Conclusion

Recall that our goal was a system that is as secure as time–sharing. We feel we have met this goal. The way you are authenticated in a time–sharing system is by knowing your password. In our system, the same is true. In time–sharing the person you trust is your system administrater, who does not do anything dirty such as change your *passwd* entry so they can impersonate you. In our system, you instead trust your network administrater who does not change your entry in the public key database. In one sense, our system is even more secure than time–sharing. In our system, it is unfruitful to place a tap on the network in the hopes of catching a password or encryption key, because we encrypt them. Most time–sharing environments do not encrypt the data emanating from the terminal; users must trust that nobody is tapping their terminal lines.

DES authentication is not the end–all authentication system for Sun. It is likely that in the future there will be sufficient advances in algorithms and hardware to render the public key system as we have defined it useless. The nice thing about DES authentication is that there is a smooth migration path for it in the future. Syntactically speaking, nothing in the protocol requires the encryption of the conversation key to be Diffie–Hellman, or even public key encryption in general. To make the authentication stronger in the future, all that needs to be done is to strengthen the way the conversation key is encrypted. Semantically though, this will be a different protocol, but the beauty of RPC is that it can be plugged in and live peacefully with the older authentication systems.

But for the present at least, DES authentication satisfies our requirements for a secure networking environment. From it, we are able to build a system secure enough for use in unfriendly networks, such as for example a student–run university workstation environment. The price for this security is not high. Nobody has to carry around a magnetic card or remember any hundred digit numbers. You use your login password to authenticate yourself, just as you did before. There is a small impact on performance, but if this worries you and
you have a friendly net, you can merely turn the authentication off.

Acknowledgements

David Goldberg was the chief designer of DES authentication as well as how it fits in with the Sun environment. Bradley Taylor did most of the implementation, some of the design and even wrote this paper. We would like to thank Bob Lyon for putting authentication at the RPC level where it belongs, and for bringing RPC to Sun in the first place.

References

Diffie and Hellman, ”New Directions in Cryptography”, *IEEE Transactions on Information Theory IT–22*, November 1976.

Gusella & Zatti, ”TEMPO: A Network Time Controller for a Distributed Berkeley Unix System”, *USENIX 1984 Summer Conference Proceedings*, June 1984.

National Bureau of Standards, ”Data Encryption Standard”, *Federal Information Processing Standards Publication 46*, January 15, 1977.

Needham & Schroeder, ”Using Encryption for Authentication in Large Networks of Computers”, *Xerox Corporation CSL–78–4*, September 1978.

server decrypts it, server encrypts reply timestamp and client decrypts it. On a Sun–3, the time it takes to encrypt one block is about half a millisecond if performed by hardware and 1.2 milliseconds by software. So, the extra time added to the round trip time is about 2 milliseconds for hardware encryption and 5 for software. The round trip time for the average NFS request is about 20 milliseconds, resulting in a performance hit of 10 percent if one has encryption hardware, and 25 percent if not. Remember that is the impact on *network* performance. The fact is that not all file operations go over the wire, so the impact on total system performance will actually be lower than this. It is also important to remember that security is optional, so environments that don't require it may turn it off if they want higher performance.

Problems: Booting and Set–uid Programs

Consider the problem of a machine rebooting, say after a power failure at some strange hour when nobody is around. All of the secret keys that were stored away get wiped out, and now no process will be able to access secure network services, such as trying to mount an NFS filesystem. The important processes at this time are usually root processes, and so if only *root's* secret key was stored away, then things would work okay, but nobody is around to type the password that decrypts it. One solution to this problem is to store the root password in a file, which the keyserver then reads to decrypt the secret key. This works fine for diskful machines which can store the root password on a physically secure local disk. It won't work for diskless machines though, since the root password must be sent in the clear over the net. Another solution is to mount all the filesystems read–only at boot–time, and then wait for the root user to return. After the root password has been entered, the filesystems can be remounted with full access rights. We will break all of the programs that depend upon writing files at this time, most notably administration and mail. The programs will have to be fixed on a per application basis. For example, mail can be sent to server machines instead of clients. As an aside, note that storing mail on the server has the windfall benefit of allowing one to read mail from several different machines, further reducing the workstation to user binding.

Another booting problem is the single–user boot. There is a mode of booting known as single–user mode, where one is given a *root* login shell on the console. The problem here is that one is not prompted for a password when attempting to do this. To fix this, in the future a password will be required in order to boot single–user. There will still be a way to set up your machine so that you can do single–user booting without a password, but you had better have physical security for your machine if you're going to turn this feature off.

Yet another problem with booting is that for diskless machines it will not be totally secure. It is possible for somebody to impersonate the boot–server, and boot you a devious kernel that, for example, makes a record of your secret key on a remote machine. The problem is that our system is set up to provide protection only after the kernel and the keyserver are running. Before that, there is no way to authenticate the replies given by the boot server. We don't consider this a serious problem because it is highly unlikely that somebody would be able to write this funny kernel without source code. Also, the crime is not without evidence. If you polled the net for boot–servers, you would discover the devious boot–server's location.

Set–uid programs will not behave as they should in all cases. If a set–uid program is owned by *dave*, and *dave* has not logged into the machine since it booted, then the program will not be able to access any secure network services as *dave*. The good news is that most set–uid programs are owned by *root*, and since *root's* secret key is always stored at boot time, these programs will behave as they always have.

the Yellow Pages. If you log into somebody else's workstation and type in your password, then your secret key would be stored in their workstation and they could use *su* to impersonate you. But this is not a problem since you should not be giving away your password to a machine you don't trust anyway. Someone could just as easily change *login* to save all the passwords it sees into a file.

Not having *su* to rely on anymore, how is one to impersonate others in the new system? Probably the easiest way is to guess somebody's password, since most people don't choose very secure passwords. We offer no protection against this; it is your own problem if you choose an insecure password. The next way would be to attempt replays. For example, let's say that I have been squirreling away all of your NFS transactions with a particular server. As long as the server remains up, I won't succeed by replaying them since the server always demands timestamps that are greater than the previous ones seen. But now suppose I go and pull the plug on your server, causing it to crash. As it reboots, its credential table will be clean, and so it has lost all track of previously seen timestamps and now I am free to replay your transactions. There are few things to be said about this. First of all, servers should be kept in a secure place so that no one like myself will go and pull the plug on them. But even if they are physically secure, servers still occasionally crash without any help. Replaying transactions is not a very big security problem, but even so, there is protection against it. If a client specifies a window size that is smaller than the time that it takes a server to reboot (5 to 10 minutes), then the server will reject any replayed transactions because they will have expired. There are other ways to break DES authentication, but they are much more difficult. These methods involve breaking the DES key itself, or computing the logarithm of the public key. But it is important to keep our goals in mind. We are not aiming at having super–secure computing. What we're after is something that is as good as a time–sharing system, and in that end, we have been successful.

There is another security issue that DES authentication does not address, and that is tapping of the net. Even with DES authentication in place, there is no protection against somebody merely watching what goes across the net. This is not a big problem for most things, such as the NFS, since very few files are not publically readable, and besides, trying to make sense of all the bits flying over the net is not a trivial task. For logins, this is a bit of a problem because you wouldn't want somebody to pick up your password over the net. For this reason, the new remote login program *slogin* will encrypt all data (though there will be a way to turn this feature off). As we mentioned before, a side effect of the authentication system is a key exchange, so that the network tapping problem can be tackled on a per application basis.

Performance

Public key systems are known to be slow, but there is not much actual public key encryption going on in our system. Public key encryption only occurs in the first transaction with a service, and even then, there is caching that speeds things up considerably. The first time a client program contacts a server, both it and the server will have to calculate the common key. The time it takes to compute the common key is basically the time it takes to compute an exponential modulo M. On a Sun–3 using a 192–bit modulus, this takes roughly 1 second, which means it takes 2 seconds just to get things started since both client and server have to perform this operation. This is a long time, but it is only the very first time you contact a machine that you'll have to wait. Since the keyserver caches the results of previous computations, it does not have to recompute the exponential every time.

The most important service in terms of performance is the NFS, but at the time of this writing, we have not built the secure NFS so there is not much to be said about it. What we can give you are estimates. The extra overhead that DES authentication creates versus Unix authentication is most likely the encryption. A timestamp is a 64–bit quantity, which also happens to be the DES block size. Four encryption operations take place in an average RPC transaction: client encrypts the request timestamp,

to type in a password. But it is not that easy to tell which machines a given machine trusts, as you have to go recursively through the *hosts.equiv* file to determine this. It is important to emphasize that anybody able to change their uid on one of these trusted machines can login using your account, without typing a password. The fundamental problem with *rlogin* though is it should not be basing its security on machines, but rather on users as that is the way the *passwd* file is set up. If a machine trusts you, you should be able to login to it from any other machine, without having to send your password in the clear. *slogin* will correct these problems because the authenticated entity will be the netname instead of the machine–name. You should never be forced to type your password since you have already been authenticated. However, if you don't type your password, then the remote machine will not be able to decrypt your secret key, and you will not be able to access the secure network services from the remote machine. To correct this problem, there will be an option to *slogin* to force the remote machine to prompt you for a password. There will also be an option to *su* to prompt you for your password and set your secret key accordingly, in case you logged in without typing a password.

The other application that we have planned is the most important: the Network File System. There are three security problems with the current NFS using Unix authentication. The first is that verification of credentials occurs only at mount time when the client gets from the server a piece of information that is its key to all further requests called the *file handle*. Security can be broken if one can figure out a file handle without contacting the server, perhaps by tapping into the net or by guessing. After one has mounted an NFS file system, there is no checking of credentials during file requests and this brings up the second problem: if one has mounted a file system from a server that serves many clients (as is typically the case), then there is no protection against someone who has sovereignty over their machine using *su* (or some other means of changing the uid) to gain unauthorized access to other people's files. The third problem with the NFS is the severe method it uses to circumvent the problem of not being able to authenticate remote client superusers, that is, denying them superuser access altogether.

The new authentication system will correct all of these problems. Guessing file handles is no longer a problem since in order to gain unauthorized access, the miscreant will also have guess the right encrypted timestamp to place in the credential, which is a virtually impossible task. The problem of authenticating root users is solved since we can now authenticate machines. The NFS server will associate with each file system that it serves a root netname which is granted full access to the filesystem. This is very important for diskless booting, so that an NFS server serving root filesystems will know which machine is associated with each root filesystem.

Actually, the level of security associated with each filesystem will be a user settable item. The file */etc/exports* currently contains a list of file systems and which machines may mount them. It will be changed in the future so that one can also specify the security required for accessing the filesystem. Unix authentication will be the default, but DES authentication can be selected as well. Associated with DES authentication is one parameter: the maximum window size that the server is willing to accept. There will also be a level of security where the actual file data is encrypted, if one is willing to accept the severe performance penalty for doing this.

Security Issues

There are several ways to break DES authentication, but we should make it clear first that using *su* is not one of them. The reason is the following. In order to be authenticated, your secret key must be stored by your workstation. This usually occurs when you login, with the login program decrypting your secret key with your login password and storing it away for you. If somebody tries to use *su* to impersonate you, it won't work because they won't be able to decrypt your secret key. Editing */etc/passwd* isn't going to help them out either, because the thing that they need to edit, your encrypted secret key, is stored in

The old Unix authentication system has a few problems when it comes to naming. Recall that with Unix authentication, the name of a network entity is basically the uid. These uids are assigned per Yellow Pages naming domain, which typically spans several machines. We have already stated one problem with this system, that it is too UNIX oriented, but there are two other problems as well. One is the problem of uid clashes when domains are linked together. The other problem is that the superuser (uid 0) should not be assigned on a per domain basis, but rather on a per machine basis. The NFS deals with this latter problem in a severe manner: it does not allow superuser access over the network by uid 0 at all.

DES authentication corrects these problems by basing naming upon new names that we call *netnames*. Simply put, a netname is just a string of printable characters and fundamentally, it is really these netnames that we authenticate. The public and secret keys are stored on a per netname, rather than per username, basis. There is also a Yellow Pages map that maps the netname into a local uid and group–access–list, though other non–Sun environments may map the netname into something else.

The internet naming problem is solved by choosing the netnames to be globally unique. This is a far easier task then choosing globally unique uids. In the Sun environment, user names are assigned per Yellow Page domain. The convention we use for assigning netnames is to concatenate the username with the domain name. For example, a user named *dave* in the domain *discovery* would have the netname *dave@discovery*. If domain names are uniquely chosen within the internet, then this scheme will work to produce a unique netname for each user in the internet. A good convention for naming domains would be to append the ARPA domain name to the local domain name. Thus, the domain *discovery* within the ARPA domain *sun.com* would be renamed *discovery.sun.com*.

We solve the multiple superusers per domain problem by assigning netnames to machines as well as to users. A machine's netname is formed in a similar manner to a user's. Example: *dave's* machine has the netname *hal!root@discovery.sun.com.* Authenticating machines is a very important capability for diskless machines that need full access to their root filesystem over the net.

Other non–Sun environments will have other ways of generating netnames, but this does not preclude them from accessing the secure network services of the Sun environment. To authenticate users from any remote domain, all that has to be done is make entries for them in two Yellow Pages databases. One is an entry for their public and secret keys, the other is for their local uid and group–access–list mapping. Upon doing this, users in the remote domain will be able access all of the local network services, such as the NFS and remote logins.

Applications of DES Authentication

We have only developed one application using DES authentication at the time of this writing, but we have two more planned. The first application we wrote was a generalized Yellow Pages update service. This service allows users to update their private fields in the Yellow Pages databases, for example, their login password or mail alias. Without the update service, we currently have a specialized daemon to update login passwords and hire a full–time person just to update the mail aliases! There are other applications for Yellow Pages updating, such as changing your login shell or adding a new machine to the host tables. No doubt many more applications will emerge.

Another application we have planned is a secure login program which we call *slogin. slogin* is aimed at being a replacement for *rlogin*, which suffers from some important security problems. *rlogin* operates in two modes: trusted and untrusted. In untrusted mode, the remote machine prompts you for a password, but then you are forced to send your password in the clear over the net. If the machine has an entry for your machine in its *hosts.equiv* file, then you can login in trusted mode, and there is no need

The particular public key encryption scheme we use is the Diffie–Hellman method. The way it works is you generate a *secret key* SK_A at random and compute a *public key* PK_A using the following formula:

$$PK_A = \alpha^{SK_A} \quad (\alpha \text{ is a well–known constant})$$

You store PK_A in a public directory, but keep SK_A a secret. Now, if I've done an analogous thing to generate SK_B and PK_B, you can figure out our common K_{AB} as follows:

$$K_{AB} = PK_B^{SK_A} = (\alpha^{SK_B})^{SK_A} = \alpha^{(SK_A SK_B)}$$

Without knowing your secret key, I can calculate the same K_{AB} in a different way as follows:

$$K_{AB} = PK_A^{SK_B} = (\alpha^{SK_A})^{SK_B} = \alpha^{(SK_A SK_B)}$$

Notice that nobody else but the two of us can calculate K_{AB}, since doing so requires knowing either my secret key or yours. All of this arithmetic is actually computed modulo *M*, which is another well–known constant. It would seem at first that one could guess your secret key by taking the logarithm of your public one, but we choose *M* to be so large as to make this a computationally infeasible task. To be secure, K_{AB} will have too many bits to be used as a DES key, so what we do is extract 56 bits from it to be used as the DES key.

Both the public and the secret keys are stored in a public database indexed by netname, but the secret key is DES encrypted with your login password. When you login to a machine, the login program grabs your encrypted secret key, decrypts it with your login password, and gives it to a secure local keyserver to save for use in future RPC transactions. Note that ordinary users do not have to be aware of their public and secret keys. In addition to changing your login password, the *passwd* program will randomly generate a new public/secret key pair as well.

The keyserver is an RPC service local to each machine that performs all of the public key operations, of which there are only three. They are:

setsecretkey(secretkey)
encryptsessionkey(servername, deskey)
decryptsessionkey(clientname, deskey)

setsecretkey tells the keyserver to store away your secret key (SK_A) for future use and is normally called by *login*. **encryptsessionkey** is called by the client program to generate the encrypted conversation key that is passed in the first RPC transaction to a server. The keyserver looks up **servername**'s public key and combines it with the client's secret key (set up by a previous **setsecretkey** call) to generate the key that encrypts **deskey**. The server asks the keyserver to decrypt the conversation key by calling **decryptsessionkey.** Note that implicit in these procedures is the name of caller, who must be authenticated in some manner. The keyserver cannot use DES authentication to do this since it would create deadlock. The keyserver solves this problem by storing the secret keys by uid, and only granting requests to local root processes. The client process then executes a set–uid process, owned by root, which makes the request on the part of the client, telling the keyserver the real uid of the client. Ideally, the three operations described above would be system calls, and the kernel would talk to the keyserver directly, instead of executing the set–uid program.

Naming

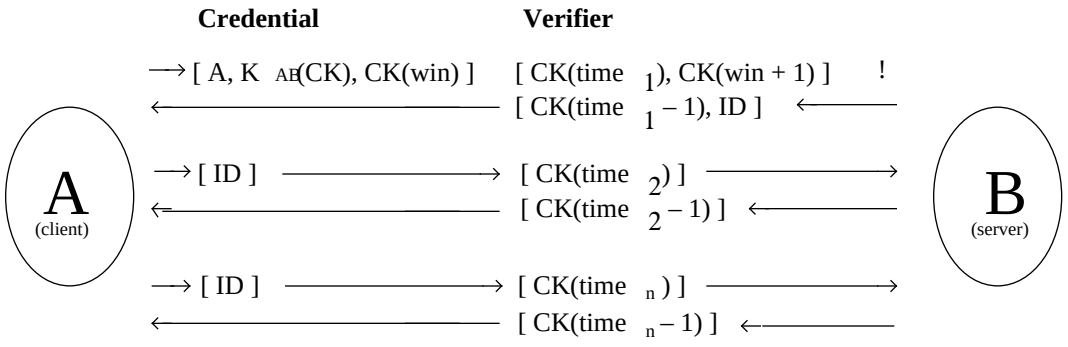

Figure 1: DES Authentication Protocol

Figure 1 above illustrates the authentication protocol in more detail, describing a client named *A* talking to server *B*. A term of the form *K(X)* means *X* encrypted with the DES key *K*. Examining the table, you can see that for its first request, the client's credential contains three things: its name A, the conversation key CK encrypted with the common key K_{AB} and a thing called the *window* encrypted with CK. What the window says, in effect, to the server is this:

I will be sending you many credentials in the future, but there may be fiends sending them too, trying to impersonate me with bogus timestamps. When you receive a timestamp, check to see if your current time is somewhere between the timestamp and the timestamp plus the window. If it's not, please be so kind as to reject the credential.

The client's verifier in the first request contains the encrypted timestamp and an encrypted verifier of the specified window, win + 1. The reason this exists is the following. Suppose somebody wanted to impersonate A by writing a program that instead of filling in the encrypted fields of the credential and verifier, just stuffs in random bits. The server will decrypt CK into some random DES key, and use it to decrypt the window and the timestamp. These will just end up to be random numbers. After a few thousand trials, there is a good chance that the random window/timestamp pair will pass the authentication system. The window verifier makes guessing the right credential much more difficult.

After authenticating the client, the server stores four things into a credential table: the client's name A, the conversation key CK, the window, and the timestamp. The reason the server stores the first three things should be clear: it needs them for future use. The reason for storing the timestamp as well is to protect against replays. The server will only accept timestamps that are chronologically greater than the last one seen, so any replayed transactions are guaranteed to be rejected. The server returns to the client in its verifier an index *ID* into its credential table, plus the client's timestamp minus one, encrypted by CK. The client knows that only the server could have sent such a verifier, since only the server knows what timestamp the client sent. The reason for subtracting one from it is to insure that it is invalid and cannot be reused as a client verifier.

The first transaction is rather complicated, but after this things go very smoothly. The client just sends its ID and an encrypted timestamp to the server, and the server sends back the client's timestamp minus one, encrypted by CK.

Public Key Encryption

ended: a variety of authentication systems may be plugged into it and coexist in the network. Currently, we have two: *Unix* and *DES*. DES authentication is the new system which is the main focus of this paper, while Unix authentication is the older, weaker system. Two terms are used when speaking of any RPC authentication system: *credentials* and *verifiers*. Using ID badges as an example, the credential is what identifies a person: his or her name, address, birthdate, etc. The verifier is the photo attached to the badge, so that one can be sure that the badge has not been stolen by checking the photo on the badge against the person carrying it. In RPC, things are much the same. The client process sends both a credential and a verifier to the server with each RPC request. The server sends back only a verifier, since the client already knows the server's credentials.

Unix authentication is the authentication system used by most of Sun's network services today. The credentials contain the client's machine–name, uid, gid and group–access–list. The verifier contains nothing. There are two problems with this system. The glaring one is the empty verifier! It is easy to use *hostname* and *su* to cook up just the right credential. If you trust all of your root users in the network, then this is not really a problem. But many times this is an unrealistic assumption. The NFS tries to combat the deficiencies in Unix authentication by checking the source internet address of mount request as a verifier of the hostname field, and accepting requests only from privileged internet ports. Still, it is not difficult to circumvent these measures, and NFS has no way of verifying the uid field.

The other problem with Unix authentication appears right in its name, *Unix*. It is another unrealistic assumption that all of the machines in your network will be UNIX machines. The NFS in fact works on MS–DOS and VMS machines, but Unix authentication breaks down considerably when applied to them. Given these two shortcomings, it is clear what one would desire in a new authentication system: operating system independent credentials and secure verifiers. This is the essence of DES authentication.

DES Authentication

The security of DES authentication is based upon a sender's ability to encrypt the current time, which the receiver can then decrypt and check against its own clock. The method used to encrypt this timestamp is DES (Data Encryption Standard). Two things must be true in order for this scheme to work: (1) the two agents must agree on what the current time is, and (2) the sender and receiver must be using the same encryption key.

If one has a network time synchronization, such as Berkeley's TEMPO, then client/server time synchonization is not a problem as it is performed automatically. However, if this does not exist, the timestamps can all be computed using the server's time instead of network time. In order to do this, the client asks the server for the time before starting the RPC session and computes the time difference between his own clock and the server's. This information is then used to offset the client's clock when computing timestamps. If the client and server's clocks get out of sync to the point where the server begins rejecting the client's requests, the DES authentication system will just resynchronize with the server again.

We now describe how the client and server arrive at the same encryption key. When a client wishes to talk to a server, it generates at random a key to be used for encrypting the timestamps (among other things). This key is known as the *conversation key*. The client encrypts the conversation key using a public key scheme and sends it to the server in its first transaction. This key is the only thing that is ever encrypted using public key cryptography. The particular scheme we use will be described further on in this document. Suffice it to say for now that for any two agents A and B, there is a DES key K_{AB} which only A and B can deduce. This key is referred to from here on as the *common key*.

Secure Networking in the Sun Environment

Bradley Taylor
David Goldberg
Sun Microsystems, Inc.

ABSTRACT

We present an authentication system that greatly improves the security of Sun's network environment. The system uses DES encryption and public key cryptography to authenticate both users and machines in the network. The system is general enough to be used by other UNIX† and non–UNIX systems.

Introduction

Sun's Remote Procedure Call (RPC) mechanism has proved to be a very powerful primitive for building network services. The most well–known of these services is the Network File System (NFS), a service that provides transparent file–sharing between heterogenous machine architectures and operating systems. The NFS is not without its shortcomings, however. Currently, an NFS server authenticates a file request by authenticating the *machine* making the request, not the *user*. On a system accessing files with the NFS, it is a simple matter of running the *su* command to impersonate the rightful owner of the file. But the security weaknesses of the NFS are nothing new. The familiar command *rlogin* is subject to exactly the same attacks as the NFS because it uses the same kind of authentication.

A common solution to network security problems is to leave the solution up to each application. A far better solution is to put the authentication at the RPC level. The result is a standard authentication system that can be used by all RPC–based applications, such as the NFS, secure logins and the Yellow Pages (a name–lookup service).

Our system allows us to authenticate users as well as machines. The advantage of this is that it gives the network environment an appearance similar to the familiar time–sharing environment. Users are not tied to machines for security purposes, just as they are not tied to terminals. They can login to any machine, and their login password is their passport to network security. No knowledge of the underlying authentication system is required. Our goal is a system that is as secure and easy to use as a time–sharing system.

Our assumptions are that any machine is capable of injecting arbitrary data into the network and picking up any data on it. We also assume that no machine is capable of packet smashing, that is, capturing packets before they reach their destination, changing the contents, and then sending it back on its original course. The attacks we consider dangerous are those involving the injection of data, such as trying to impersonate somebody by generating the right packets, or recording conversations and then replaying them later. We do not worry about passive eavesdroppers who merely listen to the network traffic, but are unable to impersonate anybody by doing so.

RPC Authentication

RPC is at the core of our security system, and in order to understand the big picture, we must first delve down to this level and understand how authentication works in RPC. RPC's authentication is open–

† UNIX is a trademark of AT&T