Tighter SSH Security with Two-Factor Authentication

Paul Sery

Abstract

How to set up two-factor authentication using a USB pendrive and ssh-agent for root logins.

I enthusiastically use two-factor authentication whenever possible, because static passwords aren't the best mechanism around any moat. Traditional passwords are vulnerable to social engineering, key-loggers, yellow post-it notes and—especially as computers become ever faster—to cracking. Tossing them in favor of two-factor authentication is a good idea and helps me sleep better at night.

Unfortunately, network-based, commercial two-factor systems are generally too expensive and complex to use at home or on small networks. But, guess what? You already have the necessary parts on your Linux computer to build a two-factor authentication system. The ubiquitous secure communication tool, OpenSSH, provides all the tools necessary to create a host-based, two-factor authentication system suitable for the home, small office and even larger networks.

This article describes how to combine removable media with OpenSSH public/private keys and the amazing ssh-agent program to achieve two-factor authentication for both regular and privileged users.

Example 1: Two-Factor User Authentication Using USB Drives

Let's start by creating two-factor authentication for regular (nonroot) users. In this case, we use the well-known SSH public key authentication facility with a small twist. Rather than store the private key in the .ssh subdirectory of your home directory, as is the default, we'll place it on a USB pendrive.

For this example, you'll be logged in as the nonprivileged user bob on a Fedora Core computer, machine1. You'll connect to the remote Linux box machine2 as bob.

Let's start by creating the public/private key pair that we'll use to log in to machine2:

```
ssh-keygen -t rsa -f key-rsa-bob@machine2 -C key-rsa-bob@machine2
```

Enter a passphrase when prompted (the longer and more random the better). By default, the ssh-keygen program creates the key pair in the subdirectory .ssh in your home directory—in this case, /home/bob/.ssh. For this example, I've chosen an arbitrary yet descriptive filename to help identify the intended user and hostname at a glance; this will be important in the succeeding examples, which use multiple keys. (I'm assuming the USB drive is formatted with a Linux filesystem like ext3; vfat works, but you'll need to change the key's file permissions to 400 after every mount.)

Mount your USB pendrive, and you should see it as as /media/usbdisk, /media/usbdisk1, /media/disk or /media/disk-1. Move your newly created private key to the appropriate directory and limit access to the owner:

```
mv key-rsa-bob@machine2 /media/usbdisk
chmod 400 /media/usbdisk/key-rsa-bob@machine2
```
Next, copy the public key (key-rsa-bob@machine2.pub) into the /home/bob/.ssh/authorized_keys file on machine2. Make the authorized_keys file readable only by the owner:

```bash
chmod 400 authorized_keys
```

Now, you can log in to the remote computer, machine2, from machine1, as bob, using the public/private key pairs (the -i option tells the ssh client what key to use):

```bash
ssh -i /media/usbdisk/key-rsa-bob@machine2 bob@machine2
```

Type in the private key passphrase when prompted, and the OpenSSH server on machine2 logs you in. Unmount and remove the USB device (or removable disc) on machine1, and your private key is protected. You've achieved two-factor authentication: one factor is the key stored on the USB device that you can keep separate from your computer, and the second one is the passphrase you store in your head.

Using SSH public key authentication is a common and familiar process to many. Putting the private key onto removable media is a simple way of physically separating one factor from another.

**Example 2: Two-Factor Root Authentication Using ssh-agent**

Example 1 shows how to log in to a remote machine securely using a USB device to separate one authentication factor from another. This works well when logging in as a nonprivileged user but not as root. We have to find a way to log in remotely as the superuser.

One solution would be simply to extend the previous example's method and configure the remote OpenSSH server to allow root logins directly from the network. No passwords or keys will traverse the network, but we would violate the age-old system administration prohibition against directly logging in as root. No shortcuts should be allowed, so we have to figure out how to first log in as a regular user and then as root.

Once again, OpenSSH comes to the rescue. In this case, we continue to use public/private keys but introduce a configuration twist. First, configure the remote SSH service to allow root logins via the internal loopback interface but not the external network. Second, configure the ssh-agent utility to allow the remote machine to authenticate root by querying the keys stored on the local machine.

Here's how the process works:

1. Create a private/public key pair for root on the local machine.
2. Copy the public key into root's authorized_users file on the remote machine.
3. Run the ssh-add utility locally to cache the private key.
4. `ssh` to the remote machine and log in as a regular user as described in Example 1; however, this time use the agent-forwarding option.
5. On the remote machine, `ssh` to the localhost interface as the root user. The remote OpenSSH daemon queries the local agent, authenticates root, and you can log in as the superuser.

The ssh-agent utility provides just the functionality we're looking for. It allows remote SSH daemons to authenticate users by querying the locally stored cache of decrypted private keys. Keys are never transmitted between machines—the private keys remain stored on removable media on your local workstation.

ssh-agent is powerful, but setting it up can be tricky. First, you need to use the ssh-add utility to decrypt your
private key and hand it to ssh-agent. Second, you need to tell ssh-add how to communicate with ssh-agent. ssh-add communicates with ssh-agent via a socket, whose location is stored in the SSH_AUTH_SOCK environmental variable. By default, ssh-agent creates sockets with arbitrary names, and setting SSH_AUTH_SOCK correctly can take some work.

Fortunately, many Linux distributions, including Fedora Core, automatically set up the necessary ssh-agent/ssh-add connections when you log in graphically (such as on GNOME or KDE). Log in at the console, open a terminal console and type the following:

```
ssh-add -l
```

As long as ssh-add can communicate with ssh-agent, you should see either a list of your public keys or a message like “The agent has no identities”.

If, for any reason, ssh-agent isn't running or your SSH_AUTH_SOCK variable isn't set, or isn't set correctly, you will get the message “Could not open a connection to your authentication agent”. In that case, run the following command:

```
eval `ssh-agent`
```

This starts an ssh-agent instance and automatically sets the environmental variables in your current shell.

Next, create a key pair for root as you did in the first example:

```
ssh-keygen -t rsa -f key-rsa-root@machine2 -C "key-rsa-root@machine2"
```

Move the private key to the removable media and give read access to the owner but nobody else:

```
mv key-rsa-root@machine2 /media/usbdisk
chmod 400 /media/usbdisk/key-rsa-root@machine2
```

Copy the public key into the /root/.ssh/authorized_keys file on the remote computer machine2.

Add root's private key on machine2 to ssh-agent by running the following command:

```
ssh-add -t 300 /media/usbdisk/key-rsa-root@machine2
```

Enter the passphrase when prompted, and ssh-agent returns the message “Identity added: key-rsa-root@machine2 (key-rsa-root@machine2)” when it adds the key. (The -t 300 option limits the lifetime of the cache to 300 seconds, or five minutes. Your keys will remain viable forever if you don't specify the lifetime.)

Log in to the remote machine as a regular user:

```
ssh -A -i /media/usbdisk/key-rsa-bob@machine2
```

Enter the passphrase when prompted, and you will log in to machine2. (This command is the same as in Example 1, except we're using the -A option, which turns on agent forwarding.)

Type `ssh-add -l` on machine2, and you should see the root key you just added to ssh-agent. For example:

```
2048 fa:5c:4b:73:88:26::... /media/usbdisk/key-rsa-root@machine2 (rsa)
```

Next, `su` to root (on machine2), and configure the SSH daemon to allow root logins on the internal loopback interface. Edit the /etc/ssh/sshd_config file and add/modify the following options:

```
PermitRootLogin yes
```
AllowUsers bob@
AllowUsers root@localhost.

(Some OpenSSH configurations require you to set the numeric loopback address explicitly: AllowUsers root@127.0.0.1.)

Save your changes, and restart the SSH daemon:

    service sshd restart

Log out of the root account, and use OpenSSH to log back in as root:

    ssh root@localhost

Now the OpenSSH daemon on machine2 accepts root logins on the loopback interface but not from the external network. It negotiates with ssh-agent on machine1 to authenticate you as the root user. root's private key never left machine1! Using OpenSSH in this way effectively allows you to replace the su (switch user) and sudo utilities.

But, we're not quite finished. You can increase security further by limiting the su command to locally connected devices. Modify /etc/pam.d/su as shown below to prevent anyone from using su over the network:

        auth  required  pam_securetty.so

The su command will work only from the console and virtual terminals.

Unmount and remove your USB device. Individuals actually will have to steal your USB drive at this point to get your keys. Even then, they have to discover your passphrase or expend lots of computing power and time cracking the key.

**Example 3: Tightening Up**

We need to close a potential vulnerability before using this system in the wild.

Using ssh-agent and agent forwarding allows the remote SSH server to query the private key stored on your local computer. However, if you use this system to log in to multiple computers, an intruder on one machine can potentially highjack those keys to break in to another machine. In that case, this system could be more dangerous than one using static passwords.

To illustrate the problem, let's expand our example network from two to three nodes by adding machine3 to the mix. Create key pairs for both bob and root on machine3, as described in Examples 1 and 2, and add root's private key to ssh-agent on machine1.

Now, ssh to machine3 as bob using the agent-forwarding option -A. Run ssh-add -l, and you can see the public keys for both machine2 and machine3:

    2048 fa:5c:4b:73:88:...: ... /media/usbdisk/key-rsa-root@machine2 (RSA)
    2048 26:b6:e3:99:c1:...: ... /media/usbdisk/key-rsa-root@machine3 (RSA)

In this example, ssh-agent on machine1 caches the private keys for machine2 and machine3. This single agent allows us to log in as root on either computer. However, using the single agent also potentially allows an intruder on machine2 to log in as root on machine3 and vice versa. This is not good.

Fortunately, we can fix this problem by using the ssh-add -c option; we can add additional security by using individual ssh-agent instances to store one root key for each remote machine. The -c option tells ssh-agent to
have the user confirm each use of a cached key. Devoting one ssh-agent instance per host prevents any as yet unknown ssh-agent vulnerability from exposing one machine's key to another.

Using the ssh-add confirm option is easy; simply set the -c option whenever adding a key to ssh-agent. Let's give it a try. Start two agents on machine1, specifying predefined sockets:

```bash
ssh-add -c /media/usbdisk/key-rsa-root@machine2
ssh-add -c /media/usbdisk/key-rsa-root@machine3
```

You'll be asked to confirm use of the key when you `ssh` to machine2 and machine3.

You also can use separate ssh-agents to store each key. Let's give it a try; start two agents on machine1, specifying predefined sockets:

```bash
ssh-agent -a /tmp/ssh-agent-root@machine2
ssh-agent -a /tmp/ssh-agent-root@machine3
```

Once again, I'm using an arbitrary yet descriptive naming convention. Set the environmental variable, and add the key for machine2:

```bash
export SSH_AUTH_SOCK=/tmp/ssh-agent-root@machine2
ssh-add -c /media/usbdisk/key-rsa-root@machine2
```

Repeat this process for machine3, making the appropriate substitutions:

```bash
export SSH_AUTH_SOCK=/tmp/ssh-agent-root@machine3
ssh-add -c /media/usbdisk/key-rsa-root@machine3
```

Now, log in to machine3 (we'll go to machine3 at this point as we just set the SSH_AUTH_SOCK variable to point to machine3's agent):

```bash
ssh -A -i /media/usbdisk/key-rsa-bob@machine2 bob@machine3
```

Run the following command to see what keys you can query on machine1:

```bash
ssh-add -l
```

You see only the key for root on machine3.

Exit from machine3, change the environmental variable to the machine2 ssh-agent socket, and log in to machine2:

```bash
export SSH_AUTH_SOCK=/tmp/ssh-agent-root@machine2
ssh -A -i /media/usbdisk/key-rsa-bob@machine2 bob@machine2
```

Check your keys again:

```bash
ssh-add -l
```

Checking your keys on machine2 and machine3 reveals only the root key for that machine. In the previous example, by using a single ssh-agent, you would have seen the keys for both machine2 and machine3.

Using separate ssh-agent instances for each machine you log in to requires more work.

Resetting the SSH_AUTH_SOCK variable every time you want to log in to another machine is impractical. To simplify the process, I've written a simple script tfssh (two-factor ssh) to simplify the process. Its syntax is:
tfssh [username@]host [keydir]

The script [Listing 1 on the LJ FTP site at ftp://ftp.ssc.com/pub/lj/listings/issue152/8957.tgz] starts ssh-agent when necessary, sets the environmental variable, adds the root keys to ssh-agent and logs in to the remote machine as the user. You also can tell tfssh to look in an arbitrary directory ([keydir]) for its keys and also set a key timeout for the key cache.

**ssh-add**

ssh-add allows you to lock and/or confirm using private keys. Use the -x and -X options to lock and unlock a key. You will create a password to lock the key, and use the password to unlock it. Using the -c option directs ssh-add to prompt you every time ssh-agent is asked to use a key. The prompt is displayed on the machine running ssh-agent and effectively prevents unauthorized users from using your keys.

**Conclusion**

Static passwords are quickly becoming more trouble than they're worth. We need to break the static habit and start using two-factor authentication. OpenSSH is a powerful system that provides the tools necessary to make that step. By using public/private keys, agent forwarding and removable media, we can use OpenSSH as a key “safe”. This, in turn, allows us to create a simple, inexpensive and effective host-based, two-factor authentication system.

This two-factor system requires a moderate amount of work to configure and use, but it is well worth the extra security. However, using the tfssh script makes the process easy to use. Using the script means you get all the benefits of two-factor authentication but almost none of the hassle.

**Two vs. 2.X Factors**

Some people count the locally stored SSH keys and their passphrases as two factors. This view is reasonable, but I feel more comfortable physically separating the key storage device from the computer. Keeping your keys on removable media reduces the opportunity for intruders to capture and crack them.

Now, it's important to realize that keeping your keys on devices like USB pendrives doesn't eliminate the ability of an intruder to spy them. Your keys are vulnerable while mounted, and you should take precautions to harden the workstation from which you connect to other computers. Use good passwords for local (console) logins, keep your workstation patched and so on.

So, you're better off using public key authentication than static passwords, as long as you adequately protect your workstation. How safe you want to be depends on your paranoia.

**Storing Keys**

You can store your keys on any type of removable media. I'm using a USB pendrive in these examples because it's easy to work with and carry around. Feel free to use writable CD-ROMs or DVDs or even floppies if you want.