Automated Process Classification Framework using SELinux Security Context

Pravin Shinde and Priyanka Sharma
Centre for Development of Advanced Computing
Mumbai, India
Email: {pravin,priyanka}@cdacmumbai.in

Srinivas Guntupalli
ProCurve, HP
Bangalore, India
Email: gsrinivas@gmail.com

***Abstract*—Stringent Quality of Service requirements from operating systems led to several extensions to the existing systems. These extensions aim at classifying the processes in a system at runtime to provide differentiated Quality of Service. Also there are many other applications which do need classification of processes for their working. The methods used for identifying the processes and grouping them, by different extensions have been ad-hoc. Enabling several of such extensions adds to the complexity of administering a system. We propose an automated mechanism to classify processes using some persistent characteristics of a process. We use persistent tokens (*security contexts*) added to all kernel objects by Security Enhanced Linux. We present the overall problem as three sub-problems viz., *Notification*, *Classification* and *Enforcement*. The proposed solution solves *Notification* and *Classification* problems. *Enforcement* is left to the specific application that uses the framework.**

I. Introduction

Traditionally operating systems(OS) have been resource managers. The resource intensive applications and hostile environment in which they run created new challenges to operating systems. Several extensions were proposed in the literature that aim at making operating system, an efficient and secure service provider rather than a mere resource manager. To provide Quality of Service(QoS) at kernel, Class based Kernel Resource Management(CKRM) has been proposed. To efficiently use memory and processors in a multiprocessor system, CPU-Set has been proposed. Enhanced Linux System Accounting(ELSA) provides system accounting in user space. Even though these efforts to achieve resource management and accounting have been independent, they address some common issues and have some common behavior. This leads to a lot of overlap and repetition of work and makes the systems more complex and error prone. Salient feature of all these systems is to classify processes based on some process feature and provide differentiated QoS. SELinux, an extension to implement Mandatory Access Control(MAC) in Linux, identifies kernel objects in a unique way, which can be used by the above systems as well. We discuss all the systems in detail in the following sections.

A. SELinux

SELinux is an implementation of MAC using Linux Security Modules(LSM) in the Linux kernel, based on the principle of least privilege[1],[2](Certain distributions of Linux included SELinux in their main line releases). MAC has been introduced as a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects. It also verifies the authority of subjects to access information of such sensitivity[4].

To implement MAC, SELinux abstracts the system into reference monitor[5],[6]. Reference Monitor classifies resources into active and passive entities. Active entities such as processes are called subjects and passive entities such as files are called objects. The Reference Monitor mechanism controls access among these subjects and objects according to the specified security policy[7].

MAC is done in accordance with the policies specified by the administrator of the system. SELinux access control is based on access control attributes associated with objects and subjects. These access control attributes are called *security context*. All objects (files, inter-process communication channels, sockets and so on) and subjects (processes) have a unique *security context* associated with them. A *security context* has three elements: *user, role,* and *type* identifiers. The usual format for specifying or displaying a *security context* is as follows:

user:role:type

The *type* identifier is the primary part of the security context that is used to write rules in the policy database and determines access. The *security context* of an object is persistent. Kernel data structures have been extended to accommodate the *security context*. A policy database governs the *security context* to be assigned to a kernel object. Linux distributions ship a comprehensive policy database and administrators can customize it as per the local needs.

B. CKRM

Control of key resources such as memory, CPU time, disk I/O bandwidth and network bandwidth is strongly tied to kernel tasks and address spaces in Linux OS. There is very limited support from kernel to enforce user specified priorities during resource allocation. CKRM allows user level controls to provide differentiated service at a user or job level. It also enables accurate metering of resource consumption in user and kernel mode[8][9].

Linux kernel represents both processes and threads as tasks. A class (used by CKRM) is a group of tasks. The grouping of tasks into classes is decided by policies. A policy database, which is a collection of rules, decides task and class mappings. Administrator of the system creates a policy and makes it available to the kernel. The kernel optionally verifies the policy for consistency and activates it. Similar to SELinux, there needs to be a mechanism to uniquely identity each task. A Task-Tag, a user-defined attribute, is associated with a task in CKRM enabled systems. Using this tag application specific criteria can be taken care of while classifying, as the tag is user specified. Typically a system call, ioctl or /proc is the interface between application and kernel, for the application to convey its criteria through a tag. This tag is used in invoking a specific classification rule. The proportions in which resources should be allocated to classes is determined by a resource manager. This could be either a human system administrator or a resource management application middleware[8][9].

C. CPU Set

CPU-Sets are light-weight objects in the Linux kernel that enable users to partition their multi-processor machine by creating non-overlapping execution areas[10][11][12]. CPU sets can eliminate the need for a gang scheduler, provide isolation of one such job from other tasks on a system, and facilitate provision of equal resources to each thread in a job. Restraining all other jobs from using any of the CPUs or memory resources assigned to a critical job minimizes interference from other jobs on the system. This results in both optimum and repeatable performance. The kernel CPU-Set facility provides additional support for system-wide management of CPU and memory resources by related sets of tasks. It provides a hierarchical structure to the resources, with file-system like name-space and permissions, and support for guaranteed exclusive use of resources.

Each task has a link to a CPU-Set structure that specifies the CPUs and memory nodes available for its use. Hooks in the system calls used for CPU placement and memory placement ensure that any requested CPU or memory node is available in that task's CPU-Set. Kernel CPU-Sets are arranged in a hierarchical virtual file system, reflecting the possible nesting of soft partitions. The kernel task scheduler is constrained to only schedule a task on the CPUs of that task's CPU-Set. A CPU-Set contains a set of tasks/processes and those tasks can use processors and memory nodes that are associated with that CPU-Set. There is no classification engine available for CPU-Set to do the above mentioned functions automatically. An administrator has to execute them manually.

D. ELSA

Enhanced Linux System Accounting(ELSA) is a user-space solution for process accounting in Linux. Work is split into the following parts in ELSA.

- Connector : A Linux kernel feature
- A user space daemon : jobd
- Per-process accounting information : BSD and/or CSA
- User space applications : webmin + jobmng + elsa

The *connector* reports process events to user-space[13]. It uses the netlink mechanism and Linux kernel must be built with the necessary configuration options. *jobd* listens to the net link messages sent by the process event connector. This way, whenever a process is forked, it will be informed. This information is used manage a group of processes. Communication between *jobd* and high level applications happens through sockets. The high level application can use this mechanisms to send requests to add or remove a process from a job. The same mechanism can be used to information about current jobs. Thus, *jobd* is under the control of a high level application. BSD accounting or CSA accounting (which is external to ELSA) is used to obtain per-process accounting information. *jobmng* is the interface to manage groups of processes. *webmin* provides per-group accounting information using the information provided by *jobd* and *per-process accounting mechanism*. The interactions and interfaces are shown in fig 1.

II. Problem Description

Most of the extensions that we discussed use a configuration filesystem for policy management. CPUSets are represented as directories in the config file system. Resources have to be assigned to the CPUSets. In order to classify the processes into these CPUSET's, one need to get their PIDs, and add them to particular file in the directory indicating the corresponding CPUSet (as per their resource requirements). So, classification can only be done after the creation of the process. And, if a process restarts, then it has to be classified again, as it gets a new PID. Similarly, after every reboot classification has to be done. This process involves lot of manual intervention. Currently, there is no way to automate the process.

Similarly, during initialization of CKRM, its resource manager commits a policy to the kernel. A default class is created to which all tasks will initially belong until resource managers policy is loaded. Policy load triggers classification. Classification refers to association of tasks to classes and resource requests to a class. Processes are classified, by default, into the class of their parents. Classification engine uses UID, GID and executable associated with the process for classification. Administrators have to manage the config file system, which contains the list of process classes and their resource shares. Classification is a continuous process and happens whenever a new task is created or attributes of a task are changed or explicit reclassification of a task is done by resource manager. Resource usage is monitored at class level and that information is used for future decisions. Resource schedulers control the resource utilization by tasks that belong to different classes. The requirement to classify processes based on their attributes is similar to CPUSet. Management of classes needs manual intervention. There is no provision to specify the rules with persistent identifiers.

ELSA also has similar shortcomings. It provides tools like *jobmng*, *ELSA* which has to be used by system administrators for manually classifying the processes into job groups. As this classification is based on PID of a process, which is not persistent, every time system or *jobd* restarts, classification has to be done again. This process can't be automated due to the lack of persistent identifiers.

III. Sub-Problems

Specified problem can be broken down into three sub-problems, and each of them can be dealt separately and independently. The identified sub-problems are as follows,

- Notification
- Classification
- Enforcement

A. Notification

Creation of new process, or alteration of process is an event that happens inside kernel and is protected from userspace. This information can be obtained by inserting hooks into kernel and trapping system calls like *fork*, *exec* and *setuid*, which are responsible for these events. But this needs modification of kernel. There are some existing solutions which provide similar notifications about process events. Connector is a Linux kernel module which is implemented using netlink sockets. Netlink sockets provide a standard way of communication between kernel and userspace processes. Connector uses the netlink sockets to notify important process events like fork, exec, id change(UID, GID, SUID etc) to userspace. For connectors to work, kernel needs to be built with required support. Once they are enabled, they will notify all process related events to the userspace. As shown in fig.1, userspace programs can receive these notifications using netlink sockets.

B. Classification

Classification problem arises because of using non-persistent attributes of processes as keys. The processes need to be classified as per the rules provided by users. Some systems use PIDs as the unique identities of processes. But, PIDs are volatile in nature and a new PID is generated whenever a process starts. Even though the same application is executed multiple times in same environment by the same user, it will get different PID every time. As we can not find out the PID's of these processes in advance, it makes it almost impossible to write rules in advance, based on PIDs. Currently, most resource management systems, and other systems where process classification is needed, use manual classification. This is done based on some simple rules, like every new process will inherit the class of its parent process. These basic process classification systems provide an interface through which users can reclassify the processes using their PID as handle to them.

C. Enforcement

As different applications provide different interfaces to classify the processes, it makes the problem of *enforcement* more application dependent. And also, these applications are different in what they do after process classification. For example, ELSA and CPUSET provide different interfaces to change the class of a process, and also handle processes differently after classification, as they have different objectives. Depending on the need of application there may be a need of one more configuration files to provide rules about behavior after classification. For example, we need a configuration file for PCSS-CPUSET classifier, which will be having rules about when a new CPUSET is to be created, what should be the values to be assigned to various configuration parameters. Such configuration file is not needed in case of ELSA, as ELSA is not having any properties associated with jobs (i.e,. classes in ELSA). As there is no generic solution for this problem, different solutions are provided for different applications.

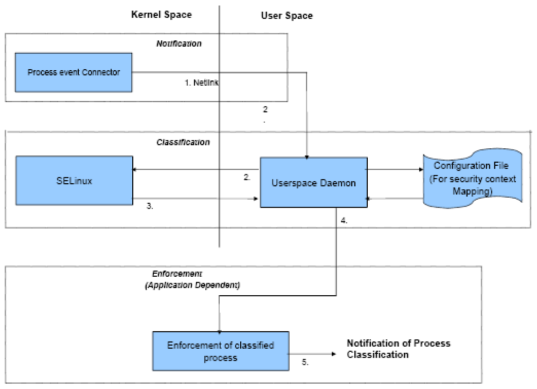
Kernel Space
User Space
Notification
Process event Connector
1. Netlink
2
Classification
SELinux
2.
3.
Userspace Daemon
Configuration File (For security context Mapping)
4.
Enforcement (Application Dependent)
Enforcement of classified process
5.
Notification of Process Classification

Fig. 1. Overview of PCSS Architecture

IV. Related Work

There are existing solutions to this problem. One of them is rule-based classification engine (RBCE), used by CKRM for automatic process classification. It classifies the processes based on the rules which can be written using UID, GID and name of the executable. But this classification is very basic and is not very flexible, as it does not consider the environment, in which a process is being executed. It also does not recognize various roles played by the user. Another problem with this solution is that, it does not make clear distinctions among above discussed subproblems (Notification, Classification and Enforcement), which restricts the implementation to be very specific to CKRM, and can not be easily reused at other places like ELSA and CPUSET.

V. Our Approch

We propose a classification solution *PCSS(Process Classifier based on SELinux Security_Context)* which is based on *security contexts* of SELinux. The class of a process can be decided based on its *security context* and the class mapped to this security context in configuration file. The configuration file is very simple and flexible. It contains only two parameters, one is *security context* and second is corresponding class name to which it should be assigned. As this module deals only with the problem of classification, it is generic enough, so that it can be used with any application that needs any kind of process classification.

A. Why SELinux Security Contexts?

The proposed solution depends primarily on SELinux kernel module. SELinux is available in Linux kernel from 2.6 onwards. Nowadays most of the distributions of Linux (i.e,. Fedora core) are shipped with SELinux enabled by default.

Major criticism against SELinux had been its complex policies which are difficult to configure. But, many user friendly tools like *seedit* are available now which simplify the policy management to a large extent. If SELinux is already enabled, then using PCSS for process classification is straight forward.

The advantages of using *security context* of SELinux for process classification are listed below:

- They are persistent (remain same across reboots)
- They are flexible (*Security context* differs depending on who is executing, which program is being executed, and in what context it is being executed)
- They are configurable (One can change SELinux policies to set *security contexts* as per one's needs. There are many user-friendly tools which help in doing this.
- Process classification rules are kept in separate file from SELinux internal policies. This helps in keeping classification rules very simple.
- User needs to modify configuration file only once, as per the local policies.
- Added flexibility is provided by supporting wild-characters in security context.

VI. Implementation details

To implement the *Notification* module, we used *connectors*, an existing infrastructure in Linux kernel. We have implemented a *userspace daemon*, which continuously listens using netlink sockets for messages from *connectors*. These messages contain information about events with the PID of the process, that caused the event. On receiving these messages from *connectors*, the *userspace daemon* finds out the *security context* of the process responsible for the event. It uses *getpidcon* system call to get the *security context* from the PID. The context is used to extract matching rules from configuration file. These rules are flexible due to *wild-character support*. If no matching rules are found, default classification is applied. If matching rules are found, classification is done by *Enforcement* module accordingly.

Different applications have different requirements from enforcement module. So, it is implemented separately for CKRM, CPUSET and ELSA. This module is mostly dependent on the interface provided by the application for classification of processes. As the interfaces provided by applications like CPUSET and ELSA, for process classification, are very much different, it is not feasible to provide a generic solution for all the applications.

A. Wild Character support

To enhance flexibility and ease of use, wild-character support has been incorporated in rule syntax. It allows users to mask irrelevant components of *security context*, and makes rules more compact. The usual format of *security context* is as follows:

user:role:type

But '*' can be put at any place of the security context replacing user and/or role and/or type. '*' instructs the *classification* module to ignore that part of *security context* while matching. Following examples illustrate the use of wild characters.

- **:*:httpd_t http_class*
  This rule groups all "httpd_t" processes in http_class, without matching user and role part.
- *user_t:*:httpd_t user_class*
  This rule groups all processes of "httpd_t" type and user is "user_t" without matching the role part.

B. PCSS CPUSET Integration

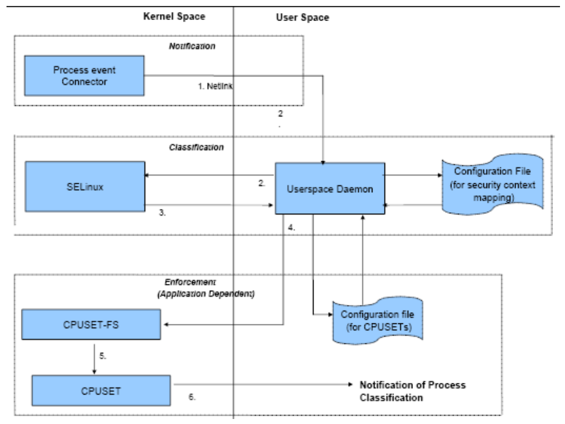
Kernel Space
User Space
Notification
Process event Connector
1. Netlink
2
Classification
SELinux
2.
3.
Userspace Daemon
Configuration File (for security context mapping)
4.
Enforcement (Application Dependent)
CPUSET-FS
Configuration file (for CPUSETs)
5.
CPUSET
6.
Notification of Process Classification

Fig. 2. CPUSET with PCSS

CPUSET classes are implemented inside kernel by CPUSET kernel module. It provides interface in the form of a virtual filesystem called CPUSET. As the CPUSET objects have many properties associated with them like *cpus, mems, cpu_exclusive, mem_exclusive, memory_pressure, etc.,* we use separate configuration files for these CPUSET properties. In integrated PCSS-CPUSET, *Notification* and *Classification* phases do remain the same as in generic framework, but *Enforcement* is implemented to handle the CUPSETfs interface provided by CPUSET. Every directory in CPUSET filesystem is a separate CPUSET class. A process is moved to a CPUSET class by appending its PID in a members file inside directory corresponding to the given CPUSET class.

Enforcement module starts by detecting the mount location of virtual filesystem provided by CPUSET. After that, whenever a process is classified into any CPUSET, the enforcer first searches to see if the specified CPUSET is already present or not. If specified CPUSET is found, then the process is directly added to that CPUSET. Otherwise, enforcer creates new CPUSET using information given by the rules in the configuration file for CPUSET. And, the process is moved to the new CPUSET.

C. Integration of PCSS and CKRM

CPUSET and CKRM are very much similar in work and implementation. CKRM also provides virtual filesystem named as CONFIGFS. CPUSET and CKRM do defer in the properties associated with these objects. So the configuration file used by classification enforcer for creating classes will differ. But, CKRM is no more supported in latest Linux kernels (after kernel-2$\dot{6}$18). Instead a more elegant resource management solution named *containers* is being brought up. So, we didn't discuss integration of CKRM and PCSS.

D. Integration of PCSS and ELSA

In the integrated PCSS-ELSA also, *notification* and *classification* phases do remain the same as in the generic framework, but *enforcement* is quite different from PCSS-CPUSET's counterpart. ELSA relies on BSD process accounting kernel module, to make the accounting data available to the userspace. ELSA jobs are managed by the same userspace program which does the classification work. Besides, it provides another interface using UNIX sockets. This interface allows users to change the classification of processes.

We are using the same mechanism provided by ELSA to classify the processes in jobs. The ELSA userspace daemon named *jobd* has been customized for this additional functionality. On receiving the notification of process related events from *notification* module, *jobd* interacts with the *classification* module to classify the processes as per user specified rules, and enforces the classification by simulating the pseudo user classification request. These pseudo requests trigger the actual process classification inside ELSA.

Above discussed implementation has been posted as a project on sourceforge. For more details, readers are requested to refer to the project web pages[14].

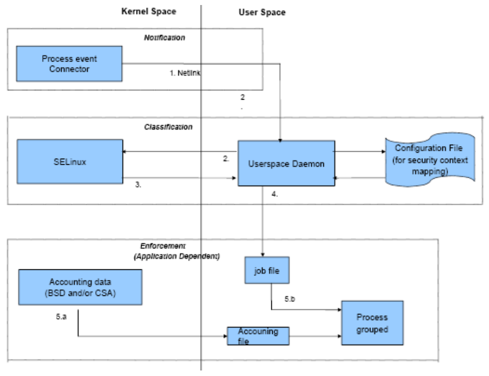

Fig. 3. ELSA with PCSS

VII. Conclusion

By breaking down the problem of process level QoS provisioning into sub-problems and providing generic solution for *Notification* and *Classification*, we abstracted out generic things from the large problem and provided a generic framework. Any application that needs *Notification* and *Classification* can use this framework. Also, use of SELinux *security contexts* for classification made the framework more flexible and easy of use.

VIII. Future work

In the current implementation, when *security context* of a kernel object change, *notification* module doesn't send any specific notification. Due to this, current solution needs to monitor all notifications to detect changes in *security contexts*, which overloads the application. If connectors can be modified to include such notification, this workload can be reduced.

Also, SELinux does not provide any explicit support for process classification from any other point of view except security. SELinux support can be enhanced, by extending its *security context* to support other types of process classification also, to include class field, as shown bellow.

user:role:type:class

This kind of modification can be used to considerably increase the flexibility of classification. Also, there are other applications and solutions like *containers*(upcoming resource management solution), which do use userspace classification in their working. This framework can be modified to incorporate *containers* as well.

Acknowledgment

The authors would like to thank C-DAC for providing the infrastructure and support

References

[1] National Secuirty Agency - Security Enhanced Linux. *http://www.nsa.gov/selinux/info/faq.cfm*
[2] Security enhanced Linux from wikipedia. *http://en.wikipedia.org/wiki/SELinux*
[3] U. S. D. of Defense, Trusted computer system evaluation criteria, DoD Standard 5200.28-STD, December 1985. *http://www.fas.org/irp/nsa/rainbow/tg003.htm*
[4] Mandatory access control from wikipedia. *http://en.wikipedia.org/wiki/Mandatory access control*
[5] J. P. Anderson, Computer security technology planning study, Tech. Rep., Oct. 1972. http://csrc.nist.gov/publications/history/ande72.pdf
[6] F. Mayer, K. MacMillan, and D. Caplan SELinux by Example: Using Security En-hanced Linux. Prentice Hall.
[7] R. Spencer, S. Smalley, P. Losocco, M. Hibler, D. Andersen, and J. Lepreau, The flask security architecture: System support for diverse security policies, no. UUCS-98-014, 1998. [Online]. Available: citeseer.ist.psu.edu/spencer98flask.html
[8] S. Nagar, H. Franke, J. Choi, and C. Seetharaman, Class-based prioritized resource control in linux, in Proceedings of the Linux Symposium, 2003.
[9] S. Nagar, R. van Riel, H. Franke, C. Seetharaman, V. Kashyap, and H. Zheng, Im- proving linux resource control using ckrm, in Proceedings of the Linux Symposium, 2004.
[10] CPUSets for Linux. [Online]. Available: http://www.bullopensource.org/cpuset/
[11] ELSA on Sourceforge. [Online] Available: http://elsa.sourceforge.net/
[12] Linux Cross Reference. [Online] http://lxr.linux.no/source/Documentation/cpusets.txt
[13] Connectors [Online] http://lxr.linux.no/source/Documentation/connector/connector.txt
[14] PCSS on Sourceforge [Online] Available: pcss.sourceforge.net