SSH exercises

1. Login to the NOC machine

We have created an account for each of you on the central NOC machine, and given you the username and password in class. You will now use ssh to login to this machine. This is very straightforward:

```
$ ssh username@noc.e0.ws.afnog.org
```

Here is what your session should have looked like, more or less:

```
$ ssh foo@noc.e0.ws.afnog.org
The authenticity of host 'noc.e0.ws.afnog.org (196.200.219.125)' can't be established.
DSA key fingerprint is 8a:d2:68:4b:4b:a5:1c:b8:b0:24:c2:51:9b:fc:a0:85.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'noc.e0.ws.afnog.org' (DSA) to the list of known hosts.
foo@noc.e0.ws.afnog.org's password: <enter password here>
... normal 'message of the day'
$
```

If everything is successful, you will be at a shell prompt on the NOC machine, and you can run commands there.

NOTE: The "fingerprint" is a unique identity for the server - it's actually a hash of the server's public key.

The first time you connect, ssh has no way of knowing whether you are connecting to the real machine, or if a man-in-the-middle is intercepting your communication. All it can do is to retrieve the public key belonging to the remote side (which may be the real machine, or if you are very unlucky it may be an attacker)

If there is any doubt that the communication is secure at this stage, then you can phone up the server administrator to verify the fingerprint. They can do this by typing

```
$ ssh-keygen -l -f /etc/ssh/ssh_host_dsa_key.pub
```

and reading the result back over the phone to you.

For now we'll trust that our communications are not being intercepted (they are within our lab, after all), so just say 'yes' to the initial prompt that you receive.

The server's public key is then recorded in *~/.ssh/known_hosts* on your own machine - you can examine this file yourself. From now on, every time you connect, the remote-side's public key is compared with the one which was learned during the first connection. If they are the same, you can be *sure* that you are connecting to the same machine.

If they are different, this means that a man-in-the-middle attack might be underway and the connection will be dropped. (Or, it might just mean that the remote machine was reinstalled and its key has changed; it's up to you to phone up to find out if that has happened)

> There are free ssh clients for Windows: "putty" and "teraterm" are two examples. So even if your laptop runs Windows, you can still login using ssh to your Unix servers.

Once you are logged into the NOC machine, you can check who else is logged in ('w' or 'who'). If we enable the talk daemon for you, you may even be able to chat to them:

```
$ talk otherusername
```

When you have finished looking around, log out. (Note: your account is purposely *not* in the wheel group! :-)

2. Set up sshd on your system

You will now set up sshd to allow remote clients to login to your system.

Remember that you must first specify in /etc/rc.conf that a service is enabled, before you can start it using a script located in /etc/rc.d/. The ssh service is an excellent example of this. So, to start the ssh daemon (server) you must first edit the file /etc/rc.conf:

```
# vi /etc/rc.conf
```

Add an extra line which says

```
sshd_enable="YES"
```

You can do this by moving to the end of the file, hitting 'o', typing the text, then hitting ESC. Then save using :wq

Notice that if you type

```
$ grep ssh /etc/defaults/rc.conf
```

you'll see:

```
sshd_enable="NO"                # Enable sshd
sshd_program="/usr/sbin/sshd"   # path to sshd, if you want a different one.
sshd_flags=""                   # Additional flags for sshd.
```

So, by putting 'sshd_enable="YES"' in /etc/rc.conf you have overridden the default setting for this service, that is, it's disabled unless you say otherwise.

Now you still need to start the ssh daemon. To do this type:

```
# /etc/rc.d/sshd start
```

That should start it running. The first time it may ask you to type random characters; if so type lots of garbage until it is happy. This is to help prevent attacks guessing the internal state of the random number generator.

To verify ssh is really running, try these commands:

```
$ /etc/rc.d/sshd status
$ ps -auxw | grep sshd
```

Did you notice in the second command where the actual sshd program file is located?

Finally, you can try connecting to your own machine; connect using the username and password you created for yourself on day 1.

```
$ ssh username@localhost
```

"localhost" translates to IP address 127.0.0.1, which means your own machine.

You should again get a prompt asking you whether you're sure the fingerprint is correct; since this time the connection doesn't even leave your machine, you can be pretty sure your traffic is not being intercepted! (Or you can verify the fingerprint in another window, using the command given above). In any case, say 'yes' when asked if you wish to continue.

Once you've convinced yourself that you've logged in successfully, type 'exit' to logout.

3. Create accounts for your neighbours

Now you will create accounts for one or more of your neighbours, so that they can login to your machine:

```
$ pw useradd <username> -m
$ passwd <username>
```

Give them the username and password you have chosen, and get them to login to your machine with ssh. Then get them to assign you a username and password on their machine, and you can login to their machine.

> Question: would you want to put their account into the 'wheel' group? Why? Or why not?

Once you've finished, remember to logout. If you forget, you may end up typing commands into a shell on *their* system rather than on your own!

> Note: sshd by default does not allow remote users to login as 'root'. This is good practice: it means you need to connect to your server as a normal user, and then use "su" to become root.
>
> Try it - that is, try logging into one of your neighbours machines as root. You know their root password!
>
> Should you need to enable logins as root, you can uncomment the setting "PermitRootLogins" in /etc/ssh/sshd_config, and set it to "yes". However we strongly recommend you do NOT do this. If/when a security hole is found in ssh, you don't want someone to gain access to your machine as root simply because you allowed root logins. And you also don't want people to be able to guess root passwords until they stumble across the right one.

---

4. Set up ssh private key authentication

This is an extra exercise, if you have spare time available. The objective is to set up secure, authenticated access to your neighbour's machine using a private/public key pair, which you will use *instead* of a password to login. We will also see how to use 'scp' to copy files between machines securely.

Firstly, you need to create yourself a private/public key pair. We will choose a 2048-bit key for good security, and we will choose *rsa* keys for ssh protocol version 2. (The other choices are *dsa* keys for ssh2, and *rsa* keys for the old ssh1)

```
$ ssh-keygen -b 2048 -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/u/home/lists/.ssh/id_rsa): <just hit Enter>
Enter passphrase (empty for no passphrase): <type a passphrase>
Enter same passphrase again: <type it again>
The key fingerprint is:
7d:04:51:06:91:7c:7d:46:69:17:f4:ac:e0:c8:e4:bd you@pcn.e0.ws.afnog.org
```

A 'passphrase' is like a password. The difference is that a password is shared between two parties (e.g. you know your password, and the remote machine validates it); this passphrase is only used on your own machine, to encrypt your private key, and is never told to anyone else. It must be at least 5 characters long, preferably much longer to make it harder to guess.

This process creates two files: ~/.ssh/id*rsa and ~/.ssh/id*rsa.pub, which are your private key and your public key respectively.

Now, in order to login using this key pair, you need to install your *public* key on the remote host. Use the username and password on your neighbour's machine which you've already been given initially. Copy your public key file across like this:

```
$ scp id_rsa.pub username@pcother.e0.ws.afnog.org:id_rsa.pub
Password:
```

What's that? scp is 'secure copy', for transferring files using ssh. You can see it's similar to a normal copy (cp from to), except you can refer to files on a remote server with a colon, as hostname:filename

OK, now ssh in to the other machine (at this stage you still have to use your password to authenticate). Check that the file id*rsa.pub was copied across. You now have to create a directory called '.ssh' in your home directory, and install the public key into a file called .ssh/authorized*keys

```
$ ssh username@pcother.e0.ws.afnog.org
Password:
[you now have a shell on remote machine]
$ ls -A
id_rsa.pub
$ mkdir .ssh         # if .ssh directory doesn't already exist
$ mv id_rsa.pub .ssh/authorized_keys
$ exit
[you should now be back on your local machine]
```

Now you should be ready to go. ssh in again, and instead of being prompted for the password, you should be asked for the passphrase of your public key:

```
$ ssh username@pcother.e0.ws.afnog.org
Enter passphrase for key '/usr/home/yourname/.ssh/id_rsa': <enter passphrase>
```

If all is well, you should be logged in successfully.

This might *look* very similar to a normal login, but it's very different. In particular:

- You are not using the password in the other system's password file. It could be changed, or even removed, and you could still login.
- If you want to change the passphrase on your key, you do it in one place (on your own system), even if you use this key to login to hundreds of different systems
- The remote system can *disable password logins* completely in /etc/ssh/sshd_config. This is an excellent idea for security, as it completely eliminates brute force and dictionary attacks on your login passwords.