



Lessons Learned Developing Cross- Domain Solutions on SELinux

March 2, 2006

Karl MacMillan, Spencer Shimko, Chad Sellers,
Frank Mayer, and Art Wilson

Copyright ©2006 Tresys Technology, LLC. All Rights Reserved.

Other names and brands may be claimed as the property of others. Information regarding third party products is provided solely for educational purposes. Tresys Technology, LLC is not responsible for the performance or support of third party products and does not make any representations or warranties whatsoever regarding quality, reliability, functionality, or compatibility of these devices or products.

Abstract

Building computer systems that allow the controlled transfer of data between security domains, commonly called cross-domain solutions (CDS) or guards, presents many common and some unique security challenges. In this paper, we explore lessons learned from building several CDS systems on SELinux. We explore the desired security properties of a CDS, define the role of the operating system in enforcing these security properties, and describe our experience using SELinux to fulfill the operating system role.

1. Introduction

The need to transfer data between security domains, typically in the form of a network, is common both in industry and in government. Firewalls, email gateways, and other types of industry standard network edge protection devices are examples of systems that address this need. These devices typically attempt to protect devices on an internal network from the larger internet by limiting or filtering network data. By controlling the flow of information across the network boundary, a firewall or other network edge device reduces the chances that a successful intrusion can be accomplished.

Connecting two or more networks that contain sensitive or classified information adds additional requirements beyond integrity protection. A common scenario in US and other government networks is that networks are separated based on the highest classification approved to be transmitted over a network. To prevent unwanted disclosure, it is assumed that all data on these networks is classified at the highest level authorized for the network. This greatly reduces the level of trust placed in the devices attached to these networks and makes it practical to use commodity software. It also means that the networks must be kept strictly separated; even though a top secret network, for example, might contain data only classified as secret it is not possible for a device on a secret network to connect to the top secret network to receive this data.

Cross-domain solutions (CDS), also called guards, are employed in scenarios where it is required that data be transmitted between security domains in a controlled way.

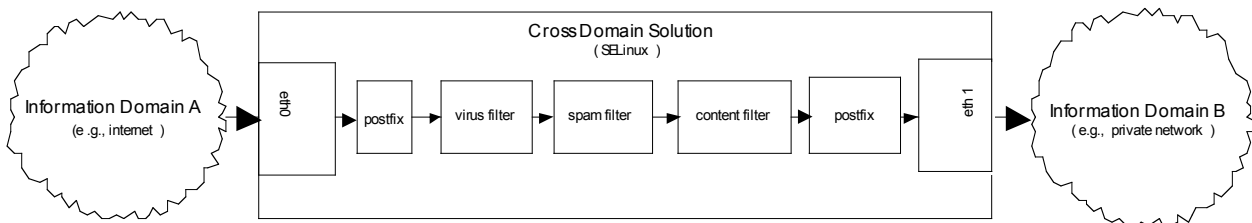


Figure 1 - Example Cross Domain Architecture

In this paper we will present lessons learned from our participation in creating several CDS systems based on SELinux. In general, the use of SELinux has been very successful. Type Enforcement (TE) is an excellent mandatory access control (MAC) mechanism for CDS systems and has allowed the creation of architectures where the operating system does much of the “heavy lifting” for the overall system security, reducing the security burden on the CDS-specific applications. This does not mean that there have not been challenges; in particular we had to develop disciplined design policies to strictly control the flow of information through the CDS system.

2. Cross Domain Solution Architecture

Figure 1 shows an example of a simple CDS architecture. This idealized system is an smtp email relay; it is intended to present the conceptual elements of what we believe is an appropriate CDS architecture and not represent an actual CDS system. For the purposes of this paper, assume that this example CDS is intended to connect a private network, on which there is highly sensitive material (e.g., medical records or corporate secrets), to the internet only for the purposes of receiving email. In order to assure that no sensitive information can be leaked, the CDS will only allow data to flow from the internet to the private network; no data from the private network will be allowed to reach the internet.

The main feature of the architecture depicted in Figure 1 is a processing pipeline, modeled on assured pipelines [1], that receives incoming email, runs the email through one or more content filtering applications (e.g., virus or spam filtering), and relays the email to one or more internal mail servers. Each of the boxes in the diagram represents one or more processes in a single TE domain type. These processes communicate using some form of inter-process communication (IPC).

2.1 Architecture Goals

The purpose of this paper is not to discuss the design goals of this example architecture, but rather to examine the challenges of implementing this architecture with SELinux. However, a brief discussion of the architecture security goals will help inform the discussion of the implementation challenges.

The overall goal of the architecture is to create a solution that reduces the trust placed in the filter applications and allows the operating system to assume the bulk of the responsibility for the system security. This allows application developers to focus on writing a good filter and reduces the chances that an application error can compromise the security of the system, resulting in higher security assurance of the CDS solution. The following five security goals all contribute to this overall goal:

1. Policy completeness – all security goals should be directly enforced by policy. This goal is not achievable in practice, but serves as the ideal to which to strive.
2. Information domain separation – no information can flow between the information domains except through the designated processing applications. Information domains are collections of network resources that all share the data sensitivity.
3. Non-bypassable processing pipeline – all information must flow through a non-bypassable processing pipeline. The information must flow through each of the steps, in the correct order.
4. System self-protection – the operating system, particularly the security mechanism, must be protected from tampering and bypass.
5. Least-privilege administration – all administrative components will be given the minimum of access and assigned only to appropriate roles.

3. CDS Architecture Implementation

In many ways, SELinux is an ideal platform for implementing this architecture. TE is well suited to creating these processing pipelines. Anyone familiar with the basic concepts of SELinux and TE can imagine how this could be encoded in policy. That being said, there are many details that make creating this architecture challenging. Most of these challenges center on the unique requirements for CDS, particularly the strict control over information flow and the mixing of data of differing sensitivities.

Existing SELinux policies and applications are more focused on least-privilege than controlled information flow.

The basic approach that we have employed to creating these systems is to create a customized version of Red Hat Enterprise Linux that includes a minimum set of applications and services, the solution specific applications, and a customized policy. In all instances we have started with an existing policy, either the NSA example strict policy [2] or the reference policy [3], and customized the policy to meet the system needs. This includes configuring and modifying existing policy modules and adding policy modules for the system specific applications.

3.1 Policy Completeness

Policy should directly enforce all security goals of the system in order to minimize trust placed in user space applications. Unfortunately, this is infeasible on CDS systems primarily because CDS applications must directly enforce much of the transfer policy. For example, in Figure 1, the virus filter is responsible for ensuring that malicious code does not reach Information Domain B. The SELinux policy can guarantee that all data traveling from Information Domain A to Information Domain B passes through the virus filter, but it cannot make the virus filter do its job. The virus filter must be trusted to enforce this security goal. However, with TE we can ensure that the virus filter is *only* trusted with its job of scanning for viruses, and not concerned with basic separation of high and low information domains. SELinux can handle that more fundamental security concern.

Additionally, compromises must be made in policy in order to get a system to function that create extraneous paths through the system. Figure 1 depicts a single path through the system from Information Domain A through postfix, each filter, postfix, and to Information Domain B. In a real-world system, other flows would have to exist for functionality such as initialization, shutdown, and management. An example of this would be a package manager such as rpm. Consequently, these applications must be trusted in addition to the policy in order to meet the security goals of the system.

3.2 Information Domain Separation

Separating information domains involves, in practice, strictly controlling network access by all applications except those at the beginning and end of the processing pipeline. Ideally, only a single application would have access to each network adaptor.

The current policies implement control over network access, but the control is coarse-grained and many applications are granted networking privileges. Typically, SELinux domain types¹ are given access to all nodes on all network interfaces (e.g., see the ‘can_network’ macro in the example strict policy). There is some control over binding to ports, but again, these domain types can typically bind to the allowed ports on all network interfaces. Had this problem of network granularity been depicted in Figure 1, the postfix process communicating with Information Domain A could just as easily have communicated with Information Domain B thus bypassing all filters. This problem is solvable, but it requires a careful audit of all domain types to remove unnecessary network access.

More problematic than a SELinux policy that provides better separation of network resources is the assumption by many applications that networking is always available. If the CDS solution uses off-the-shelf software it is not uncommon for that software to attempt more access than necessary (e.g., binding to ports on all interfaces) and fail if it is unsuccessful. This can obviously be solved if the source is available, but would still leave maintenance issues. In many cases, sources are not available.

¹ In this paper, an “information domain” is a collection of network resources, including a subset of “domain types” and local system resources, that handle information that is all of the same sensitivity. A “domain type” is the domain type concept implemented in a SELinux policy. Be careful not to confuse those two uses of the word “domain.”

The final challenge in information domain separation comes from applications that use networking for IPC. This is common for Java applications using RMI. These applications are difficult to keep separate since they use the network for IPC. One solution that has been effective for these applications has been creating multiple localhost nodes (for example, 127.0.0.1, 127.0.0.2, 127.0.0.3) with different labels, then setting up the applications to use those nodes. This allows the policy to enforce separation between information domains without having to change the underlying communication mechanisms in applications. Note that node-based separation is only useful on localhost IPC since IP addresses can be easily spoofed.

3.3 Non-bypassable Processing Pipelines

The creation of the processing pipelines is fairly simple in the basic case. TE domain types are created for each of the steps in the pipeline, some form of IPC is allowed between the TE domain types, and communication with other TE types other than the next TE type in the pipeline is eliminated or limited. The primary challenges arise when trying to enforce one-way communication via policy and limiting interaction with TE domain types not in the pipeline.

Linux provides many different IPC mechanisms, and some of these are better suited for one-way flow than others. Some are inherently bidirectional, such as stream sockets. Others can be used as one-way flow mechanisms, but may be difficult to use without opening a back channel. For instance, a directory full of files can be used between the virus filter and the spam filter. The virus filter can be granted access to create and write files to the directory, and then the spam filter can be granted read access to retrieve the files. This accomplishes the security goals, but does not provide a mechanism for deleting the files after the process is complete. The spam filter cannot delete the files, as this would be a backward channel. The virus filter has no way of knowing when the spam filter is done with the files. This problem can be addressed by allowing the backchannel based on its low bandwidth, having the virus filter guess when the spam filter is done and then delete the file, or using an infinitely large storage device.

This problem is exacerbated by the use of off-the-shelf software that was not designed to work in a one-way fashion. The CDS in Figure 1 uses postfix as a component in the pipeline. This application was designed to work over TCP networks using bidirectional communication. To pass messages to the virus filter, postfix would have to be rewritten to use a one-way mechanism internally. Alternatively, this step in the pipeline can be bidirectional, and one-way flow can be enforced in the other steps.

Utilizing existing policies for applications is problematic as well, as these policies are focused on least privilege rather than information flow. Consequently, all accesses within these policies need to be reviewed for information flow implications.

3.4 System Self-Protection

System self-protection is a security goal that is clearly aligned with existing policies, and therefore requires little or no policy modification. Obviously, more can always be done to protect the operating system, and particularly the security mechanisms. However, in the CDS systems that we have helped develop we have deemed the protections provided by the current policies as adequate. The one change consistently made is to reduce the system to the minimum set of packages possible.

3.5 Least-privilege Administration

The NSA example strict policy includes the sysadm_r role for administration. CDS systems require different administrative roles for different tasks. Dividing up sysadm_r responsibilities has proved difficult. It requires auditing the entire policy and deciding which function belongs in each role. Any time the role definitions are changed, the entire policy must be reworked again.

It is possible to define least privilege administrative roles in current policy. However, these definitions result in complex implementations that are inflexible and cumbersome to manage. Slight variances in deployment scenarios often require extensive modifications to the policy. The Reference Policy is working to solve this problem with a more flexible solution to role separation. This solution will enable administrative roles to be more easily defined and permit the capabilities given to a role to change to meet the deployment requirements.

4. Summary

Type Enforcement in SELinux provides a MAC mechanism excellent for building cross-domain solutions. We believe it is clear that TE is far more suitable for CDS system than traditional MAC systems that directly implement a static multilevel security (MLS) policy. While no MAC policy, including TE, can meet all the security goals of a CDS system, TE can directly implement most of the fundamental domain isolation security goals better than traditional MAC. TE provides an excellent mechanism for separating information domains and creating processing pipelines. SELinux also provides a means of separating administrative roles.

There are many practical issues involved in building a SELinux-based CDS system, but they are not insurmountable. In fact, many of the solutions to these problems are being used to generate tools to make the process easier, such as SEFramework[4].

Though there are many issues to work through in building a SELinux-based CDS system, it is a very attractive platform. Much of this is due to the appropriateness of TE for solving these problems compared with other MAC mechanisms. The first round of SELinux-based CDS systems are currently in development and will likely go into production use in the near future. There are many more on the horizon.

5. References

- [1] Boebert, W and Kain, R., "A Practical Alternative to Hierarchical Integrity Policies" Proceedings of the 8th National Computer Security Conference, 1985.
- [2] National Security Agency, Strict Policy, <http://www.nsa.gov/selinux>.
- [3] Tresys Technology, LLC, Reference Policy, <http://serefpolicy.sourceforge.net>.
- [4] Wilson, A., *SEFramework: A New Policy Development Framework and Tool to Support Security Engineering*, SELinux Symposium, 2005, Tresys Technology, LLC.