IPSJ Journal Vol. 51 No. 9 1670–1681 (Sep. 2010)

Regular Paper

SELinux Security Policy Configuration System with Higher Level Language

Yuichi Nakamura,[†1] Yoshiki Sameshima[†1]
and Toshihiro Yamauchi[†2]

Creating security policy for SELinux is difficult because access rules often exceed 10,000 and elements in rules such as permissions and types are understandable only for SELinux experts. The most popular way to facilitate creating security policy is refpolicy which is composed of macros and sample configurations. However, describing and verifying refpolicy based configurations is difficult because complexities of configuration elements still exist, using macros requires expertise and there are more than 100,000 configuration lines. The memory footprint of refpolicy which is around 5 MB by default, is also a problem for resource constrained devices. We propose a system called SEEdit which facilitates creating security policy by a higher level language called SPDL and SPDL tools. SPDL reduces the number of permissions by integrated permissions and removes type configurations. SPDL tools generate security policy configurations from access logs and tool user's knowledge about applications. Experimental results on an embedded system and a PC system show that practical security policies are created by SEEdit, i.e., describing configurations is semi-automated, created security policies are composed of less than 500 lines of configurations, 100 configuration elements, and the memory footprint in the embedded system is less than 500 KB.

1. Introduction

In order to prevent intrusion to a system by attackers such as malwares and crackers, virus check software and updating software are commonly used. However, since they require pattern files and security patches which must be distributed before attacks, they are not effective to zero-day attacks. Once attackers obtain root privilege by exploiting security holes of services running as root, or by exploiting vulnerabilities leading to privilege escalation [1),2)], they can do everything in traditional Linux. In order to limit root privilege, Security-Enhanced Linux (SELinux) [3),4)] provides mandatory access control where all processes including root processes can access no resources unless access rules are described in the security policy. SELinux adopts TE (Type-Enforcement) [5)] mandatory access control model where *domain* labels are assigned to processes and *type* labels are assigned to resources such as files and ports. Set of access control rules (what kind of domain can access what kind of type) is called as *security policy*. As long as the security policy is configured properly, all processes including root and malicious processes have only limited access rights. As a result, the damage by attackers is confined even when zero-day attack happens. Because of this confinement feature, SELinux is included in major Linux distributions [6)], and is used for servers that require high level security. SELinux is also useful for network connected embedded devices such as cell phones and TVs. Actually, some Linux distributions for embedded system include SELinux [7)].

To deploy SELinux to a system, a security policy must be created. However, creating security policy for SELinux is difficult because access rules often exceed 10,000 and elements in rules such as permissions and types are understandable only for SELinux experts. The most popular method to facilitate creating security policy is customizing refpolicy (Reference Policy) [8),9)] which is composed of macros and sample configurations. Refpolicy can be applied with almost no customization when configurations for applications in a target system are included in refpolicy. For example, refpolicy is almost perfectly configured for default applications included in Fedora and CentOS. However, customizing refpolicy is required for systems where refpolicy is not configured enough. For example, refpolicy is not configured for commercial package software, custom applications and embedded systems.

There are three problems with customization. First, it is difficult to describe configurations because there are more than 700 permissions and 1,000 macros. In addition, type labels must be associated with file names and network resources. Second, it is difficult to verify refpolicy. Since refpolicy is intended for multiple use cases, more than 100,000 lines of configurations are included. When engineers verify refpolicy before reuse, they have to review such a lot of configurations. Third is the problem of resource consumption. When SELinux is

†1 Hitachi Software Engineering Co., Ltd.
†2 Graduate School of Natural Science and Technology, Okayama University

applied to resource constrained systems such as embedded systems, the files used and memory consumed by the security policy are a problem because refpolicy is large.

This paper proposes a security policy configuration system SELinux Policy Editor (SEEdit) that facilitates creating security policy by a higher level language called Simplified Policy Description Language (SPDL) and SPDL tools.

- SPDL
  Instead of complicated macros, we propose a higher level language called SPDL. SPDL simplifies describing and verifying SELinux security policy configurations with two features. Firstly, integrated permissions in SPDL reduce the number of permissions by grouping related SELinux permissions. Secondly, it removes type configurations by identifying resources with names such as path name and port number.
- SPDL tools
  To solve the verification and size problems of refpolicy, the security policy is created by writing only the necessary configurations in SPDL without refpolicy. SPDL tools help the writing process by generating configurations using access logs and knowledge of users about applications.

The remaining of this paper is organized as follows. Problems in creating security policy (Section 2), approaches of SEEdit to facilitate creating security policy (Section 3) are explained. The detail of SEEdit (Section 4), experimental results (Section 5) are shown. Finally, related works (Section 6), summary (Section 7) and future works (Section 8) are described.

2. Problems in Creating Security Policy

In this section, problems in creating a security policy for a target system based on refpolicy are described after an overview of SELinux policy language, and its difficulty.

2.1 SELinux Policy Language

The security policy is loaded to SELinux kernel in binary representation. However, it is hard to handle the binary security policy because it is unreadable for humans. To represent the security policy in text, SELinux has a basic policy language [10] and which is mainly composed of four syntax elements shown in

```
(1) Type assignment
<file name>  system_u:object_r:<type>
portcon <port number> system_u:object_r:<type>
netifcon <NIC name> system_u:object_r:<type>
(2)  Label declaration
type <type or domain>, <attribute>;
(3) Allowing access
allow <domain> <type> <permission>;
(4) Conditional policy expression
if(<parameter>){<statement>}
```

Fig. 1 Main components of SELinux Policy Language.

Fig. 1. SELinux identifies resources by labels called *type*. Types are assigned to resources such as files, port number and NICs (Fig. 1 (1)). Domains and types must be declared by *type* statements (Fig. 1 (2)). <type or domain> inherits configurations described for <attribute>. The following statements are example of attribute. admin_t can read both httpcontent_t and ftpcontent_t.

```
type httpcontent_t, content;
type ftpcontent_t, content;
allow admin_t content:file read;
```

The *allow* statements (Fig. 1 (3)) comprise access rules, i.e., domains are permitted to access types using some permissions. <permission> is composed of *object classes* and *access vector permissions*. Object class means classification of resources such as *file* (normal file), *dir* (directory) and *tcp_socket* (TCP socket). For each object class, access vector permissions such as *read* and *write* are defined. For example, permission *file read* means reading normal files, *dir read* means reading directories. Conditional policy expression (Fig. 1 (4)) is written to support multiple use cases in one security policy. <statement> is effective only when <parameter> is true. For instance, when CGI is necessary, the parameter *httpd_enable_cgi* is set true, then rules related to using CGI are enabled.

2.2 Difficulties in Writing Security Policy

The fine grained access control of SELinux is effective to prevent behavior of attackers, but there are difficulties in writing security policy.

2.2.1 Amount of Access Rules

To grant enough permissions for applications to work correctly, a lot of access

rules must be configured because the SELinux security policy is white list which means applications are not granted any permissions unless rules are described. In addition, more than 700 permissions make the amount much larger. In fact, the total number of access rules in a security policy often becomes more than 10,000, and sometimes exceeds that number.

2.2.2 Describing Individual Access Rules

Access rules are composed of *allow* statements, but it is hard for engineers to write individual rules because of permission and types.

(1) Configuring permissions
Since SELinux permissions are designed from the view point of system calls, engineers are required to have knowledge of Linux kernel. For example, when engineers want to grant an application to use TCP ports. Permissions related to system call such as *listen*, *accept*, *bind* and so on should be configured. Additionally, the number of permissions which exceed 700 makes using permissions more difficult.

(2) Configuring types
Engineers have to assign types to all files and ports in a system. This means, for each file, they have to group some files and ports appropriately, then name types, and assign types. The work is very hard because there are more than 10,000 files in standard Linux systems.
In addition, there are two more difficulties in types. First, engineers have to get used to types because they have been identifying files by file names not types in traditional Linux. Secondly, there is also a problem of dependency in assigning new types. This problem is explained with an example. When the foo_t type is assigned under /foo directory and the bar_t domain is allowed to read the foo_t type, the bar_t domain can read all files under the /foo directory. If the foo2_t type is newly created, and assigned to the file /foo/foo2, then the bar_t domain can not access /foo/foo2 because the bar_t domain is not allowed to access foo2_t. In this way, the bar_t domain was able to read /foo/foo2 before assigning the new type foo2_t, but bar_t can not access /foo/foo2 after the new type is assigned to /foo/foo2.

2.3 Overview of Refpolicy

Because of the difficulties explained above, it is not realistic to create security policy from nothing. The most popular way to facilitate creating security policy is *refpolicy* which is developed and maintained by the SELinux community. Refpolicy is composed of macros and configurations for typical applications.

```
#Macro definition
define(‘r_file_perms’,‘file { read getattr lock ioctl }’)
#Example usage of the macro
allow httpd_t contents_t r_file_perms;
```

Fig. 2 Example of an m4 macro.

(1) Macros
M4 [11] macros are defined to describe frequently used phrases in short words. For instance, **Fig. 2** shows *r_file_perms* macro, which is expanded to permissions related to reading regular files.

(2) Configurations for typical applications
Configurations for applications shipped with Linux distributions are prepared by the SELinux community and Linux distributors, and they are included in refpolicy. **Figure 3** is part of the configuration for the ftp daemon. There are many macros, such as *init_daemon_domain*, *miscfiles_read_public_files* and so on. In the figure, conditional expressions are omitted, but in fact, many conditional expressions are also included because refpolicy is intended to support as many use cases as possible, such as ftp login, and DB connection.

2.4 Problems in Creating Security Policy Using Refpolicy

Customizing refpolicy is necessary when the system usage case of the system or its installed applications is beyond the expectations of refpolicy. For example, embedded systems and commercial applications are not within the scope of refpolicy. However, there are three problems in customizing refpolicy. One is the difficulty in describing configurations, second is the difficulty of verifying refpolicy and third is resource consumption.

2.4.1 Difficulty in Describing Configurations

The major difficulty in describing configurations is complicated configuration elements such as permissions, macros and types. The main reason of complexity is the number of configuration elements. For example, there are more than 700 permissions and more than 1,000 macros and 1,000 types. In addition, difficulties

```
#Assign ftpd_t domain to ftp daemon
 1 type ftpd_t;
 2 type ftpd_exec_t;
 3 init_daemon_domain(ftpd_t,ftpd_exec_t)
 4 init_system_domain(ftpdctl_t,ftpdctl_exec_t)
 5 /usr/sbin/vsftpd --gen_context(system_u:object_r:ftpd_exec_t,s0)
#Permit ftpd_t to read contents
 6 miscfiles_read_public_files(ftpd_t)
 7 /var/ftp(/.*)? gen_context(system_u:object_r:public_content_t,s0)
#Permit ftpd_t to wait connection on tcp port 21
 8 corenet_non_ipsec_sendrecv(ftpd_t)
 9 corenet_tcp_sendrecv_all_if(ftpd_t)
10 corenet_udp_sendrecv_all_if(ftpd_t)
11 corenet_tcp_sendrecv_all_nodes(ftpd_t)
12 corenet_udp_sendrecv_all_nodes(ftpd_t)
13 corenet_tcp_sendrecv_all_ports(ftpd_t)
14 corenet_udp_sendrecv_all_ports(ftpd_t)
15 corenet_tcp_bind_all_nodes(ftpd_t)
16 corenet_tcp_bind_ftp_port(ftpd_t)
17 corenet_tcp_bind_ftp_data_port(ftpd_t)
18 corenet_dontaudit_tcp_bind_all_ports(ftpd_t)
19 portcon tcp 21  gen_context(system_u:object_r:ftp_port_t,s0)
```

Fig. 3 Part of the configuration for the ftp daemon in refpolicy.

in configuring types as discussed in Section 2.2.2 still remain, and nested macro definitions make understanding macros harder.

2.4.2 Difficulty in Verifying Refpolicy

For the purpose of quality assurance for a security policy which is created based on refpolicy, refpolicy should be verified. In this context, *verify* means understand what is configured, then find misconfigurations and modify them. However, it is difficult to verify because of the complexity of the configuration elements as stated before. In addition, the following points make verification more difficult.

- Amount of configurations
  The size of refpolicy makes verification more difficult. For example, refpolicy included in Fedora 9 has configurations for almost all applications shipped with Fedora 9 and is composed of more than 2,000 types and more than 150,000 access rules.
- Conditional expressions
  Many conditional expressions are embedded in refpolicy, and they are sometimes included in macro definitions. Thus, it is difficult to figure out which configurations are enabled.
- Attributes
  Attributes are often used for types and they increase the time necessary to understand what configurations mean, as shown in the next example. The line *allow httpd_t httpdcontent:file read;* is included in refpolicy. *httpd_t* is a domain for the apache daemon, and *httpdcontent* is an attribute. To understand what kind of files httpd_t can access from the line, types that have the httpdcontent attribute have to be found by searching for type declaration statements, which are sometimes embedded in macro definitions.

2.4.3 Resource Consumption

A security policy is saved as files in storage, then it is loaded to RAM at system boot. Therefore, the security policy consumes storage and RAM. Since refpolicy is intended for multiple use cases, many conditional expressions and configurations for many applications are included. As a result, the size of refpolicy becomes large. For example the refpolicy included in Fedora Core 6 consumes 1.4 MB storage and 5.4 MB RAM. In resource constrained systems such as embedded systems, this is a problem because they often have less than 64 MB RAM and storage.

2.4.4 Tools for Refpolicy

Tools are developed by SELinux community to aid customizing refpolicy based security policy.

To help describing configurations, tools called setroubleshoot [12], SLIDE[13] and system-config-selinux [14] are developed. Setroubleshoot analyzes access logs and presents configurations when an application does not work due to SELinux access denial. SLIDE is an Integrated Development Environment (IDE) to configure refpolicy. It has features to aid describing configurations such as input completion. System-config-selinux is a tool to generate templates of configurations for new applications. It can generate templates using a wizard. These tools are helpful, but remain problems. Setroubleshoot is targeted only for minor modifications. In SLIDE and system-config-selinux complexities of SELinux policy language and refpolicy such as many permissions, types, macros are not resolved.

To aid verifying refpolicy, setools [15] has a features to query security policy, such as querying what kind of types a domain can access. It is also useful, but problems discussed in Section 2.4.2 still exist.

3. Approach to Creating Security Policy

To encounter problems in creating security policy based on refpolicy, we propose a different approach. This means that we propose a new security policy configuration system called SEEdit instead of using refpolicy. SEEdit aims to facilitate describing configurations, verifying a created security policy and creating a small security policy. The idea of the proposed system is explained in this section.

3.1 Higher Level Language: SPDL

The difficulty in describing configurations in refpolicy is caused by the large number of permissions, complicated macros and type configurations. Sophisticated macros can partly solve such problems, i.e., creating a small number of macros and removing nested macro definitions. However, type configurations are still necessary in such macros. Instead of macros, we propose a higher level language *SPDL* on top of SELinux policy language. SPDL aims to reduce the number of configuration elements by *integrated permissions* where related SELinux permissions are grouped. In addition, SPDL removes type configurations by identifying resources with their names. An example of configuration by SPDL is shown in **Fig. 4**. The configured access rules are almost the same as Fig. 3, but SPDL is simpler. Permissions related to reading files and directories are merged to integrated permission *r* and permissions to wait for connection on ports are merged to *server*. Additionally, names such as /var/ftp and port 21 are used to identify resources and assigning types to resources is not necessary. To apply SPDL configurations, the *SPDL converter* translates these configurations to SELinux policy language, i.e., SPDL converter generates the necessary type configurations, and expands integrated permissions to related SELinux permissions.

```
1 {
# Assign ftpd_t domain to ftp daemon
2 domain ftpd_t;
3 program /usr/sbin/vsftpd;
# Permit ftpd_t to read /var/ftp
4 allow /var/ftp/** r;
# Permit ftpd_t to wait connection on
tcp port 21
5 allowcom -protocol tcp -port 21 server;
6 }
```

Fig. 4 A configuration example of SPDL for ftp daemon.

The difficulty in verifying refpolicy is caused by two factors. First is the complicated configuration elements such as macros, permissions, attributes and conditional expressions. This complexity is solved by SPDL. Second is that many lines of configurations for access rules for applications not installed in the system and for rules disabled by conditional expressions are included. Our approach to solve the problem of many configuration lines is to describe only necessary configurations by SPDL without refpolicy, i.e., write configurations only for applications installed in the target system. Since neither conditional configurations nor configurations for unused applications are included, the number of configuration lines are expected to be reduced. This also helps reduce resource usage by the security policy.

3.2 SPDL Tools

In order to support writing configurations by SPDL without refpolicy, we propose SPDL tools composed of *template generator* and *allow generator*. SPDL tools aim to reduce the number of configurations written by hand during the process of creating a security policy.

Table 1 shows a typical process of creating a security policy and this process is iterated for each target applications. Configurations to assign a domain to a target application are described as lines 2 and 3 in Fig. 4 (Step 1). In order to figure out what kind of access rules should be described, access logs are obtained by running the target application (Step 2). Access rules are described using the

Table 1 Typical process of creating a security policy.

Step	Necessary works	Available SPDL tools
1	Assign domain	Template generator
2	Run application	-
3	Describe allow rules	Allow generator
4	Check configuration	-

access logs (Step 3). For example, when an access log entry shows *foo_t domain read accessed filename bar* then an access rule that allows *foo_t* to read *bar* is described. Run the application again and see whether it works correctly (Step 4). If the application does not work correctly, go back to Step 2.

Allow generator supports writing configurations allowing access in Step 3 of Table 1. We adopt an approach of audit2allow [16] to automate describing configurations, i.e., generate configurations that permit accesses appearing in access logs.

Template generator outputs configurations in Step 1 of Table 1 by using configurations typical to application categories. For example, most daemon programs require access rights to create temporary files under /var/run and communicate with syslog. To produce more configurations, template generator uses the knowledge of the tool user about the target application, such as what kind of files and network resources the application accesses.

4. Design and Implementation of SEEdit

We designed and implemented SEEdit following the approaches discussed in the previous section. SEEdit provides SPDL and SPDL tools layers on top of SELinux Policy Language as shown in **Fig. 5**. In SPDL layer, SPDL converter generates the security policy written in SELinux policy language from configuration written in SPDL. SELinux policy language is converted to binary format by *checkpolicy*, then loaded to SELinux kernel by *load_policy* command. In SPDL tools layer, SEEdit has *allow generator* and *template generator* to help writing configuration. The design of SPDL and the implementation of SPDL converter and SPDL tools are described in the following subsections.

4.1 Design of SPDL

The main features of SPDL are *integrated permissions* to reduce the number of permissions, and configurations using resource names to remove type configurations. SPDL also has an *include* statement to reduce the number of lines. The detail is explained in this section.

4.1.1 Integrated Permissions

While integrated permissions reduce the number of permissions by grouping permissions, those permissions important for security should be kept. In order to include such important permissions, integrated permissions are designed from the viewpoint of protecting the confidentiality, integrity and availability of a target system. Compromising confidentiality happens when an unexpected information goes out, and compromising integrity happens when an unexpected information comes into the system. Thus, permissions related to input and output to files, network resources and IPCs have to be included in integrated permissions. The other permissions are privileges which can be abused to compromise availability and to facilitate attacks. For example, *setrlimit* permission that controls the resource usage limit of processes can lead to compromised availability. Permission *cap_insmod* can result in installation of malicious kernel modules. Therefore, privileges have to be included in integrated permissions. The details of integrated permissions are shown as follows.

(1) Integrated permissions for files
Integrated permissions for files are taken from previous research by Yamaguchi, et al. [17] because they are designed to control input and output to files and directories. The integrated permissions are, *r* (read), *x* (execute), *s* (list directory), *o* (overwrite), *t* (change attribute), *a* (append), *c* (create), *e* (erase) and *w* (= o+t+a+c+e).

(2) Integrated permissions for network
Two integrated permissions related to input and output are designed for port numbers, NIC, IP address and RAW socket. For example, integrated permissions for port numbers are *server* (wait for a connection from outside) and *client* (begin a connection to outside).

(3) Integrated permissions for IPC
Integrated permissions for Sysv IPCs are *send* and *recv* to control input

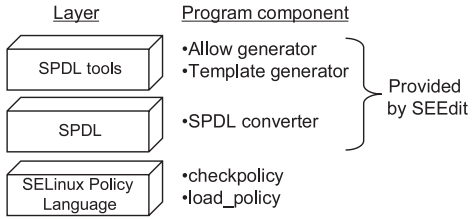

Fig. 5 The architecture of SEEdit.

and output to processes. Integrated permissions for signals are designed to control sending each signal because SELinux can only control sending of signals. For example, integrated permission *k* allows sending sigkill.

(4) Integrated permissions for other privileges
46 integrated permissions for other privileges are designed. Almost all permissions about privileges are included to prevent attackers from compromising availability and facilitating attacks. However, overlapped permissions are merged as an exception. For example, SELinux permission *capability net_admin* and *netlink_route_socket nlmsg_write* overlap each other because they are related to change kernel configuration of network. Thus, they are merged to the integrated permission *net_admin*.

4.1.2 Configurations Using Resource Names

To remove type configurations, SPDL enables configurations using resource names. SPDL statements *allow* and *allownet* are designed as shown in **Fig. 6** to enable name based configurations for files and network resources such as port number, NIC and IP address. To configure IPCs and other privileges, *allowcom* and *allowpriv* are also designed. Assigning types for IPCs and privileges is not required in SELinux, but they are shown for reference in and Fig. 6.

4.1.3 Include Statement

In order to reduce the number of configuration lines, the *include* statement imports configuration from a file. For example, when the file daemon.te includes access rules commonly used for daemon applications, describing *#include daemon.te;* imports those access rules.

4.2 Implementation of SPDL Converter

SPDL converter translates SPDL to SELinux policy language. The translation process is shown with an example of converting SPDL configurations in **Fig. 7** (1) to configurations in Fig. 7 (2).

The ftpd_t domain is allowed to read files and directories under /var/ftp in Fig. 7 (1). SPDL converter generates types from resource names. For example, it generates var_ftp_t type from filename /var/ftp, then outputs configuration to assign var_ftp_t under /var/ftp in the first two lines in Fig. 7 (2), and it generates configuration to allow access to the generated type as line 3–6 in Fig. 7 (2).

When different types are generated for files or directories under /var/ftp, accesses to such types are allowed. For example, when some domains are configured *allow /var/ftp/content1/** r;*, then configuration that assigns var_ftp_content1_t to /var/ftp/content1 is generated. SPDL converter also generates configuration for ftpd_t that allows reading var_ftp_content1_t.

```
(1) Statements
(a) Permits access to files
    allow <filename> <integrated permission>;
(b) Permits access to network resources
    allownet <resourcename> <integrated permission>;
(c) Permits access to domain using IPC
    allowcom <IPCname> <domain>  <integrated permission>;
(d) Permits usage of privilege
    allowpriv <integrated permission>
(2) Example
(a) Permits to read files under /foo/bar directory.
    allow /foo/bar/** r
(b) Permits to wait connection on tcp port 80.
    allownet -protocol tcp -port 80 server;
(c) Permits to read data from process running as foo_t domain
 via unix domain socket.
    allowcom -unix foo_t r;
(d) Permits to use chroot system call.
    allowpriv cap_sys_chroot;
```

Fig. 6 Statements in SPDL to allow access to resources.

```
(1) SPDL to be converted by SPDL converter
1 domain ftpd_t;
2 allow /var/ftp/** r;
(2) Output of SPDL converter
# Declare and assign type
1 type var_ftp_t;
2 /var/ftp(|/.*) system_u:object_r:var_ftp_t
#Allows permissions related to integrated permission r
3 allow ftpd_t var_ftp_t:lnk_file { iotcl lock read  };
4 allow ftpd_t var_ftp_t:file { iotcl lock read };
5 allow ftpd_t var_ftp_t:fifo_file { iotcl lock read };
6 allow ftpd_t var_ftp_t:sock_file { iotcl lock read };
```

Fig. 7 SPDL configurations and output of SPDL converter.

However, configurations using resource names do not work well for files dynamically created by processes. *Dynamically created files* mean files that are removed and created again. In SELinux, when a file is removed and created again, the type of the file is the same as the directory where it belongs. This behavior is sometimes a problem. For example, *allow /tmp/foo r;* is configured in foo_t domain. At first, /tmp/foo is assigned tmp_foo_t type, but when /tmp/foo is removed and created again, then the type is tmp_t. Therefore, the foo_t domain can no longer access /tmp/foo. To handle such cases, SPDL has *allowtmp* to configure assigning types correctly. The syntax of allowtmp is as follows.

```
allowtmp -dir <directory> -name <type> <integrated permission>;
```

This means files created under <directory> are assigned <type>. When <type> is *auto*, type is named automatically. For example, when foo_t domain creates temporary files under /tmp, we have to describe *allowtmp -dir /tmp -name auto r;* in foo_t domain, then type *foo_tmp_t* is generated and assigned to temporary files.

4.3 Implementation of SPDL Tools

4.3.1 Allow Generator

Allow generator outputs configurations that permit accesses recorded in the access log. The process is explained by an example below. First, allow generator reads SELinux access log, then extracts domain, resource name and permission from an access log entry. When a log entry is recorded that says *httpd_t domain process accessed filename /foo/bar whose type is foo_bar_t with permission file read*, httpd_t, /foo/bar/ and *file read* is extracted. The extracted information is not enough to create SPDL based configuration, because the permission is not an integrated permission. In order to obtain an integrated permission, *allow generator* converts SELinux permissions to integrated permissions by permission mapping, which contains mapping of integrated permission to SELinux permissions as illustrated in **Fig. 8**. In the example, recorded SELinux permission is *file read*, then permission mapping is looked up and corresponding integrated permission *file_r* meaning integrated permission *r* for file is found. As a result, allow generator is able to output SPDL based configurations *allow /foo/bar/ r;*, from obtained domain, resource name and integrated permission.

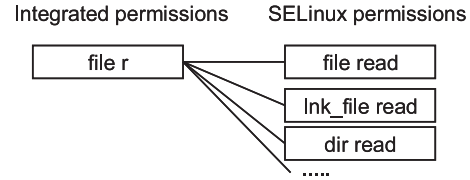

Fig. 8 An example of permission mapping.

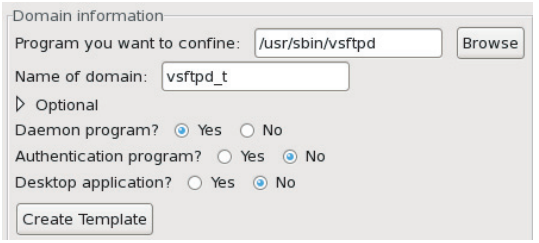

Fig. 9 Template generator GUI to generate typical configurations.

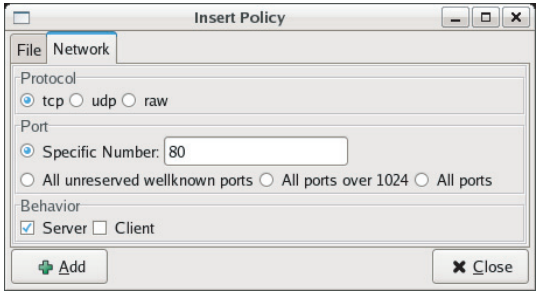

Fig. 10 Template generator GUI to generate using knowledge of users.

4.3.2 Template Generator

Template generator is implemented as a GUI. **Figure 9** is a GUI to generate typical configurations. Users choose the profile of applications, and configurations are generated based on the profile. **Figure 10** is a GUI to generate configurations from the user's knowledge. They can input their knowledge to the template generator without having to manually enter the SPDL.

5. Evaluation

5.1 Experimental Setup

In order to make sure whether SEEdit works, we used two typical systems for experiment. One is an embedded system and the other is a PC system. The embedded system is a low power consumption small server often used in home to serve as http and ftp server. It also has portmap daemon for the purpose of sharing files from PC via NFS. The PC system is intended to represent a PC server which serves typical Internet server applications (HTTP, FTP, Name server, E-mail, File sharing). The detail of these systems is shown in **Table 2**.

Five domains are configured for services running on the embedded system, 16 domains are configured for services on the PC system. Access rules are written for these services to work properly. Memory usage of the security policy on the embedded system was also measured to evaluate whether SELinux is applicable to embedded systems. The memory consumption by SELinux was defined as the difference between memory usage when SELinux enabled and that when SELinux is disabled.

For comparison, we also evaluated refpolicy based security policies. We prepared three kinds of refpolicy. The first is default refpolicy shipped with Cent OS 5 which was used without modification. Second and third are refpolicy tuned for the PC system and the embedded system. The tuning is assumed to be possible for a usual engineer who is not SELinux specialist. In the tuning, configurations for unused applications are removed by hand, but further removal is not done because only SELinux experts can understand and remove unnecessary configuration lines from each application's configurations.

Table 2 Systems used in the evaluation.

	Embedded system	PC system
CPU	SH7751R (SH4) 240 MHz	Core2Duo 2 GHz
RAM	64 MB	1 GB
Storage	64 MB Flash ROM	10 GB HDD
Linux distribution	Not used	Cent OS 5
SELinux	Linux 2.6.22	Linux 2.6.18
Services	httpd, vsftpd, syslogd, klogd, portmap	auditd, avahi_daemon, crond,cupsd, dhclient, gdm, httpd,klogd, mcstransd, named, ntpd, portmap, samba, sendmail

5.2 Result and Consideration

In the experiment, we have successfully created security policies for both the embedded and the PC system. The process of describing configurations, verifying configurations and resource consumption are reviewed and considered. At last, trade-offs in SEEdit are also discussed.

5.2.1 Describing Configurations

The first step to describe configuration is using *template generator*. To evaluate template generator, the assumption of knowledge on the part of the tool user is necessary because generated configurations depend on the user's knowledge. For evaluation, it is assumed that users know how to manage applications, i.e.: they know file path of configuration files for applications, names of log files, names of content files which applications deliver and port numbers for applications. Assuming this, template generator produced 52% of the lines of configuration for the evaluation systems. For example, total 24 lines of configurations were described for http service in the PC system, and 12 lines were generated by template generator.

Next step is to produce configurations from access logs by *allow generator*. Most of the configurations generated by allow generator were able to be used without modification except for the following two cases. First is allow statements generated for dynamically created files. These allow statements have to be replaced with allowtmp statements. For example, foo_t domain dynamically creates and removes /tmp/foo, then log entry *foo_t domain write /tmp/foo* is recorded. Allow generator outputs *allow /tmp/foo w;* from the log entry. However, it should be replaced with *allowtmp -dir /tmp -name auto w;* as shown in Section 4.2. Second is configurations generated from log entries which record access to normal files. Allow generator outputs *allow /var/www/index.html r;* for httpd_t from log entry *httpd_t read /var/www/index.html*. When the user knows http_t domain accesses /var/www directory, it is better to permit access to directory like *allow /var/www/** r;*. For the above two cases, the generated integrated permissions still can be used without modification.

Table 3 Number of permissions in refpolicy and SPDL.

	refpolicy	SPDL
File	130	9
Network	453	14
IPC	45	7
Privilege	80	46
Total	708	76

Table 4 Comparison of number of configuration lines.

	refpolicy (default)	refpolicy (PC)	refpolicy (embedded)	SEEdit (PC)	SEEdit (embedded)
Number of lines	147,218	98,915	92,019	401	174

Table 5 Result of resource consumption in the embedded system.

	refpolicy (default)	refpolicy (tuned)	SEEdit
File size (KB)	1,843	302	71
RAM usage (KB)	5,820	1,168	465

As shown above, SPDL tools generate most parts of the configurations. In addition, to modify a generated SPDL configuration is easier than modifying refpolicy because the number of permissions is reduced as shown in **Table 3**, complicated macros are not necessary, and type configurations are removed.

5.2.2 Verifying Configurations

To verify created security policy, the difficulty depends on the number of configuration lines. The number of configuration lines are shown in **Table 4**. Refpolicy has more than 90,000 lines even when it is tuned for evaluation systems. In addition, complicated permissions, macros and types are included. On the other hand, in the experiment, the total lines of configuration are 174 for the embedded system, 401 for the PC system, and they are described with SPDL. Therefore, it is easier to verify configurations in SPDL than configurations in refpolicy.

Note that verifying configurations written in SPDL is meaningful as long as the output of SPDL converter is correct. Another work is necessary to ensure the result of SPDL converter. One possible way is a test tool. The tool inputs configurations in SPDL and is run for each domain defined in the configurations, then the tool tries all access patterns to see if only accesses configured in the policy are permitted.

5.2.3 Resource Consumption

The file size of the security policy in the embedded system is 71 KB and RAM usage is 465 KB. In the system used in the experiment, storage is 64 MB, RAM is 64 MB. The consumption of storage and RAM is less than 1%. Thus, the created security policy is usable for the resource constrained embedded devices.

In addition, the resource consumption is also smaller compared with refpolicy based security policy as shown in **Table 5**. The file size of SEEdit based security policy is about 24% and the RAM usage is 40% of tuned refpolicy.

5.2.4 Trade-offs

There are two usability-security trade-offs in SEEdit. The first trade-off is integrated permissions used in SPDL because integrated permissions reduce granularity. For example, integrated permission for file r means read permissions for file, symlink and socket file. Therefore, allowing read access to symlink but not to file and directory can not be configured by r permission. To solve this problem, the security policy generated by SPDL converter has to be edited. Another solution is to create a new statement in SPDL that enables configuring SELinux permissions directly. The second trade-off is the audit2allow approach in allow generator. If there is a bug or malicious code in a program, and the program accesses files unnecessary for the program to work correctly, allow generator outputs configurations to permit access to such files. For example, if code that accesses confidential data is embedded in a CGI program by an evil programmer, then a configuration that permits access to the confidential data is outputted by allow generator after running the CGI. To prevent such a dangerous configuration to be included in the security policy, generated configurations should be checked by the SEEdit user. To help the check process, a tool that evaluates generated configurations would be useful.

6. Related Work

There are some researches for improving the usability of mandatory access control. Shan proposed a kind of new mandatory access control model, named CUMAC [18]. CUMAC can improve both compatibility and usability. Li, et al. also proposed Usable Mandatory Integrity Protection (UMIP) model [19]. UMIP

model adds usable mandatory access control to operating systems. In addition, AppArmor is implemented using the Linux Security Modules kernel interface [20]. AppArmor is a mandatory access control system based on file paths. It is easier to use than SELinux. These researches proposed new mandatory access control systems, because existing mandatory access control systems for operating systems are difficult to use. On the other hand, this paper proposes a higher level language for an existing mandatory access control system.

There are some works about higher level language for SELinux. Sniffe, et al. introduced a tool called polgen [21] with a higher level language to describe template configurations. The purpose is different from SEEdit because SPDL in SEEdit is intended to describe whole configurations. SENG [22] is a higher level language intended to replace m4 macros. However, it is not targeted to reduce to reduce permissions and remove type configurations. Sellers, et al. [23] also designed a higher level language and IDE [24] on top of the language. The language is designed from the viewpoint of information flow control, but is not intended to simplify configurations.

There are also works related to security policy verification. SLAT [25] is a security policy analyzer which points out violations of the information flow goal described by analyzers in advance. Gokyo [26] analyzes a security policy based on constraints called Access Control Spaces, then suggests configurations violating the constraints. The purpose of these tools is to assist formal verification. On the other hand, SPDL is intended to facilitate casual verification which means understanding what is configured.

7. Summary

Security policy for SELinux is usually created by customizing a sample policy called refpolicy. However, creating security policy based on refpolicy has problems in describing and verifying configurations, and in resource consumption.

We proposed a security policy configuration system SEEdit which makes creating security policy easier with a higher level language called SPDL and SPDL tools. SPDL reduces the number of permissions by integrated permissions, and removes type configurations by name based configurations. SPDL tools help in writing configuration by generating configurations based on access logs and the knowledge of tool users about applications. Experimental results on an embedded system and a PC system have shown that SEEdit resolves the problems creating security policy and practical security policy can be created with SEEdit.

8. Future Work

There are remaining issues in ensuring the results of SPDL converter (Section 5.2.2) and trade-offs in SEEdit (Section 5.2.4). Another issue is co-existing with refpolicy. Currently SEEdit can not be used with refpolicy because type configurations generated by SPDL converter conflict with existing type configurations in refpolicy. SPDL converter has to be improved to resolve such conflicts.

References

1) CVE-2008-0600: Common Vulnerabilities and Exposures (2008). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0600
2) CVE-2007-5964: Common Vulnerabilities and Exposures (2007). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5964
3) Security-Enhanced Linux: http://www.nsa.gov/research/selinux/
4) Loscocco, P. and Smalley, S.: Integrating Flexible Support for Security Policies into the Linux Operating System, *Proc. FREENIX Track of the 2001 USENIX Annual Technical Conference*, pp.29–42 (2001).
5) Boebert, W.E. and Kain, R.Y.: A Practical Alternative to Hierarchical Integrity Policies, *Proc. the Eighth National Computer Security Conference*, pp.225–237 (1985).
6) Coker, F. and Coker, R.: Taking advantage of SELinux in Red Hat Enterprise Linux, *Redhat magazine*, Issue 6 (2005). http://www.redhat.com/magazine/006apr05/features/selinux/
7) Linuxdevices.com: MontaVista readies new Linux mobile phone OS (2007). http://www.linuxdevices.com/news/NS4364061392.html
8) SELinux Reference Policy: http://oss.tresys.com/projects/refpolicy/
9) PeBenito, C., Mayer, F. and MacMillan, K.: Reference Policy for Security Enhanced Linux, *Proc. 2006 Security Enhanced Linux Symposium* (2006). http://selinux-symposium.org/2006/papers/05-refpol.pdf
10) Smalley, S.: Configuring the SELinux policy, NAI Labs Report #02-007. http://www.nsa.gov/research/selinux/docs.shtml
11) GNU m4: http://www.gnu.org/software/m4/m4.html
12) Denis, J.: Setroubleshoot: A User Friendly Tool to Diagnose AVC Denials, *Proc. 2007 Security Enhanced Linux Symposium* (2007). http://selinux-symposium.org/2007/papers/09-setroubleshoot.pdf

13) SLIDE: http://oss.tresys.com/projects/slide
14) Walsh, D.: A step-by-step guide to building a new SELinux policy module, *Redhat magazine* (2007). http://magazine.redhat.com/2007/08/21/
15) SETools. http://oss.tresys.com/projects/setools
16) Linux man pages for audit2allow(1). http://linuxcommand.org/man_pages/audit2allow1.html
17) Yamaguchi, T., Nakamura, Y. and Tabata, T.: Integrated Access Permission: Secure and Simple Policy Description by Integration of File Access Vector Permission, *Proc. The 2nd International Conference on Information Security and Assurance* (*ISA2008*), pp.40–45 (2008).
18) Shan, Z.: Compatible and Usable Mandatory Access Control for Good-enough OS Security, *Proc. International Symposium on Electronic Commerce and Security* (*ISECS*), Vol.1, pp.246–250 (2009).
19) Li, N., Mao, Z. and Chen, H.: Usable Mandatory Integrity Protection for Operating Systems, *Proc. IEEE Symposium on Security and Privacy*, pp.164–178 (2007).
20) Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P. and Gligor, V.: SubDomain: Parsimonious Server Security, *Proc. 14th Systems Administration Conference* (*LISA 2000*), pp.355–367 (2000).
21) Sniffen, B., Ramsdell, J. and Harris, D.: Guided Policy Generation for Application Authors, *Proc. 2006 Security Enhanced Linux Symposium* (2006). http://selinux-symposium.org/2006/papers/14-guided-polgen.pdf
22) Kuliniewicz, P.: SENG: An Enhanced Policy Language for SELinux, *Proc. 2006 Security Enhanced Linux Symposium* (2006). http://selinux-symposium.org/2006/papers/09-SENG.pdf
23) Sellers, C., Athey, J., Shimko, S., Mayer, F. and MacMillan, K.: Experiences Implementing a Higher-Level Policy Language for SELinux, *Proc. 2006 Security Enhanced Linux Symposium* (2006). http://selinux-symposium.org/2006/papers/08-higher-level-experience.pdf
24) CDS Framework IDE. http://oss.tresys.com/projects/cdsframework
25) Guttman, J., Herzog, A., Ramsdell, J. and Skorupka, C.: Verifying information goals in security-enhanced linux, *Journal of Computer Security*, Vol.13, No.1, pp.115–134 (2005).
26) Jaeger, T., Edwards, A. and Zhang, X.: Managing access control policies using access control spaces, *Proc. 7th ACM Symposium on Access Control Models and Technologies* (*SACMAT 02*), pp.3–12 (2002).

(Received November 30, 2009)
(Accepted June 3, 2010)
(Original version of this article can be found in the Journal of Information Processing Vol.18, pp.201–212.)

Yuichi Nakamura received his B.S. and M.S. degrees in physics from the University of Tokyo in 1999 and 2001, and M.S. degree in computer science from the George Washington University in 2006. He is working for Hitachi Software Engineering Co., Ltd. since 2001. His research interests are computer security, open source software and home network. He is a member of IPSJ.

Yoshiki Sameshima received his B.S. degree from Kyoto University in 1984, the M.S. degree in mathematics from Osaka University in 1986, and the Ph.D. degree in computer science and technology from Osaka University in 2008. He is working for Hitachi Software since 1986, and stayed at Computer Science Department of University College London from 1992 to 1994. Since 1993, he has been working in research of information security. He is a member of IEICE, IPSJ, JSSM, the USENIX Association, and the WIDE Project.

Toshihiro Yamauchi received his B.E., M.E. and the Ph.D. degrees in computer science from Kyushu University, Japan in 1998, 2000 and 2002, respectively. In 2001 he was a research fellow of the Japan Society for the Promotion of Science. In 2002 he became a research associate in Faculty of Information Science and Electrical Engineering at Kyushu University. He has been an associate professor of Graduate School of Natural Science and Technology at Okayama University since 2005. His research interests include operating system and computer security. He is a member of IPSJ, IEICE, USENIX, and ACM.