Hostbased SSH
A Better Alternative

ABE SINGER

Abe Singer is the chief security officer for the Laser Interferometer Gravitational Wave Observatory at the California Institute of Technology. He has been a programmer, system administrator, security geek, occasional consultant, and expert witness. His areas of interest are in security that actually works.
abe@ligo.caltech.edu

Almost all SSH users are familiar with two modes of authentication over SSH: passwords and SSH keys. SSH supports another method that seems to be less well known: hostbased, which allows for users to ssh securely between cooperating hosts without providing a credential. It's called hostbased because the client (source) host authenticates itself to the remote host, and the remote host then trusts the client to identify the user. The term "hostbased" is often employed to describe use of hostname or IP-address access control lists. That's not what I'm talking about, so please keep reading.

Sounds scary? Hostbased SSH can be at least as secure as SSH, or more so, and can be simpler to manage.

Hostbased SSH isn't the answer to everything, but I think it's the right answer in particular common scenarios. In the first scenario, you manage a network of computers where each user has the same account, with the same credential, across multiple machines. Once users have authenticated to one of the hosts, there's no added value in requiring them to authenticate again to other hosts on the network. The users may also want to run unattended jobs that execute commands between hosts. Clusters are a particular variant, where users log in to a head node to, in turn, access compute nodes.

In the second common scenario, you want to automate root access to multiple machines in order to control who has remote access to the root account, and from where, and to minimize having to type the root password on the remote host. The latter is especially a good idea if you are investigating a host that might be compromised. As a bonus, you might want to give a user root access to particular hosts without having to divulge the root password.

Hostbased authentication solves these problems because it doesn't require the user to have, know, or enter any credentials.

How does that work? First, a little bit of history.

The R-Commands

Some of you may remember the Good Old Days™ when we had the "r-commands": rsh, rlogin, and rcp. Rlogin worked like ssh: You would "rlogin" to an account at a remote host and, by default, be prompted for the password for that account. If you didn't want to have to type a password every time, you could create a file in your home directory on the remote host called ".rhosts" and, in the file, put a line with the local host and local username.

For example, if alice@foo.example.com wanted to log in to aliceb@bar.example.com, she'd create the file ~aliceb/.rhosts on bar, that looked like this:

```
foo.example.com alice
```

Then, when she was on foo and typed "rlogin aliceb@bar.example.com," the rlogin server on bar would read the entry in the rhosts file and log Alice into the account aliceb on bar, without prompting for a password.

rlogin used a pretty weak security model. Unlike ssh, rlogin provided no encryption, which was a serious problem. The whole protocol was done over cleartext, with no integrity checking, relied on privileged port numbers for validation, and was vulnerable to network address spoofing.

Hostbased SSH

Hostbased SSH implements rlogin style access control without the insecurity of rlogin. The SSH server uses ".shosts" files (or a global "shosts.equiv" file) that use the same format as the old .rhosts file. The SSH client host authenticates itself to the remote SSH server. It then asserts to the server its own hostname and the username originating the session, which the server then checks against the shosts.equiv file or the .shosts file for that user.

So, what prevents a rogue user from asserting any hostname and/or username? This is the awesome part: The client signs the assertion with its host key. In detail, it works like this:

- Alice, on host "client," runs "ssh alice@server."
- The client makes an ssh connection to the server, negotiates hostbased login and a session ID for alice@server.
- If hostbased negotiation was successful, the client creates an assertion consisting of the session ID, Alice's username, and the client's hostname, signs the assertion with its host key, and sends it to the server.
- The server checks that the signature on the assertion matches the client's public key in /etc/ssh/ssh_known_hosts (or ~alice /.ssh/known_hosts if enabled), and that alice@client matches an entry in ~alice/.shosts or /etc/ssh/shosts.equiv.
- If all the checks pass, the server proceeds with the login [1].

The hostbased dance works because the client's host SSH private key is (supposed to be) only readable by root, which is why the remote host trusts the key for authentication—the fact that the assertion was signed with the host key is proof that the assertion was signed by root, not the user.

ssh doesn't normally (and shouldn't) run as root. Rather, it uses the helper application /usr/lib/openssh/ssh-keysign, which runs setuid root. ssh-keysign gets invoked automagically by ssh, reads the host SSH key, and does the required signing.

This is an elegant little design. The user cannot subvert the contents of the assertion, and only the code that handles the signing has to run as root, minimizing the potentially exploitable setuid codebase.

Of course, you still have to have user accounts on the remote machine. But, because the user doesn't need to have a password, you have the option of creating accounts with no usable password, so that the only way the user can log in is by using hostbased authentication from hosts that you authorize. Definitely useful in a cluster scenario.

Hostbased SSH uses the same technology as SSH keys, so cryptographically speaking, hostbased SSH authentication is just as strong as SSH key authentication.

Obviously, you can only do hostbased SSH with a client that has a host key. Normally, that would be a host running an SSH server. Hostbased wasn't really designed to be run from a client-only setup such as a laptop, although you could in theory just generate a host key manually without running the ssh daemon. I'll leave how to do that as an exercise for the reader.

Howto

Here's how to make hostbased SSH work on the client and on the server.

On the Client

You need to have the following entries in /etc/ssh/ssh_config:

```
EnableSSHKeysign yes
HostbasedAuthentication yes
```

/usr/lib/ssh-keysign has to be setuid root (which it is by default) so that it can read the host key.

On the Server

You need the following in /etc/ssh/sshd_config:

```
HostbasedAuthentication yes
IgnoreRhosts no
```

Put the client host's SSH keys in /etc/ssh/ssh_known_hosts. The ssh_known_hosts file functions for hostbased SSH similar to how the user's authorized_keys file functions for SSH key authentication: The server will only accept hostbased authentication from clients whose host public keys are in the file. You can get public keys from clients using ssh-keyscan:

```
ssh-keyscan –t dsa client.example.com
```

Verify that the results are indeed the public key of the client by using key fingerprints:

```
ssh-keygen –l –f <public key file>
```

Remember to tell the ssh daemon to reload the configuration:

```
service ssh reload
```

Of course, you can create a script or use a configuration management tool to push the configuration and ssh_known_hosts file to several machines at once.

Note that adding keys to ssh_known_hosts does not require restarting sshd.

Now you just have to configure which users get access.

Controlling User Access

Hostbased SSH provides two flavors of controlling which users can log into accounts: ~/.shosts or /etc/ssh/shosts.equiv.

shosts.equiv allows users on clients to log in to accounts on the server with identical usernames. In other words, alice@client can log in to alice@server, but not to aliceb@server. You can enable access on a per-user basis or allow all users on the client to log in to the corresponding account on the server. The file should be editable only by root, so regular users cannot make any changes to it.

The syntax for entries in the shosts.equiv file is:

```
<remote host> [-][<remote username>]
```

sshd reads the file from the beginning and stops at the first matching line. A hostname without a username allows all users (except root) from the client to log in to the matching username account on the server. A hostname and a username allows a specific user to log in. A "-" in front of a username excludes that user—useful only if followed by a line with just a hostname—to allow all other users.

Note that you cannot enable root access with shosts.equiv. The hostname-only format excludes root, and the server will ignore explicit entries for root.

If you want to allow root login, or let users log in to accounts that have different usernames, you have to use the ~/.shosts file. The syntax of the file is identical to shosts.equiv, and multiple entries are allowed. Specifying a hostname works identically to shosts.equiv, but specifying a username allows a non-matching username login to the account.

Thus, if on the server, ~alice/.shosts contains:

```
ws1.example.com alice
ws1.example.com bob
ws2.example.com alice
```

then alice@ws1, alice@ws2, and bob@ws1 can ssh to alice@server.

~/.shosts gives you more flexibility, but shosts.equiv file gives you more control over who gets authorized, at the expense of your having to maintain it.

Yeah, You Could Use SSH Keys

Everything I've talked about could be implemented using SSH keys, but with worse failure modes. For starters, the default access mode for SSH keys is to allow access from anywhere; any restrictions applied are vulnerable to spoofing, and there is no way to say "only bob@foo can log in to carol@bar."

An unfortunately common solution to passwordless login is the use of SSH keys with no passphrase on the secret key ("passwordless keys"), which is effectively storing an unencrypted password in a text file (which users also often do), a basic security no-no.

And many scenarios require putting the user's SSH key on every host, which defeats the design of SSH keys, where the key only lives on the client.

ssh-agent makes things a little better, but it has to be manually restarted when a host reboots—a pain when you had it running on a thousand nodes that just rebooted due to a kernel patch.

Auditing access and de-authorization are more difficult with SSH keys.

Key management is always difficult. SSH key solutions require more key management than hostbased. In fact, SSH key management is difficult enough that ssh.com even sells a product focusing on just that.

Hostbased SSH is just simpler.

What, Me Worry?

Usually at some point when I'm explaining hostbased SSH, someone says "But what if the client machine is rooted? Anyone with root could log in to any user's remote account!" Yes, that is true. But then any scheme—be it hostbased, password, or SSH key—fails if the client is rooted. You have to trust root on the remote host in all cases. Hostbased SSH is just a bit easier to manage.

Let's go over the risks of using hostbased, and compare to SSH keys.

Hostbased SSH uses the same technology as SSH keys; so, cryptographically speaking, hostbased SSH authentication is just as strong as SSH key authentication.

With SSH keys and hostbased, you have to trust root (users) on the remote host and trust that the root hasn't been compromised. With hostbased, you don't have to trust that user keys are being managed properly and whether they have strong passphrases.

Hostbased does true authentication of the client, whereas SSH keys can only validate the client's hostname or IP address, which can be spoofed.

In short, hostbased SSH has fewer trust requirements than SSH keys and is harder for the users to circumvent.

The user's .shosts file can, of course, be modified by the user to allow access to other users. That can be good or bad, depending on your policies. However, with hostbased the user can only do so for remote hosts that you have explicitly authorized to do hostbased login by adding their host keys to /etc/ssh/ssh_known_hosts. With SSH keys, the user could allow someone access from any remote host.

If a user is going to allow other users access to their account, I'd like to have a log showing which other user logged in. And I really prefer an access mode that doesn't require users to share passwords.

Additional Tricks

Here are some other options that you can use with hostbased auth.

You can get the hosts to log which user logged in using hostbased access by setting the log level to "VERBOSE" in /etc/ssh/sshd_config:

```
LogLevel VERBOSE
```

You get a log message that looks like this:

```
sshd[8180]: Accepted DSA public key 7c:3d:bc:84:c5:87:71:06:93:
56:ff:d6:8c:c4:ae:66 from alice@client.example.com
```

You can let users configure hostbased SSH access to just their account from some external host by putting that remote host's public key in their ~/.ssh/known_hosts file, if you include the following directive in sshd_config:

```
IgnoreUserKnownHosts no
```

You can let clients assert their hostname with just their host key. This can be used to allow hostbased SSH from a roaming laptop whose IP address changes:

```
HostbasedUsesNameFromPacketOnly yes
```

You can force sshd to use shosts.equiv only:

```
IgnoreRhosts yes
```

You can force the server to only accept hostbased:

```
AuthenticationMethods Hostbased
```

or just do hostbased first:

```
PreferredAuthentications HostBased,PublicKey,Password
```

Restricting what commands a user can run is a little complicated. You can do it by creating a script in /etc/ssh/sshrc. The details are left as an exercise for the reader.

Hostbased SSH can be a bit painful to debug when it doesn't work right. In the interests of space, I'll refer you to [2] and [3] for some debugging help.

Secure Remote Root Access

I want to be able to remotely log in to my hosts as root for a variety of reasons, but I want to be able to do it in a secure manner.

The Problem with Remote Root Login

It used to be a "best practice" that one did not remotely log in to a host as root; rather, one would log in as a regular user and then su to root. In short, the problems are: no accounting of who has had root at a given time; an attacker with the root password gets immediate access; and trojaned SSH clients get login passwords much more easily than a password entered in a shell.

Back in the Telnet days, most distros had root login via Telnet disabled by default; you had to explicitly enable it. Telnet was also unencrypted, an additional problem.

Unfortunately, OpenSSH comes with root login turned on by default. As a result, we have a new generation of sysadmins who blithely ssh as root from anywhere, with password or SSH key or whatever. It still makes me cringe when I see someone do it.

Particularly when doing incident response, the last thing you want to do is type a password on a host that might be compromised. Attackers usually root systems without having the root password (they wouldn't need an exploit if they had it). If the root password on the compromised host is the same as on other hosts, and they allow root login, the attacker might just get root access to all your hosts, without even needing another exploit.

My Remote Root Solution

Clearly, there are a number of situations where you need remote root access. I want to run the same commands across hundreds of hosts automatically. I want to be able to run remote commands non-interactively, and I don't want to type a password on a potentially compromised host. I want to give other users selective root access to hosts without giving them the root password, and easily disable their access. And I want a log of who had access.

Hostbased SSH makes this easy.

Here's my solution:

I have a dedicated bastion host whose sole purpose is to provide root access to other hosts. The root account on the bastion host is authorized for hostbased access to my other hosts (the "target hosts") and is used to manage hostbased access on the target hosts.

Each user of the bastion host has her own account and has to use a type of two-factor authentication to get to that account.

Hostbased configuration on the bastion host is done exactly as described above. But on the target hosts, I add some more restrictions. I want to allow root ssh only from the management host. I do this using the "Match" statement in /etc/ssh/sshd_config:

```
IgnoreRhosts no
PermitRootLogin no
Match Host bastion.example.com
      HostbasedAuthentication yes
      PasswordAuthentication no
      PermitRootLogin without-password
      RhostsRSAAuthentication no
      PubkeyAuthentication no
      GSSAPIAuthentication no
```

The "IgnoreRhosts" line is required so that I can use ~root/.shosts for controlling who gets access. sshd doesn't allow that directive to be inside a match statement, so it has to be global. However, because the .shosts file only works for hosts authorized for hostbased access, the net result is the same.

Inside the match statement, I've disabled all modes of authentication except for hostbased. There's no reason to allow them. Similarly, I've enabled root login using the "without-password" option, which means that root login over SSH cannot be done with a password in any circumstances (a bit belt-and-suspenders since password authentication is separately disabled, but better to be cautious).

Then in ~root/.shosts, I put in an entry for root on the bastion host, and then for each user who gets access:

```
bastion.example.com root
bastion.example.com alice
bastion.example.com bob
bastion.example.com carol
```

Remember, Alice, Bob, and Carol don't get the root password to the bastion host nor to the target hosts; they just have credentials for their account. I can give them each access to only the hosts that they need access to, and I can quickly disable their root access to all hosts by disabling their account on the bastion server.

And Bob's your uncle.

Resources

[1] T. Ylonen, RFC 4252, The Secure Shell Protocol, Section 9 "Host-Based Authentication," January 2006.

[2] Wikibooks OpenSSH/Cookbook/Host-based Authentication: http://en.wikibooks.org/wiki/OpenSSH/Cookbook/Host-based_Authentication.

[3] Daniel J. Barrett and Richard E. Silverman, *SSH: The Secure Shell: The Definitive Guide* (O'Reilly and Associates, February 2001).