15 Minute Guide to SSH Security

johnny@ihackstuff.com

Many people rely on secure shell (SSH) as a method of securing terminal connections between hosts. I personally use SSH to connect to my servers because of the simple fact that my username, password and my entire session are encrypted, safe from prying eyes.

> "Secure Shell (SSH) is a program to log into another computer over a network, to execute commands in a remote machine, and to move files from one machine to another. It provides strong authentication and secure communications over unsecure channels. It is intended as a replacement for rlogin, rsh, and rcp. "http://www.kleber.net/ssh/ssh-faq-1.html#ss1.1

Understanding the problem

When you connect to an SSH server for the first time, the server will send you it's public key. Although this will present itself in various ways depending on the version of the SSH client used, a message similar to the one displayed by putty will be displayed:

Figure 1: PuTTY security alert

This message indicates that the SSH client has not cached the fingerprint of this server's key. According to the SSH client, you are essentially connecting to a "new" host. If you are at all interested in security, you should only cache the fingerprint of each server *once you've manually validated that fingerprint*. I will discuss manual validation later. It is important to understand that simply put, a key fingerprint confirms the identity of your SSH server. If you do not cache, or record the fingerprint, your SSH client has no way of knowing if the server you are connecting to is the same server it connected to the last time.

Let's assume for a moment that you have already validated and cached the key fingerprint for a server. If the key fingerprint for that server ever changes, the SSH client will alert you with a message similar to the one shown in Figure 2.

Figure 2: PuTTY detecting a changed key fingerprint

As an SSH user interested in conducting a secure transaction with an SSH server, you should *never* simply click anything but 'cancel' to any of the messages listed above, regardless of the SSH client you are using. Validating the fingerprint of your SSH server is of utmost importance. Why? Because if you don't you may be choosing to trust a malicious user with your SSH username and password!

Addressing the problem – Manual fingerprint validation

Before agreeing to cache (and therefore *trust*) a fingerprint, you should first verify that the fingerprint is valid. The procedure for validating an SSH key fingerprint is fairly straightforward, but it will require *local* access to the SSH server. Although this may seem awkward in this age of networking, this is a critical element in maintaining absolute trust in your SSH connection. The good news is that this procedure does not need to be performed frequently and that there are some halfway options that can be exercised, including running a cron job to post the fingerprint on an SSL-authenticated web page every so often. Understand however, that convenience options tend to cause security weaknesses. Don't make it easier for an attacker when trying to make your own life easier.

Verifying the fingerprint of a Linux-based SSH server

1. Log onto the server using a local connection, such as a console.
2. Determine the location of your server's public SSH keys. One common location is in the /etc/ssh/ directory.
3. Run ssh-keygen with the –l and –f options against your server's public key. The command line may read

```
ssh-keygen -l -f /etc/ssh/ssh_host_key.pub
```

4. This is the server's host key fingerprint. It may look like this:

```
1024 59:c7:83:9e:57:97:61:47:2f:04:da:50:98:b1:3e:85
```

This output should match the fingerprint that your SSH client produced when you first connected. If it doesn't match, you've got issues! Let's take a look at some of the reasons the key fingerprints may not match.

Problem	Resolution
You inadvertently selected 'No' or 'Cancel' the last time you were prompted with this message.	If this is the case, you did not cache the key fingerprint. Since the SSH client has no record of this fingerprint, it has alerted you. After manually verifying the fingerprint, choose 'Yes' to cache it.
You updated your SSH client or switched to a different client since you last connected to your SSH server.	Again, it is possible that you client has no record of this fingerprint if you overwrote its fingerprint cache. After manually verifying the fingerprint, choose 'Yes' to cache it.
The SSH session is using a different protocol version than was previously used.	SSH protocol 1 and 2 use different key files. Make sure you are manually fingerprinting the correct key file. Try manually fingerprinting all the `*pub` files in the server's /etc/ssh directory.
You are being spoofed by a malicious user or program... an interloper!	See below for more clues as to what may be happening. Do not connect to the server until you know you aren't being duped. Always 'cancel' the connection!

Detecting an interloper

If you absolutely positively can not get your key fingerprints to match and you're sure you're checking the right server and keys, there's a chance you're the victim of an SSH man-in-the-middle (MITM) attack. This attack subverts the normally secure SSH login process and can lead to the compromise of your SSH session, username and password. Securiteam has a great write-up on this vulnerability entitled "SSH Protocol Weakness Vulnerability (MITM)." This write-up can be found at http://www.securiteam.com/securitynews/5BP0M157PG.html.

In summary, this vulnerability allows an attacker *sitting on your network* to masquerade as your SSH server via packet and ARP spoofing. When this happens, there is a good chance that you can detect the attack by using a network sniffer like ethereal found at www.ethereal.com. In order to understand how SSH normally behaves, let's take a look at a normal SSH client connect. For purposes of this exercise I will be using the PUTTY SSH client available from http://www.chiark.greenend.org.uk/~sgtatham/putty/.

I will be using the following network configuration:

Description	IP	MAC
SSH Client	10.1.1.111	00-02-B3-1B-CA-34
SSH Client Default Gateway	10.1.1.1	00-40-10-13-b5-0e
SSH Server	216.133.72.171	not important
Attacker	not important	00-02-B3-33-B1-B7

To gather this information:

- Run **`ifconfig -a`** in a UNIX environment or **`ipconfig /all`** in Windows to determine the IP and MAC address of your SSH client (See Figure 3).
- Ping your default gateway and run **`arp -a`** to determine the MAC address of your default gateway (See Figure 4).
- Record the IP address of your SSH Server for reference.

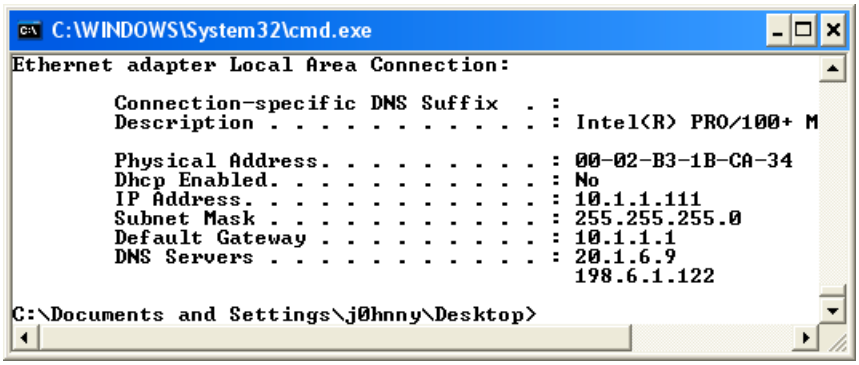

Figure 3: ipconfig /all output (Windows)

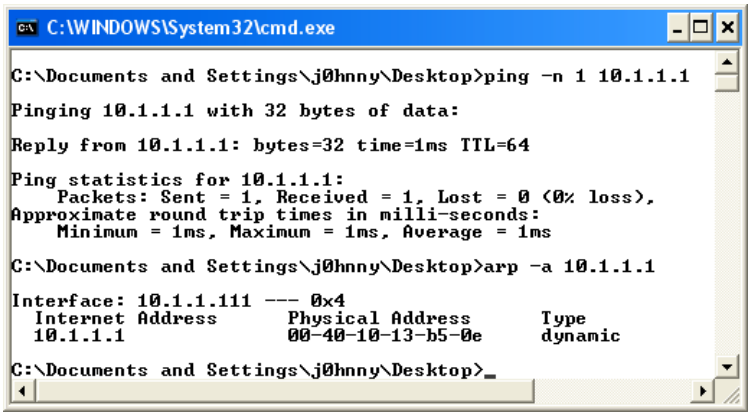

Figure 4: Determining the MAC address of the gateway with ping and arp

Once this information has been gathered:

1. Launch ethereal and begin capturing packets in promiscuous mode.
2. Lauch your SSH client (putty for this example) and wait for the PuTTY security alert as shown in Figure 1 or Figure 2.
3. Stop the Ethereal capture to view the network traffic.

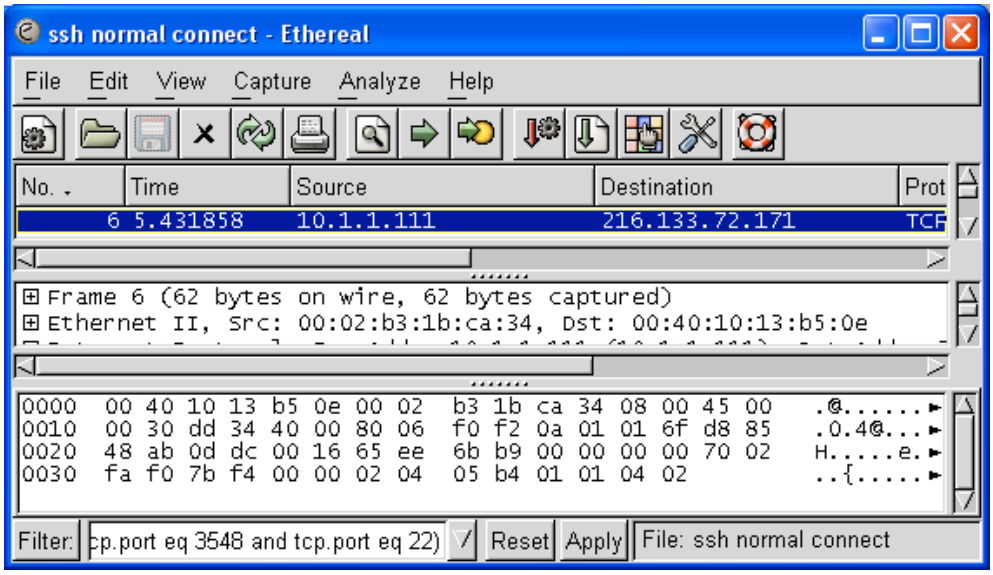

Figure 5: Ethereal capture of normal SSH session

The capture used in Figure 5 shows a filtered capture. Although it is not necessary to filter the capture, it can help to isolate an interesting connection. There are several ways to isolate the SSH connection in ethereal. One way to apply an ethereal filter is to enter a string in the filter field and click 'Apply' as shown in Figure 6. Another method requires selecting an SSH packet (TCP port 22) by left clicking the packet in the top window, then right-clicking the packet and selecting "Follow TCP Stream." as shown in Figure 7.

Figure 6: Ethereal filter input

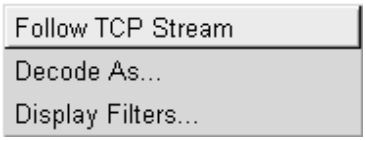

Figure 7: Ethereal 'Follow TCP Stream' option

Once the filter is applied, and the proper TCP stream is selected, select the first packet of the SSH stream (TCP port 22, your IP as the Source, your SSH server as the destination) in the top window and select it by left-clicking the line. Turning your attention to the middle window in Figure 5, notice that `10.1.1.111` (at MAC address `00:02:B3:1B:CA:34`) is initiating a connection with `216.133.72.171` (at MAC address `00:40:10:13:B5:0E`) and that these MAC addresses correspond with our SSH client and the default gateway respectively. When following through this TCP stream, you will notice that these two MAC addresses exchange packets back and forth

for the duration of this TCP connection. You are having an SSH conversation with your SSH server through your default gateway. This is normal behavior.

If an attacker on your network is attempting a MITM attack against you, a network capture will look very different as shown in Figure 8. Notice that now IP address `10.1.1.111` (at MAC address `00:02:B3:1B:CA:34`) is initiating a conversation with `216.133.72.171` (at MAC address `00:02:B3:33:B1:B7`!) The MAC address of the default gateway has changed! We are now sending packets to a different MAC address!

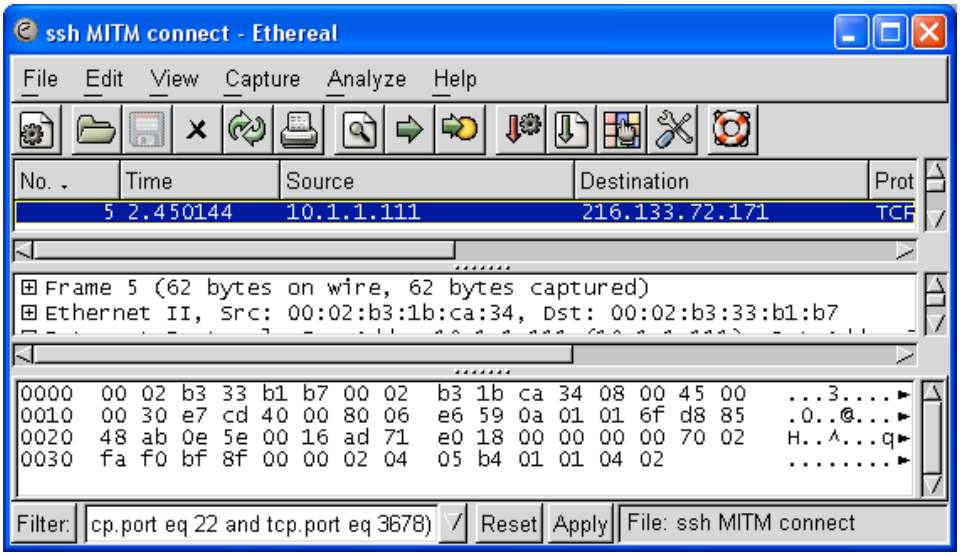

Figure 8: MITM attack capture

This network shenanigans, combined with the fact that our SSH server's host key fingerprint has changed since the last time we used it means that we are being spoofed by a malicious user. *If you do anything but 'cancel' your SSH connection, that malicious user will have your SSH username and password*.

The magic: How the bad guys do it

The malicious user, sitting *on the same network* as the SSH client victim, fires up a program like ettercap as shown in Figure 9.

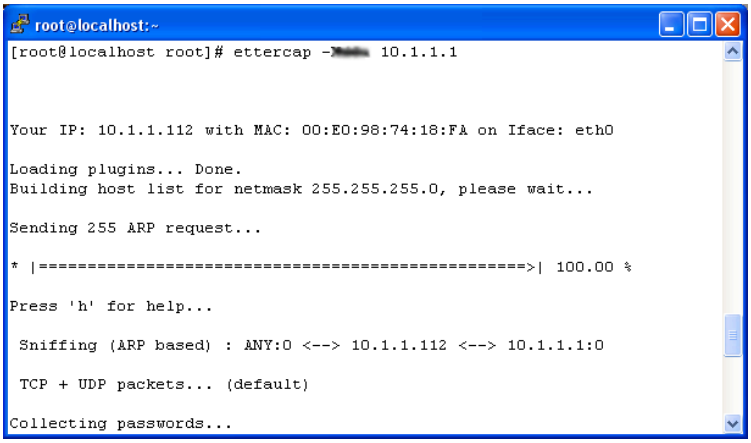

Figure 9: Ettercap launched

Using options designed specifically for the purpose of launching a MITM attack, ettercap first launches ARP queries against every device on the subnet, in this case the local class C, 255 addresses. Once ettercap has mapped the MAC address of every device on the subnet, it proceeds to inform every device on the network (via ARP replies) that the new MAC address to talk to is `00:0E:98:74:18:FA`, the MAC of the attacker. When a machine is updated with a new MAC address for an IP, it sends each packet destined for that IP to the new MAC address, effectively fooling those machines into sending the attacker packets that should be headed elsewhere. (See Figure 10) Eventually, ettercap re-sends these ARP replies, effectively updating the bogus MAC address on the other machines. In this way, the fooled machines continue to communicate with the attacker instead other machines, or the default gateway. This is called a man-in-the-middle and is the basic security flaw ettercap requires to perform it's magic.

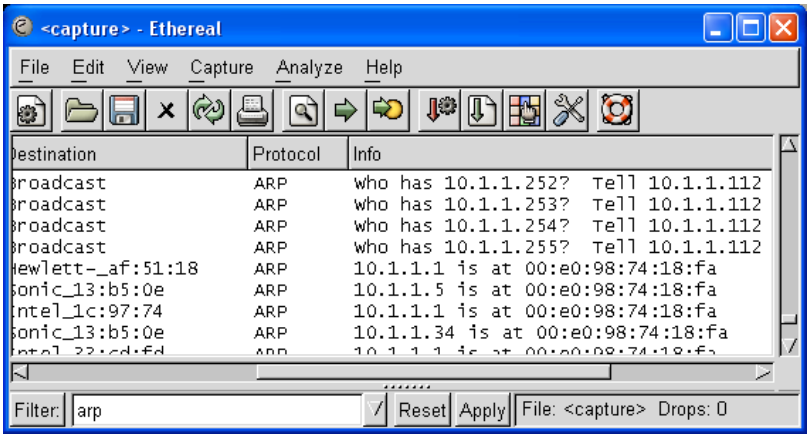

Figure 10: ettercap scanning and spoofing

Once the network is mapped and the hosts on the network have been fed false ARP information, ettercap enters a collection mode, waiting to capture passwords. If a client on the same network as the attacker launches an SSH client to connect to the outside world, ettercap spoofs a reply and a bogus key fingerprint to the SSH client, resulting in a warning like the ones shown in Figure 1 and Figure 2. If the SSH user selects anything

but 'Cancel' (a *very, very bad thing to do!*) ettercap will successfully man-in-the-middle the connection with a bogus key fingerprint and display the password as show in Figure 11.

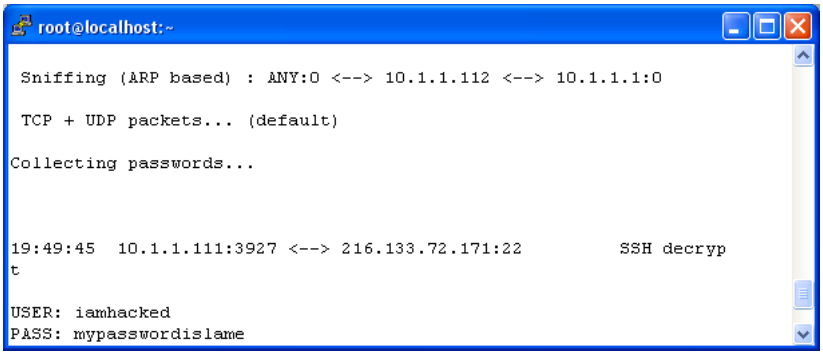

Figure 11: ettercap successful password capture

Because the MITM attack is so effective, ettercap can capture other secured usernames and passwords such as those used to access secure web pages via SSL, the standard for web security employed just about everywhere. We'll take a look at how ettercap works against SSL in a future paper. However, SSH and SSL aren't the only protocols that can be observed with ettercap. According to the ettercap man page, the tool can also sniff usernames and passwords from "TELNET, FTP, POP, RLOGIN, SSH1, ICQ, SMB, MySQL, HTTP, NNTP, X11, NAPSTER, IRC, RIP, BGP, SOCKS 5, IMAP 4, VNC, LDAP, NFS, SNMP, HALF LIFE, QUAKE 3, MSN, YMSG (other protocols coming soon...)" In addition, ettercap has many other features. If you're concerned about the risk this tool poses, I suggest you download it to learn about it's extensive list of features.

Diving into the weeds (techies rejoice!)

If you're into networking, you may have already figured out that the main security weakness here (ARP spoofing) can be taken care of with static ARP entries on the client machine. Static ARP entries are not overwritten by unsolicited ARP replies. Looking at the ARP table shown in Figure 4, the ARP entry that is being abused is marked as 'dynamic.' We can update this by adding a static ARP entry for our default gateway with the following command:

```
arp -s 10.1.1.1 00-40-10-13-b5-0e
```

When this command is run, the ARP table is updated with the address specified as shown in Figure 12. Ettercap can no longer fake this entry in our ARP table.

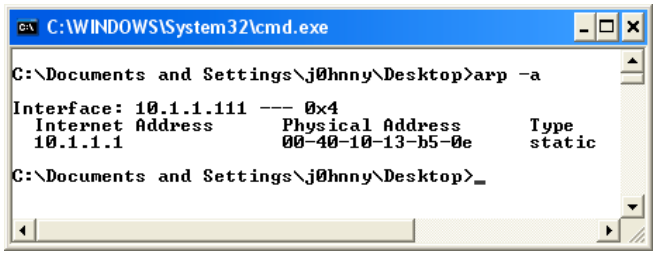

Figure 12: static arp entry

The question is, does this solve the problem? The answer is no. On the application side, we quickly discover that our SSH host key fingerprint is still wrong. When we sniff our SSH connection, we get something like what is shown in Figure 13.

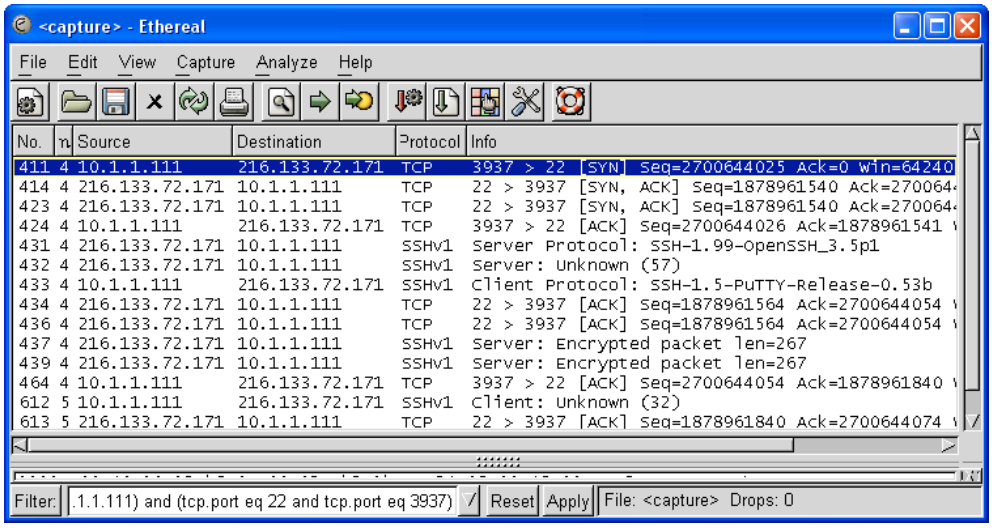

Figure 13: ethereal capture with static client ARP entry

This odd capture seems to show that each response from our SSH server is duplicated. What is not readily apparent is why this is happening. Looking closer at the capture we discover that the replies are not duplicated, but that they are subtly different. Focusing on the SYN|ACK replies, we will see that the first SYN|ACK comes from MAC `00:40:10:13:B5:0E` (the gateway) and is sent to MAC `00:E0:98:74:18:FA` (the ettercap attacker). The second SYN|ACK comes from MAC `00:E0:98:74:18:FA` (the ettercap attacker) and is sent to MAC `00:02:B3:1B:CA:34` (the SSH client).

In essence, the session happens like this:

```
GATEWAY ==> ETTERCAP ==> CLIENT
```

Ettercap is acting as the man-in-the-middle. What ettercap can do with this packet before sending it on to the client is up to ettercap. Generally speaking, though, ettercap will do bad, bad things with the packet before passing it on. The entire conversation is documented below. For the sake of brevity, some ACK and such have been omitted.

First, the client starts the TCP handshake with a SYN packet to the gateway. Since the client has the real MAC of the gateway, the packet is sent to the real gateway's MAC address:

```
CLIENT [SYN] =========> GATEWAY
```

Next, the gateway replies to the SYN with a SYN|ACK. However, the gateway has the wrong MAC address for the client, thanks to ettercap. The reply goes to the ettercap attacker first, and then ettercap forwards the reply onto to client:

```
GATEWAY [SYN|ACK]======> ETTERCAP ======> CLIENT
```

Any packet the client sends will always go to the real gateway, like this final ACK in the TCP handshake:

```
CLIENT  [ACK]==========> GATEWAY
```

The SSH server will reply with the server's version of SSH. This reply, like all others, is first delivered to the ettercap attacker then on to the client:

```
GATEWAY [SSH SERVER VERSION] ===> ETTERCAP ===> CLIENT
```

The client then sends the server information about the SSH client software and capabilities:

```
CLIENT [SSH CLIENT INFO] ===> GATEWAY
```

Last, but not least, the final blow comes in the form of a spoofed SSH host key fingerprint from ettercap. Ettercap snagged the real fingerprint in route and changed it to a new fingerprint. This is necessary so ettercap can continue to read the SSH traffic when it's encrypted:

```
GATEWAY [SSH KEY PRINT] ===> ETTERCAP ===> CLIENT
```

At this point, if the client accepts the bogus key, any future communications with the 'SSH server' will really occur through ettercap, which can read the encrypted data. In order to keep a real SSH session going, ettercap passes all the data through to the SSH server using the *real* encrypted SSH channel which was negotiated with the *real* SSH host key fingerprint, the username and the password.

Fixing the problem

There are several things that can be done to avoid becoming a victim to this type of attack.

1. Heed the warning of new or different key fingerprints. Cancel the connection (do not continue the connection with 'yes' or 'no' selections in your client).
2. Keep an eye on your ARP table wither manually with the 'arp' command or with a watchdog program. The Linux and Mac OSX kernels have the ability to log these malicious ARP requests. A cron job or something similar can be used to monitor these kernel messages.

3. Static ARP everything! This may not be practical, but if you have static ARP tables on both your client and gateway, you can't be spoofed from the local network. Bear in mind that your remote network may be another story. Read on...
4. Always manually validate your SSH host key fingerprints at the server. No amount of spoofing is going to fool a pure-and-simple manual validation.
5. Keep an eye out for "Gatekeeper," soon to be released through my site. This program will serve as a watchdog for your windows machine, alerting you when your gateways' MAC gets hosed.