2015

ACCESS CONTROL PROGRAMMING LIBRARY AND EXPLORATION SYSTEM

Zhitao Qiu
Michigan Technological University

ACCESS CONTROL PROGRAMMING LIBRARY AND EXPLORATION SYSTEM

By

Zhitao Qiu

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2015

This report has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Report Advisor:	*Dr. Jean Mayo*
Committee Member:	*Dr. Ching-Kuang Shene*
Committee Member:	*Dr. Min Wang*
Department Chair:	*Dr. Min Song*

Dedication

To my mother, teachers and friends

who didn't hesitate to criticize my work at every stage - without which I would neither be who I am nor would this work be what it is today.

Contents

List of Figures

List of Tables

Abstract

The high complexity of advanced security models in the modern trusted systems requires an effective formal education for students. Education access control tools have been promoted. Though they can benefit the learning through analyzing or visualizing access control policies, few of them are designed to teach development of access control policies.

In this report, we propose an access control programming library which can provide students hand-on experience with the effect of an access control policy on a running program. A student can write a policy and then run programs under the policy. The Programming Library provides a system call wrapper API which enforces the developed policy in the execution of a process. The program and policy exist at the user level. No administrator access is required. From another hand, students can monitor how the process is affected by the policy through this tool and adjust the rules accordingly. Furthermore, an Access Control Shell was designed as an interactive command interface to execute the wrapper APIs, as well as a test platform or a container to launch student program. Finally, we defined an interface for further communication with existing visualization tools, which depict the program execution using visualizations specific to the policy model.

Chapter 1

Introduction

1.1 Introduction

In a trusted system, the principle of least privilege is applied to make sure that only essential information or resources are provided to complete a task. Fine-grained enforcement of the policy rules and mandatory controls for the organization security policy are required. Advanced access control technologies and sophisticated abstract security models have evolved, such as Domain Type Enforcement (DTE) [1], Multi-level Security (MLS) [6] and Role-based Access Control (RBAC) [7]. These models enforce the above mentioned principle effectively in modern security systems.

The real system usually contains mixture of those advanced models which significantly

increase the complexity of learning. Hence, formal education must be provided to students to apply the sophisticated technologies correctly [2]. Yet, in order to achieve effective education in this area, students need hands-on experience to develop a deeper understanding of multiple security models.

There are some implementations of these access control security models like Redhat Linux. However, these are problematic for direct use in security education due to the complexity of large policies with mixed security models and great administration overhead. Instead, a simplified and customized pedagogical system is more suitable for teaching principles and performing assignments through combining different technologies and tools. This cuts the learning curve and encourages students to gain experience.

1.2 Motivation

Pedagogical visualization tools (DTEvisual, MLSvisual, RBCvisual) have been designed separately to depict the security models and allow students to develop or analyze a policy [3,4]. These, which deploys specific graphs to analyze the security policies , or allow students to develop a policy form scratch. These tools benefit students by building a solid understanding in the security models, and improve the effectiveness of security education.

However, in order for students to understand the effect of a policy on the dynamic system of processes that comprise a real system, an access control programming library that wraps the system call API to enforce the Mandatory Access Control policies. This allows students to monitor and explore the credentials of processes during their execution.

We developed a system call wrapper API that allows students to run a program under a policy they develop. RBAC, MLS, and DTE are all supported. Policies are implemented at the user level. No administrator privilege is required. The running program can interact with existing model-specific visualization systems to further help students understand policy development under modern models of access control.

Chapter 2

Background

This chapter introduces the fundamental technologies of Mandatory Access Control and related work.

Mandatory Access Control (MAC) is usually enforced by an organization's security policy and is not at the discretion of any single user[2]. In MAC, even the root user is constrained by the policy. In contrast, Discretionary Access Control (DAC) system users or owners have control over the resources assigned. Just like in Unix file system, DAC users can change file permissions for User, Group and Others [5].

The following sections will introduce three types of MAC security models in detail. The visualization tool for each model has been developed to assist students in analyze and developing related policies. Each tool contains different graphs based on the

characteristics of the specific models. These tools operate on the policies written in model specific languages and implement parsers to translate the policy specification to access control matrix. The syntax for each model specification is introduced in the tool section.

2.1 Domain and Type Enforcement (DTE)

2.1.1 Introduction

DTE is an enhanced type enforcement access control technology, where system is partitioned into different access control domains and types [1]. In type enforcement for UNIX, domains are defined as collections of active subjects (processes) and types are associated with passive objects (files, messages, other resources) [1].

Instead of using complex tables to stand for authorized access modes between domains and types or domains and domains, DTE System enforces the access control policies through a high level language called DTE Language (DTEL) [1]. Access to objects in different domains are decided through the specifications written in DTEL. In a DTE UNIX system, access control rules in a DTEL specification are processed and enforced at the level of UNIX kernel [1]. Access rights are granted to domains which group the processes. Therefore, even root processes in DTE UNIX are subjected to

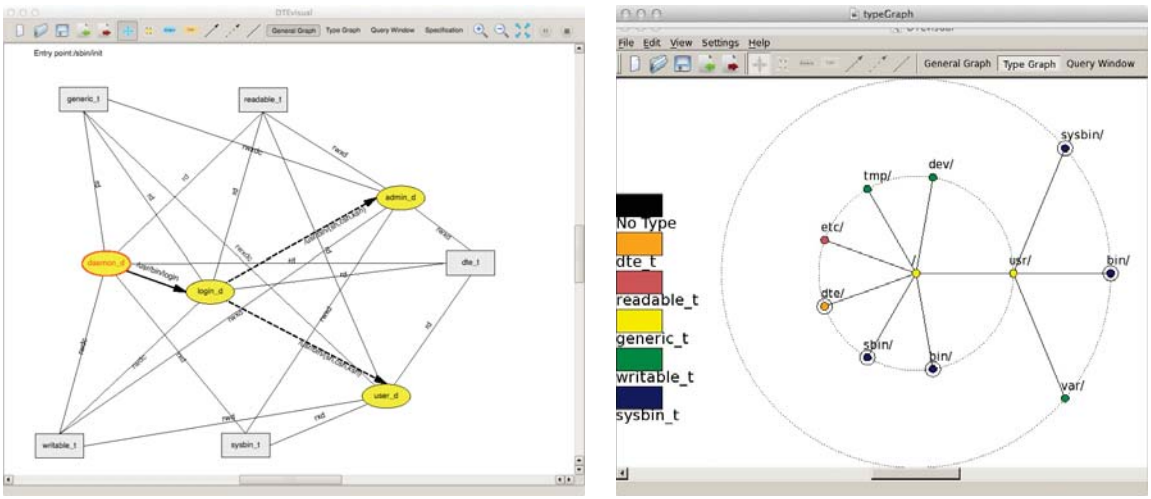
DTE policy.

DTE provides a process oriented access control[2] by placing statements on domains, which constrain the exact object types that each process or subject can access. This helps students to think about the principle of least privilege from the view of subject precisely. Aforementioned benefits of DTE and further applications in education are introduced in Carr and Mayo's paper [2], which forms one important foundation for this project.

The syntax of DTEL is described in next subsection.

2.1.2 DTEvisual

DTEvisual is a visualization tool to facilitate the learning of DTE, which provides two graphical visualizations for a selected DTE policy: the General Graph and the Type Graph [3]. The General Graph depicts domains, types, transitions between domains, and access from domains to a given type.The Type Graph shows object types and is displayed using a radial tree. All the files at the same level in the directory hierarchy are connected using a dotted circle. Directories are indicated by a trailing slash. Different types are identified through color of the labels [3]. Figure 2.2 shows the two type of graphs.



(a) General Graphs (b) Type Graph

Figure 2.1: DTE General Graph and Type Graph

Users can create a DTE specification from scratch using the graph operations, or they can import the specification, and then analyze and modify the graph.

DTE specification in DTEvisual is a text-based policy which follows specific syntax of Domain Type Enforcement Language (SDTEL). The list 2.1 is a sample of SDTEL specification. Please see detailed specification syntax introduction in Appendix A.

Listing 2.1: DTE Sample Specification

```
type dte_t,readable_t,generic_t,writable_t,sysbin_t,log_t;

domain login_d = (/usr/bin/login),

        (cdrw->writable_t),

        (exec->student_d,admin_d);

        (dr->generic_t,dte_t),

domain admin_d = (/usr/bin/{sh,csh,ksh}),

        (cdrwx->generic_t);

        (drwx->dte_t,writable_t,readable_t,sysbin_t),

domain student_d = (/usr/bin/{sh,csh,ksh}),

        (drx->sysbin_t),

        (cdrwx->generic_t),

        (dr->readable_t),

        (drw->writable_t,dte_t);


initial_domain = login_d;

assign -r log_t /usr/data/log;

assign -r generic_t /;

assign -r readable_t /etc/test/;

assign -r -s sysbin_t /usr/bin,/bin,/sbin;

assign -r writable_t /usr/data/record,/temp;

assign -r -s dte_t /dte/policy;
```

2.2 Multi-level Security (MLS)

2.2.1 Introduction

MLS is based on the Bell-LaPadula model, which is consistent with military-style classifications [13]. MLS uses categories on objects and clearances for subjects. In the confidentiality classification, security clearances reflect the order of security sensitive levels; categories comes from the "need to know" principle[13] that subjects should only be granted to read the objects required by the job. For example, the security clearances range from top-secret to public, and categories range from weapon to mobile-device. Security levels combine clearances and categories. For example, Jack is cleared into the level (top-secret, weapon, mobile-device), and a tank document is at the level of (secret, weapon). In this example, Jack can read the tank document. The dominates relation can be suggested from this example, which is defined as below:

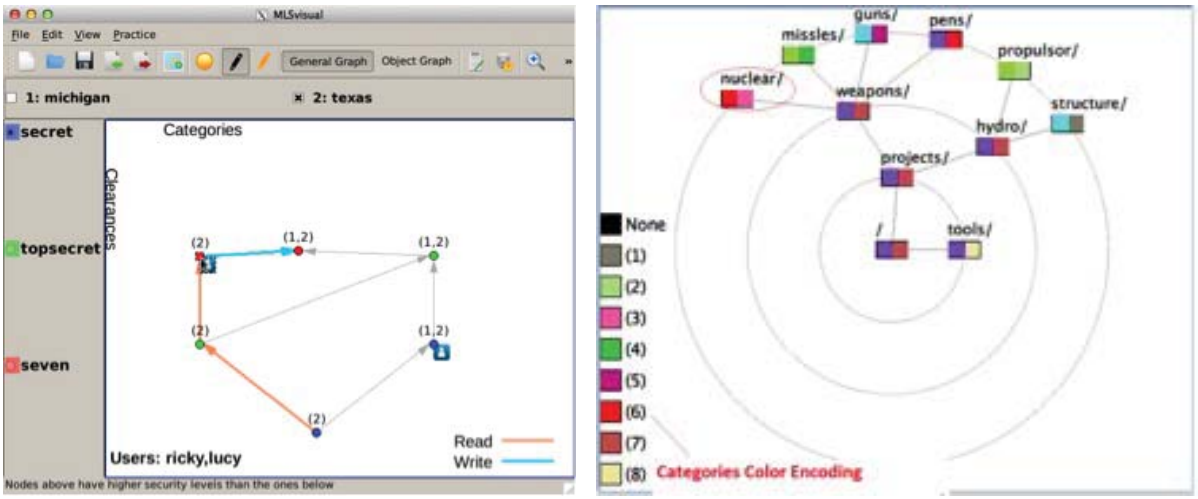
> Dominates: the security level (L, C) dominates the security level (L', C') if and only if $L' \leq L$ and $C' \subseteq C$[13].

The set of security levels form a lattice corresponding to the domination relation from the power set of that set[13]. The domination relation decides the access rights. In the example above, level (top-secret, weapon, mobile-device) dominates level (secret,

weapon). Jack can read access to the tank document. The access control process induced from the domination relation can be simply characterized by the phrase "no read up and no write down" [5]. This phrase reflects two important properties: simple security property and star-property. The simple security property prevents a subject read an object of a higher security level; meanwhile, the star-property requires a subject not write to a lower security level object [5]. Hence, in this example, Star-property further can prevent Jack write access to the tank document, which constrains the information flow from high to low.

2.2.2 MLSvisual

The visualization tool MLSvisual focuses on the interpretation of the security level hierarchy as well as the read and write permissions to files for users with different clearance levels. Figure 2.2 shows two types of MLS graphs. The General Graph allows users to explore the lattice formed by the set of security levels by building portions of interest. For example, the user may select a node and add its predecessors or successors, or select two nodes and build that portion of the lattice that connects them. The Object Graph depicts the security levels assigned to objects, which forms concentric circles surrounding the root directory. Each node consists of two colors, left for clearance and the right for the category [4].



(a) General Graphs (b) Object Graph

Figure 2.2: MLS General Graph and Object Graph

MLS Specification syntax is similar to SDTEL, but has its own semantics. List 2.2 is a sample of this. Detailed illumination can be found in Appendix A.

Listing 2.2: MLS Sample Specification

```
clearances:top<secret<topsecret<seven
#Category section: list all the categories, no order is required. Syntax: "↩
    categories: category1, category2, ..."
categories:Michigan,Washington


#Assign security levels to directories in the file system. Syntax:  "assign ↩
    clearance:category1:category2:... [-r | -s] directory1, directory2, ..."
assign secret:Michigan  -r /usr, /top
assign seven:Washington     /usr/weaponCatalog


#Assign security levels to users. Syntax:  "users clearance:category1:category2:...↩
     user1, user2, ..."
users secret:Michigan:Washington  Ning, Mike
users seven:Washington            David, Jack
```

2.3 Role-based Access Control(RBAC)

2.3.1 Introduction

Role-based Access Control models only assign access rights to roles[7]. Users are assigned to roles, and then acquire permissions through inheritance from the roles. This feature is similar to the UNIX group based access control; it can also serve as implementation for DAC, as well as MAC. As a generalized approach, RBAC models are now widely accepted[7].

2.3.2 RBCvisual

RBCvisual defines two graphical views: Matrix view and Hierarchy view. Matrix view presents two tables about the role-to-object permission and user-to-role assignment. Hierarchy view in Figure 2.3 depicts the relationships in a graph where nodes stand for users and roles, and edges show the inheritance relationship between the nodes. Another graph reflects the permission assignment corresponding to the highlighted node.

RBAC Specification syntax is more straightforward than that of the DTE and MLS

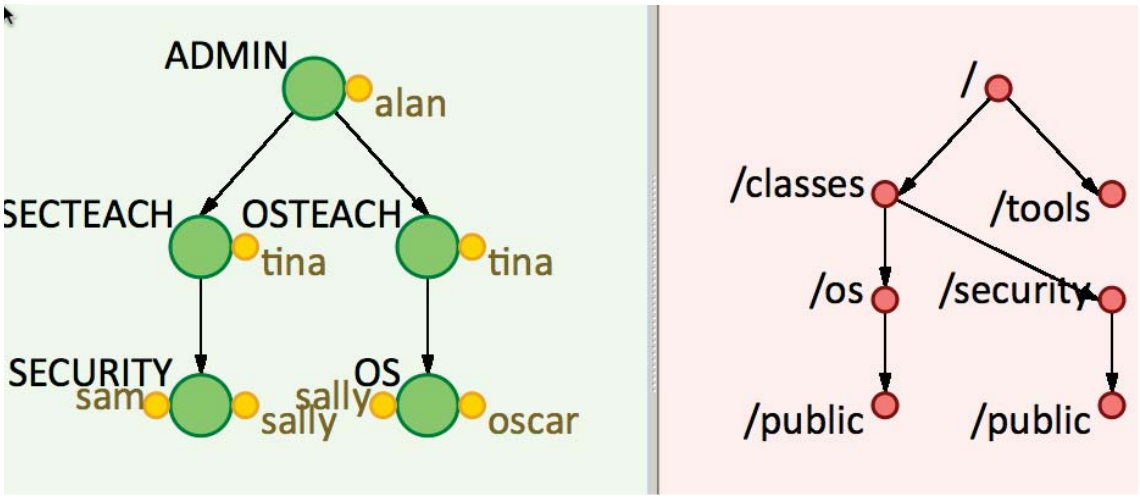


Figure 2.3: RBAC Hierarchy View

models. It contains three sections which define the hierarchy of the roles, the role-to-user assignment and role-to-object assignment. Please find a sample specification with a simple introduction in the List 2.3.

Listing 2.3: RBAC Sample Specification

```
#Section 1: define the inheritance relation between the roles

inheritance: Admin > Users

inheritance: Users > Guest


#Section 2: define the role-to-user assignment

user: Guest Gerry

user: Users Ping

user: Admin Lucy, Ping


#Section 3: define the role-to-object assignment

object: Guest,Admin, Users  r,w,x /TestFile1

object: Guest,Admin, Users  r,w /TestFile1

object: Users, Admin r,w /TestFile2

object: Admin r,w,x /TestFile2
```

Chapter 3

Implementation

3.1 Introduction

The access control programming library and related exploration system is based on the access control policy languages adopted in the visualization tools, and the parsers are extracted from the visualization tools mentioned in Chapter 2.

The system design contains two major components: a Programming Library (wrapper API), a Policy Engine (policy translation and access control analysis).The system also implements an interface to the visualization tools. The design of the exploration system is depicted in Figure 3.1. The wrapper API are written in C. The Policy Engine and other parts are written in Python.

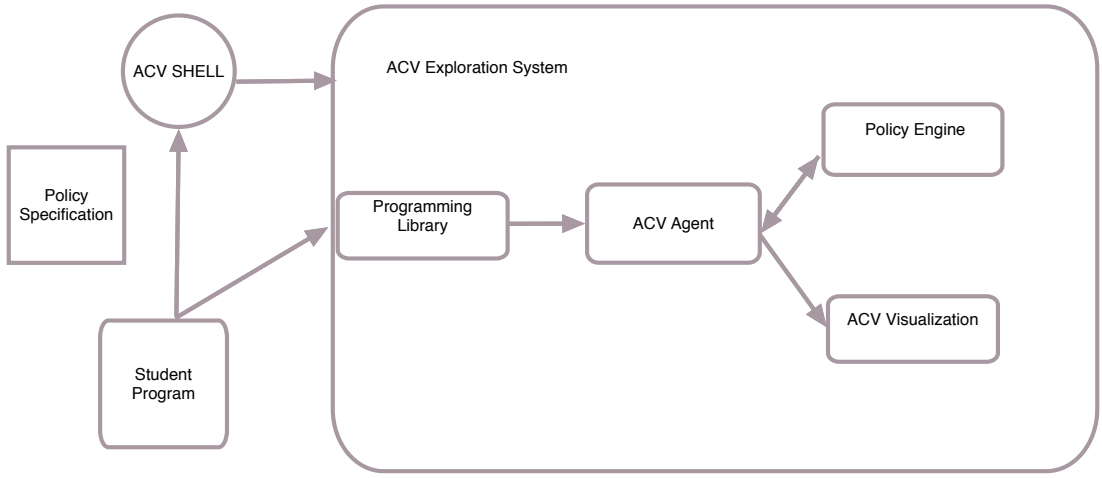


Figure 3.1: Programming Library Exploration System Design

The Programming Library provides a wrapper API for the system call API. The wrapper for a system call will make an appropriate access request before performing the requested system call. Then the Engine Agent will forward the request to the Policy Engine to make a decision. This request can trigger the visualization tool to launch if visualization is configured and tools are in place. The model-specific graphs can depict the policy rules in the visualization window.

TCP was used for the communications between modules including the connection between the library and the policy engine, and the connection between the policy engine and the visualization tools. The implementation of the process communication for the connection between the library and the policy engine is one of the difficulties in this project. Details regarding this will be addressed in the section of this chapter.

Finally, an Access Control Shell was created, which is an interactive command line

interface for the application and test of the library wrapper API. It will be introduced in Chapter 4.

3.2 Programming library

3.2.1 Introduction

The Programming library allows students to monitor the execution of a program under a selected policy. The program usually operates on some system resources such as creating a file or changing a file name. The access control wrapper API of the Programming Library will be used to replace the system call API.

The next two sections will describe the design principles of the Programming Library and introduce the implementation of the API functions.

3.2.2 Wrapper API Design Principles

The Programming library effectively wraps the system calls through which access requests are made. The names of routines in the wrapper API are comprised of the string "acv_" prepended to the corresponding system call. Additionally, some system

calls require parameters that are not part of the POSIX interface. For example, a call to execvp may contain a parameter that designates the domain in which the process will execute on successful completion of the routine.

Each model shares the same API interface, but extracts different parameters to form the access control request. For example, in the rename wrapper API, DTE extracts Domain ID, whereas RBAC extracts Role Name and User Name.

3.2.3 API Initialization and Termination Function

Before using the wrapper API, function acv_init has to be called to execute the required initialization. acv_init handles the initial setting for the access control process, such as the policy file name with the full path; launch the Engine Agent, which creates model specific policy manager and starts TCP server. acv_init will make sure only one Engine Agent exists in the system, so that only currently selected policy file will be enforced. At the end of the student programs, acv_end must be called to clean up the resources including termination of the Engine Agent.

```
int acv_init(const char *specPath,const char * acv_param1, const char *acv_param2, ↩
    const char *vflag,char *envp[])

#   acv_init initializes the access control environment variables shared by the ↩
    wrapper API based on the input, creating a new process of the Engine Agent, ↩
    which imports the policy and listens to the access control request messages.
    #Input arguments    :
        specPath: specification name,
        acv_param1, acv_param2: model specific, refer to Table 3.1
        vflag: need visualization or not
        envp: defaulted to NULL; for environmental variables
    #Usage example:
        acv_init(path, RoleName, username, "-V", NULL);


int acv_env_init(int argc, char *argv[])
#   acv_env_init is used with the test tool AC Shell, which already calls acv_init.↩
     Hence, acv_env_init will transfer the context of AC Shell to the student ↩
    program through the argc, argv variables passed by main entry function.

int acv_end()
#  acv_end cleans up resources such as closing sockets and terminating the Engine ↩
    Agent process.

#Return value for above three functions: 0 for success, -1 for failure (Upon ↩
    failure, program should exit).
```

Table 3.1
acv_init Parameters Mapping Table

Model	acv_param1	acv_param2
RBAC	RoleName	Username
DTE	InitialDomainID	NONE
MLS	UserName	NONE

3.2.4 System Call Functions

This section will introduce the system call wrapper functions in the API. A selected set of functions are implemented in the wrapper API. Each function will check the model type and makes appropriate access control requests to get the authorization before calling the POSIX system call. First, we will introduce the common steps shared by the individual function, and then illustrate the details of each function.

3.2.4.1 Common Steps of Making Access Control Request

Generally, a system call wrapper function extracts access control parameters based on the selected model, and then builds an access control request, which is further forwarded to the Policy Engine for decision making. In other words, the implementation for each security model will extract their own parameters and permission modes.

First, the wrapper API decides which kinds of objects to extract based on the targets

of the system call function.

Second, the required access privileges of the objects are converted according to the permission attributes of the operation. For example, in the RBAC model, the rename function needs $w + x$ access right for the parent folder of the target file.

Moreover, some model specific parameters like RoleName and UserName are obtained from the access control global data. Those are initialized through acv_init and can be changed in the program through the Set functions of the Programming Library:

```
int set_username(const char *ac_typevalue);
int set_domainname(const char *ac_typevalue);
int set_rolename(const char *ac_typevalue);
```

At the end, the wrapper API receives the access control result from the Policy Engine. It returns a failure code of rejection or performs the system call function if allowed. Please note that the ultimate execution of a system call function is still subjected to the underlying system access control, such as the DAC of UNIX. For example, even when a MLS policy permits the execution of acv_write to a file, if the owner of the file sets the permission mode to read only like mode 0444 in Linux, the write system call will fail.

3.2.4.2 Functions Introduction

The following functions in the wrapper API are based on related POSIX Libc function definition [8]. For detailed arguments format and usage of the functions, please refer to the GNU POSIX Manual[8]. We will mainly introduce functions of the API from the view of access control .

```
int acv_rename(const char *oldpath, const char *newpath ) ;
#   acv_rename Rename a file by wrapping LIBC function rename. acv_rename function ↩
    is used to change the name of a file.
#   acv_rename decides whether the parent folder or the file itself will be the ↩
    object for access control depending on the two cases below:
#   1) If target is a file, we are going to rename it, so the object is the file ↩
    itself;
#   2) If target is a directory, we move the file to target folder, so the object ↩
    is the parental directory of the file.

int acv_execvp(const char *file, char *const argv[],const char * acv_param );
#   acv_execvp executes a file by wrapping LIBC function rename. Child process ↩
    executing the program specified by file will be forked[13]. Execution ↩
    permission will be requested and appropriate mode value will be fetched based ↩
    on the model. acv_param is NULL, except for DTE, it is the Domain ID for the ↩
    command to execute.



int acv_creat(const char *pathname, mode_t mode);
```

```
#   acv_creat creates a file by wrapping LIBC function creat.The argument mode ↩
    follows the same definition of creat. acv_creat is equivalent to acv_open with ↩
    flags equal to O_CREAT | O_WRONLY |  O_TRUNC [14]. Permission rights ↩
    corresponding the required value in the flags are checked. Write permission is ↩
    checked for the parental folder of the pathname  (Table 2).

int acv_write(int fildes, const void *buf, size_t nbytes);
#   Write a file by wrapping LIBC function write. fildes is returned from acv_creat↩
    . Access right is checked through acv_creat.

int acv_read(int fildes, void *buf, size_t nbytes);
#   read a file by wrapping LIBC function read. fildes is returned from acv_creat. ↩
    Access right is checked through acv_creat.

int acv_open(const char *pathname, int oflags);
#   Open a file by wrapping LIBC function open.The argument mode follows the same ↩
    definition of open. Permission rights corresponding the required value in the ↩
    flags are checked.

int acv_access(const char * filename, int mode);
#   Delete directories by wrapping LIBC function rmdir. The argument mode follows ↩
    the same definition of access. Permission rights for the file corresponding the↩
     required value in the modes are checked.

int acv_mkdir(const char * pathname, int mode);
#   Create directories by wrapping LIBC function mkdir. The argument mode follows ↩
    the same definition of mkdir. Write permission is checked for the parental ↩
    folder of the pathname  (Table 3.2).

int acv_rmdir(const char * pathname);
#   Delete directories by wrapping LIBC function rmdir. Write permission is checked↩
     for the parental folder of the pathname  (Table 3.2).
```

```
int acv_chdir(const char *pathname);
#   Change to the target directories by wrapping LIBC function chdir. Search ↩
    permission is checked for the parental folder of the path (Table 3.2 For MLS, ↩
    it corresponds to the read right.

int acv_remove(const char *pathname);
#   Delete a file by wrapping LIBC function remove. Remove permission is checked ↩
    for the parental folder of the path (Table 3.2).

Return value for above  functions:0 for success, -1 for failure
```

Table 3.2
Permission Mode Mapping Table For Accessing a Directory

Model	search	read	remove/write
RBAC	x	rx	wx
DTE	d	rxd	wxd
MLS	r	r	w

3.3 Access Control Engine

3.3.1 Design Structure

Figure 3.2 depicts the design of the policy engine.

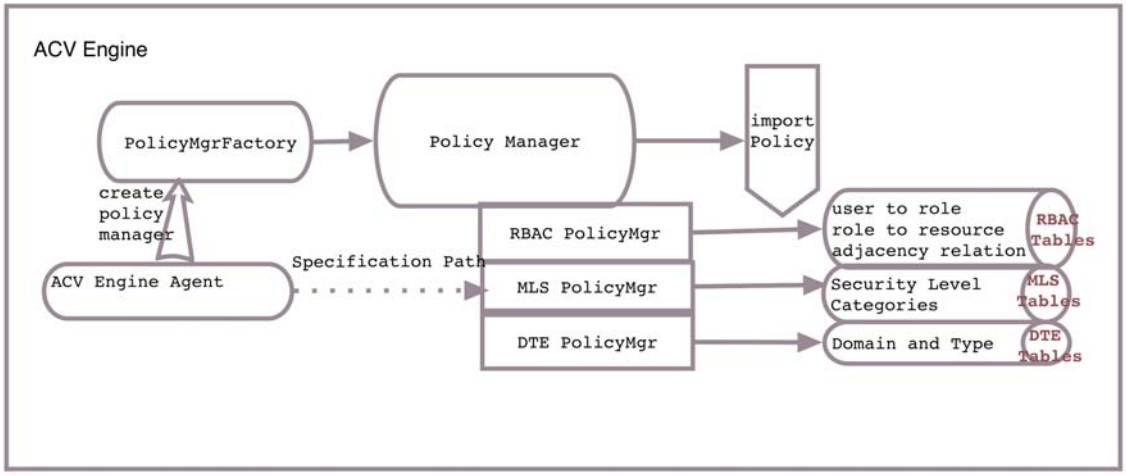


Figure 3.2: Policy Engine Structure

The Policy Engine consists of 3 components, each of which is implemented in one python class: an Engine Agent, a TCP Server, and a Policy Manager (parser and decision function). The Engine Agent containing the main method is the entrance of the Policy Engine. The Policy Engine runs as a single process which is launched through the system call of python acv_agent.py in acv_init with appropriate parameters such as a model policy and a visualization flag. This section will introduce Engine Agent

and Policy Manager. TCP Server will be introduced in next section .

During the execution of python acv_agent.py, an instance of Engine Agent is created. The Engine Agent will further create a Policy Manager instance, which further imports the policy. Then, a TCP Server instance will be created and starts to listen for access control requests. After processing by the TCP Server, the itemized message will be passed to the Policy Manager for access control decision. Meanwhile, the Engine Agent will also bring up a visualization tool if configured. A copy of the request will be transferred to the tool.

The Policy Manager is the core of the Policy Engine. It is responsible for parsing the policy and decision making for the access control requests. Abstract Factory Design Pattern which uses generic interface of the factory to create concrete objects[5] was adopted in the implementation of Policy Manager. The abstract class defines two interfaces import_policy and acv_query. A concrete policy manager for a specific model is chosen in the run time when running the program. This greatly increases the flexibility of Programming Library. Students can write the same program and execute it under different security models. The Programming Library can decide at run time which exact policy parser and decision logic to use based on the necessary input, such as the model type or policy path.

For example, given a model type of RBAC, an instance of classs RBACPolicyMgr which implements the Policy Manager for RBAC is created through the factory. Then,

RBACPolicyMgr further calls importPolicy function, where the RBAC parser will translate the policy and extract access control matrices from the specification statements written in the corresponding RBAC language mentioned in Chapter 2. Upon receiving an access control request, the decision routing will be called and appropriate response will be returned to the originating API function through TCP message.Upon receiving a request, acv_query of the Engine Agent will pass the itemized message to the queryManager function of RBACPolicyMgr, which further calls RBAC decision function queryUserAccessObjFromRole to grant the access.The decision function will be described in next subsection.

Below two lists show the code snapshot for the Policy Manager Factory and the Engine Agent.

Listing 3.1: ACV Engine PolicyMgr Factory

```
class ACVEnginePolicyMgrFactory(object):
    @staticmethod
    def create_policy_manager(policy_type):
        if policy_type == "RBAC":
            return RBACPolicyMgr()
        elif policy_type == "DTE":
            return DTEPolicyMgr()
        elif policy_type == "MLS":
            return MLSPolicyMgr()
```

Listing 3.2: Engine Agent

```
class ACV_Agent():
    def __init__(self, PolicyPath): #read from command arguments
        self.acv_policy_manager =       ACVEnginePolicyMgrFactory.↩
    create_policy_manager(getModelType(PolicyPath))
         self.acv_policy_manager.importPolicy(PolicyPath)
   def acv_query(self, itemized_acv_message):
         answer,output = self.acv_policy_manager.queryManager(itemized_acv_message)
```

3.3.2 Access Control Decision

The decision routing in each model specific policy manager handles the itemized access control requests. Access control request is itemized for each model as following: for DTE, domain ID, access object and requested permission set (RqsPermSet); for RBAC, role name, access object and RqsPermSet; for MLS, subject, object and RqsPermSet.

Each model has its own decision routing to authorize the request. But they share the same logic. Let's define PolicyPermSet as the permission set defined in the policy. First, the DTE handling routing will pull out the PolicyPermSet from the policy data matrices for above described items presented in the request. For example, to handle an access control request of DTE, routing queryDomainAccessObj of DTEPolicyManager will extract the PolicyPermSet for the domain student_d and

access object */home/zhitaoq/Document/test.log*. Then, RqsPermSet will be compared with PolicyPermSet to see it is satisfied or not for the authorization through: RqsPermSet.issubset(PolicyPermSet).

3.4 TCP - Process Communication

There are several methods to communicate between the different processes or components, but for this project, TCP socket was chosen.The reasons other methods were not selected are based on two factors: language and flexibility.

First, the Library code was written in C language which is different from the language used for the Policy Engine, which is Python. After some trials, the other methods such as message queue and pipe were found not appropriate for the cross-language communication in this project. Message queue was used initially, which was implemented in a third party module sysv_ipc adopted by the Python library to communicate with C language function directly, but sysv_ipc is not well tested and still has bugs. Sometimes the messages sent from a C process to a Python process were malformed. In contrast, the TCP Sockets are well-implemented and tested for this purpose as a relatively high level communication method between processes.

Second, while integrating the aforementioned different components into a whole exploration system, it is important that the system can be low coupling and high cohesion. TCP Socket communication can minimize the coupling between components [10]. Compared to TCP Sockets, the pipe mechanism reduces the flexibility. TCP Socket interface is especially useful in this project because the Programming Library must run in two modes with or without the visualization component. This is an example of what Robert Frost mentioned "Good fences make good neighbors." This chapter mainly focuses on introducing the implementation between the Programming Library and the Policy Engine.

In the socket communication, we use the Client and Server model. Engine Agent acts as a TCP Server to accept messages from the Programming library, which plays the role of a TCP Client.

3.4.1 TCP Server and Handler in the Python Engine

This subsection describes the main process to establish the TCP server through the Python Library's **socket** module.

First, the module tcpserver was designed. This exposes the object acv_server and the method agent_tcphandler. acv_server is a wrapper function for Python socket module, which initializes the host name, port and debugging setting. It contains a

start method for the caller to create the socket, and then bind and listen. Incoming connection requests were bound with the agent_tcphandler, which accepts the arguments of the acv_agent, tcp_connection and sock_address. In method agent_tcphandle, acv_query method of acv_agent passes the access request message to the Policy Manager. Afterward, the server responds to the client through the sendall method of the TCP connection.

If visualization is enabled, these request messages from the Programming Library will be passed to the visualization tool to display graphs for the students to monitor the execution process.

3.4.2 TCP Client in the Programming Library

In the Programming Library, the Access Control Channel function is responsible for delivering a Access Control Request message to the decision engine. Coding the TCP Client in C language to communicate with Python TCP Server is problematic and introduced a lot difficulty in two way communications, as well as in the debugging.

Later we changed the problem and fixed it by introducing the embed Python Interpreter in C language. TCP Client is then also coded in Python language. Detailed implementation as below.

1. Build a separate Python module acv_query.py to implement the actual socket client, which connects to the server and receives response.

2. extend and embed the Python Interpreter in the Access Control Channel function.

   (a) Initialize the Python Interpreter through Py_Initialize(), which creates fundamental python modules and maintains a related table for the resources, as well as calculating the module search path from the environment variables.

   (b) Import module acv_query, and save the result to a callable PyObject: pQueryFunc which is a pointer to represent the actual Python object.

   (c) Call Python/C API PyObject_CallObject to pass the ACV Request message to pQueryFunc [11].

Listing 3.3: Embed Python Interpreter

```
pQueryModule = PyImport_Import(PyString_FromString((char*)"acv_query.py"));
pDict = PyModule_GetDict(pQueryModule);
pQueryFunc = PyDict_GetItemString(pDict, (char*)"acv_query");
```

There are two notes about the usage of this Python/C library API.

1. Include header file Python.h before any other standard header files, because it contains some pre-processor definitions which might bring some impact to others [9].

2. Set PYTHONPATH TO working directory setenv("PYTHONPATH",".",1);

3.5 Visualization Interface

To launch the visualization tools in run time and allow more graphic monitoring features to be created in the future, we have defined the interface in the Programming Library and other related components.

First, the visualization can be configured through acv_init argument. The AC Shell also preserves the -V flag, which will pass the visualization request through acv_init all the way to the Engine Agent.

Second, the Engine Agent will locate the visualization tool path through the system path setting and launch the tool based on the model type for an access control request sent from the Programming Library. After that, the refresh message will be sent for each further access control request, which will be also buffered to allow replaying

function in the tools.

The process communication between the Visualization tools and Policy Engine uses TCP socket. The message handler acvisual_tcp_server and acvisua_tcp_handler are similar to the versions of TCP server and handler described in the previous section.

Chapter 4

Test

4.1 Introduction

In this chapter, first, we will introduce our test tool AC Shell, and then illustrate the correctness and robustness test of the Programming Library and the access control decision. At last, visualization interface test is performed to make sure correct message is passed to the visualization tool through TCP sockets.

4.2 Test Tool

Access Control Shell (AC Shell) was designed as a test platform for the Programming Library as well as a command line tool to practice the policy specified system programming. Basically, AC Shell is a command-line interpreter based on the open source Google Mini Shell [12], which reads user's input as a command and executes it. In this report, we reshaped the mini shell to an interactive wrapper API test tool by mapping a wrapper API to a shell built-in command. It also plays the role of program execution container, through which students write their program without much concern of the Programming Library initialization.

AC Shell collects necessary information for the Programming Library by establishing an environment with default parameter values or acquiring the specified values based on the user command line input. It will save these parameters as access control context. When the student program is launched through the container, it will start the Engine Agent and pass the AC Context to the student program. Hence, the students can focus on the system programming. When AC Shell exits, it will terminate the Engine Agent.

As an interactive command interface, it can test the wrapper APIs flexibly and conveniently. Combination test of wrapper APIs can be performed by observing the result

under different policy setting. Model specific parameters like Domain Name, Role Name or User Name can be changed as needed through running the Set command. The changed values will be reflected in the access control request immediately without the need to re-compile the program.

From another point of view, the Access Control Shell provides more hands-on experience in the system programming level for the students to explore the selected policy.

Listing 4.1: AC Shell Command Line options

```
AC Shell command line instruction:
-E for execution of program,  for example,  Prog1.exe
-M for model type ,           MLS ,DTE, RBAC
-D for initial domain ID ,    student_d is provided as default value
-U for initial user name,     Login name will be used as defaulted userName
-R for initial role name ,    stduent is the default value
-P for policy path           Default is "policy" with the model type,
                              such as, policy.rbac
```

One of model type or policy path must be specified. If model type is specified without specifying policy, default policy will be used, for example, "policy.mls", "policy.dte" or "policy.rbc".

For example, ./acshell -E Prog1.exe -M MLS

./acshell -E Prog1.exe -P policy/policy.mls

Test programs need to include below three parts to launch through AC Shell:

A) #include "acv_wrapper_api.h"

B) acv_env_init(argc,argv)

Notes: pass AC Context or environment to student program

C) acv_end();

Note: To execute programs without AC Shell, acv_init() must be called to setup all the parameters, for example,

$RBAC : acv\_init(specPath, initRoleName, \text{“}zhitaoq\text{”}, \text{“} - NV\text{”}, NULL)$;

4.3 Test Strategy

Generally the test consists two parts: correctness test of access control decision and the system robustness test.

The correctness test of the access control logic for each of the three security models as described in Section was performed separately under different set of policies. The test policies cover different access control scenarios to verify the correct decision making. The execution of the wrapper API was verified through observing the result code and output message, which was compared against the expected code and output.

Failure code (-1) and related outputs were defined into indicate the corresponding reject reasons. Test results were also observed to make sure that the underlying discretionary access control of the system itself is not violated.

For the quality assurance of the software, the system robustness test verifies that the Programming Library and exploration system function correctly in the presence of invalid inputs or stressful circumstances[3].

Based on different approaches of using AC Shell, correctness tests were divided to two parts: test through the interactive command line interface of AC Shell, and test through container by executing student level test programs.

The Programming Library API unit tests are mainly performed in the first part of Correctness Test. Integration tests are mainly performed in the second part.

The system robustness test will be described in a separate section.

4.4 Correctness Test Through Interactive Command Mode

Before the API test, the selected policy needs to be imported to the ACV Shell in the command line or through acv_init, which is described in 3.2.

Below is a sample test process which tests wrapper API acv_rename for RBAC model. The base policy is policy.rbac, as shown in Listing 4.1.

After running ./acv_shell, acv_init was executed with the arguments below, where user zhitaoq has a role of graduate. *NV* stands for No Visualization.

[ACV Shell] acv_init PolicyPath graduate,zhitaoq,NV

First the API was tested with no permission configured.

The user zhitaoq inherits the permissions from the role graduate. However, graduate only has access to /Users/zhitaoq/Wrap/trunk/src/Archive with permission mode r,w,x. Renaming file parse.py under /Users/zhitaoq/Wrap/trunk/src is not permitted by this policy. The arguments from the command line were passed to the wrapper API to execute. The ACV Request is denied by the policy engine. The final result is printed as text log in the shell.

$[ACVShell]./acv\_rename\ /Wrap/trunk/src/parse.py\ /Wrap/trunk/src/parse.sh$

Second added permission to src in the policy.rbac as below, and rerun the command in ACV Shell. The file was renamed successfully.

object: graduate r,w,x -r /Users/zhitaoq/Wrap/trunk/src/.

Benefit from the interactive mode, this part of test can adjust the model specific parameters to improve test productivity. Such as calling set_RoleName Admin,

set_username to cathy, who is configured different access rights to the object, can perform a testcase without changing rule for Role graduate.

Based on the test cases designed, two or more APIs can also be combined to test in ACV Shell interactively, such as after the command acv_rename is executed successfully, acv_execvp can be tested further. This can make sure complex scenario can be covered in the test.

Listing 4.2: Sample Base Test Policy

```
filename: policy.rbac


inheritance: dev > qc

inheritance: sales > qc , cust

inheritance: pres > sales , dev

inheritance: graduate > sales , dev


user: qc quinn

user: dev dave , dot ,zhitaoq

user: admin charles , dave , cathy

user: pres patty

user: sales sam

user: graduate zhitaoq


object: cust r,w,x /path/to/db

object: qc x /path/to/tests

object: dev r,w /path/to/tests

object: pres r,w /path/to/evidence

object: sales r,w -r /path/to/files

object: sales r,w,x /path/to/db

object: graduate r,w,x -r  /Users/zhitaoq/Wrap/trunk/src/Archive
```

Listing 4.3: Test log for the denied case

```
HomePath is : /Users/zhitaoq/Wrap/trunk/src

SpecPath is : /Users/zhitaoq/Wrap/trunk/src/policy/policy.rbac

setACVArgs start.

setACVArgs input :/Users/zhitaoq/Wrap/trunk/src/policy/policy.rbac graduate zhitaoq↩
     -NV!

setACVArgs done!


To launch ACV Agent.SPECPATH IS : /Users/zhitaoq/Wrap/trunk/src/policy/policy.rbac

Go to Loop for receiving message.

DEBUG:TCP Server:listening

Acv rename request.

DEBUG:server:Got connection

$acv_query$:zhitaoq,graduate,/Users/zhitaoq/Wrap/trunk/src,wx

Python Engine Agent receive msg.

zhitaoq

graduate

/Users/zhitaoq/Wrap/trunk/src

set(['x', 'w'])

requested permset:

set(['x', 'w'])

QueryManager: the role graduate has no permission.

deny:Role graduate has user zhitaoq assigned to it but has no access to the object ↩
    /Users/zhitaoq/Wrap/trunk/src.

Result of call: -1

Renaming is not allowed.
```

4.5 Correctness Test Through Container Mode

Test programs simulating student programs are written to test certain complex scenarios via the combinations of a subset of wrapper APIs.

A sample program is introduced here to describe the test process. Basically, it creates a file and writes some strings to the file. Next, the file is created under folder /Users/zhitaoq/Wrap/trunk/src/. At last, it opens the file to read and prints out the result.

Through this test mode, the program can be used to test policy of different security models. For example, this program tests a set of wrapper APIs for file operation. First, we run the program with RBAC default specification policy.rbac: ./acshell -E Prog1 -m RBAC

Listing 4.4: Partial RBAC policy

```
inheritance: student > sales , dev

User: student zhitaoq

object: student r,w,x  /Users/zhitaoq/Wrap/trunk/src/
```

With all the required permissions configured in the policy, this process is executed successfully with the expected result. After executing the program and deleting the read permission for the role graduate as below , the request of *acv_open* with permission mode O_RDONLY was denied.

$object : graduatew, x - r/Users/zhitaoq/Wrap/trunk/src/$

Listing 4.5: Sample Execution Log for RBAC

```
INFO:RbcPolicyManager:zhitaoq

INFO:RbcPolicyManager:graduate

INFO:RbcPolicyManager:/Users/zhitaoq/Wrap/trunk/src/

INFO:RbcPolicyManager:requested permset:

INFO:RbcPolicyManager:set(['x', 'w'])

INFO:RbcPolicyManager:configured permset:

INFO:RbcPolicyManager:set(['r'])

Deny:Role graduate

acv_create is not allowed for : ACV_TEST_PROG1.log !
```

And then, we switched to test the DTE with default specification policy.dte: ./acshell -E Prog1 -M DTE -D student_d

Listing 4.6: Partial DTE policy

```
domain student_d = (/usr/bin/{sh,csh,ksh}),
    (drx->sysbin_t,gradExec_t),
    (cdrwx->generic_t),
    (dr->readable_t,gradReadable_t,dte_t),
    (drw->studWritable_t),
    (exec->guest_d);
assign -r studWritable_t /Users/zhitaoq/Wrap/trunk/src;
```

Listing 4.7: Sample Execution Log for DTE

```
INFO:DTEPolicyManager:student_d
INFO:DTEPolicyManager:/Users/zhitaoq/Wrap/trunk/src
INFO:DTEPolicyManager:Requested:  set(['d', 'w'])
INFO:DTEPolicyManager:Configured: set(['r', 'd', 'w'])
pass:good
performed acv_create.
```

At last, we execute Prog1 under MLS policy as below with default specification policy.mls. User name is zhitaoq, with security level of secret:michigan, which has write permission to higher level object top-secret:michigan.

./acshell -E Prog1 -M MLS

Listing 4.8: Partial MLS policy

```
users secret:michigan  zhitao
assign top-secret:michigan    /Users/zhitaoq/Wrap/trunk/src
```

Listing 4.9: Sample Execution Log for MLS

```
INFO:MLSPolicyManager:
INFO:MLSPolicyManager:Subject: zhitaoq
INFO:MLSPolicyManager:Subject: secret
INFO:MLSPolicyManager:Subject: set(['michigan'])
INFO:MLSPolicyManager:Object: /Users/zhitaoq/Wrap/trunk/src
INFO:MLSPolicyManager:Object: top-secret
INFO:MLSPolicyManager:Object: set(['michigan'])
pass:good
performed acv_create..
```

The decision result can be totally different based on the rules defined in the corresponding policy.

4.6 System Robustness Test

4.6.1 Invalid Case Handling

User enters the initial setting for the Programming Library through acv_init or acv_env_init for AC Shell. So our test must make sure correct arguments are passed.

First, specification file input test. This test make sure correct policy path is specified before further handling. We verified that incorrect policy path will be detected, and program will terminate gracefully.

Second, that correct model type is selected. The model type must be one of the values "DTE", "RBAC" or "MLS", which is extracted from the policy name or through AC Shell command line input. Otherwise, the call of acv_init or acshell will indicate the failure.

Third, for other parameters, such as RoleName, UserName for RBAC, DomainID for DTE, UserName for MLS if no values are specified, default values involved in the access control process are verified.

Furthermore, tests are performed to cover the case that students write their programs without calling acv_init or acv_env_init. For example, one program calls acv_rename

directly. This test verifies that the API call fails with appropriate message which indicates AC Library not initialized.

4.6.2 Stressful Condition Test

The purpose for this test is to make sure the system still functions correctly and the right policy will be enforced under the circumstances that multiple processes could be launched at the same time.

First, tests are performed to make sure that only one policy can be selected in the system at the same time. As illustrated in Chapter 3, the library initialization function acv_init or acv_env_init (launched by acshell) must be called at the beginning of each program. The design of the function makes sure that only one policy agent is allowed to exist in the system, which enforces one policy. So, when one process already executes the policy agent, other programs can detect this and then terminate accordingly.

Multiple-process tests are performed in two cases: with the same type of policy, and with different types of policy. The same test steps of below list are followed.

Listing 4.10: Test Steps for Stress Test

```
1.Run selected programs simultaneously through shell script;
2.Verify that only one Engine Agent process starts, access control requests are↩
 handled correctly under the appropriate policy;
3.Verify that the processes spawned by the executing program terminate      ↩
      gracefully, including the Engine Agent process, the TCP Server process;
4.Verify that sockets and files are closed;
```

4.7 Visualization Interface Test

This section introduces the test performed for the Visualization Interface.

To enable visualization mode, the value of $-V$ must be entered in the ACV Shell command line or set int the acv_init function call of the Test Program. Then we verify that acv_agent function detects the visualization request and launch the correct visualization tool related to the specified model. If the visualization tool aforementioned in Chapter 2 is not installed in the system yet, a message will print out for this. Otherwise, verify that the correct policy is selected and appropriate graphs are displayed.

Following the above example of testing acv_rename for RBAC in previous section. Verify that when the permission request is rejected, the following graph is displayed

to explain the rejected request. The related rules and graph nodes are highlighted. It shows clearly that user zhitaoq which is role of graduate with limited privilege configured to access /Users/zhitaoq/Wrap/trunk/src/Archive folder. Detailed permission sets are printed in the top of right window. Figure 4.1 depicts this result visually.

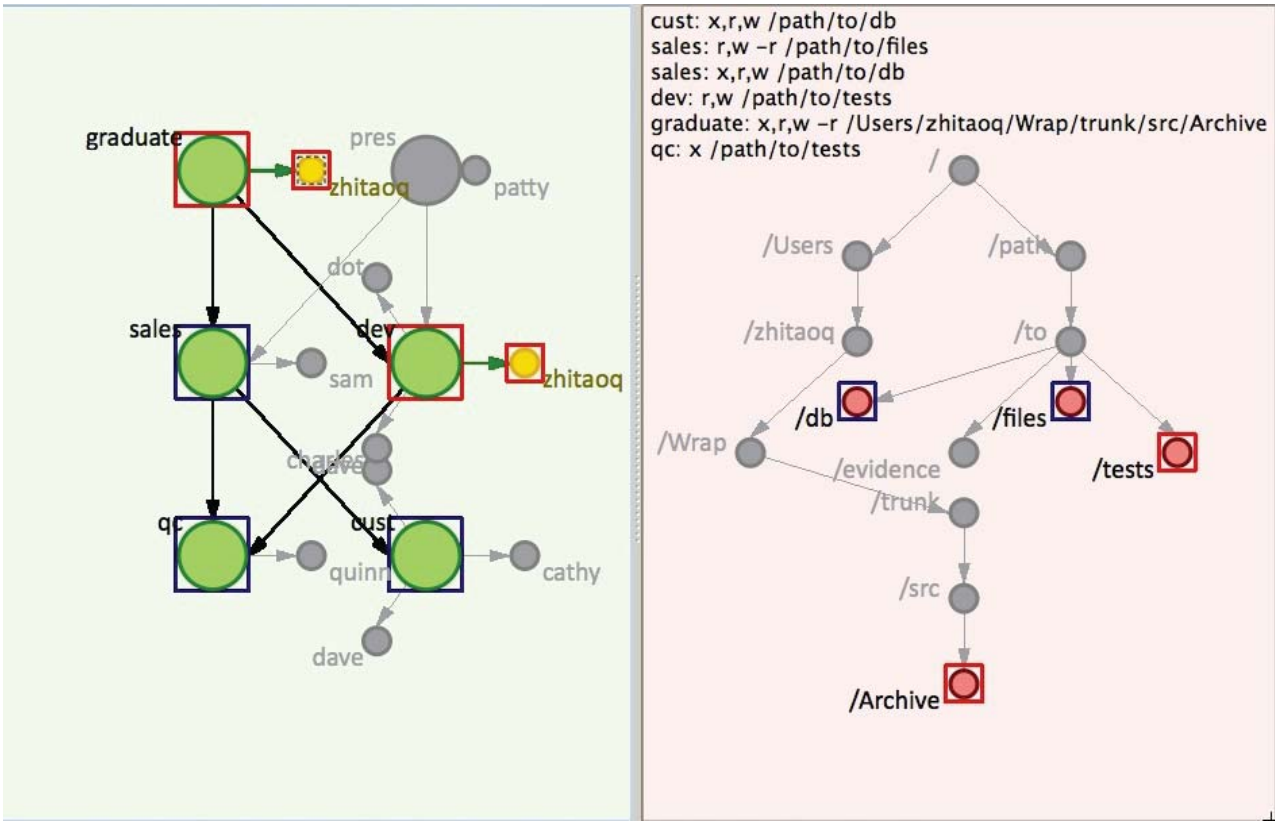


Figure 4.1: Visualization Interface Test

Chapter 5

Conclusion and Future Work

In this report, we investigated three security models of Mandatory Access Control technologies and visualization tools. We also implemented the Access Control Policy Programming Library and the Exploration Shell. A selected collection of System Call Wrapper APIs were implemented for the POSIX System Call APIs. Policy parsers were extracted from the existing tools and integrated to the Policy Engine, and the Access Control Policy Manager was also implemented to provide decision for the Access Control Request.

This Programming Library and exploration system shows the students related policy credential changes in the execution of a running program in system level. The information level for more or less logs can be adjusted conveniently. It also defined

the TCP socket interface to communicate with visualization tools, through which the abstract access control process can be depicted visually. This achieved the expected effect as a pedagogical tool to provide the availability of hands-on experience in the security formal education.

However, currently, the Programming Library just implemented a limited set of system call wrapper APIs, which focus on the file system operations and binary execution. In the future, the library can be extended to cover more system call functions, such as memory or process control, so that students can have more advanced security programming experience.

In addition, this report describes a command line tool called Access Control Shell, but it is mainly used as a wrapper API test platform and binary execution container. It can be enhanced to contain more security functions or features similar to SE-Linux to provide more hands-on experience for the students, such as adding a sesearch feature of SE-Linux, or even build an access control Linux container (LXC) [11].

Chapter 6

References

[1] Badger L., D.F. Sterne, D.L. Sherman, K.M. Walker, S.A. Haghighat. A Domain and Type Enforcement UNIX Prototype. Proc. Fifth USENIX UNIX Security Symposium, 1995.

[2] Carr S. and J. Mayo. Teaching Access Control With Domain Type Enforcement. Journal of Computing Sciences in Colleges, 27(1) pp. 74-80, 2011.

[3] Yifei Li, Steve Carr, Jean Mayo, Ching-Kuang Shene, Chaoli Wang "DTEvisual: A Visualization System for Teaching Access Control using Domain Type Enforcement." Journal of Computing Sciences in Colleges, 28 pp. 125-132, 2012.

[4] Man Wang,Steve Carr, Jean Mayo, Ching-Kuang Shene, Chaoli Wang "MLSvisual: A Visualization Tool for Teaching Access Control Using Multi-Level Security" Proceedings of

the 2014 conference on Innovation and technology in computer science education, 2014.

[5] Wikipedia Discretionary_access_control, BellLaPadula_model, Robustness_testing, Abstract Factory at http://en.wikipedia.org/wiki/, 2015.

[6] David Elliott Bell "Looking Back at the Bell-La Padula Model" Proceedings of the 21st Annual Computer Security Applications Conference, 2005

[7] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn,and R. Chandramouli. "Proposed nist standard for role-based access control". ACM Trans. Inf. Syst. Secur.,4(3):224274, Aug. 2001.

[8] GNU LIBC POSIX Manual, available at http://www.gnu.org/software/libc/manual/html_node/, 2015.

[9] Python document, available at https://docs.python.org/, 2015

[10] Andrew Hunt, David Thomas "The Pragmatic Programmer: From Journeyman to Master" Addison-Wesley Professional, 1999.

[11] SELinux Wiki, available at http://wiki.gentoo.org/wiki/SELinux/Tutorials/Controlling_file_contexts_yourself, 2015

[12] Mini-shell, Google Project Hosting, GNU GPL v3, https://code.google.com/p/mini-shell/, 2011

[13] Matt Bishop, "Computer Security Art and Science", Addison-Wesley, QA 76.9.A25 B56, 2002.

[14] Linux Programmer's Manual, Michael Kerrisk, http://man7.org/linux/man-pages/man2, 2015

Appendix A

Specification

A.1 DTE SPECIFICATION SYNTAX

```
DTE specification normally contains four sections [2,3].


Section 1   Type definition
Declares one or more object type names, which are then available to other parts of ↩
    a DTE specification. The statement starts with keyword Type and then lists all ↩
    the defined object types with suffix\_t. No ordering is required.


Syntax format:    Type  objecttype1_t, objecttype2_t , objecttype3_t,
For example:
Type  dte_t,readable_t,generic_t,writable_t,sysbin_t,projectA_t


Section 2 Domain definition
```

```
Domain definition is expressed as a list of tuples.  Domain definition statement ↩
    start with keyword domain and follows with domain ID and execution environment ↩
    .

Defines a restricted execution environment composed of three parts[1]:
    a.  Entry Program,  executed by a process in order to enter the domain, for ↩
    example, /usr/bin/login for domain login_d,
    b.  Define Access permissions to the object types, e.g. drwx->writable_t . This↩
     contains multiple lines.
    c.  Permissions to access programs in other domains e.g. exec->user_d for ↩
    domain login_d. Or allow Auoto transition to another domain, like (auto->↩
    login_d) for daemon_d.

domain daemon_d = (  /sbin/init),
        (dr->generic_t,readable_t,dte_t),
        (cdrw->writable_t),
        (drx->sysbin_t),
        (auto->login_d);
domain login_d = (/usr/bin/login),
        (dr->readable_t,generic_t,dte_t),
        (cdrw->writable_t),
        (exec->admin_d, user_d, graduate_d,faculty_d);

domain user_d = (/usr/bin/{sh,csh,ksh}),
        (drx->sysbin_t),
        (cdrwx->generic_t),
        (drw->writable_t);
        (dr->readable_t,dte_t),


Section 3 Initial domain
This section contains one statement to declare the initial DTE domain.
```

```
      initial_domain = daemon_d;

Section 4 Type Assignments

The assignments in this section will associate a type with one or more files.
Syntax format:    assign [-r] typeId_t directory1 directory2    ;

A statement may be recursive, in which case it applies to all files in the ↩
    directory tree rooted at the named directory.  Recursive assignment of a file ↩
    with prefix P overrides an assignment for a file with a prefix shorter than P. ↩
     For instance a recursive assign statement for /etc overrides a recursive ↩
    assignment for /.

Below are more examples:
assign -r generic_t /;
assign -r projectA_t    /proj1;
assign -r projectB_t    /proj2;
assign -r -s dte_t      /dte;
assign -r writable_t    /dev,/usr/var/test,/tmp/test ;
assign -r readable_t    /etc;
```

A.2 MLS SPECIFICATION

```
 MLS specification normally contains four sections.

1)  Section1: Clearance Statement
Define one or more clearance levels, which are then available to other parts of a ↩
    MLS specification. Clearance Statement starts with keyword clearances and then ↩
    lists all the defined security levels from low to high.

Syntax format:   clearances: level1<level2<

For example, there are three security levels from low to high as: unclassified, ↩
    secret, topsecret.  We define the Clearance Statement as below:

    clearances: unclassified<secret<topsecret

2)  Section2: Category Statement
Define one or more categories, which are then available to other parts of a MLS ↩
    specification. Category Statement starts with keyword categories and then lists↩
     all the categories after it without ordering.
Syntax format:  "categories: category1, category2...
    categories:weapon, nuclear, grocery

3)  Section3: Directory Security Assignment Statement
Assign security levels to directories in the file system. The statement starts with↩
     assign, and then follows with the clearance level and categories, which are ↩
    assigned to the directory occurring at the end of the statement. One directory ↩
    can be assigned to multiple categories.
```

```
Syntax format:  "assign clearance: category1: category2...  [-r | -s] directory1, ↩
    directory2,

       Here are some examples:
assign unclassified:    -r  /
assign secret:weapon -r  /weapon
assign topsecret:nuclear -r /nuclear

4)  Section4: User Security Assignment Statement
Assign security levels to users. This statement starts with keyword users and ↩
    follows with the clearance level and categories, which are assigned to the user↩
     list occurring at the end of the statement.


Syntax format:    "users clearance: category1:category2...   user1, user2 ...

For examples:
users  topsecret:nuclear:weapon    ping
users  topsecret:nuclear           david
users  unclassified:               micky, lucy
```