IEICE TRANS. INF. & SYST., VOL.E92–D, NO.11 NOVEMBER 2009

PAPER

Reducing Security Policy Size for Internet Servers in Secure Operating Systems

Toshihiro YOKOYAMA[†a)], **Miyuki HANAOKA**[†], **Makoto SHIMAMURA**[†], ***Nonmembers***, **Kenji KONO**[††], ***Member***, ***and*** **Takahiro SHINAGAWA**[†††], ***Nonmember***

SUMMARY Secure operating systems (secure OSes) are widely used to limit the damage caused by unauthorized access to Internet servers. However, writing a security policy based on the principle of least privilege for a secure OS is a challenge for an administrator. Considering that remote attackers can never attack a server before they establish connections to it, we propose a novel scheme that exploits *phases* to simplify security policy descriptions for Internet servers. In our scheme, the entire system has two execution phases: an initialization phase and a protocol processing phase. The initialization phase is defined as the phase before the server establishes connections to its clients, and the protocol processing phase is defined as the phase after it establishes connections. The key observation is that access control should be enforced by the secure OS only in the protocol processing phase to defend against remote attacks. Since remote attacks cannot be launched in the initialization phase, a secure OS is not required to enforce access control in this phase. Thus, we can omit the access-control policy in the initialization phase, which effectively reduces the number of policy rules. To prove the effectiveness of our scheme, we wrote security policies for three kinds of Internet servers (HTTP, SMTP, and POP servers). Our experimental results demonstrate that our scheme effectively reduces the number of descriptions; it eliminates 47.2%, 27.5%, and 24.0% of policy rules for HTTP, SMTP, and POP servers, respectively, compared with an existing SELinux policy that includes the initialization of the server.

key words: *secure operating system, SELinux, Internet server, policy description*

1. Introduction

Secure operating systems (secure OSes) are widely used to limit the damage caused by unauthorized access to Internet servers. For example, Hewlett-Packard Development Company (HP) developed the secure platform VirtualVault [11] based on Trusted HP-UX, the secure OS developed by HP to operate the Netscape Web server securely. PitBull LX [2] utilizes technologies based on secure OSes to operate the Apache Web server securely. A secure OS introduces mandatory access control (MAC), which can be used to ensure system-wide data confidentiality and integrity. In addition, a secure OS focuses on the principle of least privilege [18] by giving a process exactly the rights required to perform its given task. Therefore, even if a server is hijacked by attackers, they can only perform operations authorized to the server processes.

However, it is difficult to introduce and maintain a secure OS because of the complexity of its security policy configuration. In a secure OS, all subjects and objects are labeled, and the security policy in the system is defined with respect to these labels. To label subjects and objects, a server administrator is required to have expertise in the resources each process uses. Then, he or she needs to write a policy to grant access rights to the processes based on the principle of least privilege.

A particular challenge is that the policy configuration for the initialization process of the system is extremely complicated because of various transactions, such as booting, mounting file systems, and network configuration. Even if we focus on defending against remote attacks like targeted policy [10] available in SELinux [1], the policy configuration remains difficult. To defend effectively against a remote attack using a secure OS, we have to grant minimal access rights to processes running an Internet server to limit the damage when the server is hijacked by remote attackers. To accomplish this, we are required to have a detailed understanding of the behavior of the servers. However, a server initialization process has complicated steps, such as reading configuration files and linking libraries. Moreover, its initialization process tends to depend on its execution environment, such as an OS or the version of the server. When the kinds of resources that the server uses are changed (for example, by a version upgrade), it is necessary to reconfigure a security policy.

Our scheme focuses on the protection from remote attacks. Since secure OSes are often used to protect Internet servers from remote attacks, our scheme tries to simplify the policy description solely for the defense against remote attacks. In this paper, we leverages the fact that the server is never attacked by remote attackers before it establishes any connections to its clients. If we defend against a remote attack using a secure OS, access control by the secure OS needs to be enforced only after the server establishes connections.

Because remote attackers can never attack a server before it establishes connections to them, we propose a scheme that exploits *phases* to simplify a security policy description. One of the authors proposed simplifying security policy de-

Manuscript received March 18, 2009.
Manuscript revised June 28, 2009.

[†]The authors are with School for Open and Envionmental Systems, Graduate School of Science and Technology, Keio University, Yokohama-shi, 223–8522 Japan.

[††]The author is with the Department of Information and Computer Science, Faculty of Science and Technology, Keio University, Yokohama-shi, 223–8522 Japan.

[†††]The author is with the Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba-shi, 305–8573 Japan.

a) E-mail: burger@sslab.ics.keio.ac.jp

DOI: 10.1587/transinf.E92.D.2196

scriptions for sandbox systems by exploiting phases [21]. Our work applied this concept to secure OSes. Through our effort to apply the concept to secure OSes, this paper demonstrates the following three points. First, the phase concept can be incorporated into a wider range of secure platforms other than sandbox systems. Second, the engineering effort to incorporate the phase into SELinux is reasonable. Finally, and the most importantly, this paper shows the quantitative result of incorporating the phase concept into SELinux. Since the secure OSes are totally different from sandbox systems in their security models and mechanisms, the quantitative analysis shown in our previous paper is useless in the context of secure OSes.

We define the initialization phase as the phase before the server establishes any connections to its clients and the protocol processing phase as the phase after it establishes connections. There is no need to enforce access control in the initialization phase because remote attacks cannot be launched; it needs to be enforced only in the protocol processing phase. We can therefore eliminate the policy description before the server establishes connections. Additionally, we need not have detailed information about various transactions until the server establishes connections, which makes it easier to write the policy. Our scheme does not completely eliminate the need for the knowledge about server behaviors; rather, it reduces the amount of security policy descriptions.

We implemented our proposed scheme with SELinux and evaluated the effectiveness of our proposal by writing policies for three kinds of Internet servers (HTTP, SMTP, and POP servers). As a result, we were able to eliminate 47.2%, 27.5%, and 24.0% of the policy rules for HTTP, SMTP, and POP servers, respectively, compared with an existing SELinux policy which includes the initialization of the server.

The rest of the paper is structured as follows. In Sect. 2, we describe the main features of secure OSes and analyze the difficulty of writing policy for Internet servers. In Sect. 3, we introduce our proposal. In Sect. 4, we describe its implementation. In Sect. 5, we examine our proposal by comparing existing SELinux policy and our policy that exploits *phases* for three kinds of servers. In Sect. 6, we describe related work and in Sect. 7, we conclude the paper.

2. Motivation

Breaches of Internet servers routinely cause tremendous damage, such as destruction of private information or systems. Because only a handful of programmers have the mindset to write secure code, vulnerability of servers will continue to be a problem.

Secure OSes are widely used to prevent the damage caused by unauthorized access to Internet servers. For example, Hewlett-Packard Development Company (HP) developed the secure platform VirtualVault [11] based on Trusted HP-UX, the secure OS developed by HP to operate the Netscape Web server securely. PitBull LX [2] utilizes technologies based on secure OSes to operate the Apache Web server securely. Even if a server running on a secure OS were hijacked by attackers, they would only be able to perform operations authorized to the server processes.

However, the secure OS is known for the difficulty of its security policy configuration, which remains the biggest barrier to system administrators trying to introduce it. Unfortunately, few researchers are interested in simplifying policy descriptions for the secure OS, while a lot of work has been proposed to help analyze policy configuration [3], [7], [12], [13], [20], [25], [27].

In this paper, we tackle the problem of simplifying policy descriptions for Internet servers, which makes it easier to introduce secure OSes.

2.1 Secure OSes

A secure OS implements a mandatory access control (MAC), which focuses on providing an administratively-defined security policy that can control all subjects and objects, basing decisions on all security-relevant information. Subjects and objects are labeled, and the security policy in the system is defined with respect to these labels. In addition, the secure OS focuses on the principle of least privilege by giving a process exactly the rights it needs to perform its given task. Many secure OSes have been developed, such as SELinux [1], Trusted BSD [6], Trusted Solaris [14], and SEDarwin [26].

In this paper, we focus on simplifying SELinux policy descriptions for Internet servers. SELinux was originally a development project of the National Security Agency (NSA) and it implements a security architecture called Flask [22]. SELinux now implements three different security models: type enforcement (TE) [4], role based access control (RBAC) [19], and multi-level security (MLS) [8]. Among these security models, we focus on TE because it plays the predominant role in the access control.

In TE, a security attribute called *type* is bound to each subject and object. A type that is bound to a subject is also called *domain*. Objects are partitioned into *object classes* (file, directory, process, socket, etc.) and security-sensitive operations are divided into *access vectors* or *access modes* (read, write, getattribute, accept, etc.). A combination of *a user identity*, *a role*, and *a type* is called *a security context*. Whenever a subject tries to access an object in a given mode, it is checked whether that is authorized by the security policy according to *the security contexts* of the subject and the object involved in the access. The construct *allow* permits specification of the access rights granted in each domain. For example, the allow rule "allow httpd_t httpd_conf_t read;" grants the Apache HTTP server running in the domain httpd_t the access right on reading its configuration files whose type is httpd_config_t.

SELinux provides *security context transition* for changing security contexts of a subject and object. Security context transition for a subject can occur when a subject executes a program. The security context of a process

after transition is determined from the security contexts of the program and the process before the transition. Security context transition for an object occurs when a new object is created. The security context of a new object is determined from the security contexts of the subject and other related objects (for example, the parent directory for a file).

2.2 Difficulty of Policy Configuration

To grant minimal access rights to all system processes based on the principle of least privilege, we are required to have detailed knowledge of all daemons and applications running on a secure OS and write a policy for them with no mistakes. First, we need to label subjects and objects depending on the kinds of services or applications. Then, we write a policy for granting each subject exactly the access rights it needs to perform its given task.

Even if we focus on defending against remote attacks like the targeted policy [10] available in SELinux, policy configuration remains difficult. The targeted policy makes policy configuration easier than the strict policy [9] by focusing on defense against remote attacks. Its aim is to provide additional security compared with the standard Linux DAC for commonly used daemons, such as httpd, dhcpd, and mailman. To use a secure OS to defend against a remote attack, we have to grant minimal access rights to the domain bound to an Internet server to limit the damage when the server is hijacked by remote attackers. To accomplish this, we are required to have a detailed understanding of the behavior of the servers.

The initialization process of an Internet server is particularly complicated because the server process needs various kinds of resources that are scattered in a lot of directories. Making matters worse, these directories are difficult to determine from the server's primary transaction, such as dealing with web pages or e-mail. We use the initialization process of the Apache HTTP server as an example. Figure 1 shows the configuration rules for the initialization of Apache. Here, "initialization" means the transaction from the time the server program (httpd) is executed until Apache establishes any connections to its clients. This policy includes 30 types scattered in a lot of directories: /dev (device_t), /etc (etc_t), /lib (lib_t), /proc (proc_t), /root (root_t), /usr (usr_t), and /var (var_t). Apache's initialization has complicated steps, such as loading modules (httpd_module_t), linking libraries (lib_t), and reading configuration files (httpd_config_t, net_conf_t). Sixty out of the 83 rules shown in Fig. 1 are used only for the initialization of Apache, which means that the resources which Apache needs for initialization are significantly different from those needed after it establishes connections to its clients. After establishing a connection, Apache just returns the requested web pages to clients; these transactions are quite simple compared with the initialization. In addition, the resources used for initialization (for example, function libraries and configuration files) tend to depend on their execution environment, such as an OS or version of Apache, while content files used for providing services (for example, HTML files) do not depend on their execution environments. Moreover, whenever the resources needed for the server change (for example, because of a version upgrade), it is necessary to reconfigure the policy; we have to relabel objects and write a policy that grants access rights on operating server again.

3. Proposal

In this section we propose the simplification of security policy descriptions by exploiting *phases*.

3.1 Policy Description Exploiting Phases

To simplify security policy descriptions for Internet servers, we propose a scheme that exploits *phases*, dividing the lifetime of the server into the phase before the server establishes any connections and the phase after it establishes connections. Note that our scheme targets only on the protection from remote attacks. If the system is locally attacked (for example, by canonical users), our scheme provides the same security level as traditional Linux.

We define *the initialization phase* as the phase before the server establishes any connections to its clients and *the protocol processing phase* as the phase after it establishes connections. In the initialization phase, the server is untainted and can never be attacked by remote attackers, so access control need not be enforced, which results in reducing the number of access-control policy rules. On the other hand, access control needs to be enforced in the protocol processing phase because the server could be tainted and attacked.

Our scheme simplifies the description of security policies because the administrator does not have to write the policy for the initialization phase. This eliminates the need for the knowledge about server behaviors in the initialization phase, but it does not imply the need for the knowledge about the server behaviors is completely eliminated; the administrator must have the knowledge about the behavior in the protocol processing phase.

In our scheme, we write different security policies for each phase. For the initialization phase, we write a policy that grants all access to all domains because the server is not vulnerable to attack. For the protocol processing phase, we write a policy that grants minimal access rights based on the principle of least privilege to a domain bound to the server and grants all access rights to other domains.

Because access control is not enforced in the initialization phase, we need to transit the phase from the initialization phase to the protocol processing phase when the server establishes any connection. When *the phase transition* occurs, the security policy for the protocol processing phase is loaded and access control is enforced.

We define three kinds of fundamental events for the phase transition. One is the connect system call, which requests that a connection be established. Another is the accept system call, which accepts a connection, and the other

```
#the rule with (*) is used only for the initialization
allow httpd_t bin_t:dir search(*);
allow httpd_t console_device_t:chr_file { read(*) write(*) };
allow httpd_t device_t:dir search(*);
allow httpd_t etc_t:dir search;
allow httpd_t etc_t:file { getattr read };
allow httpd_t http_port_t:tcp_socket name_bind(*);
allow httpd_t httpd_config_t:dir search(*);
allow httpd_t httpd_config_t:file { getattr(*) read(*) };
allow httpd_t httpd_exec_t:file entrypoint(*);
allow httpd_t httpd_log_t:dir { add_name(*) remove_name(*) search(*) write(*) };
allow httpd_t httpd_log_t:file { append create(*) unlink(*) write(*) };
allow httpd_t httpd_modules_t:dir search(*);
allow httpd_t httpd_modules_t:file { execmod(*) execute(*) getattr(*) read(*) };
allow httpd_t httpd_sys_content_t:dir getattr;
allow httpd_t self:capability { net_bind_service(*) setgid setuid };
allow httpd_t self:netlink_route_socket { bind(*) create(*) getattr(*)
                                          nlmsg_read(*) read(*) write(*) };
allow httpd_t self:process { fork sigchld };
allow httpd_t self:sem { create(*) destroy(*) read setattr(*) unix_write write };
allow httpd_t self:tcp_socket { bind(*) create listen(*) setopt };
allow httpd_t self:udp_socket { connect(*) create getattr(*) };
allow httpd_t self:unix_stream_socket { connect(*) create(*) };
allow httpd_t init_t:fd use(*);
allow httpd_t init_t:process sigchld(*);
allow httpd_t ld_so_cache_t:file { getattr(*) read(*) };
allow httpd_t ld_so_t:file read(*);
allow httpd_t lib_t:dir search;
allow httpd_t lib_t:lnk_file read(*);
allow httpd_t locale_t:dir search(*);
allow httpd_t locale_t:file { getattr(*) read(*) };
allow httpd_t net_conf_t:file { getattr(*) read(*)};
allow httpd_t node_t:tcp_socket node_bind(*);
allow httpd_t node_unspec_t:tcp_socket node_bind(*);
allow httpd_t null_device_t:chr_file { read(*) write(*) };
allow httpd_t proc_t:dir search;
allow httpd_t random_device_t:chr_file read(*);
allow httpd_t root_t:dir search(*);
allow httpd_t shlib_t:file { execute(*) getattr(*) read(*) };
allow httpd_t sysctl_kernel_t:dir search;
allow httpd_t sysctl_kernel_t:file read;
allow httpd_t sysctl_t:dir search;
allow httpd_t urandom_device_t:chr_file { getattr(*) read(*) };
allow httpd_t usr_t:dir { getattr search };
allow httpd_t var_run_t:dir search(*);
allow httpd_t var_t:dir search(*);
```

Fig. 1 Configuration rules for the initialization of Apache.

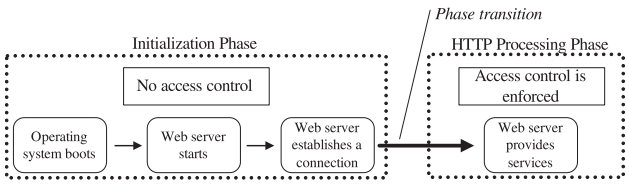


Fig. 2 The phase transition of Apache.

is the recvfrom system call, which is mainly used for receiving packets in UDP communication. Establishing a connection could lead to the server being tainted because the server may be attacked by remote attackers. Therefore, we need to detect the establishment of a connection and cause the phase transition from the initialization phase to the protocol processing phase to enforce access control by the secure OS.

For example, the phase transition of Apache is shown in Fig. 2. After an OS boots, the policy of the initialization phase that grants all access rights to all domains is loaded into a secure OS and access control is not enforced. Once Apache executes the accept system call and establishes a connection, the phase transition occurs from the initialization phase to the HTTP protocol processing phase. The policy of the HTTP processing phase is loaded into the secure OS, which grants minimal access rights to the domain bound to Apache and grants all access rights to other domains.

We developed a configuration file in XML for determining events that cause the phase transition. For example, a configuration file for causing the phase transition for Apache is shown in Fig. 3. The phase name before the transition (in this case, the initialization phase) is described in src-phase tag and the phase name after the transition (in this case, the HTTP processing phase) is in dst-phase tag. By naming all phases, we can easily understand the role of each phase. To specify that Apache causes the phase transition, the domain

bound to Apache (httpd_t) is written as an attribute value of event tag. To specify the event for the phase transition, the name of the system call for establishing a connection (accept) is written as a value of event tag. Condition tag can include a number of event tags to define a lot of events for the phase transition.

Figure 4 shows the configuration rules for the HTTP processing phase to run SPECweb2005 [24], the famous benchmark program for HTTP servers. The policy of the HTTP protocol phase includes 20 types, while 30 types are included in the initialization policy of Apache (shown in Fig. 1). Most of these 20 types are used for providing service: types used for communicating with users (http_port_t, port_t, netif_t, node_lo_t, node_t), and types used for reading content files (httpd_sys_content_t, httpd_php_exec_t). We can also understand the configuration rules of the HTTP processing phase more easily than the initialization of Apache because most of the configuration rules defined in the HTTP processing phase are for dealing with requests from clients. For example, the rules for granting the domain httpd_t rights on establishing connections to its clients, getting the requests from its clients, and reading HTML files (httpd_sys_content_t) are defined in the policy.

The phase transition occurs only once from the initialization to the protocol processing phase. However, there is an exception when two or more servers operate on one machine. Suppose that an SMTP server and a POP server operate on one machine. In this case, the phase transition occurs twice as shown in Fig. 5. If the SMTP server enters the protocol processing phase before POP server does, the SMTP policy becomes effective and the transition is made to the SMTP processing phase. Note that the POP policy is

```
<src-phase>
    Initialization phase
</src-phase>
<dst-phase>
    HTTP processing phase
</dst-phase>
<condition>
    <event domain="httpd_t">
        accept
    </event>
</condition>
```

Fig. 3 Example of the configuration file for Apache.

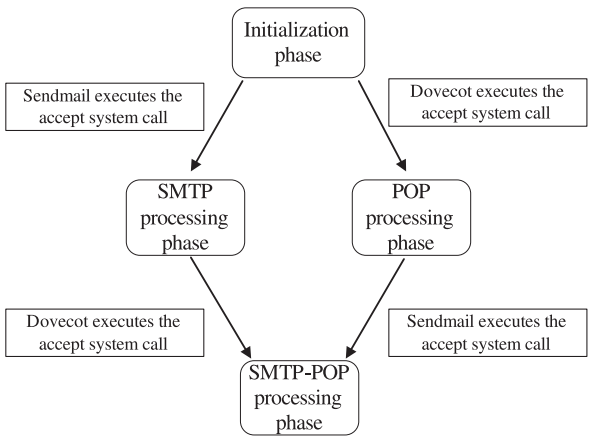


Fig. 5 The phase transition of Sendmail and Dovecot running on one machine.

```
allow httpd_t etc_t:dir search;
allow httpd_t etc_t:file { getattr read };
allow httpd_t http_port_t:tcp_socket { name_connect recv_msg send_msg };
allow httpd_t httpd_log_t:file { append lock };
allow httpd_t httpd_php_exec_t:file { getattr read write };
allow httpd_t httpd_sys_content_t:dir { getattr search };
allow httpd_t httpd_sys_content_t:file { getattr read };
allow httpd_t httpd_tmp_t:file { create getattr lock read unlink write };
allow httpd_t self:capability { setgid setuid };
allow httpd_t self:fifo_file { read write };
allow httpd_t self:process { fork sigchld };
allow httpd_t self:sem { read unix_write write };
allow httpd_t self:tcp_socket { accept connect create getattr getopt
                                read setopt shutdown write };
allow httpd_t self:udp_socket create;
allow httpd_t lib_t:dir search;
allow httpd_t lib_t:file { getattr read };
allow httpd_t netif_t:netif { tcp_recv tcp_send };
allow httpd_t node_lo_t:node { tcp_recv tcp_send };
allow httpd_t node_t:node { tcp_recv tcp_send };
allow httpd_t port_t:tcp_socket { recv_msg send_msg };
allow httpd_t proc_t:dir search;
allow httpd_t reserved_port_t:tcp_socket { name_connect recv_msg send_msg };
allow httpd_t root_t:dir search;
allow httpd_t sysctl_kernel_t:dir search;
allow httpd_t sysctl_kernel_t:file read;
allow httpd_t sysctl_t:dir search;
allow httpd_t tmp_t:dir { add_name getattr read remove_name search write };
allow httpd_t usr_t:dir { getattr search };
```

Fig. 4 Configuration rules for the HTTP processing phase.

not enabled at this time of point. After that, when the POP server establishes a connection, the POP policy is enabled and the transition is made to SMTP-POP processing phase, in which both policies for SMTP and POP are effective.

3.2 Security Concerns and Defense Mechanisms

We assume that an attacker can contrive an attack tailored to our scheme. Suppose a server running in a protocol processing phase is hijacked and attackers craft a malicious code into files which they can access with write or append permission. Next, the machine is restarted, and if the server process reads the compromised files in the initialization phase, the server might execute the code and suffer damage.

However, this type of attack does not pose a big threat to our scheme because we can defend against it completely with a taint-based approach. We regard all of the files written/appended by the processes that established connections with clients as *tainted*. And we add the *read* system call to one of the system calls for starting phase transition. After the machine is restarted, if a tainted file is to be read in the initialization phase, a server transits to the processing phase. Thus, we can defend against such an attack.

This taint mechanism might decrease the merit of our scheme because the phase transition caused by the read system call occurs earlier compared with one caused by one of the other three system calls (accept, connect, and recvfrom). However, this mechanism does not have a large influence on the availability of our scheme because the server reads the compromised files in the initialization file only rarely.

We investigated three kinds of real Internet servers (HTTP, SMTP, and POP servers) by running SPECweb2005 [24] with three-workload benchmark designs (banking, ecommerce, and support) for the HTTP server, and SPECmail2001 [23] for the SMTP and POP servers. We used Apache 2.2.6 for the HTTP server, Sendmail 8.13.8 for the SMTP server, and Dovecot 0.99.14 for the POP server. In this investigation, we found no case in which the tainted files were read in the initialization phase.

Even if the phase transition is caused by the read system call, the number of policy rules in our scheme are still fewer than existing other schemes (for example, targeted policy available in SELinux). Therefore, we believe our scheme can be of great help to system administrators introducing secure OSes for protecting Internet servers.

4. Implementation

We implemented our proposed scheme into Fedora Core 4 with SELinux and Linux kernel 2.6.18 to evaluate the effectiveness of our proposal. Our system consists of three components: a *phase management module*, a *policy load daemon*, and a *phase transition database*. The architecture of our system is shown in Fig. 6.

The phase management module is the kernel module which hooks the connect, accept, recvfrom, and read system calls to check whether a phase transition occurs, changes the current phase to the new one, and makes a request for loading a new policy to the policy load daemon. Additionally, when the phase management module is loaded, it creates a phase transition database that stores conditions of phase transitions in the kernel memory by receiving the contents of the configuration files from user space.

For implementing a taint-based defense mechanism for an attack tailored to our scheme, when the phase management module is loaded, it also receives the list of tainted files stored on a regular file from user space and copies it in the kernel memory. By hooking the write system call, if a file is written in the protocol processing phase by the process that establishes connections to clients, the phase management module updates the list of tainted files stored in both kernel memory and in the regular file.

The policy load daemon is the daemon which loads a policy specified by the phase management module. Since SELinux is designed to load a policy in user space, we developed this daemon by utilizing a load_policy program available in SELinux to simplify our implementation.

When the connect or accept or recvfrom or read system call is executed, the phase management module hooks it and checks whether a phase transition occurs by referring to the phase transition database. If there is a phase transition, the phase management module makes a request to the policy

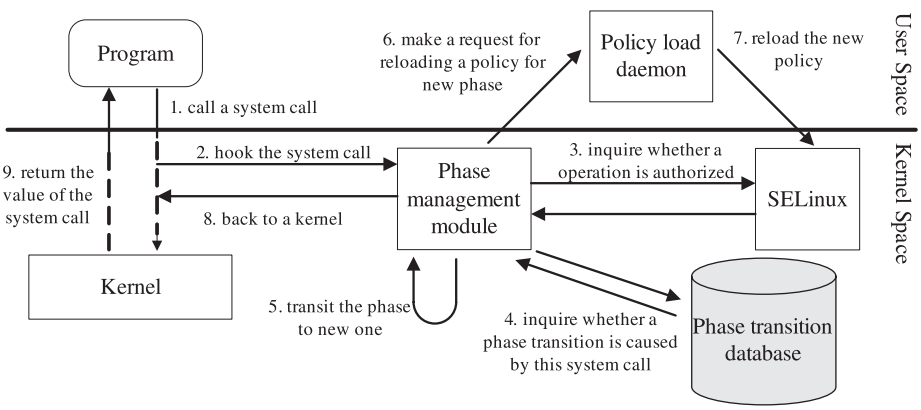


Fig. 6 Architecture of our system.

load daemon to load a new policy for the protocol processing phase, and the policy load daemon loads it into SELinux.

5. Experiments

We wrote security policies for three kinds of Internet servers (HTTP, SMTP, and POP servers) with the targeted policy available in SELinux and our policy that exploits phases, and then analyzed them to evaluate the effectiveness of our proposal. We do not address whether the policies we refactored are properly configured, because the goal of our work is to present a scheme for reducing the number of policy rules. In addition, we wrote the security policy for performing all of the transactions from the time the OS boots until the server program is executed with the strict policy, selecting daemons which are necessary for the management of Internet servers. Finally, we compared the performance of SELinux and one with our module by running benchmark programs. We used Apache 2.2.6 for the HTTP server, Sendmail 8.13.8 for the SMTP server, and Dovecot 0.99.14 for the POP server.

5.1 Simplification of the Policy Description

We wrote the security policies for three kinds of Internet servers (HTTP, SMTP, and POP servers) with the targeted policy and our policy that exploits phases (HTTP, SMTP, and POP processing phase) running SPECweb2005 [24] for the HTTP server, and SPECmail2001 [23] for the SMTP and POP servers.

We defined phase transition events for these servers as follows. Apache runs in the domain httpd_t and transits from the initialization phase to the HTTP processing phase when it executes the accept system call. Sendmail runs in the domain sendmail_t and transits from the initialization phase to the SMTP processing phase when it executes the accept system call. Dovecot runs in the domain dovecot_t for dealing with requests from clients and in the domain dovecot_auth_t for authenticating clients. When a client sends a request to Dovecot, the process running in the domain dovecot_auth_t establishes connections to the client first. Therefore, when the process running in the domain dovecot_auth_t executed the accept system call, we defined that Dovecot transited from the initialization phase to the POP processing phase.

Under these conditions, we wrote the policies with the targeted policy and our policy that exploits phases. Tables 1 and 2 summarize the results of policy descriptions.

Table 1 shows the number of types necessary for writing each policy. As for Apache, 19 types are necessary and 18 types are eliminated in our policy compared with the targeted policy, including httpd_modules_t, httpd_config_t, net_conf_t, device_t, and so on. For Sendmail, six types are eliminated in our policy, including sendmail_var_run_t, net_conf_t, var_spool_t, ld_so_cache_t, and so on. For Dovecot, three types are eliminated in our policy, including dovecot_etc_t, null_device_t, and devpts_t. This result means that these eliminated types are used only for the initialization of the servers and never for providing services to clients. Therefore, with our policy, server administrators do not have to label objects bound to these types.

Table 2 shows the number of access-control rules defined by the construct *allow*. Table 2 indicates that the number of total rules in our policy are fewer than in the targeted policy for all three Internet servers. The number of rules eliminated for Apache is particularly outstanding compared with other servers. This is because Apache has simple behavior in the HTTP processing phase; Apache just gets requests from clients and returns requested pages to them in the HTTP processing phase, while various transactions are performed in the initialization phase, such as loading modules, linking libraries, and reading configuration files.

Sendmail and Dovecot perform more transactions than Apache in the protocol processing phase. For example, Sendmail accesses mail queues, relays mail to other machines, and accesses the /proc directory to change its behavior based on the load average of the system in the SMTP processing phase. As for Dovecot, because it is designed to dynamically link libraries every time it authenticates clients, it frequently accesses libraries even in the POP processing phase. For these reasons, the number of rules eliminated for Sendmail and Dovecot are fewer than for Apache.

This experimental result suggests that a highly modularized server such as Dovecot benefits less from our scheme. To reduce the damage of remote attacks, Dovecot divides its functionalities into multiple processes. If we extend our scheme to deal with application-specific phases, monolithic (not-modularized) servers can be secured as in Dovecot because the server process is given the access rights required strictly in the current phase.

Table 3 shows the details of reduced access-control rules classified by access vectors in our policy compared with the existing SELinux policy. Many access vectors related to file transactions (read, getattr, search, write) are eliminated in our policy. In addition, access vectors related to preparation for establishing a connection (bind, listen, connect) are also eliminated.

Table 4 shows the list of access vectors that are not reduced in our policy, which indicates that they are used after the initialization of the server. These access vectors are relatively easily determined from the role of the server because

Table 1 Comparison of types.

	Apache	Sendmail	Dovecot
targeted policy	37	38	34
our policy	19	32	31

Table 2 Comparison of allow rules.

	Apache	Sendmail	Dovecot
targeted policy	127	149	171
our policy	67	108	130
eliminated rules	60	41	41
% of rules eliminated	47.2%	27.5%	24.0%

Table 3 List of access vectors reduced in our policy.

	Apache		Sendmail		Dovecot	
	targeted policy	our policy	targeted policy	our policy	targeted policy	our policy
read	23	10	24	16	37	28
getattr	18	9	32	24	22	21
search	17	9	18	16	26	24
write	11	6	14	8	15	11
create	7	3	6	4	9	5
connect	3	1	4	3	4	3
unlink	2	1	1	1	1	1
bind	2	0	1	0	3	1
execute	2	0	3	2	5	4
node_bind	2	0	1	0	1	0
add_name	2	1	2	1	2	1
remove_name	2	1	1	1	2	1
setattr	1	0	0	0	2	1
destroy	1	0	0	0	0	0
entrypoint	1	0	1	0	2	0
execmod	1	0	0	0	0	0
listen	1	0	1	0	2	0
name_bind	1	0	1	0	1	0
net_bind_service	1	0	1	0	1	0
nlmsg_read	1	0	0	0	1	1
use	1	0	1	0	2	1
lock	2	2	6	4	2	2
setopt	1	1	1	0	1	1
ioctl	0	0	3	1	0	0
noatsecure	0	0	1	1	1	0
rename	0	0	1	1	1	0
rlimitinh	0	0	1	1	1	0
sendto	0	0	1	1	2	1
siginh	0	0	1	1	1	0
chown	0	0	0	0	1	0

Table 4 List of access vectors not reduced in our policy.

	Apache	Sendmail	Dovecot
recv_msg	3	3	1
send_msg	3	3	1
tcp_recv	3	2	2
tcp_send	3	2	2
name_connect	2	1	0
sigchld	2	2	4
accept	1	1	2
append	1	0	0
fork	1	1	2
getopt	1	0	0
setgid	1	1	1
setuid	1	1	1
shutdown	1	0	0
unix_write	1	0	0
signal	0	1	0
execute_no_trans	0	0	1
setrlimit	0	0	1
sys_chroot	0	0	1
audit_control	0	0	1
audit_write	0	0	1
nlmsg_relay	0	0	1

most of them are used for dealing with requests from clients.

We also wrote a security policy for performing all of the transactions from the time the OS boots until the server program is executed with the strict policy, selecting the daemons necessary for the management of Internet servers. We selected following daemons: acpid, anacron, apmd, atd, auditd, autofs, cpuspeed, crond, haldaemon, iptables, kudzu, messagebus, netfs, network, rhnsd, sshd, and syslog. As a result, 126 types and 1219 rules are necessary for writing the policy. Compared with Tables 1 and 2, these rules are numerous. We do not have to write this policy at all in our scheme.

5.2 Performance

We measured performance of HTTP, SMTP, and POP servers running on SELinux and one with our module. We used SPECweb2005, selecting an ecommerce application for the HTTP server and SPECmail2001 for the SMTP and POP servers as benchmark programs. We call the SELinux with our module *our system* in this section.

Tables 5, 6, and 7 show the results. We can confirm that our system has almost the same performance as SELinux, although the performance of the POP server running on our system is slightly worse than the HTTP and SMTP servers.

6. Related Work

Security models have two goals: preventing accidental or malicious destruction of information or systems and controlling the release and propagation of that information. A secure OS can accomplish both goals. For example, SELinux

Table 5 HTTP server performance.

	SELinux		Our System	
Request Type	Response time (sec)	Throughput (kbytes)	Response time (sec)	Throughput (kbytes)
index	1.652	161.7	1.652	161.7
search	1.794	177.1	1.794	177.1
browse	1.786	175.2	1.786	175.2
browse_productline	1.837	177.8	1.838	177.8
productdetail	0.919	55.0	0.919	55.1
customize1	1.688	166.9	1.688	166.9
customize2	1.683	166.2	1.683	166.2
customize3	1.826	179.9	1.826	179.9
cart	0.794	76.4	0.799	77.0
login	0.482	44.9	0.501	46.7
shipping	0.457	42.8	0.456	42.8
billing	0.375	34.5	0.375	34.5
confirm	0.348	33.4	0.348	33.4
Total	1.438	138.2	1.443 (+0.35%)	138.7 (+0.36%)

Table 6 SMTP server response time.

Function	SELinux (ms)	Our system (ms)
SMTP Connect	77.45	79.14 (+2.2%)
SMTP Hello	88.06	89.66 (+1.8%)
SMTP Mail From	15.66	15.35 (−2.0%)
SMTP Rcpt To	19.79	19.23 (−2.8%)
SMTP Data	90.21	89.74 (−0.5%)
SMTP Quit	14.48	14.65 (+1.2%)

Table 7 POP server response time.

Function	SELinux (ms)	Our system (ms)
POP Connect	33.53	36.57 (+9.0%)
POP User ID	43.82	47.86 (+9.2%)
POP Password	702.05	702.50 (+0.1%)
POP Status	210.56	243.31 (+15.6%)
POP Retrieve	12.18	12.87 (+5.7%)
POP Delete	9.24	9.45 (+2.3%)
POP Quit	26.02	29.25 (+12.4%)

enforces a wide range of security policies including type enforcement that accomplishes the former goal and multi-level security [8] that accomplishes the latter. However, its policy configuration is potentially error-prone because its policy size is too large to be configured correctly by hand. Researchers have explored techniques that can be used more easily than secure OSes. Some ideas are proposed for enforcing information flow policies with higher assurance based on a programming language or an operating system approach.

Jif [15] is a programming language that provides static checking of information flow using a decentralized label model. Jif can track information flow at the level of individual variables and perform most label checks at compile time. The Asbestos operating system [5] controls information flow by providing protection through its original labeling scheme, which allows data to be sanitized by individual users. HiStar [28] was directly inspired by Asbestos, but differs in that it provides system-wide persistence.

Information flow policies are useful and important, but not enough to completely ensure a system's security. For example, an attack aimed at destroying a system cannot be prevented solely by information flow policies. In addition, these approaches are difficult to use in real situations because Jif requires programmers to have specialized knowledge about it, and Asbestos and HiStar require large modifications of kernel code and user-level library code.

There are some secure OSes whose goal is to simplify the policy description. AppArmor [17] is an application security tool designed to provide an easy-to-use security framework. AppArmor security policies completely define what system resources each application can access with what privileges. Using a combination of static analysis and learning-based tools, AppArmor security policies can be defined much easier than SELinux. TOMOYO Linux [16] simplifies the policy description by learning the behavior of target applications. By doing this, TOMOYO Linux automatically generates almost 90% of security policies.

Although our scheme is currently implemented for SELinux, it can be applied to AppArmor and TOMOYO Linux; we can introduce the phases into these secure OSes. If we use these OSes instead of SELinux, the policy description for the protocol processing phase would be simplified compare with SELinux-based systems. For example, if our scheme is based on TOMOYO Linux, the 90% of the security policy would be automatically generated by TOMOYO Linux; the administrator must write the policy for the remaing 10% for the protocol processing phase. Note that the administrator does not have to write the policy for the initialization phase.

Unfortunately, few researchers are interested in simplifying policy descriptions, while a lot of work on secure OSes has been done to help analyze policy configuration [3], [7], [12], [13], [20], [25], [27].

Jaeger *et al.* [13] presented the concept of an access control space and developed a tool called Gokyo to analyze it. An access control space represents the permission assignment of a subject and contains all permissions divided into subspaces (prohibited, permissible, and unknown spaces). Gokyo computes the unknown subspace to show system administrators the ambiguous region and enables them to reduce it.

Zanin and Mancini [27] present a formal framework

called *SELinux Access Control* (SELAC) for analyzing an arbitrary security policy configuration. They define semantics for the constructs of the SELinux configuration language and model the relationships occurring among sets of configuration rules. By using SELAC formalism, it is possible to write algorithms for checking properties of given configurations.

Hicks *et al.* [12] present the first logical specification for modeling SELinux MLS policy and implement it in an analysis tool called PALMS (for Policy Analysis using Logic for MLS in SELinux). PALMS takes two policies in SELinux MLS policy syntax and automatically determines all the information flows allowed in the policies as well as whether one policy is compliant with the other.

Lujo *et al.* [3] presented a way to eliminate a large percentage of access-control policy misconfigurations before attempted accesses. They used a data-mining technique called association rule mining and applied it to the history of accesses to predict changes to access-control policies that are likely to be consistent with a user's intention.

Apol [25] is a graphical tool to analyze a SELinux policy developed by Tresys Technology. Some of the features supported are the ability to browse and search policy components, such as types, attributes, and roles, search through type enforcement and other rules, and view file contexts from a file system. Apol is also useful for our scheme. Even if our scheme is applied, the administrator must write a security policy for the protocol processing phase. Apol helps us analyze and write the policy for the protocol processing phase.

Indeed, helping analyze policy configuration becomes helpful for introducing secure OSes, but it cannot be the fundamental solution for making policy configuration easier. Writing a policy is the first step for introducing secure OSes and remains the biggest barrier to system administrators trying to introduce secure OS. Our work differs from these previous research in aiming at reducing the policy size itself. Reducing policy size also helps analyze the policy because the simpler a policy becomes, the more easy its effect on the system can be understood.

7. Conclusion

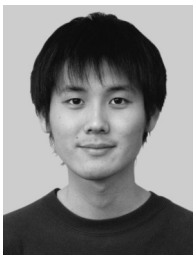
In this paper, we have proposed simplifying security policy descriptions for Internet servers by exploiting *phases*. Considering that remote attackers never attack against a server before they establish connections to it, we define the initialization phase as the phase before it establishes any connections and the protocol processing phase as the phase after it establishes connections. Access control by a secure OS is enforced only in the protocol processing phase. Thus, we can eliminate a policy description in the initialization phase. We wrote policies for three kinds of Internet servers (HTTP, SMTP, and POP servers) and our scheme was able to reduce 47.2%, 27.5%, and 24.0% of policy rules for each server respectively, compared with an existing SELinux policy that includes the initialization of the server.

References

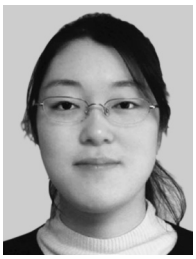
[1] National Security Agency, Security-enhanced linux. http://www.nsa.gov/selinux/
[2] Argus System Group Inc., Pitbull lx. http://www.argus-system.com/
[3] L. Bauer, S. Garriss, and M.K. Reiter, "Detecting and resolving policy misconfigurations in access-control systems," Proc. 13th ACM Symposium on Access Control Models and Technologies (SACMAT '08), pp.185–194, 2008.
[4] W.E. Boebert and R.Y. Kain, "A practical alternative to hierarchical integrity policies," Proc. 8th National Computer Security Conference, 1985.
[5] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," Proc. Twentieth ACM Symposium on Operating Systems Principles (SOSP '05), pp.17–30, 2005.
[6] FreeBSD Foundation, Trusted bsd. http://www.trustedbsd.org/
[7] J.D. Guttman, A.L. Herzog, J.D. Ramsdell, and C.W. Skorupka, "Verifying information flow goals in security-enhanced linux," J. Comput. Secur., vol.13, no.1, pp.115–134, 2005.
[8] C. Hanson, "Selinux and mls: Putting the pieces together," Technical report, NAI-02-007, 2006.
[9] Red Hat, Strict policy. http://www.redhat.com/
[10] Red Hat, Targeted policy. http://www.redhat.com/
[11] Hewlett-Packard Development Company, Virtualvault. http://www.hp.com/
[12] B. Hicks, S. Rueda, L.St. Clair, T. Jaeger, and P. McDaniel, "A logical specification and analysis for selinux mls policy," Proc. 12th ACM Symposium on Access Control Models and Technologies (SACMAT '07), pp.91–100, 2007.
[13] T. Jaeger, A. Edwards, and X. Zhang, "Managing access control policies using access control spaces," Proc. Seventh ACM Symposium on Access Control Models and Technologies (SACMAT '02), pp.3–12, 2002.
[14] Sun Microsystems, Trusted solaris. http://www.sun.com/
[15] A.C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," ACM Trans. Comput. Syst., vol.9, no.4, pp.410–442, 2000.
[16] NTT Data Corporation, Tomoyo linux. http://tomoyo.sourceforge.jp/
[17] openSUSE, Apparmor. http://en.opensuse.org/AppArmor
[18] J.H. Saltzer and M.D. Schroeder, "The protection of information in computer systems," Proc. IEEE, vol.63, no.19, pp.1278–1308, 1975.
[19] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman, "Role-based access control models," Computer, vol.29, no.2, pp.38–47, 1996.
[20] B. Sarna-Starosta and S.D. Stoller, "Policy analysis for security-enhanced linux," Proc. 2004 Workshop on Issues in the Theory of Security (WITS '04), pp.1–12, 2004.
[21] T. Shinagawa, Y. Chubachi, K. Kono, and K. Kato, "Simplifying security policies by exploiting execution phases," IPSJ Trans. Advanced Computing Systems (ACS), vol.2, no.2, pp.166-177, 2009.
[22] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The flask security architecture: System support for diverse security policies," Proc. Eighth USENIX Security Symposium (Security '99), pp.123–139, 1999.
[23] Standard Performance Evaluation Corporation, Specmail2001. http://www.spec.org/mail2001/
[24] Standard Performance Evaluation Corporation, Specweb2005. http://www.spec.org/web2005/
[25] Tresys Technology, Apol. http://oss.tresys.com/
[26] C. Vance, T. Miller, and R. Dekelbaum, "Security-enhanced darwin: Porting selinux to mac os x," Proc. Third Annual SEcurity Enhanced Linux Symposium, 2007.
[27] G. Zanin and L.V. Mancini, "Towards a formal model for security policies specification and validation in the selinux system," Proc.

Ninth ACM Symposium on Access Control Models and Technologies (SACMAT '04), pp.136–145, 2004.

[28] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," Proc. Sixth Symposium on Operating Systems Design and Implementation (OSDI '06), 2006.



Toshihiro Yokoyama received the B.E. degree from Keio University in 2007. He is currently a master student in the Graduate School of Science and Technology, Keio University. His current research interests include system software and Internet security.



Miyuki Hanaoka received her B.E. degree from the University of Electro-Communications in 2005, and M.E. from Keio University in 2007. She is currently a Ph.D. candidate in the Graduate School of Science and Technology, Keio University. Her research interests include network security and system software. She is a student member of IEEE, ACM, and IPSJ.

Makoto Shimamura received his B.E. degree from University of Electro-Communications in 2005, and Master of Engineering from Keio University in 2007, respectively. Currently, he is a Ph.D. candidate in the Graduate School of Science and Technology, Keio University. His current research interests include system software and Internet security. He is a student member of IEEE, ACM, and IPSJ.

Kenji Kono received the B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.

Takahiro Shinagawa is an assistant professor at University of Tsukuba. He received the Ph.D. degree from the University of Tokyo in 2003. His research interests include operating systems, virtual machine monitors, system software, and software security. He is a member of the IEEE/CS, ACM and USENIX.