Common threads: OpenSSH key management, Part 3

e-mail it!

Agent forwarding and keychain improvements

Daniel Robbins (drobbins@gentoo.org)
President/CEO, Gentoo Technologies, Inc.
February 2002

In this third article in a series, Daniel Robbins shows you how to take advantage of OpenSSH agent connection forwarding to enhance security. He also shares recent improvements to the keychain shell script.

Related content:
OpenSSH key management, Part 1
OpenSSH key management, Part 2
Subscribe to the developerWorks newsletter

Also in the Linux zone:
Tutorials
Tools and products
Code and components
Articles

Also in the Open source projects zone:
Tutorials
Projects and patches
Code and components
Articles

Many of us 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 on 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. For more background, see the previous installments of this series on OpenSSH key management, which cover RSA/DSA authentication (Part 1) and ssh-agent and keychain (Part 2), respectively.

Since Part 2 was published on *developerWorks* in September 2001, and later referenced on Slashdot and Freshmeat (see Resources later in this article for links to these sites), a lot of people have started using `keychain`, and it's undergone a lot of changes. I've received approximately 20 or so high-quality patches from developers around the world. I've incorporated many of these patches into the `keychain` source, which is now at version 1.8 (see Resources). I send my sincere thanks to all those who submitted patches, bug reports, feature requests, and notes of appreciation.

Tightening ssh security

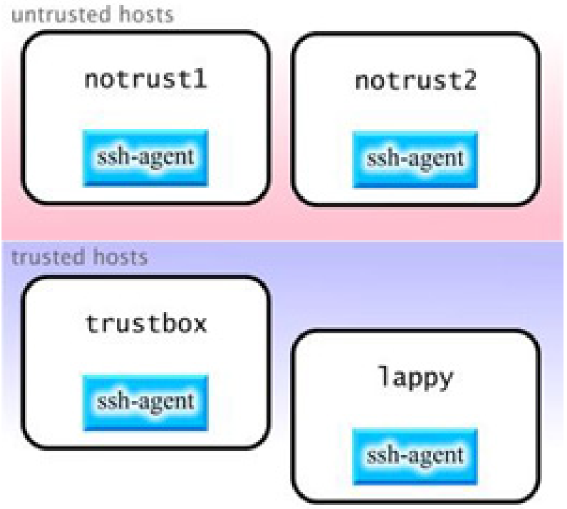
In my last article, I've spent some time discussing the security benefits and tradeoffs of running `ssh-agent`. A few days after the second article appeared on *developerWorks*, I received an e-mail from Charles Karney of Sarnoff Corporation, who politely informed me of OpenSSH's new authentication agent forwarding abilities, which we'll take a look at in a bit. In addition, Charles emphasized that running `ssh-agent` on *untrusted* machines is quite dangerous: if someone manages to get root access on the system, then your decrypted keys *can* be extracted from `ssh-agent`. Even though extracting the keys would be somewhat difficult, it is within the skill of professional crackers. And the mere fact that private key theft is *possible* means that we should take steps to guard against it happening in the first place.

To formulate a strategy to protect our private keys, we must first put the machines we access into one of two categories. If a particular host is well-secured or isolated -- making successful root exploit against it quite unlikely -- then that machine should be considered a *trusted host*. If, however, a machine is used by many other people or you have some doubts about the security of the system, then the machine should be considered an *untrusted host*. To guard your private keys against extraction, `ssh-agent` (and thus `keychain`) should never be run on an untrusted host. That way, even if the system's security is compromised, there will be no `ssh-agent` around for the intruder to extract keys from in the first place.

However, this creates a problem. If you can't run `ssh-agent` on untrusted hosts, then how do you establish secure, passwordless `ssh` connections from these systems? The answer is to only use `ssh-agent` and `keychain` on *trusted* hosts, and to use OpenSSH's new *authentication forwarding abilities* to extend passwordless authentication to any untrusted hosts. In a nutshell, authentication forwarding works by allowing remote `ssh` sessions to contact an `ssh-agent` running on a trusted system.

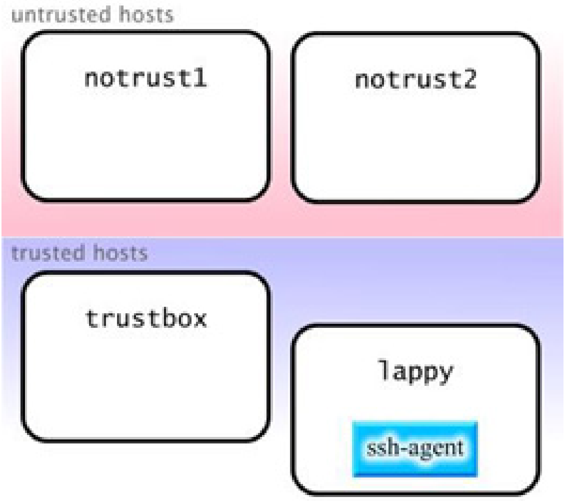
Authentication agent forwarding
To get an idea of how authentication forwarding works, let's first take a look at a hypothetical situation where user `drobbins` has a trusted laptop called `lappy`, a trusted server called `trustbox`, and two other untrusted systems that he must access, called `notrust1` and `notrust2`, respectively. Currently, he uses `ssh-agent` along with `keychain` on all four machines, as follows:

Figure 1. ssh-agent running on trusted and untrusted machines



The problem with this approach is that if someone gains root access on `notrust1` or `notrust2`, then it is of course possible for this person to extract keys from the now vulnerable `ssh-agent` process. To fix this, `drobbins` stops running `ssh-agent` and `keychain` on untrusted hosts `notrust1` and `notrust2`. In fact, to be even more careful, `drobbins` decides to only use `ssh-agent` and `keychain` on `lappy`. This limits exposure of his decrypted private keys, protecting him against private key theft:

Figure 2. ssh-agent running only on lappy; a more secure configuration



Of course, the problem with this approach is that `drobbins` can now only establish passwordless connections from `lappy`.

Let's see how to enable authentication forwarding and get around this problem.

Assuming that all machines are running recent versions of OpenSSH, we can get around this problem by using authentication forwarding. Authentication forwarding allows remote `ssh` processes to contact the `ssh-agent` that is running on your local trusted machine -- rather than requiring a version of `ssh-agent` to be running on the same machine that you are `ssh`ing out from. This usually allows you to run `ssh-agent` (and `keychain`) on a single machine, and means that all `ssh` connections that originate (either directly or indirectly) from this machine will use your local `ssh-agent`.

To enable authentication forwarding, we add the following line to `lappy` and `trustbox`'s */etc/ssh/ssh_config*. Note that this is the config file for `ssh` (*ssh_config*), not the ssh daemon `sshd` (*sshd_config*):

Listing 1. Add this line to your /etc/ssh/ssh_config

```
ForwardAgent Yes
```

Now, to take advantage of authentication forwarding, `drobbins` can connect from `lappy` to `trustbox`, and then from `trustbox` to `notrust1` without supplying passphrases for any of the connections. Both `ssh` processes "tap in" to the `ssh-agent` running on `lappy`:

Listing 2. Tapping lappy

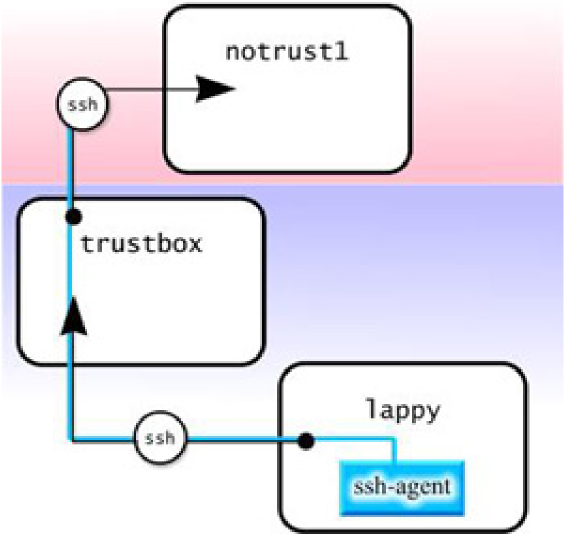
```
$ ssh drobbins@trustbox
Last login: Wed Sep 26 13:42:08 2001 from lappy

Welcome to trustbox!
$ ssh drobbins@notrust1
Last login: Tue Sep 25 12:03:40 2001 from trustbox

Welcome to notrust1!
$
```

If you try a similar configuration and find that agent forwarding isn't working, try using `ssh -A` instead of plain old `ssh` to explicitly enable authentication forwarding. Here's a diagram of what went on behind the scenes when we logged in to `trustbox` and `notrust1` using authentication forwarding, above:

Figure 3. Agent forwarding in action



As you can see, when `ssh` connected to `trustbox`, it maintained a connection to the `ssh-agent` running on `lappy`. When an `ssh` connection was made from `trustbox` to `notrust1`, this new `ssh` process maintained the authentication connection to the previous `ssh`, effectively extending the chain. Whether this authentication chain can be extended beyond `notrust1` to other hosts depends on how `notrust1`'s /etc/ssh/ssh_config is configured. As long as agent forwarding is enabled, all parts of the chain will be able to authenticate using the `ssh-agent` running on the trusted `lappy`.

Advantages of agent connection forwarding

Authentication forwarding offers a number of security advantages not touched on here. To convince me of the importance of agent connection forwarding, Charles Karney shared with me these three security advantages:

1. The private key is stored only on the trusted machine. This prevents malicious users from grabbing your encrypted key from disk and attempting to crack the encryption.
2. `ssh-agent` runs only on the trusted machine. This prevents an intruder from doing a memory dump of a remote `ssh-agent` process and then extracting your decrypted private keys from the dump.
3. Since you only need to type in the passphrase on your trusted machine, you prevent any keystroke loggers from stealthily grabbing your passphrase as it is entered.

The one drawback to relying on authentication agent connection forwarding is that it doesn't solve the problem of allowing cron jobs to take advantage of RSA/DSA authentication. One solution to this problem is to set up all cron jobs that need RSA/DSA authentication so that they execute from a trusted machine on your LAN. If necessary, these cron jobs can use `ssh` to connect to remote systems to automate backups, synchronize files, and so on.

Now that we've looked at authentication agent connection forwarding, let's turn to recent improvements made to the `keychain` script itself.

Keychain functionality improvements

Thanks to user patch submissions, many significant improvements have been made to the `keychain` source. Several of the user-submitted `keychain` patches were functionality-related. For example, you'll recall that `keychain` created an ~/.ssh-agent file; the name of this file has now been changed to ~/.ssh-agent-[hostname] so that `keychain` works with NFS-mounted home directories that may be accessed from several different physical hosts. In addition to the ~/.ssh-agent-[hostname] file, there is now a ~/.ssh-agent-csh-[hostname] file that can be sourced by `csh`-compatible shells. Finally, a new `--nocolor` option has been added so that colorization features can be disabled if you happen to be using a non-vt100-compatible terminal.

Shell compatibility fixes

While the functionality improvements have been significant, the vast majority of fixes have dealt with *shell compatibility* issues. You see, while keychain 1.0 required `bash`, later versions were changed to work with any `sh`-compatible shell. This change allows `keychain` to work "out of the box" on nearly any UNIX system, including Linux, BSD, Solaris, IRIX, and AIX as well as other UNIX platforms. While the transition to `sh` and general UNIX compatibility has been a bumpy ride, it has also been a tremendous learning experience. Creating a single script that runs on all of these platforms has been very tricky indeed, mainly because I simply don't have access to most of these operating systems! Thankfully, `keychain` users from around the globe do, and many have provided great assistance in identifying compatibility problems and submitting patches to fix them.

There are really two kinds of compatibility problems that had to be fixed. First, I needed to make sure that `keychain` only used built-ins, expressions, and operators that were fully supported under all `sh` implementations, including all the popular free and commercial UNIX `sh` shells, `zsh` (in `sh`-compatibility mode), and `bash` versions 1 and 2. Here are some of the user-submitted shell-compatibility fixes that were applied to the `keychain` source:

Since older `sh` shells don't support the ~ convention to refer to a user's home directory, lines that used ~ were changed to use `$HOME` instead:

Listing 3. Making it $HOME

```
hostname=`uname -n`
pidf=${HOME}/.ssh-agent-${hostname}
cshpidf=${HOME}/.ssh-agent-csh-${hostname}
```

Next, all references to `source` were changed to `.` to ensure compatibility with purist NetBSD's `/bin/sh`, which doesn't support the `source` command at all:

Listing 4. Humoring NetBSD

```
if [ -f $pidf ]
then
    . $pidf
else
SSH_AGENT_PID="NULL"
fi
```

Along the way, I also applied some nice performance-related fixes. One savvy shell scripter informed me that instead of "touching" a file by typing `touch foo`, you can do this instead:

Listing 5. Touching files

```
> foo
```

By relying on built-in shell syntax rather than using an external binary, a `fork()` is avoided and the script becomes slightly more efficient. The `> foo` should work with any `sh`-compatible shell; however, it does not appear to be supported by `ash`. This shouldn't be a problem for most people, since `ash` is more of an emergency-disk type shell rather than something people use on a day-to-day basis.

Platform executable issues

Getting a script working under multiple UNIX operating systems requires more than adhering to pure `sh` syntax. Remember, most scripts also call external commands, such as `grep`, `awk`, `ps`, and others, and these commands must be called in a standards-compliant way as much as possible. For example, while the `echo` included with most versions of UNIX recognizes the `-e` option, Solaris does not -- it simply prints a `-e` to stdout when it is used. So in order to deal with Solaris, `keychain` now auto-detects whether `echo -e` works:

Listing 6. Sniffing out Solaris

```
if [ -z "`echo -e`" ]
then
    E="-e"
fi
```

Above, `E` is set to `-e` if `-e` escaping is supported. Then, echo can be called as follows:

Listing 7. Better echo

```
echo $E Usage: ${CYAN}${0}${OFF} [ ${GREEN}options${OFF} ] ${CYAN}sshkey${OFF} ...
```

By using `echo $E` instead of `echo -e`, the `-e` option can be dynamically enabled or disabled as needed.

pidof, ps

Probably the most significant compatibility fix involved changing how `keychain` detects currently running `ssh-agent` processes. Previously, I was using the `pidof` command to do this, but it had to be removed since several systems don't have a `pidof`. Really, `pidof` isn't the greatest solution anyway since it lists *all* `ssh-agent` processes running on the system, regardless of user, when we're really interested in all `ssh-agent` processes owned by the current effective UID.

So, instead of relying on `pidof`, we switched over to piping `ps` output to `grep` and `awk` in order to extract the needed process ids. This was a user-submitted fix:

Listing 8. Piping better than pidof

```
mypids=`ps uxw | grep ssh-agent | grep -v grep | awk '{print $2}'`
```

The above pipeline will set the `mypids` variable to the values of all `ssh-agent` processes owned by the current user. The `grep -v grep` command is part of the pipeline to ensure that the `grep ssh-agent` process does not become part of our

PID list.

While this approach is good in concept, using `ps` opened up a whole new can of worms since `ps` options are not standardized across various BSD and System V UNIX derivatives. Here's an example: while `ps uxw` works under Linux, it does not work under IRIX. And while `ps -u username -f` works under Linux, IRIX, and Solaris, it doesn't work under BSD, which only understands BSD-style `ps` options. To get around this problem, `keychain` auto-detects whether the current system's `ps` works with BSD or System V syntax before executing the `ps` pipeline:

Listing 9. Detecting BSD vs. System V

```
psopts="FAIL"
ps uxw >/dev/null 2>&1
if [ $? -eq 0 ]
then
psopts="uxw"
else
ps -u `whoami` -f >/dev/null 2>&1
if [ $? -eq 0 ]
then
psopts="-u `whoami` -f"
fi
fi
if [ "$psopts" = "FAIL" ]
then
echo $0: unable to use \"ps\" to scan for ssh-agent processes.
Report KeyChain version and echo system configuration to drobbins@gentoo.org.
exit 1
fi

mypids=`ps $psopts 2>/dev/null | grep "[s]sh-agent" | awk '{print $2}'` > /dev/null
2>&1
```

To ensure that we work with both System V and BSD-style `ps` commands, the script does a "dry run" of `ps uxw`, throwing away any output. If the error code from this command is zero, we know that `ps uxw` works and we set the `psopts` value appropriately. However, if `ps uxw` returned a non-zero error code (indicating that we need to use BSD-style options), we do a dry-run of `ps -u `whoami` -f`, again throwing away all output. At this point, hopefully we've found either a BSD or System V variant of `ps` that we can use. If we haven't, we print out an error and exit. But it is very likely that one of the two `ps` commands worked, in which case we execute the final line in the above code snippet, our `ps` pipeline. By using the `$psopts` variable expansion immediately after `ps`, we're able to pass the correct options to the `ps` command.

The `ps` pipeline also contains a true `grep` gem, which was kindly sent to me by Hans Peter Verne. Notice that `grep -v grep` is no longer part of the pipeline; instead, it has been removed and `grep "ssh-agent"` has been changed to `grep "[s]sh-agent"`. This single `grep` command ends up doing the same thing as `grep ssh-agent | grep -v grep`; can you figure out why?

Listing 10. Neat grep trick

```
mypids=`ps $psopts 2>/dev/null | grep "[s]sh-agent" | awk '{print $2}'` > /dev/null
2>&1
```

Stumped? If you've decided that a `grep "ssh-agent"` and `grep "[s]sh-agent"` should match the exact same lines of text, you are correct. So why do they generate different results when the output of `ps` is piped to them? Here's how it works: when you use `grep "[s]sh-agent"`, you change how the `grep` command appears in the `ps` process list. By doing so, you prevent `grep` from matching itself, since the `[s]sh-agent` string doesn't match the `[s]sh-agent` regular expression. Isn't that brilliant? If you still don't get it, play around with `grep` a bit more and you'll get it soon enough.

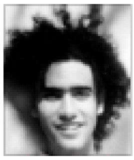
Conclusion

This column concludes my coverage of OpenSSH. Hopefully, you've learned enough about it to start using OpenSSH to secure your systems. Next month's *Common threads* column will continue with the "Advanced filesystem implementor's guide" series.

Resources

- "Common threads: OpenSSH key management, Part 1" (*developerWorks*, July 2001) covers RSA/DSA authentication.
- "Common threads: OpenSSH key management, Part 2" (*developerWorks*, September 2001) introduces ssh-agent and keychain.
- The most recent version of `keychain` is available on the Gentoo Linux Keychain page.
- Be sure to visit the home of OpenSSH development, and check out the OpenSSH FAQ.
- You can download the latest OpenSSH source tarballs and RPMs from Openbsd.org.
- PuTTY is an excellent ssh client for Windows machines.
- The book "SSH, The Secure Shell: The Definitive Guide" (O'Reilly & Associates, 2001) may be of assistance. The authors' site contains information about the book, a FAQ, news, and updates.
- Visit Slashdot for "news for nerds and other stuff that matters".
- Check out Freshmeat, which lists new releases of open source packages as they happen.
- 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 of 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 to 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 their 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