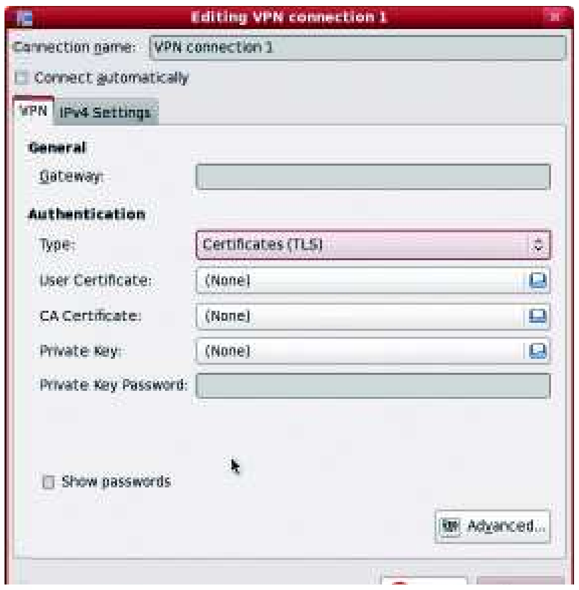
SSH: Get access

Our man on holiday **Ben Martin** tunnels SSH traffic over HTTP to get around over zealous packet filtering at Wi-Fi access points. And you can too!


Editing VPN connection 1

Connection name: VPN connection 1

☐ Connect automatically

VPN | IPv4 Settings

General

Gateway:

Authentication

Type: Certificates (TLS)

User Certificate: (None)

CA Certificate: (None)

Private Key: (None)

Private Key Password:

☐ Show passwords

Advanced...

› **While VPNs have made their way into *NetworkManager* and you can create them easily, if the hotel filters out that sort of network traffic you are out in the cold.**

Our expert

Ben Martin has been working on filesystems for over 10 years. After completing his PhD he now offers consulting services around *libferris*, filesystems and search solutions.

Murphy's law mandates that bad things happen at inconvenient times. It's always when you're out of the country for a weekend that an email arrives informing you that your server needs a little tweak to keep on ticking. That's when you discover that the jokers offering Wi-Fi at your hotel have decided to filter out all internet traffic that is not destined to port 80 (HTTP) or 443 (HTTPS). After all, if you're not just doing vanilla web browsing then it's gotta be malicious and/or nefarious activity right?

With such a filtering setup and no preparation, you're down to either paying extortionate roaming data charges or throwing away hours tracking down an internet cafe just to have that five-minute SSH session you so desperately need.

This tutorial is all about how to tunnel your SSH traffic over port 443 (HTTPS). The 'security conscious' hotel operator will only see HTTPS traffic, and you will be able to have your SSH session without having to pay your telco a pound of flesh or leave your hotel room. Your holiday memories don't need to include lugging your laptop through the streets in hour-long searches for an internet tap that flows with the sweet port 22.

To make things a little easier to explain, I'll assume we're trying to connect to a server in your home. You'll need to have *Apache* installed on the server and access to its configuration files. A cheap web hosting site might not give you enough access to set up the proxy described in the article.

If you're really game, the setup described in the article can bounce you through many HTTP proxies on the way to your SSH server. Sometimes this is actually required, if the internet connection on offer forces you to go through a web proxy (ProxA) then you'll have to first connect to ProxA and then to the web proxy on your home server before finally connecting to the SSH server. Three or more proxies along the way is generally a party trick though, but it can be done!

How it fits together

The information flow for a connection to an SSH server running on a home network firewall machine (PServer) is shown in the diagram *(above-right)*. The client machine could be your laptop. The box in the top-left is the internet connection zone. If the provider of your internet connection forces you to use a web proxy, any traffic wishing to make it to the internet must go through the web proxy server, and consequently any attempt to directly connect to the home SSH server will be dropped before they even hit the internet.

It might be that the web proxy server machine in the diagram is not part of the internet connection being used but that the router connecting the client to the internet simply drops traffic that is not destined to HTTP(S) ports. Once the connection hits your home firewall machine, PServer, it goes to port 80 or 443 at a given URL. After reading this tutorial you will have configured this URL to act as a web proxy server

» **Last month** We cleaned up the web with the Privoxy proxy server.

anywhere

in very specialised conditions and forward the connection to whatever port you are running the SSH server on. A side effect of the setup is that if you are already running a web server that's exposed to the internet you do not have to open any additional ports in order to connect to your SSH server.

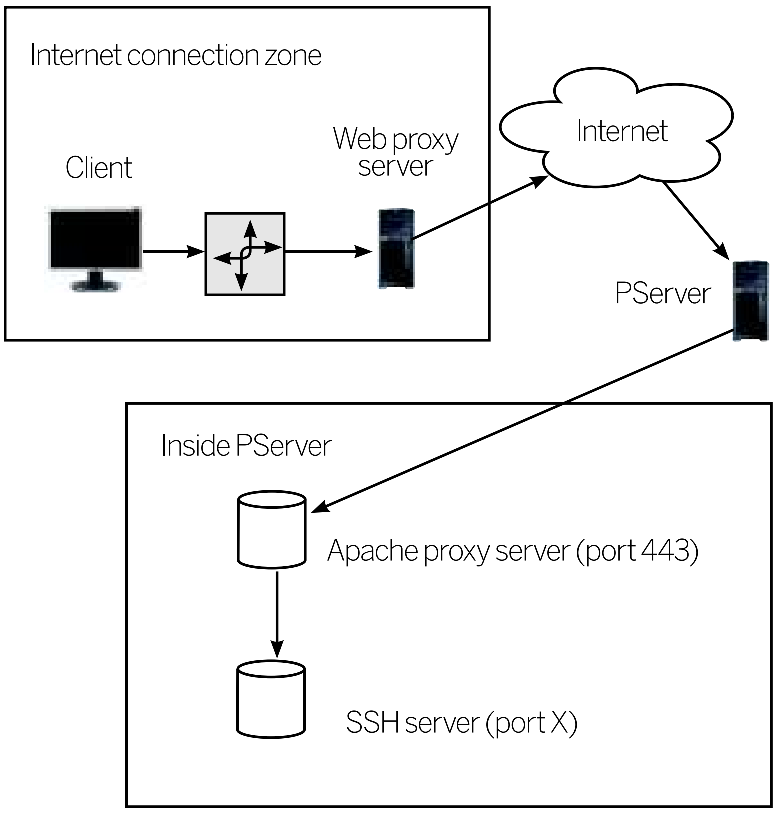
When you're playing around with this stuff you might like to use virtual machines to test that the setup, including with multiple intermediate proxies, is working as you expect. For a two-proxy server setup you'll need at least three virtual machines – two for the proxies and one for the PServer server machine (assuming that you are using your desktop machine as a substitute for the client). Using virtual machines like this also enables you to test the setup before you expose your setup to internet connections. To be more concise I'll call the PServer virtual machine vserver, and the two proxy virtual machines vproxy1 and vproxy2. (The naming is derived from them being virtual machine installations with a 'v' because they are virtual hosts.) Once the setup is working with the vserver virtual machine, the *Apache* configuration can be transferred on to the real PServer machine and the virtual machines are no longer needed. As vserver is a virtual machine version of the real PServer you can think of these two machines as filling the same role throughout the tutorial. The only difference is that vserver is a virtual PServer used for testing the configuration before going live.

Apache setup

The first setup will have an SSH server running on vserver on port 10,000 and will allow connections through an *Apache* server on vserver. The first round will not use any intermediate proxy server, but connect directly to the *Apache* proxy on vserver. Then two web proxy servers will be introduced between the SSH client and the *Apache* server.

The *Apache* setup takes advantage of the **mod_proxy** *Apache* modules. By default on a Fedora Linux *Apache* install these modules are already loaded for you. You will have to have such modules as **mod_proxy_connect** loaded in order for the configuration file to be useful. The configuration shown in **/etc//httpd/conf.d/sshproxy.conf** (Listing 1) can be placed entirely in a new file in the **/etc/httpd/conf.d** directory. The **allowCONNECT** directive is used to limit which ports can potentially be connected to through the *Apache* proxy. **ProxyRequests** turns on forward HTTP proxying and the rest of the configuration file is used to allow only proxying connections to **localhost:10000** and nothing else. With forward HTTP proxying, *Apache* will appear as another HTTP proxy server when we are trying to connect to the SSH server. Originally I tried using two *Proxy* directives for access control. The documentation for the *Apache* Order Directive would lead one to interpret the configuration shown in **apache-bad-acl** (Listing 2) as allowing access to only **localhost:10000**. First the deny is tested, and because a deny and an allow both match, the request is allowed. If you try this configuration it might work for you… for a while. The ordering of the **Proxy** directives in the *Apache* configuration file will alter if the **localhost:10000** access is allowed or denied. This order dependence goes against what is documented for the *Apache* order directive. This sort of random behaviour is obviously not desired.

With only one Proxy directive, the configuration in **/etc/httpd/conf.d/sshproxy.conf** can't change when other parts of the *Apache* configuration are modified. The regular expression will deny anything that does not match the look ahead, so anything not connecting to **localhost:10000** will »



› **How we'll be connecting to our home machine via our *Apache* proxy server.**

Resources

» Apache mod_proxy
http://httpd.apache.org/docs/2.2/mod/mod_proxy.html
» Apache Order Directive
http://httpd.apache.org/docs/2.2/mod/mod_authz_host.html#order
» Tunneling SSH over HTTP(S)
http://dag.wieers.com/howto/ssh-http-tunneling
» ssh-https-tunnel
http://zwitterion.org/software/ssh-https-tunnel

Quick tip

Once you can tunnel your SSH connection, remember that it's easy to use SSH port forwarding to get at other services over the same restricted Wi-Fi network.

be denied. As we are going to have SSH on port 10,000, it is then up to the client to be able to properly authenticate with SSH in order to get a connection. No other proxying is allowed by *Apache*.

An *Apache* web server set up to allow connections to an SSH server running on a user-defined port on localhost like this should have similar security implications to opening the user selected ssh server port to direct internet traffic. There is no extra filtering of attempts to connect to the SSH server port being done by *Apache*. Here's how we allow *Apache* to proxy SSH connections:

```
# Listing 1
# /etc//httpd/conf.d/sshproxy.conf
# Config to allow ssh tunnelling via http.
  AllowCONNECT          10000
  ProxyRequests         on
  <ProxyMatch ^(?!localhost:10000$)>
     Deny from all
  </ProxyMatch>
```

... and here's a really bad example of an access control specification that shows you what not to do:

```
# Listing 2
# BAD configuration, EXAMPLE ONLY
  <Proxy *>
    Order deny,allow
    Deny from all
  </proxy>

  <Proxy localhost:10000>
     Order deny,allow
     Allow from all
  </Proxy>
```

Quick tip

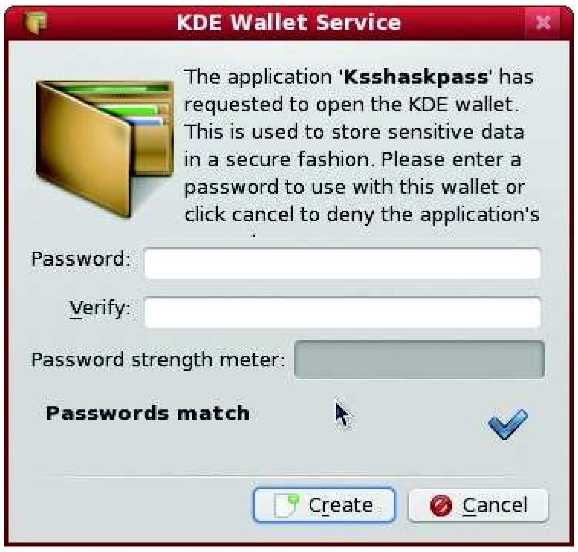
A good final test is to go to a local internet cafe and use *Iptables* to block all outgoing traffic that is not heading for port 80 or 443. Its best to know your safety net works before you make the final fall.

SSH setup

Use the **ProxyCommand** directive to set up the SSH client to use HTTP proxies. It is convenient to set up many Hosts in **~/.ssh/config** to allow easy use of proxies. Introducing many Host directives for the same server enables you to connect to the same server using different proxy options from the command line. For example, you might connect to the host with DNS name **pserver.example.com** using the hostname **pserver**. Connecting with this name might attempt a direct connection to the SSH daemon on **pserver.example.com**. If you feed the **pserver-http** destination hostname to SSH you can indicate that you want a connection to **pserver.example.com** but that SSH should use the current HTTP proxy when connecting. I use the **-hd** (HTTP direct) to connect using the *Apache* web server as the only proxy server. The **hd** hostname is useful for both verifying that the *Apache* web server configuration is working as expected and when you want to connect via the *Apache* web server but your internet provider does not require any other HTTP proxy servers.

Quick tip

Don't forget about the SSH agent, having to type your SSH passphrase all the time can lead to less secure, shorter choices. Use the SSH agent to unlock your SSH key when you unlock your laptop session.


KDE Wallet Service
The application 'Ksshaskpass' has requested to open the KDE wallet. This is used to store sensitive data in a secure fashion. Please enter a password to use with this wallet or click cancel to deny the application's
Password:
Verify:
Password strength meter:
Passwords match
Create
Cancel

› **Unlock your SSH identity files when you unlock your KDE desktop using *KWallet*.**

The **TCPKeepAlive** and **ServerAliveInterval** options are used to keep traffic on the link so that the proxies do not think the connection has stalled and close it. If you are connecting over a link such as through a mobile data service you might want to turn off keepalive messages to save on data traffic.

There are many commands available that can be used with the SSH **ProxyCommand** directive; we're about to look at using **connect-proxy** and a custom Python script for **ProxyCommand**.

It comes in handy to have a single configuration section for a host in your **~/.ssh/config** file specifying the port, DNS hostname, identity file and other options, leaving only the **ProxyCommand** to be defined for each specific connection style. The top half of **~/ssh/config-header** (Listing 3) shows the template with the lower half defining two connection styles that can be used with both vserver and **pserver.example.com**. Using the command **ssh vserver-chd** will connect to the machine with the DNS name vserver through the apache web server running on vserver. I use the **c** prefix to indicate that I'm connecting using the **connect-proxy** command and no prefix for the **http-proxy-tunnel.py** script. The postfix defines how I want to connect: **d** for direct; **hd** for direct using only the *Apache* HTTP proxy on the server itself; **http** for connection over an intermediate HTTP proxy to the *Apache* HTTP proxy and finally to the SSH server; and **https** for the same but using HTTPS instead of HTTP for the intermediate proxies.

Specifying the hostname as **vserver-chd** on the command line will cause SSH to use **connect-proxy** to connect to the HTTP proxy server on vserver and request a connection to the port that SSH is using on the same machine that's running the *Apache* server. Because the configuration of *Apache* shown earlier in **/etc/httpd/conf.d/sshproxy.conf** allows connections to the SSH port on localhost this final link should be allowed.

Here are some aliases and **ProxyCommand** settings for **~/.ssh/config**

```
# /ssh/config-header
```

Port: not just good with Stilton

The internet uses ports to allow a machine to provide multiple services. Common services such as web traffic and SSH connections have fixed ports so that client machines can select which service they want from the many that may be offered by a server. A common network safety tool is to limit the ports that traffic might be allowed to flow between. Sometimes network providers will discard traffic between ports used by common P2P networks, or the ports used by VOIP to enforce network policy. Sometimes port access is tightened too far, and then you need to create a tunnel to your final port.

```
Host *
  TCPKeepAlive yes
  ServerAliveInterval 10
Host vserver*
  HostName          vserver
  User              root
  IdentityFile ~/.ssh/root@local-virtual-machine
  Port 10000
Host pserver.example.com*
  HostName          pserver.example.com
  User              root
  IdentityFile ~/.ssh/root@local-virtual-machine
  Port 10000
Host                *-chd
  ProxyCommand      connect-proxy -H %h:80 localhost %p

Host                *-cd
  ProxyCommand      connect-proxy %h %p
```

With the configuration shown above you can very easily connect with the SSH server through *Apache*. The commands to install the **connect-proxy** command and start a connection through the *Apache* HTTP proxy are shown below:

```
# yum install connect-proxy
# ssh vserver-chd
```

Connecting to the SSH server only through the *Apache* proxy

The main drawback with the **connect-proxy** command is that it allows you to specify only one HTTP proxy server. This means that you can't use **connect-proxy** for SSH when you have to connect to the internet through a proxy server because we already need to use the one allowed proxy server for the *Apache* server on **pserver.example.com**.

http-proxy-tunnel.py

The proxy limitation is removed by using **http-proxy-tunnel.py**, which allows many intermediate proxies to be used. The complete **http-proxy-tunnel.py** is under 300 lines of code but only fragments are shown here.

The command line arguments for **http-proxy-tunnel.py** are a chain of proxies to tunnel through in the form **[proto://]host[:port]** where the first proxy may be specified as "**.**" to use the **http_proxy** environment variable. The proto and port arguments both specify the same thing – which port to connect to on the given host. The last two parameters in **~/ssh/config-prxtun1** (code included on the **LXFDVD**) will have **http-proxy-tunnel.py** connect to our *Apache* server on the HTTP or HTTPS port and then ask to connect to the port SSH is using on the same machine that's running the *Apache* server. Also included on the disc is the http-proxy-tunnel.py script that you'll need to use – see the instructions on the disc to configure your SSH client to use this.

Now that we have set up Apache on PServer (or vserver) as above and configured our SSH client to use **http-proxy-tunnel.py** we should be able to connect directly with the web server on PServer (or vserver) using SSH. The command **ssh vserver-hd** should generate a connection though *Apache*. Running a **tcpdump** while the above command is issued will show traffic similar to this:

```
15:08:59.590325 IP vproxy2.36040 > vserver.http: ...
15:08:59.602934 IP vserver.http > vproxy2.36040: ...
```

The main point is that there is no traffic to or from the SSH service in the packet capture.

Connect through intermediate proxies

If you're testing this using virtual machines for the intermediate proxy servers you might get a fatal error from the proxy script of 'HTTP/1.0 403 Forbidden'. Assuming that the virtual machines are running on the subnet 192.168.1.x, adding these two lines to **/etc/squid/squid.conf** will allow connections through:

```
acl localnet src 192.168.1.0/24
http_access allow localnet
```

With *Squid* is configured like this the commands shown here:

```
[root@vproxy2 .ssh]# export http_proxy=http://vproxy1:3128
[root@vproxy2 .ssh]# ssh vserver-http
```

will cause SSH to connect to the *Squid* proxy server on vproxy1, then to the *Apache* proxy server on vserver and finally to the SSH server on vserver. Assuming that *Apache* is set up on PServer to allow proxying, the **~/.ssh/config** file and identity files used on vproxy2 for this connection can be directly used on a laptop to connect to the home server **pserver.example.com** by changing the **http_proxy** environment variable and connecting to the **pserver**-http host with SSH.

Shown below is a connection through two user-defined web proxies and finally to the *Apache* proxy and SSH server, though most people will not need to use two intermediate web proxy servers before hitting their web server.

```
///[id:two-ud-proxy-serv]///
[root@vproxy2]# vi /etc/squid/squid.conf
... ADD
http_access allow localhost
... SAVE + QUIT
[root@vproxy2]# /etc/init.d/squid restart
[root@vproxy2]# cat /root/.ssh/config
...
Host    *-http2proxies
   ProxyCommand http-proxy-tunnel.py .  "http://
vproxy1:3128" "http://%h" "localhost:%p"
[root@vproxy2]# ssh vserver-http2proxies
Last login: Mon ... from localhost.localdomain
[root@vserver ~]#
```

With a little tinkering in your *Apache* configuration file you can set up an endpoint to allow SSH traffic to be sent over intermediate web proxies and arrive on port 80. Having such a fallback can be a godsend when you really need a connection but can't afford the time to find a less restrictive Wi-Fi hotspot.

Thanks to Russell Stuart for sharing the Python script to tunnel over multiple web proxies and providing the insight to get me started using SSH over web proxies. LXF

Breakout

While there are a plethora of VPN solutions around, many of them use custom protocols and send network packets which are not used for normal web browsing. If you have a fancy IPSec VPN setup, you might find that you can't use that with your hotel's Wi-Fi because they do not support the network traffic required. Tunnelling SSH traffic over HTTP means that the hotel will only see normal network traffic from your laptop, so you're not relying on them having any in-depth knowledge of networking in order for you to connect.

Quick tip

You might also want to pitch in with friends to rent a server in the cloud so that the entire group can use SSH tunnelling through your server with minimal cost.

» **Next month** We'll connect to the cloud with Ubuntu and Amazon's EC2.