MARK MCCULLOUGH

# secure automated file transfer

Mark McCullough is a senior system administrator at SBC, where he focuses on issues of security and system automation.

*mmccul@earthlink.net*

WHEN WE WERE ASKED TO SET UP AN automated file transfer of customer data with an outside vendor, we developed a method using SSH to do this. Soon several other projects requested the same structure to transfer files containing sensitive data. Various refinements were later made, as well as more in-depth security analysis. Much later, a variation was even developed that permitted outside contractors to securely access individual servers without compromising the security of the network. Each of these variations used SSH to transfer the files in a way that did not lower the overall security of the system, yet could be completely automated.

## Requirements

Any choice of product for file transfer must offer the following three characteristics: it must use or allow for good authentication; it must encrypt the data during transfer; the method must be completely automated.

Securing interactive access has similar requirements except, obviously, automation is no longer needed. It should go without saying that the setup must not introduce significant new security risks.

Today, business-to-business virtual private networks (B2B VPNs) provide the ability to secure network transactions from the perimeter of one company to the perimeter of another. Given the axiom that most computer intrusions come from inside the corporate intranet, stopping the security at the corporate perimeter is insufficient.

The first requirement is obviously met by SSH. Public key exchange and encrypting user authorization credentials provide good authentication. It is also possible to use true two-factor authentication with SSH.

By nature, SSH also meets the second requirement, encrypting the session. HR and legal requirements may require data confidentiality, especially if the data is personal, medical, or financial. Future developments like widespread adoption of IPv6 or use of point-to-point IPSec may mitigate this requirement by moving the burden from the application layer to the transport layer.

The third requirement appears to be readily met by SSH as well. Use of SSH keys in memory makes for easy scripting.

For the immediate problem of transferring files, several alternatives to SSH are available (e.g., rcp, FTP, NetBIOS, HTTPS, rlp). Each of these alternatives has advantages and disadvantages and may in some situations be more appropriate than SSH; in many cases, however, SSH remains the preferred choice.

## SSH for File Transfer

Unfortunately, SSH needs some work before it can be appropriately secured, especially when used for file transfer. Unlike FTP, creating individual access for downloads on SSH provides for the possibility of interactive access, something not at all desirable in most situations. SCP-only accounts are possible but difficult to properly set up and maintain. Perhaps more important, protecting the system so that users cannot overwrite key files offers a unique challenge. A user able to send an arbitrary file name could potentially overwrite the very file that prevents interactive access with an altered file.

Fortunately, this deficiency in SSH can be addressed, but it does require some work on both the client end and the server end. SSH offers the possibility of authenticating by means of a public key instead of just password authentication. One of the most useful features of public key authentication is the ability to specify a forced command that will be executed anytime a user successfully authenticates to the system using that public key.

Using forced commands, one must give up on the SCP and SFTP ease-of-use features. Instead, the fact that SSH will pass STDIN through the encrypted session to STDOUT on the remote end is used. Also, forced commands permit immediate command execution method through normal shell notation.

To specify a forced command to a system using OpenSSH, one prepends `command="some command"` to the beginning of the line containing the public key. Other SSH servers may use a different syntax. Consult your documentation on the correct format, but be sure to enclose your forced command in double quotes to protect spaces.

## Direct Server-to-Server Transfers

There are two basic ways to transfer a file: by a PUT or by a GET (to borrow from the FTP terminology). Windows systems can readily be the initiator just as easily as UNIX, but it is presumed here that a UNIX box is the server receiving the request for either the PUT or the GET.

The simplest application of file transfer assumes that the client will be directly transferring files to the next server with a PUT. All the more complex concepts in file transfer presume that one understands this direct server-to-server file transfer method.

The forced command needs to receive a file sent on STDIN to a hard-coded file name. It is important that the sender not have any input on the forced command executed to protect against tainting of the command line or overwriting the authorized_keys file. The dd command was chosen for this purpose because of its ability to handle both STDIN and STDOUT easily. If there is no fear of file clashing (the same file will be transferred once a day and should overwrite the existing file), then the forced command on the server to PUT a file would be

```
command="/usr/bin/dd of=/path/to/file"
```

Usually, it is important to protect against file-name clashes, so a date string might be added:

```
command="/usr/bin/dd of=/path/somename-\
'/usr/bin/date +%Y-%m-%dT%H:%M:%S'.$$"
```

Note the use of the shell syntax for process ID (PID); a full datestamp provides protection for a unique file name. This method does not attempt to protect against a local attack to predict the file name. The file transferred should be placed in a location of the drive where permissions can protect against unauthorized access.

Unfortunately, many times a custom file name, determined by the application, is required. For example, each send of the file may require a sequence number in the file name which changes every day. Allowing the sender to specify the file name without checking it first is unacceptable. Prepending the file name as the first line of the file can be used to embed the file name even in a binary file. In UNIX there are, of course, multiple ways to do this without editing the file. Windows users can concatenate two files with the copy command:

```
copy file1 + file2 newfile
```

A trivial script could read the file up to the first line break and use what is read to obtain the desired file name and write the rest of the file to that file name:

```
#!/usr/bin/ksh
SOURCEFILE=$1
INNAME=$(/usr/bin/head -1 $SOURCEFILE)
FILENAME=$(/usr/bin/basename $INNAME)
/usr/bin/tail +2 $SOURCEFILE > $FILENAME
```

Obviously, such a method should have sanity checks appropriate for the location to protect against deliberate or accidental overwriting of a different file. The above example could easily be modified to send files to specific directories depending on the file name. The use of a base name helps prevent key files from being overwritten.

The GET procedure is fundamentally similar. The primary restriction of this method is that each public key can only execute one function, such as get a file of a given file name. The forced command in this structure would appear as:

```
command="dd if=/file/to/send"
```

Obviously, /file/to/send is the full specification of the file name to be sent to the requesting client. That the original command line can be used as a variable to the command string might allow the requester the ability to request arbitrary file names, but extra care must be taken to detaint any information passed by the requester.

## ClientSide

The client sends the file (PUT) by piping the data into the SSH process. Under UNIX, one way to do this would be:

```
cat file | ssh remotehost
```

If a set file name is to be placed as the first line of the file, one may wish to do it at the time the file is sent:

```
(echo desiredfilename ; cat file) | ssh remotehost
```

Windows provides a similar mechanism; the PUT command using PuTTY might be

```
type file | plink remotehost
```

The GET procedure is fundamentally very similar to the PUT. However, instead of sending the data through the SSH session, it is received and then output to a file:

```
ssh remotehost | dd of=/file/to/receive
```

The Windows command is very similar:

```
plink remotehost > /file/to/receive
```

One caveat about the Windows procedure: Some Windows SSH clients may not be able to handle reasonably sized files transferred in this manner. In 2003, the then current version of SecureCRT was unable to handle more than 128KB of data. PuTTY, on the other hand, was capable of transferring well over a megabyte of data.

## "Man in the Middle" Bastion Host

Usually, direct SSH connections are not possible from the outside company directly to the endpoint, and so a bastion host is used for the transfer. Two possibilities exist. In the first case, the bastion host may be required to intercept files suspected of containing inappropriate content. In this case, some minor scripting knowledge is required. In the second case, it may be unacceptable for the bastion host to be able to intercept the files being transferred. This option requires more security awareness on the part of the destination-server administrators rather than concentrating the security awareness on a perimeter box.

In the first scenario with a bastion host, the sender transfers the file in the manner described previously, but this time sending to the bastion host. Instead of using the direct forced command, the bastion host uses a script, which receives the file as indicated and then immediately plays the role of the client, transferring the file to the final destination. In the following example, the forced command that invokes the script also passes a single argument of the final destination server:

```
#!/usr/bin/ksh
MSGFILE=/appl/safedir/tmpfile-\
'/usr/bin/date +%Y-%m-%DT%H:%M:%S'.$$
/usr/bin/dd of=$MSGFILE
/usr/bin/dd if=$MSGFILE | /usr/bin/ssh $1
/usr/bin/rm $MSGFILE
```

Once the file is transferred, the bastion host can safely remove the file. If there is reason to inspect a file, this can be done by not removing the file or by transferring it to a different server inside the organization.

As a variation, if the end server expects the file to be received by scp, then the second dd command could be changed to

```
scp $MSGFILE $1:.
```

## Pass-Through Bastion Host

In the second method, direct transfer of the file, it is unacceptable for the bastion host to intercept the files being transferred. A command sequence example is:

```
ssh -L 2001:finalhost:22 -f bastionhost sleep 60
cat /file/to/send | ssh -p 2001 localhost dd \
of=/file/being/sent
```

Note, the dd step on the second line is redundant and exists for documentation purposes only. In this scenario, the bastion host has a forced command of sleep 60 but, unlike the first method, does permit port forwarding.

Because the bastion host only serves as a port forward, it is impossible to intercept the file at the intermediate point. This method is appropriate for instances where the initiator and receiver cannot directly route to each other. It is also important to note that this method could be extended to chain multiple bastion hosts together (again, the dd command here is for documentation purposes only):

```
ssh -L 2001:bastion2:2222 -f bastion1 sleep 60
ssh -L 2002:remoteserver:22 -f -p 2001 \
localhost sleep 60
cat /file/to/send | ssh -p 2002 localhost dd \
of=/file/sent
```

The forced command on the intermediate hosts would match the specified command, a sleep command. On

the final box, the forced command matches instances, described above, where no bastion host is used.

A caveat: Because this second method permits port forwarding, it is possible for a user to port forward beyond the intended destination, permitting the user to leapfrog to other systems in your network. A leapfrog attack can be difficult to detect because, until it goes beyond the final destination, the attack resembles legitimate file-transfer traffic. As such, it is more appropriately used when the users of both the initiator and the receiver have legitimate access to the network but cannot directly talk to each other.

## Interactive Access

Interactive user access presents a different problem. Users cannot be restricted to a single command that is completely locked down. It is very difficult to ensure that a port forwarding is not created to transfer files back and forth. With B2B VPNs, the question might even be, why worry? Interactive access from an outside vendor should not result in the dumbing down of security. As with any box, the box that the vendors access should be locked down, disabling the telnet daemon, the r-commands, etc.

By directing the B2B VPN endpoint to a bastion system, users can be forced to authenticate against this system. This system would be running a specially configured SSH daemon with all port forwarding disabled. (As a note, this will disable the possibility of X11 traffic, but considering the speed of X Windows traffic over a wide area network, this restriction would hopefully not be a serious issue.) Once users land on this box, a forced script will immediate initiate a SSH to their eventual destination. It does not matter if they attempt to construct port forwards beyond the bastion box, because all port forwarding is stopped at this bastion system. They cannot construct a new port forward through that particular hop.

Set either a .profile or forced command to execute a small script for the user. Ensure that when the script exits, the user's connection is terminated. The script should initiate an SSH connection to the next system without ever offering the user the chance to get a shell prompt. This is important because a user with shell access could install their own port forwarder to bypass the protections of the bastion host.

## Configuring the Server for Interactive Access

Several of the listed options require changes to the server. Some of the options require that port forwarding be enabled, but others specifically shut it down for added security. In the OpenSSH server, there are several options that should be set to add to the security of the system. In the sshd_config file:

```
X11Forwarding no
AllowTcpForwarding no
```

For the indirect file transfer where the files land on the bastion host, also set (using command-line option names):

```
no-agent-forwarding
no-pty
```

## Conclusion

Using the built-in features of SSH permits two companies to safely transfer sensitive files in an automated manner. Of course, in any discussion of computer security, it is important to evaluate recommendations carefully prior to implementing them at any given site. It is hoped, however, that the tools described here will make it easier to handle the problems of file transfer.


### ACKNOWLEDGMENTS

The procedures described above were developed with the cooperation of Ivan Holt and Darrell Widhalm from SBC. Their input helped bring these methods to full operational use.