Security Advancement Know-how Upon Read-only Approach for Linux.

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

Abstract

It is effective for improving the security of computer systems to introduce security enhanced OSes, but entails a new burden of managing appropriate policy to control access appropriately. This paper describes a method and know-how that prevents files from being tampered with, without entailing policy management, upon read-only approach. This method is suitable for WWW servers that deal with not-so-frequently-updated information such as software and documentations. This paper also suggests a tiny kernel patch that reduces the damage in case the system has been hijacked.

1. Introduction

The fact that it is impossible to ensure computer system's security with only measures of application level is getting to be recognized [1]. Linux OS, which is rewritten based on UNIX, inherited the simplicity that is the characteristics of UNIX and the too powerful root privileges and daemon programs that require the root privileges. This mechanism is known as the vulnerability of Linux's security.

Researches on requirements and implementations of security mechanisms for trustworthy computer systems have been continued since 1980's as a procurement requirement of military systems of United States. The TCSEC (Trusted Computer Systems Evaluation Criteria)[2] published by DoD (Department of Defense) in December 1985 has been widely referenced as a practical international standard of requirement of secure computer systems, though the evaluation of the appropriateness for TCSEC has finished in 1999 and new criteria has been proposed.

NSA (National Security Agency) implemented MAC (Mandatory Access Control) that is placed as a major technology in TCSEC on mainstream Linux OS and released the implementation named SELinux (Security-Enhanced Linux) [3, 4, and 5]. SELinux realized the security level that pursues commercial trustworthy OSes (Trusted OS) as open sourced software. But SELinux required development of source codes that worth one tenth of source codes of Linux kernels, and entails new burden of managing policy files with several tens of thousands of lines to administrators.

There is a simple method that can prevent tampering without entailing policy management. That is, use data that should be protected from tampering in a read-only mode. If a filesystem is mounted for read-only, even the administrator can't write to it. But if the filesystem is remounted for read-write by a cracker who deprived the administrator's privileges, the cracker can write to it. Therefore, it is not enough to just mount the filesystem for read-only. Therefore, the authors of this paper (hereafter, we) considered and realized a tamper-proof Linux server by combining the feature of read-only mounting and the medium that can provide physical write protection, without making massive modifications to Linux OS.

2. Tamper protection by mounting read-only.

2.1. The merits of read-only mounting.

The access control in standard Linux/UNIX is performed access control information called "security bits" stored on the filesystem. This access control is called DAC (Discretionary Access Control), and the owner of resources (files or directories) can freely modify the information. The Linux kernel is made to always grant access requests if the process who requested access is running with the administrator's privileges (i.e. user id = 0) regardless of the information set or modified by the owner of the resources. Therefore all files will be tampered with or deleted if the administrator's privileges are deprived by external attackers due to (for example) buffer overflow.

The MAC is located as a method that realizes access controls that are applied without exception even to the administrator. But there are methods that can prevent resources from being tampered with without introducing MAC, such as mounting filesystem for read-only or using medium that provides physical write protection.

If the system can work with the root filesystem (a partition that is mounted on "/" directory) mounted for read-only, important data such as /etc/passwd and commands such as /bin/ls and library files under /lib directories won't be illegally replaced by the legal administrator logged into the system from the local console or the cracker who penetrated and deprived the administrator's privileges into the system from network by attacking vulnerabilities. It will become possible to protect public servers such as WWW with a high security. The read-only mounting can't perform file-grained access controls like MAC that controls based on subject and object or SELinux that controls based on context, but the read-only mounting can realize stronger protection in the point of view of tamper protection.

2.2. The problems in mounting Linux's root filesystem read-only.

Since the standard Linux assumes to be installed into a medium that is writable (i.e. HDD), it is impossible to use Linux with the root filesystem mounted for read-only without modifications. To use Linux with the root filesystem mounted for read-only, the following three requirements have to be met.

(a) The Linux boots properly and shutdowns properly.

To provide services constantly, it is not enough if the Linux can boot properly but cannot shutdown properly. If writable partitions are provided by using tmpfs or loopback mounts, it is important to unmount in a correct order so that the Linux can shutdown properly.

(b) Applications runs properly

Applications that require writable areas are not limited to DBMS and mail servers. For example, Tomcat needs a writable area for compiling Java programs, many daemon programs needs a writable area for creating lock files used for remembering process IDs and performing exclusive access control.

(c) Services and system can store log information

Suppose the service itself doesn't require writable areas, the administrator needs log information to monitor the activity of services and system. Therefore, it is necessary to provide a writable area using a writable medium such as HDDs for storing logs about the services so that the administrator can refer the logs later when the root filesystem is mounted for read-only.

2.2.1. Directories that needs to be writable in Linux.

It is possible to boot Linux in single-user mode if the root filesystem is mounted without modifications. But the Linux can't proceed to the login prompt when the runlevel is changed to the multi-user mode because the /sbin/mingetty fails to change the owner of /dev/tty* (chown() fails due to read-only filesystem).

The /tmp and /var directories are not essential for the OS itself and the users can direct the applications to use other directories. But these directories are conventionally used for creating temporary files and lock files for exclusive access control and storing process IDs for historical reasons, the administrator has to keep these directories writable.

The /proc dir is managed by the procfs filesystem and no modifications are needed if the root filesystem is mounted for read-only.

2.2.2. The way to prepare writable directories.

There are several ways for providing writable directories when the root filesystem is stored in a read-only medium.

(1) Using tmpfs

The tmpfs is an on memory filesystem and available by default. The data kept in tmpfs is lost when the power fails, but tmpfs is useful if it doesn't matter. Since tmpfs is per a filesystem (mountpoint) basis, the administrator need to create symbolic links from the read-only root filesystem to the directory on tmpfs if per a file basis is needed.

(2) Using devfs

The devfs[6], a filesystem introduced in kernel version 2.3.46, is applicable to only /dev, but devfs can provide the contents of /dev directory on memory without using tmpfs. (Also, the udev available in kernel version 2.4 and later can provide /dev like devfs.)

2.3. An example configuration

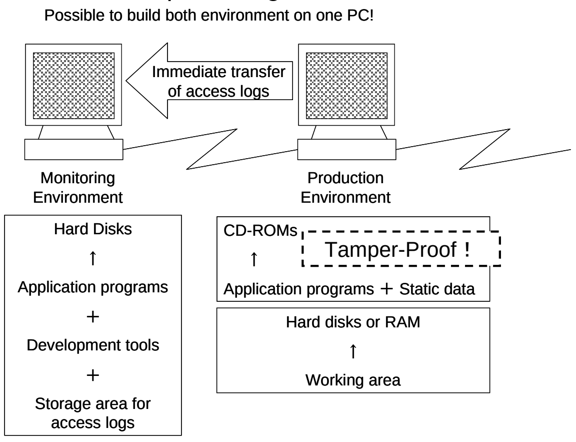

Fig. 1 Configuration for production environment

Fig. 1 is an example configuration that uses a separated monitoring environment. The production environment can be diskless, for it is possible to transfer access logs from the production environment to the monitoring environment via network.

Also, it is possible to configure that doesn't require a separated monitoring environment if the production environment has a HDD for storing access logs.

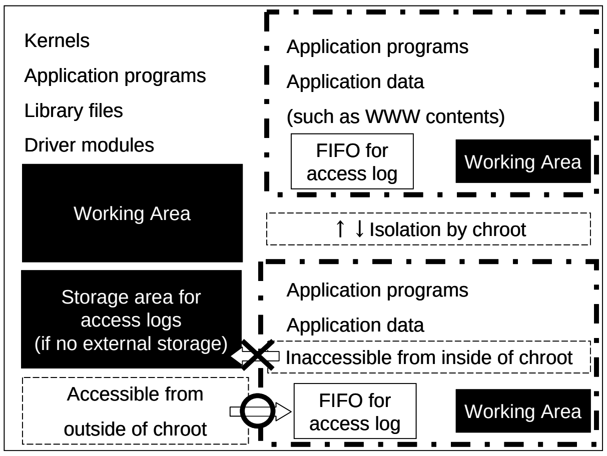

Fig. 2 Location of files.

Fig. 2 is the location of files in the production environment. The inverted area is writable (i.e. HDD or RAM). The alternate long and short dash line areas are environments isolated by chroot(). The access logs won't be tampered with if the processes running in the isolated environments are hijacked, for the access logs are kept outside the isolated environments.

3. Vulnerabilities other than tampering and countermeasurements.

It is possible to perfectly prevent the data from being tampered with if we keep files that must not be tampered with in a physically write protected medium. But it is not enough if seen from the point of view of system operations. We explain some vulnerabilities that may created by a cracker who deprived the administrator's privileges and our countermeasurements against such vulnerabilities.

3.1. Mounting writable filesystem on arbitrary directories.

The physical write protection, to prevent programs and data from being tampered with, becomes useless if some partitions are mounted upon write protected partition such as /bin, for the processes refer the faked files on the mounted partition and this is equivalent to "resources were tampered with". To prevent such scenario from being happen, we modified the function sys_mount() so that the system can mount upon only directories that are allowed in advance.

3.2. Execution of chroot() and pivot_root()

To make Linux work, as aforementioned, the /var and /tmp directories need to be writable. A cracker with administrator's privileges can prepare a fake directory tree orienting from such directories and run malicious programs after chroot'ing to such directories. To prevent such scenario from being happen, we modified the function sys_chroot() so that the system can chroot to only directories that are allowed in advance.

We also modified the function sys_pivot_root() that exchanges the "/" directory and other directory like chroot(). Specifically, we added a Boolean flag, and turn the flag on after the bootup procedure has completed.

4. Implementation

This chapter describes the implementations of our demonstration system that realizes all features aforementioned, including modifications of the kernel to ensure higher security.

4.1. Implementations for read-only mounting.

4.1.1. Modifications of linuxrc

To keep the complete compatibility with ext2 filesystem that are stored in media formatted other than ext2 filesystem, we didn't directly copy the directory tree. We made a loopback image file of ext2 filesystem and copied the directory tree to the loopback image file.

See the chapter "The boot sequences of Linux" for the behavior of linuxrc.

4.1.2. Relocating directory structure.

Prepare a writable partition and mount it.

Move the directories that have to be writable to the writable partition, and create symbolic links from the directories in read-only partition to the directories in writable partition.

Generate a script program that automatically creates directories in writable partition, which is needed only if the writable partition is volatile (i.e. tmpfs).

We don't explain detailed procedure here.

4.2. Implementations for improving security.

4.2.1. Modifications of kernel

(1) Restricting mount operations

We modified the kernel so that the kernel rejects mount requests (and remount requests) that attempt to mount on directories that are not allowed by policy. Actually, register the given combination of mountpoint and device file when the sys_mount() was called with keyword "protect". On the subsequent calls of sys_mount(), the kernel checks the combination of mountpoint and device file and rejects the request if the combination is not registered. The chance of registration is only once so that the cracker can't append combinations of mountpoint and device file, and the registered configuration remains until reboot.

(2) Restricting chroot operations

There is a possibility that a cracker who deprived the root privileges creates a directory tree oriented from writable partitions such as /var, and run programs after chroot'ed to that partition. To prevent such scenario from being happened, we modified the kernel so that the kernel rejects chroot requests that attempts to chroot to directories that are not allowed by policy. Actually, register the given directories when the sys_chroot() was called with keyword "protect:". On the subsequent calls of sys_chroot(), the kernel checks the directory and rejects the request if the directory is not registered. The chance of registration is only once, and the registered configuration remains until reboot.

(3) Restricting pivot_root operations

There is a possibility that a cracker who deprived the root privileges creates a directory tree oriented from writable partitions such as /var, and run programs after exchanging the "/" and the writable directory. To prevent such scenario from being happened, we modified the kernel so that the kernel rejects pivot_root requests if it is once forbidden by policy. Actually, call the sys_pivot_root() with keyword "protect", and on the subsequent calls of sys_pivot_root() are unconditionally forbidden.

(4) Restricting execve operations

This modification is unrelated to vulnerabilities that can't be solved by read-only mounting. This is a modification to reduce the possibility of processes being hijacked by a cracker. Many codes that attempt to hijack the system uses the system call execve() to launch a shell or a terminal. By discarding the privilege to call execve() when the process no longer need to launch a new program, it becomes difficult for crackers to hijack. Actually, we added a Boolean variable in the "struct task_struct" (database used to manage processes). The kernel checks this variable when sys_execve() is called. If sys_execve() was called with the keyword "protect:", the kernel turns on this variable of each process whose ID is given with the keyword so that the processes whose IDs are given with the keyword won't be able to call sys_execve(). This variable remains until the process terminates, and inherited by the child processes created afterwards. The current implementation is to forbid unconditionally, but it will be possible to allow execution of only specific programs (for example, allow execution of java compiler programs for java processes).

4.2.2. Newly created utility programs

These utilities are wrapper programs that enable extensions of the modified kernel aforementioned, and assumed to be used by the legal administrator. (We assume the legal administrator defines policy before a cracker deprives the administrator's privileges.)

(1) limitmount

A wrapper program that calls sys_mount() with the keyword "protect". Use this program within the bootup scripts under the /etc/rc.d directory so that this program is invoked automatically on system's bootup procedure.

(2) limitchroot

A wrapper program that calls sys_chroot() with the keyword "protect:". Use this program within the bootup scripts under the /etc/rc.d directory so that this program is invoked automatically on system's bootup procedure.

(3) limitpivot

A wrapper program that calls sys_pivot_root() with the keyword "protect". Use this program within the bootup scripts under the /etc/rc.d directory so that this program is invoked automatically on system's bootup procedure.

(4) limitexec

A wrapper program that calls sys_execve() with the keyword "protect:". This program can be invoked as many times as the administrator wants.

4.2.3. The value of these modifications.

The modifications to the kernels aforementioned are made as countermeasurements against vulnerabilities that cannot be solved by write protection only. If the administrator doesn't worry about these vulnerabilities, the administrator can operate Linux with root filesystem mounted for read-only without using our modified kernel. But we don't recommend using normal kernel. The other way around, we think the standard kernel wants some countermeasurements against vulnerabilities we have mentioned in this paper.

4.3. Implementations for building chroot'ed environments

It is important that chroot'ed processes can access minimal files if the administrator wishes to introduce isolated environments using chroot. Even if the administrator introduced isolated environments, if the processes can access to unnecessary devices and programs, the damage becomes bigger when the isolated environments are cracked. But it is not easy to pick out minimal files. Therefore, we developed a custom kernel that can automatically pick up only necessary files that are needed for applications running in isolated environments.

The principle is "Record all pathnames that are passed to functions that converts from pathnames into inode structure". But this is not enough, for this will pick out all pathnames accessed by all processes but we want only pathnames accessed by isolated applications. Therefore, we added a variable to the "struct task_struct" that holds the process ID that called the chroot() for the last time so that the kernel can determine whether a process is running in an isolated environment or not by checking the value of this variable whenever the process requests a pathname. When the chroot() is called, the process ID is recorded. Since the "struct task_struct" is inherited by child processes, it is possible to group by process ID by performing chroot for once.

Next, we explain how to obtain the pathnames. To get the list of files that are accessed in a chroot'ed environment, it is possible to copy all files into the chroot'ed environment prior to chroot and then delete files that were not accessed. But actually, we needn't to do so. We can chroot to the current "/" directory. By calling the chroot(), the process ID is recorded in the "struct task_struct". In this way, we can get only the files that are accessed by chroot'ed processes easily and cover all necessary files because the range of accessible files doesn't change.

To implement this feature, we made the following six modifications to the kernel.

(1) include/linux/sched.h：task_struct

Add a variable that holds a process ID.

(2) fs/open.c：sys_chroot()

Records the process ID into the variable in (1).

(3) fs/namei.c：link_path_walk()

Pass the requested pathname to the function in (6).

(4) fs/stat.c：sys_readlink()

This is a readout window of accumulated logs kept in the function in (6). We don't use syslog, for the function printk() is not thread-safe and the logs may get corrupted.

(5) Newly created function that calculated the current directory of the current process.

(6) Newly created function that accumulates logs. If the given pathname is a relative pathname, converts into an absolute pathname by calling the function in (5).

Since we use an existent function as a readout window, we needn't to export a newly created function from the kernel as a system call. Therefore, this implementation is applicable to other kernels that have more system calls.

Also, we developed the following tools that read the access logs and copy files into the isolated environment.

(1) A program that reads access logs using readlink().

(2) A program that copies minimal files into the isolated environment that exist and are accessed by the chroot'ed processes based on access logs.

We confirmed that we can run Apache and Tomcat in chroot'ed environments with minimal files needed for these services using this kernel. Some files such as WWW contents that won't be accessed by just starting and terminating Apache need to be copied manually, but it won't matter because the pathnames of such files are obvious. This kernel is not applicable to this system, this is widely applicable when building chroot'ed environment.

5. Existent technologies used in this system

We can't explain the actual procedure of making "/" partition that can be mounted for read-only due to limitations of space. Instead, we introduce existent technologies and tools used by this system and the result of performance test.

5.1. Technologies used in this system

5.1.1. devfs

devfs is a filesystem managed by the kernel space that provides the contents of /dev directory. At the point of kernel version 2.4.18-14, devfs is considered as EXPERIMENTAL and disabled by default. But by mounting devfs on /dev directory, it becomes possible to mount "/" directory as read-only.

Since devfs also provides the functionality of devpts filesystem, disable devpts when enabling devfs at the kernel compilation time.

5.1.2. iptables

Iptables is used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel. Several different tables may be defined. Each table contains a number of built-in chains and may also contain user-defined chains.

Each chain is a list of rules which can match a set of packets. Each rule specifies what to do with a packet that matches. This is called a 'target', which may be a jump to a user-defined chain in the same table.

By using this function, it is possible to redirect packets that arrived at privileged ports (ports whose numbers are less than 1024) to other ports. Therefore, applications that require root privileges only for opening privileged ports can start without root privileges.

5.1.3. Named pipes (FIFO)

A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the file system. It can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the file system. Thus, the FIFO special file has no contents on the file system, the file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system.

Therefore, it is possible to use FIFO in a filesystem that is physically write-protected. There is a limitation that FIFO cannot do seek operation, but the applications can write to FIFO as well as regular files in a writable filesystem if the FIFO is used for append only such as log files.

5.1.4. ISO filesystem

ISO filesystem is used widely by CD-ROMs. The program mkisofs is used to create ISO image files.

mkisofs is effectively a pre-mastering program to generate an ISO9660/JOLIET/HFS hybrid filesystem. mkisofs takes a snapshot of a given directory tree, and generates a binary image which will correspond to an ISO9660 or HFS filesystem when written to a block device.

mkisofs is capable of generating the System Use Sharing Protocol records (SUSP) specified by the Rock Ridge Interchange Protocol. This is used to further describe the files in the iso9660 filesystem to a unix host, and provides information such as longer filenames, uid/gid, posix permissions, symbolic links, block and character devices.

5.2. The features of demonstration system

- The system files and programs are never tampered with.
- Provides two services (Apache and Tomcat), which the application effect of this approach is considered larger.
- These services can't access to system files and log files even if cracked, for they run under the chroot'ed environment.
- The administrator's privileges won't be deprived via these services, for these services don't require the administrator's privileges from the beginning.
- The access logs won't be deleted even if cracked, for FIFOs are used for access logs.

5.3. The performance of demonstration system

Our approach doesn't affect the system's performance because we didn't make massive modifications to the kernel like MAC. We measured the performance of transferring WWW contents kept in the root filesystem recorded in a CD-R.

Server: Dell PowerEdge 1550
CPU: Pentium III 1GHz
RAM: 1280MB
CD-ROM: Max. X24 read
Size of WWW data: 40170370 Bytes
Measurement method: Download using wget

The result is shown in Fig. 3. For the first trial it is very slow, but for the subsequent trials they aren't. For the first trial, the necessary files are searched from CD-R and read and transferred, but for the subsequent trials, the necessary data is transferred from cache that is on the RAM.

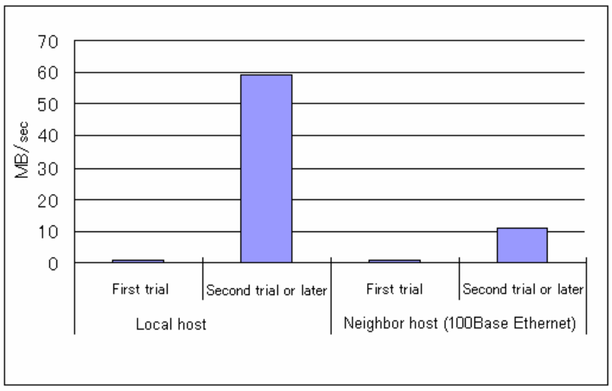

Fig. 3 The transfer speed of WWW contents

6. Reference: The boot sequences of Linux

In this chapter we briefly review the boot sequence of Linux and explain about modifications we made to provide services using read-only mounting.

6.1. From the boot to the invocation of linuxrc

(1) The bootloader program reads compressed kernel image file (vmlinuz) and decompresses it onto RAM, and gives control to the kernel.

(2) The vmlinuz reads a file named initrd.img and decompresses it onto RAM and mount it on "/" directory.

(3) The vmlinuz executes a program named linuxrc that is located in the "/" directory of decompressed image. The linuxrc is a script and interpreted by nash (very compact shell program) that is dedicated for running linuxrc script.

6.2. The role of linuxrc and modifications we made

The linuxrc performs mainly the following two tasks.

- Load device driver modules that are needed for mounting real root device (usually a HDD).
- Mount the real root device for read-only. (During this task, pivot_root() is called to switch the "/" directory.)

Our method mounts root filesystem using loopback so that the administrators can do version control easily. Therefore, we added a task that mounts pre-created image file that contains the contents of root filesystem in addition to standard tasks listed above.

6.3. After the invocation of /sbin/init

When the execution of initrd.img finished, the root filesystem is already mounted, and the kernel gives control to /sbin/init. /sbin/init executes /etc/rc.sysinit script. In this script, root filesystem is remounted for read-write. After this script, /sbin/init enters to the specified runlevel and executes startup scripts for that runlevel. Finally, it becomes possible for users to login.

We made the following modifications in startup scripts so that the services won't abort due to the root filesystem remaining mounted for read-only.

- Comment out the writability tests used to test the user is root
- Prepare the contents of /dev directory (if devfs is not used)

7. Possibility of applying our method

7.1. The use of media other than CD-R

We used CD-R as a recording medium, but it is impossible to use CD-R for protecting massive data due to the medium's capacity limitation. But the technique of mounting ext2-formatted root filesystem using loopback allows users use arbitrary media formatted with arbitrary filesystem. Therefore, it is possible to use (for example) DVD-R as well.

Also, HDDs that can do write protection at hardware level (i.e. independent with OS) are coming to market. By combining with such products, it is possible to protect massive amount of data from tampering without loosing performance.

7.2. Other applications of read-only mounting

The significance of read-only mounting is not limited to preventing configurations of system and applications.

In standard Linux, all information including access logs become unreliable if the administrator's privileges were deprived (because the administrator is free from access controls). But the policy files remains reliable if they are kept in a read-only medium using our method.

The administrator can be confident in programs for incident detection and incident response kept in a read-only medium even after the administrator's privileges were deprived.

7.3. Distribution Dependency

We think that it is possible to apply our approach to distributions other than Red Hat Linux, for our approach doesn't entail massive modifications of kernels.

The patches and the tools our system uses are very small, and the modifications are closed within specific system calls. We think it is easy to port to other distribution's kernels.

8. Conclusion

The MAC implementations on Linux such as SELinux, RSBAC[7] demonstrated the possibility of improving Linux's security. But they entail management of enormous quantity of policy and loss of performance due to authorization of system call requests, and it is difficult to say that they are aiming at everyone.

The method described in this paper that mounts root filesystem for read-only can prevent files from being tampering with even the administrator's privileges are deprived, without entailing management of policy. This method is effective by itself, but it offers stronger protection by applying tiny patches to the kernel that restricts chroot, pivot_root and mount operations. We think that these patches are also effective for systems that don't mount root filesystem for read-only.

The technologies such as devfs, iptables and chroot are already incorporated into the kernel and used, and the usage of these technologies in this paper is nothing new. The restriction of execve itself is not versatile and the applicable scope is limited, but this is a result of attempts for improving Linux's security with minimal modifications to Linux. We hope you find our approach informative.

Acknowledgment: During this research, we referred to Mr. YAMAMORI Takenori's article and his website [8]. Mr. YAMAMORI not only agreed readily to be listed in the bibliography but also pointed some of errors in our manuscript. We sincerely thank Mr. YAMAMORI. Also, we were patiently supported by Tetsuo Handa, NTT DATA CUSTOMER SERVICE CORPORATION for developing the prototype of TOMOYO Linux. We would like to thank Mr. Handa.

Bibliography

[1] Peter A. Loscocco et al, *The Inevitability of Failure: The Flawed Assumption of Security in Modern Computer Environments*

[2] United States. Department of Defense, *TCSEC (Trusted Computer System Evaluation Criteria) DDS-2600-5502-87*, 1985

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

[4] Serge E. Hallyn, *Domain and Type Enforcement for Linux,* 4th Annual Linux Showcase & Conference, 2000

[5] Chris Wright and Crispin Cowan et al, *Linux Security Module Framework*

[6] Richard Gooch, *Linux Devfs (Device File System),* /usr/src/linux-2.4/Documentation/filesystems/devfs/README, http://www.atnf.csiro.au/people/rgooch/linux/docs/devfs.html

[7] *Rule set based access control (RSBAC) for linux*, http://www.rsbac.org

[8] YAMAMORI Takenori, "Let's create original Linux live on CD-ROM", Software Design vol. December 2002, http://www15.big.or.jp/~yamamori/sun/cdlinux/ (Written in Japanese), http://www15.big.or.jp/~yamamori/sun/tech-linux-2/index_e.html

Notes

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

http://sourceforge.jp/projects/tomoyo/document/l c2003.pdf

The technology shown in this paper was incorporated into 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/