**Implementation of Mandatory Access Control System in Linux OS (Shang Da, Sen Shen)**

Shang Da

**Introduction**

Historically, Linux system have used DAC. But a problem is that, a malicious program can acquire root authority and do whatever it wants. However, security policy in MAC is administratively set and fixed. Therefore, it provides very nice access control against such malicious programs. There are many existing security enhancements for Linux OS that implemented MAC. SELinux is a good example.

In our COMS552 project, we implement a Linux kernel module that enable MAC in Linux OS in order to learn more about Linux kernel as well as security.

**System Architecture**

Our system contains the following components:

* Method to intercept and overwrite system calls.
* Security Server which store security policies and also provide enquiry for Security Check.
* Security Check for each system call to decide grant certain operations or not based on security policies.
* Security Manager who is the only user that can access and modify policies in Security Server.

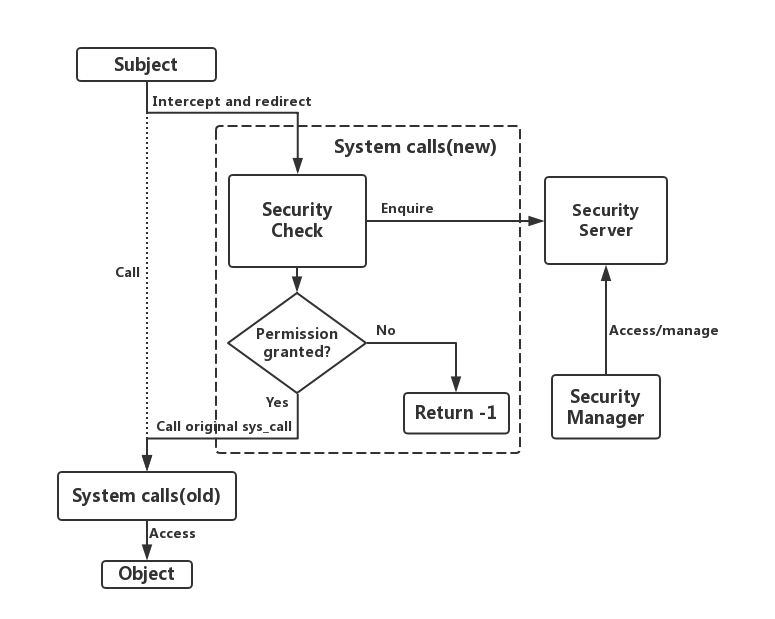


Figure 1. Architecture of our system

As is shown in the figure 1. Architecture of our system is different from SELinux, which used LSM to implement MAC. Every time a subject wants to access an object, the subject will always call the system calls. When our system is loaded, it will intercept certain system calls, replace it with a new system call. In the new system call we'll first do the security check, which enquires Security Server and get the authority, and judge if permission is granted or not. Our new system call will decide whether to call the old system call, which means permission is granted, or otherwise return -1.

**Implementation**

Our system is implemented as a Linux Kernel Loadable Module. The source code is in C-language. Figure 2.1 shows all of our source code. To make our source code modular. We put different type of methods into different files. To avoid redefinition, we created C-header files for declarations and .c files for definition. Compiling and running of our system is also the same as a normal Linux Loadable Module. Figure 2.2 shows how we compile and run and Figure 2.3 shows how we write our makefile for compiling kernel module with multi-file source code.

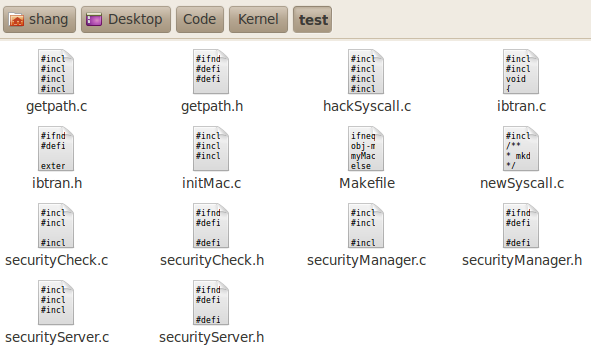
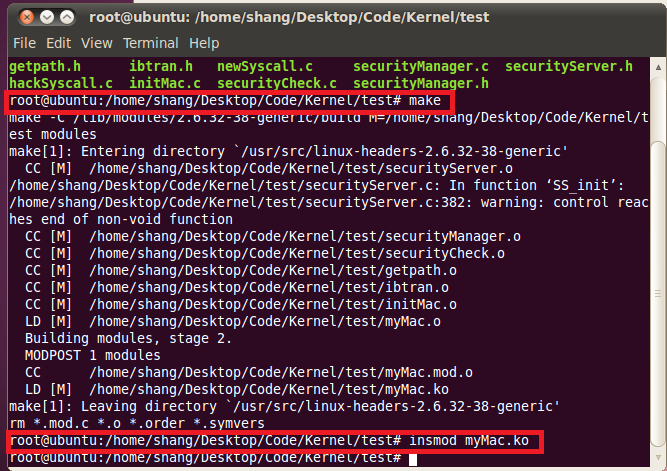
****

Figure 2.1 source code Figure 2.2 Compile and install module

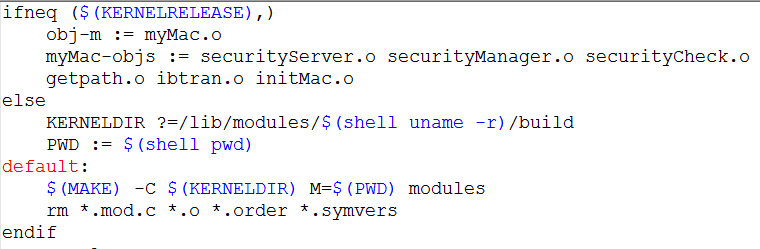


Figure 2.3. makefile

**Intercepting System Calls**

Overwriting system calls can be implemented in two ways:

1. Modify kernel source code and define system call table as an extern variable.
2. Intercept system call table through interrupt 0x80.

Because the first method will enable every one to visit system call table. This can cause some server security issues as some malicious program can access the table and change system calls directly. Therefore, we choose the second method.

Figure 4.1 shows the procedure when any system call is called. All system calls will trigger the same interrupt at address 0x80 of IDT, and goes to interrupt service routing(ISR) for this certain interrupt 0x80. ISR will call sys\_call\_table(system call table), which stores entry for service for each different system call. It will check which service it need to go to based on the system call who trigger the interrupt and in the end, it will go to the specific service.

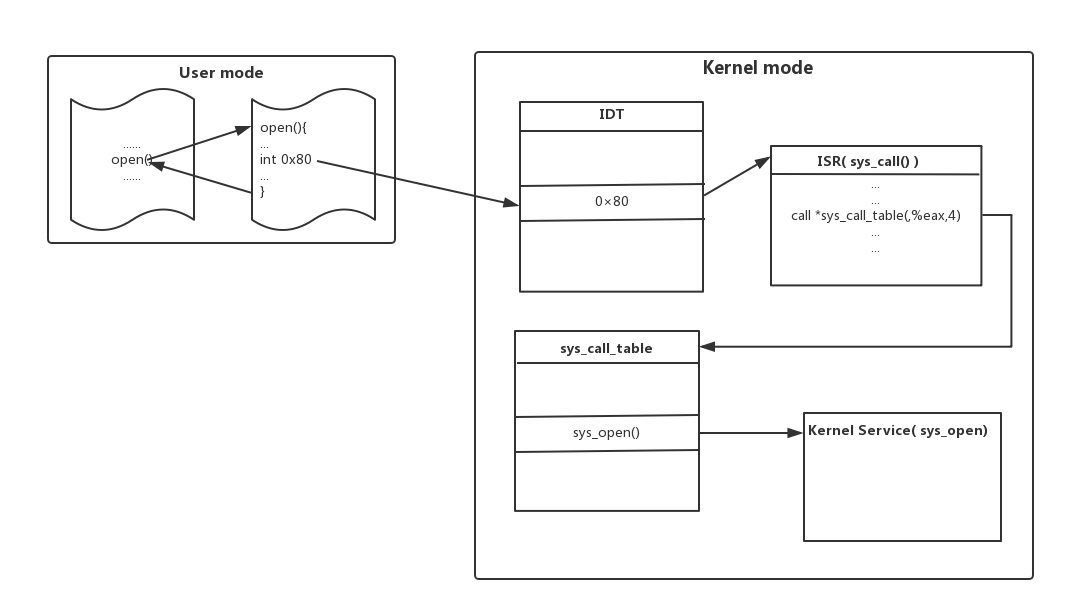


Figure 4.1 procedure of system call

To intercept system call table. We first get address of Interrupt Definition Table(IDT) by an assembly command "sidt". Then we go to address 0x80 of IDT, which stores the address of the ISR we previously talked about. In the memory ISR is stored as binary code in some certain memory buffer. Therefore, we can search line by line for the unique command "call", which is used to call the sys\_call\_table. Then we have the address of sys\_call\_table.

Plus, because sys\_call\_table is write protected. In order to be able to write to sys\_call\_table, we also need to disable write protect in CR0(set 16th digit of CR0 to 0) and set it back after we 've done replacing system calls.

Now, since that we have address of sys\_call\_table and that we 've disabled write protection. We can start redirecting system calls. Because sys\_call\_table is actually an array of address(function pointers) of different services for each system call. We can replace system calls by changing particular address to point to our own functions.

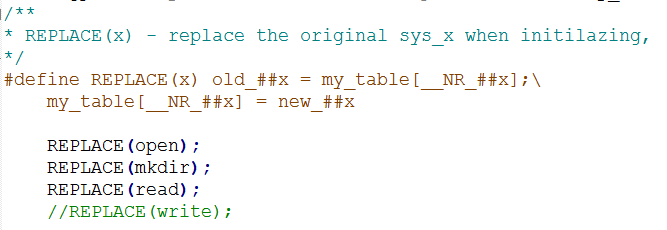


Figure 4.2. redirecting system calls

Figure 4.2 shows an example of how we redirect entry in sys\_call\_table to our new system call. Note that to make it more convenient, we used #define to make compiler translate REPLACE(x) as the sentence after it. my\_table is a pointer that points at sys\_call\_table. \_\_NR\_##x is translated as \_\_NR\_open/\_\_NR\_mkdir/... by the compiler. They are defined as numbers in file "unistd.h". These numbers actually represent relative address of certain system calls in sys\_call\_table. Therefore, we've redirected certain system calls. Similarly, new\_##x/ old\_##x is translated as new\_open/old\_open etc. by the compiler. They are defined in file "newsyscall.c". new\_##x will be our new system calls and old\_##x will store address of old system calls. After security Check, we'll decide whether to call old\_##x or not.

Because we implement our system by firstly intercept and replacing system calls. Even root user, as long as he wants to use certain system calls, which is unavoidable, cannot bypass our system.

**Security Server**

Our Security Server has the following features:

* Maintains 3 tables for subject types, object types, and authority.
* Provide methods for enquiring three tables
* ONLY allow access from Security Manager.

# Implement

Our Security Server is implemented by procfs(proc filesystem) it is a virtual file system. In user mode each proc entry(file) looks like a file. However, each of them is actually stored in the memory as a structure named proc\_dir\_entry. In the kernel mode we can access them and get the data we stored directly. Figure 5 showed an example.

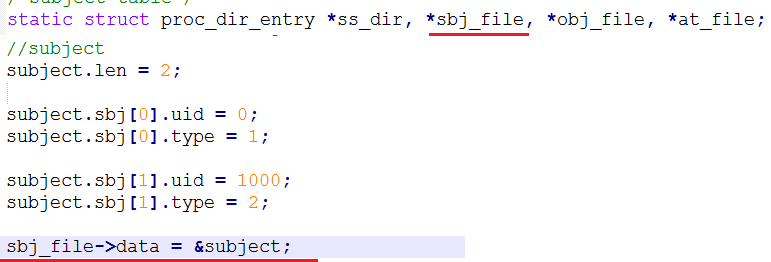


Figure 5. an example of proc entry

# Tables

There are three tables in our Security Server. They respectively store information about:

1. Subject table: Mapping from subject to subject type
2. Object table: Mapping from object to object type
3. Authority table: Authority of each subject type to each object type

# Enquiry

Because we can have access to each table directly in kernel space. Enquiry for each table are implemented by some logic that traverse all the table and search for certain subject/object/pair of types and in the end return authority.

Plus, in our Authority table, we store authorities(e.g. mkdir/opening/reading etc.) by integers. When enquiring, the SS will transfer the integer to binary and return it as a 13-digit string, each digit will represent a certain type of authority. For example, 6143 = 1011111111111(binary). The second digit is 0. Therefore, opening is not allowed. The method for transforming integer to 13-digt string is written by our self in files "ibtran.h" and "ibtran.c".

# Limit access

To allow access from ONLY the Security Manager. We set a user(Security Manager) as type 0 in Subject table as well as proc files as type 0 in Object table. Then we set 0 to authority of subject types except 0 to object type 0. As a result, tables in our security servers will be blocked by our MAC system when some other users(even root) wants to visit them.

**Security Check**

Security Check is basically some logic that call enquiry functions implemented in SS. The enquire functions will return a 13-digit string(pointer). And Security Check for each new system call will check certain digit based on the type of the new system call.

In some situation, we need to transfer relative path or file descriptor into absolute path. The former is by first get current directory and append the relative path. The latter is by visiting file descriptor table and get file structure and get absolute path. The method for getting absolute path is declared and defined in files "getpath.h" and "getpath.c".

**Security Manager**

The reason why Security Manager can change security policies in the user mode is that our SS is implement by procfs. And procfs provide API for rewriting call back function of reading and writing proc entries. Figure 6 showed an example of our new call back function for reading/writing one of the proc entries, which is the authority table. Note that at\_file is a proc entry and read\_proc/write\_proc are APIs provided by procfs. Declaration and definition of our new call back functions are in files "securityServer.h" and "securityServer.c".

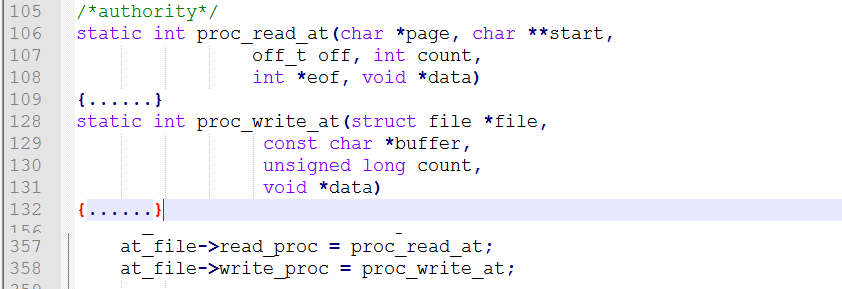


Figure 6. rewriting call back function

We define new call back functions for each of three proc entries such that, every time we read each proc entry, it present data in the ways that we want. And every time we write to the authority table, it only accepts format as follow:

"allow/deny sbj\_type obj\_type authority1| authority2|......"

And after finishing input to the proc entry. The call back function will analyze the input, enquire and set value of data in SS accordingly. Enquiry and setting values used APIs written in SS.

**Drawbacks and solutions**

Our system can provide a MAC in Linux. however, it has some drawbacks:

# System calls are vulnerable. Sometimes if we replace some certain system calls system will crash. There are some ways to solve the problem but it requires high level understanding of Linux kernel.

Solution: Using LSM can be a way to avoid this problem because LSM is deeper in the kernel and does not have to deal with communication between user mode and kernel mode.

# Mapping from subject to type and object to type is stored in tables. Problem is that there are too many files in a computer and thus the object table can be very large.

Solution: Instead of using table for the mapping, we modify certain structures in Linux kernel source code and add a flag that represent object type.

**Conclusion**

The project is very difficult and time-consuming. But I learned a lot about Linux kernel module, system calls, virtual file system and security during the project. After finishing the project, now I consider myself a kernel-beginner rather than a kernel-newbie. Next time if I meet another problem related to Linux kernel, I 'll be very confident that I can solve it more elegantly.