SSH TUNNELLING FOR FUN AND PROFIT: SSH CONFIG

By cytopia January 25, 2016 11 comments Administration ssh, ssh tunnelling, ssh_config

This series has already covered a few basics about `~/.ssh/config` in terms of how to simplify the usage of ssh tunnelling. In order to round this up a bit more, I will add some information you should be aware of about `~/.ssh/config`. This is only intended to be a quick reminder about how it is done right and some useful hints you might not have heard about.

The following uses examples with pure `ssh` connection commands, but it is also applicable to establish tunnels with `ssh` as they all read the same configuration file.

Article series

SSH tunnelling for fun and profit

TL;DR

Nope, this time you need to read it all.

Structure of SSH Config

Probably the most important part which is widely overlooked is the **order of definition blocks** in `~/.ssh/config` and accordingly `/etc/ssh/ssh_config` in terms of generalization and specialization.

You can basically categorize blocks into three stages:

1. Most specific (without any wildcards)
2. Some generalization (with wildcard definitions)
3. General section (which applies to all).

Let's define a basic `~/.ssh/config` containing the examples above and see what it does:

Wrong way

No, what many people do wrong is to define the general stuff at the top. Let's do this for a second and see what the outcoming ssh connection string will be:

```
Host *
    User root
    Port 22
    PubkeyAuthentication no
    ServerAliveInterval 30
```

```
Host c*
    User cytopia
    Port 10022
    PubkeyAuthentication yes
    IdentityFile ~/.ssh/id_rsa__c_cytopia@cytopia-macbook

Host c1
    HostName 192.168.0.1

Host c2
    HostName 192.168.0.2
```

If you want to ssh connect to `c1` (`ssh c1`), the file is read as follows:

1. Find section `Host *`
    i. Apply User: root
    ii. Apply Port: 22
    iii. Apply PubkeyAuthentication: no
    iv. Apply ServerAliveInterval: 30
2. Find section `Host c*`
    i. Ignore User (already defined above)
    ii. Ignore Port (already defined above)
    iii. Ignore PubkeyAuthentication (already defined above)
    iv. Apply IdentityFile
3. Find section `Host c1`
    i. Apply HostName: 192.168.0.1

The final connection string that will be made internally will look like this:

```
ssh root@192.168.0.1 -p 22 -i ~/.ssh/id_rsa__c_cytopia@cytopia-macbook -o PubkeyAuthentication=no -o ServerAliveInte
```

Now this is totally not what you intended to do!

Right way

Let's restructure the `~/.ssh/config` into the right order and check the resulting connection string:

```
Host c1
    HostName 192.168.0.1

Host c2
    HostName 192.168.0.2

Host c*
    User cytopia
    Port 10022
    PubkeyAuthentication yes
    IdentityFile ~/.ssh/id_rsa__c_cytopia@cytopia-macbook

Host *
    User root
    Port 22
    PubkeyAuthentication no
    ServerAliveInterval 30
```

The important part to keep track of is the `Host` section (aligned to the left). Notice here that the general definitions are at the very top and more wildcarded definitions (using the asterisk `*`) are followed below.

If you want to ssh connect to `c1` (`ssh c1`), the file is read as follows:

1. Find section `Host c1` and use its corresponding `HostName` (192.168.0.1)
2. Find more general section `Host c*` and use their values (User, Port, etc).
3. Find most general section `Host *`

i. Don't use `User` as it has already been defined for this connection in `c*`
ii. Don't use `Port` as it has already been defined for this connection in `c*`
iii. Don't use `PubkeyAuthentication` as it has already been defined for this connection in `c*`
iv. Use `ServerAliveInterval` as there is no previous definition.

So from that you must always remember that whenever a specific value has been found, it cannot be overwritten by values defined below. It is a first come first server here. The final connection string that will be made internally will look like this:

```
ssh cyptopia@192.168.0.1 -p 10022 -i ~/.ssh/id_rsa__c_cytopia@cytopia-macbook -o PubkeyAuthentication=yes -o ServerA
```

Now this is how you intended to connect. So always remember:

1. **Specific definitions at the top**
2. **General definitions at the bottom**

Why use SSH config anyway?

Simpler usage

Imagine you have a couple of dozens or even hundred servers you have to take care of. Each of them is having different login options such as: some still use passwords, others use rsa keys, others ed25519 keys, lots of different initial users to use for the connection and much more. Wouldn't it be much more simple to define everything into a file and don't care about the rest anymore?

You could for example use a naming convention for clouded vs. dedicated hosts as so:
c1, c2, c3, …, d1, d2, d3

Or you use hosts per customer:
google1, google2, google3, …, apple1, apple2, apple3

All those hosts might have completely different settings even different ports and you simply need to

```
$ ssh c1
$ ssh d2
$ ssh google1
$ ssh apple3
...
```

Other applications make use of it too

Most programs that make use of ssh can use the same alias specified in `~/.ssh/config` with the same options, simply by specifying the alias inside this program.
For example on OSX I am using Sequel Pro to manage all my MySQL connections. Instead of having to specify host, user, port and certificate (in the ssh tunnel section), I simply only specify the ssh alias and it will auto-grab all details from my `~/.ssh/config`.

I am sure there are many other programs out there that are also able to make use of it.

On top of that, if you need to alter settings of one server, you do it in a central place and it will have an effect on all tools instantly.

Autocompletion

You will have autocompletion (at least under bash or zsh) for every host and every alias defined. This is true for hosts and even IP addresses. When I type `ssh 1` and hit tab:

```
$ ssh 1
192.168.0.1     192.168.0.2     192.168.0.3     192.168.0.4     192.168.0.5     192.168.0.6
192.168.0.7     192.168.0.8     192.168.0.9     192.168.0.10    192.168.0.11    192.168.0.12
```

> *Note: I have replaced the IP addresses with internal once.*

Hostnames

```
$ ssh c
c1                              c15.example.de                  c4
c1.example.de                   c16                             c4.example.de
c10                             c16.example.de                  c5
c10.example.de                  c17                             c5.example.de
c11                             c17.example.de                  c6
c11.example.de                  c18                             c6.example.de
```

> *Note: I have replaced the domains with example.de once.*

Defaults

Within the most general configuration section you can define settings that must be applied for every `ssh ...` you type.

So hopefully the `~/.ssh/config` has raised your attention by now.

What your mother never told you about `~/.ssh/config`

Identity leak via ssh keys

If you are a big fan of ssh keys in combination with ssh-agent, then you should be aware that once you connect to any ssh server, all of your public keys that are hold by your ssh-agent, are sent to this server.

> *You can check which keys are stored inside your ssh-agent via ssh-add.*
>
> ```
> $ ssh-add -l
> 4096 SHA256:111_SSH_HASH_111 /Users/cytopia/.ssh/id_rsa__host_root@me (RSA)
> 256  SHA256:111_SSH_HASH_111 /Users/cytopia/.ssh/id_ed25519__host_user@me (ED25519)
> ...
> ```
>
> *By default, if you do not manually add any keys via `ssh-add`, all defaults (no custom name) for rsa, dsa ecdsa and ed25519 (usually `id_rsa`, `id_dsa`, `id_ecdsa` and `id_ed25519`) are added to the ssh-agent (once they are created).*
>
> *So this means, if you have created one default rsa key simply by typing `ssh-keygen`, you will have this key `~/.ssh/id_rsa` and this key will also be added to your ssh-agent by default.*

This means if you connect a lot to many untrusted ssh servers, they might log your keys (just like websites track you via cookies) and might be able to identify.

The problem has been address by https://github.com/FiloSottile/whosthere/ which can identify your github name.

> *Test if your ssh client settings are vulnerable to github identity leak:*
>
> ```
> $ ssh whoami.filippo.io
> ```
>
> *This is the example from: https://github.com/FiloSottile/whosthere/, make sure to visit this github page.*

What FiloSottile recommends is to turn off public key authentification in general and explicitly turn it on per host (where u need it):

```
# Turn on pubkey auth per specific HOST
Host c1
    HostName 192.168.0.1
    PubkeyAuthentication yes
    IdentityFile ~/.ssh/id_rsa_specific

# Turn off pubkey auth for all hosts
```

```
Host *
    PubkeyAuthentication no
    IdentitiesOnly yes
```

Securing `known_hosts`

Let's look at a line of a typical `~/.ssh/known_hosts` file:

```
cvs.example.net,192.0.2.10 ssh-rsa AAAA1234.....=
```

Space separated fields in order of occurance
1. [optional] markers
2. hostnames (comma separated)
3. Bits, exponent and modulus
4. [optional] comment (not used)

This file is pretty talkative and can tell all the hosts you have visited so far and therefore has some implications. You can read more about the problems here: Protecting SSH from known_hosts Address Harvesting

So in order to only store hashes of the hostnames inside `~/.ssh/known_hosts`, you will need to alter `~/.ssh/config`:

```
Host *
    HashKnownHosts yes
```

The hashed version for the file will look like this:

```
|1|JfKTdBh7rNbXkVAQCRp4OQoPfmI=|USECr3SWf1JUPsms5AqfD5QfxkM= ssh-rsa AAAA1234.....=
```

Note 1: Keep in mind that the hashing will start from now on and previous entries will not be hashed.

Note 2: With hashing you will loose the autocompletion feature from `known_hosts`, but when you use aliases, you still have the alias based autocompletion described above.

Multiple connections inside a single one

SSH needs some time establishing a connection. This time grows as you use stronger/bigger private/public key-pairs. However, if a server receives a lot of connections, this time might matter and fortunately there is a way to reduce it by multiplexing multiple ssh connections over a single one from the same host/user by re-using an already established connection.

So how can this be established? Again, you can configure this behavior inside your `~/.ssh/config` globally as the following example shows:

```
Host *
    ControlMaster auto
    ControlPath ~/.ssh/sockets/%r@%h-%p
    ControlPersist 600
```

Option	Description
`ControlMaster`	Tell SSH to re-use an existing connection (if there is already an established one) without having to authenticate again.
`ControlPath`	This is the path of the socket for open SSH connections. Every new connection will hook into this socket and can use the already established connection
`ControlPersist`	Keep the master (the first) SSH connection open for X seconds after the last connection has been closed. This means you have X seconds to connect again without authentification after all connections have been closed to this host.

Where does this matter?

Nagios

If your nagios server does multiple SSH checks (`check_ssh`) against one server, it is recommended to setup the nagios' servers ssh client to re-use existing ssh connections in order to speed up those checks.

Git

If you do a lot of work with git very frequently like pushing, using the autocomplete feature (which requires some remote connection to upstream), etc you are probably also a candidate to re-use existing SSH connections.

I am myself are not a fan of enabling the whole thing globally (except for the nagios server), but rather for specific use cases.
So If you want to enable this for specific hosts only you could do it like that:

```
# For some host
Host c1
    HostName 192.168.0.1
    ControlMaster auto
    ControlPath ~/.ssh/sockets/%r@%h-%p
    ControlPersist 600

# For github usage
Host github.com
    HostName github.com
    User git
    ControlMaster auto
    ControlPath ~/.ssh/sockets/%r@%h-%p
    ControlPersist 600
```

Private ssh key leak

Have you heard about the recents cve's about possible private key leaks via ssh to a malicious SSH server: CVE-2016-0777 and CVE-2016-0778.

In order to avoid this possible vulnerability add the following undocumented setting to your `~/.ssh/config` at the bottom inside the general section:

```
Host *
    UseRoaming no
```

Useful tools

As I have lots of ssh hosts configured in my `~/.ssh/config` and it would be impossible for me to remember which domain is hosted on which server (especially if a single server is having more than 20 separate domains), I am using a little helper script, that will search my ssh configuration file for a given domain or any other keyword and will present me the server it is hosted on.

For example, If I want to know on which server www.everythingcli.org is hosted I can simply type:

```
$ sshf everything
------------------------------------------------------------
 c1
------------------------------------------------------------
 @vhosts: www.everythingcli.org another-domain.com
```

So it told me, that there are two domains on server `c1` including the one I was looking for (which will be auto-highlighted via `grep --color`). No I can simply go there via:

```
ssh c1
```

So if you find this useful, you can find the script at github:

cytopia/sshf

Eof

I hope you enjoyed this little introduction to `~/.ssh/config` and noticed that it is just as complex as the corresponding server configuration. Keep in mind that I just covered some basics mixed with a few specific examples. There is much more to this configuration file, so go on and read up about the power of your ssh client: man ssh_config

_

‹ SSH tunnelling for fun and profit: Autossh

AWS Single Sign-On with AzureAD: AWS Account Structure ›

11 COMMENTS ON “SSH TUNNELLING FOR FUN AND PROFIT: SSH CONFIG”

Pingback: SSH tunnelling for fun and profit: Autossh

Pingback: SSH tunnelling for fun and profit: local vs remote

Pingback: Links 26/1/2016: MPlayer 1.2.1, Parsix GNU/Linux 8.5 | Techrights

Pingback: Configuring SSH | 0ddn1x: tricks with *nix

Arnaud April 20, 2016 Reply

Awesome article thanks

Pingback: SSH Basics and the SSH Config File – Richard Skumat's Website

Chrinstine November 21, 2016 Reply

bookmarked!!, I really like your site!

Michael March 29, 2017 Reply

I’m using your tip adding this to my nagios’ ssh config

```
Host *
ControlMaster auto
ControlPath ~/.ssh/sockets/%r@%h-%p
```

But unfortunately we’re getting a lot of errors in the form of:

> *Remote command execution failed: ControlSocket /usr/local/icinga/.ssh/sock/ nagios@80.237.155.29:22 already exists, disabling multiplexing*

I tried reproducing it manually and I noticed these kind of errors only appear when I connect to the same host multiple times simultaneously. Did you have those errors as well and how did you get rid of them if so?

As a workaround I used the script from here to start master connections beforehand. But this is not a *pretty* solution I guess.

Cheers,

Micha

tentang judi bola January 23, 2018 Reply

Oh my goodness! Incredible article dude! Many thanks,

However I am encountering issues with your RSS. I don't know the

reason why I can't join it. Is there anybody else having the same RSS issues?

Anyone who knows the answer will you kindly respond? Thanks!!

ChronoFish April 5, 2018 Reply

This is really fantastic. The absolute best resource on SSH I have seen. Thorough and simple to follow.

Thank you so much for spending the time to do this.

Michael April 15, 2019 Reply

When some one searches for his vital thing, so he/she desires to

be available that in detail, so that thing is maintained over here.

LEAVE A REPLY

Your email address will not be published.

Comment

Name	Email	Website

You can use GitHub Flavored Markdown to format your comment.

Pasting code:

```
```

code goes here
```
```

Post Comment