SSH Host Key Protection

by Brian Hatch
last updated October 14, 2004

This is the first in a series of articles on SSH in-depth. We start with looking at standard SSH host keys by examining the verification process to ensure you have not been the victim of an attack. Please note that this article applies to the widely used OpenSSH application that is bundled with most Unix based operating systems, and not the commercial version of SSH.

SSH Host Keys as a protection against Man-In-The-Middle Attacks

SSH is a ubiquitous protocol that offers secure, encrypted connections for a variety of purposes, including logging into remote machines, transferring files, setting up encrypted tunnels, running remote commands without manual authentication, and more. It was created to replace many non-encrypted protocols such as Telnet, FTP, RSH, and the like.

One of the problems with these old protocols, aside from the fact that they send everything (your password included) in the clear, is that they are vulnerable to man-in-the-middle attacks. A cracker with access to the intermediate network could intercept your packets, log them, and then send them to the actual destination. Even worse, she could re-write your packets, perhaps replacing `ls -l mydir` with `rm -r mydir`, or send you a trojaned file via your intercepted FTP session rather than the original. Since these protocols have no way to know with certainty whom you were talking to, any number of tricks were possible.

SSH offers a crucial feature -- the ability for you to verify the identity of the host to which you're connecting. If you correctly verify the host, then there's no way an intermediate device could be reading or manipulating your packets. [ref 1] Successful host verification indicates that the connection is encrypted end-to-end -- your SSH client has established a secure connection with the SSH server itself, and no intermediate machines have access to that connection.

Host verification is not unique to SSH. Any decent security-aware protocol has some sort of peer verification. For example, let's take a look at HTTPS, the SSL/TLS encrypted version of HTTP. It also offers host verification in much the same way as SSH. HTTPS host verification looks like the following. I've taken the liberty of mapping SSL-speak into SSH-speak to make it clearer to compare later on. [ref 2]

1. The client (web browser) connects to the server (HTTPS-aware webserver on port 443.)
2. The server provides its public key (X509 Certificate).
3. Some mystical, mathematical number crunching proves that the server has access to the private key associated with the public key.
4. The browser checks to see that the public key was signed by a trusted Certificate Authority (such as Verisign, Thawte, or others).
5. The browser checks that The CN (X509 Common Name) value in the server's certificate matches the host name you used. IE if connecting to https://www.example.com, then the certificate needs to have www.example.com as the CN value.

All of these step have their analogue in the world of SSH except for one: there is no Certificate Authority at all. [ref 3] Instead, the 'authority' are your personal and global configuration files, which we'll see later.

SSH Host Keys in Action

Let's see how SSH maps these steps by making an SSH connection to a machine we've never contacted before:

```
$ ssh ssh-server.example.com
The authenticity of host 'ssh-server.example.com (12.18.429.21)' can't be established.
RSA key fingerprint is 98:2e:d7:e0:de:9f:ac:67:28:c2:42:2d:37:16:58:4d.
Are you sure you want to continue connecting (yes/no)?
```

Let's say we blindly say "yes" here, and the connection will continue:

```
$ ssh ssh-server.example.com
Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'ssh-server.example.com,12.18.429.21' (RSA) to the list of known hosts.

Password: (enter password)

ssh-server.example.com $
```

The server presented its host key to the client as part of the initial handshake. The mathematical magic to prove it had access to this key is not seen above, indicating there were no errors in that regard.

Since we've never connected to this machine before, and SSH doesn't have the concept of a trusted third party like Certificate Authorities in the world of SSL/TLS, it's up to you to do all the key management yourself. Your client shows you the key fingerprint, an easy-to-read string of numbers that you can use to check the key manually -- we'll see this later. If you say "Yes, the fingerprint is correct", then your SSH client will continue logging in, allow you to type your password, and let you do your work.

When you said 'yes', above, your SSH client saved the server's host key locally in the file `$HOME/.ssh/known_hosts`. This file is, effectively, your personal Certificate Authority -- it is the list of all SSH server host keys that you have determined are accurate. Let's take a look at the last line of this file, which was just added on your behalf:

```
$ tail -1 $HOME/.ssh/known_hosts
ssh-server.example.com,12.18.429.21 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEA0
  6jFqviLMMJ/GaJNhGx/P6Z7+4aJIfUqcVjTGQasS1daDYejcfOAWK0juoD+zS3BsGKKYKPA
  5Gc5M8v+3NHLbPn1yTpDBgl6UzA0iiMPCbwnOLx61MrBTk+/qJI9kyDaJf4LEY6Chx4IJP0
  ZN5NmAlCtXQsca3jwFAF72mqPbF8=
```

(I've taken the liberty of wrapping and indenting this line for clarity.)

Each entry in `known_hosts` is one big line with three or more whitespace separated fields as follows:

1. One or more server names or IP addresses, joined together by commas.
2. The type of key (described later).
3. The public key data itself (encoded to stay within the ASCII range).
4. Any optional comment data (not present in the above output).

The next time you connect to this machine, your SSH client will go through the standard steps of verifying the remote machine and allowing you to log in:

```
$ ssh ssh-server.example.com
Password: (enter password)
```

Note that, this time, it did not ask you to verify the key fingerprint at all. That's because the key was in your `$HOME/.ssh/known_hosts` file. The SSH client actually checks in a few places:

1. The global known hosts file, typically `/etc/ssh/ssh_known_hosts`. This can be modified by changing the

GlobalKnownHostsFile parameter in the ssh configuration file (typically `/etc/ssh/ssh_config`).
2. The user's known hosts file, typically `$HOME/.ssh/known_hosts`. This can be modified by changing the UserKnownHostsFile parameter in the ssh configuration file.

As you should have noticed, when we connected originally and accepted the key, it stored both the hostname (`ssh-server.example.com`) and the IP address (`12.18.429.21`) so we can connect to this machine using either and it will be able to find and authenticate the host key from your `known_hosts` file.

Verifying the host key

I've shown you how to connect for the first time and accept the host key. But how do you know you've gotten the correct key? If you connected to the server, but an attacker has intercepted and proxied your SSH connection, she could have tricked you into accepting **her** key, not the actual host key. We've got a classic chicken and egg problem here.

The best way to verify a host key is to do so in some out-of-band method. For example the owner of the system could publish the key fingerprint on an SSL-protected web page. One place I worked actually distributed the host key of our main shell server on our laminated 'Emergency Contacts' phone number card, so you could verify it just by pulling out your wallet, even when sitting down at an Internet cafe.

If you don't have an out-of-band option, then the next best thing is to check the host key once you have logged in. The public host key is usually world readable in the `/etc/ssh/` directory, so upon logging in you can check the fingerprint of the key file (using ssh-keygen) against the key you accepted on the wire. (ssh-keygen is also used to create SSH host keys and Identities/PubKeys, which we'll discuss later.)

Going back to that first connection we made, let's see how we match up the fingerprint:

```
$ ssh ssh-server.example.com
The authenticity of host 'ssh-server.example.com (12.18.429.21)' can't be established.
RSA key fingerprint is 98:2e:d7:e0:de:9f:ac:67:28:c2:42:2d:37:16:58:4d.
Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'ssh-server.example.com,12.18.429.21' (RSA) to the list of known hosts.

Password: (enter password)

ssh-server.example.com $ cd /etc/ssh
ssh-server.example.com $ ls *.pub
ssh_host_dsa_key.pub  ssh_host_rsa_key.pub ssh_host_key.pub

ssh-server.example.com $ ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key.pub
1024 98:2e:d7:e0:de:9f:ac:67:28:c2:42:2d:37:16:58:4d ssh_host_rsa_key.pub
```

Above we see the user accept the key, log in, and then view the key with ssh-keygen manually. If the fingerprints match, you can be reasonably sure [ref 4] that your connection was made to the actual endpoint, even though you didn't know the host key ahead of time.

Host Key Checking Paranoia

SSH has three ways it can react to an unrecognised or changed SSH host key, based on the value of the `StrictHostKeyChecking` variable:

`StrictHostKeyChecking=no`

In this, the most insecure setting, it will blindly connect to the server. It will add the server's key if it's not present locally, and if the key has changed it will warn you and then add the key without asking. [ref 5]

This mode is generally not a good idea

This mode is generally not a good idea.

StrictHostKeyChecking=ask

This is the default setting. If you have no host key for the server, it will show you the fingerprint and ask you to confirm, as seen in the examples above. If you connect and the key does not match, then it will prevent you from logging in, and will tell you where to find the conflicting key inside the known_hosts file:

```
$ ssh -o stricthostkeychecking=ask ssh-server.example.com
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
23:00:20:83:de:02:95:f1:e3:34:be:57:3f:cf:2c:e7.
Please contact your system administrator.
Add correct host key in /home/xahria/.ssh/known_hosts to get rid of this message.
Offending key in /home/xahria/.ssh/known_hosts:8
RSA host key for localhost has changed and you have requested strict checking.
Host key verification failed.
```

StrictHostKeyChecking=yes

This is the most secure, perhaps even unfriendly, setting. If you have no host key for this server, then it will prevent you from logging in at all:

```
$ ssh -o 'StrictHostKeyChecking=yes' ssh-server.example.com
No RSA host key is known for localhost and you have requested strict checking.
Host key verification failed.
```

If you do have a host key but it does not match, you will get an error, just as with StrictHostKeyChecking=ask

Why Can A Host Key Change?

There are several reasons a host key can change or appear to be new. In the case of cracker activity, an error seen during the host key verification phase is the only way you'll know before you log in. However there are some other common problems that are not related to malicious activity that cause host key mismatches.

- Either the client or server software has changed, and now they are negotiating SSHv2 whereas before they were negotiating SSHv1. [ref 6]
- The machine has been re-installed with the same hostname, but the original keys were not restored. Since they've been created anew, they will of course not match your known_hosts file.
- The machine to which you wish to connect has been moved to a different DNS name or IP address, or it's been replaced by a new one entirely.

Types of Host Keys

SSH comes with two major protocols, SSHv2 and SSHv1. [ref 7]

The older SSHv1 protocol relied exclusively on RSA for it's asymmetric encryption, whereas the newer SSHv2 protocol supports ether RSA or DSA asymmetric encryption. An SSH server can use any of the three types of host keys: SSHv1 RSA keys, SSHv2 RSA keys, or SSHv2 DSA keys. I will refer to these as *rsa1*, *rsa*, and *dsa* keys respectively, as this is the terminology used by the OpenSSH tools.

SSH Host Keys are created with the ssh-keygen command. [ref 8] Most likely, when SSH was installed on your machine, the installation setup or the server startup script created these for you.

The sshd_config file, usually located in the /etc/ssh directory, lists which host keys it will load upon startup:

```
# Which protocol(s) should we support?
Protocol 2,1

# HostKey for protocol version 1
HostKey /etc/ssh/ssh_host_key

# HostKeys for protocol version 2
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
```

If you were installing from scratch, you'd want to create these three keys using ssh-keygen:

```
# ssh-keygen -t rsa /etc/ssh/ssh_host_rsa_key
# ssh-keygen -t dsa /etc/ssh/ssh_host_dsa_key
# ssh-keygen -t rsa1 /etc/ssh/ssh_host_key
```

The ssh-keygen program actually creates two files for each key. The first contains both the private and public key, and the second contains just the public key. So upon running the first ssh-keygen command above, you'd have created both /etc/ssh/ssh_host_rsa_key and /etc/ssh/ssh_host_rsa_key.pub. There's no reason not to allow the public key to be world readable, but the private key must be kept protected, since it has no passphrase associated with it. [ref 9] Luckily ssh-keygen sets paranoid permissions for you.

If you doubt that the first file, the one that doesn't end in .pub, contains the private key as well, or for some reason you lose the public key file, you can always regenerate it with the very same ssh-keygen command:

```
$ ls -1 /etc/ssh/ssh_host_rsa_key*
/etc/ssh/ssh_host_rsa_key
/etc/ssh/ssh_host_rsa_key.pub

$ cat /etc/ssh/ssh_host_rsa_key.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEApCyGZbDdzrRdszzQUZI+siu3
  /mUI57nmjzKwHS7M27AoMZNJ6yIDTn5J3/MVCDJAeyB53LvIFFD9Kzp6P9
  fhNhPm8+b0joJ5Wrn+YfUnt2moI3lkAzQUZI+siu3/mUI57nmjzKwH

$ ssh-keygen -y -f  /etc/ssh/ssh_host_rsa_key
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEApCyGZbDdzrRdszzQUZI+siu3
  /mUI57nmjzKwHS7M27AoMZNJ6yIDTn5J3/MVCDJAeyB53LvIFFD9Kzp6P9
  fhNhPm8+b0joJ5Wrn+YfUnt2moI3lkAzQUZI+siu3/mUI57nmjzKwH
```

Is there any reason to use one type of host key versus another? Not really. rsa keys will usually be a bit faster for the host key mathematics, but for greatest client interoperability it's best to include both. There are no patents on either algorithm [ref 10] so that is not a concern. If you need to support SSHv1, then obviously you must have an rsa1 key.

Tips

There are a number of tricks that can make managing host keys easier and more secure for you and/or you users.

- Maintain a global known hosts files (/etc/ssh/ssh_known_hosts) that contains all the machines to which your users will connect. If you take the time to verify these keys, then you do not need to rely on the users to do so independently. Make sure you get all three forms of the host key, rsa, dsa, and rsa1. Also, users to do so independently. Make sure you get all three forms of the host key, rsa, dsa, and rsa1. Also, should you ever need to change a host key (say the original machine has been re-installed and you forgot to save the old key) then you have only one place to update it.

- Maintain your global known hosts file with as many aliases as you might expect your users to try. For example you should include both `mail` and `mail.my_co.com` if that host can be reached by both names.
- If you have multiple machines that can be reached with the same name, such as a load balanced pair of webservers, then include both host keys under the shared name. The ssh client will check all matches until it finds one, so you can share a common DNS name without sharing the host key itself.
- Here's a very simple shell script that connects to an SSH server and adds all three potential host keys to a file. This may be useful when creating your global `ssh_known_hosts` file, but note that there's no actual verification here, that's up to you.

```
#!/bin/sh
#
# add-known-hosts
# Add all possible SSH keys for the specified hosts to the file
# specified.  It's your responsibility to be sure that the keys
# found are, in fact, valid.
#
# Copyright 2003, Brian Hatch <bri [@] ifokr.org>
#  Released under the GPL

KNOWN_HOSTS=./ssh_known_hosts
SSH_ARGS="-o StrictHostKeyChecking=no -o UserKnownHostsFile=$KNOWN_HOSTS"

if [ $# -lt 1 ] ; then
        echo "Usage: $0 hostname [hostname ...]" >&2
        exit 1;
fi

for host in "$@"
do
        ssh $host $SSH_ARGS -1 echo ''
        ssh $host $SSH_ARGS -o'HostKeyAlgorithms=ssh-rsa' echo ''
        ssh $host $SSH_ARGS -o'HostKeyAlgorithms=ssh-dss' echo ''
done
```

Configuration Options

There are several configuration options which might be useful, depending on how you are using SSH. These can live in either the global SSH configuration file, typically `/etc/ssh/ssh_config`, or the user configuration file, typically `$HOME/.ssh/config`. See the `ssh_config` manpage for more information.

- **StrictHostKeyChecking**: Determines if you should ignore errors with the server's host key when you connect, die immediately, or ask the user if they should continue.
- **CheckHostIP**: Determines if SSH will check for the server's IP address in the `known_hosts` file.
- **NoHostAuthenticationForLocalhost**: Turn off host key checking for the local machine only. Useful if you set up SSH Port Forwards to remote machines, ala `ssh -p 9999 localhost`, but you need to live with the consequences. Better to use HostKeyAliases as appropriate.
- **HostKeyAlias**: This option allows you to specify an 'alias' that will be used, instead of the actual hostname on the command line, when looking for a match in the `known_hosts` file. Particularly useful for commands

on the command line, when looking for a match in the known_hosts file. Particularly useful for commands that use ProxyCommands to connect, or are based on multiple ports on a machine that forward to different SSH servers behind it, such as a firewall.

- **HostKeyAlgorithms**: The algorithm you prefer, RSA or DSA, for protocol 2 host keys. RSA is the default, and unless you have a preference you may as well stick with it for performance purposes.

References

[ref 1] This is because the cryptography used to protect your session is based on the successful asymmetric encryption used to verify the host.

[ref 2] Note this is description is valid for any SSL/TLS capable service that has peer authentication enabled -- it need not be HTTPS, it could be MySQL or LDAP over SSL, for example.

[ref 3] There are patches to OpenSSH that allow actual X509 authentication, so you can get closer to this model if you desire.

[ref 4] It's always possible that a man-in-the-middle attacker is watching for this verification and can twiddle the output to convince you the keys match. If the attacker is this good, you've got lots of problems ahead of you.

[ref 5] It does, however, disable password authentication, so at least you must use SSH Identities/PubKeys, Challenge/Response, or some other form of authentication that is not reusable by an attacker.

[ref 6] The converse is possible, but unlikely.

[ref 7] SSHv1 is considered the less secure of the two versions. Unless you need to use client software that only supports the older SSHv1 protocol, for security reasons you are best off only enabling SSHv2 in your server. In /etc/ssh/sshd_config make sure your Protocol line reads as follows:

```
# Protocol 2,1 would allow either SSHv1 or SSHv2.
# Let's be paranoid and only support the later.
Protocol 2
```

[ref 8] ssh-keygen is the same tool that's used to create SSH Identities/PubKeys. There is really no difference between a user key and a host key.

[ref 9] The private key must be unprotected because otherwise sshd could not start up without administrator intervention upon reboot.

[ref 10] The patent on the RSA algorithm expired in 2000.

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.