ADMIN

Network & Security

Digital Special

Tricks with SSL and SSH

GET STARTED WITH SSH

SSL AUTHENTICATION WITH APACHE

Squid Proxy
Encryption in a proxy server environment

Scripting Tricks
Building OpenSSL into your Bash scripts

VNC in SSH
Remote control in an SSH tunnel

US$ 7.95

WWW.ADMIN-MAGAZINE.COM

Too Swamped to Surf?

Our ADMIN Online website offers additional news and technical articles you won't see in our print magazines.

Subscribe today to our free ADMIN Update newsletter and receive:

- Helpful updates on our best online features
- Timely discounts and special bonuses available only to newsletter readers
- Deep knowledge of the new IT

ADMIN
Network & Security

© Yuri Arcurs, fotolia.com

www.admin-magazine.com/newsletter

ADMIN Network & Security

Tricks with SSL and SSH

Contact Info

Editor in Chief
Joe Casad, jcasad@admin-magazine.com

Managing Editor
Rita L Sooby, rsooby@admin-magazine.com

Localization & Proofreading
Ian Travis
Amber Ankerholz

Layout and Graphic Design
Dena Friesen, Lori White

Advertising
www.admin-magazine.com/Advertise
Ann Jesse, ajesse@admin-magazine.com
Phone: +1-785-841-8834

Corporate Management (Vorstand)
Hermann Plank, hplank@linuxnewmedia.de
Brian Osborn, bosborn@linuxnewmedia.com

Publisher
Brian Osborn, bosborn@linuxnewmedia.com

Marketing Communications
Darrah Buren, dburen@linuxnewmedia.com

Customer Service / Subscription
For USA and Canada:
Email: cs@admin-magazine.com
Phone: 1-866-247-2802
(toll-free from the US and Canada)

www.admin-magazine.com

While every care has been taken in the content of the magazine, the publishers cannot be held responsible for the accuracy of the information contained within it or any consequences arising from the use of it.

Copyright and Trademarks © 2012 Linux New Media Ltd.

No material may be reproduced in any form whatsoever in whole or in part without the written permission of the publishers. It is assumed that all correspondence sent, for example, letters, email, faxes, photographs, articles, drawings, are supplied for publication or license to third parties on a non-exclusive worldwide basis by Linux New Media unless otherwise stated in writing.

All brand or product names are trademarks of their respective owners. Contact us if we haven't credited your copyright; we will always correct any oversight.

Printed in Germany

ADMIN ISSN 2045-0702

ADMIN is published by Linux New Media USA, LLC, 616 Kentucky St, Lawrence, KS 66044, USA.

Dear Readers:

The need for encryption has never been greater. Amateur crackers, professional cyber-thieves, corporate secret seekers, and even international espionage agents are at large right now on the Internet. Toolkits are readily available for cracking passwords and highjacking connections, and your best protection is to conceal the details of your Internet presence so a would-be intruder doesn't discover that one clue that will unravel your careful protections.

SSL and SSH are important tools for keeping your Internet communications confidential. The TLS/SSL protocol system adds encryption to the TCP/IP protocol stack, letting servers and clients operate through an encrypted channel that is immune from conventional eavesdropping. SSH is a protocol – and an accompanying set of utilities – designed to provide a secure shell environment for command-line management tasks across a public network.

This special edition of ADMIN magazine explores some tricks for using SSL and SSH effectively on real-world networks. The emphasis is on advanced techniques for experienced admins. If you are looking for strategies for locking down your network and keeping your communication private, this special edition will give you lots of answers and plenty of ideas for your own network.

Joe Casad
Editor in Chief
ADMIN Magazine

Table of Contents

4 SSL Authentication with Apache
Set up your Apache web server for SSL authentication.

8 Squid Proxy with HTTPS
We'll show you how to support secure Internet sessions in a proxy server environment.

12 OpenSSL with Bash
It is easy to integrate OpenSSL into your Bash scripts. Learn some techniques for building encryption into your automation.

16 Getting Started with SSH
Build a secure tunnel for private communication with the Secure Shell.

20 VNC in SSH
Operate a VNC remote control session through an SSH tunnel.

24 SSH Tricks
Dynamic forwarding, a TUN/TAP tunnel, and other tricks for putting SSH to work in the real world.

30 Perl: Drilling SSH
This handy Perl script that tunnels mail traffic will give you ideas for your own SSH solutions.

SSL authentication with Apache

Close the Page

Website logins with user names and passwords can be dangerous. The Apache web server provides more security with a certificate-based login. *By Florian Effenberger*

Forums, blogs, or content management systems usually have their own authentication systems. But the web server itself also allows access control – almost every user has already seen the `.htaccess` query (Figure 1). Recently, OpenID authentication has become common, in which only one identity is used as a kind of single sign-on to access different pages. One problem these possibilities have in common is that they only require two pieces of data: a user name and a password. Should this information end up in the wrong hands, all doors are open.

Authentication via certificate is a much safer alternative: No password is transmitted at login time; instead a key is stored as a file on the hard drive. This file is in turn protected by a password so that a potential intruder would need not only the key, but also the secret password. And the password can easily be changed at any time. If that is still too risky, you can save the certificate on a USB token or a smart card.

Many users are already familiar with this principle of certificate-based authentication from email encryption, because both PGP and S/MIME use password-protected keys. In most cases, people who manage many servers also use certificate-based logins via SSH that transmit a key instead of a password.

Keys, certificates, certification authority – at first, this all sounds quite complicated, but this approach offers tangible benefits. For

WHAT IS SNI?

SNI, which stands for Server Name Indication, solves a problem with SSL that has existed for years. Previously, Apache was only able to operate exactly one encrypted domain per IP address. With SNI, the browser first sends the address you want in unencrypted form, which allows the use of multiple SSL domains per IP address. And that is extremely useful – especially in times when IPv4 addresses are scarce. Newer versions of Apache (from Ubuntu 10.04 and up), as well as Firefox 4.0, support SNI. Windows Vista or later with Internet Explorer 7 or later supports SNI, but Windows XP is not SNI capable with any IE version.

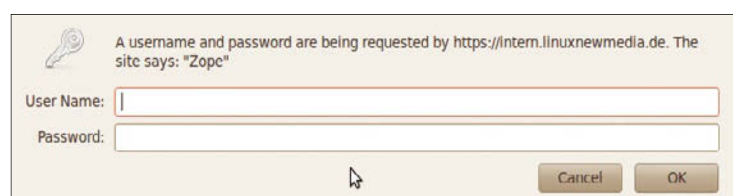


Figure 1: The classic login with .htaccess has one weak point: If the login credentials end up in the wrong hands, all doors are open.

AUTHOR

Florian Effenberger has been a free and open source evangelist for many years. Pro bono, he is on the Board of Directors at The Document Foundation. He was previously active in the OpenOffice.org project for seven years, most recently as Marketing Project Lead. He is also a frequent contributor to a variety of professional magazines worldwide.

one thing, the need for the certificate or access to the smart card represents a much greater obstacle to attackers than just a password prompt. On the other hand, many organizations already issue certificates to their employees anyway. Furthermore, certificates are much more convenient to manage centrally than simple passwords, because by design they already support limited periods of validity and blacklists.

However, this kind of authentication also has disadvantages: A quick login to the server from a colleague's PC won't work, and you can't access important data should the key be lost due to a crash. And this kind of security measure only makes sense if all employees receive their own key. But it is precisely because this method does not allow a login from just any device that it makes a significant contribution to security.

Basic Encryption

The test system on both the client and the server is Ubuntu 10.10. For the web server, we used Apache from the Ubuntu package source; the browser is Firefox. Install Apache as follows

```
# apt-get install apache2-suexec ↩
apache2-mpm-prefork
```

After installation, activate the SSL module at the command line by typing `sudo a2enmod ssl`. To enable SNI (see the box "What is SNI?"), add a `NameVirtualHost *:443` entry to your `/etc/apache2/ports.conf` file under the existing line, and finally restart Apache by entering `sudo/etc/init.d/apache2 reload`.

The web server must first be capable of delivering encrypted pages before it can accept certificates for login. For this reason, first set up a normal SSL-encrypted page and make sure it is working. Only add authentication after the initial SSL configuration has been completed. If the virtual page is missing, remedy that with the following steps:

1. In the folder `/etc/apache2/sites-available/vhost_name.DMN.tld`, create a file with the contents of Listing 1. This describes the virtual host, and will also be used later to configure the authentication. Instead of `vhost_name.dmn.tld`, use the name of the page. If a chained SSL certificate is used, then also add the following line: `be SSLCertificateChainFile/etc/ssl/certs/vhost_name.dmn.tld.chain`
2. Type `mkdir -p /var/www/sites/vhost_name.dmn.tld` to create a directory in which you will be storing the contents of the website.
3. Now activate the new page by typing `a2ensite vhost_name.dmn.tld`, and then restart Apache.
4. Start up a firewall, then open TCP port 443, for example, by typing `sudo ufw allow 443/tcp`.

The SSL certificate for operating the site and the corresponding key are created with the same tools that you also use for certificate management. For a first test, it is perfectly okay to create a certificate manually. Listing 2 shows how to do this. As an alternative – but an impractical one for large companies because of the high costs – you can use an intermediate cer-

SECURITY RISK WITH MULTIPLE VIRTUAL HOSTS

f you also define an unencrypted virtual host for the same address, as seen in Listing 4, make sure it either immediately redirects all requests, or, as in the example, it additionally refers to an empty path. Otherwise, your supposedly protected content can be freely accessed over the normal HTTP protocol, because the `SSLVerifyClient` directive does not apply.

LISTING 1: Creating a Virtual Page

```
<VirtualHost *:443>
ServerName vhost.dmn.tld
DocumentRoot /var/www/sites/vhost.dmn.tld
SSLEngine On
SSLCertificateFile /etc/ssl/certs/vhost.dmn.tld.crt
SSLCertificateKeyFile /etc/ssl/private/vhost.dmn.tld.key
</VirtualHost>
```

DON'T MISS A SINGLE ISSUE!

The first print magazine created specifically for Ubuntu users! Ease into Ubuntu with the helpful Discovery Guide, or advance your skills with in-depth technical articles, HOW-TOs, reviews, tutorials, and much, much more.

SUBSCRIBE NOW!
4 issues per year for only
£ 24.90 / EUR 29.90 / US$ 39.95

✔ Don't miss a single issue!
✔ Huge savings – Save more than 35% off the cover price!
✔ Free DVD – Each issue includes a Free DVD!

shop.linuxnewmedia.com

tificate of your own from your provider.

Now, when you access the website in your browser – if the certificate is not from a known CA (Certificate Authority) – you will see an error message. After you confirm the warning, the address bar of the browser will show a symbol highlighted in blue, indicating that encryption is active.

Admission with Valid Ticket Only

The web server must be told that not only the connection should be encrypted, but that clients should also be authenticated with an SSL certificate. The valid keys do not need to be set individually, but rather all permitted CAs are saved in a file. When a client presents a certificate signed by one of the CAs referred to in the file, Apache grants access.

If you already have a CA and the users' browsers have been configured correctly, you only need to add the lines from Listing 3 to the virtual host in `/etc/apache2/sites-available/vhost_name.dmn.tld` and restart Apache.

Line 1 says that all certificates signed by a CA listed in `/etc/ssl/certs/intranet-ca.crt` are valid for logging in. The directive in Line 2 enforces login by certificate with (`require`). Please note: Without this statement, no authentication will take place. Line 3 allows five intermediate certificates, which is particularly important for large CAs. If all certificates come directly from the CA without intermediate CAs, then enter 1 here.

Depending on the browser, authentication takes place automatically after entering the master password or after selecting the corresponding certificate. In the case of Firefox, be sure that *Ask me every time* is enabled under *Edit* | *Preferences* | *Advanced* | *Encryption* (Figure 2). This will help discover errors, at least during installation on the server. After configuring the server and adding the certificate to the browser, the browser will prompt you for it the next time you visit the page.

This technique allows password protection with `.htaccess` for individual directories or addresses. For example, in order to protect only the Wiki website, configure the virtual host as shown in Listing 4. Listing 4 also shows how you can automatically redirect all unencrypted calls with `mod_rewrite`. Caution: There is a security risk involved here that is described in the "Security Risk with Multiple Virtual Hosts" box. But you first need to activate `mod_rewrite` by typing `a2enmod rewrite`. By the way, it is not necessary to use mod_rewrite; it only saves users from having to type in the HTTPS protocol by hand.

Conclusions

Configuring authentication by certificate might take some effort, but it offers significant security benefits compared with a password-based login procedure. If you decide to set up your own certificate authority, you should plan carefully. In production operations, be sure to add a certificate revocation list (CRL). Furthermore, to protect yourself against compromise, consider the use of intermediate CAs. A good overview of the server side is provided by Apache's own documentation ([1], [2]).

LISTING 2: Creating a Certificate Manually

```
$ openssl req -new > vhost.dmn.tld.csr -newkey
  rsa:2048 -keyout vhost.dmn.tld.pem
$ openssl rsa -in vhost.dmn.tld.pem -out
  /etc/ssl/private/vhost.dmn.tld.key
$ openssl x509 -in vhost.dmn.tld.csr -out
  /etc/ssl/certs/vhost.dmn.tld.crt -req -signkey \
  /etc/ssl/private/vhost.dmn.tld.key -days 3650
```

LISTING 3: Forcing SSL Authentication

```
01 SSLCACertificateFile /etc/ssl/certs/intranet-ca.crt
02 SSLVerifyClient require
03 SSLVerifyDepth 5
```

Figure 2: Especially while setting up the server, it is helpful to have Firefox prompt for the certificate every time.

INFO

[1] Apache documentation on mod_ssl: *http://httpd.apache.org/docs/current/mod/mod_ssl.html*

[2] Apache SSL-Howto: *http://httpd.apache.org/docs/current/ssl/ssl_howto.html*

LISTING 4: Handling an Unencrypted Virtual Server

```
<VirtualHost *:80>
ServerName vhost.dmn.tld
DocumentRoot /var/www/sites/vhost.dmn.tld-80
RewriteEngine on
RewriteRule ^(.*) https://%{SERVER_NAME}$1 [NE,L]
</VirtualHost>

<VirtualHost *:443>
ServerName vhost.dmn.tld
DocumentRoot /var/www/sites/vhost.dmn.tld
SSLEngine On
SSLCertificateFile /etc/ssl/certs/vhost.dmn.tld.crt
SSLCertificateKeyFile /etc/ssl/private/vhost.dmn.tld.key
<Location /wiki>
SSLCACertificateFile /etc/ssl/certs/intranet-ca.crt
SSLVerifyClient require
SSLVerifyDepth 5
</Location>
</VirtualHost>
```

Shop the Shop

shop.linuxnewmedia.com

Want to subscribe?

Need training?

Searching for that back issue you really wish you'd picked up at the newsstand?

Discover the past and invest in a new year of IT solutions at Linux New Media's online store.

shop.linuxnewmedia.com

DIGITAL & PRINT SUBSCRIPTIONS

SPECIAL EDITIONS

TRAINING

- LPIC-1 LPI 101 - CompTIA Linux+ LX0-101
- LPIC-1 LPI 102 - CompTIA Linux+ LX0-102
- LPIC-1 - CompTIA Linux+ 101 + 102

Using a Squid proxy with HTTPS

Squid in the Middle

How do you monitor the network when your client systems are connecting to secure web servers through HTTPS? We'll show you how to keep watch using the Squid proxy server and share some inventive certificate tricks. *By Kurt Seifried*

Network- and host-based intrusion detection is pretty much a mandatory requirement now if you want to keep your network under control. Back in the good old days, when your Internet connection was a dial-up link (for the entire company), you could just keep software up to date, install a firewall, and call it a day.

Since then, things have changed significantly. Almost all computers are now attached to the Internet all the time. Most of these computers are behind firewalls and NAT-based systems – so they can use the Internet, but the Internet can't initiate connections to them.

This strategy worked pretty well until clients started using prodigious amounts of data from the Internet, especially the World Wide Web and email. Now, to add insult to injury, almost all web and email

Mark J. Grenier, Fotolia.com

clients include JavaScript support and newer technologies like HTML5 and web sockets.

To put it simply, implementing all the best practices in the world cannot guarantee that no one will break into your network. So, from a defender's point of view, you need to maintain a close watch on systems and the network for anomalous behavior (e.g., sending a terabyte of data to a country you don't have an office in).

Back in the day, tracking web traffic was trivial. You could just install a proxy server, block outgoing access (except for the proxy server), and you were done. You could see all outgoing requests and incoming responses; everything was in cleartext and could be trivially logged, searched, and stored for later use. But now, with more and more sensitive information being accessed and sent through websites, many sites are deploying SSL, and some larger sites (notably, Google properties like Gmail and Red Hat OpenShift) are defaulting to SSL for all sites and traffic.

Intercepting SSL and TLS

The good news (or bad, depending on your perspective) is that HTTPS traffic can be intercepted transparently by proxy software such as Squid. This is accomplished by executing what is essentially a man-in-the-middle attack. The proxy system opens an HTTPS connection with the outside web server and passes the web data back to the client as if it were coming directly from the server. Of course, the client is operating through HTTPS, and the proxy system must complete that connection.

The trick to pulling off this setup is to use a root certificate that is trusted by the client to sign site certificates, thereby allowing you to create fake signed certificates for sites the user is visiting. This process is possible because client software (your web browser, email client, etc.) trusts all root certificates equally. When you access a site, such as example.org, you currently have no way of knowing which certificate authority example.org used to sign the site certificate.

To set up an intercepting SSL proxy involves basically two steps. First, you need to configure a proxy such as Squid, and second, you need either to get a certificate that is not restricted from signing other certificates or to install your own root certificate on clients for which you have the private key, which you can then use to sign fake certificates. This process, of course, leads to some ethical and potential legal issues; I think the Squid site says it best:

> *Decrypting HTTPS tunnels without user consent or knowledge may violate ethical norms and may be illegal in your jurisdiction. Squid decryption features described here and elsewhere are designed for deployment with user consent or, at the very least, in environments where decryption without consent is legal.* [3]

The good news (again, depending on your perspective) is that various technologies and strategies are being created to solve this exact problem, such as Google SSL Pinning and Convergence SSL [1].

Configuring Squid to Proxy SSL

Configuring Squid for transparent SSL interception is not difficult. However, most vendor-supplied Squid packages do not include support for transparent SSL interception out of the box. This means you will either need to compile Squid from source or download a source RPM or DPKG, modify the configuration, and recompile it. Note that currently on Fedora, Red Hat Enterprise, and derivatives like CentOS, the OpenSSL package has an issue that prevents Squid from being compiled with SSL interception. Short of installing a second copy of OpenSSL from source code and compiling against that, I was unable to find a good solution. Basically, when compiling Squid, you will need to make sure the `--enable-ssl-crtd` and `--enable-ssl` options are enabled for SSL interception.

Next, you will need to configure three main options: dynamic certificate generation, SSL bumping, and the HTTPS support for clients (assuming you are not using transparent interception). For all three of these configuration setups, I highly recommend checking the Squid wiki for the latest information.

Dynamic certificate [2] generation is required because you will need to create signed SSL certificates for each site visited by users. You can do this through the `ssl_crtd` program; the main configuration option to keep in mind is the `dynamic_cert_mem_cache_size`. Setting a larger cache (I recommend 10-20MB) ensures that certificates are not constantly being regenerated (100 certificates require approximately 4MB), thereby reducing load times.

The next step is to enable SSL bumping [3]. Squid can easily be configured to proxy SSL connections using the `CONNECT` method. In this mode, it simply passes packets between the server and the client; it does not decrypt or otherwise understand the traffic being passed back and forth. To decrypt the data, you will need to enable SSL bumping, which mainly consists of adding `ssl-bump` to the `http_port` and `https_port` configuration lines.

Finally, if you're proxying HTTPS, it would probably be a good idea to talk to the proxy using HTTPS [4]; otherwise, local attackers might be able to view all the traffic. To enable HTTPS, simply use `https_port` instead of `http_port`. You should have something like

```
https_port 3130 ssl-bump ⮐
  generate-host-certificates=on ⮐
  cert=/etc/squid/myCA.pem ⮐
  key=/etc/squid/myCA.key
always_direct allow all
ssl_bump allow all
```

once you put it all together.

Squid SSL Transparent Proxy

So, now you have an HTTPS-enabled proxy that can handle SSL connections and examine them. However, you probably want to set this up transparently to avoid having to change the proxy settings on every single device – especially mobile devices that are only attached to your network for a few hours a day (and attached to other networks like coffee shops and hotels).

To do this, you can simply add the "transparent" keyword and define NAT redirection rules like this,

```
iptables -t nat -A PREROUTING -i ⮐
  eth0 -p tcp -dport 80 -j DNAT ⮐
  -to-destination 10.1.2.3:3128
iptables -t nat -A PREROUTING -i ⮐
  eth0 -p tcp -dport 443 -j DNAT ⮐
  -to-destination 10.1.2.3:3129
```

HARDWARE AND CELL PHONES

If you haven't heard about the Raspberry Pi [6] and similar computers, now would be a good time to learn about them. Basically, for US$ 25 to US$ 35, you can get a fully functioning ARM-based computer about the size of a credit card with Ethernet, USB, and HDMI output that runs on five watts or so. How is this related to intrusion detection? Attackers can now build small cheap computers that either run off of a battery or are small enough to fit in other equipment like power bars or UPSs. You can even buy a device called a "Jack PC," which is literally a full computer that fits into a standard-sized wall jack for less than US$ 300.

The problem here is that if a shipment of 10 or 20 UPSs shows up at your office, people will probably plug them in. Then, an attacker can connect to them remotely via cell phone or WiFi and attack your wireless networks. If the attacker ships an extra Ethernet cable and instructions to place the UPS inline with the wired network to prevent surges from frying the computer, I bet most users would do so. And, with this, an attacker would be able to attack your wired network. On September 30, 2012, the Power Pwn [7] penetration testing platform is expected to be commercially available with all these and other features (Figure 1), so inventorying your equipment and making sure nothing strange has sneaked in will become a new standard task for administrators – unfortunately.

Figure 1: Power Pwn.

so that outgoing traffic to port 443 is sent to the Squid transparent proxy.

Installing Certificates on Clients

Recently, a number of certificate authorities have been found selling certificates that can be used to sign arbitrary sites. These certificates are used by various SSL-intercepting proxy devices that can be purchased commercially. Chances are you won't be able to buy one of these certificates (and if you can, then you can pretty much perform a man-in-the-middle attack on any public website); however, you can generate your own root certificate and install it on client systems to accomplish the same thing. Your main options are to export the public part of the certificate and manually install it or to have users manually install it on their system. You'll likely have to do this anyway with mobile devices (see also the "Hardware and Cell Phones" box).

Your other options are to use command-line tools to insert the certificate into the certificate storage mechanism on a user's system or to modify and create customized packages for the system certificate store. Programs like Firefox and Thunderbird typically use a central certificate store in `/etc/pki/nssdb/cert8.db`; however, this is not always the case.

The programs also keep local certificate stores in `$HOME/.mozilla/firefox/[random].default/cert8.db` and `$HOME/.mozilla/thunderbird/[random].default/cert8.db`, for example. To interact with these certificate stores, you'll need the nss-tools [5] software, which is typically installed along with programs like Firefox or Thunderbird. The command line to do this should look like this:

```
certutil -A -n "$certname" 
  -t "TCu,Cuw,Tuw" 
  -i $certfile 
  -d $certdir
```

Google Chrome uses the same format of NSS files, so adding certificates to it is the same as above. However, the certificates could be stored in a different location, such as `$HOME/.pki/`.

Risks of Interception

Of course, you incur some risks when intercepting SSL connections. The most significant one being that users cannot see the real site certificate, so they cannot make an informed decision on whether or not to trust it. Instead, they must rely on the intercepting proxy to handle certificates properly. This means that problems like expired certificates, or certificates that suddenly change, might not be detected correctly.

At this time, Squid does not support SNI (essentially virtual hosting multiple SSL sites off a single IP address) very well, which might break a number of popular sites. Unfortunately, if you do not monitor SSL-based web traffic, you will see less and less as more sites move to the use of HTTPS by default, leaving no easy answers for this problem. Additionally, technologies such as Google SSL pinning will detect such interception and, depending on the configuration, possibly block users from using the site at all.

Conclusion

Ironically, what's making network intrusion detection more difficult is the simple fact that we are getting much better at security and are widely deploying things like SSL encryption. Tricks like the Squid HTTPS proxy technique described in this article will help you maintain a consistent level of oversight in the age of secure Internet connections. ■■■

INFO

[1] "Who's on First?" by Kurt Seifried, *Linux Magazine*, April 2012, pg. 70: *http://www.linuxpromagazine.com/Issues/2012/137/Security-Lessons-Fixing-SSL/(kategorie)/0*

[2] Dynamic SSL certificate generation: *http://wiki.squid-cache.org/Features/DynamicSslCert*

[3] Squid-in-the-middle SSL Bump: *http://wiki.squid-cache.org/Features/SslBump*

[4] HTTPS (HTTP Secure or HTTP over SSL/TLS) *http://wiki.squid-cache.org/Features/HTTPS*

[5] Using the certificate database tool: *http://www.mozilla.org/projects/security/pki/nss/tools/certutil.html*

[6] Raspberry Pi: *http://www.raspberrypi.org/*

[7] Power Pwn: *http://pwnieexpress.com/products/power-pwn*

LINUX UPDATE

Need more Linux? Our free Linux Update newsletter delivers insightful articles and tech tips to your mailbox twice a month. You'll discover:

- **Original articles on real-world Linux**
- **Linux news**
- **Tips on Bash scripting and other advanced techniques**
- **Discounts and special offers available only to newsletter subscribers**

LINUX UPDATE
EXPLORING THE WORLD OF LINUX

- Bash Tips: Autocompletion
- Cloud Storage Behind the Firewall
- Ubuntu for Android
- Dress Up Bash Scripts with YAD
- Cotton Candy: Tiny Cloud Computer Available for Pre-Order

FEATURED ARTICLES

Bash Tips: Autocompletion
Steer around errors and save yourself some typing by adding autocompletion to your Bash scripts. (more)

Cloud Storage Behind the Firewall
ownCloud's new commercial venture lets you manage risk and data exposure in a bring-your-own-device age. (more)

Ubuntu for Android
Canonical announces Ubuntu for Android. (more)

Dress Up Bash Scripts with YAD
If you want to add a dash of GUI goodness to your Bash scripts, you have several options. You can use Zenity or Kdialog to quickly add simple dialogs and message boxes to Bash scripts. However,... (more)

Cotton Candy: Tiny Cloud Computer Available for Pre-Order
FXI Technologies Inc. launches a community website and technical forum, announcing pre-order availability of the 21-gram Cotton Candy devices. (more)

FURTHER READING

- Hidden Meaning: Working with Microformats
- TinyDNS
- Fedora 16
- Automating Cloud Services with Open Source Tools
- Regional Failure in the Cloud

Managing Superuser Access
Privileged accounts pose a security risk for any network. Click here for some expert techniques for managing, monitoring, and auditing access to the superuser account.

25% Off On Digital Subscriptions
Want to start a Linux Magazine digital subscription or renew your existing one? Act now and save 25% on our normal price. You'll get 12 issues (one year) in PDF format. These files are fully searchable, and you have the advantage of being able to read your magazine anywhere, at any time!

Already a print subscriber of the magazine? Get a free digital subscription add-on for the same issues as your print subscription.

MOST READ

Qemu and the Kernel
Debugging the kernel of a running operating system has always been tricky, but now the Qemu emulator

www.linuxpromagazine.com/Subscribe/iframe-Newsletter

Ft Photography, Fotolia

Using the OpenSSL toolkit with Bash

Cryptic

Michal Boubin, 123rf

Cryptography is an important part of IT security, and OpenSSL is a well-known cryptography toolkit for Linux. Experts depend on OpenSSL because it is free, it has huge capabilities, and it's easy to use in Bash scripts. *By Marcin Teodorczyk*

OpenSSL[1] makes use of standard input and standard output, and it supports a wide range of parameters, such as command-line switches, environment variables, named pipes, file descriptors, and files. You can take advantage of these features to quickly write Bash [2] (Bourne-Again Shell) scripts that automate tasks, such as testing SSL/TLS (Secure Socket Layer/Transport Layer Security) connections, bulk conversions between different formats of cryptographic keys and certificates, batch signing/encrypting of files, auditing password protected files, and implementing or testing a PKI (Public Key Infrastructure).

The OpenSSL toolkit provides many modules that each perform a specific task. Each module is not a separate executable, but is, instead, selected with the first parameter of the `openssl` executable. On the other hand, each module has a separate manual page. For example, a module named `x509` manages X.509 digital certificates and a module named `pkcs12` manages PKCS12 packages.

To use `x509`, you should execute the following command:

```
openssl x509 -param1 param1value
```

but to see the manual page for it, you should type: `man x509`.

Testing SSL/TLS connections

OpenSSL provides three modules that allow you to test SSL connections: `s_client`, `s_server`, and `s_time`. The first two, as the names suggest, are for simulating a client and a server in an SSL connection. The third one is for connection timing tests. I'll start with a closer look at the `s_client` module.

`s_client` is particularly useful for checking which protocols and which ciphers the server agrees to use. This information is useful in security and functionality audits. For example, you could use this protocol information to find servers that don't accept a legitimate protocol or cipher, thus preventing a legitimate client from connecting. You could also locate servers that accept weak protocols or ciphers and could thus allow a malicious attack. With a little help from Bash, you can fully automate this process.

Assume the client and the server hostnames are `client` and `server`, and that the server listens for SSL/TLS connections on port 443.

To check which protocols `server` accepts, you could use the following parameters: `-ssl2`, `-ssl3`, `-tls1`, `-no_ssl2`, `-no_ssl3`, or `-no_tls1`.

Because SSL2 is known to have security weaknesses, you can attempt to connect to the server using the following command:

AUTHOR

Marcin Teodorczyk has been passionate about computers and Linux for more than 14 years. He works with grid environments as an Information Security Officer, and in his spare time, he juggles.

```
openssl s_client ↩
  -connect server:443 -no_ssl3 -no_tls1
```

If the server accepts any protocol other than SSL3 or TLS1, the preceding command opens a connection and waits for data. (Of course, this approach is not ideal if you plan to embed the command in a Bash script.) To close the connection immediately after establishing it, write to `s_client`'s standard input:

```
echo "x" | openssl s_client ↩
  -connect server:443 -no_ssl3 -no_tls1
```

Similarly, you can check allowed ciphers with the `-cipher` parameter. For user convenience, OpenSSL allows you to specify specific cipher suites (e.g., `DES-CBC3-SHA`) or groups of ciphers (e.g., `LOW`, `MEDIUM`, `HIGH`, `NULL`, `ALL`). Find group names and ciphers with `man ciphers`.

To check whether the server accepts connections using ciphers from group `NULL` or `LOW`, use the following:

```
echo "x" | openssl s_client ↩
  -connect $server:443 -cipher NULL,LOW
```

In Bash scripts, it is a good idea to run OpenSSL modules with a specified timeout. Otherwise, when a hostname can't be resolved, the script will hang for a long time. A special Linux utility will let you run any command with a timeout. Surprisingly, the utility is called `timeout`. For example, to check if an SSL2 connection can be established, but not wait for it longer than 10 seconds, use:

```
echo "x" | timeout 10 openssl ↩
  s_client -connect server:443 -ssl2
```

Finally, to make the command more automatic, you can use the `$?` variable to check the return code of the last command executed by BASH. If the connection is established, OpenSSL returns `0`.

Listing 1 shows a simple example script with everything I have done so far. The script reads hostnames from standard input and checks if a connection other than SSL3 or TLS1 can be established on port 443. It waits no more than three seconds. Names of hosts that allow such connections are written to the file `bad_protocol.txt`. Similarly, the hosts that allow connections with NULL or LOW ciphers are listed in `bad_cipher.txt`.

Handling PEM/DER and PKCS12 Formats

A few formats and containers are used for public cryptography keypairs and digital certificates. Without getting into details, the most common formats for my network are PEM, DER, PKCS12, or JKS. From these, only the JKS format is not supported by the OpenSSL software. PEM and DER are encoding formats – PEM is a Base64-encoded format. DER is binary. PKCS12 is a container that can hold private and public keys, as well as signed certificates and certificates chains.

To convert between the PEM and DER file formats, you can use the `-inform` and `-outform` parameters. For example, to convert all X.509 certificates from PEM to DER, you can use the following loop:

```
for file in *.pem;
do openssl x509 -inform PEM ↩
  -in $file -outform DER -out $file.der;
done
```

Another common task is extracting keys/certificates from a PKCS12 package, which is usually protected with a password. I can handle such an operation with the Bash and OpenSSL option `-passin`. This option allows me to specify passwords to access data in password-protected files in five ways. It is useful not only for PKCS12, but for every action that requires a password, for example, encrypted private keys or data.

First, I can specify a password as a `pass:password_text`, in which case `password_text` is the actual password. This is not a secure method, because the password is stored in Bash history and can be spotted with a `ps` command during execution. Second, I can specify the password with `env:var`. This method is more secure, because the password is held in the environment variable `var`. Another approach is to store the password as a `file:pathname`, which instructs OpenSSL to read the password from the first line of the file `pathname`. Or, I could use `fd:number`, which makes OpenSSL read the password from the file descriptor `number`. Finally, I can simply use `stdin` to read passwords from standard input.

Next, I'll extract all certificates from password-protected PKCS12 files in a working directory and store them without password protection. This can be done with the following:

```
for file in *.p12; do
    openssl pkcs12 -in $file ↩
    -passin file:$file.pass ↩
    -nokeys -nodes -out $file.nokeys
done
```

I will assume I have a password for each PKCS12 file written in a file with the `.pass` extension.

Bulk Encrypting and Decrypting

Common cryptography tasks include encrypting and decrypting files. Symmetric cryptography uses one key for encrypting and decrypting. Asymmetric cryptography uses a public key for encrypting and a private key for decrypting (typically implemented with PKI and X.509 certificates).

Symmetric cryptography is faster than asymmetric cryptography, and it is a better choice when there is no need to provide public access to the key.

To encrypt the `plain.txt` file with symmetric cryptography and write the output `cipher.enc`, I can use the following command:

LISTING 1: Checking Permitted Protocols

```
01 #!/bin/bash
02 while read server ; do
03     timeout 3 openssl s_client -connect $server:443 -no_ssl3 -no_tls1
04     if [ $? -eq 0 ] ; then
05         echo $server >> bad_protocol.txt
06     fi
07     timeout 3 openssl s_client -connect $server:443 -cipher NULL,LOW
08     if [ $? -eq 0 ] ; then
09         echo $server >> bad_cipher.txt
10     fi
11 done
```

```
openssl [ciphername] -a -salt ↩
  -in plain.txt -out cipher.enc
```

The system will prompt for an encryption password, which also has to be typed when decrypting later. It is not the best option for bulk operations, but I have already described several methods for specifying a password to OpenSSL.

Thus, to encrypt all `.txt` files in the current directory and write them to the `../enc` directory with the `aes-256-cbc` cipher, I can use the following loop (assuming the password is written in the `pass` file):

```
for file in *.txt; do
    openssl aes-256-cbc -a -salt ↩
    -in "$file" -out "../enc/$file" ↩
    -passin file:pass
done
```

I can decrypt all `.txt` files in the current directory and write them to the `../dec` directory with:

```
for file in *.txt; do
    openssl aes-256-cbc -d -a -salt ↩
    -in "$file" -out "../dec/$file" ↩
    -passin file:pass
done
```

again assuming that I have a password in the `pass` file.

OpenSSL and Standard Input/Output

Keeping with Unix philosophy, every argument that is passed with the `-in` parameter can be passed also using standard input. If I don't specify the output with the `-out` parameter, it is written to standard output. Hence, I can use OpenSSL for processing outputs of other commands and generating inputs for other programs with a pipe. To check whether a certificate with the serial

LISTING 2: Testing a PKI

```
01 function create_config {
02     {
03     echo "HOME              = ."
04     echo "RANDFILE          = $ENV::HOME/.rnd"
05
06     ...
07     cut for better readability
08     ...
09
10     echo "oid_section       = new_oids"
11     echo 'subjectKeyIdentifier=hash'
12     echo 'authorityKeyIdentifier=keyid,issuer'
13     echo 'proxyCertInfo=critical,language:
             id-ppl-anyLanguage,pathlen:3,policy:foo'
14     } > "$config"
15 }
16
17 function create_root_ca {
18     local keysize=$1
19     local country=$2
20     local org=$3
21     local name=$4
22     local days=$5
23     local certfile=$6
24     local keyfile=$7
25     openssl req -newkey rsa:$keysize -x509
                   -days $days -keyout $keyfile -nodes
                   -out $certfile -config $config
                   -subj /C=$country/O=$org/CN=$name
26     return $?
27 }
28
29 function create_crl {
30     local cakey=$1
31     local cacert=$2
32     local crlfile=$3
33     openssl ca -gencrl -config $config -keyfile $cakey
                  -cert $cacert -out $crlfile
34 }
35
36 function create_client_req {
37     local keysize=$1
38     local country=$2
39     local org=$3
40     local name=$4
41     local keyfile=$5
42     local reqfile=$6
43     openssl req -new -newkey rsa:$keysize -nodes
                   -keyout $keyfile -out $reqfile
                   -config $config
                   -subj /C=$country/O=$org/CN=$name
44 }
45
46 function sign_client_req {
47     local clientreq=$1
48     local days=$2
49     local cacert=$3
50     local cakey=$4
51     local clientcert=$5
52     openssl x509 -req -days $days -CA $cacert
                    -CAkey $cakey -CAcreateserial
                    -in $clientreq -out $clientcert
53 }
54
55 function revoke_client_cert {
56     local clientcert=$1
57     local cakey=$2
58     local cacert=$3
59     openssl ca -revoke $clientcert -keyfile $cakey
                  -cert $cacert -config $config
60 }
61
62 function get_cacountry {
63     cacountry="DC"
64 }
65
66 function get_caorg {
67     caorg="Dummy org"
68 }
69
70 function get_caname {
71     caname="Dummy CA"
72 }
```

number 44A2FC741D8C1755 has been revoked, I can use the following command:

```
curl -s http://localhost/crl.pem | ↩
    openssl crl -text -noout | grep ↩
    "Serial Number: 44A2FC741D8C1755"
```

This command will retrieve a CRL (Certificate Revocation List) and decode it with OpenSSL. Next, it will `grep` for a serial number. Similarly, as with the previous Bash scripts, I can add a `timeout` and check the output of the `grep` command with the `$?` variable.

Auditing Encryption Passwords

A private key should almost always be secured with a password. Often, PKCS12 files are secured with passwords, too. With OpenSSL and Bash, I can do a quick check of the passwords used to protect those files. Assume I have a text file with the most common passwords, one in a line, called `passwords.txt`. I can check each password-protected file in the current directory with:

```
while read pass; do
  for file in *.p12; do
    openssl pkcs12 -in $file ↩
    -noout -passin pass:$pass ↩
    2>/dev/null
    if [ $? -eq 0 ] ; then
        echo "Guessed password ↩
              for $file: $pass"
    fi
done
done < passwords.txt
```

using each password from the `passwords.txt` file.

Testing PKI

The last major capability of OpenSSL is implementing a Public Key Infrastructure (PKI). A PKI often has a crucial role in security, and OpenSSL can be used to implement and test a PKI.

First, I can use OpenSSL to generate key pairs and corresponding CSRs (Certificate Signing Request). Second, I can sign CSRs, thus creating a valid certificates. Third, I can revoke and generate CRLs. Fourth, I can sign/encrypt and verify/decrypt. Finally, I can change parameters on the fly, manipulating values such as algorithms, key lengths, or DN content. I can then use this data as input for other applications.

Listing 2 shows a few example functions I can use for testing various elements of a PKI. The `create_config` function has been cut for better readability. You can use the contents of your default OpenSSL configuration file for additional configuration settings. The configuration file is usually called `openssl.cnf` and placed in `/etc`.

By default, OpenSSL reads its configuration file from a specified location (usually `/etc/openssl.cnf`), but for my purposes, it is easier to create a config file on the fly. The script function create_config takes care of this by writing the configuration to the `./config` file. Later, the file created by this function is pointed to OpenSSL with the `-config` parameter.

Next I have functions `create_root_ca`, `create_crl`, `create_client_req`, `sign_client_req`, and `revoke_client_cert`, the names of which are self-explanatory. All of these functions take parameters that specify things such as a DN (Distinguished Name) string, valid period, keysize, etc.

The main part of the script (not shown in the listing) could use the functions to generate a specified number of CA's (Certification Authority) certificates and a specified number of client's certificates for each CA. Also, I could revoke some client's certificates right after generating. Thus the output of the script would be a bunch of CA certificates, revoked client certificates, and CRLs.

Summary

OpenSSL is a very flexible tool. Because you can specify all the necessary parameters using command-line switches, files, pipes, and environment variables, it is perfectly suited for Bash scripts.

This article described a few uses for OpenSSL, but bear in mind that this is only the tip of an iceberg. I encourage you to glance through the manual and experiment with your own ideas. Just don't confuse somebody else's private key with your own. ■■■

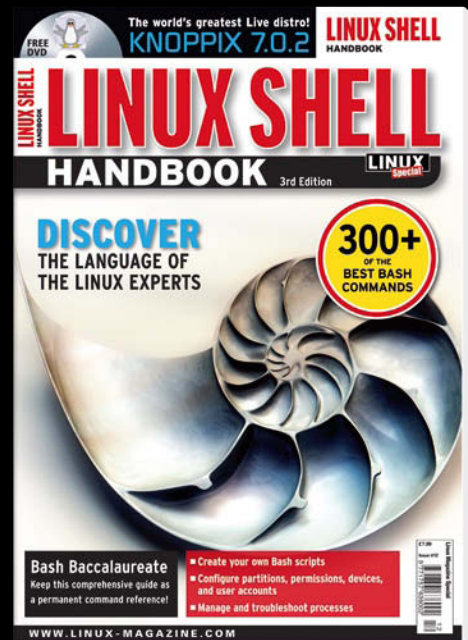
INFO

[1] OpenSSL: *http://www.openssl.org/*

[2] Bash: *http://www.gnu.org/software/bash/*

300+ of the best BASH Commands!

- Create your own Bash Scripts
- Configure partitions, permissions, devices, and user accounts
- Manage and troubleshoot processes



AVAILABLE AT YOUR NEWSSTAND!

OR ORDER ONLINE AT:
SHOP.LINUXNEWMEDIA.COM
(SELECT SPECIAL EDITIONS)

3rd edition
New and improved!

Secure connections with SSH

Tunnel Builder

Miroslaw Hejnicki, 123rf.com

Whether you need an encrypted tunnel between multiple PCs or graphical applications over a LAN, the all-purpose SSH tool leaves little to be desired. *By Joerg Harmuth*

Telnet is probably the best known solution for providing users with console access to remote machines. However convenient this dinosaur of network communication might be, it has one major disadvantage: All the data are sent in plaintext over the wire. If an attacker sniffs the connection, he or she will quickly learn the administrative password for the server. Admittedly, it probably isn't quite that easy, but the danger is there all the same. For this reason, all popular Linux distributions install the Secure Shell (SSH) as a safer alternative.

SSH's configuration files are located in `/etc/ssh`, where you will find one file for the server (`sshd_config`) and another for the client (`ssh_config`). The files contain a huge number of options, which are explained in detail in the man pages. Users don't typically need to make any major changes. The defaults used by openSUSE 11.0 are user friendly but still secure enough not to make additional configuration worthwhile.

SSH PROTOCOL VERSIONS

SSH1 and SSH2 are the current versions. SSH1 uses the insecure DES or the secure Triple DES (3DES). The Blowfish algorithm provides a fast and – so far – secure encryption technology. Version 2 includes the AES algorithm and others.

Vulnerabilities in the SSH1 protocol make it possible to hack the encryption. Version 1 relies on encryption of data with a random number that has been encrypted with the server's public key. This method is open to brute force attacks that give the attacker the plaintext key.

Protocol 2 relies on a Diffie-Hellman exchange that never transmits the key over the wire but allow server and client to generate the same key independently.

Other enhancements to version 2 include the software's ability to check the data integrity with cryptographic hashes (the Message Authentication Code method) rather than the unreliable CRC (Cyclic Redundancy Check) method. Support for multiplexing is also improved. All of the examples in this article use SSH2, although some will work with SSH1.

How It Works

The SSH client/server architecture is based on TCP/IP. The SSH server (`sshd`) runs on one machine, where it listens for incoming connections on TCP port 22. The client simply uses this port to connect to the server. When a connection is established, quite a few things happen in the background. First, the server and client negotiate the SSH protocol version to use for the communications. Currently, SSH 1 and SSH 2 are available, but SSH 2 is standard today because of its better security. Details – including details of encryption – are given in the "SSH Protocol Versions" box.

Second, the server and client negotiate the algorithm, followed by the key that

```
debian:~# ssh sector
The authenticity of host 'sector (192.168.10.100)' can't be established.
RSA key fingerprint is 81:00:6e:dc:49:e1:5b:1d:76:86:8c:a4:55:91:0d:29.
Are you sure you want to continue connecting (yes/no)? y
Please type 'yes' or 'no': yes
Warning: Permanently added 'sector,192.168.10.100' (RSA) to the list of known host
Password:
Last login: Tue Sep 27 14:45:53 2005 from 192.168.10.254
Loading /usr/share/keymaps/i386/qwertz/de-latin1-nodeadkeys.kmap.gz
SECTOR:~$
```

Figure 1: On initial login, SSH imports the host key from the remote machine.

```
debian:~# ssh sector
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
81:00:6e:dc:49:e1:5b:1d:76:86:8c:a4:55:91:0d:29.
Please contact your system administrator.
Add correct host key in /root/.ssh/known_hosts to get rid of this message.
Offending key in /root/.ssh/known_hosts:1
RSA host key for sector has changed and you have requested strict checking.
Host key verification failed.
debian:~#
```

Figure 2: If the host key changes, the SSH client will refuse to connect.

both will use for the data transfer. The key is used once only for the current communication session, and both ends destroy it when the connection is broken. For extended sessions, the key will change at regular intervals with one hour being the default.

Initial Login

The easiest approach is to log in using the classic username/password method. The SSH client will automatically use your username as the login name on the remote machine. The first time, the client will not know the server's host key and will prompt you to confirm that you really do want to set up a connection with the remote machine. The program waits for you to confirm before generating the fingerprint (Figure 1). To check the key fingerprint, contact the administrator of the remote machine. This prevents man-in-the-middle attacks, in which an attacker reroutes network traffic to his own machine while spoofing a genuine login to your machine.

If you confirm the security prompt and enter your password in such a case, the attacker will then own your password; thus, some caution is recommended. If the host key changes, the client will refuse to connect when you log in later. Figure 2 shows the output from the SSH client.

The only thing that will help here is to remove the offending fingerprint from your `$HOME/.ssh./known_hosts` file and then accept the new key after contacting the administrator on the remote machine. To configure this behavior, use the `StrictHostKeyChecking` variable in `ssh_config`.

If you do not want to use your current account name to log in to the remote machine but have a different account name, the `-l login_name` option can help you. For example, the command `ssh -l tuppes sector` will log you into the remote machine as the user `tuppes`. SSH also accepts the following syntax: `ssh tuppes@sector`.

To run a single command on the remote machine, you simply append it to the command line (Listing 1).

If you get tired of typing your password, public key authentication provides an alternative. This technique uses encryption methods such as those used by GnuPG. Before you can use the public key approach, you first need to run `ssh-keygen` to generate a pair of keys

```
ssh-keygen -b 1024 -t rsa
```

The software will tell you that it has created a keypair with a public key and a private key on the basis of the RSA approach. When prompted to enter a password, press Enter twice. The program will then tell you where it has stored the data and will display the fingerprint for the new key.

In the example here, the software generated an RSA keypair (`-t rsa`) with a length of 1024 bits (`-b 1024`). An RSA key is fine for use with both protocol versions. For security reasons, the key length should not be less than 1024 bits. To be absolutely safe, you can use a key length of 2048 bits: 2048-bit keys are regarded as safe until the year 2020 based on the current state of the art. The key length has no influence on the data transfer speed because the program does not use this key to encrypt the data.

The next step is to copy the public key to the `$HOME/.ssh/authorized_keys` file on the remote machine from, for example, a floppy disk:

```
mount /media/floppy
cat /media/floppy/id.rsa.pub >> ↩
   $HOME/.ssh/authorized_keys
umount /media/floppy
```

Certainly you should avoid transferring the key by an insecure method, such as email or FTP. Figure 3 shows the fairly unspectacular login with the new key.

Passwords protect keys for interactive sessions; otherwise, anybody with physical access to your computer could use your keys to log in to the remote machine. Key-based, password-free logins are often used to automate copying of files to remote machines. For example, if you back up your data every evening and want to copy your data automatically to a remote machine, keys without passwords are a useful approach. If the key was password protected, you would need to enter the password for the SSH key to copy the data – so much for automated copying.

Useful Freebies

The SSH package includes two more useful programs: Secure Copy (`scp`) and Se-

```
debian:~# ssh sector
Last login: Wed Sep 28 13:36:22 2005 from 192.168.10.254
SECTOR:~$
```

Figure 3: Public key authentication makes the login more user friendly by removing the password prompt.

LISTING 1: Running Commands on the Remote Machine

```
01 jha@scotti:~$ ssh sector "ls -l"
02 Password:
03 insgesamt 52
04 Drwxr-xr-x 3 tuppes users 4096 2005-08-26 12:38 .
05 Drwxr-xr-x 16 root root 4096 2005-09-07 13:47 ..
06 -rw-rw-r-- 1 tuppes users 266 2005-04-12 12:00 .alias
```

cure FTP (`sftp`). As the names suggest, these programs are used to copy and transfer files by FTP via SSH. The basic syntax for the two programs is similar.

For example, the following command copies a file named `test.txt` from your home directory on the remote machine to your current working directory:

```
scp RemoteComputer:test.txt .
```

Depending on your authentication method, you might need to enter your password to do this; however, the colon is mandatory in all cases. It separates the name of the remote machine from the pathname. Also, you need to specify the local path. The easiest case is your current working directory, which is represented by the dot at the end of the line. To copy multiple files, just type a blank-delimited list of the file names:

```
scp RemoteComputerA:test1.txt ↩
   RemoteComputerB:test2.txt .
```

If you use the standard login approach, the client will prompt you to enter your password for each file you copy. If you use the public key method discussed previously, there is no need to type a password. The command `scp Remote-ComputerA:test.txt RemoteComputerB:` copies the file from remote computer A to remote computer B. To copy a file as the user tuppes from `/home/tuppes/files` to your local directory, type:

```
scp tuppes@RemoteComputer:↩
   files/test.txt .
```

Unlike SSH, you do not specify the `-l username` option here. If you are copying in the other direction – from local to remote – the procedure is just as easy:

```
scp ./test.txt ↩
   tuppes@RemoteComputer:/files/
```

`scp` copies the `test.txt` file from your current working directory to `/home/tuppes/files` on the remote machine.

Again, watch out for the closing colon. Sftp uses the same command structure as `scp` but has two operating modes: interactive, like the one you might be familiar with from FTP, and a batch mode. To use `sftp` to retrieve the sample file from the remote machine in batch mode, type:

```
sftp RemoteComputer:test.txt .
```

If you add `remote_test.txt` to the end, the program will give that name the local copy of the file. Typing `sftp RemoteComputer` opens an interactive, encrypted FTP session on the remote machine, and the server will accept FTP commands such as `GET` or `PUT` in the session.

Building Tunnels

SSH also lets you encapsulate other protocols. For example, you can run the telnet protocol over an encrypted SSH connection and do it transparently for users. The technical term for encapsulating one protocol inside another is tunneling.

The standard specifies that programs must be running on the same machine to use the tunnel. If you want to let other machines on the network use the tunnel, you must specify `-o GatewayPorts=yes` when setting up the tunnel. The alternative approach is to set the option in the `ssh_config` configuration file.

This setup is similar to a VPN (Virtual Private Network) connection but is easier to implement. The SSH variant has the disadvantage that you can only forward a single TCP port. Thus, you need an SSH tunnel for each port you want to forward. If you want to encrypt all communications between two machines, a VPN is probably a better choice.

Any user can set up a tunnel, although tunnels for privileged ports (i.e., below 1024) are reserved for root. To open a tunnel to a remote machine encapsulating the telnet protocol (port 23), enter:

```
ssh -c blowfish -L 23:RemoteBox:23 ↩
   RemoteBox
```

```
debian:~# netstat -tlpn | grep 23 | grep ssh
tcp        0      0 127.0.0.1:23            0.0.0.0:*               LISTEN     3311/ssh
tcp6       0      0 :::22                   :::*                    LISTEN     2364/sshd
tcp6       0      0 ::1:23                  :::*                    LISTEN     3311/ssh
debian:~# netstat -tpn | grep ssh
tcp        0      0 192.168.10.254:32790    192.168.10.100:22       ESTABLISHED3311/ssh
debian:~#
```

Figure 4: The netstat program showing an existing SSH tunnel.

The command uses the `-L` option to open a tunnel from local port 23 on the local machine (the first `23`) to port 23 on the remote machine. The fast Blowfish method is used for encryption. If you type two remote machine names, you can take advantage of another SSH feature: the ability to open a tunnel from the first machine, via the second, to a third. The command

```
ssh -L 23:192.168.1.1:23 192.168.20.5
```

starts the tunnel on the local machine, and routes it by way of an intermediate station (192.168.1.1) to its endpoint. The generic syntax for opening a tunnel from the local machine to the remote computer is thus: `ssh -L LocalPort:Remote-ComputerA:RemotePort RemoteComputerB`. For a direct tunnel, the two host designations are identical.

Tunnel Tricks

In Figure 4, the `netstat` command demonstrates that I really have set up a telnet connection via SSH. The first `netstat` command tells me that an SSH process with a process ID of 3311 is listening on port 23. The second command shows that a connection to port 22 with precisely this PID (3311) exists.

If you were to look more closely at the syntax used to open a tunnel, you might be led to assume that the local and remote ports do not need to be identical – and this is true. Assuming the remote machine is running a proxy configured for transparent proxying on port 3128, you could redirect all HTTP requests:

```
ssh -o GatewayPorts=yes ↩
  -L 80:RemoteComputer:↩
  3128 RemoteComputer
```

This process of redirecting one port to another is known as port forwarding. For other computers on the network to use the tunnel, use of the `-o GatewayPorts=yes` parameter is required.

In a similar fashion, tunneling works in the reverse direction. The following syntax allows you to set up a return tunnel from the remote machine to your local computer:

```
ssh -R RemotePort:LocalComputer:↩
      LocalPort RemoteComputer
```

```
harmuth@debian:~$ ssh sector
Last login: Wed Sep 28 16:42:42 2005 from 192.168.10.254 on ttyp3
Linux SECTOR 2.4.31 #1 SMP Do Aug 4 14:24:58 CEST 2005 i686 unknown
You have mail.

Last login: Wed Sep 28 16:42:52 2005 from 192.168.10.254
harmuth@SECTOR:~$ xclock &
[1] 8056
harmuth@SECTOR:~$ echo $DISPLAY
localhost:11.0
harmuth@SECTOR:~$ ps aux | grep xclock | grep -v grep
harmuth   8056  1.0  0.3  5280 3308 ttyp3   S   16:43   0:00 xclock
harmuth@SECTOR:~$ su -
Password:
SECTOR:~$ netstat  tpn | grep xclock
tcp        0      0 127.0.0.1:52503        127.0.0.1:6011        VERBUNDEN  8056/xclock
SECTOR:~$
```

Figure 5: A forwarded X11 connection – the Xclock is running on a remote machine.

In my proxy example, this would be:

```
ssh -o GatewayPorts=yes ↩
  -R 3128:LocalComputer:80 ↩
  RemoteComputer
```

Graphical Tunnels

The X Window System is natively network-capable, but almost nobody uses this ability because communications are again unencrypted over the wire. Tunneling with SSH makes this a far more attractive proposal.

To tunnel X11, the SSH daemon (`sshd`) emulates an X server and occupies a display (number 11 by default). When you log in to the server, the server sets the `DISPLAY` environmental variable to this value, or to *localhost:11.0* to be more precise. The idea is to avoid collisions with the X server running locally. Information sent by a computer to this display is encrypted and sent to your machine.

OpenSUSE 11.0 enables X11 forwarding (the technical term for the process I just described) by default. If needed, you can disable X11 forwarding on the machine configured for forwarding by setting the `X11Forwarding` variable to `no` in `etc/ssh/sshd_config`. The `X11DisplayOffset` variable with a default value of `10` defines the distance between the virtual display and the physical display; you should keep the default here.

If the machine on which you want to display tunneled X11 is an openSUSE 11.0 machine, the `etc/ssh/ssh_config` file will already have the `ForwardX11Trusted` variable set to `yes`. This completes the configuration work.

Next, log in to the remote machine and launch, for example, the `Xclock` program. Figure 5 shows the display (*localhost:11.0*), the process, and the matching network connections.

Conclusions

The SSH package includes a collection of important programs that make working on networks far more secure. The feature scope covers anything from basic encrypted connections, through tunneling and port forwarding, to X11 forwarding, leaving very little to be desired in daily use. ■■■

REAL SOLUTIONS FOR REAL NETWORKS

Each issue delivers technical solutions to the real-world problems you face every day.

ADMIN magazine covers Windows, Linux, Solaris, and popular varieties of the Unix platform.

Learn the latest techniques for better network security, system management, troubleshooting, performance tuning, virtualization, cloud computing, and much more!

FREE CD or DVD in Every Issue!

Now 6 issues per year!

ORDER ONLINE AT: shop.linuxnewmedia.com

Key-based authentication with SSH and VNC

Crack Proof

pnphoto, 123RF.com

Learn how you can tunnel VNC remote control through an SSH connection. *By James Stanger*

When preparing for this article, I decided to do a bit of research to discover what most Linux users knew about security. I won't give the usual sardonic answer of "not much." Actually, I found that end users actually know quite a bit about security. I was pretty impressed.

Nevertheless, in the official Ubuntu Security forum [1], I found that the most common security problem facing Linux users involves attackers stealing usernames and passwords from default implementations of Secure Shell (SSH) and Virtual Network Computing (VNC) servers. This article describes how to use VNC with SSH for secure remote desktop operations. I will use Ubuntu Linux in these examples, but the tools described in this article also run on other flavors of Linux. For that matter, versions of VNC and SSH are also available for Windows and Mac OS, so these concepts will work for other systems as well.

Even though SSH [2] and VNC [3] are really cool ways to remotely control computers, neither of these tools uses the most secure protocols and procedures by default. For example, SSH defaults to using traditional usernames and passwords during authentication. The result of this default behavior is that, even though SSH encrypts all network transmissions via a very powerful encryption algorithm, such as RSA, DSA and AES, passwords are still passing across untrusted networks.

The goal is never to have any password transmitted over a network. And I mean never. When it comes to VNC, the encryption used is quite weak and can be broken quite easily. One way to fix that problem is to "tunnel" VNC traffic inside of a secure SSH connection, as shown in Figure 1.

In Figure 1, host A and host B are sending VNC traffic via an encrypted SSH tunnel. I'll next take a look at how you can secure SSH and VNC transmissions by using public key encryption and a bit of tunneling.

Public Key Encryption and Tunneling

You really don't need to know all the details of encryption algorithms to understand how to secure SSH, but a bit of an explanation is certainly useful.

Public key encryption involves the use of a mathematically related pair of values called a key pair. In SSH, this key pair is usually stored in text files on your computer. The values that comprise this key pair are so closely related that one part of the pair can be used to encrypt information, and the other half is used to decrypt that information. What one half encrypts, the other decrypts.

One part of this key pair is kept completely secret, and this part is called the private key. You don't show or give the private key to anyone, and you enact permissions on the private key to make sure that no one gets hold of it. The other half, called the "public key," is shared. You can send it to your friends or even to your enemies. It doesn't matter.

As I'll describe shortly, you can configure SSH to stop using common passwords and use a key pair to authenticate connections. That means the only data that travels across the network during the authentication sequence is freely available information regarding the pub-

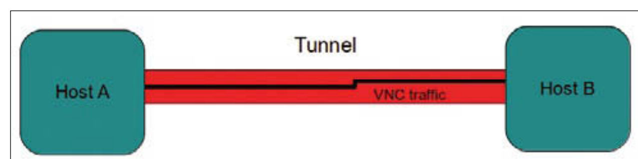


Figure 1: Tunneling SSH traffic.

lic key; nothing secret is ever transmitted. Thus, you can use SSH to authenticate users without having any passwords cross the network. So, you now know enough of the theory about key pairs and SSH to get things going.

But what about VNC and tunneling? Once you configure SSH to use public key encryption, you can use SSH to create an encrypted, secure tunnel between two computers. To begin, you establish an SSH connection from Computer A to Computer B. You can place all sorts of protocols inside of this tunnel – kind of like a poor man's Virtual Private Networking (VPN) connection. All you have to do is tell the VNC application to use the SSH tunnel you've created.

Next, I'll look at how you can enhance a default SSH implementation by configuring it to use public key authentication, and then I'll show how to tunnel VNC through an SSH connection. You can think of this article as showing you how to go public with SSH and then go underground by tunneling VNC.

SSH AND VNC OVERVIEW

SSH is a client-server protocol used to control systems remotely. It is a command-line tool; you use the `ssh` command in a plain old – but powerful – terminal. SSH was created as a replacement for old, unencrypted tools such as telnet, rhost, and rlogin. VNC is a client-server protocol that allows you to see and share an entire desktop. It is a rich, graphics-based tool that allows you to see the desktop of a remote computer as if it were that of the computer sitting right in front of you.

Both of these tools are great ways to get around; SSH lets you control a remote system via a low-bandwidth, powerful command line-based terminal. VNC allows you to control a remote system via a sophisticated GUI. Both have their uses; SSH is terrific for getting in and out of a system and executing complex terminal-based commands. VNC is perfect for manipulating a graphical environment. However, VNC doesn't use strong encryption. Also, because it supports a graphical environment, VNC requires more bandwidth than the terminal-based SSH environment.

Public Key-Based Authentication in SSH

Before I delve into the details of enabling key-based authentication, I'll explain the big picture. To properly secure SSH, you'll need to take the following basic steps:

1. Generating a key pair: Remember, SSH encrypts traffic by default. So, you can still authenticate strongly without having your passwords pass across the network.
2. Exchanging public keys.
3. Configuring SSH to use the keys you have generated.
4. Disable password-based authentication.

Generating a Key Pair

You generate a key pair using the application named ssh-keygen. Figure 2 demonstrates the typical sequence when creating a key pair. By the way, the sequence shown in Figure 2 will create a 2048-bit key pair. That's pretty strong. But, if you want to create a key pair using a stronger bit setting, you can try the following command:

```
ssh-keygen -t rsa -b 4096
```

The preceding command creates a 4096-bit key, which is much more difficult to crack. Still, I think that's a bit of overkill for most implementations. (Larger keys need more processing time.)

Regardless of the bit size you want, the sequence of events, once you run the `ssh-keygen` command, is as follows:

1. The `ssh-keygen` command creates a directory to store the key pair. This directory is usually off your home directory. If your login is "james," then the directory defaults to storing the directory in the `/home/james/.ssh/` directory. You can specify any directory you want without causing problems in the future.
2. You must enter a passphrase to protect the key. I usually take the advice of security experts who say don't use a password. If you password-protect your key, you will always have to enter that password when using your public key. Remember, you'll often be using SSH across an untrusted network. If you have to enter a password to a key during an SSH session when you're connecting via an untrusted network, then you're defeating the purpose of never having passwords – not even encrypted ones – travel across the network. Remember, that's why you're generating a key pair in the first place.
3. By default, `ssh-keygen` will then create the key pair using the RSA algorithm. You can specify DSA if you wish. I find RSA is the best to use, because it is the industry standard.
4. If you're using OpenSSH version 5.1 or later, `ssh-keygen` will then create the "randomart" visual host key, which is a unique picture in ASCII art form. The visual host key is an actual picture based on your newly generated public key.

The creators of OpenSSH created visual host keys because they thought it would be easier for individuals to recognize the public keys of systems pictorially, rather than by reading arcane words and letters such as *23:00:21:33:d4:0f:95:f1:eb:34:b2:57:cf:3f:2c:e7*. The idea is that if you regularly log into a system and see that the visual host key has changed, you will know that a problem exists.

If you open a terminal and change to the `~/.ssh/` directory, you'll see that this directory now lists the following files:

- `id_rsa`: Your private key. It is imperative that you keep this key as private as possible. Make sure that this key is always stored in a directory with restricted permissions. If this key is revealed, then anyone who obtains this key and understands SSH (e.g., anyone who reads this article) will be able to access your system and compromise its security.

```
james@jamey: ~
james@jamey:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/james/.ssh/id_rsa):
Created directory '/home/james/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/james/.ssh/id_rsa.
Your public key has been saved in /home/james/.ssh/id_rsa.pub.
The key fingerprint is:
b2:0d:53:9f:b2:38:9c:39:8d:cb:95:d8:f4:ff:69:6d james@jamey
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|                 |
|        .        |
|       . . .     |
|      + S o      |
|       @ =       |
|      =o* .    . |
|     oo=.  .  ..E|
|      *o    .oo. |
+-----------------+
james@jamey:~$
```

Figure 2: Using ssh-keygen.

SHOW YOUR RANDOMART

To make it so that your system automatically shows the randomart visual host key for a system, edit the /etc/ssh/ssh_config file so that the VisualHostKey value is no longer commented out and reads as follows:

```
VisualHostKey yes
```

You will then need to restart the SSH daemon by issuing the following command from a terminal:

```
$ sudo /etc/init.d/ssh restart
```

Once you do this, you will see the visual host key every time you log in to the system. Understand, though, that some applications and scripts might see the visual host key as a problem; so, make sure you use this feature carefully.

- id_rsa.pub: Your public key. You should re-name this key to something less generic, such as yourname_yourhost.rsa, where *yourname* and *yourhost* represent your login name and host name. That way, everyone will always know who this key belongs to.
- authorized_keys: This file will contain the public keys of other users. Any key placed into this file will make it so that this user can log on to your system without using a username and a password.
- known_hosts: This file contains the list of host keys that your local SSH server has exchanged keys with.

The ~/.ssh/ directory may also contain a file named config. This file can contain user-specific settings that aren't particularly relevant to this article.

Exchanging Public Keys: The Old-Fashioned Way

So, you've now generated a key pair, and it's time to exchange your public key. You have at least two options for doing so. You can do this manually by changing to the ~/.ssh/ directory and then renaming the id_rsa.pub file, as discussed previously. Then, using anonymous FTP, or even a USB key if you want to be uber-secure, you can transfer your public key to the ~/.ssh/ directory on the remote system. Again, your goal is to establish security without having passwords travel across the network.

Then, copy the contents of the public key file for the user on the remote system into the authorized_keys file. The syntax for copying this information is:

```
protocol | key hash | username@host
```

For example, if you had my public key and had to edit the ~/.ssh/authorized_keys file to allow me to automatically log on to your system using SSH public key encryption, you would enter the code shown in Listing 1 into the file. Once you have updated this file, you are ready to test the trust relationship you've established.

Exchanging Public Keys: The New and Improved Way

But, you don't have to do all this manually. You can use the handy ssh-copy-id command. If, for example, the remote system is already using SSH, you can transfer your public key to the remote system's ~/.ssh/ directory using the ssh-copy-id command, as follows:

```
$ ssh-copy-id sandi@hostb.company.com
```

The preceding command will automatically transfer your public SSH key into the ~/.ssh/ directory of the remote system. This command will automatically update your ~/.ssh/authorized_keys file. But, remember that you'll have to log in to that remote system using a traditional user name and password. So, to be extra-secure, I would use a less risky method, such as physically transporting the key. If that's not possible, then you'll have to take a risk.

LISTING 1: Updating ~/.ssh/authorized_keys

```
01 01 ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDeBKQeIjmDLDH4bc5pX6398OmpCIx6s22XgxvKAGL
02 Iph1OjGfMacE/O6W6LHV9ZIiTvqRasuj4a9ejlJzMgoNKJGixwpcz/
03 02 6lvU5gISbKwdW0E8p/U10mz5qtKQWaG+rx6Pai9o5+OnsTgadpZ/q6nPiXysl/PMs4AOTvjC9luQRju/
04 oo7e9jrrkiIaKbyxy9oWn3oJu+8cUJyKB/hsNjk671jrpl0yQ4HeU
05 03 7hKtHEwVYQIXX8cC49WEcCCA5d288Q+zbBnhmBFqbtqHF48f9YKcLhF/pXDRrOMDAfAQ5VEQ
06 Am47BVlGal+M+ar8DD+gOPSIkOOMk3YShFsOCR2H53 james@jamey
```

Once you've taken these steps either manually or using the ssh-copy-id command, repeat the processes of creating and exchanging keys with a second host. It is very important that you place your public keys into the ~/.ssh/ directory of the target system. This is because you will be editing the ~/.ssh/known_hosts file so that it contains a reference to the key you just placed onto the new system. Congratulations! You now have established what the security industry calls a "trust relationship" between two systems. You can now securely connect to that second remote system, because now you're still authenticating, but with public keys, rather than passwords (see the "Your Choice" box).

Disabling Password-Based Authentication

If you really want to go the extra mile, you can disable password-based authentication in SSH altogether. This step is a good idea if you've already enabled key-based authentication. To disable password-based authentication, take the following steps:

1. As root, open the /etc/ssd_config file.
2. Find the following command and uncomment it: PasswordAuthentication no. If you can't find this passage, enter it in a new line, exactly as written.
3. Restart the SSH daemon as root with the following:

```
$ sudo /etc/init.d/ssh restart
```

Now, you will only be able to use public key encryption to authenticate.

Updating your SSH Server

Here's a quick story to remind you to keep your SSH software updated: I once set up a system with strong keys, disabled password-based authentication, and made sure my computer didn't allow root-based logins via any means other than logging in physically, yet I still got hacked. Why? Because I unwisely ignored several security bulletins telling me that my SSH server version was out of date. So, be sure to use Ubuntu's update feature to keep your system current. You'll be glad you did.

Tunneling VNC

So, you've successfully set up your SSH server so that you can use public key en-

YOUR CHOICE

Why doesn't SSH use public key authentication in the first place? If you want the short answer, it's because the process of creating a key pair for individual logins is best left to end users and systems administrators. The choice of moving from user-based authentication to key-based authentication requires some personal steps, including choosing where to store the private key and whether or not to password-protect it, and choosing which encryption algorithm you want to use.

The creators of OpenSSH, which is the version used in Ubuntu, know about all of these choices, and so they leave the process to those who know best: educated end users and administrators.

cryption and tunnel connections. Now, I'll show you how to use your new and improved SSH connection to do some tunneling. The whole idea behind tunneling is to start an encrypted "pipe" on your own system that dumps out to the remote system. You can then put any protocol into that pipe, including VNC.

To create the tunnel, the following conditions must exist on the host you wish to connect to using VNC:

- It needs to be running the VNC server on the default port (TCP 5900).
- It needs to be running SSH, preferably using public key encryption.

Your own system doesn't need to be running the VNC server, but you will need a VNC client, such as the TightVNC viewer.

The first step is to create the tunnel between host a and host b. Suppose I'm at the system named `hosta.company.com`, and I want to create a tunnel that accommodates VNC traffic to `hostb.company.com`. To create a tunnel using the good old terminal-based `ssh` command, I would issue the following at the command prompt:

```
ssh -f sandi@hostb.company.com -L ⮐
    4600:hosta.company.com:5900 -N
```

The preceding command tells SSH do several things. To begin, the `-f` option tells SSH to go immediately into the background. The next part, `sandi@hostb.company.com`, tells SSH to connect to that remote system. Then, `-L 4600:`, tells SSH to start a tunnel at TCP port 4600 of my local system and connect to port 5900 of the remote system, which is the default VNC port. The `-N` option simply tells SSH not to issue a command on the remote system. This option effectively tells the remote SSH daemon that it is participating in a tunnel.

Using PuTTy to Tunnel SSH Traffic

You don't have to be so "old school" like me and use the terminal-based `ssh` command. You can use the excellent PuTTy application, as shown in Figure 3, to create the tunnel.

In this figure, you can see that I've specified 4600 as the local source port and the destination as `hostb.company.com`, port 5900. Make sure that the *Local* and *Auto* radio buttons are selected.

You can download PuTTy by using Synaptic and searching for the name, or you can download the application directly from the website [4]. Regardless, you'll still need to have the SSH daemon and client software installed.

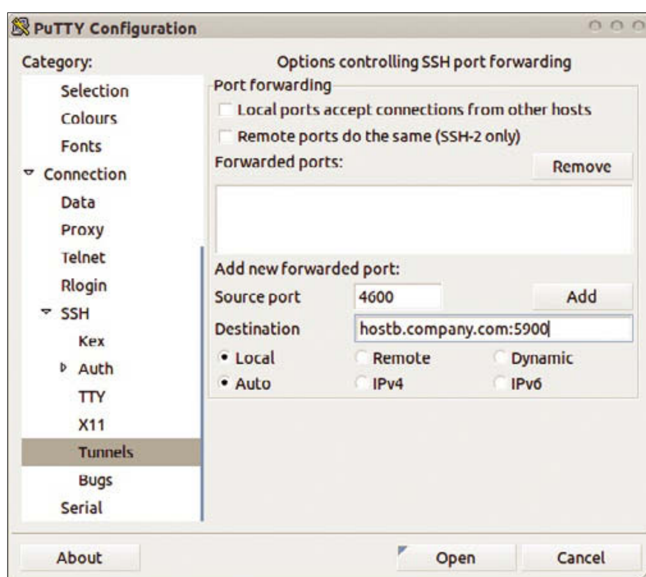


Figure 3: Using PuTTy to start an SSH tunnel.

Tunneling VNC Traffic

But, your tunnel isn't done just yet. To tunnel VNC through SSH, you can use the `vncviewer` command and do the following:

```
$ vncviewer -via localhost:4600
```

Or, if you prefer to use a graphical client, you could use the `xtightvncviewer` command and enter *localhost:4600* into the window. If you happen to be using a Windows-based application, such as the TightVNC client, you do the same thing. Just specify localhost and the appropriate port, in this case 4600.

Regardless of the client you use, you now have tunneled your VNC client through a secure SSH connection.

Problems with Tunneling?

You might find that it's impossible to tunnel connections. In that case, the problem generally comes down to a firewall issue. Although, thankfully, most ISPs do not block SSH, many workplaces will block SSH traffic.

If that's the case, then you'll have to resort to one of the following: First, you could use an approved protocol to tunnel your traffic. Second, you could try and talk your network administrator or security expert into adding a firewall rule that allows SSH traffic. Good luck with that! Most admins/security experts will view your request as punching a hole in the firewall and will likely deny your request.

Conclusion

Congratulations! You now know how to use SSH more securely by enabling public key authentication. Not only that, but you've also learned that you can tunnel VNC traffic through an SSH tunnel. Of course, you aren't limited to tunneling just VNC traffic. If desired, you could tunnel email, web, instant messaging, or any other traffic. All you need are two systems that support SSH, a client that supports the ability to specify a custom port, and a network connection that doesn't block SSH traffic. ■■■

INFO

[1] Ubuntu Security forum: *http://ubuntuforums.org/showthread.php?t=510812*

[2] SSH: *http://en.wikipedia.org/wiki/SSH*

[3] VNC: *http://en.wikipedia.org/wiki/Virtual_Network_Computing*

[4] PuTTy: *http://www.chiark.greenend.org.uk/~sgtatham/putty*

Doing more with SSH

Connection Protection

Marcio Eugenio, 123rf.com

Most admins know that SSH is a useful tool for setting up encrypted connections to a remote host. But SSH can do much more, including transferring files, forwarding ports, and even setting up a genuine VPN. *By Thomas Drilling*

The term *SSH* ("Secure Shell") refers to both the network protocol and a suite of tools that give administrators the ability to log in to a remote device via a secure connection. SSH has been around since 1995, and the popular OpenSSH free implementation [1] has existed since the turn of the millennium. SSH gives the administrator a single tool for remote login, as well as a collection of remote applications for remote execution, remote copy, and remote X11 client and a number of port forwarding scenarios.

SSH and the tools of the SSH package – `ssh`, `slogin`, `scp`, and `sftp` – have entirely replaced legacy Unix tools such as `rsh`, `rlogin`, `rcp`, and `telnet`. Each of these commands executes a shell on the remote host and allows the user to call commands. Because each client needs a user account on the host where the commands will run, all the commands mentioned thus far also perform authentication.

Because the SSH commands use a public key cryptography infrastructure, the SSH package also include the `ssh-keygen`, `ssh-agent`, and `ssh-add` tools to support authentication. The complete syntax, including a full set of options, is output when you type `ssh` without any parameters or when you type the `man ssh` command.

Interactive SSH

Once you have opened an SSH connection

```
ssh -l user_name Remote-Host
```

and entered your user password on the remote host, SSH will forward all of your keyboard input to the remote host. A lesser known fact is that you can use escape sequences to control SSH itself. The tilde at the start of a new line tells SSH to expect an escape sequence; the next character is the start of the command: for example, `~#>` lists the open SSH connections, and `~` terminates the current connection, which is useful if the shell isn't responding. For a list of all support commands, type `~?` (Figure 1). The `-e` option lets you

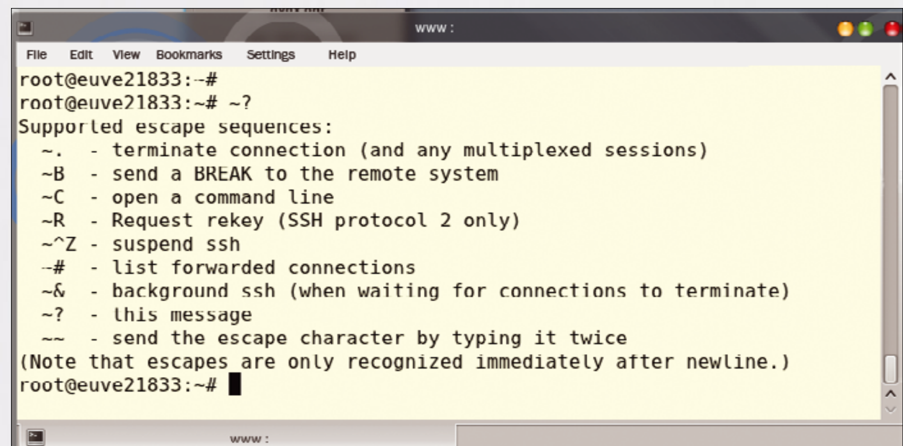


Figure 1: SSH can be controlled via escape sequences.

specify a different character for introducing escape sequences in SSH.

Depending on your keyboard layout, you might need to press the keys for creating the tilde (~) twice – as is the case on Ubuntu – before you type the required command character. (Note that the visible ~? characters in Figure 1 are for illustration purposes only. In a real situation, the escape sequence does not appear as visible text after the prompt.)

Copying Files

SSH lets you transfer files with the `scp` command; however, most administrators tend to use the far more powerful `rsync` command, which means that `scp` is becoming historic, although it is fine for most purposes. Type `man scp` for the complete syntax. In the simplest case, enter the following:

```
scp File1 user_name@Host:File2
```

To transfer files in the other direction, enter:

```
scp user_name@Host:File1 File2
```

To recursively copy complete directory trees, you can add the `-r` flag.

Sshfs

If you are looking for a more elegant approach, you will probably want to mount the remote filesystem via SSH, which you can do thanks to the `sshfs` tool (Figure 2). Because SSH authenticates and encrypts any data you transfer, `sshfs` is a very convenient tool for securely transferring data across the Internet. Sshfs gives a non-privileged user the ability to mount a remote filesystem using SSH; however, this technique does required the FUSE (Filesystem in Userspace [2]) module on the client side. On the server side, `sshfs` requires only an SSH server with an SFTP subsystem. If you install the `sshfs` package on the client side, your package manager will automatically install `fuse-utils` and `lib-fuse2`.

In the simplest case, the syntax for `sshfs` is as follows:

```
sshfs user_name@host:path ↩
  local-mountpoint [options]
```

Data Transfer with Graphical Clients

Some file managers (e.g., Midnight Commander) can use an SSH connection to access the filesystem of a remote host

```
drilling@ws1-kubu:~/sshfs$
drilling@ws1-kubu:~/sshfs$ ssh -l root www.thomas-drilling.de
root@www.thomas-drilling.de's password:
Linux euve21833.vserver.de 2.6.18-028stab094.3 #1 SMP Thu Sep 22 12:47:37 MSD 2011 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@euve21833:~#
root@euve21833:~# ls
parallels  plesk9.xml  psa-autoinstaller_3.5.0-090817.16_amd64.deb
root@euve21833:~#
```

```
drilling@ws1-kubu:~$
drilling@ws1-kubu:~$ sshfs root@www.thomas-drilling.de:/root /home/drilling/sshfs
root@www.thomas-drilling.de's password:
drilling@ws1-kubu:~$
drilling@ws1-kubu:~$ cd sshfs
drilling@ws1-kubu:~/sshfs$
drilling@ws1-kubu:~/sshfs$ ls
parallels  plesk9.xml  psa-autoinstaller_3.5.0-090817.16_amd64.deb
drilling@ws1-kubu:~/sshfs$
drilling@ws1-kubu:~/sshfs$
```

Figure 2: The contents of `/root` on a remote server: once via SSH and then as a local mount using `sshfs`.

```
Help
FIle transfer over SHell filesystem

The fish file system is a network based file system that allows you to
manipulate the files in a remote machine as if they were local. To use
this, the other side has to either run fish server, or has to have
bash-compatible shell.

To connect to a remote machine, you just need to chdir into a special
directory which name is in the following format:

/#sh:[user@]machine[:options]/[remote-dir]

The user, options and remote-dir elements are optional. If you specify the
user element, the Midnight Commander will try to login on the remote
machine as that user, otherwise it will use your login name.

The available options are:
  'C' - use compression;
  'r' - use rsh instead of ssh;
  port - specify the port used by remote server.
If the remote-dir element is present, your current directory on the remote
machine will be set to this one.

Examples:

   /#sh:onlyrsh.mx:r/linux/local

Hint: M-! will allow you to execute programs and see the output in the viewer.
drilling@ws1-kubu:~$
```

Figure 3: Midnight Commander can use SSH to access remote directories.

(Figure 3). The `mc` method doesn't require kernel support on the client. The Gnome file manager, Nautilus, also supports the SSH protocol; the administrator only needs to type the required address as a complete URL (e.g., `ssh://computername/path`) in the address bar. (The current Nautilus version 3.2.1 in Gnome 3 doesn't display the address bar by default, so you need to press Ctrl + L to switch it on, as shown in Figure 4.) Nautilus then displays the SSH login dialog.

Of course, you can log in to the SSH server with a different username:

```
ssh://user_name@hostname/path
```

If you prefer an even more convenient approach, you can also use the *File | Connect to server* wizard in Nautilus, select the *ssh* entry from the *Type* drop-down box in the *Server details* area of the *Connect to server* dialog, and enter the authentication data in the corresponding boxes in the dialog. Incidentally, this type of SFTP/SSH login will work in most other Gnome applications.

KDE has had SSH support for a long time. In Dolphin or Konqueror, you can access SSH through the `fish` KIO slave, as follows,

```
fish://computername/path
```

or you can include the user account with:

```
fish://user_name@computername/path
```

Dolphin has also stopped showing users the URL input bar by default in the latest version; however, you can quickly find this function in *View | Address | Editable address bar*. Dolphin then immediately shows you the user authentication dialog; however, it uses the currently valid local username as the default, and you will typically need to overwrite this with the remote username.

KDE users can also add an SSH network folder as a bookmark in Dolphin in *Places | Network* by selecting *Add network folder*.

Incidentally, KDE's Fish syntax also gives you access to remote files in the complete KDE context and, thus, via *File | Open* in most KDE applications. Users need to enable editable address input in the KDE file selector; to do so, right-click the current path in the file selector to open the file selector's drop-down menu and select *Edit* instead of the default *Navigate*.

Using Compression

SSH also supports compression, which can be really useful if the network is a bottleneck. Compressing the communication between `ssh` and `sshd` might involve more computational overhead at both ends, but to compensate for this, SSH only needs to transfer about 50 percent of the packets. If you permanently set up compression on the server side using a `Compression yes` line in the `/etc/ssh/sshd_config` file, you can really speed up slow DSL or ISDN transfers in combination with port forwarding.

Similarly, you can set up permanent compression on the client side with a `Compression yes` line in `$HOME/.ssh/config`. To enable compressed transmission temporarily, simply use the `-C` option – not to be confused with `-c` for the cipher specification in encrypted connections.

Master Mode

Master mode gives the administrator the ability to open multiple logical SSH connections over a physical connection by starting one SSH connection as the master. You can then route all further SSH connections to the same host with the same user account at the other end via the master connection – without needing to open a new physical connection. In this case, the client uses a Unix socket on the master and not directly on the server at the other end; you need to specify a master socket.

Figure 4: Nautilus 3.2.1 forces users to enable the address bar manually by pressing Ctrl+L.

Transferring multiple sessions via a master channel can also offer significant latency benefits. To configure this "opportunistic sharing," you just need to add the following lines to the `$HOME/.ssh/config` file:

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

`Host *` means that the subsequent configuration is used for connections from any host; alternatively, you could enter a static hostname. `ControlMaster auto` tells SSH to use an existing connection for master mode if possible; otherwise, SSH opens a new connection. The following entry

```
ControlPath ~/.ssh/master-%l-%r@%h:%p
```

tells SSH where to create the socket file representing the master connection. `%r` is replaced by the login name, `%h` by the hostname, and `%p` by the port number. The master connection is then initiated as follows:

```
ssh -M -S $HOME/.ssh/socket user_name@host
```

To open any subsequent connection to the same host with the same user account, enter:

```
ssh -S $HOME/.ssh/socket user@host
```

X11 Forwarding

X11 forwarding lets you launch programs with a graphical user interface on a remote computer via SSH but directs the input and output to the local desktop. This technique works independently of the operating system on the remote computer, assuming the program keeps to the X11 standard, which thus practically restricts the option to Linux, BSD, and Unix.

Although any number of powerful graphical, remote control alternatives are available, SSH has the advantages of being included free with any Linux system. Admittedly, GUI-based remote control with SSH isn't very fast, but for occasional system administration use, X11 forwarding is fine.

To enable X11 forwarding, use the `-X` option (not to be confused with the lowercase `-x`, which disables port forwarding). X11 forwarding only gives the program on the remote computer re-

stricted privileges for the local display, and this can cause the occasional application to fail. If you are experiencing privilege issues, you can still grant the program full access with the `-Y` option, although this option is not recommended.

You should also avoid the `-Y` option if you don't trust the administrator on the remote host; `-Y` installs a tunnel, which attackers could also use for reverse tunneling to attack your display. Incidentally, you can alternatively call SSH with the `-o` parameter instead of `-X` and pass in a value of `ForwardX11=yes`. You also have the option of setting `ForwardX11 yes` in `$HOME/.ssh/config`.

Building Tunnels with SSH

SSH also lets you secure (virtually) any other protocol, such as the legacy POP3 protocol or insecure VNC connections. Port forwarding allows administrators to redirect individual ports through a secure SSH connection, with SSH acting as a proxy that accepts the connection at one end of the SSH tunnel and connects the endpoint for the connection with the target server at the other end.

SSH supports two different operating modes, local port forwarding and remote port forwarding, which are often referred to as outgoing and incoming tunnels, although local port forwarding is used far more frequently. The direction in which you set up the tunnel – that is, local or remote port forwarding – is defined by the parameters `-L` and `-R`.

Local port forwarding forwards a connection arriving from a freely selectable local client port through the secure SSH channel to a port on the remote server; this is a classic "outward bound" tunnel. The generic syntax to initiate local port forwarding is:

```
ssh remoteuser@remotehost -L ⮐
  localport:remotehost:remoteport
```

The following example tunnels an insecure FTP connection using the standard port 21 across a secure SSH connection. An FTP server is running on the machine `www.thomas-drilling.de`, and the client computer `ws1-kubu` opens the secure SSH connection and subsequently launches the FTP client in a separate terminal session with a target address of port 4444 on the localhost.

```
drilling@ws1-kubu:~$ sudo ⮐
  ssh dilli@www.thomas-drilling.de ⮐
  -L 4444:www.thomas-drilling.de:21
```

This command opens a secure SSH connection on the local SSH client to the remote computer `www.thomas-drilling.de` using the `dilli` user account. At the same time, it is listening for any requests that reach port 4444 from `ws1-kubu` in order to forward them to port 21 on the computer `www.thomas-drilling.de`; the communications use a previously configured SSH connection. The administrator on the local machine `ws1-kubu` can set up the FTP connection in this way (Figure 5):

```
drilling@ws1-kubu:~$ sudo ⮐
  ftp localhost 4444
```

The port forwarding syntax in the man page is as follows:

```
ssh -L [bind_address:]port:host:port ⮐
  user@remotehost
```

This syntax is slightly misleading. The parameters *`host`* and *`remotehost`* in this syntax notation refer to the same remote server because *`host`* is from the viewpoint of the remote system. Thus, you could implement this as `localhost:21` instead of `www.thomas-drilling.de:21` because `localhost` relates to the remote host's perspective.

When choosing the input port (SSH), note that you are not allowed to use a privileged port below 1024 unless you are root, which explains why higher port numbers are typically used for local port forwarding. The second *`port`* parameter specifies the target port for the forwarding operation; thus, this setting refers to the port number of the service you are tunneling. If you only want to support port forwarding and prevent a shell being launched on the remote host, you can add the `-N` parameter (Figure 6).

If you want to query, say, a POP3 mail server on your virtual server via an en-

Risk-Free Trial!

GET IT NOW!
SAVE TIME ON DELIVERY WITH OUR PDF EDITIONS
(DVD NOT INCLUDED)

ORDER YOUR TRIAL NOW!
UK £ 3, Europe € 3, USA / Canada US$ 3, Rest of World (by Airmail) US$ 9

shop.linuxnewmedia.com/trial

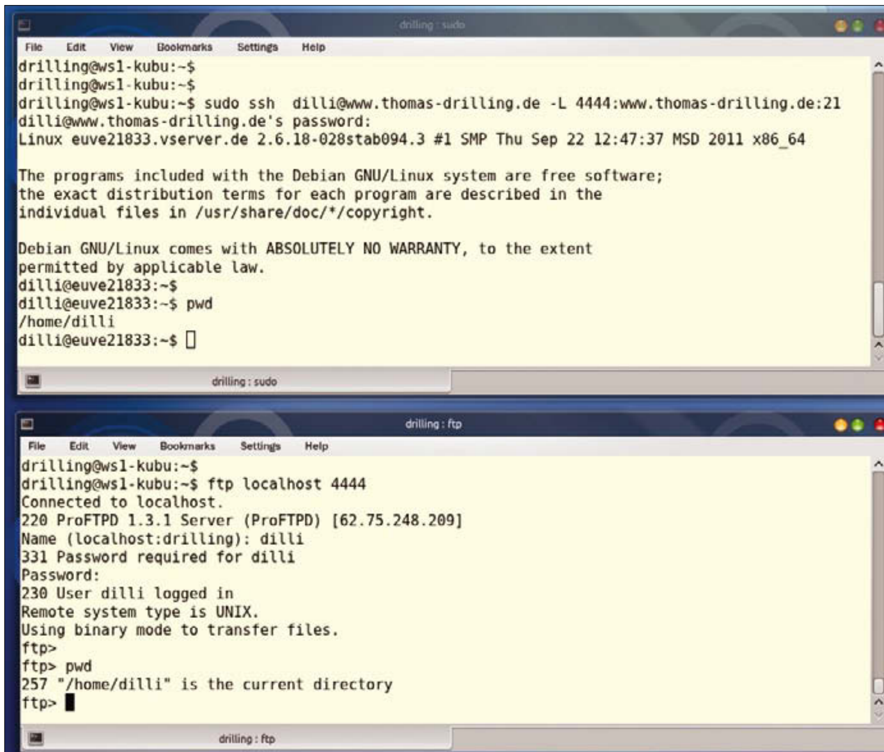


Figure 5: Setting up an SSH tunnel to protect the insecure FTP protocol. The user can conveniently open the secure FTP connection from the client side using a local port.

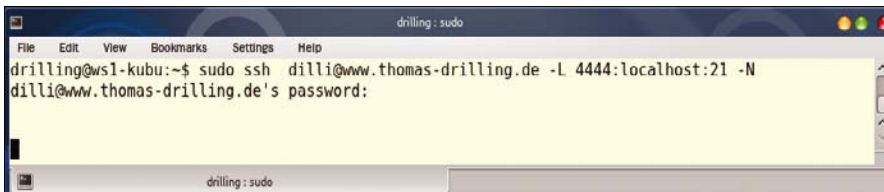


Figure 6: The `-N` parameter prevents a shell from being launched on the remote system through local port forwarding.

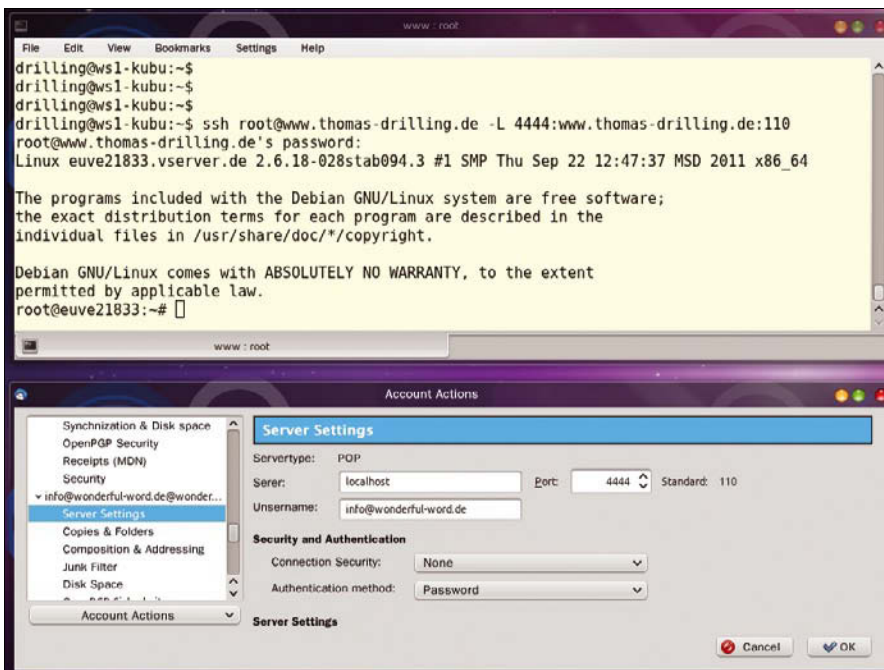


Figure 7: Local port forwarding lets you secure the inherently insecure POP3 protocol for querying a remote mail server, even if the POP server doesn't support SSL.

crypted connection, you can set up an encrypted SSH tunnel for port 110 on your Vserver using local port forwarding:

```
ssh user_name@remotehost -L ⮐
  20110:remotehost:110
```

Now, all you need to do is enter `localhost` with a port number of `4444` as your POP server in your mail client, such as Thunderbird, to encrypt mail transmission without the mail server itself supporting SSL (Figure 7).

Remote port forwarding works in exactly the opposite way from local port forwarding; in other words, the connection arrives at the host port on which `sshd` is running. The daemon forwards the data through the SSH tunnel to an arbitrarily configurable port on the client. The syntax is as follows:

```
ssh remoteuser@remotehost ⮐
  -R remoteport:localhost:localport
```

Dynamic Forwarding with SSH

Thanks to the dynamic option `-D`, an SSH client can act like a SOCKS server (SOCKS proxy) and automate access to remote servers via a secure SSH tunnel. Dynamic port forwarding is also useful if you want to access a service on your home or enterprise server via a secure tunnel from a public WLAN hotspot.

You do need a matching SOCKS client for the service, which is the case for a web browser. In the browser's connection options, enter the local SSH client as the SOCKS proxy with a freely configurable port number. Because transmissions between the client computer and the WLAN router are not encrypted on a WLAN hotspot and can thus be read by any network sniffer, this approach is always useful if you need to transmit login credentials or other sensitive data to access your own server via the web. You can set up the tunnel to the remote `sshd` as follows:

```
ssh -D port user_name@remotehost
```

Now you only need to enter the local SSH client and the port as the SOCKS proxy for your browser; Figure 8 shows an example with Firefox.

Again, the `-N` option is useful to tell the client to open the tunnel but prevent

it from starting a shell on the server. The use of an SSH tunnel as a SOCKS proxy is pretty close to a full-fledged VPN; the only difference being that, although the data traffic from the applications you use runs through the SSH tunnel set up in your proxy settings, the DNS requests don't; this means the SSH tunnel is not suitable for tasks such as anonymous surfing.

If you want to tunnel other programs or services besides HTTP via SSH, Linux users should be aware that some programs do not support SOCKS proxies. If this is the case, you can install the `tsocks` wrapper on Linux and add the following `/etc/socks/tsocks.conf` configuration file:

```
server = localhost
server_port = 12222
server_type = 4
```

VPN over SSH: A TUN/TAP Tunnel

OpenSSH Version 4.3 or later provides a `-w` option that lets users set up a VPN as a Layer 2 or Layer 3 tunnel with virtual network adapters (TUN/TAP interfaces). However, this technique involves the administrator's server and client-side loading of the kernel modules for the TUN/TAP devices using `modprobe`. In other words, the approach is not useful for ad hoc scenarios such as an Internet cafe.

To set up the required virtual network adapters, enter the following on the client:

```
ifconfig tun0 10.0.2.1 netmask ↩
  255.255.255.252
```

The server configuration looks like this:

```
ifconfig tun0 10.0.2.2 netmask ↩
  255.255.255.252
route add -host target_host dev eth0
```

After you enter these commands, the user on the client can establish a VPN tunnel:

```
ssh -l user -p sshd-port -w0:0 ↩
  target-Host
```

Additionally, you need to enable the `sshd` configuration on the Linux server by setting the `PermitTunnel yes` option.

See the article on SSH tunnel connections elsewhere in this issue for more on SSH tunneling alternatives.

Drilling Holes in the Firewall

The methods discussed thus far clearly demonstrate the power of SSH, especially for port forwarding. All of the examples have been for friendly use. But SSH can also be used for unfriendly activities. For example, if your own firewall blocks SSH port 22, but you need an option for securely accessing the data on your office machine, you can apply the following trick, which again relies on remote port forwarding.

An OpenSSH server has to be running on your company server, even if the firewall prevents requests for port 22. To enable an SSH server, you can install both the `openssh-client` and the server components for SSH in the form of the `ssh` package. If neither the SSH client nor the server were in place previously, Debian and Ubuntu administrators can also launch `tasksel` at the command line and then select the `OpenSSH server` package group.

To follow this example, you need to be the administrator of the remote server, or be authorized for the experiments, because you will very likely infringe on your company's security policies. Back home, you also need to make sure your local machine accepts SSH connections; you might need to install the server packages as well as the OpenSSH client for this.

Additionally, your home computer needs a DnyDNS address to make it remotely accessible. Once the SSH server is running on your home computer, open a remote port-forwarding SSH tunnel for port 22 on the enterprise server to `sshd` on your home computer, but using the publicly accessible DynDNS address.

```
ssh user_name@home_computer -R ↩
  4444:home_computer:22
```

Then, just leave the tunnel as is and run `ssh user_name@home_computer -p 4444` through this tunnel to open another tunnel to the enterprise server using the port number specified for remote port forwarding (4444).

This example demonstrates the principle of reverse tunneling, although this technique might not succeed in the current scenario because the company firewall will not let requests for port 22 pass. Most enterprise firewalls are configured this way, but they will allow requests for port 80 (HTTP). Nothing can stop you from letting your SSH server at home listen on a different port – for example, port 80. But, once again, please remember that you might be violating company policy or even infringing on the law in some jurisdictions.

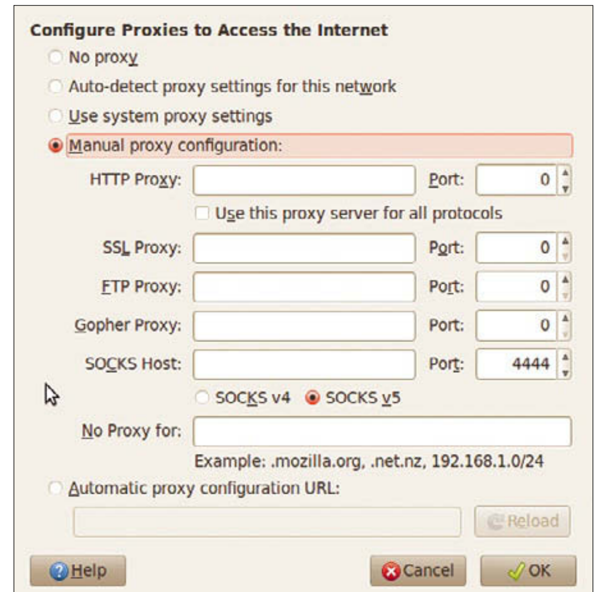


Figure 8: In dynamic port forwarding, the SSH client acts as a SOCKS proxy. The user only needs to add the localhost and the correct port number to the browser settings.

Conclusions

SSH can do much more than just open a remote shell, and many system administrators haven't yet discovered its full potential. If you take the time to investigate the options, you might just discover that SSH makes proprietary VPN solutions obsolete for secure file transfers. In fact, assuming you have the latest versions of the SSH server and client, SSH offers state-of-the-art security with authentication and encryption for a wide range of remote access scenarios. ■■■

INFO

[1] OpenSSH website: *http://openssh.org*

[2] FUSE: *http://fuse.sourceforge.net*

Perl script tunnels mail traffic on demand

Tunnel Vision

Instead of running a local mailserver, a Perl daemon listens to outgoing SMTP requests and drills a temporary SSH tunnel to a remote SMTP server on demand. *By Mike Schilli*

alemba_arts, Fotolia.com

My Internet service provider normally handles the job of shoveling data packets around fairly well. But if something fails, I often get a script-reading ignoramus on the hotline who totally ignores elementary, logical principles. They attempt to put the blame on the user instead of telling the trained system administrators who work with them that the problem is obviously on their side. Once, when I called to complain about a slow DNS server, somebody actually asked if my DSL modem was on the floor or in the bookcase.

The Age of Spam

Just recently, I had a problem with their SMTP server and wanted to avoid the frustration of calling my provider. I don't send much in the line of email from my home desktop, but when I do, I expect it to reach its destination. For example, if there's a power failure, my UPS cuts in, a fact that is noticed by Nagios, which in turn quickly sends me an email.

Of course, I could turn to my hosting provider instead, a private company who doesn't operate as a government-protected quasi-monopoly. Their SMTP server is very reliable, but in the age of spam, they won't accept mail from unknown IPs. Because the provider offers SSH access, I could drill a tunnel like

```
ssh -L 1025:localhost:25 ⮐
    mschilli@host.provider.com
```

from my local port 1025 to the SMTP port (25) on the hosted computer. From the point of view of the computer in my hosting provider's farm, it would look like the request came from the leased shared host Web server.

Dynamic Drilling

Budget hosting providers will probably not want scrooges like myself keeping SSH tunnels open day and night without typing something into their leased websites; but, if I only drill the tunnel when I want to send out email and then tear it down afterward, they'll probably be okay with it. To implement this, the `minimail` daemon, written in Perl, listens for requests from local mail clients on the SMTP port (25). The clients are blissfully unaware of the complexity behind this, they'll be under the impression that they're talking to a local mailserver.

The daemon accepts the request, opens a tunnel on local port 1025 to port 25 of the hosting provider, then waits for the connection to come up. For the local mail client, this just looks like a fairly slow mail server. The daemon then shoves the request lines from the client (local port 25) to local port 1025. Packets are entering the tunnel and pushed through to port 25 on the provider's side (Figure 1). Return packets, arriving back through the tunnel are forwarded by the

daemon to the local client, which completes the impression that it is indeed talking to the local SMTP server.

If multiple requests to send mail occur in quick succession, it doesn't make sense to break down and build up the tunnel again; to handle this case, the daemon leaves the tunnel up for 10 seconds after the last client has bailed out. To keep this looking human in the host's logs, the script adds a random number between 0 and 25 seconds to the wait.

To Root or Not to Root?

To allow the daemon in Listing 1 to `bind()` the SMTP port (25), it must run as the root user; to mitigate the security risk this implies, the daemon drops these privileges later on. A program launched with `sudo` has the `SUDO_USER` environment variable set to the account that ran the sudo command. The script drops its privileges and changes the effective user ID to this non-privileged account. The `sudo_me()` command in line 15 from the CPAN Sysadm::Install module checks if `root` ran the script and, if not, uses `sudo` to change things.

The CPAN App::Daemon module exports the `daemonize()` function which lets the script act as a daemon and process the `minimail start|stop` commands. It will put itself into the background after running through the start sequence – only the logfile reveals what

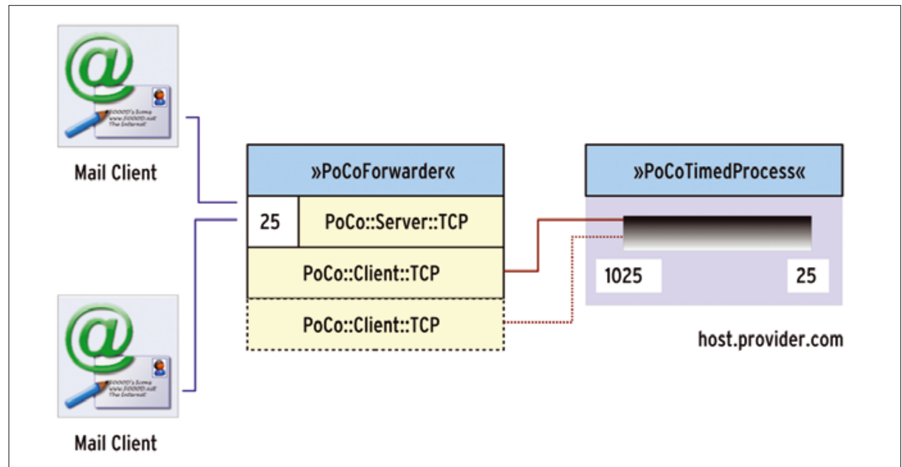


Figure 1: The mail client talking to port 25 on the forwarder, whose TCP client session talks to the tunnel.

the daemon is currently doing. The Log4perl logfile is set by the `-l` option or, programmatically, via the `App::Daemon::logfile` variable, as shown in line 18. If the daemon is launched in the foreground with the `-X` option, the log output is sent to `Stderr` instead.

The `BEGIN` block in lines 14-23 makes sure that the POE module in line 25 is not loaded until the process has been daemonized (line 22). This is important, so a helpful soul from the POE mailing list told me; otherwise, POE won't clean up the child processes it creates later on.

Because App::Daemon also offers a feature for dropping root privileges, line 16 of the module assigns a value of `root` to the `$as_user` variable and thus leaves the security switch to the script, which handles it after binding the daemon to port 25 in the forwarder code, starting at line 48.

POE to the Rescue

Writing your own network daemon normally costs plenty of blood, sweat, and tears, but, thankfully, CPAN offers a number of POE components you just need to glue together. For example, `minimail` creates the `PoCoForwarder` port forwarder from the POE::Component::Client::TCP and POE::Component::Server::TCP components. It binds with the local `$port_from` port and forwards anything

LISTING 1: minimail

```
01 #!/usr/local/bin/perl -w
02 #############################
03 # minimail - SMTP daemon
04 #        auto-opening tunnels
05 # Mike Schilli, 2010
06 # (m@perlmeister.com)
07 #############################
08 use strict;
09 use Sysadm::Install qw(:all);
10 use App::Daemon
11   qw(daemonize);
12 use Log::Log4perl qw(:easy);
13
14 BEGIN {
15  sudo_me();
16  $App::Daemon::as_user =
17    "root";
18  $App::Daemon::logfile =
19    "/var/log/minimail.log";
20  $App::Daemon::loglevel =
21    $INFO;
22  daemonize();
23 }
24
25 use POE;
26 use PoCoForwarder;
27 use PoCoTimedProcess;
28
29 my $port_from   = 25;
30 my $port_to     = 25;
31 my $tunnel_port = 1025;
32 my $real_smtp_host =
33   'host.provider.com';
34
35 my $process =
36   PoCoTimedProcess->new(
37  heartbeat => 10,
38  timeout   => int(rand(25)) +
39    10,
40  command => [
41   "ssh", '-N', '-L',
42   "$tunnel_port:" .
43    "localhost:$port_to",
44   $real_smtp_host
45  ],
46  );
47
48 my $forwarder =
49  PoCoForwarder->new(
50  port_from  => $port_from,
51  port_to    => $tunnel_port,
52  port_bound => sub {
53   INFO "Dropping privileges";
54   $< = $> = getpwnam(
55    $ENV{SUDO_USER});
56  },
57  client_connect => sub {
58   $process->launch();
59  },
60  );
61
62 $process->spawn();
63 $poe_kernel->run();
```

that arrives there to the `$tunnel_port` – and vice versa. This is no trivial matter because multiple mail clients can use the local port at the same time and would need to be served in parallel.

The second component, that is, `PoCo-TimedProcess`, uses the `launch()` method to start a process like the tunnel for a certain amount of time or extends its lifetime if it is already running. Every time the forwarder discovers a newly docked client, it calls the `launch()` method in the `client_connect()` callback (line 58). The method calls the `ssh` command in lines 41-44. The call to

```
ssh -N -L ➘
  1025:localhost:25 host.provider.com
```

thus connects to the host at `host.provider.com` via the encrypted SSH protocol, logs in when it gets there, and, thanks to the `-N` option, doesn't start an interactive shell but just hangs around forwarding datastreams back and forth.

Port 1025 is the desktop-side end of the tunnel; however, `localhost` in the `ssh` command above refers to `host.provider.com`, because the SSH session is logged in there at this point. The `25` following the colon is the SMTP port on the hosted machine. If the username on the hosted machine is not the same as on the desktop, the call needs to add a valid account name like *mschilli@host.provider.com* to tell SSH which to use.

Component Glue

What happens behind the scenes in the two POE components? Figure 1 shows the diagram with the server and client components and the port numbers they use. The port forwarder TCP server listening on port 25 winds up a TCP client session for each client to connect them to the tunnel independently.

LISTING 2: PoCoForwarder.pm

```
001 #############################
002 # POE Port Forwarder
003 # Mike Schilli, 2010
004 # (m@perlmeister.com)
005 #############################
006 package PoCoForwarder;
007 use strict;
008 use Log::Log4perl qw(:easy);
009 use
010   POE::Component::Server::TCP;
011 use
012   POE::Component::Client::TCP;
013 use POE;
014
015 #############################
016 sub new {
017 #############################
018  my ($class, %options) = @_;
019
020  my $self = {%options};
021
022  my $server_session =
023   POE::Component::Server::TCP
024    ->new(
025   ClientArgs => [$self],
026   Port => $self->{port_from},
027   ClientConnected =>
028     \&client_connect,
029   ClientInput =>
030     \&client_request,
031   Started => sub {
032    $self->{port_bound}->(@_)
033      if defined
034       $self->{port_bound};
035   },
036    );
037
038  return bless $self, $class;
039 }
040
041 #############################
042 sub client_connect {
043 #############################
044  my (
045   $kernel,  $heap,
046   $session, $self
047    )
048    = @_[
049    KERNEL,  HEAP,
050    SESSION, ARG0
051    ];
052
053  $self->{client_connect}
054    ->(@_)
055    if defined
056     $self->{client_connect};
057
058  my $client_session =
059   POE::Component::Client::TCP
060    ->new(
061   RemoteAddress =>
062     "localhost",
063   RemotePort =>
064     $self->{port_to},
065   ServerInput => sub {
066    my $input = $_[ARG0];
067
068      # $heap is the
069      # tcpserver's (!) heap
070    $heap->{client}
071      ->put($_[ARG0]);
072   },
073   Connected => sub {
074    $_[HEAP]->{connected} = 1;
075   },
076   Disconnected => sub {
077    $kernel->post($session,
078     "shutdown");
079   },
080   ConnectError => sub {
081    $_[HEAP]->{connected} = 0;
082    $kernel->delay(
083     'reconnect', 1);
084   },
085   ServerError => sub {
086    ERROR $_[ARG0]
087      if $_[ARG1];
088    $kernel->post($session,
089     "shutdown");
090   },
091    );
092
093  $heap->{client_heap} =
094    $kernel->ID_id_to_session(
095   $client_session)
096    ->get_heap();
097 }
098
099 #############################
100 sub client_request {
101 #############################
102  my ($kernel, $heap,
103   $request) =
104    @_[ KERNEL, HEAP, ARG0 ];
105
106  return if
107     # tunnel not up
108     # yet, discard
109    !$heap->{client_heap}
110     ->{connected};
111
112  $heap->{client_heap}
113    ->{server}->put($request);
114 }
115
116 1;
```

The class expects the port_from port (the one on which the server is listening to client requests), the port_to port (the desktop end of the tunnel), and two callback routines as parameters. The component jumps to the subroutine reference stored in port_bound once the server has bound to port 25 and can thus drop its root privileges.

When dropping root privileges, it is important to do it in the right order for effective and real user IDs; otherwise, the daemon could reestablish its root privileges later [2]. With multiple parallel threads, PoCoTimedProcess internally would have to prevent a race condition launching the tunnel twice. In the one-process, one-thread environment that POE provides, a simple variable check without locking is fine – robust, easy to code, and easy to understand when you come back to the program years later!

The second forwarder callback, client_connect, is accessed whenever a mail client docks on port 25. The PoCoTimedProcess component's launch() method, which is executed in the callback, then sets up the tunnel if it doesn't exist. Internally, PoCoForwarder provides a PoCo::Client::TCP type POE component for each client connection, and each connects to the desktop tunnel port. In other words, although PoCo::Server::TCP can manage any number of clients, you need to deploy a separate PoCo::Client::TCP component for each.

Closures: Confusingly Elegant

Line 32 in Listing 2 shows how the component runs the port_bound callback. The POE TCP server created in line 24 enters the Started state after launching successfully. PoCoForwarder retrieves the subroutine reference defined by minimail from the $self object hash and calls it. The callback code defined in Minimail handles everything else.

Note that $self is not in the scope of the handler assigned to the Started state. Instead, it comes courtesy of the PoCoForwarder class's new() constructor; however, the subroutine mutates to a closure that includes the lexical $self variable and thus remains valid after leaving the scope of the constructor (but only within the callback).

On the other hand, the Client-Args parameter in line 25 makes sure the server component provides the $self object hash as an argument, ARG0, if it enters the client_connect() callback function. In line 54, the component runs the client_connect callback set by the main script earlier, which launches the tunnel process. Note the timing problem that occurs here because it is difficult to predict how long the tunnel will take to come up. This means that our newly fired up TCP client might try to bind to a port later when no one is listening in.

In this case, it isn't an issue. The TCP client enters the ConnectError state (line 80), which schedules a reconnect event for one second later in POE's todo list

LISTING 3: PoCoTimedProcess.pm

```
001 #############################
002 # POE Timed Process
003 # Launcher Component
004 # Mike Schilli, 2010
005 # (m@perlmeister.com)
006 #############################
007 package PoCoTimedProcess;
008 use strict;
009 use warnings;
010 use POE;
011 use POE::Wheel::Run;
012 use Log::Log4perl qw(:easy);
013
014 #############################
015 sub new {
016 #############################
017   my ($class, %options) = @_;
018
019   my $self = {%options};
020   bless $self, $class;
021 }
022
023 #############################
024 sub launch {
025 #############################
026   my ($self) = @_;
027
028   $poe_kernel->post(
029     $self->{session}, 'up');
030 }
031
032 #############################
033 sub spawn {
034 #############################
035   my ($self) = @_;
036
037   $self->{session} =
038     POE::Session->create(
039    inline_states => {
040     _start => sub {
041      my ($h, $kernel) =
042        @_[ HEAP, KERNEL ];
043
044      $h->{is_up} = 0;
045      $h->{command} =
046        $self->{command};
047      $h->{timeout} =
048        $self->{timeout};
049      $h->{heartbeat} =
050        $self->{heartbeat};
051      $kernel->yield(
052       'keep_alive');
053      $kernel->yield(
054       'heartbeat');
055     },
056     sig_child => sub {
057      delete $_[HEAP]->{wheel};
058     },
059     heartbeat  => \&heartbeat,
060     up         => \&up,
061     down       => \&down,
062     keep_alive => sub {
063      $_[HEAP]->{countdown} =
064        $_[HEAP]->{timeout};
065     },
066     closing => sub {
067      $_[HEAP]->{is_up} = 0;
068     },
069    }
070     )->ID();
071 }
072
073 #############################
074 sub heartbeat {
075 #############################
076   my ($kernel, $heap) =
077     @_[ KERNEL, HEAP ];
078
079   $kernel->delay("heartbeat",
080    $heap->{heartbeat});
081
082   if ($heap->{is_up}) {
083    INFO
084 "Process is up for another ",
085      $heap->{countdown},
086      " seconds";
087
```

with the `delay()` POE kernel function. This game can go on for a couple of rounds, but the tunnel will come up eventually. The TCP client then binds the port, which is now working, and can enter the `Connected` state as of line 73.

The Tunnel is Ready

If Minimail sends a command, the TCP server branches to the `client_request` state and thus to the handler (lines 100-114), which checks that the tunnel is already up and ignores the client command if the connection is down. The SMTP protocol stipulates the server has to start the communication with a greeting. A well-behaved client will not start to talk until the server says hello, which only happens if the tunnel is up. With other protocols (e.g., HTTP), it is different; in this case, the forwarder would have to buffer the client commands until the connection was up, then forward them in lieu of the client in a bundle.

If the tunnel is ready, the heap variable `connected` is `1` in the `Connected` state handler. To forward the message to the tunnel, line 112 retrieves the saved TCP client heap and pulls out its `server` entry, whose `put` method is then used to forward the request to the tunnel entry the client docked onto earlier. Note that `client_request()` is a server session callback that knows nothing about the client's heap or the client, which is running in another session. The `client_heap` heap variable, set in line 93 in the server session, solves this problem.

When messages come back out of the tunnel, the TCP client switches to the `ServerInput` state in line 65, which then uses `put()` on the client reference stored on the heap, to return text to Minimail. If Minimail disconnects from the TCP server, the server enters the `Disconnected` state, and the handler sends a shutdown event to the running session (line 77), finally interrupting the client server connection.

Processes with Countdown

Handlers in the `PoCoTimedProcess.pm` component (Listing 3) set up and break down the tunnel. When `minimail` uses `spawn` (line 62) to launch the process timer's POE session, its first course of action is running the `_start` handler defined in `PoCoTimedProcess.pm` (line 40). The handler in turn uses a closure to extract all the critical parameters, such as `heartbeat` (check frequency for a timeout), `timeout` (number of seconds until tunnel breakdown), and `command` (the SSH command for setting up the tunnel) from the `self` object hash and stores them on the session's own heap. It then sets two events for processing by the POE kernel at a later stage: `keep_alive` and `heartbeat`. The former resets the heap `countdown` variable to the maximum value in seconds to keep a tunnel open, which is defined in `timeout`. Additionally, POE calls the `heartbeat` event at regular intervals, thanks to the `delay` method in line 79, every time the number of seconds defined in the heap `heartbeat` variable has elapsed.

The tunnel is closed at first, but as soon as the `launch()` method triggers the `up` event and POE activates the matching `up` handler (line 103), a POE::Wheel::Run object (line 119) fires up the SSH tunnel process. The handlers for the Unix `INT` and `TERM` signals defined in lines 134 and 136 ensure that the `minimail` process will tear down an open tunnel if the main script is killed unexpectedly.

LISTING 3: PoCoTimedProcess.pm (part2)

```
088   $heap->{countdown} -=
089     $heap->{heartbeat};
090
091   if (
092    $heap->{countdown} <= 0)
093   {
094    INFO
095 "Time's up. Shutting down";
096    $kernel->yield("down");
097    return;
098   }
099  }
100 }
101
102 ##############################
103 sub up {
104 ##############################
105  my ($heap, $kernel) =
106    @_[ HEAP, KERNEL ];
107
108  if ($heap->{is_up}) {
109   INFO "Is already up";
110   $_[KERNEL]
111     ->yield('keep_alive');
112   return 1;
113  }
114
115  my ($prog, @args) =
116    @{ $heap->{command} };
117
118  $heap->{wheel} =
119    POE::Wheel::Run->new(
120   Program     => $prog,
121   ProgramArgs => [@args],
122   CloseEvent  => "closing",
123   ErrorEvent  => "closing",
124   StderrEvent => "ignore",
125    );
126
127  my $pid =
128    $heap->{wheel}->PID();
129  INFO "Started process $pid";
130
131  $kernel->sig_child($pid,
132   "sig_child");
133  $kernel->sig(
134   "INT" => "down");
135  $kernel->sig(
136   "TERM" => "down");
137
138  $_[KERNEL]
139    ->yield('keep_alive');
140  $heap->{is_up} = 1;
141 }
142
143 ##############################
144 sub down {
145 ##############################
146  my ($heap, $kernel) =
147    @_[ HEAP, KERNEL ];
148
149  if (!$heap->{is_up}) {
150   INFO
151     "Process already down";
152   return 1;
153  }
154
155  INFO "Killing pid ",
156    $heap->{wheel}->PID;
157  $heap->{wheel}->kill();
158  $heap->{is_up} = 0;
159  $kernel->sig_handled();
160 }
161
162 1;
```

```
mybox> telnet localhost 25
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

Figure 2: When a message needs to be sent, Minimail needs to open the tunnel for the first request …

```
mybox> telnet localhost 25
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 host.provider.com    ESMTP
```

Figure 3: … then the SMTP server at the other end of the tunnel will respond within about one or two seconds …

```
mybox> telnet localhost 25
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 host.provider.com    ESMTP
HELO apple.com
250 host.provider.com
MAIL FROM: s.jobs@apple.com
250 2.1.0 Ok
```

Figure 4: … and the client can then exchange SMTP commands as if connected directly. The server thinks it is talking to a local client.

Once the tunnel has reached its maximum lifetime, line 96 triggers the `down` event and the matching handler (line 144) sends a kill signal to the `ssh` process. To let other handlers know that the tunnel no longer exists, `down()` sets the `is_up` variable to `0`. This completes the processing of the triggering signal; the call to `sig_handled()` in line 159 prevents the POE kernel from acting on it as well, which would be undesirable because the kernel's default action on these signals is to terminate the daemon.

To prevent the killed process mutating into a zombie, joining a growing army of other zombies, and finally bringing the computer to its knees, line 131 defines a `sig_child` handler, which reaps the dying process and then enters the `sig_child` state of the POE session, defined in line 56. This helps POE give the dying tunnel its last rites (internally, via `waitpid()`) and prevents it from going to zombie hell. The handler finally deletes the last remaining reference to POE::Wheel. If POE figures out it has nothing left to do, it'll neatly fold up the kernel.

Keys Instead of Passwords

Because a daemon can't use an interactive password dialog to identify itself, the `ssh` tunnel command requires the user to create a keypair:

```
ssh-keygen -t rsa
```

The keys will typically be stored in the `id_rsa` (Private Key) and `id_rsa.pub` (Public Key) files in the `.ssh` directory below the user's home directory.

To make sure the hosting service provider lets the daemon connect to it, the user has to push the public key created with the `no passphrase` option to the server. This involves appending the local content of the `id_rsa.pub` file to the `.ssh/authorized_keys` file on the hosting server. If you then enter the ssh tunnel command in Minimail manually (without the `-N` option), you should be logged in to the hosting server without being asked for your password.

```
mybox> tail -f /var/log/minimail.log
2010/04/18 23:40:56 Process ID is 17415
2010/04/18 23:40:56 Written to /tmp/mailer.pid
2010/04/18 23:40:56 Dropping privileges
2010/04/18 23:41:45 Starting program
2010/04/18 23:41:46 Process is up for another 23 seconds
2010/04/18 23:41:46 Is already up
2010/04/18 23:41:56 Process is up for another 23 seconds
2010/04/18 23:42:06 Process is up for another 13 seconds
2010/04/18 23:42:16 Process is up for another 3 seconds
2010/04/18 23:42:16 Time's up. Shutting down
2010/04/18 23:42:16 Shutting down process
mybox>
```

Figure 5: The daemon logs critical events.

Trial Run with Telnet

The Telnet command in Figure 2 with `localhost` and port 25 discovers whether the mail server that was launched by `sudo minimail start` really works. If the daemon tunnel is down, Minimail will delay the response by one or two seconds until the server provider-side responds and then patch through to the SMTP server on the other end (Figure 3).

If you speak some SMTP, you can try out a couple of tricks (for test purposes only, of course – Figure 4). The daemon will busily take note of this in the `/var/log/minimail.log` logfile (Figure 5). It will not store the mail headers or text for data protection reasons.

While running tests with the `telnet` command, you can get out of a hung session caused by a server not releasing the client by pressing the keyboard shortcut Ctrl +], which takes Telnet down into a shell that you can terminate by pressing *q*.

Waiting for a Power Failure

To launch the Minimail server automatically every time you boot your machine, you need to add

```
SUDO_USER=mschilli /path/to/minimail
```

on Ubuntu to the `/etc/init.d/minimail` file, which you might need to create, then make the file executable with `chmod +x` and finally call

```
sudo update-rc.d minimail defaults 80
```

to add the script to the boot process. When the power returns, the new mail server boots automatically and makes sure it is ready to take messages once Nagios reports that power has been restored and disaster averted. ■■■

INFO

[1] Listings for this article: *http://www.linuxpromagazine.com/Resources/Article-Code*

[2] Dropping privileges, but properly: *http://perlmonks.com/?node_id=833950*

AUTHOR

Mike Schilli works as a software engineer with Yahoo! in Sunnyvale, California. He is the author of *Goto Perl 5* (German) and *Perl Power* (English), both published by Addison-Wesley, and he can be contacted at *mschilli@perlmeister.com*. Mike's homepage can be found at *http://perlmeister.com*.