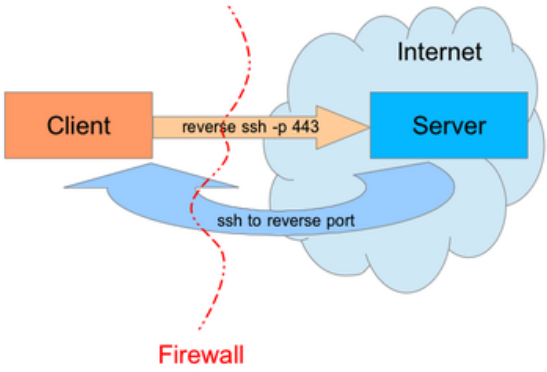
Fun and Profit with Reverse SSH Tunnels and AutoSSH

Posted on June 20, 2016

Sometimes you just need to get to things remotely. Those things might be behind an unbudging firewall with no way to forward proper ports for remote access, or perhaps you just want it setup access to resources this way. I recently needed to establish a persistent connection to a new Raspberry Pi setup remotely where the ISP did not allow proper port forwards. Luckily you can use reverse SSH tunneling and AutoSSH (automated tunnel persistence) and carry on.

How it Works

SSH reverse proxies work very much like the following illustration:



- You connect to a remote host via SSH
- Specify the -R option to open up a reverse tunnel on a local port on the remote host
- From the remote host you can access the originating machine via the local port.

Example in Practice

You connect from remote.example.com to homeserver01 via SSH on port 9991

```
[user@remote ~]# ssh homeserver01 -p 9991 -R 8081:localhost:9991
```

On remote machine homeserver01 you can now connect back to the original machine on port 8081 where you have bound a local port.

```
[user@homeserver01 ~]$  ssh localhost -p 8081
```

Let's break down the options above:

- *-p 9991* = We connect to SSH listening on the remote host (homeserver01) on TCP/9991
- *-R 8081:localhost:9991* = We bind local port 8081 on the remote host, and again specify 9991 as our initial connecting port that where SSH listens.

Automating with AutoSSH

What happens if this disconnects after you leave? Well, you can no longer access the reverse tunnel you've setup. The good news is there's a program that will set this up for you and keep it connected. You can also start it as a service or on reboot.

First, install autossh. Most distributions will have autossh packaged, if not you'll need to build it from source but we won't cover that here.

```
[root@remote ~]# yum install autossh -y
```

AutoSSH can take a configuration file or you can pass it parameters when you run it. We're going to use the runtime options for illustration here

Make AutoSSH Users

You'll want a set of dedicated users and SSH public key for AutoSSH. On the originating host (where you'll be initiating the tunnel from) you probably want to give it a false shell.

Create the originating host user

```
[root@remote ~]# useradd -m -s /sbin/nologin autossh
```

Create an SSH key for the user

```
[root@remote ~]# su - autossh -s /bin/bash
[autossh@remote ~]$ ssh-keygen -t rsa
```

You should **not set a passphrase** here so it can be automated, and go with the other defaults by hitting enter.

```
[autossh@remoteserver ~]$ ssh-keygen -t dsa
```

```
Generating public/private dsa key pair.
Enter file in which to save the key (/home/autossh/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
```

Now create a user on the destination server, this time you'll need to set a valid shell along with a password if you want to remotely copy the SSH public key.

```
[root@homeserver01 ~]# useradd -m -s /bin/bash autossh
[root@homeserver01 ~]# autosshpass=`date | md5sum | cut -c1-10` && echo "$autosshpass" \
| passwd autossh --stdin && echo "password is $autosshpass"
```

Remember the password echoed to the terminal, in this case mine was *d53a922e9e*.
You'll need this in the next step.

Note: Ubuntu users have reported that the –stdin option isn't present on some systems, you can do this instead in that case:

```
echo -e "$autosshpass\n$autosshpass" | passwd autossh
```

Copy SSH Keys

Back to the remote server *(machine you want to establish the reverse connection to)* you'll want to copy the newly created key over using the password you generated.

```
[autossh@remoteserver ~]$ ssh-copy-id homeserver01 -p 9991
```

Enter the password and you should get a nice message back saying keys are successfully copied.

Test the Connection

At this point you're ready to test the connection. Run the following AutoSSH command which should establish your tunnel. You can use the *-vvv* flags for extra verbosity to troubleshoot if it doesn't work.

```
[root@remoteserver ~]# su - autossh -s /bin/bash
[autossh@remoteserver ~]$ autossh -M 0 homeserver01 -p 9991 -N -R 8081:localhost:9991 -vvv
```

You should see something like the following.

```
OpenSSH_6.6.1, OpenSSL 1.0.1e-fips 11 Feb 2013
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: /etc/ssh/ssh_config line 56: Applying options for *
debug2: ssh_connect: needpriv 0
debug1: Connecting to homeserver01 [x.x.x.x] port 9991.
debug1: Connection established.
```

At this point you can now test the port on the remote host for an SSH connection back via the reverse proxy.

```
[user@homeserver01 ~]$ telnet localhost 8081
```

```
Connected to localhost.
Escape character is '^]'.
SSH-2.0-OpenSSH_6.6.1
```

Start on Boot – Cron Method

Great! If you got this far you're setup, now let's make this start on boot and persist with a cron job. Alternatively you can create a Systemd service if you're using a distribution that supports it.

Add the following to root's crontab, adjusting as needed

```
[root@remoteserver ~]# crontab -e
```

Add the below lines adjusting to your setup, save and quit.

```
@reboot /bin/sudo -u autossh bash -c '/usr/local/bin/autossh -M 0 -f autossh@homeserver01 -p 9991 -N -
```

Systemd Service *(Preferred)*

If you're using Systemd you can create a unit file for this. Note the binary paths below, they are needed for Systemd unit files when running as non-root users.

```
cat > /etc/systemd/system/autossh-homeserver01.service << EOF
[Unit]
Description=Keep a tunnel to 'homeserver01' open
After=network.target

[Service]
User=autossh
ExecStart=/bin/autossh -f -M 0 autossh@homeserver01 -p 9991 -N -o "ServerAliveInterval 60" -o "ServerA
ExecStop=/usr/bin/pkill autossh
Restart = always
[Install]
WantedBy=multi-user.target
EOF
```

Now enable the service.

```
[root@remoteserver ~]# systemctl enable autossh-homeserver01.service
[root@remoteserver ~]# systemctl start autossh-homeserver01.service
```

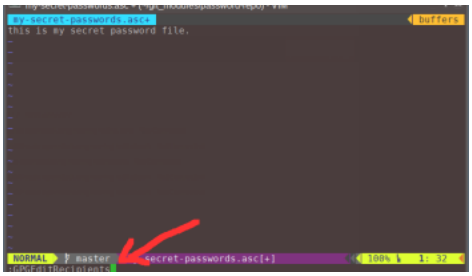
Extending Reverse SSH Tunnels

Reverse tunnels over SSH are useful for more than just SSH connections, you can also use them to access otherwise local-only listening ports. Another common example might be MySQL (TCP/3306).

Disclaimer

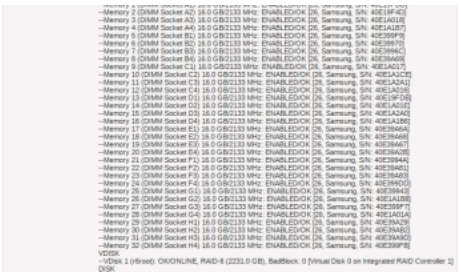
I am in no way endorsing using this for anything nefarious like getting around a corporate VPN or firewall. This is simply a guide of how to use secure, reverse tunnelling where connectivity would normally not be possible.

Related



Secure, Distributed Password Resources with GPG, Git and Vim

In "open source"



Automate Nagios Monitoring with Ansible

In "open source"

Quick and Dirty Remote Execution with Ansible

In "open source"

About Will Foster

hobo devop/sysadmin, all-around nice guy.

View all posts by Will Foster →

This entry was posted in open source, sysadmin and tagged linux, reverse proxy, security, ssh. Bookmark the permalink.

8 Responses to *Fun and Profit with Reverse SSH Tunnels and AutoSSH*

Julian Phillips *says:*

July 6, 2016 at 8:00 am

Thank you for a great article, your advice will be very helpful for me.

Like

Reply

Alex *says:*

March 17, 2017 at 12:41 pm

Thanks a bunch for very clear step-wise instructions. Everything worked as expected:-)

– Alex.

★ Like

Reply

Kyle *says:*

September 1, 2017 at 1:00 am

Amazing, this is so valuable to me! Thank you!

★ Like

Reply

Will Foster *says:*

September 1, 2017 at 11:51 am

Thanks Kyle, I'm really glad it was useful to you.

★ Like

Reply

Joey *says:*

October 7, 2017 at 6:40 pm

btw you need a uppercase m to create a user with no home dir

★ Like

Reply

Will Foster *says:*

October 8, 2017 at 7:53 pm

> *btw you need a uppercase m to create a user with no home dir*

Hey Joey, I do indeed want to create a homedir so we have a place to keep ssh keys.

★ Like

Reply

Marc Compere *says:*

December 3, 2017 at 5:00 pm

This was SUPER helpful. In the "Make AutoSSH Users" section above, my /usr/bin/passwd in Ubuntu 17.10 (Artful Aardvark) seems to

be missing the –stdin option for some reason. The line below can be used instead of the –stdin portion for creating the autossh password described above:

echo -e “$autosshpass\n$autosshpass” | passwd autossh

Other notes:
– I used the -t rsa to generate rsa keys. The dsa keys are probably fine but seem to be deprecated in openssh
– I had to add ‘AllowUsers autossh’ to /etc/ssh/sshd_config (then ‘service sshd restart’) on the ssh server to copy keys over

The next step is setting up a systemd service so the Raspberry Pi 3 device will wake up, connect to a mobile hotspot, then open the remote ssh port so login is avaialble over the cellular network. Aside from a custom written tcp tunnel, this is the only way I know how to get login access to a device over a mobile cellular network.

Mosh would be better for a cellular mobile network connection but the ssh remote port forwarding is the only way to get access to an unknown ip address and also through blocked inbound udp ports on the remote device.

Thank you, this was really helpful.

★ Like
Reply

Will Foster *says:*
December 3, 2017 at 7:03 pm

Hey Marc, thanks for the tip rsa vs. dsa as well as the lack of –stdin for Ubuntu (this is present on CentOS/Fedora for me). For ‘AllowUsers’ that’s usually never set in stock configurations that I’ve seen, being an optional setting that someone usually locks down afterwards. I’m glad that you found the guide useful and I appreciate the feedback.

★ Like
Reply