Algorithms for Automatic Analysis of SELinux Security Policy

Gaoshou Zhai, Tong Wu, Jing Bai, Tao Guo and Tianyou Li

School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China
gszhai@bjtu.edu.cn

Abstract

Configuration of security policies is an important but complicated work for running of secure operating systems. On the one hand, completely correct and consistent configuration is the necessary prerequisite for secure and credible system operation. On the other hand, errors and bugs are incidental anywhere within configuration at all time. Therefore, algorithms for automatic analysis of SELinux security policy are studied in this paper. Based on an improved analysis model similar to SELAC model, both algorithms for validity analysis and integrity analysis are designed. So that any access relations among subjects and objects with specified security contexts can be identified correctly by using the former algorithm. And all rules that could potentially influence integrity of subjects and objects can be detected based on the latter algorithm. Furthermore, a corresponding prototype is implemented in C Language and a security policy configuration as to an application system called Student-Teacher system is designed based on the architecture of reference policy in order to test the prototype. Results are satisfactory and it shows that related algorithms are potential to be used to build an appropriate tool to assist people to perform configuration work and to complete correct and reliable configuration.

Keywords: *Validity analysis, Integrity analysis, Security policy, SELinux, Access control, Secure operating systems*

1. Introduction

Operating systems are the key foundation of security for information systems while mandatory access mechanism (MAC) is a necessary part of secure operating systems [1-3].

SELinux is a typical implementation of a flexible and fine-grained MAC architecture called Flask in the Linux kernel and it can be used cooperated with discretionary access control to implement effective control whenever a subject request to access an object [4].. SELinux can enforce an administratively-defined security policy over all processes and objects in the system, basing decisions on labels containing a variety of security-relevant information. To demonstrate the architecture, SELinux provides an example security server that implements a combination of Type Enforcement (TE), Role-Based Access Control (RBAC), and optionally Multi-Level Security (MLS). These security models provide significant flexibility through a set of policy configuration files.

It is obvious that reliable access control depends on the correct policy configuration. But it is hard for people to perform security policy configuration correctly and inerrably and such task is both time consuming and tedious. Therefore, it is rather significant to study automatic analysis method about security policy configuration so as to build appropriate computer-aided configuration tools [5-7]. Accordingly, algorithms for analyzing SELinux policies are

discussed in the following sections. Considering that MLS rules are not optional for default, TE and RBAC rules are focused in this paper. In addition, constraints rules are also processed. Furthermore, it is taken aim at validity and integrity for policy analysis and it is based on an improved SELinux Access Control (SELAC) model. Comparing with SELAC model [5], scope of possible values for role is reduced and thus a great many invalid security contexts are eliminated in our model.

2. Algorithm Design

2.1. Overall Solution

Validity analysis is to make sure that the policy configuration has carried out expected access regulations, so that results for at least three cases ought to be got correctly. Firstly, if a subject is specified by the way of security context (e.g. <*user_name*, *role_name*, *type_name*>), all objects with corresponding security context and permissions (in the form of <*class_name*, *access_mode*>) that it can access can be worked out. By contrary, if an object is specified by the way of security context, all subjects with corresponding security context and permissions that it can be accessed can also be worked out. In addition, if a subject and an object are specified in the way of security context respectively, corresponding permissions can be figured out.

Integrity analysis is to verify that subjects inside Trusted Computing Base (TCB) are prohibited to read wrong information from non-trusted objects while sensible information inside TCB objects are protected from wrongly modified. If results show that no non-TCB subject or object can infect any TCB ones, it can be proved that the integrity of TCB is protected by the policy configuration. In fact, it is necessary that information flow from a non-TCB one into a TCB one in some cases. But such information flow ought to be audited, which can be ensured by a different authorization way of **auditallow** statement (opposite to **allow** authorization way). So that any information flow ought to be worked out if it could influence the integrity of TCB without audit.

In order to accomplish above objective, configuration files need to be read and some basic elements for analysis model such as attributes, types and etc ought to be extracted in the first place. Moreover, the sets of classes, two-tuples like <*class_name*, *permission_name*>, attributes, types, roles, and users are calculated and generated in turn, which can be marked as *SetC*, *SetCP*, *SetA*, *SetT*, *SetR* and *SetU* respectively. At the same time, relationships among them are also extracted. In particular, access mapping from types of subjects to types of objects with corresponding permissions is established, which can be marked as *MappingSubtoObjCP*. Security context space for subjects and that for objects are then constructed, which can be marked as SCS_{sub} and SCS_{obj} respectively. And valid access space for subjects and that for objects, which can be marked as VAS_{sub} and VAS_{obj} respectively, are generated by the way of extending from types of subjects and objects to security contexts of them within *MappingSubtoObjP*. A so-called TCB access space is also constructed in order to detect those non-TCB subjects/objects and corresponding configuration rules, which can infect the integrity of TCB.

2.2. Extracting of Basic Component

2.2.1. Classes and Permissions: Algorithm of generating *SetC* and *SetCP* can be described as follows:

step 0. Set $SetC = \Phi$ and $SetCP = \Phi$.

step 1. Read policy configuration files and check if there is any class/permission declaration statement that has not been processed. If it is false, go to step 5.

step 2. If there is a class declaration statement as **class** *class_name***;** and *class_name* doesn't belong to *SetC*, append *class_name* into *SetC* and go to step 1.

step 3. If there is a class permission declaration statement as **class** *class_name* **{***permission_name*$_1$ … *permission_name*$_n$**}** Then (i) if *class_name* doesn't belong to *SetC*, append *class_name* into *SetC*; (ii) For all *permission_name*$_i$ ($1 \le i \le n$): append <*class_name*, *permission_name*$_i$> into *SetCP*.

step 4. Go to step 1.

step 5. END.

2.2.2. Attributes: Algorithm of generating *SetA* can be described as follows:

step 0. Set *SetA* = Φ.

step 1. Read policy configuration files and check if there is any attribute declaration statement that has not been processed. If it is false, go to step 4.

step 2. If there is an attribute declaration statement as **attribute** *attribute_name***;** and *attribute_name* doesn't belong to *SetA*, append *attribute_name* into *SetA*.

step 3. Go to step 1.

step 4. END.

2.2.3. Types: Algorithm of generating *SetT* and corresponding relationships can be described as follows:

step 0. Initialization: (i) Set *SetT* = Φ; (ii) For any *attribute_name* ∈ *SetA*, initialize the associated set of mapping from *attribute_name* to *SetT*, i.e. *MappingTAtoT[attribute_name]*= Φ.

step 1. Read policy configuration files and check if there is any type declaration statement that has not been processed. If it is false, go to step 5.

step 2. If there is a type declaration statement as **type** *type_name***;** and *type_name* doesn't belong to *SetT*, append *type_name* into *SetT* and initialize the associated set of mapping from *type_name* to *SetT* (i.e. *MappingTAtoT[type_name]*= {*type_name*}) and go to step 1.

step 3. If there is a type declaration statement as **typeattribute** *type_name attribute_name*$_1$, …, *attribute_name*$_n$**;** Then (i) if *type_name* doesn't belong to *SetT*, append *type_name* into *SetT*; (ii) For all *attribute_name*$_i$ ($1 \le i \le n$): if it belongs to *SetA*, append it into *MappingTtoA[type_name]* and append *type_name* into *MappingTAtoT[attribute_name*$_i$*]* respectively.

step 4. Go to step 1.

step 5. END.

2.2.4. Roles and users: Algorithm of generating *SetR*, *SetU* and corresponding relationships can be described as follows:

step 0. Set *SetR* = Φ and *SetU* = Φ.

step 1. Read policy configuration files and check if there is any role declaration statement that has not been processed. If it is false, go to step 4.

step 2. If there is a role declaration statement as **role** *role_name* **types** *typeorattribute_name*$_1$**;** or **role** *role_name* **types {***typeorattribute_name*$_1$, …, *typeorattribute_name*$_n$**};** Then (i) if *role_name* doesn't belong to *SetR*, append *role_name* into *SetR* and initialize its dominated roles set (i.e. *SetDR[role_name]* = Φ) and the associated set of mapping from *role_name* to *SetT* (i.e. *MappingRtoT[role_name]*= Φ); (ii) For all *typeorattribute_name*$_i$ ($1 \le i \le n$): if it belongs to *SetT*, append *MappingTAtoT[typeorattribute_name*$_i$*]* into *MappingRtoT[role_name]*.

step 3. Go to step 1.

step 4. Read policy configuration files and check if there is any role dominance statement that has not been processed. If it is false, go to step 7.

step 5. If there is a role dominance statement as **dominance {role** *role_name* **{role** *role_name*$_1$**;** … **; role** *role_name*$_n$**; } }** Then, for all *role_name*$_i$ ($1 \le i \le n$): if it belongs to *SetR*, append it into *SetDR[role_name]* and append *MappingRtoT[role_name*$_i$*]* into *MappingRtoT[role_name]* respectively.

step 6. Go to step 4.

step 7. Read policy configuration files and check if there is any user declaration statement that has not been processed. If it is false, go to step 10.

step 8. If there is a user declaration statement as **user** *user_name* **roles {***role_name*$_1$ … *role_name*$_n$**};** Then (i) if *user_name* doesn't belong to *SetU*, append *user_name* into *SetU* and initialize the associated set of mapping from *user_name* to *SetR* (i.e. *MappingUtoR[user_name]*= Φ); (ii) For all *role_name*$_i$ ($1 \le i \le n$): if it belongs to *SetR*, append it and *SetDR[role_name*$_i$*]* into *MappingUtoR[user_name]*.

step 9. Go to step 7.

step 10. END.

2.3. Construction of Valid Access Space

2.3.1. Basic access rules: Algorithm of constructing *MappingSubtoObjCP* can be described as follows:

step 0. For all $t \in SetT$: Let *MappingSubtoObjCP[t]* = Φ.

step 1. Read policy configuration files and check if there is any allow-rule statement that has not been processed. If it is false, go to step 4.

step 2. If there is an allow-rule statement as **allow** *subject_type_name_list object_type_name_list***:** *class_name_list permission_name_list* (where each kind of name_list can be a sole name or a few names bracketed by **{}** and separated by blank, suppose that the sets of *SetSubType*, *SetObjType*, *SetCPList* can be extracted from these name_lists), Then (i) Compute

MappingTAtoA[SetSubType]=∪ *MappingTAtoA[t]* (*t*∈*SetSubType*) and *MappingTAtoA[SetObjType]*=∪ *MappingTAtoA[t]* (*t*∈*SetObjType*); (ii) Compute Cartesian product Δ= *MappingTAtoA[SetSubType]* ×*SetCPList* ×*MappingTAtoA[SetObjType]*; (iii) For each <t, <c, p>, t_2>∈Δ: append <t_2, <c, p> > into the set *MappingSubtoObjCP[t]*.

step 3. Go to step 2.

step 4. END.

Procedure of **auditallow** rule analysis is similar to above algorithm.

2.3.2. Security context space: The security context space for subjects can be constructed as follows:

$$SCS_{sub} = \{< u, r, t > | \, u \in SetU \wedge r \in MappingUtoR[u] \wedge t \in MappingRtoT[r]\} \quad (2.1)$$

The security context space for objects can be constructed as follows:

$$SCS_{obj} = \{< u, r, t > | \, u \in SetU \wedge r = object\_r \wedge t \in SetT\} \quad (2.2)$$

2.3.3. Valid access space: Algorithm of constructing the valid access space for subjects, i.e. VAS_{sub} can be described as follows:

step 0. For all <*u*,*r*,*t*>∈SCS_{sub}, initialize VAS_{sub} [<*u*, *r*, *t*>]=Φ.

step 1. If there is a type t_1 in <*u*,*r*,t_1>∈SCS_{sub} that has not been processed, go to step 2; else go to step 11.

step 2. Compute and get *MappingSubtoObjCP*[t_1] and SCS_{sub_t1}.={ <*u*,*r*,*t*>|<*u*,*r*,*t*>∈SCS_{sub}∧*t* = t_1}.

step 3. If there is an element <t_2, <c, p>>∈ *MappingSubtoObjCP*[t_1] that has not been processed, go to step 4; else go to step 1.

step 4. Compute and get SCS_{obj_t2}.={ <*u*,*r*,*t*>|<*u*,*r*,*t*>∈SCS_{obj}∧*t* = t_2}.

step 5. Compute Cartesian product Δ= SCS_{sub_t1} ×{<c, p>}×SCS_{obj_t2}.

step 6. If there is an element λ=<<*u_s*, *r_s*, *t_s*>, <c, p>, <*u_o*, *r_o*, *t_o*>>∈Δ that has not been processed, go to step 7; else go to step 3.

step 7. If λ satisfy all **Constrain** rules, go to step 8; else go to step 6.

step 8. If c=**process** and p=**type_transition**, go to step 9; else go to step 10.

step 9. If there are corresponding three **allow** rules required for type transition, go to step 10; else go to step 6.

step 10. Append <<*u_o*, *r_o*, *t_o*>, <c, p>> into VAS_{sub} [<*u_s*, *r_s*, *t_s*>]. Go to step 6.

step 11. END.

Algorithm of constructing the valid access space for objects, i.e. VAS_{obj} can be described as follows:

step 0. For all $<u,r,t> \in SCS_{obj}$, initialize $VAS_{obj}[<u, r, t>]=\Phi$.

step 1. If there is a type $t_2$in $<u,r,t_2> \in SCS_{obj}$ that has not been processed, go to step 2; else go to step 11.

step 2. Compute and get *MappingObjtoSubCP[t_2]*(by the way of traversing above analysis results of **allow** rule list) and $SCS_{obj\_t2}.=\{ <u,r,t>|<u,r,t> \in SCS_{obj} \wedge t = t_2\}$.

step 3. If there is an element $<t_1, <c, p>> \in$ *MappingObjtoSubCP[t_2]* that has not been processed, go to step 4; else go to step 1.

step 4. Compute and get $SCS_{sub\_t1}.=\{ <u,r,t>|<u,r,t> \in SCS_{sub} \wedge t = t_1\}$.

step 5. Compute Cartesian product $\Delta= SCS_{obj\_t2} \times\{<c, p>\} \times SCS_{sub\_t1}$.

step 6. If there is an element $\lambda=<<u\_o, r\_o, t\_o>, <c, p>, <u\_s, r\_s, t\_s>> \in \Delta$ that has not been processed, go to step 7; else go to step 3.

step 7. If λ satisfy all **Constrain** rules, go to step 8; else go to step 6.

step 8. If c=**process** and p=**type_transition**, go to step 9; else go to step 10.

step 9. If there are corresponding three **allow** rules required for type transition, go to step 10; else go to step 6.

step 10. Append $<<u\_s, r\_s, t\_s>, <c, p>>$ into $VAS_{obj}[<u\_o, r\_o, t\_o>]$. Go to step 6.

step 11. END.

2.4. Construction and Analysis of TCB Space

During the construct of security context space for objects and subjects, TCB flag is also set according to the judgement that if the corresponding type *t* is specified as a TCB type by security operator who is responsible for policy configuration.

The TCB space for subjects and objects (marked as $Space_{TCB}$) can be constructed as follows:

$$Space_{TCB} = \{< u,r,t >|< u,r,t > \in SCS_{obj} \cup SCS_{sub} \wedge t \in TCB\_type\} \quad (2.3)$$

And the TCB access space can be figured out by information flow analysis method. The algorithm of integrity analysis as to direct infection can be described as follows:

step 0. For all $<u,r,t> \in SCS_{obj} \cup SCS_{sub}$, initialize $I_{direct}[<u,r,t>]=\Phi$.

step 1. If there is an element $<u,r,t> \in SCS_{sub}$ that has not been processed, go to step 2; else go to step 5.

step 2. Compute and get VAS_{sub} [<*u*,*r*,*t*>].

step 3. If there is an element $<<u\_o, r\_o, t\_o>, <c, p>> \in VAS_{sub}[<u,r,t>]$ that has not been processed, go to step 4; else go to step 1.

step 4. If *p* is kind of **read**/**rw**, append $<<u\_o, r\_o, t\_o>, <c, p>>$ into I_{direct} [<*u*, *r*, *t*>]. Got to step 1.

step 5. If there is an element $<u,r,t> \in SCS_{obj}$ that has not been processed, go to step 6; else go to step 9.

step 6. Compute and get VAS_{obj} [$<u,r,t>$].

step 7. If there is an element $<<u\_s, r\_s, t\_s>, <c, p>> \in VAS_{sub}[<u,r,t>]$ that has not been processed, go to step 8; else go to step 5.

step 8. If *p* is kind of **write**/**rw**, append $<<u\_s, r\_s, t\_s>, <c, p>>$ into I_{direct} [$<u, r, t>$]. Got to step 5.

step 9. END.

The algorithm of TCB integrity analysis can be described as follows:

step 0. For all $<u,r,t> \in Space_{TCB}$, initialize $I_{all}[<u,r,t>] = I_{direct}[<u, r, t>]$ ($<u,r,t> \in Space_{TCB}$) and $I_{all} = \cup I_{all}[<u,r,t>]$ and $I_{all\_new} = I_{all}$.

step 1. Compute $I_{indirect} = \cup I_{direct}[<u, r, t>]$ ($<u,r,t> \in I_{all\_new}$).

step 2. Compute $I_{all\_new} = I_{indirect} - I_{all}$ and $I_{all} = I_{all} \cup I_{all\_new}$.

step 3. If $I_{all\_new}=\Phi$, go to step 4; else go to step 1.

step 4. END.

By traversing and checking elements in I_{all}, it can be decided whether there is any non-TCB subject or object that is potential influence the integrity of TCB space. If that is the fact, the corresponding unsafe **allow** rule is also recorded. Moreover, it can be verified whether there is any audit rule that can do audit for it. If it is audited, it can be decided that the TCB integrity is not threatened because the related **allow** rule has already been monitored. Or it can remind that the security operator ought to pay attention the rule and revise the configuration.

3. Empirical Studies

A prototype is implemented in C language based on above analysis algorithms. And a group of security policy configuration modules are designed based on the architecture of reference policy as to a simplified student-teacher system in order to test the prototype.

3.1. Prototype Implementation

The prototype is made up of reference policy transformation module, security policy extract module, security policy analysis module and analysis result display module. Reference policy transformation module is to read related configuration files, preprocess module interface and macro and other extended elements and generate a single file based on basic policy language of SELinux. Security policy extract module and security policy analysis module are to interpret various declaration statements and rules, extract elementary components for the analysis model and construct related access space and TCB space. Analysis result display module is to traverse corresponding access space or TCB space and output analysis results.

Design of some data structure is elaborated and refined. For example, it is troublesome to solve store problem of logic expressions of **Constrain** rules because not only subjects and objects but also unknown types and etc can be appeared there. Finally, function pointer is selected to store each logic expression and all unknown parameters are taken for a string.

3.2. Test Cases

A simplified student-teacher system is considered as application scenario (refer to Figure 1). Final scores are verified and registered by administrative teacher rather than course teacher so as to ensure absolute correctness of scores. In addition, they are many-to-many relationships among course teachers, courses and students. And some programs are introduced to perform task of identity verification. A student can't submit his homework to corresponding files or directories unless he has right identity of course election. It is similar whenever a course teacher submits scores or an administrative teacher register scores. Although related verification functions are not provided by security mechanism of SELinux, policy analysis ought to be thought on such suppositions.

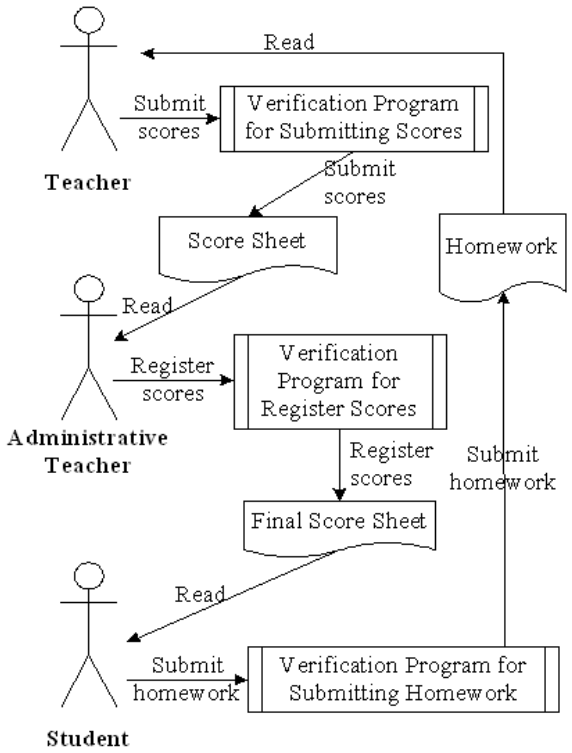

Figure 1. A Simplified Student-teacher System

Types for students, teachers, administrative teachers, files and verification programs are declared as well as interface for accessing related files and access relations between verification programs and files are specified in our policy configuration files, which can be divided into four security policy modules, i.e. student policy module (including policy file *student.te*, refer to Figure 2), teacher policy module(including policy file *teacher.te*), administrative-teacher policy module(including policy file *collegeadmin.te*) and file-and-program policy module(including policy file *collegefile.te* and *collegefile.if*, refer to Figure 3 and Figure 4 respectively).

```
########################################
#
# Declarations
#

type student_t;
typeattribute student_t domain1;
domain_base_type(student_t)

role student_r types domain1;

########################################
#
# Local policy
#

if (time_write_record)
{
collegefile_record_write(student_t)
}
collegefile_work_write(student_t)
collegefile_source_read(student_t)
if (time_read_mark)
{
collegefile_mark_read(student_t)
}

collegefile_premark_notwrite(student_t)
collegefile_mark_notwrite(student_t)
```

Figure 2. Segment of Policy Configuration File *student.te*

```
attribute domain1;

type accesswork_t;
typeattribute accesswork_t domain1;
type accesswork_exec_t;
domain_base_type(accesswork_t)
domain_entry_file(accesswork_t,accesswork_exec_t)

type coursemark_t;
```

Figure 3. Segment of Policy Configuration File *collegefile.te*

```
## <summary>College file access rules</summary>

interface(`collegefile_source_read',`
        gen_require(`
                type coursesource_t;
        ')

        allow $1 coursesource_t:file { read };
')

interface(`collegefile_work_write',`
        gen_require(`
                type accesswork_t;
                type accesswork_exec_t;
        ')

        allow $1 accesswork_exec_t:file { execute };
        allow $1 accesswork_t:process transition;
        type_transition $1 accesswork_exec_t:process accesswork_t;
')

interface(`collegefile_mark_read',`
        gen_require(`
                type coursemark_t;
        ')

        allow $1 coursemark_t:file { read };
')

interface(`collegefile_mark_notwrite',`
        gen_require(`
                type coursemark_t;
        ')

        neverallow $1 coursemark_t file { write };
')

interface(`collegefile_premark_read',`
        gen_require(`
                type coursepremark_t;
        ')

        allow $1 coursepremark_t:file { read };
#       auditallow $1 coursepremark_t:file { read };
')
```

Figure 4. Segment of Policy Configuration File *collegefile.if*

3.3. Results and Analysis

Tests are done from four aspects: (1) to verify all accessible objects in appropriate access modes when a subject is specified by security context in the form of <*user_name*, *role_name*, *type_name*>; (2) to verify all subjects that can access the given object in appropriate access modes when an object is specified by security context in the form of <*user_name*, *object_r*, *type_name*>; (3) to verify if a subject specified by given security context can access an object specified by given security context in appropriate access modes; (4) to verify the integrity check for information sources of TCBs. And the former three items focused on validity analysis (refer to Figure 5) while the last one is about integrity analysis.

```
Please choose 1~4 to decide your operation:
  1.Display all the access permission for one subject.
  2.Display all the access permission to one object.
  3.Display all the access permission from one subject to
    one object.
  4.Quit.
1
Please input the context of the subject as "user role type":
student_u student_r student_t
(file, execute)         <student_u, object_r, accesswork_exec_t>
(file, execute)         <college_admin_u, object_r, accesswork_exec_t>
(file, execute)         <teacher_u, object_r, accesswork_exec_t>
(process, transition)           <student_u, student_r, accesswork_t>
(file, read)            <student_u, object_r, coursemark_t>
(file, read)            <college_admin_u, object_r, coursemark_t>
(file, read)            <teacher_u, object_r, coursemark_t>

Please choose 1~4 to decide your operation:
  1.Display all the access permission for one subject.
  2.Display all the access permission to one object.
  3.Display all the access permission from one subject to
    one object.
  4.Quit.
2
Please input the context of the object as "user role type":
student_u object_r accesswork_exec_t
<student_u, student_r, accesswork_t>            (file, entrypoint)
<college_admin_u, collegeadmin_r, accesswork_t>         (file, entrypoint)
<teacher_u, teacher_r, accesswork_t>            (file, entrypoint)
<student_u, student_r, student_t>               (file, execute)
<college_admin_u, collegeadmin_r, student_t>            (file, execute)
<teacher_u, teacher_r, student_t>               (file, execute)

Please choose 1~4 to decide your operation:
  1.Display all the access permission for one subject.
  2.Display all the access permission to one object.
  3.Display all the access permission from one subject to
    one object.
  4.Quit.
3
Please input the context of the subject as "user role type":
student_u student_r student_t
Please input the context of the object as "user role type":
student_u object_r accesswork_exec_t
<student_u, student_r, student_t>               (file, execute)         <student
_u, object_r, accesswork_exec_t>
```

Figure 5. Some Test Results for Validity Analysis

Integrity analysis is executed on the premise that TCBs are specified. Then all information flow into TCBs is analyzed and test results including records that impaired integrity are

outputted in a file. In the supposed application scene, administrative teachers (i.e. ***collegeadmin_t***), mark-writing verification program (i.e. ***accessmark_exec_t***), mark file (i.e. ***coursemark_t***) and etc can be specified as TCBs.

It is always thought that the systematic integrity have been threatened once a TCB subject (e.g. administrative teacher) reads information from a non-TCB object (e.g. submitted mark sheet draft). For example, if some audit rules such as "**auditallow $1 coursepremark_t:file { read };**" is denoted by comment sign "#" (refer to Figure 4), records that infected integrity can be found from the output file (refer to Figure 6). By contraries, such records will be cleared away once the comment sign "#" for that rule is deleted.

```
The TCB node is <teacher_u, object_r, coursemark_t>
        The Target node is <student_u, student_r, student_t> Non-TCB. And the break rule is:
                allow collegeadmin_t coursepremark_t:file read;

The TCB node is <teacher_u, teacher_r, collegeadmin_t>
        The Target node is <student_u, student_r, student_t> Non-TCB. And the break rule is:
                allow collegeadmin_t coursepremark_t:file read;
        The Target node is <student_u, object_r, coursepremark_t> Non-TCB. And the break rule is:
                allow collegeadmin_t coursepremark_t:file read;
        The Target node is <college_admin_u, object_r, coursepremark_t> Non-TCB. And the break rule is:
                allow collegeadmin_t coursepremark_t:file read;
        The Target node is <teacher_u, object_r, coursepremark_t> Non-TCB. And the break rule is:
                allow collegeadmin_t coursepremark_t:file read;
```

Figure 6. Some Test Results for Integrity Analysis

Above results show that the prototype can get not only all objects with corresponding permissions that any subject with specified security context can access but also all subjects with corresponding permissions that any object with specified security context can be accessed. Moreover, all rules that could potentially influence integrity of TCB subjects and objects can be detected.

4. Summary

Algorithms are designed in this paper to perform validity analysis and integrity analysis of SELinux policies. And a set of example policies are taken as test cases, which is devised according to a simplified student-teacher system. The initial test results are satisfactory.

Nevertheless, the supposed student-teacher system for test is just a simple application scene and access control relations among its subjects and objects are limited and can't contain all cases. In addition, a few special signs, Boolean variables and macro blocks are ignored during analysis process in this paper. All these details ought to be full considered in the future research. In addition, both method and prototype for analysis must be improved farther for practicability. And performance of analysis ought to be considered in our future work.

Acknowledgements

This research was supported by the Fundamental Research Funds for the Central Universities (No.2009JBM019).

References

[1] G. Zhai, J. Zeng, M. Ma and L. Zhang, "Implementation and Automatic Testing for Security Enhancement of Linux Based on Least Privilege", International Journal of Security and Its Applications, vol. 2, no. 3, **(2008)**, pp. 93-100.
[2] G. Zhai and Y. Li, "Analysis and Study of Security Mechanisms inside Linux Kernel", In: Proceedings of 2008 International Conference on Security Technology (SECTECH2008), IEEE Computer Society, California, **(2008),** pp. 58-61.
[3] G. Zhai, H. Niu, N. Yang, M. Tian, C. Liu and H. Yang, "Security Testing for Operating System and Its System Calls", In: D. Slezak et al. (Eds.): SecTech 2009, CCIS (Communications in Computer and Information Science), vol. 58, Springer-Verlag Berlin, Berlin, **(2009)**, pp.116-123.
[4] F. Mayer, K. MacMillan and D. Caplan, "SELinux By Example: Using Security Enhanced Linux", Prentice Hall, New Jersey, **(2006)**.
[5] G. Zanin and L. V. Mancini, "Towards a Formal Model for Security Policies Specification and Validation in the SELinux System", In: Proceedings of the 9th ACM Symposium on Access Control Models and Technologies, Association for Computing Machinery (ACM), New York, **(2004)**, pp. 136-145.
[6] G. Zhai, W. Ma, M. Tian, N. Yang, C. Liu and H. Yang, "Design and Implementation of a Tool for Analyzing SELinux Secure Policy", In: Proceedings of 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (ICIS 2009), Association for Computing Machinery (ACM), New York, **(2009)**, pp. 446-451.
[7] G. Zhai and T. Wu, "Automatic Analysis Method for SELinux Security Policy", International Journal of Security and Its Applications, vol. 6, no. 2, **(2012)**, pp. 229-234.

Authors

Gaoshou Zhai received the B.Sc., Master and Ph.D. degrees in 1993, 1996 and 2000 respectively. From 2000 to 2002, he was a lecturer in the School of Computer and Information Technology at Beijing Jiaotong University. Since 2002 he has been an associate professor and since 2007 he has been vice director of the Department of Computer Science. From January to May in 2006, he had been to the department of computer science at UIUC as a visiting scholar. His research interests include operating systems, system security, system software design and automatic tools for software engineering, algorithm analysis and design, artificial intelligence and intelligent traffic systems. In these areas he has published more than 40 papers in journals and conference proceedings. He served as program committee member of SERA2009 and invited review specialist for Chinese Science and Technology Papers Online sponsored by Centre for Science and Technology Development, MEPRC. He is also invited to review papers for Journal of Xi'an Jiaotong University, Journal of Beijing Jiaotong University, Journal of Lanzhou Jiaotong University, ICCIT2009 and ICSAI2012. He is a member of ACM and SERSC, a senior member of CCF and IACSIT.