

# **DDOS-Dels**

## **Code Manual Documentation**

Authors: Eugene Tan, Abdurarraheem Elfandi, Jackson Klagge

### **User Documentation:**

The operating system is dependent on the Linux system. For the testing purposes of the proposed system, Ubuntu version 20.10 was used for installation, package installations, and overall testing environment for the entire system network.

Installation of the particular system is dependent on a multitude of Linux packages for metric measurement, network functionality, and dependencies on tools used within the system. Therefore, some prerequisite packages are:

- G++
- iperf
- Ifupdown
- Net-tools
- Vim (Or text editor of choice)
- Nload
- Cmake
- Git

Each of these particular packages can be installed using command: “sudo apt-get install <package>”. An example of this usage would be:

“sudo apt-get install g++”, which after providing the sudo user’s password would install the package g++ onto the Ubuntu OS, enabling use of the C++ compiler and C++ dependencies.

### **VirtualBox Configurations:**

For the purpose of the system we want our virtual machines to all be related and communicable without escaping the virtual environment provided so we will use the network type of “Host-Only Network” within the network configurations of each virtual machine. The network type of each machine should only be modified after initially performing the installations of all aforementioned packages.

To properly configure the VirtualBox settings after assuming proper installation of all packages above, we navigate into each Virtual Machine’s settings and the network section. After moving the network type to be “Host-Only”, VirtualBox will formulate a “VirtualBox Host-Only Ethernet Adapter” which has the capability of connecting all machines which are connected to this network adapter. To modify the settings for this adapter, one must navigate to VirtualBox’s Host Network Manager where we will disable the DHCP Server by unchecking the box indicating DHCP server.

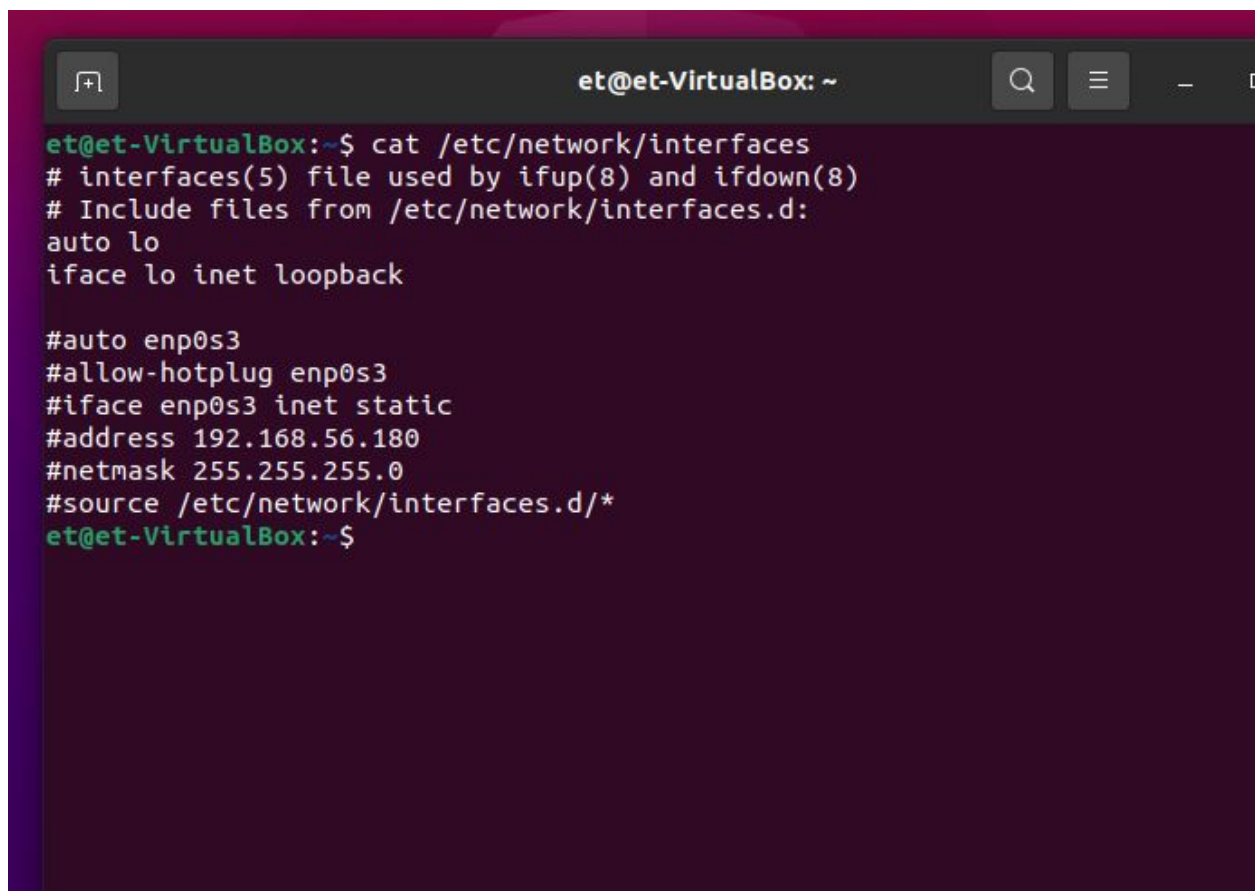
Furthermore, another modification done for the VirtualBox settings is to ensure each virtual machine has a differing MAC address by resetting each MAC address manually through virtual machine network settings.

Lastly, a shared folder for each virtual machine is an ideal add-on that enables flexibility with data transfer and moving files and packages quickly between the host machine and the virtual machine. This is easily configured through “Insert Guest Additions CD” in each virtual machine’s system settings, and mounting a virtual machine in each virtual machine’s device settings. Detailed instructions of this operation can be found in the following site:

<https://www.howtogeek.com/189974/how-to-share-your-computers-files-with-a-virtual-machine/>

### Setting a Static IP Address:

As we want all of our machines to retain the IP assigned to them we assign static IP addresses to each machine by modifying the file contents of “/etc/network/interfaces”. Within this file we configure an IP address, netmask, and configuration that will be executed upon each system boot. Therefore, a user could determine their network interface by executing a command such as “ifconfig” or “ip a” to see what their network interface is named.



```

et@et-VirtualBox: ~
et@et-VirtualBox:~$ cat /etc/network/interfaces
# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
auto lo
iface lo inet loopback

#auto enp0s3
#allow-hotplug enp0s3
#iface enp0s3 inet static
#address 192.168.56.180
#netmask 255.255.255.0
#source /etc/network/interfaces.d/*
et@et-VirtualBox:~$
  
```

An example is this particular network file which configures the loopback interface as well as the “enp0s3” interface under a static IP address of 192.168.56.180. This static IP address can be assigned to any arbitrary IP address. A detail to note is that while the above interface is named “enp0s3”, the name of the interface assigned by VirtualBox can vary depending on BIOS settings or other VirtualBox configurations.

### Building and Usage for Each Function:

As each function is dependent on the prerequisite packages mentioned in the User Documentation introduction, the following assumption is made that all mentioned packages have been successfully installed and each mentioned source file exists on the User's machine.

### Wireshark Collection:

To perform a wireshark packet collection, one needs to install Wireshark which can be easily obtained and installed via <https://www.wireshark.org/#download>. Upon completing the installation corresponding to one's host OS, execution of Wireshark will enable a user to perform a collection of their network packet data. For ample collection, a period of 1 hour should be sufficient for purposes of singular collection. When saving this particular packet capture, it should be saved in the format of a .pcap, as this is the preferred format for input into the MIMIC system.

### MIMIC System:

For convenience of installation, to build the MIMIC system quickly and efficiently on a virtual machine we included a script called "mimic-install-build.sh", a shell script that when executed automatically clones the MIMIC GitHub repository and installs all supplementary packages required in conjunction with the system. After executing this particular shell script, the MIMIC systems of MIMIC-extract and MIMIC-replay will be built and executable according to their respective use cases.

Within the scope of this system we will only be applying MIMIC-extract's behavior of extracting a provided .pcap file (formulated from Wireshark collection). To perform this action, the pcap file we wish to extract must exist on the Virtual Machine we are performing the extraction action on. Assuming the pcap file exists and we are in the MIMIC directory, the command to perform extraction is:

```
"/mimic-extract <.pcap file to be extracted> > <output file name>.csv"
```

After execution of this command, the extracted .pcap file contents will be piped into the comma separated value specified in the commandline above. This outputted file contains CONN, SEND, and WAIT events formulated from the packet data contents of the .pcap file provided.

### Parsing CSV Files Into IP Addresses:

Assuming we have completed the MIMIC-extraction process and have obtained a usable .csv file containing events read from the .pcap file provided, we can now process and obtain a set of source and destination IP addresses that indicate the addresses involved in the network traffic within the pcap file. To build the csv parsing system we first must compile the source file:

```
"g++ parseCSV.cpp -o parseCSV"
```

This command will compile our parseCSV.cpp source file into an executable file called "parseCSV". Once this file has been compiled into its executable form we can then execute this program with a .csv file containing CONN, SEND, and WAIT events (file outputted by the MIMIC system's extraction behavior):

```
"./parseCSV <.csv file containing network events>"
```

As a result of successful execution, two files will be produced:

- Source.ips: a list of all source IP addresses that initiated client connections with arbitrary outside IP addresses.
- Destination.ips: a list of all IP addresses that were interacted with and connected to during the duration of Wireshark capture.

These two IP files will be used for actions such as dynamic IP binding to a virtual machine's network interface and IP routing.

### **Dynamic IP Binding onto VM Network Interfaces:**

Assuming that we have successfully parsed a .csv file's contents into readable Source.ips and Destination.ips files, we can now dynamically assign these IP addresses temporarily to a virtual machine's network interface and route each IP address through VirtualBox. IP routing occurs simultaneously with IP binding after execution of this program, enabling an IP address arbitrarily bound onto the interface be reachable from another virtual machine on the host-only network.

Assuming all IP addresses we wish to be bound exist in a file with all IP addresses separated on a new line, to bind the addresses we first compile the program's source file:

```
"g++ ipBind.cpp -o ipBind"
```

This command will compile our ipBind.cpp source file into an executable file. To use this executable file we must also know the static IP address bound to the virtual machine, as routing IP addresses bound through this static IP address will enable it to be reached from other IP addresses routed through VirtualBox. The usage for the program is as follows:

```
"sudo ./ipBind <File of IPs> <Static IP Address>"
```

The result of executing this command is the modification of the routing table for the virtual machine. The default route set for the virtual machine will become the static IP address, and all corresponding IP addresses to be bound within the IP file will be bound and routed through this static IP address. As a result all IP addresses on the interface will be routed through VirtualBox.

To test if this operation has successfully been completed, on a secondary virtual machine within the same host-only network, one can ping any of the IP addresses listed on the primary virtual machine successfully.

Note: This IP binding file assumes the network interface for each machine is "enp0s3", should this not be true for the user, modify line 70 within ipBind.cpp to reflect the name of the interface on the user's machine.

### **Spawning iperf Servers:**

To emulate network traffic, iperf is used as a means of generating throughput between IP addresses bound on distinct machines. The tool iperf leverages clients and servers, therefore we must spawn an iperf server accordingly for each IP address bound onto an interface. An example of this is if Source.ips was bound onto this particular machine's network interface, we would use this same file to spawn the iperf servers, as we would need knowledge of which IPs exist within our current machine.

Spawning these servers is quick and complete, the result of which is a daemon server process for each 'n' IP addresses we have bound. Leveraging the daemon option provided within iperf version 2, we can have each server service multiple client connections simultaneously and persist in the background. Therefore, to use this program we must compile our source file into an executable:

```
"g++ spawnServers.cpp -o spawnServers"
```

Upon successful compilation the executable program can be executed by calling the produced executable with the same IP file that was used to bind IP addresses to the interface:

```
“./spawnServers -s <IP file>”
```

As a result of executing this command, an iperf server will be spawned corresponding to each IP bound onto the network interface; therefore, the same file used to bind IPs to the interface should be used here so that we can assign an iperf server to each IP.

To verify that iperf servers are functioning in the background as daemon processes we can use ‘pgrep’ in conjunction with “wc” to see how many servers are ongoing:

```
“pgrep iperf | wc -l”
```

Output will indicate the number of iperf servers, the integer returned should be the same as the number of lines within the inputted ip file.

### **Connecting iperf Clients to iperf Servers:**

Once iperf servers have been spawned on a different machine, and assuming all the above steps have been completed and both machines have IP addresses bound onto their corresponding interfaces, we can formulate client to server connections. To formulate client connections we must compile our client connections source file:

```
“g++ connectClients.cpp -o connectClients”
```

The executable “connectClients” requires 3 inputs with an additional optional input of bandwidth. These three inputs are the source IPs, destination IPs, and duration of the test. To better explain how these function, source IPs in this context are the IP addresses bound to the interface and the source of the iperf client connections. Destination IPs in this context are the servers we’d like to connect to and exchange data with. Lastly, duration of the test is simply how long we would like client connections to persist before all are removed. Therefore, the usage is as follows:

```
“./connectClients -c <Destination IP file> <Source IP file> <Duration of test (integer)>
```

```
<OPTIONAL: Bandwidth in bits>”
```

With iperf servers running on the IPs specified within the destination IP file, when executing this program, client connections will be randomly selected and formulated by IPs within the source IP file. Additionally, client connections persist for randomized durations of time and are rescheduled upon completion with a new destination IP from a new source IP. Therefore, all client connections run simultaneously in the background.

To verify that there is indeed network traffic existing we can run:

```
“nload”
```

In the terminal to see a visual graph indicating the flow of incoming and outgoing traffic. As all clients are instructed to perform bidirectional tests, both the flow of incoming and outgoing traffic should display statistics indicating movement.

### **Installation of the DDoS Tool**

The tool selected was the BoNeSi DDoS Botnet Simulator, a tool capable of simulating the behavior of a botnet executing a DDoS attack. Installation of the tool is fairly straightforward:

1. Clone the repository: “git clone <https://github.com/Markus-Go/bonesi.git>”
2. Update the libraries and install required packages: “sudo apt-get update”, “sudo apt-get install autoconf automake gcc make libnet1-dev:

3. Afterward we navigate into the bonesi directory and execute the configuration script: “sudo ./configure” which will configure the bonesi system.
4. Execute “autoreconf -f -i” to install auxiliary files.
5. Execute “sudo make” which will build bonesi, followed by the command “sudo make install”
6. The BoNeSi system has now been installed onto the machine and can be verified by running “bonesi -h” in the terminal.

### **Execution of the DDoS Tool:**

Bonesi supports various tools that can be applied to a variety of scenarios and thus has a multitude of options that can be used. For the scope of the system we apply a few of the options that best fit the system. To execute the given attack we use the command:

```
“bonesi -i <bots IP file provided> -p <network protocol> -r <number of packets per second> -d <network interface> <ip address>:<port number>”
```

For the purposes of testing, the network protocol chosen was ‘tcp’, -r was typically omitted to uncap number of possible packets, and the IP address and port number was a selected iperf server that was chosen to be our server of choice to DDoS. Another item to note is the bots IP file which is an included file containing 50,000 inbuilt IP addresses within bonesi, and can be used for testing purposes. The execution of this particular DDoS tool should be done on a virtual machine that is neither running server or client iperf connections but exists on the same network to best simulate a third-party attacker.

To gauge the success of the attack, we launch the iperf server to client connects simultaneously and measure the incoming packet data with ‘nload’. If the attack is successful, the terminal executing bonesi should indicate requests finished correctly and the machine experiencing the bonesi attack should have extremely large amounts of incoming data due to the flooding of packets.

### **Testing DDoS Detection Tools:**

To test DDoS Detection Tools, the ideal environment is such that we install the desired DDoS detection tool onto the virtual machine that has the iperf servers running in the background. Once we have successfully installed the desired detection tool onto the server virtual machine, we can execute client connections from the client iperf machine and begin generating arbitrary congestion-aware network traffic between the server and client machine. During this period of time we can view whether or not the detection software detects this arbitrary network traffic as malicious or not before executing a DDoS attack onto the server machine from our third virtual machine running bonesi. By performing a DDoS attack onto the machine intentionally we can determine if the detection software can detect this malicious attack traffic in the presence of our background traffic. As well as if the detection result remains the same in the presence of and without background traffic. This methodology for evaluation allows false positives and negatives to be found, should an incorrect result be produced during an intentional test.

### **Steps for System Setup:**

1. Configure VirtualBox according to “VirtualBox Configurations”
2. Create 3 Virtual Machines (Client, Server, Attacker) running Ubuntu Version 20.04 or 20.10. Follow the steps outlined in “VirtualBox Configurations” and “Setting a Static IP Address”.
3. On a host machine connected to the internet, perform a Wireshark capture of approximately 1 hour in duration, saving the resulting file in a .pcap format.

4. Provide each virtual machine with the source files of the system.
5. Provide the produced .pcap file to one of the virtual machines. On the machine containing the .pcap file, install the MIMIC system through the “mimic-install-build.sh” script.
6. Extract the .pcap file by executing MIMIC-extract: “./mimic-extract <path to .pcap file> > <name of csv file>.csv” and receive the .csv.
7. With the produced .csv file we can compile the parseCSV.cpp file and then retrieve the the destination and source IP addresses from the .pcap file:  

```
“g++ parseCSV.cpp -o parseCSV”
“./parseCSV <.csv file>”
```
8. With the produced destination and source IP files from parseCSV program, we can copy the source IP file into the shared folder and provide it to the machine serving as our server machine. Keeping the destination IP file on the client machine we can then perform IP binding.
9. To perform IP binding, on the client and server machines we compile the “ipBind.cpp” file to “ipBind” through “g++ ipBind.cpp -o ipBind”. After it has been compiled on both machines, both machines will execute ipBind with their provided IP file and statically assigned IP address:  

```
“./ipBind <ipFile> <Statically assigned IP address for routing>”
```
10. Now that IPs have been successfully bound, on the server machine we will compile “spawnServers.cpp”: “g++ spawnServers.cpp -o spawnServers”. After we have completed this task we can execute: “./spawnServers -s <ipFile used for ipBinding in step 9>”, starting iperf servers for each ip in the ipfile that was bound.
11. On the client machine containing both files of source and destination IP addresses originally parsed, we compile the “connectClients.cpp” source file: “g++ connectClients.cpp -o connectClients”. The compiled source file can then be executed to perform connections to server spawned in step 10 through:  

```
“./connectClients -c <IP file we want to connect to> <IP file we form clients from>
<Duration of the test> <OPTIONAL: Bandwidth for test>”
```
12. With Steps 10 and 11 running, there is ample congestion-aware background network traffic effectively being produced in the environment. At this time, following instructions listed above within the Code Manual one could also introduce DDoS detection tools as well as DDoS software on a third virtual machine distinct from the server and client machine residing on the same network.