Global Information Assurance Certification Paper

SSH, a practical guide to installation, configuration, and use.

GSEC Assignment version 1.4
Mark Sowerby
23 October 2002

Table of Contents

Abstract

Secure shell is more or less a drop in replacement for telnet, ftp, r exec, rlogin, rsh, rcp, and performs secured communication and authentication across a network. Telnet, ftp, rexec, rsh, rcp and ftp are all traditional methods for accessing a UNIX system via the command line and are run over TCP/IP; so SSH is primarily a method to gain access UNIX systems across a network. Secure shell (SSH) is actually a suite of programs, the main parts being a client and a server program, ssh is the client, which is used to connect to the sshd server. Clients exist for most platforms including Windows, Dos, Macintosh, Java and of course UNIX in both free and commercial versions. SSH is especially useful when the network is considered hostile, unknown or not under local control, although it is always a good philosophy to assume the network infrastructure is not always friendly or benign.

This document aims to detail how to compile, install, configure and use Secure Shell. The target audience of this document is any system administrator concerned about security, or any user of a system wishing to grasp how to use Secure Shell. Basic system administration skills are assumed in the target audience, but not necessarily experience of software compilation. The author also hopes to impart into the reader some security awareness.

Why Secure Shell?

Secure shell authenticates (validates the identity of) both the client and server ends of a "conversation" over an encrypted link by use of "keys," the user is given the assurance that the system they are typing their password on is the system they believe it to be. Users can also be authenticated by this same method of using "keys," or they can be authenticated by the use of the usual username and password combination. There is also support for strong 2 factor authentication (SecurID etc) where knowing the username and password is only the first step of the process as a further piece of information is requested from the user before they are considered authenticated. This authentication process happens over a secured communications channel that is more resilient to eavesdropping, and other attacks than telnet, rcp, rsh etc.

Access to a system can be restricted on a "per user" basis, and also on a "per host" basis. This gives the administrator reasonable access control; specific hosts or even entire subnets can be denied, so that even knowing the correct username and password combination is insufficient to gain access, the connection must originate from a permitted location; for example, this can be used to ensure that servers can only be administered from the desktop systems used by the administration team. By preventing certain accounts from "logging" in to the system, for example "oracle" or "informix" can be denied direct network login to the system; this improves the security posture of a system, as it is a quite easy matter to find out which applications are running on any particular system by "port scanning" it, if Oracle is found to be running, it is a good guess that there is an account called "oracle." Knowing a valid account name on a system is half the username / password combination required for access.

Secure shell has the ability to "tunnel" traffic through its' connection, almost like a personal point-to-point VPN, this means that the ability of secure shell to create an encrypted link can be used to encrypt traffic that is normally in plain text. A document detailing how to tunnel X through an SSH tunnel can be found at this location

http://rr.sans.org/encryption/x_tunnels.php . Specific information on tunnelling traffic through an SSH tunnel[1] can also be found here http://rr.sans.org/encryption/tunnels.php .

Note, "SSH" will be used to refer to either the secure shell protocol, or the secure shell suite of programs, also "ssh" will be used to refer to the secure shell client program, "s shd" will be used to refer to the ssh server (daemon).

Secure shell comes in two main versions, SSH1 and SSH2. SSH version 1 is becoming deprecated as it has some known weaknesses [6]. SSH Version 2 has more features, enhancements and support for different a lgorithms. From a user perspective it operates in the same manner. SSH2 supports RSA SecureID and Kerberos 5. SSH2 also has an sftp command, which allows interactive transfer of files, in a style similar to ftp, SSH1 only had an interface similar to rcp for transferring files.

Open SSH is the public domain counterpart to the commercial products. Secure shell is available from www.openssh.org , and also commercially, from www.ssh.com[4] or www.f-secure.com. The document at http://rr.sans.org/encryption/intro_SSH.php discusses the differences between SSH1 and SSH2 and also has a description of how SSH actually works[2]. Clients are available for many platforms, even java versions can be found, http://www.employees.org/~satch/ssh/faq/ssh-faq-2.html lists where many of the free clients can be obtained.

Note, secure shell is not a panacea, and there have been vulnerabilities found in most versions. Restricting access to the secure shell "login" prompt is recommended if possible, and always check that the version you are installing does not have any known vulnerabilities (check vendor web sites, www.cert.org, etc).

Installation

Secure shell is normally shipped as source code, which means that it will need to be compiled to work on each specific system. For demonstrating installation, openssh will be used on Linux running on x86 architecture, with the gnu compiler gcc due to their free availability.

What is needed

Obtain the source code for zlib, openssl, and openssh. Zlib is utilised by openssh for its' compression "features" and openssl is utilised for the cryptography routines; that way the writers of openssh do not have to re-invent or rewrite "the wheel" for the compression or cryptography routines in openssh. Please refer to the licenses for each of these components and keep in mind the credit deserved to the respective authors. Zlib and openssl will need to be compiled and installed prior to compiling and installation of openssh.

Openssh requires specific versions of zlib and openssl, as older versions may have vulnerabilities in them or features found in the latest version may be required for openssh to work. Which versions are required is detailed at this link which is the openssh install guide, ftp://ftp.ca.openbsd.org/pub/OpenBSD/OpenSSH/portable/INSTALL check the versions of zlib and openssl are suitable for the openssh version obtained. The openssh install guide also makes an excellent companion to these instructions. Root access to the system is required to install these software components, but the binaries can actually be compiled without requiring root access. Typically the download files have the suffix Z or gz

that indicates the compression method used on the files. Files with a gz suffix are compressed using a program called gzip that may already be installed on a system, if not, gzip need to be obtained from the link above, there are binaries for many systems available. Files with a Z suffix have typically been compressed by "compress" which is standard on UNIX systems. There may also be some files noticed on the ftp servers with a suffix of sig, these files contain a special signature unique to the file being downloaded, so that when downloading the file, it can be checked that it is what it pertains to be. This is beyond the scope of this document, but refer to man (1) md5sum for details of how to verify the md5 signature of a file, or for gnu privacy guard signatures, see man (1) gpg.

Zlib can be downloaded from www.zlib.org and is distributed under liberal licensing, which in brief allows zlib to be used for any purpose as long as you do not claim it to be your own work.

Openssl can be downloaded from www.openssl.org and is distributed under openBSD [3] style of license.

Openssh can be downloaded from www.openssh.org/portable.html and is available under the liberal openBSD [3] style of license.

Gcc can be obtained from gcc.gnu.org If Linux is the operating system being used, it will generally have gcc installed, but if this is not the case gcc.gnu.org/install/binaries.html has links to pre-compiled binaries for various popular operating systems. Installation of gcc is beyond the scope of this document, refer to the instructions that ship with the particular gcc package obtained. Ensure that gcc is in the search path preferably at the begining of the path to ensure that the correct compiler is called, running the command **gcc –v** will indicate that gcc is in the search path and also the gcc version. The compilation still might fail if a minimal install of the operating system is on the system, as certain programs or files required by the compilation process, such as nm or ar may not be part of a minimal installation. Ensure that nm and ar are in the search path, by default on Solaris they reside in /usr/ccs/bin.

```
[jbloggs@sneezy jbloggs]$ gcc -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/2.96/specs
gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-112)
[jbloggs@sneezy jbloggs]$ which gcc
/usr/bin/gcc
[jbloggs@sneezy jbloggs]$ which xauth
/usr/X11R6/bin/xauth
```

Notes on compilation

A default compilation is usually adequate, but it may be desirable to change certain aspects of the compilation, for example it may be necessary to install the programs in a directory other than /usr/local where they are typically installed, some features of a program may wish to be disabled or some features enabled that are not normally used. A script that ships with the source code for each program, called "configure" often controls this compilation behaviour. Refer to the instructions that ship with each program for details, but changing configuration options without understanding the impact, may lead to a program that does not work correctly.

```
[root@sneezy openssl-0.9.6g]# ./Configure --help
Usage: Configure [no-<cipher> ...] [-Dxxx] [-lxxx] [-Lxxx] [-fxxx] [-Kxxx] [rsaref] [no-
threads] [no-asm] [no-dso] [386] [--prefix=DIR] [--openssldir=OPENSSLDIR] [ --test-
sanity] os/compiler[:flags]
```

To compile anything other than the simplest program requires many steps to link all the various components together, but a program called “make” does this (which also needs to be in the search path) by reading instructions contained within the "Makefile." A program’s author will ship a “Makefile” with the source code so that the correct sequence of steps is carried out with the compilation. Explaining how “make” operates is beyond the scope of this document (see “man (1) make” for details), but has been mentioned due to its’ importance to the compilation process.

Zlib

In the directory where the zlib source code has been downloaded, unpack and un-tar the file. Running **make && make install** will compile and then copy the binaries onto the system in the directory /usr/local (/usr/local must already exist as a directory beforehand otherwise the install will fail).

```
[root@sneezy openssl-0.9.6g]# cd /tmp/ssh_install/zlib/ ; gunzip *z && tar xf *tar
[root@sneezy zlib]# make && make install
cc -O   -c -o example.o example.c
cc -O   -c -o adler32.o adler32.c
cc -O   -c -o compress.o compress.c
cc -O   -c -o crc32.o crc32.c
cc -O   -c -o gzio.o gzio.c
*
*
*
cp libz.a /tmp/usr/local/lib
cd /usr/local/lib; chmod 755 libz.a
cd /usr/local/lib; if test -f libz.so.1.1.4; then \
  rm -f libz.so libz.so.1; \
  ln -s libz.so.1.1.4 libz.so; \
  ln -s libz.so.1.1.4 libz.so.1; \
  (ldconfig || true) >/dev/null 2>&1; \
fi
[root@sneezy zlib]#
```

The screen dump above shows the compilation of zlib, at the bottom of the screen dump it is shown that the file libz.a has been placed into /usr/local/lib. By examining /usr/local/lib satisfactorily operation of the compilation can be verified, by locating the file libz.a.

Openssl

Unpack and un-tar the source code **gunzip -c openssl-0.9.6g.tar.gz | tar xf -**. Which will place the source code into a new directory. Change into this directory and execute the script **./config**, **"**config" is the name of the configuration script for openssl instead of the usual name "configure." Config will then build the system specific ma kefile. When "config" has finished, running **make && make install** will compile and install openssl.

```
[root@sneezy ssh_install]# gunzip -c openssl-0.9.6g.tar.gz | tar xf -
[root@sneezy ssh_install]# ls
openssh-3.4p1.tar.gz  openssl-0.9.6g  openssl-0.9.6g.tar.gz  zlib
[root@sneezy ssh_install]# cd openssl-0.9.6g
[root@sneezy openssl-0.9.6g]# ./config
Operating system: i686-whatever-linux2
Configuring for linux-elf
Configuring for linux-elf
IsWindows=0
CC          =gcc
CFLAG       = -fPIC -DTHREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN
-DTERMIO -O3 -fomit-frame-pointer -m486 -Wall -DSHA1_ASM -DMD5_ASM -DRMD160_ASM
*
*
*
[root@sneezy openssl-0.9.6g]# make && make install
+ rm -f libcrypto.so.0
+ rm -f libcrypto.so
+ rm -f libcrypto.so.0.9.6
+ rm -f libssl.so.0
+ rm -f libssl.so
+ rm -f libssl.so.0.9.6
making all in crypto...
make[1]: Entering directory `/tmp/ssh_install/openssl-0.9.6g/crypto'
( echo "#ifndef MK1MF_BUILD";  \
echo '  /* auto-generated by crypto/Makefile.ssl for crypto/cversion.c */  '; \
echo '  #define CFLAGS "gcc -fPIC -DTHREADS -D_REENTRANT -DDSO_DLFCN -
DHAVE_DLFCN_H -DL_ENDIAN -DTERMIO -O3 -fomit-frame-pointer -m486 -Wall -DSHA1_ASM -
DMD5_ASM -DRMD160_ASM"'; \
echo '  #define PLATFORM "linux-elf"'; \
*
*
*
installing libcrypto.a
installing libssl.a
[root@sneezy openssl-0.9.6g]#
```

The above screen dump shows the compilation of openssl. At the bottom of the screen dump libcrypto.a and libssl.a have been installed, these files are installed in /usr/local/lib and also a program, openssl should be found in /usr/local/bin , there is also a directory, /usr/local/ssl .

Openssh

Unpack and un-tar the source code. **gunzip –c openssh-3.4p1.tar.gz | tar xf -** . That will place the source code into a new directory. Change into this directory **cd openssh-3.4p1** and then execute the configuration script **./configure** that will build the system specific makefile . By typing **./configure --help** all the configuration options available for the compile via the configure script will be shown. These options are worth examining, as they will be used for the example compile. The screen shot below shows, the **"--with-md5-passwords"** argument being passed to the configure script, as the sample system does make use md5 passwords. This is not common to most UNIXs so should not be used unless it is certain the system uses MD5 passwords, some Linux systems do though use MD5 passwords. The other arguments given to configure are for the "privilege separation" feature of SSH (**--with-privsep-path=/usr/local/ssh --with-privsep-user=nobody**), the options used here for privilege separation should be suitable for most UNIXs. Privilege separation is explained further on in this document. When the configure script has completed, running **make && make install** will compile and install openssh.

```
[root@sneezy ssh_install]# gunzip -c openssh-3.4p1.tar.gz | tar xf -
[root@sneezy ssh_install]# ls
openssh-3.4p1  openssh-3.4p1.tar.gz  openssl-0.9.6g  openssl-0.9.6g.tar.gz  zlib
[root@sneezy ssh_install]# cd openssh-3.4p1
[root@sneezy openssh-3.4p1]# ./configure --with-md5-passwords --with-privsep-
path=/usr/local/ssh --with-privsep-user=nobody
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking build system type... i686 -pc-linux-gnu
checking host system type... i686 -pc-linux-gnu
checking whether byte ordering is bigendian... no
checking how to run the C preprocessor... gcc -E
*
*
*
[root@sneezy openssh-3.4p1]# make && make install
*
*
*
id nobody || \
        echo "WARNING: Privilege separation user \"nobody\" does not exist"
uid=99(nobody) gid=99(nobody) groups=99(nobody)
[root@sneezy openssh-3.4p1]#
```

SSH should now be compiled and successfully installed in /usr/local. In /usr/local/bin will be the client programs, in /usr/local/sbin will be the server (sshd), in /usr/local/etc will be the configuration files for SSH and also the server keys. The server keys enable SSH to prove its' identity (authenticate) to clients. The warning message at the end of the screen dump above is harmless, this is just the code from the installation routines; the installation script would "echo" to the screen the words "WARNING....." if the "id nobody" command had failed, but the command was successful, the results of the "id nobody **"** command are the very last line of output from the installation routines "id=99(nobody) gid=99(nobody) groups=99(nobody)." This can be demonstrated by typing **id nobody || **
echo "WARNING: Privilege separation user \"nobody\" does not exist" at the command line.

Starting SSHD

To start sshd, the daemon is simply executed on the command line **/usr/local/sbin/sshd** and sshd will start up and bind (listen on) port 22. Sshd will take the configuration from the file **/usr/local/etc/sshd_config** . A check that sshd is listening on port 22 can be made "telnetting" to port 22 at address "localhost" which loops back to the local system; this is shown in the text box below. A telnet session can be disconnected from an SSH server (sshd) by just pressing "enter" a couple of times, which causes sshd to realise that a proper SSH client is not connected. The connection will then terminate with the message "Protocol mismatch," but sshd will still be listening for client connections.
An SSH client connection can now be attempted, by typing **ssh localhost**, a password prompt will be presented, and on entering the correct password interactive shell access should be granted. Being permitted to log in to a system directly as "root," is not recommended, except at the console, this is explained later in the document, and also how to prevent this behaviour of SSH.
It is recommended to add the directory /usr/local/man to the MANPATH variable, so that the excellent manual pages that come with SSH are available for perusal.

```
[root@sneezy tmp]# telnet localhost 22
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.4p1

Protocol mismatch.
Connection closed by foreign host.
[root@sneezy tmp]# ssh localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
RSA key fingerprint is 92:85:17:16:1f:65:91:b4:b4:07:e3:d6:ff:e7:43:f8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
root@localhost's password:
Last login: Tue Sep 10 15:22:44 2002 from sneezy
[root@sneezy root]#
```

In the screen output above SSH warns about not be able to verify the authenticity of "localhost" and prompts for user input before continuing to connect. This is what happens when SSH connects to a system for the first time as SSH does not hold a record of the servers' public key, once connected SSH will save a copy of the key. When subsequently

connecting, SSH makes use of the saved public key to “cryptographically” validate the identity of the server (the word "cryptographically" is used for want of a simple explanation of the process).
When first connecting to a server through SSH, SSH cannot protect you from spoofing attacks[5] where an attackers system is "pretending" to be a friendly system; SSH does not know if it is connecting to the correct system or not at this stage. The username and password could be given away to the attacker.

Although SSH has been compiled and installed, sshd will not start automatically when the system is started, the script below can be used to stop and start sshd on a system. This script will need to be run every time your system boots, typically this will entail placing it in an "rc" directory, for example /etc/rc3.d or /etc/rc5.d. Please consult the manual pages for each specific system for details.

```
#!/bin/sh
##############################################
# OpenSSH start/stop script
# Author M Sowerby
# Arguments stop | start | restart
##############################################
PREFIX=/usr/local
[ -f ${PREFIX}/sbin/sshd ] ||  exit 1

case "$1" in
  start)
       echo  "Starting sshd"
       ${PREFIX}/sbin/sshd
       ;;
  stop)
       if [  -f /var/run/sshd.pid ]
       then
        echo  "Shutting down sshd"
        PID=`cat /var/run/sshd.pid`
        echo "on PID ${PID}"
        kill ${PID}
       else
        echo "cannot find the pid file for ssh"
       fi
       ;;
  restart)
       $0 stop
       $0 start
       ;;
  *)
       echo "Usage: sshd {start|stop|restart}"
       exit 1
esac
```

Both the SSH client (ssh) and the SSH server (sshd) have many configuration options, which can be set from configuration files or on the command line, one useful option for de -bugging SSH is to use the -v argument, which gives de -bugging (verbose) information as it connects to the server. Below is the output from connecting to the SSH server on localhost as above, but with the de -bugging information. This is useful when having problems, as a better idea of which stage in the connection set -up the problem lies.

```
[root@sneezy .ssh]#  ssh -v localhost
OpenSSH_3.1p1, SSH protocols 1.5/2.0, OpenSSL 0x0090602f
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Applying options for *
debug1: Rhosts Authentication disabled, originating port will not be trusted.
debug1: restore_uid
debug1: ssh_connect: getuid 0 geteuid 0 anon 1
debug1: Connecting to localhost [127.0.0.1] port 22.
debug1: temporarily_use_uid: 0/0 (e=0)
debug1: restore_uid
debug1: temporarily_use_uid: 0/0 (e=0)
debug1: restore_uid
debug1: Connection established.
debug1: rea d PEM private key done: type DSA
debug1: read PEM private key done: type RSA
debug1: identity file /root/.ssh/identity type    -1
debug1: identity file /root/.ssh/id_rsa type    -1
debug1: identity file /root/.ssh/id_dsa type    -1
debug1: Remote protocol version 1  .99, remote software version OpenSSH_3.1p1
debug1: match: OpenSSH_3.1p1 pat OpenSSH*
Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH  -2.0-OpenSSH_3.1p1
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex : server ->client aes128 -cbc hmac -md5 none
debug1: kex: client ->server aes128 -cbc hmac -md5 none
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_GROUP
debug1: dh_gen_key: priv key bits set: 131/256
debug1: bits set: 1605/3191
debug1: SSH2_MSG_KEX_DH_GEX_INIT sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_REPLY
The authenticity of host 'localhost (127.0.0.1)' can't be established.
RSA key fingerprint is 10:16:f7:11:92:99:43:fa:05:ee:51:31:33:bf:2b:6f.
Are you sure you want to continu  e connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
debug1: bits set: 1605/3191
debug1: ssh_rsa_verify: signature correct
debug1: kex_derive_keys
debug1: newkeys: mode 1
debug1: SSH2_MSG_NEWKEYS sent
debug1:  waiting for SSH2_MSG_NEWKEYS
debug1: newkeys: mode 0
debug1: SSH2_MSG_NEWKEYS received
debug1: done: ssh_kex2.
debug1: send SSH2_MSG_SERVICE_REQUEST
debug1: service_accept: ssh  -userauth
debug1: got SSH2_MSG_SERVICE_ACCEPT
debug1: authentications that can c  ontinue: publickey,password,keyboard  -interactive
debug1: next auth method to try is publickey
debug1: try privkey: /root/.ssh/identity
debug1: try privkey: /root/.ssh/id_rsa
debug1: try privkey: /root/.ssh/id_dsa
debug1: next auth method to try is keyboard  -interactive
debug1: authentications that can continue: publickey,password,keyboard    -interactive
debug1: next auth method to try is password
root@localhost's password:
```

Configuration

The configuration of ssh or sshd, can be controlled through their relative configuration files or through command line arguments. For example, the file /usr/local/etc/sshd_config controls the configuration of sshd, for controlling which port sshd binds to, the keyword "Port" is used within the file and by default is set to "22," as show below.

```
Port 22
```

But if sshd is started with a command line argument of -p 6060 as shown below, sshd will start up, but bind to port 6060 instead of port 22; the command line arguments take precedence over configur ation file.

```
/usr/local/sbin/sshd    -p 6060
```

SSH client is configured via the file /usr/local/etc/ssh_config. The exact file names and location vary from vendor to version etc, but the default compilation of openssh, installs the configuration files into /usr/local/etc, so this will be used as a standard for the examples. The configuration files for other distributions tend to be easy enough to find, the following command will more than likely find the configuration files, which are named intuitively and similar across the various flav ours.

```
ls  -al /etc/ssh* /usr/local/ssh*  /usr/local/etc/ssh*
```

sshd (Server) configuration

The sshd configuration file, is typical of UNIX configuration files, in that a comment begins with a "#" hash. Each line in the file consists of a keyword and value (or value list) pair. When the file is examined, it will be apparent that a lot of the configuration lines are "commented out," this typically indicates the default behaviour of sshd, so examination of the configuration file allows the default behavi our of sshd to be known. It is not the intent of this document to reproduce the excellent manual page for sshd (man (8) sshd), but some of the immediately useful configuration options will be detailed.

ListenAddress

This is the IP Address that sshd binds to. On a system with multiple IP addresses sshd can bind to a particular address. This is particularly useful if there is a separate management network, as sshd can be configured to only listen on the management address and hence it can only be accessed f rom the management network. By default sshd will bind to all the IP addresses that the system has, as indicated by the 0.0.0.0 address.

```
#ListenAddress      0.0.0.0
ListenAddress     192.168.123.123
```

PermitRootLogin

By default this is set to "yes" which will allow the system to be logged into directly as the root user. This is not recommended, as there is no accountability; administrators should always log in with their own personal account and then "su" to root, as there is actually a record of who logged in to the system. See man (8) sshd for details of the further options for this setting.

```
#PermitRootLogin   yes
PermitRootLogin    no
```

DenyUsers

This will prevent SSH "logins" to the accounts listed. Access to a system is normally gained by knowing a username and password combination. By simply port scanning [9] a system an attacker will probably be able to ascertain the applications a system is running, for example "ingres." The attacker would then know that the system is highly likely to have an account called "ingres" (and more than likely with a simple password, such as "1ngres"). This gives an attacker half the information needed for access, without having used any clever cracking techniques. By preventing the "ingres" user from having direct access, this type of attack is foiled. Group or shared accounts should not be used as they tend to be neglected; no one individual has ownership of the account; passwords tend to not be changed as often, and there is little audit trail. If log files are trawled at a later date, it may be known that the "informix" user was logged in at the time that the database stopped working, but this does not indicate the actual person who logged in. Making people login with their own personal account is always desirable.

```
DenyUsers    oracle informix   *@mailserver
```

Protocol

Weaknesses in the original SSH protocol triggered a re-write leading to SSH2; the original protocol then became known as SSH1. By default sshd can fallback to SSH1 compatibility mode and serve SSH1 clients. This is not recommended, as an attacker may exploit these weaknesses in the SSH1 protocol (which lead to the development of SSH2). It is always a good idea to disable unused features as protection will be obtained from any as yet unknown vulnerabilities in those features (if it has not got it, it cannot go wrong).

```
#Protocol 2,1
Protocol    2
```

SSH (client side) configuration

Little configuration tends to take place via the file /usr/local/etc/ssh_config as command line arguments can be specified at the time of connecting to a system, and the defaults are usually sufficient. The format of ssh_config is the same as sshd_config and some of the keywords are the same.

Host

One useful directive is the Host keyword and its' corresponding value. The behaviour of SSH client can be changed on a host to host basis, for example If there is a specific host that is always connected to with a certain set of parameters, then this can be set in ssh_config by surrounding the host specific arguments with the host keyword. For example, there may be one system named "gateway" where X applications are tunnelled back from, but it is not used on other systems, the configuration could be set -up thus:

```
Host    gateway
ForwardX11   yes
Host     *
ForwardX11   no
```

Protocol

See sshd configuration above for a description of "protocol," the same reasoning applies to the client that SSH version 1 has known weaknesses.

Usage

It has already been demonstrated how to connect to "localhost" using ssh, but to connect to a remote system running sshd, ssh must be run, followed by the name of the system or its' IP address i.e. **ssh 192.168.123.10** .
If the username on the local system is "mark," but the account on the remote system is "sowers," ssh would attempt to log on to the remote system as the local account name, ie it will try to log on remotely as "mark." As there is no "mark" account on the remote system the login would fail. Ssh gets around this problem by allowing the username to be specified on the command line in one of two ways; either b y using the -l argument to ssh, or by using user@hostname notation thus:

```
ssh  -l sowers    192.168.123.10
ssh –l sowers     gandalf
ssh   sowers@192.168.123.10
```

Key Pairs (logging in without a password)

SSH can be used to connect to a system, without usi ng password authentication by generating a "key pair," this is especially useful for automated or scripted tasks. There are different types of key pair, like having different vendors for locks, but I will stick to RSA keys for the purposes of explanation. This authentication method works by having a

"private key" and a "public key" that make up the key pair. A complex mathematical relationship exists between the two keys that cannot (easily) be reverse engineered. The private key is kept private, and on the system to login to without using a password, the public key is placed, I.E. The private key could be kept local on a desktop system, and on the remote systems a copy of the public key is placed. When a login to the remote system is attempted, SSH tests th e mathematical relationship between the public and private keys and will allow the owner of the correct private key to log in without having to ever see the private key. The fact that the private key does not need leave the local system is one of the strengths of this system.

To create a key pair, the command ssh-keygen is used with an argument of –t rsa that instructs ssh-keygen to create RSA type keys.

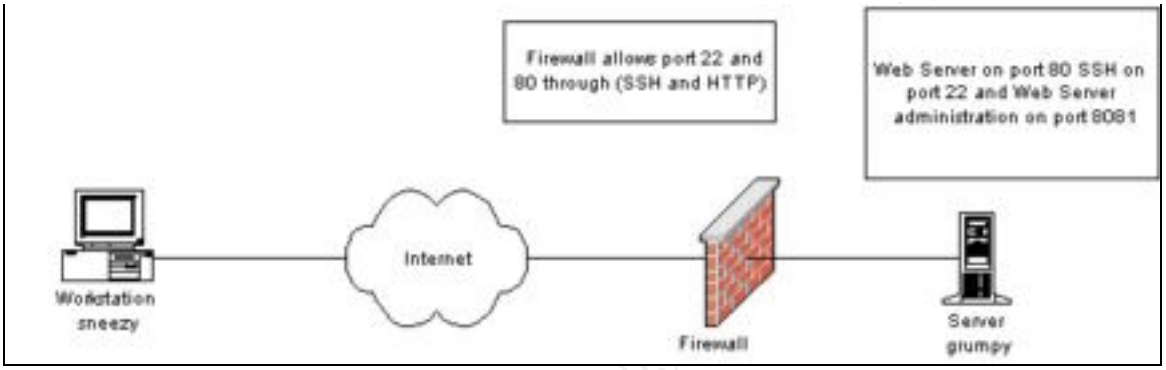
```
[mark@sneezy mark] ssh-keygen -t rsa
Generating pub lic/private rsa key pair.
Enter file in which to save the key (/home/mark/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/mark/.ssh/id_rsa.
Your public key has been saved in /home/mark/.ssh/id_rsa.pub.
The key fingerprint is:
2c:5f:05:e4:57:f1:f1:86:71:c8:9b:b4:87:50:c0:bb jbloggs@local
```

Ssh-keygen will then write the keys to ~/.ssh/id_rsa for the private key and the public key in ~/.ssh/id_rsa.pub. In the output above ssh-keygen requested a "passphrase," a passphrase can be used as protection for your private key as without a passphrase the private key is protected only by the filesystem it resides on. For example, on a UNIX system the root user would be able to read any private key, also if a home directory and hence a private key is stored on an NFS mount, then the security of the private key becomes more of an issue. If a system where ever compromised, then without passphrase protection the private keys are immediately compromised, the attacker would be able to access all the systems on the network where the corresponding public keys have been copied, without even having to type a password.

Now that the key pair has been generate d, to login to a system without using password authentication, the public key ~/.ssh/id_rsa.pub from the local system is appended into the file ~/.ssh/authorized_keys (*Note the American spelling)* on the remote server. It is important to make sure that the file system permissions on the authorized_keys file and id_rsa file are such that they are only readable and writable by the owner, otherwise SSH by default will not allow the authentication to succeed. The authorized_keys file can contain more than one pu blic key, as it may be required to connect to a server from more than one client, the public key for each client would need to be appended.

Port Forwarding

SSH can allow its encrypted link to be used to carry other connections, as well as the interactive user session. If the display variable ($DISPLAY) is set before and ssh connection to a remote system, then the remote display will automatically be "forwarded" back to the local system through the encrypted tunnel. SSH achieves this by setting $DISPLAY on the remote system to point to a special "display" which forwards the connection back to the local $DISPLAY. SSH when forwarding X through the encrypted tunnel also calls the xauth program to ensure that the remote program has the correct authentication. See man (1) xauth for details of how xauth works.



Above is shown an example of how SSH can "forward ports."

The Scenario

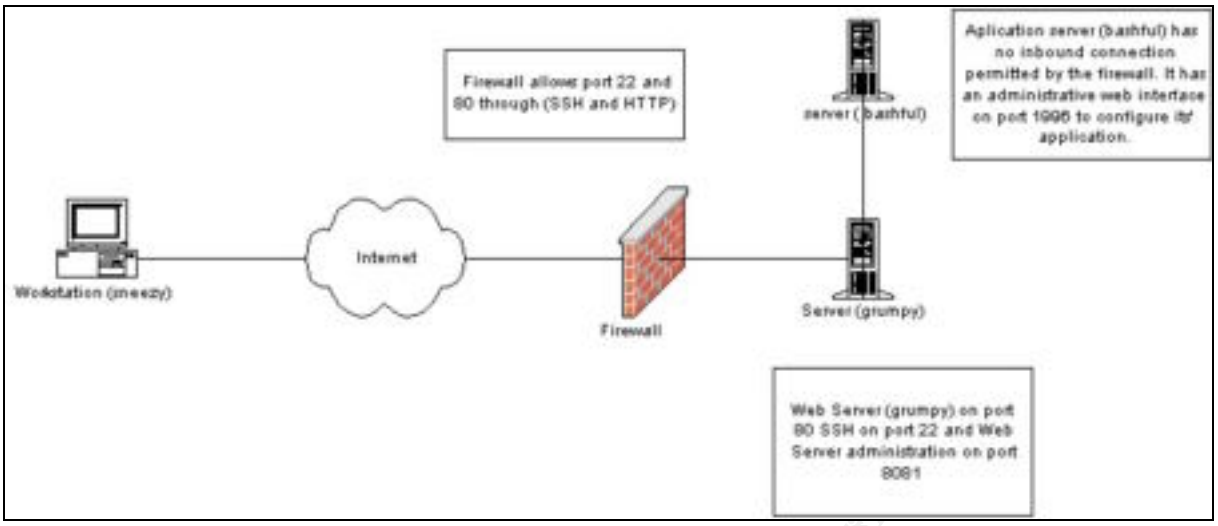
From a workstation (sneezy), it is required to use the web based administration tool on the server grumpy, but port 8081 that it listens on is blocked by the firewall. SSH can be used connect a local port on sneezy through the encrypted link to the administration port on the grumpy.

The –L argument to ssh will "forward" the local port to the remote server; **ssh –L 8081:localhost:80 81** means, on the local system port 8081 is taken through the tunnel and connects to localhost, which is on the remote system on port 8081. This could also be represented as **ssh –L 8081:grumpy:8081** which in effect is the same, as when ssh connects over to g rumpy, localhost on grumpy is grumpy.

```
[sneezy  home] ssh –L  8081:localhost:8081    -l   jbloggs     grumpy
```

When the command above is executed on sneezy and login is achieved on grumpy (as user jbloggs), a web browser on sneezy can connect to port 8081 on sneezy and ssh will forward the connection transparently to port 8 081 on grumpy. The command below when executed on sneezy will connect to the web based admin interface on grumpy through the encrypted tunnel.

```
[sneezy    home]   /usr/local/netscape/netscape http://localhost:8081    &
```



Above is shown a slightly different example of how SSH can be used to "forward ports. " From the workstation sneezy, it is required to administer an application on bashful. The application on bashful is configured via a web based interface listening on port 1996, but unfortunately the firewall does not permit any external connections to bashful. SSH will allow a local port on sneezy through the encrypted link to the administration port on the bashful, even though the firewall denies direct connection to bashful.
As already shown, by using the –L argument ssh can "forward" a local port on sneezy, through the link to a remote server: **ssh –L 8082:bashful:1996** means, on the workstation (sneezy) port 8082 is taken through the tunnel and connects to bashful on port 1996. The full command is shown in the box below, which on successful login to grumpy, will forward the port over to bashful. As far as the firewall is concerned only a connection on port 22 from sneezy to grumpy is made, but SSH does some re -direction at each end of the connection.

```
[sneezy home] ssh –L 8082:bashful:1996 -l jbloggs grumpy
```

When the command above is executed on sneezy, and authentication is succe ssful a web browser can be connected to sneezy on port 8082 and SSH will forward the connection across to bashful on port 1996. Below is shown the command to execute on sneezy to connect to the administrative web interface on bashful.

```
[sneezy home] /usr/local/netscape/netscape http://localhost:8082 &
```

File Transfer

SSH can be used to transfer files via the same secured communications channel as the interactive sessions. SSH1 has a remote copy interface for file transfers, whilst SSH2 also has an interactive FTP style interface. For the remote copy interface, the command scp is used and for the FTP interface, sftp is used. Examples are shown below for clarification.

```
[sneezy root]# scp sowers@grumpy:/home/sowers/web_server_log .
```

The above example requests the file /home/sowers/web_server_log from grumpy to be copied to the current directory. The user name sowers is used to login to grumpy. Note the username@machine notation, where a colon (:) separates the machine name from the path.

```
[sneezy root]# scp /tmp/patches.tar.z sowers@grum py:/tmp
```

The above example is to copy the file /tmp/patches.tar.z from the local system (sneezy) onto grumpy in the directory /tmp. The user "sowers" will be used to "login" to grumpy for the copy.

```
[sneezy root]# sftp sowers@grumpy
grumpy >sftp sowers@atom
Connecting to grumpy...
sowers@grumpy's password:
sftp> ls
drwxr-x--x   7 sowers   other      1024 Jul  5 11:06 .
drwxrwxr-x   6 root     sys         512 Oct 31  200 1 ..
drwxr-xr-x  11 sowers   other       512 Mar  9  2001 .dt
-rwxr-xr-x   1 sowers   other      5111 Mar  9  2001 .dtprofile
drwx------   2 sowers   other       512 Mar  9  2001 .solregis
--rw-------  1 sowers   other        49 Mar  9  2001 . Xauthority
-rw-------   1 sowers   other       962 Jun 10 09:46 .sh_history
drwxr-x--x   3 sowers   other       512 Apr 21  2001 .ssh2
-rw-------   1 sowers   other       297 Nov 15  2001 dead.letter
-rw-rw-rw-   1 sowers   other     94555 Mar  18  2002 clust_admin.pdf
drwxr-x--x   2 root     other       512 Jul  5 13:50 openssh
sftp>  ?
Available commands:
cd path                  Change remote directory to 'path'
lcd path                 Change local directory to 'path'
chgrp grp path           Change group of file 'path' to 'grp'
chmod mode path          Change permissions of file 'path' to 'mode'
chown own path           Change owner of file 'path' to 'own'
help                     Display this help text
get remote-path [local-path]  Download file
*
*
```

Above is shown how to sftp to a system, the now familiar username@hostname notation is used, and once the password for "sowers" has been entered an ftp style command prompt appears, which enables commands such as ls to be run. Files retrieved using the usual get

command or can be placed on the remote system using the put command. Refer to man (1) scp and man (1) sftp for more details

Other tricks

ssh-keygen

Above has been detailed the "keys" which SSH utilizes to prove the identity of users or of systems (authenticate them). The first time a connection is made to a system, SSH is vulnerable to spoofing, as it does not have a record of the other systems' key. In this situation, SSH will prompt the user before continuing with the connection and will display to the user the "fingerprint" of the remote systems key, to give the user a chance to make a manual comparison. The fingerprint of a key is displayed when it is created, but it can also be displayed by using the ssh-keygen command with an argument of –l ssh-keygen will prompt the user for the name of the key they wish to fingerprint. There can still be a little bit of a chicken and egg situation here, where access to the system is required to obtain the fingerprint, but perhaps secure distribution of the public keys, or fingerprints can be part of the commissioning process of each system. Refer to man (1) ssh-keygen for details.

```
[sneezy home] ssh-keygen –l
Enter file in which the key is (/root/.ssh/id_rsa): /usr/local/etc/ssh_host_rsa_key
1024 92:85:17:16:1f:65:91:b4:b4:07:e3:d6:ff:e7:43:f8
/usr/local/etc/ssh_host_rsa_key.pub
```

ssh-keyscan

The command ssh-keyscan can be used to obtain the public key of a system, and has been designed to facilitate the building of the known_hosts file Or known_hosts2 file for SSH2 keys, where SSH stores the public keys of hosts. By obtaining the public keys of the remote systems in this manner, the local system becomes less at risk to spoofing attacks, at it already has the public keys of the systems it is to connect to.

```
[ sneezy .ssh] #ssh-keyscan -t rsa grumpy >> ./.known_hosts2
# grumpy SSH-2.0-OpenSSH_3.4p1
[sneezy .ssh]
```

The command ssh-keyscan can be wrapped around with a script to obtain all the relevant public keys; the keys can then be installed onto a system as part of the commissioning process. Refer to man (1) ssh-keyscan for details.

Privilege separation

Privilege separation [7] was configured when SSH was compiled, and is designed to enhance the security of sshd, and reduce the impact of any as yet undiscovered bugs in the programming of sshd. With privilege separation, only the parts of sshd that are required by necessity to have system privileges, have the system privileges, the rest of the program runs as the lower privileged user nobody (nobody was the example used). Privilege separation also tries to confine sshd to a directory (/usr/local/ssh in the example used), so that if sshd were attacked, the attacker would be confined to the directory /usr/local/ssh and as the low privilege user nobody. Privilege separation offers protection

against specific types of attacks [8], not against cryptographic attacks for example.

ssh-agent and ssh-add

Ssh-agent and ssh-add can be used in conjunction with each other at the start of an interactive session to help manage SSH keys. This is extremely useful if the SSH keys are passphrase protected, as the key only needs to be de-crypted once for the session, which is as long as the agent remains running. The ssh-agent creates environmental variables that are used by ssh when making a remote connection, so for ssh to inherit the variables, it must be a descendant of ssh-agent. Ssh-add is used to add an "identity" into the control of our ssh-agent. For example, ssh-agent is started, and then ssh-add would be executed, for the agent to manage the identity for the user mark. The steps taken are shown below which should make the process clearer.

```
[sneezy home]# exec ssh-agent /usr/bin/ksh
[sneezy home]# set | grep SSH
SSH_AGENT_PID=8043
SSH_AUTH_SOCK=/tmp/ssh -UsIT7974/agent.7974
[sneezy home]#
```

Above, ssh-agent has been started, and is given the argument of /usr/bin/ksh, ssh-agent will therefore start a korn shell interactive session (as a child) and the environment that ssh-agent has created can be viewed using the set command. To add an "identity" to the agent, the ssh-add command is used, this makes ssh-agent hold the information from the keys, so that when an ssh session is made to a remote system, a login can be made without having to type the passphrase for the keys, or the password for the remote system. The keys are protected, as they are still stored encrypted, they are just decrypted once for the session.

```
[sneezy home]# ssh-add
Enter passphrase for .ssh/id_rsa:
Identity added: .ssh/id_rsa .ssh/id_rsa)
atom #ssh-add -l
1024 d7:5b:67:52:c2:89:e5:9e:88:42:93:be:f8:a2:0f:1b .ssh/id_rsa (RSA)
```

The above example shows the default identity being added to the agent, and then being listed by using ssh-agent –l, once the agent is managing the keys, ssh will automatically be able to use the keys for logins. The process of starting up the agent and adding the identities is best taken care of automatically within a users' profile for example, the user would then only be required to login, and then type the passphrase for the keys and the agent would take care of the keys in the background. When the identities are added ssh-add should be given an argument also of –t, which sets the lifetime of the identities. See man (1) ssh-agent and man (1) ssh-add before using this feature.

Conclusion

As the world becomes more computer literate, more computer dependant, and more connected to computers, attacks against systems will inevitably increase. A little effort spent on security will reduce the risk this threat poses to systems. Freely available and easy to use programs are becoming de-facto standards like Secure Shell, which when used properly can improve the security posture of systems.

References

1. Dunston, Duane "Encrypted Tunnels using SSH and MindTerm. " March 26 2001. URL: http://rr.sans.org/encryption/tunnels.php

2. Zwamborn, Damian "An introduction to SSH Secure Shell " May 15 2001 URL: http://rr.sans.org/encryption/intro_SSH.php

3. Open Source Initiative "The Approved Licenses." 2002. URL: http://www.opensource.org/licenses/index.php

4. Lewis, Shawn "A Discussion of SSH Secure Shell " August 4 2001 URL: http://rr.sans.org/encryption/SSH.php

5. Riser, Neil B "An Overview of some of the Current Spoofing T hreats" July 1, 2001 URL: http://rr.sans.org/threats/spoofing.php

6. OpenBSD "openssh advisories" URL: http://openssh.org/security.html

7. Provos, Niels "Privilege Separated OpenSSH" URL: http://www.citi.umich.edu/u/provos/ssh/privsep.html

8. One, Aleph "Smashing The Stack For Fun And Profit" URL: http://www.insecure.org/stf/smashstack.txt

9. Fyador "The Art of Port Scanning" 6 September 1997 URL: http://www.insecure.org/nmap/nmap_doc.html

Other references

Carnegie Mellon Software Engineering Institute "CERT Coordination Center" URL: http://www.cert.org

F-Secure Corporation "Securing the Mobile Enterprise" URL: http://www.f-secure.com/

Free Software Foundation "GCC home page" 04 October 2002 URL: http://gcc.gnu.org/

Gailly, Jean-Loup "Zlib home page" URL: http://www.gzip.org/zlib/

Linux Online Inc "Linux Online" URL: http://www.linux.org/

Netscape Communications "Netscape 7 Web Browser" URL: http://channels.netscape.com/ns/browsers/download.jsp

OpenBSD "the main OpenSS H page" URL: http://www.openssh.org/

SSH Communications Security “SSH Communications Security” URL: http://www.ssh.com/

Engelschall, Ralph S “Welcome to the OpenSSL Project” UR L: www.openssl.org

OpenBSD “Portable OpenSSH” 14 October 2002 URL: http://www.openssh.org/portable.html