CS 4404: Tools and Techniques in Network Security

Team: John Faria, Maryia Hancharova, Shuxing Li

# Mission 3: Novel IDS IDeaS

DNeScalade and IDS Defense

## 1. Introduction

In this report we will explore security goals associated with the communication on the web and ways those goals can be compromised. We also explored different ways that attackers can use to create covert channels for communication with their command and control infrastructure of bots, as well as evaluated and proposed different IDS techniques that can aid defenders in detecting malicious traffic on their network. More specifically, we explored packet inspection tools, flow monitoring tools and SDN based monitoring and factors that affect performance and scalability of such tools.

In this report we will demonstrate a set up of stealthy bot that communicates with it infrastructure by encoding the botnet command and control communications into the questions and answers of DNS queries which are bounced from the local network’s default DNS server to an attacker-controlled authoritative DNS server. With the botnet’s use of DNS tunneling, the command and control communication channel is difficult to detect because the defenders and their network tools do not clearly see any remote-access protocols nor are they able to easily detect any traffic going to or from the attacker’s IP address. We have named our command and control system DNeScalade, referencing both DNS and the castle siege term “escalade”. Then we proposed a defense that can detect such communicating by programming a basic SIEM (Security Incident Event Manager) for analyzing traffic flow on the network and provided suggestions that defenders can use to monitor and protect the network.

## 2. Reconnaissance Phase

For the reconnaissance phase of this mission we explored three types of tools, their strengths and weaknesses, how to deploy them on a network, and how the specifics such as performance and scalability can affect security goals. We also explore the goals that attackers can have when they are communicating with compromised hosts and how these tools affect those goals.

### 2.1 Security Goals

Before we explore tools available for the defenders to analyze network traffic, we first need to establish the security goals that the tools we will present later in the paper aim to aid.

The first goal of any individual user or the defender trying to protect a network serving a large population is the availability of the system. IDS tools in general and specifically flow monitoring tools aim to detect malicious traffic that is overloading the network and valid communication and give the defender an option to remove a machine that is generating malicious, interfering traffic from the network. Attackers can use their command and control infrastructure to launch DDoS attacks and compromise the goal of availability. If the goal of availability is not met, the network can be overloaded with malicious traffic, and users will not be able to access the information they are attempting to retrieve which will at best cause frustration and at worst might result in user being compromised. However, as we will demonstrate later in the report, defenders can use a variety of countermeasures to ensure availability.

In addition, integrity and authenticity of information and data flow should also be considered as goals of the IDS systems. While packet inspection and flow monitoring tools don’t primarily check for integrity and authenticity of information, SDN based monitoring depending on how it is configured can ensure the integrity and authenticity of information. We will discuss in section 3.4 further how SDN based monitoring can use a tool such as TLSDeputy to examine and verify lists of certificates and confirm that the data truly originates from a source it claims to and the data has not been altered. Attackers can often spoof data, falsify certificates or rely on browsers and devices that don’t check for certificate revocation (which is a big issue and a factor to consider) to compromise integrity and authenticity of information. If either one of those factors is compromised, the client attempting to receive the data can be fooled into thinking that the data originated from a trusted source or the corrupted data that was sent to them is actually true and suffer financial and emotional harm as a result. We will discuss further in section 3.4 how tools such TLSDeputy propose to act as a countermeasure and check for certificate revocation ensuring adherence to these security goals.

### 2.2 Packet Inspection Tools

The first IDS method we research were packet inspection tools similar to Bro that we read about in the class. We found that Ubuntu has a useful, basic packet inspection tool in the form of “tcpdump” which can be used to monitor all the packets going through a machine. It is also possible to filter out some of the packets by using the pipeline and the grep command *“tcpdump | grep ''*.

Other tools such as Bro and Snort, unlike tcpdump, are able to perform much more complex inspections, but can come with a challenge of setting them up.Bro uses its bro policy language, which is a programming policy language, while Snort uses a rule system to alert packets of interest [9]. These rules or policies can be expanded easily, and can have multiple present at the same time, making Bro and Snort more powerful than tcpdump for large and complex packet inspection. Bro has a stronger extensibility than Snort because of the policy language difference, making Bro a more complete tool for packet inspection [9]. However, Bro uses a policy language that requires a basic understanding of programming, while Snort can be easier to get started with.

Deploying these tools is not very difficult given a fairly basic understanding on computer programming and linux operating systems, however, can pose a challenge to the defenders with little technical background. For the purpose of this paper, we assume that the majority of the defenders that want to implement such tools on the network and perform complex packet analysis have a basic understanding of Linux systems and basic programming. Tcpdump, Bro and Snort can all be deployed on a Linux machine and used directly from the command line arguments. It is worth noting that performance wise, based on the research finding [10], Bro outperformed Snort in the time parsing through the tcpdump logs which might be an import factor in consideration of the defender trying to minimize the time spent analyzing data logs.

In our research, it was determined that packet inspection tools would be useful in a SIEM environment for looking at network protocol usage, as well as seeing what ports were being accessed. In a more advanced SIEM, running on a more powerful machine and much more complicated than one that we were able to program in two weeks, other details of packet inspection such as message content and DNS questions would also be useful.

### 2.3 Flow Monitoring Tools

In addition to packet inspection tools, our team also explored flow monitoring tools. One of the tools, is a Netflow protocol that can monitor network traffic. In turn, that traffic can be collected, exported and analyzer by a third party analyzer [8]. One of the software tools we looked into is Ntop which is a software for web based traffic analysis. Ntop supports the Netflow protocol, and can be deployed on all platforms via direct installation. Ntop also supports exporting data via lua scripting, or direct analysis with traffic visualization, allowing user to export and analyze information with a more readable format [7].

Another tool we explored, similar to ntop is FlowScan. FlowScan also has the ability to visualize data, and can be deployed via direct installation as well [6]. Using these flow monitoring tools, a defender has a high chance of figuring out the type of attack occurring on the network given the graphic analysis of the incoming packets. Furthermore, the defender ideally could use a combination of tools, and initially establish whether the attack is occuring using flow monitoring tools and then proceed to packet inspection tools mention in section 2.2 in order to gain a deeper insight into the source and the type of attack and engage in appropriate countermeasures.

It was determined that network flow monitoring tools would be useful for monitoring what machines were communicating with what machines, as well as how much the different machines were communicating on the network.

### 2.4 SDN-based monitoring

One of the methods for monitoring and detecting malicious traffic we explored for this mission was SDN based monitoring which can monitor traffic by using controllers and middleboxes. SDN based monitoring has several advantages and ends up being a great solution for residential networks. This solution can be “administered by end-users without any training in networking or security” [1] and can protect a variety of devices such as televisions, video game systems and home automation devices that often have security vulnerabilities [1]. SDN based monitoring relies on residential routers to address weaknesses in the end devices that themselves might not possess the capabilities to enforce security such as traditional laptops and desktops. Routers perform filtering and validation and block attacks from exploiting devices and “when routers combine software defined networking (SDN) techniques with cloud based virtual machines, these routers can ensure that middleboxes running on cloud VMs can enforce certain security goals” [1]. Researchers also estimated that the cloud hosting for the system would cost roughly $20/per month and those costs included “ two always-on VMs, network traffic transmission, and disk storage, with the majority of the cost associated with the VM uptime” [1], not to mention that given CPU and memory overheads proposed in the paper, residential networks could share these VMs or a “third-party security provider could run cloud-based VMs to provide TLSDeputy services to large numbers of residential users and achieve economies of scale” [1]. There are some concerns about cloud based middleboxes introducing network latency and bandwidth overheads, which become problems with applications such as video games [1].

One of the ways to deploy SDN type of monitoring that is described in the paper by Taylor and Shue [1] uses a combination of OpenFlow and IDSes to achieve detailed insights into communication on the network. The paper proposes a cloud based TSL validator that “monitors TLS handshake process and performs independent verification of TLS certificates and revocation checking using CRL’s” [1] in additional to creating “new communication channel between OpenFlow SDN controlled and cloud based middlebox” [1]. Certificate revocation system is crucial, and making sure that the system checks for up to date certificates greatly increases the security of the system contributing to integrity and authenticity goals we discussed in section 2.1 goals. There has been a lot of research efforts on the subject on certificate validation. Server Based Certificate Validation Protocol (SCVP) is one approach that enables clients “to delegate path construction and certificate validation to another server” [1], however it has some limitations as it requires “client-side support, which may not be feasible in legacy or embedded devices” [1]. TLSDeputy proposed in the same builds upon the goals of SCVP and previous work and experience with Skype and explores “outsourcing security applications to a public cloud” [1].

The system provides an innovative and practical approach “to enforce certificate revocation checks” [1] by using “OpenFlow enabled switches, cloud based controller and middleboxes, and custom Open Flow Agents (OFAs)” [1]. TLSDeputy middlebox runs on a VM cloud that is connected to an OVS instance and checks the “TLS handshake and checks certificates and other important information, such as the Server Name Indicator (SNI) extensions to TLS, to ensure a secure TLS connection” [1]. TLSDeputy contains a feature that checks whether the TLS request is a new one or a renegotiation. In case of a renegotiation, it “notifies the OpenFlow controller via the OFA that the communication can be transmitted directly via the residential router without further TLSDeputy inspection” [1], but if it detects a new request it performs throughout set of verification checks.

Providing efficient CRL enforcement is crucial for integrity and authenticity goals, and recent studies continue to demonstrate that many mobile browsers still don’t perform “a proper revocation checks even after the high-profile Heartbleed attack” [1] which greatly compromises above mentioned security goals. One of the main reasons behind that is performance concerns. However, TLSDeputy “caches CRLs locally rather than obtaining them on demand” minimizing performance concerns.TLSDeputy goes even further to address performance issues, and “anytime TLSDeputy encounters a certificate with a CRL not in the database we add the URI to a list of monitored CRLs and immediately begin retrieving it in the background. However, to avoid performance issues, we do not wait to check the CRL for the chain causing the first retrieval” [1].

The paper also summarizes the results for security evaluation of IoT, mobile devices and desktop browsers. The paper unsurprisingly found that both IoT devices that they tested either had a vulnerability with storing certificates or full on man in the middle attacks [1]. They also found that mobile browsers did not perform revocation checks that they should perform to avoid compromising integrity and authenticity.

## 3. Infrastructure Building Phase

In order to demonstrate our attack and defense, we organized our infrastructure using the VM’s in the following way:

|  |  |  |
| --- | --- | --- |
| IP address of the machine | Name | Description of the function |
| 10.4.20.1 | Host 1 | Router that host 2, host 3, and host 4 will use to communicate with the rest of the network |
| 10.4.20.2 | Host 2 | Bot for our bot command and control infrastructure |
| 10.4.20.3 | Host 3 | Healthy and normally functioning machine on the network |
| 10.4.20.4 | Host 4 | Healthy and normally functioning machine on the network |
| 10.4.20.5 | Host 5 | The default DNS server that resolves DNS queries that the router communicates with |
| 10.4.20.6 | Host 6 | Authoritative dns server that the attacker will use to communicate with their infrastructure |

Our infrastructure creates a local network out of the virtual machines with the IP addresses 10.4.20.2, 10.4.20.3, and 10.4.20.4. The traffic from these local machines is carried through the “router” 10.4.20.1, before the local host machines can connect to each other, or the rest of the “Internet”. The virtual machine 10.4.20.5 acts as both the default DNS server for the local machines, and as a web server that is out on the Internet. There are python scripts on the local machines that will get the web pages hosted on 10.4.20.5. 10.4.20.6 is another server located on the “internet” that is known to be malicious, and its configuration is detailed in the attack section of the report.

### 3.1 Setting Up a Local Network of Machines

We began setting up our infrastructure by setting up 10.4.20.2, 10.4.20.3, and 10.4.20.4 as our local network of machines. We set up host 2 as a malicious bot on the network that the attacker will communicate with in the attack portion, and host 3 and 4 as normal functioning machines.

We began by disabling ICMP routing and redirects by modifying “sysctl.conf” file and creating a script that copies the contents of that file into the default network infrastructure. We uncommented out tow lines to include *“net.ipv4.conf.all.accept\_redicts = 0”* and *“net.ipv6.conf.all.accept\_redirects = 0”* to disable redirection for both ipv4 and ipv6 (see Figure 18 for our sysctl.conf file). Then we created a script “route-thru-1-setup.sh” that contains several commands to set host 2, host 3, and host 4 to ignore any redirects requests because we wanted to make sure out machines communicate with the network properly only through the router which is our host 1. We used “echo 0 >” command to change the value inside all of the “accept\_redirect” files and a command to restart our networking infrastructure (please see Figure 17 for the detailed route-thru-1-setup.sh file).

Then on every machine we added a routing table rule to make sure that host 2, host 3 and host 4 communicate with the outside network through host 1 which serves as our router. We enabled root privileges on all the machines by using the command *“sudo su”* and we added host 1 as the default router to the routing table by using a command *“route add -net 10.0.0.0/10 gw 10.4.20.1”* on all of the above mentioned hosts (see Figure 1 for routing table for host 2, Figure 2 for routing table for host 3, and Figure 3 for routing table for host 4).

Next, we edited network parameters on host 2, host 3 and host 4 to point to the authoritative server. We followed the instructors from “How to implement a DNS server using BIND“ article [4] and navigated to the network directory by using *“cd /etc/network”* command and opening up interfaces file with *“nano interfaces”* command. We edited the file and inserted a line “dns-nameservers 10.4.20.5” into the file into two places (See Figure 4 for interface file for host 2, Figure 5 for host 3 and Figure 6 for host 4 respectively) to establish host 5 as our authoritative DNS server.

### 3.2 Setting Up Host 5 as DNS Server

Next we set up host 5 located on 10.4.20.5 as the default DNS server that can redirect our clients to “[www.guuglu.com](http://www.guuglu.com)”, “[www.rabbit.com](http://www.rabbit.com)”. We set up a DNS server using BIND configuration files. We cleaned and reinstalled bind to make sure we were working with a clean set of settings. Note that if you are working with a clean install of Ubuntu this might not be necessary. The command we used is *“sudo apt-get purge bind9”* to purge and clean all the settings we were not using, and *“sudo apt-get install bind9”* command to install a new bind package. In addition we also installed utilities for bind9 by using *“sudo apt-get install bind9utils”* command.

Next we proceeded to create zones for our domains. We enabled root privileges by using *“sudo su”* command and then navigated to the bind directory by using “*cd /etc/bind”* command and creates a new “zones” directory by using *mkdir zones* command. We navigated inside new zones directory and created zone files for “[www.guulu.com](http://www.guulu.com)” and “[www.rabbit.com](http://www.rabbit.com)”. We created a bind file for “[www.guuglu.com](http://www.guuglu.com)” by using “nano db.guuglu.com” command and followed the instructors from “Linux DNS server BIND configuration” article [2] and modified the appropriate values to our domain names and IP addresses (see Figure 7 for our db.guuglu.com file). We repeated the same steps and created similar file for “[www.rabbit.com](http://www.rabbit.com)”, but modified the domain name to point to the rabbit.com (see Figure 8 for our db.rabbit.com” file).

Next we modified a configuration file to point to our zones files by navigating to bind directory using *“cd /etc/bind”* and opening up named.conf.local file with *“nano named.conf.local”* command. We again followed the instructions from the “Linux DNS server BIND configuration” article [2] and replaced the appropriate fields to match our configuration (see Figure 9 for our named.conf.local file).

### 3.3 Establishing Routing Rules on Host 5

We also modified the routing table on host 5 to make sure that host 5 knows to communicate with all the machines on our network through the router which is simulated on our host 1. We created a “route-thru-1.sh” script (see Figure 27 for the script) that uses “route add -host” command along with the specified IP’s of our networks and the “gw” with the specified IP address of our router (so for example for one of the lines is route add -host 10.4.20.2 gw 10.4.20.1). After the file is saved, it restarts modifying rules allowing us to have persistent routing rules (see Figure 28 for our routing table).

### 3.4 Generating Normal Traffic and Creating a Web Server

In order to simulate more machines on the network than we have access to through the virtual machines provided to us, we set up a python script called “webserver.py” (see figure 24 for webserver.py ) on host 5 that can host multiple web servers on the same machine. The script takes 2 arguments: the html file to host and the port number for accessing the file. To run the script we used the command in the following format: *“sudo python webserver.py [filetohost] [portnumber]”* which opens the http server at the port specified. Then clients can connect to the server using the command in the following format: *“wget [serverip]:[portnumber]”*. The port numbers we used are 56000 and 56001, the order does not matter. The port numbers are hardcoded into the clients’ machine to simulate accessing different addresses. In addition, we also hosted 2 different html files on host 5: index.html (see Figure 26 for index.html file) and dif.html (see Figure 25 for dif.html file). Index.html is the homepage for guuglu.com, a powerful search engine and dif.html is the homepage for rabbit.com, the frontpage of the internet. These two files contain basic html tags and do not contain JavaScript or CSS style sheets in order to simplify the setup process.

Lastly, to create covert, regular traffic from the client, we created a finite state machine trafficfsn.py. The script can be run with the command *“python traffic-generator.py”* (See Figure 19 for traffic-generator.py source code). The script has two states: querying the DNS and sending HTTPS requests. The client always starts by querying the DNS. After the client received the DNS response, it will move on to the HTTPS request state and wait for HTTPS response. Similarly, after receiving the response, the client will once again query the DNS. The requests are made at random intervals between 0 and 10 seconds. We included a list of domains and subdomains in the file, so the client can randomly query different domains and create different combinations of requests in order to simulate regular traffic flow.

Figure 37 in the appendix displays the webservers running, and clients generating web traffic

### 3.5 IP Forwarding on Host 1

We also had to make sure that IP forwarding was enabled on host 1, so all the machines that we set up in section 3.1 could communicate properly. We changed the value inside ip\_forward and used two commands *“echo 1 > /proc/sys/net/ipv4/ip\_forward” and*  *“sudo sysctl -w net.ipv4.ip\_forward = 1”* commands (see Figure 23 representing enabled IP forwarding)

## 4. Attack - DNeScalade Botnet Command and Control

Our attack uses DNS tunneling through the host network’s default DNS server and to the attackers authoritative DNS server. The bot will query the DNS for subdomains that are recorded on the authoritative DNS server. The default DNS server will see the request for the subdomain, and forward the question to the attacker-controlled authoritative DNS server. The authoritative DNS will respond with answers to the question that will instruct the bot to carry out certain actions, such as launching a denial of service attack, or scanning a machine. The bot is able to communicate back to the botnet host and even send data via DNS exfiltration. All the while, neither remote access protocols or the IP address of the attacker are easily detectable by a network defender. We have named this botnet command and control system DNeScalade, referencing both DNS and the castle siege term “escalade”.

### 4.1 Changes to Host 5

In order to have the DNS server on Host 5 recognize Host 6 as having an authoritative DNS server for the domain “\*.sketchydomain.net”, a small change needed to be made to the named.conf.local file on Host 5. The file /etc/bind/named.conf.local was modified to include a slave zone for sketchydomain.net, located on 10.4.20.6. The syntax of the file change can be seen in Figure 38.

### 4.2 Setting Up Host 6 as an Authoritative Server for the Attacker

We also followed a similar set of instructions to section 3.2 and set up our host 6 as a DNS server containing the zone file “db.sketchydomain.net” that the attacker will use for communicating with the bot as part of the command and control infrastructure. As root, we navigated to the cd/bind/ directory and created a zones directory using the command “*mkdir zones*”. We navigated to the zones directory and created db.sketchydomain.net file by using *“nano db.sketchydomain.net”* command. We set the bind file to point to 10.4.20.6 address on host 6 and to contain the domain name associated with sketchydomain.net (see Figure 10 for our bind configuration file). Note that in section 4.3 we explain how the “db.sketchydomain.net” file will be periodically replaced during our attack, so the file is not static, but originally we began by creating the basic file.

We also edited named.conf.local file inside bind directory and added the file path for our newly created zone file (see Figure 11 for named.conf.local). The file is similar the named.conf.local file from section 3.2 with the exception of the added “allow-transfer {10.4.20.5;} line. We added allow transfer line in order to allow Host 6 to send back the file to Host 5 and communicate where sketchydomain.net is located.

We also modified named.conf.options file from the same bind directory with “nano named.conf.options” command and disabled recursion and transfer (see Figure 12 for our named.conf.options file).

Lastly, we also modified interfaces file on host 6 following a procedure similar to the one mentioned in section 3.1. We added host 6 to the interfaces file as one of the dns-nameservers by opening up the file with “nano interfaces” from the “etc/network” directory and added a line “dns-nameservers 10.4.20.6” line in two places in the file (see Figure 13 for our interfaces file)

### 4.3 Command and Control Interface

There is a python script named “*host-console.py*” that is run on the attacker’s authoritative DNS server with “*sudo python3 /home/cs4404/host-console.py*” (see Figure 14 for python code or “host-console.py file in the directory). The script acts as a console, through which the botnet manager can give instructions to, and even get responses from the bots. The five options that the attacker can run from the console are: “help” - which displays all of the console options, “wait” - which instructs the bot to do nothing, “scan <IP address>” - which instructs the bot to scan the given IP address and then waits to parse the bot’s response via DNS exfiltration, and “ddos <IP address>” - which instructs the bot to launch a denial of service attack against the given IP address.

Once the commands were entered into the console, they were inserted into a copy of the DNS db.sketchydomain.net zone file, which then replaced the existing db.sketchydomain.net zone file located in /etc/bind/zones. Then the script would restart bind so the bot would be able to detect the change in command. The new db.sketchydomain.net changes are made by writing lines from the command’s specific sketchydomain.net record, which is stored with the host-console python script in /home/cs4404/. The specific records are “ddos.sketchydomain.net”, “scan.sketchydomain.net”, and “wait.sketchydomain.net” (see Figure 20, Figure 15, and Figure 16 for ddos.sketchydomain.net, scan.sketchydomain.net and wait.sketchydomain.net respectively). These records have TTL values of 1 second, and have IP addresses associated with the “givemeinstructions” subdomain that reflect their specific command (see 4.3). The first 5 lines of the file are written to the new file using “head -n 5 <command>.sketchydomain.net > new.sketchydomain.net”. Then the python script writes the next line of the file, based on the value of the number stored in the file “number” (see Figure 36). The number indicates the serial number for the bind DNS record, and an incrementation of it is necessary for bind on the default DNS server to recognize any changes in the DNS record on the authoritative DNS server. The line is appended to the new.sketchydomain.net record by having the script execute the terminal command “echo \” “ + str(number) + “; Serial\” >> new.sketchydomain.net””. The last 15 lines of the command’s specific DNS record are appended to new.sketchydomain.net using the command “tail -n 15 scan.sketchydomain.net >> new.sketchydomain.net”. If the command is ddos or scan, and a target IP address is required than the script would append the last line by executing the terminal command “echo \”<command> IN A “ + ipAdr + “\” >> new.sketchydomain.net”. Finally new.sketchydomain.net is copied from its location in /home/cs4404/ and to the zone directory at /etc/bind/zones/db.sketchydomain.net (see Figure 10). All of these file changes and commands were automatically carried out by the python script, the botnet master just needs to enter their basic command into the console. The details of how to bot instructions were embedded are explained in section 4.3.

### 4.4 Embedding instructions in the DNS record

The bot queries the authoritative DNS server every two seconds asking for the IP address associated with “givemeinstructions.sketchydomain.net”. When the bot master has the console give instructions to the bot, the IP address associated with the subdomain “givemeinstructions.sketchydomain.net” becomes the given command, written out as the decimal equivalent of their ASCII characters, separated by periods. For example the command “wait” will be encoded into the IP address as “119.97.105.116”. The same is done for the commands “ddos” and “scan”. When the “scan” and “ddos” commands are entered into the console, than those subdomains are also added to the new DNS record for db.sketchydomain.net, and the associated IP address for them becomes the target IP address. For example if the botnet master enters the command “ddos 10.4.20.3” then the IP address associated with the subdomain “givemeinstructions” will become “100.100.111.115”, and the subdomain “ddos” will be added to the record with the associated IP address 10.4.20.3 (the IP address of the target). After the records are updated the console script will automatically restart bind.

### 4.5 Bot Payload

The bot is controlled by the python script “*payload.py*” (see Figure 22 for the python code or supplemental file included in the archive directory) which is run as root on the bot host, 10.4.20.2. A dependency of the script is to have nmap installed on the host machine. Nmap was installed by executing the command “sudo apt-get install nmap”. The script will use the terminal command “dig givemeinstructions.sketchydomain.net +noall +answer | grep A” to get the IP address associated with the subdomain “givemeinstructions” on the sketchydomain.net authoritative DNS server 10.4.20.6, which is controlled by the attacker. Because the default DNS server of the bot’s host machine is 10.4.20.5, the host machine only ever communicates directly with 10.4.20.5, and not the attacker’s server located at 10.4.20.6. The script will parse the returned IP address, converting the address back to its original command word. If the command is “scan” or “ddos”, then the bot will query for the IP address associated with the given command’s subdomain, otherwise the script will wait two seconds before querying the botnet command and control server for further instructions. If the given command was “ddos”, then the script will use the same dig command to get the IP address associated with ddos.sketchydomain.net, this will be the target IP address for the ddos. The bot will launch a denial of service attack against the target using the terminal command “*ping -c 100 -i .1 <target IP address>*”, which will rapidly ping the target IP address 100 times (see Figure 29). If the given command was “scan”, then the script will use the same dig command to get the IP address associated with scan.sketchydomain.net, this will be the target IP address for the scan. The bot will launch the scan using the nmap terminal command “*nmap -sV <target IP address> 2>&1 | tee scan.log*”, which will conduct a scan to determine open ports verbosely and output the results of that scan to the file scan.log (see Figure 30). The script will read the contents of scan.log into one string, then remove all symbols and whitespace from the string (which would otherwise interfere with DNS requests). The resulting string, which contains the results of the nmap scan, will be exfiltrated to the attacker’s command and control server by DNS queries that are forwarded to the authoritative DNS with the subdomains being sixty-character chunks of the string. The chunks are re-assembled into the original string via a listener that runs as part of the python host-console script.

After the bot has launched its attack it will loop-back and continue to request the IP address associated with “givemeinstructions.sketchydomain.net”.

## 5. Defense

Our intrusion detection system (IDS) is a SIEM (Security Incident Event Manager) operated by the python script “IDS.py”, that parses through tshark output. The IDS records and displays important information about the packets captured by tshark. Notifications of concerning network activity, such as large amounts of traffic flow and host machine’s behaving suspiciously. The IDS will also look for certain threats in the network traffic and will display a warning about the incident, pausing all output until the incident has been dealt with or verified to be legitimate activity. Information detailing the parts of the incident are displayed in the warning so that the defender who is managing the IDS knows what logged events and notifications to look for while they are hunting for the potential threat.

### 5.1 Event Logging

The python script IDS.py (see Figure 21 for the part of the python code or Figure 21 supplement file included in the zip folder directory for the entire code) should be run as root on the router, 10.4.20.1. A dependency of the script is to have tshark installed on the host machine. Tshark was installed by executing the command “sudo apt-get install tshark”. The script captures packets traveling across the router using the tshark command “sudo tshark -T fields -e frame.number -e frame.time\_delta -e \_ws.col.Source -e \_ws.col.Destination -e \_ws.col.Protocol -e ip.len -e \_ws.col.Info -e tcp.port”. The python script uses the Popen function from subprocess library to capture the output of the tshark command as a string that the script can use.

The resulting string is long, complex, and not easy for a defender to read through. The timestamp for the packet is recorded and the string is passed to the isolateSDP() function. The isolateSDP() function parses through the string and returns a list containing the source, destination, and protocol of the packet (SDP). The SDP is logged in the last30[] list, which is a list that loops around after 30 elements have been added, and contains the last 30 SDP records. The current SDP record is displayed in columns, alongside the timestamp of that record. The columns allow for the defender who is monitoring the warnings from the IDS to skim the logs, instead of reading through dense tshark output which contains unnecessary information. If the source of the current SDP record is one of the host machines on the isolated network (10.4.20.2, 10.4.20.3, or 10.4.20.4), then it is added to the list frequentTalkers[].

### 5.2 Notifications and Incident Detections

The IDS.py script will navigate through the records and look for suspicious or dangerous activity. Some of the activity is displayed as notifications. Other, more concerning, activity is displayed as a “WARNING” and requests for defender who is monitoring the IDS to deal with the threat, or potentially verify that it is normal network activity.

#### Section 5.2.1 - Detecting Blacklisted IPs

At the top of the IDS.py script there is a section for script configuration options. One of these options is a list containing IP addresses that are to be considered “blacklisted”. These IP addresses are known to belong to attacking machines, and the defender should receive a warning immediately if they are detected on the network. For every new packet, the IDS checks for the most recent record and confirms that neither the source, nor the destination IP address match any of the IP addresses on the blacklist. If the IDS does detect a blacklisted IP address, it will generate a “WARNING” for the defender immediately. The warning message will contain information about whether the blacklisted IP address was the source or destination, and what host machine that IDS detected the blacklisted IP communicating with. The script will pause and wait for the defender to deal with the threat, or verify that it was harmless network activity. For our tests we had the attacker’s command and control server listed as a blacklisted IP address (see Figure 35). Because of this basic blacklisted defense, our attacker needed a way to covertly control their bot on the network without displaying their IP address.

Figure 35 in the appendix deals with defense and detecting a blacklisted IP

#### Section 5.2.2 - Traffic Floods and Detecting DDoS Attacks

One of the configuration options at the top of the IDS.py script is the FloodThreshHold. This variable details the thresh-hold for the ratio of SDPs (source, domain, protocols) in the last 30 SDPs that can look similar before the IDS investigates the traffic flood. The function detectFlood() is used to compare the source IP addresses and their protocols for the last 30 packets. If more packets have had the same source IP address and the same protocol than the thresh-hold, then the IDS recognizes the activity as a traffic flood. If the IDS finds a traffic flood it will display a notification to the defender that there is a traffic flood and begin scanning the traffic to determine if there is a denial of service attack occurring or not.

The IDS looks for ping-based denial of service attacks by calling the function detectPingODeath(). The detectPingODeath() function looks to see if the most frequent protocol that has been detected (the protocol that must have triggered the traffic flood notification) is “ICMP”, the protocol that ping uses. If the amount of ICMP traffic is greater than the flood thresh-hold, then the IDS generates a warning for the defender, detailing the detected source of the denial of service attack, and the host machine that is being attacked. The IDS pauses and waits for the defender to “OK” the incident, indicating that they have either dealt with the source of the malicious traffic or used the log of SDPs and notifications to confirm that the traffic was actually harmless network activity.

Figure 32 in the appendix deals with defense and detecting a DDoS attack

#### Section 5.2.3 - Detecting Port Scanning

The presence of a traffic flood also triggers the IDS to look and see if it has seen any recent port-scanning on the network. Most of the port scanning generated by the nmap scan takes place over the “TCP” protocol. The IDS uses the detectScan() function to look through the last 30 SDPs (source, destination, protocol) and the recent full-tshark output to determine if it has any recent logs that would be indicative of port scanning. The IDS firsts checks to see if the amount of TCP traffic in the last 30 SDPs is greater than the defined flood thresh-hold. If so, then it confirms that the traffic has been between the same two host machine IP addresses on the local network. Finally, the function will look at the current tshark-output and isolate the ports being accessed in the most recent scanning packet. If it has detected port scanning, the IDS will display a warning to the defender, detailing the source of the scan, what computer is being scanned, and what ports are currently being looked at. The IDS pauses and waits for the defender to “OK” the incident, indicating that they have either dealt with the source of the malicious traffic or used the log of SDPs and notifications to confirm that the traffic was actually harmless network activity.

Figure 33 in the appendix deals with defense and port scanning

#### Section 5.2.4 - Noticing Communication Frequency

The IDS also examines the frequency at which host machines communicate on the network. At the top of the IDS.py script there is a configuration setting named “frequentTalkerRatio”. The ratio gives the number of communications in the last 500 packets that a machine is allowed to have before it is considered to be a “frequent talker”. If all of the host machines on the local network are doing similar web browsing activity (requesting web pages at random intervals as detailed in section 3.4), than none of the machines should be talking significantly more than any other machine. Frequent talking could indicate that something is different on the machine that is frequently talking. The IDS has a list of the most recent 500 communicating IP addresses. Whenever a new packet is logged, the IP addresses of the local network machines are added to the list. Once every 100 added IP addresses, the IDS uses the analyzerTalker() function to determine if any one IP address has been talking more than the given thresh-hold. The activity could potentially be a sign of a DDoS attack, or maybe even a command and control communication channel. A machine’s status as a frequent talker is not necessarily an indication of malicious activity and it could easily be caused by many innocent network activities, however it is suspicious and therefore the IDS displays a notification to the defender detailing what IP addresses has been talking most frequently.

#### Section 5.2.5 - Detecting Network Script Communication “Heartbeats”

One of the defense mechanisms involves analyzing the incoming packet frequency to detect repetitive network communications. Generally these communications repeat at a certain frequency, controlled by a script and the resulting network behavior is almost heartbeat-like. The same communications occuring at a specific frequency is an indicator of bot behavior. The script keeps track of a dictionary of the machines in the local network, and the timestamp whenever they sent a packet to other machines. After getting enough packets sent by the machines, the script will then parse through the timestamps and calculate out the time difference between each packet sent. If multiple packets are sent in the same time frame, these packets are considered as only one packet. The script then picks out the range of the time frame, and if the range does not appear to be a distribution of random numbers between 0 and 10, than it is very likely that there is a heartbeat bot behavior, and a warning will be shown on the screen. The number of packets to record is determined by the packets sent per second over the network, so a larger number of traffic per second will have a larger overall capture size. This also effectively reduce the occurence of false positives of reporting bots. The IDS will display a warning to the defender, and will pause until the incident has been dealt with. The details of the warning, notifications, and logs of the network traffic allow for the defender to determine the effectiveness of the defense, and if the “WARNING” may have just been a false positive referring to normal network traffic. Also the redundancy of having different forms of detection ensures that even if the defender misses an important warning about command and control communications, that they will still be alerted when other, different forms of malicious activity are detected.

Figure 34 in the appendix deals with defense and the detection of a network script’s “heartbeat”

## 

## 6. Conclusion

In this report we have thoroughly explored the security goals and vulnerabilities associated with the communication on the web. We have discussed the major security goals that defenders have when maintaining their network systems and the ways that different tools such as packet inspection, flow monitoring and SDN based monitoring can aid the defenders in detecting malicious traffic. We also successfully demonstrated a set up for a stealthy bot that communicates with it’s command and control infrastructure via DNS queries. We picked a more difficult method of communication that uses DNS tunneling and proves to be harder to detect since defenders cannot detect traffic coming from the attacker’s IP address and instead see traffic coming from compromised network of machines. In order to assist defenders in mitigating such attacks, we proposed a defense in the form of a SIEM for analyzing traffic flow. We have shown that our SIEM is resilient against DDoS and port scanning attacks launched by the attacker. Therefore, in order to ensure availability, integrity and authenticity security goals we discussed in section 2, we propose that the defenders use flow monitoring tools that exist on the market or one similar to what we have implemented in this mission along with packet inspection tools to secure their networks.

Finally, in terms of improvements to our IDS we could obviously start by monitoring DNS queries to ensure that no potentially malicious domains are being connected to, or at all queried. Not only would this help us detect command and control communications, but it is also very useful in preventing users from going to phishing sites as well as other dangerous domains. We could also implement a strike-system for the notifications and warnings. If a machine’s IP address was mentioned a certain number of times in notifications, then it could generate a warning for the defender to look at. This way even minorly suspicious network activity, if it keeps repeating, will be dealt with. There could also be a strike system for the generated warnings. If a machine’s IP address has been responsible for too many warnings that have not been taken care of by the defender than an IP-Tables rule could be automatically put into effect, and the problematic machine could be prevented from communicating on the network. This way even if the defender is not present to work with the IDS, the network will still be able to be somewhat protected by actions taken by the IDS itself. Finally, we would have liked to introduce a triaging option for the warnings. If a warning was confirmed by the defender to be a false-positive, than it could be triaged and that specific warning, between those specific addresses, using that specific protocol is not actually a threat and should not produce a warning. A triaging system would allow for the IDS to adapt and change itself based on how the network it is defending operates.

## 

## Sources

* [1] C. R. Taylor and C. A. Shue, Validating Security Protocols with Cloud-Based Middleboxes. 2019. Available at: https://web.cs.wpi.edu/~cshue/research/cns16.pdf
* [2] L. Rendek, "Linux DNS server BIND configuration - LinuxConfig.org", Linuxconfig.org, 2019. [Online]. Available: https://linuxconfig.org/linux-dns-server-bind-configuration. [Accessed: 3- Dec- 2019].
* [3] "How to Disable ICMP Redirects in Linux", Linuxpoison.blogspot.com, 2019. [Online]. Available: https://linuxpoison.blogspot.com/2010/01/how-to-disable-icmp-redirects-in-linux.html?m=1. [Accessed: 05- Dec- 2019].
* [4] "How to implement a DNS server using BIND", Scaleway, 2019. [Online]. Available: https://www.scaleway.com/en/docs/implement-dns-server-with-bind/. [Accessed: 05- Dec- 2019].
* [5] D. Brown, "Create DNS Record For Subdomain", *The Cave*, 2019. [Online]. Available: http://www.thecave.info/create-dns-record-for-subdomain. [Accessed: 06- Dec- 2019].
* [6] Center for Applied Internet Data Analysis. “FlowScan Analysis.” CAIDA, https://www.caida.org/tools/utilities/flowscan/analysis.xml.
* [7] "High-Speed Web-based Traffic Analysis and Flow Collection", *ntop*, 2019. [Online]. Available: https://www.ntop.org/products/traffic-analysis/ntop/. [Accessed: 04- Dec- 2019].
* [8] Parker, Jeff, and Jeff Parker. “Netflow - What Is It and How Does It Work? A Complete Guide.” PC & Network Downloads - PCWDLD.com, 30 Mar. 2019, <https://www.pcwdld.com/what-is-netflow>.
* [9] "What's The Simple Comparison Between Bro And Snort? | Hacker News". *News.Ycombinator.Com*, 2014, <https://news.ycombinator.com/item?id=7726881>.
* [10] Rødfoss, Jonas Taftø. “Comparison of Open Source Network Intrusion Detection Systems.” 24 May 2011, <https://core.ac.uk/download/pdf/30827238.pdf>.
* [11] "HOW DNS TUNNELING WORKS AS C&C COMMUNICATION CHANNEL FOR BOTNET", *Security4vn.com*, 2019. [Online]. Available: https://www.security4vn.com/2019/08/how-dns-tunneling-works-as-c.html. [Accessed: 06- Dec- 2019].
* [12] A. White and B. Clark, *BTFM*. Columbia, SC: CreateSpace, 2017.

## Appendix

Figure 1. Routing table on host 2

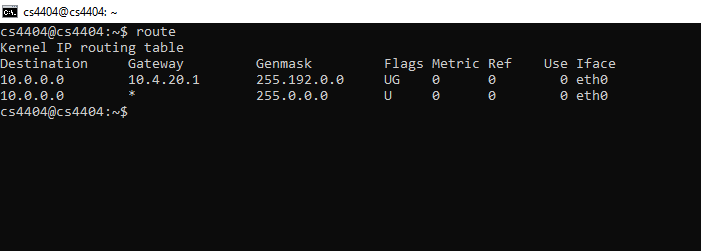


Figure 2. Routing table on host 3

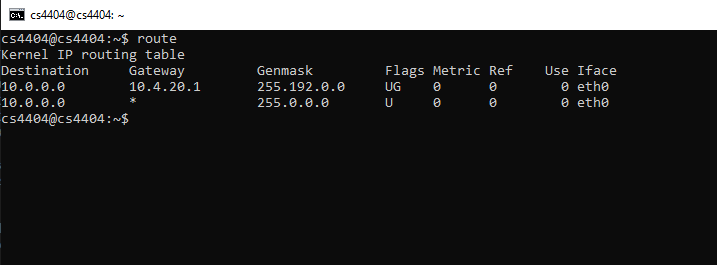


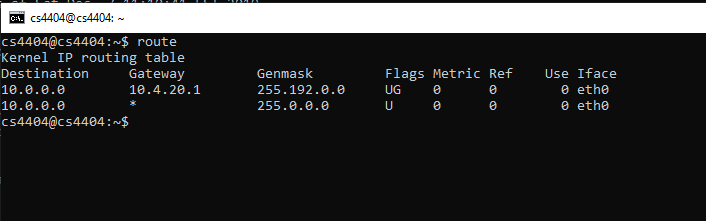
Figure 3. Routing table on host 4  


Figure 4. Interfaces file on host 2

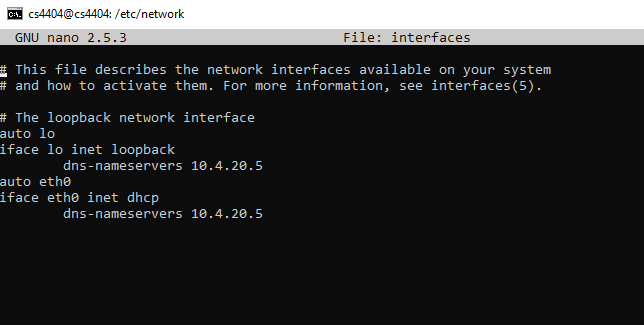


Figure 5. Interfaces file on host 3

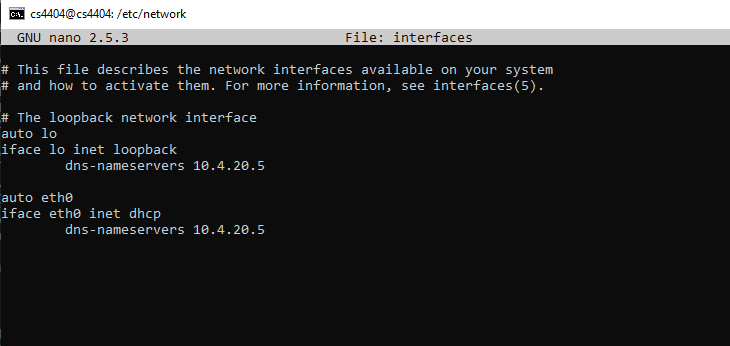


Figure 6. Interfaces file on host

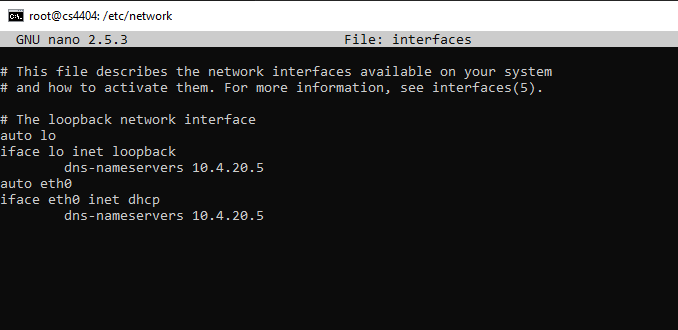


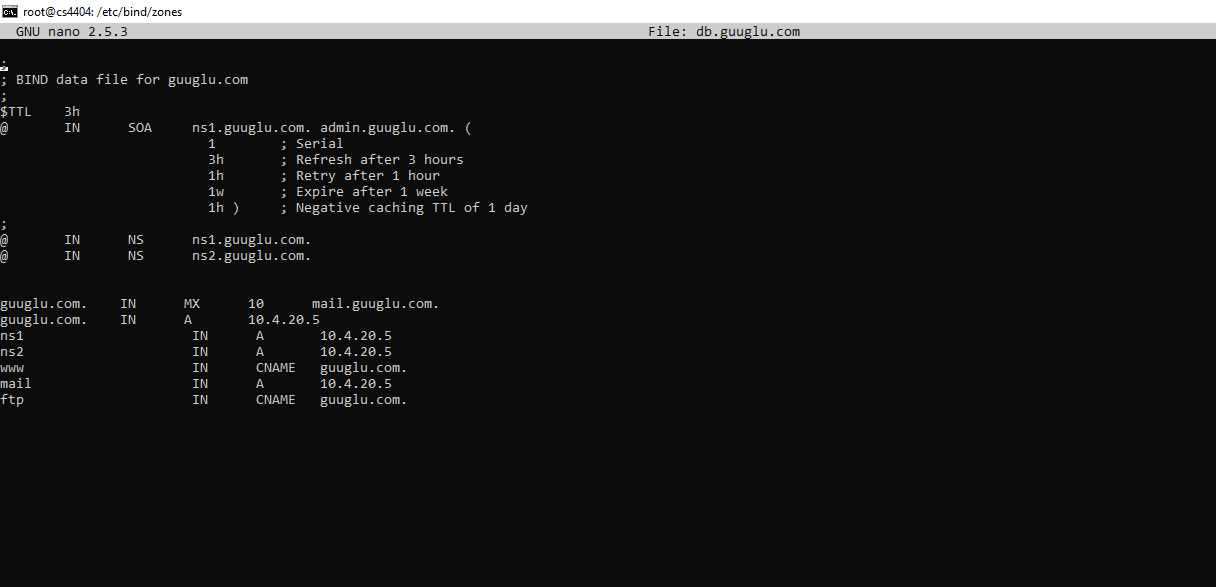
Figure 7. BIND file on host 5 for “[www.guuglu.com](http://www.guuglu.com)” named db.guuglu.com  


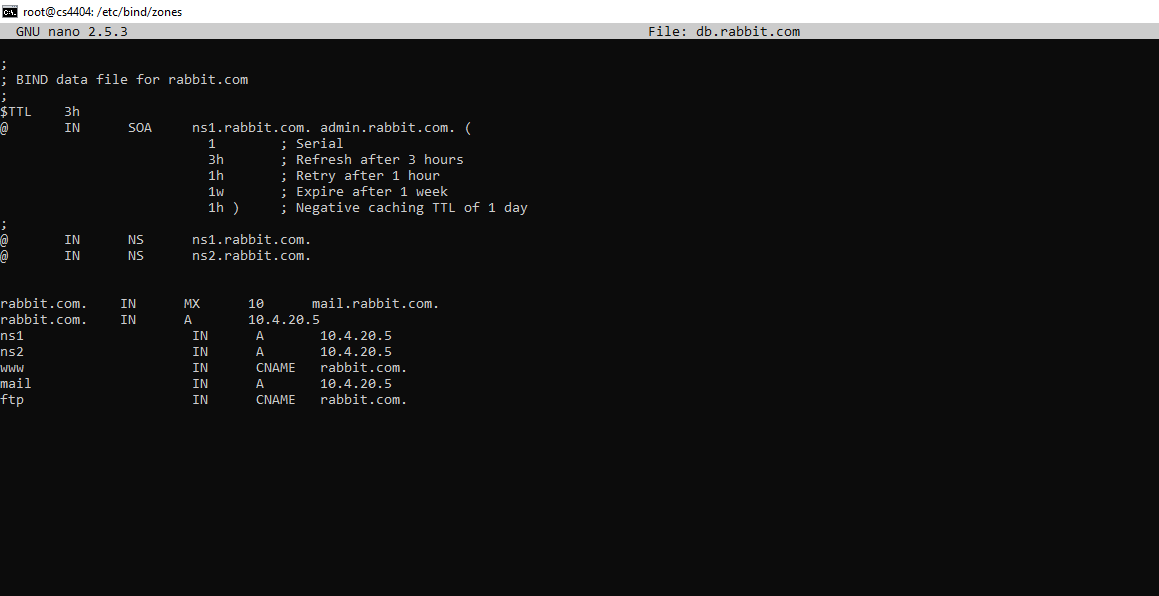
Figure 8. BIND file on host 5 for “[www.](http://www.guuglu.com)rabbit.com” named db.rabbit.com  


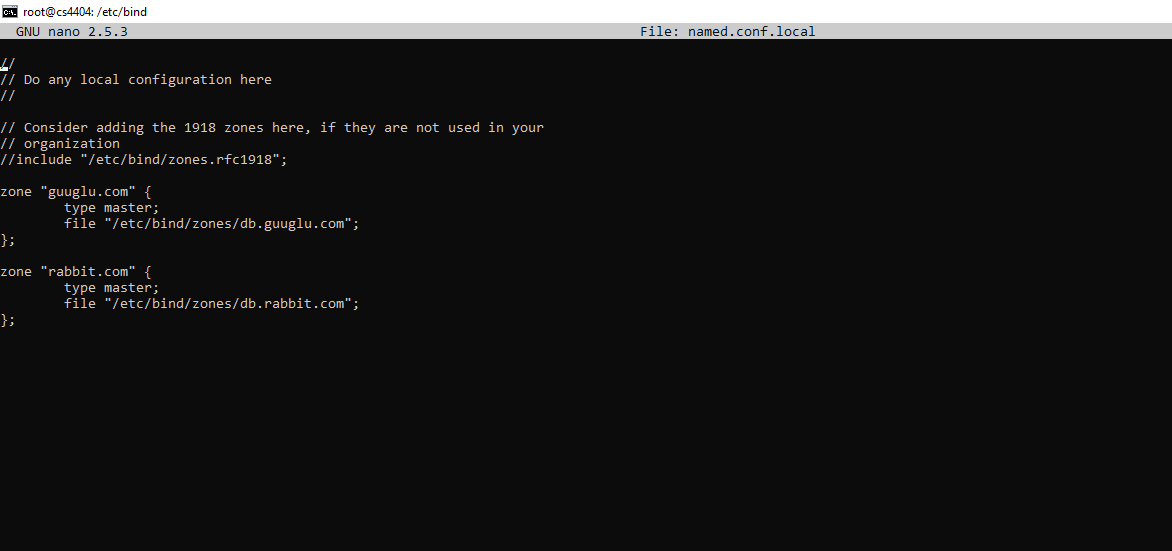
Figure 9. named.conf.local file on Host 5 containing the path to our BIND file  


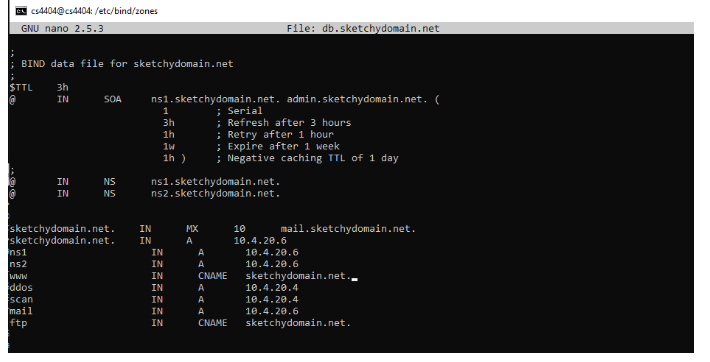
Figure 10. db.sketchydomain.net BIND configuration file located on Host 6  


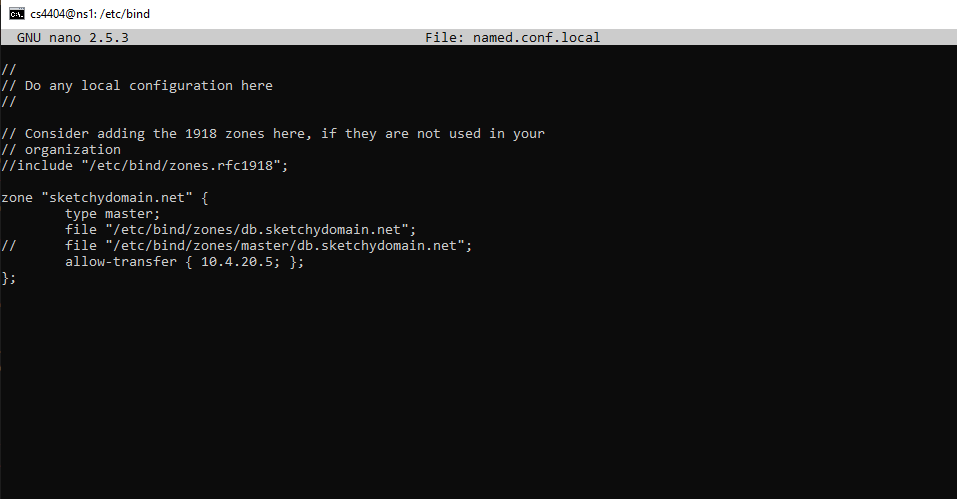
Figure 11. named.conf.local file located on Host 6 containing pointer to the location of the BIND file  


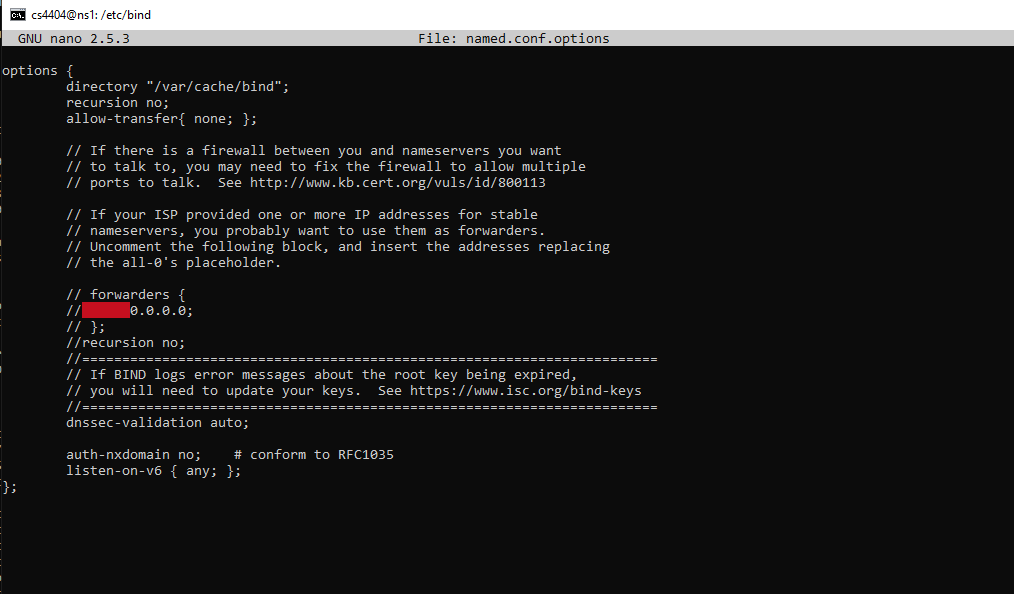
Figure 12. named.conf.options located on Host 6 with modifications to disable recursion and allow transfer  


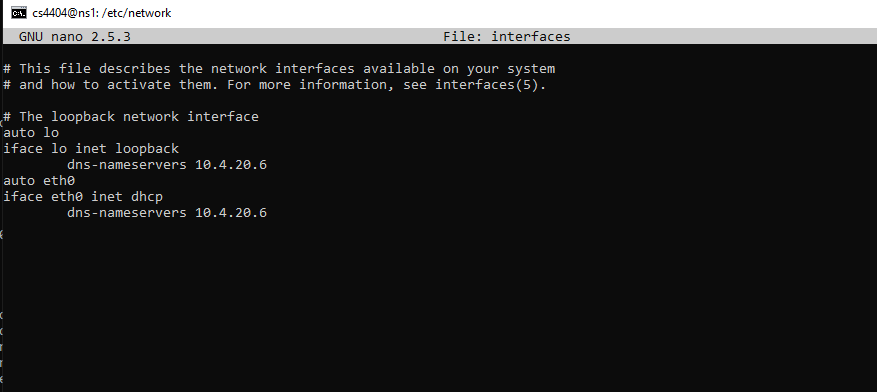
Figure 13. interfaces file on Host 6 to specify a nameserver  


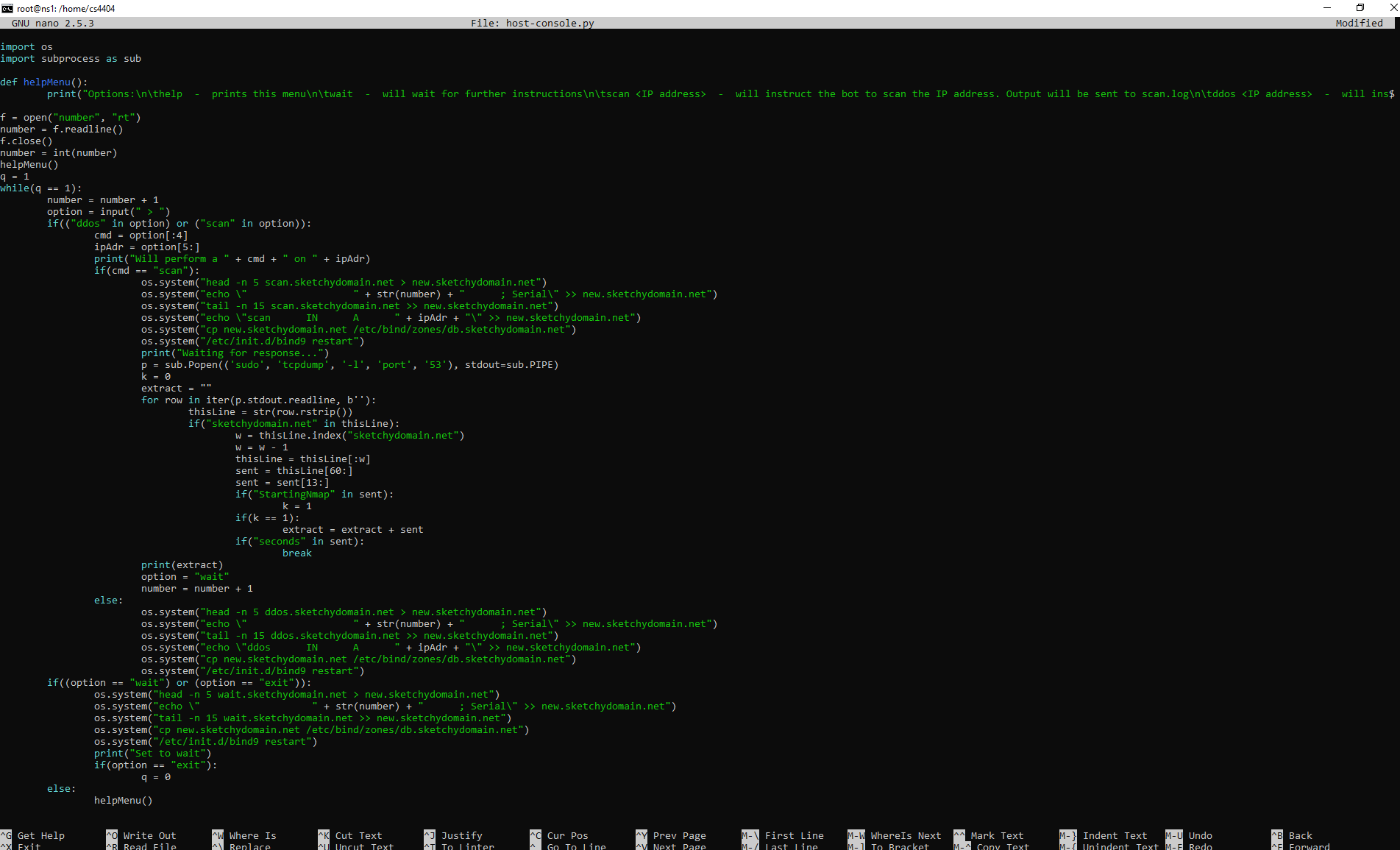
Figure 14. host-console.py python script located on Host 6  


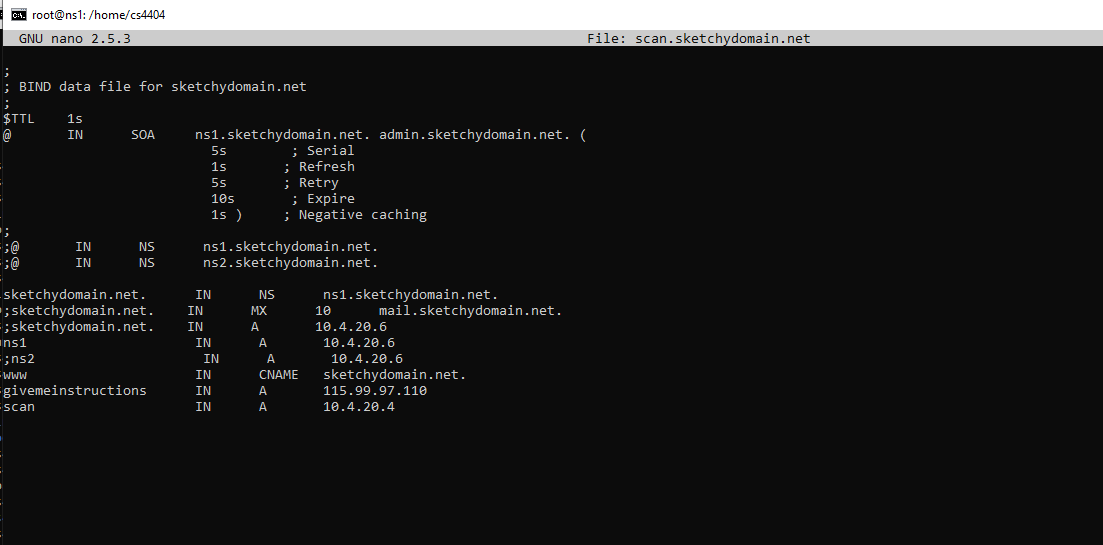
Figure 15. scan.sketchydomain.net located on Host 6  


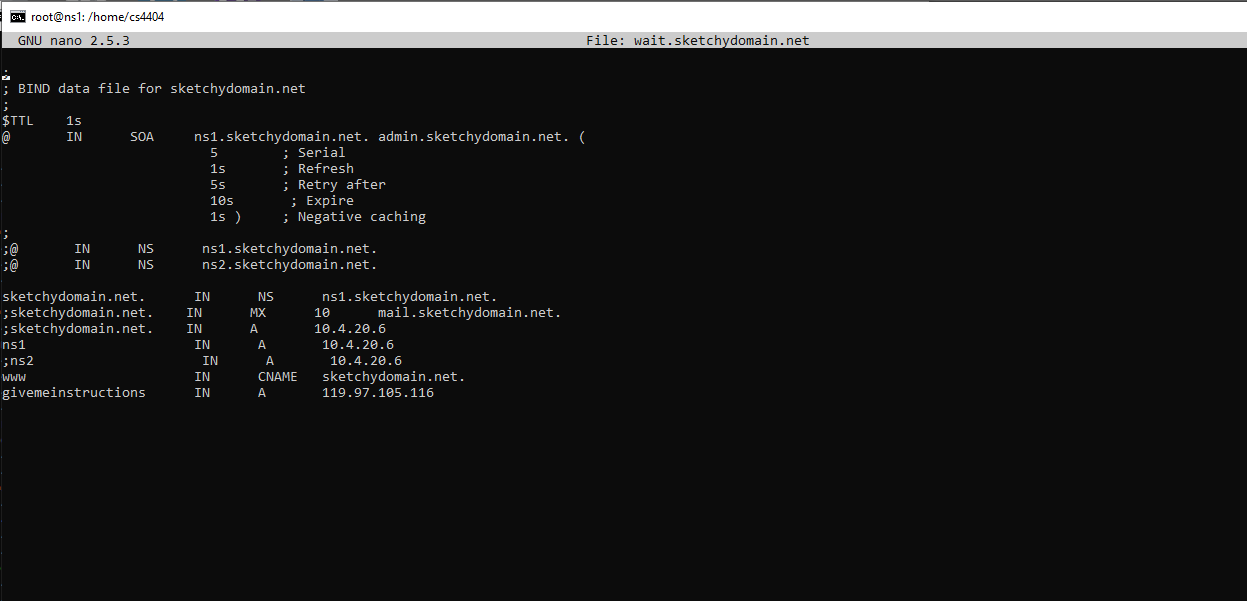
Figure 16. wait.sketchydomain.net located on Host 6  


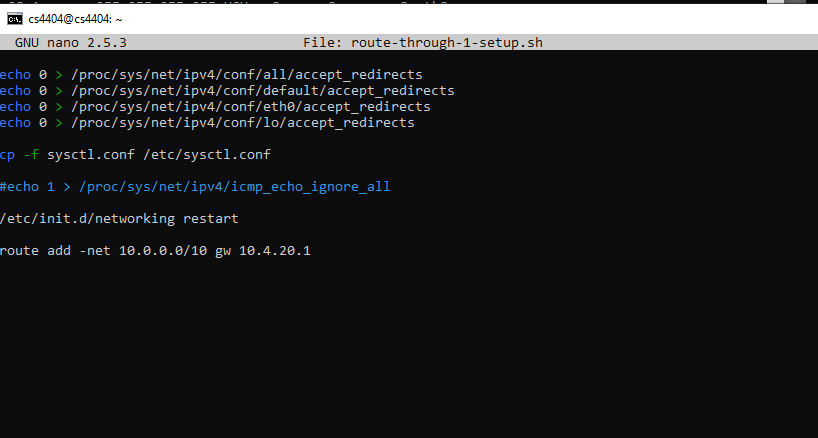
Figure 17. route-thru-1-setup.py scriot located on Host 2  


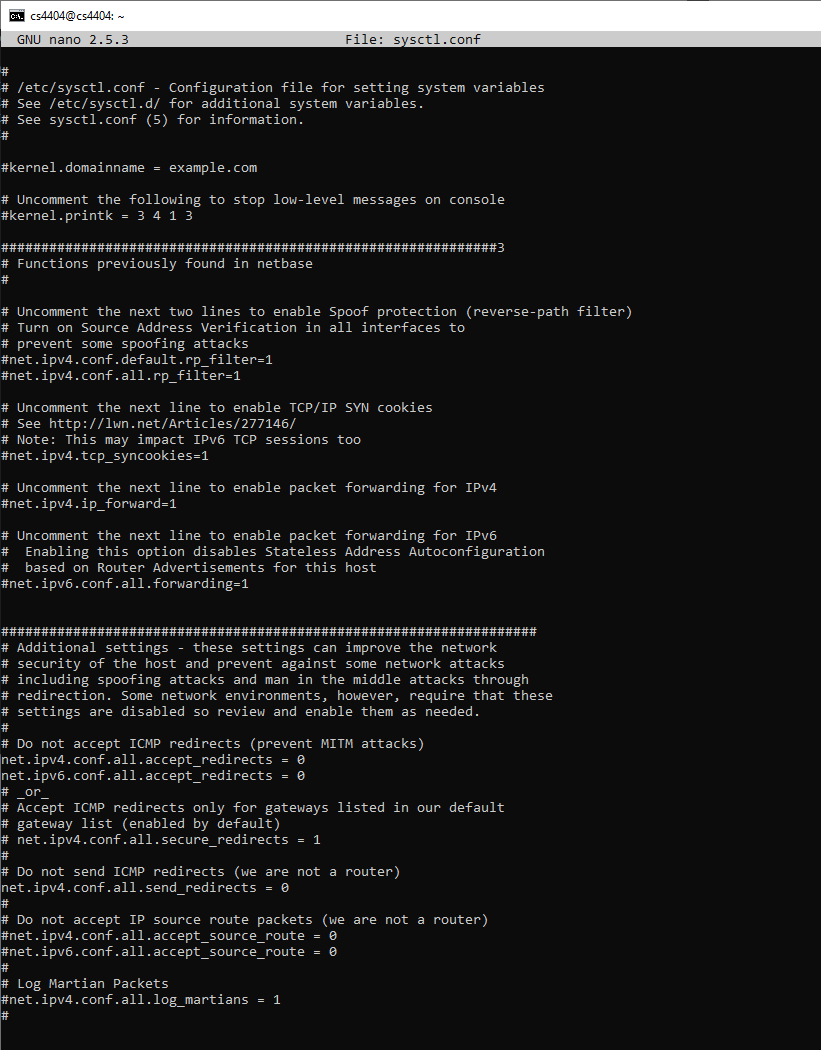
Figure 18. sysctl.conf file located on Host 2  


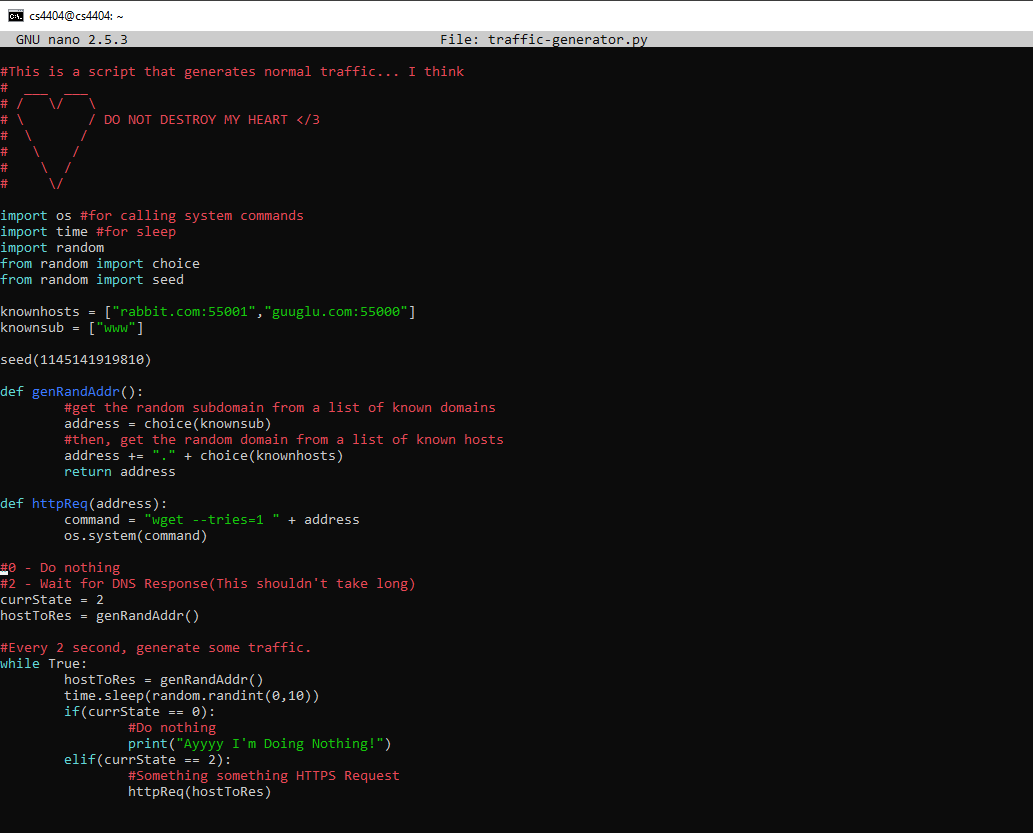
Figure 19. traffic-generator.py python script located on Host 2

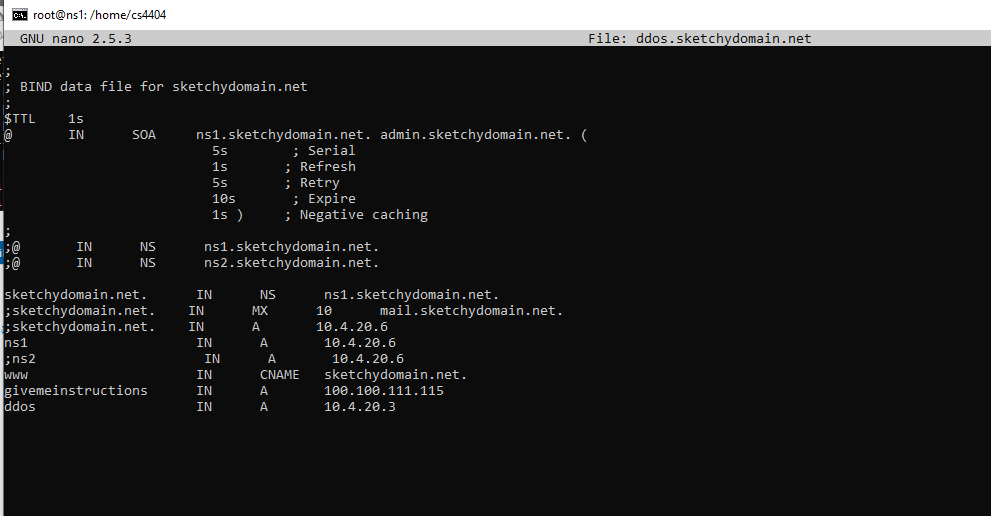
Figure 20. ddos.sketchydomain.net file located on Host 6

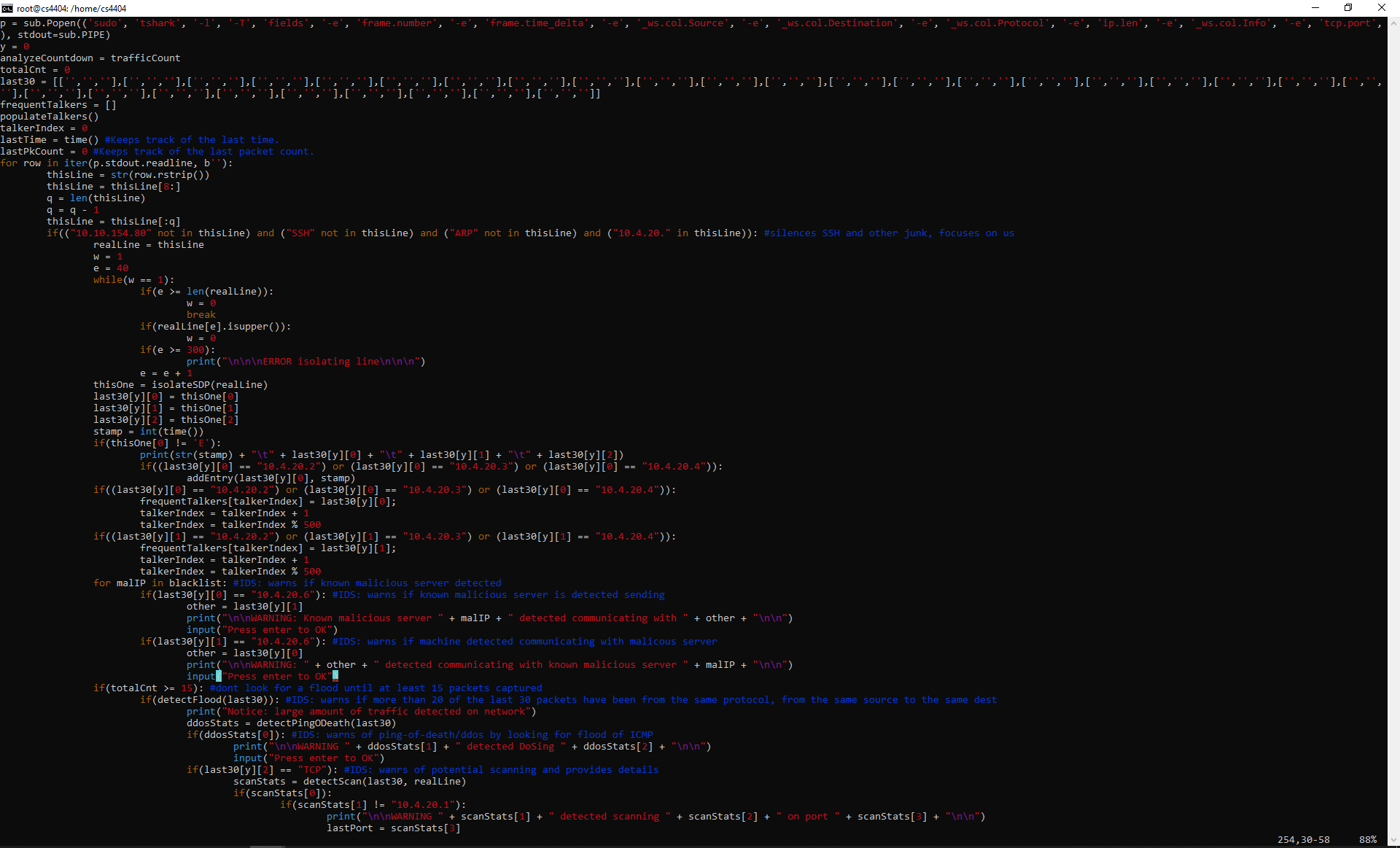
Figure 21. Part of the IDS python code that monitors traffic (please see Figure 21 supplement included in the zip file directory for the full code)

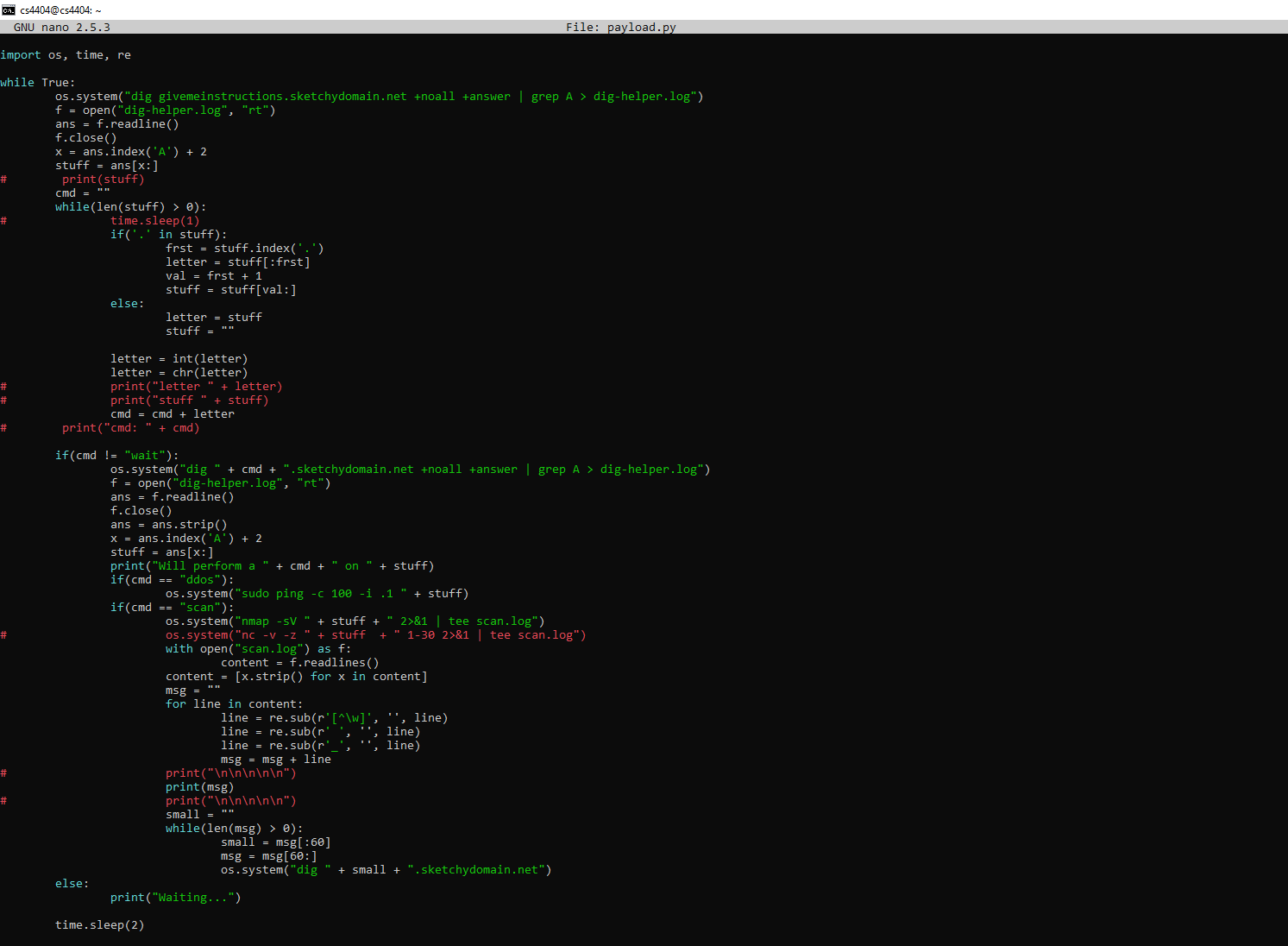
Figure 22. payload.py file located on Host 2  


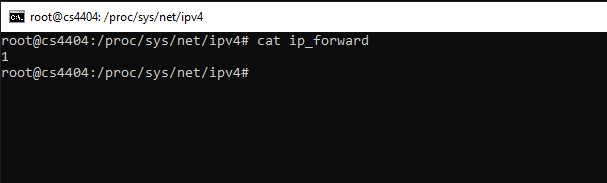
Figure 23. File showing enabled ip forwarding and “ip\_forward’ file  


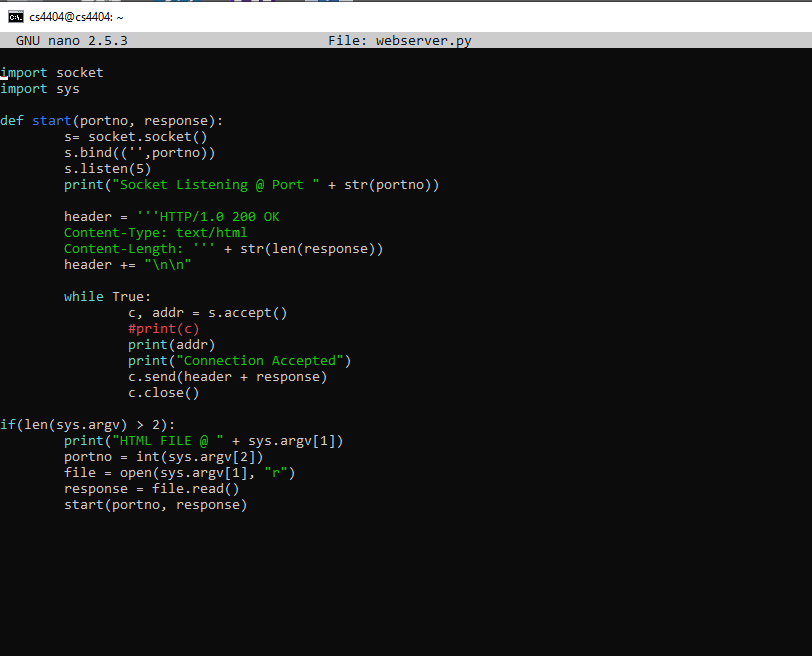
Figure 24. webserver.py code located on Host 5  


Figure 25. dif.html file located on Host 5

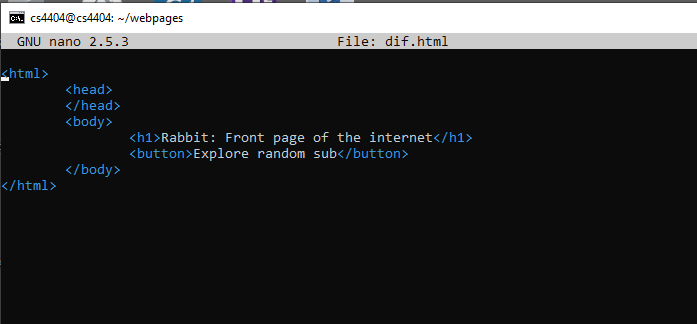


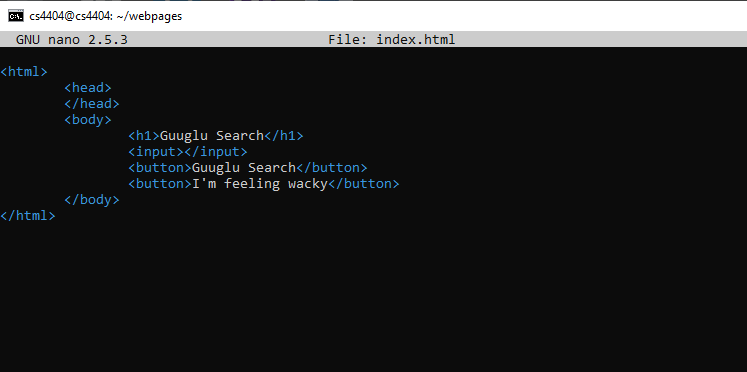
Figure 26. index.html located on Host 5  


Figure 27. route-thru-1.sh script located on Host 5

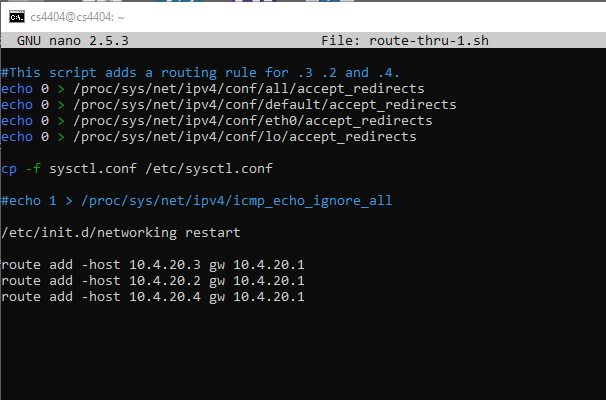


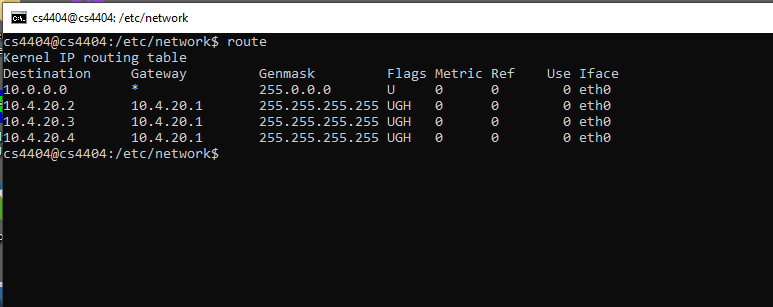
Figure 28. routing table located on Host 5 showing new routing rules  


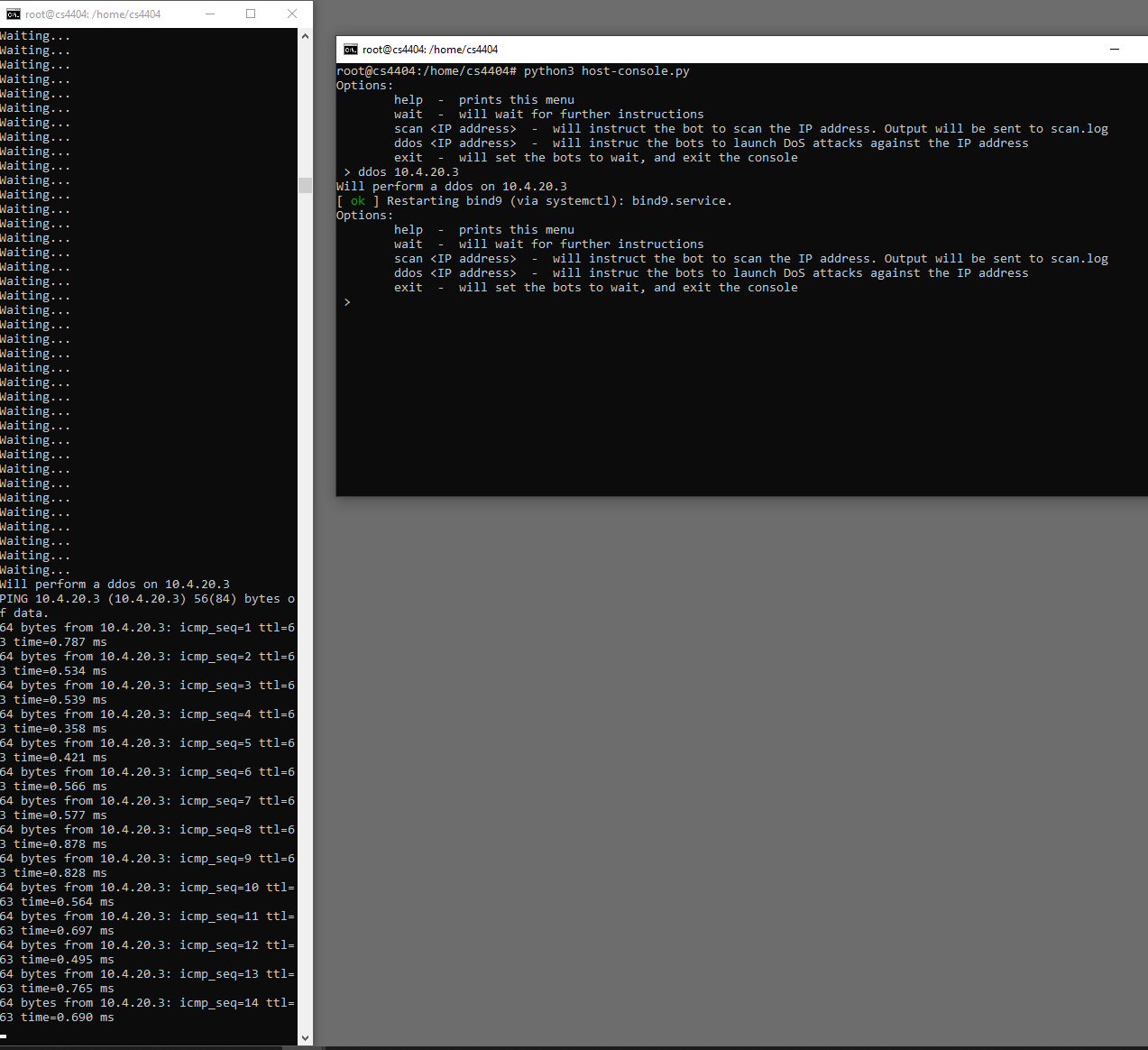
Figure 29. Attack Demonstration of DDoS Instruction

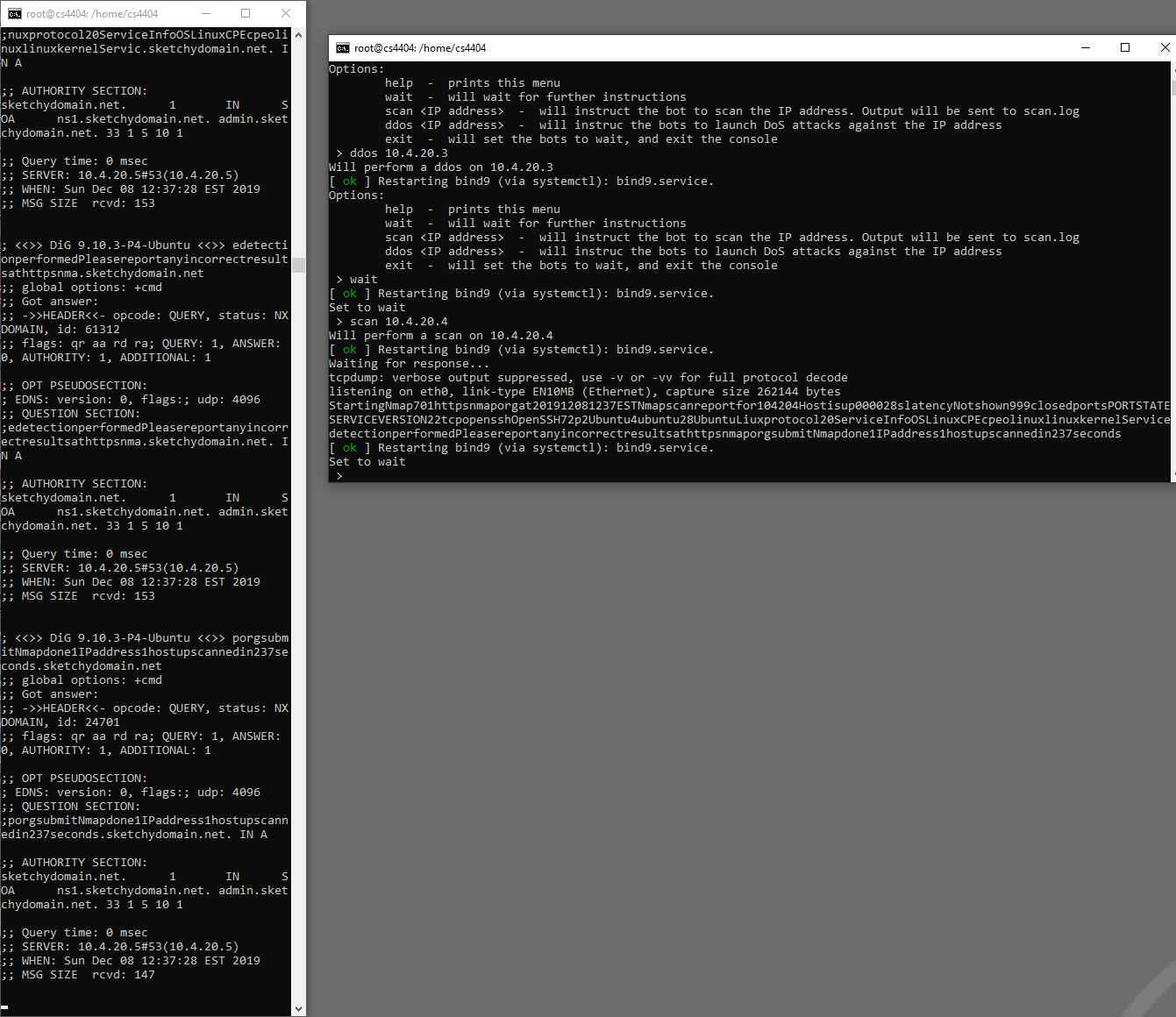
Figure 30. Attack Demonstration of the nmap scan instruction by the command and control infrastructure, and response string extracted from the bot

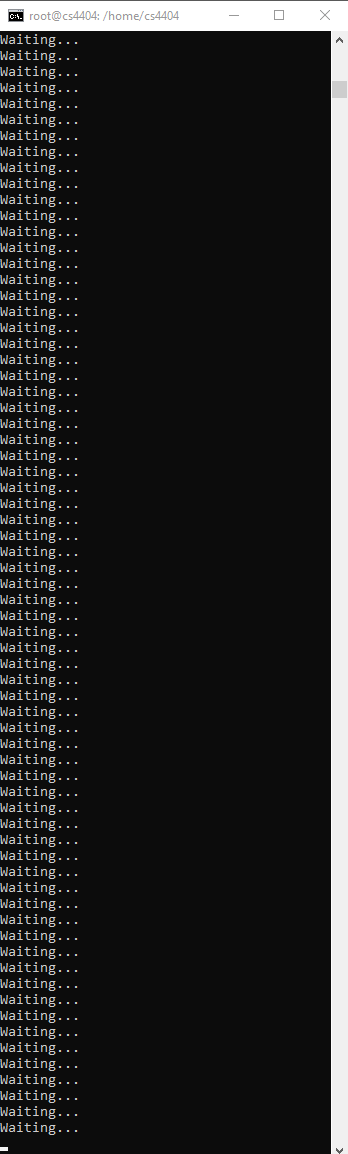
Figure 31. Attack showing bot host waiting for instruction during the attack

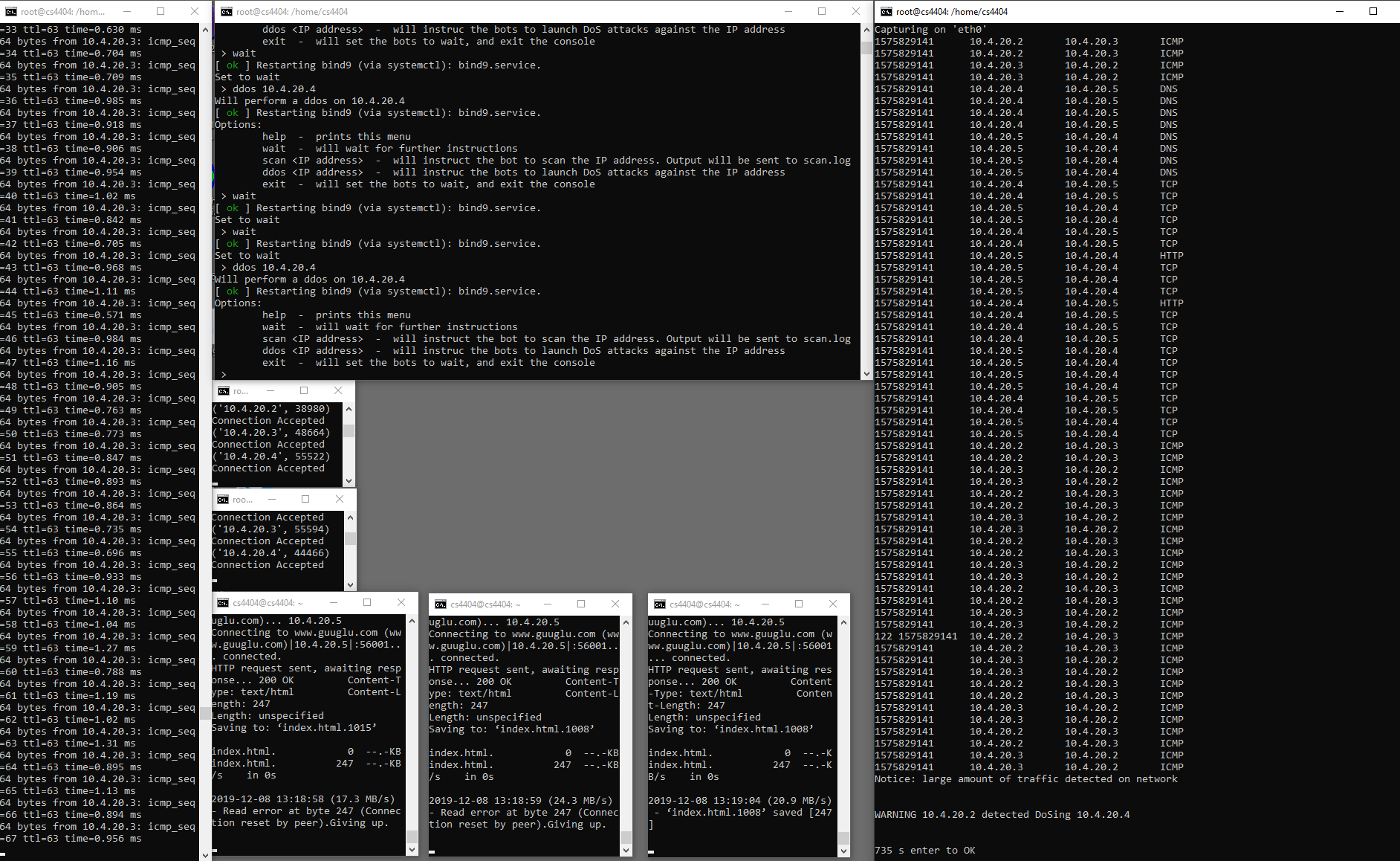
Figure 32. Defense showing detection of DDoS  


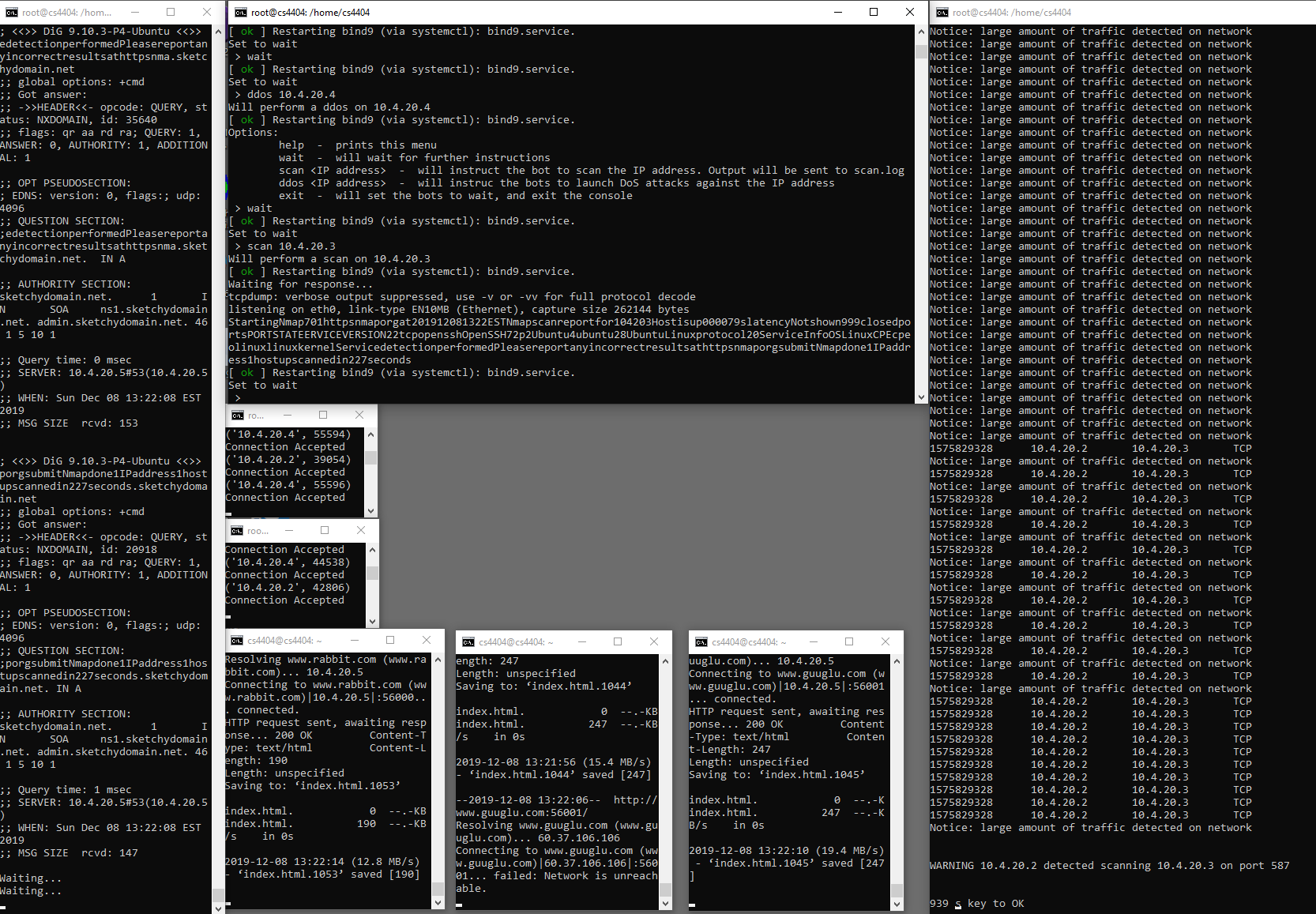
Figure 33. Defense showing detection of port scanning

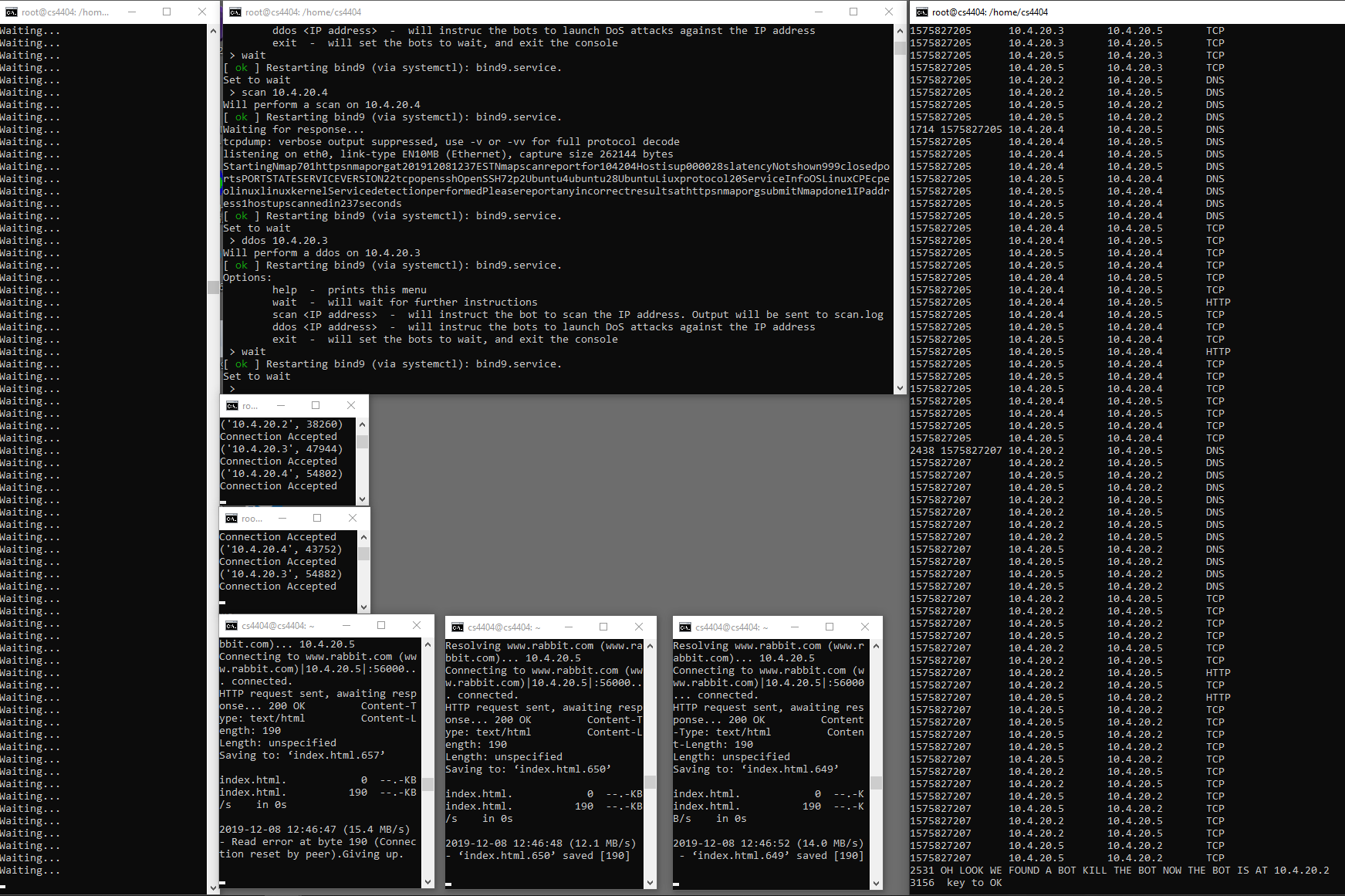
Figure 34. Defense showing the detection of script’s repetitive “heartbeat” communication

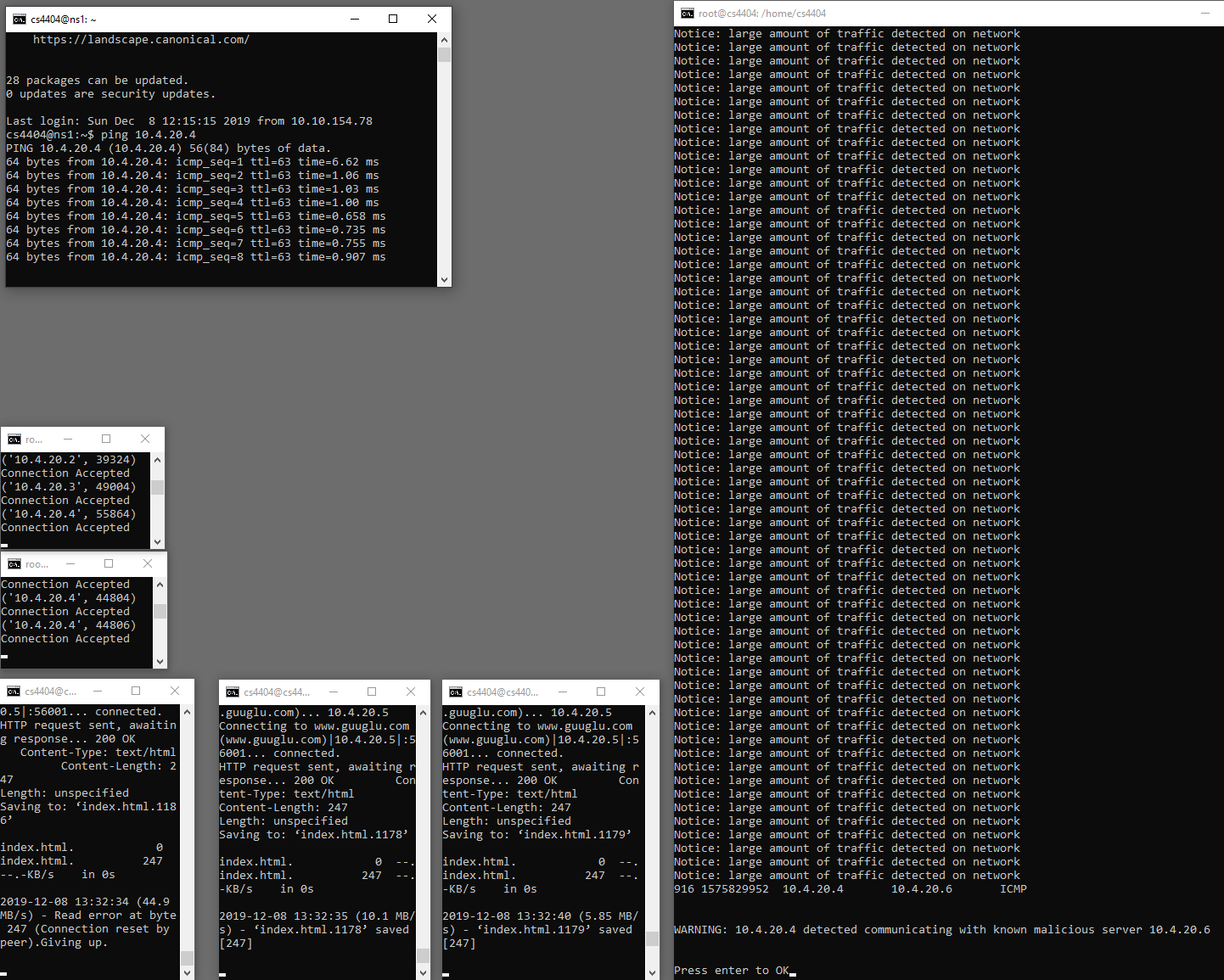
Figure 35. Defense showing detection of blacklisted IP 10.4.20.6  


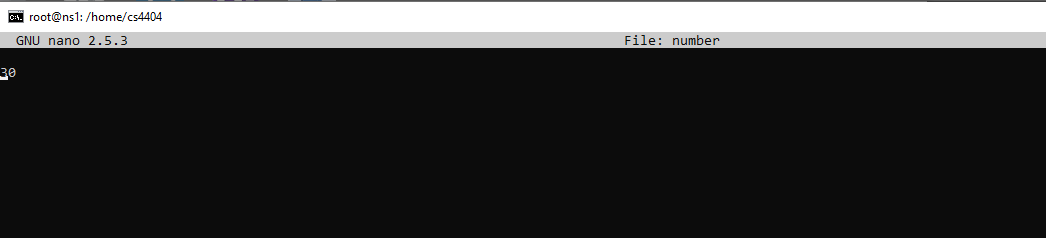
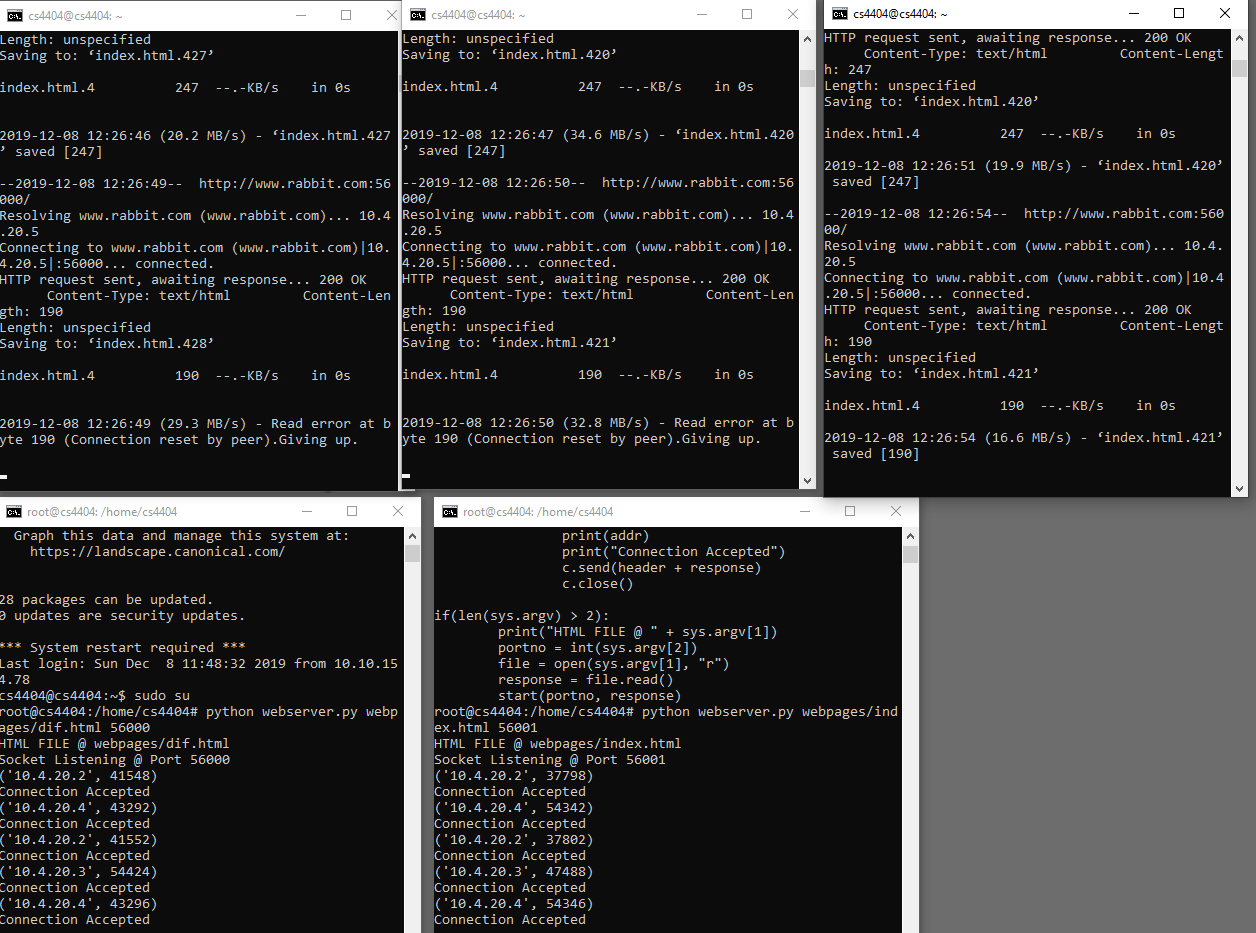
Figure 36. Number file located on host 6 

Figure 37. Demonstration of our entire infrastructure working. Webservers are on the bottom, clients reaching out and getting the hosted webpages are on top.  
  
Figure 38. named.conf.local file located on Host 5  
