A Role-Based Access Control Kernel for NetBSD

Alistair Crooks
The NetBSD Foundation
agc@NetBSD.org
August 2009

Abstract

This paper looks at the traditional Unix security models, and introduces the Role-Based Access Control (RBAC) security model, a much more finely-grained operation and capability system already deployed in some versions of Trusted Solaris and AIX 6. The development model is discussed, along with various approaches to bringing RBAC kernels to a modern BSD kernel. Some implementation details are then described, and lessons learned from these implementation details are discussed. Finally, the RBAC kernel itself is shown in operation, other systems are compared, and areas for future research are identified.

1. Introduction

The traditional Unix security model has the kernel as the main decision point for carrying out operations. File system operations will be performed if the user has the correct privileges to read from, or write to, a file. Other operations will be carried out only if the user is the *root* superuser, or if the process which is executing has root privileges. The kernel verifies that this is the case by using its *issuser()* function internally. The security model reduces to a "root can carry out any operation" question. With role-based access control, the individual operations within the kernel are organised into groups of logically-related actions - get a raw network socket, format the disk, set the kernel's idea of the time, allow process tracing, etc. Building on these groups of actions, users are given roles - so that an individual user might be able to get a network socket, but not format the disk, or set the time. This much finer-grainedcontrol over operations means that the security model has now changed. Most of our efforts at the current time are put into stopping exploits on vulnerable servers, since once control has been gained by an attacker, any operation is possible, due to trhe broad nature of the root user's privileges. With RBAC kernels, a single exploit is not nearly so harmful - in the unlikely case that an attacker gains control over the ability to open a raw network socket, then that is dangerous, but not fatal. The exploit can be viewed as an annoyance, rather than the highest priority - normal triage can take place by operations staff, forensics can take place without fear of being spoofed, trojanned or overlooked, and it's highly likely that normal operations could continue on that same machine, even if it had been vulnerable to that exploit.

2. Unix Security Models

2.1 Security Models

For the purposes of describing the security model, the whole system can be described by categorising the users and user-level programs and libraries as *"subjects"*, and these subjects perform actions upon entities known as *"objects"*. The operating system manages the ability of subjects to perform these actions. In Mandatory Access Control systems, this security policy is controlled centrally by a security policy administrator, and users do not have the ability to influence this policy. Traditional Unix systems are Discretionary Access Control systems - users have the ability to make policy decisions and assign security attributes to the objects.

Traditionally, DAC has centralised around the user. There is a single, all-powerful super user, known as root. The kernel carries out operations on behalf of the user. To perform some operations, the kernel verifies that the user requesting the operation has sufficient privileges to allow this to be done (file system permissions). This is taken to the extreme case for certain operations - opening a "reserved" network port, access to a raw device - where the kernel has traditionally used the *issuser()* function to check that the user requesting the kernel to perform the operation has an effective uid of zero.

The RBAC security model is a different one - it does not assume that there is a single, all-powerful super user. Rather, this super user's privileges are split up into a number of separate roles - the ability to format a disk, the ability to mount a file system on a disk, the ability to set the kernel's idea of the time, the ability to gain a raw socket, the ability to gain a privileged socket, the ability to trace a process, and many more. The NetBSD RBAC implementation has 57 specific roles defined at the current time, and more will be added. The RBAC security model is separate to the MAC and DAC security models - the distinctino is not how the subjects may interact with the objects, but rather the level of privilege needed to do the same. Whilst there is some overlap between MAC and RBAC (mainly visually in the way the user may interact with the operating system), they are two separate concepts.

2.2 Su and Sudo

This single, all-powerful super-user approach to security has a number of benefits - primarily those of simplicity, ease of use, and understandability. But it is also a model that was first used (in Unix) 40 years ago. Since then, times have changed, and other priorities and challenges have appeared. The Unix "substitute user" su(1) command is less used than it used to be, and typically sudo(1) will be installed on machines these days. For those of us who have just returned from Mars, sudo is used to provide finer-grained control over who can become the superuser, and what they can do as superuser. The original "substitute user" functionality is rarely used, and most people assume that su means "super user". Sudo and su both prompt the user for his own password - if it is provided, then the user is taken to have authenticated properly, and will be allowed to run the command specified on the command line. Although sudo can be used to restrict the commands which users can invoke, it is most commonly used as a production tool to enable users to run commands with an effective uid of 0, at the same time restricting the commands that may be performed in this manner.

Sudo is a fine tool, and is used right across the globe in almost every organisation. The security model it usually implements, though, is the same one of "the root user can do anything". This paper will explore different security models which take a different viewpoint.

2.3 Kauth

NetBSD has used kauth [Efrat2006] since 2006 to abstract away the use of *issuser*() inside the kernel - instead, a function call is made into the kauth with details of the operation required. Kauth is essentially a layer of abstraction which allows finer-grained control over the actions taken in the security model. The intention is for other security models to be written which can be used in operating systems research to explore new security models, and to ease their introduction.

The traditional DAC security model has been continued seamlessly in NetBSD as the *bsd44* security model (found in the *sys/secmodel* directory in the kernel source). Security models can be stacked. If a security model wishes to deny the action, a deny return code is issued. Specific approval can also be given for an action. If there is not enough information to take a decision, then a defer return code is given, and the next security model will be used to take the decision. The default action is to deny - a form of fail safe.

3. Role-based Access Control

Standard security practices in real life are to separate roles to people. In Sarbanes Oxley compliant organisations, testers can not be the same people as the reviewers of the tests.The reviewers must be different from the auditors. This separation of roles ensures that no one person can have all the responsibility, or be allowed to lock out other users on the system by his actions.

One of the ways of doing this is to provide a smaller subset of the users on the machine who are more trusted than others. In BSD-derived operating systems, this is the *wheel* group. This group is not supported by GNU su(1), as found in [Stallman2002]:

> Why GNU `su' does not support the `wheel' group
>
> Sometimes a few of the users try to hold total power over all the rest. For example, in 1984, a few users at the MIT AI lab decided to seize power by changing the operator password on the Twenex system and keeping it secret from everyone else. (I was able to thwart this coup and give power back to the users by patching the kernel, but I wouldn't know how to do that in Unix.)
>
> However, occasionally the rulers do tell someone. Under the usual `su' mechanism, once someone learns the root password who sympathizes with the ordinary users, he or she can tell the rest. The "wheel group" feature would make this impossible, and thus cement the power of the rulers.
>
> I'm on the side of the masses, not that of the rulers. If you are used to supporting the bosses and sysadmins in whatever they do, you might find this idea strange at first.

3.1 Driving Forces for RBAC

Using current secmodels, if an exploit is used to gain root privileges, then the attacker can do anything that root can do - re-format disks, impersonate users, change network settings, read private data for other users, and many other activities.

Using current secmodels, we try to protect against this by making the attacks harder to exploit. Using a different approach, if we were to limit the amount of damage that a successful exploit could lead to, then exploits would be less stressful - less damage could be caused, and the priority for remediating the exploit (and the attendant stress) would be much less than at present.

To illustrate this with an example:

/sbin/ping is a setuid binary which needs root permissions to be able to open a raw socket.

```
% ls -al /sbin/ping
-r-sr-xr-x  1 root  wheel  68448 29 Nov  2007 /sbin/ping
%
```

Root privileges are used to open this socket at the start of execution. Once the socket has been opened, root privileges are dropped, and further execution continues as an ordinary user.

If an exploit for ping is found and published, the attacker can gain complete control of the machine, and can perform any operation that would currently be allowed to be performed by root. i.e. anything.

Using current secmodels, people need to acquire a new ping binary and install it on all affected machines. For large companies, this can involve hundreds of thousands of machines.

```
% ls -al /sbin/ping
-r-xr-sr-x  1 root  net_open_sockraw_role_  68448 29 Nov  2007 /sbin/ping
%
```

The RBAC security model provides a different level of security - if a user or program possesses the "get raw socket" capability, then there is no need for root permissions, or setuid binaries; neither is there any
need to reset the privileges to an ordinary user after opening the socket. Instead, we assign the "get raw socket" capability to the ping program itself. If an exploit for ping exists, then, when it is exploited, the attacker is able to open a raw socket - which is a noticeable difference from the current security model.

This is obviously a trivial, and manufactured, example. Much richer pickings exist with programs such as tftpd, ntpd, sshd, and (possibly) apache. More realistically, any program which has been used to gain
root access - see previous Bugtraq advisories.

In summary, we are implementing finer-grained control over actions within the kernel. We are not asking if the process is root, in order to authorise the action to be performed - we are asking instead if
the process has sufficient privileges to perform the required action.

4. The RBAC Implementation

We have seen that RBAC is desirable to subdivide the operations that root has to perform. (this is done for us already by kauth), so that the system may not be compromised as easily as with the traditional

security model.

At an early stage in RBAC development, it was decided to use the existing groups mechanism to express the capabilities related to the roles needed for RBAC. Users have always been able to belong to multiple groups in the Unix model - Version 6 had a newgrp(1) command to set the current group for a user, and hence the reason for having group passwords in the /etc/group file. Binary files also have (single) group ownerships, enough for conveying the roles to the kernel.

Users traditionally belong to a number of groups; in the same way, the user can perform a number of roles. The mapping between roles and groups seems to be a useful one.

The standard kauth subsystem in NetBSD already has actions, segmented by scopes - logical groupings of operations which kauth has to allow, deny or defer to another scope.

To implement RBAC, we map a role which a user may perform, to a single group. If the user is allowed to perform the role, he will be a member of the group which maps to that role. If the user is not allowed to perform the role, then he will not be a member of that group.

We create a new secmodel - when an operation is attempted, control passes into the kauth subsystem, and the group memberships of the user are searched to see if one of the groups is the desired role. If it is, the operation is allowed. If not, then the operation is either denied, or deferred.

4.1 Implementation Stages

The initial implementation of the RBAC security model called for roles to be mapped to kauth actions, and for these to be given to hardcoded groups at user level. The two need to map 1 to 1 for RBAC to work
effectively.

- Firstly, the RBAC secmodel was added. It looks at the group membership, and takes a decision to allow, deny, or defer based on the groups to which the entity belongs. It is a small amount of code, and is segregated in its own secmodel directory in src/sys/secmodel/rbac.

- The /etc/group file needs to be changed to add all the role groups. The development of this part was aided by the mkgroups shell script (see listing 1) which takes the header file of all the defined roles, and creates lines for the /etc/group file corresponding to these roles. This way, the 1:1 mapping is kept. The header file uses hexadecimal numbers to define the roles. The groups file uses decimal numbers to denote group numbers. The group library functions cannot decipher hexadecimal numbers in the group file, so the mkgroups script has to change the group numbers from hexadecimal to decimal.

- Following that, every user-level program which uses setuid or setgid execution must be examined, along with every daemon that runs as root, whether in a chroot jail or otherwise. The setuid root programs need to modified to drop root privileges all together. Each program which needs privileged roles to operate effectively can be assigned a group ownership of that role.

Occasionally a program may need a number of roles - for this, a number of separate roles can be grouped together, solely for the benefit of this program. For example, if a program were to wish to get a raw socket, and set the kernel's idea of the time, it would

need two roles - that of procuring a raw socket, and one to set the kernel's idea of the time. These programs are, at first sight, innocuous - however, the danger of aggregating roles in such a way is that we lose the finer-grained access control provided by RBAC, and move towards the older "all our eggs in one basket" security model.

As part of the development process, each role "group" (where the roles are logically grouped together) was given a "super-group" role. In effect, this role allows all system, or all disk, or all network operations. In practice, this role has, so far, not been found to be useful, and it is highly likely that this will be removed in the final product.

This last phase is the longest - it is also the most time-consuming, but can be done by multiple people simultaneously.

4.2 Implementation Constraints

NetBSD has a maximum number of usable groups of 16. This is a hardcoded limit, and is driven by the NFS limit to the number of groups. We have at least 57 kauth actions, and so mapping a role to a group needs some sub-division.

In addition, group ids cannot have the high-bit set (a gid is a 32 bit integer, so the largest gid that is available is GID_MAX, or 2^{31}-2)

The initial implementation of RBAC initially used the 31st bit to denote that this was a role. It was considered that people tend to use low group numbers (simply because of the difficulties of copying,
interpreting and reading larger numbers), and so abusing the higher end of the gid spectrum would not be a problem for existing installations. However, once implemented, this did not scale, and was
difficult to use.

The first development attempt consisted of the following:

- RBAC secmodel added was added to the kernel
- a new RBAC kernel config was generated, with the secmodel changed to rbac
- kern/kern_auth.c modified to include a new function to check scope & bitmasks of gids in user credentials
- /etc/group was modified to include a number of new groups, using the *mkgroups* script
- RBAC kernel built with the new rbac secmodel, and the kernel installed in a virtual machine

4.3 Exporting Roles to Userlevel

The roles need to be added to the group file to maintain the 1:1 fixed mapping between roles and groups in the kernel and at the userlevel. There will be further discussion later on about the values of the gids used in the group file, but, for the purposes of RBAC development, a separate group file was used for the RBAC security model, compared to the standard bsd44 security model. The rbac group file had gids like the following added:

```
...
net_altq_role_:*:805306369:
net_bind_privport_role_:*:805306370:
net_interface_getpriv_role_:*:805306372:
```

```
net_interface_setpriv_role_:*:805306376:
net_nfs_export_role_:*:805306384:
net_nfs_svc_role_:*:805306400:
net_open_sockraw_role_:*:805306432:
net_route_role_:*:805306496:
net_socket_cansee_role_:*:805306624:
net_socket_rawsock_role_:*:805306880:
netroot:*:805307391:
...
```

added to it to support the RBAC roles needed. It is highly unlikely that the gids selected will overlap with pre-defined group ids, since it is highly unlikely that gids are ever used in practice, even in large installations.

4.4 Kauth actions

Kauth has a number of actions defined. Each secmodel must choose to allow, deny, or defer these actions, and the RBAC secmodel is no different. However, it is different to the bsd44 secmodel - there, the decision to allow or deny was taken on the effective uid of the calling process being zero (which would allow the operation), or non-zero (in which case, with a very few exceptions, the operation would be denied). Kauth groups these actions logically into groups of actions called *scopes*. There is a network scope, a system scope, etc.

```
/*
 * Network scope - actions.
 */
enum {
        KAUTH_NETWORK_ALTQ=1,
        KAUTH_NETWORK_BIND,
        KAUTH_NETWORK_FIREWALL,
        KAUTH_NETWORK_INTERFACE,
        KAUTH_NETWORK_FORWSRCRT,
        KAUTH_NETWORK_NFS,
        KAUTH_NETWORK_ROUTE,
        KAUTH_NETWORK_SOCKET
};
```

4.5 Initial bringup

The initial development of the new RBAC secmodel went well, and new listeners were written for each scope. The secmodel for the RBAC kernel config was set to *rbac* instead of *bsd44*, and a test kernel was booted. Unfortunately, this "big bang" approach to kernel development did not work particularly well. The new security model booted up into multi-user straight away, but the network had not come up as expected, and the update mount for the root file system had also not worked as expected. It was enough, however, to validate that the new secmodel was straightforward from the kernel point of view, but needed more work at the user level.

The next phase of development went back to revisit the rbac secmodel itself. A table of RBAC roles was introduced to the secmodel, and the system was migrated from the bsd44 secmodel to the rbac secmodel one role at a time. This allows much finer-grained control over development. It also allowed us to see early on how the RBAC secmodel would work in practice. Some of the effects are unexpected, as the following output shows (the role in development is that of RBAC_SYS_MKNOD):

```
rbac# id
uid=0(root) gid=0(wheel) groups=0(wheel),2(kmem),3(sys),4(tty),5(operator)
rbac# mknod -m 600 node c 0 0
mknod: node: Operation not permitted
rbac# su - agc
$ id
uid=1000(agc) gid=1000(agc) groups=1000(agc),1342177296(sys_mknod_role_)
$ mknod -m 600 node c 0 0
$ ls -al node
crw-------  1 agc  agc  0, 0 Feb  4 06:19 node
$ exit
rbac# exit
```

4.6 GIDs and Roles

We briefly described the range of gids used in developing the rbac security model. The initial idea was to use gids with the high bit set, but this proved to be unworkable - the various libc routines which manage and list group file entries cannot be larger than 2^{31} as the high bit is used internally to denote a process group. RBAC development continued by using the gids characterised by the lower 7 bits of the high byte being non-zero. Whilst this causes the rbac secmodel to use large gids, none of these gids ever needs to manipulated numerically, save for its presence in the group file.

The high byte is then split up into groups of roles, which also mirror the split into kauth action scopes. The code in kauth which inspects the gids in the group list first masks the high byte, and then uses a bit mask on the other 3 bytes to determine the rbac role. This is just a convenience, and is done this way for clarity - integer equality could just as easily have been used for the rbac roles.

4.7 Development Issues

It is interesting to note the various issues that were encountered during development:

- Early on in development, it became apparent that the size of the group name was an issue. The /etc/security script complains if the length of the group name is larger than 16 characters. This is not a fatal flaw, but rather an artefact of the way that rbac development has been carried out, with the kauth scope driving the role name, which in turn drives the group name in the group file.

- The hybrid approach to secmodel development (some roles traditional bsd44 secmodel, new ones rbac) works well enough to identify user level dependencies and pre-requisites.

- In moving roles from the bsd44 model to rbac, all the parts that use rbac authorisation need to be converted to use the new secmodel at the same time. Whilst this may seem obvious, there are often user level utilities and third party

packages which use the role, and all must be transitioned at the same time. If not, some calling sites will expect to use the effective userid of zero as an authorisation decision point, whereas the other parts of the system will use the new role presence. An example of this is *rbac_sys_shutdown* - when the time came to transition the role from bsd44 to rbac, root was still able to call shutdown(8), but after that, and during system shutdown, with /etc/nologin in place, the role was used

- Transitioning everything to the rbac secmodel is a lot of work, and most of that is in the user level. It is not particularly difficult, but identifying calling sites calls for searching the kernel for instances of the kauth action, finding the system call which causes this action to be called, and then looking through all of the userland source to see where this system call might be used. This is before third party software is even considered. For example, NET_RAW_SOCKET includes hostapd, icmp, routed, racoon, and ntp
- The transition for various programs can be distilled into the following rule: if the binary is setuid root, then it is likely that it will become setgid role. If the user has in the past used su or sudo to invoke a program, then the user will have to either have the relevant role in his list of groups (in /etc/group), or will have to define a number of users in the sudoers file with the relevant group access. In practice, a user will usually have a number of roles, but not too many - the principle of separation of duties in real life on machines which are vulnerable is often used to accomplish this. Also the MLS products and MAC achieve this in a different way.

- Some of the actions require us to take a decision based on whether the system is booting, as well as when the system has booted. One example of this is update mounts, which should be allowed when the system is moving out of single user into multi-user, and needs to remount the root file system. The global kernel variable *cold* could be used for this, except that it is only set in the time before text is output on the console (i.e. by the time "twiddle" happens, *cold* has been cleared). There are other ways of verifying the status by inspection of values of *curlwp* pointing to *proc1*, or by inspecting the uptime values, but there are security concerns about adding a variable which could be modified using a binary editor or debugger. For now, we have abstracted this into a separate *is_booting*() function. More research here is being done.

- The /etc/rc.subr routines will execute a command as a special user if *_user* is defined for that command. rc.subr will issue trhe command "su -m $_user sh -c 'shell command'", which can be very useful for running commands at start up. Please take this as justification, if any more be needed, that the /etc/rc.d should be used wherever possible.

4.8 Further Development

This paper is being written during the period that the user level utilities are still being modified to perform properly in an rbac secmodel. Some of the future areas for development are:

- Adding a role for user management, including adding additional roles to users
- Files in /etc, /var, and other binaries should no longer belong to root
- The file system scope in kauth has still to be properly defined, and so the rbac secmodel is as yet unfinished in the area of file systems
- The kauth GENERIC_ISSUSER action will be left in place
- The standard root user will be left in place, for the reasons already specified
- Standard binaries will continue to be owned by root:wheel

- Third party packages need to be modified to use the rbac secmodel

5. Future Work

The RBAC work is, at the time of writing, about to be checked into the NetBSD source code repository. As can be seen above, however, the kernel modifications are the smallest piece of the work to build a full RBAC system.

Kauth is still lacking a number of scopes, and so is still reliant upon the kauth equivalent of *issuser()* - most notably, and most voluminously, is the file system scope, about which there has been a lot of discussion on the NetBSD mailing lists surrounding its implementation.

6. Other Systems and Models

We have discussed the standard Unix security model, and we also have looked at the RBAC model. An early form of RBAC was first introduced in Trusted Solaris, and from there became part of Solaris 8. Development has continued through to Solaris 10 today [Solaris2009]. AIX gained support for RBAC in AIX 6 [AIX2008].

Both AIX and Solaris have variants of RBAC security models which can be used today. It is interesting to note that the most popular proprietary forms of Unix used in enterprises are proposing RBAC security models as a possible security model to adopt.

Other security models which have not been investigated in the paper include Mandatory Access Control (yet another MAC acronym), and Discretionary Access Control (DAC). SELinux [SELinux2007] is used in the Linux kernel as a form of MAC.

The RBAC model is a lot of work to implement - whilst it is relatively easy to implement the kernel part of RBAC, most of the work is in the adaptation of individual binaries to have the correct roles. This must be done not only for all programs in the base system, but also for third-party packages. It is also a sad fact that most setuid root programs will need different group ownership, although the benefit of doing that is that the programs need no longer be setuid root - they will most likely become setgid to a role, and thereby be less of a risk to the system as a whole.

For hosts that are situated in DMZs, or other network zones where security is paramount, an RBAC security model could be an interesting deployment. Existing root kits are of little use in a security model which does not respect root as having privileges. Such sensitive machines are also likely to have environments where third-party packages are rigorously controlled, which makes them good contenders for analysing the roles necessary for such software to operate well.

7. Conclusions

This paper has talked about the Role-Based Access Control security model, its different approach to the standard Unix security model, and shown how it was implemented in the NetBSD kernel, layered on top of the kauth subsystem. The implementation of the rbac secmodel was relatively simple - most of the work in developing a new secmodel comes in adapting user level utilities to the new secmodel, and RBAC is no exception. Developing a new secmodel does, however, have some benefits - it is a great way for a

developer to get an in-depth view of the system as a whole, and how interactions between various subsystems take place. Deployment in sensitive network zones is an interesting alternative to some of the other security models available, and would be easier to achieve than a general purpose machine, since software installed on such a machine is highly controlled. RBAC security models have also been implemented in proprietary versions of Unix, used as enterprise solutions. The NetBSD RBAC kernel shows that this can also be provided in the open source world.

8. References

[AIX2008] http://www.redbooks.ibm.com/redbooks/pdfs/sg247430.pdf
[Efrat2006] http://www.netbsd.org/~elad/recent/recent06.pdf
[SELinux2007] http://hackinglinux.blogspot.com/2007/05/selinux-tutorial.html
[Solaris2009] http://www.sun.com/software/whitepapers/wp-rbac/wp-rbac.pdf
[Stallman2002] http://www.gnu.org/software/coreutils/manual/html_node/su-invocation.html