Author: Wong Tin Kit
Student ID: 1003331

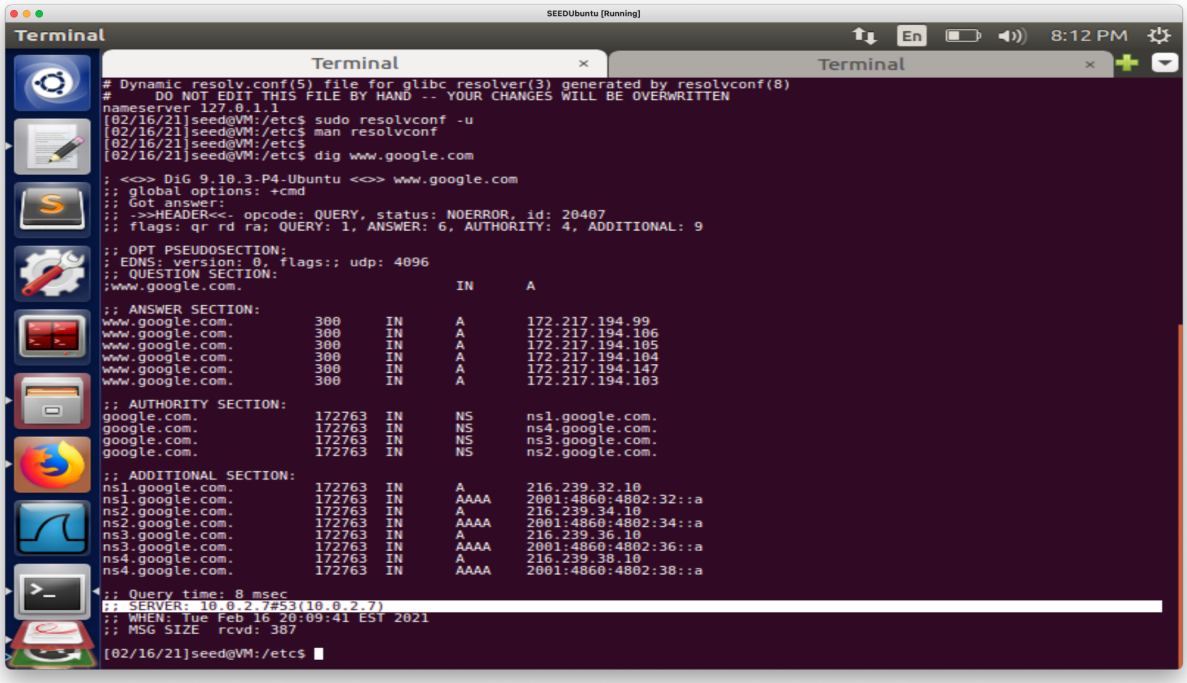
IP Mapping

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



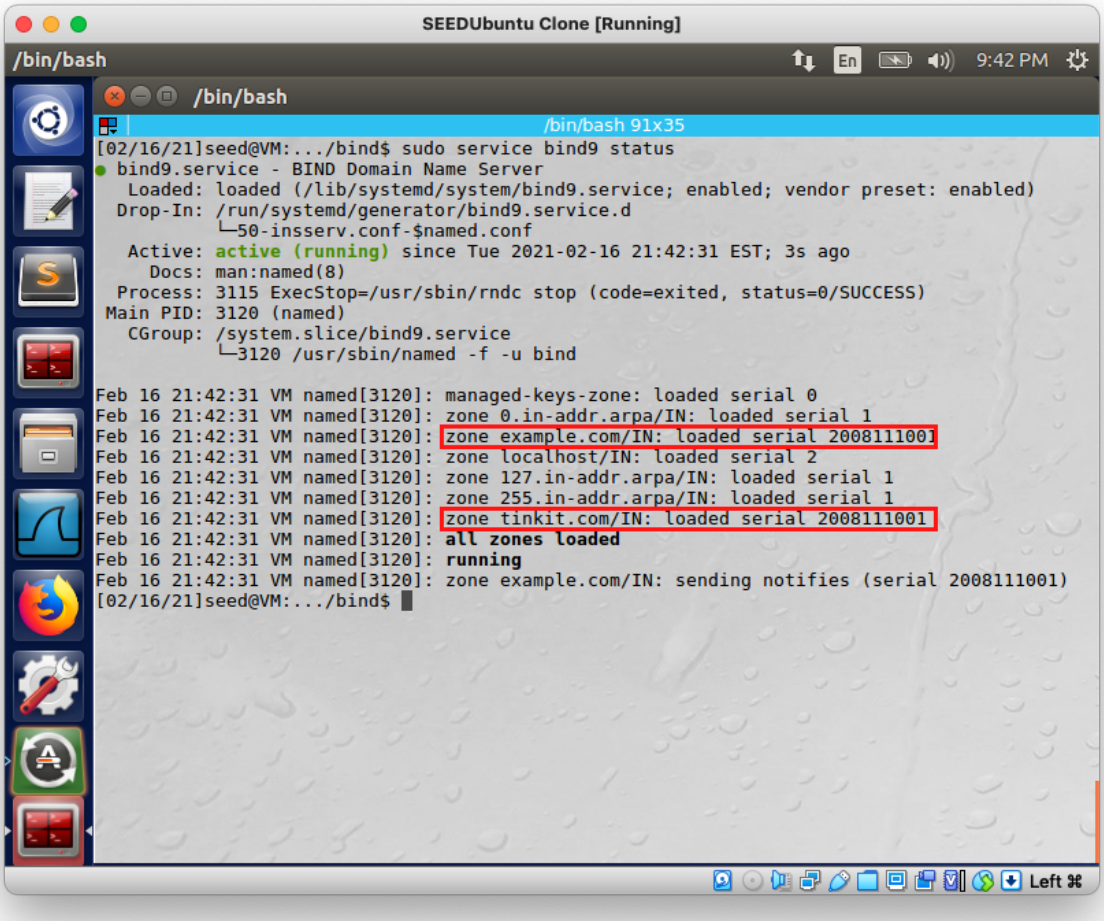
The highlighted line in the following ***dig*** command ***dig www.google.com*** shows that the local DNS has been properly configured. The highlighted output is as such
;; SERVER: 10.0.2.7#53(10.0.2.7)

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

Nothing to show.

Task 3: Configure the Attacker VM

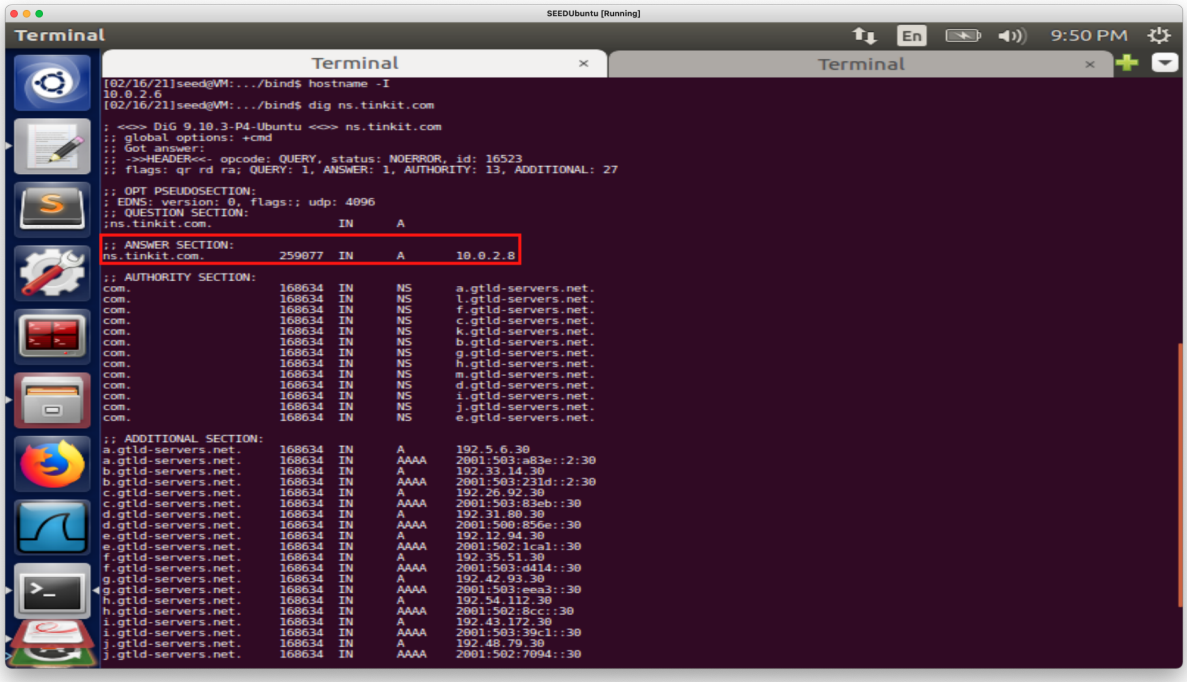
https://docs.oracle.com/en-us/iaas/Content/DNS/Reference/formattingzonefile.htm was a good read when configuring Attacker VM's zone files.



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

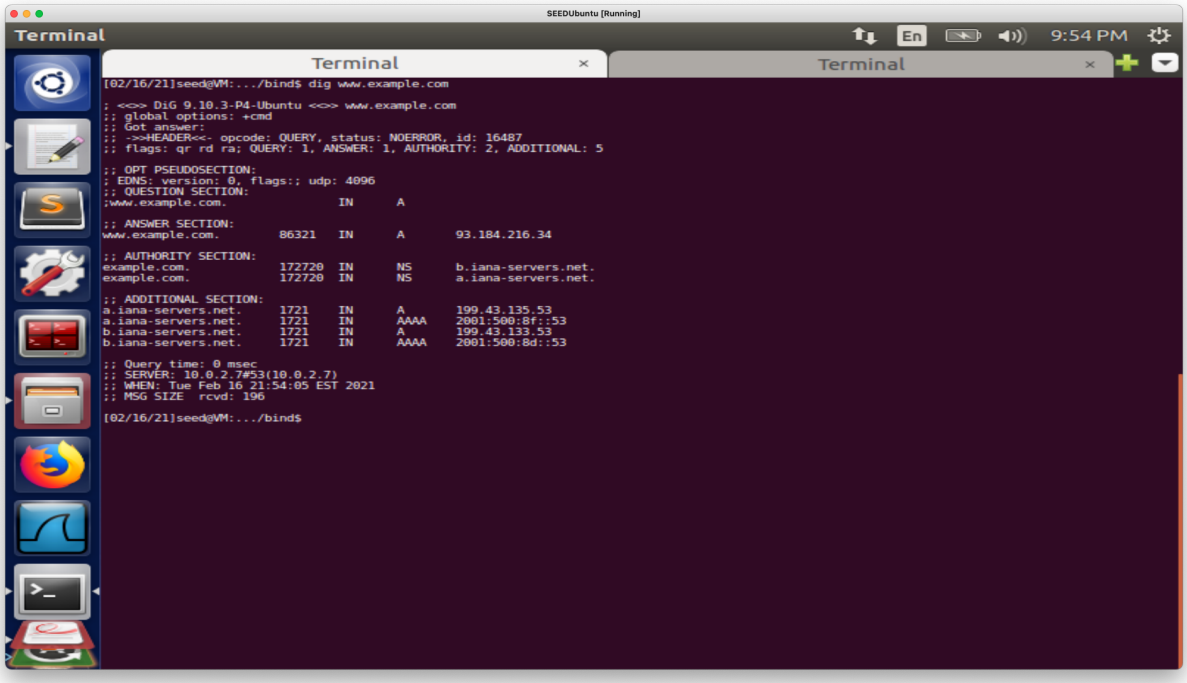
Task 4: Testing the Setup

Get the IP address of **ns.tinkit.com**.

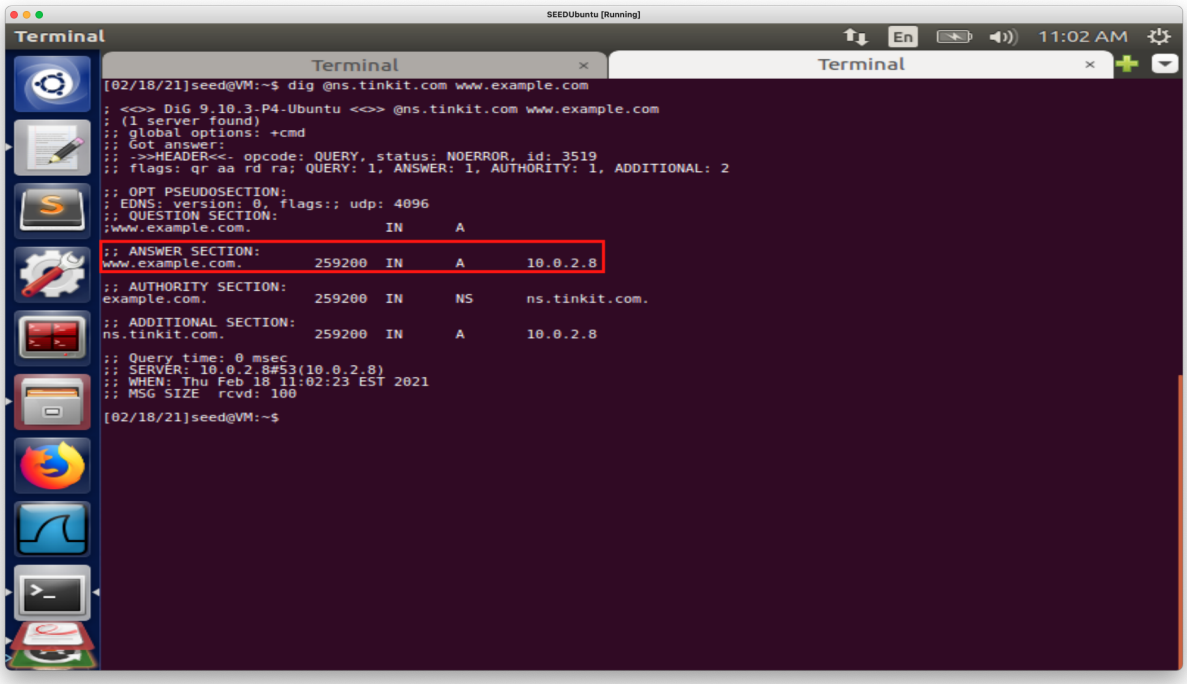


The command ***dig ns.tinkit.com*** was run on the User VM of the IP address ***10.0.2.6***. We see the answer came from ***tinkit.com.zone*** file in the Answer Section (highlighted box).

Get the IP address of www.example.com.



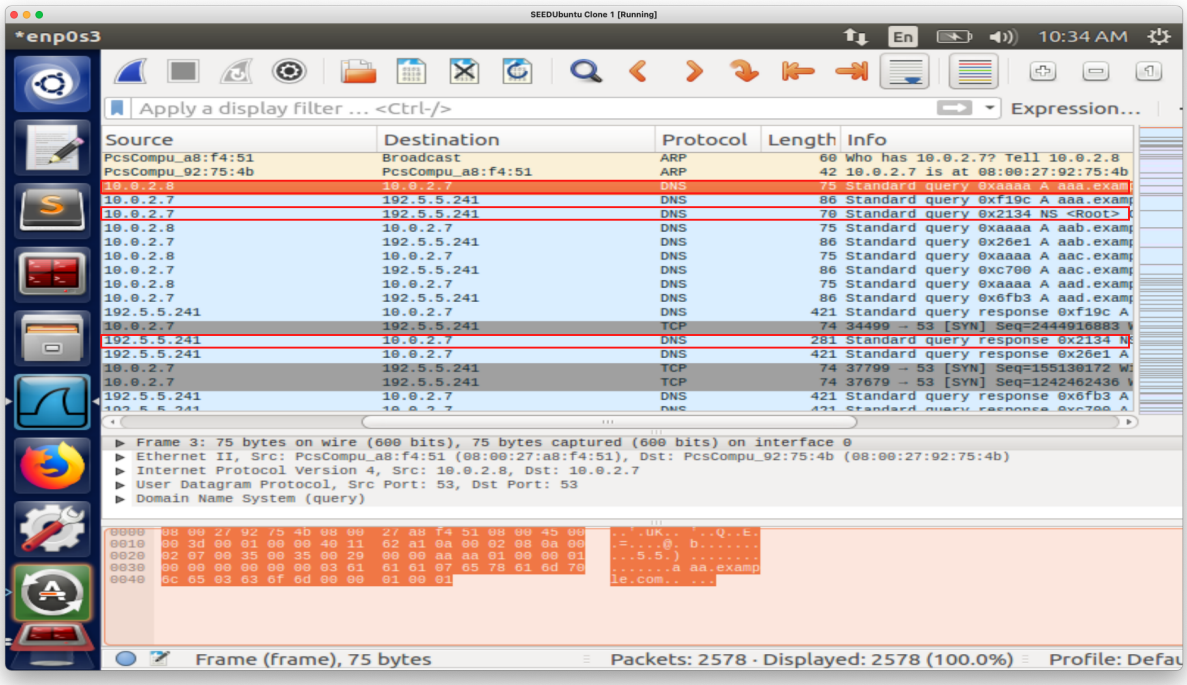
The above image shows the output of querying ***example.com's*** official nameserver. We can be sure because of the public IP address returned in the answer section.



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

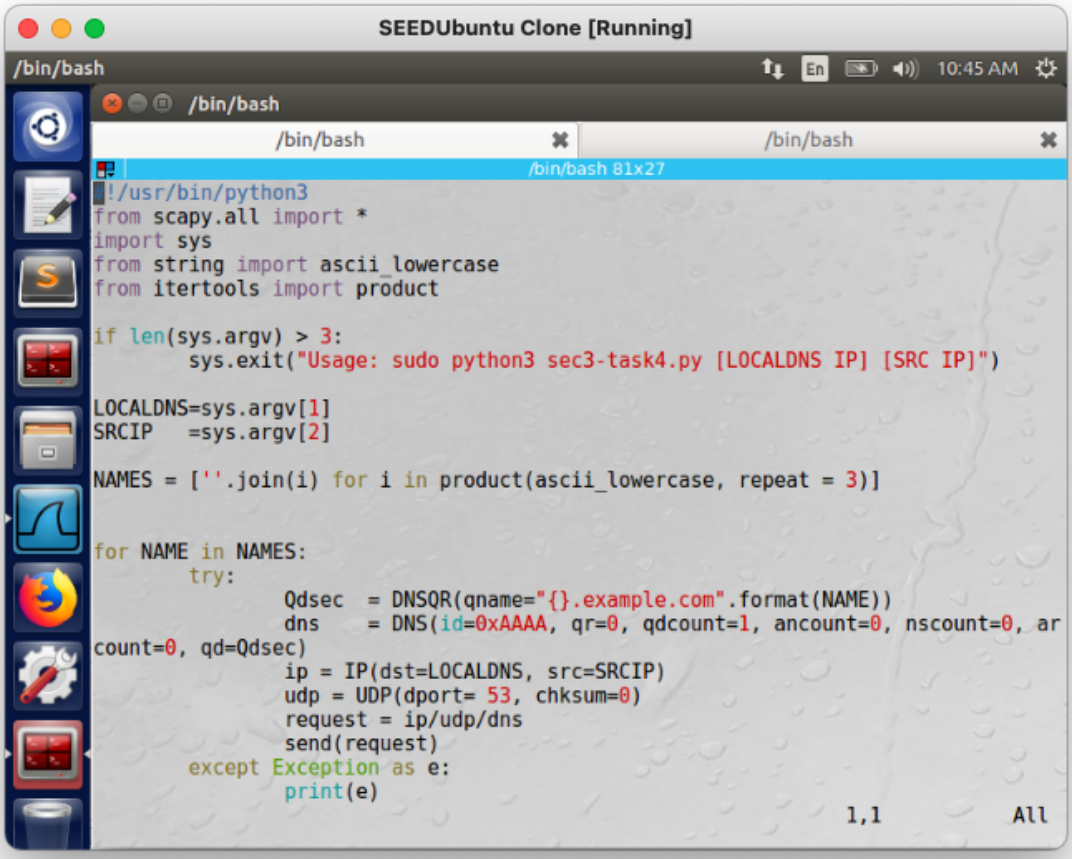
Task 4: Construct DNS request



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***.

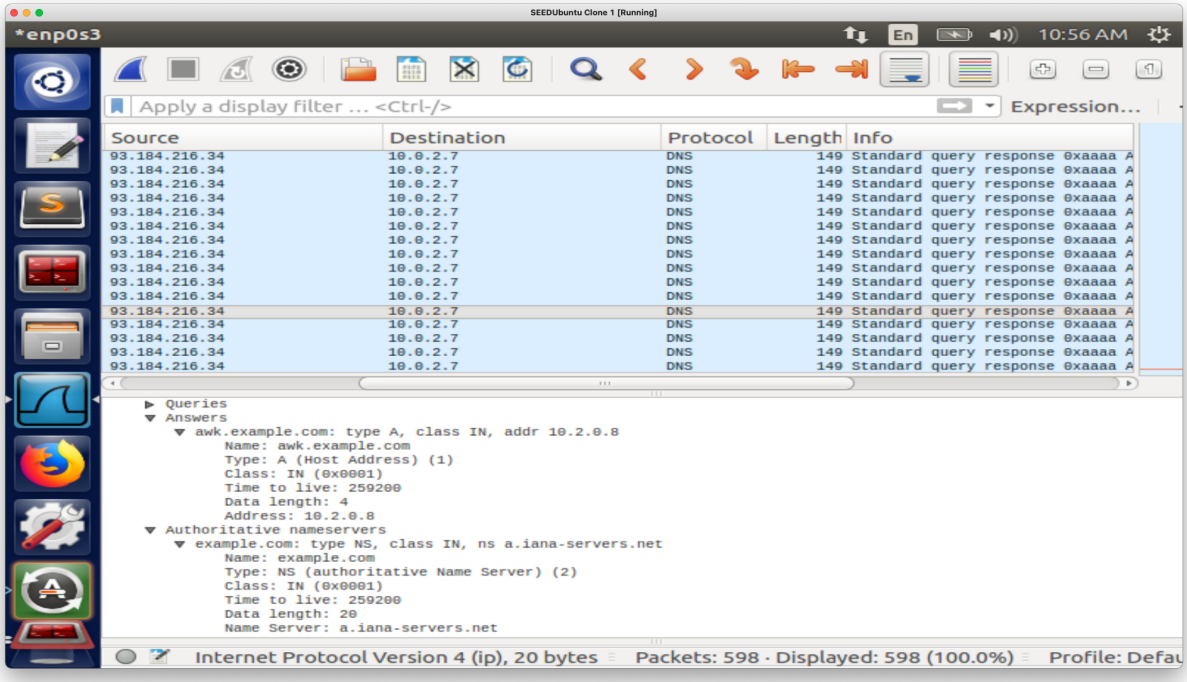
We can be sure that the attack script works as this pattern of sending out DNS queries from the local DNS Servers to the Root DNS Server as well as getting the corresponding DNS Response is seen in the above image. The DNS Responses as well as newer DNS Queries are in the later

parts of the wireshark capture.

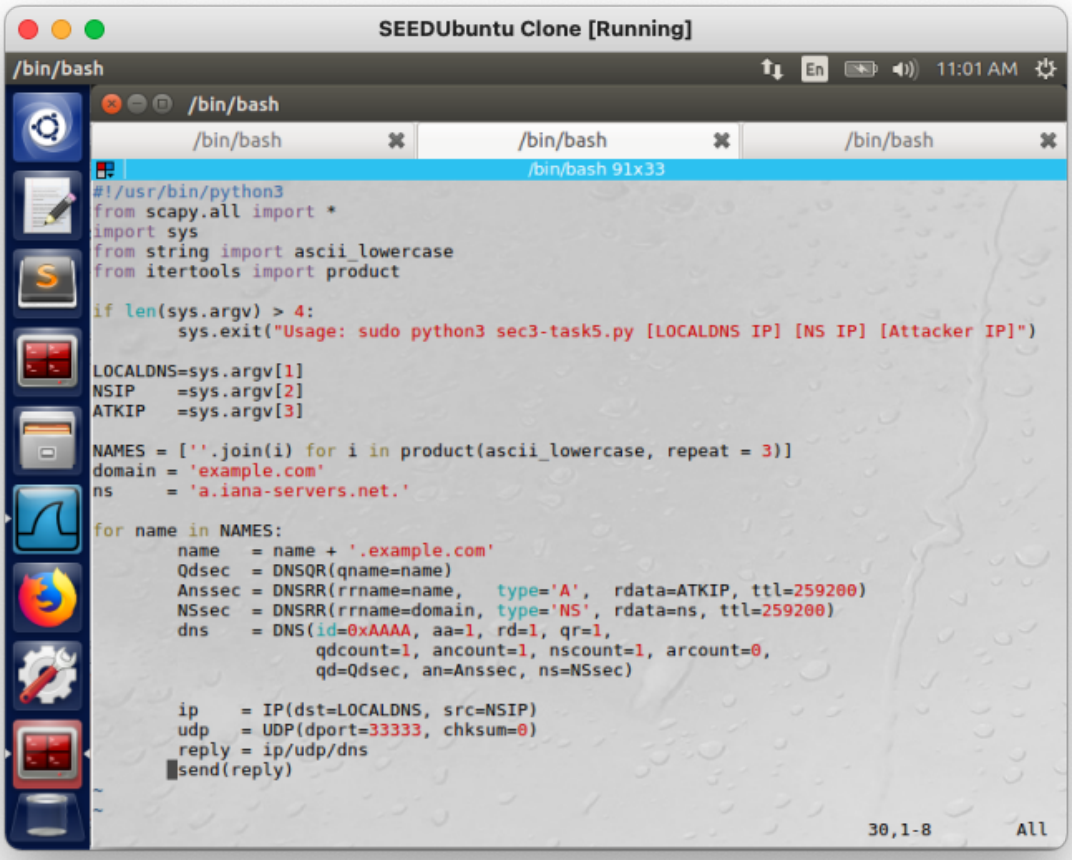


The above image shows the code used by the attacker to automate the DNS queries for Apollo. We use the script with the following command ***sudo python3 sec3-task.4py 10.0.2.7 10.0.2.8***. We first generate all possible combinations (17576) of a 3 character string and store it as a list called ***NAMES***. 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 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.

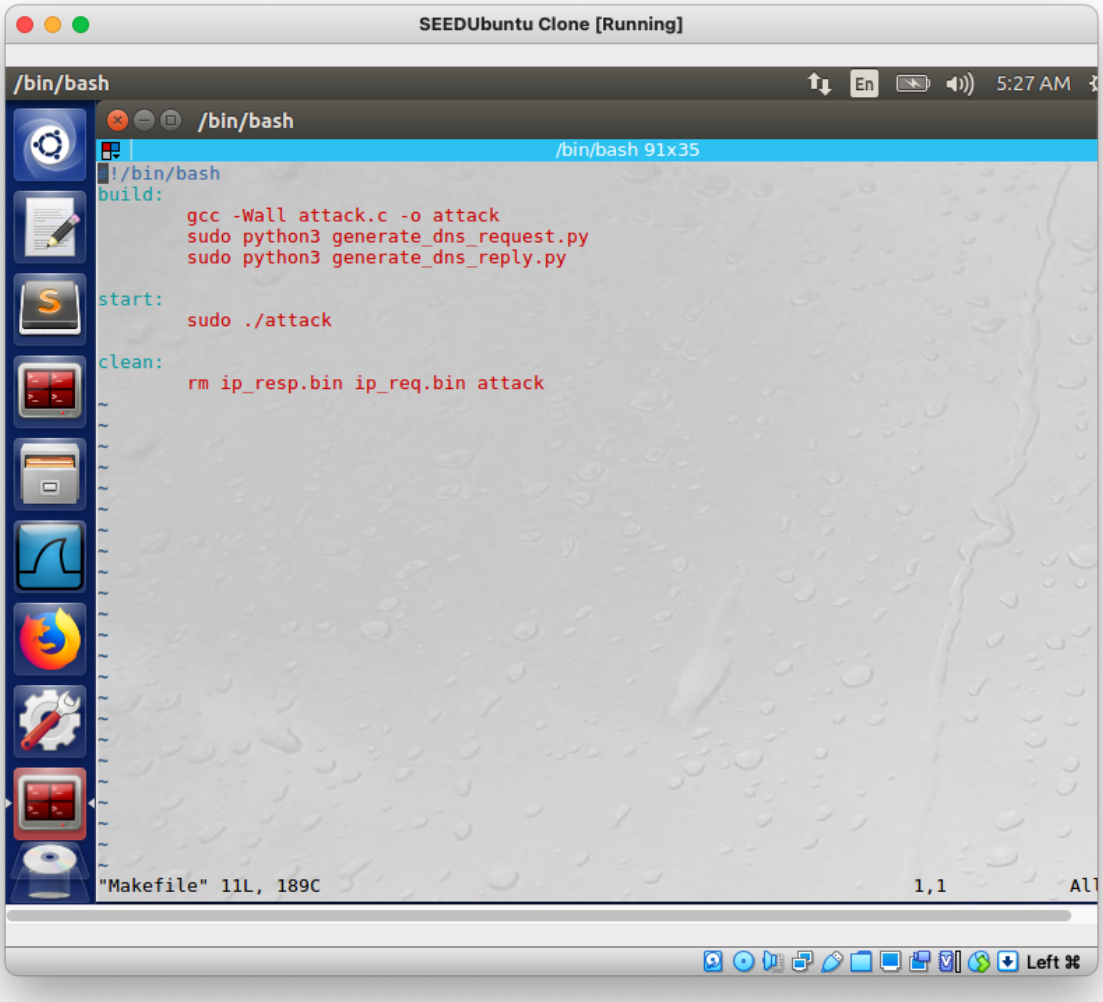
DNS(rd=1)	This bit directs the name server to pursue the query recursively.
DNS(qr=1)	Qr bit set to 1 to indicate a DNS Response Packet.
DNS(qdcount=1)	Setting qdcount=1 specifies only 1 entry in the Question Section which is true for each of our DNS Query (***.example.com*)
DNS(ancount=1)	Setting ancount=1 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 the generate DNS Query, Response binary files as well as executable ***attack.c***. Run this on the Attacker VM.

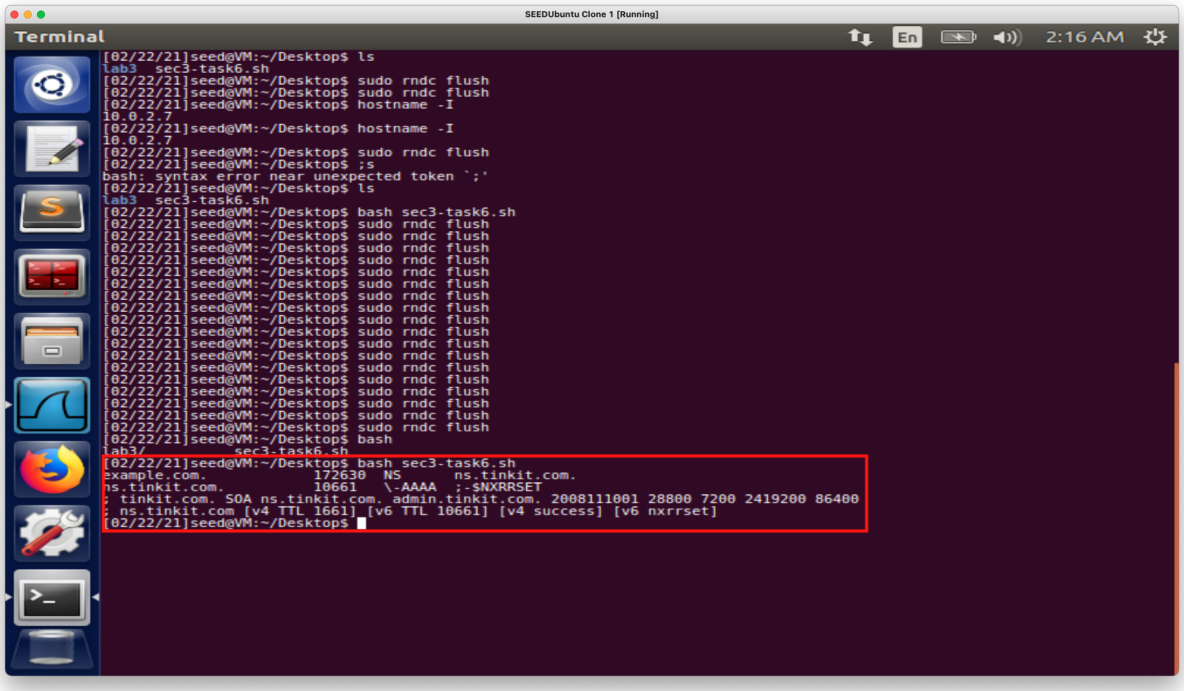
1. ***make clean***
2. ***make***
3. ***make start***
4. ***Enter the total no. of DNS Queries sent by script. (ps. I used 1000 so I don't need to keep trying.)***

Below is the image of the Makefile.



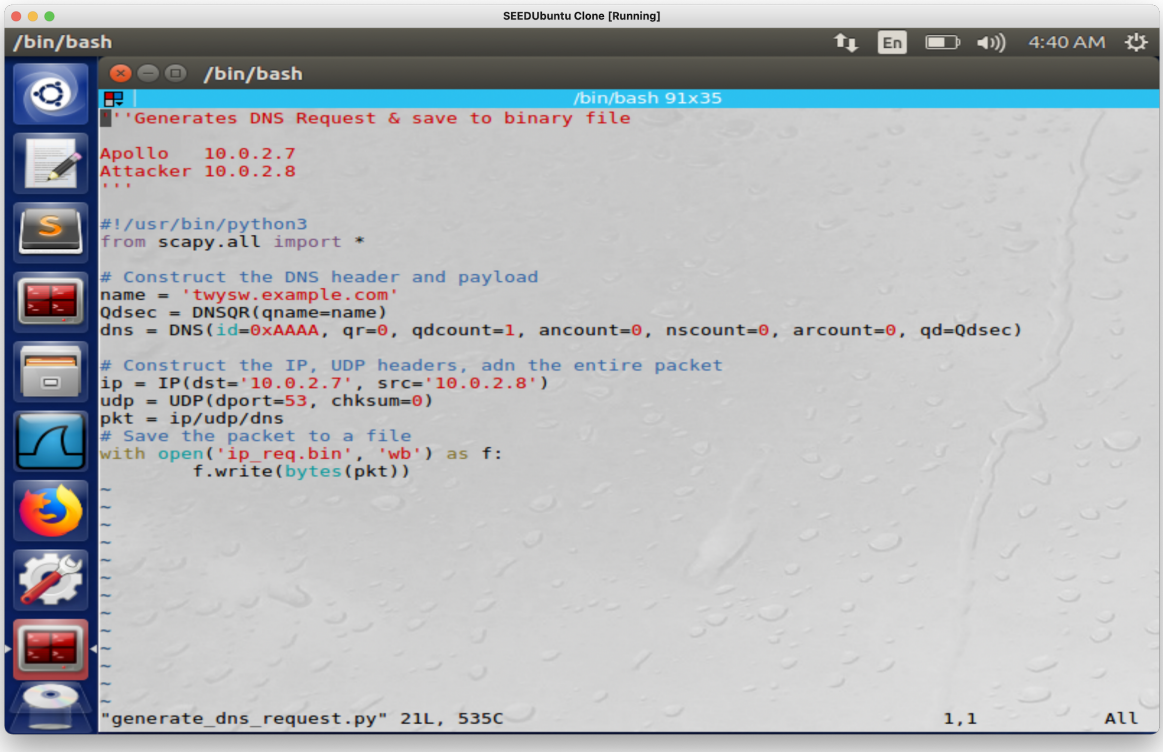
Do note that for each DNS Query we send, we will send 1000 DNS Responses.

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.

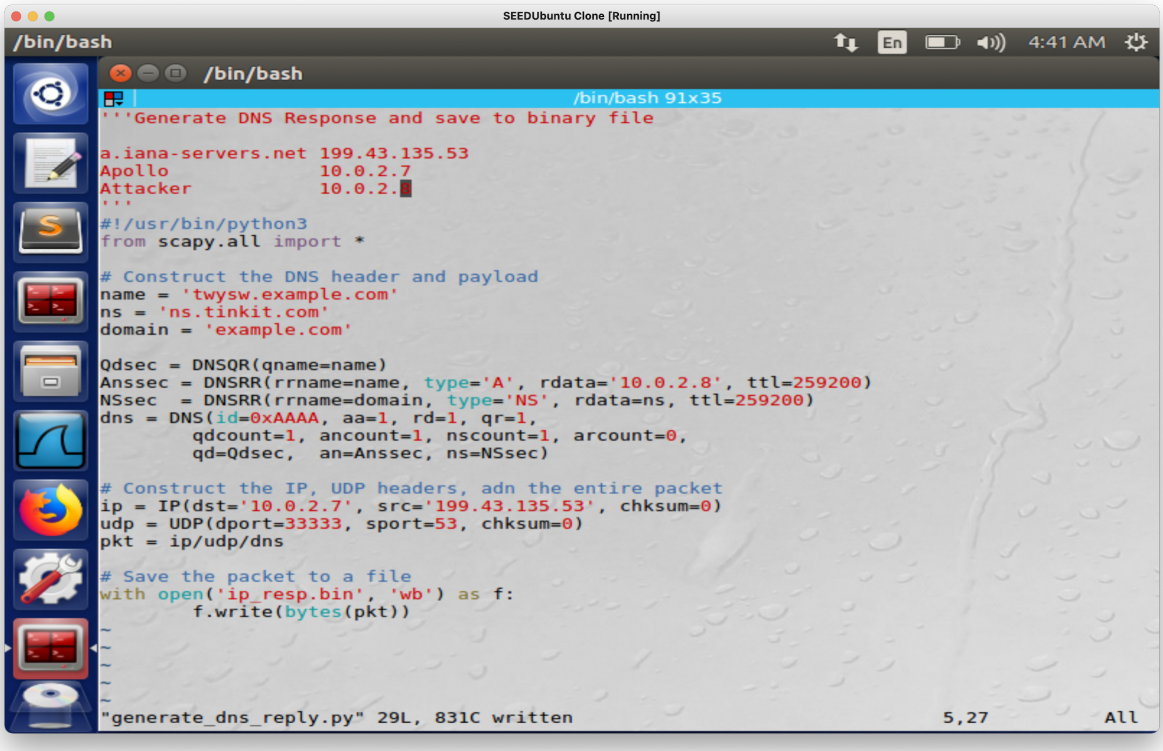


The above image shows the Kaminsky Attack has been successful. I will explain the c code and python scripts used to achieve the above results. This task suggested first to create DNS query

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



The 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.

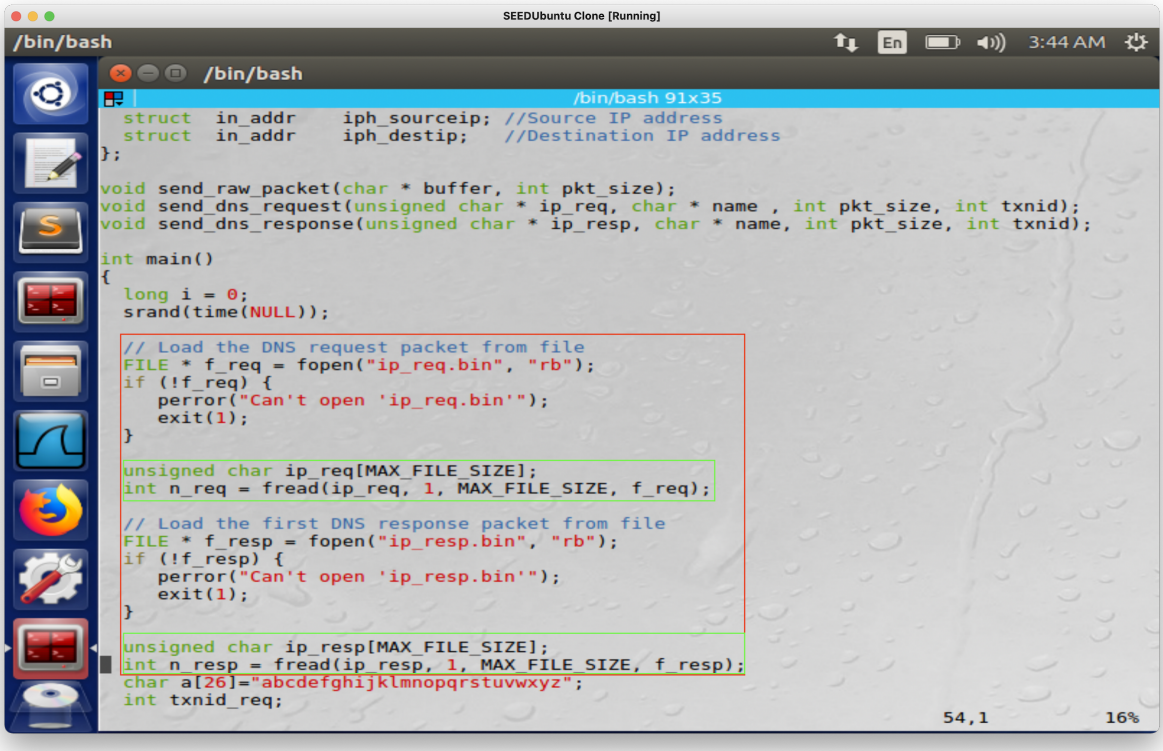


The above image shows the code ***generate_dns_reply.py*** used to construct a DNS Response Packet with scapy. ***qr=1*** confirms that it is indeed a DNS Response Packet. This is similar to the code used in ***sec3-task5.py*** (Section 3 Task 5). Please refer to the answer in Task 5 for a detailed explanation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

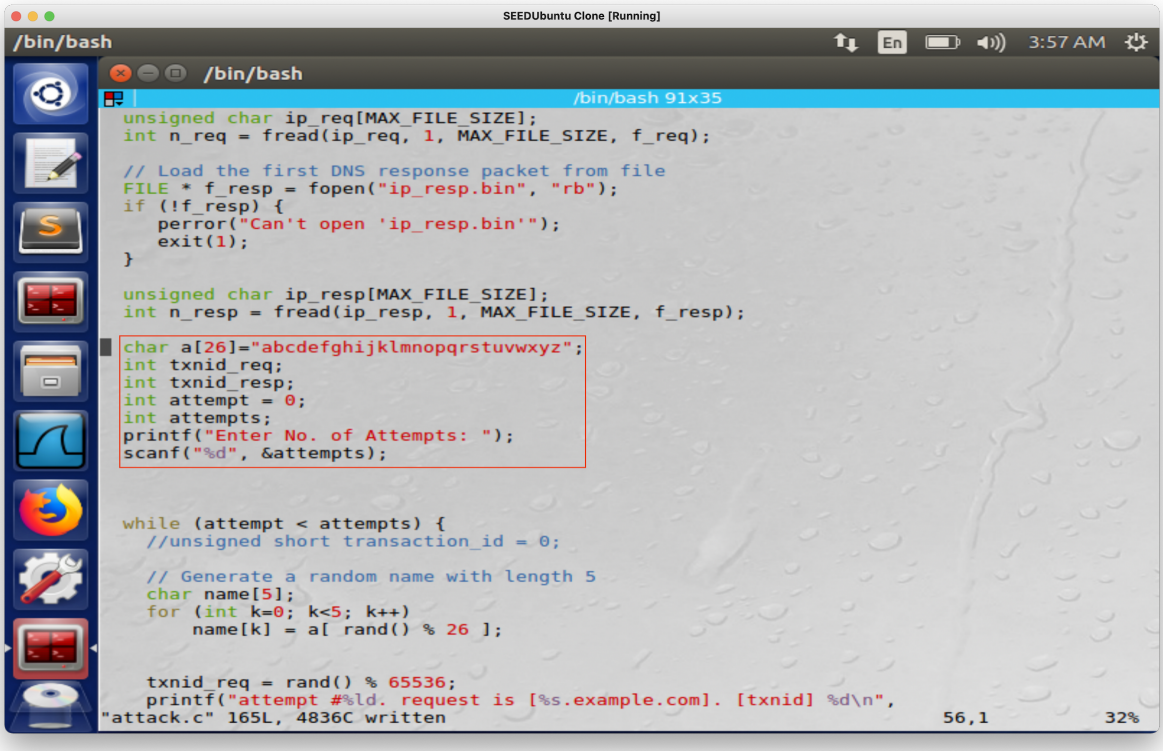
To run this file, type ***sudo python3 generate_dns_reply.py***.

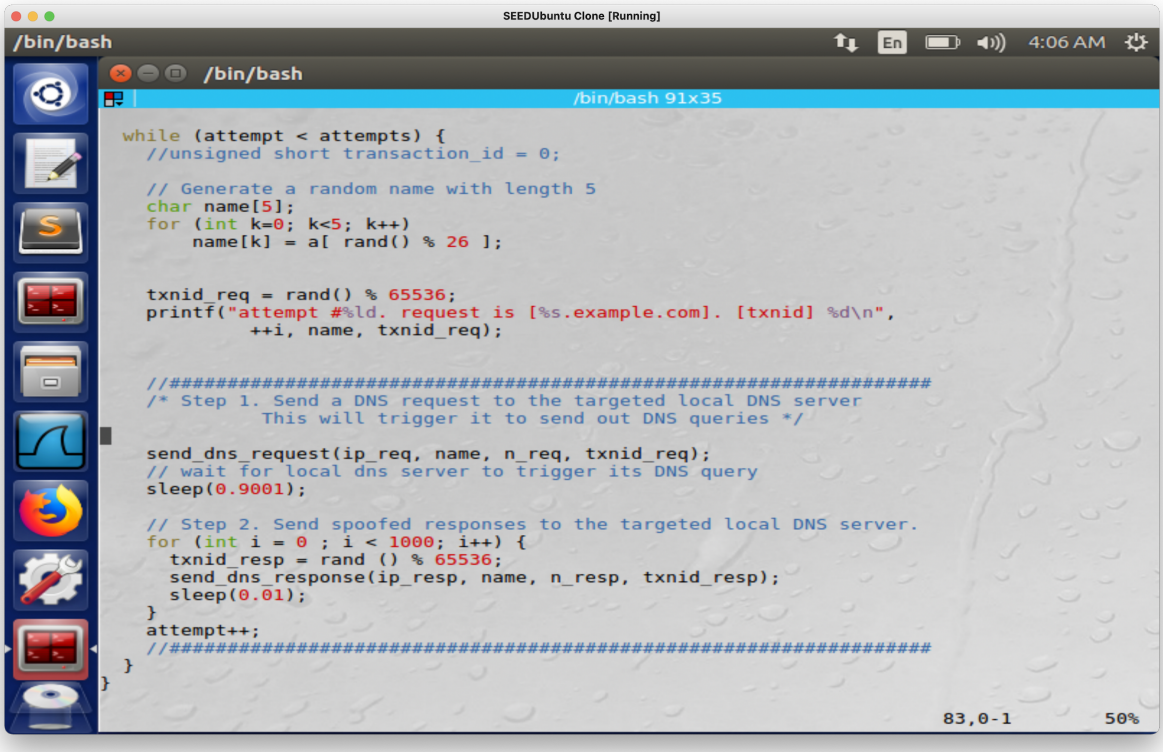
Now we have both the spoofed ***DNS Query*** and ***DNS Response Packet templates.*** We can read it into ***attack.c*** as shown below and we assign them to their respective file pointers ***FILE * f_req*** and ***FILE * f_resp*** (red highlighted box) before we read it into a buffer accessible by the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



The highlighted box below initialises the following variables:

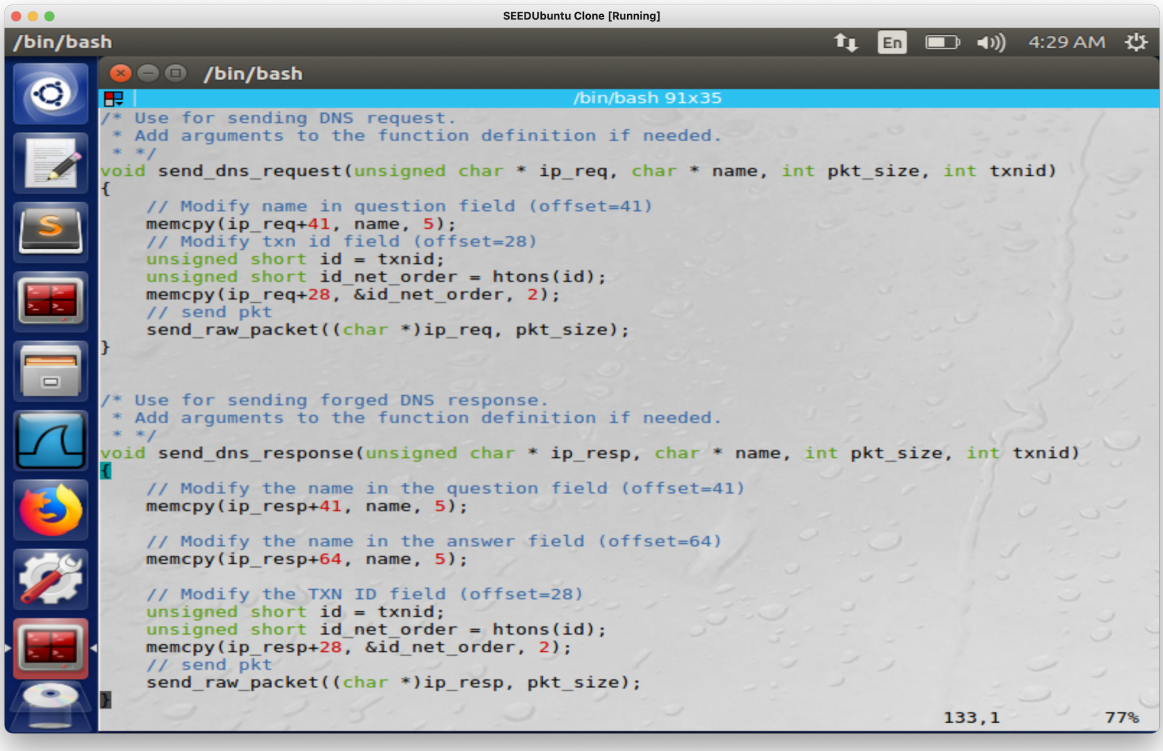
1. Character array ***a[26]*** of size 26 holding all 26 alphabets.
2. Integer variables ***int txnid_req***, ***int txnid_resp***, which are the Transaction IDs for DNS Queries and DNS Response Packets.
3. Integer variables ***int attempt*** and ***int attempts***
    a. ***attempt*** defines the current 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
En
3:57 AM
/bin/bash
/bin/bash 91x35
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",
"attack.c" 165L, 4836C written
56,1
32%


SEEDUbuntu Clone [Running]
/bin/bash
En
4:06 AM
/bin/bash
/bin/bash 91x35
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.



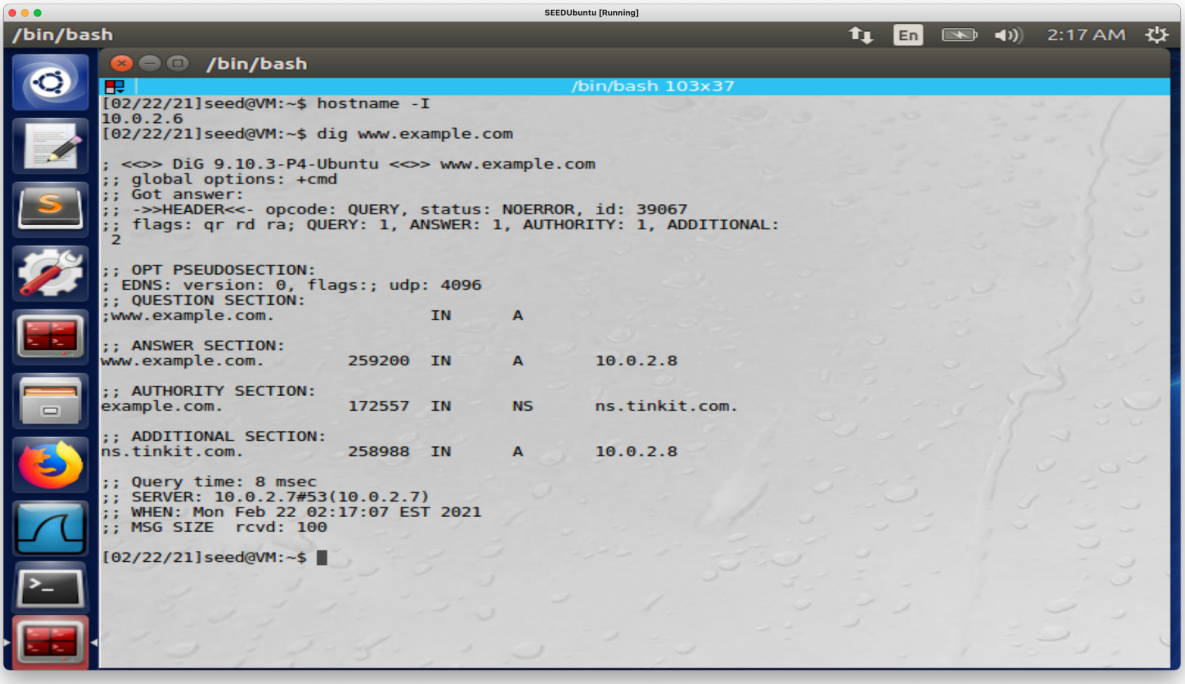
The above image shows the code snippets used to send spoofed ***DNS Queries and DNS Responses***.

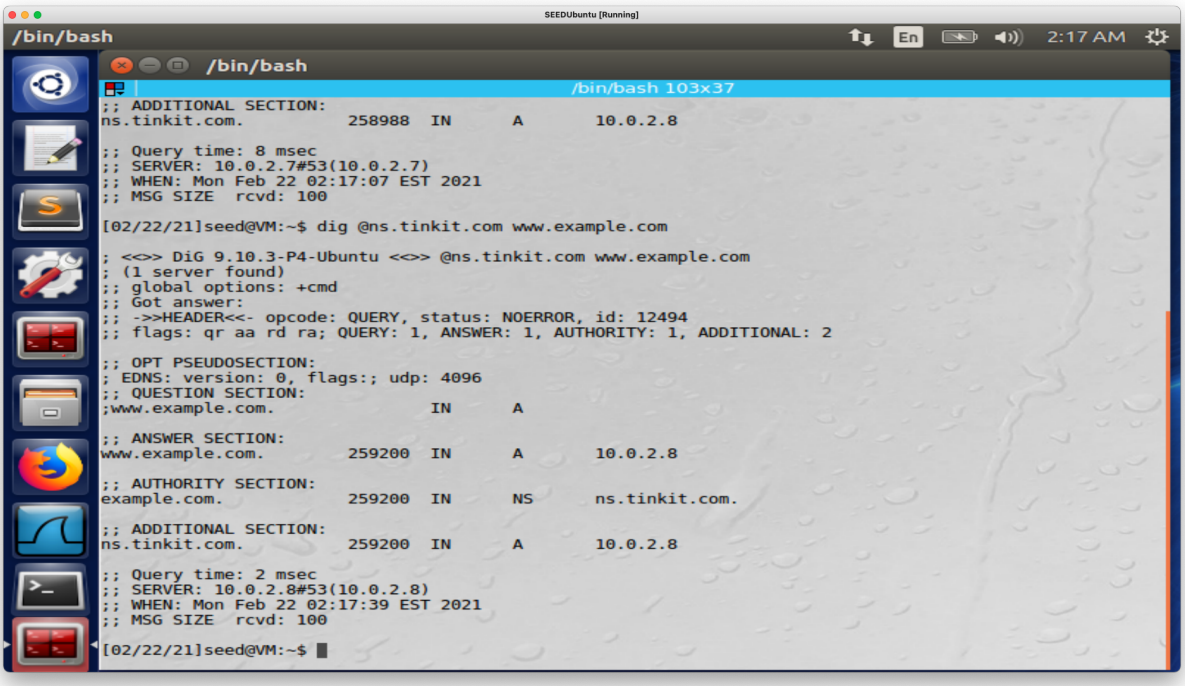
With bless (binary editor), we are able to find the offsets of the required values to change:

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

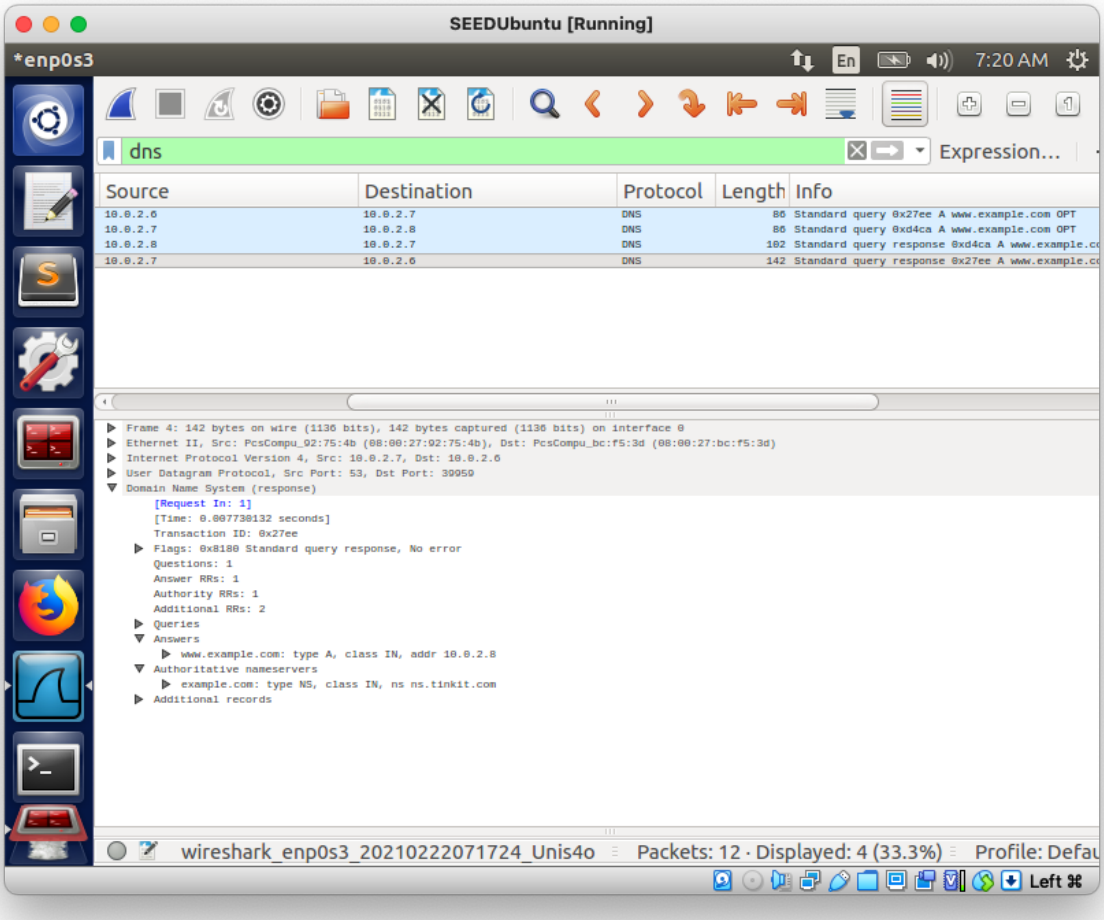
Since we do not have an answer section in a DNS Query, we do not need to change it there as shown in the ***send_dns_request***. After modifying the relevant bytes, both functions will send the updated packet by calling ***send_raw_packet()***.

Task 7: Result Verification

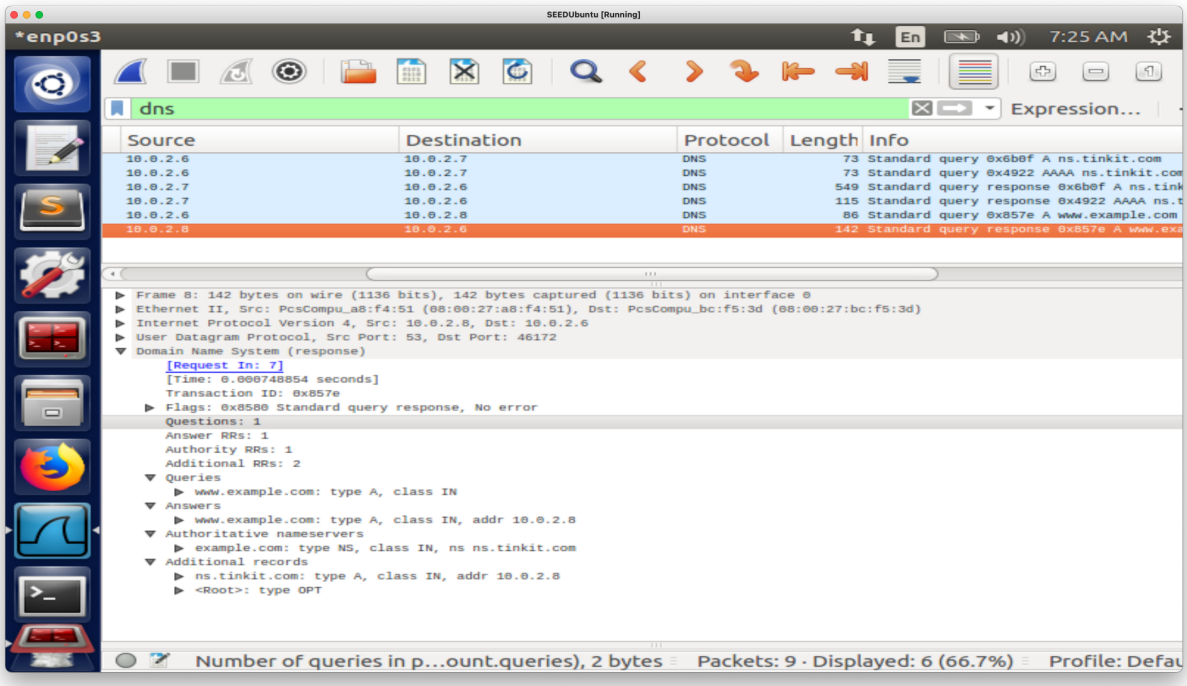




The above 2 images verifies the attack was successful as ***example.com's NS*** was mapped to ***ns.tinkit.com*** and ***www.example.com*** was resolved to the attacker's IP ***10.0.2.8***

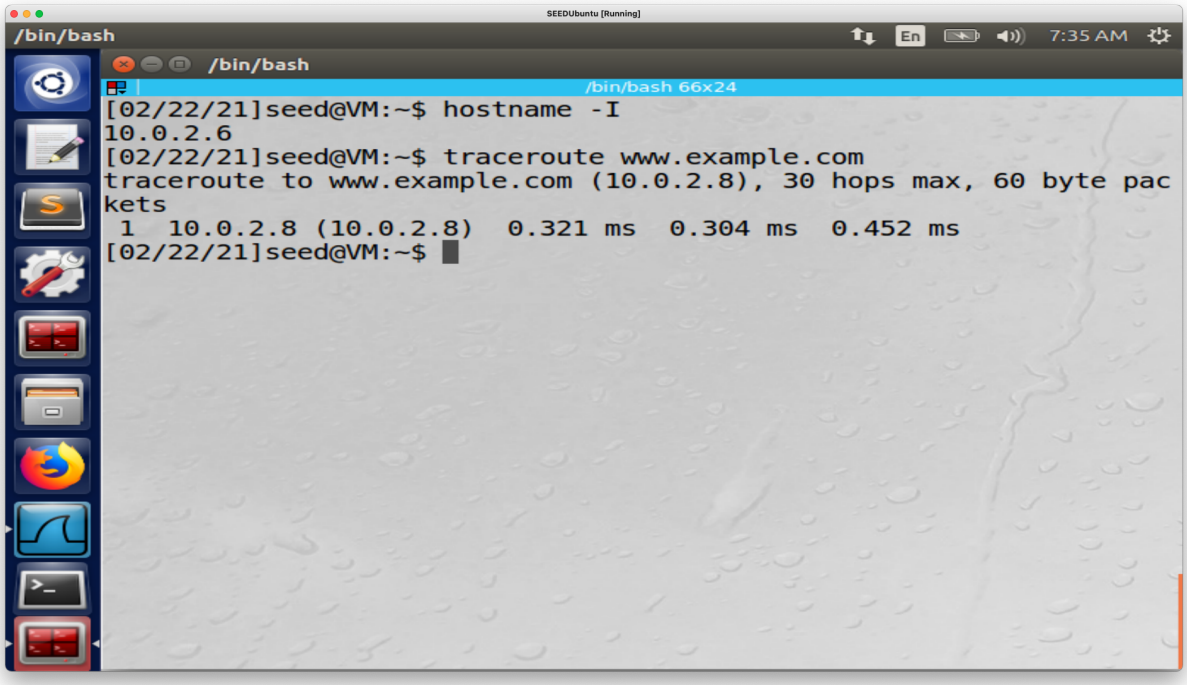


The above is a wireshark capture of ***dig www.example.com*** on the User VM (***10.0.2.6***). A DNS Query first sent to the local DNS Server ***10.0.2.7*** which was forwarded to the attacker due to the DNS Cache being poisoned. From which, the DNS Response is directed first to Apollo before back to 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***](http://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.



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 succesful.