

Man in The Middle and other Network Attacks

Created by: Peter A. H. Peterson and Dr. Peter Reiher, UCLA {pahp, reiher}@cs.ucla.edu



Important! This experiment has a tendency to stay "swapped in" because the nodes on the system appear to be "busy". We don't want our labs to waste DETER resources, so in order to make sure that unused nodes are not left idle, the **maximum** amount of time this lab will stay swapped in at once is **six hours**. If you need it for more than six hours in one sitting, you will need to swap out and back in again some time before that.

Contents

1. [Overview](#)
2. [Required Reading](#)
 1. [Sniffing Ethernet](#)
 1. [Shared Medium](#)
 2. [Switched Networks](#)
 2. [ARP Spoofing](#)
 3. [Eavesdropping](#)
 4. [Replay Attacks](#)
 5. [Insertion Attacks](#)
 6. [MITM vs. Encryption](#)
 1. [Public Key Encryption](#)
 2. [Man In the Middle Attacks](#)
 7. [Software Tools](#)
 1. [tcpdump](#)
 2. [chaosreader](#)
 3. [Ettercap](#)
 1. [Getting Started](#)
 2. [SSL Attacks](#)
 3. [Interface Tips](#)
 4. [Ettercap Filters](#)
 5. [Ettercap Links](#)
3. [Introduction](#)
4. [Assignment Instructions](#)
 1. [Setup](#)
 2. [Tasks](#)
 1. [Eavesdropping](#)
 2. [Insertion](#)
 3. [Replay](#)
 4. [MITM vs. Encryption](#)
 3. [What Can Go Wrong](#)
5. [Extra Credit](#)
6. [Submission Instructions](#)

Overview

The purpose of this lab is to introduce you to the concepts of sniffing, insertion, replay, and Man-in-the-middle (MITM) attacks, and to give you opportunities to execute these attacks in a testbed. You should have a solid grasp of networking as from an undergraduate networking course, and be familiar with basic programming principles.

After successful completion this lab, you will:

1. understand the basics of network sniffing
2. understand the basics of replay attacks
3. understand the basics of insertion attacks
 - including the development of your own filters for Ettercap
 - including common programmatic defenses against replay attacks
4. understand the general concepts of man-in-the-middle attacks
5. execute sniffing, replay, and insertion attacks against cleartext data
6. execute MITM attacks against strong cryptography

Required Reading

"Man-in-the-middle" is a term used in cryptography to describe scenarios where an attacker (the eponymous "man in the middle") between two remote parties can view or control data that would otherwise be secure. "Man-in-the-middle attack" usually refers to vulnerabilities in a key-exchange protocol whereby an attacker can subvert the encryption (typically by substituting secure keys with insecure keys) and gain access to the cleartext without the victims' knowledge. Man-in-the-middle attacks are possible due to characteristics of common networking protocols that make eavesdropping and other "insecure" activities possible.

Sniffing Ethernet

Ethernet is by far the most common data link (or "layer 2") protocol in typical LANs. Ethernets work by passing "frames" (analogous to what TCP/IP calls "packets") on a shared medium according to an algorithm that attempts to be fair. While TCP/IP packets are addressed by port and IP address, Ethernet packets are addressed by a MAC ([Media Access Control](#)) address. Every Ethernet card has a unique, manufacturer-assigned MAC address (although this can often be changed in software) that is included in the header of an Ethernet frame. This address allows Ethernet frames to be addressed to and recognized by the interface that is meant to receive it.

Shared Medium

Due to the broadcast nature of Ethernet, there is *nothing* stopping one interface from "hearing" a packet destined for another host on the same network segment or medium (colloquially called "the wire"). However, Ethernet hardware is typically configured to ignore *at the hardware level* any frames that do not match its MAC address. This ignorance is *by convention only* and serves two purposes; it reduces the amount of data an interface is required to process, and it provides a small amount of "soft security" through voluntary behavior since end users do not normally have direct access to the Ethernet hardware.

However, most (if not all) Ethernet interfaces are perfectly capable of receiving and decoding all

Ethernet frames on the wire. This usually requires putting the interface into so-called promiscuous mode, which passes *all* Ethernet frames on the network segment to the operating system.

Even though most network traffic these days uses TCP/IP or UDP/IP, those packets are still encapsulated in Ethernet frames for transport on the local network segment. This means that if an interface is in promiscuous mode, then all traffic on the segment -- including TCP and UDP -- can also be decoded.

Older network infrastructure hardware used a fully "shared media" paradigm; these network devices are called Ethernet hubs. An Ethernet hub is essentially a physical layer (layer 1 in the OSI model) repeater that broadcasts every inbound packet out on every port but the one that received the packet. This means that logically, all computers connected via a hub are sharing a single "network segment" as if they were connected to the same physical wires.

This method is simple and cheap, because it fundamentally requires very little hardware which was important in the early days of computer networks. However, this design has two significant drawbacks due to the fact that it broadcasts data to places that do not need it. First, it creates a scaling issue because hubs use more bandwidth than is necessary since traffic is broadcast on wires where it is not needed. This causes unnecessary packet collisions and directly limits the size of a hub network due to link saturation. Second, (and more importantly for the purposes of a security class) hub networks send data to interfaces that are not meant to be recipients of the data. While this is technically "correct behavior," it enables data compromise. Incidentally, 802.11 wireless Ethernet networks are extremely similar to Ethernet hub networks, precisely because all hosts share the same "medium" -- the radio waves within proximity of the access point.

This means that a computer connected to a hub or wireless network, using an interface in promiscuous mode, can listen to *all* traffic to and from any host connected to the same network segment. Because of all the above considerations, most wired Ethernets today have eliminated hubs for a faster and more efficient device, the switch.

Switched Networks

An Ethernet switch requires a certain amount of memory and a microcontroller because it is a link layer device and must understand certain elements of the Ethernet protocol. In particular, a *switch* keeps track of which MAC addresses are present on each of its ports. When an Ethernet frame comes into a switch, the switch extracts the destination MAC address from the frame header, and checks to see if it knows which port that MAC address is on. If the switch thinks it knows what port the receiver is on, it sends the frame out on that port only. If it doesn't know what port that MAC is on, it sends the frame out on all ports, in the hopes that a host will receive it and respond, at which point the switch will know where the new MAC address is located.

The "intelligent" nature of a switch (versus a hub) addresses the scaling issues of hubs by transmitting data on as few network segments as possible. This means that compared to a hub, an interface on a switch typically has much less traffic on its network segment. Where a host on a hub could see all traffic, because the hub acts as a single shared medium, a host on a dedicated switch port (even in promiscuous mode) typically sees only traffic that it sends or receives as though it were on its own network segment.

Security researchers have developed several attacks against switched networks with the intent to increase the amount of traffic that an interface can see -- in effect, to make a switch act more like a hub for the purposes of eavesdropping. One early attack was based on the fact that many switches

will "fail over" into a "hub mode" when overloaded. Fortunately, it is harder to force a switch to fail over today than it used to be. This technique is also conspicuous because of the large amounts of bogus traffic it generates.

Unfortunately, there is another technique that is almost impossible to defend against.

ARP Spoofing

ARP Spoofing is a technique for redirecting traffic that can allow an arbitrary interface on an otherwise unsecured switched network to receive traffic that would not normally be sent to it.

The [Address Resolution Protocol](#), or ARP, is used to resolve a network layer address (e.g., IP address) to a link layer address (e.g., MAC address). When a host wishes to send a TCP/IP packet to a remote host, it first creates the TCP/IP packet. However, the hardware in the network interface must encapsulate the packet in a link layer frame (commonly Ethernet) before sending the data out on the wire. In order to assign the frame to the correct destination address, the local host must use ARP to determine which hardware address corresponds to the IP address in question. This involves sending special ARP messages called *ARP Requests* on the local Ethernet and receiving replies from hosts that have the answer.

For example, the host 10.10.10.20 needs to send data to the Internet. In order to reach the Internet, it must send the data through its default gateway, which is configured with IP address 10.10.10.10. The sending host 10.10.10.20 needs to know what Ethernet address 10.10.10.10 has in order to put the correct MAC address in the frame header. In order to discover this, 10.10.10.20 needs to broadcast an ARP request to all Ethernet nodes:

```
arp who-has 10.10.10.10 tell 10.10.10.20
```

... Hopefully, 10.10.10.20 will receive a reply like:

```
arp reply 10.10.10.10 is-at 2e:2f:30:31:32:33
```

... from some host that knows the answer. Host 10.10.10.20 will update its ARP tables associating the IP 10.10.10.10 with the MAC address 2e:2f:30:31:32:33 and won't ask again for some time.

ARP Spoofing relies on the decentralized, unauthenticated, and completely trusting nature of ARP to succeed. In ARP Spoofing, the attacker (or spoofer) tells specific hosts or entire networks that *its* MAC address should be associated with the victim's IP address, rather than the correct MAC to IP mapping. This is commonly used to tell network hosts that an *attacker* is the default gateway instead of the real default gateway. The spoofer keeps the victim's *real* MAC address in its ARP table, and prepares to receive data from network hosts.

When an attacker pretends to be the default gateway for a network, all hosts that received the bogus ARP messages now believe that the *spoofers*' MAC should be inserted into any Ethernet frame destined for the Internet or another IP subnet instead of the real default gateway's MAC address. **This is especially meaningful** because Ethernet switches *build their own ARP tables* in order to

efficiently direct traffic and avoid broadcasting like a hub. This means that *Ethernet switches will be fooled in addition to the remote hosts*. As a result, switches will direct default gateway-bound traffic to the *spoofers*, not the **real** default gateway. Furthermore, a spoofing host that wishes to remain undetected can *rewrite and forward the Ethernet frames to the real default gateway*. This method silently maintains the victim's connections while routing them conveniently under the nose of the sniffer. In this way, ARP Spoofing acts as a kind of network reconfiguration at the data link layer and highlights Ethernet's dependence on ARP.

At this point, the spoofing host has almost total control of the network traffic and can see and sniff arbitrary hosts' data with the use of a promiscuous interface and packet sniffing software. ARP Spoofing does slow down network traffic because of the additional hops and processing involved, but the clever attacker will target only the hosts she wants to affect, rather than the whole network. ARP Spoofing also requires multiple spoofing actions to maintain the tables in their bogus state, but this is not a difficult burden to bear. Furthermore, the careful spoofer can restore the ARP tables of the network before quitting, resulting in minimal network disruption due to the spoofing.

While there are strategies against ARP Spoofing, notably ARP monitoring, fixed ARP tables and [VLANs](#), these are expensive in terms of maintenance, network design, and may require specialized hardware. Because of this, most networks in use today are at least partially vulnerable to ARP Spoofing attacks.

Being the "man in the middle" gives you the opportunity to perform several different kinds of network attacks. Each type of attack can be executed in many different ways with many different goals; the following is just a brief overview.

Eavesdropping

Eavesdropping generically refers to intentionally listening to a private conversation. In the context of computer networks, this is often called *sniffing*, and involves receiving and decoding network packets on your network segment that are not sent by or meant to be received by your host.

The usefulness of sniffing is immediately obvious when the data is being passed in plain text, because any "cleartext" data sniffed is data that can be immediately read. Usernames and passwords are often easy to extract with a little protocol knowledge or an ASCII decoding of the network data.

In the old days, almost no network traffic was encrypted because it required more memory and CPU power, which were both considered too valuable when compared against the likelihood of meaningful losses caused by a sniffing attack. As a result, the TELNET remote shell program remained popular into the late 1990s even though all data and authentication information were sent in cleartext.

Nowadays however, sniffing is one of the most common and easiest attacks to execute due to the proliferation of computer networks, free user-friendly sniffing tools and especially wireless LANs where sniffing requires only proximity and a laptop. Most web traffic is unencrypted, along with many other protocols such as SMTP and IMAP (email). Most remote shell traffic is now encrypted due to the tremendous risk in sending system authentication information unencrypted, but often SMTP, IMAP, or web forum passwords are still unencrypted and may often be the same as more sensitive system passwords. (For example, John's unencrypted webmail password may be the same as his login password on the same system, even though the system uses SSH for logins.) Additionally, while TELNET is not often used for sensitive Internet traffic, it is still used in many internal networks (like companies and banks) because of legacy systems that cannot handle encryption.

Tools such as [tcpdump](#) and [Wireshark](#) are both very powerful, freely available sniffing programs and are available for most computer platforms in use today. For more information, please see their manpages or online documentation.

Replay Attacks

Replay attacks can be more damaging and successful than mere eavesdropping because they can cause remote actions -- sometimes even against cryptography.

The basic idea of a replay attack is to capture packets sent on a wire between two hosts with the intent to later to replay the payloads (or more rarely the same packets) in order to effect the same result. For example, most web based CGI's are extremely susceptible to primitive replay attacks because they are stateless and respond to a single web request with the appropriate information in the request. This request is usually built up over several page requests, but can often be triggered by submitting a full request at once.

Imagine an online greeting card application that has several steps, all managed by the same CGI application, with all input saved in the background (traditionally done with hidden form fields):

1. The first page takes your login name and password: (user: *rms*, password: *gnu*)
2. The second page takes the address of the recipient: esr@catb.org
3. The third page selects an image: *h2obuffalo.jpg*
4. The fourth page provides an entry form for the message: *Happy Hacking!*
5. The fifth page previews and submits and sends the card.

This process would normally take 5 steps by a human, with the possibility of going backwards and forwards to edit form fields. However, the whole process can often be summed up in one HTTP request like this:

```
http://example.com/card.cgi?  
login=rms&pass=gnu&addr=esr@catb.org&img=h2obuffalo.jpg&msg="Happy  
Hacking!"&submit=submit
```

Loading that single URL will skip the whole process and directly send the above card. This is a simple example of the "payload" type of replay attack, where the commands included in a network stream are reissued to a server.

While this URL could be sniffed and entered manually, one can imagine capturing the entire stream of packets (including the TCP handshake) that generated the single HTTP request that resulted in sending the card. After the capture, we might simply re-inject the packets into the network. While this seems like it should work, TCP and other protocols use sequence numbers are chosen pseudorandomly upon connection initiation. Without valid sequence numbers, a naive "replay" of packets into the network will be rejected by the server. This is why "payload" replay attacks are popular, and incidentally highlights one reason why unpredictable numbers are critically important in security.

On the other hand, some protocols, such as UDP, do not necessarily use sequence numbers -- in this

case, a direct replay attack of a data stream could be successfully processed by a server. One can imagine replaying a network capture back onto the network, ignoring any response from the server or replaying the appropriate packets if necessary. If done properly, it would look as though those packets simply "appeared" in the network, and because their headers would already contain the information from the *original sender*, the traffic would appear as though it were coming *from* the original sender, not the host executing the replay attack. Regardless, the result is the same -- using commands that are known to be valid to achieve the same result without authentication.

Regardless of whether you are using "direct" packet replay attacks or more complicated "payload" replay attacks, they can be used for many purposes; one can easily imagine a web or other network conversation giving a user a raise or increasing or decreasing a balance of some kind, from something mostly harmless (like World of Warcraft hitpoints) or something more significant, like an airplane ticket purchase. If that conversation can be replayed, the same effect can be obtained repeatedly.

"Direct" Replay attacks work when there is nothing *unique* about particular transactions other than the request parameters or data payload. The main defense against such an attack are unpredictable (but easily validated) session tokens in the communication that ties a sequence of packets to a unique request. It is not enough to use the current time or a hash of the packet data, because this can easily be guessed or recalculated in a replay attempt.

Good session tokens are a form of the [cryptographic nonce](#). Ultimately, [TCP sequence numbers](#) are a kind of nonce -- they prevent the same data from being used in the future to achieve the same results.

The concept of nonces and packet sequence numbers is often applied to [RPC transactions](#) in order to make individual transactions unique regardless of the underlying network characteristics.

For example, suppose we added a nonce field to our e-card application:

```
http://example.com/card.cgi?
nonce=aksdjf2fk&login=rms&pass=gnu&addr=esr@catb.org&img=h20buffalo.jpg&msg="Happy Hacking!"&submit=submit
```

The field `nonce` consists of a random string of enough length so as to be unguessable which is assigned by the server when the session was first initiated. The nonce would no longer be valid after a certain amount of time, or after the card was successfully sent. In this way, a replay of this stream would include the invalid (already used) nonce "aksdjf2fk", which would be flagged as invalid by the e-card server and summarily rejected. (A request without a nonce would be rejected outright.)

It is also very important to understand that encryption alone does not protect against replay attacks unless there is some kind of nonce or other secret (e.g. session id) within the encrypted packets. For example, with encryption, the original e-card URL could have looked like:

```
http://example.com/cryptocard.cgi?session=1kadjf0921fkj2f0-
20f2p2fusldkjf0294u8rklj;sdf8-9081kdjsf92344f
```

... where "lkadjf092lfkj2f0-20f2p2fusldkjf0294u8rklj;sdf8-908lkdjsf92344f" represents some kind of ciphred session state *without* a nonce. There is no reason why *this* conversation couldn't be replayed with the same success as the original session, resulting in a second card being sent.

There are many tools that are capable of capturing and replaying traffic. For the purposes of this course, we'll use [tcpdump](#) for capturing, with [chaosreader](#) to extract packet payload data.

Insertion Attacks

Insertion Attacks are essentially replay attacks where an attacker changes existing or inserts new data in the network conversation. For example, in the e-card example, an insertion attack could have changed the message in the replayed card from "Happy Hacking" to "Drop Dead!" (although that wouldn't be very nice). Furthermore, this is a less impressive kind of insertion attack because it relies on the first e-card being sent before the modified message could be replayed.

In fact, typically, an "insertion attack" means that the attacker has the ability to capture, modify, and resend valid packets *as though they were the original data stream*. At the very least, this involves capturing the packet, editing, updating any relevant header information (especially the data payload checksum), and resending the modified packet.

This is a "man in the middle" attack in the sense that it requires an entity on the network that has the capability to block transmission of the original packets and send the modified packets. (We usually imagine this entity directly between two communicating parties, but that need not physically be true.) Sophisticated insertion attacks can even modify the responses in order to make the original host think everything has gone according to plan.



Sometimes people use "man in the middle" loosely to refer to any kind of network attack. However, the canonical Man in the Middle attack refers to a specific kind of attack against cryptosystems. We'll discuss this in detail in the next section.

One extreme way of doing this is to insert a computer in the middle of the network that acts as a gatekeeper and is capable of modifying traffic. In this sense, an application proxy performs beneficial "insertion attacks" on a network. For example, a web proxy can exchange all graphics for locally cached copies if they exist. In fact, this is one technique that many "[web accelerators](#)" use -- they replace images and other content with compressed versions, or in the case of images, lower quality lossy-compressed versions.

A more stealthy way of performing insertion attacks is using ARP Spoofing as described above to convince hosts to send their data to the attacker *first*, who will then typically forward the data on after modifying it. This doesn't require modifying the physical nature of the network in any way, and can often be performed undetected from anywhere (even across the Internet) as long as the attacker can directly control an interface on the LAN (e.g. after exploiting a software vulnerability to take over a host).

Encryption can obviously hinder the opportunities for successful insertion attacks, but poor use of encryption can still leave open the opportunity to decrypt the packet, change it, and reencrypt the packet before resending. This is especially true if a cryptographic nonce or other kind of session token can be reverse engineered -- if a correct nonce can be generated or recalculated (like a TCP

checksum) the nonce has lost its usefulness.

In this lab, we will use [Ettercap's](#) extensive filtering capabilities for insertion attacks.

MITM vs. Encryption

While all the above kinds of attacks require a malicious user to have special access to the network, the canonical MITM attack refers to an attack against the key exchange portion of a cryptosystem (which may otherwise be resistant to cryptanalysis). In the canonical MITM attack, a user intercepts the keys as they are exchanged, and then inserts new keys. By doing this, the attacker can encrypt and decrypt all messages, and the communicating parties are none the wiser.



While this isn't an MITM attack, insertion attacks have sometimes been used to downgrade a connection from a more secure protocol to a less secure protocol by tricking the parties into thinking that one or more clients can only support a less secure protocol. Imagine if you could make SSH simply downgrade to TELNET by claiming that the client can't perform encryption!

Public Key Cryptography

While this is not a lab on public key cryptography, the basic idea of public key cryptography is that each host or service has two mathematically related keys. One key (the public key) is used to encrypt data and is public, is registered to a single user or organization, and is widely and freely distributed. The other key (the private key) is used to decrypt data and is known only to the recipient. Because the private key remains *private*, a file encrypted with a user's public key can only be decrypted by the user holding the private key. If two parties have each others' public keys, they can encrypt a conversation using the public keys, and decrypt that conversation with the private keys that they each hold.

This is all cryptographically sound, *as long as the key exchange is performed in a secure manner*. But, aye, there's the rub: how do you exchange keys securely over an insecure medium open to **insertion attacks**? Unfortunately, there are few good answers, and none in wide use.

The Man In The Middle Attack

The man in the middle attack works like this:

Alice and Bob want to encrypt their chat session, but neither Alice nor Bob have each others' public keys. In order to communicate securely, Bob and Alice need to send their public keys (bob.pub and alice.pub) to one another. Once both hosts have each others' public keys, they can begin encrypting their packets, secure in the knowledge that only the intended recipient can decrypt the data.

However, imagine there is a user in the middle of the network, known as Eve. Eve wants to listen in on the conversation between Alice and Bob. In order to do this, Eve runs an application that accepts the real public keys alice.pub and bob.pub from Alice and Bob, but sends them both different *bogus* public keys, *bob1.pub* and *alice1.pub*. Bob and Alice don't know any better and will use the bogus public keys instead of the real public keys to encrypt packets for each other. Both Alice and Bob

believe that they are using the correct public keys, and expect to decrypt the messages with their own private keys. How can they use their original private keys to decrypt these bogus messages?

The answer is that Eve (as the "Man" in the Middle) captures and decrypts the packets with the private keys for the bogus bob1.pub and alice1.pub, inserts or merely eavesdrops on the connection, and reencrypts the data with the *original* public keys that Alice and Bob sent (alice.pub and bob.pub) and forwards the packets to their original destination. Bob and Alice decrypt the packets with their own private keys, and so don't realize that their connection has been hijacked because from their perspective, the session behaved **perfectly**.

Modern implementations of public key cryptography for network services rely on several different secure key exchange methods. By default, SSH will pass the public key to the client the first time in cleartext. During this first key exchange, the connection is vulnerable to a MITM attack, and if this is a serious concern, the key should be installed by hand from trusted media or encrypted with preexisting trusted keys before being sent over an insecure network.

After the initial key exchange, the public key is saved and associated with the IP address in the ssh configuration file ~/.ssh/known_hosts (or the Windows Registry if using PuTTY), so that if a different key is sent by a man in the middle (see the discussion of HTTPS below) or even if it is legitimately changed on the remote host it will be noticed and an alert will be generated. Compromise happens either by intercepting the initial exchange or by tricking a user into accepting an illegitimately changed key.

"Tricking the user" is easier than you might think, because due to configuration issues, sooner or later most users of ssh will encounter an unexpected key change. This is usually not a security issue and functionally speaking, the "right" thing to do is usually to ignore the warning and accept the new certificate. It is usually worth a little investigation, but most of the time the key change is either expected (e.g., server upgrade) or understood (e.g., erased known_hosts file). Sometimes a user can't be sure why the key has changed, but might be able to consider several plausible explanations that encourage them to accept the key without being *certain* what is happening. For better or worse, in the vast majority of common cases the user ends up correctly ignoring the warning, sometimes even facilitating the process by deleting an old key locally. This conditioning makes an attacker's job easier, because users have learned that key issues do not usually signal an attack.

Secure HTTP (HTTPS) uses a different strategy. Instead of hoping that the first transmission was secure or requiring the users to perform some out-of-band transfer, HTTPS key validation relies on "[trusted third party](#)" verification of public key certificates. This means that if you run a web server and you want to ensure visitors that the keys used by your website really belong to you, you buy a certificate from a trusted third party such as VeriSign or Thawte. This certificate lists the host, IP address, name, and other identifying information. When a user attempts to open a new SSL connection to your server, your browser will inspect the certificate to make sure that it is certified by a trusted party (by checking it against a list of parties that it is configured to trust) and that its identification lines up with the server on which it is being used. If it doesn't, the browser will complain and ask you whether to accept the certificate.

Unfortunately, while SSH's error message for key mismatch is loud and frightening ("**IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!**"), most web browsers key alert messages are apologetic and deferential by comparison. Most users simply accept the questionable key which may or may not be malicious. This may be related to the fact that signed web certificates are expensive and many non-commercial websites choose to use self-signed certificates instead of paying for third-party verification. Another common problem is the fact that an SSL certificates can be invalidated by

accident fairly easily (changing an IP address for example) and expire over time, or can be correct but lack the third-party endorsement. This "mostly correct" scenario *feels* more secure than one that is flagrantly wrong (e.g., one that says it's for "youarehacked.com" instead of "bankofamerica.com"), but it is *trivial* to create a certificate that has all the correct information (except the third party signature) yet is still totally bogus. What's worse, ARP Poisoning attacks can be used to intercept and modify the network transactions required to validate certificates, making forgeries appear authentic to the user and the computer.

Typical MITM attacks against cryptography **rely** on the attacker's *unexpected* ability to be interposed in the network and the haste and trust that most users have when dealing with computers. Furthermore, most users are not even *remotely aware* how likely this kind of attack might be performed and normally approve whatever is necessary to keep doing what they were doing, even if that undermines security. When an attacker is performing a MITM attack against a server with a legitimate SSL certificate, the end user usually only notices that there is "some kind of error" and they just click OK.

We will perform this kind of attack using [Ettercap](#).

Software Tools

tcpdump: sniff and analyze network traffic

[tcpdump](#) is the *de facto* network [packet sniffer](#) for most operating systems including many commercial unices, free Unix-likes, Apple's OS X, and even Microsoft Windows. Originally written at Lawrence Berkeley Laboratory (LBL), `tcpdump` is most frequently used for debugging networks and network applications because it gives a simple but comprehensive look into the traffic on an interface and requires no GUI.

It is quite common for a network segment to carry the traffic of many hosts. However, network cards typically only process network frames that are destined for their host. This means that network applications can normally only view the traffic addressed to that host, even though most of the time there is traffic belonging to other hosts on the same wire. If you are testing the behavior of a network application sending or receiving data to your host, this is appropriate.

However, Ethernet interfaces can be put into so-called promiscuous mode, where the network card passes *all* network data on the segment to the CPU and subsequently applications can see all network traffic on the interface. If a user wishes to sniff the network for more than their own traffic, the interface must be put into promiscuous mode, either by the application or by a network configuration utility (such as [ifconfig](#) on Linux).

`tcpdump` has various filtering and display options that you can investigate yourself, but the command you will probably find most useful for the course is:

```
$ sudo tcpdump -i ethN -s 1500 -X
```

This invokes `tcpdump` listening on ethernet adapter N, with a snaplength of 1500 bytes; using a smaller snaplen will not always show all the packet data (especially for large packets). The option `-X` decodes the data in hex and ASCII, similar to the display of hex editors like `hexdump`.

`tcpdump` will stream all decoded information to the terminal, too quickly to read. You can use Control-C

to kill `tcpdump` after enough time has passed and use your terminal's scrollbars to look at the output, use `less`, or use a terminal manager like `screen` for its scrollbar features.

You can also dump the capture to a special binary capture format with the `-w` switch:

```
$ tcpdump -i eth1 -s0 -w output.pcap
```

The capture file is used with a utility like [chaosreader](#).

You can also output to a text file (not a binary capture file) *and* the terminal simultaneously using the Unix utility `tee` like this:

```
$ tcpdump -i ethN -s 1500 -X | tee output.log
17:15:00.290656 IP laptop.local.39221 > server.somehost.net.ssh: . ack
2272 win 5002 <nop,nop,timestamp 327689043 282363814>

    0x0000:  4510 0034 e811 4000 4006 d56b 0a0a 0a58  E..4..@.@...k...X
    0x0010:  4051 2884 9935 0016 6f1c 67f9 7ce3 83e5  @Q(..5..o.g.|...
    0x0020:  8010 138a a57c 0000 0101 080a 1388 2353  .....|.....#S
    0x0030:  10d4 87a6
```

This output shows a packet en route from the host "laptop" from port 39221 to a server's ssh port (22). Following the header line is the decoded payload. It should be noted that `tcpdump` does not analyze the packet payload in order to determine the protocols shown in the header; it merely checks the port.

`tcpdump` also has powerful header filtering options and its own filter syntax that you can read about online or in the `tcpdump` manpage. But here's a simple example, showing only traffic to or from a specific host:

```
$ tcpdump -i ethN -s 1500 -X host 10.10.10.10 | tee output.log
```

Or a specific port instead (telnet traffic only):

```
$ tcpdump -i ethN -s 1500 -X port 23 | tee output.log
```

These filtering terms can be more expressive with the use of the Boolean operators AND, OR, NOT, etc. Please see the manpage or other online resources for more information.

Finally, if ARP spoofing has been set up (e.g. with `ettercap`), `tcpdump` is able to copy *all* the traffic sent across this interface (except decrypted SSL traffic -- see below).



[Wireshark](#) (formerly `Ethereal`) is essentially a very nice graphical application built top of `tcpdump`'s capabilities (using `libpcap`, the `tcpdump` library). If you are capturing packets in a graphical environment, `Wireshark` is highly recommended.

chaosreader: inspect and analyze `tcpdump` captures

chaosreader is an application that extracts TCP connections, names, passwords, files, etc. from inside network traffic logs as captured by [tcpdump](#). If you're curious how chaosreader works, edit the program with your favorite editor -- it's just a Perl script.

The easiest way to use chaosreader is to create a capture file with tcpdump and the -w option, and then feed that capture file into chaosreader with this command:

```
$ chaosreader output.pcap
```

This will create a set of HTML pages with all the relevant data in them. After running chaosreader, copy the files to your home computer (using `scp -r`) or use the text-only web browser elinks to navigate through the pages to find the information you want.

ettercap: play god on a switched network

[Ettercap](#) is a powerful tool for network analysis and security cracking. Ettercap has two main strengths:

First, Ettercap is capable of performing [man-in-the-middle](#) attacks on many protocols, including SSH1 and HTTPS. Ettercap is capable of "filtering" packets, where it intercepts, changes, and resends packets on the fly. This could be as simple as changing all occurrences of the word "buy" to "sell" in all web traffic, or as complex as exchanging the encryption keys in a secure transmission.

Secondly, Ettercap is capable of fully automating ARP spoofing (also known as ARP poisoning) which is a technique whereby a computer on a switched network is capable of forcing other hosts to send their data to the attacker rather than the actual network gateway or destination. This allows an attacker to force arbitrary users to send their data to him first -- after which he can choose to send the traffic on, change it, or throw it away -- all without any privileged position on the network other than the complete control of one workstation.

By the inclusion of ARP spoofing, expressive filters, and man-in-the-middle attacks, Ettercap is a one-stop-shop for many network attacks. Where such attacks used to require specialized software development (often customized for a particular network or attack), Ettercap is a user-friendly tool that makes network attacks incredibly simple.

Getting Started

To get started with Ettercap:

1. Start Ettercap: `sudo ettercap -C --` this starts Ettercap with the basic ncurses terminal interface.
2. Choose "Unified Sniffing" from the **Sniff** menu.
3. find the hosts on the network (**Hosts** menu)
4. select targets (Add hosts to a target by highlighting them and pressing 1 in the Hosts list.)
5. spoof their ARP tables (Under **MITM** menu)
6. and begin sniffing the network. (Under **Start** menu)
7. While sniffing the network, you can use `tcpdump` as described above to log the sniffed traffic.

SSL Attacks

Part of this lab includes performing MITM attacks against SSL encryption. To perform SSL decryption, you will need to modify ettercap's configuration file, which is located in `/usr/local/etc/etter.conf`.



Logging with SSL Attacks: `tcpdump` doesn't work to log SSL MITM attacks, because the decrypted data never passes through `eve`'s network device. (Think about why.) Just use copy and paste from your terminal (View | Connections in `ettercap`) to log decrypted traffic.

While Ettercap is basically "point and click," part of the this lab is learning and experimenting with these tools, so feel free to poke around and look at other resources. The beauty of doing this on a testbed is that you really can't break anything.

Interface Tips:

- Use 'Tab' and the arrow keys to change the window/menu you are interacting with.
- Use 'Control-Q' to close any active window in the Ettercap interface.
- There is more traffic than what you see in the "User Messages" window. Try looking under View | Connections and selecting a connection to view.
- To quit, press Control-X. This will "un-poison" the network and close.

Example Ettercap screen:

```
Start  Targets  Hosts  View  Mitm  Filters  Logging  Plugins  Help  NG-
0.7.3
└─Hosts
list...
|
|
| 10.10.10.10      00:13:D4:04:44:CA
|
| 10.10.10.11      00:14:6C:3B:F3:9D
|
| 10.10.10.99      00:08:74:12:19:22
|
|
|
|
|
|
|
|
|
|
|
```



```
|
|
|
└─User
messages:
|1698 tcp OS fingerprint
❖
|2183 known services
❖
|Randomizing 255 hosts for scanning...
❖
|Scanning the whole netmask for 255 hosts...
❖
|3 hosts added to the hosts list...
❖
|
◆
```

Ettercap Filters

Filters in Ettercap are tiny programs compiled by `etterfilter`, which are then interpreted by the Ettercap filter interpreter when loaded. The syntax is vaguely PHP-like so it is easy to use, but the debugging messages from `etterfilter` can be cryptic. You should **avoid** using the more complete regex filter functions, because their behavior is somewhat unpredictable. Instead, use `replace()` and other simpler functions.

Many students get confused when they do not see their filters being applied the way they expect them to. For example, students will set up a filter, tunnel in using `ssh`, and view the web page from their desktop computer. However, they won't see the changes from the filter. Why is that? This happens because when you tunnel into DETER, you are tunneling through `users.deterlab.net` to `eve`. And the network that `users` shares with `eve` is not the network being poisoned -- at least it shouldn't be -- you're poisoning the network that `alice`, `bob`, and `eve` share. Thus, 'users' (and you at home) do not see the effect of the filter. However, traffic between `alice` and `bob` should be being filtered -- you can see this using `tcpdump`, the connection viewer, or `chaosreader`.

For instructions on how to use them, see `man etterfilter` on DETER. There is also a sample ettercap filter in `/root/` on the `eve` host.

Ettercap Links:

- [SANS Paper on Ettercap](#) -- this paper is about an older version of Ettercap, but most of the

technical details are still accurate.

- [etterfilter tutorial](#)

Introduction

It turns out that the recent leak of some memos at FrobozzCo regarding imminent downsizing (due to insecure software and permissive filesystem ACLs) made the top brass are scared of malicious insiders. As a result, they've ordered a vulnerability analysis.

You've been hired at FrobozzCo ostensibly to replace one of the office assistants that has recently quit. None of the other employees know who you really are and think that you're just a temp. In reality, you've been hired by FrobozzCo's IT division to perform a [Red Team exercise](#) against a portion of the network.

Your assignment is to pretend that you are a disgruntled computer-savvy employee at FrobozzCo who recently discovered that your job is going to be eliminated. Since your job is over, you're supposed to steal as much information as you can and create as much chaos as possible; in other words the brass want to know how much damage a sufficiently motivated insider could do.

You have a personal computer on the company LAN, which is a switched network. You share a network subnet with several computers that have some important responsibilities. The brass fear that someone could do a little corporate espionage on their way out the door and take it to Zembor Corporation, FrobozzCo's biggest competitor.



WARNING: Red Team exercises *are* performed as a part of vulnerability analyses. However, they are only performed with express, written permission, non-disclosure agreements, and well-defined boundaries. Doing this without permission is illegal and carries **severe legal penalties**. So don't try this at home.

Assignment Instructions

Performing this task and all subsequent tasks requires you to successfully perform an ARP Spoofing attack against the other nodes in your network using [Ettercap](#). Other tools that might be useful for this task include [tcpdump](#) and [chaosreader](#).

Setup

1. If you don't have an account, follow the instructions in the [introduction to DETER](#) document.
2. Log into DETER.
3. Create an instance of this exercise by following the instructions [here](#), using `/share/education/MITM_UCLA/mitm.ns` as your NS File.
 - In the "Idle-Swap" field, enter "1". This tells DETER to swap your experiment out if it is idle for more than one hour.
 - In the "Max. Duration" field, enter "6". This tells DETER to swap the experiment out after six hours.

4. Swap in your new lab.
5. After the experiment has finished swapping in, log in to the node via ssh.

For this lab, you will only have console access to the host `eve.mitm.UCLAClass.deterlab.net`; the hosts `alice` and `bob` will at be configured so that you can't log into them (at least to start!).

It is **very possible** that actions you take in the course of this lab could kill important processes, erase files, or generally cause other havoc in the system. If you think that you have "broken" your lab nodes, you can always reboot the nodes, or if things are *really* screwed up, swap out the experiment and swap it back in to start fresh. As always, anything you save in your group directory on DETER will stay there -- make sure you save any important work somewhere safe.

Tasks

1. Eavesdropping

Your first task is to eavesdrop on all cleartext network traffic and document what the different traffic streams are communicating.



Do not ARP Poison any interface from `/var/emulab/boot/controlif` or starting with `192.168.*.*` -- that is DETER's control network!

Eavesdropping Tasks

Answer these questions as completely as you can:

1. What kind of data is being transmitted in cleartext?
 - What ports, what protocols?
 - Can you extract identify any meaningful information from the data?
 - e.g., if a telnet session is active, what is happening in the sesion? If a file is being transferred, can you identify the data in the file?
 - Make sure you eavesdrop for at least 30 seconds to make sure you get a representative sample of the communication.
2. Is any authentication information being sent over the wire? e.g., usernames and passwords.
 - If so, what are they? What usernames and passwords can you discover?
 - **Note:** the username and password decoding in ettercap is not perfect -- how else could you view plain text authentication?
3. Is any communication encrypted? What ports?

2. Replay Attack against the Stock Ticker

You know that stock information is communicated over your network and logged on an internal server. It takes a feed from New York that updates the current price. The local server shows the current price and all previous prices since the last reload. This local server is the information that all internal company brass use for their reports and personal stock purchases. You also know that the stock program uses some kind of encryption, but you think it may be vulnerable to replay attacks because it was written by the CTO's 13 year old nephew who [doesn't have the greatest](#)


[security track record](#).

Can you perform a replay attack against the stock ticker in order to make FrobozzCo's stock look bad and Zembor Corp's stock look good? You like imagining the internal chaos this would cause.

To do this, you will need to:

- set up ARP spoofing with Ettercap
- capture traffic with `tcpdump`
- analyze the captured traffic
- generate new requests to the stock quote service by "replaying" old requests.

You will probably find it easiest to create URL-based replay attacks using `elinks`, `wget`, or `curl` (all text-based web clients). Likewise, you'll probably want to [set up port forwarding](#) so you can play with the application in your desktop web browser.

 In this way, you are "replaying" the remote procedure call (RPC), but not the actual TCP packets. In reality, there is software that can be used to actually replay real TCP sessions back on to the network, but it is too complicated for this lab. (What about TCP packets makes replaying difficult?)

Replay Tasks


Once you figure out how to create the replay attack, include the following information in a report:

1. Explain exactly how to execute the attack, including the specific RPCs you replayed.
2. Explain how you determined that this strategy would work.
3. Execute your replay attack and show the results of your attack with a screen capture, text dump, etc. showing that you are controlling the prices on the stock ticker.

3. Insertion Attack

Ettercap has the ability to execute [regular expression](#)-like filters on cleartext traffic. This allows you to change the contents of packets as you sniff them. To do this, you need to create ettercap filter modules and load them into a running ettercap that is performing ARP spoofing.

For this part of the lab, you'll write some etterfilters, compile them, and load them into ettercap so you can use them on the network stream.


 etterfilters can be used to intercept and change traffic in either or both directions, but only between `bob` and `alice`. This means that you can change a request going to the server, and/or you can change the results that ticker users see when they view the ticker (by changing outgoing results). However, changing requests *to* the server is hard because of the nonce in the stock ticker request. This nonce hinders replay attacks and thus makes it difficult to correctly modify requests that are *inbound* to the

ticker. However, but since the ticker replies in cleartext, it's easy to change outgoing results! Again, remember that requests from `eve` will not be modified -- so "testing" from `eve` with `elinks` or a tunneled connection will not see the changes.

Insertion Tasks


For these filters, you won't change the data that's on the stock server -- just what it looks like to someone viewing the page.

1. You can change the symbols a viewer of the ticker sees by intercepting the HTML bound for their browser. Write a filter to change the symbol FZCO to OWND.
2. Write a filter to affect the *prices* a user of the stock ticker sees.
3. Include your filter sources with your submission materials.
 - Make sure you comment your code (use the `#` character) to explain what the filter does and how.

 **Warning!** It's hard to test whether your filtering setup is working, because `ettercap` is only filtering traffic to and from `alice` and `bob`. If you run `elinks` on `eve`, or forward a port through `eve` with `ssh` tunneling, the traffic you see as a viewer will not be modified! You *can* run `tcpdump` on `eve` to see the modified packets leaving `eve`, but only the outgoing packets will be modified.


For these questions, you don't need to write filters, just write short answers or pseudocode explaining how you would do it.

1. Given the power of `etterfilter` and the kinds of traffic on this network, you can actually make significant changes to a machine or machines that you're not even logged in to. How?
2. Of the cleartext protocols in use, can you perform any other dirty tricks using insertion attacks? The more nasty and clever they are, the better.

 **Hint:** There is a sample `etterfilter` source file in `/root/` on the host `eve`. You can use this filter as a starting point for your own filters. See `man etterfilter` and `irongeek`'s `etterfilter` tutorial in the "starting points" section below.

4. MITM vs. Encryption

There is at least one encrypted network stream in use on your network subnet. Lucky for you, your coworkers blindly click OK any time a certificate error pops up on their screen -- you know this because they complain loudly every time it happens -- so you are hopeful that a man-in-the-middle attack against the certificates would be successful. What's so important about that encrypted stream -- what is the data being transferred?

 Ettercap will not perform SSL decryption automatically -- you have to uncomment a few configuration items in `/usr/local/etc/etter.conf` in order to properly forward the data. Examine that file, use any available documentation and make the necessary changes so that you can man-in-the-middle their SSL connection.

When you quit using Ettercap after having made these changes, it will make some complaints -- don't worry about them.

MITM Tasks

1. What configuration elements did you have to change?
2. Copy and paste some of this data into a text file and include it in your submission materials.
3. Why doesn't it work to use `tcpdump` to capture this "decrypted" data?
4. For this exploit to work, it is necessary for users to blindly "click OK" without investigating the certificate issues. Why is this necessary?
5. What is the encrypted data they're hiding?

What can go wrong

Specific problems:

- Make sure you swap out your nodes when you are done working.
- If Ettercap doesn't seem to be working, make sure you are poisoning the correct interface. You should be poisoning the 10.x.x.x interface.
- If Ettercap still won't work, try rebooting your nodes (you don't have to swap out yet).
- If it's still not working, try swapping the nodes out and back in. (Save your work.)
- Keep your etterfilters simple -- the etterfilter scripting language can be difficult to work with.

Other tips:

This assignment is left intentionally open-ended -- rather than going into great depth about how to use tools like Ettercap, `tcpdump`, `chaosreader`, and others, we're leaving it up to *you* to do the external reading and experimentation required. We've also provided a [challenge problem](#) if the regular assignment is too easy.

You should take notes on everything that you do, including taking terminal logs or copying character data from the terminal. In a very real sense, this activity is the opposite of computer forensics, and both require lots of notes in order to ensure repeatability. In a real Red Team exercise, complete and thorough documentation is a must.

A few starting points:

- [SANS Institute Ettercap Tutorial](#) -- this is for an older version of Ettercap so don't be confused.
- [Ettercap Homepage](#)
- [etterfilter tutorial](#)

Good luck!

Extra Credit

You have a *powerful suspicion* that the encryption token used in the stock ticker application is not particularly strong.

Extra Credit Questions

Answer one or more of the following to receive extra credit points.

1. What *observable software behavior* might lead you to believe this?
2. Can you reverse engineer the token? How is the token created?
3. If you can reverse engineer it, can you write a script in your favorite language to post data of your choice?
 - Hint: all the necessary pieces are available on the servers for both Perl and bash.
4. What would be a better token? How would you implement it on both the client and server side?

Submission Instructions

Create a single .pdf, .doc, or .txt with all your answers, your etterfilter source files, and anything else you think should be included etc., and submit it to your instructor. Make sure to double check that you have answered all the questions.