ssh Quick Reference

This is just some common tricks to use for ssh

Passwordless Logins

Sometimes, you need to be able to have an automated process log in to a *Target* machine from a *Source* machine. In most cases, this is done as the root user. This decreases security somewhat since if your *Source* machine is compromised, they can gain access to the *Target* with no problem (assuming they compromise the root account). However, automated processes require this.

The best way to do it is to use the *forced-command* parameter in *Target*'s sshd config file (/etc/ssh/sshd_config) for the root account, then create entries in the /root/.ssh/authorized_keys file. However, there are some times when you strictly need full access for a short period of time. I'll cover both.

First, on *Target*, create a public key pair. First, look if you already have a public key pair as we might not want to overwrite it.

```
ls /root/.ssh
```

If that does not have id_rsa and id_rsa.pub, or if you want to trash those, then we're good. If they exist, you can either use 'dsa' (an alternative) or temporarily move them out of the way (or delete, if you don't need them). Issue the following command, as root, and press <enter> at every question (including the passphrase). Change *rsa* to *dsa* if you want to use dsa.

```
ssh-keygen -t rsa -b 4096
```

This creates a file, /root/.ssh/id_rsa.pub, which is the public key for /root/.ssh/id_rsa (which has an empty passphrase). Somehow, get the contents of id_rsa.pub onto *Target*. It is a simple text file, so copy/paste is fine.

On *Target*, do the following:

```
mkdir -p /root/.ssh
joe /root/.ssh/authorized_keys
# place contents of id_rsa.pub on a separate line, then exit the editor
chown -fR root:root /root/.ssh
chmod 700 /root/.ssh
chmod 600 /root/.ssh/authorized_keys
```

From *Source*'s root account, you should now be able to ssh to Target and not have a password requested.

Limit to a single program

A more secure way to do this is to limit the command available. In this case, edit *Target*'s /root/.ssh/authorized_key and add '*command="/some/command/here",* ' to the beginning of a line which authorizes access. The space is very important after the comma. An example is:

```
command="/opt/bin/limitaccess", ssh-rsa AAAA...341tQ== root@dd-app-021
```

Upon login, /opt/bin/limitaccess will be called with the full parameters. If that script returns success, the command is allowed. If it does not, login is rejected. A sample script follows:

access_control.pl

```
#! /usr/bin/evn perl
use warnings;
use strict;

# get IP address
my $realIP = $ENV{'SSH_CLIENT'};
$realIP =~ m/^([\d.]+)/;
$realIP = $1;

# and hostname
my ($hostname,$temp) = split( ' ', $ENV{'SSH_ORIGINAL_COMMAND'});
# if $temp exists, it is the IP
$realIP = $temp if $temp;

die "You must send hostname with command\n" unless $hostname;

# do whatever you want here.
# look through $ENV to see what you can access

my $OK = &checkPermission();

die unless $OK;

1; # we made it here, so they can issue the command
```

Port Forwarding

ssh has the ability to forward an IP:port *as seen by the local machine* to an IP:port *as seen by the remote machine*. The most common occurrence of this would be to be able to access an internal web site when you ssh into a remote machine that is on that internal network.

The syntax uses the **-L** parameter to ssh, in the form

```
ssh -L localip:localport:remoteip:remoteport something
```

Remember, the local port is as seen by the local machine, and the remote ip/port is as seen by the remote machine. Let's use an example where we want to hit an internal web site where we have remote ssh capabilities. We can log into the remote server as

```
ssh username@joe.example.org
```

The internal IP of joe.example.org is 192.168.1.5, and we want to hit an HTTPS (port 443) web site at 192.168.1.6 (same subnet). We can not use ports under 1024 unless we are root, so we'll use port 8080 on our local machine (localhost) to get to that. The following command shows the example.

```
ssh -L localhost:8080:192.168.1.6:443 username@joe.example.org
# or, you can leave off the first IP and localhost is assumed
ssh -L 8080:192.168.1.6:443 username@joe.example.org
```

When this connection is made, any traffic going to localhost:8080 will be forwarded *over the ssh connection* to 192.168.1.6 on port 443. So, we can open our web browser on our local machine and put in the URL:

```
https://localhost:8080
```

and see the normally inaccessible web site on the remote network.

Note: You should not try to use a port that is already being used on your machine. So, for example, if you have a web server running on your local machine at port 8080, ssh can get very confused. In that case, you would want to use another port. You can use any unused port between 1025 and 65535 (don't know about the first and last ones there).

Relay Port Forwarding

I don't know the actual term for this, but we can forward a port to some machine, then forward that port to still another one. In this case, we have jane.example1.org, which we can get to. We also have john.example2.org which we can not get to unless we are logged into jane. We need to get to a Windows RDP server which john.example2.org can get to (port 3389).

```
ssh -L localhost:3389:localhost:3389 username@jane.example1.org
# we make the connection to jane and get a command prompt
ssh -L localhost:3389:192.168.1.10:3389 username@john.example2.org
# we are now on john, and 3389 from jane is forwarded to windows
# server at 192.168.1.10
```

In this case, we have said *any traffic for port 3389 on my local machine is forwarded to localhost port 3389 on jane* in the first command.

The second ssh command says *any traffic for port 3389 on my local machine (jane) is forwarded to port 3389 on the machine on my same subnet at 192.168.1.10 on port 3389*

You can now open an rdp client on your local machine to connect to localhost:3389. Any traffic for that will be forwarded to jane, which will then forward to john, who will then forward to 192.168.1.10.

Shared ssh configuration

The simplest way to share ssh configurations with a group is to have a shared group of ssh configuration files. You might go so far as to have one file per client, with each file containing the config information to make an ssh connection to various machines within a client's LAN.

In this example, we will have multiple files with a .ssh suffix. A sample file would be

```
# acme.ssh
# ssh configuration file for client Acme
Host     acme.server1
Hostname 192.168.100.3
User     tech

Host     acme.router
Hostname 192.168.100.1
User     administrator
Port     2222
```

A second file might be

```
# hal.ssh
# ssh configuration file for client HAL
Host     hal.router
Hostname 192.168.55.1
User     root
Port     22

Host     hal.fileserver
Hostname fileserver.hal.local
Port     22
```

Save these two files in a location shared with the relevant technicians who require access. This can be done via an NFS share, a local directory on a terminal server, or even sync'ing software like NextCloud.

On each system, modify the system wide ssh configuration to include all files matching *.ssh. NOTE: This could also be achieved by adding the Include into individual users .ssh/config files.

```
echo 'Include /path/to/shared/ssh/configs/*.ssh' >> /etc/ssh/ssh_config
```

Discussion

- Permissions should be fairly strict, though it is not as strictly enforced as some aspects of the ssh system.
  - All directories and files should be owned by root
  - files should be 644 (rw_rw_rw_)
  - directories 755 (rwxr_xr_x).
  - This allows all users to read the files but only root can modify them.

- The .ssh suffix is only one convention; not a requirement or any kind of magic. By using “*.ssh”, you are able to place other files as needed, and disable files by changing the suffix
- Users can still have their own .ssh/config files for private configurations
- Adding a client prefix with a period in between is simply convenient. In the example, you see that both hal and acme have a connection named router. **Host** is what you type, **Hostname**, **Port** and **User** is what you get. Host is just a convenient alias for the connection.
- Host and Hostname are the only required fields. If no user is specified in the configuration, the current user is assumed. If no port is used, the default (port 22) is used. There are several other options you can include if you need.
- The example of having one file per client/department/whatever is just organizational. You could easily have all of your entries in one file, or break it down by function. It all depends on your needs.

Shared authorized_keys file

Similar and complementary to the shared configuration above is the ability to have the same set of authorized_keys files available across multiple machines. This is especially important in clusters of servers, where all members of the cluster need to have the same basic set of public keys which are allowed to access them without a password.

Again, share a single file across all machines with the public keys that are allowed to access all machines in the cluster. It does **not** hurt to have the public key of the current machine in this file, so no modification is necessary to ensure a machine does not have its own public key.

An NFS mount is likely the best way for to set this up. I'm going to assume it is mounted at /srv/common_config, with a subdirectory of ssh.

Add/replace the AuthorizedKeysFile directive in /etc/sshd_config. On some systems, you can simply create a file /etc/ssh/sshd_config.d

```
# Debian style
echo 'AuthorizedKeysFile .ssh/authorized_keys
/srv/common_config/ssh/authorized_keys' >> \
   /etc/ssh/sshd_config.d/authorized_keys.conf
```

Reload the sshd daemon.

Discussion

- Permissions very important here.
  - All files and directories must be owned by root
  - Directories must be 755, and if possible, 700
  - Files must be 644, and if possible, 600
- Users may still have a personal .ssh/authorized_keys file.
- Some systems are set for .ssh/authorized_keys2 as a second file. You can add this as a third option in the /etc/ssh/sshd_config.d/authorized_keys.conf file
- This does **not** include the known_hosts file, so the initial connection must still be made. However, the documentation in sshd_config implies this can be accomplished somehow.

Links

- https://www.ssh.com/academy/ssh/authorized-keys-openssh