Towards a manageable Linux security

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

Abstract

The LSM and SELinux were introduced into Linux kernel 2.6 and it allows us to perform "Fine-grained Access Control". But the reality is, it is difficult to say that SELinux is fully utilized since it is so difficult to define appropriate policy for SELinux. The background of this reality is that structure of Linux and implementation of SELinux are so closely related. Authors of this paper has developed "Tamper-free Linux" and "TOMOYO Linux" aiming manageable and secure Linux, and attended at Security Stadium 2004 on the defense side to test the effectiveness of these approaches. This paper describes the followings.
What kind of threats are there in the standard Linux, and how does SELinux solve them?
Why it is difficult to operate Linux with SELinux?
Why and how did the authors of this paper develop original Linux?
What did the authors learned from the Security Stadium?
What is needed for Linux's security enhancement?

1. Introduction

To improve the Linux's security, the authors of this paper (hereafter, we) have developed (code name) "SAKURA Linux"[1] that protects from tampering physically by storing almost all files in a read-only medium and (authorized name) "TOMOYO Linux"[2] that has automatic policy generation technology to avoid practical difficulty of appropriate policy definition; and published their implementations in Linux Conference 2003 and Linux Conference 2004 (hereafter, LC2004) respectively. They are developed aiming to "Protect from tampering for sure, without entailing the policy managements upon administrators." and "Provide manageable MAC for administrators." respectively. During the past one year after LC2004, many changes and enhancements are added. The functions in SAKURA Linux have incorporated into TOMOYO Linux. Administrators can choose functions they want to enable via kernel boot option. TOMOYO Linux started to support 2.6 kernels. We attended at Security Stadium 2004 (hereafter, SS2004)[3] hosted by NPO Japan Network Security Association held on November 2004 on the defense side to test the effectiveness of our approaches; with all MACs enabled including functions added after LC2004.

At first, SELinux [4] [5] developed by NSA was provided in the form of patches to standard Linux kernels. But LSM (Linux Security Modules) [6] (a generic framework to expand Linux kernels) was invented and SELinux was rewritten using LSM, and SELinux is officially incorporated into kernel 2.6 on December 2004. And now, Linux users can enable SELinux by just choosing SELinux in the kernel configuration options.

Although SELinux and TOMOYO Linux differ in their approaches and implementations, both are aiming to enhance Linux's security. This paper arranges the actualities and challenges of enhancing Linux's security by the concepts and policy specifications of SELinux and TOMOYO Linux.

2. The needs to enhance the security at OS level

The application level programs that are used for (for example) reading mails, writing documents are implemented in the combination of functions supplied by the OSes. In Linux, functions by the kernel are supplied in the form of system calls. Normally, applications don't use system calls directly; applications use system calls through libraries such as libc that provide common functions. The OS merely processes the request whenever the OS receives a request from applications or libraries, and the OS has no concern with "What services in total are individual functions provided for?" or "What's the meaning and the purpose of individual requests?" Therefore, the OS can't know if the control of a process has been hijacked due to (for example) buffer overflow. The OS passively acquiesces in the cracker's decision and cooperates in attacks on the system. Since Linux is based on a model that grants all privileges to the "root" user, it becomes a fatal problem if a process with the "root" privileges has been hijacked.

To confine such risk, it is effective to "pick out unnecessary functions and limit the execution of such functions". One of the achievements of researches since 1980's to implement this control is the MAC (Mandatory Access Control). The MAC is defined at (for example) "Class B1: Labeled Security Protection" of "Division B: Mandatory Protection" in DOD 5200.28-STD (TCSEC [7]).

3. The meaning of introducing MAC

The MAC permits the execution of individual requests only when the request is explicitly granted by the policy that is predefined to tell the OS what actions are permitted and what actions aren't. It is possible to regard that the MAC is a mechanism that is capable of "preventing the system from being hijacked completely" and "limiting and localizing the damage caused by the deprivation of privileges" by appending restrictions so that "the OS can perform only functions that are necessary to the specific purpose" to the OS that was "originally made available for generic purpose". Since the MAC is nothing but the mechanism to restrict accesses according to the policy defined by the administrator, "whether the administrator can fully utilize the MAC" depends on "whether the administrator can define APPROPRIATE policy". Introducing the MAC is a method of improving security, but you need to be careful that introducing the MAC itself doesn't improve security.

4. The merits of "Physical Tamper Protection" and its limitation

The efficacy of MAC is also demonstrated in security enhanced commercial OSes called Trusted OSes, but the MAC entails the costs of policy managements. Therefore, it is too laborious to introduce OSes with MAC for servers transmitting public information and personal WWW servers. Therefore, we have developed (code name) SAKURA Linux and published its implementations [1] in 2003. SAKURA Linux stores almost all files in the system into a read-only medium so that these files are physically protected from tampering without entailing the costs of policy managements. SAKURA Linux patches a little to the 2.4 kernels.

- Keep the content of the root fs in a medium (such as DVD-R, USB flash memory) that is physically write-protected.
- Limit the execution of "mount", "chroot" and "pivot_root".
- Forcefully apply "noexec" option to partitions that are writable (such as /tmp) so that malicious applications can't be executed.

SAKURA Linux doesn't have policy files like SELinux. But it is possible to regard that a kind of MAC that physically prevents tampering is implemented on the system, for the separation of root fs and writable partitions corresponds to implicit policy (though it's very rough-grained). Nobody can tamper with files stored in a physically read-only medium, but if one can shadow files by mounting arbitrary partitions on the root fs, the physical protection become useless. Therefore, we patched the kernels to restrict mount operations. In SAKURA Linux, files such as kernels, application programs, shared libraries and static WWW contents are physically protected even if the root privilege has been deprived, and they are never tampered with. The feature of SAKURA Linux is that the administrator needn't to manage policy files by limiting the purpose and the efficacy, though it's impossible to perform fine-grained access control. We made a presentation in SS2004 in the defense side to examine the efficacy of SAKURA Linux. Nobody could tamper with files directly, but we received the following attacks.

- A shell with root privilege was invoked by attacking the vulnerability of Samba server.
- The processes of httpd and sshd were killed.
- A fake httpd was executed.
- Some device files in /dev directory were deleted.

In SAKURA Linux, the MAC like SELinux is not implemented, and the kernel is almost the same as normal Linux kernel expect for the restriction of mount operations. Therefore, the OS passively accepts, like the normal Linux, "administrative operations" including rebooting the system if root privilege has been deprived, though the attacker still can't tamper with files. We forced the "noexec" option for writable partitions such as /tmp to prevent execution of malicious applications installed by the attacker, but the attacker wrote a tiny WWW server program using Java language and compiled and executed the fake WWW server. (The server had Java compiler and interpreter to provide Tomcat service.) As a result, the regrettable state that is "providing fake WWW contents to clients, though the WWW contents aren't tampered with" was realized. Such state will happen if the server has interpreters such as (for example) Perl or Ruby, and this threat is inevitable as long as the system depends on only physical tamper protection. Also, since tty device files in /dev directory were deleted by the attacker, the legal root couldn't log into the system from the console. (Nor using ssh, for the cracker killed sshd.) Everything, not limited to the MAC, which are implemented on the software come with bugs and vulnerability. Seen in that light, we still believe the efficacy of the physical tamper protection. But after the experience of SS2004, we recognized that to apply Linux that requires higher levels of security, the physical tamper protection is not enough, and the introduction of MAC is premised.

5. SELinux

The most well-known implementation of MAC in Linux is SELinux developed and released by NSA. SELinux implements two security models, TE (Type Enforcement) and RBAC (Role-Based Access Control), on Linux, and allows flexible and fine-grained access controls. SELinux was rewritten using LSM and was incorporated into Linux 2.6 kernels, and the barrier of introduction has disappeared. SELinux made it possible to "keep the compatibility with the existent applications" and "perform fine-grained access control with minimal penalty in performance", but there still remains the big challenge of "how to define APPROPRIATE policy".

5.1. The reality of introducing SELinux

There was a discussion focusing "How to obtain APPROPRIATE policy" in the SELinux Symposium [8] held on March 2005 in Maryland, USA. The Red Hat, Inc. (the first distributor who has introduced SELinux enabled by default, and aggressive about incorporation of SELinux) showed an opinion "Users who can't understand SELinux's policy should not touch SELinux's policy". In fact, the "targeted policy" (policy that protects partially, not the entire system) is installed and activated in Fedora Core 3 if the user enables SELinux. But the source of "targeted policy" (the definition files) is a separated RPM package and not visible to users unless the user explicitly installs that package.

In SELinux, there is a policy template called "default policy" distributed by the distributors. The default policy of SELinux is built up with a set of text files worth 30 thousands of lines. The way the SELinux should be is that "the users understand the default policy perfectly" and "the users customize the domain definitions depending on their needs" and "the users grant permissions to each domain". But it premises the detailed and precise understanding of behaviors of the OS and the applications [9][10][11]. Unfortunately, it is a tough job for almost all users. Therefore, the SELinux that provides fine-grained access control is used with looser policy made by patching to the default policy. We would like to look at the cause for this circumstance, starting from ordering the concepts of access controls in SELinux.

5.2. The concepts of SELinux

The SELinux performs access controls using concepts called "domain", "type", "context", "label", and "role" [10]. The following are the shortest description of these concepts.

In SELinux, the administrator can define "access vectors" in high granularity against objects ("object class") such as files, directories, and sockets. The access vectors are not fixated statically; they are usually associated dynamically depending on the context such as the combination of the process and the objects that the process accesses. The SELinux distinguishes these contexts so that the administrator can grant different access vectors depending on their contexts, and introduces the concept called "context" ("security context"). A "context" consists of the user, the role the user is playing, and the nameplate called "type" assigned to objects. The "type" in a "context" assigned to processes is called "domain", and the "type" in a "context" assigned to all objects other than processes is called "label". From the point of view of a process, the "domain" is nothing but a "type" that the process itself belongs to. But since the "domain" is divided according to the zones the administrator wishes to switch access vectors, the "domain" is a set of access vectors from the point of view of the administrator. Therefore, you can regard that "domain" is defined first and then assigned to processes depending on their needs. The "role" in a "context" means the role the user is playing. In the SELinux enabled systems, all users must play at least one role (you can see your role by "id -Z" or "id --context"), and it is possible to allow users play multiple roles. The "role" is considered as a set of "domains"; the user can choose any "domain" from all "domains" assigned by the "role" assigned to the user using "newrole" command to perform the user's task.

5.3. Why it is difficult to define SELinux's policy?

The first hurdle in defining SELinux's policy is to understand the SELinux's concepts described above, but that is not the highest hurdle. We consider the cause of difficulty of policy definition is due to the following three points.

- It is difficult to figure the system's behavior out in the granularity of access vectors, for the access vectors are too fine-grained. There are 13 permissions, for example "r_file_perms", applicable to files. (Moreover the 13 permissions aren't the primitive in SELinux; they are macros to help understanding.)
- The granularity of domain definition is user-defined. In SELinux, access vectors are granted to domains. It is possible to improve security by dividing domains more precisely to grant minimal access vectors. But there is no global and explicit agreement on the rule how the domains should be divided.
- Access vectors have to use labels; the administrator can't use filenames or directory names directly. In the kernel space, files and directories are managed using "i-node" object. The access controls by SELinux is performed using the type assigned to "i-node" object (i.e. label). Therefore, the label definition (i.e. associating labels with pathnames) is required and has to be maintained. But there is no global and explicit agreement on the rule how the labels should be defined.

6. The automatic policy generation system and its challenge

The policy for MAC is the key of improving Linux's security. We considered that "Why not let the kernel generate policy automatically, for it is difficult to figure out and generate policy manually?" and developed "Access policy generation system based on process execution history"[12] and published in Network Security Forum 2003 (hereafter, NSF2003). The basic idea was so simple. "If you need to control access at the kernel space, isn't it possible to generate policy for MAC by monitoring and memorizing access requests at the kernel space?" We demonstrated at the NSF2003 that it is possible to generate SubDomain-like policy files by monitoring the file accesses with the finest granularity of domain division (we used process's program invocation history starting from /sbin/init for domain definition), but we couldn't generate SELinux's policy that was the original purpose of the development of this system.

6.1. The domain definition

The SELinux's domains are flat and non-hierarchical, and transit to other domain on the invocation of only specific programs defined in the policy. Therefore, the way of domain divisions is underspecified, even if the behavior of the system that is to be restricted by the policy is completely figured out. It is impossible to automate the policy definition whose granularity of domain division is underspecified. The polgen [13] developed by MITRE uses strace to automate policy definition, but the way how to associate the information obtained from strace with SELinux's domain is entrusted to the administrators, therefore, the information obtained from strace is used only for as a guide of policy definition. The way we published in NSF2003 used the string concatenated program invocation history for domain definition like the output from pstree. Fig. 1 shows two domains distinguished by their ancestors, and monitors file accesses separately, though both processes are executing /bin/mount.

/sbin/init /etc/rc.d/rc.sysinit /sbin/initlog /bin/mount

(Domain for /bin/mount invoked by /sbin/initlog invoked by /etc/rc.d/rc.sysinit invoked by /sbin/init.)

/sbin/init /etc/rc.d/rc /etc/rc3.d/S25netfs /sbin/initlog /bin/mount

(Domain for /bin/mount invoked by /sbin/initlog invoked by /etc/rc3.d/S25netfs invoked by /etc/rc.d/rc invoked by /sbin/init.)

Fig. 1 An example of domain definition in "Access policy generation system based on process execution history"

The domain definition in "Access policy generation system based on process execution history" is equivalent to "divide domains whenever a program is invoked and transit to different domain whenever a program is invoked" in SELinux. It will be possible to define such policy; but considering the granularity of SELinux's access permissions, it is unrealistic to force the administrator manage so many domains; and even if it is possible, the penalty to the performance will become larger due to the bloated policy.

6.2. The granularity of access permissions

The granularity of access permissions of SELinux is based on the granularity of LSM (the underlying framework). Since LSM hooks the system calls in the kernel space, access permissions are defined according to the internal implementation of kernel. It is the inconsistency that the policy has to be defined considering the internal implementation of kernel although the userland applications needn't to consider the internal implementation of kernel during their development. This inconsistency is inevitable as long as MAC is implemented using the current LSM specifications.

TOMOYO Linux didn't use LSM, just independently patched to some system calls, to achieve automatic policy generation and MAC based on the automatically generated policy as its symmetry. We defined the granularity of access permission as read/write/execute and use bitwise-OR'ed numerical value so that standard Linux administrators can understand and control completely. The syntax of TOMOYO Linux's policy is obvious to all Linux users, and easy to edit the policy.

6.3. The label definition

We considered that it is essential to allow administrators intuitive policy management that the administrators can grant permissions using filenames and directory names in the policy files. To allow using pathnames in automatic policy generation and MAC using the automatically generated policy, we introduced inverse pathname resolution process to convert from requested pathnames to canonicalized absolute pathnames that don't have symbolic links. (See the paper [2] for actual implementations.)

7. TOMOYO Linux

The behavior of the MAC enabled Linux is controlled by the policy. Therefore, the policy specification mirrors the MAC enabled Linux's functionality, and the readability of the policy directly mirrors the MAC enabled Linux's operability. In this chapter, we describe the policy specification of TOMOYO Linux as of April 2005. Anybody who can understand this specification will be able to master TOMOYO Linux. Portions of this specification were extended after the demonstration at LC2004 according to the needs we realized through introducing TOMOYO Linux into actual systems.

7.1. The concepts of TOMOYO Linux

We will describe the concepts of TOMOYO Linux like SELinux below.

Every process always belongs to single domain. Divide domains and transit domains whenever a process executes a program. Restrict the resources that the domain can access on the granularity of read/write/execute, using filenames and directory names. The domain division is performed automatically by the kernel. The policy is generated automatically if the user commands to do so.

7.2. The domain definition

In TOMOYO Linux, permissions are granted to domains, not to processes. Therefore, the policy of TOMOYO Linux doesn't have the concept of the process owner (i.e. user id). The initial domain is "<kernel>", which the kernel belongs to. The rest of all domains are defined as the string concatenated program invocation history. For example, the domain for /sbin/init is "<kernel> /sbin/init" because /sbin/init is invoked by the kernel. The domain for /etc/rc.d/rc is "<kernel> /sbin/init /etc/rc.d/rc" because /etc/rc.d/rc is invoked by /sbin/init and /sbin/init is invoked by the kernel. All pathnames are canonicalized (i.e. a pathname starts with "/" and doesn't contain "//", "/../", "/./", all symbolic links are resolved, and consists of ASCII printable characters without white spaces). This simple rule allows defining all domains for all processes in only one clearly defined path.

7.3. The structure of policy files

The following policy files are used by TOMOYO Linux. You needn't to create all of them. You can create only files that correspond to MAC you want to enable.

(1) policy.txt
Define all domains, and also define access permissions for files. See 7.4.

(2) allow_bind.txt
Define access permissions for localhost's network port numbers that the domain can bind to. See 7.5.

(3) cap_policy.txt
Define access permissions for capabilities that the domain can use. See 7.6.

(4) authorized.txt
Define domains which the three types of MAC listed above are not applied to. See 7.7.

(5) initializer.txt
Define programs that reset the history of program invocation history. See 7.8.

(6) allow_read.txt
Define files that are readable to all domains.

(7) chroot.txt
Define directories that are allowed to chroot to.

(8) mount.txt
Define the combinations of partition, mount point, filesystem and options that are allowed to mount.

(9) noumount.txt

Define directories that are not allowed to unmount.

(10)syaoran.conf

Define the combinations of device files and their attributes that are allowed to be created. See 7.10.

7.4. MAC for files

Access permissions for files are given in the form of the combination of access modes (numericalized permission) and canonicalized pathnames followed on a domain definition. Lines between the line just after a definition of a domain and the line just before a definition of next domain are access permissions granted to the domain. File types are only directory or non-directory. The directory always ends with "/" and the non-directory never ends with "/". Wildcards are accepted for read and/or write permissions. An example is shown in Fig. 2.

```
<kernel>
1 /sbin/init
<kernel> /sbin/init
6 /dev/console
6 /dev/initctl
6 /dev/tty\$
4 /etc/inittab
6 /etc/ioctl.save
4 /etc/localtime
1 /etc/rc.d/rc
1 /etc/rc.d/rc.sysinit
1 /sbin/mingetty
1 /sbin/shutdown
2 /var/log/wtmp
6 /var/run/utmp
```

The kernel can do the following.

- Execute /sbin/init (--x)

The /sbin/init invoked by the kernel can do the following.

- Read/write /dev/console (rw-)
- Read/write /dev/initctl (rw-)
- Read/write /dev/tty[0-9]* (rw-)
- Read /etc/inittab (r--)
- Read/write /etc/ioctl.save (rw-)
- Read /etc/localtime (r--)
- Execute /etc/rc.d/rc (--x)
- Execute /etc/rc.d/rc.sysinit (--x)
- Execute /sbin/mingetty (--x)
- Execute /sbin/shutdown (--x)
- Write /var/log/wtmp (-w-)
- Read/write /var/run/utmp (rw-)

Fig. 2 An example policy for files (policy.txt)

Wildcards are ignored for execute permissions, due to the following reasons.

- Wildcards may break the domain transition rule defined in only one clearly defined path, resulting unintended domain transition.
- It makes difficult to display domain transition tree using policy editors if one execute permission can transit to multiple domains.
- Careless oversight will happen when an administrator reviews the domain transitions using policy editors.

Read permission is not checked when executing a program, due to the following reasons.

- Read permission checks are done by DAC.
- The content of programs (especially, script programs) shouldn't be visible to users, for the cracker attempts to analyze programs to crack them.

7.5. MAC for local ports

This policy file restricts localhost's network port numbers for TCP and UDP protocols that a domain can bind to.

An example is shown in Fig. 3. The combinations of protocol and port number are listed in the form of <protocol>-<port> followed on a domain definition. Port 0 means arbitrary port. Domain can bind to multiple ports. This function is implemented after LC2004.

<kernel> /sbin/portmap UDP-0 TCP-0	・/sbin/portmap can bind to arbitrary TCP and UDP ports.
<kernel> /usr/sbin/dhcpd UDP-67	・/usr/sbin/dhcpd can bind to UDP port 67.
<kernel> /usr/sbin/httpd TCP-443 TCP-80	・/usr/sbin/httpd can bind to TCP port 443 and TCP port 80.

Fig. 3 An example policy for local ports (allow_bind.txt)

7.6. MAC for capabilities

We implemented MAC for the following system calls after LC2004.

- socket(PF_INET or PF_INET6, SOCK_STREAM, *), socket(PF_INET, SOCK_DGRAM, *), socket(PF_INET, SOCK_RAW, *), socket(PF_ROUTE, *, *), socket(PF_PACKET, *, *)
- listen() for PF_INET or PF_INET6, SOCK_STREAM
- connect() for PF_INET or PF_INET6, SOCK_STREAM
- sys_mount(), sys_umount(), sys_reboot(), sys_chroot(), sys_kill(), sys_tkill(), sys_vhangup(), do_settimeofday(), sys_adjtimex(), sys_nice(), sys_setpriority(), sys_sethostname(), sys_setdomainname()

TOMOYO Linux didn't hook "CAP_*", the Linux's standard capability. The reason is that since the types of capabilities are few, one capability (for example, CAP_SYS_ADMIN) is used in so many purposes, and it's too rough-grained to control with the granularity of system calls. The capability checks in TOMOYO Linux and the ones in standard Linux capability are performed independently. An example policy is shown in Fig. 4.

```
<kernel> /sbin/mingetty /bin/login /bin/tcsh /usr/bin/ssh
inet_tcp_create inet_tcp_connect use_inet_udp
<kernel> /sbin/portmap
inet_tcp_create inet_tcp_listen use_inet_udp
<kernel> /usr/sbin/httpd
inet_tcp_create inet_tcp_listen use_route SYS_KILL
```

Fig. 4 An example policy for capabilities（capability.txt）

7.7. Trusted domains

There are often cases during the policy definition or in the production state that administrators want to run arbitrary commands or edit arbitrary files while MAC is enabled. It doesn't improve operating efficiency to disable MAC or appending policy for administrating operations everytime the administrator want to do. Therefore, we introduced domains that are controlled only by Linux's DAC while the MAC is enabled. For example, during the policy definition, by specifying shells invoked by sshd as trusted, the administrator will be able to edit policy effectively. An example of trusted domain is shown in Fig. 5.

<kernel> /usr/sbin/sshd /bin/tcsh	・Trust /bin/tcsh invoked by /usr/sbin/sshd
<kernel> /sbin/mingetty /bin/tcsh	・Trust /bin/tcsh invoked by /sbin/mingetty

Fig. 5 An example policy for trusted domains (authorized.txt)

A typical usage of trusted domain is the package managements. For example, when administrator runs rpm command to update an application, many commands are executed and many files are accessed, but the administrator can't know what commands are executed and what files are accessed prior to the invocation of rpm command. Administrators can use this trusted domain to execute rpm command. You need to be very careful allowing trusted domains, for this is an exception of MAC. This function is implemented to respond to the needs realized while introducing TOMOYO Linux into actual systems.

7.8. Domain transition exceptions

Some administrators remotely restart daemon programs for maintenance purpose, while daemon programs are usually configured to start automatically on bootup. Since TOMOYO Linux distinguishes domains by their ancestors, the daemon programs will belong to different domains if restarted. But it is desirable to identify the domain for automatic invocation with the domain for manual invocation because these domains are used by the same daemon program. Therefore, we introduced the "domain transition exception" function, which allows to transit to a domain just below the "<kernel>" domain. This function is used to aggregate multiple processes' program invocation histories. An example policy for /usr/sbin/httpd is shown in Fig. 7. (Files other than programs are omitted.) This function is implemented to respond to the needs realized while introducing TOMOYO Linux into actual systems.

```
/usr/sbin/httpd
```

Fig. 6 An example policy for domain transition exception (initializer.txt)

```
<kernel> /init /sbin/init /etc/rc.d/rc /etc/rc.d/init.d/httpd /sbin/initlog
1 /usr/sbin/httpd
<kernel> /sbin/mingetty /bin/login /bin/tcsh /sbin/service /bin/env /etc/rc.d/init.d/httpd /sbin/initlog
1 /usr/sbin/httpd
<kernel> /usr/sbin/httpd
1 /var/www/cgi-bin/sessioncookie.cgi
```

Fig. 7 An example policy for files using domain transition exception (policy.txt)

7.9. Event-driven domains

We found that domains for some programs such as /sbin/hotplug and /sbin/modprobe that are invoked by the kernel on demand cannot be defined in only one clear path. Since these programs are invoked on demand depending on the operating conditions of the system, there are possibilities of invocation failures due to the lack of access permissions if the system is operated using only automatically generated policy. Therefore, we introduced "event-driven domain" function, so that the fork'ed process will transit to the kernel domain when the kernel fork's the current process to invoke these programs. (The fork'ed process will belong to a domain just under the "<kernel>" domain by the subsequent program invocation by the kernel.) This made automatic policy generation of these programs invoked on demand possible, and made the system operation easier.

7.10. Protection for device files

At the point of LC2004, TOMOYO Linux distinguished only directory and non-directory, and performed MAC based on canonicalized pathnames. Therefore, if a domain has a write permission to write to a pathname, the domain can also delete that pathname and recreate. If the recreated pathname is a device file (which means the cracker has hijacked a process and commanded to do so), the content of HDD might be destroyed at a blow. To avoid this threat, we implemented an original filesystem with the following features based on tmpfs.

- Restricts the combination of pathnames and their attributes (one of directory, FIFO, UNIX domain socket, character device, block device, symbolic link, including major and minor numbers if character or block device) that can be created.

- Restricts operation on files such as deleting and changing attributes (owner, group, permission) according to the policy.
- Picks up only device files that are actually opened in the accept mode.

By mounting this filesystem on /dev, you can ensure that /dev/null (for example) is a character device file with major=1 and minor=3. Also, you can forbid changing the owner or group or permission of /dev/null.

8. Advanced Usage of TOMOYO Linux

You can use the following techniques by making use of the TOMOYO Linux's domain transition rules.

8.1. Login Authentication Multiplexing

A login authentication is performed when a user logs into the system. But the login authentication has the following vulnerability. One is that the secrecy of passwords may be broken due to (for example) dictionary attack. The other is that the authentication itself is bypassed due to vulnerability of authentication program such as buffer overflow. The MAC can solve these vulnerability simply and effectively. We will describe an example of login authentication multiplexing with customizable authentication programs using TOMOYO Linux's policy.

An example policy is shown in Fig. 8. An example authentication program is shown in Fig. 9.

```
<kernel> /usr/sbin/sshd /bin/bash
1 /bin/auth1
<kernel> /usr/sbin/sshd /bin/bash /bin/auth1
1 /bin/bash
<kernel> /usr/sbin/sshd /bin/bash /bin/auth1 /bin/bash
1 /bin/auth2
<kernel> /usr/sbin/sshd /bin/bash /bin/auth1 /bin/bash /bin/auth2
1 /bin/bash
<kernel> /usr/sbin/sshd /bin/bash /bin/auth1 /bin/bash /bin/auth2 /bin/bash
1 /bin/cat
1 /bin/vi
```

Fig. 8 An example policy for files to perform extra login authentication (policy.txt)

```
#! /bin/sh
for i in 1 2 3
do
  read -r -s -p 'Password: ' passwd
  echo
  [ "$passwd" = "SAKURA" ] && exec $SHELL
done
echo 'Incorrect password.'
```

Fig. 9 A simple authentication program

```
#! /bin/sh
for i in 1 2 3
do
  read -r -s -p 'Password: ' passwd
  echo
  [ -f /data/rootauth ] && exec $SHELL
done
echo 'Incorrect password.'
```

Fig. 10 An authentication program that doesn't depend on passwords

/bin/auth1 and /bin/auth2 are the programs added for performing extra authentications. The first domain is for /bin/bash invoked by /usr/sbin/sshd and has execute permission of /bin/auth1. The duty of /bin/auth1 is to authenticate the user and invoke the next program (i.e. /bin/bash). Therefore, /bin/auth1 needs only permissions to execute /bin/bash. In the same manner, /bin/bash is invoked after the authentication at /bin/auth2 has succeeded, and actual operations (such as executing /bin/cat or /bin/vi) starts. You can use authentication programs with time synchronized passwords something like RSA's SecurID. The elements available for authentication are not limited to passwords. You can use the existence of specific files shown in Fig. 10 and the content of specific files as passwords. You can create authentication programs in the same manner of developing normal application programs. The possible combinations of elements are infinite. See the paper [14] for more details.

8.2. RBAC

You can divide root privilege on the systems that supports RBAC (Role-Based Access Control) like SELinux. But the syntax of policy becomes more complicated because the element "role" is added. TOMOYO Linux doesn't support RBAC because TOMOYO Linux doesn't handle the concept of users in its policy syntax. But just a little ingenuity allows administrators to use RBAC-like privilege division. The basic idea is the same described in 8.1. Divide domains by invoking arbitrary programs (for example, shells) instead of authentication programs, and grant minimal permissions to each domain. An example of delegating administrator's jobs is shown in Fig. 11.

```
<kernel> /usr/sbin/sshd /bin/bash
1 /bin/bash
1 /bin/tcsh
1 /bin/zsh
<kernel> /usr/sbin/sshd /bin/bash /bin/bash
(List only the ACLs that are needed to maintain WWW service.)
<kernel> /usr/sbin/sshd /bin/bash /bin/tcsh
(List only the ACLs that are needed to maintain mail service.)
<kernel> /usr/sbin/sshd /bin/bash /bin/zsh
(List only the ACLs that are needed to maintain accounts.)
```

Fig. 11 An example policy for files to perform administrative task delegation (policy.txt)

The first domain is for /bin/bash invoked by /usr/sbin/sshd and has execute permission of /bin/bash and /bin/tcsh and /bin/zsh. When the user invokes /bin/bash, the user will transit to "<kernel> /usr/sbin/sshd /bin/bash /bin/bash" domain. And grant permissions necessary for maintaining WWW service orienting from this domain. Of course, you needn't to grant manually. You just operate as you want in accept mode. When the user invokes /bin/tcsh, the user will transit to "<kernel> /usr/sbin/sshd /bin/bash /bin/tcsh" domain. And grant permissions necessary for maintaining mail service orienting from this domain. In the same way, you can do it with /bin/zsh.

9. Conclusion

The security enhancements at the OS level, not limited to Linux, retrofit a mechanism to restrict the versatility of OSes while OSes are originally made available for generic purpose. Since the mechanism is retrofitted, not built-in, it produces distortion, and the distortion reflects the policy description. SELinux is frequently referred as "Providing flexible and fine-grained MAC". But we think it is a biased opinion. We think it is also possible to refer SELinux as "Requiring policy definition at the level of system calls" due to convenience of SELinux's implementation. It is possible, at least in principle, to realize everything using assembly language that can be realized using high-level language. In the same manner, it will be possible to achieve a manageable Linux security using SELinux, but some breakthroughs of topics mentioned in this paper are needed to bring SELinux into practice. For example, the situation will get better greatly if there were mechanisms that analysis the side effects of appending policy when a policy violation occurs or mechanisms that automates splitting existing domains. It would be nice if the package management system can negotiate policy that a package needs by embedding access vector information into the package; this will dramatically reduce the labor of administrators and come to appropriate policy.

Suppose SELinux realizes what the access controls should be on Linux coherently, our TOMOYO Linux is a result of attempts for keeping Linux not to look for trouble while being aware of usability from the beginning and keeping operability through a trial and error process. The procedure for releasing the source codes of TOMOYO Linux is now in progress. We do hope that the usability and the simplicity and the implementations of TOMOYO Linux provide the possibility of improvement in SELinux and all other security enhanced OSes, and the day people can master at ease Linux and all other OSes comes true.

Acknowledgment: We were supported from the technological study to implementation of TOMOYO Linux by Tetsuo Handa, NTT DATA CUSTOMER SERVICE CORPORATION. We would like to thank Mr. Handa.

Bibliography

[1] Toshiharu HARADA, Takashi HORIE and Kazuo TANAKA, "Security Advancement Know-how Upon Read-only Approach for Linux." Linux Conference 2003, http://sourceforge.jp/projects/tomoyo/document/lc2003-en.pdf

[2] Toshiharu HARADA, Takashi HORIE and Kazuo TANAKA, "Task Oriented Management Obviates Your Onus on Linux." Linux Conference 2004, http://sourceforge.jp/projects/tomoyo/document/lc2004-en.pdf

[3] Security Stadium 2004 (Written in Japanese) http://www.jnsa.org/active/press/vol12pdf/4_report4.pdf

[4] http://www.nsa.gov/selinux/

[5] Fedora Core 3 SELinux FAQ, http://fedora.redhat.com/docs/selinux-faq-fc3/index.html

[6] http://lsm.immunix.org/

[7] Department of Defense, *Trusted Computer System Evaluation Criteria*, December 1985

[8] SELinux Symposium, http://www.selinux-symposium.org/

[9] Stephen Smalley and Timothy Fraser. *A Security Policy Configuration for the Security-Enhanced Linux*, February 2001

[10] Stephen Smalley, *Configuring the SELinux Policy*, February 2005.

[11] Michelle J. Gosselin, Jennifer Schommer, *Confining the Apache We Server with Security-Enhanced Linux*

[12] Toshiharu HARADA, Takashi HORIE and Kazuo TANAKA, "Access policy generation system based on process execution history" Network Security Forum 2003, http://sourceforge.jp/projects/tomoyo/document/nsf2003-en.pdf

[13] MITRE – Security-Enhanced Linux, http://www.mitre.org/tech/selinux/

[14] Toshiharu Harada and Takaaki Matsumoto, "Chained Enforceable Re-authentication Barrier Ensures Really Unbreakable Security" Workshop on Informatics 2005, http://sourceforge.jp/projects/tomoyo/document/winf2005-en.pdf

Notes

This is a translation of the original paper, which was written in Japanese and published in Linux Conference 2005 held in Japan. You can obtain the original paper from the following URL.

http://sourceforge.jp/projects/tomoyo/document/lc2005.pdf

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/

The policy syntax of TOMOYO Linux has changed after Linux Conference 2005. Please refer the following URLs.

http://tomoyo.sourceforge.jp/en/doc/policy-reference.html
http://tomoyo.sourceforge.jp/en/doc/policy-syaoran.html