



# Global Information Assurance Certification Paper

Copyright SANS Institute  
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

## Interested in learning more?

Check out the list of upcoming events offering  
"Security Essentials: Network, Endpoint, and Cloud (Security 401)"  
at <http://www.giac.org/registration/gsec>

## **In Search of Secure File Transfer Across the Internet**

### ***Abstract***

Every day, organizations all over the world move valuable data across the Internet. Files containing sensitive information must be sent across an insecure channel to their destination – subject to interception along the way. The standard TCP/IP utility, File Transfer Protocol (FTP) doesn't afford any significant level of security. And although there are command extensions to the original FTP specification that enable the integration of FTP with the highly secure SSL/TLS standard, FTP still possesses a dual-port/dual-session architecture which reduces the potential for secure firewall configuration. Secure FTP, a subset of the SSH (Secure Shell) protocol, provides FTP-style file transfer across a single encrypted session. This paper discusses the advantages of using SFTP over other methods of file transfer, and compares the installation, costs, and features of three versions of Secure Shell server on the Windows 2000 platform.

### ***Friendly-neighborhood FTP.***

The Internet has become the hub of global communication, and a core set of simple, TCP/IP-based utilities serves the crucial inter-platform objective of performing standard data operations. These include: "http" which provides platform-independent text presentation and data navigation; "smtp" which provides asynchronous message/file delivery; "telnet" which offers a shell interface for network-connected nodes; and "ftp" which provides platform-independent file transfer capabilities. When the Internet was first conceived, the primary consideration in the design of these utilities was access across diverse platforms as echoed in the original FTP specifications in RFC 959:

The objectives of FTP are 1) to promote sharing of files, 2) to encourage indirect or implicit use of remote computers, 3) to shield a user from variations in file storage systems among hosts, and 4) to transfer data reliably and efficiently [1].

These objectives are effectively accomplished using the simple-to-use File Transfer Protocol. FTP clients are native to every operating system in graphical user interface (GUI) and command-line versions. Programmers and script-writers are able to automate file transfers with ease. Windows and Unix servers can easily exchange files with minimal effort. Operating systems of every flavor come with an easily configurable FTP server and client.

The FTP service offers a simple method of file storage and retrieval: connect via TCP/IP, login, download, disconnect. There are no requirements for domain membership (as is the case for typical NetBIOS file transfers in Windows domain, trusted networks). Files are available from any internet-connected computer (assuming the FTP server possesses a publicly-accessible IP address). Using anonymous FTP, there is not even a requirement for a pre-configured username and password. This is especially useful for companies who post software or product documentation for free distribution. In the same way that web pages provide 24x7 access to information, FTP servers also provide always-available service. This brings us to our problem from a security perspective.

### ***The two main drawbacks of FTP.***

#### ***Number One: Weak Password Security***

All FTP communications are sent in clear text. That means that if I log into my company's FTP server with my domain password, then I am broadcasting that password in plain sight across the Internet. A simple packet capture can yield the key to my company's sensitive data and other resources, limited only by the privileges granted to the stolen password. (This is an excellent illustration of the importance of "least privilege" and "defense-in-depth" [2]. "Least Privilege" suggests that I should set up user accounts exclusively for FTP (and perhaps other insecure, internet zone services) that are denied access to resources in the trusted network. That way, a cracked or intercepted password would have some access to files in the ftp-accessible directories, but no access beyond. "Defense-in Depth" suggests that I use multiple layers of defense. So for example, in addition to using special, internet-zone passwords to protect my FTP-accessible folders, I should also use file system access control lists; host-based packet filtering; and either read-only or write-only FTP sites).

#### ***Number Two: Perimeter Defense Compromises***

FTP utilizes at least two TCP ports on each machine per data transfer. The first port (TCP port 21) sets up the "control channel" to establish an authenticated session and an interface for subsequent commands. The replies to these commands (e.g., directory listings; transferred files) are sent via the "data channel", typically port 20. The mechanism for establishing the data channel occurs in two modes: active and passive. In Active mode, following the initiation of the control channel session, the FTP server connects from port 20 to a random port on the remote client machine (by default, the client port initially connecting to server port 21 + 1). For this to work, the client-side firewall must permit incoming traffic to all ports above port 1024: a substantial perimeter security compromise. (Some implementations of the FTP server and client offer the ability to specify a range of so-called random ports, which in turn gives tighter control over firewall thru-traffic access control). This scenario is advantageous to the FTP Server security administrator. In this case only inbound port 21 is required, although outbound access to all ports above 1024 is also required. Note that these port

rules are only required for the IP address of the FTP server. However for the client side security administrator, Active mode is simply too insecure. It would require opening the entire, client-side network (to accommodate every ftp client running on every machine) to all traffic above port 1024.

In the other transfer mode, Passive mode, the client initiates the data channel session from it's own random port (by default, the client port initially connecting to server port 21 + 1) to a random port on the server (usually port >1024) allocated and communicated by the FTP server using the PORT command in response to the client issuing the PASV command [3,4]. This alternative is more secure for the client-side firewall, since it doesn't require the firewall to allow any inbound TCP/IP traffic originating from the Internet. Unfortunately however, it does require the client's ability to initiate connections from a random port to a random port – a very open, and unacceptable firewall configuration if the firewall administrator is trying to lock down outgoing traffic (for example, to protect against outgoing transmissions originated from a Trojan, backdoor, or packet sniffer installed on the internal network). Even worse, it requires the server-side firewall to allow incoming connections to all ports above 1024: the same substantial perimeter security comprise, this time faced by the FTP server network.

### ***Locking Down FTP.***

With the explosive growth of worldwide Internet traffic and commerce, security has become a major concern. Whereas "back in the old days, we trusted our neighbors and left our doors unlocked", we no longer know who our neighbors are, and we know that some of them are primarily interested in causing mayhem in our private networks for fun or profit. Hence, the core network services (mail, web, shell, file transfer) all have more secure counterparts, with varying degrees of public acceptance.

In an effort to secure FTP, a series of RFC's and file transfer alternatives were introduced in the late 1990's. In their "FTP Security Extensions" (RFC 2228), Horowitz and Lunt describe additional FTP commands that allow for standard FTP sessions to proceed over an encrypted channel. [5]. Currently, the most noteworthy implementation of this standard is FTPS or FTP over SSL (Secure Socket Layers) and its most recent manifestation, TLS [6]. While this implementation offers strong encryption using available algorithms such as AES, 3DES and Blowfish [7], it still suffers from the feature of FTP in which two TCP ports are required. FTPS implementations generally require that both the FTPS server and client both reside on public IP addresses (no NAT'ed addresses). With the additional port negotiation complexities of FTP over SSL, the ability to create fault-tolerant, fail-over network configurations, utilize network address translation, and tighten firewall security becomes significantly more challenging.

In situations where standard FTP servers are required for their universal accessibility and administrative simplicity, there are published guidelines to help system administrators and programmers configure systems to minimize vulnerability (e.g. read-only and write-only anonymous FTP; consideration of system resource access control; firewall configuration to protect against FTP Bounce attacks) and write code which minimizes these vulnerabilities [8,9,10].

### ***New Technologies Built on Secure Foundations: Secure Shell and Secure FTP.***

SFTP (secure file transfer protocol), a feature of SSH (Secure Shell), provides a solution to the two-port problem. Whereas FTPS calls for the setup of a secure channel (using SSL/TLS) within the context of FTP's dual-channel, multi-session architecture, SFTP performs file transfer within the context of SSH's single channel architecture. SFTP has an interface and commands that closely resemble those available in standard FTP. SSH creates this secure channel through an exchange of keys or certificates. The SSH client contacts the SSH server which responds by sending the client its host public key. The client can use this response to decide whether to trust the server in several ways. First, the client can compare the host address/key combination with the client's own stored key database. When the client has previously connected to this server, and if the key is not what the client encountered previously, a warning is generated. The user must then decide whether the key has been regenerated at the server since last connection, or if the SSH server host is being spoofed by a would-be attacker. Second, the client can verify the hostname against a certificate (if this is the authentication mechanism in use). Third, the client can perform a reverse resolution (IP to DNS) to cross check and assure that the two match up. Fourth and last, the client can decline a connection if the server is running a previous version of SSH, or is not prepared to use a compatible encryption method.

Assuming that the client has validated the server's authenticity and both are able to negotiate a compatible set of encryption, authentication, and other session parameters, a secure session is established. Essentially, this Secure Shell session functions as an encrypted tunnel through which shell commands and other file and data transactions can be conducted [11].

The SFTP server is a subsystem of SSH. SFTP clients (such as Putty for Windows, and Unix native SFTP clients) initiate the SSH session and connect to a directory on the SSH/SFTP server for the sending or receiving of files. Standard FTP commands such as "cd" (change directory); "ls" (list directory), "get" (receive file), "put" (send file), and "quit" (end the session) are all supported in SFTP. This apparent similarity makes it easy for those familiar with command-line FTP clients. (For those requiring a GUI interface, there are several listed below in Appendix A).

Tatu Ylonen of the University of Finland created SSH/SFTP in 1995. The open source version of SFTP (a sub-protocol of SSH) was created in 2000 by Markus Friedl [12]. Since 1999 when SSH was first released with OpenBSD 2.6, SSH (and later, SFTP) has become a standard utility across the Unix/Linux landscape. Cisco routers and PIX firewalls utilize IOS-based SSH servers for secure configuration over the Internet. Windows 2000/XP/.Net all provide native telnet, FTP, and HTTP services, reflecting the adoption by Microsoft of TCP/IP as a central architectural feature. However, the GUI-oriented Windows platform still does not provide native SSH/SFTP support. The remainder of this paper discusses the selection, installation, features, and maintenance of SSH/SFTP server applications for the Windows platform.

### ***Choosing a Secure File Transfer Protocol Server.***

If you have decided that your organization requires a secure procedure for transferring sensitive data over the Internet – including account passwords and confidential files, you must now consider which software package best meets your needs. If you are simply tightening up your existing security and you are currently using a password-restricted FTP site, your needs may be somewhat different than if you require the rock-solid, cutting-edge security of a financial, medical, or military organization. In these situations, you must consider the objectives and resources available for the project. Ultimately, you will be considering *costs* (relating to purchase, implementation, and maintenance); *features* (simplicity versus complexity, required commands for accomplishing tasks); and *security* (does one package offer a higher level than another?). In deciding which version to implement, this writer recommends setting up a test lab, installing the versions under consideration (all versions mentioned are available as evaluations), and testing operation in your particular environment.

The following discussion focuses mainly on OpenSSH/Cygwin on Windows 2000. Subsequent brief comparisons of WinSSHD and SSH Communications (SSH.com) along with GUI screenshots round out the picture of what different versions may offer.

#### ***OpenSSH - Costs***

SFTP servers range in cost from free (but potentially labor-intensive), open source (under the GNU public license) versions (including all OpenSSH/Cygwin versions), to modestly-priced (approximately \$100 per server), minimum-functionality versions (WinSSHD.com, Foxitsoft.com), to high-priced (\$650-\$800), fully featured versions (SSH.com, F-Secure.com). This price range leads many to believe that the free versions cost the least. However, implementation costs for the free, open source versions are easy to underestimate and can be quite significant if you attempt a full Cygwin install, compile incompatible versions, or are not comfortable with the Unix shell environment. Moreover, OpenSSH configuration occurs through editing the `sshd_config` file (in other

words, you would be well advised to begin your OpenSSH evaluation by reviewing the man pages for "sshd\_config"[13]). Other manual configuration tasks for the OpenSSH version include 1) configuring valid paths and files for client key-based and host-based authentication, and 2) manually editing the passwd and group files to accept/deny individual users. Note that while the full Cygwin install generates the passwd file from the existing local or domain user database, subsequent changes to the Windows user account database do not automatically update the "passwd" and "group". A utility is available with Cygwin (mkpasswd) that can easily make these changes from the command line. However, if accounts are frequently added and removed from the account database (and this may be local, NT4 domain, or active directory), then the redundant task of updating the SSHD passwd file could be cumbersome. One final note regarding the costs of implementing the "free", OpenSSH package: all versions of OpenSSH do not run on all versions of Cygwin. In this writer's initial trials including minimal and full Cygwin installs, numerous hours were spent pursuing a viable combination of packages and versions. Although there is copious documentation on the Internet regarding installation steps for SSHD on Cygwin, all renditions may not be comprehensive. For example, following many of the Internet references for OpenSSH/Cygwin installation sources generally resulted in a reliably performing SSH server, but SFTP usually would not work. At other times, it became clear (from personal experience and exhaustive web searches) that certain versions of OpenSSH or OpenSSL (the complementary encryption package) were simply buggy. In the midst of this writer's odyssey to find a reliable procedure for installing SFTP server over Cygwin on Windows 2000, this writer discovered a free Windows executable package which copies the essential files, makes a few choice registry edits and environmental variable changes, and offers straightforward and accurate documentation on completing the manual (mkpasswd) changes to the passwd file. Once this writer removed traces of the previous Cygwin installations, this version of the Cygwin/OpenSSH/SFTP-Server package was functioning in 15 minutes [14].

### *OpenSSH - Features*

The primary benefit of using an OpenSSH SFTP server is its ubiquity: SSH servers abound in the Unix world, and client software, as well as performance expectation, is geared to that fact. OpenSSH possesses all the standard features of SSH security specifically and Unix authentication in general. However, depending on your method of installing SSH and Cygwin, you may not have all these options available. (Testing every authentication option and documenting the necessary Cygwin ".dll" file required is beyond the scope of this paper.) Suffice it to say, OpenSSH on Cygwin in any type of install will be able to handle key generation and exchange, since this is the essential feature of SSH. Features such as host-based (considered insecure when used exclusively due to IP-spoofing vulnerabilities) and public key authentication require the creation of specialized files ("hosts.equiv" for host-based authentication, and "authorized\_keys" for public key authentication) in order to function. (See next section for server-side debug output of: 1) a keyboard-interactive, password

authenticated session; and 2) an RSA public key authenticated session. In both cases, a PuTTY/PSFTP client connected to an OpenSSH server). Features such as Kerberos authentication require system wide configuration changes that are likely very difficult or impossible to implement on a Cygwin-based Windows NT installation. Finally, other features such as user and group level access lists; compression; designated ciphers; hash algorithms (MAC); logging levels/paths; TCP/IP addressing and port details; and port-forwarding can all be configured in the `sshd_config` file [13]. Note that most (not all) of these features characterize the Windows SSH server versions as well. Some Windows versions offer additional features. And all Windows versions offer a more "user-friendly" (i.e. GUI) interface.

### *OpenSSH - Security*

Although all versions have their specific vulnerabilities (and they always will), patches and new versions are always being created to fix these vulnerabilities. OpenSSH is no exception. Both Open Source and Proprietary software proponents have their arguments as to why one is more secure than the other. The Open Source camp will argue that there is an international community of seasoned and dedicated programmers testing and reviewing open source software all the time. When a vulnerability is detected, a patch may be posted and distributed within hours – as opposed to commercial software companies who may tend to address a problem if it makes financial sense in terms of "costs of developing the patch" versus "costs of lost business due to damaged consumer confidence". The proponents of Proprietary software are likely to point to the additional security offered by keeping the software code secret and protected. They will argue that public access to open source code clearly gives hackers all the material necessary to examine every line for errors (such as unvalidated data fields which could be used for buffer overflow attacks) and a lot of opportunity to discover and test potential exploits. In a casual online review of various, published SSH vulnerabilities (especially at [www.cert.org](http://www.cert.org): search for SSH + vulnerability), there seems to be a relatively equal prevalence of vulnerabilities in open source (OpenSSH, BSD) versions and proprietary (SSH Communications, F-Secure) versions. In general, however, version 2 of the SSH protocol is considered secure, while version 1 is considered much less secure. In summary, it is recommended that SSH version 2 be used; ideally with ciphers such as 3DES, Blowfish, and AES256, rather than DES (which is not considered secure); using RSA generated keys rather than DSA; and staying current with patches and strong password policies. Configuration options such as disabling root login and using privilege separation (child SSH processes run under the client user account privilege rather than the account which the SSH service is running under (typically with system or root privileges) are also good practices to consider when implementing an SSH server.

### ***Description of PuTTY/PSFTP to OpenSSH Server Sessions.***



### Case 1: Password Authentication

In this session, a connection was made from a PSFTP client (see Appendix B for PuTTY client configuration screen shots) to an OpenSSH server (on Windows 2000 with Cygwin, minimal install) using PuTTY default settings.

On line 1, the server is started in debug mode (-d) which enables verbose logging to the console:

1. F:\Program Files\OpenSSH\usr\sbin>SSHd -d

Lines 2 – 11 are pre-connection. The server displays its configured operating parameters. Items such as listening address, TCP port, and host key types are easily configurable through the sshd\_config file:

```
2. debug1: SSHd version OpenSSH_3.5p1
3. debug1: read PEM private key done: type RSA
4. debug1: private host key: #0 type 1 RSA
5. debug1: read PEM private key done: type DSA
6. debug1: private host key: #1 type 2 DSA
7. debug1: Bind to port 22 on 0.0.0.0.
8. Server listening on 0.0.0.0 port 22.
9. debug1: Server will not fork when running in debugging mode.
10. Connection from 192.168.0.250 port 4533
11. debug1: Client protocol version 2.0; client software version PuTTY-
    Release-0.53b
12. debug1: no match: PuTTY-Release-0.53b
13. debug1: Enabling compatibility mode for protocol 2.0
14. debug1: Local version string SSH-2.0-OpenSSH_3.5p1
15. debug1: list_hostkey_types: SSH-rsa,SSH-dss
```

Line 16 initiates the key exchange process. On line 18, the client sends an encryption parameter request to the server, in this case to use the AES256 cipher and SHA hash (MAC) algorithm. The key exchange completes on line 33:

```
16. debug1: SSH2_MSG_KEXINIT sent
17. debug1: SSH2_MSG_KEXINIT received
18. debug1: kex: client->server aes256-cbc hmac-sha1 none
19. debug1: kex: server->client aes256-cbc hmac-sha1 none
20. debug1: SSH2_MSG_KEX_DH_GEX_REQUEST_OLD received
21. debug1: SSH2_MSG_KEX_DH_GEX_GROUP sent
22. debug1: dh_gen_key: priv key bits set: 254/512
23. debug1: bits set: 1531/3191
24. debug1: expecting SSH2_MSG_KEX_DH_GEX_INIT
25. debug1: bits set: 1531/3191
26. debug1: SSH2_MSG_KEX_DH_GEX_REPLY sent
```

```
27.debug1: kex_derive_keys
28.debug1: newkeys: mode 1
29.debug1: SSH2_MSG_NEWKEYS sent
30.debug1: waiting for SSH2_MSG_NEWKEYS
31.debug1: newkeys: mode 0
32.debug1: SSH2_MSG_NEWKEYS received
33.debug1: KEX done
```

On line 34, the client sends an authentication method request to the server. The first request appears to be for "no authentication". The next attempt is keyboard interactive. This method allows for Smart Card, RSA SecurID authentication and other methods. In some configurations of SSH, passwords are sent during keyboard-interactive. In this session, "password" authentication (on line 44) is requested separately:

```
34.debug1: userauth-request for user rs service SSH-connection method
    none
35.debug1: attempt 0 failures 0
36.debug1: userauth_banner: sent
37.Failed none for rs from 192.168.0.250 port 4533 SSH2
38.debug1: userauth-request for user rs service SSH-connection method
    keyboard-interactive
39.debug1: attempt 1 failures 1
40.debug1: keyboard-interactive devs
41.debug1: auth2_challenge: user=rs devs=
42.debug1: kbdint_alloc: devices "
43.Failed keyboard-interactive for rs from 192.168.0.250 port 4533 SSH2
44.debug1: userauth-request for user rs service SSH-connection method
    password
45.debug1: attempt 2 failures 2
46.Accepted password for rs from 192.168.0.250 port 4533 SSH2
47.debug1: Entering interactive session for SSH2.
48.debug1: fd 4 setting O_NONBLOCK
49.debug1: fd 8 setting O_NONBLOCK
50.debug1: server_init_dispatch_20
51.debug1: server_input_channel_open: ctype session rchan 256 win 16384
    max 16384
52.debug1: input_session_request
53.debug1: channel 0: new [server-session]
54.debug1: session_new: init
55.debug1: session_new: session 0
56.debug1: session_open: channel 0
57.debug1: session_open: session 0: link with channel 0
58.debug1: server_input_channel_open: confirm session
59.debug1: server_input_channel_req: channel 0 request subsystem reply 1
60.debug1: session_by_channel: session 0 channel 0
```

61. debug1: session\_input\_channel\_req: session 0 req subsystem

As this connection was initiated completely from the PuTTY SFTP client (PSFTP), an SFTP session is automatically created following successful SSH authentication:

62. subsystem request for sftp

63. debug1: subsystem: exec() /usr/sbin/sftp-server

### *Case 2: RSA Public Key Authentication*

The main benefits of public key authentication include completely automated connections, no passwords to lose or remember, complex strings that are harder to crack with a brute force attack. Vulnerabilities include the ability of an unauthorized user to log in if they gain access to the client machine. Also, it is imperative that the private/public key pair are kept secure.

Again, the server is executed in debug mode. (In general, the server starts as a service). There are a few noteworthy differences in this session. PSFTP uses the PuTTY GUI interface to set connection parameters which are saved with an identifying session name. (See Appendix B for PuTTY client configuration screen shots). The PSFTP connection is made to that session name. In this session, compression was enabled (see line 29), keyboard-interactive was disabled, and public key authentication was enabled (key exchange on lines 39 – 56). The saved PuTTY session also allows one to configure the username for full automation. Leaving that field blank – i.e. requiring the user to enter the username -- would offer a slight measure of protection against console-based intrusions from the client machine. PuTTY is bundled with a utility called puttygen (analogous to OpenSSH's "keygen") which allows the user to generate RSA or DSA keys with variable bits. (RSA is generally seen as a more secure choice). When the private/public key pair are generated, the public key output is displayed and that content is used to create the "authorized\_keys" file in the user's home directory on the SSH server (in the "%userprofile%\ssh" subdirectory):

1. F:\Program Files\OpenSSH\usr\sbin>SSHd -d
2. debug1: SSHd version OpenSSH\_3.5p1
3. debug1: read PEM private key done: type RSA
4. debug1: private host key: #0 type 1 RSA
5. debug1: read PEM private key done: type DSA
6. debug1: private host key: #1 type 2 DSA
7. debug1: Bind to port 22 on 0.0.0.0.
8. Server listening on 0.0.0.0 port 22.
9. debug1: Server will not fork when running in debugging mode.
10. Connection from 192.168.0.250 port 4570

11.debug1: Client protocol version 2.0; client software version PuTTY-Release-0.53b  
12.debug1: no match: PuTTY-Release-0.53b  
13.debug1: Enabling compatibility mode for protocol 2.0  
14.debug1: Local version string SSH-2.0-OpenSSH\_3.5p1  
15.debug1: list\_hostkey\_types: SSH-rsa,SSH-dss  
16.debug1: SSH2\_MSG\_KEXINIT sent  
17.debug1: SSH2\_MSG\_KEXINIT received  
18.debug1: kex: client->server aes256-cbc hmac-sha1 zlib  
19.debug1: kex: server->client aes256-cbc hmac-sha1 zlib  
20.debug1: SSH2\_MSG\_KEX\_DH\_GEX\_REQUEST\_OLD received  
21.debug1: SSH2\_MSG\_KEX\_DH\_GEX\_GROUP sent  
22.debug1: dh\_gen\_key: priv key bits set: 252/512  
23.debug1: bits set: 1601/3191  
24.debug1: expecting SSH2\_MSG\_KEX\_DH\_GEX\_INIT  
25.debug1: bits set: 1587/3191  
26.debug1: SSH2\_MSG\_KEX\_DH\_GEX\_REPLY sent  
27.debug1: kex\_derive\_keys  
28.debug1: newkeys: mode 1  
29.debug1: Enabling compression at level 6.  
30.debug1: SSH2\_MSG\_NEWKEYS sent  
31.debug1: waiting for SSH2\_MSG\_NEWKEYS  
32.debug1: newkeys: mode 0  
33.debug1: SSH2\_MSG\_NEWKEYS received  
34.debug1: KEX done  
35.debug1: userauth-request for user rs service SSH-connection method none  
36.debug1: attempt 0 failures 0  
37.debug1: userauth\_banner: sent  
38.Failed none for rs from 192.168.0.250 port 4570 SSH2  
39.debug1: userauth-request for user rs service SSH-connection method publickey  
40.debug1: attempt 1 failures 1  
41.debug1: test whether pka/gpkblob are acceptable  
42.debug1: temporarily\_use\_uid: 1001/513 (e=1001/513)  
43.debug1: trying public key file /bin/rs/.SSH/authorized\_keys  
44.debug1: matching key found: file /bin/rs/.SSH/authorized\_keys, line 1  
45.Found matching RSA key: 69:f1:6a:29:f2:ae:79:e9:9c:cb:0f:f0:fa:8d:e6:51  
46.debug1: restore\_uid: (unprivileged)  
47.Postponed publickey for rs from 192.168.0.250 port 4570 SSH2  
48.debug1: userauth-request for user rs service SSH-connection method publickey  
49.debug1: attempt 2 failures 1  
50.debug1: temporarily\_use\_uid: 1001/513 (e=1001/513)  
51.debug1: trying public key file /bin/rs/.SSH/authorized\_keys  
52.debug1: matching key found: file /bin/rs/.SSH/authorized\_keys, line 1

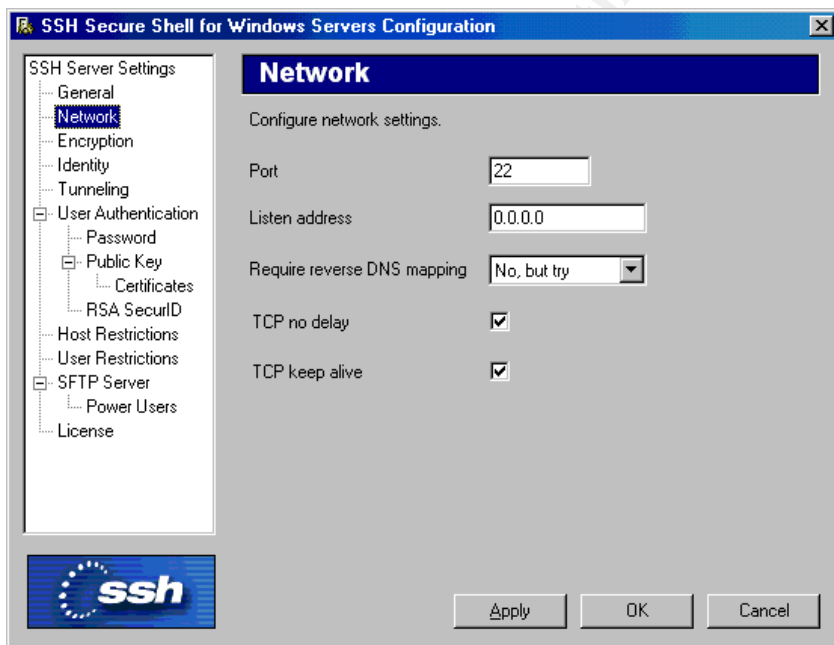
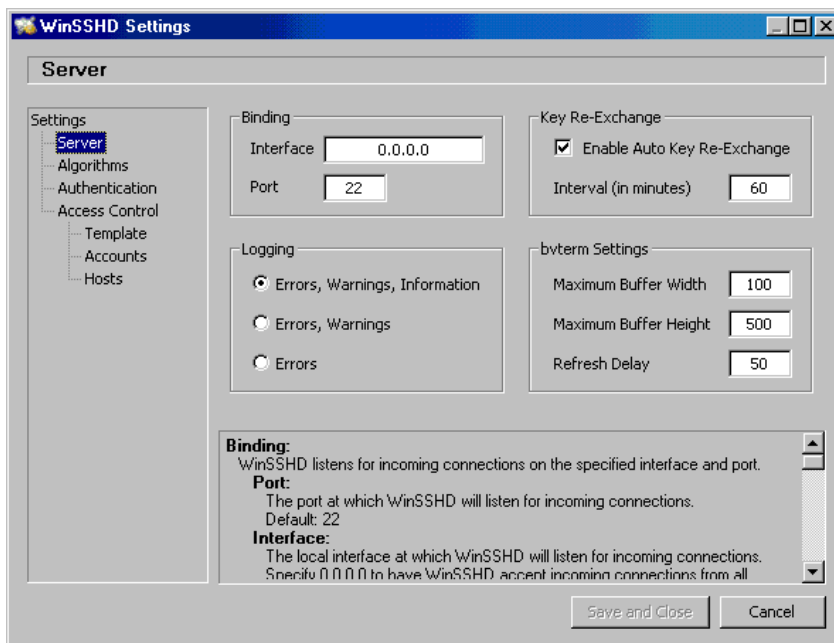
```
53. Found matching RSA key: 69:f1:6a:29:f2:ae:79:e9:9c:cb:0f:f0:fa:8d:e6:51
54. debug1: restore_uid: (unprivileged)
55. debug1: SSH_rsa_verify: signature correct
56. Accepted publickey for rs from 192.168.0.250 port 4570 SSH2
57. debug1: Entering interactive session for SSH2.
58. debug1: fd 4 setting O_NONBLOCK
59. debug1: fd 8 setting O_NONBLOCK
60. debug1: server_init_dispatch_20
61. debug1: server_input_channel_open: ctype session rchan 256 win 16384
    max 16384
62. debug1: input_session_request
63. debug1: channel 0: new [server-session]
64. debug1: session_new: init
65. debug1: session_new: session 0
66. debug1: session_open: channel 0
67. debug1: session_open: session 0: link with channel 0
68. debug1: server_input_channel_open: confirm session
69. debug1: server_input_channel_req: channel 0 request subsystem reply 1
70. debug1: session_by_channel: session 0 channel 0
71. debug1: session_input_channel_req: session 0 req subsystem
72. subsystem request for sftp
debug1: subsystem: exec() /usr/sbin/sftp-server
```

### ***WinSSHD and SSH Communications – Feature Comparison***

In researching and testing the OpenSSH/Cygwin, Bitvise (WinSSHD), and SSH Communications SSH servers, a clear grouping emerged of command-line versus GUI configuration. All three integrate with Windows NT (and later). Most features are shared across all three packages (although highlighting the differences is one of the goals of this paper). Securely configuring and administering these SSH servers seems to be a significant factor that would prevent many Windows systems administrators from working with the less accessible OpenSSH version.

In the remaining discussion, selected screen shots from Bitvise and SSH Communications are grouped based on the SSH elements they configure. Note that the lack of a feature on a corresponding page doesn't mean that the feature is absent in that application. It may simply mean that they are grouped differently. Where relevant, comparisons are made with the configuration of OpenSSH elements.

## Network Configuration Options



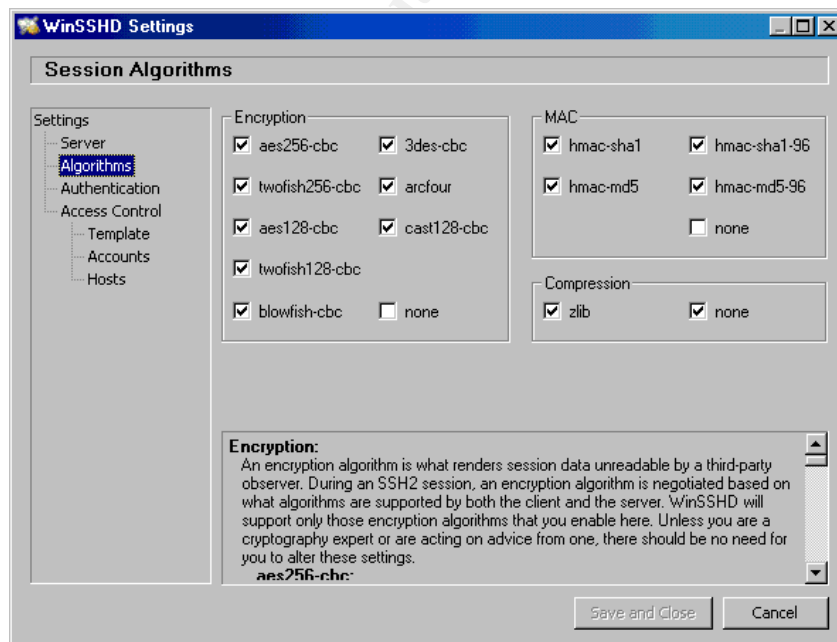
At first glance, it should already be apparent that SSH Communications (SSH.com) offers many more configuration options than Bitwise WinSSHD. (Of course, it is important to select an application based on the features required rather than the mere presence of more features). In these two pages, it is shown that both applications feature simple configuration of TCP/IP settings. For example, a multi-homed machine can be configured to listen only on the internal address. SSH.com and OpenSSH offer reverse DNS lookups for additional

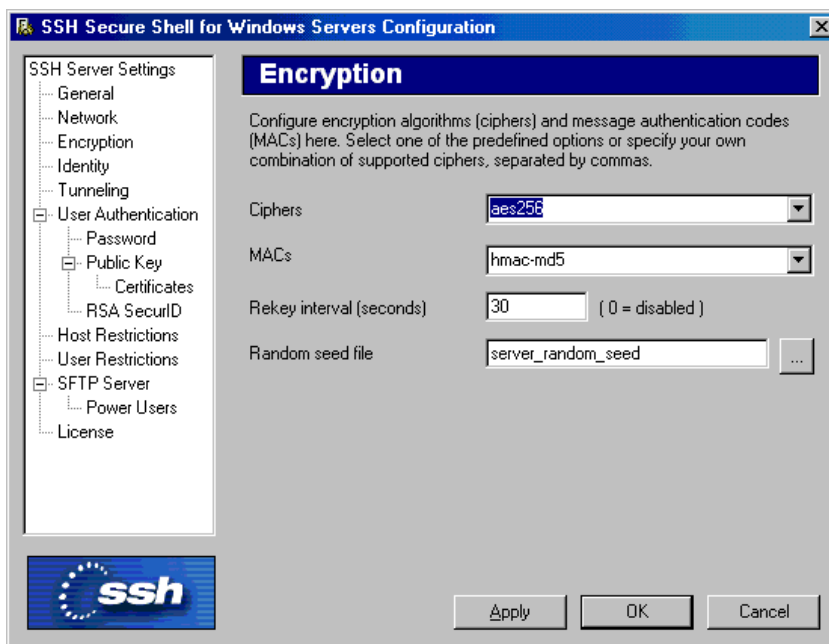
security. Although a workstation connecting to an SSH server is unlikely to possess a PTR record, this feature could be useful in a highly restricted host-based authentication scenario to reduce the likelihood of an IP Spoofing attack. (In such an attack, an unauthorized machine crafts an IP header that possesses an allowed source IP address. However, unless the would-be intruder is also able to change the reverse DNS (PTR) record on the server(s) which hosts that IP address block, such a connection would be rejected (assuming the reverse DNS feature was enabled)).

The "re-key interval" is another important feature (on Bitvise above and on SSH.com 2 screen shots below). If a would-be intruder is sniffing packets passing between an SSH client and server, it is possible that a session key could be captured and used for a man-in-the-middle, session-hijacking attack. The re-key interval generates new keys between client and server at the configured interval to prevent the likelihood of such an attack. OpenSSH also offers key generation interval configuration.

Finally, both applications permit the configuration of variable logging levels. Here is a significant advantage of the Windows-based SSH servers over OpenSSH: OpenSSH, native to Unix, looks for a local syslog service. There are syslog servers for Windows, but this is an additional set of installations, items to integrate, and logs to check (your SSH syslog and your NT event viewer). All this adds to the application's Total Cost of Ownership (TCO), which makes the OpenSSH option not so "free". Recall that in a secure environment, "Prevention is Ideal, but Detection is a Must!" [2].

### *Encryption and MAC Algorithms*





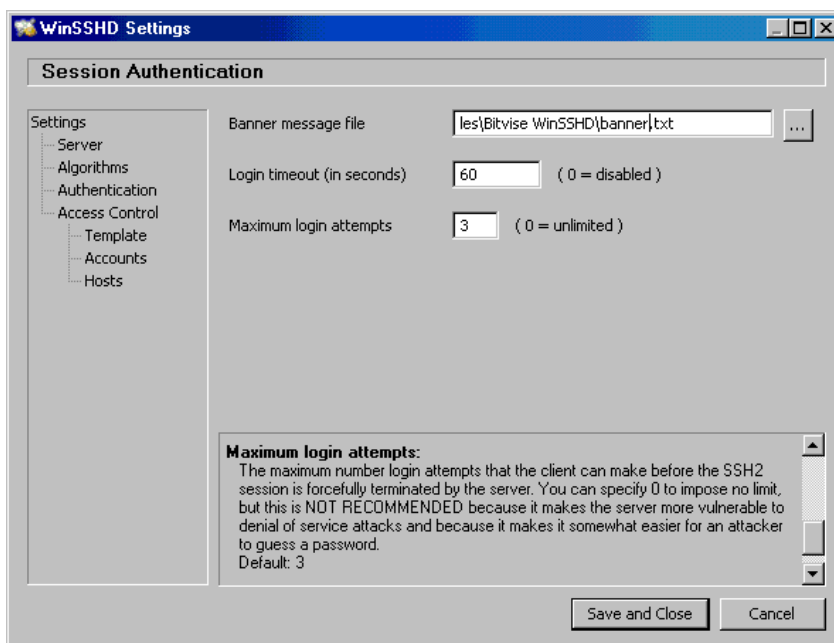
Both servers possess the ability to configure which encryption and message-hashing algorithms (Message Authentication Code) are used. (Incidentally, the Windows-based SSH servers are exclusively SSH2, proven significantly more secure than SSH1. OpenSSH supports both SSH versions, and supports all standard ciphers). SSH.com supports a wider range of encryption algorithms, including the relatively insecure DES algorithm) for backward compatibility. WinSSHD does not support DES (which is probably just as well).

Upon installation, both servers create randomness files from which host-keys are generated. SSH.com then creates a 2048-bit DSA host key (the bit length, key type, and other options can be manually configured. WinSSHD creates a 1024-bit DSA host key, but gives no options for configuring bit-length or key-type. In case of a security breach, the key pair can be manually regenerated. The OpenSSH package also offers fully configurable key generation commands.

SSH.com does not offer any compression options (for added performance over slow connections), while WinSSHD offers optional zlib compression. Compression is also a configurable option in the OpenSSH package.



## Logon Settings

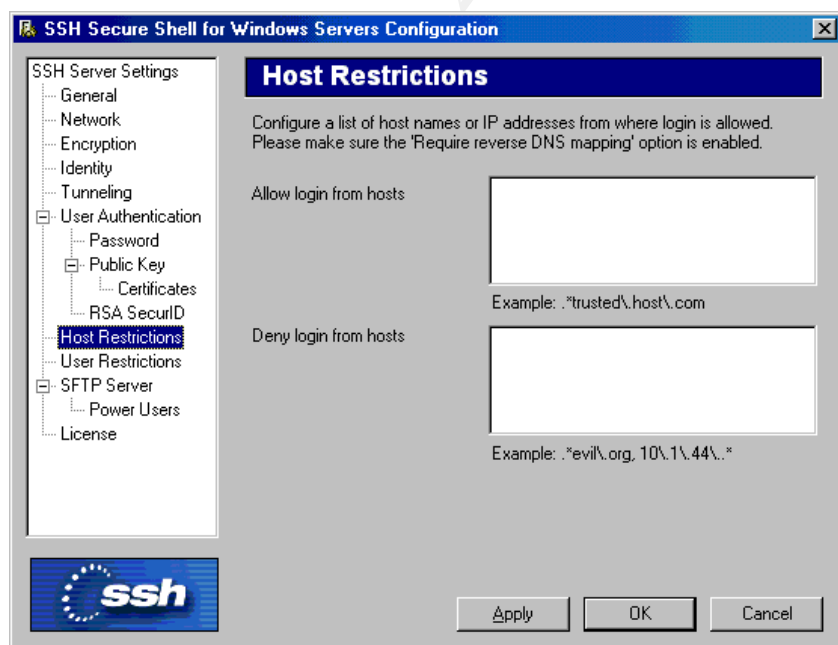
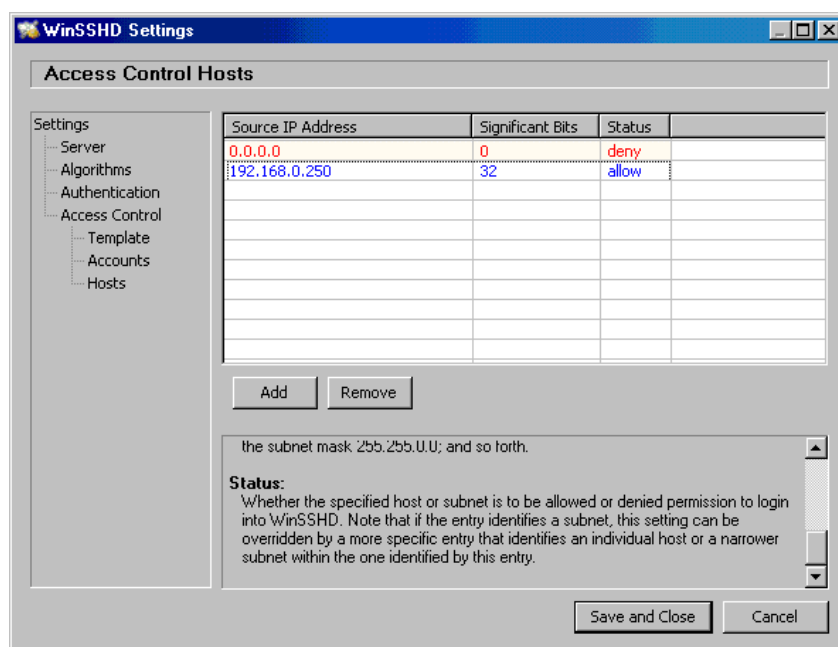


Login settings are another critical security feature. Configuring a "maximum number of login retries before disconnecting" discourages brute force attacks. The Windows-based SSH servers make this setting easy to configure. OpenSSH does not include this setting in the `sshd_config` file. This parameter would normally be configured at the os level in a Unix system. However, when attempting to connect to OpenSSH via PuTTY, the session disconnected after 2 retries. When connecting via the SSH.com client, the session disconnected after

3 tries (7 tries the first attempt, 3 thereafter). Also, the OpenSSH version of the SSH client disconnected after 3 tries. Not too shabby for a default.

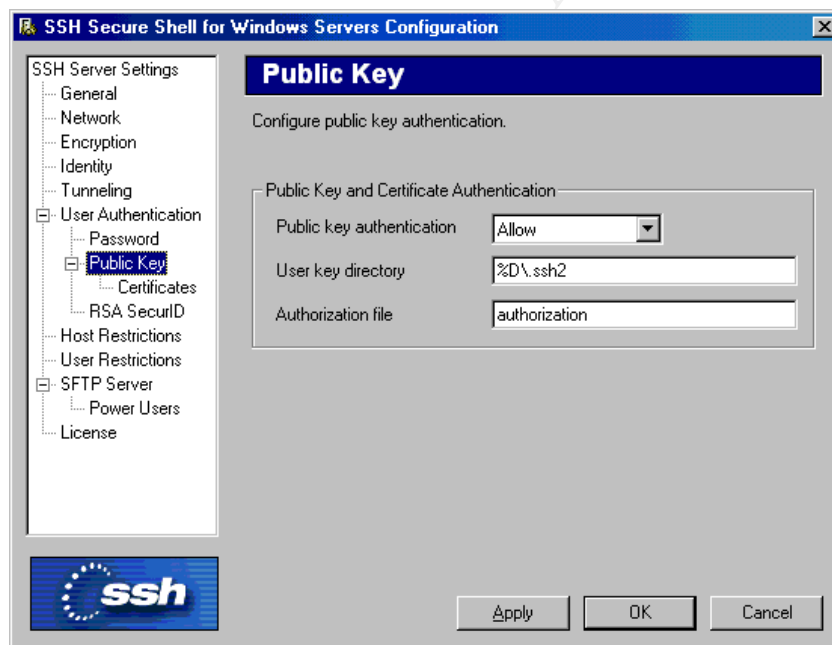
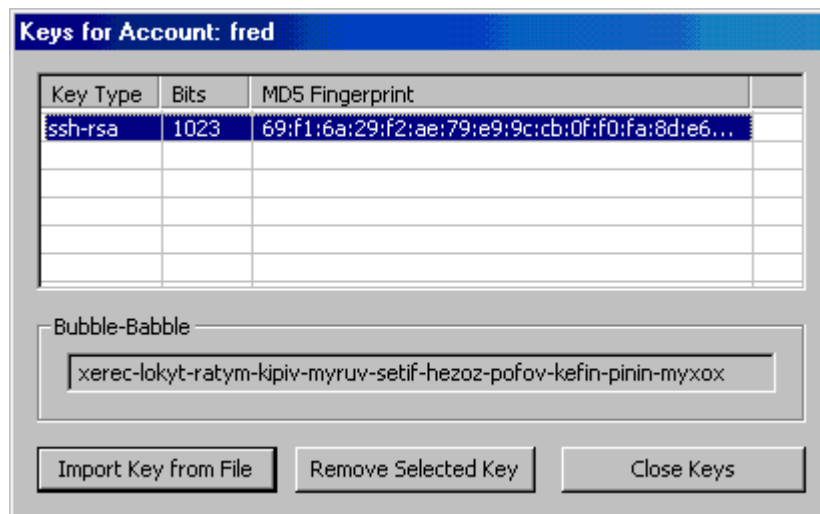
All three versions allow easy configuration of a banner message which displays upon successful login. In some areas, such a message is a legal requirement for subsequent prosecution of an illegal trespass.

### Host-Based Authentication



All three versions allow host-based authentication. In SSH2, this type of authentication is in addition to the primary authentication method (such as password or public key). Thus, even though by itself, such authentication is quite vulnerable to IP Spoofing attacks, when combined with other methods, it creates a stronger authentication process. In OpenSSH configuration, host-based authentication requires an additional file, /etc/hosts.equiv.

### Public Key Authentication

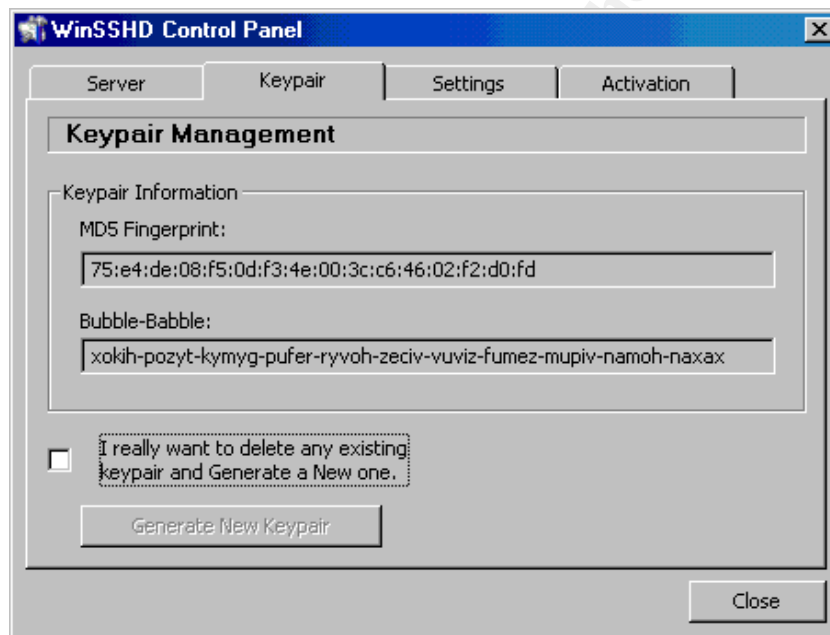


All three SSH servers allow the use of public key authentication. Of the three, WinSSHD allows the simplest way of configuring this. Simply deposit the client's public key (must be in SSH2 format, either DSA or RSA) in a secure location on the server and import it using the GUI interface. (Note that it is very important to

use a secure method to initially move the public key from the client machine to the server. A password-authenticated SSH/SFTP session can provide this). Each key is assigned to a specific user account that the key is checked against when that user logs in. The SSH.com server requires the manual creation of an authorization file that is stored in the client user's home directory. OpenSSH uses the same procedure. When logging on with PuTTY (and other clients), a profile may be created to log onto a specific SSH server with a specific username and a specific private key. This allows for complete automation of the login process. Please note that although such automation is essential for scheduled, unattended jobs, and a convenience for users who access the SSH server frequently, it is nevertheless a weakness in the overall security of the application. An unauthorized user who knows a username now has access under that user's security context. Therefore it is also imperative that the directory in which the `authorized_keys` file is stored is given adequate access restrictions to prevent unauthorized tampering.

### ***Differences between the Windows-based versions.***

#### *WinSSHD Features.*



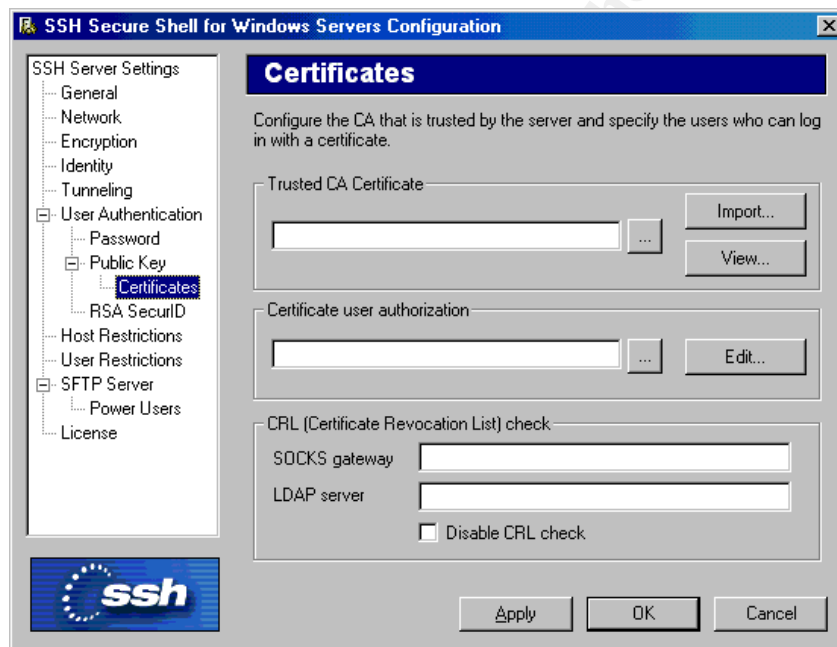
When an SSH client logs into an SSH server for the first time, the client alerts the user with an MD5 hash of the host's public key file (also called a fingerprint). At that time, a user at the client end can call the SSH server administrator and check the fingerprint stored by the SSH client against the server's fingerprint. Assuming all checks out, the user can agree to the warning and store the fingerprint. During subsequent connections, the SSH client checks the fingerprint against that server. If the server returns a different fingerprint, then the client is

alerted. The user can then verify this change with the SSH server administrator to see if the keys were regenerated. If the server administrator cannot verify this regeneration, this could be evidence of a man-in-the-middle attack. The MD5 fingerprint is the best insurance that the destination server is who it purports to be. This fingerprint-advertising feature is a standard feature of SSH. However, while the OpenSSH and SSH.com applications provide command line switches for displaying the server's fingerprint, WinSSHD displays this fingerprint in the GUI for immediate access to this critical piece of information.

Another useful feature of WinSSHD, especially for novice users, is the provision of a descriptive explanation of each feature in the lower pane of each configuration window.

Beyond this context-based help, WinSSHD offers very little documentation with the installation package (or the evaluation package at least) and little more on their web site. However, considering the simple install and the ease of configuration, this may not be a critical issue.

### *SSH Communications Features.*



The primary advantage offered by the SSH Communications version is tight integration with a Public Key Infrastructure (PKI). SSH.com has standalone add-on products for creating a new PKI or integrating with an established one. The screen shot above shows one of the configuration windows for trusting a Certificate Authority, installing client certificates, and identifying directory servers that contain databases of valid and restricted users based on their certificates.

PKI is a highly secure (when implemented correctly), enterprise-scale secure authentication system, based on centralized generation, management, and revocation of public keys which are stored in user or machine-based certificates. PKI interoperates with smart cards and tokens that actually store these certificates. A widely used subset of smart cards, RSA SecurID, is also supported in the SSH.com version. (Incidentally, such integration may also be possible with the OpenSSH version as an input option of keyboard-interactive authentication. However, this writer assumes that actually configuring this option would be somewhat difficult). RSA SecurID provides two factor authentication by requiring a consistent pin (factor 1: "what you know") along with a randomly generated numeric string that changes every 60 seconds (factor 2: "what you have").

SSH Communications offers excellent documentation both in the form of administrator and user manuals, and an FAQ database. SSH.com also offers telephone support contracts.

OpenSSH has no company or help desk to call. However there is ample online documentation accessible by searching on google.com or other search engines.

## **Summary**

Over the course of a quest to find out more about FTP and other methods of file transfer, this writer arrived at Secure Shell as a flexible, versatile, and secure application. Discovering proprietary SSH applications with GUI installers for Windows was an easy step in this process. However, remaining in front of the comfort of a graphic interface may not be the best way for anyone to learn the details and options in the configuration and operation of the software. The OpenSSH install and configuration process demanded an intense level of involvement and yielded the reward of a deeper and broader perspective. It seems useful to state this since implementing security must be such a dynamic process. Which ciphers and key types provide the strongest security? What are the vulnerabilities? What options can be used to secure SSH connections in different environments? These are all essential questions. Without closely examining the install and configuration of OpenSSH, this writer would not have been able to effectively evaluate the features of the Windows-based versions.

In addition to the features of SSH described in this paper, other features such as TCP tunneling (also known as port forwarding) deserve their own investigation. (There are several papers available in the SANS.org reading room that address this topic).

Once the basic setup and operation of SSH is grasped, file transfer is a simple process. For this reason, the paper focuses mainly on the operation of SSH. However, file transfer as a critical network task, is an extremely compelling

reason to implement SSH. Clearly, SSH can and ought to replace telnet in most situations. But with the increasing use of GUI-based remote administration, shell-access alone may not be sufficient reason to consider SSH. Of course, SSH, through TCP tunneling, can provide an encrypted tunnel for any TCP-port based application (including standard FTP!), but that is another story.

## ***Bibliography***

1. Postel, J. and Reynolds, J. "RFC 959 – File Transfer Protocol (FTP)". October 1985. URL: <http://www.ietf.org/rfc/rfc959.txt?number=959>. (28 March 2003).
2. Cole, E. "Security Essentials training". SANS. Washington, D.C. October 2002.
3. Ribak, J. "Active FTP vs. Passive FTP, a Definitive Explanation". August 2001. URL: <http://slacksite.com/other/ftp.html>. (28 March 2003).
4. Bellovin, S. "RFC 1579 - Firewall-Friendly FTP". February 1994. URL: <http://www.rfc-editor.org/rfc/rfc1579.txt> (28 March 2003).
5. Horowitz, M. and Lunt, S. "RFC 2228 –FTP Security Extensions". October 1997. URL: <http://www.rfc-editor.org/rfc/rfc2228.txt>. (28 March 2003).
6. Ford-Hutchinson, P., Carpenter, M., Hudson, T., Murray, E., Wiegand, V. "Securing FTP With TLS". April 2002 (revised). URL: [http://www.isaserver.org/articles/Securing\\_FTP\\_with\\_TLS.html](http://www.isaserver.org/articles/Securing_FTP_with_TLS.html) (28 March 2003).
7. Micke Pettersson. "SSL - Secure Sockets Layer ,TLS - Transport Layer Security". 13 June 1998. URL: [http://www3.tsl.uu.se/~micke/ssl\\_links.html](http://www3.tsl.uu.se/~micke/ssl_links.html). (28 March 2003).
8. CERT Coordination Center. "Anonymous FTP Configuration Guidelines." 1995. URL: [http://www.cert.org/tech\\_tips/anonymous\\_ftp\\_config.html](http://www.cert.org/tech_tips/anonymous_ftp_config.html). (28 March 2003).
9. Allman, M. and Ostermann, S. "RFC 2577 - FTP Security Considerations". 1999. URL: <http://www.rfc-editor.org/rfc/rfc2577.txt>. (28 March 2003).
10. Gromek, M. "Securing FTP Authentication". SANS Reading Room. February 2002. URL: [http://www.sans.org/rr/protocols/sec\\_ftp.php](http://www.sans.org/rr/protocols/sec_ftp.php) (28 March 2003).
11. Zwamborn, D. "An Introduction to SSH Secure Shell". SANS Reading Room. May 2001. URL: [http://www.sans.org/rr/encryption/intro\\_SSH.php](http://www.sans.org/rr/encryption/intro_SSH.php). (28 March 2003).
12. Author unknown. "OpenSSH: Project History and Credits". January 2002. URL: <http://www.openssh.com/history.html> (28 March 2003).
13. Friedl, M. et al. "SSHD\_Config, Man support page". 1999. URL: [http://www.nevis.columbia.edu/cgi-bin/man.sh?man=SSHD\\_config](http://www.nevis.columbia.edu/cgi-bin/man.sh?man=SSHD_config) (28 March 2003).

14. Johnson, M. "OpenSSH for Windows". Claremont McKenna College. 2003. URL: <http://lexa.mckenna.edu/SSHwindows/> (28 March 2003).

Other References and Links:

- Seeger. "FTP Related RFCs (Request For Comments)". ProFTPD project. 15 March 2001. <http://proftpd.linux.co.uk/docs/rfc.html> (28 March 2003).

**Appendix A: Links for SSH/SFTP Servers and Clients on Windows.**

*Windows Servers:*

- OpenSSH for Windows v3.5p1-3— a windows executable which performs a simple install of cygwin and cygwin-ported OpenSSH/SFTP files. Maintained by Michael Johnson at Claremont McKenna College. Includes daemons and clients. Minimal post install configuration, with good documentation. Free. <http://lexa.mckenna.edu/SSHwindows/>
- Cygwin – the prerequisite for running OpenSSH packages. <http://www.cygwin.com/>. There are many university site offering detailing install instructions including the following University of Utah site (<http://www.pcmanagers.utah.edu/software/setupSSHD.cfm>). The cygwin install can be a bear, and getting all the necessary packages with compatible versions can be a frustrating experience. Setting up SSHD is fairly straightforward but the SFTPD has it's own set of problems. Free.
- WinSSHD – an SSH/SFTP server designed for Windows servers. Simple install, easy to use and configure. Evaluation available. Not too pricey. <http://www.bitvise.com/WinSSHD.html>.
- SSH.com - SSH for Windows Servers. Tatu Ylonen's original open source work became SSH Communications. Simple install. All the configuration options and features that you might need. Works great. Pricey. <http://www.ssh.com/support/downloads/secureshellwinserver>
- Windows Access Server (includes SSH/SFTP servers). Not evaluated. <http://www.foxitsoft.com/wac/wac.zip>.
- F-Secure SSH server (includes SFTP server). The other major enterprise-level, proprietary SSH implementation. Not evaluated. <https://europe.f-secure.com/download-purchase/download-forms/sshserverwin.shtml>
- VShell (SSH/SFTP server). Van Dyke Software. Not evaluated. <http://www.vandyke.com/download/vshell/index.html>

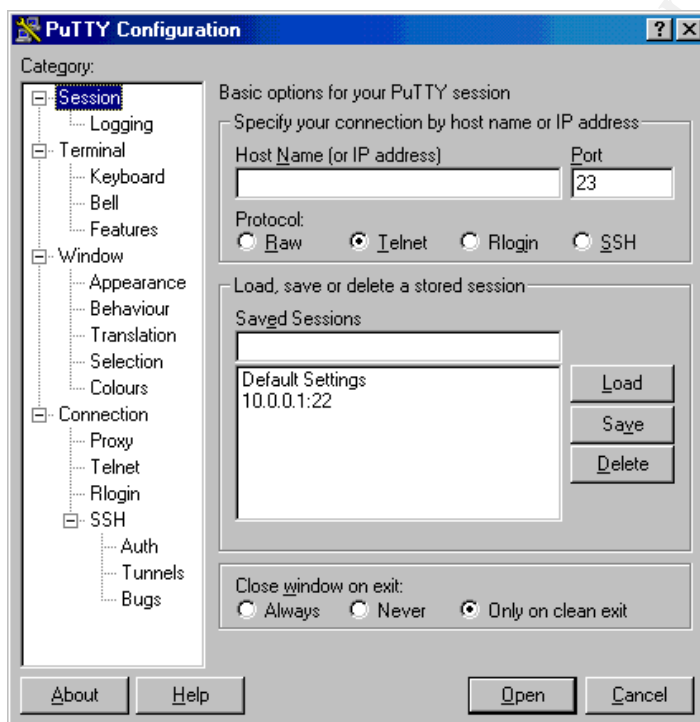


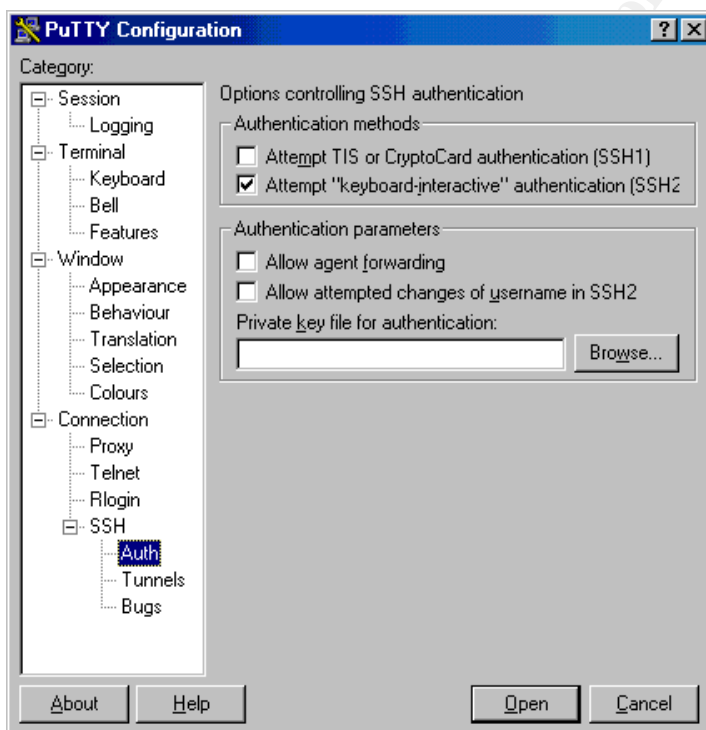
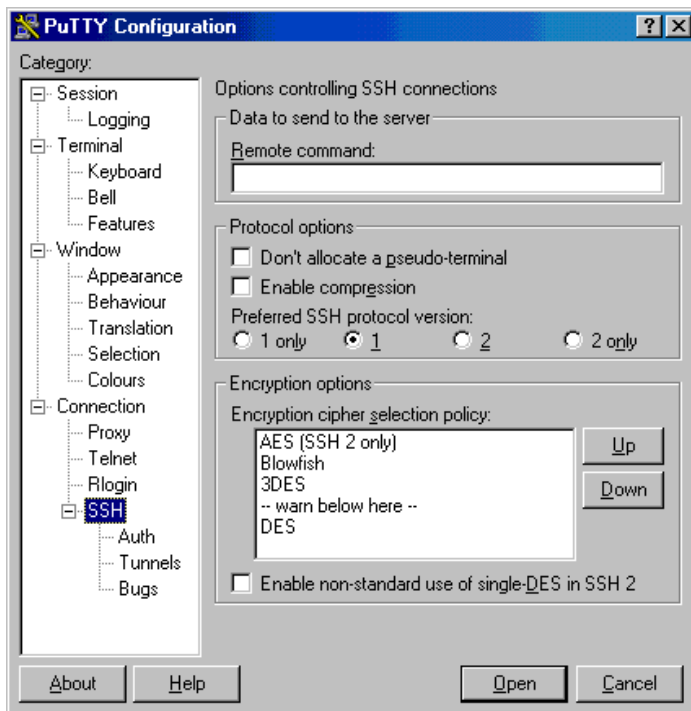
## Windows Clients:

- PuTTY/PSFTP – maintained by Simon Tatham. Command-line, Freeware. <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html> (Putty.zip has everything!).
- SecureFX from Van Dyke Technologies, GUI. Evaluation available. [www.vandyke.com](http://www.vandyke.com).
- Secure Server/Client Evaluation Page at UPENN. A good resource, perhaps a bit outdated. <http://www.upenn.edu/computing/group/secure/2000/phase1/matrix.html>

## Appendix B: PuTTY Client Configuration Screen Shots.

The following screen shots illustrate the GUI configuration environment for PuTTY. Connection profiles – which can also be utilized with PSFTP connections -- specify destination servers, encryption and hash ciphers, terminal environment, and authentication options.





The following screen shot shows the display of the puttygen.exe program. Upon generating the public/private key pair, the user is prompted to move the mouse within an area of gray space to generate randomness for the seed file which is then used as the source of the private key. The public key is displayed in the upper window and is used for creating an "authorized\_keys" file to be stored on

the server for public key authentication. Note that keys can be generated for SSH1 or SSH2 in both DSA and RSA formats with variable bit lengths.

