Secure Remote Access with OpenSSH and rssh

by David Bank CNE, CCSE, CCNA

v1.75 (2008-Dec-10)

Why...

...is there a need for secure remote access?

To enable end-user access to hosts in the Linux/UNIX environment, the old standbys have been **telnet** (to reach a command prompt) and **FTP** (to transfer files). These protocols have been around since the mid-1980s, and are defined by **RFC 854**[1] (telnet) and **RFC 959**[2] (FTP).

They're also very insecure. Both protocols conduct their communication "in the clear", meaning that not only are the login credentials sent unencrypted, but all of the data transmitted back and forth are likewise open for anyone to read. Every keystroke, every password, every byte, can be silently intercepted and/or recorded.

Even within a "closed" or heavily-firewalled network, it is easy to overlook the dangers of unauthorized ARP- or ICMP-based re-direction, which would allow someone who had gained access to the network (legitimately or not) to conduct "man in the middle" (MitM) or similar attacks. Telnet and FTP offer no defenses against this, or any way to detect such subtle intrusions.

The original Internet environment was populated by a relative few, and relied as much on trust as anything else. More than 20 years have passed since then - the modern computing environment, for better or worse, is not inherently trustworthy, and insecure tools can be as dangerous as they are ubiquitous. Telnet and FTP are both. New, secure tools are needed.

...OpenSSH and rssh?

The Secure Shell protocol is actually a series of protocols, defined by several RFCs, including:

RFC 4251[3] Secure Shell (SSH) Protocol Architecture
RFC 4252[4] Secure Shell (SSH) Authentication Protocol
RFC 4256[5] Generic Message Exchange Authentication for the Secure Shell Protocol (SSH)

OpenSSH[6] is probably the best-known implementation of the SSH suite, and arguably the one with the widest OS vendor adoption, being supported on platforms as diverse as IBM AIX, Novell NetWare, Sun Solaris and practically every Linux distribution[7].

OpenSSH combines the shell/command-line functionality of telnet with the file transfer capability of FTP. Through its *subsystem* feature, its functionality can be extended even further. Using a single software stack for these services simplifies system management, providing a single point for configuration, authentication and logging.

Important!

This paper is based on the *Portable* version of OpenSSH, which varies in a number of ways, subtle and otherwise, from the OpenBSD version of the package. If you are using the OpenBSD version and encounter documentation conflicts between this paper and the **man** pages, consider the OpenBSD **man** pages as the more-accurate source for that platform.

rssh[8] is an extension to the OpenSSH environment that allows the system administrator to more finely control access through OpenSSH. Designed to work specifically with OpenSSH, rssh is a restricted shell that can prevent or allow specific functions of OpenSSH, on a global or per-user basis.

...not "secure" telnet or FTP/S?

When considering implementation of telnet and FTP protocol variations that provide secure communications, the first thing one must ask is: "Which one?"

For example, just for FTP, there are at least two RFCs for providing "secure" FTP:

RFC 2228[9] FTP Security Extensions
RFC 4217[10] Securing FTP with TLS

Similarly, several RFCs cover options to "secure" telnet; among them:

RFC 2946[11] Telnet Data Encryption Options
RFC 3205[12] On the use of HTTP as a Substrate

None of those methods enjoys wide implementation or vendor acceptance. Further, this approach results in two software stacks for what are really very similar purposes, whereas OpenSSH is a single software stack for both activities. With the submission of RFCs for the SSH protocol, OpenSSH becomes just as "standard" as telnet and FTP have been.

...this paper?

The goal of this paper is to provide a basic road map for the installation and configuration of **OpenSSH** and **rssh** in a *NIX/Linux environment. Then, the paper will cover how this software combination can be leveraged to provide a secure communications framework. While advanced functions, such as *port forwarding* and the ability to *tunnel* other protocols through SSH, are mentioned, those topics are outside the formal scope of this paper. Similarly, integrating OpenSSH with specific authentication back-ends (such as Kerberos) or additional security tools (such as TCP Wrappers) are also outside the scope.

It is important to keep in mind that these tools are not a magic security bullet. They will not protect against an account compromised by other means (*e.g.* someone learning a user's password, perhaps by social engineering), filesystem insecurity, attacks against other services (*e.g.* a webserver exploit) or insecure configuration of other tools or services.

How To Build and Install OpenSSH and rssh

OpenSSH and **rssh** are both Free/Open-Source Software (FOSS) packages. These softwares are among the few that I prefer to build from source, so this paper covers those steps.

Why Not Packages?

Common practice in the Linux/UNIX world is to install pre-built packages, usually provided by the platform vendor or distribution maintainer, rather than to build from source. Where a package is not part of the "official" offering, it is often available from a 3rd party (*e.g.* the software author, a site like SunFreeware[13], or other independent repository). Many modern Linux/UNIX distributions include a pre-built **OpenSSH**. **rssh** is also available in package format, although it seems less common to find it already included in distributions (some versions of Gentoo and Debian have it).

Why, then, does this paper take the build from source approach?

When using a pre-built package, one must accept whatever compile-time configuration decisions were made by the package creator. These parameters may or may not be adjustable at run-time, and compile-time options selected by the package creator may or may not be appropriate to a specific environment. Building from source allows tailoring the software. As with any other system administration decision, weigh the factors in the environment and choose a course of action.

If you decide to use a pre-built package, skip past the Compilation and Installation sections - start reading at the Configuration section. You should follow the package installation documentation, and reference this paper for configuration tips. The current versions, as of this writing, are **OpenSSH v5.0p1** and **rssh v2.3.2**.

Standards and Assumptions

This paper's assumptions, for those wishing to build from source, include the use of a modern Linux or UNIX (or UNIX-like) operating system, the ability to run shell scripts, compile C code (with GNU **gcc** or an equivalent), copy and/or link files, and set file modes and ownership. While some of these tasks require *root* privilege, you should only invoke that when you specifically need it (for example, most compilation steps can probably be accomplished as an unprivileged user). Additionally, this paper assumes that **OpenSSL**, and the **zlib** data compression libraries, have already been installed and configured.

Every system admin has their own way of doing things, and their own sense of where to put files. This paper is written from my perspective on these issues, which may be different from the maintainers of the pre-built packages. Select whatever file location scheme for these tools that is appropriate to your environment, and if necessary translate my paths and locations into the scheme you select.

My personal environment is usually either SLES v9 (or later) with **gcc v3.3.3** (or later), RHEL v4 (or later) with **gcc v3.4.6** (or later) or Sun Solaris v8 (or later) with **gcc v3.3.2** (or later). Other common tools include GNU **make v3.8x** and **nano v2.0.x**; however, I try to use a platform's stock development tools such as **ld**, **ar**, **yacc** and **lex**. I also tend to create a sub-directory structure specifically for building add-on tools, generally */work*. Make sure the partition where this is located has adequate room.

I usually install add-on tools, like **OpenSSH** and **rssh**, in a sub-directory off of */opt*, and then use symbolic links in */usr/bin* or wherever else might be needed. This allows me to control access better than if everything is dumped in */usr/local*. I prefer symbolic links because I frequently make */opt* its own partition, and I can "snap-in" a newer version of a tool with a few **mv** commands, since the link is merely a pointer to a path and file name (a hard link is another inode entry and can't cross partition boundaries).

What About Deployment Tool/Technique X?

Administrators accustomed to working in a homogeneous environment may wonder why this paper does not mention or advocate the use of specific tools for deployment beyond a single machine, or uses techniques that might seem problematic when viewed from a specific environment perspective. A reader might find themselves thinking “Why not just use <*insert tool name here*>?” or “That suggestion doesn't make sense in <*insert specific environment name here*>!”

This paper is deliberately written for a generic audience, where a reader may be interested in applying the information presented in diverse environments, perhaps other than a typical Linux distribution or common UNIX variant. As a consequence, it offers ideas culled from a number of environments. The reader is encouraged to consider those ideas, techniques and tools that are applicable to their situation, and to ignore those that are not.

OpenSSH and TCP Wrappers in *NIX

On Linux and most *NIX platforms, OpenSSH can be compiled with support for TCP Wrappers[14], although doing so is not specifically covered in this paper. If TCP Wrappers is installed on the system[15] but you decide to not compile OpenSSH with TCP Wrappers support, you can still leverage TCP Wrappers by invoking **sshd** via **inetd** (or **xinetd**). Consult the TCP Wrappers documentation for details.

Note to Solaris Admins

As of Solaris v9, Sun has added TCP Wrappers support to Solaris. The package name is **SUNWtcpd**.

Download and unpack the sources

However might be appropriate for your environment, download the latest **OpenSSH** and **rssh** source packages. For example, into */work/openssh* and */work/rssh*, respectively (assuming the **gzip**-ed versions):

```
me@host /work 2 $ dir
drwx------ 2 me wheel    512 Apr 9 12:23 openssh/
drwx------ 2 me wheel    512 Apr 9 12:23 rssh/
me@host /work 3 $ dir openssh
-rw------- 1 me wheel 961213 Apr 9 15:32 openssh-5.0p1.tar.gz
me@host /work 4 $ dir rssh
-rw------- 1 me wheel 585704 Apr 9 19:47 rssh-2.3.2.tar.gz
```

Downloading and verifying the file signatures is also a wise step; however, it is outside the scope of the paper.

If your **tar** program includes the ability to decompress **gzip**-ed files, then you can use it directly; or you can call **gzip** as a separate step on the way to un-**tar**ing the files. Here is the latter method, with the three commands separated by semi-colons:

```
me@host /work/openssh 8 $ gzip -dv *.gz ; tar -xvf*.tar ; gzip -v9 *.tar
```

...and...

```
me@host /work/rssh 10 $ gzip -dv *.gz ; tar -xvf*.tar ; gzip -v9 *.tar
```

The commands will decompress the files, unpack the tarfile, then recompress the tarfile using the best compression offered by **gzip**. No sense wasting disk space leaving the uncompressed tarfile around. Or you can delete the tarfile. If you like and have it installed, substitute **bzip2** for **gzip** when re-compressing – the results tend to be better.

The source files now reside in a directory named very similar to the file from which everything was extracted. If you, like me, use the autocompletion capability of the Bash shell, this can be annoying. So I usually change the directory name to something short:

```
me@host /work/openssh 12 $ mv openssh-5.0.p1 V5.0p1
me@host /work/openssh 13 $ dir
-rw------- 1 me wheel 961231 Apr 10 19:47 openssh-5.0p1.tar.gz
-rwx------ 1 me wheel    512 Apr 10 19:47 V5.0p1/
me@host /work/openssh 14 $ cd ../rssh
me@host /work/rssh 15 $ mv rssh-2.3.2 V2.3.2
me@host /work/rssh 16 $ dir
-rw------- 1 me wheel 585704 Nov 10 19:47 rssh-2.3.2.tar.gz
-rwx------ 1 me wheel    512 Nov 10 19:47 V2.3.2/
```

Now autocompletion won't beep at me. You can get ready to build OpenSSH with:

```
me@host /work/rssh 17 $ cd ../openssh/V*
me@host /work/openssh/V5.0p1 18 $
```

OpenSSH: Compilation

Helpfully, OpenSSH uses the ubiquitous GNU *autoconf* configuration tool. The script **configure** will automatically examine your system, check dependencies, and prepare OpenSSH to be compiled with a built-in set of defaults.

However, since building from source offers flexibility, I recommend a few tweaks. For example, the default location of the OpenSSH binary is */usr/local*, and as I noted above, I prefer */opt/openssh*. You can see all the possible configuration options, and their defaults, with the "--help" parameter:

```
me@host /work/openssh/V5.0p1 19 $ ./configure --help
```

This will show you a lot of information, but won't actually configure or compile anything. What you might change from the defaults is largely dependent on your preferences and standards in your environment.

Within the context of this paper, I suggest the following defaults, which can be selected from the command-line when invoking the **configure** script:

```
me@host /work/openssh/V5.07p1 20 $ ./configure --prefix=/opt/openssh \
      --sysconfdir=/opt/openssh/conf --with-zlib=/path/to/zlib \
      --with-ssl-dir=/path/to/openssl --with-pid-dir=/var/run \
      --with-privsep-path=/var/empty/or/alternate \
      --with-privsep-user=user name \
      --with-mantype=man
```

Why these particular options and settings? Glad you asked that:

`--prefix=/opt/openssh`

See my previous discussion about file locations.

`--sysconfdir=/opt/openssh/conf`

Without this, the location of the OpenSSH configuration files and encryption keys would be in */opt/openssh/etc*. As time-honored as *etc* is, I prefer the more-descriptive *conf*.

`--with-zlib=/path/to/zlib`

This option simply tells the **configure** script where the **zlib** software is located. **configure** would almost certainly figure this out for itself, I'm just making sure.

`--with-ssldir=/path/to/openssl`

Similar to the previous option, I'm simply saving **configure** the trouble of locating the OpenSSL files. Make the script earn its keep, if you prefer.

`--with-pid-dir=/var/run`

When running, the **sshd** daemon will write its process ID in the file *sshd.pid* and place it in this directory. Actually, */var/run* is the default if this option is omitted; it's being shown here for demonstrative purposes.

`--with-privsep-path=/var/empty/or/alternate`

The concept of **Privilege Separation** is discussed in more detail below. This option merely defines the directory that will be used by the **PrivSep** process. The default is */var/empty*, but it can be anywhere (within reason). This directory will be created by the **make install** command, if it doesn't already exist. If you create it by hand (or use a pre-existing location), the directory should be owned by *root*, mode *755*, and not contain any other files or directories.

`--with-privsep-user=user name`

Using this option, you can specify a non-privileged user name for Privilege Separation. The default value is *sshd*. This user name must exist or **sshd** will not run (the software will install without the user ID having been defined, but the install script will complain).

`--with-mantype=man`

This option instructs **configure** to create documentation in **man** format. Other options include **cat** (for *catman* format) or **doc**. Set as you prefer.

At this point, the **configure** script should have been run, with any options you wanted. For the purposes of this paper, the assumption is made that it ran without errors, or that any issues have been fixed. Among other things, **configure** will generate *Makefile* in the same directory, and this is used for the rest of the process.

You're now ready to compile the OpenSSH suite:

```
me@host /work/openssh/V5.0p1 21 $ make
```

This starts the compilation process. Troubleshooting compilation issues is outside the scope of this paper, so we'll proceed to the next stage.

OpenSSH: Installation

If you've gotten to this point without any problems, the rest of the process should be equally trouble-free. To actually install the program files, you need to have *root* privilege because, among other things, you'll be setting SUID bits in file modes, and usually only *root* can do that. So, invoke **su**:

```
me@host /work/openssh/V5.0p1 25 $ su -
```

Now that you are privileged, install the program:

```
# make install
```

Again, troubleshooting installation error messages are outside the scope of this paper. When the script has run, it's a good idea to check the results, which should look like this (not all installation directories are shown):

```
# ls -la /opt
drwxr-xr-x 7 root   other      512 Apr 11 13:34 openssh/
# ls -la /opt/openssh/bin
-rwxr-xr-x 2 root   other    37676 Apr 11 13:34 scp*
-rwxr-xr-x 2 root   other    68000 Apr 11 13:34 sftp*
lrwxrwxrwx 1 root   other        5 Apr 11 13:34 slogin -> ./ssh*
-rwxr-xr-x 1 root   other   257288 Apr 11 13:34 ssh*
-rwxr-x--- 1 root   other    81488 Apr 11 13:34 ssh-add*
-rwxr-x--- 1 root   other    67680 Apr 11 13:34 ssh-agent*
-rwxr-x--- 1 root   other   106748 Apr 11 13:34 ssh-keygen*
-rwxr-x--- 1 root   other   147780 Apr 11 13:34 ssh-keyscan*
# ls -la /opt/openssh/conf
-rw-r--r-- 1 root   other   111892 Apr 11 13:34 moduli
-rw-r--r-- 1 root   other     3604 Apr 11 13:34 ssh_config
-rw------- 1 root   other      668 Apr 11 13:34 ssh_host_dsa_key
-rw-r--r-- 1 root   other      599 Apr 11 13:34 ssh_host_dsa_key.pub
-rw------- 1 root   other      524 Apr 11 13:34 ssh_host_key
-rw-r--r-- 1 root   other      328 Apr 11 13:34 ssh_host_key.pub
```

```
-rw------- 1 root   other      883 Apr 11 13:34 ssh_host_rsa_key
-rw-r--r-- 1 root   other      219 Apr 11 13:34 ssh_host_rsa_key.pub
-rw-r--r-- 1 root   other     5829 Apr 11 13:34 sshd_config
# ls -la /opt/openssh/share
-rwxr-xr-x 1 root   other      600 Apr 11 13:34 Ssh.bin
# ls -la /opt/openssh/libexec
-rwxr-xr-x 1 root   other    32492 Apr 11 13:34 sftp-server*
-rws--x--x 1 root   other   152024 Apr 11 13:34 ssh-keysign*
```

So, what are all these files? And do you need them all? The answer to the second question is that you probably need most, but not all, of them.

First, let's look in */opt/openssh/bin*. Here, you'll find secure implementations of many venerable *NIX programs client tools, as well as some new ones. **scp** is the Secure remote CoPy program, similar in function to **rcp**. As its name suggests, **sftp** is the Secure FTP client, which accepts many of the same commands as does a standard **ftp** client program. **ssh** (and its associated symlink **slogin**) is the Secure SHell client program that effectively replaces **telnet**, **login**, **rsh**, **rexec** and **rlogin**.

The programs **ssh-add** and **ssh-agent** are complementary, and form the foundation for a public-key-based single-sign-on framework, under which a user can automate authentication to other environments that support key-based authentication. **ssh-keygen** and **ssh-keyscan** are tools for users to manage their own authentication keys. Each of these has its own **man** (or whatever documentation format you chose) page. Unless you wish to deny certain client tools to your user population, you need all the programs in this directory.

Next, the */opt/openssh/conf* directory houses system-wide files. *ssh_config* is the configuration for the **ssh** program, but is ignored if the invocation of **ssh** includes the **-F** parameter. The file *sshd_config* is the configuration of the **sshd** server daemon. The server's private keys are in the files *ssh_host_key*, *ssh_host_rsa_key* and *ssh_host_dsa_key*. It is important that these files be protected (in the default installation they can only be read by *root*), as with them, anyone may impersonate your host. The server's public keys are in the files, *ssh_host_rsa_key.pub*, *ssh_host_dsa_key.pub* and *ssh_host_key.pub* - these may be read by anyone. The file *moduli* contains large prime numbers for generation of Diffie-Hellman (DH) keys and was created as part of the installation (*i.e.* is unique to your system).[16]

In */opt/openssh/share*, we find **Ssh.bin**, which is an experimental Java applet for smartcard readers using OpenSC. If this is not needed, then you may safely remove this file and directory.

Finally, in the directory */opt/openssh/libexec*, there are the programs **ssh-keysign** and **sftp-server**. The first is a helper program for host-based authentication, and is invoked by **ssh** when the **EnableSSHKeysign** directive is set to **yes** in *ssh_config*. The latter is the SFTP server *subsystem* invoked by **sshd** when a client requests that subsystem. SFTP is not provided by **sshd** without this subsystem, and subsystems in general are discussed later in this paper.

If everything is correct, and you've installed the OpenSSH client programs (*e.g.* **ssh**, **sftp**) into a location not normally in your *$PATH* (like */opt/openssh*), then you'll probably want to link the files to */usr/bin* or wherever is appropriate, like so (not all possible links shown):

```
# ln -s /opt/openssh/bin/sftp /usr/bin/sftp
# ln -s /opt/openssh/bin/ssh /usr/bin/ssh
# ln -s /opt/openssh/bin/ssh /usr/bin/scp
# ln -s /opt/openssh/bin/ssh /usr/bin/slogin
# ln -s /opt/openssh/sbin/sshd /usr/sbin/sshd
# ln -s /opt/openssh/man/man1/ssh.1 /usr/share/man/man1/ssh.1
# ln -s /opt/openssh/man/man1/sftp.1 /usr/share/man/man1/sftp.1
# ln -s /opt/openssh/man/man1/scp.1 /usr/share/man/man1/scp.1
# ln -s /opt/openssh/man/man5/ssh_config.5 /usr/share/man/man5/ssh_config.5
# ln -s /opt/openssh/man/man5/sshd_config.5 /usr/share/man/man5/sshd_config.5
```

Again, exactly what you install (or link) where is driven by how you administer your environment. This paper merely shows one possible way.

OpenSSH: Configuration

With OpenSSH installed, it's time to configure the server (**sshd**). The server is configured by the file */opt/openssh/conf/sshd_conf* (or wherever your SSH configuration files are located).

It's important to note that options selected at the command line override the configuration file. Choose one way or the other to configure **sshd**. Don't mix the two methods, or you'll create an environment that is more difficult to administer.

The default configuration file will contain many option keywords, and may include keywords for options that are not supported in your **sshd** build. For example, the standard *sshd_config* file contains references to Kerberos and GSSAPI. However, if you didn't compile in Kerberos support (or it wasn't included in the **sshd** from your package), then the options aren't supported. The content of the *man* page for *sshd_config* also varies by what the compilation options were.

You can edit the *sshd_config* file that was installed, or write your own. The included file is moderately well documented. A more complete version is in the **Reference** section below.

It's not practical to review every possible keyword, but let's take a look at some of the more important ones, as they appear in the file provided in the **Reference** section:

`Protocol 2`

In its default configuration, **sshd** supports both SSH v1 and SSH v2 protocols[17]. However, SSH v1 has a number of security weaknesses, and is generally considered

deprecated (although a number of devices - like routers - that support SSH access only support SSH v1). Unless you have some insurmountable technical challenge that prevents you from requiring clients accessing your SSH server to always use SSH v2, disable SSH v1 support.

`AllowTCPForwarding yes`

Known as "the poor man's VPN", *TCP Forwarding* allows TCP packets from an SSH client machine to be redirected, or forwarded, to other ports on the host running the SSH server. They can also be redirected to remote hosts.

Disabling TCP Forwarding does not prevent users with shell access from setting up their own forwarders. To a certain extent, you must be able to trust users with shell access.

As a rule, never allow TCP Forwarding on SSH servers that allow anonymous access, such as an *Anonymous CVS* host. This is because your network can then be probed, with an attacker using the anonymous access to redirect packets to various hosts within your network. Shell access is not required to abuse TCP Forwarding - any **sshd**-delivered service can be leveraged.

In a situation where you must permit TCP Forwarding on a host that also has anonymous access, then you may be able to use tools such as TCP Wrappers to strictly control the behavior of **sshd**. As noted previously, TCP Wrappers is out-of-scope for this basic paper; you should consult the OpenSSH and TCP Wrappers documentation concerning using TCP Wrappers to exercise control over SSH.

If you are able to prevent users from editing their *authorized_keys* file (which would generally mean that you prohibit them from writing at all to their home directory; or you use a global *authorized_keys* file that you control), then it's possible to use options on the keys in the file to limit what the users can do with respect to forwarding TCP packets.

Finally, introduced as an **sshd** option in OpenSSH v4.4, the **PermitOpen** directive affords a measure of granularity over TCP Forwarding. Consider limiting any TCP Forwarding you enable using this option.

`ChrootDirectory`

An in-depth discussion of **chroot** is out of the scope of this paper. However, it is mentioned here because, as of OpenSSH v5.0p1, the SSH server (**sshd**) natively supports the function. This means it is no longer necessary to use **rssh** (or to jail the entire SSH server) for **chroot**.

`GatewayPorts no`

If enabled, this option allows the SSH server to listen for forwarded ports on any interface. Normally, **sshd** will only listen for forwarded ports on **127.0.0.1**, the *loopback* interface. This means that only programs running on the SSH server can have their TCP packets forwarded to an SSH client. Other hosts cannot have their TCP packets forwarded through the SSH tunnel from the server to the client.

If you set this option to **yes**, then **sshd** can be told to listen on other interfaces for TCP packets hitting forwarded ports, and forward them to the SSH client that requested the packet forwarding (this arrangement, where the server forwards to the client, is sometimes called *reverse forwarding*). SSH clients request this functionality with the **-R** parameter when setting up forwarding.

TCP Forwarding presents significant security issues, and **GatewayPorts** magnifies those issues by allowing any host to forward packets through the SSH tunnel. While access to this functionality can also be controlled with TCP Wrappers, you should be sure you have thoroughly researched the effects of this option, and understand the security ramifications of allowing **GatewayPorts** and TCP Forwarding in general.

`MaxStartups 5`

Using this option, you limit the number of unauthenticated connections that the SSH server will accept at any one moment in time. The default is **10**, and you should adjust this higher or lower as your environment requires. Authenticated connections do not count against this limit.

When the limit is reached, further connection requests are summarily refused, until a pending connection authenticates, times out, or is dropped (perhaps due to too many authentication failures). If the limit is set to **0**, then there is no limit on simultaneous unauthenticated connections, and connections are always accepted (until the server runs out of resources).

The limit can also be expressed as **A:B:C**, where **A** is a lower bound, **C** is an upper bound, and **B** is a percentage. In this situation, when the number of unauthenticated connections reaches **A**, the next connection has a **B%** chance of being rejected. As the number of unauthenticated connections grows above **A**, the chance of rejection of a given new connection grows in a linear fashion, roughly expressed by the equation **((100 - B)/(C - A))%** per additional connection.

When the number of unauthenticated connections reaches **C**, the chance of rejection of a new connection is 100%. This remains true until an existing pending connection authenticates, times out, or is dropped.

`PermitRootLogin no`

When set to **no**, the *root* account is not allowed to login, even if it is listed in **AllowUsers**. Any account with a UID of 0 is affected by this option, not just the account named *root*. There are also several caveats that are important to understand.

First, listing *root* in **DenyUsers** - or a group of *root* in **DenyGroups** - will block *root* at connection time. In contrast, if the only impediment to *root* login is this option, and proper credentials are provided, the authentication process actually succeeds; the session is simply immediately terminated. This difference is important in that if you want to absolutely block remote *root* access, then doing so with **DenyUsers/DenyGroups** is the more sure method.

Next, there exists a poorly documented loophole with this option. Key-based authentication is allowed to succeed for *root*, regardless of this option, if a command has been specified that matches a command in the account's *authorized_keys* file. If the key and command match, the command is executed. Again, if you really want to block *root*, use a specific listing in **DenyUsers**.

Finally, using **without-password** instead of **no** restricts *root* login to authentication methods other than the account password, such as key-based authentication. This prevents someone who possesses the *root* password, but not the account's private key, from logging in.

If you simply have to allow remote *root* login, and your hosts must also be available from untrusted/untrustable networks (such as the Internet), then consider architecting a solution with multiple instances of the **sshd** server, augmented with host-based firewalls or tools like TCP Wrappers, where the instance of **sshd** that allows remote *root* access is only available to hosts on a trusted network.

`DenyUsers AllowUsers DenyGroups AllowGroups`

These options allow you to specify either user accounts, or groups in which user accounts are members, that are either specifically denied or permitted to login *via* SSH. These options are evaluated in the order listed above, regardless of the order in which they appear in *sshd_config*.

Evaluation of these options is by string comparison to user/group names (not UID or

GID), and wildcards (***** and **?**) may be used to form *patterns* for comparison. For example, **jo*** would match *john* and *joanne*, but not *jack*. Similarly, **t?m** would match *tim* and *tom*, but not *tommy*.

When a user attempts to login, their user name is compared against the list (or patterns) in **DenyUsers**. If **DenyUsers** is not in *sshd_config*, or if no match is found, then the user name is compared to **AllowUsers**. If that option hasn't been specified, then login is allowed to proceed to the next step. If it has been specified, then if the user name cannot be matched, the user will be denied login; otherwise, login is allowed to proceed.

If the user name is not denied, then **sshd** retrieves the groups (Primary and any Secondary) to which the user ID belongs. These are all compared against the list of group names and/or patterns in **DenyGroups**, if it is defined in the configuration. If a match is found, then the user is denied login; if **DenyGroups** is not defined, or no match was found, then the **AllowGroups** option is checked. If **AllowGroups** is not defined, or is defined and any group of which the user is a member can be matched, then login proceeds. If **AllowGroups** is defined but no match is found, then login is denied.

The general approach of these options is to deny login if any reason can be found to do so. It's important to note that if **AllowUsers** or **AllowGroups** is specified, then any user or group name not explicitly listed (or which cannot be matched to a pattern) will be denied login. When deciding how to approach access control with these options, consider carefully the ways in which they interact. Generally, it's easier to implement and understand a configuration that uses either **DenyUsers/Groups** or **AllowUsers/Groups**, but not both.

`PermitUserEnvironment no`

This option controls whether or not **sshd** will honor **environment=** parameters on keys in the user's *.ssh/authorized_keys* file, or read the user's *.ssh/environment* file. If set to **no** (the default), then the parameters and environment file, if they exist, are ignored. If this option is **yes**, then the file is read, if it exists, and any **environment=** parameters on authentication keys are honored.

Within the context of the configurations and system design presented by this paper, this option should never be enabled, as it can allow users to bypass access restrictions. Also, enabling user control of environment variables will break **rssh** security.

`Compression delayed`

Using this option, the administrator can control when, or if, **sshd** will honor client requests for data compression across the link. The default value, **delayed**, instructs the SSH server to ignore client requests for compression until after authentication is successful. This prevents unauthenticated connections from attempting to exploit data compression libraries (for example, as shown in **CVE-2005-2096**[18]), while still allowing authenticated connections to request compression.

Using **no** means that all client requests for compression are ignored; while **yes** means that any client request for compression is honored, even for unauthenticated connections.

While data compression can be useful and significantly enhance performance, especially across slower links, it's not always appropriate. SSH connections from the local LAN rarely benefit from compression, and in some cases, such links will actually get worse performance if compression is used.

Important!

Understand that using **Compression delayed** may cause connection negotiation problems for certain SSH clients. The option causes the SSH server to decline compression requests from the client during the initial (unauthenticated) session when keys are exchanged. Some clients will not re-request compression support if the first request is declined. If the client is configured to require compression support when talking to the server, then the connection negotiation will fail.

Client softwares known to have this problem include:

* Van Dyke Software, Secure CRT (v3.x, v4.01)
* SSH Communications Security Corp., SSH Secure Shell v3.2.9

Client softwares known to not have this problem include:

* OpenSSH v4.x client
* KDE kssh client v0.7

```
UsePrivilegeSeparation yes
```

The goal of this option is to enhance security by having the SSH server create unprivileged child processes to handle initial unauthenticated connections. When a connection arrives, **sshd** spawns a child, which runs under the UID/GID of the *Privilege Separation User* for which **sshd** was built. This unprivileged child is further restricted by **chroot** to the *Privilege Separation Directory* hardcoded into **sshd**.

The child process handles key exchange and receipt of authentication credentials. A successful attack at this point only compromises the unprivileged (and **chroot**ed) child process, not the privileged instance of **sshd**. The child process uses *pipes* to communicate with the privileged parent, and the parent process is responsible for verifying any authentication credentials presented by the remote client. When advised by the parent that the authentication is valid, the child passes the cryptographic and authentication state back to the parent, and terminates.

If the authentication was valid, then the privileged SSH server process spawns a new unprivileged child process, this time using the UID/GID of the authenticating user. This child process requests a PTY from the privileged parent, and then the user session begins.

By default, this process is enabled (**yes**), but the Privilege Separation user ID and directory must exist on the system. The directory should be owned by *root* and have mode *755*. If this option is disabled (**no**), then all communication between the client and server happens with the privileged **sshd** instance. If this instance is compromised, then privilege escalation attacks can occur.

Subsystem

Perhaps the least-understood option (probably because it is not well documented), this is also a powerful and useful tool. A *subsystem* can be any executable program; even, in theory, a shell script.

To understand the ramifications, it's helpful to know how the "standard" subsystem, SFTP, works. Essentially, invoking the SSH client with the the proper **subsystem** parameter (*e.g.* **ssh -s sftp**) is the same as invoking **sftp**. Replace **-s sftp** with something else that has a corresponding **subsystem** entry in *sshd_config* and practically any other program can be invoked in the same way. For example, consider this *sshd_config* line:

```
subsystem   imap     /usr/sbin/imapd
```

This subsystem can then be used to provide secure IMAP sessions between a client and a server. On the client, the command-line would be:

```
$ ssh -s imap server.domain.tld
```

By defining an executable as a *subsystem*, it can be conveniently executed by a client, and the communications will be protected by the SSH encryption. This capability can be leveraged far beyond the default SFTP server function. Note that this ability is specific to SSH v2, and is not supported under SSH v1.

When using the **`ChrootDirectory`** directive (introduced in OpenSSH v5.0p1) and the native **sftp-server** daemon, a user's SFTP session may be **chroot**ed with relative ease (as compared to using **rssh** for the same function) . The specifics of **chroot** are outside the scope of this paper.

`Match`

Introduced in OpenSSH v4.4p1, the **Match** option allows you to create conditional configuration options. In brief, the **Match** option allows you to apply a subset of the **sshd** configuration options to an arbitrary SSH session. These options override the global settings for the session.

When the sshd_config file contains a **Match** statement, the file is read when **sshd** starts, then again every time a new SSH session is initiated. Each time the file is re-read, **sshd** is looking for **Match** statements. When it finds one, it checks the criteria specified. If a match is found, the entries in the file are parsed until another **Match** statement is encountered, or EOF is reached. These entries are known as the *Match Block*.

There are four (4) criteria on which a **Match** statement may operate: **User** (the user name under which the connection is authenticating), **Group** (a Group to which the authenticating user belongs), **Host** (the host name from which the SSH session is originating) and **Address** (the IP address from which the SSH session is originating).

WARNING!

Unless you control DNS for all hostnames in a **Host** statement, it is dangerous to use it as a selection criteria, especially if the **Match** block grants permissions not found in the global configuration.

Only a relative few **sshd** configuration commands are valid in a **Match** block. The most useful ones are probably **AllowTcpForwarding**, **ChrootDirectory**, **GatewayPorts**, **PermitOpen**, **Banner**, **ForceCommand** and **X11Forwarding**. The sample *sshd_config* file available in the **Reference** section below shows some practical examples.

WARNING!

Command-line options specified when **sshd** is invoked override settings in *sshd_config*. In particular, this can affect the **GSSAPIAuthentication** option. Read the OpenSSH documentation thoroughly before using that option.

With these features and issues in mind, you're ready to edit the *sshd_config* configuration file. You're still privileged, so simply:

```
# edit /opt/openssh/conf/sshd_config
```

Note that this file must be writable by *root* only. **sshd** will refuse to run if this file is group- or world-writable, or if it is located in a directory that is group- or world-writable.

Moving on, if you've not done so yet, exit the *root* shell. Let's take a look at what we've done:

```
# exit
me@host /work/openssh/V5.0p1 26 $ which ssh
/usr/bin/ssh
```

Congratulations, OpenSSH is now installed. Next, we'll build and install **rssh**.

rssh: Compilation

Like OpenSSH, **rssh** also uses GNU *autoconf*. The **configure** script automatically examines your system, checks dependencies, and prepares **rssh** for compilation.

As with OpenSSH, I suggest a few tweaks, starting with the binary location. Again, the default location is */usr/local*, and I prefer */opt/rssh*. You can see all the possible configuration options, and their defaults, with the "--help" parameter:

```
me@host /work/rssh/v2.3.2 51 $ ./configure --help
```

As before, what you might change from the defaults is largely dependent on your preferences and standards in your environment. I like to change the following defaults:

```
me@host /work/rssh/V2.3.2 52 $ ./configure --prefix=/opt/rssh \
      --sysconfdir=/opt/openssh/conf \
      --with-sftp-server=/path/to/openssh/libexec/sftp-server
```

Reviewing these specific tweaks:

`--prefix=`*`/opt/rssh`*

See my previous discussion about file locations.

`--sysconfdir=`*`/opt/rssh/conf`*

Without this, the location of the **rssh** configuration file would be in */opt/rssh/etc*. I prefer the more-descriptive *conf*.

`--with-sftp-server=`*`/path/to/openssh/libexec/sftp-server`*

This option tells the **configure** script where the **sftp-server** subsystem exec file is located. **configure** would probably find it just fine, I'm merely making sure. Omit this if you prefer.

At this point, the **configure** script should have been run, with any options you wanted, and as with OpenSSH, troubleshooting system-specific errors is outside the scope of this paper. Among other things, **configure** will generate *Makefile* in the same directory, and this is used for the rest of the process.

You're now ready to compile the **rssh** software:

```
me@host /work/rssh/V2.3.2 53 $ make
```

This starts the compilation process. Troubleshooting compilation issues is outside the scope of this paper, and so I'll proceed to the next stage.

rssh: Installation

If you've gotten to this point without any problems, the rest of the process should be equally trouble-free. To actually install the program files, you need to have *root* privilege because, among other things, you'll be setting SUID bits in file modes, and usually only *root* can do that. So, invoke **su**:

```
me@host /work/rssh/V2.3.2 55 $ su -
```

Now that you are privileged, install the program:

```
# make install
```

Once more, troubleshooting installation error messages falls outside the scope of this paper. When there are no errors, it's a good idea to check the results, which should look like this (omitting examination of the **man** directory):

```
# ls -la /opt
drwxr-xr-x 7 root   other      512 Apr 11 17:53 rssh/
# ls -la /opt/rssh
drwxr-xr-x 2 root   other      512 Apr 11 17:53 bin/
drwxr-xr-x 2 root   other      512 Apr 11 17:55 conf/
drwxr-xr-x 2 root   other      512 Apr 11 17:53 libexec/
drwxr-xr-x 4 root   other      512 Apr 11 17:53 man/
# ls -la /opt/rssh/bin
-rwxr-xr-x 2 root   other   104664 Apr 11 17:53 rssh*
# ls -la /opt/rssh/conf
-rw-r--r-- 2 root   other     1791 Apr 11 17:53 rssh.conf
# ls -la /opt/rssh/libexec
-rwsr-xr-x 1 root   other   103740 Apr 11 17:53 rssh_chroot_helper*
```

If everything is correct, and you've installed **rssh** into a location not normally in your *$PATH* (like */opt/rssh*), then you'll probably want to link the **rssh** executable (and, optionally and not shown, the documentation) to */usr/bin* or wherever is appropriate, like so:

```
# ln -s /opt/rssh/bin/rssh /usr/bin/rssh
```

/opt/rssh/bin/rssh is the **rssh** shell program When you want the **rssh** restrictions to apply to a user, simply change the user's shell specification in */etc/passwd*, like so (this example assumes the symlink has been made):

```
luser:x:1000:500:L. User:/home/luser:/usr/bin/rssh
```

The configuration file, *rssh.conf*, is located in */opt/rssh/conf*, and this location is hardcoded in **rssh**. The next section will delve into syntax and usage in more detail. The sample file included in the install is actually fairly well commented, and shows a number of useful examples.

In */opt/rssh/libexec*, there is **rssh_chroot_helper**, which is used by **rssh** to safely transition an **rssh**-restricted shell into a **chroot** jail. Setting up a **chroot** jail in an **rssh** environment is frequently a complex, even daunting, task; and usually very specific to the system in question. You should read the documentation in the **rssh** package before attempting it, and because of the system-specific nature of the task, it's outside the scope of this paper. As of OpenSSH v5.0p1, **sshd** natively supports **chroot**, and may offer a better way of providing a **chroot** session (but is still out-of-scope for this paper).

Depending on your security model and level of paranoia, you can leave *rssh.conf* with its default ownership and mode (only editable by *root*), or make it easier for appropriate staff to update it. For example, if you use a group, say *wheel*, to list system admin IDs, then you might:

```
# chown root:wheel /opt/rssh/conf/rssh.conf
# chmod 664 /opt/rssh/conf/rssh.conf
# ls -la /opt/rssh/conf/rssh.conf
-rw-rw-r-- 2 root   wheel     1791 Apr 11 17:53 rssh.conf
```

Other optional installation steps include making a copy of the original *rssh.conf* file before any changes are made, adding **rssh** to */etc/shells* (if appropriate to your environment), and exiting your privileged state if it's not necessary for editing *rssh.conf*:

```
# cp /opt/rssh/conf/rssh.conf /opt/rssh/conf/rssh.conf.original
# exit
me@host /work/rssh/V2.3.2 57 $ cd /opt/rssh/conf
me@host /opt/rssh/conf 58 $
```

rssh: Configuration

Once an account's shell has been set to **rssh**, then when the user connects via **sshd**, after both authentication and Privilege Separation have occurred, **rssh** will consult *rssh.conf* to determine what activities, if any, the account may perform on the host. If nothing is permitted, the SSH connection will be dropped, with an explanatory message. The message text is hardcoded and if you want to change it, you'll need to edit the source code (see the file *util.c*).

The *rssh.conf* file can best be considered as having two types of entries: global and per-user. It's important to note that the entire file is parsed each time it is read, and the per-user entries override all global settings whenever a matching per-user entry is found. This means that if you use both types of entries, then you cannot depend on one to "fill in the blanks" in the other.

In a practical example, consider user *bob*, whom you have restricted using **rssh**. If, in *rssh.conf*, you set the global **allowsftp**, but also have a per-user entry for *bob*, you must specify that *bob* may use SFTP in *bob*'s per-user entry. **rssh** will not allow *bob* to "inherit" the ability to use SFTP from the global setting. This also means that if you don't want *bob* to be able to SFTP into the host, the presence of a global **allowsftp** setting won't override the account's per-user entry.

Logging is also configured via *rssh.conf*, using the **logfacility** keyword. You can only specify the *Facility* for **syslog**, and the values may be specified two different case-insensitive ways: for example, **LOG_USER** and **user** are considered equivalent. If you do not include this keyword, then the default Facility is **LOG_USER**. In general, **rssh** will log with *Severity* **INFO**, but other *Severity* levels are used. Logging begins after **rssh** drops its *root* privileges (a safety measure), and is always performed (turning off all logging would require editing the code).

The keywords **allowscp**, **allowsftp**, **allowcvs**, **allowrdist** and **allowrsync** are global settings, each used to enable a specific service through SSH. Similarly, the global **umask** keyword allows you to override any system default or shell-based file masks, and apply a consistent mask for **scp** and **sftp** operations. The **umask** should be specified using octal notation (*e.g.* **022**). These settings will apply to all user accounts that have a shell of **rssh** and for which a per-user entry is not found.

The **chrootpath** keyword, also a global, defines the path for the **chroot** jail. Specifying this keyword means that **rssh_chroot_helper** will be invoked, and the authenticating account **chroot**ed to the specified directory for all services it can access (assuming no per-user entry applies). As noted before, creating an **rssh**-based **chroot** environment is outside the scope of this paper.

per-user entries are made with the **user** keyword, and take the form:

```
user = < user ID >:< octal-specified umask value >:< access bits >:[optional chroot path]
```

The user ID must match the entry in */etc/passwd* and is the text string, not the numeric UID. The *umask* value is specified in octal notation, as with the corresponding global value.

There are five (5) *access bits*, with values of **0** or **1**, corresponding, in order, to **rsync**, **rdist**, **cvs**, **sftp** and **scp**. These operate similar to the global **allow*** keywords. If a specific bit is set to **1**, then access to that service is permitted for the user ID. If the bit is **0**, then access to that specific service is denied. These bits override any global settings, but only for the specific user.

The *chroot path* is optional, and only specified if you are **chroot**ing the user ID. If omitted, the user will not be **chroot**ed. This value allows you to place specific UIDs in specific **chroot** directories, while still being able to set a global value for all other UIDs. Once more, the nitty-gritty of **chroot** in the **rssh** environment is outside the scope of this paper (and is probably easier to do in OpenSSH v5.0p1).

You may edit *rssh.conf* with any text editor. There's no validation tool (such as is found with **sshd**), so it's a good idea to keep a user ID handy for the express purpose of testing edits. **rssh** will log an error if it has trouble parsing the *rssh.conf* file.

Tips for using OpenSSH and rssh effectively

Here are some tips and tricks that may be useful in your environment. It's important to stress that this paper is only an introduction to these tools, and there are further ways to leverage these tools to provide good communications security. A future paper will expand on the materials presented here.

OpenSSH

1) Keep on top of updates: OpenSSH is under active development, with new releases every few months. Some releases are minor improvements, while others address significant security issues or provide major improvements in your ability to configure/control the environment. Subscribe to the **openssh-unix-announce**[19] mailing list, a low-traffic, announcements-only list. Even if you use a pre-built package rather than building from source, keep an eye on developments with OpenSSH, so you know when getting and testing the new package is a priority (*e.g.* there is an important fix available) as opposed to merely routine.

2) Eliminate **telnet** and **FTP**: Once you have OpenSSH running, there's almost certainly no reason to continue to support **telnet** or **FTP** as access methods for the SSH-equipped server. And if there is a reason, then work on eliminating it. Running both sets of protocols is like replacing your home's front door with solid oak in a steel frame using a deadbolt, but leaving the back door hollow wood with a lock easily opened using a credit card.

3) Be sure to use **PrivSep**: The **Privilege Separation** function of OpenSSH is an important part of providing a secure remote access service. Don't start your SSH server without it. Avoid enabling the **UseLogin** option, which will break PrivSep. The user account for this should look something like:

```
# grep sshd /etc/passwd
sshd:x:2222:2222:Mr. SSH Daemon:/var/empty:/bin/false
```

And the Privilege Separation directory should look like:

```
# ls -la /var
....
drwxr-xr-x  2 root      other       512 May 28 11:26 empty/
....
```

To reduce the attack profile further, deny unauthenticated connections the ability to use of data compression (**Compression delayed**), or even turn off data compression (**Compression no**) entirely if it isn't of benefit in your environment.

4) Leverage subsystems: Defining subsystems is a convenient way to make specific functions available to users. The subsystem executes in the user's security context and shell, so a given user cannot execute a subsystem to which they didn't already have access. As of OpenSSH v4.4, the **subsystem** option supports command-line parameters.

While similar functionality is offered using *command* flags on key entries in the user's *authorized_keys* file, if users are allowed to edit their *authorized_keys* file, then they can alter what they are allowed to do. Subsystems centralizes the control of the function.

5) Use configuration test mode: When invoked with the **-t** parameter, **sshd** reads and validates its configuration file. It exits, reporting any problems to *stdout*. Whenever you make changes to *sshd_config*, it's a good idea to invoke **sshd -t** to verify the configuration file has no syntax errors.

6) Update back-leveled vendor versions: While many *NIX vendors have started to include OpenSSH (or their derivative of it) in their OS distributions (*e.g.* Sun Solaris v9 and later; IBM AIX v5.3 and later), some are quite tardy with their updates, and a few even are years behind in their pre-built configurations. When a vendor consistently fails to keep a tool like OpenSSH updated, or their updates are consistently many releases behind, consider building your own version from source, or finding a different source of the packages.

7) Consider **rssh** for **chroot** for older OpenSSH versions: Prior to OpenSSH v5.0p1, the stock OpenSSH environment did not support **chroot**ing SSH connections. For those older versions, there were workarounds/patches to add the functionality; however, those worked either by jailing the entire SSH server, or by patching the **sshd** server code to look for specific home directory strings so as to jail the user. **rssh** can accomplish the same task without having to jail the entire server or create home directory paths that may break other functions. The only caveat is that **rssh**-limited connections cannot access a shell, so if your goal is to provide users with **chroot**-ed shell access, then you cannot use **rssh**.

If you have OpenSSH v5.0p1 or later, then it includes **chroot** capability, and is probably a better way of implementing the function.

8) Pay attention to dependencies: OpenSSH depends on both the **OpenSSL** and **zlib** software libraries (in addition to the usual system libraries), and vulnerabilities in those softwares can potentially create exposures in OpenSSH. Keep track of security updates to OpenSSL and zlib and integrate them into your environment in a timely manner.

rssh

1) Keep **rssh** updated: While not as active a development project as OpenSSH (since **rssh** is the brainchild of one person), there are still irregular updates to **rssh**. The author has stated that with respect to feature/functionality, no further development will occur. That means that any updates are probably to fix security issues, and therefore worthy of attention. You can keep track of **rssh** via its **SourceForge**[20] project page.

2) Understand the limits imposed: By changing an account's shell specification to **rssh**, you are denying shell access to that user. The account is limited to the five access methods controlled *via* **rssh**: **scp**, **sftp**, **rdist**, **cvs** and **rsync**.

Any other commands defined using the *subsystem* option will not be available to the **rssh**-controlled user account, nor will any commands made available via entries in their *authorized_keys* file. Further,

any environment variables defined in *.ssh/environment*, in key file entries, or set by the SSH client using **SendEnv**, even if allowed by the **PermitUserEnvironment** or **AcceptEnv** option in *sshd_config*, will be ignored by **rssh**.

3) Don't change *root*'s shell to **rssh**: You probably weren't planning to do it, but just in case, it's not a good idea to use **rssh** as the shell for the *root* account. Leave *root* alone. This is especially true for Solaris environments.

> **Important!**
>
> It is impossible to over-stress the dangers of remote *root* login. A basic security premise in this paper is that remote access to privileged accounts should be blocked for any hosts with connections to untrusted networks (including the Internet). Even allowing such access within nominally "trusted" networks carries dangers. As a best practice, remote access should be limited to un-privileged accounts, and privilege escalated using appropriate tools (*e.g.* **sudo**, **su**, *etc.*). For whatever need may seem to "require" remote *root*, it's a virtual certainty that other ways exist to achieve the same goal, using methods that are more secure.

Helpful reference materials

The sample *sshd_config* file included in with the program is a bare-bones framework, and not all that well documented. Here is the template that I use:

```
# /path/to/your/sshd_config
#
#       $OpenBSD: sshd_config,v 1.74 2006/07/19 13:07:10 dtucker Exp $
#
# This is the sshd server system-wide configuration file.  See
# sshd_config(5) for more information.
#
# This sshd was compiled with PATH=/path/your/daemon/was/compiled/with
#
# The strategy used for options in the default sshd_config shipped with
# OpenSSH is to specify options with their default value where
# possible, but leave them commented. Uncommented options change a
# default value.
#
# Options with BOOLEAN values may be "yes" or "true" OR "no" or "false"
# but *must* be in lower-case
#
# Options with time values default to seconds, are additive, and values
# may optionally have a single-character suffix indicating the time unit:
#     7s = seven seconds      12m = twelve minutes
#     3h = three hours        8d = eight days
#     2w = two weeks          1h30m = ninety minutes
```

```
#
# IMPORTANT: Command-line options override settings made via this file
#
# Change Log
# Who       When        What
# ----      ----------  ----------------------------
#################################################################

##################
# Communications #
##################

# Specify the TCP Port to which sshd will bind (default is 22)
# Multiple Port statements may appear and sshd will listen on all indicated ports
# Equivalent command-line option: -p
#Port 22

# Specify IP address(es) that sshd will bind to at startup
# The default is all local addresses available
# Use these entries to limit the daemon to specific addresses
# IPv4
#ListenAddress 0.0.0.0
# IPv6
#ListenAddress ::
# Alternatively, the ListenAddress and Port may be combined in a single statement
#ListenAddress 127.0.0.1:22

# SSH Protocols that sshd will support
# Default is both 1 and 2
# Starting with v4.7p1, default is 2 only
Protocol 2

# Send "keepalive" packets to verify connection is still valid?
# Default is yes
TCPKeepAlive yes

# Terminate connection if client does not respond to
#     ClientAliveCountMax (defaults to 3) number of packets sent every
#     ClientAliveInterval (defaults to 0) seconds.
# The settings below give a client about 45 seconds
# to respond before the connection is cut (3 x 15 = 45)
ClientAliveInterval 15
ClientAliveCountMax 3

# Allow sshd to forward TCP connections from authenticated
#     SSH sessions to other ports on this host or on remote hosts
# Default is yes
# If you permit users to have shell access, then globally disabling
#   port forwarding is not an effective security measure, as users
#   will be able to run their own forwarding mechanisms
# NEVER enable if you have anonymous access to the host (e.g. AnonCVS)
AllowTcpForwarding no

# Allow sshd to bind forwarded ports to addresses other than 127.0.0.1
# Default is no
```

```
# Setting to yes allows a client to receive forwarded TCP packets
#     from any host that the server can receive from. Enabling
#     this has significant security issues and generally should not
#     be enabled without other control mechanisms, such as TCP Wrappers
#     or host-based firewalls.
#GatewayPorts no

# Limit the TCP ports to which forwarding is permitted
# Default allows any TCP port to be forwarded (subject to AllowTcpForwarding)
# If you allow TCP Forwarding, you should probably use this to
#     restrict which ports may be forwarded. This setting is not an
#     effective substitute for a more-comprehensive tool like a firewall
#     or TCP Wrappers
# Multiple whitespace-separated permissions may be listed on a single line
# The keyword “all” removes all restrictions (the default)
#PermitOpen host:port
#PermitOpen IPv4_addr:port
#PermitOpen [IPv6_addr]:port

# Max number of *unauthenticated* connections permitted at any
# one moment; authenticated connections do not count against this limit
# Defaults to 10; set to 0 for no limit
# May also be specified using A:B:C
#    where A is a lower bound, and once it is exceeded, there
#    is a B% chance that the next connection will be summarily
#    rejected. This chance increases in a linear fashion until
#    the number of connections reaches C, when the chance of refusal
#    becomes 100%. Roughly, each unauthenticated connection above
#    A adds to the chance of rejection by ((100 - B)/(C - A))%
MaxStartups 5

######################
# HostKey Locations #
######################
# Locations of the private key files that uniquely
#    identify this server
# HostKeys for protocol version 2
HostKey /opt/openssh/conf/ssh_host_rsa_key
HostKey /opt/openssh/conf/ssh_host_dsa_key

############
# Logging #
############
# These options obsolete QuietMode and FascistLogging from previous versions
# As of OpenSSH v4.4, these options also obsolete the SFTP-server-specific
#     configuration options LogSftp, SftpLogFacility, and SftpLogLevel

# Facility to which sshd logs
SyslogFacility AUTH

# Logging Level/Severity (default is INFO)
# Equivalent command-line option: -o "LogLevel VERBOSE"
LogLevel VERBOSE
```

```
##################
# Authentication #
##################

# Location (relative to user home directory) of user's public-key
#     authorization file
# May also be an absolute path
# Macro substitutions are available:
#     %h = User's home directory as defined in /etc/passwd or elsewhere
#     %u = User name
#     %% = A % sign
#AuthorizedKeysFile     .ssh/authorized_keys

# Grace period in which authentication must occur before connection is dropped
# Default is 120 seconds/2 minutes; 0 disables this (infinite time)
# Equivalent command-line option: -g
#LoginGraceTime 2m

# Limit the number of authentication attempts that may be
#    made using a single SSH connection (may discourage dictionary
#    attacks, or make them obvious); a side-effect is that a user
#    with multiple public keys in an identity file may exceed the
#    limit (so this option may not be useful in a key-oriented environment)
# Default is 6
MaxAuthTries 3

# Require important files and directories to have strictly limited permissions
#    Default is yes. When enabled, the locations must be owned by the user (or root)
#    and must NOT be group- or world-writable
# Locations checked:
#     User's home directory (~)
#     ~/.rhosts and ~./shosts
#     User's SSH configuration directory (~/.ssh)
#     User's SSH key files (~/.ssh/authorized_keys)
StrictModes yes

# Allow keyboard-interactive authentication?
# Default is yes
# Requires compile-time support for BSD-AUTH, PAM and/or SKEY; if these
#   were not included, then this option is ignored
#ChallengeResponseAuthentication yes

# Allow password-based authentication?
# Default is yes
# sshd normally uses the password in /etc/shadow, but may also use PAM or Kerberos
# Note that for NetWare, password authentication is the only supported method
#PasswordAuthentication yes

# Permit accounts with empty passwords to login?
# Default is no
# It is HIGHLY recommended that you leave this DISABLED (no)
#PermitEmptyPasswords no
```

```
# Allow SSH v2-style public-key authentication?
# Default is yes
#PubkeyAuthentication yes

# Allow SSH v2-style host-based authentication
# Default is no
# Uses /etc/hosts.equiv, ~/.rhosts, /etc/shosts.equiv and/or
#    ~/.shosts for authentication
# Host-based authentication is problematic and its use is not recommended
#HostbasedAuthentication no

# Allow hosts file in user's ssh directory to override previous setting?
# Default is no (which PERMITS the override)
# To ensure host-based authentication is disabled, these options
#     should be set to yes
# RhostsRSAAuthentication and HostbasedAuthentication
IgnoreUserKnownHosts yes
# Don't read the user's ~/.rhosts and ~/.shosts files
IgnoreRhosts yes

##################
# Access Control #
##################
# Is root allowed to login via SSH?
# Default is yes
# It is HIGHLY recommended you DISABLE this (no) - in most
#    environments, there is no reason root should login remotely
PermitRootLogin no

# List of user accounts (by name, not UID #) specifically
#    denied SSH access (even with an otherwise valid authentication, these
#    accounts are not allowed to login)
DenyUsers root daemon bin sys adm lp sshd uucp listen nobody noaccess nobody4 ftp

# List of user accounts (by name, not UID #) specifically
#    allowed SSH access (pending successful authentication via a permitted method)
#AllowUsers

# List of groups (by name, not GID #) specifically denied SSH access (even with an
#     otherwise valid authentication, accounts in these groups are not allowed to
#     login; does not require the group to be the account's Primary group)
#DenyGroups

# List of groups (by name, not UID #) specifically allowed SSH
#    access (pending successful authentication via a permitted method;
#    does not require the group to be the account's Primary group)
#AllowGroups

# NOTE ABOUT RESOLVING ALLOW/DENY CONFLICTS:
# sshd will adopt the most-restrictive interpretation of AllowUsers/AllowGroups
# and DenyUsers/DenyGroups. Thus, if any entry can be used to deny an
# account login, the account will be denied.
# The directives are consulted in the following order regardless of the
#     order they appear in this file: DenyUsers, AllowUsers, DenyGroups, AllowGroups
##############################################################################
```

```
####################
# User Environment #
####################
# List of environment variables that sshd will allow the client
#    to set using the SendEnv option in the client configuration
# Default is an empty string, which causes sshd to ignore all such client requests
#AcceptEnv

# Specify the directory to which the server should chroot() a session
#     Most useful within a Match block
# Chrootdirectory

# Override any command requested by the client and force execution of the following
#     command. Execution occurs in the user's shell environment, with the -c option
# NOTE: Generally, this is most useful in a Match block
# ForceCommand

# Allow the user to set environment variables using
#    ~/.ssh/environment and options in their authorized_keys file
# Default is no
# If you are using rssh, then do NOT enable this, as it will
#    create holes that a user can exploit to break rssh security
PermitUserEnvironment no

######################
# Encryption Ciphers #
######################
# NOTE: These options do not force a particular selection order,
#       they merely limit the ones sshd will allow a client
#       to use; clients must support at least one of
#       the listed algorithms

# List of permitted data encryption algorithms
Ciphers aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc
Ciphers arcfour128,arcfour256,arcfour,aes192-cbc,aes256-cbc,aes128-ctr
Ciphers aes192-ctr,aes256-ctr

# List of permitted integrity-checking algorithms
MACs mac-md5,hmac-sha1,hmac-ripemd160,hmac-sha1-96,hmac-md5-96
# List of permitted integrity-checking algorithms (v4.7p1 added umac-64)
MACs mac-md5,hmac-sha1,hmac-ripemd160,hmac-sha1-96,hmac-md5-96,umac-64

####################
# Kerberos options #
####################
# These options are ignored unless OpenSSH was compiled
# with Kerberos authentication support; default values are shown
# Discussion of these options is outside the scope of this paper

# Enable direct Kerberos authentication
# Equivalent command-line option: -K
#GSSAPIAuthentication no

# Delete Kerberos-forwarded credentials on logout
#GSSAPICleanupCredentials yes
```

```
# Allow sshd to submit user's SSH password for
#   Kerberos authentication (indirect method)
#KerberosAuthentication no

# Allow local password authentication if Kerberos fails
#KerberosOrLocalPasswd yes

# Delete Kerberos-forwarded credentials on logout
#KerberosTicketCleanup yes

# Requires AFS support; instructs sshd to attempt to get an AFS
#    token prior to accessing the user's home directory
#KerberosGetAFSToken no

###############
# PAM options #
###############
# Enable PAM support?
# Defaults to no
# Set this to 'yes' to enable PAM authentication (via challenge-response)
# and session processing. Depending on your PAM configuration, this may
# bypass the setting of 'PasswordAuthentication' and 'PermitEmptyPasswords'
#UsePAM no

###############
# X11 options #
###############
# Enable forwarding of X11 connections
# Default is no
# Will be disabled if UseLogin is enabled
#X11Forwarding no

# Reserve X11 Display numbers so sshd won't try to use them
# Default is 10
# Prevents sshd from clashing with existing X servers
#X11DisplayOffset 10

# Require sshd to emulate pre-v3.1 X11 server behavior
# Default is yes
# Use with older X11 clients
#X11UseLocalhost yes

#################
# Other options #
#################
# Allow use of data compression in SSH links?
# Default is delayed
# Compression is requested by the client, but this setting allows globally enables
#     or disables it; compression is generally not useful in the LAN environment
#     (and may actually hurt performance), but can help in WAN or extranet
#     environments; third option, delayed, ignores compression requests until
#     connection is authenticated
Compression delayed
```

```
# Force sshd to use system's login program?
# Default is no
# There is generally no need to enable this option, doing so
#    may compromise Privilege Separation, and will break X forwarding
#UseLogin no

# Banner message displayed prior to authentication
# Must be full, absolute path
# Default is no pre-authentication banner
#Banner /etc/warning.txt

# Print /etc/motd after successful authentication?
# Default is yes
# If your shells print motd by default, this option may be redundant
# NOTE: sshd will obey ~/.hushlogin
#PrintMotd yes

# Print date/time of user's last login?
# Default is yes
# NOTE: sshd will obey ~/.hushlogin
#PrintLastLog yes

# Use privilege separation user ID
# Default is yes
# sshd was hardcoded at compile-time to use a specific user
#    account - it is "sshd" unless something else was specified
UsePrivilegeSeparation yes

# Require a connecting host to have DNS reverse name-resolution?
# Defaults to yes
# Unless you control DNS for all hosts legitimately connecting, this
#    option is not of much value, and probably not worth the overhead
UseDNS no

# Full path to daemon's PID file
# Defaults to /var/run/sshd.pid
# Ignored in debug mode
PidFile /var/run/sshd.pid

# Specify commands that may be executed by remote clients
# Default is no subsystems
# Format:  Subsystem  <shorthand name>  <full path to executable> [parameters]
# Full path *must* be specified; as of OpenSSH v4.4, command-line options
#     are supported (in prior versions, sshd would refuse to start
#     if a “Subsystem” directive included a command-line parameter)
# Shorthand names are case-sensitive
# Maximum of 256 subsystems may be defined
Subsystem   sftp  /opt/openssh/libexec/sftp-server

#########################
# Protocol v1 settings #
#########################
# The configuration above disables SSH v1
# The following settings are ignored, but shown here for documentary purposes
```

```
# Lifetime and size of ephemeral version 1 server key
#KeyRegenerationInterval 1h
#ServerKeyBits 768

# Allow SSH v1-style public-key authentication
# Default is yes
# Changing to "no" is redundant if SSH v1 support is disabled
#    in the Protocols keyword
#RSAAuthentication yes

# Allow SSH v1-style host-based authentication
# Default is no
# Requires host key in /path/to/openssh/configuration/ssh_known_hosts
#RhostsRSAAuthentication no

# HostKey for protocol version 1
#HostKey /opt/openssh/etc/ssh_host_key

############
# Matches #
############
# Match attempts to match a session based on any of the following criteria:
#               User            Group
#               Host            Address
# If there is a match, then the subsequent options are applied, overriding any
#     global settings for those options made above. Options are read until EOF or
#     until another Match statement is encountered (whether or not it is a
#     successful Match). The available Options in a Match block are:
#           AllowTcpForwarding          Banner                  ChrootDirectory
#           GatewayPorts                GSSAPIAuthentication
#           HostbasedAuthentication     KbdInteractiveAuthentication
#           KerberosAuthentication      PasswordAuthentication
#           PermitOpen                  PermitRootLogin
#           PasswordAuthentication      RhostsRSAAuthentication
#           RSAAuthentication           X11DisplayOffset        X11Forwarding
#           X11UseLocalHost

# Display a special banner for "internal" clients
Match Address 10.1.2.*
      Banner /etc/issue.internal

# Allow user bob to forward TCP packets, including those from
#     other hosts
Match User bob
      AllowTcpForwarding yes
      GatewayPorts Yes

#################################
## End of /path/to/sshd_config ##
#################################
```

This simple shell script automates running the **configure** script with the options suggested above for OpenSSH:

```
#!/usr/bin/bash
./configure --prefix=/opt/openssh --sysconfdir=/opt/openssh/conf \
      --with-zlib=/usr/lib --with-ssldir=/opt/openssl \
      --with-privsep-path=/var/empty --with-privsep-user=sshd \
      --with-pid-dir=/var/run --with-mantype=man
```

While well commented, the sample *rssh.conf* file from the package omits some information. This template may be slightly clearer:

```
##############################################
#
# /opt/rssh/conf/rssh.conf
#
# Configuration file for SSH Restricted Shell (rssh) v2.3.2
#
# Change Log:
# Who       When        What
# ----      ----------  ------------------------------------
#
##############################################
############
# Logging #
############
# Set the log facility
# Default is LOG_USER
# The syntax "LOG_USER" and "user" are equivalent (both
#     will log to Facility USER)
# Can be any valid Facility
logfacility = LOG_AUTHPRIV

####################
# Global Defaults #
####################
# Sets defaults for all users who do not have a Per-User entry below
# If a line is commented, the activity is disabled

# Secure CoPy
#allowscp

# Secure FTP
#allowsftp

# CVS code management
#allowcvs

# rdist
#allowrdist

# rsync
#allowrsync
```

```
#########
# UMASK #
#########
# Specify the default umask for all users who do not have a Per-User
#     entry below
# Requires octal notation
umask = 077

##########
# Chroot #
##########
# If you want to chroot users, use this to set the directory where the root of
# the chroot jail will be located.
#
# If you DO NOT want to chroot users, LEAVE THIS COMMENTED OUT.
# chrootpath = /usr/local/chroot

# You can quote anywhere, but quotes not required unless the path contains a
# space... as in this example.
#chrootpath = "/usr/local/my chroot"

###################################
# Per-User Confguration Options #
###################################
# Format:
#  user=<user name>:<UMASK>:<Access Bits>:<optional chroot path>
#
# where
#
#   <user name> is the name as it appears in /etc/passwd (not the numeric)
#   <UMASK> is the octal-notated file creation mask
#   <Access Bits> are either 0 or 1 and correspond, in order, to
#      rsync, rdist, cvs, sftp and scp
#   <chroot path> is not required and specifies a path to chroot the user to
#
# The final colon is required only if a chroot path is specified

# From the supplied file:
# EXAMPLES of configuring per-user options
#user=rudy:077:00010:  # the path can simply be left out to not chroot
#user=rudy:077:00010   # the ending colon is optional

#user=rudy:011:00100:  # cvs, with no chroot
#user=rudy:011:01000:  # rdist, with no chroot
#user=rudy:011:10000:  # rsync, with no chroot
#user="rudy:011:00001:/usr/local/chroot"  # whole user string can be quoted
#user=rudy:01"1:00001:/usr/local/chroot"  # or somewhere in the middle, freak!
#user=rudy:'011:00001:/usr/local/chroot'  # single quotes too

# if your chroot_path contains spaces, it must be quoted...
# In the following examples, the chroot_path is "/usr/local/my chroot"
#user=rudy:011:00001:"/usr/local/my chroot"  # scp with chroot
#user=rudy:011:00010:"/usr/local/my chroot"  # sftp with chroot
#user=rudy:011:00011:"/usr/local/my chroot"  # both with chroot
```

```
# Spaces before or after the '=' are fine, but spaces in chrootpath need
# quotes.
#user = "rudy:011:00001:/usr/local/my chroot"
#user = "rudy:011:00001:/usr/local/my chroot"  # neither do comments at line end

################################
## End of /path/to/rssh.conf ##
################################
```

Another simple shell script to automate running the **configure** script with the options suggested above for **rssh**:

```
#!/usr/bin/bash
#
./configure --prefix=/opt/rssh --sysconfdir=/opt/rssh/conf \
      --with-sftp-server=/opt/openssh/libexec/sftp-server
```

SSH, The Secure Shell, The Definitive Guide, Second Edition by Barrett, Silverman & Byrnes, ISBN 0-596-00895-3 (O'Reilly)[21]

Let me start with the fact this book is useful, and does contain a lot of good information. That said, it can be confusing to read, because it doesn't always do a good job of changing gears when it switches from talking about the **OpenSSH** package to the commercial **Tectia** SSH implementation. The reader has to be careful to distinguish between the two when the book neglects to make the focus change explicit. Also, despite being "definitive", I found the omission of even a mention of **rssh** to be a glaring deficiency.

Footnotes

[1] http://www.rfc-archive.org/getrfc.php?rfc=854
[2] http://www.rfc-archive.org/getrfc.php?rfc=959
[3] http://www.rfc-archive.org/getrfc.php?rfc=4251
[4] http://www.rfc-archive.org/getrfc.php?rfc=4252
[5] http://www.rfc-archive.org/getrfc.php?rfc=4256
[6] http://www.openssh.org
[7] The specifics of support for platforms vary by the platform. For example, the QNX implementation doesn't support post-authentication **PrivSep**. While NetWare has included OpenSSH for years, the NetWare console environment is not analogous to the *NIX terminal/shell environment; hence, **rssh** can't really function on the NetWare/OES-NetWare platform (thus, with regards to Novell's production, this paper is really only applicable to NetWare's successor, Open Enterprise Server, specifically OES-Linux; or SLES).
[8] http://www.pizzashack.org/rssh
[9] http://www.rfc-archive.org/getrfc.php?rfc=2228
[10] http://www.rfc-archive.org/getrfc.php?rfc=4217
[11] http://www.rfc-archive.org/getrfc.php?rfc=2946
[12] http://www.rfc-archive.org/getrfc.php?rfc=3205
[13] http://www.sunfreeware.com

[14] ftp://ftp.porcupine.org/pub/security/index.html
[15] Virtually every Linux distribution includes the OpenSSH package, and practically all of them link OpenSSH against TCP Wrappers. If you choose to compile your own OpenSSH (perhaps to get a newer version than the distribution maintainer provides), then linking against TCP Wrappers is recommended.
[16] If you subsequently perform an in-place upgrade of OpenSSH, the *moduli* file is not replaced if the install script detects an existing file. The extremely security-conscious (*i.e.* really paranoid) may consider periodic re-creation of the *moduli* file to thwart cryptographic analysis attacks that try to determine the primes used in key generation as a method to narrow keyspace. Of course, if the reader is actually worried about an attack on that level, then the reader has problems no technical paper can solve.
[17] As of OpenSSH v4.7p1, the default configuration file only configures support for SSH v2 (although the package continues to support both versions of the SSH protocol). If you install v4.7p1 atop an existing, earlier, OpenSSH installation, the configuration is not changed in this respect.
[18] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2005-2096
[19] http://www.mindrot.org/mailman/listinfo/openssh-unix-announce
[20] http://sourceforge.net/projects/rssh/
[21] http://www.oreilly.com/catalog/sshtdg2/index.html

Change Log

Version	Date	Change
0.10	2006-Mar-11	Initial creation
1.00	2006-Mar-17	Initial Publication
1.05	2006-Apr-20	Fleshed out OpenSSH configuration options
1.10	2006-Apr-26	Finished OpenSSH configuration options; added Tips
1.15	2006-May-07	Corrected some typos and minor formatting issues; added References for **rssh**; corrected ownership/mode information for *sshd_config*
1.20	2006-Jun-10	More typo fixes; completed Tips
1.25	2006-Jun-11	PDF Version
1.30	2006-Jun-16	Integrated changes from HTML version; fixed typos; minor formatting changes; added Footnotes section
1.31	2006-Jul-14	Added information on client connection negotiation issues with the **Compression delayed** setting in the SSH server; minor documentation edits to *sshd_config* and *rssh.conf* templates
1.40	2006-Aug-20	Clarified TCP Wrappers integration with OpenSSH; fixed minor formatting inconsistencies
1.45	2006-Dec-16	Updated for OpenSSH v4.5p1; fixed minor typos and formatting
1.50	2007-Jan-08	Added information about **Match** directives
1.51	2007-Jan-27	Expanded **Match** information; fixed minor typos and formatting; clarified **Deny/AllowUsers/Groups** precedence in example configuration file (PDF only)
1.52	2007-Jan-29	Fixed minor typos and formatting (PDF only)
1.53	2007-Mar-19	Reformatted with open font (PDF only)
1.54	2007-May-03	Updated for OpenSSH v4.6p1; minor formatting edits (PDF only)
1.55	2007-May-07	Added example **Match** directives; updated HTML version for OpenSSH v4.6p1 and with information from PDF footnotes; minor formatting and typo fixes
1.56	2007-May-13	Minor text updates
1.57	2007-May-17	Fixed a minor typo
1.60	2007-Jun-07	Minor formatting updates
1.65	2007-Nov-11	Updated for OpenSSH v4.7p1 (HTML version only)

1.70	2007-Dec-14	Updated for OpenSSH v4.9p1; additional minor text changes
1.71	2008-Apr-14	Updated for OpenSSH v5.0p1 (PDF version only); typo fixes
1.75	2008-Dec-10	Revision of tips for OpenSSH to accurately re-cap chroot; minor typos fixed; corrected copyright statement in per-page headers (PDF version only)

End of Document