

USENIX Association

Proceedings of the FREENIX Track:
2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004



© 2004 by The USENIX Association
Phone: 1 510 528 8649

All Rights Reserved

FAX: 1 510 548 5738

Email: office@usenix.org

For more information about the USENIX Association:

WWW: <http://www.usenix.org>

Rights to individual papers remain with the author or the author's employer.

Permission is granted for noncommercial reproduction of the work for educational or research purposes.

This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

A new Distributed Security Model for Linux Clusters

Makan Pourzandi

Open Systems Lab, Ericsson Research, Town of Mount-Royal (QC) Canada.

Makan.Pourzandi@Ericsson.Com

Abstract

With the increasing use of clusters in different domains, efficient and flexible security has now become an essential requirement for clusters, though many security mechanisms exist, there is a need to develop more flexible and coherent security mechanisms for large distributed applications.

In this paper, we present the need for a unified cluster wide security space for large distributed applications. Based on these needs, we propose a new security model that implements security zones inside the cluster. The model is an extension to Mandatory Access Control (MAC) mechanisms used at node level to the whole cluster with processes as basic security entities.

We designed this model with clustered Linux servers running carrier-grade applications in mind but this model can be used in any domain that needs Linux clusters running large distributed applications continuously with no interruptions. We prove the feasibility of this approach through an open source implementation of the concept [1].

1 Introduction

Distributed applications become more and more complex. Often, they are made of different software components of different functionality. The current security mechanisms are based on user permissions, that provides solid security environment in many cases but they are not flexible enough for large distributed applications.

In many of these applications, only few users exist on dedicated clusters for running a pre-defined set of software for a long period of time without interruption. However, the user based security system does not support authentication and authorization checks for interactions between two processes belonging to the same user. This situation leads to an all-or-nothing approach, as all

users within a group or all processes of the same user have the same rights. This is most inconvenient when one wishes to compartmentalize these rather large distributed applications by restricting the access to some resources to some processes or users of the same group.

This lack of compartmentalization between different software components, results in that a vulnerable small component may compromise the whole system. This situation gets even worse with systems running untrusted software (unfortunately, in practice, it is impossible for time and economic reasons to security audit all software running for these rather large applications).

This situation is due to the use of user-level granularity as the basic entity for the security control in these distributed applications, for which this granularity is not sufficient. Therefore, there is a need for finer granularity security mechanisms which use the individual process as the basic entity.

Furthermore, the security must be pervasive and make a coherent system across the cluster. Therefore, distributed security functions must be put in place throughout the cluster. However, the current situation is based on assembling heterogeneous security solutions for different nodes. This leads to security management nightmares and stiff integration problems, and too often leaves gaps between different security mechanisms allowing security breaches.

The paper is organized as follows: Section 2 discusses the main concepts behind our approach. In Section 3, we detail the implementation of our security model. We conclude by presenting the future work.

2 Cluster-wide security space

We detail hereafter different elements of our security model.

2.1 Process level granularity

In order to implement process-level security mechanisms, we need to identify the different security contexts for individual processes inside the cluster.

A *security node identifier* (SnID) is assigned to each node. All processes and resources also receive a *security context identifier* (ScID). We define ScIDs for processes, binary files and resources on the cluster. These secure IDs are coherent and meaningful within the entire cluster. Also, ScIDs are persistent (they do not change after rebooting the system). Actually, one should think of ScIDs more like security GIDs than PIDs: ScIDs are meant to group together processes and resources that have the same security context.

Any newly created process is assigned a ScID which is based on the ScID of the parent process, the ScID stored in the loaded binary, and the general security context of the system.

In our security model, all different security mechanisms (access control, authentication, confidentiality, integrity and logging) are based on the pair (ScID, SnID). We show in §3.2, an implementation of the above model for process-level, cluster wide access control mechanisms. A more detailed article on the implementation can be found in [2].

2.2 Mandatory Access Control at cluster level

In *Discretionary Access Control* (DAC), the objects' permissions are set by their owners. So, as soon as gets hold of a (buggy) process, he gains access to *all resources* available to the owner. This is used in several buffer overflow exploits to allow attacker to gain root privilege.

Adding the *Mandatory Access Control* to Linux remedies this problem. In MAC, access control no longer solely depends on the user's decision but also on a variety of security-relevant information. For example, executing a given process requires the correct Unix permissions, but also that the current security context permits the creation of a new process. This is particularly useful when the administrator needs to execute two types of program: `secure-prog` that s/he trusts and `handle-with-care-prog` that s/he does not entirely trust. Therefore, s/he considers that `secure-prog` should be allowed to spawn new processes, but `handle-with-care` should not. This is

what MAC provides: it clearly assigns different security contexts to these two programs. With the DAC mechanisms only implemented, the administrator needed to create separate groups/users for both programs and avoid shared resources between two groups/users, and set the strict permissions for each groups/users. This is clear that this is not sustainable effort for an important number of binaries.

Even though the Linux community has not yet come up with a standard way of implementing MAC mechanisms inside Kernel Linux, several projects exist (c.f. SE Linux [5]...) implementing MAC approach at Linux kernel level. However, those solutions are still dedicated to single nodes; The DSI project allowed us to prove the feasibility of extending the MAC mechanisms to the distributed systems.

With persistent ScIDs for processes across the cluster, we extend the access control checks at kernel level from local nodes into the entire cluster. The ScID and SnID of the process initiating a connection are carried with the IP packet to the other end of the connection. This way, the permissions of the processes to access resources are verified in the entire cluster independently from their location. Note that these verifications are at kernel level, and are independent from the mechanisms to be implemented at application layer by the developers (this is particularly important when running untrusted code, or software that can not be modified with security considerations for technical or historical issues). We detail the above model in §3.2.

With MAC at the cluster level supporting process-level granularity, we have the necessary means to implement security zones inside the cluster.

2.3 Security zones

In order to compartmentalize the system, we define different security zones within the cluster and enforce the security policy for different security zones (see Figure 1).

The security zones are created by defining the security rules for interactions between different processes and resources through the cluster. For example, it is possible to define an access control rule stating that processes on node 1, with ScID of 2 may create sockets and connect to the processes of node 2, with ScID 2.

That is by assigning the same ScIDs to the processes and

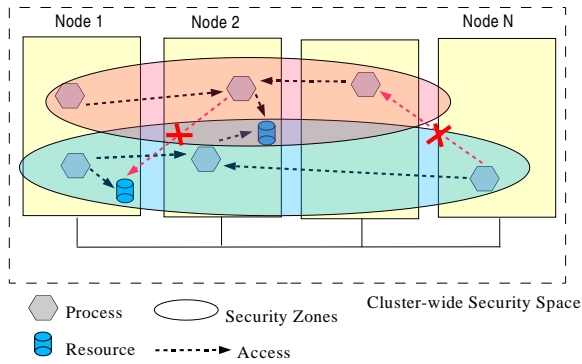


Figure 1: Security zones are defined in the logical cluster-wide security space.

resources of the same security zone, and defining rules to restrict any connection or access to resources for the zone defined by the ScID, the administrator can define a security zone throughout the cluster independently to where the processes are running or where the resources are located in the cluster. Note that the administrator can create a zone with privileges to access all zones for administrative purposes, or define the shared resources among different zones.

All security rules are collected in the *distributed security policy* (DSP) in order to define a unique, homogeneous and cluster-wide security policy to be enforced all over the nodes of the cluster.

Hence, the DSP simply consists of a list of rules to be applied to pairs of (SnID, ScID). Through the DSP, security rules can be set for each (SnID, ScID) pair, thus enabling a fine-grained process-level security policy, valid across the entire cluster.

The DSP is automatically propagated to all nodes of the cluster at initialization time, and after each change updates are sent to all nodes of the cluster. The new rules are then locally compiled and cached in the kernel memory for the fast access (this is done by the security managers in each node within the cluster, see §3).

Note that defining security zones is then simplified to editing DSP to set the same ScID for all processes and resources belonging to the same security zone (see details in §3.2).

3 An implementation of the model: DSI

To validate our approach and show its feasibility, an open source project, named *Distributed Security Infrastructure* (DSI), was initiated in 2002, so as to propose an adequate security solution for carrier-grade clusters. DSI implements the above mentioned security model [1].

DSI is composed of one security server (SS) and multiple security managers (SMs) - one per node. The SS is the central point of management of the cluster: it gathers all alarms and warnings sent by the SMs and propagates the security policy over the cluster. Each SM is responsible to enforce security on its own node.

DSI is based on open and standard software such as Linux Security Modules (LSM) for kernel level security mechanisms, OmniORB, an open-source implementation of Corba [3] and SSL/TLS for communication security.

Administrative messages between SMs and SS are sent on secure encrypted and authenticated channels, using SSL/TLS (i.e.; Secure Communication Channel in Figure 2).

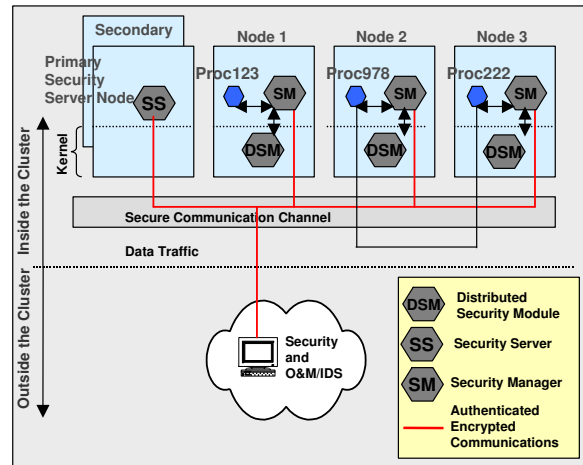


Figure 2: Distributed Architecture of DSI

3.1 The Distributed Security Policy (DSP)

In DSI, all security rules are defined through the Distributed Security Policy (DSP). The goal of the DSP is to define a unique, homogeneous and cluster-wide security policy to be enforced over all nodes of the cluster. It

```

<SOCKET_class_rule>
  <sSnID> 1 </sSnID>
  <sScID> 2 </sScID>
  <tSnID> 2 </tSnID>
  <tScID> 2 </tScID>
  <allow> CREATE </allow>
</SOCKET_class_rule>

```

Figure 3: Access control rule example for a socket.

contains customization for all security services running on DSI.

Basically, an access control rule consists in various permissions to be applied to entities (i.e.; processes, sockets,...) sharing the same security context and security node identifier (i.e.; a ScID/SnID pair). Permissions are organized in different *classes*. For example, there are permissions relative to sockets (create, bind, send, receive), others relative to process creation. All kinds of permissions have not been implemented yet. Actually, we have mainly focused on network communication and process creation so far.

Currently, the DSP supports the following rule types:

- Process class allowing or denying a given pair (ScID, SnID) permission to spawn new processes. The DSP enables control over `fork()` or `execve()` system calls (c.f. §3.2.2).
- Socket class controlling specifically permissions of sockets on the cluster. ScIDs may be assigned to sockets of a given node, for a given protocol and port (currently only TCP and UDP are supported.). Then, it is possible to set permissions between source and target sockets/processes.
- Networking class controlling network permissions on the cluster, such as allowing or denying a given pair (ScID, SnID) to receive network information from a given pair (ScID, SnID).
- Transition class defining how ScIDs are assigned to processes according to the ScID stored in their binary file and their parent process.

For instance, Figure 3.1 illustrates an access control rule. It states that resources on node 1, with ScID of 2 may create a sockets to the processes in node 2, with ScID 2. All other connections from a process with ScID 2 in node 1 to any process in node 2 will be denied.

In order to be flexible and to ease human readability of configuration file, the DSP was implemented using XML language. On top of those extensibility features,

XML comes with a variety of open source tools and with its own security mechanisms [4].

In the following, we do not explain DSI in details but rather concentrate on a practical example to illustrate our approach: the distributed access control mechanisms. Further details on DSI and DSP can be found in [1].

3.2 A practical case: the Distributed Security Access Control service (DisAC)

DisAC implements the MAC paradigm over the *entire* cluster with process-level granularity as this was discussed in §2.2.

DSP allows to maintain a homogeneous, central point of security management for the cluster. The security administrator sets up the security policy on the security server, and then, the DSP gets propagated across the cluster.

DisAC also allows administrators to simplify access control rules by setting different categories of security contexts.

In the following we detail main functionality of the system. A detailed explanation of DisAC can be found in [2].

3.2.1 Cluster-wide access control for DisAC

DisAC extends the local access control to a distributed access control for the whole cluster, using both source/target security node and security context identifiers as security information: $AccessDecision = Func(Source < SnID, ScID >, Target < SnID, ScID >)$.

The access decisions are enforced locally by DSM based on DSP rules for local and remote processes accessing local resources. For example in Figure 4, we illustrates the case where a process accesses a remote resource. First, a local check is performed to verify the access permissions of the local entity (here, process 12, ScID=10) to use network resources (DisAC by default assigns the ScID of the process creating the socket to the socket, therefore the local TCP socket ScID is 10). If permission is granted, ScID and SnID of the source entity (here process 12, ScID=10) are added to all IP packets sent from this socket. ScID and SnID are added in the IP Options

based on FIPS definition of standard security labels for information transfer [7]. When the IP packet is received on the remote node, the DSM module extracts the ScID and SsnID from IP Packet and checks the access permission to the defined resource (here port 8000, ScID=10, DisAC allows to set ScIDs explicitly for each TCP or UDP port). Furthermore, a check is performed to verify that source entity has the permission to send information to the parent process which created the socket (here this comes to check permissions between process 12 and process 14). Finally, of course, DSM verifies that the process 14 has permission to receive information from TCP socket port 8000.

Therefore, access privileges may be defined at process-level for both local and remote nodes. For example, it is possible to define that a process of type A is only able to access resources of type B on nodes M and N of a given cluster.

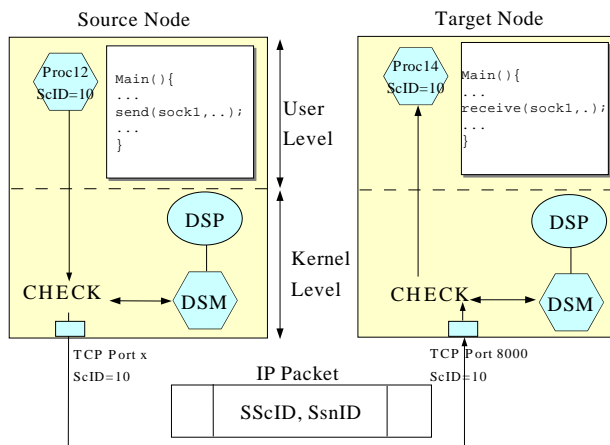


Figure 4: Secure remote access control.

3.2.2 Storing security context information (ScID) into binaries

A newly created process is automatically assigned a ScID by the DSM (see §3.2.3) based on the loaded binary file, the parent process and the security context of the system.

In DSI, we store a ScID in each binary (stored in its Elf header). This allows dividing binaries into different types based on the information for the provider of the software, or the level of trust for the binary. DSI also has mechanisms to support digital signatures for binaries in order to avoid these ScIDs to be tampered with by the

intruders.

DisAC is particularly useful for large clusters, where there is a need for compartmentalization into distinct sub-clusters with restricted/controlled connections between sub-clusters. For instance, this scenario is quite useful for telecommunication clusters that are shared among different operators: operators share the global infrastructure of the cluster providing different services, but they certainly do not wish to share their binaries or data with other operators.

Therefore, the operator assigns the same ScID for the binaries provided by each operator and the resources to be accessed on different nodes by that application. Furthermore, the operator defines the access rules among the ScIDs, setting up possible interactions between different security zones.

3.2.3 Access control at kernel level

For security not to be bypassed, the DisAC service has been implemented at kernel level, in the *Distributed Security Module* (DSM). DSM is a set of kernel functions enforcing distributed security policy, and is implemented using Linux Security Module hooks as a Linux kernel module [6].

3.3 DSI Benchmark results

Tests have been performed using LMBench 3.0 [8], on two different configurations: a 2.4.17 kernel with the LSM [6] patch, without any security check performed and the same patched kernel, with the DSM module loaded, implementing different security mechanisms defined in DisAC (see Table 1).

UDP and TCP latency tests are performed by having client and server loop on exchanging a message of 4 bytes. The RPC tests are similar, but using Sun's RPC layer over TCP or UDP (see [8]).

DSI involves minimal overhead for local operations. For TCP and UDP tests, DSM's overhead ranges from 4% to 15% as security checks have to be done before processes are allowed to communicate and at the reception of each IP packet. The TCP Connect result is particularly heavy for DSM because TCP is a "three-way handshake" and security checks are done at each stage of the handshake. For RPC tests, as the overhead due to RPC connections increases in the total time of communication, the over-

Test type	Without DSM	With DSM	Overhead
Stat	1.92	1.94	1.0%
Open / Close	2.68	2.68	0%
Fork	92.81	93.58	0.82%
Exec	322.56	328.33	1.78%
Sh proc	2140.75	2150	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%

Table 1: Comparison of performances between a LSM patched kernel without any security mechanisms implemented and a kernel supporting DSI distributed security services. Tests have been done on an Intel Pentium IV 2.4 GHz. Time units are microseconds.

head due to security checks decreases in percentage.

4 Future work

During the work with DSI, we realized that having implemented many of necessary mechanisms, one of the major challenges is creating the “right” distributed security policy (DSP). For time being, all interactions between different pairs (ScID, SnID) must be explicitly defined in the DSP. This task when many applications are involved becomes quite complex. We plan to simplify the DSP creation and sub-sequent modifications by generating the rules to allow interactions between all entities (processes and resources) of the same security zone. We avoid implementing this as a default rule inside DSP as we believe this could be too restricting in some cases. Still, one major issue would be how to handle interactions between different security zones in a user friendly way.

The security for communications between nodes is guaranteed using IPSec. We plan to implement a more fine grained security mechanism based on security policy set for each ScID. This comes to define security mechanisms for communications between processes inside a security zone independently from the location of processes within the cluster.

5 Conclusion

In this paper, we proposed a new security model for Linux clusters based on security zones allowing a flexible compartmentalization of software components comprising large distributed applications. This security model is based on defining process-level security rules and expanding MAC mechanisms to the whole cluster to enforce those rules.

References

- [1] *The Distributed Security Infrastructure Open Source Project*, <http://disec.sourceforge.net>.
- [2] M. Pouzandi, 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), Las Vegas, June 2003.
- [3] *omniORB*, <http://omniORB.sourceforge.net/>
- [4] Eastlake D., Reagle J., Solo D., *XML-Signature Syntax & Processing*, Network Working Group, RFC 3275, March 2002.
- [5] P. Loscocco, S. Smalley, *Integrating Flexible Support for Security Policies in the Linux Operating System*, in the Proceedings of the FREENIX track of the 2001 USENIX Annual Technical Conference, 2001, <http://www.nsa.gov/selinux>.
- [6] C. Wright, C. Cowan, S. Smalley, J. Morris, G. Kroah-Hartmann, *Linux Security Modules: General Security Support for the Linux Kernel*, in the Proceedings of the 2002 USENIX Security Symposium, <http://lsm.immunix.org>.
- [7] J. Morris, *SelOpt: Labeled IPv4 networking for SE Linux*, <http://www.intercode.com.au/jmorris/selopt>.
- [8] *LmBench*: <http://www.bitmover.com/lmbench>.