Experiences Implementing a Higher-Level Policy Language for SELinux

Chad Sellers, James Athey, Spencer Shimko, Art Wilson, Frank Mayer, Karl MacMillan
Tresys Technology

Abstract

Expressing security architectures that meet required security goals for a system in SELinux policy language is quite difficult, particularly for those without a strong understanding of the implications of SELinux security mechanisms and object class permissions. However, SELinux policy language is an excellent base upon which to build a higher-level policy language that more directly expresses specific security architectures. We have developed one such language, CDSFramework, for implementing security policies focused on information flow applications as typically found in cross-domain solutions. This paper presents the components of this language and describes some of the issues that we faced in implementing the language and its tools.

1. Introduction

A cross-domain solution (CDS) requires a high degree of confidence in its implementation and is ideally developed using a formal engineering process. This process requires the definition of a high-level security architecture for the CDS system that captures all the security goals, and assigns the implementation of these goals to identified portions of the architecture.

Implementers, typically software developers, use this security architecture as a specification to follow throughout the development process. We believe that one of the goals of any CDS security architecture is to ensure that as many of the security requirements are directly enforced by the underlying operating system (OS) using strong mandatory access controls (MAC). Crucial security goals for any CDS security architecture include the fundamental requirements of information domain isolation and controlled information flow. Such an architecture greatly improves the assurance of the overall system, by ensuring that the CDS guard applications themselves are limited in their trust to the small task for which they are designed, and the more primitive OS security mechanisms ensure separation and isolation. SELinux with TE is a great improvement over traditional multilevel security (MLS) MAC in achieving these goals.

In current practice, when using SELinux, a policy developer takes the security architecture and manually crafts policy to enforce the security goals that the architecture assigns to the OS. Likewise, the CDS application developer separately develops the guard applications. This manual, disconnected process is laborious and prone to errors. Each permission granted by the SELinux policy must be carefully evaluated to ensure that it meets the security goals and implements the security architecture. The guard applications may make assumptions about allowed access that the TE policy disallows.

Our goal with the CDSFramework project [2] is to create a means for a CDS system designer to specify a security architecture from which the SELinux TE policy is *directly derived* and that the CDS application developers cannot subvert in their application implementations. We would like to collapse security architecture specification and TE policy development into a single step, and thereby remove a large area of possible error. Ultimately, it should be possible to develop a CDS system with higher assurance that is more easily accredited at a lower cost.

In this paper, we outline the concepts, language, and implementation of the CDSFramework, and discuss some of the implementation challenges we had to overcome.

2. CDSFramework concepts

The CDSFramework is designed to allow the system designer to express security goals and properties in a manner that focuses on the security concerns without worrying about low-level details, while enabling the application developer to implement those details within the constraints set by the designer.

CDS systems are primarily concerned with information flow and information domain separation security goals, in addition to the containment and least privilege security goals of any TE policy. Consequently, the current

CDSFramework language primarily focuses on meeting the information flow and domain separation goals, though we plan to generalize the language in the future. With this end in mind, we created a framework that consists of four concepts: domains, shared resources, access, and decomposition.

2.1. Domains

CDSFramework *domains* are security boundaries that can contain a number of system objects, including active entities such as processes, passive objects such as files and sockets, and transition objects such as file entrypoints. Domains therefore represent the "information domains" from CDS systems. A key characteristic of an information domain is that within a domain, all resources have the same sensitivity, and all processes have the same level of trust. This definition of domain is broader than the SELinux concept of domain types (which are simply types that may be used for processes).

A basic rule for our domain concept is that within the security boundary of the domain, most accesses are granted by default for contained processes to all contained objects. From a security perspective, a domain, and all the processes, files, and other objects it contains, all have the same security attributes, making them security equivalent. On the other hand, access between domains is only allowed via shared resources, described below. CDSFramework domains allow us to achieve data separation by placing information domains and all associated objects (such as network interfaces, files, and directories) in a single domain. Likewise we can separate the stages in the CDS application pipeline into their own domains. When graphically depicting domains, we use boxes, as show in Figure 1.

2.2. Shared resources

A *shared resource* is a purely passive entity used for communications and information sharing between domains. Since all objects within a CDSFramework domain are private to that domain and not shared, the only way for domains to interact is via a shared resource. Domains must be given explicit access (in a well-defined form) to shared resources.

While shared resources may seem similar to SELinux object classes, they are in fact a higher-level concept. A single shared resource may contain any number of system object classes (files, pipes, sockets, etc.) that together represent a single conceptual resource. For example, the definition of a shared resource called *unix-StreamSockets* would include permissions on sock_files, unix_stream_sockets, and a directory to put the sock_files in.

Shared resources are represented as circles in the graphical depictions, as show in Figure 1.

2.3. Accesses

Accesses define the interaction between domains and shared resources. Thus, indirectly, accesses define the information flow between two domains. CDSFramework currently supports three forms of access in support of information flow applications: read, write, and readwrite. A read access transfers information from a shared resource to a domain, while a write transfers information from a domain to a shared resource. A readwrite access represents an information flow that is inherently and indivisibly two-way.

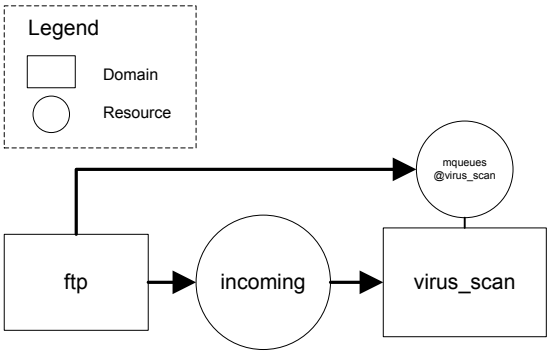

Figure 1 - A graphical representation of a CDSFramework policy.

SELinux provides significantly more granular forms of access permission than read, write, and readwrite, although most SELinux permissions imply information flow in one or both directions. Thus, it is necessary for CDSFramework to support the ability to map between its simple access categories and the more complex SELinux object class permissions. For example, both random access writing and appending to a file can both be characterized as a write in terms of information flow.

As a further example, for the resource definition *unix-StreamSockets* described above, write access may be defined as the ability to create sock_files in the direc-

tory, set up a unix stream socket, and delete those sock_files, but not delete the directory.

Accesses are represented as arrows in the graphical language, as show in Figure 1. Write access arrows point from a domain to a resource, and read access arrows point from a resource to a domain, indicating the direction of information flow.

2.4. Domain decomposition

Our objective is to allow the designer to create a security architecture at a level of detail appropriate for the system, focusing his energy on where security properties are enforced, and not on how to implement the security architecture. While the domain, shared resources, and access concepts provide this ability to focus, these concepts by themselves do not provide sufficient ability to construct complex architectures. Therefore, we include the concept of *domain decomposition*, which allows a domain to be decomposed into sub-domains, shared resources, and accesses between them.

For example, we might initially define a CDS security

```
rdef unixStreamSockets
[desc: "Unix stream sockets, sockfiles, and the directory they live in"]
{
          requires { dir }
          owner {
                    resource {
                              dir { add_name read remove_name search write }
                              sock_file { create getattr unlink write }
                    }
                    self {
                              unix_stream_socket { accept bind connect connectto
                                                   create write listen read shutdown }
                    }
          }
          read {
                    default { read }
                    read
                    [desc: "Read data from Unix stream sockets"]
                    {
                              resource {
                                        dir { search }
                                        sock_file { getattr }
                              }
                              self {
                                        unix_stream_socket { connect create read }
                              }
                              other write {
                                        unix_stream_socket { connectto }
                              }
                    }
          }
          write {
                    default { create write }
                    write
                    [desc: "Write data to Unix stream sockets"]
                    {
                              resource {
                                        dir { search }
                                        sock_file { write }
                              }
                              self {
                                        unix_stream_socket { accept bind create listen
                                                             shutdown write }
                              }
                    }
                    create
                    [desc: "Create and delete socket files"]
                    {
                              resource {
                                        dir { add_name remove_name write search }
                                        sock_file { create unlink write }
                              }
                    }
          }
}
```

Figure 2: CDSFramework resource definition in the dictionary

architecture with three domains: the low information domain, the guard application domain, and the high information domain. As the design evolves, we would likely add more resolution to this architecture. Using decomposition, we can populate the guard domain with several sub-domains, representing individual guard stages or applications, each of which has a subset of the security permissions of the parent domain. In this way we can provide the ability to increase the assurance of the security architecture by allowing stepwise introduction of sub-domains, while keeping the focus on the security goals and least privilege.

As described earlier, all processes in a domain have essentially unlimited access to objects in that domain. With decomposition, we can refine access within a domain by dividing the parent domain into sub-domains and shared resources in order to explicitly define what accesses are allowed within the domain. As always, sub-domains obey the same rules as domains and may only interact via shared resources.

An additional rule that results from decomposition is that domains cannot access shared resources inside another domain. A resource inside a domain is implicitly not shared with other domains.

A decomposed domain must contain at least one sub-domain. In other words, no domain can contain solely shared resources. This prohibition exists because a parent domain itself is not an active entity. Instead, it is merely a container that, taken alone, represents an active entity with private resources.

Access from children of a decomposed domain to an external shared resource is constrained by the access of the parent to that resource. Similarly, child domains and resources can only associate with resource definitions also associated with the parent.

Decomposition may be successively applied to sub-domains as needed. With decomposition, we allow a developer to produce a detailed and precise security policy, while maintaining the security goals of the architecture and preserving the high-level simplicity of the original design.

3. Implementation experiences

We have an initial implementation of the CDSFramework, which so far has proven successful. Our implementation includes a CDSFramework language that represents the concepts, a dictionary that describes the meaning of resources and accesses in SELinux policy language, a compiler that generates SELinux policy directly from the framework language, and an IDE that helps the developer use the CDSFramework. We designed CDSFramework to be expressible in both a textual and a graphical language. Examples of the graphical and textual policy languages are in Figure 1 and Figure 3, respectively.

Designers can use these tools to generate CDS security architectures and directly derive the associated TE policies for SELinux. We envision their use for many other solutions that require information flow security goals as well. In implementing the CDSFramework, we addressed several problems that will be common to any higher-level policy language development. Those issues are detailed in the following sections.

3.1. Dictionary

In order to write CDSFramework policy, we needed a way to specify the different kinds of CDSFramework resources and the accesses associated with those resources. This specification also needed to support the ability to directly derive SELinux TE policy from CDSFramework language. Thus we required a way to map CDSFramework resources and accesses to SELinux object classes and sets of permissions. Additionally, we need this mapping to be customizable, so that resources can be added or modified. The CDSFramework dictionary addresses these needs.

The dictionary uses a language syntax that allows us to create CDSFramework *resource definitions* as groupings of SELinux object classes and permissions. These resource definitions can include any combination of SELinux object classes. For example, a single resource could group the SELinux *netif*, *tcp_socket*, *node*, and *port* object classes to intuitively represent a "network resource." Although primarily used for shared resources, we have other types of resource definitions in the dictionary, including an entrypoint definition that defines the possible information flows for domain-to-domain transition. In addition, a resource definition may be associated with a domain to provide access to system resources internal to that domain.

Using these resource definitions, the CDSFramework presents users with a conceptually abstract resource that represents system objects in a form meaningful for their application.

In Figure 2, the *unixStreamSockets* resource definition provides an example of the dictionary syntax. This definition includes several blocks for specifying differ-

ent components of the shared resource. In general, resource definitions include several types of statement blocks:

- "owner" - This group of permissions is given to a domain when the resource definition is associated with it. Recall that domains have effectively unlimited access to private domain resources, and the permissions contained in this block should facilitate that.

- "read", "write", and "readwrite" - These blocks group access definitions by verb. An access definition found under "read" can only be used in "read" accesses in the CDSFramework policy language.

- "default" – Lists the access definitions used for an access when that access only specifies the category, i.e. "read", and not also specific access definitions within "read".

- Access definitions – within "read", "write", or "readwrite", groupings of permissions that represent individual conceptual operations on a resource. In this example, "write" allows a domain to use a shared resource with this resource definition to write data through the Unix stream, while "create" allows a domain to create the stream.

- Access targets – within access definitions, indicate what the target of the permissions is, either "resource", the shared resource itself, "self", the domain itself, or "other" and a verb, which would give permissions on all other domains that access this resource via that verb.

- System-resource association requirements – "requires" - This type of block is used to support the association of real system objects with the CDSFramework abstractions. The association is then used to generate labeling policy. There are two types of system-resource association blocks: "requires" and "optional". Each block specifies kinds of object instances that must or may be associated with the shared resource. In the *unixStreamSockets* resource definition, a "dir" object class is listed as required, which means that when a shared resource associates this resource definition, a specific directory path must be provided so that labeling can be performed correctly.

In Figure 3, we define a simple CDSFramework policy that defines a domain "ftp" and a shared resource "incoming" as well as write access between them. In this figure, we show how our tools would use the dictionary to translate the CDSFramework language into SELinux policy statements. The domain, shared resource, and access declarations are used to create the associated "allow" rules and type declarations.

3.2. Linking with base policies

The dictionary provides a means for mapping abstract resources and accesses to SELinux object classes and permissions and allow rules. However, the dictionary abstraction does not account for making these derived policy statement working with a complete SELinux policy. For a system to function, the CDSFramework-derived policy statements must be integrated into an underlying *base policy* for the operating system. The base policy will define the core security properties for the OS upon which our CDS application will run. For our CDSFramework policy to be effective, it must integrate with and build upon the implementation details of this base policy.

Our goal is to allow for CDSFramework generated policy statements to integrate with any reasonable base policy through a linking layer. Linking with a base policy may include adding attributes to certain CDSFramework defined types or calling a base policy macro to access base types. The implementation of linkage will differ from one base policy to another.

We investigated a number of mechanisms to link to a base policy. We concluded that there needed to be a set of well-defined interfaces that would be custom-fit depending on the base policy, but present the same set of capabilities to the CDSFramework compiler. We currently link to our sister Reference Policy[1] policy project, but the linkage layer is designed to be customizable to any form of base policy.

In the example shown above in Figure 3, in translating the "domain ftp" declaration, our compiler calls the linkage interface that inserts a standard *linkage macro*. These linkage macros, when processed by M4, in turn insert the proper macros and policy statements for the underlying base policy. In general, our linkage macros ensure that certain characteristics are associated with the CDSFramework defined types.

3.3. Label policy issues

For the CDSFramework-generated policies to be useful, we need to associate real system object instances with our abstract CDSFramework domains and shared resources. In SELinux, this association is primarily accomplished through object labeling. In order to do system-to-resource association we needed a way to generate security contexts for install time (e.g., application files and directories installed when the application is installed), as well as a method for ensuring that any resource created at run-time (e.g., files in a directory whose label are derived via a transition) are labeled properly.

CDSFramework language:

```
domain ftp;
resource incoming { unixStreamSockets };
access ftp incoming write;
```

SELinux policy using Reference Policy:

```
CDSFramework_domain(ftp)
CDSFramework_resource(incoming)
CDSFramework_files_type(incoming)
allow ftp incoming:dir { add_name remove_name search write };
allow ftp incoming:sock_file { create unlink write };
allow ftp self:unix_stream_socket { accept bind create listen shutdown write };
```

Figure 3: Mapping example

In SELinux. many object classes inherit security contexts from the creating process if no explicit labeling policy exists, addressing part of the run-time issue. SELinux also allows labels to be transitioned from other objects using type_transition rules. Finally, objects can be statically labeled using file_contexts, nodecon, and so on. With these mechanisms in mind, we examined several approaches to manage labeling and the association of actual system objects with our abstract resources.

In our first approach we marked objects associated with resources as either run-time or install-time, and then used the appropriate labeling method. Unfortunately, this approach was cumbersome and not always effective. It also required us to expose too many SELinux details to our user when our goal was the opposite.

In our second approach we treated all associated objects as both install-time and run-time. For file-related objects, for example, we generated both a file_contexts entry and a type_transition rule. While this approach is generally workable, it tends to produce policies with extraneous rules and can result in unintended information flows.

We eventually chose to use the simplifying assumption that only "container objects" are explicitly labeled through association. Examples of container objects include directories, which contain filesystem objects, and network interfaces, which contains network traffic. By explicitly labeling container objects and implicitly labeling the objects they contain, we solve most of the labeling challenge using inheritance. We found this approach simplified our implementation without adding any significant limitations to the CDSFramework concepts. As an additional benefit, this approach prevented many inadvertent information flows. For example, when two domains can access different files with different contexts in the same directory, the domains can exchange information using only the filenames.

Another problem with system-to-resource association involved the directory "search permission." In order to access a directory, a program must have permission on that directory's parents, all the way up to the root directory. Since those types are external to CDSFramework, this involves special linkage with the underlying base policy. Consequently, we integrated the CDSFramework compiler with libselinux to query the appropriate security context within the base policy and add the necessary rules where appropriate.

3.4. Process type labeled objects

Some of the SELinux object classes are not labeled via filesystem inheritance, explicit labeling policy, or type_transition rules, but are assigned the same label as the creating process type. Examples of this phenomenon include sockets, IPC mechanisms such as message

queues, and signals. We cannot change the types given to these objects in the current implementation of SELinux. In CDSFramework, these object classes cannot be placed in normal resource definitions and used in shared resources, because instances of these classes share the creating domain's type and cannot be separately labeled. These object classes cannot become part of the domain either, because accesses that use these classes would connect domains directly to domains. Our goal is to model inter-domain communication strictly via shared resources.

Conceptually, these object classes are passive facilitators of communication between domains, just like shared resources. Therefore, to handle these objects we created a special kind of shared resource called a *control resource*. A given control resource is always linked to a specific CDSFramework domain and is labeled with the type of the domain. A domain can only have one control resource associated with each control resource definition, as all message queues created by that domain will share the same type, that of the domain.

For some of these objects, such as IPC, control resources are not the optimal solution. Rather, we would like to eventually modify the kernel to support labeling these objects uniquely at runtime. Nevertheless, a control resource concept will always be required for objects such as signals, because signals do not exist as objects the way the other forms of IPC do. Instead, signals are themselves the communication.

To help differentiate control resources from ordinary shared resources, we construct their names by concatenating the resource definition name, an '@', and the domain's name. A graphical example of a control resource can be seen in Figure 1. The resource *mqueues@virus_scan* is a control resource attached to *virus_scan*, and represents message queues created by processes in the *virus_scan* domain.

4. Conclusions and future work

The SELinux policy language provides a good base for higher-level policy languages. The CDSFramework provides such a language, and is suitable for expressing security policies for certain targets, in particular cross-domain solutions. We are planning to continue development on the CDS problem domain, as well as using the lessons we are learning from building actual CDS systems to improve the concepts. The CDSFramework is under development, but versions are being released for testing at (Ed. to be determined; will be released before paper publication).

References

[1] Tresys Technology, *Reference Policy*, http://serefpolicy.sourceforge.net.

[2] Wilson, A., *SEFramework: A New Policy Development Framework and Tool to Support Security Engineering*, SELinux Symposium, 2005, Tresys Technology