

Porting Legacy Multilevel Secure Applications to Security Enhanced Linux

Andy Suchoski
Rick Supplee

Hewlett Packard Company

andrew.suchoski@hp.com rick.supplee@hp.com

Abstract

This paper reflects a work in progress and will discuss issues in migrating applications from legacy Multilevel Secure (MLS) systems to Security Enhanced Linux (SELinux). Initially, architectural similarities and differences will be discussed. This will provide a basis for discussing the actual migration of code from Trusted Solaris to SELinux. Although the examples in the paper are simple, they illustrate basic principles that will be used in porting code and performing policy work in moving applications from Trusted Solaris to SELinux. The areas covered are not exhaustive but do discuss how the main security features of sensitivity labels, process privileges, roles, and authorizations in Trusted Solaris map to SELinux. Additionally, areas where functionality does not currently exist in SELinux will be noted. Finally, conclusions will be offered and direction for further work.

1 Introduction

The Department of Defense Trusted Computer System Evaluation Criteria (DOD 5200.28 STD), commonly called the Orange Book, provided a standard against which to evaluate security. Last revised in 1985, this publication put emphasis on information confidentiality. It specified both Discretionary Access Controls (DAC) in Division C and Mandatory Access Controls (MAC) in Division B of the specifications. Most operating system vendors were able to incorporate Division C requirements into their commercial operating systems but incorporating MAC resulted in incompatibilities that were too extensive for the standard operating system release. Furthermore, as implemented the mandatory access controls were based on sensitivity labels which were used within the Department of Defense to control the dissemination of information. There was also a need for MAC in commercial environments but for information integrity as well as confidentiality. As a result, products which satisfied the MAC requirements of the Orange Book were special releases of their commercial counterparts and often lagged several releases behind the commercial release. Finally, most operating system vendors stopped updates over five years ago and as a result hardware which runs this software is only available through non-standard sources like eBay.

Linux offers DAC in the basic kernel and MAC is implemented through the optional Security Enhanced

Linux (SELinux) Linux Security Module (LSM). Unlike legacy MLS systems, there is no special release of the operating system. SELinux offers flexibility in the MAC security policy so that an appropriate level of protection can be chosen. The ability to choose a security policy is an important difference between the legacy MLS systems and SELinux. With the inclusion of sensitivity label enforcement policies in Red Hat Enterprise Linux 5 and Common Criteria evaluation efforts, SELinux can satisfy many of the security requirements of legacy MLS environments. With the obsolescence of hardware capable of running legacy MLS systems, it is time to seriously consider migration to a newer environment.

2 Security Feature Architectural Similarities and Differences

The legacy MLS system vendors took the approach of modifying the kernel security policy to enforce the MAC requirements of the Orange Book. Furthermore, privileges *replaced* the root privilege, thus a process that ran as EUID=0 now could only accomplish its task by having the appropriate privileges. The root user ID is simply another user UID on the system with no privilege. Authorizations and roles are implemented by privileged processes checking the user's credentials and then using their privileges to effect administrative functions.

SELinux takes a different approach because the default Linux DAC policy is left in place and MAC security is

added when the LSM is compiled into the kernel. This means that operations requiring EUID=0 still require this but the required privilege must be granted in the MAC security policy in SELinux. Roles are implemented through policy configuration rather than individual programs. Note that Linux API calls setting capabilities can be used in place of EUID=0 but will not be discussed in this paper.

2.1 Privileges

Legacy MLS systems use process privileges to allow trusted programs to perform tasks that are not permitted to normal processes. Privileges effectively separate the monolithic EUID=0 global override into discrete privileges which provide selective overrides in a specific area, much like Linux capabilities, but more fine grained. There are several process privilege sets so that a process could activate a privilege when needed and drop the privilege when not needed. This is meant to limit damage that could result from a malicious subversion or unintentional program error. However, when activated, the scope of the privilege is system wide. There are no bounds on what a process could do with a privilege once it is raised. Activating privilege requires Application Programming Interface (API) calls specific to the MLS system to be inserted into the code. Privilege could be *forced* on a process but that is discouraged because of the system-wide scope of privileges.

In SELinux, a process has a security context consisting of four security elements; type, user, role, and range. The type of the process defines the domain in which it executes. Type enforcement policy statements give the domain access to various object types. Additionally, policy statements can assign capabilities and type attributes to the domain which provides the equivalent of legacy MLS privileges with two important differences. The first difference is that the privilege resulting from type attributes and capabilities is limited to access defined in type enforcement statements. Secondly, there is no API so programs do not need to be modified.

The different ways in which a process has authority to perform actions not permitted to unprivileged processes illustrates two important differences between SELinux and legacy MLS systems. First, to affect privilege in a trusted manner, a proprietary API is needed in legacy MLS systems which introduces binary incompatibilities

between programs written for the commercial version of the operating system and the special release trusted version. Second, because of the nature of how privileges are activated, trusted programs are in effect implementing their own security policies. In SELinux, the capabilities and type attributes which enable a process to perform trusted actions are controlled by the policy itself; not the process. There are no code changes required. In effect, legacy MLS programs make policy while SELinux programs follow policy.

2.2 Sensitivity Labels

Probably the most distinguishing security feature of the legacy MLS systems is the presence of sensitivity labels to control access to information. The MAC policy followed was the famous Bell-LaPadula policy which specified “No write down; No read up”. Sensitivity labels are manipulated by trusted programs and there are a large number of library calls to support a variety of operations on labels in the various sensitivity label formats; text, binary, and hexadecimal. Most operations are performed on the binary format.

Instead of a single label as the MAC attribute, SELinux employs a security context consisting of four elements, three of which are used for type enforcement and the fourth attribute, range, is used for label security. The range element combines the legacy MLS sensitivity label of the process with the process clearance. The Bell-LaPadula rules are embodied in the policy MLS constraint file. In user space, there is only a single format of the range and that is as a string. There are presently no sensitivity label-specific library calls to deal solely with the sensitivity label. SELinux treats the range as simply one of the security context elements with no support for application decisions based on sensitivity label comparisons alone.

On legacy MLS systems, in addition to the Bell-LaPadula access rules, other factors affect the validity of sensitivity labels. Many of the legacy MLS systems use the MITRE encodings format to exercise additional controls on how labels are constructed. Using the MITRE encodings, minimum and maximum sensitivity levels could be required for certain categories. Additional rules can specify that certain categories can not appear in the same label with other categories (mutually exclusive) or certain categories must appear together in a sensitivity label. In the SELinux MLS constraint file, categories are enumerated over

sensitivity levels enabling some of the encodings grammar to be implemented.

2.3 Process Clearance

Legacy MLS systems also have the feature of a process clearance which is the highest sensitivity level at which a process can access data. The Bell-LaPadula read/write rules are based on the sensitivity level of the process but the process also has a clearance which corresponds to the user clearance; the highest level of information for which the user is authorized on the system. Typically, users operate at a specific level less than their clearance. With privilege, processes could access information up to the clearance.

In SELinux, the process sensitivity level and clearance are combined into a single security element; range. The range security element consists of a low sensitivity level and a high sensitivity level corresponding to the legacy MLS process sensitivity level and clearance. Both processes and objects may have a range instead of a single sensitivity label. Ranged objects are not discussed in this paper.

2.4 Roles and Authorizations

Legacy MLS systems have roles and authorizations to avoid giving privilege directly to the user. The model is that a process can query the user's authorizations or roles to determine if privileges should be used to accomplish a task. The program would query the user's authorizations or roles using proprietary API calls and if the user were authorized for a specific task, the process would raise privileges to accomplish the task.

In SELinux, roles serve a gatekeeper function to determine whether a certain domain is accessible within that role. Domains are assigned to roles in the policy and users are assigned roles by the security administrator. A process does not have to query whether the user has been assigned the role. Policy itself enforces whether a process running in a particular domain can be run by the user.

The difference between legacy MLS systems and SELinux in the areas of roles and authorizations is similar to privileges. SELinux policy defines valid contexts and roles are defined in policy not by the individual process.

2.5 Audit

An audit facility provides an unalterable record of security relevant events on the system. It is a basic security requirement to insure users are accountable for their actions. Legacy MLS systems and Linux both provide this facility but controlling the generation of audit records is accomplished differently. Audit records are generated by the kernel and privileged processes. In legacy MLS systems, a specific privilege is needed to write to the audit trail. In SELinux, a process must have the EUID=0 or have the AUDIT_WRITE capability and the process domain must be appropriately configured. Architecturally, both legacy MLS and SELinux accomplish audit in much the same way writing to a kernel buffer and relying on an audit daemon to write the information to the file system. In legacy MLS systems, the audit trail is usually in binary format requiring a utility program to translate it while the Linux audit trail is in plain text.

2.6 Other Security Features

Although privileges, sensitivity labels, authorizations, roles, and audit provide the basis of security features in legacy MLS systems, several other security features are important. Filesystem polyinstantiation, trusted networking and trusted windowing are features of the legacy MLS systems which are being or have been developed for SELinux but will not be discussed in this paper. Trusted networking and filesystem polyinstantiation are present in Red Hat Enterprise Linux 5. Information labels were also part of some legacy MLS systems. These were used as an advisory mechanism rather than an enforcement mechanism. Initially intended to aid the systems security officer in identifying the actual level of information in a file, they were not extensively used and were eventually removed from some of the legacy MLS systems.

3 Porting code from Legacy MLS systems to SELinux

As stated in the Introduction section, there are many reasons to consider porting applications from legacy MLS systems to SELinux. However, porting from a legacy MLS system to SELinux is not trivial. There are no standards in legacy MLS systems or SELinux for security API calls. Hence *all* security legacy MLS API

calls must be changed. Furthermore, the base of most of the legacy MLS systems was the Unix flavor particular to the vendor. For HP it is HP-UX; for Sun it is Solaris; for IBM it is AIX, and so forth. Besides porting the specific security API calls, the basic operating system API calls need to be ported as well.

4 Porting Trusted Solaris Applications to SELinux

The first part of this paper discussed legacy MLS systems as a whole. The general architecture differences and similarities pertain to multiple legacy MLS systems. However, in order to discuss specific porting issues, it will be necessary to narrow the discussion to a single legacy MLS system. Therefore, the rest of this paper will focus on one specific legacy MLS system, Trusted Solaris, although similar discussions apply to all legacy MLS systems.

4.1 Porting Solaris Code to Linux

Before porting the Trusted Solaris API calls to SELinux, the first task is to port the standard Solaris operating system and library calls to Linux. Fortunately, standards, such as POSIX.1, UNIX 95 and UNIX98 exist and make this task somewhat easier. Porting guides are available to assist with this task.

4.2 Trusted Solaris Code to SELinux

After dealing with the standard Solaris code to Linux porting, the next task is to evaluate what needs to be done with the Trusted Solaris calls. Some calls will be ported to comparable SELinux library routines while other calls will result in policy work. Most of the Trusted Solaris library calls are in the `tsol` library, `/usr/lib/libtsol.so.1`, and the audit calls are in `/usr/lib/libbsm.so.1`. Several other libraries contain routines but these are the main libraries.

5 Porting Sensitivity Label Code

Trusted Solaris labels all subjects and objects with a CMW label. The CMW label consists of the sensitivity label and the information label. The information label is similar to the sensitivity label, consisting of a hierarchical level plus categories, but it also contains

handling caveats or directives on how to properly handle and dispose of the information. The use of information labels was deprecated in Trusted Solaris 7 but the structure still remains and has a default value of unclassified. The library routines to get and set sensitivity labels on objects and subjects still use the CMW label structure and additional library calls are needed to deposit or extract the sensitivity label from the CMW label structure.

In SELinux library routines are used to get and set the security context on subjects and objects. The basic structure is the security context. The sensitivity label, or range, is one of the elements in the security context. In order to get or set the sensitivity level of a subject or object, the entire security context is set or retrieved and the range is extracted or deposited in the security context.

A simple Trusted Solaris program to display the sensitivity label of a file is shown in Table 1.

```
#include <tsol/label.h>
main()
{
    int retval, length = 0;
    bclabel_t fileCMWlabel;
    bslabel_t fsenslabel;
    char *string = NULL;
    /* Get file CMW label */
    getcmwlabel("/app/foobar",
                &fileCMWlabel);
    /* Get sensitivity label portion */
    getcsl(&fsenslabel, &fileCMWlabel);
    /* Translate file SL and print */
    bsltos(&fsenslabel, &string, length,
           LONG_CLASSIFICATION);
    printf("File sensitivity label =
           %s\n", string);
}
[OUTPUT]
File sensitivity label = CONFIDENTIAL
```

Table 1

The program reads the CMW label from a file, extracts the sensitivity label, changes the binary sensitivity label to a string and prints it out. Note that only essential library routines are included and error handling code has been left out for clarity.

The same operation in SELinux is similar but the steps vary slightly. In SELinux, the security context string is retrieved from the file and then put into the `context_t` structure from which individual security attributes can

be retrieved. Equivalent SELinux code is shown in Table 2.

```
#include <stdio.h>
#include <selinux/selinux.h>
#include <selinux/context.h>
main()
{
    int retval;
    security_context_t seccon;
    context_t contxt;
    /* Get file context */
    retval=getfilecon("/app/foobar",
        &seccon);
    /* Convert to a context_t struct*/
    contxt=context_new(seccon);
    /* Print the range */
    printf("File Range is %s\n",
        context_range_get(contxt));
}
[OUTPUT]
File Range is Confidential
```

Table 2

The previous simple programs illustrate that the process of retrieving and displaying the sensitivity label of a file is similar for Trusted Solaris and SELinux. The process to set the sensitivity label would basically be the reverse. However, each program would need privilege to work since they would be changing a MAC attribute. The process of retrieving and setting the sensitivity label of a process is the same except with library calls directed at processes instead of files.

5.1 Sensitivity Label Library Differences

Since Trusted Solaris sensitivity labels and clearances can be in one of three formats, there are several routines to convert labels between the different formats. One of these, `bsltos()`, used in the example program translates the binary sensitivity label structure to a string. There are routines to translate any label or clearance structure from one format to another. None of these types of routines are needed in SELinux since the format of all security attributes is always a string in user space.

Trusted Solaris contains additional routines to not only initialize sensitivity label and clearance structures but also to set specific sensitivity levels. Trusted Solaris library routines exist to initialize a binary sensitivity label or clearance structure with the lowest sensitivity level or the highest sensitivity level on the system. SELinux has no comparable routines that allow an application to get the highest or lowest sensitivity level

on the system. In order to do this, support is needed from policy since there is no way of knowing `s0` is the lowest defined label or that `s15:c0.c255` or `s15:c0.c1023` is the highest label.

Trusted Solaris also contains routines to compare sensitivity labels for equality, dominance, strict dominance and whether a label is within a range. These are needed for program run time decisions based on label values. No comparable routines are present in SELinux which simply treats the label as an unordered value. Some support for these types of operations exists using the `security_compute_av()` or `avc_has_perm()` library routines but these routines evaluate access based on the entire security context rather than just the range field. This could present some extra work for porting application code to SELinux.

Trusted Solaris also contains library routines to determine the least binary level that dominates two label structures passed to it as well as the greatest binary level that is dominated by two binary labels. These routines are useful for programs that must combine information from different sensitivity levels and set a minimum level that dominates both or a level that is dominated by two labels. Support for comparable operations is not found in SELinux. Lack of support for these types of label operations could require additional coding and logic.

Finally, Trusted Solaris extends API coverage to retrieve and set sensitivity labels on additional objects like interprocess communication objects (message queues, shared memory areas and semaphores). SELinux limits library support for retrieving and setting security context to processes and files. APIs do not presently exist for IPC objects. This may be a problem for some applications.

In summary, the basic operations of setting and retrieving sensitivity labels and clearances in Trusted Solaris will port easily to comparable SELinux library routines for files and processes. Operations based on the value of labels or that depend on the relationship of labels in Trusted Solaris may not port as easily to SELinux. Furthermore, the lack of SELinux security context library support extending to all objects may prove also prove to be a hindrance.

6 Porting Process Privileges

Process privileges exist in Trusted Solaris to allow trusted programs to override the security policy. Trusted Solaris defines over 80 discrete privileges that override both DAC and MAC policies for various subjects and objects. Processes get privilege through inheritance from the parent process and from privileges on the executable file. The individual privileges that a process can use are limited by the “allowed” privileges on the program executable file. Since the kernel only checks for privileges in the “effective” set, trusted programs used API calls to raise a privilege when needed for an operation and lower the privilege when no longer needed.

SELinux accomplishes the equivalent of Trusted Solaris privileges by using Linux capabilities and SELinux type attributes in policy creation. However, there is a significant difference in that capabilities and type attributes are assigned to domains instead of processes. This means that porting a Trusted Solaris program that uses privileges will involve creating a domain with the appropriate capabilities and type attributes to perform the equivalent task. At the code level, the Trusted Solaris routines manipulating privileges would be removed. Since a new domain is going to be created, appropriate type enforcement rules will also need to be written to ensure that the trusted process is allowed to access only what is required.

Another change may have to be made in the UID under which the process runs. Trusted Solaris *replaced* the root account with discrete privileges. Simply being EUID=0 in Trusted Solaris gives simply that DAC identity; no privilege. SELinux leaves the entire Linux root account privilege in place and *adds* additional MAC controls after DAC checks are done. A Trusted Solaris process could run with any UID and rely on privilege for traditional root override. In SELinux, a process will have to run with the EUID=0, or use capabilities, to pass traditional Linux DAC checks, plus have capabilities and type attributes assigned to the domain in which it runs to pass MAC checks.

6.1 Process Privilege Example

A Trusted Solaris program to change the label of a file is illustrated in Table 3. The sensitivity level of the process and the file are both Unclassified. The file is owned by a user that is not the EUID of the executing process.

```
#include <tsol/label.h>
#include <tsol/priv.h>
#include <stdio.h>
main()
{
    int err, len= 0;
    bclabel_t flCMWlab;
    bslabel_t newsl, fsl;
    char *string=NULL, *string1=NULL, *str;
    set_effective_priv (PRIV_SET, 0);
    /* Get file CMW label */
    getcmwlabel ("/app/foobar", &flCMWlab);
    /* Set the SL to CONFIDENTIAL */
    set_effective_priv (PRIV_ON, 4,\
    PRIV_FILE_DAC_WRITE, PRIV_FILE_MAC_READ,\
    PRIV_FILE_UPGRADE, PRIV_SYS_TRANS_LABEL);
    stobsl("CONFIDENTIAL", &newsl, NEW_LABEL,\
    &err);
    setcsl(&fileCMWlabel, &newsl);
    setcmwlabel ("/app/foobar", &flCMWlab,\
    SETCL_SL);
    /* Reread the file CMW label */
    getcmwlabel ("/app/foobar", &fcl);
    getcsl(&fsl, &fcl);
    bsltos (&fsl, &string1, len,
    LONG_CLASSIFICATION);
    printf ("NEW file SL = %s\n", string1);
    set_effective_priv (PRIV_OFF, 4,\
    PRIV_FILE_DAC_WRITE, PRIV_FILE_MAC_READ,\
    PRIV_FILE_UPGRADE, PRIV_SYS_TRANS_LABEL);
}
[OUTPUT] New file SL = CONFIDENTIAL
```

Table 3

Privileges to override both MAC and DAC policies are needed. All effective privileges are initially cleared and raised only when needed for changing the file sensitivity label and to re-read the label now that the label of the file dominates the label of the process.

SELinux code to perform the comparable task under the same circumstances is shown in Table 4.

```
#include <stdio.h>
#include <selinux/selinux.h>
#include <selinux/context.h>
main()
{
    security_context_t seconstr, con;
    context_t seconstrct, secl;
    /* Get file context */
    getfilecon("/app/foobar", &seconstr);
    seconstrct=context_new(seconstr);
    /* Assign new Sensitivity label */
    context_range_set(seconstrct,"Confidential");
    seconstr=context_str(seconstrct);
    setfilecon("/app/foobar",seconstr);
    getfilecon("/app/foobar", &con);
    secl=context_new(con);
    printf("NEW sensitivity label is %s\n",\
    context_range_get(secl));
}
[OUTPUT] NEW sensitivity label is Confidential
```

Table 4

Note that although the code in Table 4 requires privilege to change a MAC attribute, there are no API calls to establish that privilege. The privilege API usage in the Trusted Solaris program results in creating a domain in which the SELinux process can run. The SELinux policy to create the domain with the required type enforcement statements, capabilities and type attributes is illustrated in Table 5.

```

policy_module(localmisc, 0.1.5)
require {
    type user_t;
    type user_tty_device_t;
    type user_devpts_t;
    attribute mlsfilereadtoclr;
};
# Create new type and domain
type chglab_t;
type chglab_exec_t;
domain_type(chglab_t)
# Assign the new domain to user_r role
role user_r types chglab_t;
# Define transition to the new domain
domain_entry_file(chglab_t, chglab_exec_t)
domain_auto_trans(user_t, chglab_exec_t,
chglab_t)
# Assign type attributes and capabilities
domain_obj_id_change_exemption(chglab_t)
typeattribute chglab_t mlsfilereadtoclr;
mls_file_upgrade(chglab_t)
allow chglab_t self: capability { fowner };
# Rules for terminal communication
allow chglab_t user_tty_device_t:chr_file {
read write getattr ioctl};
allow chglab_t user_devpts_t:chr_file { getattr
read write };
domain_use_interactive_fds(chglab_t)
# Need access to shared libraries
libs_use_ld_so(chglab_t)
libs_use_shared_libs(chglab_t)
# Type enforcement rules for file access
allow chglab_t user_t: file { read getattr
relabelfrom relabelto };
allow chglab_t user_t: dir search;
allow chglab_t user_t:process sigchld;
fs_associate(user_t)

```

Table 5

Instead of raising privileges, the domain in which the program ran would need the `mlsfilereadtoclr` type attribute assigned to the process domain. Additionally, the `mlsfileupgrade` type attribute is needed which is provided through the `mls_file_upgrade()` policy interface. Those are the minimum type attributes needed but if the user element in the file security context differs from the user element in the domain, the `can_change_object_identity` attribute would also need to be assigned to the domain using the `domain_obj_id_change_exemption()` policy interface. Finally, if the owner of the file differs from the effective UID of the process, the process would have

to run as root and the `CAP_FOWNER` capability would have to be assigned to the domain. Various other type enforcement statements are required to enable the program to write to the terminal device and to access the file itself.

Thus, an operation in Trusted Solaris requiring several privileges results in changing library calls and some policy work in SELinux. A new domain would be created with attributes and capabilities to allow the process running in it to perform the required tasks. Attributes are assigned to the domain to satisfy policy constraint for changing SELinux security attributes. Capability is needed to provide a root file ownership override to the MAC policy. Furthermore, the process will need to run as the root user to be able to pass Linux DAC access controls associated with file ownership. Additionally, type enforcement rules permitting access between the domain of the process and the types of any objects accessed will have to be derived and implemented in the policy. In the simple example of changing the range attribute of a file and displaying the resulting label, several additional type enforcement rules permitting access to terminal devices were needed that are not needed in Trusted Solaris.

7 Porting Roles and Authorizations

Trusted Solaris roles and authorizations are assigned to the user's account. Programs implement these features by querying the account to see if a role or authorization is assigned and if so, the program raises privilege to accomplish the task corresponding to the role or authorization. This requires more proprietary API calls.

SELinux requires no application level code for role implementation. Roles are defined in policy and domains are assigned to roles. Programs which perform administration tasks will run in domains that have capabilities and type attributes to perform administration tasks. The roles are then assigned to SELinux users. Policy statements replace coding.

8 Porting Application Audit Code

Trusted applications write information directly to the audit trail to enable audit administrators to better understand what security relevant events are being performed. Trusted Solaris provides an easy to use interface via a single `auditwrite()` routine. This

routine uses tokens paired with values to enable the application program to write multiple records in a variety of formats until an AW_END token is passed. The process writing to the audit trail requires the `priv_proc_audit_appl` privilege to write audit event numbers outside of the range of the Trusted Solaris Trusted Computing Base (TCB) event number range.

Linux provides several `audit_log_user*()` routines to write information to the audit trail. Each of these routines has a fixed format of message type, character string message, with optional hostname, address and tty, and a required success code. The program must be running as the root effective UID to write audit records since audit is a Linux facility rather than a function done by SELinux. In addition to running as root, the domain of the process must have the CAP_AUDIT_WRITE capability and several permissions for the `netlink_audit_socket` object type.

9 Conclusions

While the examples discussed in this paper are somewhat simplistic, basic principles are illustrated in porting code from Trusted Solaris to SELinux. From these examples several conclusions can be drawn as porting issues.

- Porting code requires policy development work as well as code work.
- SELinux has additional MAC restrictions with type enforcement that may require additional work.
- Application code will have fewer proprietary library routines because SELinux uses a centralized policy.
- Some sensitivity label processes may require additional code due to lack of specific APIs.
- More processes will have to run with the root EUID because DAC checks are still in place.

10 Summary

Trusted Solaris, like other legacy MLS systems, benefits from over a decade of active use and development. Supportability problems resulted from the intrusiveness and inflexibility of the security policy. Although the technology base of SELinux is not new, widespread deployment of the technology is recent. There are many security features comparable to legacy MLS systems and porting from legacy MLS systems is a possibility.

The current lack of a trusted X window system as part of a standard Linux distribution limits the deployment of SELinux to server applications. However, SELinux security policy is more flexible and is included with several standard Linux distributions. These facts, along with the fact that Red Hat Enterprise Linux Release 5 with SELinux is in Common Criteria Labeled Security Protection Profile evaluation¹, should mitigate the usability and support problems of the legacy MLS systems. It is time to begin the migration.

References

- [1] Trusted Solaris Developer's Guide, Copyright 2001 Sun Microsystems, Inc.
- [2] man pages section 3: Library Functions, Copyright 2000 Sun Microsystems, Inc.
- [3] Legacy MLS/Trusted Systems and SELinux – concepts and comparisons to simplify migration and adoption, (White Paper) 2006 Hewlett-Packard

1 niap.bahialab.com/cc-scheme/in_evaluation.cfm