Tunneling through a corporate gateway

At one site where I provide technical assistance, access to the network is all done via ssh over the Internet. However, one can't get access to the machines directly; one must first make an ssh connection to a gateway box I'll call martha. martha doesn't allow one to use regular passwords or keys, though. We're all given a one−time password devices the size of a credit card. When we ssh to martha, it asks us for our one−time password. The card displays a random password which we then type into the martha's password prompt. At that point we're into the system.

As the name implies, the one−time password we get from our cards can only be used once. If I tried to use that password again, martha would not let me in. This protects me from someone looking over my shoulder as I log in and trying to use that same password to get in a few minutes later.

The problem comes when I want to do anything useful with this network. Lets say I'm trying to kill a process that's using too much processor time on waldo, our web server. Step 1: in an unused terminal window I ssh to martha. I enter my PIN number on the credit card and type the password it displays to get a shell prompt on martha. Step 2: I then type "ssh waldo" to get a shell prompt on waldo (as I've already installed my ssh key there; read on if you'd like an easy way to install keys). Step 3: Then I type "su −" and enter the root password to become root. Then I can type "killall −TERM runaway_app_name ; sleep 2 ; killall −9 runaway_app_name" to kill it off.

That terminal is certainly available to run other commands now, but if I want to connect to a different protected machine at the same time, I have to do the above all over again, including opening a new connection to martha with the requisite one−time password. This is hardly conducive to managing large numbers of machines at once.

What I'd ideally like is a way to get directly to the target machine without having to log into martha every time. As I respect, and frankly agree with, the one−time password approach, I'm not looking to get around this stronger authentication method but simply wish to avoid having to log into martha each time I want a new ssh connection to some other machine.

Here's the approach we'll take. I'll first log in to martha *once*. Assuming there are 5 machines behind martha to which I may wish to ssh, I'm going to forward 5 local ports to the ssh ports on each of those machines. So, for example, localhost port 22005 will be forwarded to port 22 on 192.168.0.5, localhost port 22006 will be forwarded to port 22 on 192.168.0.6, and so on up to port 22009 being port forwarded to port 22 on 192.168.0.9. The choice of local ports is arbitrary; I'm simply matching the last octet of the address as a way of organizing the ports.

I leave the terminal to martha open for the entire time I'm doing work with the other systems. I can leave some program running on that terminal or I can leave it idle at a command prompt, it doesn't matter.

To ssh to waldo (ip address 192.168.0.6), I instead type "ssh localhost 22006". The ssh connection gets tunneled to martha, who carries the packets over to waldo's ssh port. At that point, my ssh client program can log into waldo.

To open another session to waldo, I can type "ssh localhost 22006" again, without having to open another connection to martha. Similarly, I can open any number of connections to our database server (Darma, ip address 192.168.0.7) by typing "ssh localhost 22007".

This approach allows me to authenticate myself *once* with a one−time password, but make multiple connections to any number of machines behind the gateway box directly. There's some overhead in that all connections to machines behind martha are encrypted twice by the local machine, but this is probably not a problem.

Let's actually set this up and try it out. While my previous port forwarding examples used additional parameters on the "ssh martha" command line, we're going to do all this port forwarding in ~/.ssh/config . The following examples assume you're using Openssh; if you're using a different version of ssh, check your documentation to see if there are any changes in file location or syntax.

In ~/.ssh/config , add the following lines:

```
Host martha
#blinky
        LocalForward    22005   192.168.0.5
#waldo
        LocalForward    22006   192.168.0.6
#darma
        LocalForward    22007   192.168.0.7
#annabelle
        LocalForward    22008   192.168.0.8
#endor
        LocalForward    22009   192.168.0.9
```

This says that when we first open a connection to martha, we want that ssh session to listen on all local ports from 22005 to 22034. Now let's add some sections that make the connections to the protected machines easier:

```
Host blinky
        Hostname        localhost
        Port            22005
        HostKeyAlias    blinky
Host waldo
        Hostname        localhost
        Port            22006
        HostKeyAlias    waldo
Host darma
        Hostname        localhost
        Port            22007
        HostKeyAlias    darma
Host annabelle
        Hostname        localhost
        Port            22008
        HostKeyAlias    annabelle
        User            william
Host endor
        Hostname        localhost
        Port            22009
        HostKeyAlias    endor
Host *
        Protocol 2
        User wstearns
        Compression yes
        IdentityFile /home/wstearns/.ssh/id_dsa
```

Now when I type "ssh waldo", the ssh client program doesn't try to make a connection directly to waldo's IP address. Instead, it looks in ~/.ssh/config and figures out that it needs to connect to localhost, port 22006.

When it connects, it will associate the name waldo, instead of localhost, with the host key it will store in ~/.ssh/known_hosts . In addition, it will pull the following defaults from the "Host *" section: use ssh protocol 2, assume the target user is wstearns, compress the data being sent, and try to get an ssh key from id_dsa if the ssh−agent doesn't have one.

The defaults in the "Host *" section can be overridden, as we've done for annabelle's section as my username on annabelle is "william", not wstearns. We can also override the defaults *and* the Host specific sections with commands on the command line; for example, "ssh root@annabelle" will ignore the "User wstearns" and the "User william" lines and instead try to log in directly as root.

Once we've made our one connection to martha, connecting to waldo or any of the other machines is as simple as typing "ssh waldo". To get a root prompt on waldo, "ssh root@waldo" will suffice. Even the scp (secure copy program) can now work; "scp −p /website/index.html waldo:/var/www/index.html" will send a file right to waldo. Without the above port forwarding trick, I would have had to scp it first to martha, then scp it from martha to waldo.

fanout

What happens when I want to run one command on multiple machines? Perhaps I want to see who's logged into the 5 main servers. To check just one machine, I would have typed:

```
ssh blinky w
```

Instead of giving me a command prompt, the ssh program opens up a connection, logs me in automatically (as I'm using ssh−agent and ssh keys), runs the "w" progam, and then logs out automatically. Quite useful for running a single command line program.

For 5 machines, then:

```
ssh blinky w
ssh waldo w
ssh darma w
ssh annabelle w
ssh endor w
```

Still not too bad. Unfortunately, that's going to get very old very fast when I have to run 30 commands on 100 machines. :−)

Lets get the computer to do the work. I want to give a command to run and a list of machines to run it on and come back to an orderly output of what's going on on those machines. fanout is the tool for the task.

```
fanout −−noping 'blinky waldo darma annabelle endor' 'w' | less
```

We specify −−noping as the first parameter because fanout usually wants to ping the target machines first to see if they're alive; as we're playing port forwarding tricks, the machines aren't really reachable by ping, so we tell fanout to connect without pinging first. The next parameter is a list of machines to which to connect. The last parameter is the command to run on all. Here's what we get back:

```
Starting blinky
Starting waldo
Starting darma
Starting annabelle
```

```
Starting endor
```

, then a brief pause as fanout runs the command on each target machine, then:

```
Fanout executing "w"
Start time Tue Aug 20 00:04:04 EDT 2002 , End time Tue Aug 20 00:04:18 EDT 2002
==== On blinky ====
   12:04am  up 75 days,  8:08,  0 users,  load average: 0.00, 0.00, 0.00
  USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT

==== On waldo ====
   12:04am  up 39 days, 14:31,  2 users,  load average: 0.03, 0.04, 0.00
  USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
  bparker  pts/0    martha           Fri 8am  3days  0.18s  0.13s  -bash
  bparker  pts/1    martha            1:00pm  2:27m  0.16s  0.09s  -bash

==== On darma ====
   12:04am  up 123 days, 12:38,  1 user,  load average: 0.93, 0.39, 0.14
  USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
  bparker  pts/0    martha           Mon 7am  1:48m  0.06s  0.06s  -bash

==== On annabelle ====
   12:04am  up 39 days, 17:54,  2 users,  load average: 2.37, 2.06, 1.98
  USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
  wstearns pts/0    martha           Sat12pm  9:39m  2.14s  1.15s  -bash
  wstearns pts/1    martha           11Aug02  2:49m  1.84s  0.86s  -bash

==== On endor ====
   12:04am  up 101 days,  7:15,  2 users,  load average: 0.00, 0.00, 0.00
  USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
```

Now I have a concise summary of who's logged into the servers.

Note that this approach doesn't work for interactive commands. In fact, it won't work if you even have to type a password for ssh; that's why I'm encouraging you to use keys and ssh−agent. I couldn't run an editor on multiple machines at a time with fanout. However, there are lots of commands that work just fine non−interactively; here are some examples:

fanout −−noping "blinky waldo" 'if [−f /var/log/dmesg]; then cat /var/log/dmesg ; else dmesg ; fi' | less
: See dmesg boot output, falling back on current dmesg info if your distribution doesn't store it.

fanout −−noping "blinky waldo" 'df' | less
: How much disk space is left?

fanout −−noping "blinky waldo" "df | egrep '(100%|[7−9][0−9]%)'" | less
: Are any of my partitions 70% full or more?

fanout −−noping "blinky waldo" 'echo My PID is \"\$PPID\"' | less
: Look at an environment variable in the remote shell.

fanout −−noping "blinky waldo" "/sbin/ifconfig | grep 'inet addr'" | less
: What IP addresses are live on this system?

fanout −−noping "blinky waldo" 'last −a | head −−lines=20' | less
: Who has recently logged in?

fanout −−noping "blinky waldo" 'locate formmail' | less
: Is the formmail program anywhere on the system?

fanout −−noping "blinky waldo" 'ps axf' | less
: What tasks are running now?

fanout −−noping "blinky waldo" 'rpm −q redhat−release' | less
: What version of redhat am I running?

fanout −−noping "blinky waldo" 'rpm −qa | grep −i openssl' | less
What openssl rpm versions are installed?
fanout −−noping "blinky waldo" 'sudo rpm −U /tmp/openssl' | less*
Upgrade the openssl rpms with the new files in /tmp. Note that rpm will also accept ftp://... and http://... URLs for the files, so you don't have to pull them down in advance.
fanout −−noping "blinky waldo" 'uname −a' | less
What kernels am I running on these systems?
fanout −−noping "blinky waldo" 'uptime' | less
How long have the systems been up? Good for checking if any have been rebooted recently.
fanout −−noping "blinky waldo" 'w | grep tty' 2>/dev/null | less
Who's logged in on one of the text terminals?
fanout −−noping "blinky waldo" 'w | grep pty' 2>/dev/null | less
Who's logged in from the network?
fanout −−noping "blinky waldo" 'whoami' | less
returns your regular user account name if your key is correctly in place.
fanout −−noping "blinky waldo" 'sudo whoami' | less
returns "root" if you have your key is in place and you have the sudo rights to run whoami as root without a password.
fanout −−noping "blinky waldo" "uname −a ; (if [−f /var/log/dmesg]; then cat /var/log/dmesg ; else dmesg ; fi) | egrep −i '(hd[a−h]|sd[a−h])' ; ls −al /proc/kcore ; cat /proc/cpuinfo" | less
Pulls kernel, drive, memory, and processor info.
fanout −−noping "blinky waldo" "uname −a ; rpm −qa | egrep −i '(openlinux|redhat−release)' ; uptime ; df −P / ; netstat −a | grep ':*'" | less*
Pulls kernel, OS version, uptime, free disk space on the root partition, and all listening ports on the system.

The following doesn't work:
fanout −−noping "blinky waldo" 'sudo echo 0 >/proc/sys/net/ipv4/tcp_ecn' | less
This looks like a good way to echo a "0" into the kernel location that controls ECN (Explicit Congestion Notification). The problem is that although the echo command is run as root, the shell that's running it and handling the ">" redirection is still running as a normal user. It turns out there's a way around this; the "dd" command:

But this one does:
fanout −−noping "blinky waldo" 'echo 0 | sudo dd of=/proc/sys/net/ipv4/tcp_ecn' | less
Since the dd command opens its own files after it's started as root, if can successfully open /proc/sys/net/ipv4/tcp_ecn for writing.
fanout −−noping "blinky waldo" '[`cat /etc/passwd | grep "^bparker:" | wc −l` −eq 0] &sudo adduser bparker' | less
With this one, we want to make sure bparker has an account on both systems, but we don't want to try to create another one if she already has one. The section from [to] tests to see if the account *doesn't* exist; if it doesn't, we adduser bparker. In the test, we count the number of lines in /etc/passwd that have "bparker:" at the beginning of the line; if this equals 0, we know the account doesn't exist.
fanout −−noping "blinky waldo" 'if [`cat /etc/passwd | grep "^bparker:" | wc −l` −eq 0]; then echo No bparker account, making one. ; sudo adduser bparker ; else echo bparker has an account ; fi' | less
By using bash's if − then − else − fi block, we can specify what to do in the cases that bparker has or does not have an account.

Sudo, all the way up to root without a password

While most of the "look at the system" commands above could be run as myself, there are still some that need to be run as root. I've written about sudo in the past, but in that article I showed you how to run commands with sudo *in a way that required the user password*. Unfortunately, fanout really wants to run commands that don't require any input whatsoever, even a user password. Is there a way to run sudo without requiring the user to enter any password at all?

Yes, and it all depends on that users line in /etc/sudoers. To grant "user password required" privileges:

```
wstearns        ALL=(root) PASSWD: /bin/dd,/sbin/insmod,/sbin/ip,/sbin/tc,/usr/sbin/wshaper,/sbin
```

The above lets me run the above 7 commands as root, but when I try, it will ask me to enter my user password.

On the other hand, to grant "no password needed at all" privileges:

```
wstearns        ALL=(root) NOPASSWD: /bin/dd,/sbin/insmod,/sbin/ip,/sbin/tc,/usr/sbin/wshaper,/sb
```

Now I can run those same commands, but I'm no longer prompted for my password at all.

You can have a mix where some commands require a password, others do not:

```
wstearns        ALL=(root) PASSWD: ALL, NOPASSWD: /bin/dd,/sbin/insmod,/sbin/ip,/sbin/tc,/usr/sbi
```

With this line I can run anything I wish as root using sudo. If I run one of the seven commands, I don't need to enter a password, but I do need my user password if I want to run anything else.

Portnames − the icing on the cake

When I use "netstat −a" to see what network connections are open, I usually see the remote IP address or hostname in the listing. This tells me where my connections are going. Unfortunately, as far as netstat is concerned, all our ssh connections (except the one to martha) are to localhost. How can we keep them straight in the netstat listing?

We'll label the *ports*. Simply add the following lines to /etc/services:

```
blinky−fwd      22005/tcp
waldo−fwd       22006/tcp
darma−fwd       22007/tcp
annabelle−fwd   22008/tcp
endor−fwd       22009/tcp
```

Now the "netstat −a" output includes a reminder of listening ports and open connections:

```
[root@sparrow etc]# netstat −a
Active Internet connections (servers and established)
Proto Recv−Q Send−Q Local Address           Foreign Address         State
tcp        0      0 localhost:blinky−fwd    *:*                     LISTEN
tcp        0      0 localhost:waldo−fwd     *:*                     LISTEN
tcp        0      0 localhost:darma−fwd     *:*                     LISTEN
tcp        0      0 localhost:annabelle−fwd *:*                     LISTEN
tcp        0      0 localhost:endor−fwd     *:*                     LISTEN
```

```
tcp        0      0 localhost:47914         localhost:waldo-fwd     ESTABLISHED
...
```

The first 5 are the ports ready to accept connections; the last is an established connection to waldo.

Installing ssh keys − the easy way

I mentioned using ssh keys in an earlier article. With the ssh1 protocol, installing keys is reasonably straightforward; the target directory and filename stay constant; the public key is always copied, verbatim, to ~/.ssh/authorized_keys on the ssh server. As long as you remember that and remember to check the ownership and permissions on that file, your home directory, and ~/.ssh, this approach works well.

The problem is that the ssh1 has some problems that make it significantly less secure than the ssh2 protocol; we should all be working to get away from ssh1.

ssh2 clients and servers can use keys as well, but there are some problems with ssh2's keys. Openssh and commercial ssh2 (both of which support the ssh2 protocol) use different file formats and filenames for their private and public keys. In some cases, two files have to be created on the server.

O'Reilly's "SSH: The Definitive Guide" covers keys in marvelous detail if you want to know the gory details, and they are quite gory. Most of us, however, would much rather get a tool to do the work for us as opposed to learning how to get two programs to interoperate. As always, I have an easy way out. :−)

The ssh−keyinstall utility does all the work for you. It can test for which protocol to use, knows where the public key should go, and knows whether the public key needs to be converted before it can be used.

Lets say you've been using a password to get access to www.goober.org all this time, but now want to use ssh keys, perhaps so you can use fanout as we've shown above.

Pull down the ssh−keyinstall utility and install it. If your local username and your username at www are the same, running it is as simple as typing "ssh−keyinstall −s www.goober.org". I'll annotate a sample run:

```
[test@sparrow test]$ ssh-keyinstall -s www.goober.org
ssh-keyinstall, version 0.1.9

Please report any successes or failures to wstearns@pobox.com . Please
note the client and server ssh versions, type of key used (rsa=ssh
protocol 1 or dsa=ssh protocol 2), whether you created a new key or used
an existing key, and whether you forced a particular remote command and
send that information to William at the above address as he is hoping to
test every combination.
Please enter a truly hard to guess passphrase.
Generating public/private dsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Here's where you assign a passphrase to the private key; this encrypts it on the disk only. Before you can use the private key, you'll be asked to enter the passphrase. This should be sentence length, upper and lowercase letters, with strange characters and digits mixed in.

```
Your identification has been saved in /home/test/.ssh/id_dsa.
Your public key has been saved in /home/test/.ssh/id_dsa.pub.
The key fingerprint is:
f6:5d:50:78:67:66:1d:47:c4:44:f6:1c:2b:fd:52:2b test@sparrow.stearns.org
```

The key has been created for you.

```
You will be asked to enter your remote password for each of the
following commands. If you disagree with a particular command, or simply
wish to perform it yourself, enter an incorrect password.
+ ssh test@www.goober.org 'chmod 700 ~ ; mkdir −p ~/.ssh ; chmod 700 ~/.ssh ; rm −f ~/.ssh/newpub
The authenticity of host 'www.goober.org (192.168.12.13)' can't be established.
RSA key fingerprint is 47:ef:9d:f7:d8:67:e9:95:27:06:90:80:00:36:c8:3d.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'www.goober.org' (RSA) to the list of known hosts.
```

You'll see the above warning and check if this is the first time you've connected to the target machine. Enter yes only if you're quite sure this is the machine to which you wanted to connect.

```
test@www.goober.org's password:
+ '[' −n '' ']'
+ '[' −n /home/test/.ssh/id_dsa.pub ']'
+ scp /home/test/.ssh/id_dsa.pub test@www.goober.org:.ssh/newpubkey
test@www.goober.org's password:
id_dsa.pub             0% |                                |     0       −−:−− ETAid_dsa.pub
+ ssh test@www.goober.org 'cat ~/.ssh/newpubkey >>~/.ssh/authorized_keys2.temp'
test@www.goober.org's password:
+ ssh test@www.goober.org 'chmod 600 ~/.ssh/authorized_keys2.temp ; rm −f ~/.ssh/newpubkey ; mv −
test@www.goober.org's password:
+ set +x
```

The installation requires 4−6 seperate ssh or scp connections to the ssh server. Because your key is in the process of being installed, you'll have to enter your password for each step. The '+' lines show you what command is about to be run.

```
The public key has been installed on www.goober.org . If you wish to test
this, you can run the following command, which should prompt you for
your passphrase and return the string HI.
   ssh −l test www.goober.org '/bin/echo HI'
If you are prompted for your remote pass_word_, the key installation did
not work. In this case, please let William Stearns <wstearns@pobox.com>
know of the failure. Please include the ssh client and server versions
and the exact command you just ran.
```

The program gives us a command line to run to test that the key works; we run it.

```
[test@sparrow test]$ ssh −l test www.goober.org '/bin/echo HI'
Enter passphrase for key '/home/test/.ssh/id_dsa':
HI
[test@sparrow test]$
```

At this point, you can set up ssh−agent (see my ssh−agent tutorial for how to do so) and you can make ssh connections to thousands of machine simultaneously without having to use a password *or* passphrase.

William is an Open−Source developer, enthusiast, and advocate from New Hampshire, USA. His day job at SANS pays him to work on network security and Linux projects.