Global Information Assurance Certification Paper

An Ettercap Primer

Duane Norton
GIAC Security Essentials Certification
Practical Assignment Version 1.4b
Option 1
April 14, 2004

Abstract

Ettercap is an open-source tool written by Alberto Ornaghi and Marco Valleri (a.k.a. ALoR and NaGA). *Ettercap* is described by its authors as "a multipurpose sniffer/interceptor/logger for switched LANs [1]." Since it incorporates a variety of features necessary for working in switched environments, *ettercap* has evolved into a powerful tool that allows the user to launch several different types of man-in-the-middle attacks. In addition, *ettercap* makes available many separate classic attacks and reconnaissance techniques within its interface.

The versatility of *ettercap* is a double-edged sword. It is easy to label this utility as a hacker tool for script kiddies, and it certainly can be used as such. However, because *ettercap* includes such a broad spectrum of attack and reconnaissance functions, it may also be used to teach LAN hacking techniques to students of network security. As such, the purpose of this paper is to raise awareness of the flexibility of *ettercap's* features, to demonstrate several of its specific capabilities, and to offer defensive strategies. While there are countermeasures that may be implemented to prevent successful *ettercap* attacks, many LANs remain all too vulnerable.

Introduction

Ettercap is a versatile network manipulation tool. It uses its ability to easily perform man-in-the-middle (MITM) attacks in a switched LAN environment as the launch pad for many of its other functions. Once *ettercap* has inserted itself in the middle of a switched connection, it can capture and examine all communication between the two victim hosts, and subsequently take advantage of these other features:

- **Character injection:** Insert arbitrary characters into a live connection in either direction, emulating commands sent from the client or replies sent by the server
- **Packet filtering:** Automatically filter the TCP or UDP payload of packets in a live connection by searching for an arbitrary ASCII or hexadecimal string, and replacing it with your own string, or simply dropping the filtered packet.

- **Automatic password collection for many common network protocols**: The *Active Dissector* component automatically recognizes and extracts pertinent information from many protocols including TELNET, FTP, POP3, RLOGIN, SSH1, ICQ, SMB, MySQL, HTTP, NNTP, X11, NAPSTER, IRC, RIP, BGP, SOCKS 5, IMAP 4, VNC, LDAP, NFS, and SNMP
- **SSH1 support:** Capture username, password, and the data of an SSH1 connection
- **HTTPS support:** Insertion into an HTTP SSL session, as long as a false certificate is accepted by the user
- **PPTP suite:** Perform man-in-the-middle attacks against PPTP tunnels
- **Kill any connection**: View and kill arbitrary active connections [1]

It also has many useful reconnaissance tools built in, to ensure that an attacker can stealthily gain awareness of the LAN topology before launching MITM attacks:

- **Active OS fingerprinting:** Directly probe a LAN host to identify its operating system, using the *nmap* database [2]
- **Passive LAN scanning:** By listening to and analyzing passing frames, collect information about LAN hosts such as the operating system, open ports, running services, and IP and MAC addresses
- **IP and MAC-based sniffing:** Listen to LAN traffic in promiscuous mode and capture passing traffic. This feature is similar to common packet capture utilities, such as *tcpdump*, and allows filtering by IP or MAC address.
- **Search for other ARP poisoners and promiscuous mode NICs:** Detect other systems that are currently sniffing on the LAN, or performing ARP cache poisoning attacks.
- **Packet forge:** Construct and send custom Ethernet frames and IP packets to test the responses of network devices. This function has features similar to the tool *hping2* [3], and may be used to manually set header flags and spoof IP and MAC address [1].

Overview of Plugins

Ettercap is also extensible; the developers wrote support for plugins so that anyone can add new functionality, such as support for a new protocol dissector. The *ettercap* distribution includes a library of these plugins. The naming convention for these plugins (and for *ettercap* itself) is based on the names of monsters from the role-playing game Dungeons and Dragons.

There are two types of plugins, which can be differentiated by their names. Hooking plugins are named with the prefix *Hxx_* (e.g. *H09_roper*). These plugins are designed to accept sniffed data from a hijacked connection directly from the *ettercap* sniffing engine. In this way the plugins are said to be *hooked* into *ettercap*, communicating directly with the engine through a predefined application

programming interface (API). External plugins are named simply, e.g. *ooze*. These plugins are standalone features that do not expect data directly from the sniffing engine as input.

- **H00_lurker** – Search the LAN for other Ettercap poisoners.
- **H01_zaratan** – Broker/redirector for GRE tunnels
- **H02_troll** – ARP Reply spoof tool
- **H0*_hydra** – Suite of plugins to manipulate PPTP tunnels
- **H09_roper** – Blocks ISAKMP key exchange in IPSEC traffic
- **H10_phantom** – Sniff/Spoof DNS requests
- **H1*_giant** – Suite for SMB attacks
- **H20_dwarf** – Log all mail activity (e.g. POP, SMTP)
- **H30_thief** – Steal files from an HTTP stream
- **arpcop** – Report suspicious ARP activity
- **banshee** – Kill all connections between two hosts
- **basilisk** – Checks for successful ARP poisoning
- **beholder** – Find connections on a switched LAN
- **confusion** – Force a switch to send another host's data to your port
- **golem** – Denial of service attack
- **hunter** – Search for network interface cards that are in promiscuous mode
- **imp** – Collect Windows NetBIOS names from a host
- **lamia** – Manipulate Spanning Tree Protocol mappings on a switch
- **leech** – Isolate a host from the LAN
- **ooze** – Ping a host
- **phantom** – Sniff/Spoof DNS requests
- **shadow** – A simple SYN/TCP port scanner
- **spectre** – Flood the LAN with random MAC addresses
- **triton** – Try to discover the default gateway for the LAN [1]

In addition, the developers provide two dummy plugins, which have no function other than to serve as examples of the framework that programmers must use to write new *ettercap* plugins.

Ettercap Installation

Ettercap is freely available for download from *http://ettercap.sourceforge.net*. The most recent stable release is v0.6.b. *Ettercap* has been ported to many major UNIX variants, including Linux, FreeBSD, Solaris, and Mac OS X. There is also a version that runs on Windows 2000 and XP, although development definitely favors the UNIX platform for stability and new functionality.

In order to use the SSH1 and HTTPS sniffing features, *ettercap* requires that you install the OpenSSL libraries first, to allow support for Secure Sockets Layer (SSL) and Transport Layer Security (TLS) [1]. Many UNIX distributions include OpenSSL with their default installations, but the most recent OpenSSL libraries

are available for download from *http://www.openssl.org*. The latest stable version as of this writing is OpenSSL 0.9.7d.

Download the latest version of the *ettercap* source code from *http://ettercap.sourceforge.net/index.php?s=download*. After downloading the file *ettercap-0.6.b.tar.gz*, uncompress the file to an installation directory.

```
# tar xvzf ettercap-0.6.b.tar.gz
```

This creates the folder *ettercap-0.6.b*. Now install *ettercap* with all its plugins:

```
# cd ettercap-0.6.b
# ./configure
# make complete_install
```

Red Hat Linux - Kerberos Installation Errors

For my test installations of *ettercap*, I used Red Hat Linux 9.0 as the base operating system. I discovered that the installations repeatedly failed during compilation due to a missing Kerberos include file. The error was as follows:

```
# make complete_install
gcc -O2 -funroll-loops -fomit-frame-pointer -Wall -I. -
[Compilation output truncated]
In file included from /usr/include/openssl/ssl.h:179,
                 from src/ec_dissector_ssh.c:40:
/usr/include/openssl/kssl.h:72:18: krb5.h: No such file or
directory
```

A search of the user forums at *http://ettercap.sourceforge.net/forum/index.php* revealed that Red Hat Linux places the Kerberos include files in a different location than most other Linux distributions [4]. The missing *krb5.h* include file in Red Hat Linux 9.0 is located in */usr/kerberos/include*, but the default *ettercap* installation searches */usr/include/openssl* instead. To correct this, one must add the path */usr/kerberos/include* to the *Makefile.in* file, which *configure* uses to build the final *Makefile* for compilation:

1. Edit the file *ettercap-0.6.b/Makefile.in*
2. Find the COPTS (compiler options) variable
3. Add *–I/usr/kerberos/include* to the end of the COPTS line. This will tell *make* where to find the Kerberos include files.
4. Save *Makefile.in*

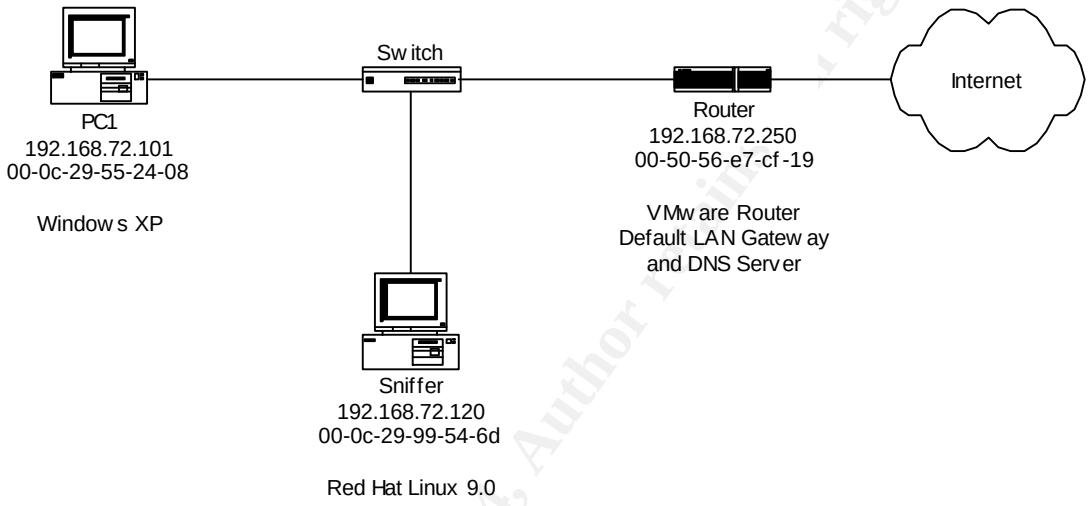
This change will cause *configure* to build the *Makefile* correctly for installation under Red Hat 9.0, and the compilation will now find the Kerberos files in the correct directory. We can now cleanly install *ettercap*, including all its plugins:

```
# ./configure
# make complete_install
```

```
[Compilation output truncated]
```

Example LAN Details

It is important to note that *ettercap* attacks can be very disruptive to a live production network, so it is imperative to experiment on an isolated test network. My network for this primer was configured using VMware Workstation 4.0, Windows XP, and Red Hat Linux 9.0. The following diagram shows the network topology:



IP and MAC Addresses

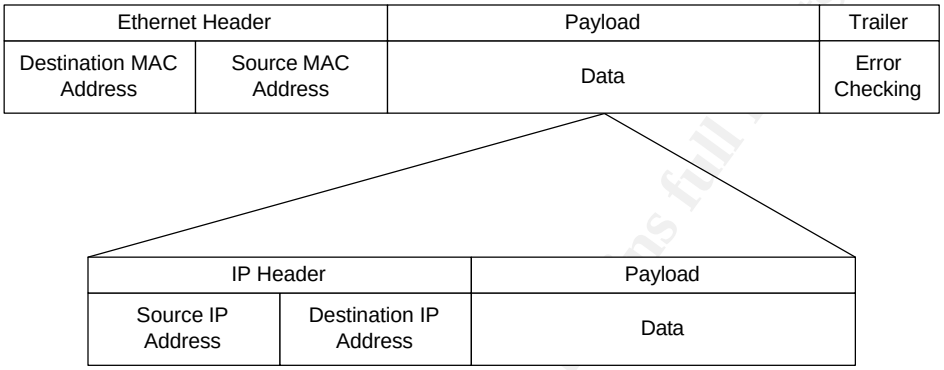
Applications use the IP protocol to communicate with each other. When a web browser sends HTTP requests to retrieve a web page from a distant server, it encodes each IP packet header with a source and destination IP address that allow the packets to be routed correctly to the web server [5].

However, within an Ethernet LAN segment, all communication between physical network interface cards (NICs) is sent using MAC addresses. A MAC (Media Access Control) address is the hardwired physical address of each network interface card, uniquely assigned by the manufacturer of the NIC. A MAC address is often represented as a 12-digit hexadecimal number, such as *00-0c-29-99-54-6d* [6]. NIC manufacturers encode the MAC address permanently onto each NIC in a ROM chip.

Individual MAC addresses are also called *unicast addresses* because they identify one particular NIC on the LAN. The system of MAC addresses provides the ability to send a frame to more than one MAC at a time, by sending to the broadcast address (*ff-ff-ff-ff-ff-ff*) instead of a unicast address. A broadcast frame

is sent to every device on the Ethernet LAN, and each NIC that receives a broadcast frame will accept and process it.

Each IP packet produced by an application is encapsulated inside an Ethernet frame, which is then labeled with the physical address of the destination NIC, and sent onto the wire. Here is a simplified diagram of the structure of an Ethernet frame encapsulating an IP packet [7].



Hubs and Switches

Ethernet networks join computers physically together using hubs or switches. A hub does not examine the Ethernet frames that pass through it. Hubs make no decisions based upon a frame header's contents: they simply forward every incoming frame out all ports, regardless of the destination MAC address. This makes it trivial to listen to all traffic on a hub-based LAN. Since every frame that traverses the LAN is sent to all ports on the LAN, sniffing only requires a NIC that is configured in promiscuous mode, listening for all passing traffic.

If the LAN uses switches instead of hubs, every frame is no longer automatically sent to every port. A switch increases LAN speed and reduces congestion by learning which MAC addresses are connected to its individual ports, and storing these mappings in a forwarding table. A switch extracts the source MAC address from passing frames, notes the port on which the frame arrived, and adds the entry to the table.

When an incoming frame arrives, the switch examines the frame's destination address and consults its forwarding table. If it hasn't yet learned which port hosts that MAC address, the switch will forward the incoming packet out all ports. However, if the forwarding table already contains a port for the MAC address, the frame will be sent only out that port. In addition, switches will forward frames with a destination MAC of *broadcast* out all ports.

This design makes sniffing a switched LAN more of a challenge, since the switch limits the frames that are sent out each port. A sniffer plugged into a switch port

will only be forwarded traffic that is either sent unicast directly to it, or is broadcast to the entire LAN. This is the reason for the common belief that switches offer some protection from sniffers. Since the traffic passing through the switch is selectively forwarded to only specific ports, a sniffer must use another method to actively intercept traffic.

The ARP Protocol

When a computer encapsulates an IP packet inside an Ethernet frame, it knows the source MAC address (its own), but it may not know the destination MAC address. However, it does know the destination IP address from the packet's IP header. The sender needs some method of discovering the MAC address for a known IP address; it uses the ARP protocol to perform this task.

The Address Resolution Protocol (ARP) is used on Ethernet TCP/IP networks to associate an IP address with a MAC address [8]. ARP is described in RFC 826. ARP uses two different types of messages to allow hosts to perform MAC discovery:

- **ARP Request** messages are normally sent to the Ethernet broadcast address, and ask the question "What is the MAC address of the computer that has IP address w.x.y.z?"
- **ARP Reply** messages are sent as a unicast response to an ARP Request: "I have that IP address, and my MAC address is aa-bb-cc-dd-ee-ff."

Each system maintains a database of previously learned IP to MAC mappings, known as the ARP cache. If a system needs to send a packet to a particular IP address, it first checks its ARP cache to determine if it already knows a MAC address for that IP address. If it finds such an entry, the system uses that MAC to address the frame.

If the destination address is not in the cache, the system sends an ARP Request to every host on the Ethernet. If a host on the LAN recognizes that IP address as its own, then it sends an ARP Reply, containing its IP and MAC address. The sender adds the ARP Reply data to its ARP cache for future reference, and can now address and send the frame. Cache entries expire after a period of several minutes, after which they are deleted from the cache.

The following example shows *PC1's* ARP cache, containing the IP and MAC addresses of *Router*:

```
C:\>arp -a

Interface: 192.168.72.101 --- 0x2
  Internet Address      Physical Address      Type
  192.168.72.250        00-50-56-e7-cf-19     dynamic
```

ARP Cache Poisoning

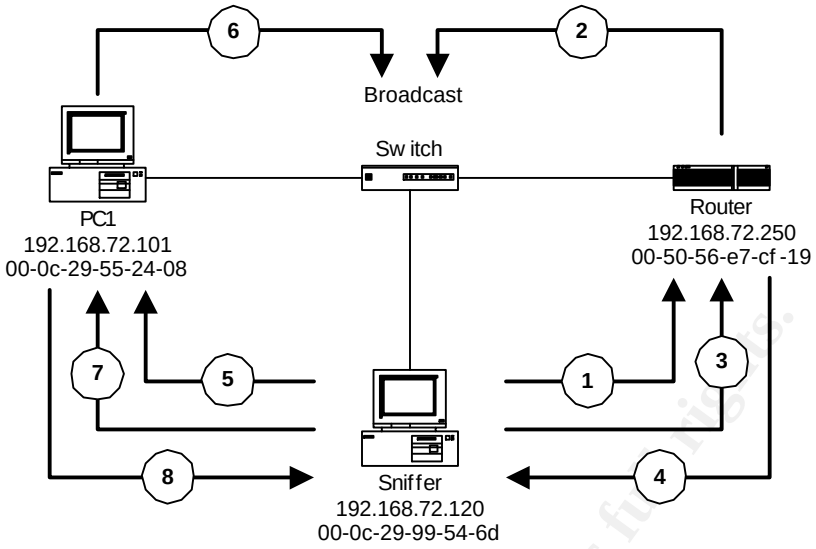
By manipulating the ARP cache on each victim host, it is possible to change the normal direction of traffic between two hosts, and redirect it to flow through the attacker's machine instead. ARP is a stateless protocol, and updates are not checked to authenticate the sender or validate the new information. Specially crafted ARP Reply packets sent to each host will force an update in their respective ARP caches, and the hosts will then send frames based on the updated ARP cache entries.

Although ARP Replies are accepted without validation, there are certain conditions that must be met, as described in *Bruschi et al.*:

> "Some operating systems, e.g. Solaris, will not update an entry in the cache if such an entry is not already present when an unsolicited ARP reply is received. Although this might seem an effective precaution against cache poisoning, the attack is still possible. The attacker needs to trick the victim into adding a new entry in the cache first, so that a future (unsolicited) ARP reply can update it. By sending a forged ICMP echo request as if it was from one of the two victims, the attacker has the other victim create a new entry in the cache. When the first victim receives the spoofed ICMP echo request, it replies with an ICMP echo reply, which requires resolving first the IP address of the original ICMP request into an Ethernet address, thus creating an entry in the cache. The attacker can now update it with an unsolicited ARP reply. [9]"

In addition, some operating systems only accept the first received reply to their ARP Request, forcing a race condition for arrival between the attacker's reply and the actual reply. If the attacker sends the poison ARP Reply immediately after the spoofed ICMP packet, the real ARP reply will likely arrive too late, and will be discarded as invalid [10].

The ARP poisoning process is shown below:



1. *Sniffer* spoofs a ping from *PC1's* IP address to Router
2. *Router* broadcasts an ARP Request to find *PC1's* MAC address.
3. *Sniffer* immediately sends a poison ARP Reply to *Router*, telling it that the IP of *PC1* has the MAC address of *Sniffer*, and winning the race condition. *Router* adds the fake ARP mapping to its ARP cache.
4. *Router* sends the ICMP ping reply bound for *PC1's* IP to *Sniffer's* MAC address.
5. *Sniffer* spoofs a ping from *Router's* IP address to *PC1*
6. *PC1* sends an ARP Request to find *Router's* MAC address.
7. *Sniffer* immediately sends a poison ARP Reply to *PC1*, telling it that the IP of *Router* has the MAC address of *Sniffer*, and winning the race condition. *PC1* adds the fake ARP mapping to its ARP cache.
8. *PC1* sends the ICMP ping reply bound for *Router's* IP to *Sniffer's* MAC address.

This exchange can also be seen in the following *tcpdump* output:

```
1.  0:c:29:99:54:6d 0:50:56:e7:cf:19 ip 42: pc1 > router: icmp:
      echo request [tos 0x7,CE]
2.  0:50:56:e7:cf:19 Broadcast arp 60: arp who-has pc1 tell router
3.  0:c:29:99:54:6d 0:50:56:e7:cf:19 arp 42: arp reply pc1 is-at 0:c:29:99:54:6d
4.  0:50:56:e7:cf:19 0:c:29:99:54:6d ip 60: router > pc1: icmp: echo reply
5.  0:c:29:99:54:6d 0:c:29:55:24:8 ip 42: router > pc1: icmp:
      echo request [tos 0x7,CE]
6.  0:c:29:55:24:8 Broadcast arp 60: arp who-has router tell pc1
7.  0:c:29:99:54:6d 0:c:29:55:24:8 arp 42: arp reply router is-at 0:c:29:99:54:6d
8.  0:c:29:55:24:8 0:c:29:99:54:6d ip 60: router > pc1: icmp: echo reply
```

When this is accomplished, all traffic flowing between *PC1* and *Router* will be sent to *Sniffer* instead of directly to its intended destination. *Sniffer* periodically

sends another pair of poisoned ARP Replies to *Router* and *PC1* to prevent the poisoned ARP cache entries from timing out.

```
0:c:29:99:54:6d 0:50:56:e7:cf:19 arp 42: arp reply 192.168.72.101 is-at 0:c:29:99:54:6d
0:c:29:99:54:6d 0:c:29:55:24:8 arp 42: arp reply 192.168.72.250 is-at 0:c:29:99:54:6d
```

PC1 now believes that its default gateway (*Router*) has the MAC address of *00-0c-29-99-54-6d*, and it will send all traffic bound for other networks, including the Internet, directly to *Sniffer*, based on the information in its ARP cache:

```
C:\>arp –a

Interface: 192.168.72.101 --- 0x2
  Internet Address      Physical Address      Type
  192.168.72.250        00-0c-29-99-54-6d     dynamic
```

Likewise, the reverse is true for *Router*. It believes that the MAC address of *PC1* is also *00-0c-29-99-54-6d*, and will now send all traffic bound for *PC1* directly to *Sniffer*. The users of *PC1* never know that their traffic has been rerouted through a third party, and the attacker on *Sniffer* now has the ability to examine frames that were previously unavailable to it when sniffing in the switched LAN.

Limitations of ARP Cache Poisoning Techniques

This attack has several limitations. It is important to note that *Sniffer* must forward all intercepted packets to the correct victim hosts, or the result would be a denial of service, as no frames sent between the two hosts would ever reach their destination if *Sniffer* merely discarded them. ARP poisoning attacks will also degrade network performance, as the attacking system must intercept, analyze, and forward each frame sent between the two victims. Finally, one cannot poison the caches of computers on a different subnet or VLAN because ARP broadcasts only reach systems within a single Ethernet broadcast domain.

Using Ettercap

As long as */usr/local/sbin* is in your PATH, you can start *ettercap* in a terminal window by simply typing

```
# ettercap
```

By actively probing with a storm of ARP broadcasts, *ettercap* can quickly learn all the MAC addresses present on the LAN. Upon startup, *ettercap* broadcasts an ARP Request to every IP address on its subnet. This step can be time-consuming, based on the network subnet configuration; a Class C network (netmask 255.255.255.0) has 2^8-2=254 hosts, and discovery takes only seconds. However, if *ettercap* is started on a Class B network, which has a netmask of

255.255.0.0 with 2^{16}-2=65534 hosts, it may take a significant amount of time to scan the network.

```
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.1 tell 192.168.72.120
0:50:56:c0:0:8 0:c:29:99:54:6d arp 60: arp reply 192.168.72.1 is-at 0:50:56:c0:0:8
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.2 tell 192.168.72.120
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.3 tell 192.168.72.120
[Output truncated]
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.101 tell 192.168.72.120
0:c:29:55:24:8 0:c:29:99:54:6d arp 60: arp reply 192.168.72.101 is-at 0:c:29:55:24:8
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.102 tell 192.168.72.120
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.103 tell 192.168.72.120
[Output truncated]
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.250 tell 192.168.72.120
0:50:56:e7:cf:19 0:c:29:99:54:6d arp 60: arp reply 192.168.72.250 is-at 0:50:56:e7:cf:19
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.251 tell 192.168.72.120
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.252 tell 192.168.72.120
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.253 tell 192.168.72.120
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.254 tell 192.168.72.120
0:50:56:f7:14:ca 0:c:29:99:54:6d arp 60: arp reply 192.168.72.254 is-at 0:50:56:f7:14:ca
0:c:29:99:54:6d Broadcast arp 42: arp who-has 192.168.72.255 tell 192.168.72.120
```

If *ettercap* knows its DNS server, it then attempts to resolve the DNS hostnames of any system that responded to the ARP Request storm. Again, this process may be time-consuming with a sizable network. After this is done, *ettercap* has an accurate map of hosts on the switched network.

Useful Command-line Options

Ettercap allows the user to modify its startup behavior, allowing stealthier probing of the network. Here are some of the more useful command-line options, which may be specified in either short or long form (*ettercap –z* or *ettercap --silent*) [11]:

Do not perform the ARP Request storm on startup

```
# ettercap –z (--silent)
```

Change the interval between ARP storm requests for stealth

```
# ettercap –Z (--stormdelay) 5000
```

Send ARP Requests only to specific IP addresses

```
# ettercap –H (--hosts)192.168.72.101,250
```

Enter passive sniffing mode, and also save the results to a file

```
# ettercap -Ok (--passive --savehosts)
```

Load the saved host map from a file

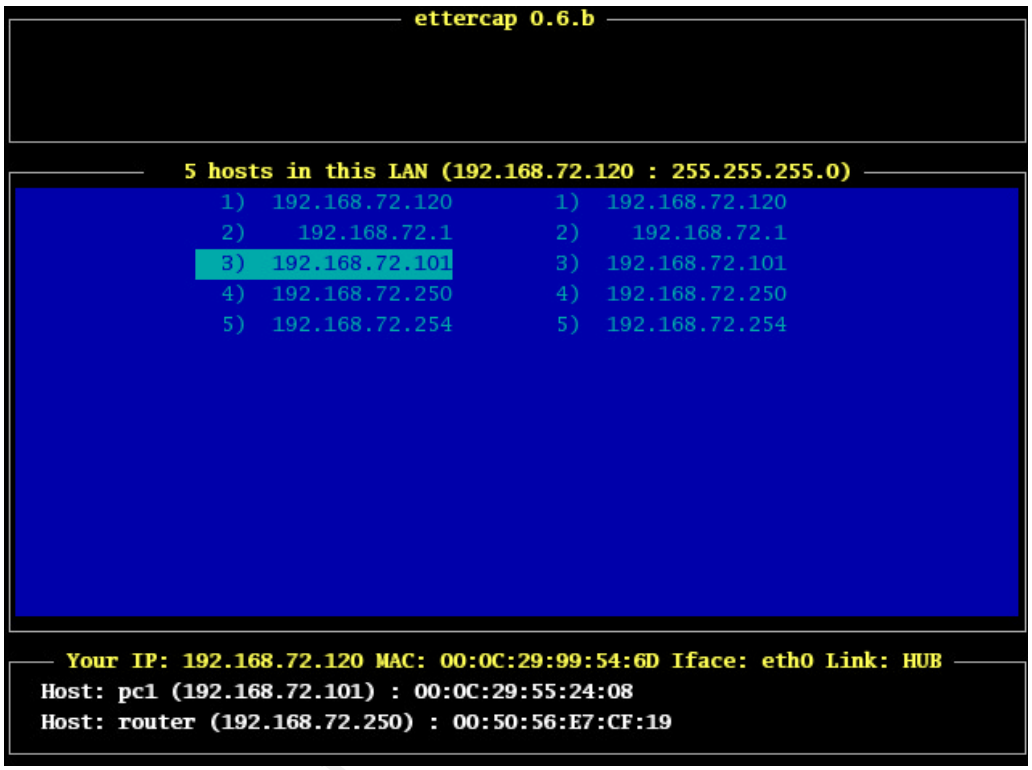
```
# ettercap –j (--loadhosts) 192.168.72.0_255.255.255.0.ehl
```

You can also run *ettercap* in simple mode (*-N* or *--simple*). This option does not start the user interface, and therefore allows *ettercap* to be used in scripting. For example, you can quickly create a map of the network by running *ettercap* as follows, which launches an ARP storm, saves the results in a file, and exits:

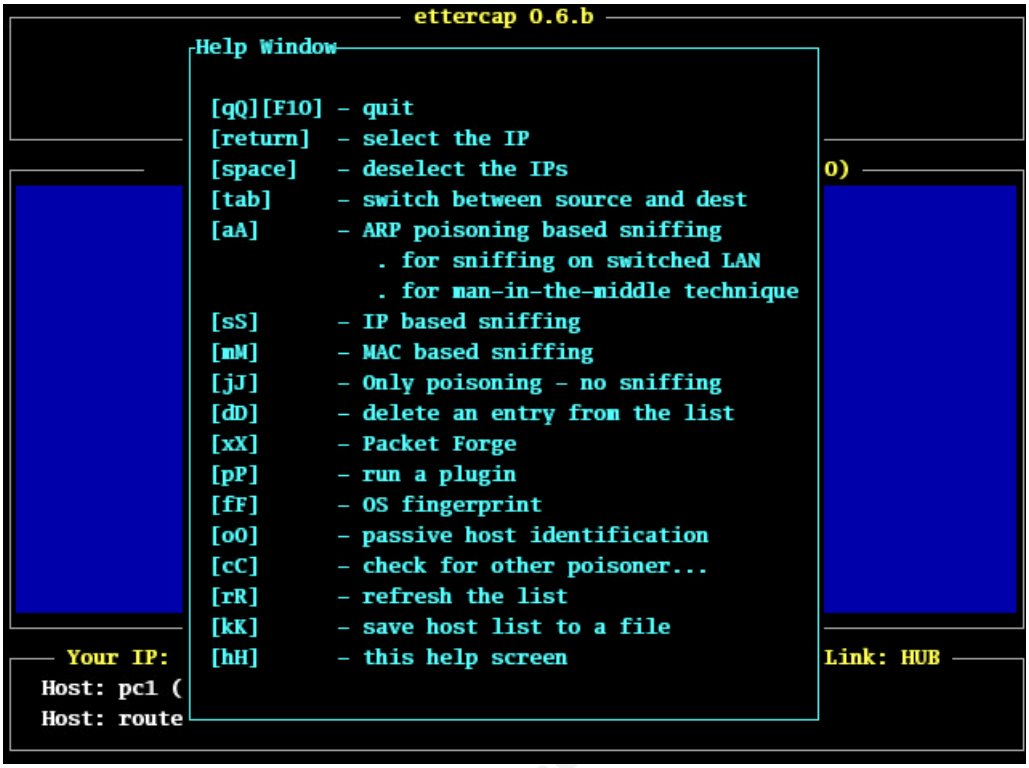
```
# ettercap –Nk
```

The Ettercap Interface

When *ettercap* is started in interactive mode, the user is presented with two columns that each list all the IP addresses which the ARP Request storm detected. The source IP column is on the left, and the destination IP column is on the right.



Pressing '*h'* on any screen presents a context-sensitive help menu. All functions in this interface are launched by pressing single keys:



Pre-Poisoning Reconnaissance

Before any ARP poisoning is performed, the user has a list of all hosts that responded to the ARP Request storm. At this point, there are several features that are useful for further reconnaissance of the network, including:

- c Search for other ARP poisoners on the LAN
- f Fingerprint the selected host's operating system and services
- k Manually save the map of discovered hosts to a file
- x Packet Forge – craft customized packets
- p Run a plugin that does not rely on ARP poisoning

For example, to run the plugin *imp*, which collects NetBIOS names, against *PC1:*

1. In the destination IP column, select *192.168.72.101* and press Enter
2. Press *p*, then select *imp* and press Enter

```
Try to retrieve some Windows names from 192.168.72.101…
Retrieved 4 names:
1) PC1 (Unique)
2) WORKGROUP (Group)
3) PC1 (Unique)
4) PC1 (Unique)

imp plugin ended.  (press 'q' to quit…)
```

A Simple Attack against an FTP Session

Let's demonstrate a simple ARP cache poisoning attack. In this example, the attacker, using *ettercap* on *Sniffer*, wants to capture all traffic going from *PC1* to the Internet. To do so, he selects *PC1* as the source IP, and PC1's default gateway (*Router*) as the destination. Since all traffic sent by *PC1* outside the local subnet passes through *Router*, ARP poisoning these two hosts will capture all Internet traffic as well.

Initializing the ARP poisoning attack is simple:

1. Select a source IP of 192.168.72.101 (*PC1*)
2. Select a destination IP of 192.168.72.250 (*Router*)
3. Press 'a' to poison the ARP tables on the selected hosts.

Ettercap poisons the ARP cache on each victim as described above, and resends the poisoned ARP Replies every thirty seconds to ensure that the poisoning will continue.

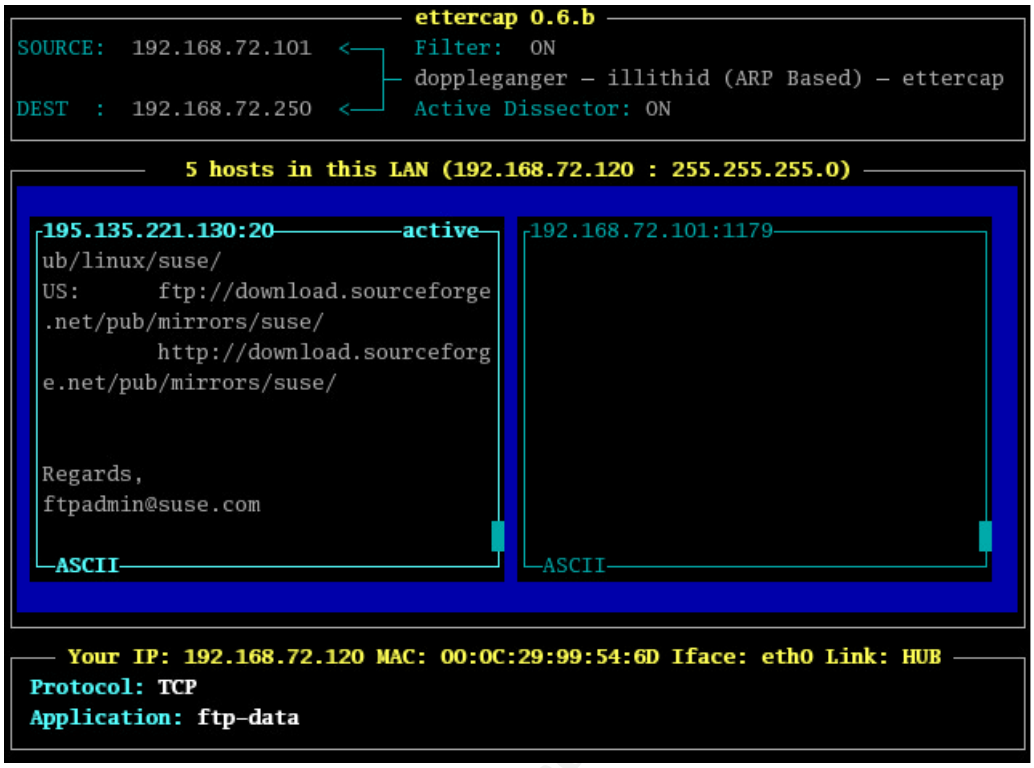
Now all Ethernet traffic between the two hosts is being intercepted by *Sniffer*. Active Dissector is on by default, and it automatically extracts the usernames and passwords from any active connection whose protocol it recognizes. Every stream between the victim hosts is captured and analyzed without the user having to select any particular connection.

The user on *PC1* starts an FTP session to ftp.suse.com, logs in, and downloads a text file.

```
C:\>ftp ftp.suse.com
Connected to ftp.suse.com.
220 "Welcome to the SuSE ftp server: Please login as user 'ftp'"
User (ftp.suse.com:(none)): ftp
331 Please send your email address as a password.
Password:
230 Login successful. Have a lot of fun.
ftp> cd pub/
[Output truncated]
250 CWD command successful.
ftp> get README.txt
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for README.txt (4046
bytes).
226 File send OK.
ftp: 4046 bytes received in 0.20Seconds 19.83Kbytes/sec.
ftp> bye
221 Goodbye.
```

We can see both the ftp session (on destination port 21) and the ftp data connection (on source port 20). Highlighting the first connection reveals the username and password that were sent in cleartext. Pressing 'l' at this point will log any captured passwords to a file named in the format "yyyymmdd_Dumped_Password.log".

Selecting the data connection shows the contents of the downloaded README.txt file.



Again, pressing 'l' will log the contents of this stream to a file, effectively saving the intercepted text file.

Unpoisoning

To reverse the ARP cache poisoning, *ettercap* sends ARP Reply packets containing the correct information to the two victim hosts and gracefully resets their respective ARP caches back to normal.

```
[Send correct information to each victim ]
0:c:29:99:54:6d 0:50:56:e7:cf:19 arp 42: arp reply  pc1 is-at 0:c:29:55:24:8
0:c:29:99:54:6d 0:c:29:55:24:8 arp 42: arp reply  router is-at 0:50:56:e7:cf:19

[Send unicast ARP Requests to victims to test unpoisoning ]
0:c:29:99:54:6d 0:50:56:e7:cf:19 arp 42: arp who-has router (0:50:56:e7:cf:19) tell pc1
0:c:29:99:54:6d 0:c:29:55:24:8 arp 42: arp who-has pc1 (0:c:29:55:24:8) tell router

[Send correct information once more ]
0:c:29:99:54:6d 0:50:56:e7:cf:19 arp 42: arp reply pc1 is-at 0:c:29:55:24:8
0:c:29:99:54:6d 0:c:29:55:24:8 arp 42: arp reply router is-at 0:50:56:e7:cf:19
```

HTTPS Interception Attack

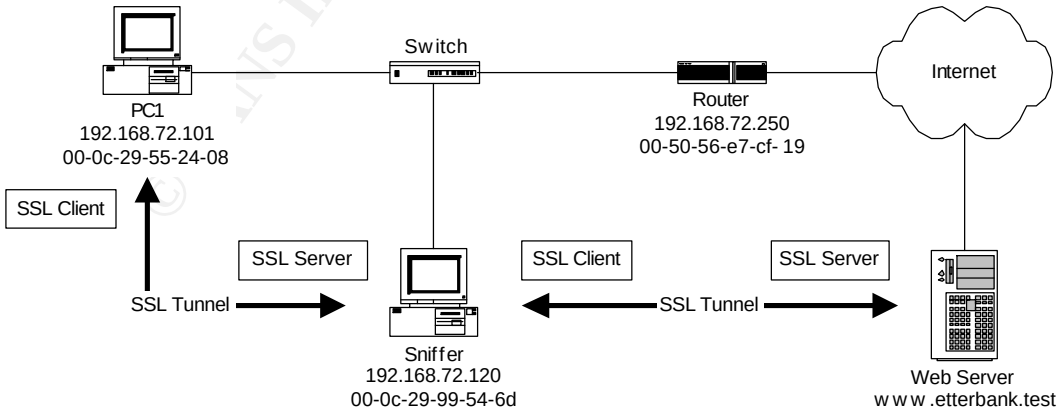
So far, we have seen how simple it is for *ettercap* to automatically extract data from cleartext traffic. However, *ettercap* can also be used to perform a more complicated attack on SSL web sites.

When a client normally accesses a web page using HTTPS, the client and server create an SSL encrypted tunnel through which all HTTP data passes. If this traffic were to be captured by a third party, the encrypted data would be unreadable to the attacker. Here is a simplified version of how an SSL tunnel is created.

1. The client's browser requests a secure web page.
2. The web server sends the website's certificate to the browser.
3. The browser checks the certificate's validity.
4. If the certificate is valid, the browser generates a session key, encrypts it with the public key from the server's certificate, and sends it to the server.
5. The server decrypts the session key.
6. Both sides use the symmetric session key to encrypt the subsequent HTTP communication.

The validity of the certificate depends on three things. The certificate must be signed by a trusted certificate authority, such as Verisign or Thawte. The certificate also must not have passed its expiration date. Finally, the hostname in the certificate must match the name of the website that the browser is attempting to display. If any of these three conditions are not met, the browser displays a dialog box that explains the error, and requests permission to continue establishing the SSL session using the questionable certificate. Valid certificates usually pass these tests unnoticed by the casual user, and because of this many users are unfamiliar with the validity requirements, or even the existence of a certificate that enables SSL encryption.

Ettercap can be used to establish a MITM attack in an HTTPS session if the victim ignores the validity warnings and accepts an invalid certificate. It does so by setting up two separate SSL tunnels:

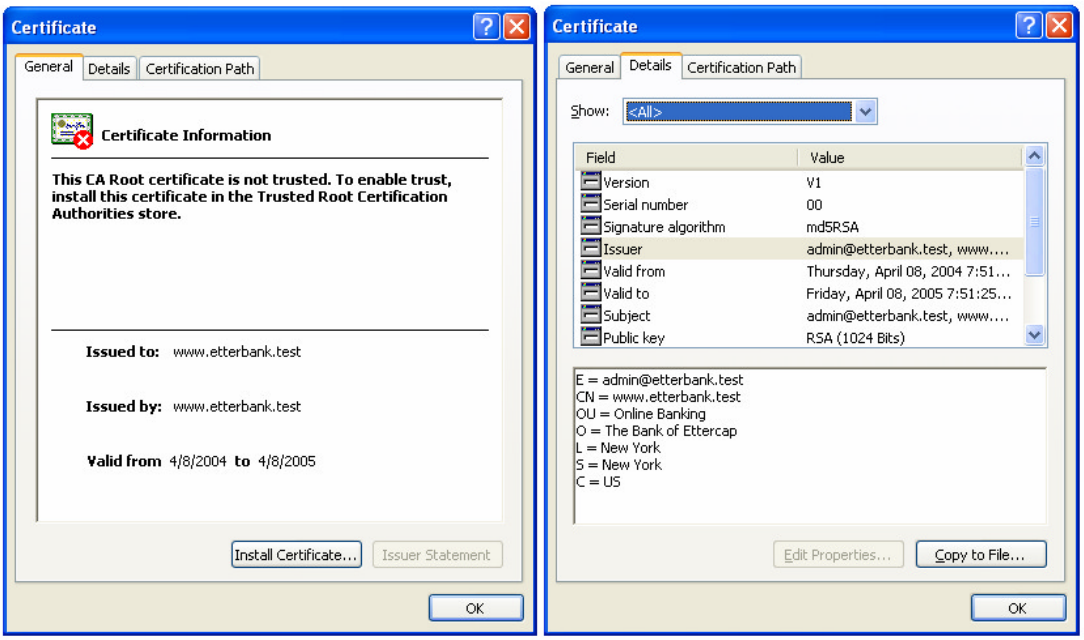


After ARP poisoning the victim's computer and gateway, *ettercap* intercepts the victim's SSL request, and presents the victim's browser with a false certificate. If the victim accepts the invalid certificate, *ettercap* establishes an SSL tunnel from the victim to itself, masquerading as the secure web server. It then establishes a second SSL tunnel to the real web server, with itself as the SSL client. Since it can now decrypt HTTPS traffic from the victim, it can easily analyze that traffic before encrypting and forwarding it to the real web server.

It is easy to imagine a scenario where this attack would work. An attacker may notice that her coworker regularly accesses his online banking at the Bank of Ettercap's secure website. Since the attacker has access to the same office LAN, and knows which secure website the victim is likely to use, she could use *ettercap* to preemptively poison the workstation and lie in wait for the next online banking session. She could also generate a customized and convincing SSL certificate, and install it in */usr/local/share/ettercap*:

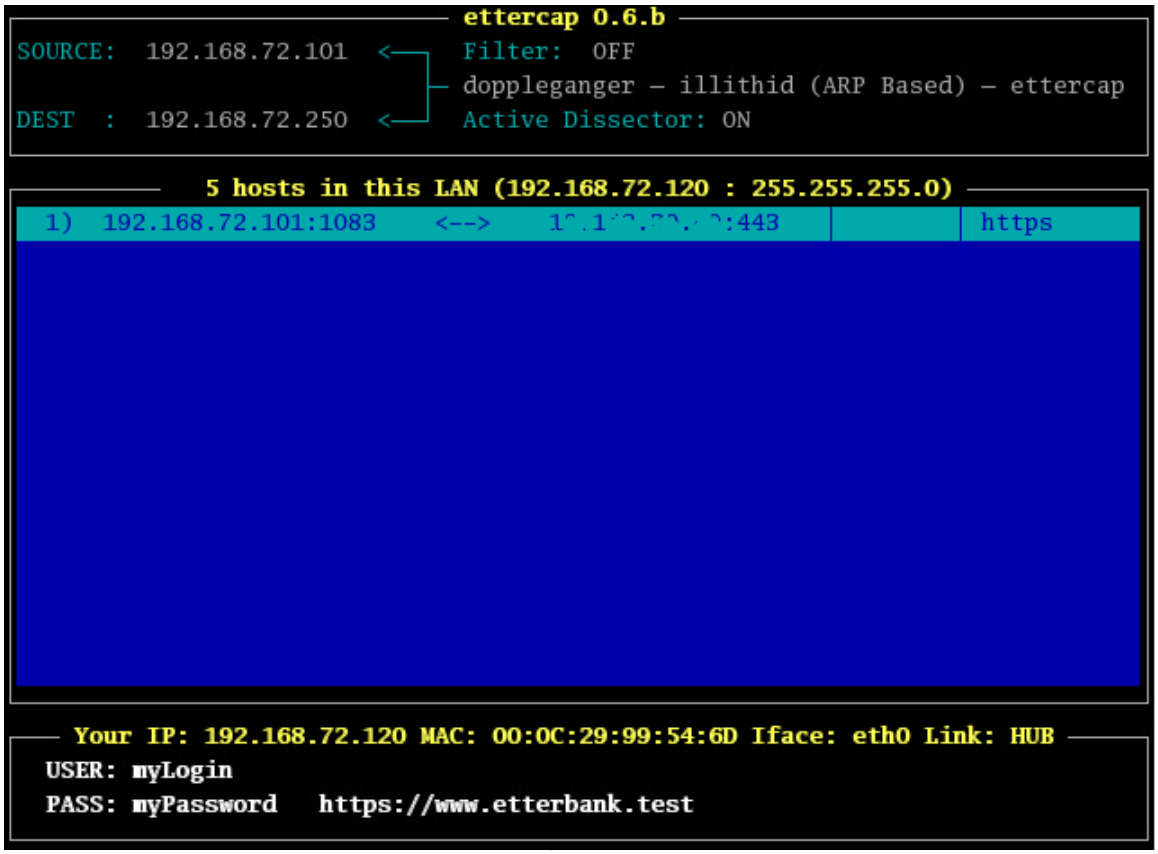
```
# ettercap --newcert

Generating Openssl [etter.ssl.crt] certificate...

Generating RSA private key, 1024 bit long modulus
.....++++++
[Output truncated]
Country Name (2 letter code) [GB]:US
State or Province Name (full name) [Berkshire]:Washington
Locality Name (eg, city) [Newbury]:Redmond
Organization Name (eg, company) [My Company Ltd]:The Bank of
      Ettercap
Organizational Unit Name (eg, section) []:Online Banking
Common Name (eg, your name or your server's hostname) []:
      www.etterbank.test
Email Address []:admin@etterbank.test
Getting Private key
[Output truncated]
Openssl certificate generated in ./etter.ssl.crt

# cp etter.ssl.crt /usr/local/share/ettercap
```

Now the customized fake certificate is available for use. Since the information in the certificate is targeted to look as much like the real bank's certificate as possible, we can view the certificate as a form of social engineering designed to convince the victim that nothing is wrong.



The certificate appears to be valid, except that it has not been signed by a trusted certificate authority. The warning message generated by a browser that has been presented with this false certificate is shown below:

Once the two SSL tunnels have been established, the packets from *PC1* are available in cleartext to *Sniffer*, and Active Dissector can extract usernames, passwords, and data as easily as with any normal HTTP traffic before forwarding the packets to the real web server.



An HTTPS man-in-the-middle attack of this type relies on the acceptance of an invalid certificate by a naïve or untrained user. While this method might require luck or social engineering to get the user to comply, it seems likely that it would have a good chance of success. If a user gets an error message that he does not understand, and which appears to be hindering his progress toward getting his bills paid on time, it is quite reasonable to assume that he might click *Yes* just to move forward. One cannot assume that most people will even read the content of error messages, or will not reflexively click whichever option will make an error disappear.

It should also be noted that any HTTPS session the poisoned host browses to would use the same false certificate regardless of the URL, but if the victim doesn't actually look at the certificate, this may not matter. In a large office with many LAN ports, it would only take one impatient victim to make the attack worthwhile.

Filters and Character Injection

Ettercap's ability to analyze passing data streams is greatly enhanced by its filtering capabilities. Filters can be configured to search for and replace arbitrary

text or hexadecimal strings in data streams before they are forwarded to the destination host. For example, a filter can be configured to automatically replace the text string *www.domain1.com* with *www.domain2.com* in all frames sent to a destination port of 53. If the filter is enabled while the victim hosts are poisoned, any subsequent DNS traffic would be automatically changed, and the victim will receive incorrect or malicious data from an otherwise valid DNS request. Passing data could also be filtered to dynamically change words in emails or to replace text in a loading web page.

In addition to simple replacement, *ettercap* also has the ability to inject additional characters into an active data stream while dynamically recalculating the proper IP sequence numbers and packet checksums required to keep the connection alive. This allows, for example, the insertion of malicious code such as JavaScript into a web page, or viruses into an email. An attacker could also inject new commands to a server as if they came from the client, or vice versa [12].

Downgrade Attacks

It should be noted that many of the more advanced capabilities of *ettercap* fit into the general category of *downgrade attacks*. The client originally attempts to connect using a protocol that is secure and not vulnerable to sniffing. However, since *ettercap* has inserted itself in the middle of the Ethernet connection, it can then attempt to block the completion of the secure connection, even if it is not capable of analyzing that protocol. If the client is configured to use a less secure fallback choice when the more secure method fails, then *ettercap* may still be able to compromise that connection.

For example, an attacker could use filters that force an SSH client to initiate an SSH1 connection instead of the more secure SSH2 [13]. As long as *ettercap* recognizes the beginning of an SSH2 connection, it can manipulate the exchange before it progresses. When the server replies that it supports both SSH1 and SSH2, the filter could change the response to say that it only supports SSH1. The client would then request only an SSH1 connection, from which Active Dissector can readily extract usernames and passwords [14]. In doing so, the attacker has worked around the use of a protocol that *ettercap* cannot crack.

Ettercap Future Development

The next release of *ettercap*, version 0.7.0 (also known as *ettercap NG*) is in alpha testing as of this writing. The new version promises to be a thorough rewrite of the sniffing engine, providing a more efficient and stable attack platform. In addition, *ettercap NG* comes with a redesigned GUI built with the GIMP toolkit's GTK+ libraries [15]. The original *ncurses* interface is still available as well.

The developers have also dropped the old plugin nomenclature, replacing the obscure mythological names with more descriptive ones. For example:

- chk_poison
- dns_spoof
- dos_attack

One notable future direction is the expansion of Active Dissector for the SSL protocol. The *NG* release will continue to support HTTPS session detection, but future releases will also allow interception of POP3, IMAP, and FTP over SSL as well. Also noteworthy is the introduction of a “unified sniffing” method for packet analysis. The new release separates the sniffing and filtering module from the MITM functions, so that *ettercap* can be used to provide a filtered active data stream to either its own attack modules or those of another third-party tool. [16]

Defenses Against Ettercap

As we have seen, it is quite easy for an attacker using *ettercap* to launch a man-in-the-middle attack once he has a LAN connection. How does one defend against this? First, one should not underestimate the need to educate users. While the average employee certainly does not need to know the details of Ethernet addressing and ARP, a user who has been trained not to accept fake SSL certificates could alert the network administrators that there may be a larger problem loose on the LAN.

Some defense tools monitor the network for changing ARP data or watch for ARP attack signatures. One such tool is *arpwatch*, which maintains a database of current IP and MAC address mappings, and can report changes to this database through email. [17]. One can also use *ettercap* itself to actively search out other ARP poisoners. As a preventative measure, one could regularly run the *H00_lurker* plugin interactively to detect other systems using *ettercap* on the LAN. Also, a properly configured and positioned intrusion detection system will likely notice both the startup ARP storm and the crafted ARP Reply packets, possibly alerting network security personnel to an ARP poisoning in progress.

Port security is another valuable part of defense in depth. Managed switches let administrators configure strict limits on which MAC addresses are allowed to connect to certain switch ports. Limiting and specifying the MAC addresses on switch ports helps to prevent unauthorized systems from connecting to the LAN, and can ensure that MAC addresses are not hijacked or spoofed. However, *ettercap* does not change its own MAC address to perform ARP cache poisoning, and therefore port security is not effective against this type of attack [18].

A *static ARP table* is a list of valid IP to MAC address mappings that is set into the ARP cache at system boot time. As the name implies, the entries in a static ARP table do not allow dynamic updates, and do not time out from the cache.

While this would prevent classic ARP poisoning, these static tables may be difficult to manage; there is no centralized ARP network information service, so changes and additions to the list would have to be propagated manually to all LAN hosts [19].

Secure ARP

These solutions assume a defensive posture against the real problem, which is the insecurity of the ARP protocol. ARP was conceived in the early 1980s as a simple method to allow Ethernet communication, but it was never designed to authenticate or validate its own information [8]. After what we have seen, there is an obvious need for an updated secure protocol that can perform IP to MAC address resolution and ensure that dynamically updated MAC information is not subject to abuse.

Interestingly enough, one of the creators of *ettercap* has co-authored a paper that proposes an update to the ARP protocol that protects against ARP poisoning. In **"S-ARP: a Secure Address Resolution Protocol"**, Alberto Ornaghi helped to design a secure extension to ARP called S-ARP, or Secure ARP. In this system, all hosts on the LAN would replace their use of ARP with S-ARP, which relies on a lightweight PKI-based authentication scheme to validate the sender of ARP messages:

> "Since S-ARP is built on top of ARP, its specification (as for message exchange, timeout, cache) follows the original one for ARP. In order to maintain compatibility with ARP, an additional header is inserted at the end of the protocol standard messages to carry the authentication information […] In S-ARP all reply messages are digitally signed by the sender with the corresponding private key. At the receiving side, the signature is verified using the host public key. If the public key of the sender host is not present in the receiving host key ring or the one in the key ring does not verify the signature, the public key of the sender is requested from the AKD [Authoritative Key Distributor] [20]. "

The proposed S-ARP protocol is still in its infancy, but it promises to be a robust solution to a nagging problem. As long as ARP cache poisoning is easy to perform and difficult to detect, it is clear that man-in-the-middle attacks will remain popular and effective.

Summary

Ettercap has developed into a tool that encompasses a wide range of available LAN attacks. Since it combines many separate attacks into one convenient interface, *ettercap* is also a great way for new security practitioners to learn the technical basis for many LAN attacks; discovering how a hacker would use these tools is valuable training. As it becomes more popular to extend the LAN through

wireless access points in coffee shops and restaurants, it becomes clear that the security community needs to promote change. The development of secure replacements for older protocols such as ARP would go a long way towards eliminating some of the more common attack strategies.

Finally, although the white hat security community can use this tool to further its own understanding of hacking techniques, remember to take *ettercap's* exit message to heart:

```
ettercap 0.6.b brought from the dark side of the net by ALoR and
NaGA...

may the packets be with you...

They are safe!!  for now...
```

References

[1] Ornaghi, Alberto, and Marco Valleri. "Ettercap." http://ettercap.sourceforge.net (March 31, 2004).

[2] Fyodor. "Remote OS Detection via TCP/IP Fingerprinting." http://www.insecure.org/nmap/nmap-fingerprinting-article.html (March 20, 2004).

[3] "Hping home page." http://www.hping.org (March 20, 2004).

[4] "Ettercap Forum." http://ettercap.sourceforge.net/forum/viewtopic.php?t=1193&highlight=redhat (April 3, 2004).

[5] Hunt, Craig. TCP/IP Network Administration. Sebastopol: O'Reilly & Associates, Inc., 1992. pp. 8-17.

[6] Odom, Wendell. Cisco CCNA Exam #640-507 Certification Guide. Indianapolis: Cisco Press, 2000. pp. 96, 138-139.

[7] Ibid, pp. 75-86.

[8] Plummer, David C. "RFC 826: An Ethernet Address Resolution Protocol." November 1982. http://www.ietf.org/rfc/rfc0826.txt?number=826 (March 15, 2004).

[9] Bruschi, D., A. Ornaghi, and E. Rosti. “S-ARP: a Secure Address Resolution Protocol.” December 2003. http://www.acsac.org/2003/papers/111.pdf : p. 2 (April 10, 2004).

[10] Ornaghi, A. and Marco Valleri. “Man In The Middle Attacks Demos.” Black Hat Conference USA, 2003. http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-ornaghi-valleri.pdf : p. 27 (April 5, 2004).

[11] Ornaghi, A. and Marco Valleri. Ettercap 0.6.b man page.

[12] Ornaghi, A. and Marco Valleri. “Man In The Middle Attacks Demos.” pp. 4-5, 13-14 (April 5, 2004).

[13] Krahmer, Sebastian. “SSH for fun and profit.” July 1, 2002. http://segfault.net/~stealth/ssharp.pdf (April 9, 2004).

[14] Ornaghi, A. and Marco Valleri. “Man In The Middle Attacks Demos.” p. 19 (April 5, 2004).

[15] “GTK+ The GIMP Toolkit.” March 16, 2004. http://www.gtk.org (March 15, 2004).

[16] Ornaghi, A. and Marco Valleri. Ettercap NG man page.

[17] “SecurityFocus HOME Tools: Arpwatch” http://www.securityfocus.com/tools/142 (April 9, 2004).

[18] Bruschi, D., et al. “S-ARP: a Secure Address Resolution Protocol.” p. 8 (April 10, 2004).

[19] Ornaghi, A. and Marco Valleri. “Man In The Middle Attacks Demos.” p. 28 (April 5, 2004).

[20] Bruschi, D., et al. “S-ARP: a Secure Address Resolution Protocol.” p. 3 (April 10, 2004).

General References

Song, Dug. “dnsiff Frequently Asked Questions.” December 7, 2001. http://naughty.monkey.org/~dugsong/dsniff/faq.html (March 5, 2004).

Whalen, Sean. “An Introduction to Arp Spoofing.” April 2001. http://packetstormsecurity.org/papers/protocols/intro_to_arp_spoofing.pdf (March 6, 2004).