

ENTERPRISE OPERATIONS MANAGEMENT

THE SSH PROTOCOL

Duncan Napier

INSIDE

A Brief History of the Secure Shell Protocol and Its Implementations; How SSH Works; Installation; Configuration; Using SSH

INTRODUCTION

SSH (Secure Shell) is a protocol for running secure network services over an insecure network. The protocol serves as the basis for many implementations of SSH that are now widely available as commercial or non-commercial products. These products encompass a wide variety of platforms, including virtually all flavors of UNIX, DOS/Windows, and Macintosh operating systems as well as many other environments, including OpenVMS, BeOS, OS/2, PalmOS, and Java to name a few. SSH runs on top of TCP/IP and is generally invisible to the software applications layer. As a result, SSH can be made completely transparent to end users and does not require any additional user training.

The term “Secure Shell” originates from the early days of SSH in 1995, when Tatu Ylönen, a researcher at Helsinki University of Technology, wrote an application to facilitate secure, encrypted login access for UNIX hosts. SSH was originally designed as a secure drop-in replacement for rsh, UNIX remote shell, as well as remote login and file transfer applications such as telnet and rcp. These traditional UNIX services either bypass or offer limited user authentication (i.e., logon and password). They are vulnerable to IP spoofing or DNS table manipulation because they offer neither the means to authenticate the identity of hosts being logged on to, nor that of the login client. They also pass all data — including login names and passwords — in plaintext over connections that can be silently hijacked. Plaintext data can be easily modified or corrupted in transit without the knowledge of end users.

Conceptually, the SSH protocol runs as an application on top of the TCP/IP layer. The SSH protocol is implemented through separate client and server applications that authenticate and negotiate protocols before

PAYOFF IDEA

SSH (Secure Shell) is a protocol for authenticating, encrypting, and checking the integrity of information traversing TCP/IP-based networks. This article describes SSH, how to install it, and how to use it.

deciding whether to establish a secure connection. The protocol provides for cryptographic host and user authentication, strong encryption, integrity protection, and the simultaneous tunneling of multiple data channels. These features lend themselves to more than just secure remote logins. The feature set of SSH includes:

- Secure remote login
- Secure remote command execution
- Secure remote file transfer
- TCP port forwarding
- Cryptographic key control
- Authentication agents (including single sign-on)
- Configurable access control
- Data compression

Before discussing the above features in detail, it is useful to summarize the history of the SSH protocol to understand some of the rationale behind its design and its numerous implementations.

A BRIEF HISTORY OF THE SECURE SHELL PROTOCOL AND ITS IMPLEMENTATIONS

The first incarnation of the SSH protocol (known henceforth as SSH1) was developed by Tatu Ylönen in 1995. SSH1 was released on the Internet as free software with source code in July 1995. Ylönen documented the software as an Internet Engineering Task Force (IETF) Internet Draft that became the specification for the SSH1 protocol.

By the end of 1995, there were an estimated 20,000 users and Ylönen started SSH Communications Security (<http://www.ssh.com>) to commercially sell, support, and develop SSH. The freeware versions of SSH continued to be available, but SSH Communications Security imposed restrictions on the terms of their use. In 1996, SSH Communications Security introduced a new version of the protocol, SSH2. The IETF founded a public working group for standardization of the secure shell, SECSH, in 1996. By February 1997, the first Internet Draft for the SSH2 protocol was completed. In 1998, SSH Communications Security released an SSH2 implementation based on the SSH2 protocol.

The SSH2 protocol fixed some problems and shortcomings in SSH1 but these fixes rendered SSH2 incompatible with SSH1. SSH Communications Security also placed more restrictions of the use of its SSH2 product. These limitations may explain why after all this time, SSH1 is still probably the more widely used protocol at the time of writing. SSH Communications Security has since removed some of the restrictions on its SSH2 product and there has also been a steady proliferation of free and Open Source implementations that support both SSH1 and SSH2 protocols. An indication that SSH is entering the public mainstream is the SSH functionality offered with many of the staple terminal emulation packages, both

free (e.g., TeraTerm) and commercial (e.g., Van Dyke Secure CRT and April Systems Anita).

The many free, Open Source, and commercial implementations of the SSH1 and SSH2 protocols make it difficult to talk about the practical properties and features of SSH in general without biasing the discussion in the direction of a particular implementation. While virtually all implementations are faithful to the fundamental features of the protocol and can be made to interoperate smoothly, different implementations may have enhancements or idiosyncrasies that are unique to a particular product. In addition to the obvious incompatibility between SSH1 and SSH2 protocols, the author has found subtle and not-so-subtle problems, often relating to the interoperability of current products with many older releases.

To alleviate the problem in describing a fictitious “generic” implementation of the SSH protocols, the focus here is on the OpenSSH implementation of SSH. Open SSH is based on Björn Grönvall’s fix of Ylönen’s last free version of SSH1 (release 1.2.12), which Grönvall named OSSH. By early 2000, the OpenBSD (<http://www.openbsd.org/>) team had taken over OSSH and renamed the project OpenSSH using Grönvall’s work. OpenSSH has reached its current form through the work of Markus Friedl and others.

The decision to use OpenSSH in this discussion is based on a number of reasons, namely that OpenSSH:

- Has been ported to a wide variety of platforms encompassing all the major UNIX flavors, Windows/DOS, MAC OS X, and others
- Is free and has no patented algorithms in its source tree
- Supports both SSH1 and SSH2 in a single, seamless package
- Derives its code base from the OpenBSD project, which has an almost unparalleled track record in the development of secure, stable, and reliable software
- Releases tightly controlled upgrades on a regular and timely basis

OpenSSH is a work in progress and its feature list continues to grow with each successive release.

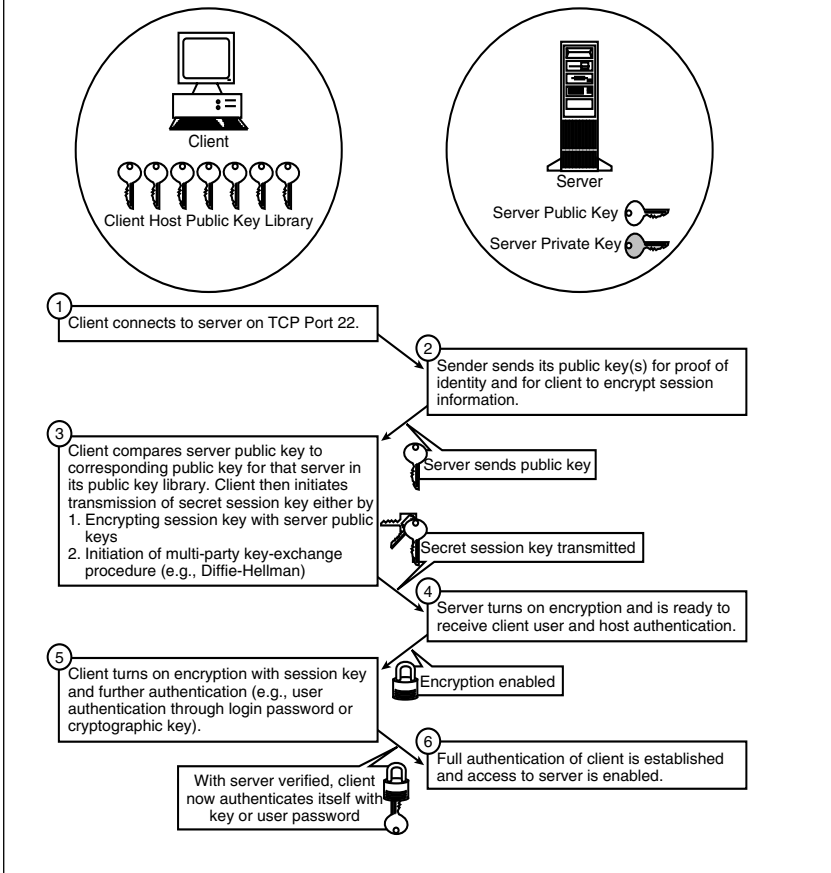
HOW SSH WORKS

[Exhibit 1](#) shows a conceptual schema of how SSH authentication and encryption works. The actual details of the implementation may vary, depending on the version of the protocol (SSH1 or SSH2), the specific implementation involved, and the user’s choice of authentication protocols. Essentially, the steps are as shown in [Exhibit 1](#) and describe below.

Step 1

The client host (left) connects to the server host (right), conventionally on TCP port 22. The server and client exchange the protocol versions

EXHIBIT 1 — Steps through which SSH Initiates, Authenticates, and Encrypts Communications



they support and, if compatible, continue the connection. Otherwise, the connection is terminated. The connection may be terminated if SSH1-only and SSH2-only implementations are involved because the two protocols are incompatible. The connection at this stage is unencrypted but uses check-bytes for integrity checking and plaintext-attack prevention on top of the TCP connection to ensure that the connection is not attacked or hijacked at this stage.

Step 2

The server sends its authentication information and session parameters to the client. This includes the server host's public key component of its own public/private key pair, as well as a list of the encryption, compression, and authentication modes that the server supports. In SSH1, an ad-

ditional server key is sent. Public key algorithms supported by most SSH implementations include RSA (Rivest-Shamir-Aldeman) and DSA (Digital Signature Algorithm).

Step 3

The client host checks the server host's public key against the client host's library of public keys. If this is the first time that this particular client and this particular server host have connected over SSH, the user is asked to verify the addition of new a server host public key to the client's public key library. Any future connections to that particular server host will now be verified against that public key reported from their first contact. Once the identity of the server is verified, a secret session key is generated.

In SSH1, the session key is encrypted with the server host public key and the server host's server public keys and sent back to the server host. Because the encrypted session key can only be decrypted by the server host's private key, the secret session key can be safely transmitted over the insecure link.

In SSH2, a multi-party key-exchange algorithm is used. Key-exchange algorithms allow for a shared secret to be agreed to and then securely transmitted between parties. The original and perhaps best-known key-exchange algorithm is the Diffie-Hellman algorithm.

One might wonder why a secret session key is required when public/private key pairs are available. The reason is that while many encryption methods use public key (or asymmetric) encryption, public key encryption is much slower than symmetric encryption methods in which all parties share a single, shared secret key. As a result, the secret key is transmitted and kept secret using public key encryption, but the actual scrambling of the data is done more speedily with the shared secret key called the session key. Public keys also just happen to be ideal for authentication and identification purposes. Secret key algorithms that SSH supports include 3DES (triple Data Encryption Standard), IDEA (International Data Encryption Algorithm), and Blowfish.

Note that at this point, the communication is still unencrypted.

Step 4

Once the secret session key is in the possession of both parties, encryption and integrity checking are turned on.

SSH1 uses a single session key for each session. For improved security, SSH2 can periodically change session keys, a process known as "re-keying." Session keys are typically stored only in memory and are not written to storage for security purposes. SSH1 uses the weak CRC-32 (Cyclic Redundancy Check) method for checking data integrity. SSH2 uses cryptographically stronger MAC (Method Authentication Code) integrity checkers.

Step 5

The client host and the client host user (“the user”) can now be authenticated to the server host without fear of the authentication and access transmission being intercepted or corrupted in transit. Methods by which the users authenticate themselves to the server include plaintext user login passwords, a user public key certificate (e.g., RSA, PGP), host public keys, Kerberos, Gauntlet’s TIS, Sun Microsystem’s PAM — the list goes on and on. Once the user is authenticated, appropriate access to the server and its services are granted to the user. The two machines are now connected through a secure, encrypted connection which can be used as a secure tunnel for all manner of services such as shell login sessions, UNIX XWindows sessions, POP3 or SMTP connection to a secure mailer, or a PPP-based VPN connection through an Internet gateway.

The above description highlights some of the many differences between the SSH1 and SSH2 protocols, and may give the reader some idea as to why they are incompatible. SSH2 also has a much more modular design, and features such as the authentication and encryption methods supported are not hardcoded into the protocol as was the case with SSH1. SSH2 has much more of an extensible “plug-in” philosophy, in which new protocols and methods can be used as drop-in replacements to the existing ones.

INSTALLATION

The SSH suite of programs is available as freeware, commercial, and open source software for a wide range of platforms (refer to list below). The author cannot vouch for all products and distributions, and one’s mileage with each of them may vary. There are good reasons to purchase vendor-supported products, but the focus here is on OpenSSH, which is generally compliant with the IETF standards for SSH (both SSH1 and SSH2).

OpenSSH has been ported to all major operating systems and platforms, and is available with source code free of charge to download and test. Releases of OpenSSH are written for the OpenBSD operating system. A second portability team at the OpenSSH project then typically follows up with a portable distribution that is identified with a “p”-designation in the version number. For example, OpenSSH 2.9 is the release of OpenSSH for OpenBSD and OpenSSH 2.9p1 is the first (as designated by the p1) portable version for all other operating systems. OpenSSH can be downloaded from the official OpenSSH Web site at <http://www.openssh.com>. A compatibility list is available at the official SSH Web site.

OpenSSH is also bundled into several OS distributions, including RedHat Linux, Debian Linux, Suse Linux, and Open and FreeBSD. Official and unofficial binary-package distributions of OpenSSH exist for such platforms as Linux (as rpms; <http://www.redhat.com>), Solaris

(<http://www.sunfreeware.com/>), AIX (<http://www.rge.com/pub/systems/aix/bull/>), and HP-UX (<http://eigen.ee.ualberta.ca/>). OpenSSH can be installed on WIntel (Windows 95/98/NT/2000) platforms as part of the Cygwin package (<http://www.cygwin.com/>).

Source code can be downloaded for compilation from the links on the official OpenSSH Web site. For the UNIX source distribution, OpenSSH requires that the zlib data compression library be installed, along with the OpenSSL library, from which OpenSSH draws its cryptographic components. GNUMake and a recent GCC compiler are also highly recommended. At the present time, all major UNIX flavors are supported, including Linux, Solaris, AIX, HP-UX, Digital UNIX/Tru64, SCO, IRIX, Open and FreeBSD, NeXT, and more. OpenSSH releases are initially OpenBSD. The package can be downloaded as a tarball (.tar.gz or .tar.Z), un-archived, and then run with the “configure,” “make,” and “make install” sequence. If one requires any (typically nonstandard) tweaks to the default compilation (e.g., if one’s system uses MD5 passwords or has the shadow password file disabled), one may need to manually change the compilation flags in the application Makefile. Note also that OpenSSH uses PAM (pluggable authentication modules) by default for login/password authentication. If the OS is configured to use PAM, one will need to make PAM aware of the SSH applications. This is commonly done by copying the appropriate PAM config file for SSH into the PAM configuration tree (usually requires copying a sample sshd.pam file from the /contrib directory of the distribution into /etc/pam.d/sshd). One’s distribution documentation is the ultimate source of information on the full installation procedure.

The default installation directories for OpenSSH in UNIX are typically /usr/local/bin for the binary executables and /usr/local/etc for host configuration and host cryptographic key storage. User keys and other specific user authentication information are usually stored in a subdirectory of that user’s home directory, ~/.ssh. For the Cygwin MS Windows distribution, the applications and config files reside in analogous locations in the Cygwin directory hierarchy.

The OpenSSH is built around two fundamental applications:

- *ssh*: a basic rlogin/rsh-like client program
- *sshd*: the server end that handles the client connections

The applications that control and regulate the cryptographic keys are:

- *ssh-keygen*: the cryptographic key generation tool
- *ssh-agent*: the authentication agent that stores private keys
- *ssh-add*: the tool that adds keys into the above agent

A suite of secure file transfer utilities is also included. These are:

- *scp*: file copy program that acts like rcp
- *sftp*: FTP-like program that works over the SSH1 and SSH2 protocols
- *sftp-server*: SFTP server subsystem

The server applications are usually set up to run as daemons in UNIX (and services in Windows) at start-up. If the SSH default TCP port 22 is used in a UNIX environment, then the server must be run as root because all applications that listen on ports 1023 and below require root privileges. One is free, however, to run SSH on any port of one's choosing, and this can be accomplished by changing the SSH configuration parameters.

CONFIGURATION

All server configurations are set using the `sshd_config` file. Global client configurations are set from the `ssh_config` file. Per-account server and client settings can also be implemented. SSH can be run as a stand-alone daemon (the most common) or, for the case of UNIX, through `inetd`, or as a service in WindowsNT/2000.

Typically, the first part of the server configuration file `sshd_config` is concerned with TCP/IP settings such as port number (Port 22), enable port forwarding (`AllowTCPForwarding`), timeouts (`IdleTimeout`), treatment of failed logins (`LoginGraceTime`), Reverse IP mappings (`RequireReverseMapping`), and numerous other settings. Access control settings (i.e., restricting/denying access to specific users) and authentication methods are also set here. The remaining configuration parameters are largely concerned with defining key generation, encryption algorithms, and protocols.

The `ssh_config` file controls the client connection, authentication, encryption methods, etc. An example of an `ssh_config` file is shown in [Exhibit 2](#). Many of these options and other options can be specified from the command line. For a list of command-line options, type `$ssh -h` at the command prompt.

To enable access to an SSH server running `sshd` behind a firewall, the access to and forwarding from port 22 on the firewall needs to be enabled. SSH clients initiate outbound connections from ports in the range 513 to 1023. This often causes problems when SSH is installed on firewalls that only allow outbound packets on ports numbered greater than 1023. The solution is to either run SSH with the `-P` option to use an unprivileged port on outbound client connections, or configure the firewall to allow ports 513 to 1023 for outbound connections to port 22 destinations.

USING SSH

Now consider some example applications of SSH.

EXHIBIT 2 — Example of an ssh_config Configuraiton File for an SSH Client

“Host *” indicates that SSH is allowed to connect to all hosts. The remaining parameters deal with forwarding, authentication modes, fallback to the rsh utility, the ability to run batch jobs by suppressing standard output, and host and key checking.

```
# Site-wide defaults for various options
```

```
Host *
ForwardAgent yes
ForwardX11 yes
RhostsAuthentication yes
RhostsRSAAuthentication yes
RSAAuthentication yes
PasswordAuthentication yes
FallBackToRsh no
UseRsh no
BatchMode no
CheckHostIP yes
StrictHostKeyChecking no
IdentityFile ~/.ssh/identity
Port 22
Protocol 2,1
Cipher blowfish
EscapeChar ~
```

Remote Login with Password Authentication

To start the client in order to log in to a remote system, at the shell prompt (or for Windows, in the DOS console), type

```
$ ssh duncan@server1.mycompany.com
```

where it is assumed that ssh is in the default path. This command opens a login session to the host server1.mycompany.com as user “duncan.” The login name is followed by an “@” optional, and if left out, the login name will default to the user name the client is currently running under. The host server then prompts for user duncan’s password. If cryptographic user identification is used instead of password authentication, the user types in the passphrase for their cryptographic key. This generates an authenticator that is encrypted with the user’s private key and verified with a copy of their public key that has previously been stored on the server. This greatly enhances security. First, passphrases are usually more difficult to attack with dictionary attacks and just an authenticator — not the passphrase — is sent over-the-wire. Authentication requires two components: the public/private user key pair that is retrieved from disk as well as the user’s manually entered passphrase.

Remote Copy

At the shell prompt, type

```
$scp /home/bob/movethis.txt \  
duncan@server1.mycompany.com:/home/duncan/tohere.txt
```

where the file `/home/bob/movethis.txt` on the local host is copied to the host `server1.mycompany.com` using user `duncan` as the login authenticator. Once again, the user will be prompted for a password. The reverse copy procedure would be

```
$scp duncan@server1.mycompany.com:/home/duncan/fromhere.txt \  
\home/bob/tohere.txt
```

To avoid retyping passwords, or to carry out operations between pairs of remote hosts that have mutual trusts established, one can resort to using SSH agents.

Remote Execution

To run a directory listing (`ls`) of the `/etc` directory on `server1.mycompany.com` using login `duncan`, type

```
ssh duncan@server1.mycompany.com ls /etc
```

Port Forwarding

The author has saved the best for last. One of the more intriguing features of SSH is port forwarding. Local port forwarding maps selected TCP ports or sockets from the present host to a remote host through an SSH encrypted tunnel. Because this is done at the TCP level, it is done transparently to the overlying applications. Remote port forwarding allows a host machine to forward connections to a given TCP port to another remote machine, effectively acting as a proxy, all through an encrypted tunnel. SSH has a very extensive XWindows/X11 forwarding system built into it. The primary security concern of XWindows users is that while X11 has nominally secure authentication, all traffic to remote XWindows is transmitted in plaintext and user keystrokes are vulnerable to capture. When set up with Xforwarding options enabled, SSH fully encrypts all XWindows traffic and enhances X client/server authentication. SSH also has a very flexible, general port forwarding mechanism which is discussed below.

To create an encrypted tunnel from port 2001 on your local host to port 23 (the telnet port) on a remote host, `server1.mycompany.com`, using user `duncan`'s account, simply type

```
$ssh -L 2001:localhost:23 duncan@server1.mycompany.com
```

and authenticate onto the server. A shell session will begin by default (running `ssh` with the `-N` option will initiate the connection but leave no shell). One has now opened a tunnel to `server1.mycompany.com`. Now one can telnet onto `server1.mycompany.com` by typing

```
$telnet localhost 2001
```

This connects one to TCP port 2001 on one's local host, which then forwards one to the telnet port (TCP 23) of the remote host `server1.mycompany.com`. If the telnet daemon is listening on port 23 of `server1.mycompany.com`, one will connect through an encrypted tunnel. One now has a fully encrypted telnet session.

To make `server1.mycompany.com` a general encrypting and port forwarding proxy for `myserver2.mycompany.com`, one can connect to `myserver1.mycompany.com` and run the following:

```
$ssh -R 2001:localhost:23 duncan@server2.mycompany.com
```

where the “-R” flag for remote port forwarding is used, instead of the “-L” for local port forwarding used in the previous example. Authenticate and log onto the server, establishing an SSH connection. Now anyone telnetting onto `myserver1.mycompany.com` will be forwarded over an encrypted link to `myserver2.mycompany.com`.

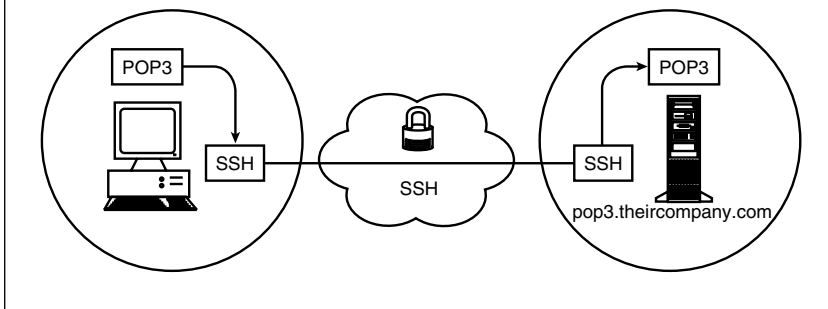
These examples seem a little contrived, but here are a couple of real-life situations that one might encounter.

Problem: You get your mail from a POP3 server that is on an untrusted network through a mail agent (e.g., Eudora or MS Outlook). You are concerned about your password being sniffed each time you log onto the server. You can create an encrypted tunnel to the POP3 server assuming you have access to a shell account on the POP3 server and the server has SSH installed on it. On your mail client, you type:

```
$ssh -L 110:localhost:110 accountname@pop3.mycompany.com
```

and authenticate. Port 110 is the conventional POP3 port. You now have an encrypted tunnel to the POP3 server. Note that to access a low-numbered port (i.e., a port number lower than 1023), you will require root access on the client. You now have to set your POP3 server to “localhost” and you should have encrypted access to your remote POP3 server. [Exhibit 3](#) shows schematically how port forwarding works in this case. Note that the TCP port 110 is redirected internally through SSH to the existing SSH tunnel.

EXHIBIT 3 — Constructing an Encrypted POP3 Connection with SSH Local Port Forwarding (shown with lock symbol)



Problem: You are at a remote site from your SMTP server and are unable to send outgoing e-mail because your SMTP e-mail gateway has anti-relaying rules and restricts access to hosts connecting from designated on-site IP/domains. Referring to [Exhibit 4](#), your laptop has been configured with a foreign IP/domain name (offsite.theircompany.com). You configure your workstation (onsite.mycompany.com) that is on site to relay SMTP traffic from your off-site machine to the on-site SMTP gateway server. On your workstation (onsite.mycompany.com), you type

```
$ssh -R 25:localhost:25 accountname@smtp.mycompany.com
```

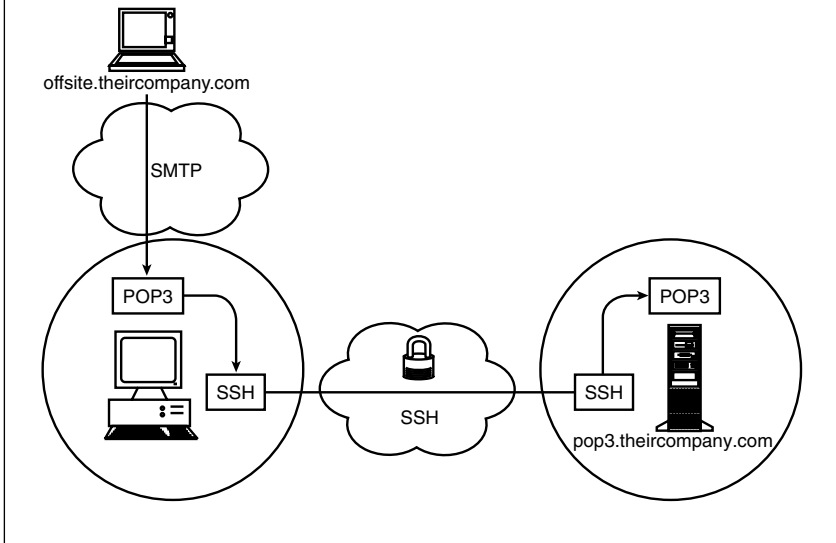
creating a tunnel from your workstation to the SMTP server. Your workstation now listens for and forwards SMTP traffic to the SMTP server smtp.mycompany.com. Because the workstation is on-site, it is permitted to use the SMTP server as an e-mail gateway. Set your SMTP server on your mailer to your workstation and you have a forwarded SMTP connection. *Note:* You may want to disable the port forwarding as soon as you no longer require an SMTP gateway because of the risk of turning your workstation into an SMTP open relay for mail spammers. [Exhibit 4](#) illustrates the layout of this remote port forwarding.

Port forwarding can be used to encrypt all manner of TCP/IP connections, and is used extensively with application such as AT&T Labs VNC, a free multiplatform terminal (<http://www.uk.research.att.com/vnc/>). Port forwarding is a useful tool to for creating secure connections from and between proxying firewalls. A functional VPN that uses PPP (the point-to-point protocol) over SSH can also be implemented.

CONCLUSION

This article has only scratched the surface of the capabilities and features of SSH. The author's hope was to introduce, those of you who were un-

EXHIBIT 4 — Creating a Remote SMTP Forwarder for an SMTP Server



familiar or only marginally familiar with SSH, to the potential of its capabilities. The applications of SSH in the real world are only limited by the imagination and ingenuity of those who choose to implement it. With SSH, it would appear that one of the “holy grails” of Internet privacy advocates, that of universal encrypted and authenticated communications, is well at hand.

Recommended Reading

1. *SSH. The Secure Shell. The Definitive Reference*, D. J. Barrett and Richard Silverman, Sebastopol CA: O'Reilly & Associates, 2001. For anyone who is interested in SSH, this book is a must-have.
2. *Securing Windows NT/2000 Server for the Internet*, Stefan Norberg, Sebastopol CA: O'Reilly & Associates, 2001. Contains useful information about bare-bones Cygwin ssh install on an NT server and use of VNC.
3. *Applied Cryptography*, 2nd edition, Bruce Schneier, New York: John Wiley & Sons, 1995. A comprehensive guide to cryptography algorithms and protocols. A floppy disk of source code is available from the author.

Duncan Napier is the owner/operator of Napier Systems Research, an Information Technology and Systems consultancy based in Vancouver, British Columbia. Duncan's educational background is in computer science and computational chemistry, and his company specializes in the design, configuration, and management of networks and network solutions. He can be reached by e-mail at napier@computer.org.