Role-Based Access Control (RBAC) Lab – Minix Version

Copyright © 2006 - 2009 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 two-fold. First, this lab provides students with an oppurtunity to integrate two access control principles, capability and the Role-Based Access Control (RBAC), to enhance system security. Second, this lab allows students to apply their critical thinking skills to analyze their design of the system to ensure that the system is secure.

In this lab, students will implement a simplified capability-based RBAC system for `Minix`. The simplification on RBAC is based on the RBAC standard proposed by NIST [1]. 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.

2 Lab Tasks

2.1 Task 1: Capabilities (40 points)

In a capability system, when a process is created, it is initialized with a list of capabilities (tokens). 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. In this lab, we have defined 80 capabilities, but only 6 of them are meaningful and need implementation; the others are just dummy capabilities.

1. `CAP_ALL`: This capability overrides all restrictions. This is equivalent to the traditional "root" privilege.
2. `CAP_READ`: Allow read on files and directories. It overrides the ACL restrictions regarding read on files and directories.
3. `CAP_CHOWN`: Overrides the restriction of changing file ownership and group ownership. Recall that for security reasons, normal users are not allowed to call `chown()`. This capability overrides the restriction.
4. `CAP_SETUID`: Allow to change the effective user to another user. Recall that when the effective user id is not root, calling `setuid()` and `seteuid()` to change effective users is subject to certain restrictions. This capability overrides those restrictions.
5. `CAP_KILL`: Allow killing of any process. It overrides the restriction that the real or effective user ID of a process sending a signal must match the real or effective user ID of the process receiving the signal.
6. `CAP_ROLE_Delegate`: This capability is related to roles. It will be discussed in the RBAC section.

7. `CAP_7`, ..., `CAP_80`: These are dummy capabilities. They will not affect access control. We just "pretend" that these capabilities can affect access control. We want to have a significant number of capabilities in this lab to make the management (the next part) more interesting.

You need to demonstrate how these capabilities affect your access control. Although the dummy capabilities will not affect access control, they need to be included in your system, so we can assign them to roles in the RBAC part. Moreover, you should be able to show their existence in your demonstration. One possible way is to implement a mechanism that can be used by administrators to print out any process's capabilities.

You are warned that the person who provides the above capability requirements have not fully thought through the security consequence of the requirements. Therefore, if you implement the above requriements as they are, your system might be flawed. Remember that an important goal of designing these capabilities is to divide the super-powerful `root` privileges into smaller less powerful privileges, so they can be used to achieve the principle of least privileges in applications. If a person who is assigned a privilege A can get more privileges using A, your system has a security flaw.

It is your responsibility to revise the above requirements to make them secure. You need to fully analyze their security consequences, document your analysis, and provide a revised and secure set of requirements in your report. If your system is flawed, we will deduct up to 30 points, regardless how beautiful your system is or how many nice features you have implemented.

2.2 Task 2: Managing Capabilities Using RBAC (40 points)

With these many (80) capabilities and many 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.

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 `Fedora Linux` and `Trusted Solaris`.

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. In this lab, students need to implement RBAC for `Minix`. The specific RBAC model is based on the NIST RBAC standard [1].

(A) Core RBAC. Core RBAC includes five basic data elements called users (USERS), roles (ROLES), objects (OBS), operations (OPS), and permissions (PRMS). In this lab, permissions are just capabilities, which are consist of a tuple (OPS, OBS). Core RBAC also includes sessions (SESSIONS), where each session 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 (e.g. via `login`), a new session is created. All the processes in this login session belong to the same RBAC session.

When the user logs out, the corresponding RBAC session will end. [1] A user can run multiple login sessions simultaneously, and thus have multiple RBAC sessions, each of which can have a different set of roles. In `Minix`, we can create a maximum of 4 login sessions using ALT-F1, ALT-F2, ALT-F3, and ALT-F4.

Based on these basic RBAC data elements, you should implement the following functionalities:

- **Creation and Maintenance of Roles:** Roles in a system cannot be hard-coded; administrators should be able to add/delete roles. To simplify implementation, we assume that the role addition and deletion will only take effects after system reboots. However, you are encouraged not to make this simplification.

- **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.

- **Enable/Disable/Drop Roles:** When a user initiates a new session, all the user's roles will be in a disabled state (we call them inactive roles); namely, the roles will not be effective in access control. Users need to specifically enable those roles. [2] An enabled role is called an active role. The following functionalities should be supported:

  - During a session, a user can enable and disable any of their 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.
  - If a session does not need a role anymore, it should be able to permanently drop the role using `DropRole`. Once a role is droped from the session, there is no way for the user to regain that role during the current session. However, new sessions will still have that role.

[1] You need to pay attention to the following situation: if some processes (possible for `Unix` OS) are left behind after the user logs out, what will happen to those process? Do they still have the privileges associated with the original session? You should describe and justify your design decision in your report regarding this issue.

[2] For example, they can put the enabling commands in their `.login` file.

(B) 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 adminstrators. 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.

2.3 Task 3: Supporting the Set-UID Mechanism (20 points)

Sometimes, to conduct an operation, a user might need additional privileges. To enable this operation, we can assign the required privileges to the user; however, once the privileges are assigned to the user, it is difficult to prevent the user from abusing the privileges (i.e., using the privileges on other undesirable operations). A solution to the dilemma is to use the `Set-UID` mechanism, which is implemented in most of the `Unix` operating system. With this mechanism, we can mark certain programs as `Set-UID` programs. Whoever runs a `Set-UID` program will run the program with the program owner's privileges. Therefore, users gain the required privileges only temporarily and only within the scope of the program.

In the particular `Unix` implementation, whoever runs a `Set-UID` program will run the program using the program owner's id as its *effective* user id; this way, the user can gain the program owner's privileges because access control is mostly based on the effective user id. In this task, we would like to extend the `Set-UID` concept to *roles*. More specifically, with the extension, a `Set-UID` program will *allow users who run the program to gain the roles of the owner of the program*. For example, if the owner of the `Set-UID` program is U, a user who runs this program will run this program using U's roles, instead of his/her own roles. Your extension should be compatible with the origianl `Set-UID` mechanism, i.e., if your implementation is correct, all the `Set-UID` programs in the original `Minix` system should work as usual.

You should be very careful when dealing with the relationship of the obtained roles and the session. If not carefully, you might introduce some major flaws into your system through this mechanism, because the mechanism allows users to gain additional privileges.

2.4 Task 4 (Optional): Set-Role Mechanism (10 bonus points)

The above `Set-UID` mechanism allows a user to grant all his/her privileges to a program, such that whoever runs the program will gain those privileges within the scope of the program. This is not desirable, especially

if the users have too much power. A better alternative is to allow the privileged user to grant a *subset* of his/her own provileges to a program, instead of all his/her privileges.

In this lab, the above goal can be achieved by associating a subset of the user's roles to the program, such that, whoever runs this program will run this program with the associated roles, instead of with his/her own roles. We call this mechanism the `Set-Role` mechanism. A challenging issue of this method is to find a place to store the role information. A good choice is the `I-nodes`.

2.5 Implementation Strategies

You can start your design and implementation by assuming that all capabilities are dummy. Namely, you do not need to concern about how those capabilities will be checked by the system. This can make your life easier. You basically assume that the capability will be eventually be used by access control. This way, you can focus on how to enable RBAC and capability in `Minix`, such that when access control needs to use those capabilities, they can find the capabilities in an efficient way.

You should be able to test you implemenation independently, regardless of whether the capabilities are dummy or not. Of course, you need to implement some utilities, which allow you to print out the role and capability information of a session and process.

After your RBAC part is implemented and fully tested, you can focus on the capability part. More specifically, you need to modify `Minix`'s access control, so those non-dummy capabilities can actually affect access control.

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.

3.1 Initialization

When a user logs into a system, a new session will be initialized. There are two important questions that you need to think about regarding this initialization: (1) where does this session get the initial roles? and (2) which program assigns these roles to this session? You might want to take a look at `login.c` under the `usr/src/commands/simple` directory.

3.2 Capability/Role in Process or Session

You need to consider the following issues related to processes:

- Since capabilities are the one used by the system for access control, the OS needs to know what capabilities a process has. How do we let OS know the capabilities. Should each process carry just roles, or both roles and capabilities, or just capabilities? You need to justify your design decisions in your report.
- Where do you store roles/capabilities? They can be stored in kernel space (e.g., capability list), in user space (e.g., cryptographic token), or in both spaces (like the implementation of file descriptor, where the actual capabilities are stored in the kernel and the indices to the capabilities are copied to the user space). Which design do you use? You should justify your decisions in your lab reports.

- You need to study the process-related data structures. They are defined in three places: file system (`/usr/src/servers/fs`), process management (`/usr/src/servers/pm`), and kernel (`/usr/src/kernel`).
- How does a newly created process get its roles?
- When system boots up, a number of processes (e.g. file system process and memory management process) will be created; do they need to carry roles?

3.3 Use Capabilities for Access Control

When a process tries to access an object, the operating system checks the process' capability, and decides whether to grant the access or not. The following issues will give you some hints on how to design and implement such an access control system.

- To check capabilities, you need to modify a number of places in `Minix` kernel. Be very careful not to miss any place; otherwise you will have a loophole in your system. Please describe these places and your justification in your lab report.
- Where do you check capabilities? You should think about applying the *reference monitor* principle here.
- The capability implemented in this lab co-exists with the `Minix`'s existing ACL access control mechanism. How do you deal with their relationship? For example, if a process has the required capability, but ACL denies the access, should the access be allowed? On the other hand, if a process does not have the required capability, but ACL allows the access, should the access be allowed? You need to justify your decisions in your reports.
- *Root's privileges:* should the super-user `root` still have all the power (i.e. having `CAP_ALL`)? This is your design decision; please justify your decisions.
- *Compatibility issue:* Keep in mind that there will be processes (especially those created during the bootup) that are not capability-enabled. The addition of capability mechanism will cause them not to work properly, because they do not carry any capability at all. You need to find a solution to make your capability system compatible with those processes.

3.4 Helpful Documents.

We have linked several helpful documents to the lab web page. Make sure you read them, because they can save you a tremendous amount of time. These documents cover the following topics: (1) how to add new system calls? (2) how are system calls invoked? (3) process tables in the file system process and the memory management process.

Important Reminder. Please remember to backup a valid boot image before you make modifications; you might crash your systems quite often.

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 demonstraiton:

- 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. Whe 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 gain 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.