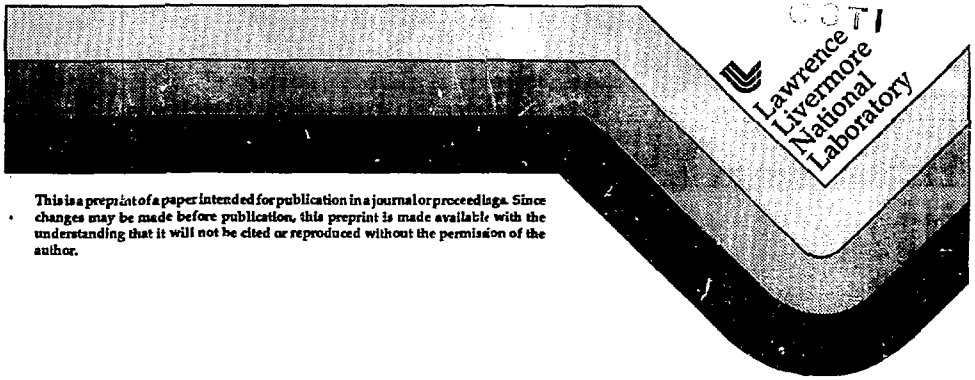
UCRL-JC-112301
PREPRINT

Practical Authorization in Large Heterogeneous Distributed Systems

J.G. Fletcher
D.M. Nessett

This paper was prepared for submittal to
PSRG Workshop on Network and Distributed System Security
San Diego, CA
February 11-12, 1993

November 1992



Practical Authorization in Large Heterogeneous Distributed Systems

J.G. Fletcher and D.M. Nessett

Lawrence Livermore National Laboratory
Livermore, CA

ABSTRACT

Requirements for access control, especially authorization, in practical computing environments are listed and discussed. These are used as the basis for a critique of existing access control mechanisms, which are found to present difficulties. A new mechanism, free of many of these difficulties, is then described and critiqued.

INTRODUCTION

Over the past decade and a half, system researchers have thoroughly investigated distributed computing, analyzing its important issues and proposing various ways of treating them. However, the services they have developed sometimes poorly fit the problems arising in practical computing environments. We concentrate on how this is so for distributed access control.

Access control is implemented through two component services : 1) authentication and 2) authorization. The problem of authentication has received significant attention and we believe the mechanisms developed so far are adequate in most situations. Consequently, we concentrate here on distributed system authorization, a problem requiring more attention.

This paper is organized as follows. First, we analyze the characteristics of certain practical distributed computing environments and develop requirements for distributed system access control. We use these requirements to critique existing distributed system access control mechanisms, particularly those aspects related to authorization. We describe an authorization method that meets our criticisms, pointing out its strengths and weaknesses and providing a compromise containment analysis for it. We then describe a production application that uses our authorization scheme.

ACCESS CONTROL IN PRACTICAL COMPUTING ENVIRONMENTS

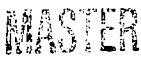
Researchers interested in distributed system security have extensively investigated the issue of access control. For the most part, they have concentrated on the problem of authentication, while on the whole limiting their investigations of authorization to the smaller sub field of distributed operating systems. With a few exceptions, architects of distributed systems other than distributed operating systems have relied on the existing non-distributed mechanisms of hosts to support authorization.

We believe that much of the previous work on distributed system authorization rests on assumptions that only rarely exist in practice. To support this claim, we analyze the characteristics of a typical distributed system supporting scientific and engineering applications and in section 3 discuss how existing distributed system access control techniques fail to operate correctly in the presence of these characteristics. While it would be appropriate to do so, we do not analyze systems that are used primarily for business applications, since we have little experience with them. However, our intuition suggests that many of the characteristics we describe are relevant for those systems as well.

The security environment of a distributed system supporting science and engineering

There are two classes of distributed application that use security services. The first class supports system level activity that is generally administered by system programmers and carried out to supply infrastructure services to distributed system customers. The second class involves computational activity initiated by non-privileged users, generally focused on solving some scientific, engineering or other customer related problem. These two classes of application possess contrasting security traits. Applications in the first class enjoy extraordinary security privileges, such as root access. Applications in the second class generally are not granted special security privileges.

Distributed applications supporting scientific or engineering work are initiated by customers rather than by system software or system programmers. Thus, they are an example of the second class of distributed application. They customarily grow from a central point and expand out into a distributed system. As with most distributed applications, their activity is organized around the client/server model. However, it is rare for the servers of these applications to exist prior to the initiation of an application run. Instead, servers are dynamically created when the application grows and are terminated when the application finishes. This pattern of behavior strongly influences which access control mechanisms are suitable for such applications. Generally, there must be an unprivileged server that permanently runs on hosts and that allows the creation of dynamic servers running in the context of a distributed application user. It is



the permanent server that makes the appropriate distributed system authorization decisions.

There is a very large investment in programs that analyze various scientific and engineering research problems. These programs use linear system solvers, implicit and explicit difference equation solvers and relaxation methods to solve partial differential and integral equations. It is far too expensive to rewrite this software for a particular distributed application. Instead, a distributed application must be able to incorporate this software without modification. Consequently, scientific and engineering distributed applications are not at liberty to change the way these programs do file I/O, terminal I/O or graphics I/O. While it is possible to write driver routines that call these programs and handle communications with other distributed application components, the underlying system service calls must not be disturbed.

Heterogeneity is an important characteristic of practical distributed systems [1, 2]. We are amazed at the number of designs that ignore this pivotal concern. Heterogeneity exists in the physical security environment of distributed system equipment, in the behavior of the organizations that administer this equipment, in the protocols used within a distributed system, in the level of vulnerability each host operating system experiences, and in the security mechanisms supported by hosts[1].

Previous work has dealt with security heterogeneity by organizing collections of similarly trusted hosts into pools known variously as Domains of Trust [3, 4], Authentication Domains [5], Inter-Organization Networks [6], Realms [7,8], and Administrative Domains [2]. Within these domains, security mechanisms may also display a certain amount of heterogeneity[2]. For example, a domain may support the Kerberos authentication mechanism [7, 8] on some hosts, while others may rely on the normal UNIX /etc/passwd file mechanism. Even within hosts, some applications may support Kerberos authentication (e.g., *rlogin*, *rcp*, *rexec*), while others may rely on /etc/passwd (e.g., Telnet, FTP).

Customer initiated distributed applications face considerable difficulties when run over resources located in multiple security domains. They do not have special privileges and therefore must use infrastructure security services provided by the domains. While there are authentication facilities available to accommodate multiple security domains [7, 8, 9], existing authorization mechanisms require either transmitting a user's password in the clear over a potentially hostile network, or the installation of software, such as a Kerberized or DASS-enhanced *rexec* daemon, that requires root privilege. Generally, system administrators are reluctant to install software provided by customers that require root access. Consequently, if systems on which the non-privileged distributed applications execute do not support the appropriate root privileged software, customers are forced to use dubious security practices, such as storing their passwords in files and passing them in the clear through vulnerable intermediate computing and switching equipment. A practical distributed system authorization method should eliminate these security hazards.

Most current distributed computing is what might best be described as network computing. Generally, hosts in the distributed system act as independent computing agents that retain a significant identity from an application's standpoint. While distributed operating systems may provide a more coherent and an ultimately superior performing base for distributed applications, so far, they have not been highly successful in the marketplace. Our own distributed operating system, LINCS [4, 10], failed not for technical reasons, but rather because we could not afford to support it as a unique LLNL specific product. Nothing is currently available from computer system vendors that provides its functionality.

Our experience with LINCS leads us to conclude that network computing will remain the predominant distributed computing model for some time to come. This means that distributed system support must be built on top of existing host operating systems, which today are largely some variation of UNIX™.

Given the ubiquity of UNIX, we are forced to consider its security properties. Most fielded UNIX operating system implementations contain significant security hazards. Moreover, there are few if any mainstream UNIX operating systems for state-of-the-art computing equipment evaluated according to the Trusted Computer System Evaluation Criteria [11][3]. We don't have much confidence in the idea that this situation is about to change. Consequently, we believe any distributed system security mechanism must operate in an environment in which the constituent hosts have intrinsic vulnerabilities. To be more precise, we believe that when a host compromise occurs, the security mechanisms should be architected to minimize the number of compromised resources and provide some kind of compromise containment support. Along these same lines, when system administrators discover a misbehaving user, they should be able to quickly and efficiently revoke that user's privileges to distributed system resources.

---

[1] Some may reject our thesis that a distributed system experiences heterogeneity in host security mechanisms, since we postulate the pervasive use of UNIX. However, variants of UNIX do not all support exactly the same security mechanism. For example, many versions of UNIX allow any user to obtain the contents of the etc/passwd file, while others hide its contents from public view.

[2] The work described in [w] argues against this practice. The definition of Administrative Domain given there insists that all constituent hosts use the same security mechanisms.

[3] Even if there were, we don't have a high regard for such evaluations, since they do not raise our confidence adequately to justify their cost. Furthermore, once evaluated systems are placed in the field, many of their handling constraints, such as the prohibition against customer operating system modifications, are impractical. We have other criticisms of the whole concept of evaluated systems, but this is a topic for another paper.

Requirements for distributed system access control mechanisms

We use the characteristics described above to develop requirements for distributed system access control. Specifically:

1) Access control facilities must not require existing scientific program modules and equations solvers to be modified. If these programs access stand-alone system resources, such as files, terminals, graphics equipment, etc., they must be able to do so in exactly the same way when they are integrated into a distributed application.

2) The support of customer initiated scientific distributed applications requires that the access control mechanisms operate without root privileges.

3) Distributed system access control must operate on systems running Unix.

4) Distributed system access control must operate in an environment of vulnerable hosts. When a host is compromised, the access control software must not allow the intruder to compromise the complete distributed system.

5) When system administrators discover a misbehaving user, the access control mechanisms must allow them quickly and efficiently to revoke his access to distributed system resources.

6) Access control facilities must not encourage users to engage in unsound practices such as storing unencrypted passwords in files or transmitting them in the clear over networks.

7) Access control must operate in a heterogeneous environment. It must work across multiple domains that may support different underlying access control methods.

A CRITIQUE OF EXISTING ACCESS CONTROL MECHANISMS

We investigate some popular distributed system access control mechanisms either in use or proposed to determine whether they meet our requirements. While our focus is authorization, some of our requirements are affected by the authentication service used for access control, so we briefly analyze several authentication schemes from this perspective. We concentrate on Kerberos [7, 8], DASS [9] and /etc/passwd based authentication.

Most distributed system access control schemes can incorporate any of the authentication mechanisms named above. However, /etc/passwd based authentication requires the transmission of a password in the clear from the client to the server, which violates requirement 6. Both Kerberos and DASS support authentication without transmitting cleartext passwords, so these authentication strategies are preferable for our applications.

Existing distributed system authorization mechanisms fall into one of two categories : 1) access control list based, or 2) capability based. Most distributed operating systems that have been developed so far have used capabilities. However, the majority of distributed system software used in practical computing environments uses access control lists, so in this critique we focus on that technology.

Access control list systems also fall into two categories : 1) those that hold the access list information in a file or database on each machine (per machine database authorization), or 2) those that hold all or part of this information on centralized servers (centralized database authorization). The most common approach to distributed authorization uses the authorization information maintained by host operating systems, which is a per machine database strategy.

Systems that use a centralized database for authorization data include Moira [8], the proxy-based ticket approach developed for Kerberos [12] and the DCE authorization mechanism [13]. The Moira approach, developed for Project Athena, keeps authorization information on a centralized server. This information is distributed to individual servers on a periodic basis. Servers use this data to make authorization decisions after a user has been authenticated by Kerberos.

The proxy-based ticket approach is based on the use of Kerberos tickets that are passed between principals. An authorization server, to which servers grant full access rights, creates restricted proxy tickets for principals according to authorization information it retains. Within the ticket may be information that restricts its use in some way. A principal proves its has obtained the ticket in a legitimate manner by carrying out a protocol with a server that uses the session key the ticket contains. This key is passed between principals when the ticket is passed.

Systems running DCE software from Open Systems Foundation authenticate the user using a Kerberos protocol exchange, the established identity being used for authorization decisions. DCE also supports a registry service that maintains the set of groups to which a principal belongs. This information is sealed in a Privilege Attribute Certificate and passed from client to server in support of authorization. Each DCE server is configured with DCE's access control list software that maintains full access lists for each resource. These lists contain entries that identify a user, a group and other information along with permission data for these identifiers. Since an access control list can contain multiple

entries, more fine grained control is supported than can be achieved with standard Unix permission bits. Moreover, a proposal to support access rights delegation is currently being studied as an enhancement to this scheme.

Layered authorization

Independent of where the access control information is stored, distributed system authorization services may be implemented in one of two ways. The first approach layers the distributed authorization mechanism over existing host authorization services. The second assumes all distributed system resources are managed and owned by servers, which multiplex their use among the server's clients.

Currently, most fielded authorization systems rely on the access list mechanisms supplied by host software. For example, a host authenticates a user through a service such as Kerberos, DASS, or by use of its own /etc/passwd file and from this obtains a local user identifier (uid). Then the authorization mechanism changes the security context of the executing process through the setuid system call, using the uid as input.

Layering distributed system access control over existing host authorization services allows program components such as existing system solvers to access stand-alone system resources without modifying their code. Thus, a layered approach satisfies requirement 1.

However, most layered authorization schemes require the software supplying distributed access control services to run as root. Thus, requirement 2 is not met by these approaches. Below we describe a layered authorization technique that does not use root privileges.

The layered approach meets requirement 3, since it utilizes distributed system authorization on each machine and we assume hosts support some variant of Unix. Its resistance to host compromise rests principally on the resistance of the authentication mechanism to this threat. Kerberos and DASS authentication mechanisms are relatively robust in the face of host compromise. Users that directly enter their Kerberos or DASS passwords on compromised machines are themselves compromised. The proxy-based ticket approach has the additional vulnerability that servers on compromised hosts possessing forwardable tickets allow them to be compromised. However, in a large distributed system these compromises give the intruder access to a small proportion of the total distributed system resources. Compromise of the Kerberos authentication and TGT servers compromises the whole distributed system, but these are special systems that may be strongly protected using high-grade physical and operational protection strategies. The use of /etc/passwd authentication is also fairly robust when a host is compromised, since users entering their passwords for other hosts are compromised on those hosts, but generally this does not compromise the whole distributed system. Consequently, requirement 4 is met by most of the popular authentication mechanisms.

If the authentication mechanism allows the quick removal of users from its databases, which is true for Kerberos and DASS, then requirement 5 is met. However, if the /etc/passwd mechanism is used, quick revocation is unlikely, especially in a large distributed system.

As specified above, only layered authorization mechanisms that rely on Kerberos or DASS satisfy requirement 6. Those that rely on /etc/passwd authentication fail in this regard.

Requirement 7 generally isn't met by most layered authorization schemes, because they do not interoperate with each other. For example, a user operating under an /etc/passwd based scheme cannot access resources in other domains controlled under a Kerberos based scheme. While there is an effort underway to harmonize Kerberos and DASS authentication, such a facility still will not interoperate with an /etc/passwd based facility.

Server-centric authorization

It is possible to design a distributed system authorization mechanism that does not rely on the authorization mechanism of hosts. Specifically, resources on the underlying machine can be owned and managed by a server, which multiplexes them among its clients (server-centric authorization).

Server-centric authorization doesn't meet requirement 1, since access to distributed system resources occurs not through system calls, but rather through server requests. This implies that existing libraries and programs must be modified to use resources managed by distributed system servers.

However, server-centric authorization does satisfy requirement 2. Servers multiplex access to stand-alone system resources, relying on the host operating system authorization mechanism to grant them access to the resources they own. This does not require root access privileges. Furthermore, this approach will operate on any Unix operating system, so requirement 3 is satisfied.

The compromise of one host may or may not compromise other distributed system hosts depending on how the authorization mechanism operates. It is possible to devise a server-centric authorization method that has good compromise containment properties. For example, the LINCS distributed operating system used the server-centric approach for its Unix guest file server. Since files were accessed through capabilities, the compromise of one host only compromised those files with capabilities on that host.

If the server-centric authorization mechanism relies on an appropriate authentication mechanism, such as Kerberos or DASS, then system administrators can quickly revoke a misbehaving user's access control rights. Consequently, requirement 5 can be met.

This approach gives the access control architect the flexibility to create a mechanism that does not encourage the user to engage in unsound security practices. For example, LINCS guest file server capabilities can be protected against both forgery and theft.

Finally, server-centric authorization can easily be made to work in a heterogeneous environment, since the difference in access control mechanisms are hidden by the server implementation. In effect, each server acts as an access control gateway, translating from the distributed system access control mechanism into the access control mechanism used by the host. Of course, if the server-centric mechanism is to operate between domains that use different authentication schemes, such as Kerberos or DASS, then either the servers must be instrumented to handle all such authentication mechanisms or there must be authentication gateways that translate from one scheme to another. This last approach is being taken in the effort to harmonize DASS and Kerberos.

Critique summary

Both the layered and server-centric approaches to authorization present difficulties when used in large practical distributed systems. Server-centric authorization imposes burdens on existing software, requiring it to be reimplemented for use in distributed applications. Most schemes that layer distributed system authorization on host authorization require servers to run at root and do not adequately cope with heterogeneity.

In the next section a layered authorization mechanisms is described that does not require root privileges and that accommodates heterogeneity by supporting several different authentication mechanisms concurrently. This is done in such a way that it also presents good compromise containment properties.

A PRACTICAL AUTHORIZATION SCHEME

The authorization technique described here is used by *Remoxe*, a remote execution service for Unix developed and in use at the Lawrence Livermore National Laboratory. A Remoxe server executes as a daemon on each computer where the service is provided. A client process on any computer can send to a server (generally on a different computer) a message asking that it execute some application. The client and the application may then communicate either through the server (in which case the application thinks that it is dealing with a controlling terminal) or directly (using sockets). The executing application has access to a *context* chosen by the client, where a context consists of the resources available to a particular user on the service computer. This choice of what constitutes a context is dictated by the nature of typical Unix systems; it could readily be modified for systems with other forms of local access control (such as capability-based systems). The access to a context is authorized without a password having to be typed.

Access lists and capabilities are frequently described as alternative means for authorizing access to resources. However, particularly in a distributed environment, the techniques are often complementary and are used together. For example, consider conventional remote access using such facilities as *telnet* or *ftp*. Access lists on the service (remote) computer (typically in the rather coarse-grained form of *owner*, *group*, and *world* access permissions) are used in connection with a user name and password provided from the client (local) computer. The user name and password together effectively constitute a capability, a coded record that establishes the client's relationship to the access lists (by defining and verifying the owner's identity).

Remoxe makes use of a capability we call a *xap* (execution access protector, pronounced "zap"). It is a coded record that is originally generated by the server and sent to the client to be stored until needed. It is sent back to the server as a parameter in messages requesting remote execution or other action. It identifies and authorizes access to a context and includes the following information:

- the TCP/IP address of the Remoxe server for which the xap is valid,
- the local user name associated with the context on the service computer,
- permission bits,
- authentication information (e.g., a GSSAPI global name), and
- a DES encryption key for the local password associated with the user name.

One permission bit enables remote execution; the others enable various "housekeeping" actions in regard to the xaps themselves, such as issuing additional ones or revoking existing ones.

A xap should be kept in a safe place, such as in a file accessible only by the user (owner) on a client computer that has a secure operating system. This last condition especially is difficult to meet for all too many Unix systems. So there may be a problem of xaps being stolen, that is, illicitly copied. The purpose of the authentication information is essentially to provide a degree of protection against the theft of a xap by limiting the effectiveness of the xap to situations in which additional information is also supplied, authenticating that the client has the right to use the

xap. Each xap employs one of three authentication options (listed in order of increasing security):

- No authentication is required. So there is no protection against theft: a purloined xap may be used by the thief (or anyone else) from any client computer. This option is provided only as a last resort for situations (we hope that there are none) where the other options are infeasible or (would that it were so!) where there is no danger of theft.

- The use of the xap is limited to a particular client computer (more precisely, a particular client IP address). The thief cannot hide himself in a distant part of the network while h.. misuses the xap. This option is provided for use where the necessary infrastructure for the next option is not available.

- The use of the xap is limited to the user with a particular global name as defined by an authentication system based on GSSAPI (namely, Kerberos or DASS). The xap must be accompanied by the evidence (context token) required by that system for establishing that the user has that name, and the degree of protection depends on how secure that system is. This is the preferred option.

The 64-bit local password encryption key appears in a xap exclusive-or'd with a DES cryptographic digest computed using all the other information in the xap and a master key that is known only to the server. The xap thereby not only conceals the encryption key, but also is protected against forgery. Anyone trying to generate a xap (either out of whole cloth or by altering a few bits, such as permission bits, in a valid xap) has only one chance in 2^{64} of correctly rendering the encryption key (effectively only one in 2^{55} because of the way DES uses keys). When a user first establishes himself with the server, at which time he must supply his local user name and password for the service computer (but not a xap) in a secure manner, the server randomly generates an encryption key just for that user. The key is then used to encrypt the user's local password. The server stores the encrypted password (in association with the local user name) in its records with sufficient redundancy that it can with high confidence recognize an improperly decrypted password before attempting to use it. The server remembers *neither* the unencrypted password *nor* the password encryption key, but it includes the latter in a xap which it issues to the new user (Fig. 1).

Therefore the server can obtain the password only when a client provides a valid xap. This means that compromise of a user's password requires "breaking into" *both* the user's records on a client computer *and* the server's records on the service computer. It is our view that such is an obstacle sufficient to render Remoxe acceptably secure.

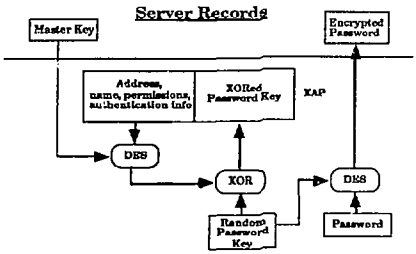


Figure 1. Concealment of password, using xap.

There are "housekeeping" chores i dealing with xaps. Remoxe provides for establishing a new user with the server and issuing the initial xap, for issuing additional xaps that may have reduced permissions and/or differing authentication information (in particular, allowing access from a different client computer), for changing the password on the service computer (both as known to Unix and as known to the server), for revoking all the user's existing xaps (by changing the password encryption key) and issuing a new xap to replace them, and for deleting the user from the server's records (which of course also effectively revokes all existing xaps). Note that, since the content of a xap does not depend on what the password is, changing the password does *not* affect the validity of existing xaps.

It is possible for each user to have a Remoxe server of his own, running on a service computer with its own "well known" TCP port (that is, a fixed port number that can be "built into" client programs). However, the user then assumes the burden of installing his server and assuring that it is always up and running. Also, there is an inefficiency in having many separate servers on a computer, all performing basically the same job. So the intent is that there be only a few Remoxe servers (often just one) on each service computer, each installed and maintained by a single user, its *sponsor*. This user owns and could access the files in which the server keeps its records. He therefore must be someone who can be trusted by the other users not to abuse his position and invade their privacy by either misusing the records himself or through carelessness letting them be accessed by others. That is, each server and its sponsor corresponds to a community of trust. The sponsor could be the "superuser", but we have not required this, because we want a user to be able to install and begin using Remoxe without waiting for an administrative bureaucracy to give its approval and then take action. The server's records are kept in files in a subdirectory of its sponsor's home directory; this directory is created when the sponsor installs the Remoxe server. The records are accessible only by the sponsor, who is their owner.

When a client requests the running of an application, it sends the server a xap accompanied by any necessary authentication information. The server verifies the authentication information (e.g., by calling the appropriate GSSAPI procedures) and also decrypts and verifies the password in the xap (Fig. 2). It then forks a process but does not directly "exec" the application. Instead, it "execs" the standard privileged Unix utility *su* and delivers the user's name and password to it as input. After *su* has converted the forked process into a shell associated with the user's id, additional input effects the "exec" of the desired application. In this way the application runs with the environment and access rights of the user, rather than with those of the sponsor.

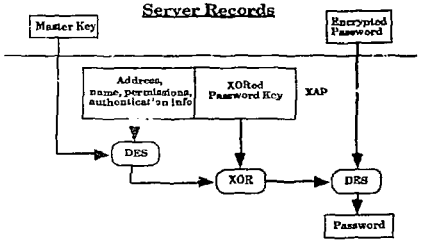


Figure 2. Recovery of password, using xap.

This rather roundabout, complex, and no doubt inefficient technique for establishing the proper context is necessitated by the peculiarities of Unix. Perhaps the designers of future operating systems will consider the following suggestions:

- The natural way for a program to interface to the operating system is through a privileged procedure (system call), rather than by forking and "exec'ing". So there should be a privileged procedure to which one can pass a user name and password (or other system specific access control information) and which will then set the user id to that of the user. Similarly, there should be a privileged procedure for changing a password.

- In fact, there should be a simple, direct way without administrative intervention for any user to establish a service and for other users to be able to grant that service such access to their resources as they choose. They should not have to grant this access in an "all or nothing" fashion, but should be able to adhere to the principle of least privilege. (A capability-based system would achieve this.)

An analysis of remoxe authorization

We briefly analyze the advantages and disadvantages of the Remoxe authorization method and then provide a crude compromise containment analysis. Servers, such as Remoxe, that utilize our authorization technique can be brought up *immediately on any* machine on which the user(s) have accounts. No system administrator help or approval is necessary. If the distributed system hosts are configured to accept a global password for local authentication, such as a Kerberos or DASS password, users need only remember one. This reduces the possibility of password compromise and the inconvenience associated with reissuing passwords that the user has forgotten. The Remoxe authorization scheme provides these advantages along with security that is at least as good, if not better, than other approaches.

One disadvantage of the Remoxe authorization scheme stems from one of its advantages, the lack of involvement of a system administrator when bringing up *permanent servers*. Without root privilege or system administrator help, keeping these servers up across system crashes is problematical. Normally, permanent servers are brought up at reboot based on an entry in the rc.local file. Since this file is owned by root, this technique is not available to the unprivileged user.

We are exploring use of the "At" utility to overcome this problem. In particular, a non-privileged server or a "persistence daemon" can periodically call "At" to schedule a check that the appropriate servers are still running. If the check fails, the server can be restarted. However, this approach requires that the activity scheduled by "At" survive across system crashes. This may or may not be true, depending on the particular variation of Unix on which the server runs.

The Remoxe authorization method has quite good compromise containment properties. If an intruder gains access to the files in which the Remoxe server master key and encrypted passwords are stored, these cannot be used, since the intruder lacks the Xaps that contain the keys to the encrypted passwords. If the intruder obtains a Xap, either by reading it from an improperly protected file on a client machine or by capturing it as it travels over a network, it cannot be used (assuming the GSSAPI authentication option is employed) since the intruder cannot manufacture the necessary time-limited credentials required by the supporting authentication technology.

If an intruder obtains root access on the server machine, he has compromised all resources on it. He need not take advantage of servers using the Remoxe authorization scheme. However, if he manages to compromise a sponsor, he gains access to the resources of all other users that have *entrusted their access rights to the server the sponsor* controls. This is a good reason for supporting several servers on a machine, one for each community of interest that runs there.

If an intruder obtains root access on a client machine, he can wait for its users to type their global (i.e., GSSAPI-based authentication) passwords and thereby compromise their resources in the distributed system. This would be true

whether Xaps are stored on the machine or not. Consequently, the Remoxe authorization scheme does not introduce any new vulnerabilities for this situation. In fact, since the intruder may not have access to all the users Xaps, i.e., those stored on other systems, the Remoxe scheme potentially can lessen the damage caused by a client machine compromise.

USE OF REMOXE

The need for Remoxe originally arose from the following typical situation: A user has a number of source files that he maintains and edits on his workstation, which offers him convenience, economy, and high interactivity. However, many of the sources are intended to be compiled and executed on a supercomputer, which offers power. After editing, the user transports the updated sources to the supercomputer and there compiles and executes them. He would like to have the required updating occur automatically in response to a single, simple typed command, such as "make".

The standard Unix utility *make*, used in conjunction with standard utilities that provide remote access, such as *ftp*, *telnet*, and *rsh*, would seem to provide the required facility. That is, *make* would invoke *ftp* to transfer the sources to the supercomputers and then invoke *telnet* or *rsh* to execute remotely the compiler, other utilities, and the compiled applications. However, there are two difficulties:

- *Make* makes decisions based on the exit status of the programs that it runs, which is not available for programs run remotely by *telnet* or *rsh*.

- Each time that *make* invokes a utility providing remote access, that utility prompts and waits for the input of a password, severely inconveniencing the user and requiring his continued attendance at the terminal; it would be difficult to view the activity as truly automated. Common means of circumventing this problem, such as identifying oneself as "anonymous" or "guest", making appropriate entries in the .netrc or .rhost files, and/or using the Network File System (NFS), open up privacy or security loopholes that are often unacceptable. These remarks also apply to use of the standard utility *rdist*.

A client utility has been provided for use with Remoxe that avoids both of these problems. In regard to the first problem, the protocol between the client and server is such that status information is returned after each remote execution; this status is in turn returned as the status of the client. To effect execution on a supercomputer, *make* invokes the client utility, which then (through the remote server) invokes the remote application. *Make* will then correctly interpret the returned status as that of the application.

There is no need for passwords, because authorization is effected using xaps appropriate to the remote servers. (In fact, the xaps specify which remote servers — which remote computers — are to be used.) These xaps are fetched from files specified in the commands to the client. The client also carries out any needed GSSAPI-based protocol. In addition, a file transfer utility has been provided to be run under the control of the server. Files may be transported between this utility and the client. A sample commented makefile for remote execution is displayed in Fig. 3. Note that *make* invokes a second, remote *make*.

```
#   This makefile effects the remote compilation and
# execution of the application "test", which tests a subroutine
# package.

#   The directory "shadow" contains empty shadow files,
# one for each file that is to be sent to the remote computer.
# These files "stand in" for their remote counterparts when
# "make" tests their ages.  After a file is sent, its shadow is
# aged by using "touch".
#   The client utility is "clnt"; "-f sc" specifies that the xap
# is to be found in the file "sc", and "-d dir" specifies that
# remote execution is to occur in the directory "dir".  The
# remote commands are "make" and "test", the former
# referring to a remote makefile that should effect the
# compilation of the three ".c" and ".h" files into the
# executable file "test".

update: shadow/test.c shadow/subrs.c shadow/subrs.h
        clnt -f sc -d dir make
        clnt -f sc -d dir test

#   Before the remote "make" is invoked, any updated ".c"
# and ".h" files are transported using the remote utility
# "rmx", which interprets "put xxx.x" as a request to have
# the file "xxx.x" sent from the client to the remote computer.

shadow/test.c: test.c
        clnt -f sc -d dir rmx put test.c
        touch shadow/test.c

shadow/subrs.c: subrs.c
        clnt -f sc -d dir rmx put subrs.c
        touch shadow/subrs.c

shadow/subrs.h: subrs.h
        clnt -f sc -d dir rmx put subrs.h
        touch shadow/subrs.h
```

Figure 3. A sample simple makefile.

ACKNOWLEDGMENTS

"Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract number W-7405-ENG-48."

BIBLIOGRAPHY

1. D. M. Nessett, "Factors affecting distributed system security," IEEE Transactions on Software Engineering, vol. SE-13, no. 2, Feb., 1987, pp. 233-248.

2. D. M. Nessett and G. M. Lee, "Terminal services in heterogeneous distributed systems," Computer Networks and ISDN Systems, Vo. 19, pp. 105-128, 1990, Elsevier Science Publishers B.V., Amsterdam, The Netherlands.

3. J.G. Fletcher, "Software Protection of Information Networks," Infotec State-of-the-Art Report, Future Network, vol. 2, 1978, pp. 149-164.

4. R.W. Watson and J.G. Fletcher, "An Architecture for Support of Network Operating System Services," Computer Networks, vol. 4, Feb., 1980, Elsevier Science Publishers B.V., Amsterdam, The Netherlands, pp. 33-49.

5. D.M. Nessett, "The Inter-Authentication Domain (IAD) logon protocol (Preliminary specification and implementation guide)," Lawrence Livermore Nat. Lab. Rep. UCID-30207, 1984.

6. D. Estrin, "Non-Discretionary Controls for Inter-Organization Networks," Proc. IEEE Symposium on Security and Privacy, IEEE, Los Alamitos, CA., April, 1985, pp. 56-61.

7. J.G. Steiner, C. Neuman and J.I. Schiller, "Kerberos: An authentication Service for Open Network Systems," Proc. Winter Usenix Conf., Usenix Association, Berkeley, CA., 1988, pp.191-202.

8. S.P. Miller, B.C. Neuman, J.I. Schiller, and J.H. Saltzer, "Kerberos Authentication and Authorization System," section E.2.1 of *Project Athena Technical Plan*, MIT, Dec. 1987.

9. J. Linn, "Practical Authentication for Distributed Computing," Proc. IEEE Symposium on Research in Security and Privacy, IEEE, Los Alamitos, CA., May, 1990, pp. 31-40.

10. J.G. Fletcher and R.W. Watson, "Service Support in a Network Operating System," CompCon 80, Spring, 1980.

11. "Department of Defense Trusted Compter System Evaluation Criteria," DOD 5200.28-STD, Department of Defense, Washington, DC, December, 1985.

12. B. Clifford Neuman, "Proxy-Based Authorization and Accounting for Distributed Systems," University of Washington Technical Report 91-02-01, Department of Computer Science and Engineering, University of Washington, FR-35, Seattle, Washington.

13. "OSF DCE 1.0 Application Development Guide; Revision 1," Dec. 27, 1991, Open Software Foundation, Cambridge, MA.