OpenSSH from the Administrator's Perspective – Part II

Tunnel Vision

The Secure Shell protocol is not only used to provide secure shells, but also to forward other types of TCP connection through a safe tunnel. But you need to get the key length and software version right, to ensure that SSH is really safe – and there are quite a few pitfalls to watch out for when using SSH across firewalls. BY ANDREW JONES

The Secure Shell, SSH, – the name promises safety, and has every right to do so. We introduced you to several secure services in the first part of this series [1]. One of the most interesting features is the facility to provide secure tunneling for any TCP protocol. We will be concentrating on that aspect of SSH in this part of the series.

First a word of warning to underline our statement in the intro: The SSH package is only secure if you use an up to date software version – vulnerabilities have been discovered time and again in OpenSSH. The developers have always resolved them in a timely fashion see *http://www.openssh.com/security.html*, but obsolete SSH servers are still a security risk.

You can use the *scanssh*[2] tool to search for obsolete SSH servers. The tool will scan individual hosts or complete subnets for SSH servers, and output the version number. Just pass the IP address and the corresponding subnet mask as arguments when launching the tool. The syntax for a single host is as follows:

```
scanssh 192.168.10.3/32
```

Figure 1 shows an example for a complete subnet, where the output contains the version number of the installed servers.

Figure 2 shows scanssh investigating individual hosts and shows that OpenSSH and Ssh.com both use their own software on their webservers. Scanssh does not require root priveleges – our only quibble is the fact that the tool does not use host or domain names, and can only locate SSH servers listening on port 22.

How long does an RSA key need to be?

There was a big scare with respect to the security of SSH and other crypto-programs in the middle of March this year. Surprisingly enough, it was not caused by a software vulnerability. It is claimed that 1024 bit RSA keys can be broken within a reasonable period and using affordable resources. To be more precise, the target was a PGP key, but the problem also affects SSH. Dan Bernstein, the author of the Qmail mail server, published his research into highly specialized parallel computers, designed for factoring integers, in the autumn of 2001 [3]. Afterwards, a discussion on the possible requirement to withdraw 1024 bit PGP keys ensued in the Bugtraq mailing list. Crypto guru Bruce Schneier added a few clarifying statements to settle this issue ([4],[5]). According to Bruce, the following key lengths can be considered as secure until the year 2005:

- Private persons: 1280 bit
- Corporate: 1536 bit
- Government: 2048 bit

Longer RSA keys are just a waste of time, according to Schneier. If you want to take Schneier's advice, but have been using a default 1024 bit RSA key for SSH so far, you will need to update your key. The following command creates a new RSA key for Version 2 of the protocol:

```
ssh-keygen -b 1280 -t rsa ➥
 -f ~/keynew/id_rsa ➥
```

Figure 1: The scanssh tool searches for SSH servers, here in the 129.168.10.0/24 subnet, and outputs the exact version

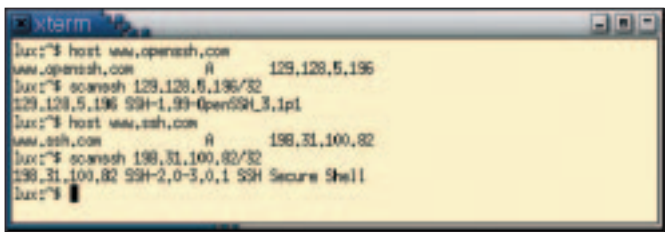

Figure 2: Scanssh investigating the SSH versions installed by the OpenSSH Project and SSH Communications Security on their own web servers

```
-C "1280 bit key for webmaster"
```

The RSA keypair (*id_rsa* and *id_rsa.pub*) with a key length of 1280 bits is written to the *~/keynew/* directory. You can use the *-f* flag to specify a target directory, if you want to avoid overwriting the existing keys under *~/.ssh/*. The *-C* option adds a comment to the Public Key, however, this is used only to distinguish the key more easily, and has no influence on functionality.

Tunneling: Forwarding TCP Ports

In addition to its original task of allowing secure remote logins, SSH can be used to secure almost any other protocol. Port forwarding allows you to relay TCP ports through the secure SSH connection. In this scenario SSH plays a similar role to a proxy, receiving connections at one end of the SSH channel and relaying them to the servers at the opposite ends.

SSH can perform two port forwarding variants: Local port forwarding and remote port forwarding. Local port forwarding is what you will need in most typical circumstances.

In this case, a connection that reaches a local (client-side) port, is forwarded across the secure SSH channel to a port on a remote server. You could also describe this technique as an egress tunnel. The syntax for this command is quite simple:

```
ssh login@remote_host ➥
 -L local_port:➥
 remote_host:remote_port
```

You can use forwarding to open up a secure POP3 connection to your mailbox, for example – in Part 1 of our series on OpenSSH[1] we already mentioned the potential vulnerability of POP3. After all, the POP client transmits the POP password to the server in the clear, which makes it easy to steal the password off the wire. To avoid this, you can of course tunnel the POP3 connection through SSH, even if your provider does not offer POP SSL:

```
ssh kh@pop.remote.com -C ➥
 -L 25025:pop.remote.com:110
```

Now, if we are so bold as to *telnet localhost 25025*, we can view the banner issued by the remote POP3 server. It works – and you don't need to be root. All you need to do now, is to set the POP client to localhost and port 25025, to allow it to poll mail as usual.

Figure 4 illustrates this procedure: The SSH command opens a normal SSH connection to the server, *pop.remote.com*, and also opens the tunnel. This forward will then remain active while you are logged on.

If a POP3 client (or a telnet command) now requests port 25025 on the client (i.e. on localhost), the SSH client will answer the connection request. SSH opens port 110 server-side and forwards any data.

You can also use a similar forward to secure the connection to a Webmin server (see the Boxout "Webmin Configures SSH Server"):

```
ssh kh@admin.remote.com -C ➥
 -L 33337:admin.remote.com:10000
```

Now the browser can talk to the Webmin server via the tunnel on *https://localhost:33337/*.

Lots of TCP based services can be forwarded and tunneled in this way – SMTP, IMAP, LDAP, or NNTP, but not FTP. FTP uses both a control channel and a data channel, whose ports are negotiated within the control channel. So, although it is trivial to secure the control channel, the data channel will still be in the clear. SSH provides *scp* and *sftp* as replacements.

Forwarding for Arbitrary Hosts

The kind of forwarding we have looked at so far relied on the hosts at both ends of the SSH connection having the application client and server software installed. But all of the programs involved, the application client, the SSH client, the SSH server and the application server could equally run on a host of its own. So forwarding can involve up to four hosts for a single instance.

This kind of off host forwarding can be used to create unusual network connections, and SSH tunnels, however, keep security in mind, when you are planning practical implementations. For one thing, only the connection between the SSH client and the SSH server is secured, and an attacker with access to the local port, but not to the target port on the server, can always use the tunnel to access a service that would normally be inaccessible.

To mitigate this danger, OpenSSH by default only allows connections from the local host to the forwarded port, although you can use the *-g* switch to change the default behavior. A sensible, practical application for off host forwarding would be a connection to a server where the user does not have an SSH account. In this case the user will need

Figure 3: The Web-based administration program, Webmin, provides a module for configuring SSH server. However, you will need to put some thought into this (see boxout)

an SSH server with a secure connection to the POP3 server in the vicinity of the target server. This might be the case if both servers are in the demilitarized zone behind a firewall, but the user requires remote access to the network:

```
ssh kh@ssh.remote.com -C ➥
 -L 25025:pop.remote.com:110
```

The forward is illustrated in Figure 5: An SSH tunnel is established between the client and *ssh.remote.com*. The mail client connects to its local port 25025. This connection is accepted by the SSH client, and the SSH server then provides the counterpart on port 110 between *ssh.remote.com* and *pop.remote.com*. Only the connection between the client and the SSH server is encrypted; a standard TCP connection is established between the SSH server and the POP3 server. From the viewpoint of the POP3 server, the connection originates from *ssh.remote.com* and not the client.

Reverse Forwarding

Remote port forwarding is the exact opposite of local port forwarding: The connection request is for a port on the host running the SSH server. Data is forwarded via the SSH channel to the client, where it is sent to an arbitrary port. You could also regard this as an ingress tunnel. The syntax is as follows:

```
ssh login@remote ➥
 -R remote_port:➥
 local_host:local_port
```

To determine what kind of port forward you need, you need to look at the scenario from the viewpoint of the TCP client application. If the TCP client application is local to the SSH client machine, local forwarding is the right option. If it is running on the remote SSH server machine, you should opt for remote port forwarding.

Not Always All Ports

OpenSSH permits TCP forwarding by default, and allows any free local and remote ports above 1024. Root is additionally permitted to forward local privileged ports below 1024.

A user with a genuine SSH login can also achieve the same goal without any support from SSH, using Netcat, (*nc*), for example. To do so, the user would need to connect a Netcat server and a Netcat client via an SSH shell pipe. The *AllowTcpForwarding no* directive in the server configuration file, *sshd_config*, is thus only partially effective.

Through the Firewall

One of the more interesting tasks for TCP forwarding involves transparently tunneling protocols through a firewall which permits SSH. A homeworker might need access to data stored on an Intranet web server, for example, although the server is only accessible on the company's internal LAN. A firewall prevents access from outside, but permits SSH logins on the gateway. Let us assume that the following computers are involved:

- Home desktop *hd*
- Office desktop *od*
- Gateway *gw*
- Internal web server *ws*

The user runs the following command on his home desktop:

```
ssh gw_login@gw -L 2001:ws:80
```

This opens an SSH session to *gw*, and at the same time forwards the local port 2001 to TCP port 80 (HTTP) on the internal web server *ws* via the SSH channel. This assumes that port 2001 on the local machine has not already been assigned to another service. Now the LAN web server can be accessed from the home desktop using the following URL: *http://localhost:2001*.

This variant is risky. Any user logged on to *hd* can use the open port, provided the tunneled session to *gw* is active. If the user also used the *-g* flag, port 2001 on *hd* will also be accessible to other hosts. If you cannot trust your users, you should be careful here, otherwise you might find them poking holes in your firewall. But it would be wrong to blame SSH for this: Any connection that goes through your firewall can be misused to tunnel other protocols.

SSH on SSH

Keeping to our home office example, let's assume that an employee would like to be able to log on to her office desktop

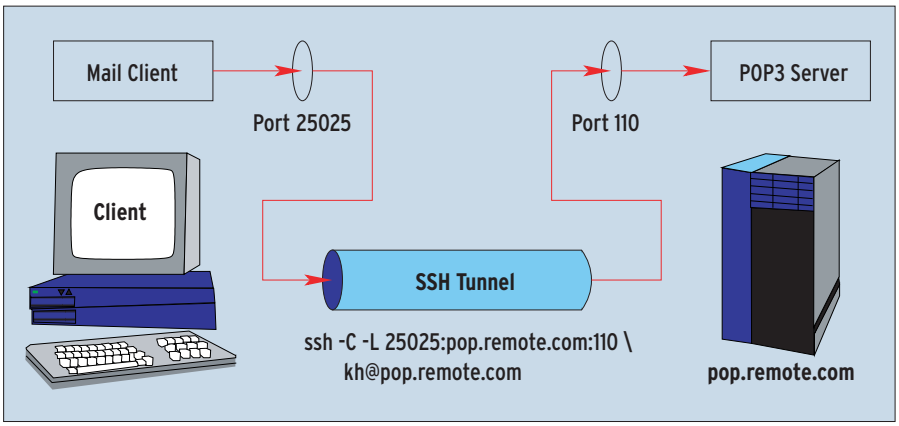

Figure 4: Local forwarding means that SSH will forward a connection that enters the client on port 25025 through the tunnel to the server, where it reaches its target, port 110

Listing 1: Allowing SSH to the Firewall

```
01 # SSH-Port
02 export SSH="22"
03 [...]
04 # Drop-Policy
05 $IPTABLES -P INPUT DROP
06 $IPTABLES -P OUTPUT DROP
07 $IPTABLES -P FORWARD DROP
08 [...]
09 # Rules for SSH access to the
gateway
10 $IPTABLES -N ssh_gate
11 $IPTABLES -A INPUT -p tcp -m
state --state NEW -d $EXT_IP --
dport $SSH -j ssh_gate
12 # Gate should permit outgoing
and ingoing SSH (to the LAN)
13 $IPTABLES -A OUTPUT -p tcp -m
state --state NEW --dport $SSH -j
ssh_gate
14 $IPTABLES -A ssh_gate -j
ACCEPT
15 [...]
16 $IPTABLES -A INPUT -m state --
state ESTABLISHED,RELATED -j
ACCEPT
```

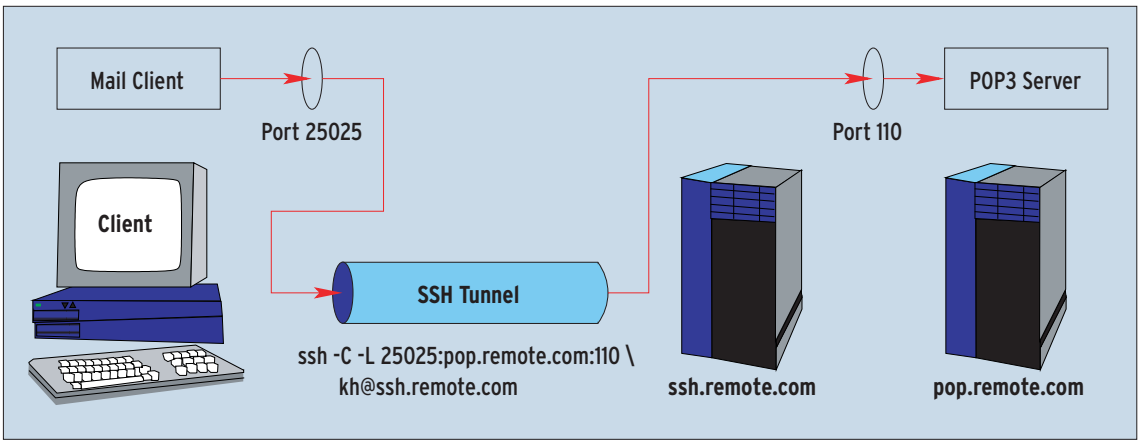

Figure 5: SSH can also relay TCP connections to a server running on a machine without the SSH daemon. The connection between *ssh.remote.com* and *pop.remote.com* is not secure in this case.

using her home office desktop. An SSH connection in an SSH tunnel provides an elegant and secure solution:

```
ssh gw_login@gw -L 2002:od:22
ssh od_login@localhost -p 2002
```

The first command opens up a tunnel from the local port 2002 to the gateway *gw*, which forwards this connection to the SSH port, 22, on *od*. The second command uses this tunnel to connect to port 2002 on localhost (option *-p*), thus creating an SSH on SSH connection.

Alternatively, the homeworker could log on to *gw* and move on to *od* from there. This solution would mean the user storing her SSH key on the gateway, enabling a forwarding agent, or using a normal password. The SSH on SSH method avoids this. The gateway has no access to the data being forwarded: *hd* is directly connected to *od* via the tunnel, and this means that user will be working with her account on *od*.

From the viewpoint of the tunneled connection it does not matter whether NAT (Network Address Translation) is involved, even multiple NAT will not cause any problems.

A Backdoor to Your Own Network

Let's look at another example that seems to be more complex at first: The user does not have a login on the gateway, and the firewall prevents her from connecting to the internal network. In this case remote port forwarding can provide a backdoor to the corporate network. The home desktop will need access to the Internet, and must be able to accept external SSH logins. The user must know the external IP address of her home desktop, but this should not be too difficult to determine, even for a dynamic IP address, in the light of services such as DynDNS. The user then enters the following command on her office desktop:

```
ssh hd_login@hd -R 2003:od:22
```

Instead of terminating this login, the user then leaves the tunnel open (see Figure 6). When home, she can use the tunnel to log on to her office desktop:

```
ssh od_login@localhost -p 2003
```

If the corporate gateway does not permit outgoing SSH connections for some reason, the user can simply have her SSH server on *hd* listen to a permitted port; port 80 looks promising in this case. This just goes to show how easy it is for users to poke holes in your firewall, if they really want to, of course. As soon as you open any port, users can tunnel through it. Of course, this normally means contravening corporate regulations, so if you want to keep your job, you should be very careful about tunneling, and seek prior authorization from your admin.

In the context of port forwarding the options *-N* and *-f* can be quite useful: *-N* prevents SSH from running commands server-side, and allows only the specified ports to be forwarded. *-f* sends the SSH client into the background, after authentication has been completed, i.e. after the user has entered her password or passphrase.

Special Cases: X11

X11 forwarding involves a special kind of SSH port forwarding. X11 always uses a network protocol. Even if the graphic output of a program running on the local machine is displayed on a local monitor, data have to be transferred between the client and the server.

The X11 server is responsible for the screen display in this case, and it also

Configuring SSH Servers via Webmin

The first article on OpenSSH [1] discussed the configuration of *sshd* amongst other things. If you prefer GUI based admin tools, you can use the corresponding Webmin module [6]. Webmin writes modified settings directly to the server configuration file */etc/ssh/sshd_config*. Figure 3 shows you what Webmin's SSH module looks like.

If you intend to use Webmin, you should be aware that this tool consists of a large number of Perl CGI scripts, that are accessible on port 10000 (TCP und UDP) of the Webmin server. To achieve a modicum of security, you will need to enable SSL encryption in your Webmin configuration, this will ensure that your login, password, and the changes you make in Webmin are not transmitted in the clear.

Also be aware that the Webmin distribution uses a 512 bit RSA key and a self-signed certificate for SSL. Of course, the certificate is not assigned to your own server. But the fact that anybody downloading the package will be aware of the purportedly secret key is probably worse. In other words, it does not really matter that the key length is insufficient. You would need your own SSL key and your own server certificate, or an SSH tunnel, to provide genuine security.

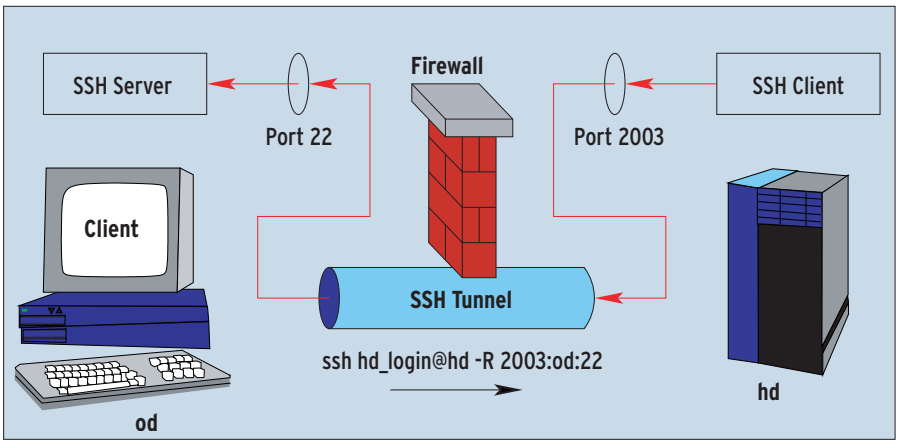

Figure 6: SSH tunnel: *od* first connects to *hd* and then opens a tunnel via reverse port forwarding, allowing *hd* to open a second SSH connection in the opposite direction to *od*

reads keyboard and mouse input. X11 clients are programs that use X11 for their input and output. X11 servers normally listen on port 6000. If a computer has more than one screen, keyboard, and mouse, additional X11 servers will use ports 6001 upward. The client program reads the environment variable *$DISPLAY* to discover what server it should display on.

If you can access an X11 server, you can display an X11 client on that server, however, you can also grab screenshots or sniff keyboard events. So without additional security measures X11 would be a security nightmare.

But rest assured, X11 uses an authentication system of its own. MIT Magic Cookies are the most common implementation in this area. Since you need authentication, a port forward alone is not sufficient for X11. So SSH provides a mechanism that allows you to relay the graphic output of a remote computer to your local display. This mechanism handles X11 authentication, sets the *$DISPLAY* variable when you log on, and forwards the connection through the tunnel.

Several conditions must be met. The configuration file for the remote SSH server, *sshd_config*, must contain the lines *X11Forwarding yes* and a directive of the type *X11DisplayOffset 10*.

On the SSH client side, you will need to run an X11 server and enable X11 forwarding, for example, by using the SSH option *-X* or *ForwardX11 yes* in */etc/ssh/ssh_config* or *~/.ssh/config*.

The profile files on the remote computer, for example, *~/.profile* or *~/.bashrc* can prove to be another pitfall. Some of these scripts attempt to set the *$DISPLAY* variable, without being aware of SSH. They may even overwrite the correct settings and this could cause some surprises if the X11 client talks directly to the X11 server, and simply ignores the tunnel, although SSH and X11 forwarding have been enabled.

After fulfilling the conditions for X11 forwarding, you can run any X11 program on the remote computer. The SSH tunnel forwards the display to the local display and encrypts the data transmission. When dealing with SuSE servers with Yast 2, or Mandrake hosts with DrakConf, admins can use this method for secure remote administration via an SSH tunnel.

Configuring a Firewall for SSH

We have already mentioned how a user can undermine a firewall using a tunnel. But no security conscious admin would want to attach her computer to the Internet without a firewall. The firewall is often the Internet gateway for an internal LAN configured with private IP numbers (RFC 1918).

Our task is to configure the firewall to allow an SSH login on the firewall host, and to provide access to the servers in a DMZ or on the LAN from that point. Listing 1 shows how you can use the firewall subsystem of the Linux 2.4 kernel to do so; it illustrates only the relevant sections of the *iptables* rules.

This set of rules uses a *DROP* policy for *INPUT*, *OUTPUT* and *FORWARD*. By default, the kernel will not permit any IP packages to enter or leave the computer, and will not forward any IPs. Interfaces, IPs and Ports must be specified explicitly – i.e. the basic principle, "anything not explicitly permitted is denied", applies. This policy will not even allow connections to a host loopback device without explicit permission.

An *INPUT* rule allows SSH connections to the gateway via the external interface. An *OUTPUT* rule allows SSH logins via the Gateway to computers on the LAN or in the DMZ. These rules do not permit you to log on directly to any internal computer. The last line in Listing 1 allows the kernel to recognize the packets belonging to a permitted connection and also permit them. This kind of statefulness became available with the network stack of the 2.4 kernel.

For more detailed information you might like to refer to the commented iptables scripts produced by Bob Sully [7], or to *man iptables*, and the iptables HOWTOs [8]. ■

INFO

[1] "Out of Sight: OpenSSH from the Administrator's Perspective", Linux Magazine Issue 24

[2] Scanssh: *http://www.monkey.org/~provos/scanssh/*

[3] Daniel J. Bernstein: "Circuits for Integer Factorization: A Proposal": *http://cr.yp.to/papers/nfscircuit.ps*

[4] *http://www.counterpane.com/crypto-gram-0204.html#3*

[5] *http://www.counterpane.com/crypto-gram-0203.html#6*

[6] Webmin SSH module: *http://www.webmin.com/download/modules/sshd.wbm*

[7] IPtables scripts by Bob Sully: *http://www.malibyte.net/iptables/scripts/fwscripts.html*

[8] HOWTOs for IPtables: *http://www.digitaltoad.net/docs/iptables-HOWTO-1.html*

THE AUTHOR

Andrew Jones is a contractor to the Linux Information Systems AG http://www.linux-ag.com in Berlin. He has been using Open Source Software for many years.

Andrew spends most of his scarce leisure resources looking into Linux and related Open Source projects.