USE OF SSH ON A COMPARTMENTED MODE WORKSTATION[1]

Johnny S. Tolliver (jxt@ornl.gov)
Oak Ridge National Laboratory
P.O. Box 2008, Oak Ridge, TN 37831-6418

and

David Dillow (il1@ornl.gov)
Lockheed Martin Energy Systems
1099 Commerce Park Drive, Oak Ridge, TN 37830-8027

Abstract

SSH stands for "Secure Shell." It is not a user shell like *csh* or *ksh*. Instead it is a widely-used means to accomplish secure, encrypted communication among cooperating nodes. It is a secure replacement for the "r-commands" *rsh*, *rlogin*, and *rcp*. SSH is free for noncommercial use and builds and runs on most any Unix platform. A Compartmented Mode Workstation (CMW) is an example of a secure or "trusted" operating system. The use of SSH on a CMW introduces security problems unless the SSH source code is modified to take advantage of the security features of the CMW. This paper describes the port and use of SSH on one particular brand of CMW.

Introduction

Compartmented Mode Workstations (CMWs) were invented by and for military users. As such, they are generally used on a closed, trusted network where an assumption of network security exists. However, as the military market decreases and CMW technology matures, some users and vendors are striving to apply CMWs to other venues. Use of a CMW as a Kerberos key server, for example, makes sense because a CMW is more difficult to penetrate using any of the latest CERT "bug-of-the-week" methods. Others have proposed using CMW technology for Web servers when proprietary data is involved. The Secure Socket Layer (SSL) protocol may provide adequate network security for HTTP data transfers but does nothing to protect the server operating system itself from attack. If the network becomes too difficult to "sniff", then attackers will move to attacking the endpoints instead of the network. Therefore, it is not unreasonable to use stronger endpoints, such as CMWs, when one really wants to protect the data stored there. Several famous Web

[1]This work was supported by the U. S. Department of Energy Office of Safeguards and Security under DOE Project No. BR-GD060403 with Lockheed Martin Energy Systems, Inc.

server endpoints have been attacked — CIA, NASA, the Justice Department, etc. One must wonder if the attackers would have been as successful had the Web servers been hosted on a CMW or another trusted operating system. If a CMW is used on an untrusted network, then improved network security is needed. The Secure Shell (SSH) is one method to provide better network security. SSH has been configured to build on most any popular Unix platform, but it must be modified to operate properly within the security environment on a CMW. This paper describes the port and use of SSH on one particular brand of CMW — the HP-UX CMW 10.16.

In order to set the context, we will begin by briefly describing some of the features of SSH. However, this is not a paper on SSH; please see the SSH home page at *http://www.cs.hut.fi/ssh* for more complete documentation on SSH itself. SSH is not a user shell like *csh* or *ksh*. Instead it is a means to accomplish secure, encrypted communication among cooperating nodes. It is a secure replacement for the "r-commands" *rsh, rlogin,* and *rcp*. SSH is free for noncommercial use and builds and runs on most any Unix platform. A commercially available client exists for Windows with a Mac version promised soon. SSH was created mostly by one person, Tatu Ylönen (ylo@cs.hut.fi), at Helsinki University of Technology, Finland. Some features and advantages of SSH are[1]:

- Strong authentication to close several security holes: IP routing, DNS spoofing, password sniffing.

- All communications are encrypted automatically and transparently. In addition to privacy, encryption also protects against spoofed packets and hijacked connections.

- X11 traffic is also encrypted to provide secure X11 sessions.

- Built-in multiple user-convenience features designed to make SSH easier to use than its nonsecure counterparts.

Basic Operation of SSH

SSH has two basic functions: strong authentication and protection of privacy. In the authentication role, SSH replaces the standard "r-commands"*rsh*, *rcp*, and *rlogin*. It uses public-key cryptography for authentication to permit remote shell execution, file transfer (remote copy), and remote login without cleartext passwords on the network. In fact, when properly set up, no passwords at all are required while still maintaining strong authentication. Unlike the r-commands, which also offer password-free operation, SSH does not need *.rhosts* files which provide only minimal authentication and are subject to IP-address-and-username spoofing.

The other major function of SSH is protection of privacy by using strong encryption of *all* traffic using a session key generated "on the fly," foiling the would-be network eavesdropper. The session key is communicated between the daemon and client via the same public key mechanisms used for authentication. Without SSH, X11 sessions can be easily sniffed. But since SSH also automatically supports encrypted X11 traffic, this major security problem in X11 is solved

Some of the functions provided by SSH are also available with Kerberos — no cleartext passwords and, optionally with Kerberos, encrypted sessions. But SSH is *much* smaller (about 45,000 lines vs 250,000 lines of code) and easier to build, install, and administer than Kerberos. SSH is also easier to use than Kerberos and can be used by individual users with no central administration necessary. There is no central repository of keys, for example; no need to maintain and secure a central Kerberos Key Distribution Center computer. The client side of SSH may be built and installed by a normal user without root privileges. Thus ordinary users can build and use the SSH client to communicate with a remote machine running SSH without any intervention from the client machine's system administrator. At least one user[2] has compared the use of encrypted sessions with SSH and Kerberos and reported that an SSH session has significantly better interactivity (quicker echoing of keystrokes) than an encrypted Kerberos v5 beta 5 session.

SSH has essentially three modes of operation, all three providing both strong authentication and privacy:

1. Secure (encrypted) remote login (using the *slogin* command)
2. Secure remote command execution (using the *ssh* command, much like the nonsecure *rsh* command)
3. Secure remote copy (the *scp* command, much like *rcp*)

The first mode is a simple login session, much like *telnet* or *rlogin*, but with more features. First, it provides strong authentication without passwords, so it is easier than *telnet* and more secure than *rlogin* with *.rhosts*. Second, it encrypts all traffic for privacy. Third, if you have a DISPLAY environment variable set on your local machine, the appropriate value is automatically propagated to the remote machine. That is, you don't have to manually set your DISPLAY on the remote machine in order to run X applications. SSH also propagates the TERM variable and the terminal modes (e.g., the erase character). These feature makes SSH easier on the user than either *telnet* or *rlogin*. Finally, any X application that you run during the *slogin* session will also be fully encrypted.

Remote command execution is perhaps the most common mode of operation. It uses the *ssh* command, much like the common *rsh* command but with strong authentication and encryption. A special case of this mode is *ssh host xterm* in which the remote command is *xterm*. This *xterm* window will automatically be displayed back to your originating machine. Since *xterm* gives you a window on the remote machine with a user shell running in it, this command amounts to a login session too. This usage is much like *rsh host xterm* except that *ssh* again propagates your DISPLAY variable without additional user actions.

The third mode of operation is secure, encrypted, strongly authenticated file transfer using remote copy *(rcp)* syntax.

SSH on CMW

A Compartmented Mode Workstation (CMW) is an example of a “trusted operating system.” Based on standard Unix, a CMW OS offers much more inherent security at the OS level than regular Unix. But porting software to a CMW offers new challenges. As an example, whenever a regular Unix kernel performs any security-related chore, it first checks if the effective UID is 0 (root) or not. If root, then the operation proceeds; if not, the operation fails. Although there can be a UID 0 on a CMW, the kernel does not look at the effective UID. Instead, the kernel checks whether or not the process attempting the operation holds the specific “privilege” to perform that operation. As an example, to override write protection on a file might require that the process attempting to write to a file hold the ALLOWDACWRITE privilege. DAC stands for discretionary access control and may be thought of as the normal Unix file modes (read, write, and execute privileges) that a user may apply to his files at his own discretion. If a file is protected against writing (-r--r--r--) then the Unix kernel disallows writing to that file. On a normal Unix OS, root can override that protection and write the file anyway. This override is accomplished when the kernel checks to see if the process attempting the write has effective UID of 0 or not. But on a CMW system, UID 0 has no special permission. No process, even one with effective UID of 0, can write the file unless that process has the ALLOWDACWRITE privilege. In a typical CMW, there are about 100 such privileges, including ALLOWDACREAD, to permit reading a file, CHOWN, to permit changing the ownership of a file, and many others.

On regular Unix machines, the *sshd* daemon must be running as root, listening on port 22 for incoming *ssh* connections. When an *ssh* client makes the connection, *sshd* does some bookkeeping and eventually spawns a child process for the incoming user and switches the UID and GID of the child process to that of the incoming user. In this way, the user gets a process on the host machine with his own UID and GID while the parent *sshd* process continues running as root to service other incoming requests. Switching the UID and GID is very common and straightforward on a normal Unix machine using the *setuid()* and *setgid()* system calls. Other examples of programs that switch the UID are *telnet* and *login*. Both are essentially root processes that start up a session for a user and then switch the UID to that user.

On a CMW, however, with its rich set of privileges, much more than merely switching the UID is necessary. Without modifications to the *sshd* source code, a remote user could get a process on the host that would appear to belong to the user — i.e,. it would have the correct UID (and *whoami* would return the correct username) — but that user would have all of the privileges of the *sshd* daemon itself. This is because child processes inherit the privileges of the parent and merely switching the UID of a child process does not remove the privileges that were already held by that child. If the *sshd* daemon holds many privileges, which it must in order to function properly, the user would get all those privileges too. Without code modifications, a normal user, holding the privileges inherited from *sshd*, would be able to *'su'* without supplying root's password to get the root UID. He would then have access to anything root could do, including reconfigure the security characteristics of the machine. This unacceptable situation arises because the unmodified *sshd* source code, knowing nothing about the fine-grained CMW privileges, merely switches the UID and GID and does nothing with privileges.

Clearly, the correct steps must include modifying the source code to look up the security profile of the incoming user and set the privileges of the child process to those appropriate for that user. The discussion to follow is based on experience with one particular brand of CMW — the HP-UX CMW, version 10.16 — but very similar behavior is to be expected from other vendors' CMW implementations. Hewlett-Packard's CMW is derived from the SecureWare implementation. Other non-SecureWare-based implementations may differ in the details, but the system calls are often very similar to those described here. Each user on any CMW has an extensive user profile stored in a secure database on the machine. On HP, this database is called the "protected password" database. All of the user's privileges are specified by the Information System Security Officer (ISSO) and stored in that database. The CMW OS provides a means for a trusted program to query the database for user characteristics. For example, a user's base privilege set is available via the *getprpwuid()* system call. This call is somewhat like the normal Unix *getpwuid()* system call but returns much more information. However, not just any process is allowed to query the database. Only processes with ALLOWDACREAD can retrieve information with the *getprpwuid()* call. A "trusted program" is one which is given the proper privileges to make such calls. The privileges permitted to any program are defined and maintained by the ISSO. So not only must the *sshd* source code be modified to call *getprpwuid()* when needed, the program must be enabled with the authority to make that call. That is, the program must explicitly be assigned the ALLOWDACREAD privilege by the ISSO. Once the user's base privileges are obtained, the base privileges of the child process can be set with the *setbaseprivs()* call which requires yet another privilege, CHSUBJPRIV.

The proper form when writing a trusted program is to use "privilege bracketing" to enable (or "raise") privileges when needed and disable ("lower") them when no longer needed. This procedure is referred to as the Principle of Least Privilege because the trusted program is written to hold minimal privileges most of the time and to raise higher privileges only when necessary and to hold them only as long as needed. In addition to the special privileges needed above, calling *setuid()* and *setgid()* requires another privilege, CHSUBJIDENT. One can see that porting *sshd* to the CMW system requires nontrivial code changes to raise and lower the necessary privileges around such sensitive system calls plus the additional code needed to obtain and set a user's privileges for the child process. Fortunately the code structure and the privileges needed are such that the code can be tightly bracketed. In almost all cases, only one line of code, making just one system call, needs to be bracketed with privilege-raising and lowering calls. One example is when binding to a port to listen for incoming *ssh* connections. Under normal Unix, only root can bind to a port numbered below 1024. However, under HP-UX CMW, binding to a port below 1024 requires the NETPRIVADDR privilege. So we raise NETPRIVADDR, keeping track of the currently held privileges, call *bind()*, and then restore the previous privileges. The rest of the approximately 45,000 lines of code continue to operate in a purely unprivileged mode, and the code is unprivileged almost all of the time.

In addition to the normal UID and GID, CMWs also hold another variable called the “login” UID or LUID. This is a permanent, indelible “login” user id that trusted systems keep in addition to the normal UID. When the effective UID is changed by a suid program, the LUID does not change. The LUID was designed for auditing purposes. Since the effective UID can change all auditing records are kept against the LUID. In this way, an audit trail is kept of the original login user’s activities. Since the *sshd* daemon runs as root with UID 0, the LUID is also 0. When *sshd* switches the UID and GID of the child process to become the remote user, the LUID remains 0. Thus, the child process has the correct UID but retains LUID 0. All audit records of that user’s activities would be kept against root instead of the real user. In addition to protecting the audit trail, the LUID is used as a key to look up “command authorizations” which can be thought of as collections of privileges that a user can obtain when requested. A normal user with LUID 0 would receive all of root’s command authorizations, obviously not a desirable situation. Clearly, the *sshd* daemon must somehow change the LUID to that of the incoming user. Despite the fact that the LUID is supposed to be permanent and indelible, the system obviously provides a way of changing it during a normal user login procedure since a normal login process must begin as root and then switch the UID to that of the incoming user. In order for the LUID to be meaningful, the login process must also switch the LUID to that of the user as well. Obviously the modified *sshd* code must perform a similar operation. In fact, there is a *setluid()* system call and a corresponding privilege, CHSUBJLUID, to permit its use. The *sshd* daemon must raise the CHSUBJLUID privilege and change the LUID to that of the incoming user.

Current Status and Remaining Problems

We have implemented all the changes described above, along with several others needed for very similar reasons, in the SSH version 1.2.20 source code on HP-UX CMW version 10.16. We can now securely connect from an unlabeled client machine to the CMW using any of the three tools provided by SSH — *slogin, ssh,* and *scp*. By “unlabeled” we mean a “normal” nontrusted operating system that does not contain the special security attributes that exist in trusted operating systems. We have successfully tested SSH connections from IBM, HP, and SunOS clients running SSH 1.2.20. Since no changes are needed on the client machine, any commercial implementation of the SSH client (for Windows machines, for example) should work as well although we have not made such tests. Conversely, we can also connect from a CMW to an unlabeled host. This functionality requires small changes to the client code on the CMW but no change to the *sshd* daemon running on the unlabeled host.

Both of the connection types described above (unlabeled-to-CMW and CMW-to-unlabeled) are using unlabeled network packets. When two trusted operating systems communicate with one another, they use labeled packets that contain additional security attributes not present in unlabeled packets. The more interesting application of SSH would be between two CMW machines. Here, it will be necessary to read the security information on the incoming labeled packet and set up the incoming user's security attributes accordingly. With these techniques, a properly authorized user running at a high clearance on a client machine should be able to open a similarly-cleared session on a remote CMW. These changes are not complete at the time of this writing. Progress on the code changes required to implement CMW-to-CMW communications secured by SSH will be reported in the oral presentation.

Summary

In conclusion, SSH offers secure communications with strong authentication and strong encryption of all traffic, including encrypted X11 traffic, a feature currently lacking in Kerberos. SSH is relatively small, easy to install, and easy to administer. Porting to a CMW requires code modifications to properly use the security environment on a CMW. We have completed the necessary changes, using tightly-bracketed code, to implement unlabeled-to-CMW and CMW-to-unlabeled secure communications using SSH. We will report on CMW-to-CMW communications using labeled packets in the oral session.

References

1. See the SSH home page at *http://www.cs.hut.fi/ssh*.

2. See Wayne Schroeder of the San Diego Supercomputer Center *(http://www.sdsc.edu/~schroede/ssh_cug.html)*.