

# Mission 3

## Analysis of Covert C&C Messaging

Jared Grimm and Roger Wirkala  
CS4404 B 2018

**Table of contents**

Introduction	2
Reconnaissance	
Confidentiality	2
Integrity	3
Availability	3
Detection	4
Infrastructure	5
Attack	9
Defense	13
Conclusion	16
Works Cited	17

## Intro

With millions of network connected devices surrounding us in every aspect of our life, from computers, and phones, to smart home IoT devices, these devices act as prime targets for attackers to generate a botnet to carry out malicious attacks, whether harmful to other infrastructure or for personal gain. Botnets are a collection of devices with malicious code installed on them by attackers without the true owners knowledge. The malicious files on bots are susceptible to Advanced Persistent Threats (APT), such that an attacker is able to communicate and issue Command and Control (C&C) messages to them over covert communication channels. In this mission, we analyze the various goals, exploits and countermeasures for C&C messages to propagate from an attacker to previously infected hosts, specifically focusing on the possibility of utilizing the TTL field of DNS Resource Record as such a covert channel.

## Recon

### Confidentiality

The goal of confidentiality is the most crucial goal to an APT. Confidentiality for an APT would be defined as ensuring that their C&C messages are able to propagate to other infected hosts without detection of the specific commands. C&C commands need to appear as normal network interactions, as well as remain uncorrelated with the attackers identity. If this confidentiality is broken, analysts would be able to see the malicious commands being issued to the hosts, and possibly even trace it back to the APT as the source. This would first result in them being able to cut off communications to the distributed bots, rendering the APT defeated as it can no longer issue new commands. Next, if the command messages can not only be read, but also their source can be identified, then it will allow for analysts to trace the traffic back to them, leading to legal action against the attacker. In order to insure covert C&C transmission and avoid traceback, most attackers exploit unused or nonessential bits in existing protocol headers. In particular, manipulating the four byte Time To Live (TTL) of a DNS response, allows for characters to be encoded through ASCII and multiplied such that they fit an expected normal time range. Other countermeasures for avoiding traceback to the attacker include insuring the IP or other identifiers about the attacker are never transmitted with the message to the botnet. Instead, the bots pull their information from a server that has been taken over by the attacker. The traffic of the attacker updating the server with commands will not be tied to the bots pulling these commands (through DNS queries in our implementation) from the compromised server.

### Integrity

Integrity is a secondary goal of an Advanced Persistent Threats C&C messages. It entails the guarantee that the messages an infected host receives are only from the authorized attacker, as well as the command has not been altered by other attackers or analysts in transit. If this goal is

violated it adds uncertainty to the stability of the attackers bot net. If other attackers are able to manipulate the commands of an authenticated APT, they could send a harmful and obvious attack on a target, trying to purposely draw attention to the botnet, which would still be tied to the original attacker. If an unauthorized attacker is able to issue their own C&C commands to the network, it could be negligible as long as the invading attacker does not draw attention and expose the bot net, or block the original attacker from being able to send their C&C messages. As a countermeasure to broken integrity and unauthorized use of C&C channels to communicate with the infected bots, the bots can be put to sleep, and only wake upon receiving known a known command from the developer of the bot. This causes any other C&C messages received over the covert channel to be ignored by the sleeping bot, yet as soon as the attacker sends the wake code, the bot is ready to execute incoming commands, and can go to sleep after receiving a command.

### **Availability**

As confidentiality is the main priority of the bot net, it is preferred that all of the bots are communicated to surreptitiously over having all of them respond quickly to messages. We only need a reasonable percentage to respond to carry out an effective attack. While timing for some attacks such as DDOS, is necessary that all machines are actively carrying out the instructions in unison, others may have no time limit for execution. Thus even if traffic or execution is delayed by hours to days, once the command is carried out, it can still be a successful attack. The availability of 100% bots responding quickly to an attackers C&C command is not a mandatory goal of Advanced Persistent Threats. If availability was compromised no messages would be able to reach the infected hosts. If no new commands can be issued, the hosts would either loop their current attack, or simply lay idle forever, not able to transform or permeate further from the machine that can not be contacted. In this scenario the attacker would ultimately be defeated at that host, and need to infect new hosts if they wish to further carry out their attack. The main counter measure for maintaining availability is to have a larger bot net. The more infected hosts, the more hosts you can have carry out your attack. Another counter measure is simply to ensure confidentiality is not broken. Using high bandwidth connections insures that payloads can be transferred quickly to their destination, which may have the tiniest effect of avoiding detection, by spending shorter time on the wire. Further, avoiding single points of failure in the distributed system is key to insuring that your bots will be resilient from detection and disabling. By using several C&C servers, and propagating commands from various hosts, it insures that if one server is taken down, or one host identified, that commands will not be completely blocked from being able to reach the network. If confidentiality is violated and your communications are discovered, traffic will certainly be cut off, thus losing availability.

### **Detection**

When trying to defend against a botnet, detection is often a crucial first step. There are several symptoms to watch for if one is concerned about being a part of a botnet. All bots require

some form of command and control to be able to receive and send messages. Examining network traffic could reveal important clues. Activity such as repetitive IRC traffic over a select range of ports could suggest a bot communicating over the network. These ports can be commonly used ports such as 80 and 25. This would provide cover traffic for the command and control mechanism. Further examination of the packets could yield spoofed emails or tcp traffic with very obscure options. These are two examples of covert botnet command and control.

In our case, to detect our botnet the DNS traffic of the machine would have to be analyzed. DNS traffic almost always passes through firewalls, where mail or tcp traffic is subject to firewall inspection and could be dropped. However, patterns in DNS behavior could raise suspicion. Constantly querying a DNS server when the mapping is already cached could prompt further inspection. If these suspicious queries aligned with other system activity such as restarts or performance loss it is likely that the machine has been compromised and is part of an active botnet.

Researchers and cyber security professionals have an every growing arsenal of tools to identify and examine botnets. One popular method of botnet research utilizes honeypots. A honeypot in this instance is a machine with known vulnerabilities that has hidden tracking software or backdoors installed in it. Once the machine is exploited and made part of a botnet, researchers can gain access to examine how the bot operates and attempt to determine its command and control infrastructure.

For those not wishing to ever be part of a botnet, there are intrusion detection systems. These systems operate in real time and attempt to analyze and classify all network traffic without dropping any packets. They can be trained with a wide range of techniques. The most recent and ambitious of these systems are being built on complex machine learning models. There are essentially two different categories of activity classification: anomaly based and signature based. Anomaly based detection leverages the known behavior of benign programs, or network traffic. It understands what traffic and activity is normal of properly functioning, benevolent programs. If a program does not exhibit this behavior it is flagged by the IDS. Also if traffic of unusual flow rates, protocols, or IPs is seen entering the network it gets flagged. Signature based detection uses a manually maintained list of malware characteristics, and is commonly employed by anti-virus companies. This, however, is a retroactive method and only detects known malware. It cannot defend against a new, never before discovered malware. IDS systems also struggle with high volumes of traffic, and are susceptible to normalizer attacks on start up, when it has not yet classified acceptable traffic.

Ultimately having a decentralized and unassociated system is key to the survival of a botnet, and protection of the attacker from being traced back to. Decentralized in the fact that there are multiple servers or access points for bots to receive their commands from, such that if one location is discovered and cutoff, the bots will still be able to receive messages from alternate sources. Next, being unassociated in such a way that information about the attacker such as IP, is never involved in the propagation of attacks.

# Infrastructure

Our botnet command and control implementation is based off of DNS. We found that DNS is a great distributed management system that scales nicely and is resilient given the scenario that some servers may periodically or permanently go offline. The three DNS botnet servers we created are using BIND9.

## 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.bombast.com
```

```
$ sudo vi /etc/bind/zones/db.bombast.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. (
; Serial
                        4      ; Refresh
                        604800 ; Retry
                        86400  ; Expire
                        2419200 ; Negative Cache TTL
                        604800 )

; 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

```

Figure 1: Creating DNS Forward Zones

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. (
; Serial
                        3      ; Refresh
                        604800 ; Retry
                        86400  ; Expire
                        2419200 ; Negative Cache TTL
                        604800 )

; 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

```

Figure 2: Creating Reverse Zones

Checked the BIND configure syntax

```

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

```

We then restarted BIND

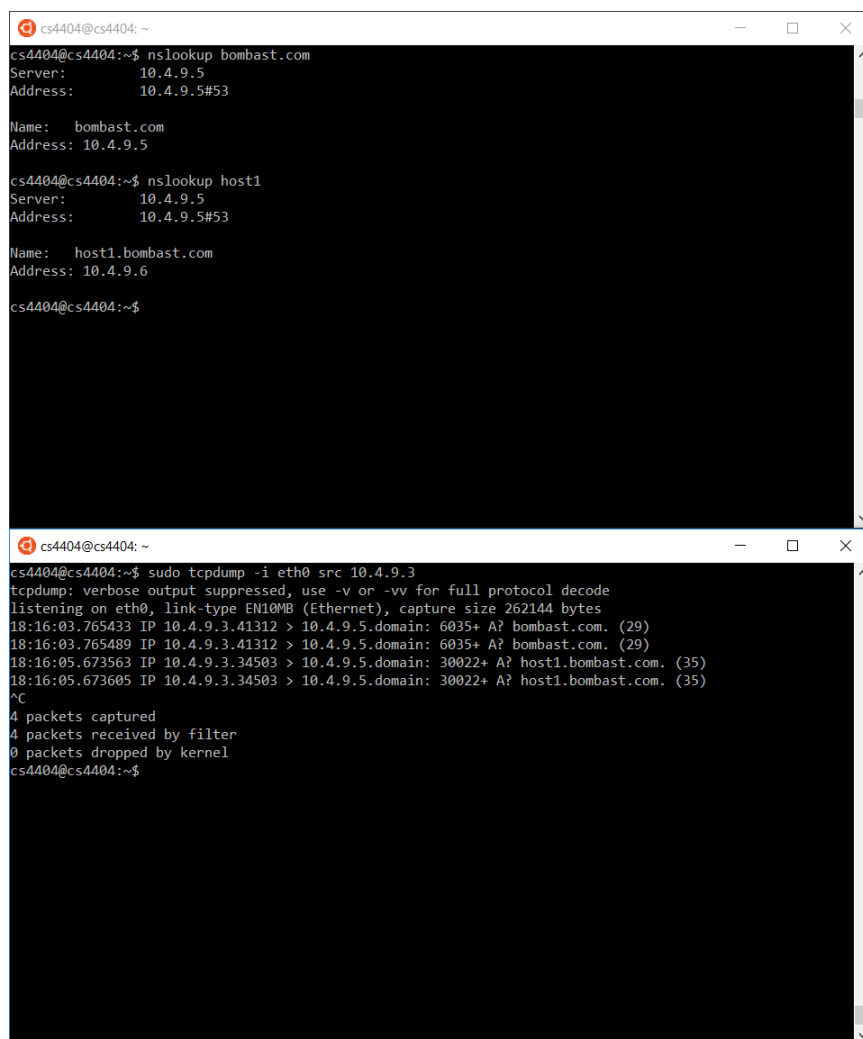
```

$ 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 3: DNS Demo

## Router Setup

A router was setup to imitate a central server is an AS or ISP. Essentially all traffic passes through this router and can be examined and modified.

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:
From 10.4.9.2: icmp_seq=1 Redirect Host(New nexthop: 10.4.9.5)
64 bytes from 10.4.9.5: icmp_seq=1 ttl=63 time=0.586 ms
From 10.4.9.2: icmp_seq=2 Redirect Host(New nexthop: 10.4.9.5)
64 bytes from 10.4.9.5: icmp_seq=2 ttl=63 time=0.873 ms
From 10.4.9.2: icmp_seq=3 Redirect Host(New nexthop: 10.4.9.5)
64 bytes from 10.4.9.5: icmp_seq=3 ttl=63 time=0.874 ms
^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=0.988 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:~$

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
18:13:20.708184 IP 10.4.9.3 > 10.4.9.5: ICMP echo request, id 1293, seq 1, length 64
18:13:20.708272 IP 10.4.9.3 > 10.4.9.5: ICMP echo request, id 1293, seq 1, length 64
18:13:21.709713 IP 10.4.9.3 > 10.4.9.5: ICMP echo request, id 1293, seq 2, length 64
18:13:21.709813 IP 10.4.9.3 > 10.4.9.5: ICMP echo request, id 1293, seq 2, length 64
18:13:22.711536 IP 10.4.9.3 > 10.4.9.5: ICMP echo request, id 1293, seq 3, length 64
18:13:22.711616 IP 10.4.9.3 > 10.4.9.5: ICMP echo request, id 1293, seq 3, length 64
^C
6 packets captured
6 packets received by filter
0 packets dropped by kernel
cs4404@cs4404:~$ sudo tcpdump -i eth0 src 10.4.9.5
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
18:13:34.047978 IP 10.4.9.5 > 10.4.9.3: ICMP echo request, id 1609, seq 1, length 64
18:13:34.048011 IP 10.4.9.5 > 10.4.9.3: ICMP echo request, id 1609, seq 1, length 64
18:13:35.049509 IP 10.4.9.5 > 10.4.9.3: ICMP echo request, id 1609, seq 2, length 64
18:13:35.049553 IP 10.4.9.5 > 10.4.9.3: ICMP echo request, id 1609, seq 2, length 64
18:13:36.050815 IP 10.4.9.5 > 10.4.9.3: ICMP echo request, id 1609, seq 3, length 64
18:13:36.050852 IP 10.4.9.5 > 10.4.9.3: ICMP echo request, id 1609, seq 3, length 64
18:13:36.103461 IP 10.4.9.5.44830 > 8.8.8.8.domain: 16473+ A? ntp.ubuntu.com. (32)
18:13:36.103640 IP 10.4.9.5.42795 > 8.8.8.8.domain: 6782+ AAAA? ntp.ubuntu.com. (32)
18:13:36.103646 IP 10.4.9.5.41104 > 8.8.8.8.domain: 58246+ NS? . (17)
^C
9 packets captured
9 packets received by filter
0 packets dropped by kernel
cs4404@cs4404:~$
```

Figure 4: Router Demo

## Attack

The attack we implemented leverages the commonality and permeability of DNS traffic by encoding ASCII characters in the TTL fields of DNS answers. The TTL field is the perfect covert channel to carry out botnet command and control. DNS traffic is ever present and extremely common activity in any network. The sheer amount of everyday DNS traffic provides

the perfect cover traffic to mask any suspicious queries/requests. Furthermore, DNS traffic is one of the few kinds of network activity that nearly always passes through firewalls without question. DNS TTLs are also not uniform and vary widely. Altering only the DNS TTL will go undetected since the rest of the DNS packet will be standard, accepted traffic, and there is no expected value for the TTL, nor will it get altered across hops to routers. Large amounts of DNS traffic were analyzed to study patterns in DNS TTL values. The average TTL was found to be 98742 seconds which is equivalent to 27.4 hours, and a standard deviation of 81766, which is 22.7 hours. This defined the range to be 4.7 to 50 hours that most traffic will fall into. Our encoding scheme has a one standard deviation range of 13.04-17.02 hours, so even with no expected value, our methods come close to normal average values for TTLs.

Our implementation of the command and control for our five machine botnet consisted of three command and control servers running the BIND9 DNS resolver. These servers were innocent.com (10.4.9.1), notsus.com (10.4.9.3), and regularserver.com (10.4.9.4). The VM at 10.4.9.2 acted as our ISP's server which all traffic was routed through. The remaining VM's were infected bots, but they did not have any command and control capabilities.

We could not create an infection mechanism so some liberal assumptions were made about infected bots. We assumed that a compromised machine that was designated as a command and control bot would have BIND9 installed and records for a fake domain inserted into the resolver. We also assumed the bot master would have remote access with root privileges to at least one command and control bot to begin sending messages. The infected bots, both regular and command and control, would have a list or database of command and control bots to listen to for messages.

Infected hosts often have a pre-generated list of possible commands, waiting for triggers to cause their execution, as well as the ability to allow the bot to include new commands. As a way to implement both of these possibilities, and communicate covertly to our infected hosts, we decided to exploit DNS packets, specifically the Time-To-Live field. Our communication scheme draws from the paper "A Covert Channel in TTL Field of DNS Packets".

The command and control is handled by the python script "cc.py". This program operates with three command line arguments. The first being the file to edit, which is the DNS record for the fake domain being queried by the bots. The second and third are two characters that will be encoded into the TTL field of the DNS record. The DNS TTL field is four bytes long, so we could encode four bytes, meaning four characters, directly into the TTL, however the four characters would cause a TTL over 27 weeks which would be out of a normal range, and suspicious to an IDS. Alternatively, we use just two bytes to encode the ASCII value of characters into, alongside a simple formula to normalize those two values to a TTL within the range of expected normalcy. The ASCII values are obtained by using ord() shown below.

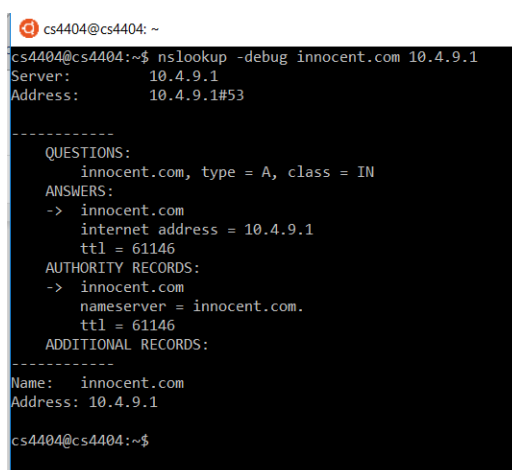
```
num1 = ord(char1)
num2 = ord(char2)
```

The numerical representations of these characters are then obfuscated, as well as the TTL normalized, through the equation below. In this case, 256 and the multiplier of 2, are essentially

shared keys which the infected hosts software will know as well in order to recover the original characters.  $TTL = ((num1 * 256) + num2) * 2$

For our implementation we have a predefined set of commands the bots can carry out. They are “fb” which executes a fork bomb “:(){ :|: & };:”, “rf” which removes all files on the machine, and “ds” which carries out a DDoS attack. These commands, however, will fall on deaf ears unless the bot is awake. This mechanism was included in hopes to prevent replay attacks. If somebody were to be watching the network traffic and discover the TTL related with a DDoS attack they wouldn’t be able to carry out that attack without first waking up the command and control bot. To wake a bot the “wm” command needs to be sent. A bot is put to sleep by issuing the “sl” command.

The getTTL.py program constantly queries the command and control bots and looks for altered TTL fields. This program is constantly calling nslookup with debug options on the host and server that are given as command line arguments.



```
cs4404@cs4404: ~
cs4404@cs4404:~$ nslookup -debug innocent.com 10.4.9.1
Server:      10.4.9.1
Address:     10.4.9.1#53

-----
QUESTIONS:
  innocent.com, type = A, class = IN
ANSWERS:
-> innocent.com
  internet address = 10.4.9.1
  ttl = 61146
AUTHORITY RECORDS:
-> innocent.com
  nameserver = innocent.com.
  ttl = 61146
ADDITIONAL RECORDS:
-----
Name:   innocent.com
Address: 10.4.9.1

cs4404@cs4404:~$
```

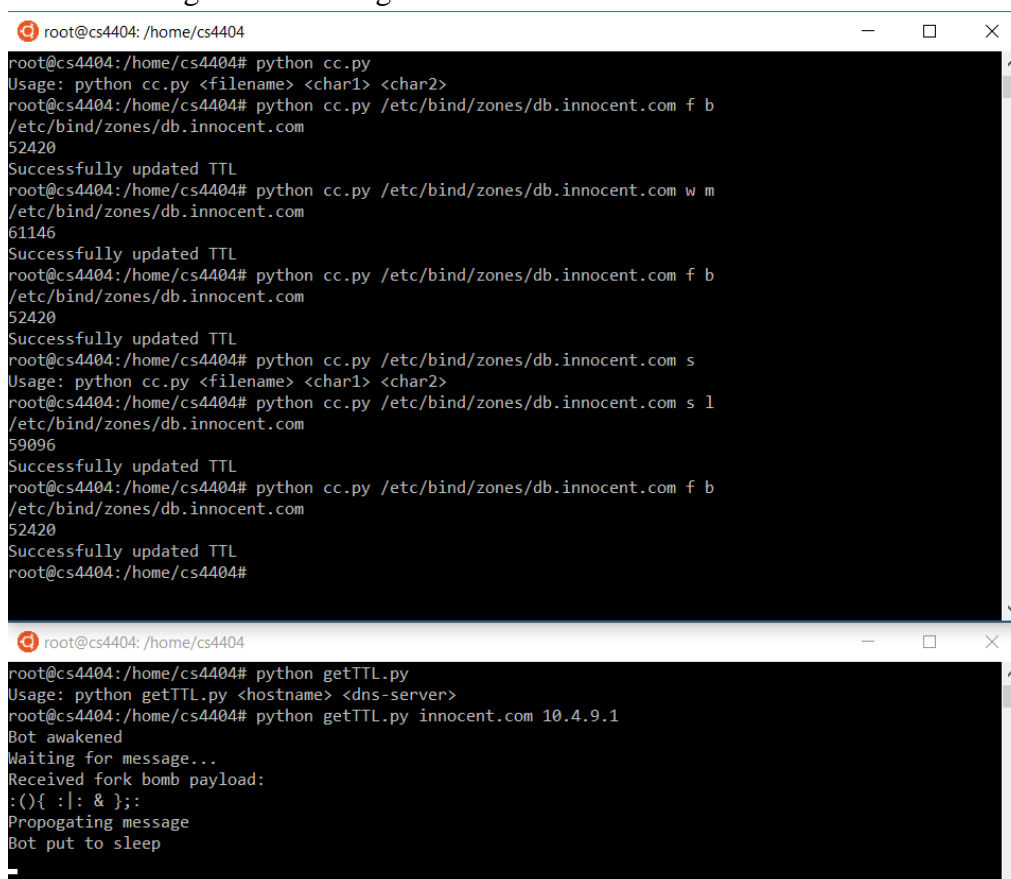
Figure 5: nslookup with debug to view TTL

The program parses the resulting information to find the number that follows “ttl”. It then decodes that TTL fields to decipher messages, shown below. Nothing else about the DNS packet is altered. For the sake of our demo the getTTL program is polling the command and control bots every second. This would be very obvious to detect on a network. To make this communication more covert, it could poll the bots at random times over every few hours or days depending on the desired level of stealth.

```
half = encoded / 2
for count1 in range(32, 128):
    value = count1 * 256
    for count2 in range(32, 128):
        if (half - value == count2):
```

```
byte1 = chr(count1)
byte2 = chr(count2)
```

Once the characters have been decoded, they are matched to a lookup table of commands. The infected hosts carries out the commands with the ability to further propagate the commands in some instances. In our testing environment, executing the commands was simply printing to command line, however our commands for fork bomb and rm could easily be actually ran on the command line causing serious damage.



```
root@cs4404: /home/cs4404
root@cs4404:/home/cs4404# python cc.py
Usage: python cc.py <filename> <char1> <char2>
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.innocent.com f b
/etc/bind/zones/db.innocent.com
52420
Successfully updated TTL
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.innocent.com w m
/etc/bind/zones/db.innocent.com
61146
Successfully updated TTL
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.innocent.com f b
/etc/bind/zones/db.innocent.com
52420
Successfully updated TTL
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.innocent.com s
Usage: python cc.py <filename> <char1> <char2>
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.innocent.com s l
/etc/bind/zones/db.innocent.com
59096
Successfully updated TTL
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.innocent.com f b
/etc/bind/zones/db.innocent.com
52420
Successfully updated TTL
root@cs4404:/home/cs4404#

root@cs4404: /home/cs4404
root@cs4404:/home/cs4404# python getTTL.py
Usage: python getTTL.py <hostname> <dns-server>
root@cs4404:/home/cs4404# python getTTL.py innocent.com 10.4.9.1
Bot awakened
Waiting for message...
Received fork bomb payload:
:(){ :|: & };;
Propagating message
Bot put to sleep
```

Figure 6: Bot receives, decodes and matches the characters 'fb', causing a fork bomb

One of the main advantages of using DNS as a command and control mechanism is the scalability of controlling and distributed system it gives the user. A message only needs to be sent to one command and control bot to propagate through the entire botnet and remain undetected. All bots listening to the command and control domain will wake and receive the new message. If the bot is designated as a command and control bot itself, it will also propagate that message to all bots listening to it. This is beneficial for two main reasons: covertness and resiliency. Command and control messages do not all originate from the same host, making it harder to track down. This also means that if one command and control bots is taken down commands can still be issued and disseminated through any of the other command and control

bots. Communication could be made even more secret by adding a random time delay for the propagation of messages, allowing for a very slow, stealthy command and control scheme.

Future work for our attack implementation would include building a random propagation speed tool for stealthy messaging. Such a system could utilize clock triggers for the bots to wake and pull commands, rather than constant polling of the DNS server for changes. We would also have liked to build some kind of interface or series of commands to allow our bots to learn new commands and accept new payloads. We also limited each query to contain only one answer, to protect the channels secrecy, in case of an IDS detecting a large increase of answers to DNS queries. Leveraging more DNS Answers per query would allow us to send more characters at once, enabling robust new commands to be executed with one sent C&C message.

A demo video of our botnet receiving and propagating commands can be viewed [here](#).

## Defense

In order to prevent such a covert and unexpected attack, on a field which has no expected or verifiable payload, our method of defense will be to regulate and round down the TTLs of all network DNS answers to the nearest minute. We assume that a company/companies can implement this defense on their internal routers, therefore inspecting all network traffic, and enabling preventative measures like regulating TTLs. Since TTL is an arbitrary value that has no impact on the DNS answer details, rather just how long the record is cached by an authoritative server. By rounding the TTLs to the nearest minute or even hour, we break any encoding scheme, thwarting covert C&C channels.

## Configuration

In our infrastructure, 10.4.9.2 was the router selected to be the network regulator. We setup NetfilterQueue, a python package that allows to access to filtered network packets, which can 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 regulator 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 traffic, we needed a way to manipulate the DNS answer packets such that no TTL is has a specified time down to the seconds, enabling character encoding. To achieve this we utilized scapy, a python based packet manipulation library, 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` [5]. With our necessary tools in place, we created the main defense file named `regulator.py`, hosted on 10.4.9.2. The defense can be activated by simply executing `python regulator.py`. 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 --sport 53 -j NFQUEUE --queue-num 2')`

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(2, 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)
```

Using Scapy to sniff the packet layers, we allow any packets that are not DNS Resource Records to exit our filter and allow them to continue through to their destination. All remaining Resource Records in our queue will be processed with scapy and regulated.

```
if not pkt.haslayer(DNSRR):
    packet.accept()
```

Pull the Time To Live of the filtered packet, from the DNS Answer's TTL field, to be rounded.

```
oldTTL = pkt[DNS].an.ttl
```

Since the TTL is stored in the resource record in seconds, it must first be divided by 60 to be converted into minutes, and then rounded to the nearest minute. The rounded minutes is then converted back into seconds and stored as a newTTL to be inserted into the regulated DNS response packet.

```
mins = oldTTL / 60
rounded = round(mins, 0)
newTTL = rounded * 60
```

We then use scapy to construct a new packet containing all the same attributes as the originally filtered packet, except we replace the TTL with our newly calculated rounded TTL.

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

Once fully constructed, route the regulated DNS Response packet to its intended destination, and wait for the next packet to enter the queue.

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

The packet now continues to the infected bot, however once its TTL is attempted to be decoded, no intelligent or readable character set can be generated, thus breaking the communication channel, leaving the infected bots isolated and the attacker disarmed.

```

root@cs4404:/home/cs4404# python getTTL.py notsus.com 10.4.9.3
^CTraceback (most recent call last):
  File "getTTL.py", line 99, in <module>
    newTTL = getTTL(sys.argv[1], sys.argv[2])
  File "getTTL.py", line 9, in getTTL
    proc stdout = process.communicate()[0].strip()
  File "/usr/lib/python2.7/subprocess.py", line 792, in communicate
    stdout = _eintr_retry_call(self.stdout.read)
  File "/usr/lib/python2.7/subprocess.py", line 476, in _eintr_retry_call
    return func(*args)
KeyboardInterrupt
root@cs4404:/home/cs4404#

root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.notsus.com w m
/etc/bind/zones/db.notsus.com
61146
Successfully updated TTL
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.notsus.com s l
/etc/bind/zones/db.notsus.com
59096
Successfully updated TTL
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.notsus.com w m
/etc/bind/zones/db.notsus.com
61146
Successfully updated TTL
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.notsus.com w m
/etc/bind/zones/db.notsus.com
61146
Successfully updated TTL
root@cs4404:/home/cs4404# python cc.py /etc/bind/zones/db.notsus.com f b
/etc/bind/zones/db.notsus.com
52420
Successfully updated TTL
root@cs4404:/home/cs4404#

Regulated Info:
SRC IP = 10.4.9.1
DST IP = 10.4.9.3
NEW TTL = 61140.0
^Cexiting....
root@cs4404:/home/cs4404# vim regulator.py
root@cs4404:/home/cs4404# python regulator.py
^Cexiting....
root@cs4404:/home/cs4404# python regulator.py

Original Packet Info:
SRC IP = 10.4.9.3
DST IP = 10.4.9.4
oldTTL = 61146

Regulated Info:
SRC IP = 10.4.9.3
DST IP = 10.4.9.4
NEW TTL = 61140.0

Original Packet Info:
SRC IP = 10.4.9.3
DST IP = 10.4.9.4
oldTTL = 61140

Regulated Info:
SRC IP = 10.4.9.3
DST IP = 10.4.9.4
NEW TTL = 61140.0

Original Packet Info:
SRC IP = 10.4.9.3
DST IP = 10.4.9.4
oldTTL = 52420

Regulated Info:
SRC IP = 10.4.9.3
DST IP = 10.4.9.4
NEW TTL = 52380.0

```

*Figure 7: With the defense running, the TTL to wake and execute commands is Rounded down, causing unintelligible commands, leaving the bot unresponsive*

As shown, the bot is not even able to be woken up as the command for w m expects a TTL of 61146, but is regulated to 61140, which is an unintelligible time to be decoded by the bot. The attacker sees that they successfully update the TTL on the DNS server, and would not expect that their TTL would be manipulated before reaching their bot. Further, the sleeping bot receives a command with TTL 52380, instead of 52420, which again would be unintelligible.

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 regulated such that no malicious TTL C&C messages can get through. On key intercepts, the iptable rule will be removed, the NetfilterQueue will be unbinded from the network, and any queries in the queue at time of unbinding will be dropped, such that no DNS resource records on the network during the time the defense is running can possibly be malicious and get tho the bots.

Future work for a robust defense scheme that includes automated detection could be achieved with machine learning. By using a tool such as TensorFlow for python and analyzing large network captures of DNS traffic, one might be able to train a model to identify suspicious changes in TTL values or hosts that change TTL values often enough to raise suspicion. This approach would rely heavily on a strong dataset to be able to appropriately train a model to classify such trends with accuracy,

## Conclusion



Ultimately, we exploited a flexible, yet extremely secret channel to transmit command and control messages from a bot master to a network of bots. This covert channel was created through the Time-To-Live field of DNS Resource Records. The unspecificity and resilience of the DNS field made it an ideal exploit for communications. Our attack is based on command and control bots running the BIND9 DNS resolver. These bots can be directly accessed by the botmaster who sends commands using the cc.py program. All other bots in the botnet are listening to these malicious hosts by running the getTTL.py program in the background. On an update of the TTL field they decipher the message and act upon it. This attack has the scalability and flexibility to be extremely robust by leveraging several attributes of DNS traffic including resilience and commonality. We were then able to implement a reasonable defense, to block attempts at covert communication, by regulating all DNS Resource record TTLs to the nearest minute. This will have no impact on performance or DNS answers, just the time they are cached by the authoritative server by seconds. However it is strong enough to insure that the DNS TTL cannot be used to transmit messages. This lightweight defense is highly practicable and easily applied to company networks at gateways into their network. Finally we stated future work that could be done to both extend the reach and portability of our attack, as well as more dynamic defenses utilizing AI to learn to better detect intrusion of DNS encoded messages, or block against attacks done by others in our class.

#### Works Cited

- [1]<http://ipset.netfilter.org/iptables.man.htm>
- [2][https://drive.google.com/file/d/1Tu7KceAo1W5wlR48QBcgYEvD63X\\_yRVe/view](https://drive.google.com/file/d/1Tu7KceAo1W5wlR48QBcgYEvD63X_yRVe/view)
- [3][https://www.researchgate.net/publication/283257776\\_A\\_Review\\_Paper\\_on\\_Botnet\\_and\\_Botnet\\_Detection\\_Techniques\\_in\\_Cloud\\_Computing](https://www.researchgate.net/publication/283257776_A_Review_Paper_on_Botnet_and_Botnet_Detection_Techniques_in_Cloud_Computing)
- [4][http://faculty.cs.tamu.edu/guofei/paper/Gu\\_ACSAC09\\_BotProbe.pdf](http://faculty.cs.tamu.edu/guofei/paper/Gu_ACSAC09_BotProbe.pdf)
- [5]<https://scapy.readthedocs.io/en/latest/introduction.html>

[6]<http://www.zytrax.com/books/dns/ch15/>