RBAC In The Real World

Christine Occhipinti

RBAC In The Real World

Christine Occhipinti

SANS GSEC Practical v1.4 Option2

Online Course

Overview

In the computer industry, access control refers to managing the ability for people to access computers and computer resources. Access control should enhance security without hindering someone from performing his or her job in the organization. There are three different types of access control models: mandatory access control, discretionary access control and non-discretionary access control. Discretionary access control is based on a user's access needs. A system administrator provides access to an object based on a user's need and the user then has the discretion as to whether to pass on this access to other user's or not. Mandatory access control is more restrictive and is normally used in military systems. With mandatory access, all objects and users in the system are assigned a label. A user can only access an object based on the permissions of the label assigned to him/her. Non-discretionary access control is based on roles. Privileges are granted based on a user's role in the organization. A mixture of these different types of access control models are usually required to meet all the security needs in a system. After some research on these mechanisms, Role-Based Access Control (RBAC), a type of non-discretionary access control, was chosen as the best solution to mitigate the risk from vulnerabilities on a system I worked on.

First, a description of RBAC is given and the different components of RBAC. Next, the system is described in order to understand the vulnerabilities that were faced. Then the vulnerabilities and the risks that they posed are explained and the different solutions that were considered. The paper takes you through each vulnerability as the RBAC solution for each is described. Finally, some problems that were confronted are mentioned and the conclusion of how the solutions have secured the system.

RBAC Description

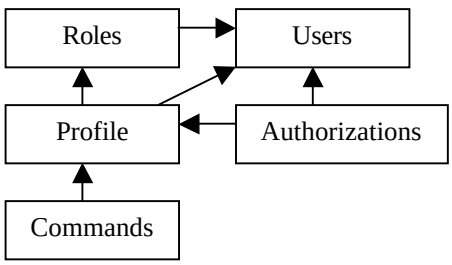
With RBAC, access control decisions can be based on the functions that a user performs in his job assignment. Control is given to a user based on a role. A set of operations is assigned to a role and the role is then assigned to only those users who need access to those operations. A user is only allowed to execute the functions in the roles that are assigned to him/her. This framework maps well to the organization of a company. For example, a user could be assigned the doctor role. With this role, the user has permission to administer prescriptions. Another user could be assigned the pharmacist role. This role would allow a user to provide the medication for a prescription. The user with the doctor role only has permission to administer prescriptions and would not be allowed to fill the prescription. The pharmacist role, on the other hand, would be allowed to distribute the medication, but would not be allowed to give out prescriptions. Another benefit of RBAC is that it allows system administrators to grant a small subset of superuser privileges to users without providing access to the superuser account. [3] The more users on the system that have superuser access the more

susceptible the system is to a security breach, so it is necessary to limit access to the superuser account.

RBAC supports a variety of security policies including the principle of least privilege and separation of duties. [2] RBAC is useful in enforcing the principle of least privilege. The user is given privilege to perform only operations necessary for his job and nothing more. This is often difficult in less precisely controlled systems due to various constraints. Providing more privilege than necessary could often mean unlawful access. [5] The separation of duties is enforceable through RBAC by using roles. For example, an accountant and an auditor have different jobs to perform. An accountant keeps track of a company's spending and profit. The auditor will audit the records that the accountant keeps to verify that the company is following the law. If both the accountant and auditor are the same person then they could perform illegal actions without getting caught. Separation of duty is important in order to keep each other in check.

The three main components of RBAC are profiles, roles and authorizations. Profiles are the entities that the commands are assigned to. Each command can be assigned the real user ID, effective user ID, real group ID and effective group ID that it will execute as. One or more profiles can be assigned to a role or a user. A role is similar to a normal user except that a user cannot log directly into a computer as that role. A user must **su** to a role in order to assume that role. Roles are assigned to users. Authorizations are special accesses granted to a user or role. RBAC-compliant software can check for this authorization before granting access to the user. A profile can be directly assigned to a user in order to skip having to enter a password in order to switch to a role. This is not recommended since a user could inadvertently cause problems with these extra privileges. [7]

The information described above is saved in four different RBAC database files: user_attr, prof_attr, exec_attr, and auth_attr. The user_attr file assigns the profiles and authorizations to either roles or users. The prof_attr file defines the profiles and their assigned authorizations. The exec_attr file assigns commands and their security attributes to profiles. The auth_attr file defines authorizations and their attributes. The user_attr file is located in /etc and all other files are located in /etc/security. The following diagram, from the white paper "RBAC in the Solaris Operating Environment", shows how the RBAC elements work together:



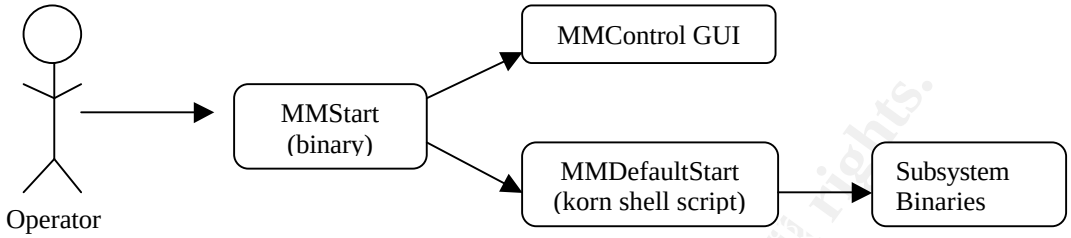
In order to execute the profile commands as the correct user, the commands must be run in a profile shell or with the pfexec command. The profile shells are custom built shells that perform as normal shells except for the limitations that the profile enforces. These shells are pfcsh, pfsh, and pfksh. The pfexec command takes commands and executes them according to the profile setup in the RBAC database files. When inside a profile shell, it automatically performs the pfexec. These 3 profile shells and pfexec are all located in /usr/bin. Profile shells are normally assigned to roles as their login shells in the /etc/passwd file; this restricts someone from logging in directly as that user.

System Description

Two of the vulnerabilities in the system originated from a group called the Monitoring and Metrics (MM) group. This group was in charge of the starting and stopping of processes in the system and the monitoring of these processes. The monitoring of these processes was done through a Graphical User Interface (GUI) called the MMControl GUI so that the operator could see whether a process was up or down. The initialization of the system began when the operator kicked off a binary called MMStart. All processes that were brought up by the MMStart binary ran under the operator id. The MMStart binary automatically brought up the critical processes of the system, but all other processes were started through the GUI. These critical processes were brought up via the MMDefaultStart script. The arguments to the MMDefaultStart script were a directory and a binary name followed by any arguments the binary required. The directory passed in was the directory where the binary resided. The MMDefaultStart script performed a change directory to the directory that was passed in and then did an exec on the binary name and its arguments, if any were passed in. For example, if I am starting a binary called CriticalProcess which resides in /usr/local/bin with the argument –a then it would get called like this: MMDefaultStart /usr/local/bin ./CriticalProcess –a. If the binary needs to be started on a remote machine then a secure shell is established that executes the MMDefaultStart on that machine.

Since only critical processes were automatically started during initialization, the operator used the MMControl GUI to bring up all other processes. The operator was allowed to start/stop only certain processes in the system. When the operator selected to start a process from the GUI, the GUI then kicked off a MMDefaultStart with the appropriate arguments as described above. When a process needed to be stopped, the operator selected to stop the process from the GUI and the GUI then executed the MMDefaultKill script. This script was a root owned setuid script that performed a grep for the binary and sent a kill signal to the binary, if it was found. The MMDefaultKill script takes the following arguments: environment name, binary name, and identity. The environment name was the name of the environment that the script is running in. The binary name was the name of the binary to send the kill signal to and the identity was a unique name for any binaries that might have more than one instance running in the environment. For instance, if the environment was “tstenv” and using the

example of the CriticalProcess from the paragraph above, MMDefaultKill would be called like this: MMDefaultKill tstenv CriticalProcess CriticalProcess. The binary name CriticalProcess was used twice since there is only one instance of this binary in the environment. The following diagram depicts the initialization of the system:



Vulnerabilities in the System

I worked with a team consisting of several people to identify the vulnerabilities and their risks, and to design a solution for their problems. I performed the implementation of the design alone. When I began testing the implementation, I had someone from each group helping me since it was their code and they knew it best. I regrouped with my team occasionally to go over possible solutions to any problems I ran into.

The first vulnerability I faced was with the MMDefaultKill setuid root script. As described above, this script reads in a process name as an argument. This script performs a grep for the process passed in and sends it a term signal, if it was found. If the term signal does not work, it then sends a kill signal. Some of the processes it kills run as a different user than the actual script, so therefore it needed special privileges in order to perform the kill. SUID programs are one of the major weaknesses that attackers have used to gain root access. One major problem with a setuid script is that it is possible to manipulate the command interpreter to perform actions that were not intended to be executed by the script. These unintended actions could be used to violate system security. An example of this would be where a site has a bourne shell setuid script. It is possible for an attacker to manipulate the shell, which interprets the commands, to lead it to believe the attacker is a person logging into the system. This produces an interactive shell that the attacker can then use to execute other commands. [1] Another problem with the MMDefaultKill script was that the kill command was not called with its full path name. This would enable an attacker to manipulate the PATH variable to run a personal version of the kill script. This personal kill script could execute commands to compromise the security of the system.

The second potentially threatening situation was that the whole system executed as the operator who started the MMStart binary. Some processes in some of the subsystems needed special access privileges. In order to grant these processes special privileges, these privileges would have to be granted to the operator;

although, the operators themselves would not need these privileges. This would violate the principle of least privilege. For example, one of the subsystems deals with transferring confidential data between two entities via tape. To get this information on and off of the tape, the subsystem performed a tar into and out of certain directories. Since this was confidential data, only certain people were allowed to view this information. With the unsecured configuration of the system where all binaries executed as the operator's id, this would mean that this subsystem would be running under the operator's id. Therefore, the operator would then have permission to look at the information that was put onto tape and taken off of tape. The operator does not have a need to view this data in order to perform his/her job. The risk here is that the operator could use this information to his/her advantage by selling the data for money or by destroying the data if they ever became disgruntled with their employer.

The third vulnerability dealt with the Configuration Management (CM) team for the system. They were in charge of deploying and configuring the system in the test and production environments. The system was built in increments and each increment extra functionality was added to the system. After development was done for an increment, all the code was built and deployed into a test environment where it was integrated and tested together. The CM team was in charge of the process of deploying and configuring the code in this test environment. They had scripts that copied the binaries from the baseline for that increment into the environment. There were certain situations where they needed to be able to change ownership of a file/directory in the environment. The operating system was locked down to only allow root to perform a change of ownership. For example, after creation of the logs directory for each user, it needed to be changed to the ownership of that user. The creation of the environment and deployment of the code was done as user cmuser. This user had to then chown the log directories for each user in the environment. This was to allow the user write permission to the directory and disallow other's access to the directory. We did not want to give the cmuser user access to the root account. This user did not need the extra permissions that root had. The user only needed to be able to chown. The less number of people who had access to the root account meant the less likely there was to be a security violation due to root access.

Possible Solutions

Before deciding to use RBAC to fix the vulnerabilities, other access control mechanisms were investigated. A discretionary access control mechanism available to use was Access Control Lists (ACLs). An ACL is a way to control which users have access to a specific file/directory. A list is built for the file/directory and in that file is a list of users who have permission to read/write/execute that file/directory. ACLs do not allow someone to execute a binary as another user id, but instead only give access to that binary; therefore, ACLs were not chosen. Sudo was also investigated as an option. Sudo and RBAC basically provide the same type of functionality. They allow the system

administrator to specify that a user can execute a particular command as some other user. One advantage of RBAC over sudo is that RBAC has a finer grain of control over the real and effective Ids that the command runs as. With sudo the real and effective ID's are assigned the same, but in RBAC a system administrator is able to set the effective ID rather than the real ID. [6] During auditing this allows the real user to be held accountable. Accountability is essential when resolving a security breach in a system. RBAC is also easier to maintain than sudo. Sudo requires compilation unlike RBAC. Since sudo does not support NIS mapping, a system administrator will need to keep each sudoers file up-to-date on each different host in the system. In RBAC when using NIS, there is only one set of files to maintain that are centrally located for all hosts to access. Another big factor was that on our system we had to get permission to use freeware. Since we were already using Solaris 8 and RBAC comes with Solaris 8, it was much easier to use and RBAC provides the same functionality as sudo. On top of that, the process of getting permission is somewhat involved and tedious.

RBAC Solution

The first two vulnerabilities were solved in conjunction with each other. It was decided that we would not use roles in RBAC, but instead would simply assign the profiles that were created to a user. Since all processes are started and stopped either through MMStart or through the MMControl GUI, then it would not be an easy task to require the operator to enter a password when starting and stopping processes. This password would be required if roles were being used. Our first step was to decide whom the system should be running as. After talking with the MM group, we determined that the MMControl GUI would have to run as the operator, but everything else could be run as some other user. Since all processes except the MMControl GUI are started via the MMDefaultStart script, we decided this would be a good place for the change of user. The pfexec could be called on the MMDefaultStart script and would then run as a unique MM user; therefore, the processes that the MMDefaultStart script exec'ed would be executed as that same unique MM user. To create this unique id, I decided it would use the environment name followed with "mm". For example, if the environment name was "myenv", then the unique id would be "myenvmm". The following example shows the necessary lines in the RBAC database for operator "tstusr" to be able to run the system as the "myenvmm" user:

```
user_attr:
tstusr::::type=normal;profiles=MM Start

prof_attr:
MM Start:::Ability to execute MMDefaultStart as myenvmm usr:

exec_attr:
MM Start:suser:cmd:::/products/swit/MM/bin/MMDefaultStart:uid=myenvmm
```

Adding the pfexec call was very simple. The code used a configuration file from which it determined what script to execute to start all the binaries. Normally, this was set to "./MMDefaultStart". I changed this value to "/usr/bin/pfexec ./MMDefaultStart".

The first problem I ran into with this solution was that one of the security features of RBAC is that when it calls setreuid() to change users, it removes the LD_LIBRARY_PATH variable. Unfortunately in the system, we had a lot of COTS products that depended on this variable to be set. In order to fix this, I had to reset this variable. In our program there was a team in charge of maintaining and updating the .cshrc's for all the operators and would also be in charge of maintaining the .cshrc for the MM user (i.e. myenvmm)--all operators of the system used csh instead of ksh. One option was to reset the variable in the MMDefaultStart script, but this would mean extra maintenance. Not only would this team have to change the LD_LIBRARY_PATH in the .cshrc's, but a developer would also have to change the LD_LIBRARY_PATH in the MMDefaultStart script. Not to mention that these could easily get out of sync if the appropriate people were not informed when the LD_LIBRARY_PATH got changed. Since the team was already maintaining a .cshrc, I decided the best solution would be to change the MMDefaultStart script from a korn shell to a c-shell. When a c-shell is executed, it sources in the .cshrc of the user executing the c-shell. Therefore, when the pfexec is done on the MMDefaultStart script, RBAC would change users by executing setreuid() and then would execute MMDefaultStart as the MM user and it would then pick up the correct environment variables. There was also the added benefit of extra security since no one would know the password for the MM user except the system administrator who created this user. Therefore, no one would be able to change the .cshrc of this user in order to execute a rogue script/binary; except of course the team in charge of maintaining this .cshrc.

The next step was to solve the vulnerability with the root owned setuid MMDefaultKill script. In the system, when the operator selected to stop a process, the MMControl GUI sent a message to one of the MM servers running. This server then executed the MMDefaultKill script with the appropriate arguments. There was also a command line binary that called the MMDefaultKill script. The operator ran this binary in order to shut down the whole system. I did not want to give the operator permission to run the kill command as root. This would allow them to then run kill on the command line and terminate anything running in the system. Instead I added an extra layer of security. I decided it would be best if the MMDefaultKill script executed as the MM user and then give the MM user permission to run the kill command as root. As mentioned earlier, no one would actually be logging in as this MM user or would know the MM user password except the system administrator who created the account. Using the example above with the "tstusr" operator and the MM user "myenvmm", the following information shows the RBAC configuration:

user_attr:
tstusr::::type=normal;profiles=MM Kill
myenvmm::::type=normal;profiles=Kill Admin

prof_attr:
MM Kill:Ability to execute the MMDefaultKill script as MM user:
Kill Admin:Ability to execute /usr/bin/kill as root:

exec_attr:
MM Kill:suser:cmd:::/products/swit/MM/bin/MMDefaultKill:uid=myenvmm
Kill Admin:suser:cmd:::/usr/bin/kill:uid=0

The pfexec command had to be added in two places. Adding it to MMDefaultKill was as simple as adding it to MMDefaultStart. There was a variable in the configuration file that specified which kill script to be execute to terminate processes in the system. I changed this variable from "./MMDefaultKill" to "/usr/bin/pfexec ./MMDefaultKill". I then had to change the MMDefautKill script by adding a pfexec call in front of the kill command. I also hard-coded the path of the kill command. This was necessary because RBAC would complain about not being able to find the real path name of the binary unless I entered the complete path name or some relative path to it. Also hard-coding the path would not allow an attacker to run a personal version of the kill command. I also hard-coded the pfexec command because of the same reasons.

The third problem was allowing the CM team to run chown as root without giving them root access. The simplest way to solve this was to give the user permission to run chown as root through RBAC. This would be a major security hole since it would essentially allow them to access any file in the system. For example, the user could chown /etc/password to be owned by him. After changing the contents of the file by giving himself root privilege, the user could then change the ownership of the file back to root. Instead of this solution, I wrote a binary called PSChown. This binary allowed someone to only change ownership of files in specific directories. Since the CM group only copied binaries/files to certain base directories, this seemed the best approach. The PSChown binary also checked for symbolic links to verify that someone was not trying to change a file outside of those directories. It also cleared the setuid and setgid bits to not allow someone to create a root owned setuid script/binary. The base directories were different for the different environments so I did not want to hard-code the base directories in the binary. Instead I created a properties file called PSChown.properties which listed all the valid base directories.

The cmuser user through RBAC had permission to run the PSChown binary as chownusr. This chownusr was a user created by the system administrator and whose password the administrator only knew. Chownusr had permission via RBAC to execute chown and chmod as root. This user needed permission to execute chmod as root in order to remove the setuid/setgid bit on the file.

Through the use of RBAC and the PSChown binary, the cmuser user now had the ability to change ownership of files existing in certain directories. The following lines were added to the RBAC files to grant these permissions:

user_attr:
cmuser::::type=normal;profiles=PS Chown
chownusr::::type=normal;profiles=Root Chown/Chmod

prof_attr:
PS Chown:Execute PSChown as chownusr:
Root Chown/Chmod:Execute chown and chmod as root:

exec_attr:
PS Chown:suser:cmd:::/usr/local/bin/PSChown:uid=chownusr
Root Chown/Chmod:suser:cmd:::/usr/bin/chown:uid=0

The PSChown binary was written in C++ and was located in /usr/local/bin. The PSChown.properties file was located in /etc/system. The PSChown binary was owned by chownusr with read and executable permission for all. I gave execute permission to all since it would not work for those users who did not have the correct permissions setup in RBAC. The PSChown.properties file was also owned by chownusr and only provided permission for the owner to read and write. It was very important to lock down this file since adding a directory to this file would allow a user to change ownership of files in that directory.

Problems

During the implementation and testing of my RBAC configurations, I ran across some peculiar problems that were worth noting:

- In order for RBAC to work correctly, the Name Service Cache Daemon had to be running on the machine.
- RBAC removed the LD_LIBRARY_PATH (This is described above).
- Could not use symbolic links in the exec_attr file for the command. When pfexec was called on a command, RBAC used the resolvepath() command to resolve that command name. If that command name did not match the one in the exec_attr file then RBAC did not perform the setreuid().
- It was possible to add more than one command to a profile in the exec_attr file. A line needed to be created for each command.
- It was necessary to give a fully qualified path or a relative path for the command that was being pfexec'd. If this was not done then RBAC complained that it could get the real path. It did not seem to search through the PATH variable for the binary.

Conclusion

Access control is an important part of a system's security needs. Without access control users would be allowed to perform any function on the system. There are different types of access control. Role-based Access Control allows the system administrator to implement access controls that map well to an organizations structure. With RBAC, users are granted privilege to only execute functions that are needed to perform their tasks.

After applying my solutions with RBAC, I believe the state of the system was more secure. The root owned setuid MMDefaultKill script could have easily been manipulated to gain root access to the system. With the fixes in place this was not possible anymore. The only problem that I see with the solution to the first two vulnerabilities would be if a hacker compromised the system and was able to assume the MM user account. Since the MM user account has the ability to run the kill command as root, then the hacker would be able to cause problems in the system terminating anything and everything. The hacker would also have special access privileges based on the access privileges given to the MM user. For example, the situation that I described earlier about only certain users being able to access the confidential data that is written to and taken off of tape. Whoever compromises this account would then be able to access this confidential data. The system has security measures in place to prevent user accounts from being compromised, but this goes beyond the scope of this paper.

With the CM team, RBAC was setup to allow them to be able to change the ownership of files in certain directories. The solution that was put into place achieved this through RBAC and the PSChown binary. The one problem that I see with this solution is that the user given these privileges could maliciously manipulate files under these directories. This problem boils down to a trust factor. There has to be a certain amount of trust with the users in the system because it is necessary for some users to have extra privilege in order to perform their tasks.

1. Bishop, Matt. “How To Write a Setuid Program”. Jan./Feb. 1986. URL: http://nob.cs.ucdavis.edu/~bishop/papers/Pdf/sproglogin.pdf (14 June 2002).

2. Ferraiolo, David F., Janet A. Cugini and D. Richard Kuhn. “Role-Based Access Control (RBAC): Features and Motivations”. 9 Nov. 1995. URL: http://hissa.ncsl.nist.gov/rbac/newpaper/rbac.ps (9 July 2002).

3. Lindstrom, Nathan W. “An Introduction to Role-Based Access Control”. URL: http://www.nathanlindstrom.com/rbac_introduction.html (1 Aug. 2002).

4. Miller, Todd C. “Sudo Manual”. URL: http://www.courtesan.com/sudo/man/sudo.html (14 June 2002).

5. NIST CSL Bulletin on RBAC. “An Introduction To Role-Based Access Control”. URL: http://csrc.nist.gov/publications/nistbul/csl95-12.txt (9 July 2002).

6. Sun Microsystems. “RBAC in the Solaris Operating Environment”. 2000. URL: http://wwws.sun.com/software/whitepapers/wp-rbac/wp-rbac.pdf (3 June 2002).

7. Sun Microsystems. Solaris 9: System Administration Guide: Security Services. Santa Clara: 2002. Chapter 17.