

SSH User Identities

Brian Hatch 2004-11-03

OpenSSH supports more than just simple passwords for authentication. It can be configured to use PAM (Pluggable authentication modules), Challenge/Response protocols, Kerberos authentication, authenticated host-based trust[1], and there are even patches for other methods, such as X509 keys. However the most popular alternate authentication method is Identity/Pubkey authentication.

The goal of using Identity/Pubkey authentication is to remove the need for static passwords. Instead of providing a password, which could be captured by a keystroke logger or witnessed as you type it, you have a key pair on your disk that you use to authenticate. Your account on the SSH server has a list of Identities/Pubkeys that it trusts, and if you can prove you have the public and private key then you are granted access without supplying a password.

Some of the nice features of this form of authentication are:

- No one can shoulder-surf your password and log in to your accounts - they'd need both your Identity passphrase and the private key from your machine.
- The server administrator could disable password authentication entirely, to prevent password guessing attacks.
- You can use the `ssh-agent` and SSH agent forwarding to have your authentication credentials 'follow' you.
- You can place restrictions on Identities/Pubkeys, for example forbidding port forwards, forcing predetermined commands, regardless of what the user wanted to run, and more.

In this week's article we'll show how you create keys and configure your account to allow them to log in. In later articles we'll go into some of the other capabilities of SSH identities.

Creating an Identity/Pubkey

In the original SSHv1 protocol implementation, you could create an *Identity*, which was an RSA public/private key pair. The SSHv2 protocol changed the format of these keys, and supported both RSA and DSA keys, and renamed this functionality *Pubkey* authentication. I'll use these two terms interchangeably, since they have the same functional purpose.

The `ssh-keygen` program is used for creating our key.[2] So let's jump right in and create one:

```
mydesktop$ ssh-keygen -t rsa
```

```
Generating public/private rsa key pair.
```

```
Enter file in which to save the key (/home/xahria/.ssh/id_rsa):
```

```
Enter passphrase (empty for no passphrase): (enter passphrase)
```

```
Enter same passphrase again: (enter passphrase)
```

```
Your identification has been saved in /home/xahria/.ssh/id_rsa.
```

```
Your public key has been saved in /home/xahria/.ssh/id_rsa.pub.
```

```
The key fingerprint is:
```

```
2c:3f:a4:be:46:23:47:19:f7:dc:74:9b:69:24:4a:44 xahria@mydesktop
```

```
mydesktop$ cd $HOME/.ssh
```

```
mydesktop$ ls -l
```

```
-rw----- 1 xahria hatchclan 883 Jan 21 11:52 id_rsa
-rw-r--r-- 1 xahria hatchclan 223 Jan 21 11:52 id_rsa.pub
```

```
mydesktop$ cat id_rsa
```

```
-----BEGIN RSA PRIVATE KEY-----
```

```
MIICWgIBAAKBgQCc+1oixZ/g84gpZH0NeI+CvVoY500FOCSpFCbhUGJigQ6VeKI5
gpOlDztpJ1Rc+KmfZ2qMaftwwnLmefhklwPcvfZvvLjfdmHY5/LFgDujLuL2Pv+F
7tBjlyX9e9JfXZau2o8uhBkMbb3ZqYlbUuuoCANUtL5uZUiiHM0BAtnGAd6epAYE
gBHwlxnqsy+mzbuWdLEVF7crlUSsctwGapb6/SEQgEXFm0RITQ3jCY808NjRS3hW
Z+uCCO8GGUsn2bZpcGXa5vZzACvZL8epJoMgQ4D0T50rAkEA0AvK4PsmF02Rzi4E
mXgzdlyCa030LYR/AkApG1KT//9gju6QCXlWL6ckZg/QoyglW5myHmfPR8tbz+54
/lj06BtBA9iag5+x+caV7qKth1NPBbbUF8Sbs/WI5NYweNoG8dNY2e0JRzLamAUk
jK2TIwbHtE7GoP/Za3NTZJm2Ozviz8+PHPIEyyt9/kzT0+yo3Kmgss1qWIBIwKB
XdBh42izEWSWpXf9t4So0upV1DEcjq8CQQDEKGAzNdgzOoIozE3Z3thIjrmkimXM
J/Y3xQJBAMEqZ6syYX/+uRt+any1LADRebCq6UA076Sv1dmQ5HMfPbPuU9d3yOqV
j0Fn2H68bX8KkGBzGhhuLmbrgRqr3+SPM/frUj3UyYxns5rnGsprkGB3AkALCbzH
9EAV8Uxn+Jhe5cgAC/hTPPdiwTJD7MpkNCpPuKRwrohytmNAmtIpKipAf0LS61np
wtq59ssjBG/a4ZXNn32n78DO0i6zVV5vwf8rv2sf
```

```
-----END RSA PRIVATE KEY-----
```

```
mydesktop$ cat id_rsa.pub
```

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAcMJy5nn4ZNcd3L32b7y433Zh2IEAnPt
```

```
aIsWf4POIKWR9DXiPgrlaGOTtBTgkqRQm4VBiYoE0lXiiOYKtpQ87aSdUXPipn
2dqjGn7OfyxYA7oy7i9j7/hYytkyMGx7ROxqD/2WtzU2SZtjs74s/PjxzyBMsr
ff5M09PsqNypoLLLZas= xahria@mydesktop
```

```
# Note: the 'ssh-rsa...xahria@mydesktop' stuff is all on one line,
# I've wrapped it for legibility.
```

As you can see, `ssh-keygen` created two files, `id_rsa` and `id_rsa.pub`. The first file is the private key, protected with the passphrase you typed at key creation time. You do not want to let anyone get at this file. (See the 'Passphrase Security' section later in this article.) The second file is the public key - this doesn't contain anything secret or compromising, so there's no need to take precautions about keeping this file safe. In fact, it can be re-generated from the private key if necessary.

Types of SSH Keys

When we created the key, we included the option `-t rsa`. This told `ssh-keygen` what kind of key pair to create. Like SSH host keys, an SSH user key can be in one of three different formats, `rsa1`, `rsa`, or `dsa`, as described in the table below:

Type	Command line arg	Default filename	Protocol	Example public key snippet
RSA1	<code>-t rsa1</code>	<code>identity</code>	SSH 1 protocol only	1024 35 118118225938285...
RSA	<code>-t rsa</code>	<code>id_rsa</code>	SSH 2 protocol only	ssh-rsa AAAAB3NzaC1yc2E...
DSA	<code>-t dsa</code>	<code>id_dsa</code>	SSH 2 protocol only	ssh-dss AAAAB3nzaC1kc3M...

You can specify the filename to use when creating a key by using the `-f filename` option. If you do not specify a filename, then it will create the key in the `$HOME/.ssh/` directory, using the default filename listed above.

Allowing Identity/Pubkey Authentication on the Server

Now, having created a key, we want to cause it to be trusted by our account on the SSH server.

First, in order to allow Pubkey or Identity authentication, the SSH server must have the proper settings in its `sshd_config`:

```
# Should we allow Identity (SSH version 1) authentication?
RSAAuthentication yes

# Should we allow Pubkey (SSH version 2) authentication?
PubkeyAuthentication yes

# Where do we look for authorized public keys?
# If it doesn't start with a slash, then it is
# relative to the user's home directory
AuthorizedKeysFile      .ssh/authorized_keys
```

The settings above are the defaults, which enable Identity/Pubkey authentication for both SSH version 1 and 2, and check for public keys in user's `$HOME/.ssh/authorized_keys` file.

Make sure you have appropriate entries in `sshd_config` (typically located in the `/etc/ssh/` directory) and restart the server if necessary.

Setting up the `authorized_keys` file

The `authorized_keys` file is simply a text file with a list of keys, one per line. Lets install the key we created above on our remote machine:

```
# Copy the RSA Pubkey to the server using scp.
# You can use any method you like, including using
# copy/paste if it's convenient.
mydesktop$ cd $HOME/.ssh
mydesktop$ scp id_rsa.pub ssh-server:id_rsa_mydesktop.pub
xahria@ssh-server's password: (enter password)

# Now let's log in and create the authorized_keys file
mydesktop$ ssh ssh-server
xahria@ssh-server's password: (enter password)

# Create the .ssh dir if it doesn't already exist
```

```
ssh-server$ mkdir .ssh
ssh-server$ chmod 700 .ssh
ssh-server$ cd .ssh

# Concatenate the RSA Pubkey we just uploaded to
# the authorized_keys file. (This will create
# if it doesn't already exist.)
ssh-server$ cat ../id_rsa_mydesktop.pub >> authorized_keys

# Let's check out the file
ssh-server$ cat authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAcMJy5nn4ZNcD3L32b7y433Zh2IEAnPt
aIsWf4POIKWR9DXiPgr1aGOTtBTgkqRQm4VBiYoE0lXiiOYKTPq87aSdUXPipn
2dqjGn70fyxYA7oy7i9j7/hYytkyMGx7ROxqD/2WtzU2SZtjs74s/PjxzyBMsr
ff5M09PsqNypoLLLZas= xahria@mydesktop

# Make sure permissions are paranoid.
ssh-server$ chmod 600 authorized_keys

# Excellent, let's log out.
ssh-server$ exit
```

In the above session we've copied our public key to the server using `scp`, then used `cat` to append it to our (potentially existing) `authorized_keys` file. You want to make sure you don't simply copy a key over the `authorized_key` file, because you'll nuke any other keys you currently have trusted.

Once you've taken the above steps, you'll see a difference when you attempt to log in to the remote account:

```
mydesktop$ ssh ssh-server
Enter passphrase for key '/home/xahria/.ssh/id_rsa':
```

Your SSH client will try Pubkey/Identity authentication first, before falling back to password authentication. It will look for the three default key filenames if you don't specifically choose one, offering each to the server.

So, here's how an SSH connection looks when Pubkey/Identity authentication is thrown into the mix:

- `/usr/bin/ssh` connected to ssh-server's SSH port.
- The SSH client and server performed their handshaking. (Client verified server's host key, encryption keys and algorithms were negotiated, etc.)
- The SSH client looked for any default Pubkey/Identity files[3].
- If a default key was found, it sends the public key to the server and offers to authenticate with this key.
- The server checks the user's `authorized_keys` file, and determines if the public key is present. If it is, the server offers to accept this key from the client by presenting a mathematical challenge.
- If the private key is protected by a passphrase, `/usr/bin/ssh` asks the user to provide the passphrase to decrypt the private key.
- The client performs some magical mathematics[4] with the key to prove that it has both the public and private key.
- If the mathematical contortions are successful, the server logs the user in without asking for their actual Unix password.
- If the client cannot prove it has the key, it may offer other keys instead.
- If the client has no other keys it can offer, then the server offers to authenticate the user using standard Unix password authentication.

Luckily, all this happens behind the scenes. If you want to see it in action, use the `-v` flag to see the authentication in more detail:

```
mydesktop$ ssh ssh-server
OpenSSH_3.8.1p2, SSH protocols 1.5/2.0, OpenSSL 0x009060cf
...
debug1: identity file /home/xahria/.ssh/identity type 0
debug1: identity file /home/xahria/.ssh/id_rsa type 1
debug1: Remote protocol version 1.99, remote software version OpenSSH_3.7.1p2
...
debug1: SSH2_MSG_SERVICE_ACCEPT received
debug1: Authentications that can continue: publickey,password,keyboard-interactive
debug1: Next authentication method: publickey
debug1: Offering public key: /home/xahria/.ssh/id_rsa
debug1: Server accepts key: pkalg ssh-rsa blen 149 lastkey 0x601d0 hint 1
```

```

debug1: PEM_read_PrivateKey failed
debug1: read PEM private key done: type <unknown>
Enter passphrase for key '/home/xahria/.ssh/id_rsa': (Enter wrong passphrase)

debug1: PEM_read_PrivateKey failed
debug1: Next authentication method: keyboard-interactive
debug1: Authentications that can continue: publickey,password,keyboard-interactive
debug1: Next authentication method: password
xahria@ssh-server's password: (Enter Unix password)

```

For example's sake, we enter the wrong passphrase for the `id_rsa` Pubkey, so you can see how it fails back to standard password authentication. You can also notice at the beginning that it found two identities, `identity` and `id_rsa`, though it can only use the later with the SSHv2 protocol.

Selecting Keys

If you only have one key of each type, then you can use the standard filenames for them and your SSH client will find and offer them automatically. There may be times when you have more than one key and need to use a specific one for authentication, for example:

- You already have personal default keys, but want to use a group key (for example your departmental SSH key, etc) for certain hosts or target users.
- You have a bunch of keys that are offered automatically^[5] and the server is kicking you out for offering too many keys it doesn't trust.
- You want to use a specific key because it has special features associated with it, such as a restricted 'rsync only' key, as controlled by `authorized_keys` options.

To preference a particular key, use the `-i private_key_file` option on the command line:

```

mydesktop$ ssh -i ~/.ssh/special_ssh_key ssh-server
Enter passphrase for key '/home/xahria/.ssh/special_ssh_key':

```

Another option is to create an entry in your `~/.ssh/config` file to indicate the key to use:

```

mydesktop$ cat ~/.ssh/config
Host ssh-server
IdentityFile ~/.ssh/special_ssh_key

```

```
mydesktop$ ssh ssh-server
```

```
Enter passphrase for key '/home/xahria/.ssh/special_ssh_key':
```

Note that the `config` variable is `IdentityFile`, regardless if you're using a Pubkey or Identity.

Passphrase Security

Your private keys can, and almost always should, have a passphrase used to protect them. The passphrase is used to encrypt the key, making it unusable to anyone who does not know the passphrase. This prevents any local permissions problems, such as having the file world readable, from allowing others to read your key and use it to log into your other accounts.

Even if your key has restrictive permissions[6], you still should have a passphrase protect the key, lest the `root` user on your system be able to access your keys. It's true, `root` (or any cracker who has gained `root` access) has ultimate control, and could attack you in other ways, but there's no reason to make that job trivial.

We'll see in a future article how you can use the `ssh-agent`, which allows you have your keys protected with a passphrase, and yet not need to type it every time you use it.

authorized_keys2

Older versions of OpenSSH used two separate files for public key trust on the server, `authorized_keys` for Identities (SSHv1), and `authorized_keys2` for Pubkeys (SSHv2). Later on they decided this was just silly, and newer version will check for keys of either kind in `authorized_keys`, and will fall back to the `authorized_keys2` file in the event matching public keys are not found. Later versions of OpenSSH may well stop supporting the `authorized_keys2` file all together.

In order to not think about this limitation, what I do is maintain one file with all my keys named `authorized_keys`, and make a hard link under the `authorized_keys2` name in case I'm on one of these older OpenSSH servers:

```
ssh-server$ cd .ssh
```

```
ssh-server$ ls -l
```

```
-rw----- 1 xahria hatchclan 883 Jan 21 11:52 authorized_keys
```

```
# make a hard link so they are the same file, just different
# file names.
ssh-server$ ln authorized_keys authorized_keys2

ssh-server$ ls -l
-rw-----  2 xahria  hatchclan  883 Jan 21 11:52 authorized_keys
-rw-----  2 xahria  hatchclan  883 Jan 21 11:52 authorized_keys2
```

This setup will satisfy the needs of any version of OpenSSH.

Conclusion

This concludes our discussion on SSH Identity/Pubkey authentication. Stay tuned for future articles in this series, where we'll look at the use of ssh-agent as well as address other capabilities of SSH identities.

Notes:

- [1] This, RhostsRSAAuthentication, allows you to establish trust based on IP or hostname, where the connecting machine must prove its identity. It is similar to the rhosts authentication on which rsh and other obsolete protocols were based, but cannot be abused by IP spoofing.
- [2] Yes, ssh-keygen is the same program that was used to create our SSH host keys as well. There is no difference whatsoever between an Identity/Pubkey and a host key, they are all simply cryptographic public/private key pairs.
- [3] It would only use identity for an SSHv1 connection, or id_rsa/id_dsa for an SSHv2 connection.
- [4] The same magical mathematics performed by an SSH server when proving its host key identity.
- [5] One common way to have multiple keys beyond the three default files is by using the ssh-agent, which will be covered in a future article.
- [6] ssh-keygen will make the private key readable only by the user by default.

About the author

Brian Hatch is the author of [Hacking Linux Exposed, 2nd Edition](#), [Building Linux VPNs](#), and of the [Linux Security: Tips, Tricks, and Hackery Newsletter](#). In order to exit an xterm, he frequently needs to log out of ten or more SSH connections, each with cascaded port forwards, to get back to his desktop shell. And that's not even including all the virtual `/usr/bin/screen` TTYS.

More articles by this author

View [more articles](#) by Brian Hatch on SecurityFocus.

[Privacy Statement](#)

Copyright 2006, SecurityFocus