

Linux Role-Based Access Control (RBAC) Lab

Copyright © 2006 Wenliang Du, Syracuse University.

The development of this document is funded by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at <http://www.gnu.org/licenses/fdl.html>.

1 Lab Description

The learning objective of this lab is three-fold: (1) The lab provides students with an opportunity to design an access control mechanism, Role-Based Access Control (RBAC), for a real operating system. In this design, students need to integrate two access control principles, capability and RBAC, to enhance system security. (2) Not only should the students implement RBAC correctly, they need to apply their critical thinking skills and the principles of security to analyze their design and to ensure that the new system, after addition of RBAC, is secure. (3) The new access control mechanism needs to be implemented at the kernel level, but students do not need to modify the kernel, instead, they will only modify the Linux Security Module (LSM), which provides a generic framework for the implementation of different access control mechanisms in Linux without the need for modifying the kernel. From this lab, students can get familiar with the concept behind LSM. They should be able to implement a variety of other access control mechanisms using the LSM concept.

The RBAC access control required for this lab is based on the RBAC standard proposed by NIST [1]. We have made certain simplifications to make this lab suitable for a course project. This lab is quite comprehensive, students should expect to spend 4 to 6 weeks on this lab. Students should have a reasonable background in operating systems, because kernel programming and debugging are required.

The RBAC access control will be implemented as a Linux Security Module, which was designed for developers to extend the Linux's original access control mechanism (i.e., Discretionary Access Control and Access Control List). LSM is a loadable kernel module, which can be directly loaded into the kernel. LSM has been used by several security systems in Linux, including SELinux and AppArmor. Although RBAC is already implemented in SELinux, which is a part of Fedora Linux, you need to implement your own (much simplified one). There will not be conflicts, because we will temporarily "suspend" SELinux in this lab.

2 Lab Tasks

RBAC (Role-Based Access Control), as introduced in 1992 by Ferraiolo and Kuhn, has become the predominant model for advanced access control because it reduces the complexity and cost of security administration in large applications. Most information technology vendors have incorporated RBAC into their product line, and the technology is finding applications in areas ranging from health care to defense, in addition to the mainstream commerce systems for which it was designed. RBAC has also been implemented in SELinux and Trusted Solaris.

In this lab, we will implement RBAC for Fedora Linux. Fedora Linux version 9 has implemented more than 30 capabilities, which are used in access control. These capabilities can be considered as special privileges, and they usually allow a user with the capabilities to carry out a privileged operation. For example, a user can read any file if she/he has the proper capability. Without capabilities, users who need

privileged have to be the root, which has more power than what she/he needs. With capabilities, if a user needs special privileges, we can assign the corresponding capabilities to the user, instead of turning the user into root.

In a system with many different types of privileged users, it is difficult to manage the relationship between capabilities and users. The management problem is aggravated in a dynamic system, where users' required privileges can change quite frequently. For example, a user can have a manager's privileges in her manager position; however, from time to time, she has to conduct non-manager tasks, which do not need the manager's privileges. She must drop her manager's privileges to conduct those tasks, but it might be difficult for her to know which privileges to drop. Role-Base Access Control solves this problem nicely.

With RBAC, we never assign capabilities directly to users; instead, we use RBAC to manage what capabilities a user get. RBAC introduces the role concept; capabilities are assigned to roles, and roles are assigned to users. Since the number of roles is much less than the number of possible combinations of capabilities, and roles can be easily associated to people's job functions, managing roles is much easier.

In this lab, we will implement the RBAC access control for `Linux`. The specific RBAC model is based on the NIST RBAC standard [1]. The basics of this model is called *Core RBAC*, which includes five basic data elements called users (`USERS`), roles (`ROLES`), objects (`OBS`), operations (`OPS`), and permissions (`PRMS`). In addition, Core RBAC also include sessions. The concept of users, objects, operations are straightforward, and they are elements in most operating systems. Permissions and roles are what you need to focus on in designing your RBAC system.

2.1 Permissions: Capabilities

Permissions decide whether an action (e.g. reading a file) can be carried out or not. Permissions consists of a tuple (`OPS`, `OBS`), which indicates whether an operation (`OPS`) can be carried out on objects (`OBS`). In this lab, capabilities are permissions, i.e., each permission is a capability.

Capabilities are like tokens: users who have a token can access the corresponding objects associated to this token. In a capability system, when a process is created, it is initialized with a list of capabilities. When the process tries to access an object, the operating system checks the capabilities of the process, and decides whether to grant the access or not. Recent `Linux` kernels have already implemented about 30 capabilities. In this lab, we will use these capabilities as the permissions.

2.2 Sessions and Processes

Sessions is a mapping between a user and an activated subset of roles that are assigned to the user. Each session is associated with a single user and each user is associated with one or more sessions.

In this lab, we use *login session* as RBAC session. Namely, when a user logs into a system (either locally or remotely), a new session is created. When a user's session is created, it is initialized with the set of roles assigned to this user. A user can run multiple login sessions simultaneously, and thus have multiple RBAC sessions. For example, a user can use `ssh` to remotely log into a computer; each remote login is a session.

Each session starts with a new process (e.g. when a user logs into a system, a shell process will be created normally), so the first process will be initialized with the set of roles assigned to the user. Within a session, many new processes can be created, their role initialization will be based on their parent processes. *You need to decide how a child process's roles are initialized.*

2.3 Roles

A role is basically a set of permissions. In RBAC systems, permissions are assigned to roles, and roles are then assigned to users; permissions are not directly assigned to users. For a practical RBAC system, the

following functionalities are needed:

- **Creation and Maintenance of Roles:** Roles in a system cannot be hard-coded; administrators should be able to add/delete roles. You need to implement a utility to allow system administrators to add/delete roles.
- **Creation and Maintenance of Relations:** The main relations of Core RBAC are (a) user-to-role assignment relationship (UA), and (b) permission-to-role assignment relation (PA). Please be noted that **both UA and PA relations can be modified during the run time**, but the change of UA and PA relations will not affect existing sessions; it only affects new sessions.
 - **Update PA Relationships:** A privileged user should be able to add permissions to or delete permissions from a role. Such a modification should be persistent; namely, the relationships will be retained even after the system is shut down.
 - **Update UA Relationships:** A privileged user should be able to add users to or delete users from a role. Similar to the PA relationships, the modification should be persistent.
 - **Delegating/Revoking Roles** Delegation/Revocation is another way to update UA relationships. A normal user with the capability `CAP_ROLE_Delegate` should be able to delegate his/her own roles to other users, and also be able to revoke the delegated roles. When a role is delegated to a user, a new user-to-role instance will be created; new sessions of the user will be affected by this new UA instance. However, this user-to-role instance is volatile; namely, it will be lost if the system is shut down. The user who delegates his/her roles can later revoke those roles. Once a delegated role is revoked by the user, the effect should be seen immediately; namely, all the involved sessions (current) will lose that delegated role immediately.

2.4 Manipulating Roles

Each new process created in a session will be initialized with a set of permissions based on the session or the parent process. After a process is created, users can dynamically modify the roles within the process (the modification can only be applied to the current process and its descendents). Here is a list of functionalities that you need to implement in your RBAC system:

- **Enable and Disable Roles:** A process can enable and disable any of its roles. Functions related to role enabling and disabling are `EnableRole` and `DisableRole`. The `DisableRole` function does not permanently drop a role, it only makes the role inactive. A disabled role can be enabled later.
- **Dropping Roles:** If a process does not need a role anymore, it should be able to permanently drop the role using `DropRole`. Once a role is dropped from a process, there is no way for the process to regain that role.

2.5 Separation of Duty

Separation of duty relations are used to enforce conflict of interest policies that organizations may employ to prevent users from exceeding a reasonable level of authority for their positions. NIST RBAC standard defines two types of separation of duty relations: *Static Separation of Duty (SSD)* and *Dynamic Separation of Duty (DSD)*. SSD enforces the separation-of-duty constraints on the assignment of users to roles; for example, membership in one role may prevent the user from being a member of one or more other roles, depending on the SSD rules enforced. DSD allows a user to be assigned conflicted roles, but ensures that

the conflicted roles cannot be activated simultaneously. In this lab, your system should support both SSD and DSD rules.

SSD and DSD policies (i.e. rules) are set by the system administrators. You can define your own format for these policies. Moreover, you can decide where to store the policies, how to effectively check these policies, and how to update these policies. We also assume that any update of the policies only affect new sessions and future operations. It is important to identify where SSD and DSD policies should be checked.

- SSD policies need to be checked every time a role assignment occurs. There are two places where a role might be added to a user: one is conducted by the privileged users. To simplify your design, you can delay the enforcement of SSD until a user creates a new session (i.e. login), rather than at the point when the privileged users add the role. Another place where a role is added to a user is via delegation. You need to make sure that any delegation that violates the SSD policies will fail.
- DSD policies need to be checked every time a role become active. There is only one place where a role can become active. That is when the function `EnableRole` is called. Note that the previous statement is true because all roles are in a disabled state initially, including those roles that are delegated from other users.

3 Design and Implementation Issues

In this lab, you need to make a number of design choices. Your choices should be justified, and the justification should be included in your lab report.

IMPORTANT NOTICE: Please backup a valid boot image before you make modifications; you might crash your systems quite often. If you are using *VMware*, you can use its snapshot feature.

3.1 Initialization

When a user logs into a system, a new session will be initialized, and the first process of this session will be initialized with the roles assigned to this user. The first process of a session will be the ancestor of all the processes within the same session; as long as this process is initialized with a role list, all the new processes in the same session can be initialized based on the role list of their parents.

The key challenge facing the initialization is when to use the user's role list to initialize a process (only applies to the first process of a session), and when to use the parent process's role list to initialize a process. In other words, how to decide whether a process should be the first process that marks the beginning of a session?

Two approaches can be used to achieve session initialization: user-level approach and kernel-level approach. In the user-level approach, a user-level program tells kernel when a session should be initialized. For example, you can modify `login.c`, which is called when a user logs in. The disadvantage of this approach is that you have to modify some existing applications, and you have to cover all the entry points, including local login and remote logins, such as `telnet`, `ssh`, and `ftp`. This is not a desirable approach.

A desirable approach is to implement the initialization in the kernel, i.e., let the kernel decides when to initialize a session. One of the system calls that is used by most of the login programs is `setuid()`. Usually, the process that deal with logins runs as `root`; it has to create the first process for a user (say user A) when the user logs in. Initially, this user process is also owned by `root`, so it calls `setuid()` to turn the real user id (as well as the effective user id) to A, before handing over the control to the user (e.g. invoking a shell program). This mechanism is used by most of the login programs, including `login`,

telnet, ssh, and ftp. Therefore, you can use the following rule to decide whether a new session should be started:

Session initialization rule: A new session is started if the real user id of a process is modified. The process should be initialized with the real user id's roles originally assigned by the system administrators.

3.2 How Capabilities Works in Linux

Because the RBAC in this lab is based on capabilities, you need to get familiar with how capabilities works in Linux. In particular, you need to understand the following questions: where the capabilities of a process is stored? how to change the capabilities of a process? how does the operating system use capabilities to conduct access control? If you have done the capability lab (a separate lab that we have designed, see [5]), you should be familiar with Linux's capabilities already. We have also developed a document to help you understand capability [6].

3.3 Compatible with Set-UID Mechanism

To be backward-compatible, your system should still allow the the existing Set-UID mechanism, namely, when a Set-UID program is run, the the program owner's user id is used as the *effective* user id of the running process. When you design your RBAC system, you need to think about this compatibility issue. You may want to think about the following questions:

- If a process already carries several roles, when this process executes a Set-UID program (a child process will be created for this program), what roles should the child process have? Should it inherit the parent's roles? Why?
- Should the child process be initialized with the effective user's default roles? Why?

When making your decisions, you should always keep security in minds. Some developers like to add features to their systems, without carefully thinking about the security consequence. For the decisions that you make, you need to justify them lab reports, and be prepared to defend them during the demonstrations.

3.4 Compatible with the File Capability Mechanism

Since Linux kernel version 2.6.24, capabilities can be assigned to files (i.e., programs) and turn those programs into privileged programs. When this type of privileged program is executed, the running process will carry the capabilities that are assigned to the program. With this mechanism, many Set-UID programs can become non-Set-UID programs (see our *Capability Exploration Lab* for details).

Your RBAC-enabled system should be compatible with the file capability mechanism. Since programs with capabilities are privileged program, you need to be very careful when dealing with them. The following questions might be able to help you think about your design:

- When a privileged program is executed, what should be the capabilities carried by the new process? Remember, you are dealing with two sets of capabilities: one is from the program, the other is from the parent process.

Whatever decisions you make, you should justify them in your reports and be prepared to defend them during the demonstrations.

3.5 LSM Documents.

To implement this lab, you need to understand and learn how to use the Linux Security Module (LSM). We have developed several documents to help you, and we suggest that you read them first, become familiar with LSM, before you start designing your RBAC system. The links to these documents are put in the lab web page (http://www.cis.syr.edu/~wedu/seed/Labs/RBAC_Linux/). Here we only describe the topics covered by these documents:

- *LSM: A small example* [2] – To show you how to use LSM for access control, we have built a very simple example. (1) In this example, the kernel checks whether a USB drive with a specific ID is attached to the computer: if not, the execution privilege will be disabled, and the user cannot execute any command. (2) To make this example a little bit more complicated, the kernel attach a small token to any process created when the USB is inserted. With this token, the process can kill any process they want.
- *LSM: How to add/maintain your own kernel data structure* [3] – You have to store the data structures related to roles and capabilities in the kernel. These data structures also need to be attached to the process data structures, so during access control, the kernel can find the role/capability information using the process context. This document tells you how to maintain such data structures in LSM, how to link them to processes, and when to initialize them, etc.
- *LSM: Using `ioctl()` to interact with LSM* [4] – User-level programs should be able to interact with LSM; for example, a program might want to disable a role. However, all the data in LSM are within the kernel space; applications cannot directly access those data. They have to ask LSM to do it for them. The question is how to instruct LSM to do things like this. A generic method is to treat LSM as a device, and then use `ioctl` to control this device by instructing it to do things that you want.

4 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to us. Please sign up a demonstration time slot with the TA. Please take the following into consideration when you prepare for demonstration:

- The total time of the demo will be 15 minutes, no more additional time would be given. So prepare your demonstration so you can cover the important features.
- You are entirely responsible for showing the demo. We will NOT even touch the keyboard during the demonstration; so you should not depend on us to test your system. If you fail to demo some important features of your system, we will assume that your system does not have those features.
- You need to practice before you come to the demonstration. If the system crashes or anything goes wrong, it is your own fault. We will not debug your problems, nor give you extra time for it.
- During the demo, you should consider yourself as salesmen, and you want to sell your system to us. You are given 15 minutes to show us how good your system is. So think about your sales strategies. If you have implemented a great system, but fail to show us how good it is, you are not likely to get a good grade.
- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.

5 An Important Message

I sent the following message to my students after their final demonstration of this project. It is a lesson that we should all learn from.

I was upset by the design and implementation decisions that you guys have made regarding CAP_SETUID and CAP_CHOWN. Most of you (except 4 people) demonstrated to me a flawed system. When I confronted them about this flaw, many said that they knew this problem, they just didn't have enough time to fix the problem. Let me show you the logic: fixing the problem takes only about less than 30 minutes, but you guys would rather spend 10 hours to make the role delegation work, rather than spend 30 minutes to fix such a major security flaw in your system. This is not what I have taught you in my class.

You are not alone; many software developers have the same attitudes like yours: they would rather spend many many hours on some nice features (so they can sell the product with a good price) than spending some time ensuring that their systems are secure. After all, security does not make money, nice features do. When they are under the pressure of deadlines, many developers choose features, like what you guys did. Just remember, although security does not make money, a simple flaw like what you guys made can cause millions of dollars in loss and damage of reputations.

I have deducted 10 points from your grade if your system is flawed. This is only symbolic. I should have deducted 50 points, because you guys are trying to "sell" me a flawed system at the end of a computer SECURITY class. This is such an irony! What makes the thing even worse is that many of you know the flaws, but feel the priority of fixing the flaws is too low for you to spare 30 minutes of your time.

As I said in the last lecture of the course: you may forget the contents of my lectures after your final exam, but you should keep the "sense of security", and take that sense to your jobs. I hope that you can learn a lesson from these 10 points. If in the future, you are facing a similar choice: features or security (I am sure you will face this kind of choice quite often), I hope that you remember this lesson.

– Kevin Du, April 30, 2008.

References

- [1] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001.
- [2] Jinkai Gao. How to use Linux Security Module: A small example. In *SEED Project Documentation*. Available at <http://www.cis.syr.edu/~wedu/seed/Documentation/>.
- [3] Jinkai Gao. How to add/maintain your own kernel data structure in Linux Security Module. In *SEED Project Documentation*. Available at <http://www.cis.syr.edu/~wedu/seed/Documentation/>.
- [4] Jinkai Gao. How to use `ioctl()` to interact with Linux Security Module. In *SEED Project Documentation*. Available at <http://www.cis.syr.edu/~wedu/seed/Documentation/>.
- [5] Wenliang Du. Linux Capability Exploration Lab. In *SEED Labs*. Available at http://www.cis.syr.edu/~wedu/seed/Labs/Capability_Exploration.

- [6] Jinkai Gao. How capabilities works in Linux. In *SEED Project Documentation*. Available at <http://www.cis.syr.edu/~wedu/seed/Documentation/>.