Using NFS to Implement Role-Based Access Control

Mats Gustafsson Benoit Deligny Nahid Shahmehri
Dept. of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden
Email: {matgu,e96bende,nsh}@ida.liu.se

Abstract

In this paper we present a design for a modified NFS *server that through simple additions makes it possible to place an exported file system under Role-Based Access Control.*

*Role-Based Access Control (*RBAC*) is an efficient way for managing access control information. However, most access control systems today do not support* RBAC*. As a solution to this problem, the modified* NFS *server we present here makes it possible to introduce* RBAC *into existing environments in a transparent manner.*

We have implemented our design as an extension to the Linux User Space NFS Server *running on a Linux 2.0 system. Our implementation demonstrates the feasibility of our main idea. However, tests show that performance of our server needs to be improved, something we believe can be achieved through code optimizations.*

Keywords: Role-based access control, NFS, distributed systems, legacy systems

1. Introduction

Whenever new technologies or novel designs and ideas emerge in some area there arises a problem of how to introduce these new technologies without having to entirely discard the old. In an area as rapidly evolving as computer science, this problem is accentuated. Design compromises must often be made in order to accommodate *legacy systems*, into which much money and competence are invested.

An interesting development in recent years is the interest devoted to *Role-Based Access Control* (RBAC). RBAC provides a conceptually simple model for organizing and representing access control information. Several authors have discussed various aspects of role-based access control and its relation to other paradigms for access control, e.g. [6, 11, 12, 16, 17]

The main advantages of RBAC lie in facilitated administration and better overview of security information. RBAC achieves this by providing an abstract view that more easily can reflect security policies within an organization. This abstraction uses *organizational roles* to structure the information contained in the access control database. A role corresponds to some position or function within the organization. This way of specifying access control information can reduce the conceptual gap between security policies and access control mechanisms.

However great the benefits of adapting a role-based view for expressing access control information may be, the issue of legacy systems remains. In this paper we present a design that using the Network File System (NFS), specified by Sun Microsystems Inc [21], makes it possible to introduce RBAC into distributed systems.

NFS makes files on a server available to clients over a network. In the design we present here, an NFS server is modified to use access control information from a role-based security information database. When clients access files on the server, an access control check is performed and information from the database is used to specify access attributes for the file.

1.1. Scope and paper outline

To achieve a secure system and to successfully deploy a system built to our design, several other components are necessary. An important requirement is to have an infrastructure that provides users with access to security services. Such an infrastructure allows information about roles and other security related information to be communicated securely over the network. Another issue to consider is that NFS rests upon the RPC protocol. In its original form, RPC and NFS are vulnerable to certain attacks. Examples of such attacks are client impersonation and file handle guessing (see for instance [3]). However, in this paper we focus on our main ideas, rather than trying to address problems in many areas.

We begin by introducing the NFS file system in section

2. Section 3 presents the design for our augmented NFS server. Moving from design onwards to implementation we describe in section 4 how we have used and modified an existing server to realize our ideas. In section 5 we report on some of the results and experiences from our implementation. We conclude the paper by giving a summary and some avenues for future work in section 6.

2. The Network File System

The Network File System, NFS, provides transparent access to remote file systems and was introduced by Sun Microsystems Inc. in 1985. Sun chose to make NFS an "open" protocol by publishing the specification. This has made NFS widely used and ported, also to non-Unix platforms. The first published version of NFS was called Version 2 [21]. In 1995, several vendors, including Sun, IBM, and Digital, published an extended Version 3 of the NFS protocol [5, 14].

NFS is a client/server protocol. An NFS server makes file systems available over the network in a transparent way. Clients perceive the exported file system as being a part of their own local file systems.

Important prerequisites for NFS are the *Virtual File System* (VFS) and the *Remote Procedure Call* mechanism (RPC). The VFS is an interface to an abstract file system and provides the indirection necessary in order to present a single file system to the client while there physically actually may be several. RPC, together with the *External Data Representation* standard (XDR), provides the network transport mechanism used by NFS. The VFS, RPC and XDR are all technologies introduced by Sun [8, 19, 20].

The operation where a client incorporates a remote file system into its own virtual file system is called *mount*. Once a client has mounted an NFS file system, the NFS protocol specifies a number of functions a client may use to access the server. In NFS version 2 there are 18 functions (of which one is obsolete and one is for use in later versions), whereas in version 3 the number has been extended to 22. The base server we use implements NFS version 2.

NFS services rather closely mirror file level operations available in a Unix system. However, by using a subset of the functions, or by providing suitable mappings, NFS has also been implemented for other operating systems, such as MS-DOS, Novell Netware and VMS (see for instance [2, 9, 10]).

3. Design for the RBAC NFS server

The NFS specification ([21]) does not define the permission checking to be used by servers. However, the specification mentions that *"... it is expected that a server will do normal operating system permission checking using AUTH_UNIX style authentication as the basis of its protection mechanisms."* Using this style of authentication, the server receives with each request the effective user and group identities (UID and GID respectively) of the caller. During normal operations, these identities are used when the server makes file accesses. However, the main responsibility for access control still rests with the kernel of the client that will allow/disallow access based on the file attributes of a file, as reported by the NFS server.

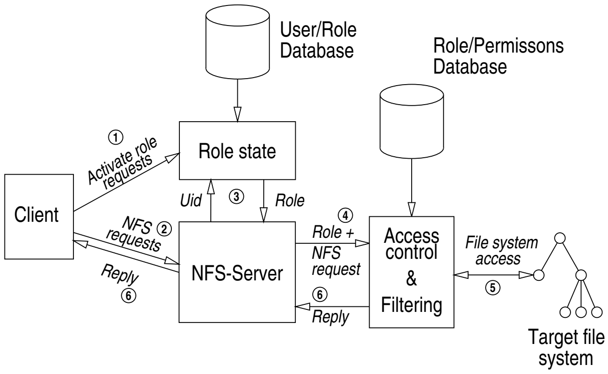

Figure 1. Main server components

Figure 1 shows the main components of our proposed NFS server implementing role-based access control. The central idea of our design is to mount the remote file system as usual but then to side-step the default permission checking behavior. The numbers in the figure show a typical flow of events as follows.

In ①, a user starts out by using a special application to *activate* one or more roles. This application communicates with the **role state** monitor module adjacent the NFS server itself. A user can only activate roles to which she has been assigned in the **User/Role database**. Information about active roles is kept in the **role state** module. As a rule, a user must independently establish an active role before she can access files through the RBAC NFS server. The exception to this rule is when access to some directory or file has been explicitly granted a user, for instance to a home directory.

After having activated a set of roles, a user application can access a file on the remotely mounted file system. When this happens, the client machine sends an NFS service request to the server ②.

When the server receives a request, it uses the UID of the calling client process as a key to acquire information about what *roles* the user has active ③.

Having retrieved the currently active roles for the user making a request, the NFS server passes this information to an access control decision function ④. To make a decision, this function uses information found in the **Role/Permission database**. This database assigns permissions to roles authorizing them to access files, directories or directory trees.

If the requested access is permitted given the set of active roles, the corresponding operation is carried out on the target file system ⑤.

In our RBAC NFS server design, owner and permission attributes stored for a file on the **Target file system** are not used. Instead, in a **filtering** step, those NFS requests that query permission attributes are intercepted and attribute values based on the information in the **Role/Permissions Database** are substituted in the reply. The result is then passed back to the client ⑥.

3.1. Considerations

Our design implies a change in the conceptual model describing the remote file system. User identity (UID) and group identity (GID) are no longer central concepts, except for conveying information that can be used to tie the UID to a role and for providing access to files that are indeed personal in nature.

However, an important design goal is to achieve a solution that requires no, or small, changes in client systems and applications. Since client programs should run unmodified, it is obvious that whatever access rules and access modes we define using the **Roles/Permissions Database**, we must ultimately map these permissions into file attributes used by the client.

In the basic Unix file system model, three basic access permissions are used, *read*, *write*, and *execute*. These permissions are used both for directories and files. Although there are only three kinds of permissions, we can identify ten basic kinds of access modes, *create/delete file*, *create/remove directory*, *read/write of file*, *execute file*, *search directory* and *create hard/symbolic link* .

Table 1 shows what permissions are necessary in order to access a file or directory in different ways. In addition to the permissions shown in the table, it is necessary to have at least *execute* permission to all directories in the path leading to the accessed file or directory.

It is the kernel of the client machine that enforces access control using the permissions assigned to a file. This is also the case when a file system is mounted over NFS, it is still the client machine that is the access control enforcer, using the permission attributes reported by the NFS server.

The granularity of Unix permissions and access modes is limited. From table 1 it can be seen that there are some access modes that only can be granted together with others. For instance, it is not possible to allow file creation without at the same time allowing creation of directories.

An access mode that often is desirable is *append file*. However, this mode is not present in the Unix model. There is also no way to express that a permission should apply to an entire directory (sub-)tree.

Access mode	Necessary permissions	
	To file	To parent dir
Create file		`-wx`
Create dir		`-wx`
Remove file	`-w-`	`-wx`
Delete dir	`-w-`	`-wx`
Read file	`r--`	`--x`
Write file	`-w-`	`--x`
Execute file	`--x`	`r-x`
Search dir		`r-x`
Make symbolic link		`-wx`
Make hard link		`-wx`

Table 1. Correspondence between access modes and necessary UNIX permissions

In the design of the RBAC NFS server, we have tried to address these shortcomings by introducing a different set of permissions. In our design, the basic permissions *read*, *write*, and *execute*, remain the same but we distinguish between access to files and access to directories. We also introduce an *append file* permission and allow permissions to apply generally for a directory subtree.

An important file access mode is the operation of changing file permissions. In our system, we use a permission database that in an independent manner assigns file access permissions to roles. Permission to modify this database is not given out by default. However, in order for it to be possible for a user to write a usable shell script or program, she should be able to set the executable permission for that file. For this reason, we include a special permission that allows this specific operation.

In our server, we implement explicitly permissions for eleven different access modes. These permissions are summarized in table 2.

Some special issues were considered when designing the RBAC NFS server. None, however, constitute any serious problem and we therefore discuss them here only briefly. The special cases we identified were

Symbolic links. No special steps need be taken in order to properly handle symbolic links. The permission attributes of a symbolic link are never used, instead access control is performed when the target the link points to is referenced. By convention, the permissions for a symbolic link is listed as "`lrwxrwxrwx`", we too adhere to this convention when returning attributes for symbolic links.

Hard links. Hard links in fact constitute an alias mechanism that allows a file to have more than one name. This can often lead to confusion. For this reason we

Operation	Mnemonic
Read file	FR
Create file	FC
Write file	FW
Append to file	FA
Delete file	FD
Execute file	FX
Create directory	DC
List directory	DL
Remove directory	DR
Toggle execute bit	XT
Create symbolic link	LC

Table 2. Permissions in the RBAC NFS server

do not allow hard links to be created, although existing links in the server file system will work correctly, as hard links per definition are indistinguishable from the file itself. We do not believe that this is a significant limitation as symbolic links can be used instead.

Device files. Device files are always created directly on the server file system using the `mknod` command, never using NFS. The handling of access to device files is no different from that of access to other files.

Set UID/GID attribute. Unix file systems offer a special *set user/group id upon execution* attribute for executable files. This is an important tool for security administration. As our server does not take into account owner or group ID of a file, the concept of *set uid/gid* becomes undefined and we do not support this attribute at present. In effect, file systems behave as if they have been mounted using the `nosuid` option.

Presenting file information. As user and group access attributes are not used, a natural approach is to only use the *others* file access attributes to convey access control information from the server to the client. This is also the default behavior in our system. An exception is made for symbolic links, by convention permission for such links are shown as "`lrwxrwxrwx`". In order for the *others* attributes to apply, it is necessary that none of the *user* or *group* attributes apply. We achieve this by presenting all files and directories as belonging to user `nobody`, group `nobody` (having UID/GID 65534).

However, upon closer examination, it turns out that only the owner of a file can toggle the *execute* bit. If the owner is set to *nobody*, the request is blocked by the kernel of the client machine and not even forwarded to the NFS server. If the effective permissions to the file include the *execute bit toggle* permission (XT), we must set the owner of a file to be the current user. We must also use the *user* position when specifying access attributes. A positive side effect of this is that there is a visible distinction between files to which XT permission is granted and files to which it is not. Figure 2 shows a typical directory listing.

```
-------r-- 1  nobody  nobody  259709 Feb 27 20:15 FILES
lrwxrwxrwx 1  nobody  nobody       8 Feb 11 16:00 INSTALL -> /var/adm/
-rw------- 1  jsmith  nobody   27954 Feb 27 20:14 README
-------r-- 1  nobody  nobody   24541 Feb 27 20:16 RELNOTES
lrwxrwxrwx 1  nobody  nobody      14 Feb 11 16:00 linux -> /usr/src/linux/
drwx------ 1  jsmith  nobody    1024 Feb 11 16:45 mail/
-------rw- 1  nobody  nobody    2265 Feb 15 17:17 test.txt
-------r-x 1  nobody  nobody     375 Feb 16 16:47 fixit
```

Figure 2. Typical directory list appearance

Specifying RBAC permissions. Permissions are assigned to roles in order to express an *access policy* for some system resource. In the case of the RBAC NFS server, the system resources in question are directories, directory trees, and files

For simplicity, the current implementation uses a text file for specifying permissions to access files and directories assigned to roles. Figure 3 shows a simple permissions file using the mnemonics specified in table 2. **everyone** is a pseudo role that is always active. When determining the actual permissions for a role to a path, the most specific prefix of the path found in the file for that role is used.

```
#
# The '#' character introduces a comment stretching to the
# end of line.
#
# General syntax is   path  role  permissions [; role permissions]+ #
/    sysadm FR:FC:FW:FD:FX:DC:DL:DR:XT:LC     # Give all rights to sysadm
/          *everyone* DL                      # Let everyone browse FS
/sbin      *everyone*                         # Deny rights
/usr/sbin  *everyone*                         # Deny rights
/usr/apps/dbms/audit.log  manager F=RCAD; clerk   F=CA
/usr/apps/dbms            manager F=RX;   clerk   F=RX
/home/jsmith              USER:jsmith F=CRWDX:D=CLR:XT:LC
```

Figure 3. Sample RBAC permission database

4. Implementation

The implementation was made using a publicly available NFS server as a starting point. The server software is in common use on Linux systems [18]. Our augmented server currently runs on a Pentium PC running Linux 2.0.29 and is accessed over a standard 10 MBit/s Ethernet network (for an introduction to Linux, see for instance [1, 4, 22]).

As can be seen from figure 1, two main additions have been made to the original NFS server. One deals with the assignment of users to roles and the management involved when users activate and deactivate roles. The other addition is responsible for implementing the new, role-based, access control semantics as described in section 3.

4.1. Implementing roles

In order to use the concept of role, there must be a way to *define* roles, it must be possible to *assign* roles to individuals, and there has to be a mechanism whereby roles may be *activated*. As stated initially, these tasks would normally be handled by an underlying security infrastructure. However, for the purpose of demonstration, we have implemented a client program that allows a user to communicate with a role state server adjacent to the NFS server. Using the client program, the user can activate and deactivate roles according to the assignments made in the **User/Role database**. This database is a simple text file that for each role lists which users may activate it.

4.2. Introducing RBAC semantics

By definition, any NFS server is built around the functions implementing the services defined in the protocol. What we have done is to introduce a two tier enforcement of RBAC in the server. The first tier delegates access control to the client machine by assigning suitable file attributes to referenced files. In many cases this is sufficient in order for the client machine itself to enforce access control. In the second tier, a check is made upon each service request to determine whether performing the request is allowed or not given the set of active roles for the current user.

In the original server, the first step taken when a service function is called is to authenticate the file handle, the calling machine, and the calling user. To this sequence we have inserted code that establishes a set of *active roles* for the duration of the call.

In the function that reads file attributes we have inserted a filter that clears out the access mode bits and substitutes a mode derived from combining together (using bitwise or) the *necessary access modes* (see table 2) corresponding to each permission granted for the file to each active role. We also preserve the original value of the execute bit.

In general, for calls to the NFS server that entails reading or modification of a file or directory, access is only permitted after a check for necessary permissions. A few calls (to read file attributes, read directory contents and read symbolic links) are permitted without check.

Overall, the changes made to the original server are small. About 500, out of a total of circa 13000, lines of code were added or modified.

5. Results

We have tested our implementation on two Linux machines communicating with each other. One of the machines is a 200MHz Pentium processor workstation while the other is an older laptop machine with a 66MHz 486DX2 processor.

Server version	Command	**Server running on** Pentium PC	486 PC
Original	`ls -lR`	22.33	46.0
Modified	`ls -lR`	24.98	1:49.0
Increase		12%	236%
Original	`tar`	24.89	1:02.7
Modified	`tar`	37.87	3:30.0
Increase		52%	335%

Table 3. Execution times for `ls -lR` and `tar` commands

Some simple experiments were carried out to assess the impact of our modifications on performance. Two identical directory trees were created on each of the machines. The trees contain a total of 2876 files and 115 directories. The builtin *time* command of *tcsh* was used to time two different commands performed on the directory tree. The first command was a full recursive listing of all files using the Unix command "`ls -lR`". To execute this command, the attributes for every file must be fetched from the server. The second command read circa 6 Mbyte of data from the directory tree using the GNU *tar* command. In both cases the output was discarded by directing it to `/dev/null`. The operations were carried out using both the unmodified and the modified server, and in both possible directions. To make sure the servers did not use any information cashed from previous calls, the servers were restarted and remounted before running each command.

Table 3 shows the results from both experiments. The values shown are computed means from three test runs and show measured total execution time.

From the results we see that our modifications incur a rather heavy performance penalty, especially when the server is running on the machine with a slower processor. Even though the values shown for the faster machine seem to be acceptable, it is obvious that our rather straight forward approach to implementation needs further refinement. As our modifications to the original server were not written primarily with performance in mind, we believe performance can be improved through code optimizations and use of caching and in-memory databases.

6. Conclusion and future work

In this paper we have shown how NFS, an existing technology, can be used to introduce an interesting access control paradigm, role-based access control, in existing systems. By introducing the proposed design, both administrative and practical problems can be solved. Administratively, RBAC has important merits as it contributes to narrow the conceptual gap between security policies and security measures. A more down to earth benefit of our design is that it solves the dilemma that often occurs when there are two user groups that both should have access to the same resource. Using traditional group access semantics this situation cannot be directly modelled, whereas it is very easy to do so using RBAC by assigning the same permission to two different roles.

As we have seen in section 5, the implementation needs to be improved with respect to performance. Use of caching and in-memory databases are obvious enhancements not present in our demonstration implementation. A closer integration between our modifications and the original server is also likely to lead to further performance gains.

However, at this stage, performance has not been the main issue. Instead, we have demonstrated by example our main idea, that RBAC can be implemented using NFS. If further developed, we believe strongly that the ideas presented here can be used to enhance security in many existing systems. As mentioned initially, a first step in this direction would be to integrate the demonstration system we have presented here with an infrastructure for distributed security. This would for instance include structured storage for roles and access control data (see for instance [7, 15]). One framework upon which an infrastructure can be built that we are looking into is provided by SESAME [13].

Before deploying our server it is also necessary to address vulnerabilities found in the transport used by NFS. For instance, strong authentication should be used at the RPC or IP level.

7. Acknowledgments

This work has been supported by the ECSEL graduate school at Linköping University and by the Swedish Foundation for Strategic Research.

References

[1] The Linux Home Page. World Wide Web page. URL: http://www.linux.org/.

[2] Attachmate. PathWay Server NFS for OpenVMS. Product information, World Wide Web document. http://www.attachmate.com/products/ServerNFS-VMS.html.

[3] J. Barkley, editor. *Security in Open Systems*, volume SP–800–7 of *NIST Special Publications*. United States National Institute of Standards and Technology (NIST), Dec 1994. URL: http://csrc.ncsl.nist.gov/nistpubs/.

[4] S. N. Bokhari. The Linux operating system. *Computer*, 28(8):77–9, Aug 1995.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, June 1995.

[6] D. F. Ferraiolo, J. A. Cugini, and D. R. Kuhn. Role-Based Access Control (RBAC): Features and Motivations. In *11th Annual Computer Security Applications, Proceedings*, 1995.

[7] M. Gustafsson and N. Shahmehri. A Role Description Framework and its Applications to Role-Based Access Control. In *NORDSEC '96 – Nordic Workshop on Secure Computer Systems*, Gothenburg, Sweden, Nov 1996. SIG Security/Dept. of Computer Engineering, Chalmers University of Technology.

[8] S. R. Kleiman. Vnodes: An Architecture for Multiple File Systems Types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Conference*, pages 238–247, Jun 1986.

[9] S. Microsystmes. Solstice Network Client – PC-NFS^{TM} 5.1. Product information, World Wide Web document. http://www.sun.com/sunsoft/solstice/Networking-products/PC-NFS51.html.

[10] Novell, Inc. IntranetWare NFS ServicesIntranetWare and NetWare 4 Edition: Executive Summary. World Wide Web document. http://www.novell.com/catalog/qr/sne34210.html.

[11] M. Nyanchama and S. Osborn. Role-Based Security, Object Oriented Databases & Separation of Duty. *SIGMOD RECORD*, 22(4):45–51, Dec 1993.

[12] M. Nyanchama and S. Osborn. Access Rights Administration in Role-Based Security Systems. In J. Biskup, M. Morgenstern, and C. E. Landwehr, editors, *Database security, VIII*, pages 37–56. IFIP, North-Holland, 1994.

[13] T. Parker and D. Pinkas. SESAME V4 – OVERVIEW. World Wide Web document, Dec 1995. URL: http://www.esat.kuleuven.ac.be/cosic/sesame/doc-ps.html.

[14] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proceedings Summer 1994 USENIC Conference*, pages 137–151, Boston, 1994.

[15] K. Rappe. Roles and Role Management in Role-Based Access Control – Model, design and implementation. Master's thesis, Linköping University, Dept. of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, December 1996.

[16] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control: A Multi-Dimensional View. In *Proceedings of the 10th Annual Computer Security Applications Conference*, Orlando, Florida, Dec 5–9 1994.

[17] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb 1996.

[18] M. Shand, D. Becker, R. Sladkey, O. Zborowski, F. van Kempen, and O. Kirch. The LINUX User-Space NFS Server(1), Version 2.2. ftp://ftp.mathematik.th-darmstadt.de /pub/linux/okir/nfs-server-2.2.tar.gz, December 1995.

[19] Sun Microsystems, Inc. XDR: External Data Representation Standard. RFC 1014, June 1987.

[20] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1057, June 1988.

[21] Sun Microsystems, Inc. NFS: Network File System Protocol Specification. RFC 1094, March 1989.

[22] T. Yager. Linux matters. *BYTE*, 21(2):123–4,126–8, Feb 1996.