PIGA-HIPS: Protection of a Shared HPC Cluster

M. Blanc*, J. Briffaut, D. Gros, C. Toinard
Laboratoire d'Informatique Fondamentale d'Orléans
*CEA, DAM, DIF F-91297 Arpajon, France, mathieu.blanc@cea.fr
ENSI de Bourges – LIFO, 88 bd Lahitolle, 18020 Bourges cedex, France
{jeremy.briffaut,damien.gros,christian.toinard}@ensi-bourges.fr

***Abstract*—Protecting a shared High Performance Computing cluster is still an open research problem. Existing solutions deal with sand-boxing and Discretionary Access Control for controlling remote connections. Guaranteeing security properties for a shared cluster is complex since users demand an environment at the same time efficient and preventing confidentiality and integrity violations. This paper proposes two different approaches for protecting remote interactive accesses against malicious operations. Those two approaches leverage the SELinux protection. They have been successfully implemented using standard MAC from SELinux, and guarantee supplementary security properties thanks to our PIGA HIPS. The paper compares those two different approaches. It presents a real use case for the security of a shared cluster that allows interactive connections for users while preventing confidentiality and integrity violations. That paper takes advantage of previous works and goes one step further for protecting shared clusters against malicious activities. It proposes a new framework to share a cluster among partners while guaranteeing advanced security properties. This solution aims to prevent complex or indirect malicious activities that use combinations of processes and covert channels in their attempt to bypass the required properties.**

Keywords - Security Property, Mandatory Access Control, High Performance Computing Security.

I. Introduction

Protecting a HPC cluster [1] against real world cyber threats is a critical task nowadays, with the increasing trend to open and share computing resources. As partners can upload data that is confidential regarding other partners, a company managing a shared cluster has to enforce strong security measures. It has to prevent both accidental data leakage and voluntary data stealing.

Security of the clusters accesses usually relies on Discretionary Access Control (DAC) and sand-boxing such as chroot, BSD Jail or Vserver. The DAC model has proven to be fragile [2]. Moreover, virtual containers do not protects against privilege escalation where users can get administration privileges in order to compromise data confidentiality and integrity.

Mandatory Access Control (MAC) can be used to confine the various cluster populations. MAC such as in NSA's Security-Enhanced Linux (SELinux) provides a powerful protection scheme. However, defining efficient SELinux policies is complex. Moreover, advanced security properties cannot be enforced by that approach. For example, SELinux in its current design does not control information flows involving multiple processes and ressources.

Solutions such as [3] propose enforcement of advanced security properties for SELinux. However, efficient protection of remote accesses has not yet been proposed using those kinds of solutions. Moreover, specific contexts of use such as a cluster shared between various entities require a new protection scheme since the protection must scale well.

That paper answers those two questions. First, it presents two different solutions for the protection of a shared cluster against malicious usage. Second, it discusses the advantages and presents the solution chosen to protect a large scale cluster such as the ones deployed by the CEA. Finally, it shows how to compute the remaining risks associated with a given SELinux policy, how to analyze the remaining risks and how to prevent them. The result is that, however complete, the proposed SELinux policy still cannot prevent about a million of indirect illegal activities. But, the PIGA HIPS enables to prevent against these remaining risks. Then, the paper presents a real case study showing realistic scenarii of attacks, and how the PIGA HIPS can prevent them.

II. Related Work

High performance computing architectures are extremely specialized, compared to general computing facility. As such, they present specific security issues and properties. As outlined by William Yurcik [4], these issues must be addressed in a way that is relevant, with a combination of general techniques and one that are specific to cluster architectures.

Sandboxing such as [5] or [6] provides a mean to confine processes. However, in the context of a shared cluster where processes can communicate and share files, the confinement cannot be hardened and information can flow between processes and resources.

Under Linux, there are at least four security models available to ensure a Mandatory Access Control policy: SELinux, grsecurity, SMACK and RSBAC. But none of these solutions can ensure a large set of security properties. In the best case, they can ensure one or two limited properties such as the Bell and LaPadula confidentiality or the Biba integrity. Under the BSD family, solutions such as Trusted BSD (available within the following Operating Systems: FreeBSD, OpenBSD, MacOSX, NetBSD) provide more or less the same kind of a Mandatory Access Control as SELinux. But, again, they fail to ensure the large majority of requested security properties.

The major limitation is related to indirect information flows, allowed by the considered protection policies, that enable to violate a security property. All those approaches fail to correctly manage indirect information flows, consequently authorizing illegal activities.

Several studies address how to manage indirect information flows within an Operating System. The HiStar Operating System [7] associates each object or subject with an information flow level. The problem of HiStar is that it is very close to the the Biba integrity model and suffers from the same limitations. The Flume Operating System [8] is very close to HiStar. However, Flume does not control efficiently the information flows.

Asbestos [9] reuses the idea of HiStar by considering four different levels of information. The protection rules can only express pairwise relationship patterns. Again, information flows involving multiple interactions and processes cannot be controlled easily.

Works about the enforcement of dynamic policies such as [10], [11], generally consider how to detect simple conflicts within dynamic policies. For example, they detect if it is safe to remove or add a role or a context, otherwise the considered access control could become invalid, conflicting or not supported. So, they address conflicting rules but do not enforce a large set of security properties.

Briffaut and al. [3] presents how to reinforce the security of SELinux MAC policies. However, security properties for protecting a HPC cluster must be defined. Moreover, the protection system must scale well in order to minimize the performance overhead and ease the deployment.

III. Security objectives

In this section, we present our security goals regarding our experimental shared HPC clusters. These goals are the basis of our security policy, and thus of our SELinux configuration. They can be resumed in five points:

- Ensure the confidentiality of data uploaded by partners;
- Confine user profiles and services so that a malicious elevation of privileges does not compromise the security of the operating system;
- Differentiate public SSH access and administrator SSH access;

The following subsections give details on each point.

A. User containers and data confidentiality

Users from the same projects should be able to exchange files freely. Hence there is a particular set of Unix groups 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, accidental leakage of information due to incorrect permissions is considered harmless.

Definition 3.1 (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 3.2 (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, aMAC mechanism will confine users in their container, and then inside a container users can restrict access to their own files with DAC permissions.

B. 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 3.3 (Confinement): Any person gaining administrative privileges on a system must not be able to break the confidentiality property, either legitimate user or external attacker.

C. SSH access

There should be two different points of access on the cluster nodes: a public access for standard users, and an administrative access reserved to system administrators. These accesses are always setup with a ciphered protocol like SSH. Even in the case a vulnerability is exploited in the server and gives an administrative access to an attacker, the public access should never allow users to configure the security mechanisms. Only the administrative access should.

Of course interactive user access is not always enabled. For example, there are some parts of the clusters like computing nodes where standard user access should be disabled. These nodes should only be accessed through the batch scheduler. The same restriction goes for the storage nodes, accessible only through mounted network file systems, and so on. These are only examples, each cluster has its specific areas.

IV. SELinux Solutions of Protection

A. Solution 1: SSH users confinement with chroot and SELinux

When the SSH daemon receives a connection the user is authenticated, it forks and executes the user's shell from `/etc/passwd`. Our confinement system provides a chroot confinement for this shell, strengthened by SELinux rules via a SELinux module. First, we create a Linux sub tree in which the SSH daemon will chroot the user. The main idea is to build one confinement tree per user, and each user has different SELinux types. We use the base types defined for / tree, adding the username of the user linked to the confinement. For example, if we confine Bob in `/cage`, files in `/cage/etc` will have

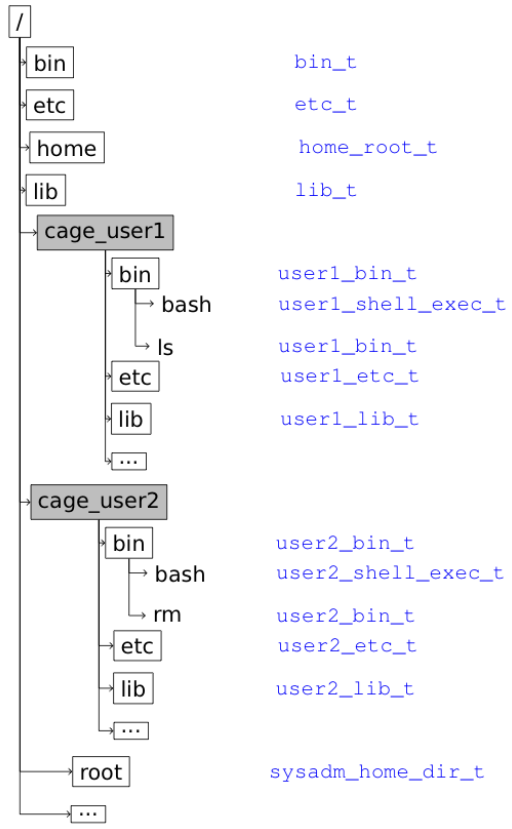

Fig. 1. Jail and SELinux contexts.

the `Bob_etc_t` type, as files in `/etc` have the `etc_t` type. See figure 1 for an example of SELinux types we use.

The goal of our SELinux module is to make an automatic transition of the user confined shell to a unique type, and to give rights to this type to interact only with objects that own a type in the user's sub tree. So, each user's processes and objects have a different SELinux type and consistent protection rules prevent from unauthorized accesses. Even in case of chroot or application corruption, a process (e.g. user1 subject) is prevented from accessing unauthorized resources (e.g. user2 objects).

Here are the main steps to confine a user. The first step is to create the sub tree. Jail is a simple Linux tool (not to be confused with FreeBSD Jail) to build such a sub tree. Then, we build the SELinux module to strengthen the confinement. We write the source code of the module, compile it and load it to SELinux. The next step is to configure OpenSSH to chroot this user in the requested sub tree, and apply the correct SELinux labels to the user file tree (relabeling operation). As the SELinux module has been loaded before relabeling the tree, it will get types we defined in our policy module, not default types.

A bash script provides automation for these different steps. The script just needs the user name and the path you want to confine him in. For our tests we used the SSHJail patch

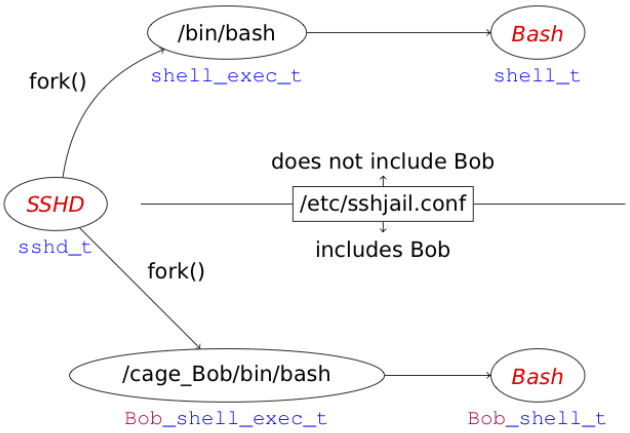

Fig. 2. Jail and SELinux transitions.

for OpenSSH, which use the `/etc/sshjail.conf` file to define user's chroot. The script must be adapted if using another system. Of course, the script checks if this user is already confined in another sub tree or if the sub tree is already used to confine another user. The SELinux rules defined for the modules are very restrictive, they just allow the user to login and run a few commands such as `ls`. Other commands can be added on demand based on application requirements.

This system requires to set up each user on the machine. So, our script needs to be executed after the `useradd` command in order to confine every user. It is a good thing to gather these two steps into a single one. The solution is to provide an alias for the `useradd` command that runs the two steps. A SELinux rule prevents the real `useradd` from being run directly. Thus, only the alias is allowed for execution.

B. Solution 2: SSH users confinement with Port differentiation and SELinux

The idea is to have different SSH ports for each user categories. In the sequel, we consider only two types of users. One has the context `ccc_guest_t` and offers a restricted access. The other one is `unconfined_t` which gives full privilege access. To separate these two kinds of interactive accesses, we introduce two new contexts for two SSH servers, `sshd_public_t` and `sshd_admin_t`. The first one offers restricted access whereas the second one gives a privileged access.

This is implemented in two different SELinux policy modules described in the following parts: `ccc_guest` and `sshd_admin`.

1) `ccc_guest`: By default, all the users are placed in the context `(user_u, system_r, unconfined_t)`. The goal of our SELinux module is to provide two confined user profiles: `ccc_guest` and `ccc_xguest`. The first one is associated to SSH connections, and the second one is associated to X11 sessions. They are originally derived from the `guest` and `xguest` profiles of Fedora 10, provided by Dan

Walsh [12]. They were subsequently adapted to our specific needs.

```
1  unprivileged_user(ccc_guest_t, ccc_guest);
```

This template deals with the creation of the basic rules for the ccc_guest_t type. When the user logs on the system, he receives a set of access rights that allows him to perform basic actions. This unprivileged_user template also enables to use a part of the network (important for the use of SSH).

```
1  userdom_restricted_xwindows_user_template(ccc_xguest);
2  use_kde(ccc_xguest_t);
```

The xguest profile derived of Fedora 10 allows us to define rules for the X11 forwarding in SSH. There are specific rules for the use of X11 and graphical environment such as KDE.

```
1  corecmd_shell_domtrans(sshd_public_t, ccc_guest_t);
```

This template enables the ccc_guest_t type to interact with the sshd_public_t. The corecmd_shell_domtrans allows the public SSH context to transit to the guest type, and only this one. That way, the public SSH server only gives restricted access.

```
1  use_local_home_dir(ccc_guest);
```

The use_local_home_dir template allows the user to manipulate and more especially to create, manage file permissions and relabel certain types of files and directories, for example the user_home_t type.

For the use of NFS and LUSTRE file system, specific rules have been defined.

These different templates allow us to use SSH with a specific type (sshd_public_t) and to confine the shell of the user. The objective of these confined user profiles is to limit the administrative privileges accorded to users to the minimum. For example, a standard user logged on the system via SSH will have the context (ccc_guest_u, ccc_guest_r, ccc_guest_t). Trying to exploit a vulnerability in any system command or service, he may obtain a root access, but he will still have the same confined SELinux context and will not be able to take advantage of this root access.

Moreover, several rules protect the system against malicious code execution by the user. For example, the stack is protected against the execution with the following rule.

```
1  allow $1_t self:process ~{ setcurrent setexec execmem execstack };
```

2) sshd_admin: By default, the SSH server is not associated to a particular context, it is actually executed in the unconfined_t type. As the SSH service must be accessible to all the cluster partners from the Internet, it is heavily exposed to attacks. This policy module answers two goals. 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. Secondly, the attainable SELinux roles from the SSH server should be limited to what is strictly necessary. That is why we create two different contexts for two SSH servers running at the same time (on different TCP ports): sshd_public_t and sshd_admin_t.

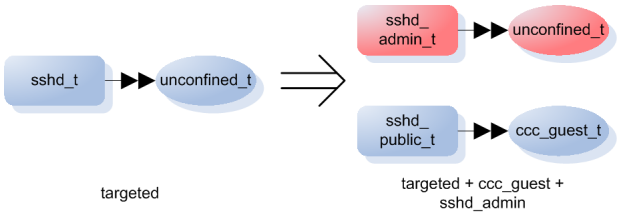

Fig. 3. Two levels of SSH access.

The first context is for the public SSH access. We call it "public" but it may already be filtered at the network level. This context can only transit to the ccc_guest_t type. The second context is reserved for administrative access, and can transit to the unconfined_t type. This is illustrated in figure 3.

The profile for the admin is close to the ccc_guest profile. It can perform several operations on files and directories. The policy adds the right for the admin to list and read the root directory. The admin can also use X11.

```
1  unprivileged_user(ccc_admin_t, ccc_admin);
2  userdom_restricted_xwindows_user_template(ccc_admin);
```

The admin is able to transit to the unconfined_t thanks to this specific rule whereas there is a rule that denies the transition for the user to transit to the unconfined_t.

```
1  neverallow sshd_public_t unconfined_t:process { transition };
2  allow sshd_admin_t unconfined_t:process { transition };
```

The admin has to be able to administrate the ccc_guest. A special program named xbe is used. The use_xbe template provides the set of rules allowing to manage the guest processes and change the permissions on the guest files.

```
1  use_xbe(ccc_admin, ccc_guest);
```

For example, the following rule is included in the use_xbe template. This rule enables the ccc_admin_t subject to control the ccc_guest_t process.

```
1  allow ccc_admin_t ccc_guest_t:process { siginh rlimitinh noatsecure };
```

C. Discussion of the two SELinux solutions

When a user logs on the cluster, he receives his allowed set of permissions. He will be able to access only the files that are allowed to him.

Our intended protection is to provide containers. For the first solution, containers are associated with the user identities. Each user can be seen as a container. In the context of a shared cluster, it is not a scalable approach since the users cannot easily share data. A complex SELinux policy must be defined in order to allow the required sharing. However, that approach enables a complete control of the resources's accesses. For the second solution, containers are associated with the server port. In the sequel, we consider only two SSH server ports. But, in the context of multiple partners sharing the cluster, each partner would use a dedicated server. That approach scales

better since a container is associated to each partner. So, one does not need to define complex sharing policies.

Our main objective is to prevent against indirect information flows i.e. covert channels. While offering a very robust protection in case of an application security vulnerability, SELinux cannot control those flows. In the following section, we show how the PIGA security tools are used to assess those indirect flows. Dedicated security properties are proposed to control the flows of our second solution.

V. Analysis of the Cluster Protection

In order to analyze and to reinforce the security provided by the SELinux policy of our second solution, several security properties are proposed using the Security Property Language defined in [3]. The advantages of our SPL are 1) to enumerate the remaining risks within a given SELinux policy and 2) to provide a mean to prevent these risks. That section will describe the first point in order to analyse the remaining risks in our SELinux policy. The second point will be developed latter on in the paper when considering the enforcement of the Security Properties in order to prevent againts these remaining risks.

A. Security Properties

Using the SPL described in [13], several security properties templates are proposed. In order to analyse the remaining risks within our SELinux MAC policy, that section defines dedicated security templates such as *transitionsequence*, *cantransit*, *dutiesseparationbash* and *dutiesseparationreadwrite*. That section shows how the usage of these security templates enables to analyze the security risks of our second solution. Each security template can be considered as a generic security objective. A template enables to define an instance of the considered security property associated with relevant security contexts. Each instance of a security property corresponds to a security objective that the target operating system must satisfies.

1) Templates of Security Properties: The first template of security property enables to protect against confidentiality violation between a source security context $sc1$ and an object security context $sc2$. Thus, one can prevent both direct and indirect information flows from $sc2$ to $sc1$.

```
1  define confidentiality( $sc1 IN SCS, $sc2 IN SCO ) [
2          ST { $sc2 > $sc1 },
3                  { not(exist()) };
4          ST { $sc2 >> $sc1 },
5                  { not(exist()) };
6  ];
```

The second template of security property enables to protect against integrity violation from a source security context $sc1$ against an object security context $sc2$. Thus, one can prevent both direct and indirect integrity violations from $sc1$ against $sc2$.

```
1  define integrity( $sc1 IN CSS, $sc2 IN CSO ) [
2          Foreach $eo IN is_write_like(IS)
3                  ST { $sc1−> { $eo } $sc2 } ,
4                          { not(exist()) };
5          Foreach $eo IN is_write_like(IS)
6                  ST { $sc1 => { $eo } $sc2 } ,
7                          { not(exist()) };
8  ];
```

The third template of security property enables to detect all the transitions to the existing source security contexts. That template is usually not used to prevent transitions but to detect the transitions carried out on the target system.

```
1  define transitionsequence( ) [
2          Foreach $sc1 IN { system_u:system_r:init_t }, Foreach $sc2
                  IN CSS
3           ST { $sc1 __> $sc2 },
4                  { not(exist()) };
5  ];
```

The fourth template of security property enables to prevent transitions from a source security context $SCFROM$ to another source security context $SCTO$. Thus, one can prevent processes to transit into specific contexts to get illegal privileges.

```
1  define cantransit( $SCFROM IN CS, $SCTO IN CS) [
2          Foreach $sc1 IN $SCFROM, Foreach $sc2 IN $SCTO
3           ST { $sc1 __> $sc2 },
4                  { not(exist()) };
5  ];
```

The fifth template of security property enables to prevent indirect activities from violating the existing SELinux direct policy. Thus, a process cannot get indirect privileges that are conflicting with the allowed direct SELinux permissions.

```
1  define consistentaccess($sc1 IN CS, $sc2 IN CS) [
2          ST { $sc1 >>> $sc2 },
3                          { exist[$sc2 > $sc1] };
4  ];
```

The sixth template of security property enables to prevent bash activities to write and then read scripts. Thus, one can prevent attacks consisting in using bash to execute illegal scripts.

```
1  define dutiesseparationbash( $sc1 IN CS ) [
2          Foreach $eo1 IN is_write_like(IS), Foreach $eo2 IN
                  is_execute_like(IS), Foreach $eo3 IN is_read_like(IS),
3          Foreach $sc2 IN $CSONE, Foreach $sc3 IN CS,
4          Foreach $a1 IN ACT, Foreach $a2 IN ACT
5                  ST { ( [ $a2 := $sc1 −> { $eo3 } $sc2 ] o ( [ $a1 :=
                          $sc1 −> { $eo2 } $sc3 ] o $sc1 −> { $eo1 }
                          $sc2) ) } ,
6                          { INHERIT($a2 , $a1) };
7  ];
```

The seventh template of security property enables to prevent malicious activities to write and then read files. Thus, one can prevent attacks consisting in writing data in order to forward the produced information.

```
1  define dutiesseparationreadwrite( $sc1 IN CS ) [
2          Foreach $eo1 IN is_write_like(IS), Foreach $eo2 IN
                  is_read_like(IS), Foreach $sc2 IN $CSRW
3                  ST { ($sc1 −> { $eo2 } $sc2 o $sc1 −> { $eo1 }
                          $sc2) } ,
4                          { not( exist() ) };
5  ];
```

2) Analysis of SElinux through instances of the security properties: The templates defined previously enable to define several instances of the security properties for protecting the administrator role, as in the following listing. The first property prevents the admin role to transit to the guest role. The second, third and fourth properties force the admin role to satisfy various separation of duties. For example, the admin role cannot uses bash to execute illegal scripts. He can only execute scripts legally installed on the target SELinux. The fifth

and sixth properties prevent against integrity violation carried out from the admin role. The seventh property prevents the admin role from indirectly getting permissions that are not available in the SELinux policy. The eighth property prevents the admin role to violate the confidentiality of the guest role either directly or indirectly.

Listing 1. Security properties for admin

```
1 cantransit( $SCFROM:=".*:ccc_admin_r:.*", $SCTO:=".*:ccc_guest_r
      :.*" );
2 dutiesseparation( $sc1:=".*:ccc_admin_r:.*" );
3 dutiesseparationbash( $sc1:=".*:ccc_admin_r:.*" );
4 dutiesseparationreadwrite( $sc1:=".*:ccc_admin_r:.*" );
5 integrity( $sc1:=".*:ccc_admin_r:.*", $sc2:=".*:.*:.*_exec_t" );
6 integrity( $sc1:=".*:ccc_admin_r:.*", $sc2:=".*:.*:.*etc_t" );
7 consistentaccess($sc1:=".*:ccc_admin_r:.*" );
8 confidentiality( $sc1:=".*:ccc_admin_r:.*", $sc2:=".*:ccc_guest_r:.*" );
```

Several similar properties are defined for protecting the guest role. Those properties enable to prevent the guest role to violate the confidentiality of the admin role.

Listing 2. Security properties for guest

```
1 cantransit( $SCFROM:=".*:ccc_guest_r:.*" , $SCTO:=".*:ccc_admin_r
      :.*");
2 dutiesseparation( $sc1:=".*:ccc_guest_r:.*" );
3 dutiesseparationbash( $sc1:=".*:ccc_guest_r:.*" );
4 dutiesseparationreadwrite( $sc1:=".*:ccc_guest_r:.*" );
5 integrity( $sc1:=".*:ccc_guest_r:.*", $sc2:=".*:.*:.*_exec_t" );
6 integrity( $sc1:=".*:ccc_guest_r:.*", $sc2:=".*:.*:.*etc_t" );
7 consistentaccess($sc1:=".*:ccc_guest_r:.*" );
8 confidentiality( $sc1:=".*:ccc_guest_r:.*", $sc2:=".*:ccc_admin_r:.*" );
```

B. SELinux Policy Analysis with PIGA

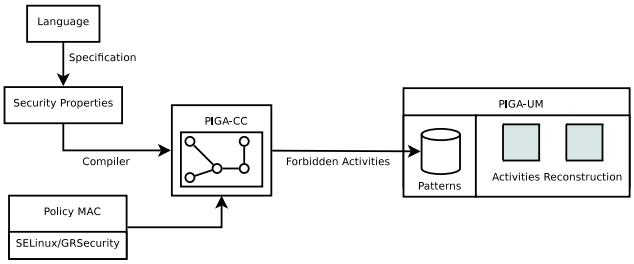

Fig. 4. Process of analyzing a SELinux policy and generating the patterns violating the security properties

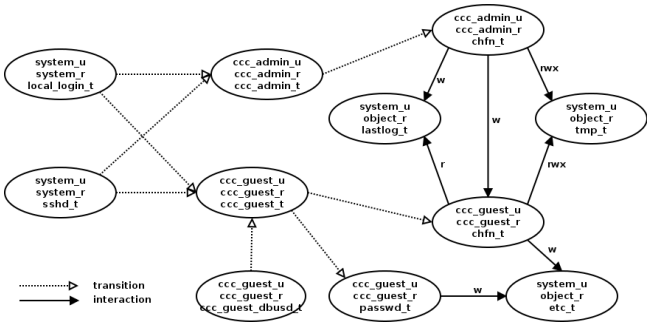

Fig. 5. Interaction graph

The objectives are first to enumerate the possible risks included into a given SELinux policy and second to provide a mean to prevent against these remaining risks. For these purposes, Figure 4 shows how to analyze a SELinux policy in order to enumerate the remaining risks. A compiler, PIGA-CC, a module of PIGA, is used to enumerate from a given MAC policy (i.e. a SELinux policy) the set of the illegal activities associated with requested security properties. This section only addresses the first point i.e. how the remaining risks can be enumerated. Latter on in the paper, we will address the way the enumerated remaining risks enable to guarantee the security properties.

The PIGA-CC module builds a graph that represents the access control policy (the SELinux policy). For each security property instanciation, PIGA-CC enumerates paths into this graph. Each enumeration, a combination of paths, corresponds to a possible violation of a required security property starting from the considered SELinux policy. Thus, the compiler enumerates all the forbidden activities i.e. all the sequences of system calls allowing the violation of a considered security property. The compiler sends those illegal activities to PIGA-UM in order to prevent the violation of the requested security properties.

Figure 5 gives an example of the sub-graph computed by PIGA-CC for the SELinux policy considered in this paper. In this graph, each edge between two contexts sc_1 and sc_2 corresponds to a direct interaction $sc_1 \rightarrow sc_2$, and each path between two contexts sc_1 and sc_n corresponds to an indirect interaction $sc_1 \Rightarrow sc_n$. This graph enables to enumerate all the terminals of the SPL language.

For example, when PIGA-CC analyses the confidentiality property:

```
1 define confidentiality( $sc1 IN SCS, $sc2 IN SCO ) [
2         ST { $sc2 > $sc1 },
3                 { not(exist()) };
4         ST { $sc2 >> $sc1 },
5                 { not(exist()) };
6 ];
```

It extracts the edge between sc_1 and sc_2 and then it generates a violation for each operation between these two contexts that correspond to a possible information transfer, i.e., a read operation between sc_1 and sc_2 or a write operation between sc_2 and sc_1. Next, PIGA-CC enumerates the set of paths between sc_1 and sc_2 where each edge is an information transfer correctly oriented.

Let us give an example for the Figure 5 and the following security property:

```
1 confidentiality( $sc1:=".*: ccc_guest_r:.*", $sc2:=".*:ccc_admin_r:.*" );
```

As shown in Figure 5, an information transfer is possible between the context $ccc\_admin\_u : ccc\_admin\_r : ccc\_admin\_t$ and $ccc\_guest\_u : ccc\_guest\_r : ccc\_guest\_t$ using the intermediate object $system\_u : object\_r : lastlog\_t$. Thus, this policy does not respect this confidentiality property since there is an activity allowing a forbidden information transfer between $ccc\_admin\_u : ccc\_admin\_r : ccc\_admin\_t$ and $ccc\_guest\_u : ccc\_guest\_r : ccc\_guest\_t$. As presented in the sequel all the forbidden activities can be used to prevent the considered flows i.e. to guarantee the requested properties.

C. Results of the SELinux policy analysis

This section presents the results provided using PIGA-CC. These results corresponds to the security analysis of the SELinux policies available for our second solution. These results are presented in two tables. The table I describes the analysis for the administrator role and the table II for the guest role. These tables are divided into two columns.

The first one deals with direct activities.These activities are blocked by SELinux. The second one deals with indirect activities. These flows cannot be prevented by SELinux but they can be blocked byPIGA-SP an extended MAC approach reusing the enumerated activities. The PIGA-SP MAC approach will be presented in the sequel.

For example, in the table I, PIGA-CC was able to find 4 direct activities dealing with integrity property and 8 indirect activities.

Table I shows, for the admin role, all the direct illegal activities and all the indirect illegal activities. This table is based on the listing 1. The direct illegal activities can be prevented simply by a modification of the SELinux policy. But, PIGA-SP MAC can prevent all those direct and indirect violations. As shown in table I, the confidentiality property can prevents against 548.858 potentially illegal activities. Those illegal activities are vectors that enable intruders to develop exploits against the SELinux protection by using for example a combination of buffer overflows and covert channels.

TABLE I
RESULT FOR THE ADMINISTRATOR ROLE

Security property	direct activities	indirect activities
cantransit	0	0
dutiesseparation	0	469
dutiesseparationbash	0	124288
dutiesseparationreadwrite	0	1191
integrity	0	0
integrity	4	8
consistentaccess	0	1474
confidentiality	98	548858

Table II shows the illegal activities for the guest role.This table is based on the listing 2. It shows that PIGA-SP MAC can prevent against about 1 million of illegal activities that could compromise the SELinux protection. Those results demonstrate that SELinux is not able to guarantee advanced security properties such as the ones required for protecting a shared cluster. In contrast with SELinux, PIGA-SP MAC can efficiently enforce all the requested properties. The sequel presents the way PIGA-SP reuses the illegal activities for preventing the violation of the requested properties. PIGA-SP is an advanced Host Intrusion Prevention System. But the illegal activities can be used to provide an Host Intrusion Detection System. Before explaining the internals of the PIGA-SP MAC approach, let us first give an example of typical scenarii of attacks allowed by the SELinux policy.

TABLE II
RESULT FOR THE GUEST ROLE

Security property	direct activities	indirect activities
cantransit	0	0
dutiesseparation	0	1118
dutiesseparationbash	0	328362
dutiesseparationreadwrite	0	2530
integrity	0	0
integrity	8	16
consistentaccess	0	3026
confidentiality	96	728214

D. Case study of remaining risks within the SELinux policy

Amoung the millions of potentially illegal activities, let us choose some of them to show the typical scenariis of attacks that PIGA can prevent.

Regarding the confidentiality of the admin role, the line 2 of the following listing shows a direct activity where the `chfn` application (that changes your finger information) uses a covert channel, i.e. writing in a fifo file, with the guest role. One can imagine an exploit against the `chfn` application using that direct covert channel to make the admin's data flow to the guest role. The line 3 shows an indirect activity, where the admin role writes a log file that is then read by the guest role. If such an activity occurs, PIGA-SP guarantees that the reading of the log file will fail.

```
1  confidentiality( $sc2:=".*:ccc_admin_r:.*", $sc1:=".*:ccc_guest_r:.*" );
2  2$55 : ccc_admin_u:ccc_admin_r:chfn_t −( fifo_file { write } )−>
       ccc_guest_u:ccc_guest_r:chfn_t
3  3$280219 : ccc_admin_u:ccc_admin_r:ccc_admin_t −( file { append
       write } )−> system_u:object_r:lastlog_t ; ccc_guest_u:
       ccc_guest_r:ccc_guest_t −( file { read } )−> system_u:object_r
       :lastlog_t
```

Regarding the confidentiality of the guest role, the line 2 of the following listing shows a direct activity where the SELinux policy enables the guest role to use a covert channel, i.e. writing in a fifo file, with the admin role. The line 3 shows an indirect activity, where the guest role writes a temporary file that is then read by the admin role. It is an indirect covert channel that PIGA can prevent. Thus, the reading of the temporary file by the admin role will fail.

```
1  confidentiality( $sc1:=".*:ccc_admin_r:.*", $sc2:=".*:ccc_guest_r:.*" );
2  0$90 : ccc_guest_u:ccc_guest_r:ccc_guest_t −( fifo_file { write } )−>
       ccc_admin_u:ccc_admin_r:ccc_admin_t
3  1$471971 : ccc_guest_u:ccc_guest_r:ccc_guest_t −( file { append
       write } sock_file { append write } )−> system_u:object_r:tmp_t ;
        ccc_admin_u:ccc_guest_r:ccc_guest_t −( file { read } sock_file
        { read } )−> system_u:object_r:tmp_t ;
```

Regarding the integrity of the admin role, the line 2 of the following listing shows a direct activity where the `chfn` application can write the `/etc` files. One can imagine to use `chfn` to get an admin role and thus modifies the configuration files that are present into the directory `/etc`.

```
1  integrity( $sc1:=".*:ccc_guest_r:.*", $sc2:=".*:.*:.*etc_t" );
2  15$3 : ccc_guest_u:ccc_guest_r:chfn_t −( file { write } )−> system_u:
       object_r:etc_t
```

The following extract of the SELinux policy shows that the guest role can execute the `chfn` application. Moreover, `chfn` as the setuid bit set which enables guest to get the admin role

when executing `chfn`. That example shows that such scenarii of attacks are very easy to carry out. PIGA can efficiently protect against them.

```
1        role ccc_guest_r types chfn_t;
2        allow ccc_guest_t chfn_t:process transition;
3
4   −rws−−x−−x root root /usr/bin/chfn
```

Regarding the integrity of the guest role, the line 1 of the following listing shows a first activity that enables the `dbus` process to transit to the guest role and then transit to the password type. Then, the line 2 shows that a process with the password type can write the `/etc` files and thus modify the passwords. PIGA protects against the combination of those two activities. Thus, if an exploit on `dbus` enables to transit to the guest role, the attempt to then write into `/etc` will fail.

```
1   16!1$11 : ccc_guest_u:ccc_guest_r:ccc_guest_dbusd_t −( process {
          transition } )−> ccc_guest_u:ccc_guest_r:ccc_guest_t ;
          ccc_guest_u:ccc_guest_r:ccc_guest_t −( process { transition } )
          −> ccc_guest_u:ccc_guest_r:passwd_t
2   16!0$11 : ccc_guest_u:ccc_guest_r:passwd_t −( file { write } )−>
          system_u:object_r:etc_t
```

VI. Security Properties Enforcement

That sections adresses the second objective of that paper i.e. how the enumerated illegal activities can be reused to guarantee the requested security properties. It describes the PIGA MAC approach enabling to guarantee all the security properties expressed using our SPL language.

A. Implementation of the PIGA MAC protection

As described in Figure 4, our MAC protection model is divided into two stages. First, the security administrator defines a set of security properties. Generally, the administrator reuses and configures existing canevas such as the various properties proposed in this paper. However, he can also use our SPL language to define dedicated security properties. Then, he uses the PIGA-CC compiler that compares the requested security properties against a mandatory policy (SELinux or grsecurity) in order to compute all the illegal activities existing in that SELinux policy. The set of forbidden activities is then compressed and stored in a database of patterns.

Second, the security administrator can use our PIGA-SP MAC solution in order to ask the enforcement of these security properties by the operating system. PIGA-SP uses a combination of a kernel module PIGA-KM and userland application PIGA-UM. The kernel module hooks the system calls and sends the corresponding traces to the userland application. Each system call is thus suspended and the kernel module waits for an authorization or a deny response. The userland application computes and verifies that the system call corresponds to an activity available in the database of patterns. Next, PIGA-UM allows or denies the considered system call aiming at preventing the occurrence of the forbidden activities. Thus, the system call fails if its execution could lead to the violation of the security properties.

Figure 6 shows the process that allows a target operating system, in our case Linux, to enforce the security properties. Extension of the classical protections is proposed. First, the kernel computes the classical Discretionary Access Control. Second, LSM (Linux Security Module) hooks are applied. LSM enables to stack several protection mechanisms such as SELinux or SMACK. In our approach SELinux is processed before running PIGA-SP. SELinux uses the mandatory security policy in order to allow or deny the system call. In our solution, a system call has to be allowed regarding wth the DAC policy, the SELinux policy, and also the requested PIGA Security Properties.

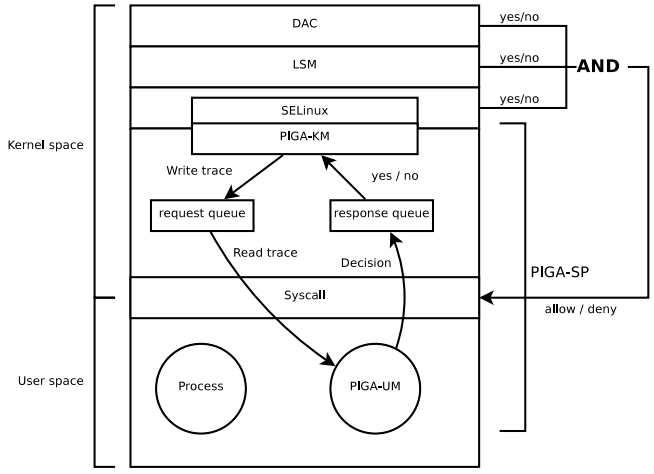

Fig. 6. Process to ensure the security properties at the Operating System layer

PIGA-KM, the kernel part, analyzes all the relevant informations for the considered system call (who made it, from where, what is its type, etc...) in order to generate a trace (i.e. a string) describing the current system call request. PIGA-KM sends that trace to a $request\ queue$ in direction to the userland application, PIGA-UM. The system call is pending, while PIGA-KM is waiting for a message from the $response\ queue$. PIGA-UM computes the response and writes it in the $response\ queue$. When PIGA-KM gets the response from the queue, it sends a decision back to SELinux. The final decision is a logical AND between the PIGA-SP, SELinux and DAC decisions. That final decision allows or denies the system call execution. If allowed, the kernel runs the system call.

The $request\ queue$ is a sequence file in the proc file system. In practice, this file allows to pass a request from the kernel space to the userspace. Once into the userspace, the userland application PIGA-UM reads the request and reconstructs the activities associated to that system call. If a reconstructed activity matches with some available patterns (generated at the compilation stage), PIGA-UM takes the decision to deny that system call.

B. Example of Security Property Enforcement

This section describes how PIGA-SP prevents against real scenarii of attacks. In this example, an administrator uses a ssh connection with the ccc_admin_r role. In this role, he can copy a critical file like $/etc/shadow$ into a file in $/tmp$. This copy could be intentional or produced by an malware

downloaded an executed by the administrator. Next, a user can connect to the system with the ccc_guest_r and read the file, copied by the administrator, in $/tmp$.

Theses actions represents a violation of the following security property that prevents information flows from the ccc_admin_r to the ccc_guest_r role:

```
1  confidentiality( $sc1:=".*:ccc_admin_r:.*", $sc2:=".*:ccc_guest_r:.*" );
```

The result of the analysis of the SELinux policy by PIGA-CC indicates that this violation is possible if we only consider the SELinux policy. This violation corresponds to the following activity:

```
1  3$366122 : ccc_admin_u:ccc_admin_r:ccc_admin_t −( file { write } )
       −> system_u:object_r:tmp_t ; ccc_guest_u:ccc_guest_r:
       ccc_guest_t −( file { read } )−> system_u:object_r:tmp_t
```

The first part of the attack could be simulate by a connexion with the ccc_admin_r following by the copy of $/etc/shadow$ into $/tmp/test$ and a modification of the permissions of the created file:

```
1  # ssh connexion with ccc_admin_u:ccc_admin_r:ccc_admin_t
2  $cat /etc/shadow >> /tmp/test
3  $chmod 777 /tmp/test
```

At the kernel level, the copy of the $/etc/shadow$ file involves a LSM hook specifying that an information has been written by the administrator into a temporally file:

```
1  dec 22 11:23:21 pigaos kernel: type=1400 audit(1277198601.563:1560):
       avc: granted { write } for pid=2056 comm="cat" ppid=1988 dev=
       sda3 ino=534412 scontext= ccc_admin_u:ccc_admin_r:
       ccc_admin_t tcontext= system_u:object_r:tmp_t tclass=file
```

This interaction is allowed by SELinux and also by PIGA-SP because it does not represent a violation of a security property.

Next, a user connects to the same host with the ccc_guest_r role. This user tries to read the content of the file created by the administrator:

```
1  # ssh connexion with ccc_guest_u:ccc_guest_r:ccc_guest_t
2  $ cat /tmp/test
3  cat: /tmp/test: Permission denied
```

This interaction generates the following trace at SELinux Level:

```
1  dec 22 11:25:45 pigaos kernel: type=1400 audit(1277199254.272:1732):
       avc: granted { read } for pid=2080 comm="cat" ppid=1876 dev=
       sda3 ino=534412 scontext= ccc_guest_u:ccc_guest_r:
       ccc_guest_t tcontext= system_u:object_r:tmp_t tclass=file
```

The read interaction of a temporally file by a user in the ccc_guest_r is allowed by SELinux. PIGA-SP prevents this violation because this interaction corresponds to a security property violation.

The trace generated by PIGA-SP indicates the number of the SELinux interaction denied and the corresponding forbidden activity:

```
1  dec 22 11:25:45 pigaos kernel: 3$366122 operation 1732 denied:
       ccc_admin_u:ccc_admin_r:ccc_admin_t > system_u:object_r:
       tmp_t > ccc_guest_u:ccc_guest_r:ccc_guest_t
```

This example illustrates a simple case of security property enforcement. Even if individuals interactions are allowed by SELinux, PIGA-SP is able to control indirect flows or a combination of these indirect flows. Thus, PIGA-SP can guarantee advanced security properties preventing against sophisticated scenarii of attacks including 0-Day attacks.

ACKNOWLEDGMENT

We would like to give special thanks to Jonathan Rouzaud-Cornabas for his participation in the development of PIGA-KM and Maxime Fonda for the development of the SSH confinement with chroot and SELinux.

VII. CONCLUSION

This paper presents two solutions based on SELinux to protect a shared HPC cluster. The first one deals with chroot to confine the user but the approach prevents the user from sharing easily data. The second one is based on two SSH server ports and enables user to share data. This paper focus on the difficulties to prevent sophisticated attacks, using for example several indirect flows, that SELinux cannot control.

That paper shows the efficiency of using SELinux plus PIGA in order to find the illegal activities into the proposed SELinux policy. The found illegal activities can be used to improve the SELinux protection with our PIGA MAC approach in order to better guarantee the confidentiality or the integrity of a shared cluster. PIGA MAC prevents against all the risks of the SELinux policy regarding the various security properties expressed using our Security Property Language. For that purpose, PIGA MAC reuses all the precomputed illegal activities in order to guarantee the required confidentiality and integrity properties. In contrast with SELinux, indirect illegal activities are controlled, permitting thus the prevention of sophisticated attacks.

PIGA MAC can be seen as an advanced HIPS guaranteeing that a system call, terminating an indirect illegal activity, will fail. The PIGA approach can be used also as an HIDS to detect the violations. It is better to use the HIDS approach for properties that correspond to auditing facilities. Moreover, the HIDS approach could be computed on a dedicated cluster. Thus, the impact on the HPC cluster performances is limited. However, PIGA provides a very efficient HIPS approach. Cluster experimentations show that SELinux adds 10% of processor overhead, while PIGA also adds an overhead of 10%. It is a very low overhead for the considered protection that goes much further than the related works. That paper details the implementation of PIGA MAC and gives examples of real scenarii that are blocked by our HIPS.

Finally, the proposed protection enables the real sharing of an HPC cluster while guaranteeing confidentiality and integrity for the partners. Future works deal with the automation of the definition of efficient security properties for the sharing of an HPC cluster. The major difficulty is to adjust the security properties in order to make a good balance between the protection and the usability of the shared cluster.

REFERENCES

[1] M. Blanc, J. Briffaut, T. Coullet, M. Fonda, and C. Toinard, "Protection of a shared hpc cluster," in *Proceedings of the 2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*, ser. SECURWARE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 273–279. [Online]. Available: http://dx.doi.org/10.1109/SECURWARE.2010.51

[2] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Commun. ACM*, vol. 19, no. 8, pp. 461–471, 1976.

[3] J. Briffaut, J.-F. Lalande, C. Toinard, and M. Blanc, "Enforcement of security properties for dynamic mac policies (best paper award)," in *Proceedings of the 2009 Third International Conference on Emerging Security Information, Systems and Technologies*, ser. SECURWARE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 114–120. [Online]. Available: http://dx.doi.org/10.1109/SECURWARE.2009.25

[4] W. Yurcik, G. A. Koenig, X. Meng, and J. Greenseid, "Cluster security as a unique problem with emergent properties: Issues and techniques," *5th LCI International Conference on Linux Clusters*, May 2004.

[5] P. henning Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *In Proc. 2nd Intl. SANE Conference*, 2000.

[6] F. L. Camargos and B. des Ligneris, "Automated oscar testing with linux-vservers," in *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 347–352.

[7] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 19–19.

[8] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," vol. 41, no. 6. New York, NY, USA: ACM, 2007, pp. 321–334.

[9] P. Efstathopoulos and E. Kohler, "Manageable fine-grained information flow," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, pp. 301–313, 2008.

[10] M. L. Damiani, C. Silvestri, and E. Bertino, "Hierarchical domains for decentralized administration of spatially-aware rbac systems," in *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 153–160.

[11] L. Seitz, E. Rissanen, T. Sandholm, B. S. Firozabadi, and O. Mulmo, "Policy administration control and delegation using xacml and delegent," in *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 49–54.

[12] Dan Walsh, "Confining the User with SELinux," http://danwalsh.livejournal.com/10461.html, 2007.

[13] J. Briffaut, J.-F. Lalande, and C. Toinard, "Formalization of security properties: enforcement for mac operating systems and verification of dynamic mac policies," *International journal on advances in security*, pp. 325–343, 2010.