SELinux and MLS: Putting the Pieces Together

Chad Hanson
Trusted Computer Solutions, Inc.
chanson@TrustedCS.com

Abstract

Multi-Level Security (MLS) has been implemented on many different operating systems. We will discuss the reasons and motivations behind the improvements to the MLS model in SELinux that were accepted into the 2.6.12 Linux Kernel. An introduction to SELinux MLS representation, policy creation, and integration is provided to help further the adoption and use of this technology.

1. Introduction

Multi-Level Security (MLS) policy was first formalized by Bell and LaPadula (BLP) [1] in the 70's. Systems implemented with MLS policies were primarily used to enforce confidentiality. In the 80's and 90's, the Compartmented Mode Workstation (CMW) used MLS as the primary Mandatory Access Control (MAC) [2] mechanism for evaluation to the Orange Book [3]. Today, MLS is still a key requirement to meet the Common Criteria [4] Label Security Protection Profile (LSPP) [5] and future Medium Robustness Multi-Level Operating System Profiles for Common Criteria.

BLP defines two MLS properties: simple and star. The simple security policy requires that a subject must dominate the object to have read access. The star property requires that a subject can write to an object only if the object dominates the subject. These properties are the basis for the MLS policies in SELinux. In our implementation, we have restricted the star policy to require a strict equality. We want to note that the DCID 6/3 PL4 [6] confidentiality requirements augment the write access to be confined by the subject clearance, so a write equality policy is a further refinement of this requirement.

SELinux is based on the Flask security architecture [7,8]. The Flask architecture is designed with flexibility to support multiple security policies. The SELinux security server supports security policies for Type Enforcement (TE) [9, 10], MLS, and Role Based Access Control (RBAC). Although SELinux supported an example MLS component, it was experimental and not suitable for commercial needs. This paper describes how SELinux was modified within the definition of the Flask architecture to improve MLS support to make it more responsive to real world MLS requirements and compatible with other MLS systems.

2. MLS Security Model

In 2003, Trusted Computer Solutions (TCS) began to look at the MLS security model in the SELinux framework. At this point, MLS support was experimental and required many steps for proper operation. The MLS model would have to be modified to work transparently in order to be accepted in the Linux kernel and mainstream distributions.

2.1. Motivation

The TE security model in SELinux is very powerful for meeting many security requirements. TE is very flexible and configurable, allowing fine-grained MAC enforced by a predefined security policy. TE provides a strong model for controlled access and execution paths of applications. It also provides a mechanism for system integrity and controlling non-hierarchical relationships.

However, it is not ideal for MLS needs. MLS, as noted above, excels in providing confidentiality through straightforward rules. A combination of MLS and TE creates a stronger, more functional system that benefits from the strengths of the two complementary models.

The premise for the strength of the pairing of the two models is by looking at their individual weaknesses. Existing MLS models do not lend themselves easily to static analysis. Also, many security goals, such as privilege models, cannot be mapped into MLS concepts. TE has the rigidity and complexity of a predefined policy matrix. Also, TE has deficiencies in handling a large number of labels or a dynamic work set of label names, especially in contrast to integrity concerns. Given these weaknesses, a pairing of MLS and TE security policies provide a much stronger platform than the existing MLS systems which are deployed currently.

2.2. Policy Enhancements

The overhaul of the MLS policy was the largest task. A flexible mechanism that could allow the ability to grant policy overrides on a very granular level, ideally within the existing SELinux framework, was desired.

The existing MLS policy mapped permissions of a security class to a set of MLS base permissions. The base permissions consisted of *none*, *read*, and *write*. After a few internal prototypes, TCS decided to remove the existing permission schema and utilize the existing SELinux constraint language to model the MLS security policy. This seemed by far the most elegant approach as the high level language allows the granular ability to define constraints based on class-permission pairing. It also gives the desired ability to allow policy overrides, such as trusted objects, in the language instead of hard-coded in the security server. The ability to define the model in the high level language also allows for research and experimentation of different MLS policies without needing to alter the underlying security server.

In utilizing the constraint language, we had to make a few extensions to support MLS, but most of the core functionality was already in place. The language originally supported user, role, and type constraint expressions. TCS expanded the language to support the *mlsconstrain* token, along with the ability to use the low and high sensitivity labels of the MLS Range. Also, there is a special case in MLS that requires a third target when transitioning labels. For this, a second token was added, *validatetrans*, which requires the third target of user, role, or type. An example of the *mlsconstrain* usage for file read access is shown below.

```
# the file "read" ops (note the check
# is dominance of the low level)
mlsconstrain { dir file lnk_file
chr_file blk_file sock_file fifo_file }
{ read getattr execute }
    (( l1 dom l2 ) or
     (( t1 == mlsfilereadtoclr )
           and ( h1 dom l2 )) or
     ( t1 == mlsfileread ) or
     ( t2 == mlstrustedobject ));
```

2.3. Dynamic Loading

The most important task in the process of getting MLS was changing the original model of a separate kernel compile option and separately compiled policy commands. With these existing limitations, the chances for MLS being accepted by the community were slim.

This required the development of a method to load either a normal policy (TE and RBAC) or an MLS policy (TE, RBAC, and MLS) into the kernel. This was accomplished by checking for an MLS tag during the policy load phase and storing the policy state. Once a policy type has been loaded, either normal or MLS, it cannot be changed until a system reboot.

Lastly, we needed to update the user-space tools, *checkpolicy* and *libsepol*, to have options to support both normal and MLS policy. In the checkpolicy command, adding a command line option to determine whether MLS language was needed in the policy solved the problem.

3. Security Context and MLS

The main visible component of SELinux is the security context. The security context is utilized by SELinux to work within the Flask architecture. The context is made up of four fields: SELinux User, Role, Type, and MLS Range. The last portion, MLS Range, is an optional component and is only available when a SELinux policy containing MLS is loaded in the kernel. TCS made a few minor improvements to the existing MLS representation to fit real world needs.

3.1. MLS Representation

The MLS Range contains two components, the low and high (clearance) sensitivity label, in which the high must always dominate the low. Each sensitivity label is comprised of a hierarchical classification and a set of non-hierarchical compartments. The MLS policy in Fedora contains 16 classifications and 256 compartments. These settings are configurable and can be changed in the *mls* file in the SELinux policy source.

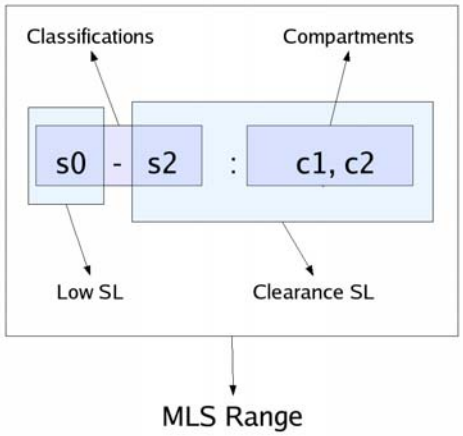

Figure 1. Components of the MLS Range

Since the number of compartments is dynamic and could grow quite large, a compact notation was introduced to help limit the size of the SELinux context. The compact notation allows the collapse of adjacent compartments by denoting the first and last compartment.

Figure 2. SL Compact Notation

The need for this feature became readily apparent when testing out labels with a large number of compartments. Minimizing the size of the security context is important in processing and storage. First, the notation helps with speed and efficiency when reading and writing this value to the Linux kernel. The security context is also stored in ASCII format within audit records. An MLS label with a large number of compartments would dramatically increase the audit record size and hinder processing.

On an MLS system are two special labels, *SystemLow (s0)* and *SystemHigh (s15:c0.c255)*. *SystemLow* is the lowest classification and contains no compartments, thus dominated by every label on the system. *SystemHigh* is the highest classification and has all compartments, thus dominates every label on the system and also benefits greatly from the new compact notation.

3.2. MLS Translation

Within the SELinux framework, we have introduced a translation mechanism to give a more literal meaning to the machine-like policy used in the MLS *sensitivity* and *category* declarations. This is needed for special labels, such as *SystemLow* and *SystemHigh*, along with the ability support different industries and more complicated government relationships. Also, this is useful to allow third parties to create specialized translation engines.

The process of MLS range translation occurs transparently in *libselinux* through the dynamic loading of the *libsetrans* when a policy containing MLS is loaded in the kernel. The translation library takes the native MLS Range and translates the sensitivity label components into a more Human Readable form. The reverse operation, taking a Human Readable form and converting back to a native MLS Range, is supported when submitting requests to *libselinux* interfaces.

Figure 3. SL Translation

Translations were first introduced in Fedora Core 5. The current users of the translation interface are the MLS policy and the Multiple Category Security (MCS) policy [11] that uses the MLS Range portion of the security context. TCS supports the MITRE Label Encoding Format through this mechanism. The MITRE Label Encodings Format is utilized by the Defense Intelligence Agency to handle the representation of complex relationships.

4. SELinux MLS Policy Creation

After discussing the motivation for MLS and how it is represented in SELinux, the next step is describing the methodology used to create and configure the SELinux MLS policy. The process of creating MLS system policy is not a trivial task.

The first task in policy creation is to define the valid labels to be used on the system. With the Fedora Policy, there are a few example classifications and compartments. Using the translation library, there shouldn't be any need to change the sample policy; refining the translation definition should be enough.

The second task is defining the labels from the objects, subjects, initial security identifiers (SIDs), and generic file system (genfs) contexts on the system. The initial MLS Policy for Fedora Core 5 will be used for this in the sections below.

4.1. Objects

Every object on the system must have an MLS label, either explicit or implicit. The implicit labels generally come from initial SIDS or genfs contexts. In the Flask architecture, these attributes are used for objects that do not have explicit labels. Most files on a system have explicit labels and, except for user and security relevant data, will be labeled *SystemLow*. This is done for usability and secondarily an integrity mechanism, since most processes dominate *SystemLow*. Examples of files

in this category are binaries, libraries, etc., required by services and users alike.

Certain categories of files such as devices, audit logs, and security configurations should be analyzed for the correct label. Certain device files, such as hard drives and kernel memory (*/dev/kmem*), are labeled *SystemHigh*. This is required since direct use of these raw devices does not enforce granular MLS access to the raw data. Normally, the disk-based file systems mediate MLS access to the raw data through system calls such as *open()*. The MLS write policy of equality between subject and object doesn't work well for special devices, such as the null device (*/dev/null*), which discards all input provided to the device. For these files, there is a trusted object attribute that allows for MLS policy overrides on most write permissions and should be used for devices where no data is stored.

The basic guideline or test for verifying an object labeling decision is to determine whether confidentiality can be breached via access of the object. In the section above, audit logs are mentioned. The audit log contains records from processes running at numerous labels from *SystemLow* to *SystemHigh*. With the possibility of *SystemHigh* data, the audit logs must be labeled *SystemHigh*. The shadow file contains the users' encrypted passwords. These passwords don't have any inherent security classification, so they can be kept at *SystemLow*. This methodology needs to be applied to the entire system, which is a bit of a daunting task.

In SELinux policy, the MLS labeling of objects occurs in the file contexts along with initial SIDS and genfs contexts. The file contexts database, which contains libaries, devices, etc., is used for initial creation of explicit labels and relabeling operations based on labeling methodology described above.

4.2. Subjects

Every subject on the system has an MLS Range. The labels in the range are inherited from the parent upon a *fork* system call. MLS labels will stay unchanged upon *exec* of a new image, except in cases of policy rules (*range_transition*) or process attribute (s*etexeccon*), and are the preferred method label changing. An example of the *range_transition* rule takes place during system initialization, the *init* process transitions from the kernel SID, which has an MLS Range of *SystemHigh*, to the defined MLS Range of *SystemLow-SystemHigh* when executing the *init* image. The process attribute change, the *setexeccon* interface, is available in *libselinux* and changes the value of */proc/<pid>/attr/exec*. Only a privileged process, a domain with the specified MLS attribute, has the ability to change the current process label within the subject clearance. This can be performed via the *setcon* interface in *libselinux* or directly changing the value of */proc/<pid>/current*.

5. Application Support

The last major step in getting an MLS system is application support. This support must be achieved through policy and code enhancements. Policy additions are the largest change for the system initialization process. A number of the system services, such as *init*, need MLS privileges or label transitions upon execution to perform their tasks. Code modification and creation is needed in areas such as *PAM*, *cron*, and other utilities to create a usable a MLS subsystem. A number of these issues are addressed in a separate paper [12].

6. Conclusions

Implementing MLS within the SELinux Flask Architecture in a transparent manner was not a trivial task. As described in this paper, great progress has been made on implementing MLS and creating a sample policy which now exists on a major Linux distribution. This functionality allows for the ability to meet the LSPP, CAPP and RBAC Protection Profiles and an evaluation is currently in progress.

However, there is still much ongoing work on the MLS front. Currently, the MLS policy applies to a server-only system. More applications and a user desktop should be added to create a secure workstation.

By utilizing the SELinux MLS security model, a solid foundation now exists in Linux for creating and transitioning existing MLS solutions to meet the needs of the security community.

7. References

[1] D.E. Bell and L.J. LaPadula, *Secure Computer Systems: Mathematical Foundations and Model*, Technical Report M74-244, The MITRE Corporation, Bedford, MA, 1973.

[2] P. A. Loscoco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner and J. F. Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," In *Proceedings of the 21st National Information Systems Security Conference*, pp. 303-314, Crystal City, VA, October 1998.

[3] DoD, Trusted Computer System Evaluation Criteria, Department of Defense Standard 5200.28-STD, December 1985, Accessed at http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html.

[4] Common Criteria Project, Common Criteria Version 2.3, Accessed at http://www.commoncriteriaportal.org/public/expert/index.php?menu=2.

[5] National Security Agency, Information Systems Security Organization, "Labeled Security Protection Profile," v1.b, October 8, 1999, Accessed at http://niap.nist.gov/cc-scheme/pp/PP_LSPP_V1.b.html.

[6] DCID 6/3, *Protecting Sensitive Compartmented Information Within Information Systems*, June 5, 1999, Accessed at http://www.fas.org/irp/offdocs/dcid.htm.

[7] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," In *Proceedings of the 8th USENIX Security Symposium,* pp. 123-139, Washington, DC, August 1999.

[8] National Security Agency, *SELinux*, Accessed at http://www.nsa.gov/selinux/.

[9] W. Boebert and R. Kain, "A Practical Alternative to Hierarchical Integrity Policies," In *Proceedings of the Eighth National Computer Security Conference*, 1985.

[10] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "Practical Domain and Type Enforcement for UNIX," In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pp. 66-77, May 1995.

[11] J. Morris, *A Brief Introduction to Multi–Category Security (MCS*), LiveJournal.com, September 16, 2005, Accessed at http://www.livejournal.com/users/james_morris/5583.html.

[12] J. Desai, G. Wilson, and C. Sellers, "Extending SELinux to meet data import/export requirements," SELinux Symposium, March 2006.