Future Generation Computer Systems 29 (2013) 876–885

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Improving Mandatory Access Control for HPC clusters

M. Blanc[a], J.-F. Lalande[b,*]

[a] CEA/DAM/DIF, Bruyères-le-Châtel, 91297 Arpajon, France

[b] Centre-Val de Loire Université, LIFO, ENSI de Bourges, 88 bd Lahitolle, 18020 Bourges, France

ARTICLE INFO

Article history:
Received 13 April 2011
Received in revised form
16 February 2012
Accepted 22 March 2012
Available online 5 April 2012

Keywords:
HPC clusters
Access control
Benchmarking

ABSTRACT

HPC clusters are costly resources, hence nowadays these structures tend to be co-financed by several partners. A cluster administrator has to be designated, whose duties include, amongst others, the prevention of accidental data leakage or theft. Linux has been chosen as an operating system for the CEA's computing platforms. However, strong system security solutions such as SELinux are usually difficult to set up in large environments.

This article presents how we have adapted a MAC mechanism in order to enforce confidentiality and integrity between a large number of users. First we define our security objectives, and show how they direct our technical choices. Then we present how confinement was achieved using the SELinux security mechanism, and how various attack scenarios were addressed. We then focus on the use of Mandatory Categories, access control on high bandwidth network filesystems and the integration of new users and applications. We discuss some residual technical challenges. Finally, we present benchmark results and validate the acceptable performance impact of our deployment on a modern cluster.

1. Introduction

The security of operating systems in computing clusters is often overlooked, the focus being on performance. However, these computing architectures are often shared between several partners, and companies running large HPC clusters often sell computing time to interested third parties. The data and computation performed by a party on these clusters are almost always confidential for this party. Of course they have to trust the cluster administrator, but they do not necessarily trust other parties using the same computing resources. Hence, it is the responsibility of the company that owns the computing facility to ensure that proper security policies and mechanisms are in place in order to protect all partners and clients data and processes and prevent any unauthorized access.

High performance computing architectures are extremely specialized in comparison with general computing facilities. Fig. 1 shows a typical HPC cluster architecture. This architecture has specific security issues and properties. As outlined by William Yurcik [1], these issues must be addressed in a way that is relevant, using a combination of techniques dedicated or not to cluster architectures.

As a simplified example, a cluster can be perceived in two different ways: either as a system connected to other networks and with specific needs in terms of access, or as a combination of heterogeneous systems (login nodes, computing nodes, storage nodes and so on). In this distributed representation of a cluster, interactions between nodes must be protected and monitored [2]. However, the security of a cluster cannot rely solely on the observation of the communication between nodes. It is also essential to control what operations are performed on the nodes.

The aim of this paper is to present the way in which we addressed the different security needs on a shared computing platform and to measure their impact on performance. The guaranteed security properties are high and we take a worst-case hypothesis: a malicious user can exploit a system vulnerability and gain administrator privileges. Even in this case, the proposed security measures should keep the intruder confined and the other partners protected.

Firstly, we present the security objectives that our HPC cluster requires. Secondly, related works that deal with security of Linux HPC clusters and performance issues are described and compared to the security objectives. Then, Section 4 deals with our configuration of SELinux and how it achieves our security aims. The following section explains the issues encountered during the implementation of our SELinux policy on our experimental computing clusters and the remaining issues with this implementation. Finally, the paper describes the secured production clusters and studies the performance impact of our security solution.

* Corresponding author. Tel.: +33 248484058.
E-mail addresses: mathieu.blanc@cea.fr (M. Blanc), jean-francois.lalande@ensi-bourges.fr (J.-F. Lalande).

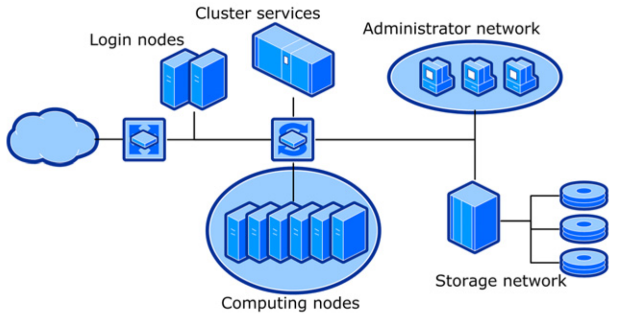


Fig. 1. Graphical representation of a typical cluster.

2. Security objectives and technical solutions

In this section we define the security objectives that we want to achieve in our cluster architecture. First, we make a brief synthesis of mandatory access control before moving on to the security properties that we want to enforce. Then, we describe the technical choices we made in order to meet our objectives.

2.1. Access control

Historically, Linux cluster security is based on access control models that have been deployed on single hosts. Two families of security policy models have been developed: the *Discretionary Access Control* model [3], that delegates the security policy to the users, and the *Mandatory Access Control* model [4], that gives the delegation to a third party authority. The MAC model implements strong security guarantees in an operating system as any user, even with administrator privileges, is constrained by the defined policy. Typical mandatory access control models implemented in current systems are RBAC and DTE. RBAC [5] (Role-Based Access Control) associates roles with users, which restricts their abilities when using a given role. DTE [6] (Domain and Type Enforcement) is more focused on the interaction between the subjects of the system (who could be users) and objects (the data used). The definition of domains grants a set of access permissions to a group of subjects. The definition of types follows the same process for the objects. In the proposal in this paper, the required security needs are closely linked to these notions. Users accessing the cluster have different roles and, depending on their credentials, should be isolated in domains.

For operating systems, the first MAC models proposed are often difficult to implement. For example, the Bell–LaPadula model [4] introduces the "No Read up" and "No Write down" principles to enforce confidentiality in a system. Based on the classification level, a subject of the system that does not have an appropriate clearance level should not be able to read or write an object. This rule is technically difficult to enforce, for example in the */tmp* directory where any process can create a file. The dual model, which ensures system integrity, is the Biba model [7]. The "No Read down" and "No Write up" rule suffers from similar issues when implemented at kernel level in operating systems. This is why the DTE model is more convenient for operating systems as it tries to confine resources that are accessed by a program even for those programs that have "root" privileges.

The implementation of access control mechanisms in Linux clusters has followed the implementation of MAC mechanism in GNU/Linux operating systems. Several attempts like Medusa [8], RSBAC [9], LIDS [10], have led to the emergence of two typical implementations: SELinux, a result of the DTOS project [11], and grsecurity [12]. Both implement the RBAC and DTE models.

2.2. Desired security properties

Here we present the security aims for an experimental shared HPC cluster. These form the basis on which to implement an access control mechanism in our cluster and to build its configuration. They can be summarized in four points:

- Ensure the confidentiality of data uploaded by partners (isolation in [13]).
- Confine user profiles and services so that a malicious elevation of privileges does not compromise the security of the operating system.
- Differentiate public and privileged remote access.
- Secure and control interactions with batch schedulers and authentication mechanisms.

The subsections that follow give details on each point.

User containers and data confidentiality. Users from the same projects should be able to exchange files freely. Hence, there is a particular kind of user group called "containers". These containers represent people working on the same project or users that are granted access by the same administrative procedure (for example a national research agreement). In these containers, any accidental leakage of information due to incorrect permissions is considered harmless.

In an actual cluster, these containers can be implemented with standard Unix groups for a DAC mechanism, or with security labels for a MAC mechanism. Indeed, a model like BLP [4] defines a security label as the combination of one level and one or several categories. Technical details about the use of categories will be given in Section 4.4.

Definition 2.1 (*Container*)**.** In a container A, with $\{a_1, a_2\} \subset A$, a_1 accessing a file f belonging to a_2 does not break the confidentiality of file f.

Which means, in terms of confidentiality:

Definition 2.2 (*Confidentiality*)**.** Data confidentiality means that a user a from container A must not be able to read a file f belonging to a user b from container B, whatever permissions are set on f.

Of course, this does not mean that any user can access all the files of all other users in the same container. Typically, a MAC mechanism will confine users in their container, and then inside a container users can restrict access to their own files using DAC permissions.

Besides, we emphasize confidentiality from among other classical security properties. This is because the most critical scenario against which protection is sought, is the theft of one partner's information by another. Still, the concept of container as it is implemented in our solution will also guarantee the integrity of files, but this is not the primary focus.

Confined users and services. Users and services should be confined in order to prevent any tampering with the security mechanisms. A first example is a malicious hacker exploiting a flaw in a network service in order to gain administrative access to a login node. Exploiting a flaw should not allow him to break the data confidentiality of another container. Another example is a legitimate user downloading a malicious code from the Internet and using it to gain administrative privileges on his node. Even if this succeeds, this user should not be able to access files outside his container.

Definition 2.3 (*Confinement*)**.** A person who obtains unauthorized access because of a system vulnerability must be placed in a particular container, disjoint from legitimate containers. Thus the effects of a security breach can be controlled, and an attacker will be confined in a specific container.

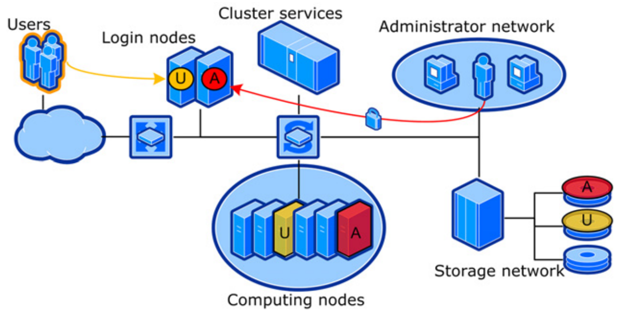


Fig. 2. Different kinds of users log into different containers.

Another requirement of this objective is for administrators to be placed in a specific container. Thus, gaining administrative privileges does not break the confidentiality property.

There are still technical limitations with this kind of confinement. In particular, a vulnerability in the core of the operating system or in the MAC mechanism will allow anyone to break the confinement. These limitations will be discussed in Section 5.

Remote access. There should be two different points of access on the cluster nodes as represented in Fig. 1: a public access for standard users, and an administrative access dedicated to system administrators. These accesses should always be set up using an authenticated and ciphered protocol. Even in the case of a vulnerability exploited in the server that gives an administrative access to an attacker, public access should never allow users to configure the security mechanisms.

Of course, interactive user access is not always enabled. For example, there are some parts of the cluster, like computing nodes, where standard user access has to be limited. These nodes should only be accessed through the batch scheduler as shown on Fig. 1. The same restriction applies to the storage nodes, accessible only through mounted network file systems. These are only examples, as each cluster has its specific areas.

Fig. 2 illustrates how the remote access should place users in their containers (for example U), and how the separate administrative access places privileged users in a specific container (for example A).

Schedulers and authentication services. The security mechanisms implemented on the node must protect the batch scheduler mechanism and the authentication services. For the scheduler, both the job submission and the command execution on computing nodes should be taken into account by the security mechanism. It is especially important to protect and confine the job execution service as this is the equivalent of a remote access on all cluster nodes. For example, if there is a vulnerability on the cluster operating system, a malicious user could launch a command exploiting it on all nodes by just a batch submission. That is why user jobs should be executed in a confined context on the computing nodes.

For authentication services, user's credentials should be protected using a MAC mechanism.

2.3. Technical choices

In this part we discuss the limits of current solutions, and the technical choices we made in order to implement our aims. First, we present the way in which we overcame the limitations in DAC with the deployment of SELinux. Then, with respect to the standard SELinux policy, we address the main issues regarding the confinement of users, that concern network services and services specific to HPC environments.

The limits of DAC. As indicated previously, DAC is an access control model in which the owner of a file is responsible for the access permissions on this file. In order to create containers, specific Unix groups have to be created. Each group contains users belonging to a specific project, division or company. It is then possible for particular folders to be created called "lock folders" for which group ownership is given to a specific container, and permissions to be set so that only that group can traverse the folder. In this way we have effectively created a file container, and the files in it can only be accessed by users belonging to this container.

However, this method only works if we exclude the risk of software vulnerability. In the event of a user successfully exploiting a vulnerability and gaining access to the *root* account, he will be able to read any folder or file on the system. This breaks the confidentiality property defined previously, and is the main limitation of the DAC model. Indeed, the *root* account cannot be confined or limited in its rights. Hence, we need a MAC model in order to enforce a security policy, and limit superuser privileges on a per-process basis.

Constraints from the environment. Usually, HPC clusters, such as those considered here, are so big that they require constant hardware and software maintenance. Therefore, the cluster is delivered by the builder including software and maintenance. In our case, the operating system installed on the cluster is based on RedHat Enterprise Linux. The kernel is heavily modified in order to provide additional functionalities, such as specific network technologies and distributed filesystems. Thus, we cannot easily add our own modifications to the kernel, integrate a new security patch such as grsecurity, or propose a specific system built for security as described in [14,15].

However, the RedHat kernel does integrate the SELinux mechanism with a choice of supported security policies. That is why we chose to build our access control solution on SELinux.

Security-Enhanced Linux. Security-Enhanced Linux or SELinux is a security mechanism integrated into the Linux kernel [16,17]. Traditionally, the security of Unix-like operating systems relies on the Discretionary Access Control (DAC) [3]. In this model, users control the access permissions for their own files. But SELinux provides a Mandatory Access Control (MAC) mechanism [4,7]. With MAC , access permissions are enforced by a separate access control policy written by a security administrator.

SELinux operates by associating security contexts with processes and files. Then, the access control policy states how security contexts are allowed to interact with each other. A security context is usually composed of three mandatory identifiers and two optional ranges. The three identifiers are the SELinux user (independent from a system user), the role and the type. The two ranges are the levels from the Multi-Level Security (MLS) model and the categories from the Multi-Category Security (MCS) model. One example of a context is (`root, system_r, unconfined_t, s0:c0.c1023`). Here `root` is the SELinux user (managed separately from the standard system users), `system_r` is the SELinux role (giving access to a set of types), `unconfined_t` is the type (this shows what the resource is), and the final part is the sensitivity (`s0`) combined with a set of categories (here `c0.c1023` if the set is of all categories). This context would typically be assigned to a process belonging to the root user and would not be restricted in its rights. This is reflected by the `unconfined_t` type. Explaining the inner workings of SELinux is out of the scope of this article, for further information refer to [18,19].

The main benefit of an operating system controlled by SELinux is a strong protection mechanism that is extremely hard to defeat. In a previous work [20], we presented a honeypot project, i.e. a dedicated computer exposed to attacks from the Internet. It has been deployed for two years with open access for intruders. Secured by SELinux, the honeypot has never been reinstalled and the system has never been damaged. This demonstrates that even if a malicious user has access to a SELinux protected cluster, the corruption of the system and the theft of information are hard challenges to overcome.

The advantages and drawbacks of SELinux have been widely debated but agreement can be reached on one point: SELinux can be very complex to deploy if a specific policy has to be created to enforce specific security properties. One single error in the configuration can make a system unusable and it is an onerous task to write a consistent and complete configuration from scratch. In order to be efficient, the SELinux configuration must include only what is necessary and sufficient, that is to say that it should be complete but it should not include any rule allowing an unnecessary action for a particular user or process. However, the availability of pre-installed SELinux configurations has simplified its use. The default SELinux policy provided in current Linux distributions is quite robust and usable, and tools are provided to write complementary policy modules. These modules allow a specific company to add access permissions for specific applications or usage scenarios [21].

Limitations of the standard SELinux policy. The cluster operating system is configured using a SELinux policy named *targeted* in a standard way. This policy is quite light, easy to apply and needs few adjustments for a typical server setup. However, in our case, this policy suffers from several shortcomings.

Firstly, this policy does not include the possibility of creating confined user profiles. This means that any user on the system is in the `unconfined_t` context. Thus, only DAC permissions protect the users and if someone gains root access, he can break our desired security properties, especially 2.2 as the *root* user can read the files of any user.

Secondly, this policy does not control the context of the remote access service (SSH), whereas this service is greatly exposed to attacks. It is the typical service that can be contacted from outside the company. In the standard policy, any incoming user is labeled the same way by SELinux. In our case, we require that the different kinds of users that may enter the cluster using SSH be isolated, if possible, in order to restrict to the minimum the permissions that are given to each user. This has to be setup if we want to ensure property 2.3.

Thirdly, the usual cluster services like network authentication and batch scheduling are not controlled in this policy. For example, in the case of the Kerberos mechanism, authentication relies on user ticket caches. These should be protected in a way that ensures that only users from the same container may have access to it, even if the system is compromised. In this way, it is possible to prevent a malicious user from stealing another user's credentials in order to authenticate and try to access his files.

Hence, we have to build and provide a complementary policy for SELinux that controls these various points. The method used to build such a policy is detailed in Section 4.

2.4. Synthesis

As stated previously, our solution cannot rely on the DAC model as it is too weak to provide confinement in the event of a successful attack against the operating system. That is why we need a MAC model. As explained in 3, there are several implementations of MAC for Linux systems. However, SELinux is the optimal choice given our environment constraints. Using mechanisms like grsecurity or RSBAC would require new patches to be applied to the kernel and as we explained, this is very difficult in our case. Cluster security solutions also exist [22,23,15], but they do not include all the features integrated in SELinux, in particular the MCS model. Furthermore, studies such as [24] seem to conclude that the SELinux implementation is robust. However, performance aspects of SELinux have not been widely studied, and for these reasons, we had to carefully study those aspects in this paper.

Thus, in order to guarantee confidentiality, our solution must support the creation of containers to prevent unauthorized users from accessing files in the container, even if they maliciously obtain a *root* access. That is why we chose to use the standard SELinux policy included in the RedHat operating system, with the addition of several modules that we have built.

Finally, we use the MCS mechanism included in SELinux in order to associate users with categories, thus creating the containers needed to ensure the confidentiality property.

3. State of the art

This section presents the papers related to the security enforcement in HPC environments. It describes in what way previous works have addressed our security requirements and shows that our present objectives are not covered by these works. It also describes related works that are connected to our proposal but that do not meet our technical constraints as they deal with virtualized environments and grid middleware.

3.1. Security in HPC clusters

One of the first approaches in integrating MAC in clusters was DSI [14] by Ericsson. This was the first to provide a policy deployment framework specifically aimed at HPC architectures. The main drawback was that a specific policy language had to be designed, and this was far from SELinux in terms of features. Besides, the development of DSI was discontinued a long time ago, hence it is not intended to work on recent Linux distributions.

Some companies offer preconfigured operating systems for HPC clusters, like NimbusOS from Linux Labs International. Unfortunately, no precise descriptions of the policies are available. The proposed policy may be the standard targeted SELinux policy that provides confinement for network services; however it does not include rules to confine the users. Regarding our objectives, such a solution is not sufficient. As described in Section 2, we want to guarantee isolation even if a malicious user gains administrator privileges. Moreover, such a solution imposes the use of a particular operating system with integrated security, whereas we already have our own cluster operating system which we want to enhance from a security perspective.

In a more recent paper, the authors of [15] present a work that is similar to our paper. They propose a full security architecture for a clustered system. Their goal is to bring Reliability, Availability, Serviceability to a cluster by integrating HA-OSCAR [23] and DSI [22,25]. HA-OSCAR provides a transparent architecture to deploy or re-deploy systems into a cluster. Each service is monitored by a daemon and is repaired in case of failure. DSI implements a MAC mechanism at kernel level, allowing isolation of processes. The policy is similar to an SELinux policy with the use of classes for subjects and objects. Performance results presented in [25] are very good for system calls (impact lower than 1%). Another advantage of this solution is that it can control network operations. Nevertheless, the proposed solution would be difficult to integrate as it is in our cluster operating system. Firstly, it requires the associated security context to be written in the program files, and secondly, the set of system calls that can be controlled is relatively reduced.

3.2. Performance and security

A large number of works have investigated the impact of security solutions on the performance of operating systems. For a high performance cluster dedicated to computing, performance measurements are critical.

In [26], an empirical study describes the impact of grsecurity and SELinux on the performance of the system. This study shows that grsecurity is lighter than SELinux, in particular regarding memory access (no overhead for process creation or arithmetic operations). Nevertheless, the authors chose not to compare the performance to a standard system without any security mechanisms. However, for recent systems that have multi-core architectures, the overhead becomes very small [27].

The solution proposed by Leangsuksun et al. [15] presents heterogeneous results regarding performance benchmarks. The authors report a small overhead for file manipulation operations controlled by DSI, which is a very similar result to the performance impact of SELinux. On the other hand, DSI can induce more than 20% of overhead for the control of interactions between nodes, as the network packets may be filtered.

3.3. Performance and virtualization

For the HPC community, the computation performance of a cluster is an important subject of investigation, especially if security measures are implemented. In [28], the XEN paravirtualization technique is evaluated on MPI processes run on a cluster of four nodes, each with four processors. The main result is that the virtualization has no significant impact on the performance. Nevertheless, the deployment of such a technology is not trivial and the security benefit is not clear, contrary to the benefits brought by SELinux.

A similar work to the one proposed in this paper, but based on virtualization techniques, is described in [13]. Its aim is to achieve strong isolation between users in what is called "containers". The paper discusses how to set up containers to obtain a good trade-off between performance and isolation. This isolation is applied to both resources (e.g. files) and system objects (e.g. memory, cpu). The results compare different virtualization techniques to two standard Linux kernels. Depending on the benchmark software, the results show a loss of 0% to 30% in disk performance and no difference for network throughput. The results are good for lightweight virtualization techniques, for example using VServer. As the kernel is shared, the system calls have no extra cost. Nevertheless, setting up a container requires the inclusion of all the data and the binaries used by the user that will be confined in this container. This is difficult to set up and the reconfiguration of such a system for a new partner using the cluster cannot be easily performed.

3.4. Grid computing

HPC cluster security is often related to services provided in grid infrastructures. Security studies regarding grids focus on the middleware that manages the authentication, delegation of rights and authorization mechanisms. Advanced techniques allow the policies to be dynamically adapted to the context of the service requests [29]. This is technically possible because the middleware enforces the majority of the security mechanisms and can adapt its configuration dynamically. On an operating system level, grids do not provide new security solutions and often rely on classical MAC mechanism of virtualization techniques. For example, [30] uses virtualized images, deployed on demand, to isolate users and confine the software installed for these users.

Grid security mechanisms are out of the scope of this paper. In fact, we are focusing on the HPC cluster level that does not implement specific APIs in order to access the computing resources. Nevertheless, recent works like [31] try to introduce the notion of "elastic clusters" that combine grid structures and cloud computing elasticity. However, regarding security aspects, the proposed solutions rely on classical grid and cloud security features and do not propose new work.

3.5. Building working policies

Setting up the security mechanisms presented is not an easy task. The optimal configuration of some components may be impossible to compute. As reported by Leangsuksun et al. [15], the security components are mainly patches that have to be applied to the kernel. Finding the right patches and a working order in which to apply them is complex. Furthermore, when the components are correctly installed, the right security policy has to be built [32].

Building an access control policy is often accomplished through a learning process. In [26] the `gradm` and `audit2allow` tools are used to build the grsecurity and SELinux policies. However, the resulting policies may contain conflicting rules because of a bad learning mechanism. Moreover, what has been learned could conflict with the rules entered by hand representing the security properties that the administrator wants to enforce. Thus, efforts have to be made to find such conflicts or to check if the desired properties are well implemented. For example, checking if a Trusted Computing Base is not violated is a complex task [33].

Another paper [24] contains an interesting study about the impact of a random bit error introduced in the binary file of the loaded policy of SELinux. In 33% of the cases, the SELinux module rejects the policy because the inconsistency has been detected. The authors have identified 8 cases where the introduced flipped bit leads to dangerous vulnerabilities in the desired security properties (information flow or possible violation of integrity). Nevertheless, the results consolidate the administrators trust in SELinux as it seems very difficult to voluntarily perform such an attack.

3.6. Synthesis

The aim of this paper is to present the integration of MAC in an existing operating system for HPC clusters, with several constraints:

1. to integrate the security objectives described in Section 2;
2. to refrain from modifying the operating system from the cluster builder, in particular the kernel, otherwise commercial support can not be guaranteed;
3. to have a minimal impact on performance.

As explained previously, related works do not cover all the security requirements for our clusters. Thus, we will propose a new solution in Section 4 that extends the RBAC and DTE models in order to achieve the extra security needs of Section 2.

4. Proposed security solution

This section describes the policy that has been designed to fulfill the security aims expressed in Section 2. First the modularity of the solution is expressed, using the new SELinux module architecture. Then, the section describes the methodology used to create the policy file. At the end, we discuss the possible limitations of the proposed solution and we show that the implementation is quite flexible.

4.1. Implementation as policy modules

When SELinux was introduced in the Linux kernel as a patch [34], the policy was only composed of a large monolithic text file. It was written according to the SELinux grammar, and compiled by a tool called `checkpolicy` in order to obtain a binary representation of the policy. This binary blob was then loaded into the kernel, and applied as defined by the SELinux mode, either *permissive* or *enforcing*.

From this monolithic form, the policy was then split into several text files, that were combined in order to produce the policy file before compilation. After that, binary modules were introduced. With this modular binary policy representation, installing all the policy source files was no longer necessary. A binary policy module file could be directly distributed and loaded. With the `semodule` command, an additional binary module could be merged with a *base* module (containing base types and policy) to produce the new binary policy. This could be done at any time on a running system and the policy loaded dynamically. Leveraging this feature, our approach was to produce binary modules only, without any modification to the Red Hat Linux Enterprise targeted policy, which included the Multi-Category System (MCS). We did not modify the base module and the additional modules present in the targeted policy, but only added two modules to define the new contexts and user profiles specific to our security aims.

The two modules we implemented follow the two following objectives:

- **ccc_guest** implements the confined user profiles called *guest* for SSH access and *xguest* for graphical access;
- **sshd_admin** implements the two levels of SSH access, public and administrative. Specific security contexts are associated with two separate SSH servers.

4.2. Policy build process

There are two ways to obtain the SELinux policy for a piece of software. The ideal way is to integrate the policy definition in the application development process. This produces a policy that is both consistent and complete. However, this requires the developers to know exactly what their applications do from an operating system point of view (for example which system calls are used) and they do not always have that knowledge. Another issue is that SELinux is currently not widely deployed, so it is still not that relevant for software developers to take time to learn how to write a SELinux policy module. But in the future it may become more useful.

A more realistic learning process is described in [21]. A system is set up with the base SELinux policy running in *permissive* mode. This means that SELinux will not forbid any action, it will only log system calls that should have been forbidden. Then each software that needs specific rules is run in the most typical usage scenarios to produce relevant SELinux logs that can be converted into relevant rules. In this process, the administrator should carefully review the produced rules. Indeed, the learning process will eventually capture useless rules that can expose critical parts of the operating system and make the confinement ineffective.

In order to write our new SELinux policy modules, we followed the second approach as we do not necessarily have access to the source code of every application running on our shared clusters. This is especially true in the case of partners applications. Partners may not want to share the algorithms or techniques used to implement their computing codes. Therefore, we have to take a practical approach where we run the code and analyze any failures through the study of SELinux logs. Often this is not as automatic as described in [21], either because we obtain very cryptic logs, or sometimes because we do not obtain any logs at all. Thus, the systematic approach we used in the development of our SELinux policy was as follows:

1. Load the default "targeted" policy on our test platform, based on RedHat Enterprise Linux 5.4.
2. Load an initial or development version of our policy modules.
3. Run applications in standard usage scenarios.
4. Identify execution issues and broken functionalities.
5. Review SELinux logs (by default in `/var/log/audit/audit.log`).
6. Convert the relevant logs into rules using `audit2allow`.
7. Review the rules and integrate the relevant ones in the sources of the policy modules.
8. Go back to step 2.

Steps 5 and 6 are generally performed with the `audit2allow` command which produces a good digest of the SELinux logs. Afterwards we analyze the rules produced by this command, and integrate the relevant rules into the sources of our policy modules. This is currently performed only on one node of the cluster at a time, but it could be performed on many nodes at the same time. We could then centralize the logs and perform correlation using techniques and tools such as proposed in [35] which can guarantee a more unified policy. Nevertheless, auditing the nodes independently ensures the most accurate policy will be obtained because each node can receive its specific policy module.

4.3. The policy components

The resulting policy of our writing process is composed of the standard "targeted" policy and two modules called `ccc_guest` and `sshd_admin`.

`ccc_guest`. By default, with the targeted policy, all users are placed in the context `(user_u, system_r, unconfined_t)`. The module proposes specific security contexts to confine the users: `ccc_guest` and `ccc_xguest`. The first one is associated with SSH connections, and the second one is associated with X11 sessions. They were originally derived from the `guest` and `xguest` profiles of Fedora 10, provided by Dan Walsh [36] and limit the administrative privileges accorded to users. They were subsequently adapted to our specific needs following the process described previously.

The objective of these confined user profiles is to limit to the minimum the administrative privileges accorded to users. For example, a standard user logged on the system via SSH will have the context `(ccc_guest_u, ccc_guest_r, ccc_guest_t)`. When trying to exploit a vulnerability in any system command or service, a malicious user may obtain a `root` access. Nevertheless, he will not be allowed to interact with other security contexts of the system as the policy does not allow `(ccc_guest_u, ccc_guest_r, ccc_guest_t)` to do so. Listings 1 and 2 are a sample of the definition of the guest user profile.

```
interface('cea_unprivileged_user','
  userdom_unpriv_login_user($2)
  userdom_basic_networking_template($2)
  kernel_read_network_state($1)
  corenet_udp_bind_all_nodes($1)
  corenet_udp_bind_generic_port($1)
  corenet_tcp_bind_all_nodes($1)
  corenet_tcp_bind_generic_port($1)
  cea_userdom_login_user($2)
')
```

Listing 1: Sample from `ccc_guest.if`

```
policy_module(ccc_guest, 1.0.1)
cea_unprivileged_user(ccc_guest_t, ccc_guest);
cea_unprivileged_user(ccc_xguest_t, ccc_xguest);
userdom_restricted_xwindows_user_template(ccc_xguest);
gen_user(ccc_guest_u, user, ccc_guest_r ccc_xguest_r, s0, s0 -
     mls_systemhigh, mcs_allcats)
```

Listing 2: Sample from `ccc_guest.te`

`sshd_admin`. In the targeted policy, the SSH server is not associated with a particular context, it is actually executed in the

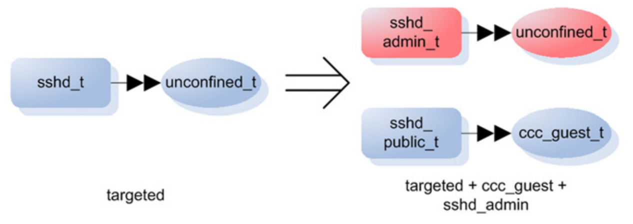


Fig. 3. Two levels of SSH access.

`unconfined_t` type. As the SSH service must be accessible to all the cluster partners from the Internet, it is heavily exposed to attacks. Moreover, any process in the same context can attack the SSH server. The proposed policy module `sshd_admin` achieves two aims. First, we want to confine the SSH server so that if an attacker exploits a vulnerability in it, he will only reach a confined profile. Second, the obtained SELinux role for the SSH server should be limited to what is strictly necessary. That is why we have created two different contexts for two SSH servers running at the same time (on different TCP ports): `sshd_public_t` and `sshd_admin_t`.

A first context `sshd_public_t` has been created for the partner SSH access. This context can only transit to the `ccc_guest_t` type. The second context `sshd_admin_t` is restricted to administrative access and can transit to the `unconfined_t` type to obtain full access on the processes and resources in this context.

The targeted policy and the proposed policy including the `ccc_guest` and `sshd_admin` are compared in Fig. 3. Nevertheless, these two modules do not achieve one of the security aims: the isolation of users which is presented in the next section.

4.4. Users isolation

As presented briefly in Section 1, MCS stands for Multi-Category System. This is a part of the security model from Bell and LaPadula [4]. The principle is to associate categories with users and files. To read a specific f file, a user U must at least be associated with all the categories of f. In other words, for U to access f, the category set of f must be a subset of the categories of U:

$$U \text{ can read } f \Leftrightarrow Cat(f) \subset Cat(U).$$

In the targeted policy for SELinux, the standard available range of categories is c_0 to c_{1023}, thus a total of 1024. In order to create containers for users, as defined in Section 2, a category is associated with a set of users who have the same origin (specific partnership, a research program, a particular university). This solution provides isolation between users of different categories, as shown in Fig. 4. It supposes that each file should only have one category and that DAC permissions are properly set up.

The categories provide an additional benefit to finely administrate the rights of a specific partner in one container of category c_1. A special category c_{1000} could be applied to this partner and some of the partners files: this results in this partner being able to read any file that has category c_1 but no other partner of the container will be able to read the file tagged with categories c_1, c_{1000}. Thus, the number of subsets isolating users and files is very high (2^{1024}) and sufficient to implement any isolation policy.

Then, when a user logs on the cluster, he receives his allowed set of categories. He will be able to access only those files tagged with a subset of these categories as in Fig. 4. Our intended policy regarding files is that each file should only have one category to ease administration. Of course, the categories are only an additional restriction regarding the DAC . If this user has the right category for a file, but does not have the standard read right (`r`), he will not be able to read the file.

In the case of a malicious user exploiting a vulnerability in a program or service, SELinux will ensure that the fraudulent access obtained will be confined. There are two main scenarios in this case:

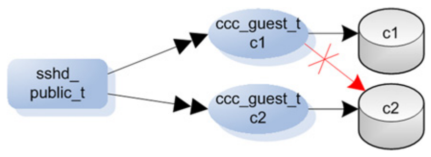


Fig. 4. File access control with MCS .

- The malicious user exploits a vulnerability in a system service or in the remote access server (SSH). All services run with specific security contexts that limit their access permissions on the system. Besides, they run without any assigned category, so they are outside any legitimate user container. In the case of a successful vulnerability exploitation, an attacker will actually obtain an access to a program that runs with one of these confined contexts. According to the SELinux policy, the attacker will not be able to change this security context, even if he gains access to the *root* account.
- An attack can be performed against a user process containing a vulnerability. In this case, the attacker will gain access to the security context of this user, and will be able to access the same files as this specific user. In the worst case, if a malicious user obtains a *root* access in this context, he will be able to access all files with the same SELinux category, and thus will be restricted to this container.

5. Complex challenges

The section that follows presents the complex technical issues that have been encountered in the process of writing the proposed SELinux configuration.

5.1. NFS homes and user containers

File security contexts and categories are stored in special extra attributes on the disk. In a large HPC cluster, the data is accessed via networks mounts. The support of such extra attributes across the network, for example over NFS , is very experimental as pointed out in a previous work [37] and thus not available on the Red Hat Enterprise Linux 5.4 operating system. It is possible to apply a specific context to a file system with the `mount` command, as shown in this example:

```
mount —t nfs —o context=system_u:object_r:nfs_t:c1 server:/
    home /home/c1
```

However, this technique does not work because of the way the Linux kernel handles the NFS filesystems. When many filesystems are mounted from the same IP address, the kernel applies the same options to all the mount points. Thus it is not possible to specify different security contexts for several NFS filesystems mounted from the same server.

To solve this issue, a workaround is proposed: the context and MCS category are applied directly to a local directory, and the NFS filesystem is mounted under that directory. For example, the experimental cluster uses paths like `/c1/home/user`. Here the `c1` directory with the category $c1$. `home` is the mount point for the sub directory of the NFS filesystem containing the home directories for users in the $c1$ container. This workaround should only be temporary. Specification for *Labeled* NFS have been published [38], and work on the integration of SELinux in NFS v4 is in progress [39].

5.2. Lustre

The Lustre filesystem driver seems to support SELinux labels and categories. Our investigations have shown that Lustre interacts very badly with SELinux which confirms what is reported in [40].

Table 1
Some production clusters at CEA.

	Defense app. cluster	Computing cluster	Visu. cluster
Total nodes	4300	1140	50
CPU/node	4 Xeon 7500	2 Nehalem Quad	2 Xeon 5450
RAM/node	128 GB	24 GB	64/128 GB
Network FS	20 PB	500 TB	100 TB
User accounts	…	≈4000	
Avg. running jobs	…	100–150	10
Avg. queued jobs	…	100–1000	N/A

For example, when first mounting a Lustre filesystem, the files are seen labeled with the security context `unlabeled_t`, which is actually the default type when SELinux contexts are not supported. When creating a new file, it receives the type `file_t`, the default type when contexts are supported. But when remounting the filesystem, the newly created file will be unlabeled like the other files. Moreover, as it is not possible for a sub directory to be mounted from a Lustre mount point, we cannot apply the same workaround as for NFS filesystems.

At this time a Lustre filesystem has to remain declared as a shared zone, not isolated. The only way to resolve this issue is to implement proper support for SELinux in the Lustre driver, which is not in the current roadmap [41].

5.3. Remaining issues

As powerful and configurable as SELinux is, it is always important to remember that it cannot protect against certain threats. Essentially, in the case of a Linux kernel vulnerability, the confinement mechanism of SELinux can be broken, as SELinux is a part of the kernel. So far, several pieces of code have been published that exploit Linux kernel vulnerabilities and at the same time defeat the SELinux confinement. In the discussion of [42], Spengler explains that SELinux should be used with other security tools like grsecurity/PaX systems. The discussed exploit has used the KERNEXEC permission to execute an arbitrary code, located in the userland, in the kernel. This way, the RBAC protection brought by SELinux has been disabled.

However, with SELinux it is possible for the exploitation vector to be blocked for a kernel vulnerability, or the result of a successful attack to be mitigated. Regarding the exploit codes that have been published for recent kernel vulnerabilities, they usually only give more privileges to the shell of the user running the exploit. For example, the widely known exploit code for the vulnerability in the `sys_vmsplice()` kernel function merely changes the UID of the user shell to 0 (the `root` UID). We tested this vulnerability and the user shell under `root` UID remains in the `ccc_guest` context and therefore remains confined.

6. Experimental results

Before we reach the conclusion of this article, we will describe the HPC clusters where our SELinux policy is being deployed and will present the results of the impact of SELinux on the I/O performance of a computing node.

6.1. CEA clusters

The SELinux configuration proposed in Section 4 is currently intended for deployment on two clusters open to partnership. The first cluster is dedicated to scientific computing, and is composed of 1068 computing nodes (including 2 interactive nodes), 48 GPU nodes and 24 I/O nodes. It achieves a peak performance of 100 Tflops CPU and 200 Tflops GPU. The second cluster is dedicated to data reduction and visualization. It is composed of 38 computing nodes (each node equipped with a Quadro FX 5800), 2 login nodes and 10 I/O nodes.

A new cluster called "Tera-100" has been recently set up, and is dedicated to defense applications. It is presented here as a reference. The security measures deployed on this cluster will not be detailed here because of its strategic nature. Its peak power is 1.25 Petaflops as established in the TOP 500 [43]. It is composed of 4300 nodes, with a total of about 140,000 Intel Xeon 7500 cores. It integrates 300 TB of memory, and 20 PB of storage. This storage achieves a bandwidth of 500 GB/s, which is currently a world record.

Table 1 compares the different clusters. The storage is not proportional to the number of nodes, as the storage depends on the use of the cluster and the applications deployed on it. The number of jobs given in Table 1 gives an idea of the use of the clusters. On average, on the computing clusters, a job requires between 10 and 100 nodes to run, and takes several hours to complete. This explains the large number of queued jobs waiting to be processed. Occasionally some jobs can even run on the whole cluster.

6.2. Performance benchmarks

One of the aims of this paper was to present the performance impact on the cluster of the proposed solution. Several benchmark runs were conducted, essentially on the Input/Output performance on SELinux-enabled computing nodes. Indeed, as SELinux mainly adds checks in system calls, the I/O operations are the greatest points of impact. Two types of benchmarks were performed on nodes similar to the nodes of the presented production clusters, with a small Lustre filesystem composed of 1 MDS and 2 OSSs: IOR tests the throughput of MPI Input/Output operations in parallel file systems (”separated” means that all tasks work on different files, ”grouped” means they work on the same file) and bonnie++ [44] evaluates the performance of operations on file metadata, for example creation and deletion of files.

The Figs. 5 and 6 show the data throughput benchmark performed with IOR. According to these first two graphs, the impact of SELinux remains relatively low.

In Fig. 7 the metadata operations per second are evaluated with bonnie++, and we can observe that SELinux has a great impact on file creation time (about 30%). This test was repeated more than ten times to be sure that this result was not just a specific case. This is probably related to the Lustre issue described in Section 5. More precisely, when creating a new file, SELinux asks the Lustre filesystem driver to set a specific context on this new file, as the corresponding function exists in the Lustre driver. However, this function is actually not implemented yet, and a call to it probably results in a timeout. This is currently under investigation by our team contributing to the Lustre source code.

As a complementary check we performed the bonnie++ benchmark on a local filesystem (not a Lustre filesystem) and observed a maximum 10% impact from SELinux on file creation time. This impact is relatively low and is the expected value for a system running SELinux. Hence, this indicates that the performance impact on Lustre filesystems really comes from an issue in the Lustre driver, rather than in the way the kernel handles file creation.

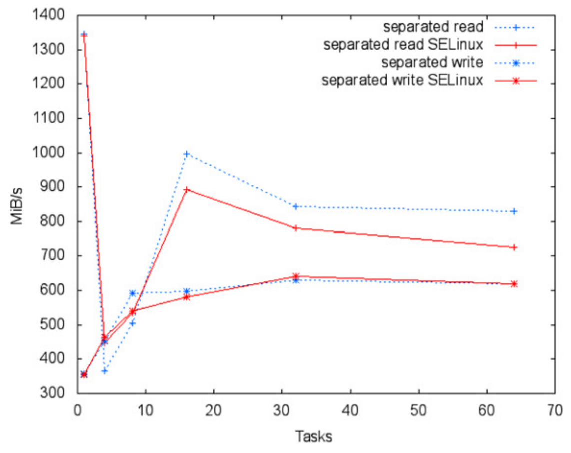


Fig. 5. Separated MPIIO benchmark.

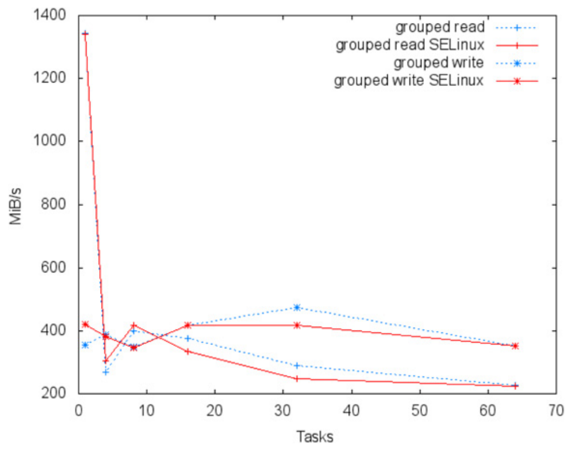


Fig. 6. Grouped MPIIO benchmark.

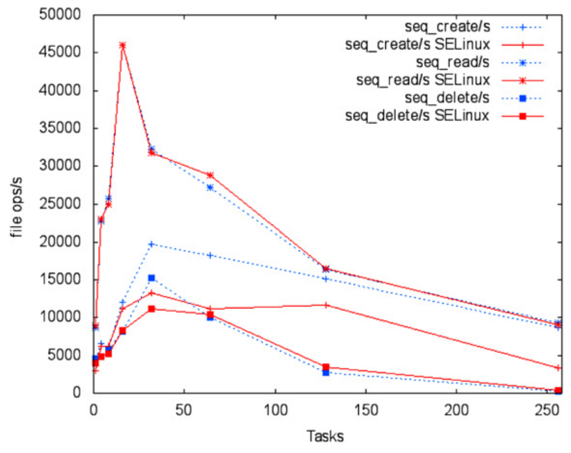


Fig. 7. Sequential metadata benchmark.

7. Conclusion and future work

In this article we have addressed the issue of confidentiality on a shared cluster, with a solution that offers strong protection. A detailed analysis of the implementation using a SELinux policy has been given. The built policy enforces strong security properties for NFS and SSH services and isolates the users, even if a vulnerability is exploited. The paper then presents the most complex technical issues encountered and how we propose to mitigate them, as well as the remaining issues. Finally, we present the performance impact of the proposed solution.

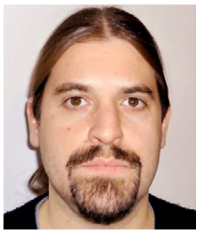
While performing this study of the definition and the deployment of a SELinux policy, we were faced with several challenges. First, we had to perfect our understanding of the inner working of SELinux and its policy grammar. Then we had to check how the current administrative procedure could be modified to integrate well with the activation of SELinux. Indeed, usual cluster administration can sometimes be very contradictory to the most basic rules of the SELinux policy, and this does not necessarily indicate a lack of security in usual cluster administration procedures.

The next steps of this work will be to solve the incompatibility between Lustre and SELinux, to add the support of contexts on NFS filesystems, and to propose solutions to efficiently perform the regular adaptation of the policy to the new applications deployed on our clusters. This last task is the most difficult problem to solve: as the users needs change over time, the installed policy should be adapted or updated. For this evolution, the administrator needs new tools to perform safe updates and to verify that the new policy will not introduce security issues.

References

[1] W. Yurcik, G.A. Koenig, X. Meng, J. Greenseid, Cluster security as a unique problem with emergent properties: issues and techniques, in: 5th International Conference on Linux Clusters, Austin, Texas, USA, 2004.
[2] M. Pourzandi, D. Gordon, W. Yurcik, G. Koenig, Clusters and security: distributed security for distributed systems, in: International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Cardiff, UK, 2005, pp. 96–104. http://dx.doi.org/10.1109/CCGRID.2005.1558540.
[3] B.W. Lampson, Protection, in: The 5th Symposium on Information Sciences and Systems, Princeton University, 1971, pp. 437–443.
[4] D.E. Bell, L.J. La Padula, Secure computer systems: mathematical foundations and model, Technical Report M74-244, The MITRE Corporation, Bedford, MA, May 1973.
[5] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, C.E. Youman, Role-based access control models, IEEE Computer 29 (2) (1996) 38–47. http://dx.doi.org/10.1109/2.485845.
[6] W.E. Boebert, R.Y. Kain, A practical alternative to hierarchical integrity policies, in: The 8th National Computer Security Conference, Gaithersburg, MD, USA, 1985, pp. 18–27.
[7] K.J. Biba, Integrity considerations for secure computer systems, Tech. Rep. MTR-3153, The MITRE Corporation, Jun. 1975.
[8] Medusa DS9, Medusa ds9 security system, 2006. URL http://medusa.fornax.sk/.
[9] A. Ott, Rule set based access control as proposed in the 'generalized framework for access control' approach in Linux, Master's thesis, Universität Hamburg, Nov. 1997. URL http://rsbac.org/doc/media/linux-kongress/index.html.
[10] P. Biondi, Security at kernel level: LIDS, in: Free and Open source Software Developers' European Meeting, 2002. URL http://www.lids.org/document/fosdem-ksec.pdf.
[11] S.E. Minear, Providing policy control over object operations in a mach based system, in: The 5th conference on USENIX UNIX Security Symposium, USENIX Association, Salt Lake City, Utah, 1995, pp. 141–156.
[12] B. Spengler, Detection, prevention, and containment: a study of grsecurity, in: Libre Software Meeting, Bordeaux, France, 2002.
[13] S. Soltesz, H. Pötzl, M.E. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors, European Conference on Computer Systems 41 (3) (2007) 275–287. http://dx.doi.org/10.1145/1272998.1273025.
[14] M. Pourzandi, A. Apvrille, E. Gingras, A. Medenou, D. Gordon, Distributed access control for carrier class clusters, in: Parallel and Distributed Processing Techniques and Applications Conference, Las Vegas, Nevada, USA, 2003, pp. 132–137.
[15] C. Leangsuksun, A. Tikotekar, M. Pourzandi, I. Haddad, Feasibility study and early experimental results towards cluster survivability, in: 5th International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2005, pp. 77–81. http://dx.doi.org/10.1109/CCGRID.2005.1558537.
[16] S. Smalley, C. Vance, W. Salamon, Implementing SELinux as a Linux security module, Tech. Rep., NSA, Dec. 2001.
[17] P. Loscocco, S. Smalley, Integrating Flexible Support for Security Policies into the Linux Operating System, 2001, pp. 29–42. URL http://portal.acm.org/citation.cfm?id=647054.715771.
[18] S. Smalley, T. Fraser, A Security Policy Configuration for the Security-Enhanced Linux, Tech. Rep., NSA, Dec. 2000.
[19] Frank Mayer, Karl MacMillan, David Caplan, SELinux by Example, Prentice Hall, 2006.
[20] J. Briffaut, J.-F. Lalande, C. Toinard, Security and results of a large-scale high-interaction honeypot, Journal of Computers, Special Issue on Security and High Performance Computer Systems 4 (5) (2009) 395–404.

[21] D. Walsh, A step-by-step guide to building a new SELinux policy module, Red Hat Magazine (2007).
[22] M. Pourzandi, I. Haddad, C. Levert, M. Zakrzewski, M. Dagenais, A distributed security infrastructure for carrier class Linux clusters, in: Ottawa Linux Symposium, 2002, pp. 439–450.
[23] I. Haddad, C. Leangsuksun, S.L. Scott, HA-OSCAR: the birth of highly available OSCAR, Linux Journal (115).
[24] Michael Ihde, Tom Brown, An experimental study of file permission vulnerabilities caused by single-bit errors in the selinux kernel policy file, UIUC Military Geographic Information System meeting.
[25] M. Pourzandi, A new distributed security model for Linux clusters, in: The annual conference on USENIX Annual Technical Conference, Boston, MA, USA, 2004.
[26] M. Fox, J. Giordano, L. Stotler, A. Thomas, Selinux and grsecurity: a side-by-side comparison of mandatory access control and access control list implementations, Tech. Rep. URL http://www.cs.virginia.edu/~jcg8f/GrsecuritySELinuxCaseStudy.pdf.
[27] C. Wright, C. Cowan, J. Morris, S. Smalley, G. Kroah-Hartman, Linux security modules: general security support for the Linux kernel, in: Foundations of Intrusion Tolerant Systems, IEEE Computer Society, Los Alamitos, California, 2003, pp. 213–226. http://dx.doi.org/10.1109/FITS.2003.1264934.
[28] L. Youseff, R. Wolski, B. Gorda, C. Krintz, Evaluating the performance impact of xen on mpi and process execution for hpc systems, in: First International Workshop on Virtualization Technology in Distributed Computing, VTDC 2006, IEEE Computer Society, 2006, http://dx.doi.org/10.1109/VTDC.2006.4.
[29] Y. Demchenko, O. Mulmo, L. Gommans, C. de Laat, A. Wan, Dynamic security context management in grid-based applications, Future Generation Computer Systems 24 (5) (2008) 434–441.
[30] M. Smith, M. Schmidt, N. Fallenbeck, T. Dörnemann, C. Schridde, B. Freisleben, Secure on-demand grid computing, Future Generation Computer Systems 25 (3) (2009) 315–325.
[31] G. Mateescu, W. Gentzsch, C.J. Ribbens, Hybrid computing where hpc meets grid and cloud computing, Future Generation Computer Systems 27 (5) (2011) 440–453.
[32] S. Smalley, Config uring the SELinux Policy, Tech. Rep. NAI Labs Report #02-007, 2003. URL http://www.nsa.gov/research/_files/selinux/papers/policy2.pdf.
[33] T. Jaeger, R. Sailer, X. Zhang, Analyzing integrity protection in the SELinux example policy, in: USENIX Security Symposium, San Antonio, Texas, USA, 2003.
[34] P. Loscocco, S. Smalley, Integrating flexible support for security policies into the Linux operating system, in: 11th USENIX Security Symposium, Boston, Massachusetts, USA, 2001.
[35] M. Blanc, P. Clemente, J. Rouzaud-Cornabas, C. Toinard, Classification of malicious distributed selinux activities, Journal of Computers, Special Issue on Security and High Performance Computer Systems 4 (5) (2009) 423–432.
[36] Dan Walsh, Confining the User with SELinux, 2007. http://danwalsh.livejournal.com/10461.html.
[37] M. Blanc, K. Guérin, J.-F. Lalande, V.L. Port, Mandatory access control implantation against potential nfs vulnerabilities, in: W.W. Smari, W. McQuay (Eds.), Workshop on Collaboration and Security 2009, IEEE Computer Society, Baltimore, 2009, pp. 195–200.
[38] D. Quigley, J. Morris, MAC Security Label Requirements for NFSv4, Tech. Rep., IETF, Jun. 2008.
[39] SELinux Project, Labeled NFS, Dec. 2009. http://selinuxproject.org/page/Labeled_NFS.
[40] Jeff Bastian, files on Lustre filesystem are unlabeled_t, Oct. 2009. https://bugzilla.redhat.com/show_bug.cgi?id=489583.
[41] Lustre, Lustre Roadmap, Nov. 2009. http://wiki.lustre.org/index.php/Lustre_Roadmap.
[42] Brad Spengler, On exploiting null ptr derefs, disabling SELinux, and silently fixed Linux vulns, Mar. 2007. http://seclists.org/dailydave/2007/q1/224.
[43] TOP500.org, 36th top500 list of the world's most powerful supercomputers, Nov. 2010. URL http://top500.org/lists/2010/11.
[44] R. Coker, The bonnie++ i/o benchmark. URL http://www.coker.com.au/bonnie++.



M. Blanc is a French researcher in the field of information systems security. He graduated from ENSEIRB in 2002 and earned a Ph.D. in Computer Science from the University of Orleans in 2006. Since 2007, he has been a research engineer at the Commissariat a l'Energie Atomique (Atomic Energy Commission). His research interests are primarily focused on the safety of high-performance computing architectures and the evaluation of security systems. He participates in various program committees of conferences in the areas of security and scientific computing. Moreover, he is regularly asked for security assessments within the CEA. An occasional writer of French security magazine, he was involved in several IEEE conferences around the security of distributed systems. He teaches at ENSEIRB and the University of Versailles - Saint Quentin, and helps organize various conferences as SSTIC and RMLL.

J.-F. Lalande has been Associate Professor at ENSI de Bourges, in the Laboratoire d'Informatique Fondamentale d'Orléans (LIFO) since 2005. His research domains are the security of operating systems, the security of C embedded software, especially for smart cards. He works on mandatory access control policies, intrusion detection tools and software code analysis. He obtained his PhD in Computer Science from the University of Nice. Before 2005, he worked on dimensioning telecommunication networks in the Mascotte team (INRIA/I3S/CNRS/UNS).