XML distributed security policy for clusters

A. Apvrille, M. Pourzandi*

Open Systems Lab, Ericsson Research Canada, 8400 Décarie Blvd,
Town of Mount-Royal, QC H4P 2N2, Canada

Received 15 January 2004; revised 28 July 2004; accepted 1 September 2004

KEYWORDS
Security policy; Cluster; XML; Distributed management

Abstract With the increasing use of clusters, efficient and flexible security has now become an essential requirement, though it has not yet been addressed in a coherent fashion for distributed systems.

This paper presents a new security policy language for clusters: Distributed Security Policy (DSP). Based on XML, this language offers a precise and easy way to customize security of clusters. Contrary to other existing security policy languages, it is not limited to access control, and may be used for other security services such as cluster’s inner communication.

Finally, the paper also explains how this security policy is used in practice, and how it is transparently enforced onto all nodes of the cluster.

© 2004 Published by Elsevier Ltd.

Introduction

In the last 5 years, clusters have risen as a new alternative to massively parallel supercomputers such as Cray/SGIs. Defined by the IEEE Task Force on Cluster Computing as "general purpose systems consisting of loosely coupled components [also called nodes] built up from single or multi-processor workstations or PCs", clusters have numerous advantages such as lower cost and better flexibility. They have consequently opened up to a large panel of applications such as medical disease diagnosis or next generation telecommunication servers.

However, with their increasing exposure to Internet and the proliferation of security threats, cluster’s security has now become an essential requirement. As a consequence, the Distributed Security Infrastructure (DSI) research project has been initiated and proposes a security framework for clusters. It implements several security services such as communication security, access control or file integrity checking within the whole cluster. Much of this work has already been detailed in Pourzandi et al. (2003), yet, an important aspect of its management is missing so far: security policies for clusters.

In Sloman (1994), security policies are defined as rules governing choices in behavior of a system. More precisely, Bishop (2003) defines them as a statement that partitions the states of the

* Corresponding author. Tel.: +1 514 345 7900 x5405.
E-mail addresses: axelle.apvrille@trusted-logic.com (A. Apvrille), makan.pourzandi@ericsson.ca (M. Pourzandi).

0167-4048/$ - see front matter © 2004 Published by Elsevier Ltd.
system into a set of authorized, or secure, states and a set of unauthorized, or nonsecure, states. Hence, security policies have an essential role in the overall security of the system. Numerous security policy languages do already exist and Sloman and Lupu (2002) covers several approaches. Damianou et al. (2001) identifies multiple requirements for policy languages, such as support for access control, capability to detect conflicts and inconsistencies, extensibility, ease of use... Nevertheless, security policies for clusters have requirements and priorities of their own, which we list below:

Adaptability. Clusters are made of nodes, hence, security policies should integrate this notion and be able to address one or several nodes. Also, clusters are meant to be performant, consequently special care should be taken while enforcing security policies over the cluster so that it only has minor impact on performances. For instance, in the telecommunication world, acceptable performance degradation should not exceed 10% at connexion setup time and 5% during data exchange (Carrier Grade Linux Requirements Subgroup, 2003).

Flexibility. Security policies often focus on access control only, however clusters also need policies for other services such as communication security level between nodes, logging and auditing policy, etc.

Ease of use. From a language perspective, this is not a new requirement: many policy languages such as Damianou et al. (2001) and Verma (2002) already take into account extensibility, readability and simplicity. However, for clusters, it also means ease of deployment: clusters may have numerous nodes, their administrators cannot go through the burden to set up a lengthy configuration for each of them.

In this article, we propose an original security policy which meets the above requirements. It is based on a homogeneous view of clusters, and proposes a Distributed Security Policy language, shortened DSP. Our contribution is divided into two parts. In the former, we detail a security policy language, based on XML, and specifically intended for clusters. In the latter, we describe a propagation mechanism which automatically distributes the policy over all nodes, and hence facilitates administration.

This article is organized as follows. First, we introduce the DSI framework and focus more particularly on the role of its security policy. Then, we detail the DSP semantics and explain our propagation mechanism. Further, we give experimental results obtained so far with our DSI prototype and briefly compare our DSP with related work. Finally, we conclude on our implementation and future work to be done in this area.

The DSI framework

DSI is a security framework which targets distributed environments, and is meant to address any specific security issue such platforms may be concerned with. Mainly, its goals are to provide a homogeneous and helpful security architecture for cluster administrators. Currently, it offers three security services: (1) the Distributed Access Control service (DisAC), which handles all access control requests, (2) the Distributed communications’ Confidentiality and Integrity service (DisCI), which is responsible for securing communications between various distributed applications, and (3) the DigSig service, which dynamically verifies the signature of binaries used on the cluster. Other security services are being investigated (for instance, DisLog, the secure distributed logging service), but not yet implemented (see Fig. 1).

All services rely on top of an architecture which is composed of a single Security Server (SS) and multiple Security Managers (SMs) — one per node (see Fig. 2). To simplify administrator’s task, the SS is the central point of management of the cluster: it gathers all alarms and warnings sent by the SMs and propagates the distributed security.
policy to all SMs. On their side, SMs are responsible to enforce security on their own node, by communicating with the DSI Security Module (DSM), which is implemented at kernel level.

In this architecture, the Distributed Security Policy (DSP) plays a fundamental role. First, it centralizes configuration of all security services presented above, and therefore represents a key element to security. Second, it helps homogeneously manage security by propagating the same policy to all nodes. This is discussed in more detail in section ‘Distributing the security policy’. Finally, the DSP also enables fine grained process-level control of the cluster identifying processes with security node identifiers (SnID) and security context identifiers (ScID). Each node is assigned an SnID, and each process receives an ScID, either explicitly from the administrator or implicitly from the DSM. ScIDs are global over the cluster, and persistent (they do not change after rebooting the hosts). ScIDs are to be seen as security groups: they identify a security context, so processes which share the same security context are assigned the same ScID.

12 DSP semantics

In a distributed framework such as DSI, security policies have been given a key role, because the whole infrastructure is quite useless if it cannot be customized. This section presents the security policy language DSI uses, and, in particular, explains the choices we made for the contents of DSPs (see Fig. 3).

Basically, a DSI security policy is an XML document whose language is specified by an XML schema (2001). It consists in:

- a version number. This optional item is merely provided to help the administrator of the cluster organize various security policies he/she might want to set up. It is not intended to identify a particular version of the policy language: indeed, in XML, those are typically differentiated by a different namespace.
- a mode. The policy may be restrictive (any undefined action is forbidden, this is also referred to as a closed policy) or permissive (an action must be explicitly forbidden — this is an open policy).
- a default ScID to assign to objects that are not explicitly defined (the definition of ScIDs is detailed in the next paragraphs).
- a list of security rules to apply to the cluster.

As previously defined in first section, a security policy indicates the authorized and unauthorized states of the system it applies to. So, adapted to the case of clusters, it must be able to represent possible states of the cluster. In DSI, those states are classified in groups and named classes. Each security rule belongs to one unique class, and may authorize or not a given set of states for this class.

Globally, security rules of a DSP follow the traditional Subject/Resource/Action model (Damianou et al., 2001; Hallam-Baker and Maler, 2002; Godik and Moses, 2003). In DSI, subjects and resources are similarly identified by their “physical” location within the cluster — the node identifier (SnID) — and their security context (ScID). Subjects are referred to as source objects, whereas resources are the targets. In practice, ScIDs and SnIDs are represented in the DSP by

```xml
<complexType name="DSI_POLICY_type">
  <sequence>
    <element name="version" type="dsi:VERSION_type" minOccurs="0"/>
    <element name="mode" type="dsi:DSI_MODE_type"/>
    <element name="default_ScID" type="dsi:INTEGER_ID_type"/>
    <element name="securityRules" type="dsi:SECURITYRULES_type"/>
  </sequence>
</complexType>
```

Figure 3 XML schema of a DSP policy.
positive integers, ranging from 0 to 65535 (see Fig. 4). Note here DSPs initially concentrate on an internal view of security, as according to the annual CSI/FBI survey (Gordon et al., 2004) they still represent about 50% of breaches. On the contrary, from an external point of view, clusters are seen as a unique entity. Consequently, the DSP introduces the wildcard ‘ALL’, which addresses in one single step all nodes of the cluster: ALL is a shortcut to mention any ScID or any SnID.

As for actions, they naturally depend on the class the security rule belongs to. The DSP currently offers six different classes, each of them corresponding to a given XML tag: PROCESS, TRANSITION, NETWORK, SOCKET-INIT, SOCKET and DISC.

The first two are crucial to offer process-level control over the cluster. The process class deals with permissions for process related operations for a given ScID, whereas transitions define how ScIDs are assigned to processes according to their binary file and their parent process. For instance, a process class rule grants or denies a process defined by its ScID and SnID the permission to create other processes. A transition rule makes it possible, for a given node, to allow processes with a given ScID to pawn another process with a given new ScID. The other classes all focus on inter-node communication. Networking is particularly interesting for distributed environments because securing distributed applications requires the capability to control security between nodes. More precisely, the NETWORK class works at IP level, and checks that a given node is allowed to receive IP packets from a given source ScID and SnID. At an upper layer, it is also possible to control TCP or UDP sockets. To do so, sockets are explicitly assigned an ScID in the

```xml
<simpleType name="ID_type">
    <union>
        <simpleType>
            <restriction base="nonNegativeInteger">
                <minInclusive value="0"/>
                <maxInclusive value="65535"/>
            </restriction>
        </simpleType>
        <simpleType>
            <restriction base="string">
                <enumeration value="ALL"/>
            </restriction>
        </simpleType>
    </union>
</simpleType>

<group name="SOURCE_type">
    <sequence>
        <element name="sScID" type="dsi:ID_type"/>
        <element name="sSnID" type="dsi:ID_type"/>
    </sequence>
</group>

<group name="TARGET_type">
    <sequence>
        <element name="tScID" type="dsi:ID_type"/>
        <element name="tSnID" type="dsi:ID_type"/>
    </sequence>
</group>
```

Figure 4  XML schema for source and target ScID/SnID.
DSP using a SOCKET-INIT class rule, or they implicitly inherit the ScID of the process which creates them. Then, SOCKET class rules grant or deny specific socket permissions between source and target ScID/SnID pairs.

All these classes (process, transition, network, socket-init and socket) are access control security rules. They define actions in terms of permissions. But, DSPs also allow other types of rules, such as quality of protection rules. This is a major difference with other security policies such as XACML that only offer access control policies. A more detailed comparison shall be found later in section ‘Related work’. Those qualities of protection rules are represented by the DisCI class, and they define actions as a given security. For instance, a DisCI rule may define a given ScID/SnID should communicate with another ScID/SnID using IPSec with the AH tunneling mode.

In all cases, actions are defined as a new type, within the DSI namespace, consisting of an enumeration of possible values. An example of socket permissions and DisCI security levels is illustrated in Fig. 5.

The possible actions for each class of security rule are defined as an enumeration of string values. This keeps the XML schema easily extendable in case a new action needs to be defined (just add an enumerate to the list) while helping policy’s validation. Indeed, the XML parser does not have to understand the content itself: it just needs to recognize keywords.

In first section, a few requirements for cluster security policies had been identified, mainly adaptability, flexibility and ease of use. The DSP language meets them in the following manner.

First, DSPs inherit from XML’s natural capabilities in terms of extensibility and readability. Extending the DSP to support a new type of security rule simply consists in defining a new type for the class and adding it as a new choice item to the security rules type. Supporting new actions is no more difficult: new enumerate values are to be appended to possible cluster states. In terms of readability, XML has clear advantage over binary formats: the security policy is directly understandable, viewed from any editor. Furthermore, various style sheets can be applied to the policy to highlight such or such configuration issue. Also, on a conceptual point of view, the DSP language is rather simple: all elements are XML tags or contents (no XML attributes), only few keywords to memorize (allow, deny, class-XXX-rule, ScID, SnID...) and security rules may be written in any order. All of this contributes to language’s simplicity: administrator should easily grasp concepts without any specific training.

```xml
<simpleType name="SOCKET_permissions">
  <restriction base="string">
    <enumeration value="CREATE"/>
    <enumeration value="CONNECT"/>
    ...
  </restriction>
</simpleType>

<simpleType name="IPSEC_modes_type">
  <restriction base="string">
    <enumeration value="ESP"/>
    <enumeration value="AH"/>
    <enumeration value="NO_SEC"/>
  </restriction>
</simpleType>

<complexType name="DisCI_modes_type">
  <sequence>
    <element name="IPSEC_mode" type="dsi:IPSEC_modes_type"/>
  </sequence>
</complexType>
```

Figure 5 XML schema for actions of the SOCKET and the DisCI class.
Another important point for cluster security policies is their adaptability to the specificities of cluster architecture. DSPs handle this by modeling DSI’s ScID/SnID view of clusters. In addition, they focus on representing cluster states — and not simply a single node’s state.

Finally, the last requirement concerned flexibility. Of course, this is partly taken in charge by XML’s intrinsic extensibility, but beyond language’s flexibility, it is important to note DSPs are not limited to an access control model. They may express quality of protection issues (notably via the DisCI security rule class), or auditing, logging, etc.

Distributing the security policy

In the previous section, we have proposed a security policy language which is (1) easy to use (extensible, human readable and simple), (2) flexible enough to extend to any distributed security service (i.e. not limited to access control rules only), and (3) capable of describing characteristics of clusters’ environments. From the requirements’ list we made in first section, we still have to cope with performance and ease of deployment on clusters: this is what this section concentrates on.

Ease of use is an important issue for security policies. Similar to Verma (2002) where the advantages of policy based management are highlighted for large network infrastructures, managing security of a cluster is a daunting task. Writing and manually enforcing a security policy on each node of a cluster would be so cumbersome from an administration point of view. This is why, as a second part to our contribution, we propose a propagation mechanism which automatically distributes our DSP to all nodes of the cluster.

We explain more in details hereafter how this mechanism works.

Initially, the DSP is created and updated on the Security Server (SS — please see section ‘The DSI framework’). Then, the SS is responsible to perform syntactic and semantic checks on the DSP. After validation of the DSP, we enter the actual propagation stage where (1) each security rule update is transformed into a simplified rule message, (2) the message is distributed to all nodes over an encrypted and authenticated channel, and (3) each SM finally enforces the rule at kernel level in the DSM. Fig. 6 summarizes the whole process.

For readers familiar with the IETF DMTF policy framework (The IETF Policy Framework Working Group), administrator’s favorite XML editor can be seen as the policy management tool, the SM is a policy decision point, the DSM is a policy enforcement point, and, similar to Verma (2002), the DSP propagation mechanism we propose offers centralization of management.

Let us explain more precisely the different steps of propagation. The syntactic validation consists in checking the DSP for typical XML syntax errors, unknown tags or invalid values. This validation is done against the XML schema of our security policy language. The semantic validation concerns the meaning of the security rules. It checks there are no explicit contradictions indicating obvious input errors, and then tries to group similar rules together. Currently, only simple cases of contradictions and similarities which concern the same subjects and resources have been studied. Some typical examples are shown in Tables 1 and 2.

Now, from a performance perspective, the propagation mechanism proposes two features. First, we reduce communication volume between the SS and SMs, by sending only modifications to the current policy. This is done on the SS by comparing the new set of security rules and the current one, and then, for each update, sending a removal of the former rule followed by an addition of the new one.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>An example of contradiction in a DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transition rules</td>
<td></td>
</tr>
<tr>
<td>Parent ScID</td>
<td>SnID</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

It is impossible for a same binary, with the same parent context to “transit” either to ScID = 20 or 50: this would not be deterministic.
The second feature consists in a single point of processing for DSPs. Once the DSP has been analyzed on the SS, the actual addition or removal messages that are sent to each SM do not use XML but a very simple and pre-processed syntax defined in DSI. This syntax is extremely close to the communication syntax between the SM and the DSM, so that SMs only have very minor processing to do when they receive messages from the SS. The reason to have two different syntaxes — XML for the DSP and DSI’s syntax for propagation to the SMs — is the following. If we had chosen an “all XML” solution, an XML message would have been sent to all n SMs of the cluster. Then, an XML parser and validator would have been required for each SM. So, we would have ended up parsing and validating n times each message. As those are only internal messages, we preferred parsing and validating the DSP only once on the SS, and then assume the SS would send correct messages to the SMs. With this choice, we are able to keep SMs and DSMs much lighter, which has beneficial impact on performances.

Finally, to complete explanation of the propagation mechanism, when the SM receives the addition/removal messages, it simply forwards them to the DSM (in kernel space). The DSM stores security rules in a hash table for fast access, with the triplet (ScID, SnID, class) as key. As security rules expressed in DSPs may use wildcards (ALL), there is not always an exact match between the action that is being checked and the security rules. We have consequently designed an algorithm which is in charge of finding the best rule as quickly as possible. The details of this algorithm are out of the scope of security policies presented in this article, however more details may be found on DSI’s web site at Sourceforge.

To conclude, with this DSP propagation mechanism, administrator can manage their clusters from the SS only. They do not need to bother about how to enforce a security policy on each node of their cluster. Furthermore, we emphasize that, as the DSP is capable of addressing specific cluster nodes, it is possible to centralize all rules for all nodes in a same and unique policy. We believe such a feature is very valuable because it simplifies the maintenance process, helps greatly reduce risks of vulnerabilities due to misconfiguration of complex distributed systems.

**DSI prototype results**

In practice, DSI has been implemented for Linux-based clusters. Most components (SS, SM, the policy update tool, etc) work in user space. As for the secure communication channels between the SS and SMs, it is implemented using SSL/TLS over CORBA event channels. Consequently, the addition and removal security rule messages discussed in previous section are propagated to each SM as CORBA events, and they use a channel configured to guarantee confidentiality, integrity and mutual authentication.

The only component to work in kernel space is the DSM. It is implemented as a kernel loadable module, for Linux 2.4.17 kernels, and has recently been successfully ported to 2.5.66. It receives DSP addition and removal messages the SM receives through a dedicated character device: this achieves communication between user-land and kernel land. Then, it updates the DSP rules and stores them internally in a hash table (see previous section). Finally, it relies on several LSM hooks (Wright et al., 2002) to enforce the various security services the policy specifies.

On a performance point of view, we have benchmarked DSI with LMBench 3.0, on a Pentium IV 2.4 Ghz machine, running a Linux 2.5.66 kernel.

In the following, we illustrate how inter-node communications are controlled in the DSP with the example of a two-node cluster.

On one hand, a UDP server is installed on node SnID = 1. It is capable of (1) creating a UDP socket, (2) binding on port 9000, and (3) receiving test messages. On the other hand, a UDP client runs on node SnID = 2. It simply connects to the socket and sends a few messages. Both UDP client and server use default ScID = 2 (see Fig. 7).

The first step consists in explicitly assigning ScID = 10 to UDP sockets on port 9000 (see rule 1 in Fig. 8). Note that socket on client’s side is automatically assigned its parent’s ScID, i.e. ScID = 2.

Then, the UDP server (source ScID = 2, SnID = 1) needs permission to create the socket (target ScID = 10, SnID = 1) and, reciprocally, the UDP client needs permission to create client socket (rules 2 and 3 in Fig. 9).
Also, the server socket needs to be able to receive messages from the client socket (source ScID Z 2, SnID Z 2). On the other side, the client socket must be allowed to connect and send messages to the server socket. All those permissions are grouped in rule 4 in Fig. 10. The fifth rule allows server’s node (SnID Z 1) to receive packets from subjects with ScID Z 2, SnID Z 2 (client side). Without this final rule, communication between nodes is blocked in the IP stack and never gets to the server socket.

Finally, note this security policy is typically written on a single node (for instance, SnID Z 1). It is then automatically propagated to the other node via addition security rule messages on a secured CORBA event channel. Naturally, for a two-node cluster, the benefit is minimal but, in real-life, with several nodes, this considerably eases administrator’s task.

**Not inherently distributed languages**

These languages express the security policy for an individual node. One of the best examples of these languages is Java security language (Gong, 1998). Intrinsically, the security policy languages for nondistributed systems do not easily adapt to the distributed environment as many fundamental concepts like locality of the security entities cannot be expressed in these languages. By locality, we mean the node where the process (cf. application or software component) runs. For a cluster, from a security point of view, this information is an important part of its security context. For example, a process (application or software component) may only run on a defined set of nodes. The security policy can forbid that process from running on sensitive nodes and its presence on those nodes can be interpreted as an indication of a possible security breach.

One workaround consists in mapping the distributed security policies to each individual local security policy. For instance, the security policy for a distributed application needs to be mapped to the local security policy of each node it operates on for each different processes the application involves.

However, the mapping is a complex cumbersome task and error prone. In practice, this exercise ends up always lacking to map perfectly the distributed policy into individual local policies, exposing gaps to the intruder for an attack.

**User centric languages**

Many security languages, though presenting well the concept of locality of elements, are user centric. One example is SAML (Hallam-Baker and...
Maler, 2002). SAML is mainly a request/response protocol around authentication and authorization for users (called “subjects”). Its first phase is much concerned about single sign on and the later phases mainly concern the digital identity for different users in the Internet. The main functionality is defined for authenticating users/subjects in an interactive way, then carrying around the authentication assertion from one locality to another. However, in our clustered servers the DSP is used internally to compartmentalize the distributed applications. Therefore, the authentication and authorization rather concern processes dynamically created in the system and not the real users or subjects. This makes the interactive aspects of the protocol rather difficult to be implemented on run time at kernel level. Furthermore, the SAML framework supposes the authentication is performed at a distinct entity: identity provider. Once authenticated the user carries the authentication assertion around for authentication purposes. This is a natural bottleneck and single point for failure for a cluster. Definitely, even though a mapping could be possible between users/subjects and processes, from performance point of view this approach is unrealistic in a server dynamically creating many processes per second to handle the incoming requests. This is because this approach necessitates many real time interactive authentications between these processes and the identity provider.

**Access control centric languages**

These languages are the closest to DSP. One example is XACML (Godik and Moses, 2003). They express the access control rules for different users. XACML is a generic language that can be used to express DSP like rules. Though in XACML, security rules are to be matched against XACML contexts. For clusters, this means that the access control mechanism of each node – which is located in kernel space – would have to generate an XACML context for every single access control request. For DSI, this would result in major negative impact on performances, because intensive work would be done at kernel level. Furthermore, XACML does not cover generic functionality like general security context for example paranoid or permissive mode or more specific security level used for communications (cf. DisCl in section ‘The DSI framework’).

**Figure 9** XML socket rule giving permissions to create sockets.

```
<!-- Rule 2 -->
<class_SOCKET_rule>
  <sScID>2</sScID>
  <sSnID>1</sSnID>
  <tScID>10</tScID>
  <tSnID>1</tSnID>
  <allow>CREATE</allow>
</class_SOCKET_rule>
```

**Figure 10** XML socket rule assigning specific socket permissions and XML network rule unblocking network communications for a given ScID/SnID.

```
<!-- Rule 4 -->
<class_SOCKET_rule>
  <sScID>2</sScID>
  <sSnID>2</sSnID>
  <tScID>10</tScID>
  <tSnID>1</tSnID>
  <allow>CONNECT SEND RECEIVE</allow>
</class_SOCKET_rule>
```

```
<!-- Rule 5 -->
<class_NETWORK_rule>
  <sScID>2</sScID>
  <sSnID>2</sSnID>
  <tSnID>1</tSnID>
  <allow>
    NETWORK_RECEIVE
  </allow>
</class_NETWORK_rule>
```
Conclusion and future work

This paper has focused on the definition of an adequate security policy for clusters. We have shown that existing security policies are unable to handle all security aspects distributed infrastructures deal with.

To solve this problem, an XML Distributed Security Policy has been proposed. It integrates the concept of cluster nodes through security identifier attributes, and also defines an adapted set of states, in particular, inter-node communication permissions. We believe that our approach is general enough to be used for clusters with need for fine grained access control at cluster level.

This paper has also proposed a propagation mechanism of the security policy, which automatically enforces the same policy on all nodes at kernel level, and helps cluster’s administration.

Concepts presented in this paper have successfully been implemented over Linux-based clusters and are part of the Distributed Security Infrastructure project. However, there are still multiple areas to investigate such as improving the validation step to detect more contradictions or reduction cases, adding policy rules to control communication with machines outside the cluster, and integrating rules for new security services.

References


Pourzandi M, Apvrille A, Gingras E, Medenou A, Gordon D. Distributed access control for carrier-class clusters. In: The proceeding of the parallel and distributed processing techniques and applications conference (PDPTA’03); June 2003.


Uncited references

Lopez et al., 2002; Sandhu et al., 1996.

Acknowledgments

For this paper, we’d like to thank Eric Gingras and Alain-Patrick Medenou for their initial work on DSI’s security policy. We are also very grateful to David Gordon for his valuable technical help regarding how security rules are processed at kernel level in DSI.