Pro OpenSSH

ISBN (pbk): 1-59059-476-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewer: Darren Tucker
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Managers: Beckie Stones and Laura Brown
Copy Edit Manager: Nicole LeClerc
Copy Editors: Ami Knox and Damon Larson
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Kinetic Publishing Services, LLC
Proofreader: Lori Bring
Indexer: Michael Brinkman
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 4

The OpenSSH Server

Chapter 3's coverage of the OpenSSH file structure sets the stage for this chapter, in which you will learn about the server configuration options and directives within the OpenSSH main configuration file, `sshd_config`. This chapter also offers some troubleshooting hints and general management tips. More advanced management tips, including architecture and key management, are covered in Chapter 8.

The next chapter will discuss the OpenSSH client and its configuration in detail. The goal for these two chapters is to provide you with the knowledge to make your OpenSSH implementation as secure and usable as possible. The discussion of the server will begin by providing information that will enable you to quickly identify mistakes in configuration files, and then move into instructions for running a daemon in debug mode. Following the debugging discussion, I cover the order of precedence for the server configuration files and directives, with a detailed at look at the directives in the `sshd_config` file.

OpenSSH Server Testing

When the OpenSSH daemon becomes the lifeblood of your administration infrastructure, careful change control is essential for optimal operation. Making a mistake in a configuration file can be the difference between having the newest, coolest script known to UNIX and having to drive out to that remote data center to reconfigure your OpenSSH server.

Checking the Syntax of sshd_config

By default, the OpenSSH server, or `sshd`, resides on TCP port 22. But when testing a new configuration on a server where connectivity services must remain available, it is best practice to place the `sshd` daemon on a different port to test functionality. Before using a new `sshd_config` file, whether it be on port 22 or in testing, it is a good idea to run `sshd -t`, which validates the syntax of the server configuration file. Keep in mind that the permissions on `sshd_config` should only allow for root to read from and write to this file, so running `sshd -t` against a properly secured configuration file requires root authority. In the following example, the `sshd_config_new` file has the invalid string `Version`. This was designed to be a comment, but because it was not prefaced with the # character, `sshd` sees it as an unknown directive. Running `sshd -t` locates the error and provides the line number where the problem occurs.

```
rack:/ # head -4 /etc/ssh/sshd_config_new
Version 2
# This is the sshd server system-wide configuration file.  See
# sshd_config(5) for more information.
rack:/ # /usr/sbin/sshd -t -f /etc/ssh/sshd_config_new
/etc/ssh/sshd_config_new: line 2: Bad configuration option: Version
/etc/ssh/sshd_config_new: terminating, 1 bad configuration options
rack:/ #
```

As with many UNIX commands, no output from running `sshd -t` means everything is fine.

Changing the Default Configuration File and Port

By default, `sshd` will use TCP port 22. However, if an alternative configuration file for your server is desired, and attempting changes on the production OpenSSH server already found on port 22 could create issues, you can run a separate instance on an alternative port. In the following example, the file `sshd_config_new` has been created, and it contains some different server options and needs to be tested.

```
rack:/ # /usr/sbin/sshd -f /etc/ssh/sshd_config_new -p 99
```

This tells `sshd` to start utilizing `/etc/ssh/sshd_config_new` as the server configuration file and to listen on port 99. While any port can be used, make sure you choose a port not already in use on your system. To establish a connection to the newly created port 99 server, use the `ssh` client software with the `-p` option, which specifies the remote port to connect to.

```
rack:/ # /usr/bin/ssh -p 99 127.0.0.1
```

Running the OpenSSH Server in Debug Mode

The `-d` option of `sshd` is another tool that will help you troubleshoot your OpenSSH configuration if you suspect something is not behaving as desired. The debug mode of `sshd` logs output to `stderr` instead of to the default `AUTH` facility of `syslogd`.

```
rack:/ # /usr/sbin/sshd -d -p 99 -f /etc/ssh/sshd_config_new
debug1: sshd version OpenSSH_3.9p1
debug1: private host key: #0 type 0 RSA1
debug1: read PEM private key done: type RSA
debug1: private host key: #1 type 1 RSA
debug1: read PEM private key done: type DSA
debug1: private host key: #2 type 2 DSA
debug1: rexec_argv[0]='/usr/sbin/sshd'
debug1: rexec_argv[1]='-d'
debug1: rexec_argv[2]='-p'
debug1: rexec_argv[3]='99'
debug1: rexec_argv[4]='-f'
debug1: rexec_argv[5]='/etc/ssh/sshd_config_new'
debug1: Bind to port 99 on ::.
Server listening on :: port 99.
```

```
debug1: Bind to port 99 on 0.0.0.0.
Bind to port 99 on 0.0.0.0 failed: Address already in use.
Generating 768 bit RSA key.
RSA key generation complete
```

As you can see, the output from debug mode is sent to the terminal. This will continue as long as this debug session of `sshd` is running. To stop it, press Ctrl-C from the controlling window or use the `kill` command. Of course, the debugging information can also be redirected to a file for further analysis. If a user logs in to a daemon running in debug mode, he or she is also presented with some additional information.

```
stahnke@rack: ~> /usr/bin/ssh -p 99 localhost
stahnke@localhost's password:
Environment:
  USER=stahnke
  LOGNAME=stahnke
  HOME=/home/stahnke
  PATH=bin:/usr/bin:/usr/local/bin
  MAIL=/var/mail/stahnke
  SHELL=/bin/bash
  SSH_CLIENT=::ffff:127.0.0.1 55894 99
  SSH_CONNECTION=::ffff:127.0.0.1 55894 ::ffff:127.0.0.1 99
  SSH_TTY=/dev/pts/4
  TERM=xterm
stahnke@rack: ~>
```

Additionally, information can be gathered by increasing the verbosity of debugging simply by using `-dd` or `-ddd`, and recent versions of OpenSSH may also need `-e` to force the debug messages to appear on `stderr`. Level three is the highest debug level for OpenSSH servers.

Reloading Configuration Files

Sometimes the need arises to change a directive on a running `sshd` server. After verifying the configuration file syntax and testing the new configuration as necessary, edit the main configuration file with your changes. Then to reload an SSH configuration, you can simply run `HUP` or `kill -1` on the PID. Sending a `-1` or `HUP` signal does not drop any connections currently utilizing the `sshd` server. If the server cannot restart for any reason, possibly due the configuration file having a syntax error, then the server will silently die.

Managing the OpenSSH Server

The contents of the OpenSSH server file, `sshd_config`, are set for the entire system. The server tells the client what authentication methods are offered, what encryption types to use, and so on. The client then has the choice of selecting from those offered settings when connections are established. When selecting the options that compose the `sshd_config` file, administrators must take extra care to ensure usability while at the same time striving for a configuration that mitigates risks their organization has deemed too great to accept.

As with any configuration file, keep in mind the defaults are not always suitable to your needs. Version changes, upgrades, or just forgetting what the defaults are can lead to undesirable behavior in your OpenSSH environment. I normally make a copy of the `sshd_config` file that includes heavy commenting to have as a reference for auditors, compliance managers, and technical reviews. When it comes to installing them onto machines, I often utilize the power of `grep` to strip the comments, leaving a bare-bones configuration file for my machines.

Also keep in mind that maintaining backup copies of configuration files, whether via manual backups or version control, or just keeping MD5 hashes around to ensure integrity is always considered best practice. Just because OpenSSH is a secure connectivity suite is no reason to side-step good system administration techniques for managing and protecting the appropriate configuration files.

Proper planning and architecture are essential when building a configuration for `sshd`. The settings chosen for the OpenSSH server will directly impact how users connect and which client options (as discussed in Chapter 5) will be usable for them. Security practices are established for the OpenSSH environment within the `sshd_config` file. If poor security is implemented in the server configuration, clients can, and most likely will, take advantage of it.

For example, allowing empty passwords seems like a bad idea, but if you accidentally enable that possibility, the benefits of having encrypted communication to protect authentication credentials have been trumped by the lazy user who decides passwords are more of a hindrance than a protective measure for him or her. The lazy user still has an encrypted session, without a password, but any user can, and probably will, try to authenticate without a password. Network sniffing is not required in this case, because poor security practices were accidentally in place. Of course, the operating system also would need to allow passwords for this to be real risk, but this is nonetheless demonstrative of the type of care required when selecting several directives for the server provided with OpenSSH.

Every administrator has encountered a user who thinks the rules apply to everyone except for him or her. Your job in planning your OpenSSH implementation is to make sure even the most innovative of users cannot bypass the intended security of your environment. Luckily, with OpenSSH, this is easy to do. The `sshd_config` file cannot be overridden by the client, therefore if the system administrator controls the `sshd_config` file, the user has no technical way of preventing the settings from affecting him or her. Of course, the user can still make a case to management about how the rules provided by the system administration team are ridiculous and prevent his or her productivity.

Several questions come into play when deciding what the right choice is for your particular application of OpenSSH. Keep these in mind as you study configuration options, and they will be addressed with best practices toward the end of the chapter in a sample configuration file.

- Do you have environments that require a stricter level of security than others—that is, DMZs, extranets, government contract networks, and so on?
- Are you intending to allow any user on a given system `ssh` access?
- Do you plan to make use of port forwarding? (Port forwarding is covered in Chapter 7.)
- Does your environment rely on the X-Windows System?

OpenSSH sshd_config

The OpenSSH `sshd_config` file is found in the configuration directory for OpenSSH along with your host keys and master client configuration file. Most installations place `sshd_config` in `/etc/ssh`; however, if you installed from source and did not specify another configuration directory, then the files will be found in `/usr/local/etc`. The `sshd_config` file is read at `sshd` startup time and not read again unless specifically told to by a command-issuing `HUP` signal, as was discussed earlier.

The `sshd_config` file should be readable and writable only by the root user. Allowing users to read this file may allow them to search for bypasses and exploits on particular settings. The server configuration file is fairly straightforward to manage. A # indicates a comment, which continues until the end of a line. Keywords are case insensitive, but arguments are not. For example, `PermitRootLogin no` can be expressed as `permitrootlogin no`. Also, many features of the OpenSSH server configuration are not used on a regular basis. Some settings are specific to working within certain system configurations, such as the existence of smart cards or Kerberos implementations. Other settings involve only SSH protocol 1, and thus will have little need to be included if protocol 1 is specifically disabled. The `sshd_config` file provided with the source or from your distribution of OpenSSH is probably a very good starting point from which to build a configuration file. Normally, the configuration files have a number of comments that explain the reasoning behind several of the settings. In the discussion that follows, each directive will appear in alphabetical order. Following the presentation of the directives, I will show you some scenarios in order to help you decide what type of configuration will benefit your environment the most.

AcceptEnv

By default, environment variables provided from the client (using the `SendEnv` directive) are not parsed. By enabling `AcceptEnv`, a client can send over environment variables upon connection. This can be convenient for setting `$PATH` and other variables, but can also lead to a user bypassing some security settings. Each instance of `AcceptEnv` uses the variable the OpenSSH server should allow as an argument. Multiple variables can be separated by a space or put on separate `AcceptEnv` lines. Variables can also be expressed with simple regular expressions.

Because this directive requires settings in the client configuration also, a complete example is provided in Chapter 5 in the section “SendEnv.”

```
AcceptEnv       PATH
AcceptEnv       DB_HOME
```

AllowGroups

The `AllowGroups` directive pertains to UNIX/Linux groups normally found in the `/etc/group` file (or similarly provided entities inside of LDAP/NIS, etc.). The token can grant specific groups usage of the OpenSSH server. Simple regular expressions, such as * and ? characters, can be used to express multiple groups. The argument requires the group to be used by name, not by GID. Multiple groups can be used with spaces between each name. For example, to allow everyone in the users, user, and wheel group access to the OpenSSH server, the following line could be placed into the `sshd_config`. The `*admin` allows anyone in the linuxadmin, dbadmin, and ldapadmin groups to use the `sshd` server.

```
AllowGroups user* wheel *admin
```

AllowTCPForwarding

`AllowTCPForwarding` tells the `sshd` server whether or not it should allow tunneled connections of other TCP protocols over SSH. Tunneling is discussed in detail in Chapter 7, but at this point, keep in mind that tunneling is sometimes a security risk. Malicious protocols or applications can be tunneled, and an administrator will not always be able to detect its behavior, only its existence.

Disabling the forwarding will prevent users from forwarding TCP connections over OpenSSH. Some security organizations prefer to disable forwarding because it is difficult to see what types of connections are established and whether they are persistent.

Many times tunnels are used when crossing firewalls between a low-risk area and a higher-risk network. If a tunnel is used, a user (malicious or otherwise) could use the tunnel to connect to systems in the lower risk area. This behavior is normally not desired because high-risk systems do not initiate communication with low-risk machines.

The `AllowTcpForwarding` directive takes a `yes/no` argument:

```
AllowTCPForwarding no
```

AllowUsers

Much like the `AllowGroups` directive, `AllowUsers` specifies, by name, users permitted to connect to the OpenSSH server. Multiple names are delimited using a space. The default is to allow all local users access to `sshd`. Simple regular expressions are allowed. Additionally, a pattern can consist of `user@host`. The user and host are then each checked, which can allow users from controlled machines. This is a nice feature in a large network where the dba account, or something similar, might be different people coming from different machines. This allows you to make that distinction to grant only the proper users access.

```
AllowUsers stahnke  *admin dba@trustedhost
```

AuthorizedKeysFile

The `AuthorizedKeysFile` directive tells the OpenSSH server the location of file containing the public key strings for users to authenticate against. The default location is `$HOME/.ssh/authorized_keys`. The file `authorized_keys2` is also checked to allow for backward compatibility. On legacy systems, sometimes symbolic links are made between `authorized_keys` and `authorized_keys2`, which can help in ensuring authentication via public keys will take place as desired. This directive takes a path relative to the user's home directory as an argument, as the following example shows:

```
AuthorizedKeysFile .ssh/authorized_keys
```

Key authentication is covered in Chapter 6.

Banner

Many organizations require some legal verbiage such as the following before accessing a network or networked device:

```
Warning: Unauthorized access to this system is strictly prohibited.
Use of this system is limited to authorized individuals only.
All activities are monitored.
```

The `Banner` directive specifies a path to the file that should be displayed before login occurs. Many times it will contain notices about monitoring activity, and usage permitted per security policy, etc. No banner is enabled by default. The `Banner` directive does not apply to protocol 1. Following is an example of this directive:

```
Banner /etc/issue
```

ChallengeResponseAuthentication

The `ChallengeResponseAuthentication` directive allows for authentication using one-time passwords that use S/Key algorithms originally invented at Bell Labs. This was designed to make eavesdropping on network transmissions useless, as the authentication credentials expire after a single use. OpenSSH supports S/Key authentication, but eavesdropping on network connections is not a significant concern because OpenSSH encrypts all traffic sent across the network. The possible arguments to this directive are `yes` and `no`.

Ciphers

The `Ciphers` directive specifies which cipher to use for encryption using protocol 2. Multiple ciphers can be specified and separated via commas. The default value for the `Ciphers` directive is `aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,arcfour,aes192-cbc,aes256-cbc,aes128-ctr,aes192-ctr,aes256-ctr`.

For most practical purposes, the default `Ciphers` arguments are acceptable. AES is a very fast and strong cipher, which is used by default. The vast array of cipher options is chosen to be compatible with many different types of clients and programming libraries.

Selecting the encryption methods can become a debate with an encryption specialist due to performance gains of certain algorithms and whether or not some encryption methods and strengths can be exported and used internationally. Please remember that in certain parts of the world, lower levels of encryption are required such as 128-bit instead of 256-bit. It is recommended that encryption experts be consulted before implementing international encryption schemes. Take some time to determine what circumstances apply to your organization.

ClientAliveCountMax

The `ClientAliveCountMax` directive sets the number of attempts the OpenSSH server should make contacting the client before issuing a disconnect. This is dependent on `ClientAliveInterval` being set. Three attempts is the default value for this directive. In the following example, ten attempts is specified. Multiplying `ClientAliveCountMax` by `ClientAliveInterval` will tell you how long a session can be idle before it is terminated. If the result is zero, the session will stay connected indefinitely.

```
ClientAliveCountMax 10
```

With the `ClientAliveCountMax` and `ClientAliveInterval` selected, a connection can be nonresponsive for 100 seconds before it is dropped.

ClientAliveInterval

Using protocol 2, the OpenSSH server can send a request for response from the client. If a response is received, the connection continues. If no response is received, `sshd` will send a request for response `ClientAliveCountMax` number of times before dropping the connection. By default, the server does not request responses from the client, which is a setting of `0`.

Some firewalls drop connections if they are detected as idle. The `ClientAliveInterval` directive can prevent that because messages and acknowledgements will be sent between the client and the OpenSSH server.

Following is an example of this directive:

```
ClientAliveInterval 10
```

Compression

The `Compression` directive directs OpenSSH as to whether or not compression is enabled. The arguments are `yes` (the default, as shown in the syntax example) and `no`. Enabling this feature entails a small CPU overhead, but it also can be overcome by transferring less data.

```
Compression yes
```

DenyGroups

The `DenyGroups` directive will deny UNIX groups based on name. If a user is member of a group on this list, the user is not permitted to use `sshd`. By default, all group members are allowed to log in. This directive recognizes simple regular expressions, and multiple arguments are separated by spaces:

```
Denygroups wheel lp bin ?adguy
```

DenyUsers

`DenyUsers` will ban specific users based on name. If you have multiple users with the same UID (not a best practice), all of their names need to be specified to eliminate that user. Simple regular expressions are parsed. Multiple arguments are separated by spaces. By default, all users are allowed to use `sshd`. If a `user@host` pattern is specified, then user and host are evaluated separately. This can prevent certain users coming from untrusted machines.

If you are using `AllowUsers` and `DenyUsers` simultaneously, a wildcard on `DenyUsers` as shown in the following example will override anything you have on `AllowUsers`. It is not possible to say `DenyUsers *`, but `AllowUsers root`.

```
DenyUsers stahke@badhost.com badguy hack*
```

GatewayPorts

When using forwarding, normally the forwarded connection is only bound to the localhost, because it binds to the loopback address. If gateway ports are specified, remote machines can use a forwarded port on the local system because the forwarded connections bind to all addresses

on the machine. This can sometimes defeat the point of tunneling, or it can allow for sophisticated setups, such as remote machines connecting to bastion hosts through firewalls for tunneled connections. The `GatewayPorts` directive is disabled by default (set to `no` as the example shows); you can enable it by setting it to `yes`. Most often, the default is adequate unless a specific application of the technology is in mind. More information on tunnels is provided in Chapter 7.

```
GatewayPorts no
```

GSSAPIAuthentication

The GSSAPI is a relatively new application programming interface (API) that allows for programs to use common mechanisms for authentication. Thus far, most GSSAPI work has been done in conjunction with Active Directory. GSSAPI usage for UNIX systems currently consists of Kerberos implementations. The `GSSAPIAuthentication` directive is disabled by default and only works with protocol 2.

GSSAPICleanupCredentials

If using GSSAPI, it is a best practice to destroy user credentials that are cached when the user's session ends; otherwise a malicious user could reuse your authentication credentials to pose as another identity. This is done by default. The `GSSAPICleanupCredentials` directive takes `yes/no` arguments and applies only to protocol 2.

HostbasedAuthentication

The `HostbasedAuthentication` directive tells OpenSSH whether or not host-based authentication is permitted via a `yes/no` argument, as shown in the example that follows. This directive applies only to protocol 2. Because the intention is to derive the maximum security out of OpenSSH, host-based authentication is not recommended unless evaluated thoroughly and the risk accepted. More discussion on host-based authentication occurs in Chapter 6.

```
HostbasedAuthentication no
```

HostKey

The `HostKey` directive requires a path to the private host key as an argument. By default, the file `/etc/ssh/ssh_host_key` is used for protocol 1. The files `/etc/ssh/ssh_host_rsa_key` and `/etc/ssh/ssh_host_dsa_key` are used for protocol 2. Protection of the host key files is paramount, and OpenSSH will refuse to use host keys that are accessible to the group or world. Multiple host keys are permitted. Protocol 1 uses `rsa1` keys. DSA and RSA key algorithms are used in the version 2 implementation of OpenSSH:

```
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
```

IgnoreRhosts

The IgnoreRhosts directive allows an administrator to enhance security by ignoring the legacy .rhost file from users. This is a best practice, in case rsh/rlogin are enabled or could accidentally become enabled.

System wide, /etc/hosts.equiv and/or /etc/ssh/shosts.equiv are still permitted for use. Once again, it is best practice to shy away from host-based authentication. Secure host-based authentication is covered in Chapter 6.

This directive takes a yes/no argument and by default is set to yes:

```
IgnoreRhosts yes
```

IgnoreUserKnownHosts

The IgnoreUserKnownHosts directive is designed to protect against users setting up host-based authentication with hosts other than those set up by the administrator. For host-based authentication to work, a cache of the remote host's public key must exist. If the user's cache is ignored, the system administrator must set up the cache. The default is no. For security purposes, it is often best to change this argument to yes.

```
IgnoreUserKnownHosts yes
```

KerberosAuthentication

If the KerberosAuthentication directive is set to yes, sshd PasswordAuthentication will take place though the Kerberos Key Distribution Center (KDC). For this to be enabled, Kerberos must be in place. The default is no.

KerberosGetAFSToken

When using Kerberos with the Andrew File System (AFS), OpenSSH can access the AFS home directory before authentication completes. The KereberosGetAFSToken will enable public key authentication even with the public key stored on a remotely mounted directory. By default, this is set no.

KerberosOrLocalPasswd

If Kerberos authentication fails, then other authentication mechanisms will be allowed, such as password authentication, public key authentication, or host-based authentication if the KereberosOrLocalPasswd directive is set to yes. The default is yes.

KerberosTicketCleanup

Much like the GSSAPICleanupCredentials directive, the KerberosTicketCleanup directive specifies whether or not the authentication ticket stored in cache should be deleted upon session termination. This defaults to yes.

KeyRegenerationInterval

In SSH protocol 1, the host key changes at a certain intervals, specified in seconds. By default, 3600 seconds is the interval for the KeyRegenerationInterval directive. The key is not stored on disk; it resides in memory, so it is less likely to be compromised. If the value is 0, the key is not regenerated.

ListenAddress

sshd can be configured to listen to all addresses of a system, or tied to specific addresses and ports using the ListenAddress directive, which requires the argument of an IP address. This directive supports IPv4 and IPv6. The default is for sshd to listen on all addresses. Multiple ListenAddress lines are permitted. For example, these configurations could be used on my workstation (192.168.1.101):

```
# Does not bind to 127.0.0.1
ListenAddress 192.168.1.101
#Only port 22
 ListenAddress 192.168.1.101:22
#Requires DNS or /etc/hosts
ListenAddress rack:22
# All addresses at port specified by the Ports Directive
ListenAddress *
```

Machine names and IP addresses can be used interchangeably. By default, all local addresses listen on the sshd port.

LoginGraceTime

sshd can be configured to drop connection attempts if a successful login has not occurred with the LoginGraceTime parameters. By default, the grace time is 120 seconds, as in the following example. If set to zero, authentication attempts have an unlimited amount of time.

```
LoginGraceTime 120
```

LogLevel

The sshd process is capable of logging at different verbosity levels. Listed from the lowest verbosity to most, the possible arguments for LogLevel are QUIET, FATAL, ERROR, INFO, VERBOSE, DEBUG, DEBUG1, DEBUG2, and DEBUG3. Normally running at the VERBOSE level, as shown in the following syntax line, will provide good amounts of information for the security conscious administrator. DEBUG and DEBUG1 levels are equivalent.

```
LogLevel Verbose
```

MACs

MACs are a type of hash used to verify data including the utilization of a secret key. MACs, or Message Authentication Code algorithms, are verified and computed using the same secret key to prevent nonintended recipients from verifying the message integrity. MACs can be

specified inside the sshd_config file. The MACs directive applies only to SSH protocol 2. Different options are available. By default, the MACs directive is set to hmac-md5,hmac-sha1,hmac-ripemd160, hmac-sha1-96,hmac-md5-96.

MaxAuthTries

The MaxAuthTries token is designed to combat a brute-force password attack, or the user who cannot remember his or her password. The number MaxAuthTries takes as an argument will tell sshd how many attempts a connection is allowed to authenticate; after failing MaxAuthTries times, the failed attempts are logged. By default, a user gets six attempts. Oftentimes six attempts is higher than local security policy recommends, so you may want to set this directive as follows:

```
MaxAuthTries 5
```

MaxStartups

The MaxStartups directive specifies how many unauthenticated sessions (users attempting to log in via passwords, keys, etc.) sshd will allow at once. By default, it is set to 10, as shown in the syntax line. Other attempts to authenticate will not be honored until one of the previous sessions authenticates, closes, or times out.

```
MaxStartups 10
```

PasswordAuthentication

By default, OpenSSH allows password authentication to occur. Sometimes, especially for very secure systems, PasswordAuthentication is set too, because normally public key authentication is thought to be much more secure than traditional password-based authentication. If setting this to directive to no, ensure some other form of authentication such as public key authentication is enabled.

```
PasswordAuthentication yes
```

PermitEmptyPasswords

By default, the OpenSSH server does not allow empty passwords. If the PermitEmptyPasswords directive is set to yes, the operating system must also be configured to allow for empty passwords. If you are using empty passwords, you are foregoing some basic security principles, such as integrity and perhaps confidentially, because you might not know who is working with data or what they have done to it. Usage of empty passwords is discouraged:

```
PermitEmptyPasswords no
```

PermitRootLogin

The PermitRootLogin directive offers special behavior for the root account outside of AllowUsers and DenyUsers. Root's login can be controlled using four possible arguments:

1. yes: This allows root to log in via any form of allowable authentication, including passwords. In most operating systems, the root account does not lock after a number of unsuccessful authentication attempts. This can leave your system open to brute-force password attacks against the root account.
2. no: Many administrators feel that no is the best argument for PermitRootLogin. Allowing root to log in using most services is generally a bad idea. For auditing purposes, normally a user should log in using a standard account and then run su or sudo to invoke root-level access.
3. without-password: The without-password argument allows root to log in, but using methods that do not use passwords. Public key authentication, for example, is still permitted. This allows for scripting and access to the root account. Keys of appropriate length are extremely difficult to crack, thus still securing the root account. Allowing root to log in using keys via SSH still does not mean direct root access should be available in the form of other services.
4. Forced-commands-only: This implies public key authentication with the command= option in root's public key. Keys are covered in more detail in Chapter 6. Using this directive can restrict the root account to only running specific commands such as scripts or reporting mechanisms.

PermitUserEnvironment

PermitUserEnvironment tells sshd whether or not it should allow a user to specify a user environment within $HOME/.ssh/environment or through the environment= option inside of an authorized_keys file.

Passing environment variables can sometimes be a security risk or cause problems, if usernames differ between systems or environments/shells differ greatly between systems. Additionally, sometimes environment variables contain sensitive information such as which sockets to connect to for ssh-agent authentication. Therefore, although the PermitUserEnvironment directive takes arguments of yes/no, this directive defaults to no, as you see in this syntax line:

```
PermitUserEnvironment no
```

PidFile

The PidFile directive takes a path to a file that will contain the process identifier (PID) of the sshd daemon. By default, on most systems, this is /var/run/sshd.pid. The PID file can be used in scripts to see if sshd is running or to kill sshd by issuing kill `cat /var/run/sshd.pid`.

```
PidFile /var/run/sshd.pid
```

Port

By default, sshd listens on TCP port 22. However, it is possible to configure the server to listen on multiple ports by using the Port directive. This can be useful if only certain points are permitted through a firewall, or if different sshd server configurations are running on different ports (perhaps one for the administrators and one for the end users).

For example, if sshd listening on port 22 and 99 is desired, the following options may be set:

```
Port 22
Port 99
```

Be careful when using multiple port statements in conjunction with the ListenAddress directive. If you specify ports inside of ListenAddress, they can conflict and take precedence over the port statement. Using a -p on the command line when starting sshd will also allow for port specifications for that instance of the daemon.

PrintLastLog

Setting PrintLastLog to yes will cause the user's last login time to be displayed on the screen at the time of login. This is enabled by default, as in the syntax line that follows, because it can empower the user to check for security. A user might be alerted if he or she sees that the last time he or she supposedly logged in was 3 a.m. last Saturday (if this is not the user's normal behavior).

```
PrintLastLog yes
```

PrintMotd

If the PrintMotd directive is set to yes as shown in the syntax line, sshd will print /etc/motd (Message of the Day) when an interactive login occurs. The default is to print /etc/motd. Sometimes the contents of /etc/motd are displayed via other configuration files, so displaying it twice could be redundant.

```
PrintMotd yes
```

Protocol

The Protocol directive allows you to specify which version of the SSH protocol to use. While by default protocol 1 and 2 are supported, using only protocol 2, as shown in the syntax example, is strongly recommended. Protocol 1 has several known exploits that are believed to be fixed now, but most material and documentation recommends against its usage. It also has some flaws that are inherent to the protocol, and not fixable.

Specifying 2,1 is the same as 1,2 for this directive. Protocol 2 is the primary focus of this book and most modern OpenSSH documentation. Protocol 1 exists for backward compatibility and is still sometimes seen for communication with embedded systems and network appliances.

```
Protocol 2
```

PubkeyAuthentication

The PubkeyAuthentication directive, which takes a yes/no value, tells the OpenSSH whether or not public key authentication should be used. The default is yes, as shown in the following example, and should be left as such. Properly managed keys are more secure and convenient than passwords. This option is only applicable to protocol 2.

```
PubkeyAuthentication yes
```

RhostsRSAAuthentication

The RhostsRSAAuthentication directive, which applies to protocol 1 only, specifies whether .rhosts or /etc/hosts.equiv (associated with the Berkeley r-utilities) authentication is allowed. This setting defaults to no, as shown in the syntax line example, and normally should be left at this setting. Setting RhostsRSAAuthentication to yes could open the systems to potentially weak authentication methods if rlogin/rsh becomes enabled, because they use the same files for authentication.

If you are using host-based authentication, using protocol 2 is a best practice. Host-based authentication utilizing protocol 2 is discussed in Chapter 6.

```
RhostsRSAAuthentication no
```

RSAAuthentication

If the RSAAuthentication directive is enabled (set to yes), RSA authentication is allowed via protocol 1. RSAAuthentication is set to yes by default, but if protocol 1 is not enabled, this option will have no bearing on the OpenSSH configuration.

ServerKeyBits

ServerKeyBits defines the key length for the protocol 1 host key. By default, the bit length is 768, with a 512-bit minimum. If protocol 1 is disabled, this setting can be left alone. If protocol 1 is enabled, using 1024-bit or stronger keys is encouraged.

StrictModes

The StrictModes directive determines whether sshd should check permissions on the user's files and home directory before allowing a public key authenticated login. This check is designed to nullify authorized_keys files that are left open for group/world members to insert their public keys into. Leaving StrictModes in place is a best practice, as without it, users can assume another user's identity and bypass several security mechanisms. StrictModes by default is yes:

```
StrictModes yes
```

Subsystem

By default, no subsystems, applications relying on the SSH protocol for transport, are utilized from the source version OpenSSH. To implement sftp (Secure File Transfer Protocol) inside of sshd, you must specify the subsystem name and subsystem path. Mine is in /usr/local/libexec, but this will be wherever your libexec directory is configured on your system. The Subsystem directive applies to protocol 2 only. Most binary distributions of OpenSSH have the sftp-server enabled:

```
Subsystem       sftp    /usr/libexec/sftp-server
```

SyslogFacility

Logging is important for connectivity daemons to see who is using services, monitoring machine utilization, change control, and troubleshooting. The `SyslogFacility` directive takes syslog's facilities as arguments. By default, `sshd` logs to the `AUTH` facility, as shown in the syntax example. In some Linux systems, this is set to `AUTHPRIV`, which puts the authentication information in a log only readable to the root user. Argument possibilities are `DAEMON`, `USER`, `AUTH`, and `LOCAL1–LOCAL7`.

```
SyslogFacility AUTH
```

■ **Tip** When changing the facility, be sure to check on what else is using that facility, as having `sshd` information in the `boot.log` file does not often make sense.

TCPKeepAlive

If `TCPKeepAlive` is enabled (the default), `sshd` will send keep-alive requests to the clients. If they are being sent, a network glitch will cause the session to terminate. If they are not being sent, the session might terminate on the client side, but the server will still have resources allocated to the client process. That scenario can cause unnecessary resource constraints on the system running `sshd`. When set to `yes`, as in the following example, the server will properly terminate an abruptly ended session from the client side, such as a client-side crash or network disconnection:

```
TCPKeepAlive yes
```

UseDNS

If the `UseDNS` directive is enabled (the default), `sshd` will attempt to resolve the host name for the remote IP address, and then ensure that hostname maps back to the IP address that is attempting the connection. This is a security feature to prevent rogue clients from connecting on a network, because oftentimes DHCP address spaces do not have reverse records in DNS. This type of setting restricts the client usage of `sshd` to those with a reverse record. Of course, this setting assumes DNS is working accurately and reverse records are found for all relevant IP addresses.

```
UseDNS yes
```

UseLogin

`UseLogin` is another `yes/no` directive that defaults to `no`, as shown in the syntax example. `/bin/login` or `/usr/bin/login` is the normal way in which most legacy connectivity services allow users to access system. `sshd` provides its own mechanism, which is more sophisticated and can allow X11 forwarding over the SSH protocol. The default value of `no` should be used in

most cases. If other legacy applications are being used, perhaps through graphical user interfaces, sometimes `UseLogin` is required. If SSH is being used to execute commands remotely, even with `UseLogin` set to `yes`, login is not used.

```
UseLogin no
```

UsePAM

The `UsePAM` directive, which takes `yes/no` values, determines whether PAM (Pluggable Authentication Module) authentication should be used. If your system requires PAM, OpenSSH should be configured with the `--use-pam` option and the `UsePAM` directive set to `yes`, as shown in the syntax line. I normally build with PAM support and enable it in my configuration files for Linux systems. Other systems that do not use PAM should set the `UsePAM` directive to `no`. Binary packages of OpenSSH from UNIX/Linux vendors normally have PAM configured to match the requirements of the specific system.

```
UsePAM yes
```

UsePrivilegeSeparation

Privilege separation is a nice feature that lets each child process of `sshd` run as the user invoking it rather than root. This practice should protect against attacks trying to gain root access via the OpenSSH server because the attempts can be made only against a standard user process. The default to the `UsePrivilegeSeparation` directive is `yes`, and should be enabled:

```
UsePrivilegeSeparation yes
```

X11DisplayOffset

When forwarding X11 connections, `sshd` sets the display variable to something like `localhost:10.0`. If you run more than ten X11 servers on the machine running `sshd`, increment the `X11DisplayOffset` variable as needed. More information about X11 forwarding is in Chapter 7.

```
X11DisplayOffset 10
```

X11Forwarding

X11 forwarding is one of the great benefits of OpenSSH if X-Windows applications must be used. When set to `yes`, as in the syntax example, `X11Forwarding` will establish a tunnel for X11 traffic over an SSH connection. This will encrypt X11 traffic and automatically set your `$DISPLAY` variable. By default, `X11Forwarding` is not enabled. If your organization does not rely on X11 technology, it is best to leave it that way. If `UseLogin` is enabled, this directive will automatically be disabled. X11 forwarding is discussed in detail in Chapter 7.

```
X11Forwarding yes
```

■ **Caution** As with other environment variables, a user can reset his or her `$DISPLAY` variable. If set to a valid address such as `rack:0.1`, then encryption has been defeated. Many users automatically set their `$DISPLAY` variable in their `.profile`. Additionally, a user could set their `$DISPLAY` to another user's tunnel and `.Xauthority` file, and display X11 applications on an X server that is not their workstation.

X11UseLocalhost

Setting the `X11UseLocalhost` directive to `yes` will allow remote connections to use the forwarded X11 connection because it binds to the wildcard address. Some older X11 clients require a name other than localhost. This should normally be left at the default (enabled).

```
X11UseLocalhost yes
```

XAuthLocation

The `XAuthLocation` directive takes a path as an argument to the `xauth` program. Normally, `xauth` is found at `/usr/X11R6/bin/xauth`, as shown in the following example, which is the default on many Linux systems. The `xauth` program is used by OpenSSH to authorize the X11 clients. Running `configure` will normally find the `xauth` program before compiling OpenSSH.

```
XAuthLocation /usr/X11R6/bin/xauth
```

Building the sshd_config File

As mentioned before, the provided `sshd_config` file, whether it is from the OpenSSH source or from your UNIX/Linux vendor, is normally a good place to start. From there, analyze your needs, and decide what directive arguments make the most sense for your environment.

Keep in mind this file can be changed, and `sshd` can be reread without dropping connections. My standard OpenSSH configuration file looks like this; however, it often changes to suit specific needs.

```
# sshd_config file
# VERSION 1.0
# Network Information
Port 22
#Disables protocol 1
Protocol 2

# Protocol 2 HostKeys
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key

# Logging
# On Linux
SyslogFacility AUTHPRIV
```

```
# Other UNIX
#  SyslogFacility AUTH
#shows key fingerprints
LogLevel VERBOSE
# Users/groups
AllowUsers *
#Deny Users badguy
AllowGroups *
#DenyGroups badgroup

# Authentication:
LoginGraceTime 120
MaxAuthTries 5
PermitEmptyPasswords no
PasswordAuthentication yes
#public keys allowed for root login
PermitRootLogin without-password
# Users must have good permissions on .ssh and authorized_keys
StrictModes yes
# protocol 1
RSAAuthentication no
#recommended authentication method
PubkeyAuthentication yes
AuthorizedKeysFile      .ssh/authorized_keys
ChallengeResponseAuthentication no
# Do not have Kerberos environment
KerberosAuthentication no
#GSSAPI not needed
GSSAPIAuthentication no
#Configured --with-pam
UsePAM yes
UseLogin no

# Host-based Authentication
#protocol 1
RhostsRSAAuthentication no
#protocol 2
HostbasedAuthentication no
#must be set by system Admin
IgnoreUserKnownHosts yes
# Squash Rhosts

IgnoreRhosts yes

# FORWARDING
# I like to know what is being forwarded
AllowTcpForwarding no
```

```
GatewayPorts no
X11Forwarding yes
X11DisplayOffset 10
X11UseLocalhost yes
# Environment
PrintMotd yes
PrintLastLog yes
Banner /etc/issue
AcceptEnv PATH

# System Settings
UsePrivilegeSeparation yes
PermitUserEnvironment no
Compression yes
PidFile /var/run/sshd.pid
#set to no if no reverse records
UseDNS yes
Subsystem      sftp    /usr/libexec/openssh/sftp-server
MACs hmac-md5,hmac-sha1,hmac-ripemd160,hmac-sha1-96,hmac-md5-96
# Leave ciphers at default values, will not mention

# Client Connection
TCPKeepAlive yes
ClientAliveInterval 5
ClientAliveCountMax 3
MaxStartups 10
```

The configuration file I normally use tries to provide security and usability at a decent trade-off. For example, I allow X11 forwarding because I know users like to use X11 applications for backups/restores and modeling applications; however, I do not allow TCP forwarding because I would like to know about the tunnels being requested, and I can set those up for a user.

To make this file readable for coworkers and auditors, I try to break the directives into sections. Most directives are explicitly specified. I leave Kerberos and GSSAPI directives alone after specifying that I am not using them. `Ciphers` I also leave alone, because some clients prefer different ciphers over others.

Different scenarios could cause this file to be heavily modified. For example, if host-based authentication is allowed, several directives in this file must change. Allowing protocol 1 connections also would cause me to include more protocol 1–specific directives. Normally, I do not include protocol 1 directives because they create confusion. The client configuration, however, does need to have protocol directives in it if the possibility of connecting to a protocol 1 client exists.

■ **Note** The server configuration, in this state, will allow for nearly every client to connect to the `sshd` processes without issue. For OpenSSH to accept clients from Microsoft Windows GUIs such as PuTTY or WinSCP, no additional configuration is necessary. If a client is specifically for protocol 1 only, however, the connection will not work.

Ensure Security of sshd_config

After creating, testing, and editing the `sshd_config` file, remember once again to check the ownership and permissions of the file. It is also a good idea to have a digest (SHA-1 or MD5) of the file, so you can quickly tell whether the file has been tampered with in any way. The following examples must be done as root:

```
rack: / # ls -l /etc/ssh/sshd_config
-rw-------  1 root root 1817 Mar 01 14:52 sshd_config
rack: / # md5sum /etc/ssh/sshd_config
b8f83d7c556fd5b37898f350e1c65ebc  /etc/ssh/sshd_config
```

After recording the MD5 sum, its best to move that MD5 sum digest to a safe remote location. Additionally, a pristine `sshd_config` file should be kept somewhere not in production for verification purposes. I normally run processes that check against that MD5 sum and ensure the file is correct. If it is not, I follow up with administration staff to see why it is not. Normally, a change was made for application migration or debugging. I then overwrite the configuration file with my good one.

Managing the OpenSSH server can seem difficult at first, but in time it will become quite simplified. With the OpenSSH server, you can reload configuration files without dropping connections, so simple changes can be made without impacting end users. The ability to run in debug mode and validate syntax of configuration changes is also a feature of `sshd` that administrators should be sure to take advantage of. Testing changes in directives is the most daunting task, but hopefully going forward you will have a good foundation of what those entail.

Summary

Building the server configuration file takes time and testing to best fit your environment. After creating a server configuration that is workable, it is time to build the client configuration. Chapter 5 covers the command-line `ssh` client in the same level of detail presented here about the server. The client settings need to be carefully examined to ensure that users are not circumventing your security design. In the later chapters that discuss forwarding and key-based authentication, any deviation from the server configuration file presented in this chapter is shown. Chapter 6 will get you started with key-based authentication that hopefully will enable you to have new authentication practices to securely and efficiently manage your systems and OpenSSH services. Chapter 7 covers forwarding and tunneling, including X11 forwarding.

`sshd` from OpenSSH provides a practical and secure alternative to the legacy protocol connectivity daemons such as `rshd` and `telnetd`, and should ultimately lead to their elimination on your network. This transition will not occur in a single night, but with proper planning and utilizing the best possible directives in your configuration files, you can make the implementation a success.

Hopefully, your OpenSSH server configuration can be added with no compromises for backward compatibility and applications that still rely on `.rhost` files. If you are running into issues with older batch jobs, scripts, and applications, check out Chapter 9, which covers scripting with `ssh`.