

RASS Framework for a Cluster-Aware SELinux

Arpan Darivemula¹, Chokchai Leangsuksun¹, Anand Tikotekar¹

Makan Pourzandi²

Louisiana Tech University¹
Open Systems Lab, Ericsson Research Canada²

apd005@latech.edu¹
box@latech.edu¹
aat007@latech.edu¹
makan.pourzandi@ericsson.com²

Abstract. The growing deployments of clusters to solve critical and computationally intensive problems imply that survivability is a key requirement through which the systems must possess Reliability, Availability, Serviceability and Security (RASS) together. In this paper, we conduct a feasibility study on SELinux and the existing cluster-aware RASS framework [5]. We start by understanding a semantic mapping from cluster-wide security policy to individual nodes' Mandatory Access Control (MAC). Through our existing RASS framework, we then construct an experimental cluster-aware SELinux system. Finally, we demonstrate feasibility of mapping distributed security policy (DSP) to SELinux equivalences and the cohesiveness of cluster enforcements, which, we believe, leads to a layered technique and thus becomes highly survivable.¹

1. Introduction

Cluster computing has increasingly become a vital infrastructure in a wide arena today to meet the growing demand for critical IT power. The role of clusters varies from one organization to others; where one's need may require high performance, another may require high availability and very robust security. The proliferation in the use of clusters has resulted in an upsurge in security issues. Such an example is the cluster wide security attack in spring 2004 [2] which only highlight the vulnerability in the cyber era. Although there are many existing security solutions, most previous works focus on independent computers (servers) and not coordinated computer nodes within a cluster. Furthermore, they are unsuitable for a downtime-sensitive cluster computing environment. In our earlier work [5], we presented a proof-of-concept RASS framework based on HA-OSCAR [3] and Distributed Security Infrastructure (DSI) [1] projects. This paper discusses a feasibility study on how to extend our exist-

¹ Research supported by Office of science, U.S. Department of Energy, the *fastOS* program, Grant # DE-FG02-04ER4614.

ing framework by incorporating SELinux. The idea behind our focus is wide acceptance and support of SELinux in the mainstream kernel code line, and similarity with our existing cluster MAC model.

2. SE LINUX Review Stage

SELinux [9] is recognized as the most complex security framework for Linux-based server systems. It provides fine granularity and robust security assurance. SELinux controlled access has approximately 458 permissions definitions for 30 security classes. End systems must be able to enforce the separation of information based on confidentiality and integrity requirements to provide system security, known as MAC (mandatory access control). MAC [6] is an administratively defined security policy which provides finer grained decision control over all processes and objects. This allows security to be separated into domains, the ability to limit process privileges, and authorization limits for legitimate users. The highlight of SELinux is its Flask Architecture [10]. Flask is a flexible MAC architecture based on Type Enforcement which provides support for dynamic security policies. It is integrated into the kernel and labels kernel objects with security contexts thereby able to identify and enforce access control decisions on all kernel operations.

3. Cluster Security

Traditional security implementations based on user login (e.g. SSH)/privileges do not support a fine-grained security approach. SELinux addresses this issue with the Flask architecture. Efficient and strong security is an essential requirement and to our knowledge has not yet been addressed in a coherent fashion on cluster systems. Challenges lie in a wide spectrum of concerns such as manageability, performance, reliability, survivability and robustness to name a few. In order to survive from attacks, cope with outages and sustain quality of service at expected levels, one must ensure Reliability, Availability, Serviceability and Security (RASS) together. This is where HA-OSCAR and DSI (Distributed Security Infrastructure) come into the picture. The combined capabilities offer promising solutions for cluster installations. Since they enforce rules much in the same way as SELinux, but on a cluster instead of end systems, it is possible to provision, control and map cluster-wide security to individual SELinux policies on the cluster nodes to achieve the same security for the cluster-wide environment.

4. HA OSCAR and DSI

HA-OSCAR [3] is a highly reliable cluster software stack that currently provides Reliability, Availability, and Serviceability (RAS) framework for a Linux HPC cluster. It supports HA and fault tolerance through “self-awareness approaches” including

self-installation, self-configuration and self-healing. On the other hand, DSI provides a distributed security framework for authentication, communication integrity, access control, and auditing. DSI [8] extends the concept of Mandatory Access Control (MAC) to a Linux cluster environment. Together HA-OSCAR and DSI provide strong, cohesive, and highly survivable and distributed control framework with centralized RASS management.

5. The Architecture

Our project main objective is to combine the strength of HA-OSCAR and the security of DSI. In [5], we implemented a prototype combining HA OSCAR and DSI. We have extended our framework by incorporating National Security Agency SELinux with our cluster-aware RASS concepts. To our knowledge, the success of this proposal will render the first field-grade open source RASS framework for Linux clusters. The DSI implements a kernel level module called the DSM (Distributed Security Module) which interacts with the kernel and enforces security rules on behalf of DSI. These rules govern the behavior of DSI are written in XML which aims for improved readability, implementation and manipulation. These rules collectively form the Distributed Security Policy (DSP). The DSM however poses a compatibility issue. As many of socket hooks used as a patch in 2.4.17 have not been accepted in main stream Linux 2.6, there is need to manually patch the kernel and re-report DSM to the newer versions. Therefore, we decided to replace DSM with the SELinux LSM to promote a reuse with the main stream Linux distributions. In addition, our goal is to combine the security granularity of SELinux and the distributive characteristic of DSI into a complete cluster security solution.

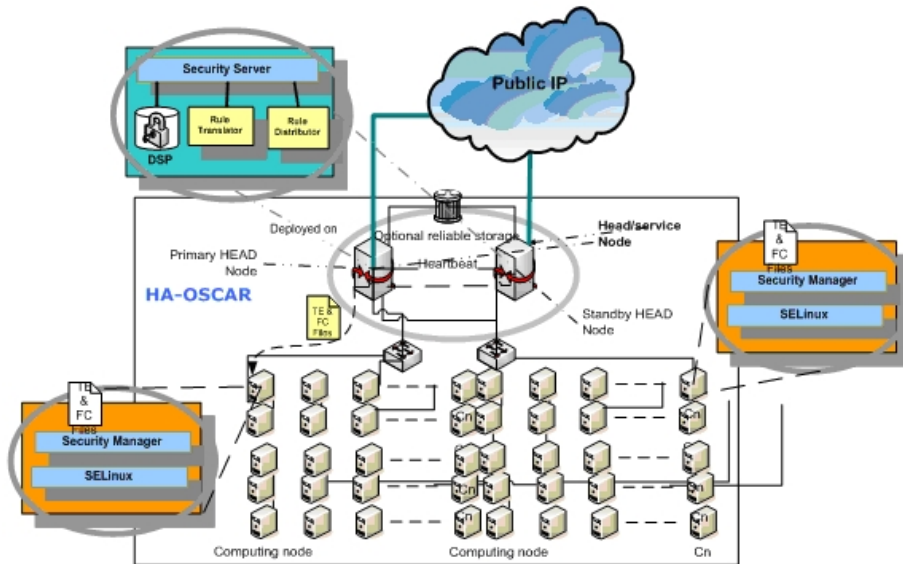


Figure 1: RASS Architecture

SELinux on the other hand, is well recognized as one of the most evolved security-oriented operating system and has also been incorporated into the Linux kernel (2.6 and higher). Using SELinux to enforce security in DSI would alleviate the compatibility issues with the DSM. SELinux provides its security policy using a declarative language. SELinux has an active community and also has ongoing support.

Our short term goal consists of replacing the DSM module with SELinux, and enforcing the DSP via Type Enforcement (TE) policy. This will remove the need for DSM while still providing similar robust security with SELinux. Although, SELinux supports other security models like role based access control (RBAC) and Multi-Level Access (MLS) approach, we currently use only the type enforcement approach.

We achieve our goal by provisioning and translating the DSI rules into SELinux rules. However, since writing security rules in a SELinux declarative language is quite a daunting task, we would like to retain the similar method of writing cluster security rules (i.e. DSP style) while having a cluster-aware rule generator that takes care of the corresponding translations. This also alleviates the compatibility issue since XML is platform independent. Another main reason to use DSP instead of directly using SELinux TE rules is the distributed aspects supported inherently in the DSP like node locality and inter-node communications. Hence, we can achieve a higher degree of abstraction with DSP than the simple use of TE rules.

In the following section, we detail how to implement a distributed security framework to enforce access control in the SELinux cluster.

5.1. Initial Scope

First we define a unique security policy for the entire cluster. This approach allows the administrator to use the same security label in the entire cluster and simplifies the access rules provisioning process. A security policy then can assign the access rules for all processes which are part of a security domain in a centralized manner, and enforce that policy on designated cluster nodes. For the time being however, the administrator assigns node IDs manually. In the future, we intend to develop a tool that automatically assigns and manages the node identification process. Our long term goal is to provide complete RASS aware cluster management that will handle system and security configuration, and provisioning.

5.2. Distributed Security Policy

The distributed security policy (DSP) is a security control definition over access control in which an administratively determined security policy can be written using a variable level of granularity. The main goal of the DSP is to define security policies to be enforced over the entire cluster. Alternative goals are ease of human readability and provisioning with syntax flexible enough to support changes. The DSP syntax has been defined using the XML language. Figure 2 shows the DSP structure file.

```

<policy>
  <dsi_policy>
    <version>
      <versionID_major>...</versionID_major>
      <versionID_minor>...</versionID_minor>
      <date> ... </date>
    </version>
    <mode>...</mode>
    <default_ScID> ... </default_ScID>

    <securityRules>
      ... (different rules)
    </securityRules>
  </dsi_policy>
</policy>

```

Figure 2: DSP File structure

For more DSP information, please refer to chapter 7 in Distributed Security Infrastructure (DSI) document [1].

Table 1: DSP security classes /rules

CLASS	ELEMENTS	PERMISSIONS
Process	ScID, SnID	CREATE
Socket	sScID, sSnID, tScID, tSnID	CREATE, CONNECT, SEND, LISTEN, RECEIVE, SHUTDOWN, GET_OPTIONS, SET_OPTIONS, GET_SOCKNAME, GET_PEERNAME
Socket_Init	Protocol, Port, ScID, SnID	TCP, UDP, RAW
Network	sScID, sSnID, tSnID	NETWORK_RECEIVE, NETWORK_SEND
Transition	parent_ScID, SnID, binary_ScID, new_ScID	

6. Cluster-Aware Security Environment

With our cluster-aware security framework, we propose to extend the type enforcement mechanisms provided by SELinux to the entire cluster environment. We adopt the existing DSI approach where the administrator modifies the DSP. The policies are then translated into SELinux rules for and propagated to different nodes in the cluster. In a typical scenario, the security server (SS) forwards the new SELinux rules

to all security managers (SMs) residing on the cluster nodes. The SMs then take care of enforcing the new rules².

Our approach is to dynamically convert all the DSP classes to its respective TE policy on individual nodes within the same environment. The DSP is a cluster-aware policy in that rules may possess security elements with various nodes' identity. The SELinux TE policy currently does not provide intrinsic cluster support. A rule translator with corresponding semantic described in Table 2, will generate SELinux equivalence rules for a given cluster-aware DSP. These rules are specific to each node and vary from node to node. Whenever a change to the DSP policy is made, this translator will dynamically generate and propagate equivalent SELinux changes to targeted nodes.

Detailed information about our RASS implementation, and DSP details can be found in [8][5]. Due to limited space, in this report only focuses on new developments compared to [5].

6.1. Translating DSP rules into SELinux TE rules

To translate the DSP rules to TE Rules, we need to:

1. Define file contexts for the enforcements with respect to executables, e.g. binary and script files
2. Assure the right labeling for created processes from those executables
3. Define access rules for interactions between these processes in the entire cluster

We also want to define the TE labels in order to reflect the process locality in our framework. For example, using a TE label "2" on all nodes of the cluster would not reflect the desired functionality of distinguishing between the processes of the same type running on different nodes. To define a cluster aware security labeling of the processes, we adopted a naming convention that reflects the node ID on the security label of the process. A strategy to extend these labels through the cluster is under investigation.

6.2. Labeling Binaries

To bootstrap the security, we need to assign the file contexts to the binary files. These file contexts are assigned under the privileged SELinux role (`sysadm_r`).

The binaries should be labeled in all cluster nodes. In the future, the SMs would label binaries on different nodes according to their path name, the node it runs on, and the specified process ID. Each label would contain the process ID and the node information, thus ensuring unique labels within the entire cluster. Such an approach would also facilitate administering the entire cluster from a single node.

²At the time of writing, this is on-going development.

6.3. Mapping DSP classes to SELinux rules

Table 2 lists the mapping between DSP rules and their associated TE rule syntax. For the time being, we concentrate our efforts in creating access rules for communications between different processes on various nodes in the cluster. Using this table as a reference, we develop SELinux specific rules in the following section for our usage scenario.

Table 2: SELinux – DSP mapping

DSP Security Class	Specific TE Rules
PROCESS	<process where> <process target> : process { perms }
TRANSITION	type \$_exec_t file_type, sysadmfile, exec_type; type_transition \$_t \$_exec_t :process \$_t; <what> <through what> <to what> allow \$_t \$_t :process transition; <from> <to> <how>
Nodes	type node_\$_t, node_type; <what> <type>
NETWORK	<from> <to> :rawip_socket <node> netif_type :netif {perms}; tcp_socket,udp_socket,unix_stream_socket
SOCKET	<from> <to> :<type of socket> {perms} } <from> <to node> :node { perms }

7. Simple scenario

In previous section, we explain the mapping between the DSP classes and SELinux security rules. We will illustrate this mapping through a simple usage scenario. Let's consider a simple scenario; where we have a UDP server and a UDP client. The UDP server binds on a pre-defined port and prints the incoming messages to the console. The UDP client connects to the pre-defined port and sends the messages to the UDP server (see **Error! Reference source not found.**, our example can also be extended to TCP client/server applications). In the following subsections, we explain step by step how we define file contexts for binaries, the transitions rules assigning the right security labels for processes created from those binaries, and the actual rules to create sockets and access the network.

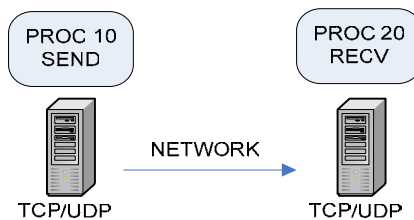


Figure 3: Simple UDP client-server scenario.

7.1. Labeling Binaries

According to SELinux conventions, we create a `dsiudp.fc` file containing all file contexts and a `dsiudp.te` file containing the actual rules³. We use SELinux command `setfiles` to label the binaries with the context specified in the `dsiudp.fc` file:

```
dsiudp.fc

/home/dsi/UDPServer          --
user_u:object_r:test_20_2_exec_t
/home/dsi/UDPClient          --
user_u:object_r:test_30_1_exec_t
```

The file contexts reflect the process security context and the locality of the binary in the cluster⁴. For example `test_20_2_exec_t` defines the ScID 20 on node 2.

7.2. Creating processes with right security labels

Once we set the file contexts, we define rules assigning the right security labels to the created processes from those binaries. We call this a transition. In DSP, the rule defining the transition is:

```
UDP Server
<class_TRANSITION_rule>
  <parent_ScID> 1 </parent_ScID>
  <SnID> 2 </SnID>
  <binary_ScID> 50 </binary_ScID>
  <new_ScID> 20 </new_ScID>
</class_TRANSITION_rule>
```

The above rule states that a binary with security context 50 should transit to security context 20 on the node with ID 2 and with parent security context 1;

```
UDP Client
<class_TRANSITION_rule>
  <parent_ScID> 2 </parent_ScID>
  <SnID> 1 </SnID>
  <binary_ScID> 60 </binary_ScID>
  <new_ScID> 30 </new_ScID>
</class_TRANSITION_rule>
```

³ In the current SELinux implementation, the file contexts and rules can be all in the same file.

⁴ In our simple scenario, we do not consider the case of shared file systems. A detailed solution for shared file systems will be provided in future publications.

This rule states that a binary with security context 60 transits to security context 30 on the node with ID 2 and with parent security context 2. The above rules are translated to the following TE rules:

dsiudp.te

```
#UDPServer

#define the domain
type test_20_2_t, domain, privlog;

#make sure test_20_2_exec_t can be executed
type test_20_2_exec_t, file_type, sysadmfile, exec_type;

#macro to transit from one domain to another
domain_auto_trans(user_t, test_20_2_exec_t, test_20_2_t);

#provide permissions for user to execute
allow user_t test_20_2_exec_t:file{ execute read };
role user_r types test_20_2_t;

#UDPClient

#define the domain
type test_30_1_t, domain, privlog;

#make sure test_30_1_exec_t can be executed
type test_30_1_exec_t, file_type, sysadmfile, exec_type;

#macro to transition from one domain to another
domain_auto_trans(user_t, test_30_1_exec_t, test_30_1_t);

#provide permissions for user to execute
allow user_t test_30_1_exec_t:file{ execute read };
role user_r types test_30_1_t;
```

To show some useful functionality of DSP, we consider the scenario that the client process forks to create several clients. To allow a process to fork in DSP, we simply add the rule:

```
<class PROCESS Rule>
  <ScID>30</ScID>
  <SnID>1</SnID>
  <allow>CREATE</allow>
</class Rule>
```

This rule states that a process with security context 30 on node ID 1 can create a process/call a system fork. The only rule supported by DSP right now is CREATE a process.

dsiodp.te

```
Client
#allow the process to fork
allow test_30_1_t self:process{ fork };
```

7.3. Access Rules

The illustrated rules so far guarantee that the client and server processes will have the right labels. Now, the next example shows the access right rules to allow access to network resources.

To permit communication between processes on different nodes, we include the following rules in DSP:

```
Server
<class_SOCKET_rule>
  <sScID>20</sScID>
  <sSnID>2</sSnID>
  <tScID>20</tScID>
  <tSnID>2</tSnID>
  <allow>CONNECT LISTEN SHUTDOWN CREATE RECEIVE</allow>
</class_SOCKET_rule>

<class_NETWORK_rule>
  <sScID>20</sScID>
  <sSnID>2</sSnID>
  <tSnID>1</tSnID>
  <allow>NETWORK_RECEIVE</allow>
</class_NETWORK_rule>

Client
<class_SOCKET_rule>
  <sScID>30</sScID>
  <sSnID>1</sSnID>
  <tScID>30</tScID>
  <tSnID>1</tSnID>
  <allow>CONNECT LISTEN SHUTDOWN CREATE SEND</allow>
</class_SOCKET_rule>

<class_NETWORK_rule>
  <sScID>30</sScID>
  <sSnID>1</sSnID>
  <tSnID>2</tSnID>
  <allow>NETWORK_SEND</allow>
</class_NETWORK_rule>
```

These rules state that processes with security contexts 20 and 30 can use sockets with the above permissions and also send and receive via the network. These rules are translated to the following TE rules:

dsiudp.te

```
#Network permissions for Server/Client interaction

# UDP Server
allow test_20_2_t node_t:udp_socket node_bind;
allow test_20_2_t port_t:udp_socket name_bind;
allow test_20_2_t self:udp_socket { bind create read };
allow test_20_2_t user_devpts_t:chr_file { getattr read write
};
allow user_t test_20_2_t:process { noatsecure rlimitinh
siginh };
allow test_20_2_t netif_lo_t:netif udp_rcv;
allow test_20_2_t node_t:node udp_rcv;
allow test_20_2_t port_t:udp_socket { name_bind rcv_msg };

# UDP Client
allow test_30_1_t netif_lo_t:netif udp_snd;
allow test_30_1_t node_t:node udp_snd;
allow test_30_1_t port_t:udp_socket send_msg;
allow test_30_1_t self:udp_socket { connect create write };
allow test_30_1_t user_devpts_t:chr_file { getattr read write
};
allow user_t test_30_1_t:process { noatsecure rlimitinh
siginh };
```

8. Prototype

We have prototyped a simple TE Rule generator: GenRule. GenRule parses the DSP XML file, and translates the DSP rules into TE equivalences. In a given example, the mapping between the binary files and SELinux file contexts is manually done through a web interface. We first ask the administrator to give as an input the DSP file that he or she wishes to enforce on the cluster. We then parse the XML from that file and get a list of classes and rules. We then ask the administrator to map the IDs present in the DSP to actual files (since SELinux does not deal with IDs as mentioned earlier), see Figure 4.

SECURITY POLICY ADMINISTRATION

Please map the ids with their filenames (full path)

ID	FILENAME with path
50	
20	/home/dsi/UDPServer
60	
30	/home/dsi/UDPCClient

Figure 4: Web Interface for file contexts input.

Once we have the mapping table, we then translate each class in DSP separately according to predefined constraints into SELinux policy. The translations are grouped together based on which nodes they pertain to and are written to files. The accuracy of rules generated has been manually verified for this scenario and a few others.

Our experimental cluster system was based on Fedora Core 3 Linux distribution with a linux-2.6.11 kernel, configured for SELinux strict policy (v 1.19.10-2). We installed OSCAR on this system, followed by HA-OSCAR for the High Availability aspect. Please note, as per the OSCAR documentation, SELinux must first be disabled before OSCAR installation and re-enabled after HA-OSCAR installation. In our simple scenario setup, we have introduced 54 new rules to the existing SELinux rule set (more than 296000 rules). We have not observed any performance degradation due to the introduction of our rules. Though, we are at the early stages of testing, and we will continue the validations with more complex scenario setups. A more detailed performance analysis can be found in Section 10.

In DSP provisioning, node information was obtained from the OSCAR database (mysql). This removes the need for the user to update multiple locations each time a node is added or deleted. When a node is removed, all rules pertaining to that node are invalidated. All process ScID and the corresponding filename is stored in an additionally created table in the OSCAR database. The supplementary prototype database schema is shown in Figure 5. The **bold** fields are used by GenRule when generating rules.

oscar.nodes	
PK	id
	gateway
	dns_domain
	cpu_type
	hostname
	installer
	domain
	default_gateway
	image_id
	swap
	id
	cpu_speed
	name
	cpu_num
	fqdn
	cluster_partition_id
	ram
	units

oscar.maps	
PK	id
	id
	xmlid
	filename

Figure 5: Minimal OSCAR Database Schema.

9. Security Policy Propagation

In the process of transitioning from DSP to SELinux rules, we decided to implement the Security Server on the Head Node as a Perl client, and the various Security Managers on the Client Nodes as Perl servers. This client-server setup currently accepts three commands – *compile*, *reload*, and *exit* which compile a new policy, reload an existing policy, and exit the server respectively, see Figure 6. We consider further extension of this set of commands to include operations such as re-labeling files, sending errors back to the head node which may occur during policy compile a policy load, etc.

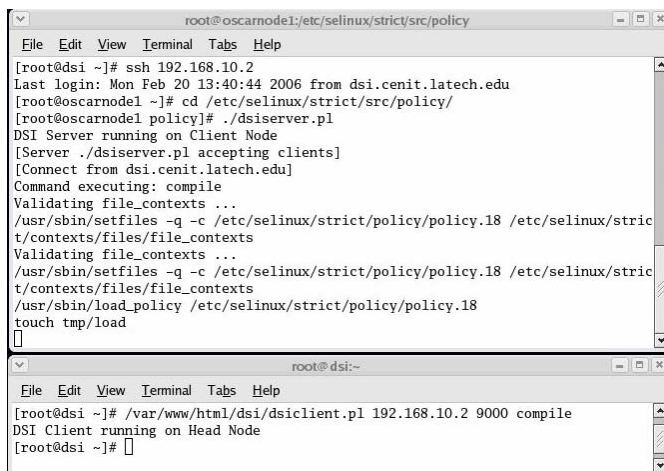


Figure 6: Perl Client-Server setup.

In addition, work is currently underway to automate this process as soon as changes in rules are observed. Translated SELinux rules are propagated via SCP to all the client nodes. This happens as soon as new rules are generated by GenRule. The status of all operations is shown in the web interface. Figure 7 shows a sample status page for GenRule.

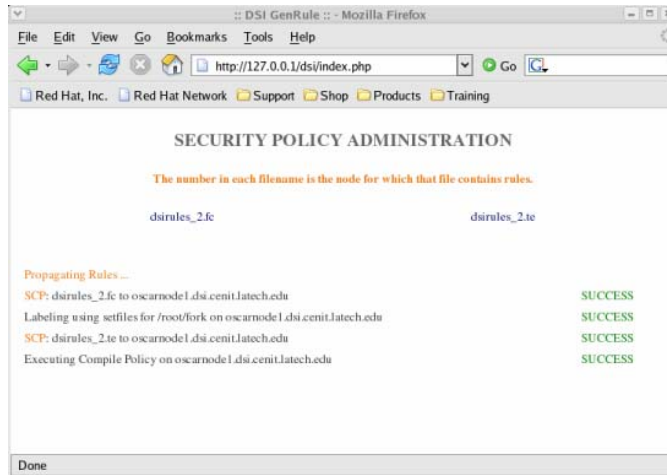


Figure 7: Sample GenRule Status Page.

The complete provisioning framework for GenRule is depicted in Figure 8.

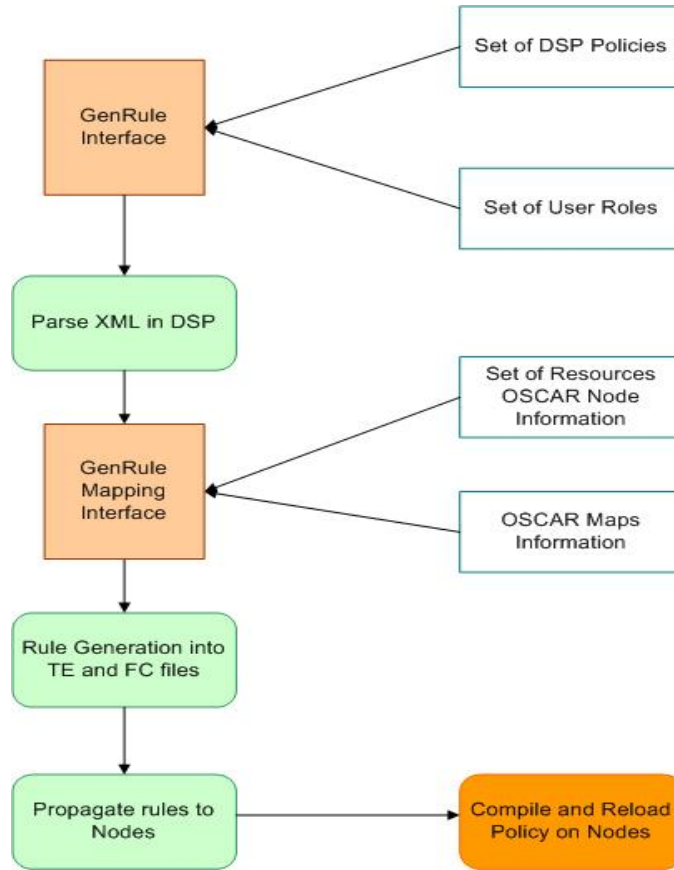


Figure 8: GenRule Provisioning Framework.

10. Performance Analysis

A performance analysis of SELinux and DSI is shown in this section. Table 3 shows the overhead of SELinux.

Table 3: SELinux – Measurements are in microseconds. Measurements below the bar represent round trip latency for various forms of IPC.

Micro bench- mark	Base	SELinux	Overhead
null I/O	1.45	1.93	33%
Sat	8.06	10.3	28%
open/close	11.0	14.0	27%
0KB create	22.0	26.0	18%
0KB delete	1.72	1.90	10%
Fork	499	505	1%
execve	2730	2820	3%
Sh	10K	11K	10%
Pipe	12.5	14	12%
AF_UNIX	20.6	24.6	19%
UDP	310	356	15%
RPC/UDP	441	519	18%
TCP	389	425	9%
RPC/TCP	667	726	9%
TCP connect	675	738	9%

Table 4: DSI – Comparison of performances between a LSM patched kernel without any security mechanisms implemented and a kernel supporting DSI distributed security services. Time units are microseconds.

Test type	Base	DSM	Overhead
Stat	1.98	1.94	-2.0%
Open/Close	2.68	2.68	0%
Fork	92.81	93.58	0.82%
Exec	322.56	328.33	1.78%
Sh proc	2150	2140.75	-0.43%
UDP	9.68	10.61	9.6%
RPC/UDP	17.66	18.7	5.9%
TCP	11.08	12.68	14.4%
RPC/TCP	23.42	24.3	3.75%

Comparing overhead between DSM and SELinux, we notice that DSM imposes lesser overhead than SELinux is every aspect except in TCP. Since SELinux was developed as a general purpose security system, it has a higher overhead of policy enforcement. DSM on the other hand was developed as part of a cluster-aware environment and exhibits lower overhead but is not as feature-rich as SELinux. In addition, SELinux has the advantage of being part of Linux mainstream. Therefore, in despite of greater overhead of SELinux, we consider using SELinux instead of DSM.

11. Conclusion

We have introduced a cluster-ware security framework with SELinux enhancement. Our approach of integrating HA-OSCAR (RAS) and DSI (S) is an integral attempt to view the perspective of RASS and the cohesiveness of cluster security, which, we believe, leads to a layered technique and thus becomes highly survivable. Currently we have translated rules and demonstrated feasibility of mapping DSP classes to SELinux equivalences. We are in the process of extending more DSP classes and making the whole process automated. We also aim to address the ease of cluster security policy expression and provision as well as striving for efficacy of the overall cluster RASS management. This work demonstrates the feasibility of a unified security configuration mechanism for cluster computing. Perhaps the mechanism itself is not a breakthrough, but an XML-based unified operating environment for cluster-based computing is prerequisite for future advancement.

12. References

- [1] Distributed Security Infrastructure (DSI) document, Revision 0.3, May 29, 2003, http://www.linux.ericsson.ca/dsi/master_dsi.pdf
- [2] "Hacker Attack Prompts Academic Supercomputers to block Outside Access Temporarily." *The Chronicle of Higher Education*, April 15, 2004.
- [3] HA-OSCAR: <http://xcr.cenit.latech.edu/ha-oscar>
- [4] C. Leangsuksun, et al, "Availability Prediction and Modeling of High Availability OSCAR Cluster", *IEEE International Conference on Cluster Computing*, 2003.
- [5] C. Leangsuksun, A. Tikotekar, M. Pourzandi, I. Haddad, "Feasibility Study and Early Experimental Results Toward Cluster Survivability", *1st Intl. Workshop on Cluster Security (Cluster-Sec) held in conjunction with the 5th Intl. Symposium on Cluster Computing and the Grid (CCGrid), 2005*
- [6] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments." in *Proceedings of the 21st National Information Systems Security Conference (NISSC), 1998, pp. 303-314.*
- [7] P. A. Loscocco, S. D. Smalley. "Integrating Flexible Support for Security Policies into the Linux Operating System" in *Proceedings of the Usenix 2004 Annual Technical Conference (FREENIX '01)*, 2001.
- [8] M. Pourzandi, A. Apvrille, E. Gingras, A. Medenou, D. Gordon, "Distributed Access Control for Carrier Class Clusters", in the *Proceedings of the Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, 2003.
- [9] SELinux: <http://www.nsa.gov/selinux/>
- [10] R. Spencer, S. Smalley, "The Flask Security Architecture: System Support for Diverse Security policies", in *Proceedings of the Usenix Security Symposium, 1999*