SEEdit: SELinux Security Policy Configuration System with Higher Level Language

Yuichi Nakamura
Hitachi Software Engineering Co., Ltd.
ynakam@hitachisoft.jp

Yoshiki Sameshima
Hitachi Software Engineering Co., Ltd.
same@hitachisoft.jp

Toshihiro Tabata
Okayama University
tabata@cs.okayama-u.ac.jp

Abstract

Security policy for SELinux is usually created by customizing a sample policy called refpolicy. However, describing and verifying security policy configurations is difficult because in refpolicy, there are more than 100,000 lines of configurations, thousands of elements such as permissions, macros and labels. The memory footprint of refpolicy which is around 5MB, is also a problem for resource constrained devices.

We propose a security policy configuration system 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 label 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 semiautomated, 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 500KB.

Tags: security, security policy, configuration, SELinux

1 Introduction

Attackers can do everything in traditional Linux when they obtain the almighty root privilege by exploiting security holes in services running as root, or by exploiting vulnerabilities leading to privilege escalation[3][4]. To restrict such behavior of root, Security-Enhanced Linux (SELinux)[1][2] has mandatory access control feature; all processes including root processes can access resources only when a security policy permits the access. The mandatory access control model is called TE (Type-Enforcement)[5]. In TE, processes are assigned *domain* labels, and resources such as files and ports are assigned *type* labels, and what kind of domain can access what kind of type is described in a security policy. If the security policy is properly configured, all processes including root, attackers processes and viruses have only limited access rights. As a result, the damage by attackers and viruses is confined. 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. The security policy is usually created by customizing a sample policy called refpolicy (Reference Policy)[8][9]. 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, such as embedded systems and systems where commercial applications are deployed.

There are three problems in the customization. First, it is difficult to describe configurations because there are more than 700 permissions and 1000 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, many configurations, more than 100,000 lines, are included. When engineers verify refpolicy before reuse, they have to review such a lot of configurations. Third is a problem of resource consumption. When SELinux is 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 refpolicy.

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], which is mainly composed of the following four syntax elements.

(1) Assigning types
In SELinux, *type* labels must be assigned to resources to identify them. For example, the following statement is written to assign types to files.

<file name> system_u:object_r:<type>

Similar statements are used to assign types to network resources such as port numbers and NICs.

(2) Label declaration
Domains and types must be declared by *type* statements as shown below.

type <type or domain>, <attribute>;

<attribute> is used to inherit configurations which are described for <attribute>. For example, in the following statements, admin_t can read both httpcontent_t and ftpcontent_t.

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

(3) Allowing access
The allow statement permits a domain to access a type as in the following syntax.

allow <domain> <type> <permission>;

<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.

(4) Conditional policy expression
To support multiple use cases in one security policy, SELinux policy language has conditional policy expressions as follows.

if(<parameter>){<statement>}

When <parameter> is true, then <statement> is enabled. For example, when CGI is necessary, the parameter *httpd_enable_cgi* is set true, and then accesses related to using CGI are permitted. Change of such parameters are applied without reloading security policy, because <statement> is embedded in the security policy.

2.2 Overview of refpolicy

To grant enough permissions for applications to work correctly, a lot of access rules should be described. In fact, the total number of access rules in a system often becomes more than 10,000, and sometimes exceeds 100,000. Therefore, it is not realistic to create security policy by writing configurations in SELinux policy language from nothing. To facilitate creating security policy, a sample policy called *refpolicy* is developed and

maintained by the SELinux community. Refpolicy is composed of macros and configurations for typical applications.

(1) Macros
M4[11] macros are defined to describe frequently used phrases in short words. Below is an example.

```
allow httpd_t contents_t r_file_perms;
define('r_file_perms','file { read
 getattr lock ioctl }')
```

r_file_perms is a 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 1 is part of the configuration for the http daemon. There are many macros, such as *init_daemon_domain*, *apache_content_template* 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 CGI, PHP and DB connection.

2.3 Problems in creating security policy using refpolicy

Customizing refpolicy is necessary when the use case of the system or its installed applications are 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.

2.3.1 Difficulty in describing configurations

The major difficulty in describing configurations is complicated configuration elements such as permissions, macros and types. The main reason of difficulty 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, nested macro definitions make understanding macros harder.

There are two more difficulties in types. First, engineers have to get used to types because in traditional Linux, they have been identifying files by file names not types. 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

```
# Assign httpd_t domain to http daemon
1  type httpd_t;
2  type httpd_exec_t;
3  role system_r types httpd_t;
4  init_daemon_domain(httpd_t,httpd_exec_t)
5 /usr/sbin/httpd -- gen_context(system_u
:object_r:httpd_exec_t,s0)
# Permit httpd_t to read /var/www
6  apache_content_template(sys)
7 /var/www(/.*)? gen_context(system_u
:object_r:httpd_sys_content_t,s0)
8  allow httpd_t httpd_sys_content_t:dir
 list_dir_perms;
9  read_files_pattern(httpd_t,httpd_sys_
content_t,httpd_sys_content_t)
10  read_lnk_files_pattern(httpd_t,httpd_
sys_content_t,httpd_sys_content_t)
# Permit httpd_t to wait connection on
  tcp port 80
11 corenet_all_recvfrom_unlabeled(httpd_t)
12 corenet_all_recvfrom_netlabel(httpd_t)
13 corenet_tcp_sendrecv_all_if(httpd_t)
14 corenet_udp_sendrecv_all_if(httpd_t)
15 corenet_tcp_sendrecv_all_nodes(httpd_t)
16 corenet_udp_sendrecv_all_nodes(httpd_t)
17 corenet_tcp_sendrecv_all_ports(httpd_t)
18 corenet_udp_sendrecv_all_ports(httpd_t)
19 corenet_tcp_bind_all_nodes(httpd_t)
20 corenet_tcp_bind_http_port(httpd_t)
21 gen_context(system_u:object_r:http_port
_t,s0)
```

Figure 1: Part of the configuration for the http daemon in refpolicy

type, the bar_t domain can read all files under the /foo directory. Next, if the foo2_t type is newly created, and assigned to the file /foo/foo2. 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.

2.3.2 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.

2.3.3 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 1.4MB storage and 5.4MB RAM. In resource constrained systems such as embedded systems, this is a problem because they often have less than 64MB RAM and storage.

3 Approach to creating security policy

We propose a security policy configuration system SEEdit, which facilitates 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 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 Figure 2. The configured access rules are almost the same as Figure 1, 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/www and port 80 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 httpd_t domain to http daemon
2 domain httpd_t;
3 program /usr/sbin/httpd;
# Permit httpd_t to read /var/www
4 allow /var/www/** s,r;
# Permit httpd_t to wait connection on
  tcp port 80
5 allowcom -protocol tcp -port 80 server;
6 }
```

Figure 2: A configuration example of SPDL for http 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 contributes to reducing resource usage by the security policy.

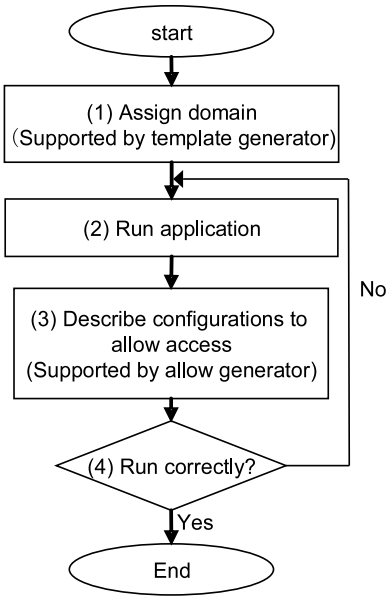

Figure 3: Typical process of creating a security policy

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.

Figure 3 shows a typical process of creating a security policy and this process is iterated for each target applications. (1) Configurations to assign a domain to a target application are described as in Figure 2 lines 2 and 3. (2) In order to figure out what kind of access rules should be described, access logs are obtained by running the target application. (3) Access rules are described using the access logs. 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. (4) Run the application again and see whether it works correctly. If the application does not work correctly, run the application again and add configuration elements until the application works correctly.

Allow generator supports writing configurations allowing access in Figure 3 step (3). We adopt an approach of audit2allow[12] to automate describing configurations, i.e. generate configurations that allow accesses appearing in access logs.

Template generator outputs configurations in figure 3 step (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.

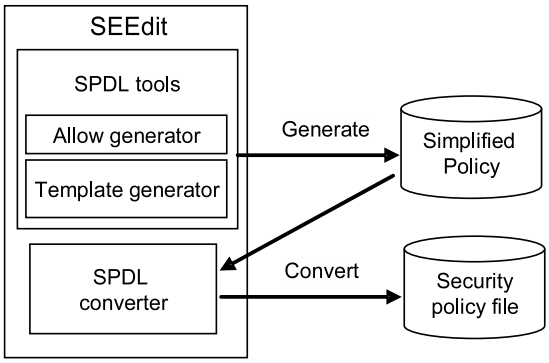

Figure 4: The architecture of SEEdit

4 Design and implementation of SEEdit

We designed and implemented SEEdit following the approaches discussed in the previous section. SEEdit is composed of SPDL tools and SPDL converter as shown in Figure 4. The security policy written in SPDL, called *simplified policy*, is created by a text editor or SPDL tools composed of allow generator and template generator. SPDL converter generates the security policy written in SELinux policy language from simplified policy. 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, permissions important for security should be kept. In order to include such important permissions, integrated permissions are designed from the viewpoint of protectiong 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. *cap_insmod* permission can result in installation of malicious kernel modules. Therefore, privileges have to be included in integrated permissions. The detail 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[13] 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 and output to processes. Integrated permissions for signals are designed to control sending each signal because SELinux can only control sending 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 Table 1 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 Table 1.

```
domain httpd_t;
allow /var/www/** r;
```

Figure 5: Simplified Policy to be converted by SPDL converter

```
# Declare and assign type
1 type var_www_t;
2 /var/www(|/.*)
   system_u:object_r:var_www_t
#Allows permissions related to integrated
 permission r
3 allow httpd_t var_www_t:lnk_file { iotcl
  lock read  };
4 allow httpd_t var_www_t:file { iotcl
 lock read };
5 allow httpd_t var_www_t:fifo_file {
  iotcl lock read };
6 allow httpd_t var_www_t:sock_file {
  iotcl lock read };
```

Figure 6: Output of SPDL converter

4.1.3 Include statement

In order to reduce the number of configuration lines, the *include* statement imports configuration from a file.

#include *filename*;

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 Simplified Policy in Figure 5 to configurations in Figure 6.

The httpd_t domain is allowed to read files and directories under /var/www in Figure 5. SPDL converter generates types from resource names. For example, it generates var_www_t type from filename /var/www, then outputs configuration to assign var_www_t under /var/www in the first two lines in Figure 6. Next, it generates configuration to allow access to the generated type as line 3-6 in figure 6.

When different types are generated for files or directories under /var/www, accesses to such types are allowed. For example, when some domains are configured

Statement	Meaning	Example
allow *filename integrated permission*;	Allows access to *filename* using *integrated permission*.	*allow /foo/bar/** r;* permits to read files under /foo/bar directory.
allownet *resourcename integrated permission*;	Allows access to *resourcename* using *integrated permission*.	*allownet -protocol tcp -port 80 server;* permits to wait connection on tcp port 80.
allowcom *IPCname domain integratedpermission*;	Allows access to *domain* using IPC *IPCname* and communicate using *integrated permission*.	*allowcom -unix foo_t r;* permits to read data from process running as foo_t domain via unix domain socket.
allowpriv *integrated permission*;	Allows usage of privilege *integrated permission*	*allowpriv cap_sys_chroot;* permits to use chroot system call.

Table 1: Statements in SPDL to allow access to resources

*allow /var/www/cgi/** r;*, then configuration that assigns var_www_cgi_t to /var/www/cgi is generated. SPDL converter also generates configuration for httpd_t that allows reading var_www_cgi_t.

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.

```
#Integrated permission
<macro value="allow_file_r" /> @@@
#Corresponding SELinux permissions
  <secclass value="file" />
  <secclass value="lnk_file" />
  <secclass value="dir" />
  <permission value="read" />
...<snip>..
```

Figure 7: An example of permission mapping file

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 Figure 7. In the example, recorded SELinux permission is *file read*, then permission mappping is loooked up and corresponding integrated permission *file_read_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.

4.3.2 Template generator

Template generator is implemented as a GUI. Figure 8 is a GUI to generate typical configurations. Users choose the profile of applications, and configurations are generated based on the profile. Figure 9 is a GUI to generate configurations from the user's knowledge. They can input their knowledge to the template generator without typing SPDL manually.

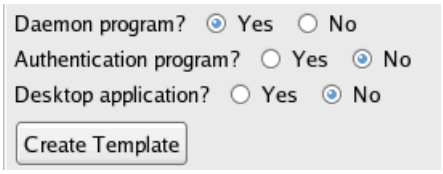

Figure 8: Template generator GUI to generate typical configurations

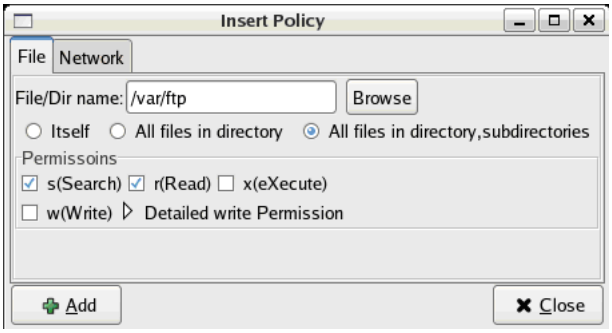

Figure 9: Template generator GUI to generate using knowledge of users

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 configured for a small server, the other is a PC system configured for PC server as shown below.

(1) Embedded system

- CPU: SH7751R(SH4) 240MHz
- RAM: 64MB
- Storage: Flash ROM 64MB
- Linux distribution: not used
- SELinux: Linux 2.6.22
- Running services: httpd, vsftpd, syslogd, klogd, portmap

(2) PC system
Virtual machine (VMware 5.5) is used.

- Linux distribution: Cent OS 5 used for PC servers
- Running services: auditd, avahi_daemon, crond, cupsd, dhclient, gdm, httpd, klogd, mcstransd, named, ntpd, portmap, samba, sendmail, sshd, syslogd

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.

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.

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

Table 2: Number of permissions in refpolicy and SPDL

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 are reduced as shown in Figure 2, 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 in refpolicy is more than 100,000 with complicated permissions, macros and types, thus verification of refpolicy based security policy is difficult. 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 the 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. Next 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 71KB and RAM usage is 465 KB. In the system used in the experiment, storage is 64MB, RAM is 64MB. The consumption of storage and RAM is less than 1%. Thus, the created security policy is usable for the resource constrained embedded devices.

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. This can be a problem in the embedded systems used in evaluation. In the embedded system, busybox[14] was used for system commands. In a system where busybox is installed, commands are executed via symbolic links to /bin/busybox(busybox executable). When /bin/ls is symbolic link to /bin/busybox and /bin/ls is executed, ls functions in /bin/busybox are called. If a domain foo_t needs access to busybox commands and is configured *allow /bin/** r;*, foo_t domain can access symbolic links under /bin, and foo_t can use busybox commands. However, if a confidential command file /bin/secret exists, foo_t can also access /bin/secret. If access to symbolic links were configured separately, foo_t would not be able to access /bin/secret. 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

Linux distribution Fedora includes security policy configuration tools called setroubleshoot [15], SLIDE [16] and system-config-selinux [17]. 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. The above tools are intended to aid configurations using refpolicy. The purpose is different from SEEdit because SEEdit does not use refpolicy.

polgen[18] is a security policy generator with a higher level language. Users of polgen first describe template configurations for the target applications using the language, then run the application. Next, polgen generates recommended security policy from access logs. The purpose of the higher level language of polgen is to de-

scribe template configurations, and users have to handle types and SELinux permissions after writing a template. The purpose is different from SEEdit because SPDL in SEEdit is intended to describe whole configurations.

SENG [19] is a higher level language for SELinux security policy. It is intended to replace m4 macros, not to reduce the number of configurations and remove type configurations.

Sellers et al.[20] also implemented a higher level language and IDE called CDS Framework[21]. It is also used in the FMAC[22] project in OpenSolaris. It enables configuration from the viewpoint of information flow control, but is not intended to simplify configurations.

There is also work related to the verification of security policy. Apol included in setools[23] has a features to query security policy, such as querying what kind of types a domain can access. SLAT[24][25] is a system to analyze the security policy based on information flow goals. Analyzers describe an information goal, then SLAT finds violations of the information flow goal. Gokyo[26] analyzes the security policy based on Access Control Spaces, then suggests configurations which violate constraints. These tools are for SELinux policy language, but they can be applied to configurations which are converted from SPDL.

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 have 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.

9 Availability

SEEdit is available from sourceforge[27]. It is licensed under the GPL.

References

[1] Security-Enhanced Linux, http://www.nsa.gov/research/selinux/

[2] 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)

[3] CVE-2008-0600: Common Vulnerabilities and Exposures (2008), http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0600

[4] CVE-2007-5964: Common Vulnerabilities and Exposures (2007), http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5964

[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., Coker,R.: Taking advantage of SELinux in Red Hat Enterprise Linux:Redhat magazine Issue 6 April 2005 (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] Linux man pages for audit2allow(1), http://linuxcommand.org/man_pages/audit2allow1.html

[13] 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)

[14] Wells, N.: BusyBox: A Swiss Army Knife for Linux, Linux Journal, vol.2000, n.78es (2000)

[15] 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

[16] SLIDE: http://oss.tresys.com/projects/slide

[17] Walsh, D.: A step-by-step guide to building a new SELinux policy module: Redhat magazine(2007), http://magazine.redhat.com/2007/08/21/

[18] 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

[19] 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

[20] 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

[21] CDS Framework IDE, http://oss.tresys.com/projects/cdsframework

[22] OpenSolaris Project: Flexible Mandatory Access Control, http://www.opensolaris.org/os/project/fmac/

[23] SETools, http://oss.tresys.com/projects/setools

[24] Guttman, J., Herzog, A., Ramsdell, J. and Skorupka, C.: Verifying information goals in security-enhanced linux: Journal of Computer Security., 13(1), pp 115-134 (2005)

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

[26] Jaeger, T., Edwards, A. and Zhang, X.: Managing access control policies using access control spaces: Proc the seventh ACM symposium on Access control models and technologies (SACMAT 02), pp. 3-12 (2002)

[27] SELinux Policy Editor Website, http://seedit.sourceforge.net/