SUN BLUEPRINTS [illegible] APPROVED

# Configuring OpenSSH for the Solaris™ Operating Environment

*By Jason Reid - Solaris System Test (SST)*

*Sun BluePrints™ OnLine - January 2002*

**http://www.sun.com/blueprints**

**Sun Microsystems, Inc.**
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 816-3250-10
Revision 1.0, 12/17/2001
Edition: January 2002

# Configuring OpenSSH for the Solaris™ Operating Environment

Networks have never been secure. As the demand on open networks for remote access has grown, the risks of compromised systems and accounts has kept pace. Tools for securing networks, such as OpenSSH, were developed to counter the threats of password theft, session hijacking, and other network attacks. However, these tools come with the price of planning, configuration, and integration. This article provides recommendations for configuring and managing OpenSSH.

Specifically, this article deals with client and server configuration, key handling, and the integration of OpenSSH into existing environments that run the Solaris™ Operating Environment (Solaris OE.) For details about the compilation of OpenSSH's components, consult the Sun BluePrints™ OnLine article "Building and Deploying OpenSSH for the Solaris Operating Environment" (July, 2001).

This article does not discuss general OpenSSH usage. Consult the OpenSSH man pages and *SSH, The Secure Shell* for that information. For technical details about the underlying protocols, refer to the Internet Drafts of the Secure Shell (SECSH) working group.

This article was drafted using OpenSSH 2.9p2.

## Security Policy

The primary purpose of security policy is to inform those responsible for protecting assets such as hardware, software, and data of their obligations. Management establishes a security policy based on the risks it is willing to tolerate. The policy itself does not set goals, but serves as a bridge between management's goals and the technical implementation.

OpenSSH was designed to be a secure replacement for unsafe network commands such as `rlogin`, `rsh`, `rcp`, `telnet`, and `ftp`. The way you configure OpenSSH should reflect a site's local security policy. For example, you might consider whether password authentication is appropriate, or whether a more rigorous two-factor (public-key based) authentication is required. Further, you might consider whether the policy allows OpenSSH to tunnel TCP and X windows connections and whether it allows for remote access to internal web sites. Again, OpenSSH configuration should match local policy.

If a site does not have a security policy, one should be crafted before configuring OpenSSH. For guidance on crafting a security policy, consult the references in the Bibliography.

## Configuration

OpenSSH has many capabilities not all of which are appropiate depending on your local policy. Configure OpenSSH to conform to your policy. OpenSSH is configured in three places: compilation (compile time), server configuration, and client configuration. Compile time configuration covers basic details such as which entropy source to use, the location of configuration files, and whether binaries are `SUID`. Compile time configuration has the advantage that it can not be overriden. Server configuration concerns how and to whom the OpenSSH server should present itself on the network. Server configuration details include which protocols and authentication methods are offered, which users have been granted access, and how much logging of each connection should be done. Server configuration can not be overridden by the client. Client configuration covers which server to communicate with, server verification, and user ease of use.

Configuration in order of precedence is: software compile time, the server configuration file (`sshd_config`), client command line options, individual client configuration file (`~/.ssh/config`), and the global client configuration file (`ssh_config`). The location of `sshd_config` and `ssh_config` vary depending upon compile time options. They are usually located in `/etc`, `/etc/ssh`, or `/usr/local/etc`.

A defensive in-depth strategy of setting the preferred configuration redundantly at compile time, server configuration, and client configuration is recommended. This reduces the chances that a single accidental misconfiguration will weaken the integrity of the system.

Example client and server configuration files that document the recommended configuration are presented later in this document. Not all of the options presented in the files are described in this document.

# Recommendations

OpenSSH offers a number of features to protect network connections between two hosts. There are choices of protocol, authentication method, port forwarding, user access, and network access. When setting up OpenSSH, you will have to make trade offs between security, ease of use, and legacy compatibility. The choices you make depend on local security policy.

## Protocol Support

There are two major versions of the secure shell protocol: SSH1 and SSH2. SSH1 was the first protocol developed and has been replaced with SSH2. It is highly recommended that you disable the use of SSH1 since the protocol has been found to have several vulnerabilities including packet insertion attacks and password length determination. In `sshd_config` and `ssh_config`, set `Protocol` to `2`, as follows.

```
# Protocol 2 only is recommended.
Protocol 2
```

For legacy client and server support, allow SSH1 but set the default to SSH2, as follows.

```
# Enable legacy support but default is Protocol 2.
Protocol 2,1
```

Unfortunately, many legacy clients and servers only support SSH1. Consider upgrading legacy clients and servers to those that support SSH2. If you wish to audit installed base of ssh servers, consider using `scanssh` by Niels Provos. It is designed to scan a network and report the version strings of any ssh servers found. (You can also use `ssh-keyscan` and shell scripting to accomplish the same thing in a much less efficient manner.)

## Network Access

By default, the OpenSSH server daemon listens to all network interfaces. For workstations and other systems where accessibility is desired on all interfaces, this is not a problem. For architectures where a single interface is dedicated to management or administration, it is preferable not to expose OpenSSH to the other networks. Limit network access with `ListenAddress` in `sshd_config` as shown here.

```
# Listen only on management network
ListenAddress 192.168.0.10
```

To further narrow down what the server will listen to (for example, a specific address range or single host), use either a host-based firewall or a tool like tcpwrappers.

---

**Note –** OpenSSH does support the use of TCP wrappers but support needs to be compiled into the server. Consult the build documentation of OpenSSH for information.

---

## Connection Forwarding

OpenSSH can create a secure tunnel to provide some protection for insecure protocols. This is referred to as connection forwarding and only works for TCP-based connections. During connection forwarding, a local TCP port is opened and OpenSSH waits for a connection. When OpenSSH receives a connection, it forwards the data to the OpenSSH server on the other end. The server then sends the data to its final destination. Responses follow the same process, in reverse.

---

**Note –** Data is protected only until it reaches the OpenSSH server. After that, it is handled the same as normal network traffic.

---

Connection forwarding is useful for protecting commonly used, noncryptographic protocols like IMAP, which is used for email. It can also be used to provide remote users with access to internal resources such as news, email, and web access. If policy is such that remote users are to be granted access to these resources, enable connection forwarding.

There are two caveats with connection forwarding. Firstly, connection forwarding is an all or nothing mechanism. Once forwarding is allowed, the client can forward any port to any location on the remote side. If this is an issue, consider using host-based firewalls on the OpenSSH server to limit connections. Secondly, because traffic that travels through connection forwarding is encrypted, neither a firewall, nor an intrusion detection system can detect when abnormal events occur. The OpenSSH server on the remote side is traffic agnostic. It does not know if data coming out is a normal IMAP request for a message or if it is buffer overflow exploit against the IMAP server. Plan firewall and intrusion detection sensors accordingly. Add the following to `sshd_config` to allow TCP forwarding.

```
# Server configuration
AllowTCPForwarding yes
```

An example of a client forwarding in `ssh_config`.

```
# client configuration
# Allow remote users access to an internal web server.
LocalForward 8080 www.corp.acme.com:80
```

## Gateway Ports

Gateway ports work in conjunction with connection forwarding. Normally, connection forwarding allows only the local host to send data to the other side of a connection. By using a gateway port, you enable other machines to connect and forward data. In effect, gateway ports create a tunnel from one network to another network. This is highly risky and in general should always be disabled. For example, an user sitting in an airport connected over a 802.11b wireless link with gateway ports turned on and a local forward to an internal web server would allow everyone in the immediate vicinity access to the web server. Set the following in both `sshd_config` and `ssh_config`.

```
# Server and client configuration
GatewayPorts no
```

## X Forwarding

OpenSSH can also securely tunnel X traffic. Because the X protocol travels in the clear, it is vulnerable to sniffing and hijacking. OpenSSH emulates an X server on the remote side and passes traffic back through the tunnel to the local client. In addition to its usefulness for remote users, this can also help decrease the potential for users to use `xhost +` to disable all access controls. Add the following lines to both `sshd_config` and `ssh_config`.

```
# Server and client configuration
X11Forwarding yes
```

The following is an example of the values of $DISPLAY on a local host and over an X forwarded tunnel.

```
host $ echo $DISPLAY
:0.0
host $ ssh remotehost
user@remotehost's password:XXXXXXXX
remotehost $ echo $DISPLAY
remotehost:11.0
```

## User Access

Some sites require that a banner be displayed once users connect to a system, but before they log in. If this is required, set the banner to `/etc/issue` in `sshd_config`, as shown in the following example, so only one banner exists for the whole system.

```
Banner /etc/issue
```

The default login grace time is ten minutes. This value is too high. Consider reducing it in the `sshd_config` to thirty or sixty seconds as shown here.

```
LoginGraceTime 60
```

User access control lists can be specified in OpenSSH; however, no part of the Solaris OE honors this access control list (ACL). The two available options are to allow only specified users access, or to specifically deny a user access. The default is to allow anyone access. You can also specify access with group membership. Note that the groups options only apply to the primary group (the group listed in `/etc/passwd`). An example of both allow and deny ACLs in `sshd_config` appears as follows.

```
# Allow sysadmin staff
AllowGroups staff
```

```
# Or limit a particular user’s access off a machine
DenyUsers kaw alex
```

By default, the root user can log in using OpenSSH. This is fine for systems without user accounts. However, disabling root logins and requiring administrators to use `su` to root is more secure and leaves an audit trail. If you have remote jobs that run as root, you can configure OpenSSH to only execute scripts. This requires the use of two-factor (key-based) authentication. If root logins are required at your site, only use key-based authentication as discussed later in this article. To set this up, add the following to `sshd_config`.

```
# Only add one of these settings.
# Forces sysadmins to su.
PermitRootLogin no
# If remote jobs require root priviledges.
PermitRootLogin forced-commands-only
```

## Authentication

OpenSSH supports multiple forms of authentication: the traditional login and password, two- factor (public-key-based), and host-based. Each method has different benefits. Password authentication fits well in existing structures. Two-factor authentication offers improved security, although with higher maintenance costs. Host-based authentication provides the most convenience, although it is extremely unsafe and easily abused.

Password authentication is the most common way for systems to authenticate users. The drawback to this method is that passwords can be shoulder-surfed, guessed with dictionaries, and sniffed in transit across the network. While OpenSSH protects passwords by encrypting them, this only prevents sniffing while they are in transit, and can’t do anything to minimize the effects of other threats. To counter other threats, OpenSSH provides two-factor or key- based authentication.

Key-based authentication is a challenge and response system which is grounded in the mathematics of public-key cryptography. There are essentially two elements: a public key that resides on all servers the user will access, and a private key that only the user knows. The private key is additionally protected by a passphrase. This system is more secure than passwords alone because in addition to being based on a passphrase the user knows, it is also based on something the user has in their possession, the private key.

The system works roughly as follows. OpenSSH generates a key pair, stores the public key on the OpenSSH server, and leaves an encrypted version of the private key on the user's machine with a passphrase. When the user connects to the server, OpenSSH prompts the user for a passphrase to decrypt the private key. The OpenSSH client and server then go through a challenge and response to prove that the two keys are related. If the server agrees that user really does have the private key, it grants the user access. The private key is never stored on the server or transmitted to it, and the public key is useless without the private key, and vice versa. For a system to be subverted (leaving out program flaws like bugs), someone would have to acquire a copy of the private key and the passphrase.

Because private keys are often stored on NFS home directories, good passphrases are critical to the success of this approach. Examples of bad phrases might include simple sentences with no punctuation and no capitalization, or extremely common phrases like "to be or not to be." Examples of good phrases include phrases or words the user can easily remember and won't write down. If a user looses a passphrase, you will need to generate a new key pair, as a passphrase cannot be recovered. Further, passphrases tend to be resistant to shoulder surfing due to their length.

Host-based authentication trusts a connection based on where it comes from. This is very unsafe and easily abused. `Rlogin` and `rsh` also use this method of authentication as denoted by their dependence on `.rhosts` files.

It is recommended that sites disable any semblance of host-based authentication. Sites that support a large number of internal users should consider staying with passwords to reduce training costs. Sites with remote users and sites that need to automate jobs should consider using key-based authentication. Add the following to `sshd_config` for the preceding recommendations.

```
# Disable unsafe hosts based authentication
HostbasedAuthentication no
RhostsAuthentication no
IgnoreRhosts yes
# Empty passwords are trivial to guess
PermitEmptyPasswords no
# For internal servers, passwords ok. Bastion hosts - no.
PasswordAuthentication yes
# For remote access, automated jobs, and advanced users
PubkeyAuthentication yes
```

## Key Handling

Public-key cryptography is used in two places: server identification and two-factor authentication. This means that there are keys to be managed, protected, transported, and eventually destroyed. Key handling is the largest obstacle to the wide-scale deployment of OpenSSH. Because OpenSSH was designed as a point-to-point solution with no public-key infrastructure in place, all key operations must be done manually. This is not a problem for small deployments; however, the problem does not scale.

Because they are the foundation for systems security, keys must be handled with care. If private keys are divulged, security is compromised because the system appears to be secure when, in fact, it is not.

### Host Keys

Server identification is accomplished by a host key pair. The `openssh.server init` script, which you can find on the Sun BluePrints website, generates a key set if it cannot find a host key. This key set is used to identify the server to the client. The private key remains private to ensure the integrity of the system. The client downloads the public key and compares it to its copy in known_hosts. If the key is different than it is expected to be, a warning message is printed and the connection is refused. The following is an example of this warning.

```
$ /opt/OBSDssh/bin/ssh some_host
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-
middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
c3:6f:30:ff:84:e1:e0:d6:ef:28:e7:76:f2:49:ea:be.
Please contact your system administrator.
Add correct host key in /export/home/user/.ssh/known_hosts to
get rid
of this message.
Offending key in /export/home/user/.ssh/known_hosts:2
RSA host key for some_host has changed and you have requested
strict checking.
$
```

The problem is how to get the public host key to the client in the first place. Another problem is what to do when the public host key has been regenerated due to loss, server upgrade, or compromise. Having multiple users call support because of the preceding warning message could create quite a support headache. Further, having users change keys manually would be even less desirable.

The client configuration option StrictHostKeyChecking controls how the client reacts to new hosts keys. If you set this option to yes, OpenSSH will not make connections to unknown servers. If you set the option to ask, OpenSSH will prompt users to accept a new host key if the server is unknown. If you set the option to no, OpenSSH will add new host keys without prompting users. The no option setting will allow connections to servers with modified host keys.

The easiest solution is to simply disable StrictHostKeyChecking by setting it to no. Blindly accepting new keys allows man-in-the-middle attacks and is not recommended. If your users can be trusted to act responsibly, then set the option to ask. Users can manually verify the host key by comparing the value in known_hosts to the value ssh_host_key.pub, ssh_host_dsa_key.pub, or ssh_host_rsa_key.pub depending on the protocol and public cryptographic system used to connect. If the values don't match, something odd has happened. This could be caused by an active attack or possibly just a server reinstallation. Respond according to your local policy.

Another solution is distribute a `known_hosts` file to users; however, it is difficult to do this in a secure fashion. You must decide how to securely collect public host keys and how to securely distribute the file to the users. Again, there are problems of scalability with any solution. Fortunately, changes to host keys should be infrequent.

The most secure method of gathering keys is to log in to every server and manually copy the public host key to a portable medium such as a floppy disk, CD-RW, or smartcard. For sites with a large number of machines, or during the first deployment of OpenSSH, this burden is significant. Alternatively, you can configure a client with `StrictHostKeyChecking` set to `no`, access every single host, copy the public host key, exit, and then compare the key with the value in `known_hosts`. Display a warning message for any server with a differing key. This can be automated using Korn shell, PERL, or some other scripting language.

`Ssh-keyscan` can also be used to generate a list of host keys. The risk is getting the host key of a compromised machine. None of the solutions are perfect. There are some serious tradeoffs between convience and security. At the minimum, set `StrictHostKeyChecking` to `ask` and train your users to check the host keys.

A novel use of `ssh-keyscan` is to regularly check for altered keys. At routine intervals, probe the servers and check if keys have been altered. This can provide warning of an intrusion or a non-logged installation.

With public host keys gathered, you must decide how to securely distribute the file to users. An easy solution is to integrate the file into the deployment packaging such as the OBSDssh package. The file can be placed on an ftp or http server. Also distribute a preferably signed hash (MD5 or SHA-1) of the file so the user can verify the integrity of the file. (OpenSSL has the capability of performing the hashes.)

For sites with a public-key infrastructure, a pretty good privacy installation, or a Gnu privacy-guard installation, distribute the file and its hash cryptographically signed.

With the hassle of users seeing an unfamiliar warning about a changed host key, there is the temptation to archive the public and private host key pairs onto a system. The key pairs would be replaced after a system was reinstalled or replaced. This is risky and not recommended. Any system storing the keys would be a tempting target and if it was comprised all keys within it would also be compromised. It is better to deal with the occasional host key change through user education and notices of reinstallation. If it is necessary to archive keys, store them offline, in encrypted format, and in secure storage such as a safe.

In the event of a server compromise, destroy host keys. An attacker with knowledge of the private portion of a host key could impersonate the host and perform a man-in-the-middle attack.

## User Identity Keys

Users may optionally authenticate themselves using cryptographic keys. Public-key authentication is more secure than password authentication for following reasons. First, the private-identity key is protected by a passphrase which may be much longer than the eight character password limit. Second, neither the passphrase nor the private key is ever transmitted to the server. There is no secret information to snoop off the network. Third, in order to compromise an account, the intruder must first gather the private key stored on the users machine and determine the passphrase in the user's head. Fourth and finally, computer generated cryptographic keys are infeasible to guess and not subject to dictionary attacks.

---

**Note –** Poor passphrases are susceptible to dictionary attacks, so good password/ passphrase discipline is still required.

---

For public-key authentication, the user creates an identity key pair with `ssh-keygen`. The resulting public key, either `id_dsa.pub` or `id_rsa.pub`, is then stored in `~/.ssh/authorized_keys2`. For hosts where users are unable to place their public keys, such as bastion hosts, public keys may be emailed to the IT support staff. Have the staff verify out of band the key fingerprint. Once public keys are placed into `~/.ssh/authorized_keys2`, users are no longer prompted for a password. Instead, they will be prompted for the passphrase for the private key.

The following is an example an user identity key generation.

```
/home/user/.ssh $ /opt/OBSDssh/bin/ssh-keygen -b 2048 -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_dsa):
Enter passphrase (empty for no
passphrase):XXXXXXXXXXXXXXXXXXXXXXXX
Enter same passphrase again: XXXXXXXXXXXXXXXXXXXXXXXX
Your identification has been saved in /home/user/.ssh/id_dsa.
Your public key has been saved in /home/user/.ssh/id_dsa.pub.
The key fingerprint is:
9b:9c:c4:fb:30:66:25:46:5b:b1:95:d9:a1:90:86:f9 user@host
/home/user/.ssh $ ls
id_dsa id_dsa.pub known_hosts2 random_seed
/home/user/.ssh $ cat id_dsa.pub > authorized_keys
/home/user/.ssh $ chmod 600 authorized_keys
/home/user/.ssh $ /opt/OBSDssh/bin/ssh remote_host
Enter passphrase for key '/home/user/.ssh/id_dsa':
XXXXXXXXXXXXXXXX
Last login: Sun Jul 15 13:37:45 2001 from host
Sun Microsystems Inc.   SunOS 5.8       Generic February 2000
remote_host /home/user $ ^D
Connection to remote_host closed.
/home/user/.ssh $
```

User-identity private keys still need some protection even when they are stored encrypted. It is preferable not to store them on NFS shares where they can be copied unnoticed. If this is not avoidable, stress the importance of good passphrases lest the keys are decrypted offline through a passphrase dictionary attack. In the event of portable computer theft, revoke all effected keys by removing them from the `authorized_keys` file and generate new keys. In case of a server compromise, check for the addition of backdoor user identity keys.

## Integration

Integrating OpenSSH into daily usage is not difficult. It can be used as a straightforward method for replacing `rlogin`, `rsh`, and `telnet` for interactive host logins that requires minimal user retraining. It can also provide added security to remote jobs and file transfers, it can tunnel through proxy servers to secure connections to outside the corporate intranet, and it can add single-sign on type convenience. You can also add your desired local configuration to the OBSDssh package for easy deployment.

## ssh-agent

Agents perform an action on the behalf of something else. Ssh-agent performs cryptographic operations on the behalf of an ssh process. Instead of an ssh process knowing the key to a remote host, the agent holds the key and does the work. Ssh-agent works by setting two environment variables: SSH_AUTH_SOCK and SSH_AGENT_PID. Because environment variables are inherited by children, setting up ssh-agent when first logging on to an environment like CDE means all terminal window shells will know about the agent and use it if possible. The following example shows a system with ssh-agent running.

```
host $ env | grep SSH
SH_AUTH_SOCK=/tmp/ssh-PNq12519/agent.12519
SSH_AGENT_PID=12520
```

A ssh-agent can do nothing until a key is loaded into it. Once a key is loaded, all of the ssh processes that are aware of that agent may use that key. This provides a form of single signon. The drawback is that if a shell is compromised, whatever access the loaded keys granted may be abused. Agent functionality is used to automate actions and make user's lives easier.

Ssh-add is used to add and list keys (referred to as identities by OpenSSH.) Ssh-agent can remove all keys held in memory and can only add a single key at a time. The following is an example of identity management.

```
host $ ssh-add
Need passphrase for /home/user/.ssh/identity
Enter passphrase for user@host <passphrase>
Identity added: /home/user/.ssh/identity (user@host)
host /opt/OBSDssh $ bin/ssh-add -l
2048 d1:0b:59:c3:ff:8a:20:ff:98:84:15:98:ff:63:e8:41 user@host
(RSA1)
host $ bin/ssh-add -D All identities removed.
host $ bin/ssh-add -l
The agent has no identities.
```

## Autonomous Actions

You can use OpenSSH to greatly improve the security of automated scripts and file transfers. However, note that any kind of unattended authentication still presents security risks. It is recommended that you use plain-text public-key authentication (keys are not protected by a passphrase.) The file permissions of the keys must be strict to ensure that others cannot read them. Even with this obvious flaw, this scheme is more secure than host-based authentication and embedding passwords into scripts.

This process requires more setup than the traditionally insecure method of `.rhosts` or `/etc/host.equiv`. You must generate a keyfile with `ssh-keygen` and distribute it to the remote hosts and the script calling host. Next, replace `rsh` calls with `ssh` calls as follows.

```
rsh host -l user <command>
```

```
ssh host -i keyfile -l user <command>
```

Then, replace `rcp` calls with `scp` calls as follows.

```
rcp file user@host:<destination>
```

```
scp -i keyfile file user@host:<destination>
```

Sites desiring a more secure approach should use agents. At the system boot time, a user would provide the passphrases needed. This scheme would not work in a lights-out style environment.

Keys do need to be protected. Where security is a concern, load them by hand to prevent tampering. For sites when scalability is the largest concern, place a copy of the keys on your JumpStart™ server and copy them at installation time.

## Common Desktop Environment (CDE)

A limited form of single sign on can be accomplished with `ssh-agent`, an X windows-based passphrase requestor, and some shell code in a user's `~/.dtprofile`. Users will enter their passphrases once and will then be able to log in to any host that honors the key from any local shell window. The downside is that the security of the system is limited by the screensaver password and by user vigilance never to leave an unattended, unlocked session.

A fairly simple X passphrase requestor is `x11-ssh-askpass` available at `http://www.ntrnet.net/~jmknoble/software/x11-ssh-askpass/`. The tool is simple to build and easy to install. After building the tool, integrate it into OpenSSH by installing it as `ssh-askpass` in `<installpoint>/libexec`. You can also integrate this tool into the deployment mechanism. An updated version of makeOpenSSHPackage.ksh will add x11-ssh-askpass if it is present during the OBSDssh package creation.The following are instructions for integration.

```
$ su -
<password>
# cp x11-ssh-askpass /usr/local/ssh/libexec/ssh-askpass
# cd /usr/local/ssh/libexec
# chmod 555 ssh-askpass
# chown root:other ssh-askpass
```

The following code fragment is needed in the ~/.dtprofile. When the users log in to a CDE session, x11-ssh-askpass (ssh-askpass) prompts them for the passphrases to their keys. If users have multiple keys to add, then successive calls to ssh-add with the keys identity strings will be needed.

```
# This example is specific to OBSDssh
# ssh agent support
# if /opt/OBSDssh/bin/ssh-agent does not exist, then do not run.
if [ -f /opt/OBSDssh/bin/ssh-agent ]; then
        eval `/opt/OBSDssh/bin/ssh-agent'
# add keys here. Need one ssh-add per key. Consult the man page.
#       Only add keys if the X passphrase requestor is present.
        if [ -x /opt/OBSDssh/libexec/ssh-askpass ]; then
                /opt/OBSDssh/bin/ssh-add
        fi
fi
```

## Proxies

You can integrate OpenSSH with a SOCKS proxy with the runsocks command. Unfortunately, this requires the user to type a long command line or requires the creation of a small shell script. The following is an example proxy connection shell script.

```
#!/usr/bin/ksh
# Some sites may require SOCKS_SERVER and LD_LIBRARY_PATH
explicitly set
/usr/bin/env SOCKS_SERVER=sockserver:1080 LD_LIBRARY_PATH=/usr/
local/socks/lib \
/usr/local/socks/bin/runsocks /opt/OBSDssh/bin/ssh
remote.host.com
```

Within OpenSSH, there is also the `ProxyCommand` user configuration option. You can use this option to specify a helper application that OpenSSH will read and write to for accessing the remote host. The creation of this application is outside the scope of this article. Consult the man page for `ssh(1)` and *SSH, The Secure Shell* for more details.

### makeOpenSSHPackage.ksh

You can add local configuration files to the OBSDssh package by replacing `sshd_config.out` with your server configuration and by replacing `ssh_config.out` with your global client configuration. Then, run the `makeOpenSSHPackage.ksh` script to generate the OBSDssh package. You can also modify this script to include `x11-ssh-askpass` as `ssh-askpass` into the package as well. (Find the section where the `sftp-server` executable is packaged.)

## Summary

The network was never secure. OpenSSH provides strong authentication, protected network connections, support for wrapping legacy protocols, and improved remote X windows security. The price of this protection is careful consideration of configuration details and the issues with key handling. Being aware of the difficulties should provide a successful integration of OpenSSH into the enterprise.

Consider disabling unsafe network services such as `telnetd`, `ftpd`, `rlogind`, and `rshd` after the deployment of OpenSSH.

# Appendix A

Example server configuration

```
#
#       Example sshd_config with recommended server defaults.
#
# Protocol two for security
Protocol 2
# Only if legacy clients are an issue
# If legacy SSH version one support is turned on, there are
other
# configuration options to consider. Consult the sshd(8)
manpage.
#Protocol 2,1
#
# If your jurisdiction requires a banner
#Banner /etc/issue
#
# Allow encrypted tunnels for insecure protocols
AllowTCPForwarding yes
GatewayPorts no
X11Forwarding yes
X11DisplayOffset 10
XAuthLocation /usr/X/bin/xauth
#
KeepAlive yes
#
# Turn on for BSM auditing. Feature is not compatible with X
forwarding.
# Do NOT turn on with a version of OpenSSH previous to 3.0.2 due
a local root exploit.
UseLogin no
#
# Allow sftp access.
Subsystem       sftp    /opt/OBSDssh/libexec/sftp-server
```

```
#
# Authentication methods
# Do not allow weak rhosts style authentication
HostbasedAuthentication no
RhostsAuthentication no
IgnoreRhosts yes
# Do not allow empty passwords
PermitEmptyPasswords no
# Force users to su to root
PermitRootLogin no
# If machine lives on the Internet, public key only
PasswordAuthentication no
PubkeyAuthentication yes
# Sixty seconds to login
LoginGraceTime  60
#
# User management details
# Login shell should check for email and display Message Of The
Day
CheckMail no
PrintMotd no
PrintLastLog yes
# Prevent tampering of user’s ~/.ssh due to poor permissions
StrictModes yes
#
#
# Legacy Protocol one options
# Use only if supporting legacy clients
#KeyRegenerationInterval 1800
#ServerKeyBits 768
#RSAAuthentication yes
#RhostsRSAAuthentication no
```

Example client configuration

```
#
#       Example ~/.ssh/config with recommended user defaults.
#
# standard host with a nickname
Host foo
HostName foo.eng.acme.com
#
# standard host with a port forwarded
Host test
HostName test.corp.acme.com
# Allow HTTP access to the corporate internal server
LocalForward 8080 www.corp.acme.com:80
#
# Host with only legacy SSH1 support
Host legacy
HostName legacy.acme.com
Protocol 1
User oldtimer
#
# Global defaults
Host *
# Only allow SSH version two protocol except where specifically
listed.
Protocol 2
# After three connection attempts give up
ConnectionAttempts 3
# Allow X display forwarding
ForwardX11 yes
# Do not allow other hosts to connect to forwarded ports
GatewayPorts no
# Check if host key has changed due to DNS spoofing
CheckHostIP yes
# Never use the insecure rsh
FallBackToRsh no
# If encountering a new host, ask about accepting the host key
StrictHostKeyChecking ask
# Solaris location of xauth
XAuthLocation /usr/X/bin/xauth
# Detect if unable to connect to the server temporarily
KeepAlive yes
```

# Bibliography

Barret and Silverman, *SSH The Secure Shell*, 2001. O'Reilly & Associates.

CORE SDI S. A., "SSH Insertion Attack," `http://www.corest.com/pressroom/advisories_desplegado.php?idxsection=10&idx=131.`

Griffin, Wesley. "Storing SSH Host Keys in DNS," `draft-ietf-secsh-dns-key-format-00.txt`, NAI Labs, Glenwood, MD, May 2001.

Knoble, Jim, x11-ssh-askpass, `http://www.ntrnet.net/~jmknoble/software/x11-ssh-askpass/`

NEC Corporation, SOCKS, `http://www.socks.nec.com/`

Noordergraaf, Alex, "Solaris Operating Environment Minimization for Security: A Simple, Reproducible and Secure Application Installment Methodology - Updated for Solaris 8 Operating Environment" Sun BluePrints OnLine, November 2000, `http://www.sun.com/blueprints/1100/minimize-updt1.pdf`

Ornaghi, Alberto and Valleri, Marco, Ettercap `http://ettercap.sourceforge.net`

Provos, Niels, `scanssh`, `http://www.monkey.org/~provos/scanssh`

Reid, Jason and Watson, Keith, "Building and Deploying OpenSSH for the Solaris Operating Environment," Sun BluePrints OnLine, July 2001, `http://www.sun.com/blueprints/0701/openSSH.pdf`

RSA Laboratories, RSA Cryptography FAQ, `http://www.rsa.com/rsalabs/faq/index.html`

SECSH IETF Working Group, `http://www.ietf.org/html.charters/secsh-charter.html`

Solar Designer, "Passive Analysis of SSH Traffic," `http://www.openwall.com/advisories/OW-003-ssh-traffic-analysis.txt`

van der Lubbe, Jan C A, *Basic Methods of Cryptography*, 1998. Cambridge University Press

Weise, Joel, "Public Key Infrastructure Overview," Sun BluePrints OnLine, August 2001, `http://www.sun.com/blueprints/0801/publickey.pdf`

Weise, Joel and Martin, Charles, “Developing a Security Policy,” Sun BluePrints OnLine, Decemeber 2001, `http://www.sun.com/blueprints/1201/secpolicy.pdf`

## Author’s Bio: Jason Reid

*Jason Reid is a test engineer in the Solaris System Test Group. He has also been an SQA engineer in the Developer Tools Group. Prior to joining Sun, Jason worked at the Purdue University Computing Center as a UNIX® system administrator, while obtaining his BS in Computer Science.*