Common threads : OpenSSH key management, Part 2

e-mail it!

Introducing ssh-agent and keychain

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

Related content:
OpenSSH key management, Part 1
More dW Linux resources
More dW Open source project resources

Many developers use the excellent OpenSSH 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 upon a pair of complementary numerical "keys". One of the main appeals of RSA and DSA authentication is the promise of being able to establish connections to remote systems *without supplying a password*. In this second article, Daniel introduces `ssh-agent` (a private key cache) and `keychain`, a special bash script designed to make key-based authentication incredibly convenient and flexible.

Introducing ssh-agent

`ssh-agent`, included with the OpenSSH distribution, is a special program designed to make dealing with RSA and DSA keys both pleasant and secure (see Part 1 of this series for an introduction to RSA and DSA authentication.) `ssh-agent`, unlike `ssh`, is a long-running daemon designed for the sole purpose of caching your decrypted private keys.

`ssh` includes built-in support that allows it to communicate with `ssh-agent`, allowing `ssh` to acquire your decrypted private keys without prompting you for a password for every single new connection. With `ssh-agent` you simply use `ssh-add` to add your private keys to `ssh-agent`'s cache. It's a one-time process; after using `ssh-add`, `ssh` will grab your private key from `ssh-agent`, rather than bugging you by prompting for a passphrase.

Using ssh-agent

Let's take a look at how this whole `ssh-agent` key caching system works. When `ssh-agent` starts up, it spits out a few important environment variables before detaching from the shell and continuing to run in the background. Here's some example output generated by `ssh-agent` when it begins:

```
% ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-XX4LkMJS/agent.26916; export SSH_AUTH_SOCK;
SSH_AGENT_PID=26917; export SSH_AGENT_PID;
echo Agent pid 26917;
```

As you can see, `ssh-agent`'s output is actually a series of bash commands; if executed, these commands would set a couple of environment variables, SSH_AUTH_SOCK and SSH_AGENT_PID. Due to the included export commands, these environment variables would be made available to any additional commands run later. Well, all that would happen *if* these lines were actually evaluated by the shell, but right now they're simply printed to stdout. To fix this, we can invoke `ssh-agent` in the following way:

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

This command tells bash to run `ssh-agent` and then evaluate `ssh-agent`'s output. Invoked this way (with back-quotes, not normal single quotes), the SSH_AGENT_PID and SSH_AUTH_SOCK variables get set and exported by your shell, making these variables available to any new processes you may start during your login session.

The best way to start `ssh-agent` is to add the above line to your ~/.bash_profile; that way, all programs started in your login shell will see the environment variables, be able to locate `ssh-agent` and query it for keys as needed. The environment variable of particular importance is SSH_AUTH_SOCK; the SSH_AUTH_SOCK contains a path to a UNIX domain socket that `ssh` and `scp` can use to establish a dialogue with `ssh-agent`.

Using ssh-add
But of course, `ssh-agent` starts up with an empty cache of decrypted private keys. Before we can really use `ssh-agent`, we first need to add add our private key(s) to `ssh-agent`'s cache using the `ssh-add` command. In the following example, I use `ssh-add` to add my ~/.ssh/identity private RSA key to `ssh-agent`'s cache:

```
# ssh-add ~/.ssh/identity
Need passphrase for /home/drobbins/.ssh/identity
Enter passphrase for /home/drobbins/.ssh/identity
(enter passphrase)
```

As you can see, `ssh-add` asked for my passphrase so that the private key can be decrypted and stored in `ssh-agent`'s cache, ready for use. Once you've used `ssh-add` to add your private key (or keys) to `ssh-agent`'s cache *and* SSH_AUTH_SOCK is defined in your current shell (which it should be, if you started `ssh-agent` from your ~/.bash_profile), then you can use `scp` and `ssh` to establish connections with remote systems without supplying your passphrase.

Limitations of ssh-agent
`ssh-agent` is really cool, but its default configuration still leaves us with a few minor inconveniences. Let's take a look at them.

For one, with `` eval `ssh-agent` `` in ~/.bash_profile, a new copy of `ssh-agent` is launched for every login session; not only is this a tad bit wasteful, but it also means that you need to use `ssh-add` to add a private key to each new copy of `ssh-agent`. If you only open a single terminal or console on your system, this is no big deal, but most of us open quite a few terminals and need to type in our passphrase every single time we open a new console. Technically, there's no reason why we should need to do this since a single `ssh-agent` process really should suffice.

Another problem with the default `ssh-agent` setup is that it's not compatible with cron jobs. Since cron jobs are started by the cron process, they won't inherit the SSH_AUTH_SOCK variable from their environment, and thus won't know that a `ssh-agent` process is running or how to contact it. It turns out that this problem is also fixable.

Enter keychain
To solve these problems, I wrote a handy bash-based `ssh-agent` front-end called `keychain`. What makes `keychain` special is the fact that it allows you to use a single `ssh-agent` process *per system*, not just per login session. This means that you only need to do one `ssh-add` per private key, period. As we'll see in a bit, `keychain` even helps to optimize the `ssh-add` process by only trying to add private keys that aren't already in the running `ssh-agent`'s cache.

Here's a run-through of how `keychain` works. When started from your ~/.bash_profile, it will first check to see whether an `ssh-agent` is already running. If not, then it will start `ssh-agent` and record the important SSH_AUTH_SOCK and SSH_AGENT_PID variables in the ~/.ssh-agent file for safe keeping and later use. Here's the best way to start `keychain`; like using plain old `ssh-agent`, we perform the necessary setup inside ~/.bash_profile:

```
#!/bin/bash
#example ~/.bash_profile file
/usr/bin/keychain ~/.ssh/id_rsa
#redirect ~/.ssh-agent output to /dev/null to zap the annoying
#"Agent PID" message
source ~/.ssh-agent > /dev/null
```

As you can see, with `keychain` we source the ~/.ssh-agent file rather than evaluating output as we did when using `ssh-agent` directly. However, the result is the same -- our ever-important SSH_AUTH_SOCK is defined, and `ssh-agent` is running and ready for use. And because SSH_AUTH_SOCK is recorded in ~/.ssh-agent, our own shell scripts and cron jobs can easily connect with `ssh-agent` just by sourcing the ~/.ssh-agent file. `keychain` itself also takes advantage of this file; you'll remember that when `keychain` starts up, it checks to see whether an existing `ssh-agent` is running. If so, it uses the ~/.ssh-agent file to acquire the proper SSH_AUTH_SOCK setting, thus allowing it to use the existing agent rather than starting a new one. `keychain` will start a new `ssh-agent` process only if the ~/.ssh-agent file is stale (points to a non-existent `ssh-agent`) or if ~/.ssh-agent itself does not exist.

Installing keychain
Installing `keychain` is easy. First, head over to the keychain project page and download the most recent available version of the `keychain` source archive. Then, install as follows:

```
# tar xzvf keychain-1.0.tar.gz
# cd keychain-1.0
# install -m0755 keychain /usr/bin
```

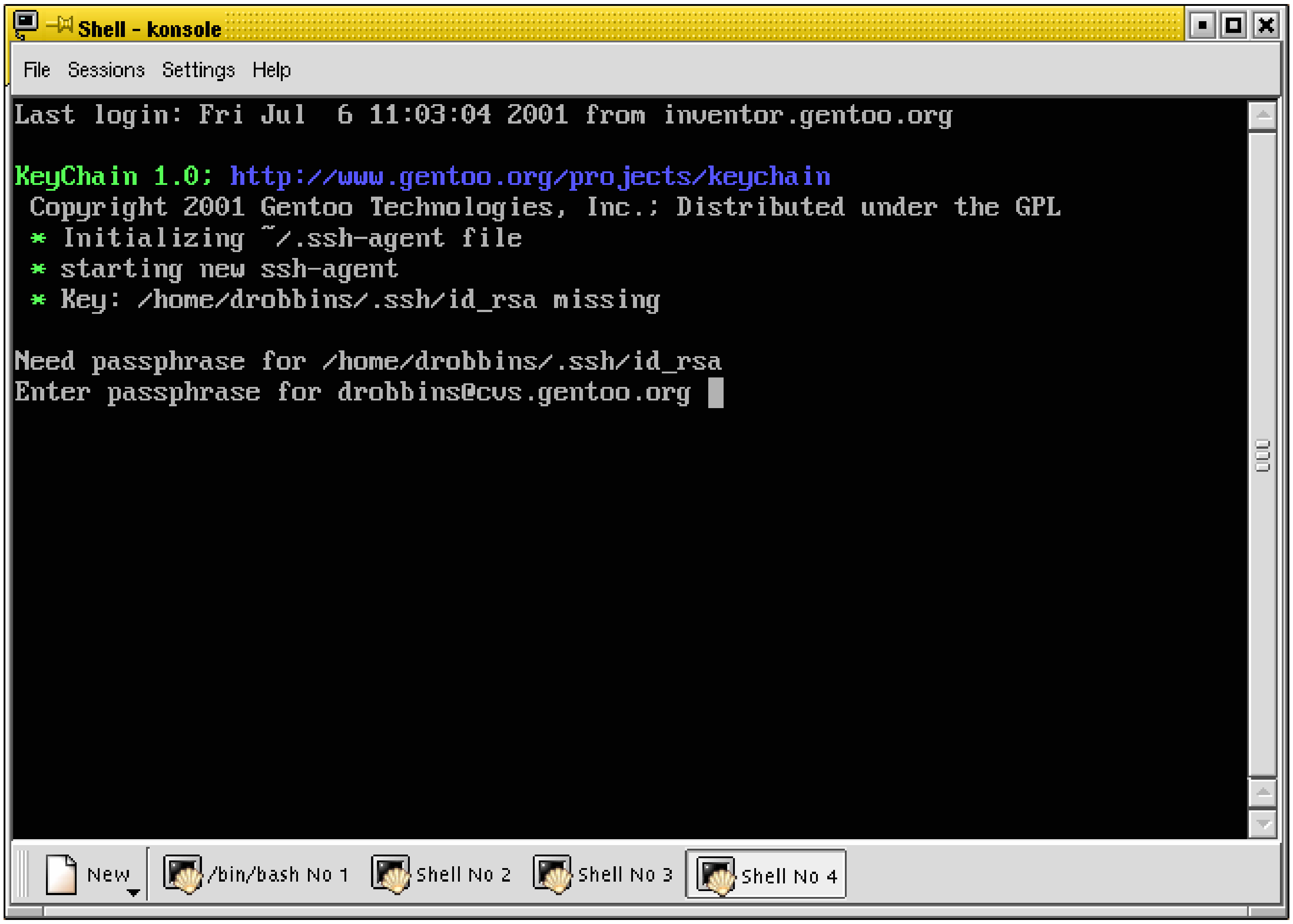
Now that `keychain` is in /usr/bin/, add it to your ~/.bash_profile, supplying paths to your private keys as arguments. Here's a good standard `keychain`-enabled ~/.bash_profile:

An example keychain-enabled ~/.bash_profile

```
#!/bin/bash
#on this next line, we start keychain and point it to the private keys that
#we'd like it to cache
/usr/bin/keychain ~/.ssh/id_rsa ~/.ssh/id_dsa
source ~/.ssh-agent > /dev/null
#sourcing ~/.bashrc is a good thing
source ~/.bashrc
```

Keychain in action
Once you've configured your ~/.bash_profile to call `keychain` at every login, log out and log back in. When you do, `keychain` will start `ssh-agent`, record the agent's environment variable settings in ~/.ssh-agent, and then prompt you for passphrases for any private keys specified on the `keychain` command-line in ~/.bash_profile:

Keychain starts for the first time



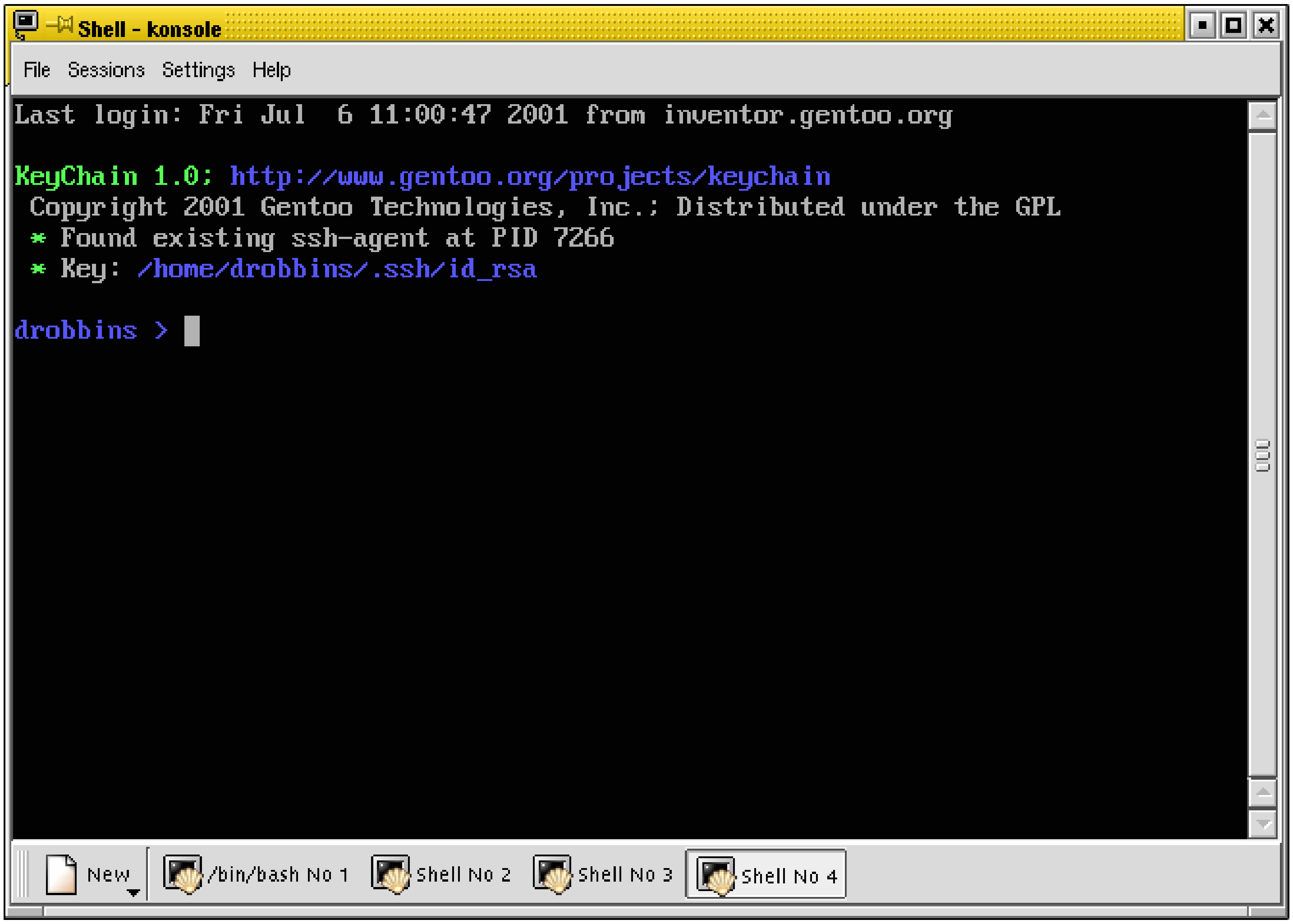
```
Last login: Fri Jul  6 11:03:04 2001 from inventor.gentoo.org

KeyChain 1.0; http://www.gentoo.org/projects/keychain
 Copyright 2001 Gentoo Technologies, Inc.; Distributed under the GPL
 * Initializing ~/.ssh-agent file
 * starting new ssh-agent
 * Key: /home/drobbins/.ssh/id_rsa missing

Need passphrase for /home/drobbins/.ssh/id_rsa
Enter passphrase for drobbins@cvs.gentoo.org
```

Once you enter your passphrases, you private keys will be cached, and `keychain` will exit. Then, ~/.ssh-agent will be sourced, initializing your login session for use with `ssh-agent`. Now, if you log out and log back in again, you'll find that `keychain` will find the existing `ssh-agent` process; it didn't terminate when you logged out. In addition, `keychain` will verify that the private key(s) you specified are already in `ssh-agent`'s cache. If not, then you'll be prompted for the appropriate passphrases, but if all goes well, your existing `ssh-agent` will still contain the private key that you previously added; this means that you're not prompted for a password:

Keychain finds an existing ssh-agent



```
Last login: Fri Jul  6 11:00:47 2001 from inventor.gentoo.org

KeyChain 1.0; http://www.gentoo.org/projects/keychain
 Copyright 2001 Gentoo Technologies, Inc.; Distributed under the GPL
 * Found existing ssh-agent at PID 7266
 * Key: /home/drobbins/.ssh/id_rsa

drobbins >
```

Congratulations; you've just logged in and should be able to `ssh` and `scp` to remote systems; you didn't need to use `ssh-add` right after login, and `ssh` and `scp` won't prompt you for a passphrase either. In fact, as long as your initial `ssh-agent` process keeps running, you'll be able to log in and establish `ssh` connections without supplying a password. And it's very likely that your `ssh-agent` process will continue to run until the machine is rebooted; since you're most likely setting this up on a Linux system, it's possible that you may not need to enter your passphrase for several months! Welcome to the world of secure, passwordless connections using RSA and DSA authentication.

Go ahead and create several new login sessions, and you'll see that `keychain` will "hook in" to the exact same `ssh-agent` process each time. Don't forget that you can also get your cron jobs and scripts to "hook in" to the running `ssh-agent` process. To use `ssh` or `scp` commands from your shell scripts and cron jobs, just make sure that they source your ~/.ssh-agent file first:

```
source ~/.ssh-agent
```

Then, any following `ssh` or `scp` commands will be able to find the currently-running `ssh-agent` and establish secure passwordless connections just like you can from the shell.

Keychain options

After you have `keychain` up and running, be sure to type `keychain --help` to familiarize yourself with all of `keychain`'s command-line options. We're going to take a look at one in particular: the `--clear` option.

You'll recall that in Part 1, I explained that using unencrypted private keys is a dangerous practice, because it allows someone to steal your private key and use it to log in to your remote accounts from any other system without supplying a password. Well, while `keychain` isn't vulnerable to this kind of abuse (as long as you use encrypted private keys, that is), there is a potentially exploitable weakness directly related to the fact that `keychain` makes it so easy to "hook in" to a long-running `ssh-agent` process. What would happen, I thought, if some intruder were somehow able to figure out my password or passphrase and log into my local system? If they were somehow able to log in under my username, `keychain` would grant them instant access to my decrypted private keys, making it a no-brainer for them to access my other accounts.

Now, before I continue, let's put this security threat in perspective. If some malicious user were somehow able to log in as me, `keychain` would indeed allow them to access my remote accounts. Yet, even so, it would be very difficult for the intruder to steal my decrypted private keys since they are still encrypted on disk. Also, gaining access to my private keys would require a user to actually *log in* as me, not just read files in my directory. So, abusing `ssh-agent` would be a much more difficult task than simply stealing an unencrypted private key, which only requires that an intruder somehow gain access to my files in ~/.ssh, whether logged in as me or not. Nevertheless, if an intruder were successfully able to log in as me, they could do quite a bit of additional damage by using my decrypted private keys. So, if you happen to be using `keychain` on a server that you don't log into very often or don't actively monitor for security breaches, then consider using the `--clear` option to provide an additional layer of security.

The `--clear` option allows you to tell `keychain` to assume that every new login to your account should be considered a potential security breach until proven otherwise. When you start `keychain` with the `--clear` option, `keychain` immediately flushes all your private keys from `ssh-agent`'s cache when you log in, before performing its normal duties. Thus, if you're an intruder, `keychain` will prompt you for passphrases rather than giving you access to your existing set of cached keys. However, even though this enhances security, it does make things a bit more inconvenient and very similar to running `ssh-agent` all by itself, without `keychain`. Here, as is often the case, one can opt for greater security or greater convenience, but not both.

Despite this, using `keychain` with `--clear` still has advantages over using `ssh-agent` all by itself; remember, when you use `keychain --clear`, your cron jobs and scripts will still be able to establish passwordless connections; this is because your private keys are flushed at *login*, not *logout*. Since a logout from the system does not constitute a potential security breach, there's no reason for `keychain` to respond by flushing `ssh-agent`'s keys. Thus, the `--clear` option an ideal choice for infrequently accessed servers that need to perform occasional secure copying tasks, such as backup servers, firewalls, and routers.

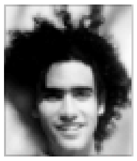
We're done!

Now that the OpenSSH key management series is complete, you should be very familiar with RSA and DSA keys and know how to use them in a convenient yet secure way. Be sure to also check out the following resources:

Resources

- Read Part 1 of Daniel's series on OpenSSH key management on *developerWorks*.
- Visit the home of OpenSSH development.
- Get the most recent version of `keychain`.
- Find 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" book helpful. The Authors' site contains information about the book, a FAQ, news, and updates.
- 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.

e-mail it!

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