Linux Distributed Security Module[1]

By

Miroslaw Zakrzewski and Ibrahim Haddad

This article describes the implementation of Mandatory Access Control through a Linux kernel module that is targeted for Linux clusters.

Introduction

Traditionally, the telecom industry has used clusters to meet its carrier grade requirements of high availability, reliability, and scalability, while relying on cost-effective hardware and software. Efficient cluster security is now an essential requirement that has not yet been addressed in a coherent fashion.

To answer the need for advanced security features on Linux clusters in the telecom world, the Open Systems Lab at Ericsson Research (Montreal, Canada) started a project Distributed Security Infrastructure (DSI), with the main goal to design and develop a secure infrastructure that provides advanced security mechanisms for telecom applications running on carrier grade Linux clusters.

One important component of DSI is the Distributed Security Module (DSM) that provides an implementation of mandatory access control within a Linux cluster.

In this article, we discuss the goals of having a distributing security module, architecture, features, performance, and implementation status. We will also go through a tutorial that explains how to install DSM and experiment with it.

Mandatory Access Control

Currently implemented security mechanisms rely on discretionary access control mechanisms. These mechanisms, however, are inadequate to protect against the various kinds of attacks in today's complex environments. The access decisions are based on user identity and ownership. Consequently, these mechanisms are easy to bypass and malicious applications can easily cause failures and breaches in system security.

Various research results have shown that mandatory security provided by the operating system is essential for the security of the whole system; furthermore, they proved that mandatory access control mechanisms are very efficient in supporting complex relationships between different entities in the computing environment.

As part of the DSI project, we address the design and implementation of a framework for the mandatory access control. We are implementing cluster-aware access control mechanisms as a Linux loadable module. Our work will help position Linux as a secure operating system for clustered servers.

---

[1] To be published in the Linux Journal.

The work is mainly based on the Flask architecture and the Linux Security Module (LSM) framework; however, the focus is on Linux clustered servers, not single Linux servers. We address the performance challenges of the cluster security since enforcing security may introduce degradation in the system performance, an increase in administration, and some annoyance for the user. Our implementation considers these factors. One important aspect of our DSM implementation is its distributed nature. This aspect provides location transparency of the security resources in the cluster from the security point of view.

Since our implementation relies on the Linux Security Module, we will discuss it in the next section.

Linux Security Module (LSM)

The LSM framework does not provide any additional security in the Linux kernel; rather it provides the infrastructure to support the development of security modules. The LSM kernel patch adds security fields to kernel data structures and inserts calls (called hooks) at special points in the kernel code to perform a module specific access control check.

LSM adds methods for registering and un-registering security modules, in addition to a general security system call that allows the communication between user programs and the LSM for security aware applications. Each LSM hook is a function pointer in a global structure called security_ops. Because the hooks are embedded in the kernel and are called even before a security module is installed, this structure is initialized to a set of functions provided by a dummy security module. These functions are just placeholders for more useful security mechanisms that can be loaded as a Linux module.

A register_security method is introduced to allow a security module to set its own security functions (to overlay the dummy functions). An unregister_security method is used to return to the dummy functions.

The LSM methods are organized into two categories:

1. Hooks to handle the security fields
2. Hooks to perform access control

When a Linux resource is created, the security label is attached to it. These labels are used to enforce mandatory access control using the security hooks. When the object is destroyed, the label is removed. Hooks to handle the security fields are used for label creation and removal. An example of those hooks are: alloc_security and free_security in the task_security_ops structure.

The process of the mandatory access control using LSM is presented in Figure 1.

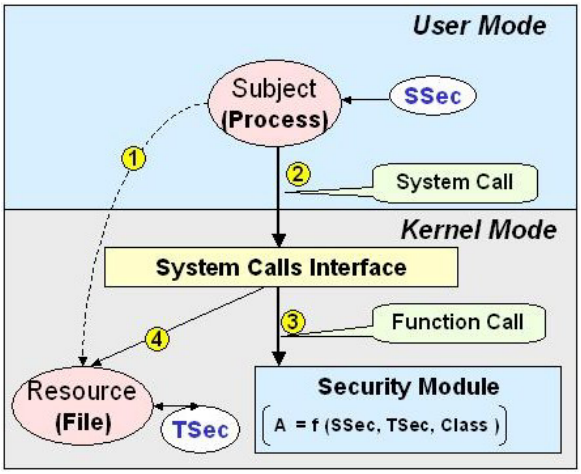

Figure 1: Access control with LSM module

Let us assume that the subject (a process in this case) has a security id SSec is trying to access (1) the resource (a file in this case) with the security id TSec.

To perform the access, the subject issues the system call (2). The system call is handled by the Linux kernel code (the system call interface in Figure 1). Before the access decision is taken, the kernel consults (using security hooks) LSM module (3), where the user specific security is implemented as a function "f". LSM will compute the function "f" and return the results to the kernel. The kernel will then either grant or deny access to the target resource (4).

The Distributed Security Module (DSM)

The Distributed Security Module is part of DSI and the purpose of implementing DSM is to enforce access control and to provide labeling for the IP messages across the nodes of the cluster with the security attributes of the sending process and node.

We started the development of DSM using Linux kernel 2.4.17 and the appropriate security patch for that kernel version (lsm-full-2002_01_15-2.4.17.patch). The implementation of DSM was based on CIPSO and FIPS-188 standards, that specify the IP header modification (adding options to IP header), and on hooks added to the IP stack.

Since DSM is a component of DSI, we will cover DSI very briefly.

Distributed Security Infrastructure (DSI)

As part of a carrier grade Linux cluster, DSI must comply with carrier grade requirements such as reliability, scalability, and high availability.
Furthermore, DSI supports the following requirements: coherent framework, process level approach, pre-emptive security, dynamic security policy, transparent key management, and minimal impact on performance.

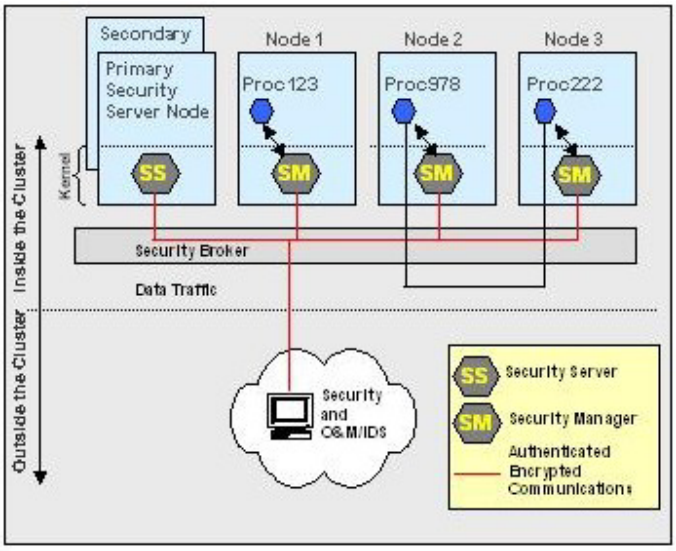

Figure 2: DSI Architecture

The security server is the central point of management. It is the central security authority for all the security components in the system. It is also responsible for the Distributed Security Policy. It

also defines the dynamic security environment of the whole cluster by broadcasting changes in the distributed policy to all security managers.

Security managers enforce security locally at each node of the cluster. They are responsible for enforcing changes in the security environment. Security managers only exchange security information with the security server.

For a detailed description of DSI, please see references.

Distributed Access Control Architecture

Designing an efficient solution to the cluster mandatory access control is a complex task. There are many factors involved in defining the access rights because the subjects and resources can be located on different nodes in the cluster. To simplify the relationships, we can handle the access control at two levels:

1. Local, when subject and resource are located on the same node, and
2. Remote, when subject and resource are located on different nodes.

For local access control, the access rights are the functions of the security IDs of the subject (SSID) and the resource (TSID) (see Figure 1). This is based on the FLASK architecture:

Access = Function (SSID, TSID)

The FLASK architecture can serve as a solution for the single node processing. When the nodes are presented as a cluster, security solutions become more complicated. In this case, we extend the FLASK architecture to the cluster remote access model (Figure 3).

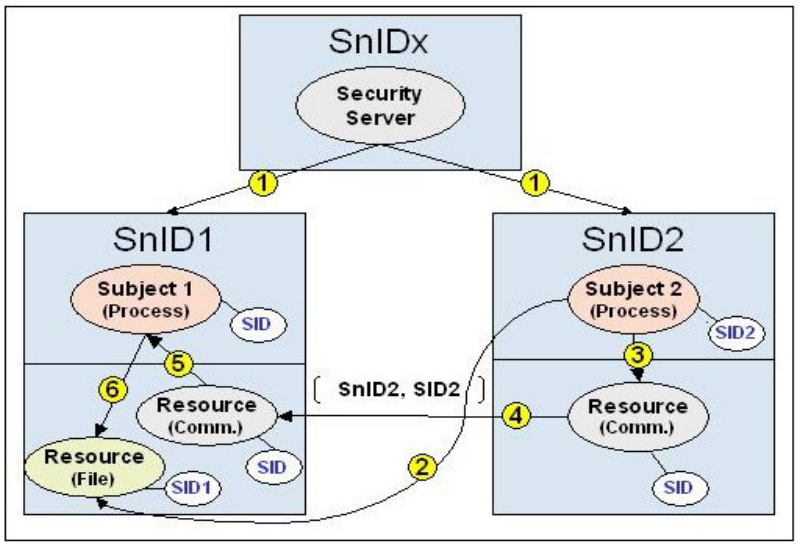

Figure 3: Distributed access control architecture

One of the new parameters is the security node ID (SnID), which defines the node interms of the security. Access rights are no more just the function of the subject and target security ID's, but as well, the function of the security node ID.

The architecture of the distributed access control is illustrated in Figure 3.

Access = Function (SnID2,SID2,SnID1,SID1)

An important part of the distributed system is the network, which spans the nodes of the cluster. To apply the access control functions in the cluster, there must be a way to pass the security parameters between the nodes in a transparent fashion. Our prototype implementation of the distributed mandatory access control will by exercised in the Linux kernel, which provides us security transparency and better performance.

In Figure 3, we illustrate an example of how the distributed access control works. The security server is responsible for passing the security policy to the security module. It is also responsible to provide the security node ID to each node of the cluster (1). Let us assume that the subject2 in node SnID2 tries to access a resource (file) on the node SnID1 (2). In this case, the subject2 must first get an access to the local communication resources (3) and then get a pair of identifiers (SnID2, SID2) that must be passed to the remote node (4). Those identifiers will be validated on the remote node SnID1 and when the access is granted, the message can be passed to the process1 (5). Now the process1will perform an access of behalf of the process 2 (6).

In the next section we will see in more details what happens on a single node of a cluster.

Access control on a node

Access control on any node of the cluster consists of two parts (Figure 4):

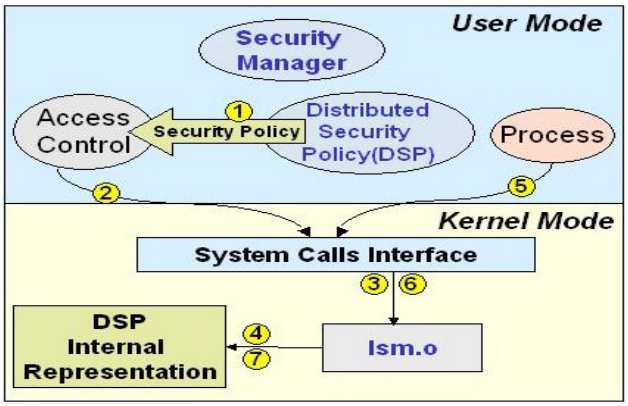

Figure 4: Access control on a node

1. ***kernel-space*:** This part is responsible for implementing both the enforcement and the decision-making tasks of access control as separate responsibilities. The kernel-space part maintains the security policy upon which it bases its decisions. The security policy is supplied by the security server and stored in the local memory for fast access (hash table).

2. ***User-space*:** This part has many responsibilities (figure 4). It takes the information from the Distributed Security Policy (1) and from the Security Context Repository, combines them together, and feeds them to the kernel space part in an easily usable form (2,3,4). It propagates back alarms from the kernel space part to the security manager, which will feed them to the Auditing and Logging Services and if necessary propagate to the security server through the Security Communication Channel (see Figure 2).

Both parts, kernel-space and user-space, are started and monitored by the local Security Manager (SM) on each node. The SM also introduces them to other services and subsystems of

DSI with which they need to interact. When a user process tries to access the system resource (5), the system call is forward to DSM (6) where the decision is taken based on the DSP internal representation (7).

Labels

All the subjects and resources must be labeled. Since the security module can be loaded at run-time, we distinguish two modes of subjects labeling.

1. Before the module is loaded there are no labels attached to any subject or resource in the system. At the module initialization time, all the running tasks are scanned and the labels are attached to them.
2.
3. When a new process is created after the security module is loaded, the security hooks are used to do the labeling.

Since Linux stores the process descriptor and the Kernel Mode process stack in a single 8KB memory area, we can use this fact and avoid allocating memory for labeling the subjects (Figure 5).

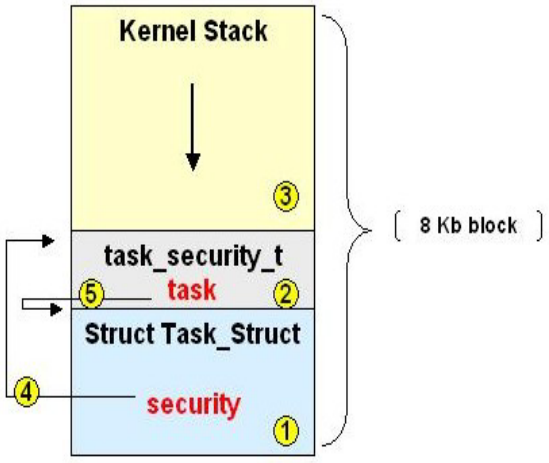

Figure 5: Task security label allocation

The other labels are attached to the resources at run-time, which implies that the module checks if the label is there. If the label is not attached, a new label will be created.

Network Labels

Because the access in the cluster can be performed from a subject located on one node to a resource located on another node (Figure 3), there is a need to control such accesses as well. When a process on one node makes an access to a resource on another node, first the local access to the communications resources (socket, network interface) is checked. When the local access is granted, the message can be sent to the remote location.

In order to identify the sending subject, the Security Node ID (security node identifier) and the Security ID of the subject (security subject identifier) are added to the IP packet.

For this implementation, we use the IP protocol for the security information transfer. A new option is added after the IP header based on the hooks in the IP protocol stack. On the receiving side,

this information (Security Node ID and Security SID) is extracted(based on the hooks in the IP stack) and used to build the network security ID (NSID).

NSID = Function (SnID, SID)

This function can be specified by the security server in form of the conversion table (for current implementation a simple mathematical function is used). The receiving side looks up into the table by specifying SnID and SID and extract the Security Network ID. Now the security network ID can be used as a local label to all the access controls.

Experimenting with the Distributed Security Module

There are several steps you need to follow to compile, load, and experiment with DSM. For illustration purposes, we assume that your machine is a Red Hat 7.2 with Linux kernel 2.4.17 (from kernel.org).

Here are the main steps involved; we explain them in details in the following sections:

- Apply the LSM patch for kernel 2.4.17
- Modify the kernel options and rebuild the kernel with the new options
- Update the boot options in /etc/lilo.conf
- Reboot the machine with the new kernel
- Compile and load the security module
- Perform some testing to validate that the module is working correctly

Rebuilding the kernel with LSM options to support the Distributed Security Module

The distributed security module is based on the LSM infrastructure, which is a set of hooks added to the kernel installing the security patch from the LSM web site. The site contains many different patches; you need to match the patch with the appropriate kernel version, in our case kernel 2.4.17.

Here are the steps to patch the kernel with LSM:

1. Download `lsm-full-2002_01_15-2.4.17.patch.gz` into `/usr/src`
2. Unzip the patch:
   ```
   % gunzip lsm-full-2002_01_15-2.4.17.patch.gz
   ```
3. Go to the linux source tree and apply the patch:
   ```
   % cd /usr/src/linux
   % patch -p1 < /usr/src/lsm-full-2002_01_15-2.4.17.patch
   ```

After applying the appropriate patch, we need to reconfigure the kernel options to support our distributed security module by following these steps:

1. Go to `/usr/src/linux`
2. If you are in XWindows, start the kernel configuration tool:
   `% make xconfig`
   If you are not in XWindows, you can use: `% make menuconfig` instead.

3. The options to be modified are the following:

   In CODE MATURITY LEVEL:
   Prompt for development and/or incomplete code/drivers must be set to "y".

In LOADABLE MODULE SUPPORT:
Set version information on all module symbols must be set to "n"
This option is to avoid problems with versioning.

In NETWORKING OPTIONS:
Network packet filtering must be set to "y" to enable the netfiltering hooks for IP packet modification.
Kernel httpd acceleration (EXPERIMENTAL) must be set to "m". This option will include tcp_sync_mss in the kernel.

In SECURITY OPTIONS:
Capabilities Support set to "m"
IP Networking Support set to "n"
NSA SELinux Support set to "m"
NSA SELinux Development Module set to "y"
NSA SELinux MLS policy (EXPERIMENTAL) set to "n"
LSM port of Openwall (EXPERIMENTAL) set to "n"
Domain and Type Enforcment (EXPERIMENTAL) set to "n"

4. Once you support these options, you need to build the kernel and produce a new kernel image. To do so, you need to do the following:
   a. Build dependencies: `% make dep`
   b. Build kernel image: `% make bzImage`
   c. Build modules: `% make modules`
   d. Install modules: `% make modules_install`
   e. Copy the new kernel image and symbol file to the boot directory:

```
% cp /usr/src/linux/arch/i386/boot/bzImage /boot/vmlinuz-2.4.17
% cp /usr/src/linux/System.map  /boot/System.map-2.4.17
% ln -s System.map-2.4.17 System.map
```

5. The only remaining step at this point is to update `/etc/lilo.conf` file to add an entry for the new kernel. Edit the `/etc/lilo.conf` file and add a new entry as follows:

```
image=/boot/vmlinuz-2.4.17
label=2.4.17-lsm
root=/dev/hda1 # change this to reflect your own partition
read-only
```

   and then update the lilo configuration by applying: `% /sbin/lilo`
   This will add an entry called "`2.4.17-lsm`" that will be presented at LILO on boot time.

6. You are now ready to reboot your machine. When LILO comes up, choose to boot "`2.4.17-lsm`".

Installing and experimenting with DSM

Since not all the components of DSI are currently implemented, we have created some test programs to emulate some parts of DSI such as loading the security policy and alarm receivers. Before the module can be used it must be loaded into the kernel by root:

```
% /sbin/insmod lsm.o
```

Next, the policy can be supplied into the security module. For this exercise, the security file is a normal ASCII file with four fields:

- source security ID
- target security ID
- class (for now only three classes are implemented: fork, socket, and network)
- permission

An extract of the policy file looks as follows:

1 1 1 0x01
1 1 2 0x07
1 1 3 0x01

The policy can be loaded with our test program called UpdatePolicy:

```
% UpdatePolicy policy_file
```

The alarms can be received by our test program called CheckAlarm. The program is started using:

```
% CheckAlarm
```

The default label for a process can be overwritten by changing the first byte of the padding field in the ELF format of the process image (witch is the eighth byte in the file).

Test configuration

We performed three types of testing to get a preliminary performance evaluation of the security module. The tests include process creation with fork, UDP local access, and UDP remote access. The UDP tests were performed with and without IP packet modification in order to see how much performance was lost during IP packet modification.

These tests were executed on a Pentium III 650 MHz machine with 256 MB or RAM. Below we explain the testing methods:

1) Process Creation: This test measures the time a process can fork a child that immediately exits. The parent process loops 100,000 performing fork and wait calls.

2) UDP Local Access: This test measures the time needed by a process to send a UDP message. It sends 500,000 UDP messages in a loop. The sending process does not check if the message was sent outside the node and does not wait for the confirmation. In this case, it is not important whether the server has DSM installed or not.

3) UDP Remote Access Testing: This test measures the time needed by a process to send a UDP messages and receive a UDP response from a server. This test sends and receives 100,000 UDP messages in a loop. The client process will send a new message after receiving the confirmation from the server. In this case, it is important that the server runs the DSM software for the permission to be checked on the receiving side. For our test, the second server is a Pentium II 300 MHz desktop with 128 MB RAM.

Performance tests results

Based on the testing performed, we present the results in Figure 6 and Figure 7. All numbers are in microseconds.

	Linux 2.4.17	Linux 2.4.17 with DSM	% Overhead
Fork	167	169	+1.20%
UDP Local Access (Send Message)	16.388	19.7	+20%
UDP Remote Access (Loopback)	133.44	173.88	+30%

Figure 6: Performance results with IP packet modification

	Linux 2.4.17	Linux 2.4.17 with DSM	% Overhead
UDP Local Access (Send Message)	16.388	17.084	+4.2%
UDP Remote Access (Loopback)	133.44	140.64	+5.4%

Figure 7: Performance results without IP packet modification

1. Process Creation Results: We have a 1.2% increase as overhead. This is because the system has to perform a permission check on the fork operation and to spend extra time on labeling the child process.

2. UDP Local Access Results: The average overhead for the setting with DSM module against the setting without the DSM module is 20%. This overhead consists of performing permission check on the socket send message and sk_buff label attachment for each message sent plus the labeling of IP messages. When the IP packet modification is disabled (Figure 7) the overhead drops to 4.2%.

3. UDP Remote Access Results: The average overhead for the setting with DSM module against the setting without the DSM module is 30%.

   The overhead consists of the following:
   - Performing a permission check on the send socket side,
   - Attaching a label to sk_buff,
   - Attaching the security information to the IP message,
   - Retrieving the security information on the receive side,
   - Attaching the network security ID to sk_buff,
   - Performing the permission checking on sk_buff,
   - Performing the security checking on the socket, and,
   - Repeating all the above operations on the return message.

   When the IP packet modification is disabled (figure 7), the overhead drops to 5.4%.

In both UDP testing cases, most of the overhead occurred is related to IP packet modification. The security module causes a small fraction of the overhead.

Ongoing Work

Our ongoing work and plans for DSM include:

a. Fully implementing the framework of the distributed access control
b. Optimizing the code for better performance
c. Providing additional functionality for the server resource to access on behalf of a client
d. Providing security information protection
e. Providing security information transfer at lower levels of the protocol stack
f. Testing the cluster security against different types of attacks

Conclusion

As previously mentioned, DSM is one component of the DSI architecture. So far, we have a basic implementation of DSM. The performance results we presented must be regarded as an upper bound because no single security operation has been optimized.

We have done some work optimizing the IP packet modification. The primary results have shown significant improvements. The UDP local access with IP packet modification (Figure 6) has dropped from 20% overhead to 8%. Similarly, the UDP remote access (Figure 6) has dropped from 30% overhead to 14%. These results are promising as we see yet many opportunities to have more optimization to reach a lower overhead. Nevertheless, the results demonstrate the challenges facing the development of efficient distributed security.

As far as testing in real environments, we tested the framework with buffer overflow attacks and it proved that our current solution could guard against these types of attacks.

As a final note, we did our best to describe DSM within the limited space we have. However, if you would like to have more details, please feel free to contact us. We hope you try out the source code for DSM and the supporting test programs and send us your feedback. All source code is available for download from the Linux Journal web site.

References

DSI reference web site	http://www.risq.ericsson.ca/dsi
DSI Linux Journal article	http://www.linuxjournal.com/article.php?sid=6053
Linux Kernel	http://www.kernel.org
Linux security modules	http://lsm.immunix.org
Flask architecture	http://www.cs.utah.edu/flux/fluke/html/flask.html
SelOpt patch	http://www.intercode.com.au/jmorris/selopt/old/
CIPSO and FIPS-188	http://csrc.nist.gov/publications/fips/fips188.html
DSM presentation	http://www.risq.ericsson.ca/dsi/AccessControl_OLS.pdf
DSM at Linux Symposium	http://www.risq.ericsson.ca/dsi/access_control_ols_paper.pdf