Common threads: OpenSSH key management, Part 1

Understanding RSA/DSA authentication

Daniel Robbins (drobbins@gentoo.org)
President/CEO, Gentoo Technologies, Inc.
July 2001

In this series, you'll learn how RSA and DSA authentication work, and see how to set up passwordless authentication the right way. In the first article of the series, Daniel Robbins focuses on introducing the RSA and DSA authentication protocols and showing you how to get them working over the network.

Contents:

Related content:

Addressing security issues in Linux
More Linux resources
More Open source resources

Many of us use the excellent OpenSSH (see Resources later in this article) as a secure, encrypted replacement for the venerable `telnet` and `rsh` commands. One of OpenSSH's more intriguing features is its ability to authenticate users using the RSA and DSA authentication protocols, which are based on a pair of complementary numerical keys. As one of its main appeals, RSA and DSA authentication promise the capability of establishing connections to remote systems *without supplying a password*. While this is appealing, new OpenSSH users often configure RSA/DSA the quick and dirty way, resulting in passwordless logins, but opening up a big security hole in the process.

What is RSA/DSA authentication?

SSH, specifically OpenSSH (a completely free implementation of SSH), is an incredible tool. Like `telnet` or `rsh`, the `ssh` client can be used to log in to a remote machine. All that's required is for this remote machine to be running `sshd`, the `ssh` server process. However, unlike `telnet`, the `ssh` protocol is very secure. It uses special algorithms to encrypt the data stream, ensure data stream integrity and even perform authentication in a safe and secure way.

However, while `ssh` is really great, there is a certain component of `ssh` functionality that is often ignored, dangerously misused, or simply misunderstood. This component is OpenSSH's RSA/DSA key authentication system, an alternative to the standard secure password authentication system that OpenSSH uses by default.

OpenSSH's RSA and DSA authentication protocols are based on a pair of specially generated cryptographic keys, called the *private key* and the *public key*. The advantage of using these key-based authentication systems is that in many cases, it's possible to establish secure connections without having to manually type in a password.

While the key-based authentication protocols are relatively secure, problems arise when users take certain shortcuts in the name of convenience, without fully understanding their security implications. In this article, we'll take a good look at how to correctly use RSA and DSA authentication protocols without exposing ourselves to any unnecessary security risks. In my next article, I'll show you how to use `ssh-agent` to cache decrypted private keys, and introduce `keychain`, an `ssh-agent` front-end that offers a number of convenience advantages without sacrificing security. If you've always wanted to get the hang of the more advanced authentication features of OpenSSH, then read on.

How RSA/DSA keys work

Here's a quick general overview of how RSA/DSA keys work. Let's start with a hypothetical scenario where we'd like to use RSA authentication to allow a local Linux workstation (named *localbox*) to open a remote shell on

remotebox, a machine at our ISP. Right now, when we try to connect to *remotebox* using the `ssh` client, we get the following prompt:

```
% ssh drobbins@remotebox
drobbins@remotebox's password:
```

Here we see an example of the `ssh` *default* way of handling authentication. Namely, it asks for the password of the *drobbins* account on *remotebox*. If we type in our password for *remotebox*, `ssh` uses its secure password authentication protocol, transmitting our password over to *remotebox* for verification. However, unlike what `telnet` does, here our password is encrypted so that it can not be intercepted by anyone sniffing our data connection. Once *remotebox* authenticates our supplied password against its password database, if successful, we're allowed to log on and are greeted with a *remotebox* shell prompt. While the `ssh` default authentication method is quite secure, RSA and DSA authentication open up some new possibilities.

However, unlike the `ssh` secure password authentication, RSA authentication requires some initial configuration. We need to perform these initial configuration steps only once. After that, RSA authentication between *localbox* and *remotebox* will be totally painless. To set up RSA authentication, we first need to generate a pair of keys, one private and one public. These two keys have some very interesting properties. The public key can be used to encrypt a message, and only the holder of the private key can decrypt it. The public key can only be used for *encryption*, and the private key can only be used for *decryption* of a message encoded by the matching public key. The RSA (and DSA) authentication protocols use the special properties of key pairs to perform secure authentication, without needing to transmit any confidential information over the network.

To get RSA or DSA authentication working, we perform a single one-time configuration step. We copy our *public key* over to *remotebox*. The public key is called "public" for a reason. Since it can only be used to *encrypt* messages for us, we don't need to be too concerned about it falling into the wrong hands. Once our public key has been copied over to *remotebox* and placed in a special file (~/.ssh/authorized_keys) so that *remotebox*'s `sshd` can locate it, we're ready to use RSA authentication to log onto *remotebox*.

To do this, we simply type `ssh drobbins@remotebox` at *localbox*'s console, as we always have. However, this time, `ssh` lets *remotebox*'s `sshd` know that it would like to use the RSA authentication protocol. What happens next is rather interesting. *Remotebox*'s `sshd` generates a random number, and encrypts it using our public key that we copied over earlier. Then, it sends this encrypted random number back to the `ssh` running on *localbox*. In turn, our `ssh` uses our *private key* to decrypt this random number, and then sends it back to *remotebox*, saying in effect "See, I really *do* hold the matching private key; I was able to successfully decrypt your message!" Finally, `sshd` concludes that we should be allowed to log in, since we hold a matching private key. Thus, the fact that we hold a matching private key grants us access to *remotebox*.

Two observations

There are two important observations about the RSA and DSA authentication. The first is that we really only need to generate one pair of keys. We can then copy our public key to the remote machines that we'd like to access and they will all happily authenticate against our single private key. In other words, we don't need a key pair for *every* system we'd like to access. Just one pair will suffice.

The other observation is that our *private key should not fall into the wrong hands*. The private key is the one thing that grants us access to our remote systems, and anyone that possesses our private key is granted exactly the same privileges that we are. Just as we wouldn't want strangers to have keys to our house, we should protect our private key from unauthorized use. In the world of bits and bytes, this means that no one should be able to read or copy our private key.

Of course, the `ssh` developers are aware of the private keys' importance, and have built a few safeguards into `ssh` and `ssh-keygen` so that our private key is not abused. First, `ssh` is configured to print out a big warning message if our key has file permissions that would allow it to be read by anyone but us. Secondly, when we create our public/private key pair using `ssh-keygen`, `ssh-keygen` will ask us to enter a passphrase. If we do, our private key will be encrypted using this passphrase, so that even if it is stolen, it will be useless to anyone who doesn't happen to know the passphrase. Armed with that knowledge, let's take a look at how to configure `ssh` to use the RSA and DSA authentication protocols.

ssh-keygen up close

The first step in setting up RSA authentication begins with generating a public/private key pair. RSA authentication is the original form of `ssh` key authentication, so RSA should work with any version of OpenSSH, although I recommend that you install the most recent version available, which was openssh-2.9_p2 at the time this article was written. Generate a pair of RSA keys as follows:

```
% ssh-keygen
Generating public/private rsa1 key pair.
Enter file in which to save the key (/home/drobbins/.ssh/identity): (enter a
passphrase)
Enter passphrase (empty for no passphrase): (enter it again)
Enter same passphrase again: (hit enter)
Your identification has been saved in /home/drobbins/.ssh/identity.
Your public key has been saved in /home/drobbins/.ssh/identity.pub.
The key fingerprint is:
a4:e7:f2:39:a7:eb:fd:f8:39:f1:f1:7b:fe:48:a1:09 drobbins@localbox
```

When `ssh-keygen` asks for a default location for the key, we hit enter to accept the default of /home/drobbins/.ssh/identity. `ssh-keygen` will store the private key at the above path, and the *public* key will be stored right next to it, in a file called identity.pub.

Also note that `ssh-keygen` prompted us to enter a passphrase. When prompted, we entered a good passphrase (seven or more hard-to-predict characters). `ssh-keygen` then encrypted our private key (~/.ssh/identity) using this passphrase so that our private key will be useless to anyone who does not know it.

The quick compromise

When we specify a passphrase, it allows `ssh-keygen` to secure our private key against misuse, but it also creates a minor inconvenience. Now, every time we try to connect to our *drobbins@remotebox* account using `ssh`, `ssh` will prompt us to enter the passphrase so that it can decrypt our private key and use it for RSA authentication. Again, we won't be typing in our password for the *drobbins* account on *remotebox*, we'll be typing in the passphrase needed to locally decrypt our private key. Once our private key is decrypted, our `ssh` client will take care of the rest. While the mechanics of using our remote password and the RSA passphrase are completely different, in practice we're still prompted to type a "secret phrase" into `ssh`.

```
# ssh drobbins@remotebox
Enter passphrase for key '/home/drobbins/.ssh/identity': (enter passphrase)
Last login: Thu Jun 28 20:28:47 2001 from localbox.gentoo.org

Welcome to remotebox!

%
```

Here's where people are often mislead into a quick compromise. A lot of the time, people will create unencrypted private keys just so that they don't need to type in a password. That way, they simply type in the `ssh` command, and they're immediately authenticated via RSA (or DSA) and logged in.

```
# ssh drobbins@remotebox
Last login: Thu Jun 28 20:28:47 2001 from localbox.gentoo.org

Welcome to remotebox!

%
```

However, while this is convenient, you shouldn't use this approach without fully understanding its security impact. With an unencrypted private key, if anyone ever hacks into *localbox*, they'll also get automatic access to *remotebox* and any other systems that have been configured with the public key.

I know what you're thinking. Passwordless authentication, despite being a bit risky does seem really appealing. I totally agree. But *there is a better way!* Stick with me, and I'll show you how to gain the benefits of passwordless authentication without compromising your private key security. I'll show you how to masterfully use `ssh-agent` (the thing that makes *secure* passwordless authentication possible in the first place) in my next article. Now, let's get ready to use `ssh-agent` by setting up RSA and DSA authentication. Here step-by-step directions.

RSA key pair generation
To set up RSA authentication, we'll need to perform the one-time step of generating a public/private key pair. We do this by typing:

```
% ssh-keygen
```

Accept the default key location when prompted (typically ~/.ssh/identity and ~/.ssh/identity.pub for the public key), and provide `ssh-keygen` with a secure passphrase. Once `ssh-keygen` completes, you'll have a public key as well as a passphrase-encrypted private key.

RSA public key install
Next, we'll need to configure remote systems running `sshd` to use our *public* RSA key for authentication. Typically, this is done by copying the public key to the remote system as follows:

```
% scp ~/.ssh/identity.pub drobbins@remotebox:
```

Since RSA authentication isn't fully set up yet, we'll be prompted to enter our password on *remotebox*. Do so. Then, log in to *remotebox* and append the public key to the ~/.ssh/authorized_keys file like so:

```
% ssh drobbins@remotebox
drobbins@remotebox's password: (enter password)
Last login: Thu Jun 28 20:28:47 2001 from localbox.gentoo.org

Welcome to remotebox!

% cat identity.pub >> ~/.ssh/authorized_keys
% exit
```

Now, with RSA authentication configured, we should be prompted to enter our RSA *passphrase* (rather than our *password*) when we try to connect to *remotebox* using `ssh`.

```
% ssh drobbins@remotebox
Enter passphrase for key '/home/drobbins/.ssh/identity':
```

Hurray, RSA authentication configuration complete! If you weren't prompted for a passphrase, here are a few things to try. First, try logging in by typing `ssh -1 drobbins@remotebox`. This will tell `ssh` to only use version 1 of the ssh protocol, and may be required if for some reason the remote system is defaulting to DSA authentication. If that doesn't work, make sure that you don't have a line that reads `RSAAuthentication no` in your /etc/ssh/ssh_config. If you do, comment it out by pre-pending it with a "#". Otherwise, try contacting the *remotebox* system administrator and verifying that they have enabled RSA authentication on their end and have the appropriate settings in /etc/ssh/sshd_config.

DSA key generation
While RSA keys are used by version 1 of the `ssh` protocol, DSA keys are used for protocol level 2, an updated version of the `ssh` protocol. Any modern version of OpenSSH should be able to use both RSA and DSA keys. Generating DSA keys using OpenSSH's `ssh-keygen` can be done similarly to RSA in the following manner:

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

Again, we'll be prompted for a passphrase. Enter a secure one. We'll also be prompted for a location to save our DSA keys. The default, normally ~/.ssh/id_dsa and ~/.ssh/id_dsa.pub, should be fine. After our one-time DSA key generation is complete, it's time to install our DSA public key to remote systems.

DSA public key install

Again, DSA public key installation is almost identical to RSA. For DSA, we'll want to copy our ~/.ssh/id_dsa.pub file to *remotebox*, and then append it to the ~/.ssh/authorized_keys2 on *remotebox*. Note that this file has a different name than the RSA authorized_keys file. Once configured, we should be able to log in to *remotebox* by typing in our DSA private key passphrase rather than typing in our actual *remotebox* password.

Next time

Right now, you should have RSA or DSA authentication working, but you still need to type in your passphrase for every new connection. In my next article, we'll see how to use `ssh-agent`, a really nice system that allows us to establish connections *without* supplying a password, but also allows us to keep our private keys encrypted on disk. I'll also introduce `keychain`, a very handy `ssh-agent` front-end that makes `ssh-agent` even more secure, convenient, and fun to use. Until then, check out the handy resources below to keep yourself on track.

Resources

- Be sure to visit the home of OpenSSH development.
- Take a look at the latest OpenSSH source tarballs and RPMs.
- Check out the OpenSSH FAQ.
- PuTTY is an excellent `ssh` client for Windows machines.
- You may find O'Reilly's *SSH, The Secure Shell: The Definitive Guide* to be helpful. The authors' site contains information about the book, a FAQ, news, and updates.
- Read Addressing security issues in Linux on *developerWorks* for an overview of data encryption and many other security topics.
- Browse more Linux resources on *developerWorks*.
- Browse more Open source resources on *developerWorks*.

About the author

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of Gentoo Technologies, Inc., the creator of Gentoo Linux, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at **SONY Electronic Publishing/Psygnosis**. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah. You can contact Daniel at drobbins@gentoo.org.

What do you think of this article?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?

Submit feedback