by Peter Collinson, Hillside Systems

JANE MARINSKY

Those 'r' Commands

As often happens when I sit down to write this monthly missive, last month's column started off in one direction and ended up wandering into somewhat different territory. About halfway through the composition process, I changed the title of the piece to more accurately reflect the contents of the emergent article. In case you missed it, the column looked at `rlogin` and how it works (see "The rlogin Program," December 1998, Page 22). My original idea was sparked by email from Mark Hale (of the Georgia Institute of Technology) who suggested that I talk about another of those "r" commands, `rsh`. Well, to be more precise, he suggested that I discuss the authentication mechanism that underlies the "r" commands: `rlogin`, `rsh` and `rcp`. Let's look at the `rsh` command first, so we can understand the problems it solves and those that are solved by the authentication system.

The `rsh` command allows you to run an arbitrary command remotely on another machine. To find out who is logged into a remote machine, for example, you can say:

```
$ rsh machine who
```

The `rsh` command looks up the name of the machine in the name service to get an IP address and connects to the appropriate remote machine. On the remote machine, a daemon is started to handle the connection. The daemon is called `in.rshd` on SunOS and Solaris. (In this article, I'm going to stick to BSD practice and ignore Sun Microsystems Inc.'s renaming of this daemon. I will call it `rshd`.) The daemon is passed the command (`who` in the above example) and runs it. The `who` command will generate output on its standard output channel. This data stream is captured by the remote `rshd` and sent back to the local `rsh` program so it can be displayed on your screen.

The intermachine connections between `rsh` and `rshd` are TCP/IP connections. The `rsh` program is designed to handle programs that take data streams as input and write data streams as output. It's really intended for use with noninteractive programs, or pipelines of commands that process data. If you want to run an interactive program on a remote system, then you need to use `rlogin`.

Within these constraints, the `rsh` program tries to follow the maxim that I stated last month: It wants to work with no surprises. The `rsh` program attempts to arrange things so that typing the command, preceded by `rsh` and a machine name, will behave in exactly the same fashion as if the command was being run on the local machine.

Working with no surprises means we need to provide more than one data stream from the remote machine to the local machine. A UNIX command can write data to its standard output channel and emit error messages to its standard error channel. The channel split needs to be reflected back on the local machine that's running rsh because the user may use the standard UNIX shell redirection mechanisms to divert the standard channels to separate destinations.

To handle a UNIX command, we need to provide a standard input channel that runs from the user, across the network and into the remote command. We need to provide a standard output channel that takes data from the remote command and sends it back to the local rsh program. These two channels can share the same TCP/IP connection. We now have the problem of handling the additional error channel that's needed to move error responses from the remote process to the local rsh.

One option might be to multiplex the data across the TCP/IP connection. We will tag each piece of data that is being sent with a channel value and decode the tagging information before writing it to the appropriate local destination. Tagging the information means that we are creating a protocol. I suspect that when the system was designed it was felt that multiplexing information was fraught with recovery and sequencing problems. It's easier to provide another connection. So the first piece of information the rsh program sends to rshd is the port number of the TCP/IP connection on which it expects to receive error messages.

Context

The who command is a good initial example because it simply interrogates the contents of log files residing in a known place in the file system and prints the result. But what about a common command like ls? What should the following command do?

```
$ rsh machine ls
```

Well, we'll make the same connection to the remote machine and run the command. Again, it will generate output on the standard output channel and send it back to the invoking system for printing on the screen. However, UNIX makes considerable use of implied context when commands are executed. Each command is run in a particular directory, the current working directory, and so when rshd is run it needs to be placed in the file system where it will cause minimum surprise to the user. By default, the rshd program will change into the user's home directory, so the above command will always list the file names in the user's home directory on the remote machine.

Incidentally, if you send the ls command to a remote machine, you may get a surprise. The output from the ls command is displayed one file per line and not the column output you see when typing the command on the local machine. This is because the ls command looks to see if it is talking to a terminal and will only generate column output when it is sure it's doing so. When running remotely using rsh, the ls command sends output to a network connection and will default to its standard behavior of sending one file per line. Some people feel that creating a program that behaves differently depending on its environment is a bad thing.

Finding the user's home directory implies that the rshd that is started on the remote machine knows who the caller is–that is, who is running the command–and, therefore, its password file can be interrogated to obtain the home directory information. Actually, I've also ignored the context of the command itself. When you type a command like ls, the shell has to find the binary of the ls program to be able to run the command. The PATH environment variable is used as a search list to locate this binary. If you use csh or one of its derivatives, then the path variable is used to determine where the command lives. Either way, we will need to run the user's shell before we can run the command because the PATH/path variables are usually established in the shell start-up files for the user.

Sometimes you'll find that the contents of shell start-up files inhibit the correct use of the "r" commands. It's especially important not to place commands that establish terminal state into the shell start-up file that's run whenever a shell is initiated. They should be put into the file that's executed when you log in and nowhere else. Coding your start-up files so that they distinguish between interactive and noninteractive access can have a positive effect on the speed at which rsh commands are executed.

Checking Identity

Knowing the identity of the caller is also a good thing for security reasons. We want to impose the same security model on any command that a user may issue from a remote machine that we apply to commands issued locally. It's not much use having controls on whether I can look at a specific file if I can just go to a remote machine and execute a command that will circumvent the security.

By default, the rsh command sends your local login name to the rshd process. For flexibility, two names are sent: the name on the local machine and the name on the remote machine. Solaris 2.6 documentation detailing the order of sending these names is incorrect. It says the names are sent in remote machine, local machine order. However, on my BSDI system, where I can look at the source, the name on the local machine is sent first. Nevertheless, the systems interwork. Using the snoop program proves that the Sun documentation is incorrect.

The two names are sent as null terminated text strings. Two names are used because there are legitimate situations where you may wish to execute a command on a remote machine as "someone else." It's usual to have the same user name across a site, but there are situations where this is not the case. One common scenario is when you are superuser on one machine and wish to execute a command using your "real" user identity on another. Most of the "r" commands have a -l (*ell* not *one*) option that allows you to specify the user you wish to be on the remote machine. If this switch is not given to the command, the remote name is set to the same value as the local name.

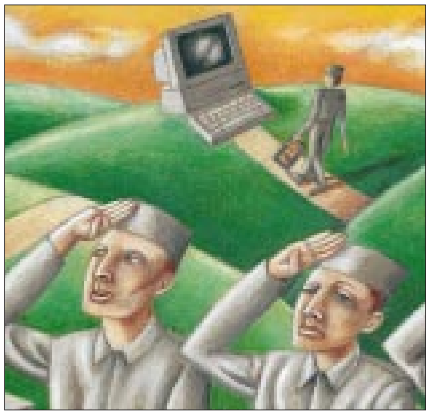
You'll notice there is no provision for sending a password. The remote machine believes what you tell it. The original thinking on the security aspects went like this: A UNIX program can be trusted to send the correct information because we program it appropriately. However, we need a way to ensure that the standard UNIX program is being run and not some version that a user has coded to break security. Theoretically, a normal user cannot become a superuser, so setting the setuid bit and running the local `rsh` program as superuser should be enough to guarantee the correct program is being run.

We now need some kernel interlock that will insist that the `rsh` program *must* be run as superuser and will cause the connection to fail when a mortal user attempts to program their own version. The TCP/IP implementation on UNIX allows the superuser and no one else to obtain port numbers between zero and 1023. If a remote machine is sent a connection from a port number in that range, then it knows that the superuser is running the correct program at the local end. The situation is now secure and the remote machine can believe what it is told.

However, the use of trusted port numbers was not the total solution. The authentication system recognized that, on the whole, machines operate in groups. When a connection is made over TCP/IP, the server end of the connection can see the IP address of the originator of the message and can translate that IP address into a machine name. This knowledge is used by an access control file called `/etc/hosts.equiv`. This file contains one machine name per line. When a connection is made to a server, it checks the name of the client machine in the `hosts.equiv` file. If the name is present, the server trusts the client and believes what it is told about the local and remote user name. The theory here is that the creation of an access control list in `/etc/hosts.equiv` allows the systems administrator to specify a set of machines that are to be regarded as trustworthy.

One problem with this system on Sun machines is the `/etc/hosts.equiv` file is maintained by NIS, and the standard system is distributed with the file containing a single "+." This is the magic NIS character that matches any name in the NIS table, so the vanilla system trusts everyone who wishes to connect with `rsh`. Removing this "+" is the first task when installing a new Sun system on a network.

There are other situations where it seems desirable to permit the user to nominate a remote machine that is permitted to access the local machine as that user. In addition to the `/etc/hosts.equiv` file, each user can create an access control file in their own home directory: the `.rhosts` file. Again, this generally contains one machine name per line. When the user sends `rsh` from a machine that has its name present in the file, then that machine is trusted to send the correct user information. In addition, the user can also supply a user name along with the machine name. This allows the `.rhosts` file to say: "It's OK to be *me* when using an 'r' command from *this* machine as *this* user."



The `.rhosts` file needs protection by the file system. For instance, if I can come onto your machine and edit your `.rhosts` file so that it contains my machine and my user name, then I can retreat back to my machine and use `rsh` to run commands as you. So the `rshd` program refuses to use the contents of an `.rhosts` file unless it's owned by the user who is trying to access it, and its file permissions only permit access by that user. In fact, forgetting to check and set file permissions on the `.rhosts` file is the largest source of mysterious failures when attempting to use an "r" command.

Incidentally, the `root` user on any machine is treated somewhat differently by the authentication system. Superuser must permit access to the machine via an `.rhosts` file on their home directory (conventionally the root of the file system). Any entry in `/etc/hosts.equiv` does not apply to superuser. This is done because systems may be administered by different people and we are unable to determine access rights from their name. They are all using `root` as a login name.

Security

What I've described above is the vanilla implementation of the authentication system. If you take a look at the Sun documentation you'll find that many frills have been added over time, for example, the `.rhosts` file can *deny* access to specific machines.

The fundamental problem is this system relies too much on the security supplied by the UNIX kernel. When PC systems appeared on the local Ethernet, they had no definition of a user, and certainly no superuser. It is possible for a PC system to be programmed to send connections to `rshd` from a low-numbered port number and lie about who the local user is. The notion of "trusted port numbers" flew out the window very quickly.

Once a UNIX system is compromised, an intruder can easily rush around the local network using `.rhosts` access to move from machine to machine. On November 2, 1988, `.rhosts` files were exploited by the Internet Worm, a program whose task was to replicate itself as it moved from machine to machine across the Internet. Its urge for life was somewhat too strident because it overloaded its target machines, making them unuasable. I suppose some good did come from this. The incident did raise the whole problem of security on the Internet, turning the issue into a hot topic.

In general, users are unaware of security when they wish to get a specific job done and will leave unwanted contents in `.rhosts` files. The basic problem is the `.rhosts` file puts security control in the hands of the users who are often unaware of the dangers. Of course, every systems administrator knows that their job would be much easier if there were no users.

Finally, the system depends on the integrity of the name service because it's doing a reverse lookup on an IP address to determine the name of a machine. If the name service can be spoofed into thinking that a specific IP address belongs to a different machine, then the system is compromised.

Many system security texts suggest that users are dissuaded from using `.rhosts` because it's too open to abuse.

The systems administrator has no real control over what systems are entered in a user's `.rhosts` file and, therefore, doesn't know where logins and remote execution requests are coming from. One solution has been to use Kerberos to provide trusted validation of machines at a session level so that the validation only lasts for a small period of time and depends on a login made using a password at some point during the user's session.

The reason alternative solutions have been sought is because it's convenient to use `rsh` and its friends. Security is always a compromise between usability and the need to keep the bad guys out. Firewalls have become a popular way to allow lax security inside a network while providing a secure gateway to the outside world. This is probably not good thinking. Maintaining a good level of security across a site can help if a firewall is breached.

I don't have a firewall. I rely on the security systems on all my machines to prevent unwanted access from the outside world. This works because my site has a small number of users and we are aware of the security aspects of what is done on the site. I also use packet filtering on my router to prevent external sites from connecting to any `rshd` daemon on any machine. Of course, this means that when I am away from home and want to `rlogin` into one of my machines, I cannot and I am forced to use `telnet` and give a password.

One real alternative for remote access over the Internet is `ssh`, this system behaves like the `rlogin` command but uses encryption over a single connection from the client to the server.

Finding Out More

Most of the information in this article can be found in the standard issue manual pages that are supplied with your system. Pages of interest are `rshd` (for a Sun system, it is in Section 1M, and for a vanilla BSD system, it is in Section 8), which describes the authentication protocol in detail; and `rhosts` (Section 4 on a Sun, Section 5 on a BSD system), which contains information on the valid contents of the `/etc/hosts.equiv` and `.rhosts` files.

If you want to find out more about security there are several books on the topic. However, many of them tend to dismiss the "r" commands with a "just say no" statement. One book that does suggest practical help is *UNIX System Security* by David A. Curry (published by Addison-Wesley Publishing Co., 1992, ISBN 0-201-56327-4). This book also has a good section on the Internet Worm. You can find out more about `ssh` from `http://www.ssh.fi/sshprotocols2/index.html/`.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests. He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.