Global Information Assurance Certification Paper

A practical guide to OpenSSH

GIAC Certification

GIAC Security Essentials Certification Practical

Olivier De Lampugnani

Version 1.4b (Option 1)

December 2003

Table of Content

Abstract :

The aim of this paper is:

- to provide system administrators who might not be aware of the security threat of using telnet, ftp and r-commands with a solution using OpenSSH to mitigate the risk.

to convince system administrators who may already be aware of SSH but still do not use it because they find it inconvenient, or because they lack necessary knowledge

- to use it.
- to bring them tips and techniques for every day life in an encrypted world.
- to cover many of the different problems that come with OpenSSH.

The intent is not to provide a step by step guide on how to implement SSH on all the UNIX flavors, but rather provide a part of my experience on the path to a secure remote shell environment .
Through reading this paper you will see that OpenSSH provides you with a powerful environment that needs to be fully understood in order to take advantage of its security added-value.
This paper is based on OpenSSH. OpenSSH was chosen for two main reasons: it is available on many operating systems and it is free. You should be aware that other products exist which will have more or less the same capabilities.

Introduction :

The use of clear text services such as Telnet, FTP and R-commands has been identified by the SANS Institute as number five in the most commonly exploited vulnerable services in Unix in the 2003 Twenty Most Critical Internet Security Vulnerabilities [1]. These services are generally installed and enabled by default on most UNIX flavors. They transmit unencrypted data over the Network, allowing sniffing of credentials and session hijacking.

Some clear text protocols have a secure version, like HTTP with HTTPS, or can be encapsulated in another encrypted channel through SSL wrappers or SSH tunneling, for example POP, IMAP and SMTP. Telnet, FTP and RSH (standing for Remote Shell) have their secure substitute in SSH (standing for Secure Shell). SSH is a protocol, and also a program created by Tatu Ylölnen.
Two versions exist; SSH1 and SSH2, which are incompatible one with each other. The last freely available version is 1.2.12 which is now commercialized under the brand ssh.com from Tatu Ylönen.
SSH1 and SSH2 are two entirely different protocols. SSH2 is a complete rewrite of the protocol with improvements to security and performance. It is strongly recommended to avoid using SSH1 unless there is no alternative.
A version of SSH is available, under GPL (GNU Public License), from the OpenBSD Team project under the name of OpenSSH. OpenSSH is developed for OpenBSD and has been ported to all other UNIX and Linux flavors, as well as under Windows through Cygwin.

OpenSSH is a suite of tools including: a server (sshd) and a client (ssh) equivalent to telnet and rlogin; scp, an equivalent to rcp; a secure ftp server (sftp-server) and client

(sftp); and some key management utilities: ssh-keysign, ssh-keyscan, ssh-keygen, ssh-add and ssh-agent.
All these utilities will be described in more detail later into this document.

SSH protocol

As SSH1 and SSH2 are two entirely different protocols, these two protocols are totally incompatible.
This document only covers the latter, as SSH1 was never standardized and SSH2 was conceived with security as a fundamental aspect. It has been standardized through the Internet Engineering Task Force (IETF) [2].

SSH version 2 protocol is a layered architecture made of three layers. Each layer has its own role in the SSH communication.
The Transport layer is set up at the session initiation during the establishment of the encrypted channel. It will assume such tasks as: host authentication, key exchange, compression, encryption, and data integrity.
Once the encrypted channel is established, the integrity and confidentiality of the communication are assured, and the User authentication layer can be put in place. This layer is responsible for the different user authentication methods such as Password, Hostbased, or Public key. Once the user is granted access rights, the Connection Layer can be established to provide interactive login session, file transfer and forwarding.

Caution: By default OpenSSH will initiate the session using SSH2 protocol but if it fails for whatever reason it will switch to protocol version 1. It is possible to prevent this switch using the OpenSSH configuration.
To do so change the option “Protocol 2,1” to “Protocol 2” in the configuration file “sshd_config”.

Using OpenSSH

To connect to a remote system with OpenSSH it can be as simple as with telnet.
For example, the following command:
ssh –l <UserID> <Remote_system_hostname>
will simply prompt you for a password.
In doing this, you will be connected in exactly the same way as with telnet, except that no credentials have been transmitted in clear text over the network. Anybody with a Network Analyzer tool listening to the communication will only see nonsense character chains.
But does this mean that the communication channel is secured? The answer is no.
What happens if a attacker inserts himself in the middle of the communication and makes you think you are directly connected with your server?
This is called a Man in the Middle attack (MITM), and don’t think it’s Science Fiction.
Sshmitm, a tool from the “dsniff” [3] program suite, is exactly designed for this purpose.
In fact this is not strictly correct as sshmitm only deals with the SSH version 1 protocol, but this does not mean that it is impossible to insert in the middle of the communication.
SSH provides you with a way to protect against the MITM attack through the establishment of a secure connection, data encryption of the traffic and also ensures data integrity.

Host authentication

The secure connection begins with host authentication, and is performed using a pair of public and private host keys. The private one will be used by the SSH remote server. The public one will serve all the other machines that connect to this remote server.

Three separate host keys should have been provided to you during the sshd daemon installation. You can also generate the host keys by running the following commands as root:

/usr/local/bin/ssh-keygen -t rsa1 -f /etc/ssh/ssh_host_key -N ''
/usr/local/bin/ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key -N ''
/usr/local/bin/ssh-keygen -t dsa -f /etc/ssh/ssh_host_dsa_key -N ''

The first one will generate an RSA key pair for connections over the SSH Version 1 protocol. The other two will generate RSA and DSA key pairs respectively for connections over the SSH version 2 protocol. Caution: the private "host key" is by definition private. You should take particular care to protect it and monitor access to this file against unauthorized persons as this key cannot be protected by a pass-phrase. An attacker with a valid key pair will be able to successfully mimic your remote server.
The public host keys will serve to verify the authenticity of the SSH daemon you are connecting to.
You do not need to pay special attention to protect it.

At each connection attempt to your SSH remote host the server will provide its public key for verification with the one cached on the client side in the "known_hosts" file.
This file is located in the *.ssh* subdirectory under the client home directory for a per client configuration or under the directory where the SSH configuration files are stored for a per server configuration.
If the client does not have a copy of the server host key the connection is suspended.
The client is required to verify that the provided key is really the one that belongs to the server you want to connect to.
For that the client will be provided the key fingerprint of the server public host key in order to compare it with the one really installed into the server.
To get this fingerprint from the remote SSH server if you are the system administrator issue one of the following commands, depending on the protocol and the type of key you are using:

ssh-keygen –l –f /etc/ssh/ssh_host_key
ssh-keygen –l –f /etc/ssh/ssh_host_rsa_key
ssh-keygen –l –f /etc/ssh/ssh_host_dsa_key

Otherwise you should contact your system administrator who should provide you a way to get it in a safe way. In all cases do not accept the remote server public host key unless you have verified that you can trust it.

Once you have verified that you have a matching fingerprint you can continue the connection. The server key will now be permanently added to the client list of known hosts (the “known_hosts” file).
The complete message will be similar to this:

```
        bash-2.05b$ ssh  192.168.121.134
        The authenticity of host '192.168.121.134 (192.168.121.134)' can't be
established.
        RSA key fingerprint is 70:5b:45:e8:01:9b:03:d3:d5:f6:8c:cf:82:5a:92:57.
        Are you sure you want to continue connecting (yes/no)? yes
        Warning: Permanently added '192.168.121.134' (RSA) to the list of known hosts.
        User1@192.168.121.134's password:
```

The entry in the “known_hosts” file will appear as follows:

```
192.168.121.134 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEAu7hsHxMvne7muXqIGlyRkfdpAJhKYicLwl9EEa
Lcksqr3Th2AR5tUH7fOLI1d22ag36jGhhx+kGJ4qP0O952D0KhRv6pIH4vutXbI89Wwdn
9cbHmvxR+rw43ZxDW8t94d7Q7EWIAA9EZFOZkWr8hgpUmSYUHGDwKctcZI6QNOp
E=
```

Once this is done the host authenticity verification at each connection will be processed in the background and will be transparent for the user, except if public key does not match anymore. If this happens the user will be warned that an error has occurred and the connection will be aborted except if you have specified the otherwise in the client “ssh_config” file. This is done by changing the default value of the “StrictHostKeyChecking” option to “no”.

The message will appear as follows:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
70:5b:45:e8:01:9b:03:d3:d5:f6:8c:cf:82:5a:92:57.
Please contact your system administrator.
Add correct host key in /home/user1/.ssh/known_hosts to get rid of this message.
Offending key in /home/user1/.ssh/known_hosts:21
RSA host key for 192.168.121.134 has changed and you have requested strict
checking.
Host key verification failed.
```

If you get this message take particular care of what you do next.
Do not modify your *~/.ssh/known_hosts* file until you are sure that nothing is wrong.
This message could simply mean that the system administrator of the remote host has changed the host key pair for some reason. It could also mean that some ill-intentioned person is trying to get your credentials, or trying to insert into your connection as a man-is-the-middle. In this case you should refer to you security policy for guidance on how to react according to your corporate rules.

This phase is very important. This is the reason that you should not set the "StrictHostKeyChecking" option to "no", because you could potentially miss this warning message and the communication will continue.
If you do not pay attention you could get into a full compromise of the whole session, as the data encryption begins at this point.
The server and the client will choose the symmetric key algorithms, IDEA, 3DES, blowfish, DES and AES; the first that matches in their lists.
The client will use the remote known server public key to encrypt the "session keys" that will be used to encrypt and decrypt the data with the algorithms chosen below.
The remote server will then decrypt the "session keys" using its private host key and will reply to the client a standard message encrypted with the "session key". Once the session keys have been exchanged they can initiate two secure channels; one for transmission and one for reception, using separate session keys. The remote server is the only one that can decrypt the message containing the "session keys", unless someone has managed to steal its key. The client can be sure that it is talking with the right server.

Now that the communication is safe we can go further with the authentication.

Password authentication

The default type of authentication has already been mentioned above, and it is telnet-like. You have only to provide your standard userid on the system and its associated password.
This method is secure as long as you have implemented a strong password policy, in order to avoid any blank passwords, or passwords equal to userids.
You can enforce this authentication mechanism using Kerberos 5 authentication on the remote server, as there is a patch available for OpenSSH to support GSSAPI [4].
Disabling remote access by root is also recommended as this will enforce full user identification and authorization. Additionally this can also prevent root access in the case of a compromise of the credentials. You can disable remote root access by adding "PermitRootLogin no" in the configuration file of the SSH server daemon "sshd_config" located in the config files directory.

host-based authentication

Another authentication method that could be implemented is only based on the hosts that connect, delegating the responsibility of user authentication to the client system. This is host-based authentication.

Historically SSH was developed to securely replace r-commands such as rsh, rlogin and rcp, so in the first implementation of SSH it was possible to implement this kind of authentication exactly as it was implemented with R-commands and files, using */etc/rhosts* or *~/.rhosts*, without verification of the remote host except by its IP address. A slight variation based on files */etc/hosts.equiv* and */etc/shosts.equiv* could be used instead of a per user configuration in *~/.rhosts*. This was only available in SSH version 1 and highly vulnerable to spoofing. It has now been removed from the OpenSSH release. The problem of IP spoofing has been mitigated in SSH version 2 by hardening the verification of the trusted host authenticity.
This is performed by adding a private and public host key verification of the client.
Briefly in SSH version 2 it is now performed as follows:
The client initiates the connection. The host authenticity verification mechanism, as described above in the password authentication, takes place. If this is successful the client will then negotiate with the server a host-based authentication. This is not enabled by default it has to be configured on both the server side and on the client.
If the server accepts then it will check in the */etc/hosts.equiv* like file or in *~/.rhosts* like file for an entry that matches the client host and user. If verified, the client will send a signature created with its own private host key through the ssh-keysign program.
The server will compare it to the one stored at configuration in the file "ssh_known_host" or "~/.ssh/known_hosts". It will also crosscheck the hostname associated to this host key in his server known host list to the one provided by the client. If these match, the remote access is granted.

If host-based authentication fails it will switch to another available authentication method.
As this method of authentication is only based on a trust relationship between the client and the server it will never be as secure as password authentication.
In any case this can be useful in certain circumstances, for example in a clustered environment when you use some monitoring management software or some failover script that requires a user to be able to login to remote computer in the node without a prompt for a password.
In this particular case care should be taken in hardening the client security. Otherwise you could get into a full compromise of your whole environment.
Caution: Use Host-based authentication only when the client and server systems are both in the same trusted environment, otherwise I highly recommend to consider public key authentication, which is discussed later in this document, before taking any decision.

As this kind of authentication is not enabled by default, some configuration is needed on the client and on the server in order to ensure maximum security.

Implementing host based authentication

Client side configuration:

First, you should have a valid key pair of private and public "host keys".
If these are not available see the password authentication section for how to generate a valid key pair.

Enable the host-based authentication by setting the “HostBasedAuthentication” and “EnableSSHKeysign” options to “yes” in the “ssh_config” file located in the OpenSSH configuration files directory or in the user *.ssh* subdirectory.

Server side configuration:

First, you should define which users will be able to logon using host-based authentication.
These users must exist with the same username on both machines. In this way only users on the client host that can be matched to the same username on the remote server will be granted access.
Append the client hostname or IP address in the */etc/hosts.equiv* or */etc/shosts.equiv* depending on you preferences.
These files will allow the configuration to be performed for the whole server as opposed to the RSH equivalent using the *~/.rhosts* or *~/.shosts* files which will allow a per user configuration.
Avoid using association between client systems userid and hostname in these files as this will enable the specified client userid to log as anybody on the remote server.
For more information on using these files see the “hosts.equiv” man pages [5].

Next, you should add the client public host key and its associated fully qualified domain name to the server known host list.
The server “known host” file can be a global or a per user configuration, using respectively “ssh_known_hosts” and “~/.ssh/known_host”.

Lastly, you should prepare your SSHD server daemon.
In the “sshd_config” file set the “HostBasedAuthentication” option to yes in order to enable host-based authentication.
If you want to base your authentication only on the */etc/hosts.equiv* like file set the “IgnoreRhosts” option to “yes”.
If you want to avoid the server to look into the users known host file instead of the global server one set the “IgnoreUserKnownHosts” option to “yes”.

Restart the SSHD daemon and you are ready.

Public key authentication

The third method of authentication that can be implemented is based on asymmetric cryptography; RSA or DSA depending on your preferences. As far as I know there are no big differences of efficiency between RSA and DSA. This is called Public key authentication.
In the password authentication you still need to send your secret password to the remote host but over an encrypted channel as opposed to public key authentication where no passwords circulate at all.
Instead the remote system will grant you access according to if you are able to prove you are the one you pretend to be. The private key is owned by the client and the public key is distributed on each remote server on which you need to be authenticated.
The session initiation is still handled in the same way as for the other authentication methods. Once done the authentication method starts.

The user sends the user name he wants to connect with and the associated public key to the remote server. The server will search for the provided user public key in the "authorized_keys" file located under the subdirectory *.ssh* in the home directory of the user he want to connect to. If that matches, and no particular restriction is implied in this file, the process can continue.
Now the remote server generates a random number, puts it in a message, encrypts it using the public key provided before, and sends it to the client.
The client in order to prove he has the private key has to decrypt the message, get the random number and send it as an answer to the server. If the server recognizes the random number the access is granted without any prompt for a password.
Being authenticated without providing a password can be convenient for a user but it means that if someone manages to get read access to the private key of the user he will be able to get the same rights as the user.
The private key of the user will appear as a jewel for an attacker and you never leave your jewels easily available to anyone. Instead you protect them in a safe.
To protect your safe from being opened by anybody you will chose a particularly difficult key code. In our case it is the same, but with a passphrase instead of a key code.
As you will choose a difficult key code for your safe you will have to find a passphrase that is hard to guess to protect your key but it also has to be easy for you to remember. Otherwise you are tempted to write it down somewhere in order to avoid forgetting it. If you lose it you will have to regenerate your key and redistribute it to all your target machines.
A good approach to help you find a good passphrase is the Shocking Nonsense approach explained in the Passphrase FAQ [6].
Similarly another important parameter to protect is the length of your key. It could be compared to the armor of the safe.
By default when you generate your key pair it uses a length of 1024 bits. This is considered as sufficient in most cases but you might want to use a stronger one for particularly sensitive keys.
Be aware that it will decrease the performance of the public key operation. A good source of information on choosing key length is to be found on the RSA security site [7].
To generate your key pair you can use the ssh-keygen program on any machine. If you generate them on a system different to the one that will really use it ensure that all transfers are performed securely and that the private keys are stored safely.

Implementing Public key authentication

Generate your key pair.
For this you will have to use the ssh-keygen command.
Issue *ssh-keygen –t rsa* or *–t dsa* depending on your preferences.
If you want to enforce the key length to 2048 instead of 1024 (the default value) simply add –b 2048.
You will be prompted for the filename where the key will be saved. By default it is "id_rsa" for an RSA based key and "id_dsa" for a DSA one. They are located the *.ssh* subdirectory of the home directory of the user.
You can also define the name and the path you want. If you choose something different to the default you will have to declare it in the client "ssh_config" file with the option "IdentityFile", followed with the full path to the key.

Next you will be prompted for your passphrase. Do not use an empty passphrase. Choose one that you are comfortable with as explained above.
Once you have your key pair generated distribute it to the remote servers you wish to be authenticated on in the file “authorized_keys” located under the subdirectory *.ssh* of the home directory of the user you want to be authenticated as.
The format of the “authorized_keys” is as follows:
Each key entry must be placed in the same line and is composed of four different fields. The first field is for the option where you can specify restrictions to the user associated with the key. These restrictions can be set on host or domain login restriction, command restriction, environment variables, port forwarding, and more. See the man pages of sshd for a complete description [8].
The next two fields are specific key fields. The last one is for the comment that can be added at the creation of the key. This may help identifying users and associated keys in the case that you have multiple users login to the same remote account.

You are now ready to use the public key authentication. If everything is correctly set up you will be prompted for a passphrase each time you connect using your private key. This could quickly become tiresome if you have to connect to multiple systems. You might be tempted to take a step back and remove your passphrase. Do not do so.
I highly recommend considering the ssh-agent program that can be considered as a kind of Single Sign On.
With the ssh-agent program you can load your credentials into the memory of your system using the ssh-add program. You will be prompted for your passphrase and that will be all for your local sessions. Then each time you connect to your remote host using public key authentication the ssh-agent will reply for you without any new prompt for a passphrase.
To use the ssh-agent facility just launch the ssh-agent command. It will start the agent in the background and provide you two environment variables: “SSH_AUTH_SOCK” and “SSH_AGENT_PID”. These must be exported to your local system environment in order to allow other applications to talk with ssh-agent. The problem with this is each time you start a new terminal on your local system you will have to export these variables. A simple workaround to this problem is to let the agent start your X session. This way all applications will be able to talk with the ssh-agent and the agent will end at the session logout.
To load your key into the agent use the ssh-add command followed by the path to the private key file you want to be loaded. By default it will load all the keys located in the . *ssh* subdirectory of the user that launches the command. You will simply be prompted for your passphrase.
You can now connect to your remote SSH server without a prompt, except if your public key is not listed in the “authorized_keys” file of the remote user you want to connect to, in which case you will switch to the password authentication where you will have to provide the standard password for the user.
One main problem that appears when you load your key into ssh-agent is that the key is now left unencrypted somewhere in the system. There are some things that you can do to mitigate this risk.
First you can set a lifetime for your key when you load it using the “-t “ option of the ssh-add command followed by the number of seconds you want your key active in the ssh-agent. This way once the lifetime expires your key is unloaded from the system. The

most important thing to understand about ssh-agent is that it should only be used on a machine where you could guarantee a high level of security.

OpenSSH is a powerful program that provides different and convenient ways to connect to a remote host securely, but handled in the wrong way it can be more or less the same thing as continuing using R-commands.

Now that you are aware of how to connect, let's have a look at the functionalities provided by OpenSSH.

File Transfer

scp

As well as providing a way to get a secure remote shell on a remote machine with OpenSSH you can also transfer files in a secure way between a client and a remote server, or between two remote servers.
This can be done with the scp command that securely replaces rcp with all the benefits of SSH discussed above, such as user authentication, host authentication, data encryption, and data integrity.
I will not cover the usage or give examples of the scp command as you will find it well documented in the man pages [9].

sftp

Another well known way to transfer files over the internet that is recognized to be insecure is FTP. OpenSSH provides a secure substitute with SFTP, and it obviously does not support Anonymous login. It works as a subsystem of SSH and can be interpreted as an FTP front-end to SSH commands. In OpenSSH you will have to enable it by adding the following line in the "sshd_config" file:
"Subsystem sftp <Path to your sftp-server program>"
or if you know the path to the sftp-server subsystem on the remote host you can use the sftp client with the –s option in addition to the path of the remote sftp-server.

rsync

Another interesting solution to transfer files from a client machine to a remote SSH server is rsync.
Rsync is a tool designed to transfer files in an efficient way by synchronizing only the pieces of the file that have changed since the last synchronization, rather than transferring the whole file as with SFTP or SCP. The most important point is that it is designed in such a way that it can use all the SSH capabilities. It can also transfer a whole directory, preserve their properties, can be easily scripted and does not require any special privileges to install. All of this makes rsync the ideal solution for backup or file synchronization over slow network links. For example, a simple rsync command using SSH could be:

```
rsync -e ssh user1@server.example.com:/home/user1/data /backup/user1/data
```

A good idea before any such file transfer is to do a dry run using the –n option in order to avoid any mistakes.

Forwarding

In SSH, port forwarding enables you to give additional security to an application that was not originally designed to be secure. SSH provide you a way to redirect TCP/IP traffic from one port on the client machine through the encrypted channel of the SSH session and redirect it to the specified port on the remote SSH server. In order to make forwarding work an interactive session must be established.

X11 forwarding

X11 port forwarding is a good example of this possibility. When you connect to a remote SSH server you might want to use some graphical user interface. For that you will export the display using the X11 protocol to your client machine. As X11 is not encrypted you will expose your X-windows communication, allowing it to be exploited by an unauthorized person on your network path. In order to fully benefit from SSH communication you can use the X11 tunneling capability enabled by default in OpenSSH. It is totally transparent to the user. You don't need to redirect ports, and the display variable is automatically exported at logon. In fact the only thing that needs to be done is to enable it on the server side in the "sshd_config" file.

Authentication forwarding

The ssh-agent also uses this forwarding capability to provide an authentication forwarding functionality. To enable this option use the –A option of your ssh command, or add the option "ForwardAgent" to yes in the "ssh_config" file of the server running ssh-agent. This option is interesting in the case where you need to connect to an intermediate remote server as a step to access another remote server.
The SSH agent forwarding function will provide your credentials to the final remote server through the already established SSH connection on the intermediary server. In a nutshell you create an authentication chain starting from your client station running the ssh-agent through as many hosts you need without having to spread your private key on all the servers that form part of the chain, as long as all the remote SSH servers have your public key in their "authorised_keys" file of the user you are connecting to.

Local and remote port forwarding

In addition to the built in port forwarding capabilities in OpenSSH discussed above, you can also statically forward a defined tcp port on the SSH client system to a remote tcp port on the remote SSH server, or to a system behind the SSH server. This is the local port forwarding –L option of the ssh command. You can also forward a tcp port of the remote SSH server to a tcp port on the client system, or to a system behind the client. This is the remote port forwarding –R option of the SSH program. In a nutshell you perform local port forwarding when you forward port in the same direction as the SSH connection while you perform remote port forwarding in the opposite direction of the SSH

connection. Playing with both options will provide you a way to encrypt traffic that was not designed to, such as telnet, SMTP, POP3, IMAP and more, as long as they only establish one connection. It will not work with FTP (in either active or passive modes) as it needs a second connection to be established for the FTP data transfer. And it is impossible to predict in advance the chosen TCP port that will have to be forwarded. In practice you could use local port forwarding to secure unencrypted outgoing traffic from your local network to a server behind a firewall that only allow outgoing SSH traffic, and remote port forwarding to secure incoming traffic.

Syntax of the commands:
ssh –L <local_port_to_forward>:<target_system_of_fw>:<port_on_the_target_system> <ssh_server>
ssh –R <remote_port_to_forward>:<target_system_of_fw>:<port_on_the_target_system> <ssh_server>

The *<local_port_to_forward>/<remote_port_to_forward>* can be whichever you want, as long as the client program that will connect is configured to access this port. If it is lower than 1024 it will need root access to establish a connection as only root can forward a privileged port.
The *<target_system_of_fw>* could be either the remote SSH server or a system accessible from the remote SSH server.
The *<port_on_the_target_system>* should be the TCP port number the target server daemon is listening on. For example it should be port 25 for SMTP.
The *<ssh_server>* is your remote SSH server where you should have at least user access.

Local port forwarding example:

```
client.intranet.com$ ssh –L 2323:telnet_server.example.com:23
ssh_server.example.com
```

This command will open an SSH encrypted channel from the client system to ssh_server.example.com, and will forward all traffic from the client system on port 2323 to the port 23 of the telnet_server.example.com through this channel. In this way a telnet command from the client system to its own port 2323 will provide telnet access to telnet_server.example.com, but the communication will only be encrypted between the client system and the remote SSH server but not between the SSH server and the Telnet server.
If you can't afford having a part of your traffic that goes unencrypted then you should establish a second tunnel between the SSH server and the Telnet server. Instead you can create the SSH connection between the client system and telnet_server.example.com, as long as the client can have network access to telnet_server.example.com traffic will never go unencrypted. An example of the command could be:

```
client.intranet.com$ ssh –L 2323:telnet_server.example.com:23 telnet_server.example.com
```

This example is not particularly useful as it reproduces the basic SSH functionality. But can be useful for encrypting POP, IMAP or SMTP for example.
If you use the "–g" option in the ssh command then the client will act as a gateway. All systems that have access to the client system on port 2323 will be able to telnet to the remote telnet server. Filtering on port 2323 would be mandatory to protect the remote connection. Again all traffic before the SSH client system is unencrypted.

Remote port forwarding example:

```
client.intranet.com$ ssh –R 2323:telnet_server.intranet.com:23
ssh_server.example.com
```

This command will open an SSH encrypted channel between the client system to ssh_server.example.com. And will forward all traffic from ssh_server.example.com port 2323 to the port 23 of telnet_server.intranet.com through this channel. Traffic from client.intranet.com to telnet_server.intranet.com will be unencrypted. As with local port forwarding the system target of the port forwarding could also be the SSH client itself and the –g option will make the SSH remote server to act as a gateway. Again the "–g" option should be used in association with filtering on the port to forward in order to have a minimum of control over who will be able to use this port forwarding.

A good example of port forwarding practical application is provided by Sys Admin Magazine in an article "Encrypted NFS with OpenSSH and Linux" [10]

Dynamic port Forwarding

Using Dynamic port forwarding is no more than making the SSH encrypted channel act as a socks proxy server and will enable TCP/IP traffic initiated from the client side of the encrypted channel to be automatically forwarded to the other side. In order to make this work you should use the "–D" option of the OpenSSH ssh command followed by the local port number on which the SSH client should be listening.
Recent versions of OpenSSH now support socks V5 protocol [11].
Once the SSH session is establish the SSH client will listen on the specified port.
Syntax of the command:
ssh -D <TCP_port_to_listen_on> <ssh_server>

The <TCP_port_to_listen_on> can be whichever you want, as long as the socks client program that will connect is configured to access this port. If it is lower than 1024 root access is required as only root can forward privileged port. By default the socks client will use port 1080 to connect.

In order to enable the TCP/IP traffic of the SSH client to be forwarded on the other side of the encrypted channel you should install on the SSH client system a socks client program. An example of a such a socks client for Windows is the Hummingbird SOCKS [12] and for Linux you can use Dante[13].
You need to configure your socks client to handle TCP/IP traffic destined to the foreign IP address, or IP address range, located behind your SSH server and send it to the port that the SSH client is listening on.

Caution: if the traffic is addressed to a system behind the SSH server then the traffic will go unencrypted between the SSH server and the target system. If you combine the "–D" option with the "–g" option mentioned above then you will allow all systems able to connect to your SSH client to use it as a socks proxy server without authentication, and to connect with systems on the other end of the SSH encrypted traffic. Again the traffic to the SSH client will not be encrypted.
FTP in passive mode is a good example of protocol that needs Dynamic port forwarding in order to be forwarded through SSH. In passive mode FTP opens a second channel, that it is hard to predict, to establish the data session, therefore it is quite impossible to use local or remote port forwarding to encrypt it in an SSH session.
Using Dynamic Port Forwarding is not particularly useful for FTP, as SFTP is better, but you can use this configuration to forward all kinds of application that use TCP without the need to configure each one manually. It should be noted that it can be used to forward Peer to Peer applications very easily outside of your corporate firewalls and that way bypassing your firewall rules. As this traffic is encrypted it is indistinguishable from other SSH traffic. Even a TCP scan could be performed through this configuration.
Disabling port forwarding on the SSH remote server side, by adding the line "AllowTcpForwarding no" in the "sshd_config" of your SSH server, has no added security value as a client can use their own forwarder. I highly recommend considering always blocking SSH access to your internal network from untrusted networks, and only allowing outgoing SSH traffic from your network to needed systems exclusively.

Debugging

With an encrypted protocol such as SSH it can be a little bit more difficult to analyze connection problems than on unencrypted communication, where you can use protocol analyzer tools to understand what's going on. There is a way by using the SSH debugging option that can be enabled on both sides.
To enable it on the client side use the ssh command with the "–v" option, and on the server side start your sshd daemon from a command prompt with the "–d" option.
Caution: using the –d option when running sshd will only permit one connection.
In a production system you may need to use the logging facility through syslog to debug in order not to disturb user access to SSH. In business-as-usual mode it is always good practice to log at least security and authorization messages. By default OpenSSH logs at an informational level. You can increase this verbosity by playing with the syslog facility option "SyslogFacility" and the log level one "LogLevel" in the "sshd_config" file. Another good security practice is to forward all syslog messages to a centralized syslog server for monitoring and consolidation.

Conclusion

OpenSSH is a powerful tool that can allow you to greatly improve the security of your communication as a substitute for plain text services. Unfortunately it can also be a threat to your security. As it is installed by default in most Unix based operating systems it is widely used in the field and it is becoming a very popular target for hackers that make them a good vector to potentially get into most operating systems.
As an example: in July 2002 some copies of the source code for the OpenSSH package were modified to contain a Trojan horse [14].

Also, OpenSSH is not exempted from security vulnerabilities. You can find on the OpenSSH web site a list of all the security vulnerabilities discovered since the beginning [15]. The only way to contain this threat is, as with any program, to keep yourself informed about security vulnerabilities through registration to a security mailing list and to update your system accordingly.

Secure Shell appeared in the SANS Institute 2003 Twenty Most Critical Internet Security Vulnerabilities [1] because of the risks produced by bad patch management and poor configuration. These were the main reasons that made me decide to write this paper. A badly designed OpenSSH environment is a threat that should not be under-estimated. As you have seen throughout this paper, OpenSSH is relatively easy to use, but depending on the way it is used it can be easy to misconfigure, which may give an intruder easy access to your systems.

As an example; if you use one password to connect to your favorite server using telnet, or another plain text protocol, and you use this same password on your secure server where you are only allowed to connect using OpenSSH: do you think that OpenSSH will be of any help?

There are two key points to protect against that: education and security policies. Education of the people that will implement and maintain your infrastructure is required as they need to fully understand how it works in order to put the appropriate bricks in the security wall defenses that you are building with OpenSSH.

The users of this infrastructure need also to be educated, as they will be the cement of this wall. If they fail to understand what good practice is in keeping their communication secure, such as protecting their credentials, or their private key, or ensuring that they are connected with the right server for example, then your security wall will be like a sieve.

References:

[1] SANS Institute, "The Twenty Most Critical Internet Security Vulnerabilities". Version 4.0 October 8, 2003
URL : http://www.sans.org/top20

[2] Internet Engineering Task Force, "Secure Shell (secsh)" November 3, 2003
URL: http://www.ietf.org/html.charters/secsh-charter.html

[3] Song Dug, "Dsniff".
URL: http://www.monkey.org/~dugsong/dsniff/

[4] Simon Wilkinson, "Kerberos/GSSAPI support in OpenSSH"
URL: http://www.sxw.org.uk/computing/patches/openssh.html

[5]] FreeBSD, "Hypertext man pages". hosts.equiv
URL:
http://www.freebsd.org/cgi/man.cgi?query=.rhosts&sektion=5&apropos=0&manpath=OpenBSD+3.3

[6] Grady Ward, "Passphrase FAQ" Version 1.0. 2 October 1993.
URL: http://www.unix-ag.uni-kl.de/~conrad/krypto/passphrase-faq.html

[7] RSA Laboratories, "Frequently Asked Questions About Today's Cryptography 4.1"
URL:http://www.rsasecurity.com/rsalabs/faq/3-1-5.html

[8] FreeBSD, "Hypertext man pages". sshd
URL:
http://www.freebsd.org/cgi/man.cgi?query=sshd&apropos=0&sektion=0&manpath=OpenBSD+3.3&format=html

[9] FreeBSD, "Hypertext man pages". scp
URL:
http://www.freebsd.org/cgi/man.cgi?query=scp&apropos=0&sektion=0&manpath=OpenBSD+3.3&format=html

[10] James Strandboge, Sys Admin Magazine. "Encrypted NFS with OpenSSH and Linux". March 2002.
URL: http://www.samag.com/documents/s=4072/sam0203d/sam0203d.htm

[11] Network working group, "Request for comment 1928, Socks protocol version 5". March 1996
URL: http://www.ietf.org/rfc/rfc1928.txt

[12] Hummingbird, "Hummingbird SOCKS"
URL: http://www.hummingbird.com/products/nc/socks/index.html

[13] Inferno Nettverk A/S, "What is Dante"

URL: http://www.inet.no/dante/

[14] OpenSSH, "Security advisory (adv.trojan)". 1 August 2002
URL: http://www.openssh.com/txt/trojan.adv

[15] OpenSSH, "Related security vulnerabilities list"
URL: http://www.openssh.org/security.html

Bernard Perrot, "Sécuriser ses connexions avec ssh". version 1.01
URL: **http://www.security-labs.org/full-page.php3?page=405**

Daniel Robbins, "Common threads: OpenSSH key management, Part 1". 1 July 2001
URL: http://www-106.ibm.com/developerworks/linux/library/l-keyc.html

Daniel Robbins, "Common threads: OpenSSH key management, Part 2".1 September 2002
URL: http://www-106.ibm.com/developerworks/library/l-keyc2/

Daniel Robbins, "Common threads: OpenSSH key management, Part 3". 1 February 2002
URL: http://www-106.ibm.com/developerworks/library/l-keyc3/