PRACTICAL PORTS

SSH Tips and Tricks

BY BENEDICT REUSCHLING

SSH is a multipurpose tool used by pretty much anyone running and administering a Unix machine, logging in from one to the next more often than not. Secure, encrypted logins as SSH's core functionality are a great day-to-day help, but the daemon can do much more. Plenty of use cases are possible and I will only describe the ones that I use, which is by no means a complete list. I'll start with some basic hardening options and then move on to more sophisticated uses.

Login Options

SSH users that want to log into a system should use SSH keys instead of the keyboard-interactive authentication. Ideally, the latter authentication method (showing a password prompt) should be completely disabled to prevent any password script kiddie from constantly trying to break into poorly secured systems with weak user passwords. On some systems that I run for less tech-savvy users (I tried educating them on ssh keys, but to no avail), I still have to enable it. However sad this may be, I can still lock down the system to limit logins to a certain group of users and force some users to not use the keyboard-interactive method. Let's look at these one by one.

SSH users that want to log into a system should use SSH keys instead of the keyboard-interactive authentication.

The SSH daemon is configured by the configuration file `/etc/ssh/sshd_config`. You can limit the users who are able to log in with the `AllowUsers` or `AllowGroups` directive. Even if a local user tries to log in and provides the correct password, as long as they are not listed in those directives, the login is still denied. In my use case, all the non-tech savvy usernames start with "abc", followed by their individual user ID. We can use wildcards to match on this username like this:

```
AllowUsers abc*
```

Use `sshd -t` or even `-T` (for more tests) to check the validity of the `sshd_config` file before you restart the daemon to make these changes take effect. This line excludes the script kiddies from trying usernames like root, admin, and such. It's not completely secure but adds an extra hurdle to overcome. The AllowGroups directive does the same for a group of users.

User groups are easier to manage at a central location. Your new colleagues will certainly appreciate getting access immediately without having to visit your office first and beg to be let into that server. If they're added to the group, they can log in right away. Otherwise, you have to modify the `AllowUsers` line each time. The same is true when someone leaves, so do yourself a favor and use groups so as not to spend the rest of your days managing SSH access to a system that the whole company logs into.

Denying logins for certain users or even groups is possible with the `DenyUsers` or `DenyGroups` command, respectively. April Fools jokes aside, this is useful for restricting certain users in a group from accessing the system until they've returned the only existing keys to the server room (or are back from their vacations). The following order of directives is used when processing a mixture of such entries: `DenyUsers`, `AllowUsers`, `DenyGroups`, and then `AllowGroups`.

Restricting individual users' login methods is also possible using a match statement. This is comparable in function to an if statement and when such a match occurs, they override the global settings of `sshd_config`. Other match lines further down the file could undo settings made in previous match blocks, but let's keep this example simple for now.

Restricting individual users' login methods is also possible using a match statement.

The system described above where both keyboard-interactive and public key are used is monitored by a special user called monitoring. It periodically logs in with its own special SSH key, runs certain checks (disk full?), and reports the results back to the central monitoring server. This user should not be allowed to log in via the keyboard, since compromising this user would basically mean root access as some of the checks run with elevated privileges. That is why we tell the SSH server to only allow public key authentication when this user logs in. The match statement looks like this:

```
Match User monitoring
    AuthenticationMethods publickey
```

Other users still use the global settings, but once the monitoring user comes along, the AuthenticationMethods setting gets overridden for this case. Other criteria for a match statement are Group, Host, LocalAddress, LocalPort, RDomain (the routing domain), and Address. Be careful here not to trust certain networks a user pretends to come from as this may be spoofed or rerouted. Find something that matches as little as possible to avoid long processing times or matching too many items, defeating the purpose of matching.

Note that not all keywords from `sshd_config` are changeable in a match statement, but many of them are. A full list is available in `sshd_config(5)`. Happy matchmaking!

Disconnecting Hung SSH Sessions

Has the following ever happened to you? You are logged in remotely on a server — doing some work — when suddenly your terminal freezes and you can't send any more commands to it? Or, you closed the lid of your laptop and opened it again at a different network location and can't get back to the session leaving you stuck at the terminal? I imagine you nodding in agreement, so here's why: it is because the server did not notice the network interrupt that may have happened and can't re-establish the connection. If this annoys you, check out the **`net/mosh`** package and see if that helps you.

How do you get the frozen shell back or at least properly disconnect? Even though it seems that there is no more communication happening between your SSH client and the server, there are still special commands that the server understands. Enter the following sequence of commands on the frozen terminal:

```
Enter ~ .
```

This sends a special interrupt command causing the server to immediately disconnect the session, returning control to your local shell session. Try it on a live session by hitting Enter, the tilde character followed by the dot. Be quick about it. Once it works, you'll quickly memorize this as a good practice if only to impress your co-workers with your knowledge.

Other sequences are documented in the ESCAPE CHARACTERS section of **`ssh(1)`**. Typing the sequence Enter, ~, and ? displays a list of escape sequences. Note that not all are supported by each SSH daemon, but in my experience, the disconnect works reliably.

The Hidden Login Script

Did you know that you can run a script each time a user successfully logs into a system via SSH? The file **`/etc/ssh/sshrc`** that does this magic does not exist by default. When it is created and made executable, the SSH daemon will execute the commands listed in it. This happens after the environment files are read and right before the user's shell (or a command) is started.

Why would that be useful, you ask? It allows for custom initialization routines to run for this user. For example, a shell script could use the user's name for log in (available as **`$USER`**) and ensure access permissions and ownership on the home directory are still restricted to that user. An error is echoed to **`stderr`** if the home directory does not exist for some reason. A simple script that does this may look like the following:

```
#!/bin/sh

HOMEDIR="/home/${USER}"

# Restore restrictive home directory permissions
if [ -d ${HOMEDIR} ]; then
  chmod 0700 ${HOMEDIR}
  chown ${USER}: ${HOMEDIR}
else
```

```
  echo "Home directory ${HOMEDIR} does not exist" >&2
fi
```

Make sure the file is executable just like any other shell script to activate it the next time someone logs in. Other environment variables are available to use as well. Note that this runs every time a user logs in — even for file transfers via `scp/sftp`. Don't put in complicated code that takes a while to execute or the user will have to wait to finish the login process. For sneaky, behind-the-scenes actions, this is a good way as every user who successfully logs has to pass through the script.

Cheap, Yet Effective VPN

Virtual private networks have sprung up as paid services. They allow a secure connection between endpoints by tunneling the traffic and encrypting it over the internet. This helped people stay connected with the office during the pandemic, and even before that, when support personnel had to fix a server at ungodly hours of the day to prevent business interruptions. For Josephine random person, the aforementioned paid services enabled them to buy things cheaper by faking their origin connection to come from a different country. This could range from airplane tickets to as yet unreleased episodes of your favorite show on your favorite streaming service. While this may not yield success in all cases, it is nevertheless a convenient service to use.

Virtual private networks have sprung up as paid services.

How (and when) does SSH come into play? Well, each time we are on an untrusted, unencrypted network and we don't want prying eyes reading our traffic. This is often the case at train stations, airports, hotels, and libraries that offer free public WIFI. The connections there are often not (or not well) encrypted and shared between many different users--a perfect use case for an SSH-based VPN solution. It does not cost us anything since we have all the tools available. All that is needed is a publicly available host reachable on the internet that you can legitimately log in to.

Instead of directly connecting from our origin host O to the destination host D, we let our traffic take a little detour via host P. This takes care of giving the network packets a different origin address. Instead of your original host O, the packets will all be fetched by host P and forwarded to you via a SOCKS proxy. The destination D will communicate with host P, handling all requests, and each answer or result sent to P is forwarded to O in return. The SOCKS proxy will allow your browser to send and receive the packets, just like you'd browse a normal web page directly. It may be a bit slower than you are normally used to — because of the redirections between you and host P (the proxy) — but this is worth it to hide our origin address and the extra encryption you'll get — for free.

Here's How To Do It

Open a new terminal and type in the following, replacing the **sshproxyhost.example.com** with your SSH host the internet:

```
$ ssh -vD8080 -fCN sshproxyhost.example.com
```

This looks complicated, so let's explain each of the options provided:

- **v:** Gives SSH verbose output and may be omitted later once you're familiar with what's going on. At the beginning, it helps to debug the process and will emit any error messages that you wouldn't see normally.
- **D 8080:** This defines the local, dynamic port for the forwarding. On your local machine, the provided port (Note: 8080 in this case, a different one can be selected as long as it is unused by other daemons) is opened as a local socket with the other endpoint being the secure connection to the proxy server (creating it in the process).
- **f:** Puts the SSH process in the background so that the shell can still take other commands. Note that if you need to provide a password to log in, this will not work well. Generate an SSH key (using **ssh-keygen(1)**) for this connection and exchange it with the proxy host (**ssh-copy-id(1)**) for passwordless logins. This option is not strictly necessary, but useful once the whole process works.
- **C:** Compresses the encrypted VPN data. Depending on your network and processor speed, this may slow down or speed up the connection. Experiment with this option and remove it if it is too slow on the shabby hotel service where you're staying for one night only.
- **N:** SSH expects to run an interactive shell on the remote host, but we don't need this for our VPN. This option will not let SSH open a terminal and will only forward the ports--which is what we want.

Each of these options is explained further in **ssh(1)**.

A typical session will emit similar messages to his one when using **-v**:

```
OpenSSH_8.6p1, LibreSSL 3.3.6
debug1: Reading configuration data /Users/bcr/.ssh/config
debug1: /Users/bcr/.ssh/config line 1: Applying options for *
debug1: /Users/bcr/.ssh/config line 16: Applying options for sshproxyhost.example.com
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: /etc/ssh/ssh_config line 21: include /etc/ssh/ssh_config.d/* matched no files
debug1: /etc/ssh/ssh_config line 54: Applying options for *
debug1: Authenticator provider $SSH_SK_PROVIDER did not resolve; disabling
debug1: auto-mux: Trying existing master
debug1: Requesting forwarding of dynamic forward LOCALHOST:8080 -> *
debug1: mux_client_request_session: master session id: 7
...
debug1: Connection to port 8080 forwarding to socks port 0 requested.
debug1: channel 3: new [dynamic-tcpip]
...
debug1: channel 8: new [dynamic-tcpip]
```

```
debug1: channel 7: free: direct-tcpip: listening port 8080 for sshproxyhost.example.
com port 443, connect from 127.0.0.1 port 58994 to 127.0.0.1 port 8080, nchannels 9
```

This confirms that the VPN is established. The line

```
debug1: Requesting forwarding of dynamic forward LOCALHOST:8080 -> *
```

shows how it works: you connect to your localhost port 8080. From there (follow the arrow), connections are established into the wide, wild world (www) of the internet. Replies are sent back in the reverse direction.

Your browser (or any other application that should use this VPN tunnel) simply needs to be set to use this SOCKS proxy in their connection settings. Look for an option like "manual proxy configuration", set the socks host to "127.0.0.1" (localhost, see above), the port to 8080 (or the one you specified), and SOCKS5 proxy if there is such an option.

That's it. Now, we should check if it works by browsing to a service like https://www.whatismyip.com (or similar sites) that display your public IP address. If this shows the IP of the host you used in your SSH command (sshproxyhost.example.com in my example), the VPN works. Wherever you next point your browser, the websites will establish connections, exchange traffic with this particular host and dutifully send you the traffic. Nice, isn't it?

Some Words of Warning

As long as the SSH connection is open to the target system, the VPN tunnel is established. Be sure to re-establish the tunnel after your laptop goes to sleep as it may have disconnected you after some time of inactivity. If you're renting a server on the internet to be a proxy for this purpose or someone else pays for the traffic on this system, don't overdo it, as it may drive up costs. This is not a free solution in that case and if you use this often, you might as well pay for a professional VPN solution that gives you a couple servers across the world to choose from.

Also, be aware that you are not completely invisible. The SSH logs of the server used as a proxy will record your login information. Don't do any malicious or harmful activities with this. We won't send your next issue of the *FreeBSD Journal* to the prison planet that they put you on when they catch you.

I hope these tips and tricks were useful and will help in your day-to-day SSH interactions. Make sure to check out the man pages for both the client (**ssh(1)**) and the server (**sshd(8)**). For a more fun and comprehensive reading experience about all things SSH, I highly recommend the *SSH Mastery* book by Michael W. Lucas.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.