No More Passwords (with SSH)

Ted Dustman

March 30, 2009

Contents

1 Introduction

The purpose of this article is to introduce the use of the Secure Shell (SSH) and its *Public-Key Authentication* features which allow, among other things, password-less logins.

This discussion in this article applies to the SSH implementation known as OpenSSH but should apply to other ssh implementations as well.

This article assumes you are using an sh'ish type shell (bash, ksh, etc.).

In addition to this article you may want to read the manual pages for `ssh`, `scp`, `ssh-keygen`, and `ssh-agent`. Also see the book `SSH The Secure Shell, The Definitive Guide`.

This document is online at `www.cvrti.utah.edu/~dustman/no-more-pw-ssh`.

1.1 Local or Remote?

In the discussions below the machine you are presently logged into is called the *local* (or *client*) machine . This may be your desktop or portable computer at work or home or a "main frame" that you have logged into from home or work.

A *remote* machine is one you login to from your local machine (called a remote login) or it's one on which you execute a command without actually performing a login (remote command execution).

"Local" and "remote" are relative terms though. A "remote" machine becomes a "local" machine once you have logged into it.

1.2 SSH Command Set

SSH consists of the following commands:

`ssh` Permits secure logins and remote command execution.

`scp` Performs secure remote file transfers.

`ssh-keygen` Generates public and private authentication keys for use by the other SSH programs.

`ssh-agent` Manages public and private keys on behalf of its user. This command is the key to our password-less world.

`ssh-add` Adds a key to the list of keys that are managed by an agent.

`sftp` A secure replacement for FTP which uses the SSH protocol. Not discussed further in this article.

Below I will show you how to use these programs to perform secure logins, remote command execution, and remote file transfers without using passwords.

2 Authentication Keys

SSH *authentication keys* are data that allow an ssh server running on a remote computer to verify that you are who you claim to be. The following sections show you how to generate keys on your local machine and how to install them on remote machines.

2.1 Generating Keys

We will generate three key pairs (each pair consists of a public key and a private key). One to support the old SSH1 protocol still in use by some facilities. And two others that can be used with the SSH2 protocol.

The first key is generated as follows:

```
ssh-keygen -t rsa1
```

`Ssh-keygen` will respond as follows:

```
Generating public/private rsa1 key pair.
Enter file in which to save the key (/Users/dustman/.ssh/identity):
```

Just press `return`. Then `ssh-keygen` will ask for a *passphrase* (twice). A passphrase is like a password except it can be any length and can contain whitespace, punctuation, numbers, almost anything. "Aunt Nellie's toe nails smell like rotten eggs to me" is a valid passphrase.

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Don't create an empty passphrase!

`Ssh-keygen` will then display some information and quit.

Now issue the following `ssh-keygen` commands to create the other keys:

```
ssh-keygen -t dsa
```

then

```
ssh-keygen -t rsa
```

Note that you are entering 3 separate passphrases. They may all have the same value or each may be different from the others.

2.2 Copying Keys

You should now have 3 pairs of keys in the directory `~/.ssh`. The files with the `.pub` extension are the public key files. The others are the private key files.

```
identity  identity.pub
id_dsa    id_dsa.pub
id_rsa    id_rsa.pub
```

You must copy the content of the public key files to every remote machine of interest.

You may need to create the directory `~/.ssh` on the remote machines if it doesn't exist:

```
cd ~
mkdir .ssh
chmod 700 .ssh
```

Now you've got to get the content of the public key files to the remote computers. The easiest way is to open an editor on the local machine and then to connect to a remote machine (via ssh of course) and start an editor there. Then you can cut and paste between the editors (using whatever cut and paste mechanism your operating system provides).

The content of `~/.ssh/identity.pub` on the local machine goes in the file `~/.ssh/authorized_keys` on the remote machines.

The content of the `~/.ssh/id_dsa.pub` on the local machine goes in `~/.ssh/authorized_keys` on the remote machines.

Likewise, the content of the `~/.ssh/id_rsa.pub` on the local machine is appended to `~/.ssh/authorized_keys` on the remote machines.

The file `authorized_keys` may contain public key content from many clients. Simply put each public key entry on a separate line.

3 The Agent

The *Agent* is the program, `ssh-agent`, that runs on the local machine and acts as your proxy when an ssh command requires a passphrase.

Normally you would type a password or passphrase when requested by `ssh` or `scp`. However, the agent can provide the passphrase for you. The trick is telling the SSH commands to get the passphrase from the agent rather than you. First you must "prime" the agent and then you must "attach" the agent to one or more processes.

3.1 Priming the Agent

To "prime" the agent issue the following commands (on the local machine):

```
ssh_info_file=~/.ssh-agent-info-‘hostname‘
ssh-agent >$ssh_info_file
chmod 600 $ssh_info_file
. $ssh_info_file
ssh-add ~/.ssh/identity
ssh-add ~/.ssh/id_dsa
ssh-add ~/.ssh/id_rsa
```

Each `ssh-add` command will prompt you for the appropriate passphrase.

Note the output of `hostname` is appended to the name of the ssh agent info file. This distinguishes the name of the file from other instances of the file that may be created in a multi-host, shared home directory environment.

It's convenient to capture this sequence in a shell script:

```
#!/bin/bash
# Creates an ssh-agent, writes ssh agent info
# to the file '~/.ssh-agent-info-‘hostname‘' and then prompts
# user for keys.  Then any shell can use the agent
# by sourcing the contents of ~/.ssh-agent-info-‘hostname‘:
#  . ~/ssh-agent-info-‘hostname‘

ssh_info_file=~/.ssh-agent-info-‘hostname‘
ssh-agent >$ssh_info_file
chmod 600 $ssh_info_file
. $ssh_info_file
for i in identity id_dsa id_rsa
do
    ssh-add .ssh/\$i
done
```

Save this script as `ssh_prime` in your home directory. Now you can type:

```
source ssh_prime
```

You need only "prime" your agent *once* each time you *reboot* your machine. The agent will stay active across logins.

Now you should be able to connect, without entering a password, to all remote machines that have the public keys you generated earlier.

Note that the shell command `source ssh_prime` both primes the agent and attaches it to the current shell process (and its children). However, in some cases (example given below), you may need to explicitly attach the agent to a process after the agent has been primed.

3.2 Attaching the Agent

Attaching the agent to a process is easy. For example, start a new terminal session and type:

```
. ~/.ssh-agent-info-‘hostname‘
```

This attaches the agent to the terminal process (as well as any future processes that are children of the terminal process).

Now you should be able to connect, without entering a password, to all remote machines that have the public keys you generated earlier.

See the examples below for more ssh tricks.

4 Examples

The following examples show some uses of the ssh passphrase agent. All examples assume you have generated keys and have moved the public keys to remote machines of interest.

4.1 Login Initialization

It is easy to forget to prime an agent. The login script presented below will “remind” you to prime your agent. When inserted into your login initialization script (`.profile` or `.bash_login`) the script will check for a running agent. If the script does not find a running agent then it will create one (by running `ssh_prime`). The script then attaches your shell to the agent.

Insert the following script into a file named `ssh_agent.sh`.

```
ssh_start_agent() {
    ssh-agent >$SSH_AGENT_INFO
    chmod 600 $SSH_AGENT_INFO
}
```

```
ssh_add() {
    source $SSH_AGENT_INFO
    for i in id_dsa id_rsa
    do
      ssh-add ~/.ssh/$i
    done
}

ssh_ask_start_agent() {
    echo -n "Start ssh agent (y,n)? "
    read yorn
    if test "$yorn" == "y"
then
ssh_start_agent
ssh_add
source $SSH_AGENT_INFO >/dev/null
    fi
}

ssh_setup_agent() {
  hostname=‘hostname‘
  export SSH_AGENT_INFO=~/.ssh-agent-${hostname}
  if test -f $SSH_AGENT_INFO
  then
      source $SSH_AGENT_INFO >/dev/null
      if ! (ps -p "$SSH_AGENT_PID" | egrep ’^ *’"$SSH_AGENT_PID"’ .*ssh-agent$’) >/dev/null
      then
  echo Found inactive $SSH_AGENT_INFO
  ssh_ask_start_agent
      else
          echo Using active $SSH_AGENT_INFO.
  source $SSH_AGENT_INFO >/dev/null
      fi
  else
      echo Found no $SSH_AGENT_INFO
      ssh_ask_start_agent
  fi
}

ssh_setup_agent
```

Put the following into your `.profile` or `.bash_login` file:

```
test -f ssh_agent.sh && source ssh_agent.sh
```

4.2 Mac X11

Mac X11 is an X11 server that run, among other things, terminal applications that can be used to connect to remote machines.

In this example we will create an agent and attach it to an X11 server so you never need to type another password when executing SSH commands from within the X11 environment.

1. Startup Apple's `Terminal` application.
2. Type `~/ssh\_prime` (but only if you've not done this during a previous login).
3. Type `. ~/.ssh-agent-info-‘hostname‘` (but only if you skipped the previous step)
4. Start the X11 server by typing:

   ```
   open -a /Applications/X11.app
   ```

5. Quit the Terminal application.

Now from within X11, start a terminal application (*e.g.,* eterm or xterm) and "ssh" to a remote machine:

```
ssh a.remote.machine
```

Perhaps you have an item in the Mac X11 Applications menu that connects to a remote machine. Try it out.

Now try copying a local file to a remote machine using `scp`:

```
scp afile a.remote.machine:
```

Now try copying a remote file to your local machine:

```
scp a.remote.machine:afile .
```

Now try issuing a command on the remote machine:

```
ssh a.remote.machine 'uptime'
```

Try the same command from within emacs or vi (using their shell command execution features).

In all cases above you should not need to enter any passwords.

4.3 Linux and XFree86

[Note: The information in this section may be out of date or otherwise incorrect—I haven't had a linux box in a while.]

If your local machine is a Linux box you will want to prime the agent after logging in but before starting XFree86 (the X11 server on most Linux boxes). This example is the Linux equivalent of the previous example.

To do this you may need to change the operating system's *run level*. You will want to change the run level (which is normally 5) so the system boots into multiuser mode with networking support but does not start XFree86. This is run level 3. You'll need super user privileges to do this.

Edit the file `/etc/inittab`. It will look something like this:

```
id:5:initdefault:
si::sysinit:/etc/rc.d/rc.sysinit
l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6
```

Change the first line to this:

```
id:3:initdefault:
```

Save the file and reboot your machine.

Your machine will not start up with its usual pretty login screen. Instead your fancy monitor will be turned into an old fashioned dumb terminal (but only temporarily).

Type in your user name and password when prompted. Now prime your passphrase agent the usual way:

```
~/ssh_prime
```

Note that you only need to do this if haven't done it during a previous login.

Now attach to the agent (but only if you skipped the previous step):

```
. ~/.ssh-agent-info-‘hostname‘
```

Now start up XFree86:

```
startx
```

You now get back your pretty GUI *and* you no longer need to type in any SSH passphrases. Try executing some SSH commands.

4.4 Shared Home Directory

In a work or school environment you may have an account which gives you access to more than one computer. Usually your account is setup so that your home directory is shared by all machines. It is easy to setup password-less SSH amongst all computers. There are two cases to consider though.

Case 1 You want password-less access to all machines on a remote network from your local machine.

In this case you only need to copy your local public keys to your account on the remote computer network. Then you will have password-less SSH access from your local machine to all machines on the network (after priming and attaching the passphrase agent on the local machine of course).

Note that this does not give you password-less access *between* the machines on the remote network. The next case handles this situation.

Case 2 *Note: This section is confusing (and only marginally useful). Beware.*

You want password-less access between the machines on the remote network (or local network if you login to a console or terminal attached to a machine on the network).

Log in to any host on the network. Generate keys as shown previously. Copy the content of the public key files to the `authorized_keys` and `authorized_keys2` files in your `~/.ssh` directory. Prime and attach the agent. Be sure to append the name of the host to the name of the ssh agent info file.

Log into another host on the network, either from the original local host or from a host on the network. Prime and attach an agent on the new host, remembering to append the name of the host to the name of the ssh agent info file. Repeat the priming and attaching steps each time you log into a host—you will create password-less logins between hosts on the network.

4.5 Cron Jobs

Say you are generating a bunch of data on your local machine once each hour. And you've also got to copy that data to a remote machine once each hour. So you think "Gee, that's a job for cron". So you write a script for cron to execute:

```
#!/bin/sh
makedata input output               # a program that generates data.
scp -q -B output a.remote.machine: # copy output to remote machine.
```

But then you realize that each time your cron script tries to copy data to the remote machine it (your script) will be asked for a passphrase that it cannot provide.

Ah, but you are smarter than the average Nerd. So you modify your script as follows:

```
#!/bin/sh
makedata input output              # a program that generates data.
. ~/.ssh-agent-info-‘hostname‘  >/dev/null   # attach agent to script's process.
scp output a.remote.machine:       # copy output to remote machine.
```

Now the agent will provide the password and your cron job will succeed.

4.6 Tramp

This section assumes that emacs (or xemacs) has been started as the child of a process that has been attached to an ssh agent, *e.g.,* X11 or OroborOSX (see section **??**).

Tramp is lisp package that gives emacs the ability to edit remote files as if they were local files.

Tramp may use a number of methods for accessing files on a remote machine. The only transfer method discussed here is the "scp external" method because it works well with the ssh techniques discussed in this article.

You must first install Tramp as per its instructions. Then add the following to your `.emacs` file:

```
(add-to-list 'load-path "path/to/tramp/lisp/")
(require 'tramp)
(setq tramp-default-method "scp")
```

where `path/to/tramp/lisp` is the path to the Tramp lisp files (you may not need this line— talk to your sysadm).

Now within emacs you may refer to remote files and directories much as you would local ones only using a slighty different syntax. For example, the path to your home directory on the machine named **remote** is `/remote:`. Note mandatory use of the leading "/" character and the trailing ":" character.

And `/remote:.emacs` refers to the `.emacs` file in your home directory on the machine **remote**. Finally the path `/eegah.cvrti.utah.edu:/cvswork/docs/no-more-pw-ssh.tex` refers to, well, you get the idea.