Task Oriented Management Obviates Your Onus on Linux.

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

Abstract

The security enhanced OSes are getting to be introduced as a method to improve security of computer systems. Many of security enhanced OSes performs access controls based on roles assigned to users and labels assigned to resources, but this approach tends to require very complicated policy files and is not always manageable for everyone. This paper describes the features and implementations of TOMOYO Linux, a security enhanced Linux kernel with automatic policy generation technology. TOMOYO Linux doesn't require labeling, and performs access control on processes. The approach of TOMOYO Linux makes it possible to perform simple but effective Mandatory Access Control.

1. Introduction

The access controls on files and directories in normal Linux/UNIX are performed based on access control information called "security bits" that are kept within the filesystem, and the owner of files and directories can change the access control information freely. Also, if the process that requested accesses is running with administrator's privileges, the access requests are always granted regardless of the access control information. Therefore, it is inevitable that the system gets completely damaged if the control of any processes that are running with administrator's privileges is wrested by (for example) attacking buffer overflow. This is known as security problem [1, 2]. To solve this security problem, MAC (Mandatory Access Control) was invented and open sourced implementations of MAC such as SELinux [3, 4] are appearing. Also, LSM (Linux Security Modules) [5], the framework to make security expansion easier, was incorporated into Linux 2.6 kernels and became available.

The most of MAC implementations on Linux performs access control by "hooking system calls in the kernel space" and "checking the validity by comparing with policies supplied by administrators" and "processing only if the request is valid". In Linux, all functions are processed via system calls provided by the kernel, the method of hooking system calls can improve Linux's security for sure. But that method doesn't solve all problems. The introduction of MAC entails policy definition and management depending on the granularity of individual MAC implementations. Also, in SELinux, the identifier called "label" has to be assigned ALWAYS APPRICIATELY to resources such as files and directories. As a result, though the existent MAC implementations on Linux fill the requirement of functionality, there still remains the large barrier when considering actual introduction and operation. The authors of this paper (hereafter, we) have developed our original security enhanced Linux "TOMOYO Linux" to solve this barrier.

2. TOMOYO Linux

TOMOYO Linux is the acronym for "Task Oriented Management Obviates Your Onus on Linux" and is a security enhanced Linux kernel with the following features.

- Can generate policy automatically
- MAC based on the "struct task_struct"
- No labeling needed
- Possible to combine with SAKURA Linux
- Can discard unnecessary privileges voluntarily
- Beginners-friendly policy syntaxes
- Has CUI based policy editor

We explain each feature below.

2.1. Automatic policy generation technology

The MAC accepts or rejects strictly and precisely the access request based on policies predefined and supplied by administrators. Therefore, the administrators have to figure out all resources that might be accessed and grant permissions to them, and this makes actual operation more difficult. It is easy to operate if all files and directories that (for example) Apache (the HTTP server) accesses are listed in the Apache's configuration files, but it is impossible to do so. There are dynamically linked library files (that can be found by using "ldd" command), modules loaded on demand (that cannot be found by using "ldd" command), files accessed via multiple names due to hard links and symbolic links, and (for example) CGI programs and perl and ruby. The administrator has to figure out all files and directories and their access modes. It is easily understand that this is almost impossible. But the administrator has to do so to define policy needed for MAC.

We focused attention on this problem, and have developed "Access policy generation system based on process execution history"[7]. This system captures and records access requests in the kernel space, and can generate about 90% of policies needed for MAC by just performing a series of operations the administrator wishes to allow. TOMOYO Linux implemented MAC using this technology. TOMOYO Linux can provide policy to administrators by booting with "accept mode" and performing a series of operations the administrator wishes to allow. The administrator can define just enough policy by editing and authorizing the policy provided by "accept mode". The biggest feature of TOMOYO Linux is that TOMOYO Linux has both "accept mode" that assists generating policy for MAC and "enforce mode" that performs MAC based on policy.

Also the unique domain division rule allows administrators grant permissions to minimal resources to domains that are running the same program depending on their contexts.

2.2. MAC based on the "struct task_struct"

In Linux, the "struct task_struct" is designed and implemented to be inherited from parent process to child processes using fork() and execve() system calls. We focused on this feature and implemented original MAC without using LSM. The patch of our MAC implementation for standard kernels is very compact, and allows access controls based on process execution history.

2.3. No labeling needed for files and directories

In many MAC implementations including SELinux, the "label" is assigned to files and directories first and then policies are defined using "label". For example, to define policy that controls access to /etc/passwd, the administrator has to assign a label to both /etc/passwd and programs that access to /etc/passwd (such as /usr/bin/passwd).

There are two problems. One is that it is laborious for administrators to assign appropriate labels to all resources. The other one is that it may happen that the binding of pathnames and labels accidentally get corrupted and as a result the appropriate access controls cannot be performed. To avoid these problems, TOMOYO Linux uses "canonicalized pathnames" instead of "labels", and allows administrators intuitive policy definition.

2.4. Possible to combine with SAKURA Linux

It becomes more secure by combining with (code name) SAKURA Linux [6] we have explained in Linux Conference 2003 to ensure physically programs and files are not being tampered with. (The logical protection (i.e. mounting read-only) is not always tamper-proof, because the content of read-only mounted medium will be destroyed by directly writing to device files that corresponds to the medium. Therefore, physical protection is important.)

2.5. Possible to discard voluntarily unnecessary privileges

The security enhanced Linux implementations including SELinux use existent programs without modification, and restricts their behavior using policies. TOMOYO Linux needn't to modify existent programs, but TOMOYO Linux provides existent programs a mechanism to discard unnecessary privileges voluntarily. It is possible to (for example) "discard a privilege to execute new program (i.e. calling execve())" or "discard a privilege to reacquire root privileges (i.e. becoming euid = 0)" by just inserting one line to existent programs. For example, if /usr/sbin/httpd needn't to execute new programs (for example, /bin/sh), by inserting a line that discards the privilege to call execve(), the control of /usr/sbin/httpd won't be wrested by execution of exploit code. There is no need for policy management for this mechanism. It is possible to apply this mechanism to restrict more tightly that are already restricted by MAC's policy.

3. Considerations on "domain"

3.1. The "domain" and "domain transition diagram"

To perform MAC, the domain is assigned to each process, and restricts accessible resources for each domain. The state of a process changes when (for example) a new program is executed, and the process transits to different domain if conditions defined in the policy are met. By appropriately restricting domains that are allowed to transit to, the system's appropriate behaviors and securities are ensured.

There is a flowchart called "domain transition diagram" to figure out the domain transitions visually. Whether the domain transition diagram is accessible for the administrator directly mirrors whether the administrator can figure out domain transitions correctly and define appropriate policy.

The three types of domain transition diagram are shown below.

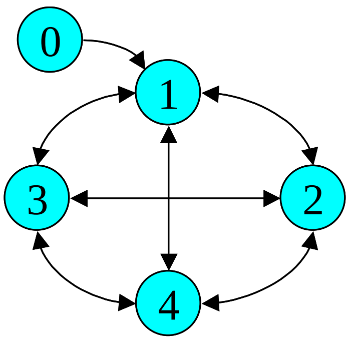


Fig. 1 No restrictions for domain transition

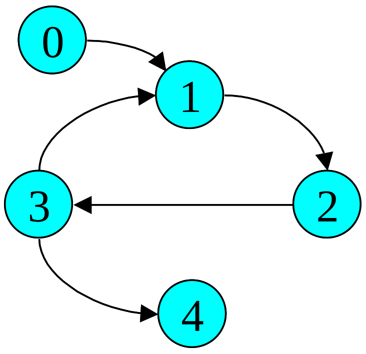


Fig. 2 Considering only the current domain for domain transition

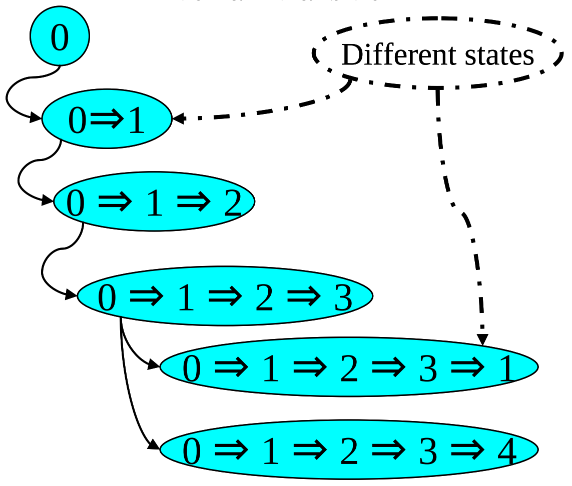


Fig. 3 Considering the whole past domains for domain transition

It makes possible to restrict the domains that can transit to by introducing MAC. In other words, before introducing MAC, a process can transit to arbitrary states as shown in Fig. 1. After introducing MAC, a process cannot transit to arbitrary states as shown in Fig. 2 and Fig. 3. Now, let's consider which of Fig. 2 and Fig. 3 is easier to understand.

The Fig. 2 can reduce the total number of domains, but may yield infinite domain transition loop. If the total number of domains is small, the Fig. 2 is convenient. But if the total number of domains becomes larger, you can't trace the all of domain transitions. You can't even draw the domain transition diagram that precisely mirrors the system's behavior. Also, Fig. 2 makes more difficult to figure out the range of domains that are affected by the insertion or deletion of domains. You have to verify that the insertion or deletion of domains doesn't break the existent paths needed for the system's appropriate behavior.

The total number of domains in Fig. 3 becomes largest since all states are differentiated. But you can trace the all of domain transitions regardless of the total number of domains because the domain transitions are tree structured. You can always draw the domain transition diagram that precisely mirrors the system's behavior. Also, Fig. 3 makes obvious to figure out the range of domains that are affected by the insertion or deletion of domains. You can easily verify the range. The Fig. 3 entails the cost of defining all domains. But the cost is dramatically reduced because the domains are assigned mechanically by adopting "Access policy generation system based on process execution history"[7] approach. We think the cost of defining all domains mechanically is much smaller than the cost of verifying all domains whenever a domain transition path is inserted or deleted manually.

3.2. The "domain" in SELinux

The SELinux adopts the domain transition of Fig. 2. Therefore, it makes difficult to figure out visually if the administrator attempts to divide existent domains into smaller domains to perform accesses controls more tightly. Also, since SELinux limits the letters that can be used for the name of domains and labels (for example, "/" is not allowed), the administrator can't use pathnames for the name of domains and labels, leading to non-intuitive bindings of "the name of domain and the name of individual programs" and "the name of label and the name of individual files".

3.3. The "domain" in SubDomain

The SubDomain[8] is a Linux that supports MAC developed by Immunix. The SubDomain can perform MAC mainly on files and directories and the execution of programs. The SubDomain adopts the domain transition of Fig. 3. Though it is impossible to perform MAC as fine-grained as SELinux, it is accessible for beginners because the structure is simple and the policy is defined using pathnames (unlike SELinux that requires labels). Our TOMOYO Linux supports some functions similar to SubDomain.

3.4. The "domain" in TOMOYO Linux

The TOMOYO Linux adopts the domain transition of Fig. 3. Also, TOMOYO Linux uses the pathnames of programs for the name of domains. Therefore, it makes easy to figure out and edit visually. Unlike Fig. 2, Fig. 3 can grant permissions to minimal resources for the same program considering the process's execution history.

4. Implementations of MAC

Our SAKURA Linux [6] can prevent files from being tampered with even if the system is wrested by the cracker. Though the cracker can't tamper with files, the cracker can execute arbitrary programs and read arbitrary files because the access control itself is not enforced, and this is a security problem. But by combining this system with TOMOYO Linux, it makes possible to perform the access control while preventing tampering.

This chapter describes the abstract of MAC implementation in TOMOYO Linux.

4.1. Access control based on canonicalized pathnames

Many of MAC implementations perform access controls based on labels. These labels are, aside from existent pathnames, newly assigned identifiers used only for performing MAC. But, by right, there is no need to assign labels for files and directories, for the canonicalized pathnames are unique on that system and can be used as identifiers.

By using pathnames for labels, it makes possible to figure out viscerally the binding of the subjects of accesses (i.e. processes) and the objects of accesses (i.e. resources). Also, the implementation becomes simpler because the system can always assume the correctness of labeling. This will become a great advantage in actual operations. We will briefly describe the procedure to derive the canonicalized pathnames below.

4.1.1. The way of handling pathnames from the

user's view

The userland programs handle pathname as a NULL terminated string. It is essential for users that users can handle pathnames using understandable string.

4.1.2. The way of handling pathnames from the Linux kernel's view

The kernel handles pathname using a structure named "struct nameidata", not a NULL terminated string. The kernel can't handle pathnames in the form of string. The procedure that converts the NULL terminated string into the "struct nameidata" is link_path_walk() defined in fs/namei.c. The "struct nameidata" contains the "struct dentry" and the "struct vfsmnt". The former holds the information of the file in the filesystem, and the latter holds the information of the mountpoint.

4.1.3. The way to convert into canonicalized pathnames

The "struct nameidata" is the form of representing pathnames in the kernel. The procedure that converts the "struct nameidata" into the NULL terminated string is __d_path() defined in fs/dcache.c. By using __d_path(), we can obtain the canonicalized pathnames (that has no symbolic links) that has already converted into the "struct nameidata".

Since the __d_path() calculates up to only the process's "/" directory, we can't obtain the canonicalized pathnames if the process has already chroot'ed to somewhere. Therefore, we created a new function that ignores the process's "/" directory to obtain the canonicalized pathnames based on __d_path().

4.2. Domain transitions

In TOMOYO Linux, every process always belongs to a domain, and the access permissions for files and directories are granted to each domain. But, the definition of domain in TOMOYO Linux differs from the definition in SELinux.

The initial domain is the kernel, and is represented as "<kernel>". Since /sbin/init is invoked by the kernel, the domain for /bin/init is represented as "<kernel> /sbin/init". The domain for /etc/rc.d/rc invoked by /sbin/init is represented as "<kernel> /sbin/init /etc/rc.d/rc".

Since the process invocation history that tells how the current process is invoked is maintained, the same program belongs to different domains if their parent domains differ. This allows administrators grant permissions to minimal resources for the same program depending on their contexts. Also, it is easy to edit the domain transition because the domain transition is tree structured.

In the kernel space, there is a domain index table that converts the domain names into the domain numbers. From the point of process's view, the domain transition means the change of domain number.

4.3. Access controls on execution of programs

When invoking a program, the execution permission is checked.

Filename	Function	Location to check
fs/exec.c	do_execve()	open_exec()

The do_execve() function performs the following steps.

(1) Get the canonicalized pathname of the requested program.

(2) If the kernel is running with accept mode, grant the execute permission of the pathname obtained in step (1) to the domain the subject process belongs to. If the kernel is running with enforce mode, check whether the execute permission of the pathname obtained in step (1) is granted to the domain the subject process belongs to, and return error if not granted.

(3) Concatenate the name of the subject process's domain and the canonicalized pathname obtained in step (1), and hold the result as the name of the domain that subject process will belong to.

(4) If the kernel is running with accept mode, register the name of domain obtained in step (3) into the domain index table. If the kernel is running with enforce mode, check whether the name of domain obtained in step (3) is registered in the domain index table, and return error if not

registered.

(5) Continue the standard procedure of do_execve(), and transit to the domain obtained in step (3) if this procedure succeeded. In the standard procedure of do_execve(), we don't give the canonicalized pathname obtained in step (1), for there are programs that behave differently depending on their invocation name. For example, /sbin/pidof is a symbolic link to /sbin/killall5, and /sbin/pidof and /sbin/killall5 behave differently depending on their invocation name.

4.4. Access controls on read accesses

When opening files for reading or loading shared libraries, the read permission is checked.

Filename	Function	Location to check
fs/open.c	filp_open()	dentry_open()
fs/exec.c	sys_uselib()	read_lock()

The filp_open() function performs the following steps.

(1) Convert the requested path into the "struct nameidata" using open_namei().
(2) Get the canonicalized pathname of the "struct nameidata" obtained in step (1).
(3) If the kernel is running with accept mode, grant the read permission of the pathname obtained in step (2) to the domain the subject process belongs to. If the kernel is running with enforce mode, check whether the read permission of the pathname obtained in step (2) is granted to the domain the subject process belongs to, and return error if not granted.
(4) Open the file pointed by the "struct nameidata" obtained in step (1) using dentry_open().

Regarding sys_uselib(), the read permission is checked before starting binary file type matching test.

4.5. Access controls on write accesses

When opening files for writing, the write permission is checked in the same manner of opening files for reading. But there are more write operation other than opening files for writing. They are, newly creating files and directories, deleting existent files and directories, creating and deleting hard links, renaming existent files and directories, creating and deleting symbolic links, truncating files, and so on. Therefore, we check the write permission in the following places.

Filename	Function	Location to check

fs/namei.c	open_namei()	vfs_create()
		vfs_truncate()

	sys_mknod()	vfs_create()
		vfs_mknod()

	sys_mkdir()	vfs_mkdir()
	sys_rmdir()	vfs_rmdir()
	sys_unlink()	vfs_unlink()
	sys_symlink()	vfs_symlink()
	sys_link()	vfs_link()
	do_rename()	vfs_rename()
fs/open.c	filp_open()	dentry_open()
	do_sys_truncate()	vfs_truncate()

We check write permission before calling the VFS functions (functions whose name begins with "vfs_"). Since the VFS functions don't handle operations that cross mountpoints, the VFS functions don't receive the information of mountpoints (i.e. "struct vfsmnt" parameter). As a result, we can't obtain the canonicalized pathnames after entering into the VFS functions.

There are more operations that needs DAC's write permissions such as changing owners and permissions and timestamps, but we don't check MAC's write permissions for such operations. We check MAC's write permission for operations that changes the content of files and directories, for we have only one write permission.

We perform in the following way.

(1) Convert the requested pathname into the "struct nameidata".
(2) Do the basic sanity checks for VFS functions, such as crossing mountpoint checks, the function that actually performs is implemented, DAC's permission checks. If any error occurs, return error.

(3) Get the canonicalized pathname from the "struct nameidata" obtained in step (1).

(4) If the kernel is running with accept mode, grant the write permission of the pathname obtained in step (3) to the domain the subject process belongs to. If the kernel is running with enforce mode, check whether the write permission of the pathname obtained in step (3) is granted to the domain the subject process belongs to, and return error if not granted.

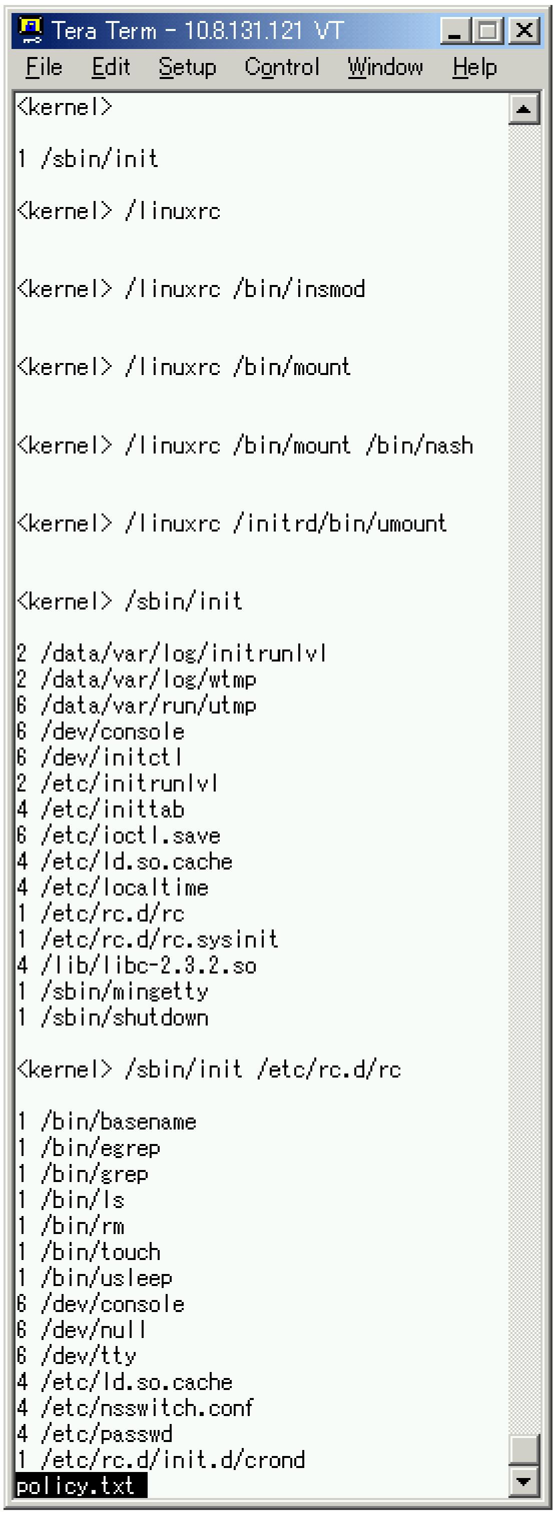
(5) Call the VFS function.

4.6. The timing of loading and saving policy

The policy files are loaded just prior to starting /sbin/init. The policy files are saved just prior to power fails in the shutdown script (i.e. /etc/rc.d/init.d/halt) if the kernel is running with accept mode. It is possible to save policy files in arbitrary timing if the system is running with accept mode.

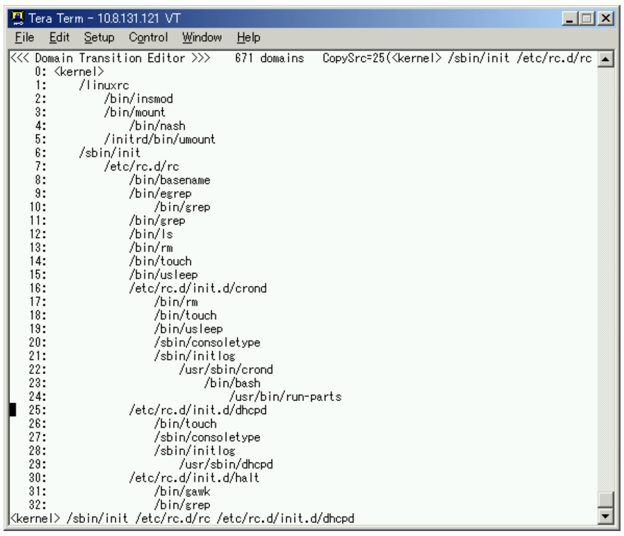
4.7. The policy editor

The policy files of TOMOYO Linux consist of ASCII printable text that is separated with spaces and carriage returns, as shown in Fig. 3. You can edit the policy files using arbitrary text editors such as emacs. You can edit the domain transition using CUI based policy editor, as shown in Fig. 4. To represent all possible characters, the characters whose value in ASCII codebook are less or equals to 32(SP character) and greater or equals to 127(DEL character) are represented using "\ooo" style octal value, and "\" itself is represented using "\\". For example, the SP character is represented as "\040".


Tera Term - 10.8.131.121 VT
File Edit Setup Control Window Help

```
<kernel>

1 /sbin/init

<kernel> /linuxrc


<kernel> /linuxrc /bin/insmod


<kernel> /linuxrc /bin/mount


<kernel> /linuxrc /bin/mount /bin/nash


<kernel> /linuxrc /initrd/bin/umount


<kernel> /sbin/init

2 /data/var/log/initrunlvl
2 /data/var/log/wtmp
6 /data/var/run/utmp
6 /dev/console
6 /dev/initctl
2 /etc/initrunlvl
4 /etc/inittab
6 /etc/ioctl.save
4 /etc/ld.so.cache
4 /etc/localtime
1 /etc/rc.d/rc
1 /etc/rc.d/rc.sysinit
4 /lib/libc-2.3.2.so
1 /sbin/mingetty
1 /sbin/shutdown

<kernel> /sbin/init /etc/rc.d/rc

1 /bin/basename
1 /bin/egrep
1 /bin/grep
1 /bin/ls
1 /bin/rm
1 /bin/touch
1 /bin/usleep
6 /dev/console
6 /dev/null
6 /dev/tty
4 /etc/ld.so.cache
4 /etc/nsswitch.conf
4 /etc/passwd
1 /etc/rc.d/init.d/crond
policy.txt
```

Fig. 3 An example policy


Tera Term - 10.8.131.121 VT
File Edit Setup Control Window Help

```
<<< Domain Transition Editor >>>    671 domains   CopySrc=25(<kernel> /sbin/init /etc/rc.d/rc
   0: <kernel>
   1:     /linuxrc
   2:         /bin/insmod
   3:         /bin/mount
   4:             /bin/nash
   5:         /initrd/bin/umount
   6:     /sbin/init
   7:         /etc/rc.d/rc
   8:             /bin/basename
   9:             /bin/egrep
  10:                 /bin/grep
  11:             /bin/grep
  12:             /bin/ls
  13:             /bin/rm
  14:             /bin/touch
  15:             /bin/usleep
  16:             /etc/rc.d/init.d/crond
  17:                 /bin/rm
  18:                 /bin/touch
  19:                 /bin/usleep
  20:                 /sbin/consoletype
  21:                 /sbin/initlog
  22:                     /usr/sbin/crond
  23:                         /bin/bash
  24:                             /usr/bin/run-parts
  25:             /etc/rc.d/init.d/dhcpd
  26:                 /bin/touch
  27:                 /sbin/consoletype
  28:                 /sbin/initlog
  29:                     /usr/sbin/dhcpd
  30:             /etc/rc.d/init.d/halt
  31:                 /bin/gawk
  32:                 /bin/grep
<kernel> /sbin/init /etc/rc.d/rc /etc/rc.d/init.d/dhcpd
```

Fig. 4 Domain Transition Editor

5. Implementations of voluntarily discarding privileges

We implemented a mechanism that allows processes discard unnecessary privileges voluntarily in addition to MAC.

For example, the process can "discard a privilege to execute new program (i.e. calling execve())" or "discard a privilege to reacquire root privileges (i.e. becoming euid = 0)". This mechanism allows restricting more tightly that are already restricted by MAC's policy.

We describe a method that can improve security with very simple implementations. Some part of this method was already implemented in SAKURA Linux [6].

5.1. The reason to use "task_struct"

In Linux, every process has a database named "struct task_struct", as shown in Fig. 5.

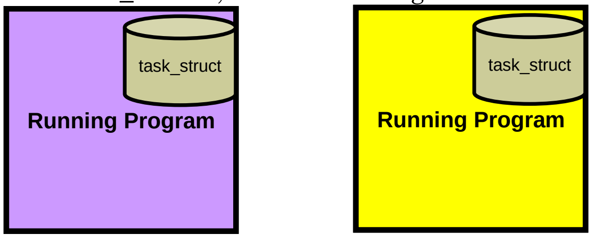


Fig. 5 "struct task_struct"

This database holds information such as process id, the owner id of the process, memory usage, process's "/" directory, as shown in Fig. 6.

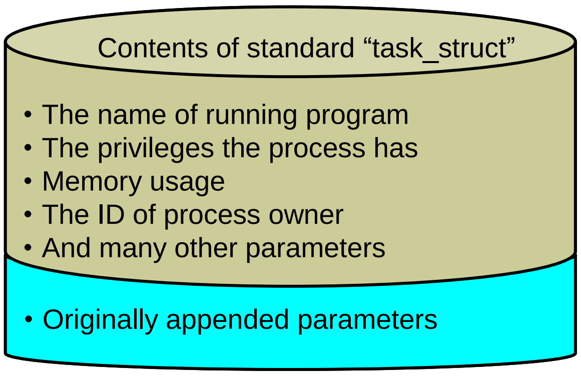

Contents of standard “task_struct”
• The name of running program
• The privileges the process has
• Memory usage
• The ID of process owner
• And many other parameters
• Originally appended parameters

Fig. 6 The content of "struct task_struct"

This database is duplicated (as shown in Fig. 7) when the parent process creates a child process (i.e. calling do_fork() defined in kernel/fork.c), and updated (as shown in Fig. 8) when the process executes a new program (i.e. calling do_execve() defined in fs/exec.c).

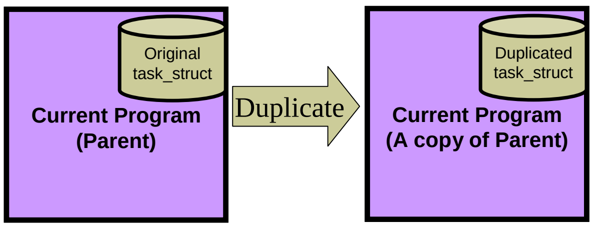


Fig. 7 Duplicating "struct task_struct"

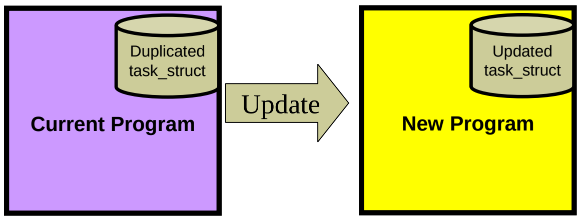


Fig. 8 Updating "struct task_struct"

All processes are the descendant of the /sbin/init (the first program executed by kernel), and they can inherit some contents of /sbin/init's task_struct.

If a special content is added to this database, the added contents are also inherited to descendant processes. This means that, if the parent process records a list of unnecessary privileges, the child process created afterwards inherits the list declared by the parent process.

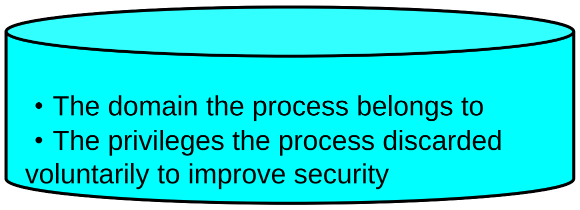

• The domain the process belongs to
• The privileges the process discarded voluntarily to improve security

Fig. 9 Originally appended parameters

We implemented the discarded privileges are not recoverable. Actually, we added a variable (as shown in Fig. 9) that records privileges that were declared as "unnecessary to the process". When the process declared the corresponding system calls, the corresponding bit is set. When the system calls that can be prohibited using this mechanism are called, the bit state is checked, and rejects the request or terminates process if the bit is set.

"Who knows best about the necessary and unnecessary privileges for programs?" "It's not the kernel, but the programs themselves."

"The experts know best." is the principle of this voluntarily access control mechanism. This mechanism improves security by letting processes declare unnecessary things at first. This mechanism is exact reverse of MAC that lets the kernel perform all access controls.

5.2. Disabling do_execve()

A process can discard the privilege to execute a new program. If the parent process discarded this privilege, the child processes created afterwards cannot execute a new program. For example, /bin/ls needn't to execute other programs. In this case, by discarding the privilege to execute a new program as soon as /bin/ls starts, /bin/sh won't be invoked if /bin/ls has a vulnerability that allows execution of arbitrary code. Most programs, excluding some programs such as shells, needn't to invoke a new program (i.e. calling do_execve()), though they need to create their copies (i.e. calling do_fork()). By discarding the privilege to call do_execve(), it makes difficult for crackers to execute shellcode by attacking vulnerabilities.

5.3. Disabling sys_chroot()

A process can discard the privilege to change the "/" directory. The behavior is similar to do_execve().

(TOMOYO Linux can also restrict the directories that are allowed to chroot to using sys_chroot() by policy, in addition to this discarding privilege mechanism. In Linux, a process can chroot to arbitrary directories, but it is not preferable in the point of view of security. By combining with SAKURA Linux [6], you can forbid chroot'ing to writable directories so that the cracker can't build the directory tree with malicious programs.)

5.4. Disabling sys_pivot_root()

A process can discard the privilege to exchange "/" directory. The behavior is similar to do_execve().

(TOMOYO Linux can also forbid calling sys_pivotroot(), in addition to this discarding privilege mechanism.)

5.5. Disabling sys_mount()

A process can discard the privilege to mount a filesystem. The behavior is similar to do_execve().

(TOMOYO Linux can also restrict the combinations of mountpoints and devices that are allowed to mount using sys_mount() by policy, in addition to this discarding privilege mechanism. In Linux, a process can mount on arbitrary directories, but it is not preferable in the point of view of security. You can forbid mounting (for example) tmpfs so that the cracker can't build the directory tree with malicious programs.)

5.6. Disabling reacquiring root privileges

In Linux, it is recommended to keep the exercise of the system administrator's privileges in the routine work as minimum as possible. This is because the damage becomes larger if the administrator did wrong operations. Therefore, the style "Login as non-root, do regular operations as non-root, and become root using /bin/su only when the root privileges are essential" has been established.

But we think that it seldom happens that a process reacquires the root privileges (except for the case administrator uses /bin/su) after the process discarded the root privileges. An example of reacquiring the root privileges happens when the cracker attempts to acquire the root privileges using a hijacked process.

Based on these assumptions, we implemented a mechanism that can forbid reacquiring the root privileges once the process discarded the root privileges.

The checking of reacquisition of the root privileges is done in the function link_path_walk() defined in fs/namei.c. In nature, this checking should be done in the system calls that changes the process's privileges, but we dare to do it in the link_path_walk() due to the following two reasons.

- Unexpected switching to the root privileges may happen due to the vulnerability of the system calls.
- The function link_path_walk() is always called to invoke the shell program with the root privileges.

Strictly speaking, we can't forbid reacquiring the root privileges. But the most operations that handles pathnames call link_path_walk(), it is possible to detect immediately regardless of the location the process reacquired the root privileges.

The process is forcefully terminated if the function link_path_walk() detected the process has reacquired the root privileges while the process was forbidden to reacquire the root privileges.

It is also possible to discard this privilege while the process has the root privileges. In that case, it takes place when the function link_path_walk() detected that the process has discarded the root privileges.

5.7. Some other examples

At this time, we only implemented discarding privileges that are related to filesystem, but it is possible to discard privileges that are not related to filesystem in the same way. For example, it is possible to discard a privilege to use TCP/IP, a privilege to create a child process.

In contrast to discarding privileges, it is possible to inherit only a part of the system administrator's privileges using this mechanism. For example, modify the kernel to allow the use of TCP port 80 for processes that aren't running with root privileges but has the information that allows the use of TCP port 80 in their "struct task_struct". The Apache without the root privilege can use the TCP port 80 by invoking the Apache after registering the information that allows the use of TCP port 80 in the parent process's "struct task_struct" that is running with the root privileges.

5.8. Problems of voluntarily discarding privileges features

There is no official interface that provides the voluntarily discarding privileges mechanism. The current implementation uses do_execve(), for this function can receive variant numbers of string parameters. This mechanism takes place when the special keyword is passed to do_execve().

The modifications of source codes needed for using this mechanism is trivial. It only requires the insertion of a several lines, for all you need to do is that "call the function that provides this mechanism (in this case, do_execve()) with the privileges to discard and the process ID".

We would like to entrust the arguments about "What privileges should be discardable?" and "What interface should be prepared?" to the experts of Linux kernels. The contents described in this chapter are our suggestions that it is possible to improve security easily by utilizing the "struct task_struct" aside from the MAC's policy approaches.

6. Example applications for TOMOYO Linux

We confirmed the following applications are runnable in a read-only medium by combining SAKURA Linux [6] with TOMOYO Linux's kernel. We are operating the following applications in a USB flash memory with 1GB capacity for demonstration environment.

- WWW Server (httpd-2.0.40-21.11)
- LXR + glimpse (lxr-0.3.1 + glimpse-4.17.4)
- Another HTML-Lint
- WWW Server (Tomcat 4.1.30 + Java 1.4.2_04)
- FTP Server (vsftpd-1.1.3-8)

- DNS Server (bind-9.2.1-16 + caching-nameserver-7.2-7)
- DHCP Server (dhcp-3.0pl1-23)
- SAMBA Server (samba-2.2.7a-8.9.0)
- F-Secure AntiVirus for SAMBA (fsavsamba-4.51-04011901)
- Mail Server (sendmail-8.12.8-9.90)
- OpenSSH Server (openssh-server-3.5p1-11)
- Text Editor (emacs-21.2-33)

The most part of policy files necessary for these applications are defined using accept mode. There may be some applications that are not suitable for operating in a read-only medium, but TOMOYO Linux itself can support arbitrary applications.

7. Conclusion

We think that the security enhanced Linux becomes popular and the MAC becomes no wonder in a near future. But the task of defining policies for MAC is very laborious, and it is hard to say that the MAC is manageable for everyone.

Our pathname-based MAC with accept mode described in this paper can automate the very laborious task of figuring out the necessary access permissions for files and shared libraries, and is easy to introduce and manage.

We also think that it is possible to improve security by just adding several lines to existent program's source code and recompiling, without requiring policy files for MAC.

The MAC described in this paper can control only files and directories, but we hope that this approach helps developing MAC in a straightforward way.

Acknowledgment: 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] Toshiharu HARADA, "Building secure systems." (Written in Japanese) Nikkei SYSTEM INTEGRATION vol. April 2004, no. 132.

[3] 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.

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

[5] Stephen Smalley et al. *Implementing SELinux as a Linux Security Module*

[6] 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

[7] 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

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

Notes

This is a translation of the original paper, which was written in Japanese and published in Linux Conference 2004 held in Japan. You can obtain the original paper from the following URL.

http://sourceforge.jp/projects/tomoyo/document/lc2004.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/