Access policy generation system based on process execution history

Toshiharu HARADA　Takashi HORIE　Kazuo TANAKA
Research and Development Headquarters, NTT DATA CORPORATION
e-mail: {haradats, horietk, tanakakza}@nttdata.co.jp

Abstract

MAC (Mandatory Access Control) has the ability to improve security of Linux operating system dramatically. However, defining and managing proper policy is not easily achieved because program dependencies are usually invisible from system administrators. This paper presents the challenges in providing automatic policy generation based on process execution history.

1. Introduction

The realization of appropriate access control plays the core role when we consider about the security of computer systems, and the researches on security models for the OSes and their implementations has been continued from early on. There are two indicators of access control mechanism, the granularity of access controls and the enforcement of access controls. The SELinux (Security-Enhanced Linux)[1, 2] developed by NSA (National Security Agency) fully realized these indicators on Linux OS, and the SELinux is carefully watched around the world.

While the author of this paper (hereafter, we) evaluated some MAC (Mandatory Access Control) implementations on Linux such as SELinux, SubDomain [3] and RSBAC [4], we noticed that there is an essential difficulty due to the fundamental structure of Linux OS in applying fine-grained access control on Linux, and the difficulty finally incarnates in the form of the most laborious task for administrators to develop just enough policy.

This paper describes the operational issues and backgrounds when applying MAC on Linux, and demonstrates an automatic access policy generation system for SELinux we have developed as a method to solve the operational issues.

2. Mandatory Access Control

2.1. Linux's Access Control

The Linux's built-in access control is performed in the following way.

The owner of files or directories specifies access permissions in the combination of read/write/execute for files (or read/write/search for directories) to three groups categorized by owner/group/others, and the specified access permissions are kept and managed by the owner.

The access requests are unconditionally authorized if the process that requests accesses is running with the system administrator's privilege, for the UNIX is parental of Linux and Linux inherited the traditional UNIX's access control. Therefore, it is inevitable that the standard Linux gets fatally damaged if the cracker usurps the administrator's privilege. This is known as a vulnerability of Linux.

2.2. Abstract of MAC and merits of applying MAC on Linux

The MAC is described as the core function of "Class B1: Labeled Security Protection" in TCSEC (Trusted Computer System Evaluation Criteria) [5] published by DOD (Department of Defense) in 1985.

The feature of MAC is that access requests are processed only after the each access request was authorized by predefined policy information. The MAC itself is a generic approach that is independent to specific OSes. This method is very effective when applied on Linux because Linux has too simple

access control model and vulnerability caused by the model already mentioned. The MAC is supported by commercial trusted OSes and used in attempt to improve security at the OS level without exception.

2.3. Problems of MAC

The implementation of MAC in OS is divided into the following three main branches.

(a) Insertion of conditional branching for access requests inside kernel space.

(b) Mechanism to keep policy.

(c) Mechanism to authorize individual access requests based on policy.

Retrofitting MAC entails extensive modifications of kernels and inevitable loss of total system performance due to its positioning. It also brings an important problem of burdens of developing just enough policy on introduction phase to the surface. These are the problems of MAC generally recognized, (though we think it is not appropriate to recognize as "problems" because this is the nature of MAC). Needless to say, it is impossible to check the validity of individual access requests automatically. The MAC provides functions or mechanisms that enforce access controls based on policy, but it is necessary for making use of MAC properly to authorize properly necessary access requests. The system's security won't improve if the administrator authorizes unnecessary access requests, but the system won't run if the administrator failed to authorize necessary access requests for services.

2.4. Applying on Linux

In Linux, requests from application processes are finally processed in the kernel space via system calls. Therefore, the system calls are usually chosen as the location to enforce MAC. It means that by performing authorizations of accesses for each system calls, the system can monitor and control application processes behavior completely. The Linux's built-in access information for files and directories are called "security bits" and are kept as an attribute of filesystem. But since the range of access requests that the MAC controls is not limited to files and directories, and conditions is more complicated than standard "security bits", the access information for MAC is kept in a newly defined data structure managed by the kernel and is referred from the kernel space.

We think that the biggest problem of introducing MAC into Linux is the policy management, especially developing just enough policy. The basic style of operating Linux is that "Implement special procedure using C language" and "Combine with commands supplied by OS using (for example) shell scripts" to provide functions and services [6]. Normally, the system administrator needn't to know "How the services and programs I'm running are configured with and realized by individual components". But to introduce MAC to reject unnecessary access requests, the system administrator has to know "What programs are invoked to provide services I need" and "What resources does each program access" and develop the list of the just enough access permissions as a policy.

We don't go into more detail about the problems of policy definition. But we would like to mention that the evaluation criterion of policy is not the one and only. Suppose the administrator knows the access permissions in kernel space and the configurations of applications to run and all necessary access permissions for each application, the content of the policy the administrator develops depends on the requirement of protection level, and the answer is not the one and only.

The difficulty of operating Linux with MAC support is that the administrator can't determine whether the policy is just enough or not.

3. SELinux

To explain the method we have devised and developed to improve policy definition for MAC in Linux, we briefly explain SELinux as one of the MAC implementations in Linux. The MAC introduced by SELinux is based on Flask [7] that NSA has been researching and developing. The labels are assigned to both the subject processes and the object resources, and label-based access controls are performed.

The label assigned to processes (i.e. the subjects of accesses) is called "domain".

Since the actual services such as WWW and FTP provided on Linux generally work in concert with multiple programs, SELinux defines domain transitions (the relationship between programs) to control program invocation between programs. We don't explain for details due to limitations of space. Please refer to information [8, 9] by (for example) IPA for the abstract of SELinux, and the paper [10] by SELinux's developers.

3.1.1. The features of MAC in SELinux

The policy definition in SELinux is mainly categorized into the following parts.

(a) Definition of domains and the range of accessible resources for each domain.

(b) Associating each domain with the range of accessible resources for each domain using labels.

(c) Definition of domain transition.

The policy definition in SELinux cannot get away from the problems already mentioned. The policy definition in SELinux entails the following difficulties.

- The administrators have to determine on their own the standard of the grouping of classified access permissions needed for handier policy definition and the grouping of programs needed for domain assignment.
- The administrators have to understand exactly all permissions needed for services.

  SELinux provides a script program (as a utility to assist policy definition) that converts error messages from the kernel into access permissions that can solve the cause of error messages. The administrators can use this script to append missing access permissions at the end of tuning phase, but they can't use this script to determine the validity of policy configuration at the beginning of operation phase. (It is possible to detect missing access permissions from error messages, but it is impossible to detect redundant access permissions.)
- It is impossible to define the range of accessible resources based on program execution history, for the domain doesn't include the domain transition history. But the access permissions have to be granted to domains that don't include the domain transition history. Therefore, access permissions have to be granted as a union for domains that are transited from multiple domains.

  In the default policy files distributed with SELinux, a domain "shell_exec_t" is defined for shell (command interpreters used in Linux) programs. Normally, the administrators modify policy files so that minimum domains can transit to the "shell_exec_t" domain. Shells are the most frequently used programs in Linux, and it is very dangerous if shells are invoked by attacking the vulnerability of any programs because various programs that are allowed to invoke from shells will be invoked by the attacker. But the services can't work immediately if the invocation of shells is unconditionally prohibited. To grant access permissions for minimal accessible resources when invoking shell programs, the administrators have to consider "what contexts are the shell programs called from" and determine minimal accessible resources according to each context. We used shell programs as an example, but the administrators have to deal with all programs and all files that are accessed by each program in the same manner, and it is significantly difficult task.

4. Improving policy definition for MAC on Linux

The resources that MAC can control depend on its implementation. Our study to assist generating just enough policy aims at file access only, for almost all MAC implementations can control and that constitutes a large share of policy definition. We also devised a system for evaluation purpose. The steps until authorization of access permissions are independent of individual MAC implementations, but we explain the final policy output using SELinux's policy syntax.

4.1. The goal settings and approaches

It is impossible to completely automate the task of shaping up to obtain less redundant policy. Also, there is not the one and only solution. Therefore, we set the goal as developing a support environment that allows system administrators generate just enough policy. The approach is, based on the analysis of the difficulty of policy definition for MAC on Linux already mentioned, "Run programs and services actually" and "Capture access requests by the system calls and marshal" and "Provide the system administrators the result using GUI so that they can edit and authorize", and we considered that this approach will allow system administrators just and enough policy definition.

4.2. The system configuration

The configuration for this approach is shown in Fig. 1.

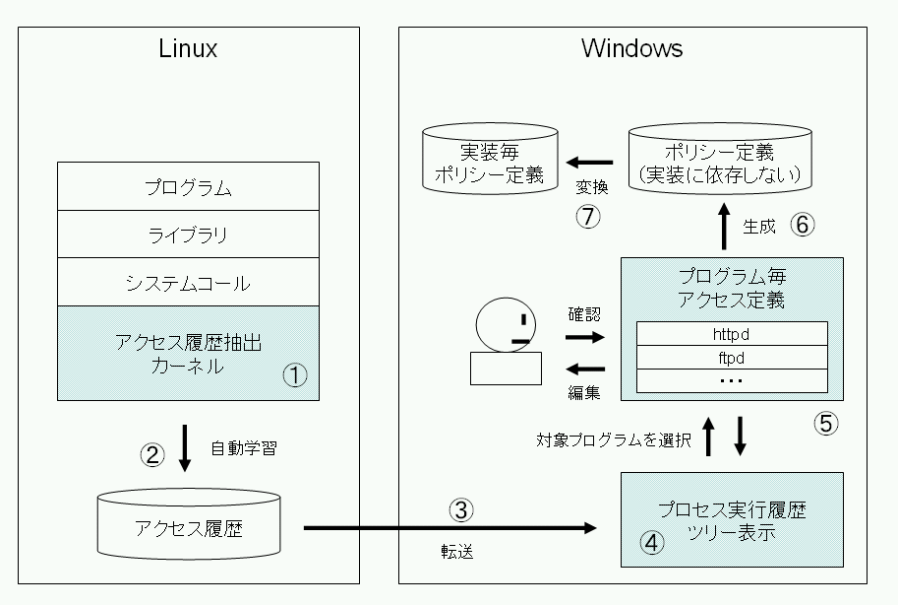

Fig. 1

We explain the flow using the numbers assigned in Fig. 1.

(1) Develop a customized version of Linux kernel that can log information such as "the fullpath of programs that called the system call", "the fullpath of files or directories that are accessed" and "the mode of access requests" when any system calls such as opening files that are controllable by access permissions are called. Run applications such as WWW services on this customized kernel. (Run a series of key operations.)

(2) Save the log information kept in the kernel memory onto the disk as a file.

(3) Transfer the file to a PC that is used for policy definition using ftp.

(4) Analyze the file and generate process transition tree starting from /sbin/init and show the tree using GUI.

(5) Choose the program from the tree to edit access permissions using GUI. Confirm the access permissions, and if necessary, choose different access permissions from the list. Since access permissions captured by the kernel are shown by default, the administrator can start editing from the policy that likely has just enough access permissions needed for individual applications.

(6) After the administrator authorized all access permissions for all applications, merge the individual access permissions and save as the policy file.

(7) Convert the saved policy file into individual policy for MAC implementations.

Although steps (1) and (2) have to be performed on Linux, steps (3) and after can be performed on any platform. We used Windows to improve the efficiency of GUI development.

5. Implementation

5.1. Patching to the kernel

5.1.1. The method of obtaining access history ant its granularity

We chose the location and the content of collecting execution history, based on the research on other MAC implementations including SELinux, as shown in Table 1. We installed RedHat Linux 8.0 as the subject for experiment and installed the modified 2.4.20 kernel that has necessary hooks inserted.

Table 1

Filename	Location	Permission	Note
fs/open.c	sys_open()	rw	Detects file open.
	BufferedLog()	Access log	Newly created function.
fs/exec.c	do_execve()	x	Updates process execution history.
fs/stat.c	sys_readlink()	l	Detects reference of symbolic links. (Also reads out access logs.)
kernel/fork.c	do_fork()		Duplicates process execution history.
kernel/exit.c	do_exit()		Destroys process execution history.
include/linux/sched.h	struct task_struct		A variable is added to hold process execution history.

It is possible to use LSM (Linux Security Modules) [11], the framework of security extension for Linux that SELinux also uses, for auditing access history. But we didn't use LSM so that our approach can also work on MAC implementations that don't use LSM, and we modified the vanilla kernel directly. The common method of auditing logs is to transfer messages generated via printk() by the kernel to /var/log/messages via syslogd. But this method works only while the syslogd is running. Also, since the buffer used by printk() is not thread-safe, the messages may be corrupted if multiple processes called printk() simultaneously. Therefore, we didn't use printk() for auditing.

The function BufferedLog() is the newly created function that holds access logs in kernel space. This function allocates 128KB of memory statically using kmalloc(), and holds access logs for opening files and resolving symbolic links. We sucked out the access logs via sys_readlink(). Though we used sys_readlink() as a readout window, any system calls that have string parameter for passing keywords and receiving results and accessible from userland applications are acceptable. Since the userland application sucks out the access log using existent system calls and saves into files, this approach has the two merits, "The developer needn't to make the modified or newly created functions visible to all userland applications" and "The administrator can save logs after the syslogd stopped until the filesystem is remounted as read-only".

In Linux, all processes run as child or descendant of /sbin/init. When the child process is to start, the image of current process is duplicated via sys_fork(), and the duplicated image is replaced with new image via do_execve().

The processes are managed by the structure called "task_struct" and this structure is also duplicated by sys_fork().

When the process terminate, this structure is destroyed by do_exit().

This structure is assigned for every process on one-to-one basis, and holds information such as memory usage, filesystems and privileges. We added a string variable (char *) to this structure to hold the process execution history, and modified sys_fork(), do_execve() and do_exit() to duplicate, update, destroy this variable respectively.

We modified the kernel to call BufferedLog() to dump the current value of process execution history, the requested pathname and requested access mode(read/write/execute) when access requests are detected at sys_open() and sys_readlink(). An example of output is shown in the appendix.

The access logs kept in the BufferedLog() are immediately sucked out by the userland application via sys_readlink() and stored into log files.

5.2. Filter out invalid access logs

Filter out invalid access logs from the access logs obtained by the customized kernel. Since the pathnames are recorded regardless of the existence of that pathname, we need to check the existence of that pathname using stat(). The filtered access logs that contain only valid pathnames are transmitted to a PC for editing via ftp. Since permissions that can be granted for files and directories differ, we also have to check whether the pathname is a directory or not so that policy editor can show different menus of permissions depending on the type of pathname.

5.3. Policy Editor

The policy editor consists of three functions ("Appending or deleting domains", "Granting ACLs for domain" and "Converting to policy files for individual MAC implementations").

"Appending or deleting domains" is used for appending execute permission of specific programs that were not audited accidentally or removing execute permission of specific programs that shouldn't be allowed for the operation phase such as debuggers used for developing and examining applications.

"Granting ACLs for domain" is used for editing read/write/execute permissions and authorizing them for files and directories for each selected domain. By repeating this step for all domains, the administrator can obtain just enough policy.

"Converting to policy files for individual MAC implementations" generates policy files for specific MAC implementation from authorized ACLs.

5.3.1. Editing domain transitions using process transition history

Fig. 2 is a screenshot of domain transition editor. This program dumps domain transition in a tree-view style according to automatically generated process execution history by analyzing access history information. Each process transition history is mapped to each domain directly.

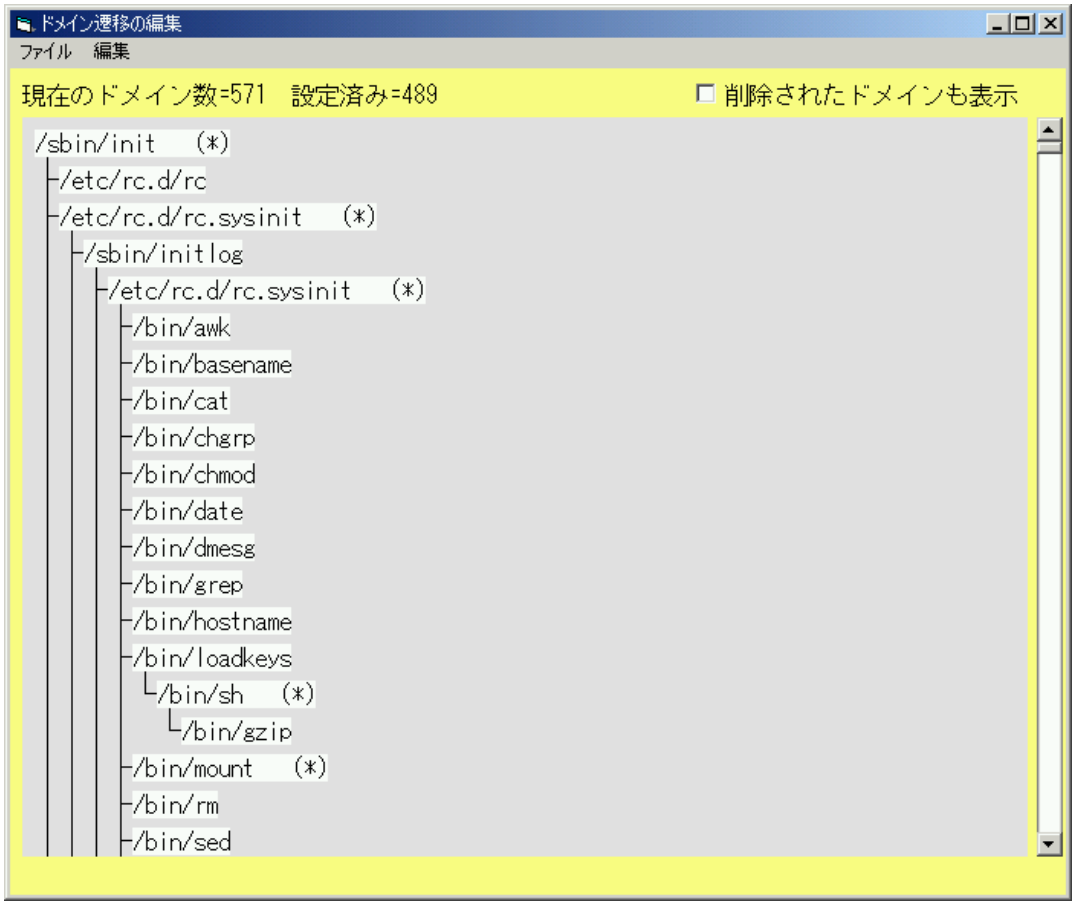

Fig. 2

In this screen, all domain transition patterns, total number of domains and number of domains whose ACLs are already configured are displayed. Choose individual domains and edit ACLs for the domain (see next section). The addition or deletion of domains corresponds to the addition or deletion of executable programs respectively. To reduce the labor of initial configuration, automatic configuration based on predefined conditions is possible.

All domains are distinguished by the process execution history starting with /sbin/init, and domain transitions are always unidirectional. For example, if /bin/tcsh is invoked by /bin/sh and /bin/sh is invoked by /bin/tcsh (that is invoked by /bin/sh), the former /bin/sh and the latter /bin/sh are distinguished as different domains, and the domain transitions never become circular.

5.3.2. Granting ACLs for individual domains

The aim of our research is to generate generic policy definition file for MAC implementations automatically on Linux. It would be possible to define the granularity of access logs based on process execution history that are used as the source of policy definition as fine-grained as the granularity of MAC implementations, but more user-friendly granularity is essential for administrators who edit and authorize policy. For extreme example, it is possible to perform the finest grained access control if access permissions are granted per a system call basis, but it won't be accepted by administrators nor there are no such implementations.

We defined the granularity of access permissions of files and directories for SELinux based on possible access permissions used by NTFS. The possible access permissions for our system are shown in Table 2.

Table 2

Files	Directories
only Read	only Read
Read/Write	Read/Write
Execute	only Create
only Scan	Allow All
Allow All	Forbid All
Forbid All	

Fig. 3 is a screenshot of access permissions editor.

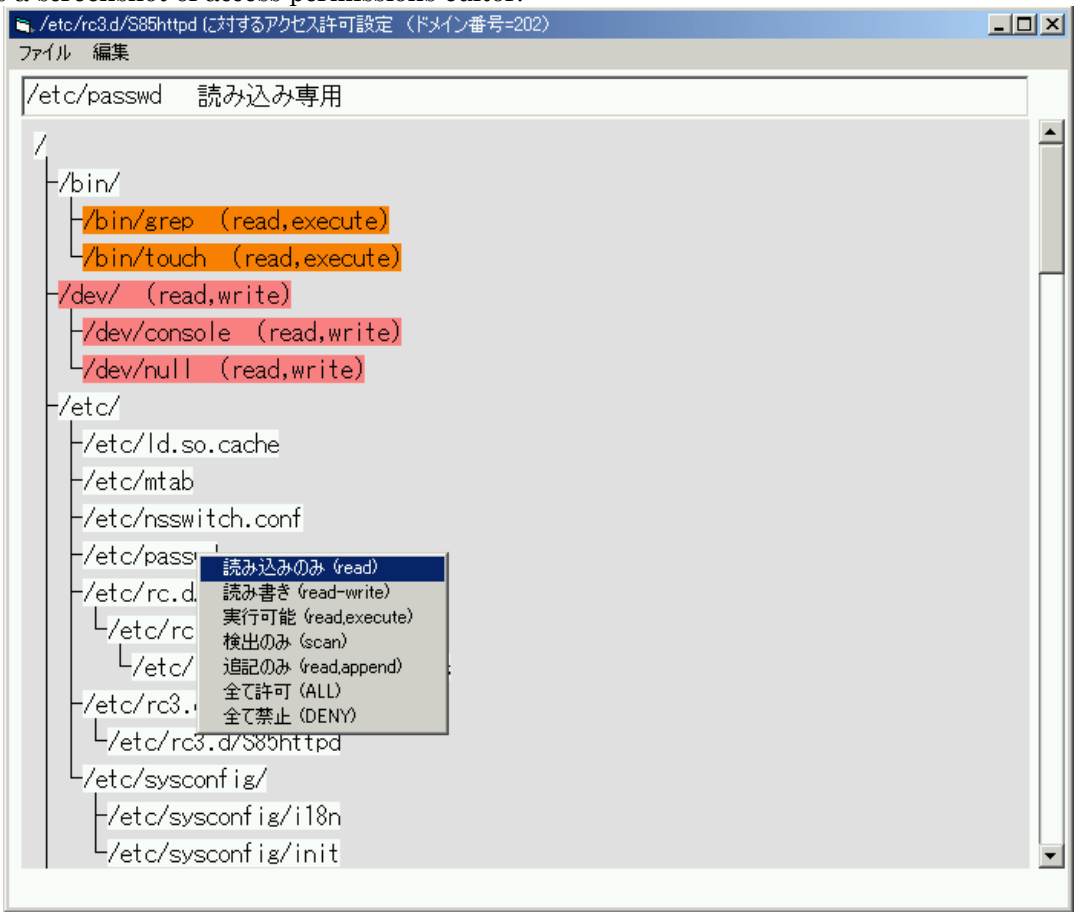

Fig. 3

The pathname of executable program whose domain is being edited is shown in the title bar. The list of files and their access permissions are shown in tree-view style starting from root directory ("/"). The current value of access permissions are shown in parentheses after filenames. The current value of access permission is read-only for filenames that has no parentheses. To change current value of access permission for files and directories, click the pathname and the possible access permissions will popup, and choose from the possible access permissions.

5.3.3. Generating the final policy

At the time of writing this paper, the policy converter cannot generate the SELinux's policy, but we are considering it's possible to generate by the following way. We show some examples of output, but since we haven't finished authorizing all access permissions for all programs, some domains cannot be converted properly (such domains are shown as file_NOT_FOUND_t). The part after "#" character is a comment part generated by the policy converter to help administrator's understanding.

First, convert the process execution history that is defined by concatenating all pathnames of each program using colon to the name of domain in SELinux. Since it is not allowed to include slashes or

colons in the name of SELinux's domain, we assign them using serial numbers.

```
type domain_0_t, domain;     #   => /sbin/init
type domain_1_t, domain;     # /sbin/init => /etc/rc.d/rc.sysinit => /sbin/initlog => /etc/rc.d/rc.sysinit => /bin/chmod
type domain_2_t, domain;     # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /bin/sed
type domain_3_t, domain;     # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /bin/sleep
type domain_4_t, domain;     # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /etc/sysconfig/network-scripts/ifup-post
type domain_5_t, domain;     # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /etc/sysconfig/network-scripts/ifup-post => /bin/basename
type domain_6_t, domain;     # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /etc/sysconfig/network-scripts/ifup-post => /bin/grep
type domain_7_t, domain;     # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /etc/sysconfig/network-scripts/ifup-post => /bin/hostname
type domain_8_t, domain;     # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /etc/sysconfig/network-scripts/ifup-post => /bin/sed
type domain_9_t, domain;     # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /etc/sysconfig/network-scripts/ifup-post => /etc/sysconfig/network-scripts/ifup-aliases
type domain_10_t, domain;    # /sbin/init => /etc/rc.d/rc => /etc/rc3.d/S10network => /sbin/initlog => /etc/sysconfig/network-scripts/ifup => /etc/sysconfig/network-scripts/ifup-post => /etc/sysconfig/network-scripts/ifup-aliases => /bin/sed
```

Next, assign labels to files and directories that are accessed by programs using serial numbers.

```
/                          system_u:object_r:file_0_t
/etc/                      system_u:object_r:file_1_t
/etc/ld.so.cache           system_u:object_r:file_2_t
/lib/                      system_u:object_r:file_3_t
/lib/i686/                 system_u:object_r:file_4_t
/lib/i686/libc.so.6        system_u:object_r:file_5_t
/lib/libacl.so.1           system_u:object_r:file_6_t
/lib/libattr.so.1          system_u:object_r:file_7_t
/lib/i686/libm.so.6        system_u:object_r:file_8_t
/lib/i686/libpthread.so.0  system_u:object_r:file_9_t
/lib/librt.so.1            system_u:object_r:file_10_t
/bin/                      system_u:object_r:file_11_t
/bin/basename              system_u:object_r:file_12_t
/bin/grep                  system_u:object_r:file_13_t
/bin/hostname              system_u:object_r:file_14_t
/bin/sed                   system_u:object_r:file_15_t
```

Next, define domain transitions.

```
# Allowed process transition from /sbin/init
domain_trans(domain_0_t, file_281_t, domain_111_t)     # /etc/rc.d/rc
domain_trans(domain_0_t, file_620_t, domain_222_t)     # /etc/rc.d/rc.sysinit
domain_trans(domain_0_t, file_NOT_FOUND_t, domain_518_t)     # /sbin/mingetty
domain_trans(domain_0_t, file_NOT_FOUND_t, domain_519_t)     # /sbin/shutdown
domain_trans(domain_0_t, file_NOT_FOUND_t, domain_521_t)     # /sbin/update

# Allowed process transition from /sbin/init:/etc/rc.d/rc
domain_trans(domain_111_t, file_58_t, domain_27_t)     # /etc/rc3.d/S12syslog
domain_trans(domain_111_t, file_77_t, domain_33_t)     # /etc/rc3.d/S13portmap
domain_trans(domain_111_t, file_87_t, domain_39_t)     # /etc/rc3.d/S14nfslock
domain_trans(domain_111_t, file_287_t, domain_47_t)     # /etc/rc3.d/S20random
    (Snipped)
domain_trans(domain_111_t, file_309_t, domain_350_t)     # /etc/rc6.d/K80random
domain_trans(domain_111_t, file_310_t, domain_358_t)     # /etc/rc6.d/K86nfslock
domain_trans(domain_111_t, file_311_t, domain_366_t)     # /etc/rc6.d/K87portmap
domain_trans(domain_111_t, file_312_t, domain_373_t)     # /etc/rc6.d/K88syslog
domain_trans(domain_111_t, file_313_t, domain_381_t)     # /etc/rc6.d/K90network
domain_trans(domain_111_t, file_62_t, domain_408_t)     # /sbin/consoletype
domain_trans(domain_111_t, file_63_t, domain_409_t)     # /sbin/initlog
domain_trans(domain_111_t, file_280_t, domain_500_t)     # /bin/egrep
domain_trans(domain_111_t, file_314_t, domain_517_t)     # /sbin/runlevel

# Allowed process transition from /sbin/init:/etc/rc.d/rc:/etc/rc3.d/S85httpd
domain_trans(domain_115_t, file_13_t, domain_116_t)     # /bin/grep
domain_trans(domain_115_t, file_54_t, domain_117_t)     # /bin/touch
domain_trans(domain_115_t, file_62_t, domain_118_t)     # /sbin/consoletype
domain_trans(domain_115_t, file_63_t, domain_119_t)     # /sbin/initlog
```

Finally, grant permissions of files and directories to domains.

```
# Allowed operations for /etc/rc3.d/S08ipchains
allow domain_547_t file_0_t:{ dir link_file } { r_dir_perms }; # /
allow domain_547_t file_11_t:{ dir link_file } { r_dir_perms }; # /bin/
allow domain_547_t file_15_t:{ file_class_set } { rx_file_perms }; # /bin/sed
allow domain_547_t file_83_t:{ file_class_set } { rx_file_perms }; # /bin/uname
allow domain_547_t file_16_t:{ dir link_file } { r_dir_perms }; # /dev/
allow domain_547_t file_17_t:{ file_class_set } { rw_file_perms }; # /dev/console
allow domain_547_t file_18_t:{ file_class_set } { rw_file_perms }; # /dev/null
allow domain_547_t file_1_t:{ dir link_file } { r_dir_perms }; # /etc/
allow domain_547_t file_55_t:{ dir link_file } { r_dir_perms }; # /etc/init.d/
allow domain_547_t file_56_t:{ file_class_set } { r_file_perms }; # /etc/init.d/functions
allow domain_547_t file_2_t:{ file_class_set } { r_file_perms }; # /etc/ld.so.cache
allow domain_547_t file_19_t:{ file_class_set } { r_file_perms }; # /etc/mtab
allow domain_547_t file_20_t:{ file_class_set } { r_file_perms }; # /etc/nsswitch.conf
allow domain_547_t file_21_t:{ file_class_set } { r_file_perms }; # /etc/passwd
allow domain_547_t file_75_t:{ dir link_file } { r_dir_perms }; # /etc/profile.d/
```

allow domain_547_t file_76_t:{ file_class_set } { r_file_perms }; # /etc/profile.d/lang.sh

6. Remaining Problems

We could confirm that it is possible to generate just enough policy definition information by collecting file access histories with process execution history at the kernel space. But we noticed that there are some problems that have to be solved when generating SELinux's policy from finally authorized file access information.

- The labels have to be assigned to files to grant permission to files. Regarding current implementation, since we assign each label to each file using serial numbers, the final size of SELinux's policy gets bloated in compensation for granting minimal necessary permissions.
- In this research, we assigned processes with domains on a one-to-one basis, but it is desirable to reduce the amount of policy by grouping multiple processes into one domain. But to achieve that, it is necessary to understand the relationship and semantic content between processes. Therefore, it is impossible to achieve that by simply processing file access information shown in this paper.
- We attempted to generate the policy for file access part that constitutes a large share of policy definition and is expected to hamper the actual operations, but the policy generated by this system doesn't take /dev and /proc into consideration and have to be merged with SELinux's default policy.

From now on, we want to support /dev and /proc and consummate automatic policy generation about file access for first, and then support automatic policy generation for other than files using this method. Also, we want to support generating policies for some other MAC implementations such as SubDomain.

7. Conclusion

In the mailing list for SELinux's developers, the way the content of SELinux's policy should be for individual programs has been discussed even nowadays. It is possible to gain higher security level if the administrator could manage MAC implementation (not limited to SELinux). But it causes an invincible barrier for the administrator that the current MAC implementation is too fine-grained to manage and their behaviors on Linux are black boxed. The administrator can't utilize them without some assistant technologies or guidance. It is easily happen that the services and programs aren't properly protected because the administrator granted unnecessary permissions or the services and programs can't behave properly because the administrator failed to grant necessary permissions.

The method shown in this paper is effective for obtaining policy semi-automatically that only allows services and applications the administrator wishes to run and forbids all unnecessary accesses the administrator doesn't wish to allow, without requiring the system administrator a lot of time and higher skill. This method solves operational issues of MAC and brings out advantages of MAC at maximum. The part of collecting file access histories with process execution history at the kernel space is implementation-independent and applicable to Linux/UNIX in general. We hope that this research helps developing countermeasures against unauthorized accesses.

8. Bibliography

[1] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.

[2] National Security Agency, *Security-Enhanced Linux*, http://www.nsa.gov/selinux/

[3] Crispin Cowan et al, *SubDomain: Parsimonious Server Security*, 14th USENIX Systems Administration Conference (LISA 2000), December 2000.

[4] Amon Ott, *The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension*,

International Linux Kongress 2001
[5] United States. Department of Defense, *TCSEC (Trusted Computer System Evaluation Criteria) DDS-2600-5502-87*, 1985
[6] Brian W. Kernighan and Rob Pike, *UNIX PROGRAMMING ENVIRONMENT,* 1985
[7] National Security Agency, *Flask: Flux Advanced Security Kernel*, http://www.cs.utah.edu/flux/fluke/html/flask.html
[8] Information-technology Promotion Agency, Japan, *A research on extending security functions of OSes* (Written in Japanese) http://www.ipa.go.jp/security/fy13/report/secure_os/secure_os.html
[9] *Introducing the ultimate secure OS easily* (Written in Japanese), Nikkei Linux 2003.5
[10] Stephen Smalley, *Configuring the SELinux Policy*, NAI Labs Report #02-007
[11] Chris Wright and Crispin Cowan et al, *Linux Security Module Framework*

9. Appendix (An example of access log produced by kernel)

(1) /bin/awk:x:/sbin/init:/etc/rc.d/rc.sysinit:/sbin/initlog:/etc/rc.d/rc.sysinit
(2) /bin/awk:x:/sbin/init:/etc/rc.d/rc:/etc/rc3.d/S10network
(3) /bin/awk:x:/sbin/init:/etc/rc.d/rc:/etc/rc3.d/S25netfs
(4) /etc/mtab:rw:/sbin/init:/etc/rc.d/rc.sysinit:/sbin/initlog:/etc/rc.d/rc.sysinit:/bin/mount
(5) /etc/mtab:r:/sbin/init:/etc/rc.d/rc:/etc/rc3.d/S25netfs:/sbin/initlog:/bin/mount

Each item is separated by a colon. The first item is the fullpath of a file or a directory that was accessed. The second item is the access mode (Read/Write/Execute). The items from third to last are the process execution history of a process that accessed the first item. The last item is the fullpath of the application that accessed the first item.

The line (1), for example, shows that the current process is /etc/rc.d/rc.sysinit and the current process was invoked by /sbin/initlog that was invoked by /etc/rc.d/rc.sysinit that was invoked by the initial process (i.e. /sbin/init), and we define the whole part (/sbin/init:/etc/rc.d/rc.sysinit:/sbin/initlog:/etc/rc.d/rc.sysinit) as a domain.

The lines from (1) to (3) access the same file to execute /bin/awk, but /bin/awk is executed in the following different domains respectively because all the items after the third field differ.

/sbin/init:/etc/rc.d/rc.sysinit:/sbin/initlog:/etc/rc.d/rc.sysinit:/bin/awk
/sbin/init:/etc/rc.d/rc:/etc/rc3.d/S10network:/bin/awk
/sbin/init:/etc/rc.d/rc:/etc/rc3.d/S25netfs:/bin/awk

The lines (4) and (5) show the situations that /bin/mount is accessing /etc/mtab. In generic way, the administrator has to grant read/write permissions of /etc/mtab to /bin/mount. But by introducing process execution history, the administrator can grant read/write permissions if accessed from the domain "/sbin/init:/etc/rc.d/rc.sysinit:/sbin/initlog:/etc/rc.d/rc.sysinit:/bin/mount" and grant only read permission if accessed from the domain "/sbin/init:/etc/rc.d/rc:/etc/rc3.d/S25netfs:/sbin/initlog:/bin/mount". This means that the administrator can grant minimal permissions depending on the context.

Notes

This is a translation of the original paper, which was written in Japanese and published in Network Security Forum 2003 held in Japan. You can obtain the original paper from the following URL.

http://sourceforge.jp/projects/tomoyo/document/nsf2003.pdf

Though the authors of this paper couldn't generate SELinux's policy after all, the technology shown in this paper was inherited by TOMOYO Linux.

TOMOYO Linux was released on November, 11, 2005. You can get more information at the following URLs.

http://tomoyo.sourceforge.jp/
http://sourceforge.jp/projects/tomoyo/