Remote Terminal Services

COSC301 Laboratory Manual

> **Note**
>
> Note that you should do this lab on your Client1 and Server1. Both of them should be up running, assuming Server1 can well function as NAT for Client1 already.
>
> You are going to test **ssh** using a new server machine called `vertex.otago.ac.nz`. If your Server1 does not work properly as NAT, your Client1 should use an extra adapter as NAT to access `vertex.otago.ac.nz`. The username and password on `vertex.otago.ac.nz` are the same as your university account.

The Secure SHell (SSH) is a very powerful tool for administering both servers and clients. The server side is generally found on Unix like machines or other devices with a dominant Command-Line Experience (eg. enterprise-grade routers), though there are clients for most operating systems in use today.

The most basic use of SSH is as a secure replacement for **telnet**. As SSH is such a useful and powerful tool for admins and users alike, half of today's lab will be on using **ssh**, and will be done on your Client1. The other half will be done on your Server1, and will be about administering SSH. A significant part of the lab manual is for your reference (labelled as optional), should you want to use this outside of the lab.

```
Client1$ ssh username@vertex.otago.ac.nz
The authenticity of host 'vertex.otago.ac.nz (139.80.32.49)' can't be established.
ECDSA key fingerprint is SHA256:z9M2TMCOyl0hCrcsvMcxLevUs7xEs0Pw/bsA7Fg94GU.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'vertex.otago.ac.nz,139.80.32.49' (ECDSA) to the list of known hosts.
Password: This is not echoed.
The following text is the message of the day /etc/motd
Last login: Thu Feb 21 15:55:02 2019 from somemachine.otago.ac.nz
This system is managed by CFEngine 3
And now we get our interactive shell.
theauthor@vertex:~ logout
Connection to vertex.otago.ac.nz closed.
Returned to where we started.
Client1$
```

1. Logging in Without a Password

In the lecture on SSH you learned about public-key authentication so let's try that now. Do this on your local machine.

> **Note**
>
> Note that if you've done this before, you would know your old keys will be overwritten, so you may want to move them out of the way for the duration of this lab. You can use this command to move the `~/.ssh` directory to `~/.ssh.off`. Make sure to move the old keys back if you want to remove the effect of this lab.

```
Client1$ mv ~/.ssh{,.off}
```

1. Create yourself a keypair. We'll create a RSA key with 2048 bits. Be sure to use a strong passphrase. Press **Enter** to accept the default location when asked where to put the key. Passphrases, unlike passwords, can contain spaces, hence their name. A good passphrase will have plenty of letters, mixed case, numbers, and symbols.

```
Client1$ ssh-keygen -b 2048 -t rsa
Generating public/private rsa key pair.
Just press enter for this next one to accept the default.
Enter file in which to save the key (/Users/theauthor/.ssh/id_rsa):
Use a strong passphrase! These won't be echoed.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/theauthor/.ssh/id_rsa.
Your public key has been saved in /Users/theauthor/.ssh/id_rsa.pub.
The key fingerprint is: Yours will be different…
1a:91:52:bf:41:78:7b:bc:19:a7:c1:ea:1a:cd:1f:2d Comment
The key's randomart image is:
+---[RSA 2048]----+
|o.o o.o . o.     |
| = o = . o       |
|. X + B o        |
| = * O *         |
|. . o O S        |
|.. . + o o       |
|=.. . . o .      |
|E+ .   . .       |
|O=.     .        |
+----[SHA256]-----+
```

   If you mistype the passphrase, you may need to do it all over again, and again, until you get it right. It's not uncommon for us to have to have three goes before we get it right. Some passphrase is very long; longer even than the length of the passphrase entry dialog box.

   If you find that it is taking a long time to create the key, the operating system entropy pool (which feeds the random-number generator) may have run dry. You can "stir in" some randomness by doing something like moving the mouse a lot, or causing a lot of system activity. The kernel uses the timings of such things as a source for entropy.

Important

Remember the private key `~/.ssh/id_rsa` must be kept secret, so it should be encrypted with a suitable passphrase.

2. Now we are ready to install the public key of the key-pair that we generated on Client1 into your account on Vertex. Still on Client1, output the public key, and copy it to the clipboard:

```
Client1$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3Nza…  copy this long line to the clipboard
```

   Now SSH into Vertex, and edit the file `~/.ssh/authorized_keys`, adding the public key to this file, which adds your public key to the list of keys allowed to access your account. Note that you will likely need to create the directory, and beware that the filename uses American English.

```
vertex$ mkdir -m 0700 ~/.ssh
```

Don’t break the lines!

The lines in this file are very long, and would wrap onto multiple lines in your terminal. Ensure that your editor does not insert newlines to break the lines, otherwise the entire file will be unusable.

This is particularly important for users of the **nano** editor. If you really want to use **nano**, use **nano -w**, which turns off wrapping.

```
$ vim ~/.ssh/authorized_keys   American English!
Paste in the public key into a single line.
ssh-rsa AAAAB4Nza…
```

3. Now log out of your SSH session from Vertex. You can login again without using a password.

```
Client1$ ssh username@vertex.otago.ac.nz
```

1.1. [Optional] Using the Mac OS X Keychain for SSH keys

When Apple released Mac OS X 10.5 “Leopard” they integrated the **SSH** agent with the local Keychain subsystem, which is the part of the system responsible for remembering passwords since then.

The system Keychain is generally (rarely is it otherwise) encrypted with the users login password, and here we shall take pains to try and get to you understand something. Which is likely to be stronger: a typical users login password or a typical users **SSH** passphrase? Actually, this is something of a moot point due to the *typical* qualifier: a typical user will likely have as poor of a password and passphrase as they can.

A security-conscious user on the other hand will have a fairly good password and hopefully an even better passphrase. When a key is added to the users Keychain, it is first decrypted by asking the user for their passphrase, and then adding the decrypted private key to the user’s Keychain. Remembering that the user’s Keychain is encrypted on disk with the user’s login password, and that a well-chosen passphrase should be stronger than a well-chosen password, you should be able to see that we have likely reduced the strength of the protection for the private key by using the login password to encrypt the Keychain.

Below are the steps to add your SSH key into the Keychain of MacOS and to enable the **ssh-agent** to handle the key for you automatically. Once successful, you can login to vertex without either password or passphrase.

1. First, store the key in the Keychain.

```
workstation$ ssh-add -K ~/.ssh/id_rsa
Enter passphrase for /home/User/.ssh/id_rsa:
Enter your key passphrase and you won't be asked again!
Identity added: /home/User/.ssh/id_rsa (User@workstation.otago.ac.nz)
```

2. Create a `~/.ssh/config` file and add the following lines.

```
Host *
```

```
UseKeychain yes
AddKeysToAgent yes
IdentityFile ~/.ssh/id_rsa
```

where it tells SSH to use the Keychain and to add the following keys to the SSH agent. Note that you can add more keys with additional lines of `IdentityFile`.

1.2. [Optional] Logging in Without a Password From Windows

This section is optional; but it is useful to know in order to access a server from off-campus using a Windows machine. Public-key authentication can be really useful, and sometimes it can even be required (to prevent dictionary attacks on passwords); we shall see more of this in the section on Section 7, "[Optional] Preventing Dictionary Attacks".

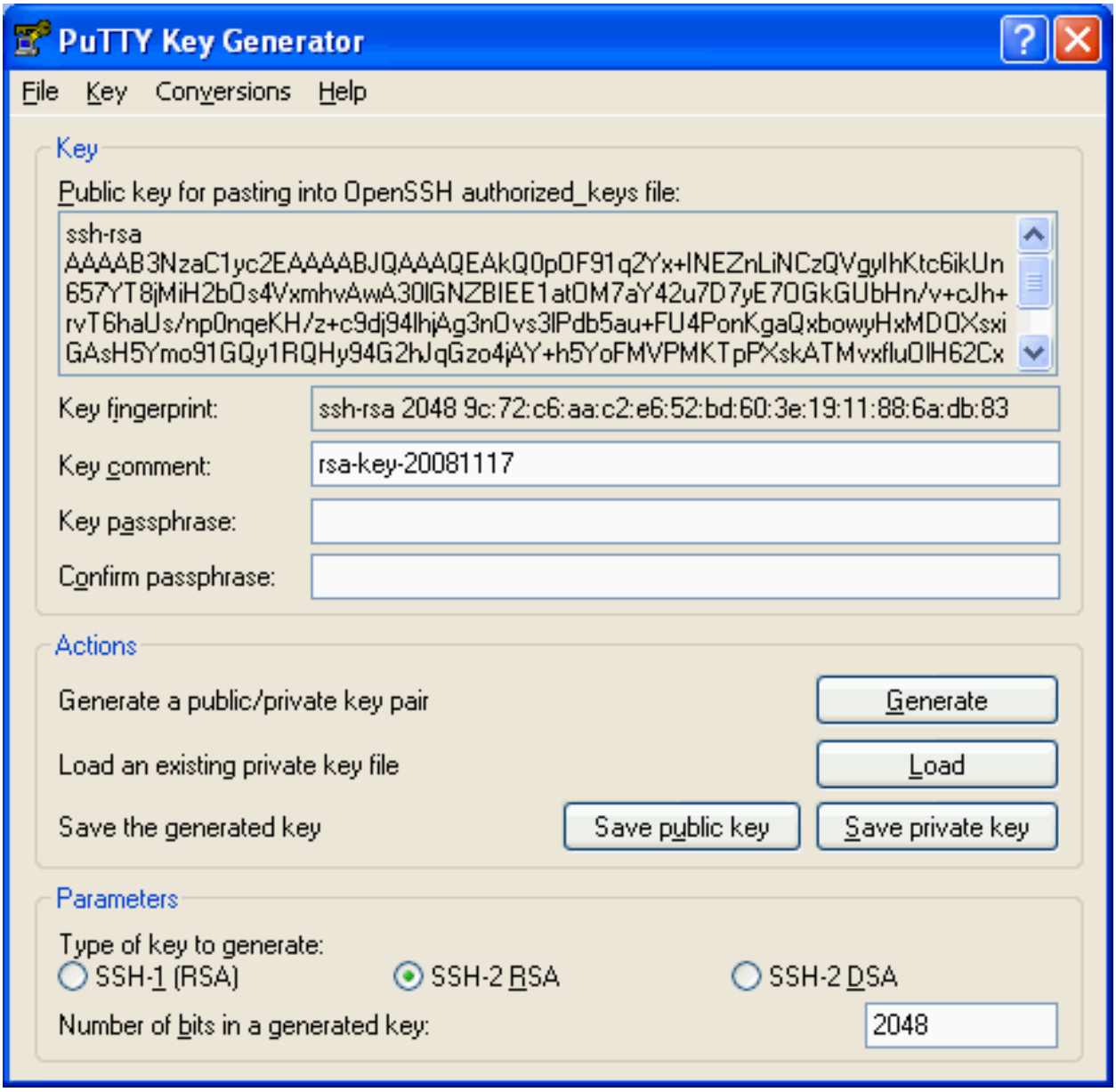
Windows, in the past, did not supply SSH client software, so third party software must be used. With the recent Windows 10 update there is a built-in SSH client, however, we have not tested it. PuTTY is a very common application for this task, which does a good job and is freely available (another option is SecureCRT, however this costs money). PuTTY is also available on other platforms such as Linux; its position in that market can be seen as a GUI SSH client.

In this section we shall look at PuTTYgen for generating our key-pair; PuTTY for our SSH client program; Pageant for our SSH Key agent and WinSCP for transferring files using **SFTP**. All software mentioned here is also Open Source software.

You could reuse (import) an existing key-pair, but for this section we shall assume that we will create a new key-pair for ourselves. To do this we shall use PuTTYgen. Launch PuTTYgen from within the Start menu (you'll find it under the PuTTY folder). Select the SSH-2 RSA option (it will likely already be the default), and make the number of bits in the key 2048 instead of 1024, which is a little short for today.

Click Generate; you will notice the helpful progress meter. You will need to move the mouse as instructed… it appears that this is the only source PuTTYgen gets it entropy from. When this has completed, you should see a window similar to the following:

> **Tip**
>
> Despite the screen-shots being from Windows XP the proces (and interface) has remained unchanged over the years.



What you should see after initial creation of a key-pair using PuTTYgen. The default comment is not useful and should be changed, and the passphrase needs to be set.

Change the comment to something more meaningful, and set a good passphrase. When done, you need to click Save public key and also Save private key. There appears to be no standard location for this, such as `~/.ssh` on Unix hosts, so I suggest you create a folder `My Documents\SSH` and store them in there. Give the public and private key the same base name, but distinguish them by putting `public` and `private` at the end of the name (not as the extension).

Before you can use the key to log into an OpenSSH server, you must first convert the public key to the format expected by OpenSSH. Fortunately, PuTTYgen recognises this as a very common requirement and in its main window has the public key which in the correct format. Select and copy the public key.

Now login to your server using PuTTY. You will want to make some changes and set up a session for the particular server. This includes data such as: what host you wish to connect to; what username (if any) you want to log in as by default; which private key you want to use to authenticate and whether agent forwarding should be enabled. Save the session using the Save button in the Session pane. The first time you log into a machine from the client you will be prompted regarding the server's fingerprint, as the client will not have seen the server's certificate before and by default has no way of knowing if it is legitimate. Click Yes, noting that it is right to be suspicious if you don't expect this dialog.

It is best to check the fingerprint of the key to ensure that you are connecting to who you think you are connecting to.

You will see that PuTTY outputs `Server refused our key` on the terminal. This is because we have yet to install our public key in the target account's `~/.ssh/authorized_keys` file. Open this file in an editor and paste your key into it ensuring it is stored as a single line.

```
Server refused our key
Using keyboard-interactive authentication.
Password:
```

The server refuses our key because our public key has not been installed into the account.

Tip

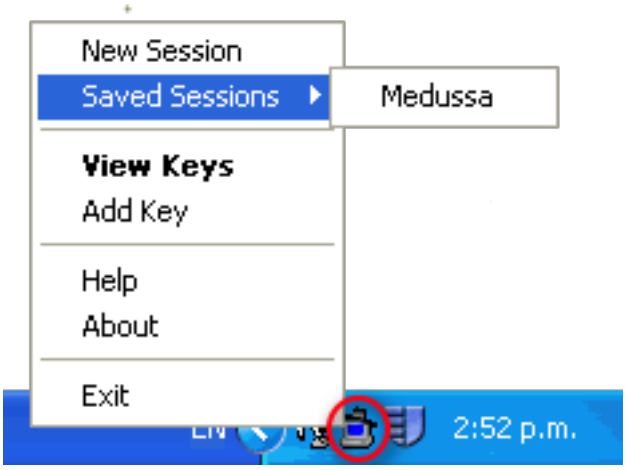
To paste in PuTTY simply right-click. This is counter-intuitive (note that middle-click doesn't work, but a Windows user wouldn't expect that to work), and Ctrl-V doesn't work because that is passed through the SSH channel.

Log out when you are done. Now when you log into the server again you should be asked `Passphrase for key "key comment"` rather than for the password. We will be asked this each time because we're not using a key agent (Pageant) at this time; let's remedy this now.

```
Authenticating with public key "Cameron CKERR-XP"
Passphrase for key "Cameron CKERR-XP":
```

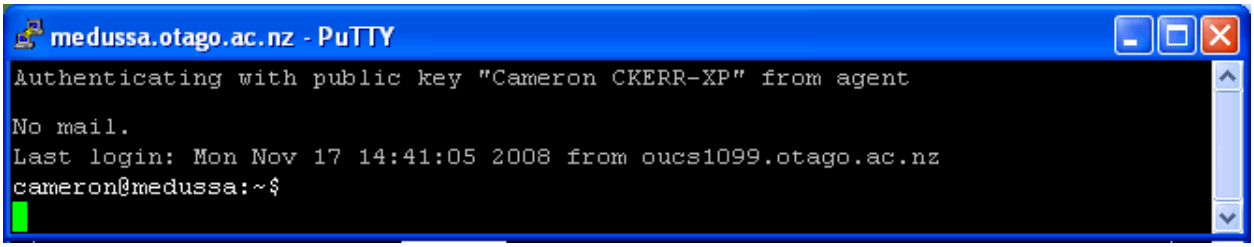
We are asked for our passphrase each time because there is key agent currently running.

Start Pageant from within the PuTTY menu of the Start menu. A new icon will appear in your system tray. If you right-click on this icon you will get a menu from which you can interact with Pageant or start any saved PuTTY session. Use the View Keys item to add the key to your key list; you will be asked to unlock your private key using by entering your passphrase.



Pageant is software that caches unlocked private keys; it plays the role of the key agent.

Now, when you use PuTTY to connect Pageant will be consulted for the unlocked private key instead of prompting the user for the passphrase.



Now that our private key has been unlocked and loaded into the key agent our SSH client doesn't need to ask the user.

Now let's try WinSCP, which is made by different people. WinSCP also includes Pageant and PuTTYgen as part of its installation, so it should work.

Launch WinSCP. Fill in the entries for hostname, username and private key (*not* password). We would also suggest setting the SSH protocol version to 2 only and only selecting Enable compression if you are going over an Internet link or WAN. Click Save… to save your session details. Then click Login to log into the server.

You will notice that a different server key cache is used, so you will need to tell WinSCP to trust the server's key since this is the first time you will have connected to the given server using WinSCP on this machine. WinSCP is not a bad program, but it does have one major annoyance: it doesn't have good support for Unicode filenames… the WinSCP program was not written as a Unicode application, and it would apparently take a lot of work to make it so.

2. Using ssh

Now that we've got fast, secure, and convenient access, the world (or at least, our network) is our oyster. Lets try using **ssh** in various ways. We've already seen how we can get a simple terminal, how about something more exciting? Try these commands.

- Run a command on a remote host and receive its output.

```
Client1$ ssh username@vertex.otago.ac.nz uptime
Banner removed if there is a banner set up on vertex. If you don't see a banner it is ok.
 11:29:06 up 41 days, 33 min, 10 users, load average: 0.00, 0.00, 0.00
```

- Interactive programs often require a terminal (or rather, a psuedo-terminal)

```
Client1$ ssh -t username@vertex.otago.ac.nz top
Banner romoved.
The 'top' program should run. Type q to quit.
```

- What about an X11 program? We can do that too. This requires LAN speeds to be really effective, although with compression turned on can also be used over a broadband internet link. Servers have this turned off by default.

```
Client1$ ssh -X username@vertex.otago.ac.nz xclock
Banner romoved.
A clock should pop up on your display.
It may not work on MacOS if the X11 server is not available.
```

- Banners are written to stderr, so we can redirect that if we really don't want to see banners (note though that this might also cause other things to be ignored, which could be bad). In this case we're counting the number of lines in the output (which as it happens, isn't very interesting at all.)

```
Client1$ ssh username@vertex.otago.ac.nz uptime 2>/dev/null \
> | wc -l
      1
```

- **ssh** is just another process, so you can still easily pipe the output. In the command `local_cmds` **|** **ssh ...** `remote_commands`, the output of `local_commands` is passed to the local command **ssh**. **ssh** logs into the remote end. On the remote end, the remote SSH (**sshd**) starts `remote_commands`. The local and remote side send any data between them, essentially gluing the output of `local_commands` to `remote_commands`.

  This example shows how you can use **ssh** to send a very simple backup (the output of **tar** in this case) to a remote machine. The command **cat > backup.tgz** will be run on the remote end. We need to escape special characters in the command to prevent the shell from interpreting them on the local side.

```
Client1$ tar -zcv some_directory | ssh username@vertex.otago.ac.nz \
> cat \> /tmp/$USER-backup.tgz
$USER isn't escaped, so it will be expanded locally.
```

- What if the command we want to run remotely contains shell meta characters? In the first command below, the * is expanded on the local side, expanding to a set of files particular to the local machine. In the second command, the * is escaped, making it be expanded on the remote side, which is what we want.

  You can get some really complex commands going; for complex commands, you should put them into a script, if you value your sanity, as the amount of escaping that is required grows wildly.

```
Your output for these commands will look different.
Here the * is expanded on the local side.
Client1$ ssh username@vertex.otago.ac.nz du -sh /tmp/*
```

```
Your output will differ, but it will still likely fail.
du: cannot access `/tmp/501': No such file or directory
…
Here the * is expanded on the remote side.
Client1$ ssh username@vertex.otago.ac.nz du -sh /tmp/\*
0       /tmp/file-on-vertex
4.0K    /tmp/another-on-vertex
…
```

- What about running **ssh** in the background? It may need to ask you for authentication (either a password, of a pass-phrase if it can't get it from an SSH Agent), so we can't use & to background it normally, so we inste-ad use the option `-f`, which tells **ssh** to go into the background only after it has completed the authentication process.

```
Client1$ ssh -f -X username@vertex.otago.ac.nz xload
```

- When scripting, you generally don't want **ssh** asking for any interaction from the user, otherwise it would hang indefinately waiting for input. We can use a particular option to tell **ssh** not to do anything that requires interaction. If it needed to, it would fail immediately.

```
Client1$ ssh -o 'BatchMode=yes' username@vertex.otago.ac.nz uptime
 11:29:06 up 41 days, 33 min, 10 users, load average: 0.00, 0.00, 0.00
```

- Things not working as expected? The first thing to try would be turning on some debugging messages. If this or the system logs on the server don't reveal the problem, then you can start the server with debugging turned on. Consult sshd(8) for more information. In a production system where others are expecting to use the SSH service, you will probably want to start the debugging server on a different port (but check firewall configuration) so other users can continue to log in normally… assuming of course SSH is currently usable.

  On the client-side, you can enable more verbose output using `-v`, although to be honest in our experience they don't tend to be very useful for solving most authentication-related problems.

```
Client1$ ssh -v username@vertex.otago.ac.nz
…
```

- Sometimes, you need to abort a connection. This is most common when you've lost your network connection (eg. you've forgotten to log-out before disconnecting from the network on your laptop). You can't use **Ctrl+C**, because that gets passed through to the remote side.

  Instead, type the sequence **Return ~ .**. That's three keys, one at a time, and remembering that **~** also needs **Shift**. If you make a mistake, you must start over, as a backspace will break the sequence. This key sequence will abort the session and return you to your local prompt. The **~** is the escape character, and is only recognised at the beginning of a line (hence the **Return** first). The dot after the escape character says to abort the connection. Another **~** just sends a ~. You can see a list using **~ ?**

3. scp & sftp

scp is a service that is provided by the SSH protocol, and its all about copying files (and directories). Its not as flexible as **ssh** however, and slower compared to unencrypted traffic, but still useful for the task it was designed for. You can copy remote to local, local to remote, local to local, and even remote to remote, although that's inefficient compared to direct copying between the two machines.

- Local to remote. It will end up in the home directory (the directory you end up in just after you login) on the remote system. Try this between your local machine and vertex. Note that the colon (`:`) is important as it signifies a remote file.

```
Client1$ scp myfile.txt username@vertex.otago.ac.nz:
…
```

- Remote to local. Here we show how you can specify a different usercode on the remote system. Remember, `.` is the current directory.

```
Client1$ scp username@vertex.otago.ac.nz:myfile.txt .
…
```

- You can copy a directory using the `-r` (recursive) option.
  Use a small directory for trying this command.

```
Client1$ scp -r source-dir username@vertex.otago.ac.nz:target-dir
…
```

FTP is not a protocol that can have cryptography easily wrapped around it. We can use **sftp** to provide the feel of a fairly basic ftp client, though it is, in no way, the FTP protocol. Actualy, it uses the scp mechanism.

```
workstation$ sftp vertex.otago.ac.nz
Connecting to vertex.otago.ac.nz...
sftp> Type `?' if you don't know how to use ftp.
```

4. Server (sshd) and Client Configuration in Linux

In this section we shall move into our virtual environment, so start up Server1 and Client1.

Install and configure sshd

Install the `openssh-server` package on Server1 only.

The configuration files for the SSH server are stored in `/etc/ssh/`. The main configuration file is `sshd_config`, which controls the daemon. There is also `ssh_config`, which controls the default behaviour of the SSH client program. Finally, there are a number of keys, which are the public and private keys for the server. The only thing we need to edit is `sshd_config`.

You can get information on what the various keywords mean by consulting the manual page for sshd_config(5) and/or sshd(8). You'll need to harden the configuration and tailor the settings of the daemon to suit our requirements. Note that not all the options may have an effect, as not all possible features are compiled into the binary at compile-time.

Make the following settings in `sshd_config` on your SSH server. You should consult the appropriate manpage.

- Turn off anything to do with rhosts, or host based authentication. (Should be off by default anyway).
- Do you want to allow root to login? On a workstation this can be useful for management tools. For example, on a management machine you could imagine a manager wanting to run an administrative command on each machine, perhaps in a loop. For example:

```
example$ for wsn in $(cat workstations-lab-A.txt)
example> do
example>   ssh -o BatchMode=yes root@${wsn} apt-get install …
example> done
```

At other times this is inadvisable. You should never enable this option on servers. On Ubuntu it is, by default, pointless. This is because root's account is locked by default.

- Do you want X-Forwarding available? You'll find it will default to different values on different systems, so it pays to check and make any configuration explicit. This is useful if you want to use GUI programs on the server, but have them display locally. Most servers ought not to have GUI environments, but many do. It's more useful on workstations.
- In some countries, in order to get protection from the law, you have to show a banner when someone tries to login. Make a simple banner `/etc/issue.net` that says something containing the hostname (hint: look at the **banner** program available in the 'sysvbanner' package), and the string AUTHORISED USERS ONLY, and tell **sshd** to display it.

Tip

Choose your words carefully. Avoid using words like "welcome", which could allow an attacker to defend their actions.

Restart the server (**/etc/init.d/ssh restart**) and check that it's running.

Currently, you will find that you won't be able to access the SSH service, because most versions of OpenSSH are configured to use TCP Wrappers, which we have previously configured to default-deny.

You should normally decide on an appropriate access policy, whether people can access any machine from the internet (typically this is bad, due to password dictionary attacks), or whether they must first log into another machine.

You could use one of the following lines in `hosts.allow` to allow the appropriate level of access to SSH:

```
sshd: ALL
OR
sshd: 127.0.0.1 [::1] 192.168.1.0/24 [fd6b:4104:35ce::]/64
```

Breaking an entry into multiple lines

If the above square brackets around the ::1 address or any IPv6 addresses are forgotten, it will cause the whole line to be ignored by **tcpd**. This will prevent any access to SSH. If instead we had put in at least two lines (one for IPv4, one for IPv6), it would have only been a partial break if we missed the square brackets for IPv6, instead of a complete break. At least IPv4 would work for SSH in that situation. Therefore, it is better to have multiple lines in `hosts.allow`, one for each IP address or each class of IP addresses.

Double-check that `hosts.deny` has `ALL:ALL`, or at least `sshd:ALL` to block unwanted SSH or other connections.

Check that you can login to Server1 from Client1 as the user "mal". Before doing so, make sure the server and the client are connected via an internal network in VirtualBox.

```
Client1$ ssh mal@server1.localdomain
```

Set up ssh-agent for password-less login

Once you can login to Server1 from Client1, logout Server1 and follow the steps below to configure **ssh** in Client1 so that you can login to Server1 without password.

1. Generate a RSA key in Client1 with **ssh-keygen** as done previously.

2. Copy the public key in `~/.ssh/id_rsa.pub` in Client1 to the file `~/.ssh/authorized_keys` in Server1 under the home of `mal`, similar to what you have done before. This will tell **sshd** in Server1 to use the key instead of the password for identity check of `mal` at login time. Note you need to create the directory `.ssh` if it does not exist.

3. Start **ssh-agent**:

```
mal@client1:~$ eval $(ssh-agent -s)
```

   The **eval** will execute **ssh-agent** and export the environment variables used by **ssh-agent** to the current login session.

4. Configure **ssh** for **ssh-agent** to use the generated RSA key. Create a file `~/.ssh/config` with the following content in Client1.

```
Host *
  AddKeysToAgent yes
  IdentityFile ~/.ssh/id_rsa
```

   You can add more keys as you wish with more lines of `IdentityFile`. Alternatively, you can use **ssh-add** to add keys manually but you will have to use the command again after logout as the keys are not permanently added by **ssh-add**. To permanently add a key for **ssh-agent**, edit the above `~/.ssh/config` file.

5. Now try login to Server1 from Client1:

```
mal@client1:~$ ssh mal@server1.localdomain
```

   You need to type the passphrase for the first time. Then for future logins, you need to type neither passphrase nor password. Try logout from Server1 and login again. Do this for a couple of times.

Though we have used **ssh-agent** successfully, the previous work was only temporary. If the system reboots, you will have to repeat most of the above procedure. To avoid that, you need to edit some files as below.

1. First, kill the **ssh-agent** process:

```
$ ssh-agent -k
```

2. Add in `~/.profile` the following command for starting **ssh-agent** automatically at login time.

```
eval $(ssh-agent -s)
```

   You can just add it at the end of the file.

3. Add in `~/.bashrc` the following line to kill the **ssh-agent** process after logout.

```
trap '/usr/bin/ssh-agent -k' EXIT
```

Again you can just add it at the end of the file.

Now reboot Client1 and see if everything works as expected. You can try every **ssh** command from the previous sections. If you want to try X11 related applications like **xclock**, you should install them on Server1 as below.

```
$ sudo apt install x11-apps
```

Then you should see the following command works.

```
$ ssh -X mal@server1.localdomain xclock
```

5. Local Port Forwarding

Port forwarding is an advanced topic, and not something I require you to know how to do, so don't worry about it for this lab. It is such a powerful tool though, it would be disappointing if the more adventurous of you didn't try it.

Here is an example of how to use Local Port Forwarding, which is the most common type. Note that just because you *can* do something, it doesn't mean you should. Seek permission from any administrators before potentially violating policy.

In the lecture on SSH, you were given an example of how you could use local port forwarding to tunnel a clear-text protocol inside an encrypted SSH session. Earlier in the course, we implemented a simple, clear-text file-transfer service called "tinyfs". Let's enarmour tinyfs so we can use it over the network. We can use a tool such as **tcpdump** or **wireshark** to verify that nothing about tinyfs is being sent in the clear over the network.

```
Start the tunnel
Client1$ ssh -fNL 9000:localhost:900 server1.localdomain
Banner removed.

See what's listening
Client1$ lsof -Pni    Don't need root privs here.
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
An established connection to the SSH service on Server1
ssh     2481  mal    3u  IPv6  16213      0t0  TCP ↩
    [fd6b:4104:35ce:0:a00:27ff:fe99:c27d]:36218->[fd6b:4104:35ce::1]:22 (ESTABLISHED)
And IPv4 and IPv6 listening ports on loopback TCP/9000
ssh     2481  mal    4u  IPv6  16285      0t0  TCP [::1]:9000 (LISTEN)
ssh     2481  mal    5u  IPv4  16286      0t0  TCP 127.0.0.1:9000 (LISTEN)

Start up wireshark, starting a capture immediately
Don't forget to dismiss the warning window about running as root.
Client1$ sudo -b wireshark -i eth0 -k

Connect to our forwarded port
Client1$ echo '/etc/hostname\r\n' | nc -q-1 ::1 9000
OK     Success, output from the server
server1

When you're done, tear down the tunnel
Client1$ ps -eo pid,command | grep ssh
  825 /usr/sbin/sshd
 1256 /usr/bin/ssh-agent /usr/bin/dbus-launch --exit-with-session gnome-session
```

```
 2481 ssh -fNL 9000:localhost:900 server1.localdomain    Note
 2601 grep --color=auto ssh
Client1$ kill 2481
```

Have a look in Wireshark. Did you see any TCP/900 traffic? What if you right-click on one of the SSH packets and select Follow TCP Stream? You should see that nothing is sent in the clear over the Ethernet, and the SSH traffic is unintelligable because it is encrypted.

Repeat the experiment, but this time start the capture on the “lo” loopback interface. What do you find now? Should you be worried?

You may be wondering what the significance of the port numbers 9000 and 900 is. There is nothing really significant here, except that port 900 is the TCP port that our tinyfs service uses, and that port 9000 is above 1024 and so doesn’t need root privileges to accept connections. Otherwise, there is no limitation, so long as the port is not already used. The two port numbers could even be the same, but usually we end up using a high-numbered port for the local side.

6. User Imposed Restrictions on Public Keys

We’ve talked a lot about using keys to gain access. But how can you restrict what a key may be used for? Edit, creating the directory (mode 0700) and/or file if they don’t exist, `~/.ssh/authorized_keys` on the machine(s) you wish the key to be used to log into. We can restrict what a key can do by inserting some rules at the start of each line (note the lines are very long and will wrap).

The various restrictions you can put on a key are outlined in the man page for sshd, under the section titled “AUTHORIZED_KEYS FILE FORMAT”. There are some examples further down the page. There aren’t many, so you can read them all.

1. Add a restriction such that you can no-longer request X11 forwarding. Test it. You may remove the restriction after you have completed the test.

7. [Optional] Preventing Dictionary Attacks

> **Note**
>
> This section is purely for your information; you are not expected to implement these protection measures, but it will be useful to you later on if you ever operate internet-facing SSH servers. These measures are also applicable to other services, not just SSH.

As you will rapidly come to realise when you have an internet facing (meaning able to accept connections from the internet) SSH server, there are many attempts of people trying to break into your system using known/harvested username/password pairs, or just trying to find weak passwords by brute force. Most of the time it just creates a lot of noise in the system, but sometimes they succeed, and then they have a vector into the rest of the network, and possibly an easier method to get root access to the system (if that was their goal, they may just want to enlist your server in their illicit activities.

Split access policy

For this reason, it can be quite useful to disable password authentication, and prefer to use authentication systems such as public-key authentication or other authentication systems such as one-time pads, where the password is different each time or calculated via some other parameter. However, to use public-keys, you need to able to put the key into the account. It also becomes more important to carry your encrypted private key (and usually also your public key) around with you when you travel, typically on a USB flash drive. In order to install your public key into your account, you either need to a) access the system locally and copy the key over, b) access another system that has the same home directory mounted, c) ask a system admininistrator to install it for you, or more likely d) authenticate via a password and install the key so you can subsequently authenticate using your public-key.

(d) above would be nice, but it does require that we are able to configure **sshd** to have a policy that allows password authentication to known client systems, and disables it for systems on the internet. Fortunately, recent versions of OpenSSH give us a way to do just that, using the `Match` keyword.

Let's say for now that we only want to allow password authentication to IPv4 clients in 192.168.1.0/24, and public-key authentication can be used from anywhere. Phrased another way, be default we want password authentication to be disabled, and allow it explicitly.

Edit `sshd_config`, find, uncomment and disable the PasswordAuthentication option. Then, *at the end of the file*, add the following:

```
PasswordAuthentication no         Deny by default
…
at end of file
Match Address 192.168.1.0/24      Effective until next Match block or End-of-File
     PasswordAuthentication yes
```

To test, we shall need to first ensure that the client won't authenticate using public-keys (temporarily). We can do this by renaming `~/.ssh/authorized_keys` on Server1 to something else, such as `~/.ssh/authorized_keys.OFF`. That way, it will fall-back to trying a password.

Thus, according to our policy, only login via IPv4 should work. This is because public key authentication is the only allowable method for IPv6, and we've essentially broken that temporarily for our testing.

The easiest way in our particular case would be to force the use of IPv4 or IPv6 using the `-4` or `-6` option to **ssh**. Alternatively, we could specify the host as either `server1.ipv4.localdomain` or `server1.ipv6.localdomain`. Remember that typically, `server1.localdomain` will use the IPv6 address first.

When using such a policy, it becomes important to educate your users on the new policy, and provide resources for them to understand how to use the new authentication mechanism, and why this policy is in place to begin with.

Dynamic Banning

Another way to protect yourself is to scan your logs periodically (perhaps every 10 minutes), and dynamically ban — using `hosts.deny` or a firewall rule — any IPs where repeated authentication failures are happening, especially if the username that is being authenticated does not exist in the system. There are a number of products out there to assist with this.

DenyHosts is one that scans the logs and edits `hosts.deny`, depending on particular rules that are set by the administrator. However, because an attacker might get lucky and successfully authenticate before being banned, DenyHosts supports a *synchronised* mode, whereby many participating servers tell a central repository which machines they have banned, so other systems can ban them also.

8. Self-assessment

1. Create an SSH keypair on Client1, and demonstrate that you can use it to login to Server1 without a password and use the **ssh** command in various ways.

2. Your server and client configurations must be suitably well-configured. You should understand the relevance of some of it to the behavior of SSH.

3. Give three useful policies or practices that users should be encouraged or even required to do with regard to the secure use of SSH public keys, particularly with regard to Mac OS X 10.5 “Leopard”.