Enforcing Security Enhanced Linux policies in a networked policy domain

Joshua Brindle, Karen Vance, Chad Sellers
Tresys Technology

Abstract

Significant progress toward general acceptance of applying mandatory access control to systems has been made recently. Security Enhanced Linux (SELinux), in particular, has been enabled by default in some Linux distributions for several years. However, current SELinux deployments only effectively control a single system. Inter-system and remote resource access control capabilities are starting to appear in SELinux, but extending policy management capabilities to cover these networked systems remains an open problem. This paper discusses issues that must be addressed to support security policies distributed across a network, including policy development changes needed to be able to express a coherent security policy for a network of systems, managing the distribution of a multi-system policy, and synchronizing policy updates, and compares options to provide this support.

1. Introduction

Infrastructure to manage Security Enhanced Linux (SELinux) systems is starting to mature and user-friendly interfaces to use this infrastructure are on the horizon, but these are limited to single systems. Even in cases where multiple system management is possible, many assumptions are made about the similarity of the systems, their access matrices, and the homogeneity of the network [9] [10].

Supporting distributed systems is the next logical step for SELinux. Distributed systems are becoming much more prevalent as cheaper hardware and technologies such as hypervisors make it more economical to run many systems for redundancy and integrity than in the past. Current efforts are developing the infrastructure needed to extend security mechanisms to enforce mandatory access control (MAC) between systems and on remote objects. This will change many of the assumptions made by current SELinux deployments, but will especially affect the policy.

Current SELinux Reference Policy [1], which is now used in all SELinux distributions, assumes that all systems running the software protected by the policy are relatively the same. This may be a fair assumption for isolated systems but the ability to enforce inter-system access breaks that assumption. Therefore, the existing concept of security policy must be extended to handle a network of systems that belong within a single policy domain, but in which similar applications may be treated differently depending on the system upon which they run or the manner in which they are accessed (e.g., locally vs. from a remote machine). Within a network of systems, there will be many security servers, each responsible for providing security decisions to a set of object managers, each responsible for enforcing security decisions on their objects [11]. The objects can also be very similar, for example `/etc/shadow` on system 1 and `/etc/shadow` on system 2 may be identical but are different objects controlled by different object managers.

Within a policy domain enforced by a single security server, the security label namespace could be considered to be unique and any objects that the policy administrator wishes to be treated the same will be given the same security label. In the same way, objects that should be treated differently – even if the objects are similar as described above – can be given distinct labels. When extending beyond the realm of that single security server, the namespace can no longer be considered to be unique. Objects may have the same security label associated with them but need to be treated differently by different security servers, or objects that the policy administrator would like treated as equivalent could have distinct security labels.

In addition to the above, since the namespace is no longer guaranteed to be coherent, analysis of the entire security policy must be handled differently. Current analysis tools assume that the security policy is within a single namespace. When multiple namespaces are introduced, a means of reconciling these namespaces must be provided.

We will be discussing two ways of addressing these challenges. The first provides a coherent policy domain-wide security policy. The second maintains the individual security policies but introduces mechanisms to synchronize points where the security policies overlap, i.e., where the systems interact. This paper discusses how each of these strategies could be implemented and analyzes the strengths and weaknesses of each method.

For purposes of discussion, this paper assumes that all interacting machines are running SELinux or provide object and subject labels. There is no attempt to handle interactions between enforcement mechanisms. Furthermore, all machines are within a single administrative domain – there is no attempt to address issues of negotiation and trust between policy domains. Finally, some of the applications on the systems may be active participants in the security policy interactions between the entities, but for the most part the applications running on these systems are unmodified and are unaware of the inter-machine interactions taking place. This assumption must be made, since it is unrealistic to require that existing applications be replaced to interact with another entity. In some cases however, aware applications may be required to participate in the access control. For example, to limit access to only certain rows or columns of a database, the database server must participate in the access control, since an external mechanism would be unable to apply security contexts at that granularity.

2. Inadequacies of existing policies

Existing policies are created for a single system, i.e., they only express access control for subjects and objects on one system, whether it is an installation on a computer, in a virtual machine, or on an embedded device. The single system policy may specify network access controls, but it has no way to synchronize access on subjects or objects of another system. With the security contexts being extended to use in a network of systems, a traditional single system policy no longer is sufficient.

2.1 Single system access control

Before delving into the challenges of multi-system access control, it is important to define existing access control in SELinux systems. Access control systems specify the allowed access between subjects, which are the active entities – i.e., processes – on the system, and objects, which are the passive entities on the system such as files. With type enforcement (TE), the primary access control mechanism in SELinux, every subject and object is given a context that comprises the security attribute used to determine what access allowed.

The Flask framework [4] upon which SELinux is based already implements distributed access control within a single system; each object manager controls its own objects and asks the security server for access control decisions. In this way, the enforcement itself is distributed throughout the system. Distributed Trusted Operating System (DTOS) [3] provides a good example of distributed access control since it is a microkernel architecture and therefore has object managers running in separate memory spaces. SELinux uses the same concepts but most of the object managers are part of the monolithic kernel. Notable exceptions are the user space object managers X Windows, DBUS, and passwd, all of which control their own objects.

Since the underlying mechanism can already support distributed access control much of the initial work to support this has already been done. The effort then falls on effectively scaling this mechanism to entire networks where before it was only effectively applied to single systems.

2.2 Expressing network-wide security goals

The first step in policy development is that of determining security goals the policy must satisfy. Existing SELinux policy assumes these goals relate to single systems and network borders. However, with security becoming more of a concern on networks, the capability to define and enforce network-wide security goals must be supported. A network-wide security goal may involve separating internal only and external data or limiting data access to customers and employees or even locking down workstations. Whether using a single system-wide security policy or maintaining individual security policies, these goals must be both attainable and analyzable.

2.3 Analyzing policy domain interactions

Types are unambiguous security equivalence classes only within the realm of a security server, and are the primary security attribute used by SELinux in the TE security model. Every subject and object has a type,

and any subject or object with the same type is treated identically by SELinux. When performing policy analysis, the TE mechanism and therefore the types are the primary consideration [11]. However, once outside the realm of that security server, label translation may be necessary in order to perform analysis of the combined policy.

On a single system, suppose there are two Apache web server instances, one that is for internal use only and one that is for external use. Each Apache server would need different types; otherwise the internal only Apache server could be accessible externally. Separating them into different types ensures that through analysis the policy will not allow internal Apache instances to be accessible externally. If the two web server instances reside on different machines and have the same type, even though they are on different systems this guarantee cannot be made without some mechanism for specifying equivalences between types.

In some cases, systems are identically configured. In cluster systems, for example, each system would have the same policy and it is likely that each instance of an application would share the same type. This is what gives SELinux most of its flexibility. Subjects that need to be treated the same use the same type; otherwise they get a different one. Obviously, one can have some portions that should be treated identically and other portions that must be handled distinctly across systems – in the Apache example there may be an internal and external Apache that must be handled differently, but every DNS server may be the same and so the same type would be used for each of them.

In all cases, analysis of a network policy requires types to be equivalent between systems, either by using a single policy that covers all systems being protected or by providing an equivalence mechanism. An analysis of the policy then can ensure the same guarantees that analyzing a single system policy can.

2.4 Handling distributed subject contexts

MAC has been used across systems for quite some time. Systems using the Bell LaPadula (BLP) model [6] have been able to propagate contexts to other systems for enforcement using technologies such as CIPSO and RIPSO [2]. These technologies are of limited use for SELinux, however, as they were designed for Multi-Level Security (MLS) systems, which have inflexible policies and homogeneous security contexts. Supporting flexible MAC introduces a whole new set of issues that must be addressed.

Recent changes to SELinux have introduced the ability to propagate contexts to other systems. IPSEC [8] protects both the integrity and confidentiality of the communication between SELinux machines and the context that is propagated. While this is a very significant advancement, it also means additional policy development and analysis effort. The traditional single system policy development model is illustrated to be further insufficient in the networked environment, since a context that is used to access other systems may have different security properties than the local equivalent would have.

Consider the example of a database server and client. For a client running on the same system as the server, its type might be `database_client_t`. However, if there is a client on another system that wants to access the database remotely, a decision must be made. Is the remote client equivalent to the local client? If it is equivalent, its type should be `database_client_t` on both systems. If the two were not equivalent, however, they would need different types. Suppose that the local client was primarily used for administrative purposes and therefore needed additional access; it might then instead need the type `database_admin_client_t`. Therefore, although the clients are the same application on different systems, they may need to be treated distinctly to meet the policy domain security goals. While this may be a simple concept when considering a single client and server on two systems, extending the concept to a network of workstations with different security properties and servers with different services vastly increases the complexity of the problem.

2.5 Handling distributed object contexts

In addition to subject contexts being used across systems, labeled network file systems bring the possibility of object contexts traversing the network. This means the network policy need not only treat subjects distinctively but also objects. Using the previous Apache example, if the two Apache instances are serving different data internally and externally, the data would need different types. This prevents the external Apache from accidentally or maliciously disclosing internal data whether it resides on the same system or a labeled network file system gets mounted to the system on which it runs.

Again, on single systems, assuming each Apache instance is on a different system, this would be unnecessary; on a distributed system the data and its context can be accessed on different systems, however. It would likely violate the policy domain's security goals if the external web server could access the internal data, even if the data is on a network file system that is accessible by the external server. A network-wide policy must take these matters into consideration.

3. Representing a policy domain using a single networked policy

In the previous section, we determined that the existing policy mechanisms are inadequate to represent a distributed system. One option is to represent the entire distributed system using a single networked policy. Further issues arise when determining the best way to express and develop a policy in this manner. Enforcing access control within a network of systems also introduces new challenges in policy management and distribution. Relevant portions of the coherent policy must be sent to the individual systems without unnecessary overhead. To this end, the policy must be partitioned and only the appropriate sections sent to individual systems. Finally, extra care must be taken in ensuring the atomicity of policy changes. While this is a concern on single systems, the problem is amplified when many systems must enforce a coherent policy on a high latency medium.

3.1 Preserve equivalence where possible

Although types must be different between systems in some cases, the policy should not introduce unnecessary differences. Types are security equivalence classes; if two subjects or objects have the same security properties on different systems, they should use the same type. For this reason, it is not advisable to adopt a policy-wide mechanism for separating all types on a per-system basis. One should not require, for example, that the shadow file on system 1 has the type `system1_shadow_t` and the shadow file on system 2 has the type `system2_shadow_t`. If these shadow files have the same security properties, they should be a part of the same equivalence class. Requiring per-system types would result in a policy far bigger than necessary and much more difficult to manage inter-system access on a large scale.

Instead, if two or more systems have very similar duties, e.g., load balancing mirrors or redundant routers, it is likely that they will have very similar or even identical policies. This concept can, and should, be applied on a per-application basis. As mentioned earlier, if two Apache instances need different types but the DNS servers on the same systems do not, the DNS servers should share the same type while distinct types should separate the Apache instances.

3.2 Unified namespaces

Each part of an SELinux policy – types, roles, users, etc. – has a namespace that ensures uniqueness. When a single coherent policy is used for a network of systems, these namespaces must be unique across the entire network. Any type in the policy has the same policy associated with it no matter what system is enforcing access.

This follows the type enforcement methodology of using types as equivalence classes and is conducive to policy analysis. It should also make the policies and systems generally more legible and comprehensible. A policy analysis should not have to consider whether or not a subject type, for example `httpd_t`, on one system has the same security properties as that subject type on another system.

3.3 Analysis of a single networked policy

Analysis of the resulting policy would work very much the way it currently does. Since there is a single canonical version of the policy applied to the entire network, an analyst can use it to study information flow throughout the entire network instead of only a single system.

3.4 Multi-system policy distribution

For a single canonical version of the policy to be applied to each system in a network, an intelligent distribution scheme must be introduced. Simply sending the entire policy to each system does not scale well in large networks. A policy management server (PMS) [5] provides these intelligent partitioning and distribution mechanisms for a network. Its functionality is described in the subsequent sections.

3.4.1 Partitioning policy

The SELinux policy is surprisingly easy to partition. The TE policy, which is the vast majority of the SELinux policy, is keyed on the source type, the target type and the object class. With the Flask architecture, each object manager enforces access on its objects. These objects in turn are part of some object class, for example file, shm (shared memory) or even window with X Windows as an object manager. Since object managers can only enforce on their objects, they would have no need for policy relating to other object classes. In this way, policy partitions can be made according to object classes.

The object managers ask the SELinux security server for access control decisions, which in turn queries the policy. The security server, therefore, only needs policy for object classes that its object managers can enforce. If, for example, X Windows were not present on a web server, the security server on that system would not need policy for the window object class.

Since the policy uses object class as part of the key, a policy can be produced for each system that has inapplicable object classes removed, without changing the intent of the policy. If an object manager subsequently is added to a system, the policy for that system would need to be reproduced with the object classes that are necessary. Figure 4.1 shows how such a system might look.

This model also allows for multiple security servers on a single system. For example, a system may have a kernel, user space, and hypervisor security server. Each of these security servers would provide access decisions to the object managers for which they are responsible. In this case, the system's policy would be a conjunction of the object classes needed by each of the security servers for their object managers.

3.4.2 Distributing relevant policy

The above scheme gives the ability to remove some of the policy that is not applicable to a given system, but most systems will have the standard Linux object classes such as `file`, `dir`, `tcp_socket`, and so on, so this does not break up the policy sufficiently. A further possibility is to determine types that are applicable to a system and remove the irrelevant rules. However, it may be very difficult to determine which types are applicable to a system since new subject and object types can come from network sources.

Even without network labeling, it is not trivial to determine every type applicable to a given system. It may be possible to get a set of object types from context files and subject types from type transitions on those object types, but types set by SELinux-aware applications or administrators would not be included in the policy, which could result in a non-functioning system.

To address this issue of which types are appropriate to send to a given system, hierarchical policy could be used to specify a starting set of types needed by a given system to get it running. However, this would be a tedious burden to place on the policy author to specify which pieces of policy went to which system on the network, so this is not desirable.

Additionally, once running, the system may encounter types that were not part of its policy as a result of contexts traversing the network, either on network file systems or over labeled network sockets. If systems interact with one another frequently, they could be put in a group so that each system gets the types. In the case where interaction is not common but otherwise allowed, the security server must retrieve policy in order to know how to apply access controls to the subject or object. The appropriate policy for the unknown subject or object is fetched by the policy management server, which combines it with the existing policy for the system and sends the new policy to the system.

There is significant overhead in the policy fetch process. This should be considered a corner case as the common case is that each system has all the policy it needs, but this is still a problem. However, having additional policy on the system in case it was needed would use resources unnecessarily, so this is not a good option either.

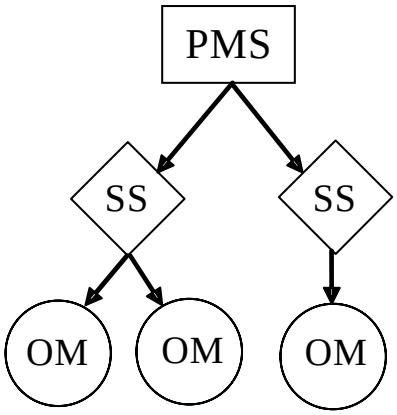

Figure 4.1

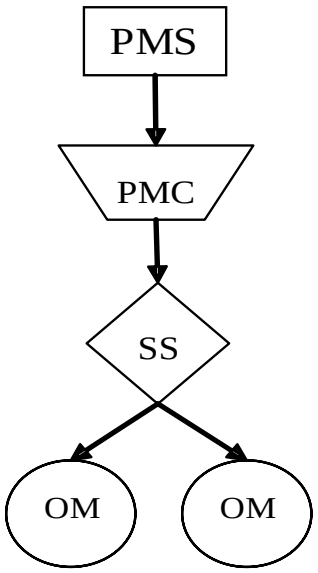

Figure 4.2

3.4.3 Multi-system policy management infrastructure

As Figure 4.1 shows, each object manager would know what kinds of objects it manages and the security server it will use to get access control decisions. The object managers, therefore, are responsible for informing the security server of the object classes for which it requires access decisions. The security servers then request policy from its policy management server for those object classes. The policy management server must know the SELinux identifier for its system and to which policy groups that system belongs, something that is likely to be a configuration setting. Unfortunately, each system would need a policy management server with this architecture.

Rather than forcing every system on the network to have policy management servers running and be policy management peers, a lighter weight daemon called the policy management client (PMC) could be installed on most systems. Figure 4.2 illustrates that the PMC behaves similarly to the PMS, except that it only receives policy from a PMS and does not send policy to other systems. The PMC still manages policy for its security servers and also alerts a PMS when it needs additional policy.

The policy management by administrators or security engineers would be done with tools like semanage and semodule, which would be extended to be able to send updates to any PMS. These changes would then propagate to other PMS' and down to security servers and object managers.

4. Providing inter-system equivalences to coordinate inter-system policy interactions

An alternative to the single network policy described in section 3 is to have individual and distinct policies on each system within the administrative domain, and then provide mechanisms to create the equivalences for interacting machines, i.e., translating the incoming object and subject labels to something that has meaning on the local machine. Note that this could result in translations on both sides of an interaction, where each machine creates a local equivalence for the across-the-network labels. The policy could be further separated to have distinct policies for each security server on a single system. This would allow each security server and object manager to have policy relevant to them without using unnecessarily using system resources.

Current SELinux mechanisms only support inter-machine transfer of subject labels. However, future modifications could allow for object labels to be transferred, so both should be considered in this implementation.

4.1 Administering policy equivalences

For this option to work, administrative tools must be put in place to allow the policy administrator to specify the local equivalences. Once the equivalences, or translations, are specified, the policy management infrastructure must be modified to perform this translation when object or subject labels are received from another machine.

4.1.1 Inter-system equivalence

Since this option allows for systems with disparate policy namespaces, but requires interactions between those systems, a mechanism for extending equivalence classes across namespace boundaries is necessary. So, a mapping between namespaces is necessary. This mapping could be to a common network-wide namespace, or could be a simple mapping between two system namespaces. The latter option would require a separate mapping table for each external system interaction, which may or may not be desirable. This mapping also need not be a one-to-one mapping, but instead will likely be a many-to-one mapping. This means that it is possible that multiple labels on the remote system will translate to a single label on the local system. The reverse, however, is not true, as all remote labels must translate to one and only one label on the local system.

An example of this may be that all users on a workstation are logged into a server as unprivileged user. The mapping would therefore say that user_t, staff_t and sysadm_t all map to user_t on the server.

The management infrastructure on each system must be able to support setting up these tables. Then tools can be built on top of this infrastructure to provide the policy administrator with the ability to initiate new relationships and to specify new equivalences.

4.1.2 Infrastructure translation

Since SELinux subject labels are currently transferred using racoon, the ISAKMP server for IPSec on Linux, it must be modified to request context translations from the SELinux infrastructure. This work is already in progress and allows a server to be written that uses a local socket to communicate with racoon and provide translations.

Network object contexts are handling in the Linux kernel so they must be handled separately. More research must be done to determine the best method of providing context translation services to an in-kernel object manager.

It is likely that other object managers will need translation services, anything that uses objects with contexts provided by a security server other than the one in use by that object manager will potentially need translation services. A general solution that provides translation services to all object managers requiring it would be ideal.

4.2 Developing network policy

Having distinct system policies does not preclude an administrator from developing network-wide policies. Instead of handling this through infrastructure, as in the single network policy option, this could be handled through development tools. Development tools can allow specification of network-wide policy, and then generate the machine-specific policy and translation tables necessary to enforce this network-wide policy. These tools can potentially provide a policy administrator with very similar functionality to a similar network policy, while providing the increased flexibility of disparate system policies.

In this model, the individual systems on the network do not have to be completely subject to a policy authority on the network. They can each have individual policies, and only require coordination of the pieces of policy expressing network interactions. So, a system can have a standard base policy with additional policy modules and translation tables for network interactions. This provides a simpler administrative scheme than a completely centralized policy for an entire network.

4.3 Analyzing policy for a policy domain

Analysis tools must also be expanded to handle multi-system interacting policies. These tools must be able handle the label equivalences that the policy administrator has specified.

In order to fully analyze the security of a network of systems, it is necessary to look at all the policies collectively, as well as the interactions between those policies. So, analysis tools must be modified to look at multiple policies and keep track of their separate namespaces. Additionally, these tools must be modified to provide linkage between these disparate namespaces by

utilizing the translation mappings. Then, current analysis techniques such as information flow analysis can be performed on an entire network of systems.

5. Conclusions

To effectively apply mandatory access control to networks and distributed systems, the single system policy paradigm must evolve. Either a single coherent policy must be developed for all the systems in a policy domain to ensure network security goals are met and to allow for effective analysis or the individual system policies must be synchronized at points of overlap. If using a single policy for the policy domain, it must be intelligently partitioned and distributed to send only the necessary pieces to each system, to reduce the overhead of distributing policy over a possibly high latency network. Furthermore, all policy distribution must be synchronized so that the policy across all systems is consistent at any given time. If synchronizing the individual system policies at the points of overlap, the subject and object labels being sent between systems must be translated to provide local equivalences. Furthermore, analysis tools must evolve to support these new equivalences.

In general, it is unrealistic to expect that an entire policy domain can be controlled using a single policy. Therefore, the option of synchronizing individual system policies must be explored in greater depth.

Finally, although the need to express distributed policies has prompted augmentation to the SELinux infrastructure and policy language, these changes will greatly facilitate the process of developing, distributing, and enforcing network policies.

6. References

[1] PeBenito, Christopher, Frank Mayer, and Karl MacMillan. "Reference Policy for Security Enhanced Linux.." In *Proceedings of the 2006 Security Enhanced Linux Symposium*, Baltimore, Maryland, March 2006.

[2] "COMMERCIAL IP SECURITY OPTION (CIPSO 2.2)." IETF CIPSO Working Group. <http://ietfreport.isoc.org/all-ids/draft-ietf-cipso-ipsecurity-01.txt>.

[3] Minear, Spencer E. “Providing Policy Control Over Object Operations in a Mach Based System”. In *Proceedings of the 5th USENIX Security* Symposium, Salt Lake City, Utah, June 1995.

[4] Spencer, Ray, Stephen Smalley, Mike Hibler, David Anderson, and Jay Lepreau. “The Flask Security Architecture: System Support for Diverse Security Policies.” In *Proceedings of the 8th USENIX Security Symposium*, Washington, D.C., August 1999.

[5] MacMillan, Karl, Joshua Brindle, Frank Mayer, David Caplan, and Jason Tang. “Design and Implementation of the SELinux Policy Management Server.” In *Proceedings of the 2006 Security Enhanced Linux Symposium*, Baltimore, Maryland, March 2006.

[6] Bell , D. E. and L. J. La Padula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244

[7] Secure Computing Corporation. "Assurance in the Fluke Microkernel." Technical Report. 1999. <www.cs.utah.edu/flux/fluke/html/final.ps.gz>.

[8] “Security Architecture for the Internet Protocol.” Network Working Group. <http://www.ietf.org/rfc/rfc4301.txt>

[9] MacMillan, Karl. Address. Tresys Technology. 2006 Security Enhanced Linux Symposium, Baltimore, Maryland. March 2006. <http://selinux-symposium.org/2006/casestudies.php#websphere>.

[10] Ashworth, Christopher, and James Athey. “Developing SELinux Management Tools.” In *Proceedings of the 2007 Security Enhanced Linux Symposium*, Baltimore, Maryland, March 2007.

[11] Mayer, Frank, Karl Macmillan, and David Caplan. SELinux by Example. Prentice Hall, 2006. .