

# Performing backups over an ssh connection

Cameron Kerr

29 January 2002

## Abstract

In Unix History, backups that have been made over a network have been done using the Berkeley `r`-commands. Nowadays, these commands are insecure, and should not be used, not just because of the data security, but because using the `r`-commands opens up the server host far more than is comfortable.

Here, I outline an approach that can be used to send your backups over the network, using the popular SSH 2 protocol (OpenSSH in particular). I show how to set up keys for automated login, and restrict access, and how we can use the forced command feature of SSH 2 to manage our backups.

## 1 The Sending Side

The sending side refers to the computer that is making the backup. I shall assume that this will be the user ‘root’, since it is that account that will likely need to run the backup.

### 1.1 Making the backup

Do this however you want, but I suggest that you verify it and the compress it before sending it over the network. This does require extra disk space, however, which may not always be available. If you run tight on disk space, you could write the backup directly to the receiver, then read it back when you’ve finished, in order to verify. Do verify you’re backup, you’ll be very sorry if you need to call on a corrupt backup.

I shall assume that you have created the backup, ready for transportation, to a file referenced as *ARCHIVEFILE*.

### 1.2 Creating the Senders Keys

We don’t want to get any prompts for password or pass-phrases when connecting to the receiver. Fortunately, ssh can help us here. Normally, for an interactive session, you *really should* encrypt your private key with a pass-phrase. If you don’t its like writing you password into a file. If you use a pass-phrase, you should also use `ssh-agent` and `ssh-add`, so you only have to enter your pass-phrase once.

For automated tasks, even using a pass–phrase can be unsuitable, so we can use a private key that doesn’t have a pass–phrase attached to it. Because this opens a trust relationship, we will use a separate *identity* for when we use this key to connect to the receiver, and the receiver will restrict what this key can do. Sound ok? Good.

If you’re confused by any of this, I suggest you read *SSH The Secure Shell: The Definitive Guide*, by Barrett and Silverman, published by O’Reilly.

We create the keys using the following command. This will create a DSA public and private keys, of 1024 bits, as an identity called `sbackup_csatm5_id_dsa`, in the `/root/.ssh/` directory.

```
# ssh-keygen -b 1024 -t dsa -f /root/.ssh/sbackup_csatm5_id_dsa+
```

Enter no pass–phrase when prompted. You should copy your public key (the contents of the file ending in `.pub`) that matches the given identity. You will need it later (just copy it the clipboard).

## 2 The Receiving Side

The receiving side is the machine that will house the backups once its been made. Because we’re using pass–phrase–less keys to connect to this computer, we will create a new user and group on the receiving host, and use that to both protect the archives and to restrict access.

### 2.1 Creating the Account

I called my backup user and group as `sbackup`, so the archives will get an ownership of `sbackup.sbackup`. Make sure the `uid` and `gid` are unique. I made the home directory for this user `/scratch/csatmlbackup/`, you should change yours to suit. Put it somewhere with *lots* of space.

The `sbackup` account should not be able to login (so don’t give it a password, leave the account disabled). Leave their login shell as `/bin/bash`, or a restricted version of something.

Make sure `sbackup.sbackup` owned their home directory.

To do the following, it would be advisable to `su` to `sbackup` from the root account.

### 2.2 Creating the Receivers Keys

Create the key DSA key for the `sbackup` account. This isn’t actually needed, I don’t think, since their private/public keys are not used to let others log into their account. Perhaps they should not be present.

But anyway, here’s what I did.

```
$ ssh-keygen -t dsa -b 1024
```

What’s really important, is the `~/.ssh/authorized_keys2`. We put the public key you copied into this file in order for the owner of that public key to log in using just public/private key authentication.

Save the file, and from the sender account on the sender host, make sure you can log into the receiver using the identity we set up.

```
# ssh sbackup@receiver.example.com -i ~/.ssh/sbackup_csatm5_id_dsa
```

All going well, you should get a shell, and not be asked for a password or pass-phrase. If not, invoke ssh with the -v flag to get some clues.

## 2.3 Restricting access to the Senders Key on the Receiver

Now that the sender can log into the receiver, its time to restrict that the receiver can do, severely. There are three main things we would like to control. They are 1) What host(s) the sender may login from, 2) What capabilities the sender may be given by the receiver, such as port-forwarding, and 3) What commands the user may run.

To do all of this, we can edit the public key in the authorized\_keys2 file. Each key can have a set of options, delimited by a comma, and containing no white space that is not enclosed in double quotes. See the sshd manual page for what these options are. All options go at the start of the line, no line breaks are allowed.

To restrict the hosts that may use the account, we can add `from=csatm1eth.localeth`. Note that the argument should be the fully qualified, canonical host name. You can use the host program to find this information.

To restrict what services we offer to the key, we can include these options, look in the man page if you want to know what they are: `no-port-forwarding`, `no-X11-forwarding`, `no-agent-forwarding`, `no-pty`.

## 2.4 Using a Forced-Command for the Senders Key on the Receiver

Finally, the user of this key only needs to do one thing, and we can prescribe what that thing is by using a forced command, given by the `command=""` option. All we need it to do, is stick input into a file, which means we need the name of the file, and the contents of the file itself. If we're not worried about clobbering files, we can use this command option. Note that its all on one line.

```
command="cd; echo \"Enter filename, followed by a single newline\";
      read filename;
      echo \"Dumping to $filename\";
      echo \"Send archive stream.\";
      cat > $filename;
      echo \"Archive received, calculating md5sum\";
      md5sum $filename | tee -a CHECKSUMS.md5"
```

You should note that this could (and probably should) go into a script file. What this does is first asks for a filename, then everything after the first newline is considered data. When EOF has been detected, you can do other things, like calculate an md5sum, or print out a receipt, or anything, really.

### 3 Using the Method

Transferring the file is relatively simple in itself. You can easily do it as part of a script, as shown.

```
# ARCHIVENAME is the basename of the archive we have already created
( echo "$ARCHIVENAME"; cat $ARCHIVENAME ) | \
  ssh sbackup@receiver.example.com -i ~/.ssh/sbackup_csatm5_id_dsa
```

Assuming *ARCHIVENAME* contains the text `csatm1_home_daily_20020129.tar.gz`, you will see the following output

```
Enter filename, followed by a single newline
Dumping to csatm1_home_daily_20020129.tar.gz
Send archive stream.
Archive received, calculating md5sum
892a33dceb2b205257ab7d818ee51af1 csatm1_home_daily_20020129.tar.gz
```

If you have any question or suggestions, please let me know at [ckerr@cs.otago.ac.nz](mailto:ckerr@cs.otago.ac.nz)