Evaluation Report

Evaluated by Group 4 (Anthony Saieva && Qing Teng)

Evaluation for Group 3 (Sophia Yao && Yoongbok Lee)

Link to project: https://github.com/sophiay98/4182_fuzzer

Evaluation Report

- Part 0: Summary
- Part 1: Installation
- Part 2: User Guide
- Part 3: Program Functionality
  - Fuzzer Functionality
  - Server Functionality
- Part 4: Error Handling
  - The following stress tests passed with no issues
    - Using non hex characters in the payload
    - Running the fuzzer with invalid arguments
    - Running the fuzzer with no arguments
    - Running the server with invalid arguments
    - Passing negative integer arguments into the command line
    - Running the server without sudo
    - Printing the entire usage message properly when required
    - Specifying an input file for ip layer tests without the ip layer file fuzzing included
  - Mismatches between layer fuzzing and arguments given
    - Running Fuzzer with empty payload file
    - Running the server with an empty pattern file
  - The following stress tests passed with minor issues
    - Exits without printing an error message when a non integer is used as source port
    - Server doesn't throw an error when invalid hex is passed in as the search pattern
    - Fuzzer Can't read valid hex in a file
    - Running the fuzzer without sudo
  - The following stress tests passed with major issues
- Part 5: Clarity of the Code/Comments
- Part 6: Code Modification

Part 0: Summary

Here are the ratings we assigned for each criteria. You can find more specific explanations in Part 1 to 5. In Part 6, we modified the code to add logging features for both the fuzzer and the server.

Criteria	Rating
Installation	9/10
User Guide	9/10
Program functionality	44/50
Error handling	12.5/20
Clarity of the code/comments	10/10
Total	84.5/100

Part 1: Installation

We cloned the repo and followed the instructions in `installation_guide.md`. We are installing the fuzzer on a Ubuntu 18.04 Desktop running in a VirtualBox VM. To ensure that the dependencies are fully listed in the installation guide, we are starting from a clean virtual environment.

The installation works per the instructions with no problems. We are able to copy and paste most of the commands without modifying anything. The only exception was step 9, which is "run the client by `sudo python3 *src* *dst* *sport* *dport*`". We believe the command should be `sudo python3 client.py *src* *dst* *sport* *dport*`, so we are **deducting .5 points**.

The instructions are mostly clear, being specific shell commands, with the exception of "using net-tools, find the ipv4 address of the server VM". While this is clear enough to people who are familiar with net-tools, we believe the instruction could be made more friendly to the general user by directing listing the `ifconfig` command, so we are **deducting .5 points**.

Overall we are assigning a rating of 9 for installation, with .5 points deducted for each of the two issues mentioned above.

Part 2: User Guide

We followed the instructions in `user_guide.md`.

The user guide includes instructions on how to run default tests on the IP layer, including how to specify which fields to fuzz. An example is included, though there is a typo in the example. We will not deduct any points for this minor issue.

The user guide includes instructions on how to run tests on the IP layer when values are read from a file, including how to specify the file name and the format of the csv file. An example file, `ip.csv`, is provided, and the fuzzer works when running with the example file.

The user guide includes instructions on how to run default tests on the TCP layer, including how to specify which fields to fuzz. An example is included, though specific values are not given.

The user guide includes instructions on how to run tests on the TCP layer when values are read from a file, including how to specify the file name and the format of the csv file. An example for the file is provided, and the fuzzer works when running with the example.

The user guide includes instructions on how to run default tests on the application layer, including how to specify the range of payload size (or exact size) and the number of tests. An example is included, though specific values are not given.

The user guide includes instructions on how to run tests on the application layer when payloads are read from a file, including how to specify the file name and the number of tests. There is a default payload file, though it is not mentioned in the user guide. We will not deduct any points for this minor issue.

The user guide includes how to specify the destination of the packets.

The user guide includes comments on the server, including where to specify the pattern. However, we believe this part could be better organized. There is no example of how to run the server (though it is included in `installation_guide.md`). For these reasons we are **deducting 1 point**.

Overall, the fuzzer instructions are clear and cover each layer and most of them include examples. The instructions for how to use the server and specify the pattern are less clear and do not include examples. The program works per the instructions with no problems. Therefore, we are assigning 9 for user guide.

Part 3: Program Functionality

Fuzzer Functionality

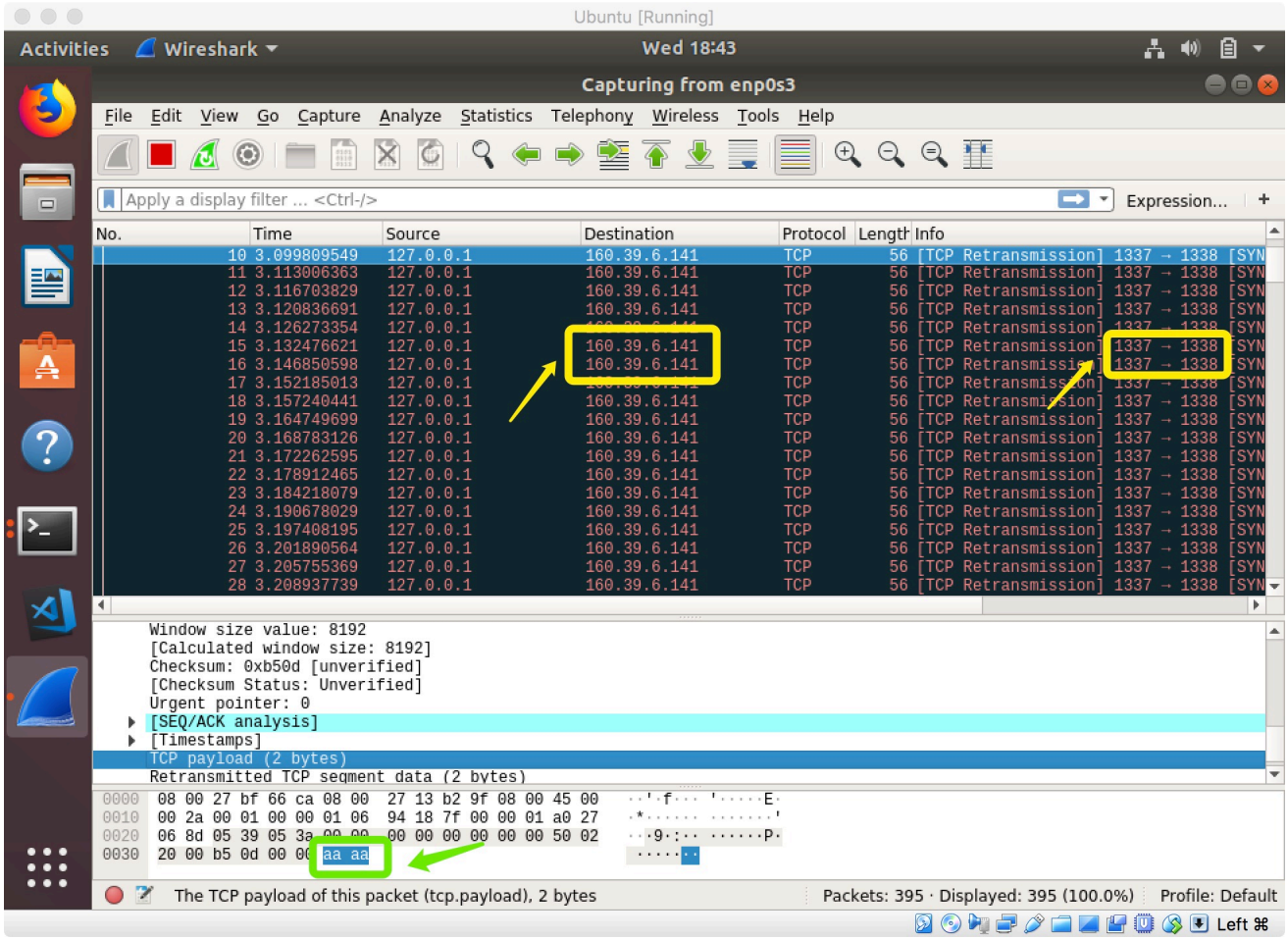
We performed a series of tests on the fuzzer. First, we tested the general functionality, which included specifying the destination address and port and including a default payload. Second, we tested the IP, TCP and application layers separately. The specific tests are described below in detail.

The following types of tests passed with no issues or minor issues:

(1) General: specifying the destination address and port. — No issues

Command: `sudo python3 fuzz.py -I -ittl -D 160.39.6.141 -DP 1338`

Result: packets are being sent to the specified destination address and port (see screenshot).



(2) General: including a default payload. — No issues

Command: `sudo python3 fuzz.py -I -ittl -D 160.39.6.141 -DP 1338`

Result: the packets sent include the default payload (see screenshot for 1).

(3) IP layer: fuzzing all fields. — No issues

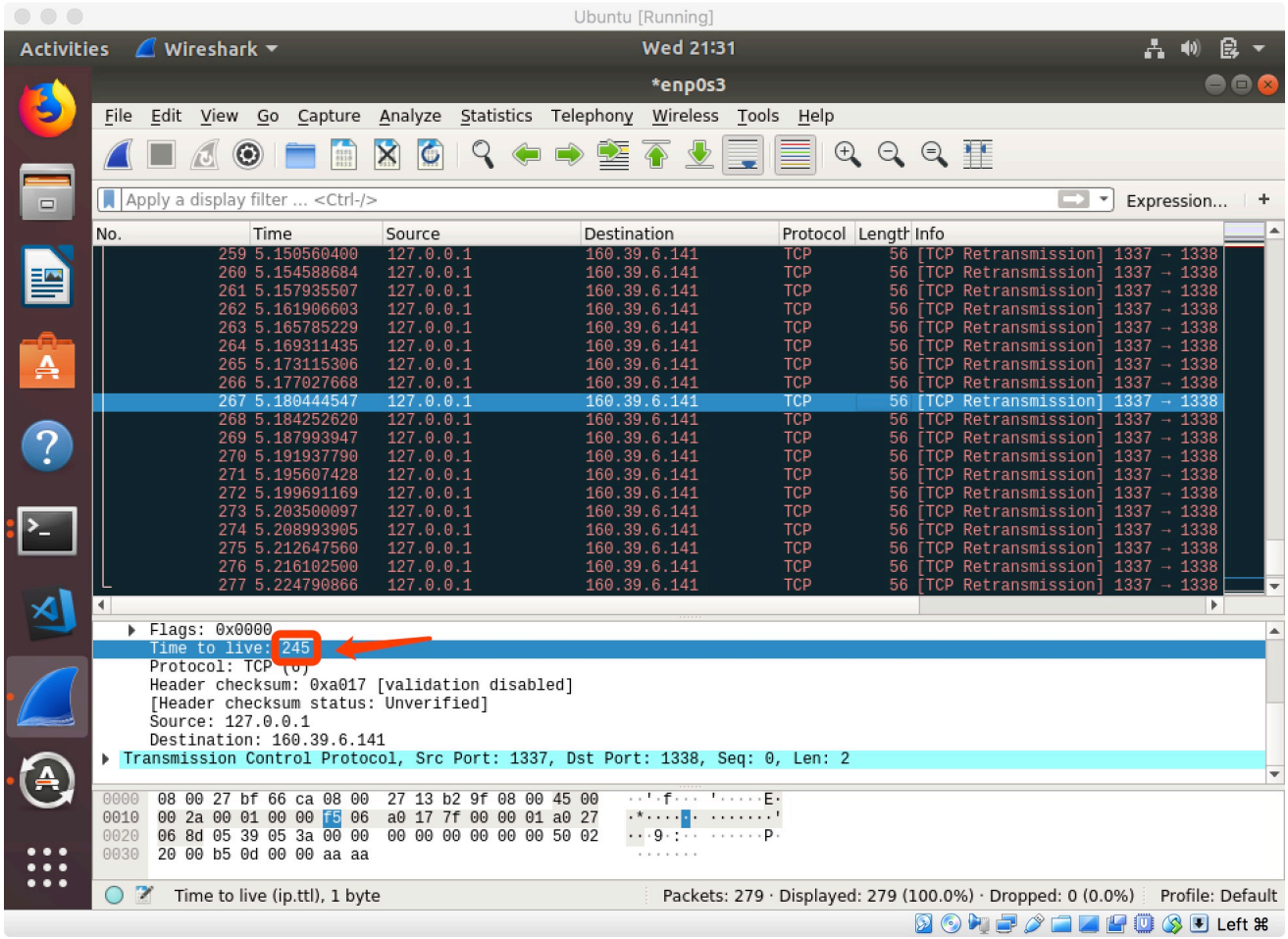
Command: `sudo python3 fuzz.py -I -D 160.39.6.141 -DP 1338`

Result: all IP header fields are fuzzed. Screenshot not included, because the packets cannot be displayed in one screenshot (there are over 200000 packets!).

(4) IP layer: run default set of tests (all values). — No issues

Command: `sudo python3 fuzz.py -I -ittl -D 160.39.6.141 -DP 1338`

Result: This test differs from 3 in that we are only testing a subset of fields, but we want to make sure all possible values are fuzzed. In this example we are performing a test on the `ttl` field. We can see that the `ttl` field is increasing and every possible value is tested.



(5) IP layer: run custom set of tests. — No issues

Command: `sudo python3 fuzz.py -I -ifile ip.csv -D 160.39.6.141 -DP 1338`

Result: the packets specified in `ip.csv` are sent to the destination address and port.

(6) TCP layer: fuzzing all fields. — No issues

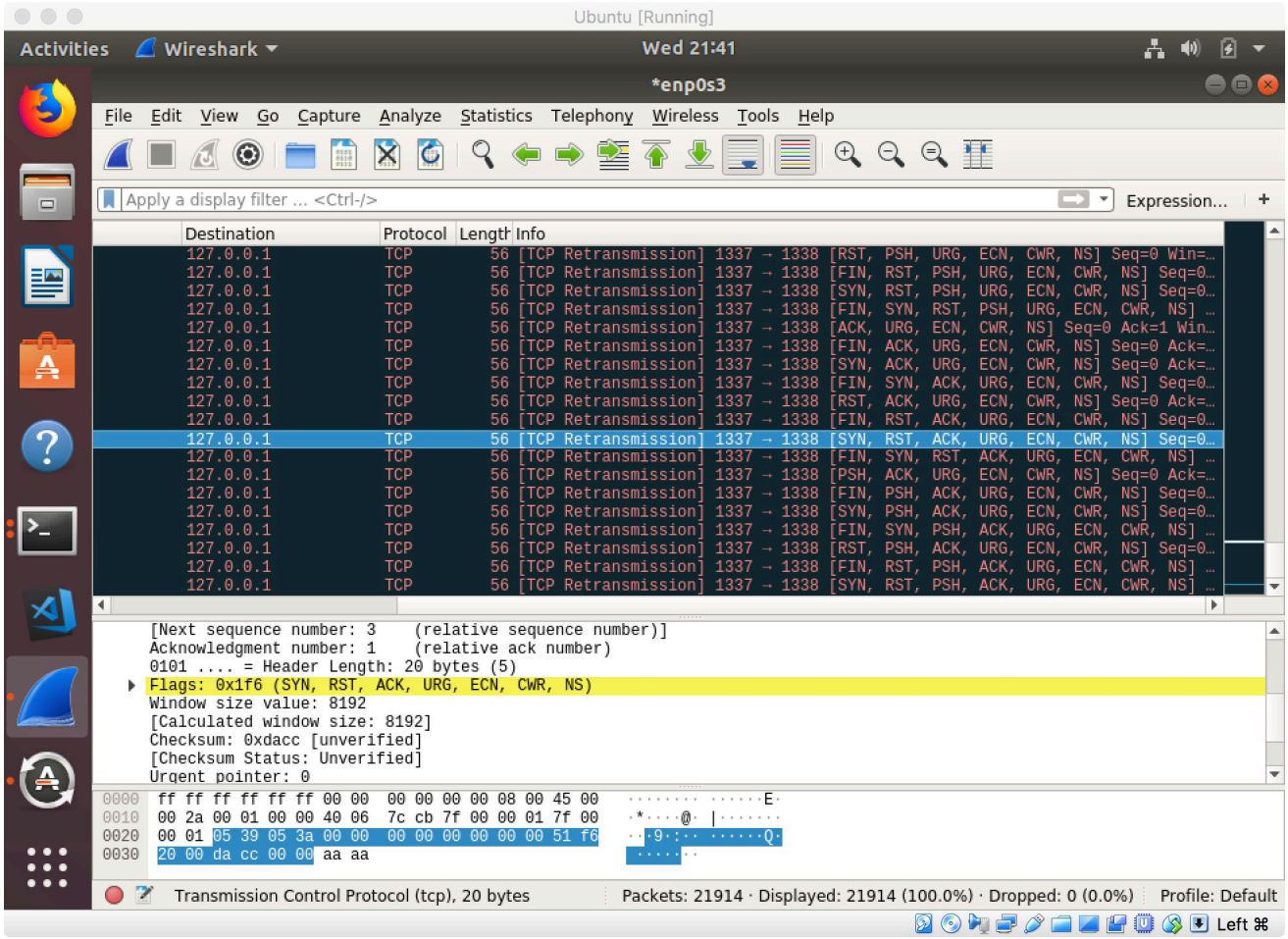
Command: `sudo python3 fuzz.py -T -tall`

Result: all TCP header fields are fuzzed. Screenshot not included, because the packets cannot be displayed in one screenshot (there are over 200000 packets!).

(7) TCP layer: run default set of tests (all values).

Command: `sudo python3 fuzz.py -T -tflags`

Result: This test differs from 6 in that we are only testing a subset of fields, but we want to make sure all possible values are fuzzed. In this example we are performing a test on the `flags` field. We can see that every combination of TCP flags are being tested.



(8) TCP layer: run custom set of tests.

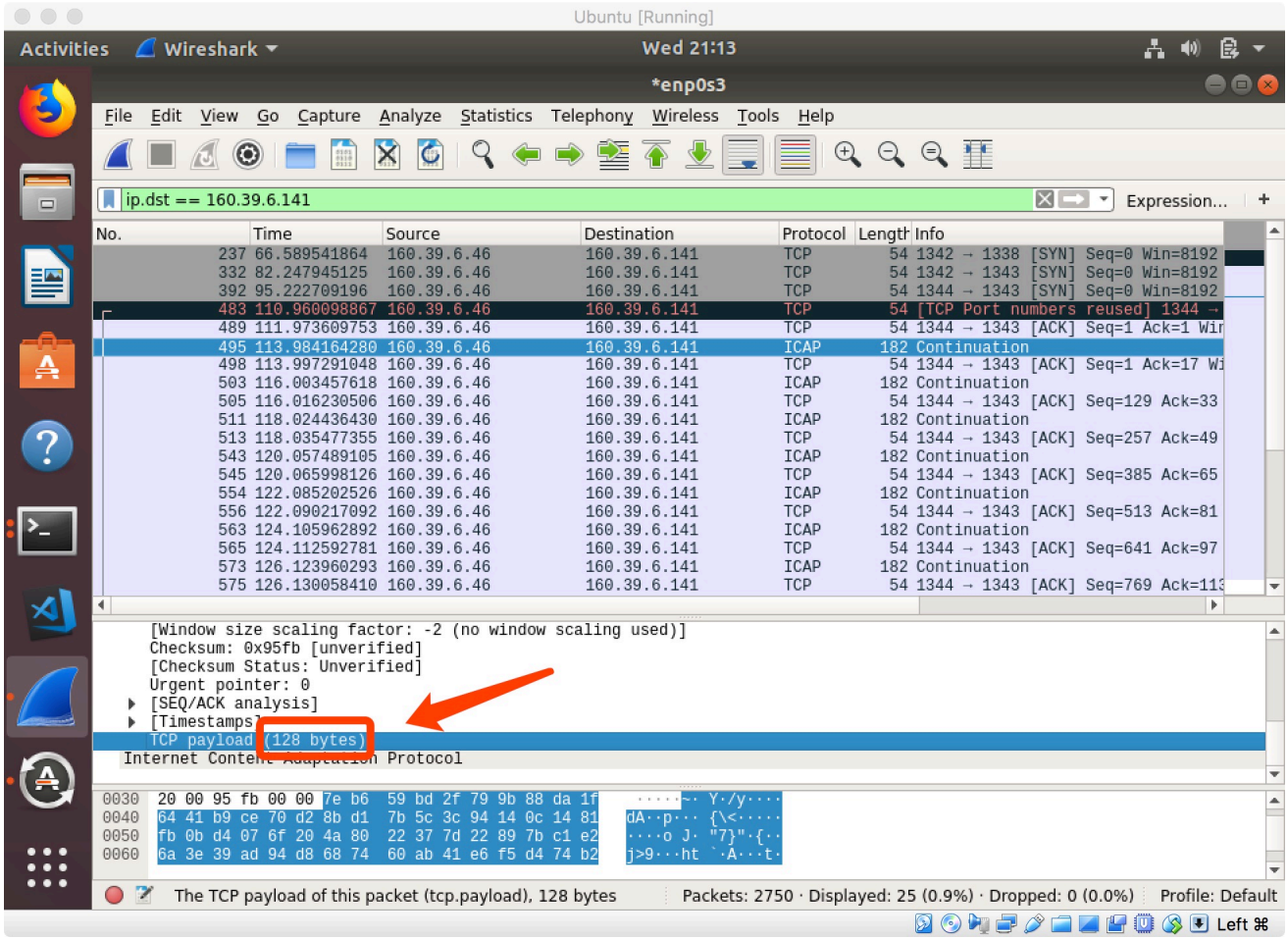
Command: `sudo python3 fuzz.py -T -tfile tcp.csv -D 160.39.6.141 -DP 1338`

Result: the packets specified in `tcp.csv` are sent to the destination address and port.

(9) Application layer: run default set of tests. — Minor issues

Command: `sudo python3 fuzz.py -A -amin 4 -amax 16 -N 10 -D 160.39.6.141 -DP 1343 -S 160.39.6.46 -SP 1344`

Result: This is supposed to generate 10 random packets with payload lengths between 4 to 16 bytes. However, we found that the packets generated all have 128 bytes for payloads. This can be seen in the screenshot below. So we think the command line arguments for `amin` and `amax` are not working, and we are **deducting 1 point**.



(10) Application layer: receive and process server's response. — No issues

Command: `sudo python3 fuzz.py -A -amin 4 -amax 16 -N 10 -D 160.39.6.141 -DP 1343 -S 160.39.6.46 -SP 1344`

Result:

```
sending packets to the server...
Connected to 160.39.6.141
finished sending
total count: 10
valid count: 0
invalid count: 10
```

The following test failed:

(1) Application layer: run custom set of tests. — Failed

Command: `sudo python3 fuzz.py -A -afile custom_payloads -D 160.39.6.141 -DP 1344 -S 160.39.6.46 -SP 1343`

Test file:

```
0102030405
aabbcc
0a0b0c0d
```

Result:

```
"0102030405
" cannot be parsed as hex
continue parsing the remaining of the file ./custom_payloads ...
"aabbcc
" cannot be parsed as hex
continue parsing the remaining of the file ./custom_payloads ...
"0a0b0c0d
" cannot be parsed as hex
continue parsing the remaining of the file ./custom_payloads ...
```

We are **deducting 3 points** for this issue.

This fuzzer was able to pass most of the tests we performed, but 4 points were deducted in total for issues with application layer fuzzing.

Server Functionality

When testing the server, we are using the application layer fuzzer to send packets, since the server was not required to work with IP and TCP layer fuzzing. We performed the following four tests:

(1) Read hex pattern from file. There is no output that suggests the server is reading from a file called `pattern.txt` in the same directory. However we do see it in the code and error checking

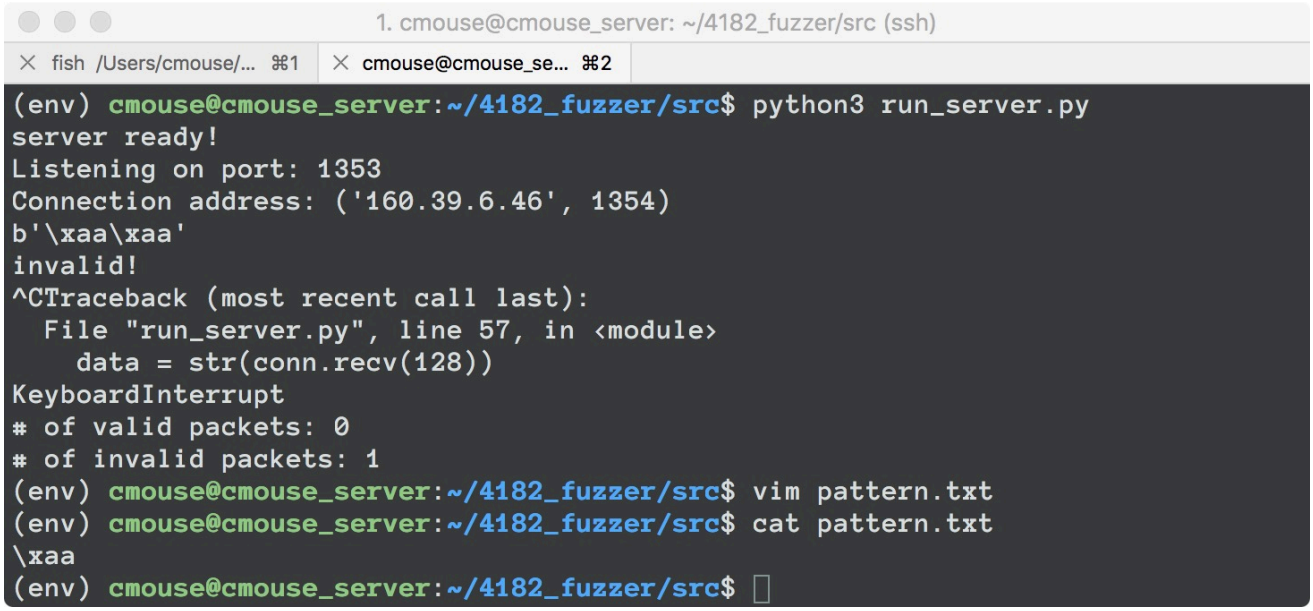
```
try:
    pattern = open("pattern.txt", 'r')
    pattern = pattern.read()
except IOError:
    print("no pattern.txt found. falling back to \x00.")
    pattern = "\x00"
```

```
$ python3 run_server.py
no pattern.txt found. falling back to .
server ready!
Listening on port: 1338
```

(2) Match with correct patterns. — Failed

Command: `sudo python3 fuzz.py -A -afile default_payload_old -D 160.39.6.141 -DP 1353 -S 160.39.6.46 -SP 1354`

Result: the server marks the payload as invalid, while it should have been valid, so we are **deducting 2 points**.



(3) Not match with incorrect patterns. — No issues

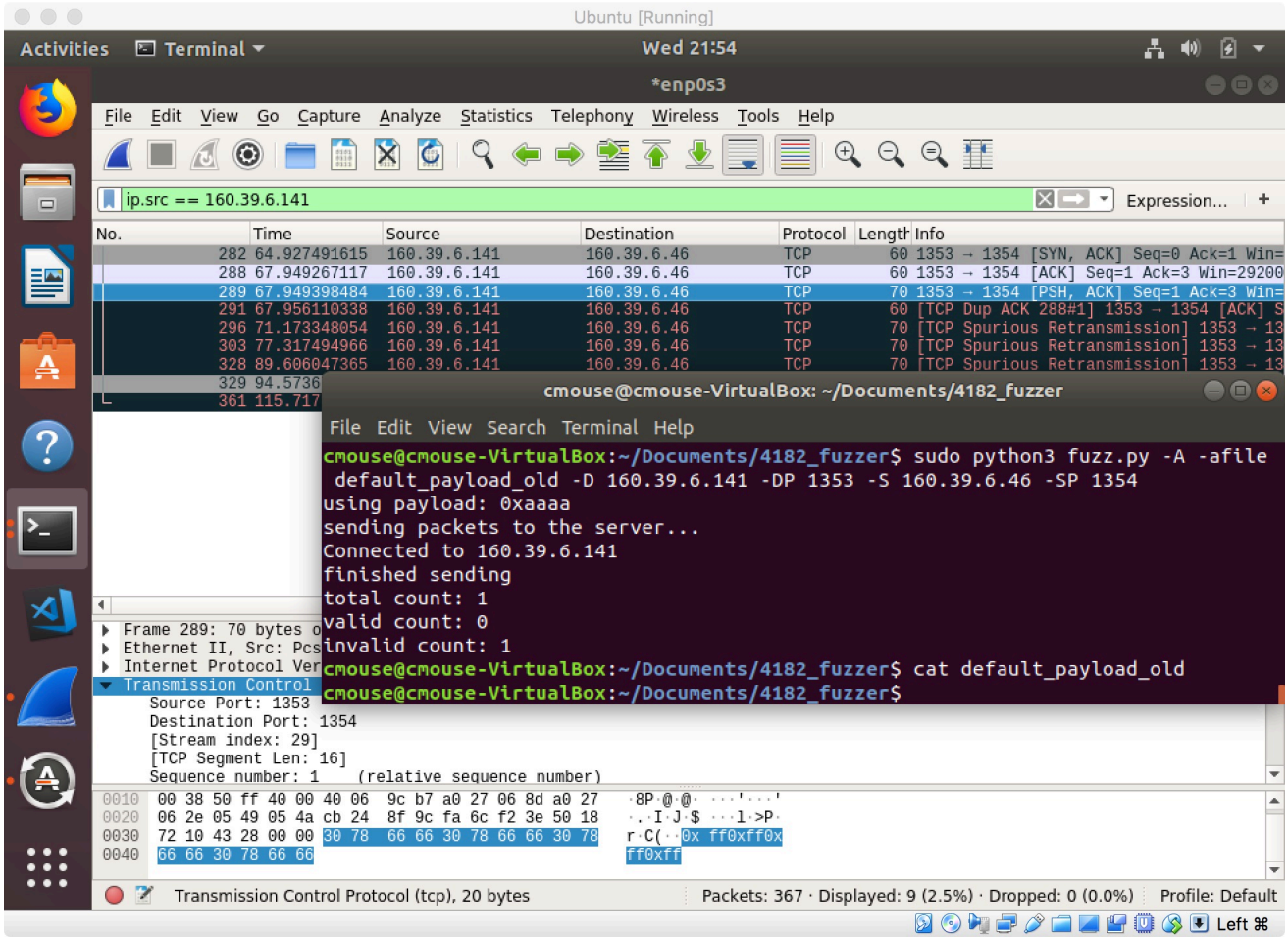
Command: `sudo python3 fuzz.py -A -afile default_payload -D 160.39.6.141 -DP 1353 -S 160.39.6.46 -SP 1354`

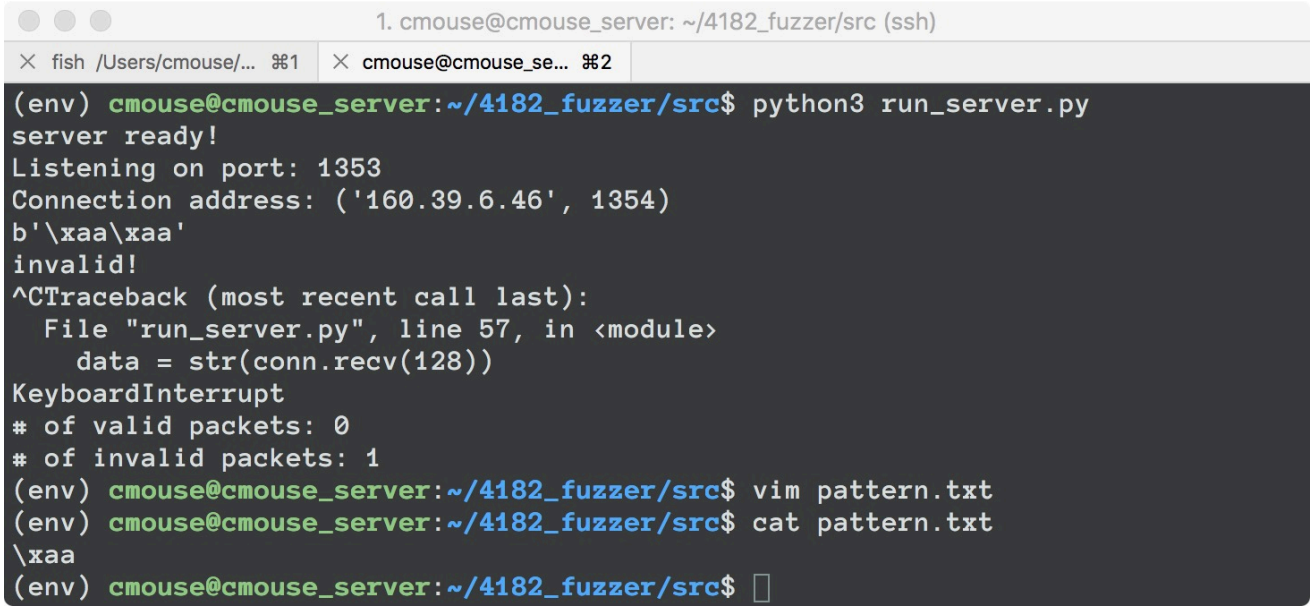
Result: the server marks the payload as invalid.

(4) Send appropriate response to client. — Failed

Command: `sudo python3 fuzz.py -A -afile default_payload_old -D 160.39.6.141 -DP 1353 -S 160.39.6.46 -SP 1354`

Result: As we can see from the screenshots, `default_payload_old` contains the same pattern as `src/pattern.txt`. However, the server sends `0xff0xff0xff0xff` to the client. This is the same issue as 3, so we will not deduct points twice.





Part 4: Error Handling

The following stress tests passed with no issues

Using non hex characters in the payload

```
/tmp/log.txt
"aaaa!
" cannot be parsed as hex`
```

Running the fuzzer with invalid arguments

```
$ sudo python3 fuzz.py --garbage
usage: fuzz.py [-h] [-S SRC] [-D DST] [-SP SP] [-DP DP] [-IF IFILE_NAME]
               [-TF TFILE_NAME] [-AF AFILE_NAME] [-PF PAYLOAD_FILE] [-I] [-T]
               [-A] [-tA] [-iA] [-N N] [-v V] [-amin AMIN] [-amax AMAX]
               [-L LEN] [-Z APP_LOG_FILE] [-tseq] [-tack] [-tdataofs]
               [-treserved] [-tflags] [-twindow] [-tchksum] [-turgptr]
               [-toptions] [-ilen] [-iproto] [-iihl] [-iflags] [-ifrag]
               [-ittl] [-itos] [-iid] [-ichksum] [-iversion]
fuzz.py: error: unrecognized arguments: --garbage
```

Running the fuzzer with no arguments

```
$ sudo python3 fuzz.py
usage: fuzz.py [-h] [-S SRC] [-D DST] [-SP SP] [-DP DP] [-IF IFILE_NAME]
               [-TF TFILE_NAME] [-AF AFILE_NAME] [-PF PAYLOAD_FILE] [-I] [-T]
               [-A] [-tA] [-iA] [-N N] [-v V] [-amin AMIN] [-amax AMAX]
               [-L LEN] [-Z APP_LOG_FILE] [-tseq] [-tack] [-tdataofs]
               [-treserved] [-tflags] [-twindow] [-tchksum] [-turgptr]
               [-toptions] [-ilen] [-iproto] [-iihl] [-iflags] [-ifrag]
               [-ittl] [-itos] [-iid] [-ichksum] [-iversion]
```

Running the server with invalid arguments

```
python3 run_server.py -L /tmp/log.txt adkjfladf
server ready!
Listening on port: 1338
```

Passing negative integer arguments into the command line

```
sudo python3 fuzz.py -amax -1
All possible integer optional arguments has to be positive.
```

Running the server without sudo

```
$ python3 run_server.py -L /tmp/log.txt adkjfladf
server ready!
Listening on port: 1338
```

Printing the entire usage message properly when required

```
python3 fuzz.py -h
usage: fuzz.py [-h] [-S SRC] [-D DST] [-SP SP] [-DP DP] [-IF IFILE_NAME]
               [-TF TFILE_NAME] [-AF AFILE_NAME] [-PF PAYLOAD_FILE] [-I] [-T]
               [-A] [-tA] [-iA] [-N N] [-v V] [-amin AMIN] [-amax AMAX]
               [-L LEN] [-Z APP_LOG_FILE] [-tseq] [-tack] [-tdataofs]
               [-treserved] [-tflags] [-twindow] [-tchksum] [-turgptr]
               [-toptions] [-ilen] [-iproto] [-iihl] [-iflags] [-ifrag]
               [-ittl] [-itos] [-iid] [-ichksum] [-iversion]

Fuzzing IP, Transport(TCP), Payloads with scapy.

optional arguments:
  -h, --help            show this help message and exit
  -S SRC, -src SRC      select source ip address
  -D DST, -dst DST      select destination ip address
  -SP SP, -sport SP     select source port
  -DP DP, -dport DP     select destination port
  -IF IFILE_NAME, -ifile IFILE_NAME
                        select file to read the fuzzing data for ip layer
  -TF TFILE_NAME, -tfile TFILE_NAME
                        select file to read the fuzzing data for tcp layer
  -AF AFILE_NAME, -afile AFILE_NAME
                        select file to read the fuzzing data for application
                        layer
  -PF PAYLOAD_FILE, -payloadfile PAYLOAD_FILE
                        select file to read the default payload data
  -I, -ip               run fuzzing for IP layer
  -T, -tcp              run fuzzing for TCP layer
  -A, -app              run fuzzing for application layer
  -tA, -tall            run fuzzing for all fields for TCP layer
  -iA, -iall            run fuzzing for all fields for IP layer
  -N N, -num N          number of tests to run
  -v V, -verbose V      set verbosity level
  -amin AMIN            minimum length for payload
  -amax AMAX            maximum length for payload
  -L LEN, -len LEN      length of the payload
  -Z APP_LOG_FILE, -log-app-payload APP_LOG_FILE
                        store app layer payload of each app layer packet
                        fuzzed in a file
```

```
  -tseq                 Add seq to TCP fields for fuzzing
  -tack                 Add ack to TCP fields for fuzzing
  -tdataofs             Add dataofs to TCP fields for fuzzing
  -treserved            Add reserved to TCP fields for fuzzing
  -tflags               Add flags to TCP fields for fuzzing
  -twindow              Add window to TCP fields for fuzzing
  -tchksum              Add chksum to TCP fields for fuzzing
  -turgptr              Add urgptr to TCP fields for fuzzing
  -toptions             Add options to TCP fields for fuzzing
  -ilen                 Add len to IP fields for fuzzing
  -iproto               Add proto to IP fields for fuzzing
  -iihl                 Add ihl to IP fields for fuzzing
  -iflags               Add flags to IP fields for fuzzing
  -ifrag                Add frag to IP fields for fuzzing
  -ittl                 Add ttl to IP fields for fuzzing
  -itos                 Add tos to IP fields for fuzzing
  -iid                  Add id to IP fields for fuzzing
  -ichksum              Add chksum to IP fields for fuzzing
  -iversion             Add version to IP fields for fuzzing
```

Specifying an input file for ip layer tests without the ip layer file fuzzing included

```
python3 fuzz.py -IFILE_NAME
usage: fuzz.py [-h] [-S SRC] [-D DST] [-SP SP] [-DP DP] [-IF IFILE_NAME]
               [-TF TFILE_NAME] [-AF AFILE_NAME] [-PF PAYLOAD_FILE] [-I] [-T]
               [-A] [-tA] [-iA] [-N N] [-v V] [-amin AMIN] [-amax AMAX]
               [-L LEN] [-Z APP_LOG_FILE] [-tseq] [-tack] [-tdataofs]
               [-treserved] [-tflags] [-twindow] [-tchksum] [-turgptr]
               [-toptions] [-ilen] [-iproto] [-iihl] [-iflags] [-ifrag]
               [-ittl] [-itos] [-iid] [-ichksum] [-iversion]
fuzz.py: error: argument -I/-ip: ignored explicit argument 'FILE_NAME'
```

Mismatches between layer fuzzing and arguments given

```
python3 fuzz.py -I -TFILE_NAME
usage: fuzz.py [-h] [-S SRC] [-D DST] [-SP SP] [-DP DP] [-IF IFILE_NAME]
               [-TF TFILE_NAME] [-AF AFILE_NAME] [-PF PAYLOAD_FILE] [-I] [-T]
               [-A] [-tA] [-iA] [-N N] [-v V] [-amin AMIN] [-amax AMAX]
               [-L LEN] [-Z APP_LOG_FILE] [-tseq] [-tack] [-tdataofs]
               [-treserved] [-tflags] [-twindow] [-tchksum] [-turgptr]
               [-toptions] [-ilen] [-iproto] [-iihl] [-iflags] [-ifrag]
               [-ittl] [-itos] [-iid] [-ichksum] [-iversion]
fuzz.py: error: argument -T/-tcp: ignored explicit argument 'FILE_NAME'
```

All invalid combinations of ip/tcp/app layer fuzzing and ip/tcp/app layer arguments were tested in this case but are not included for brevity

Running Fuzzer with empty payload file

```
sudo python3 fuzz.py -ip -N 2

 cannot be parsed as hex
```

Running the server with an empty pattern file

```
$ python3 run_server.py
no pattern.txt found. falling back to .
server ready!
Listening on port: 1338
```

The following stress tests passed with minor issues

Exits without printing an error message when a non integer is used as source port

- `python3 fuzz.py -A -S 127.0.0.1 -D 127.0.0.1 -DP 1338 -SP NONSES`
- **This is true about all command line parameter checking including invalid parameters. There are many tests due to the same mistake in error handling that I am not including here (invalid filepaths etc.)** (-3)

Server doesn't throw an error when invalid hex is passed in as the search pattern

```
$ cat pattern.txt
\x15!!!
$ python3 run_server.py -L /tmp/log.txt
server ready!
Listening on port: 1338
```

(-1.5)

Fuzzer Can't read valid hex in a file

```
sudo python3 fuzz.py -A -AF ./default_payload -S 127.0.0.1 -D 127.0.0.1 -DP 1338
-SP 1337 -Z /tmp/log.txt
/tmp/log.txt
"000
" cannot be parsed as hex
```

(-1.5)

Running the fuzzer without sudo

```
$ python3 fuzz.py -A -AF ./default_payload -S 127.0.0.1 -D 127.0.0.1 -DP 1338 -SP
1337 -Z /tmp/log.txt
```

(-1.5)

The following stress tests passed with major issues

We found no major issues like core dumps or stacktraces

Part 5: Clarity of the Code/Comments

The code is well organized and well commented.

For the fuzzer, the code for the IP, TCP and application layers are distinctively separated into the files `ip_fuzzer.py`, `tcp_fuzzer.py` and `app_fuzzer.py` and represented by the classes `IPFuzzer`, `TCPFuzzer` and `APPFuzzer`. We believe this shows good practice.

Inside each class, different functionalities are also separated into different methods, such as `_get_payload`, `_fuzz_from_file` and `_fuzz_by_fields`. As we can see, the names of the methods are very representative and they make the code very readable.

Ample comments are provided and it is easy to understand what the code is doing. Here is an example from `src/ip_fuzzer.py`:

```
try:
    f = open(self._payload_addr, "r")
    payloads = f.readlines()
    if len(payloads) < 1:
        f.close()
        raise IOError

    # try parsing the first line as hex
    try:
        payload = bytes.fromhex(payloads[0]) # only reads the first line of the
file
        print("using payload: 0x" + payloads[0])
    except ValueError:
        # if cannot be parsed as hex, raise exception
        print("%s cannot be parsed as hex" % (payloads[0]))
        raise FieldAttributeException
except IOError:
        #file cannot be read. Create with 0x00
        ...
```

Server code is relatively short and it is also easy to understand. We do recommend moving `fuzz.py` to `src/` for better organization. Also `src/fuzzer-main.py` and `src/server2.py` do not seem to be needed. In general, we did not find any reasons to deduct points for this part, so we are assigning a 10.

Part 6: Code Modification

We made the following modifications to the code to add the logging functionality:

Original fuzzer's `fuzz.py`:

```
if args.A:
    appfuzz = APPFuzzer(source=args.src, dest=args.dst, sport=sp, dport=dp,
verbose=v)
    if args.Afile_name:
        appfuzz.fuzz(test=N, size=len, file=args.Afile_name, min_len=amin,
max_len=amax)
    else:
        appfuzz.fuzz(test=N, size=len, min_len=amin, max_len=amax)
```

Modified fuzzer's `fuzz.py`:

```
if args.A:
    appfuzz = APPFuzzer(source=args.src, dest=args.dst, sport=sp, dport=dp,
verbose=v,app_log_file=args.app_log_file)
    if args.Afile_name:
        appfuzz.fuzz(test=N, size=len, file=args.Afile_name, min_len=amin,
max_len=amax)
    else:
        appfuzz.fuzz(test=N, size=len, min_len=amin, max_len=amax)
```

We also added an argument to the argument parser:

```
parser.add_argument('-Z', "-log-app-payload", action='store', default=None,
help='store app layer payload of each app layer packet fuzzed in a file',
dest='app_log_file')
```

Original fuzzer's `src/app_fuzzer.py`:

```
def __init__(self,source="127.0.0.1", dest="127.0.0.1", sport=1337, dport=1338,
verbose=0):
    self._pckt = Ether()/IP(dst=dest,src=source)/TCP()
    self.client = Client(source,dest,sport,dport,verbose=verbose)
    self.verbose = verbose
```

```
print("sending packets to the server...")
self.client.connect()
for pckt in pckts:
    try:
        self.client.send(pckt)
        time.sleep(2)
    except OSError as err:
        print(err)
```

Modified fuzzer's `src/app_fuzzer.py`:

```
def __init__(self,source="127.0.0.1", dest="127.0.0.1", sport=1337, dport=1338,
verbose=0, app_log_file=None):
        print(app_log_file)
    self._pckt = Ether()/IP(dst=dest,src=source)/TCP()
    self.client = Client(source,dest,sport,dport,verbose=verbose)
    self.verbose = verbose
    self.app_log_file = app_log_file
```

```
print("Log file is: {}".format(self.app_log_file))
print("sending packets to the server...")
self.client.connect()
for pckt in pckts:
    try:
        self.client.send(pckt)
        if self.app_log_file is not None:
            with open(self.app_log_file, 'a') as log:
                log.write(bytearray(pckt).hex()+ "\n")
        time.sleep(2)
    except OSError as err:
        print(err)
```

Original server's `run_server.py`:

```
valid = 0
invalid = 0
```

```
# max data length = 128 bytes
data = str(conn.recv(128))

#server doesn't accept empty payload
...

#send valid/invalid
...
```

Modified server's `run_server.py`:

```
valid = 0
invalid = 0

log_file = None
if len(sys.argv) > 2 and sys.argv[1] == '-L':
    log_file = sys.argv[2]
```

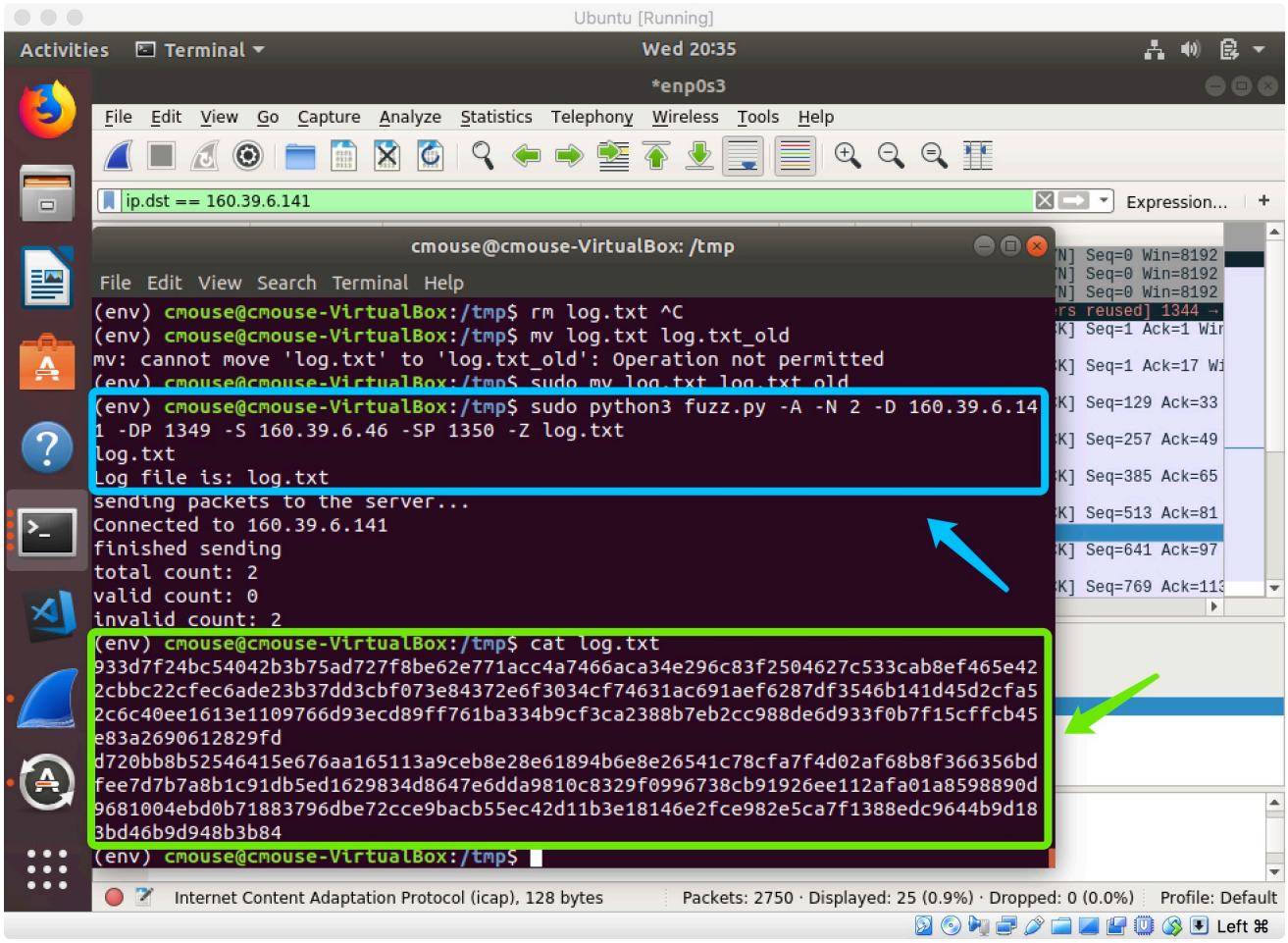
```
# max data length = 128 bytes
rawdata = conn.recv(128)
data = str(rawdata)

#server doesn't accept empty payload
...

#send valid/invalid
...

if log_file is not None:
    with open(log_file, "a") as f:
        f.write(bytearray(rawdata).hex() + "\n")
```

Then we tested the logging functionality:

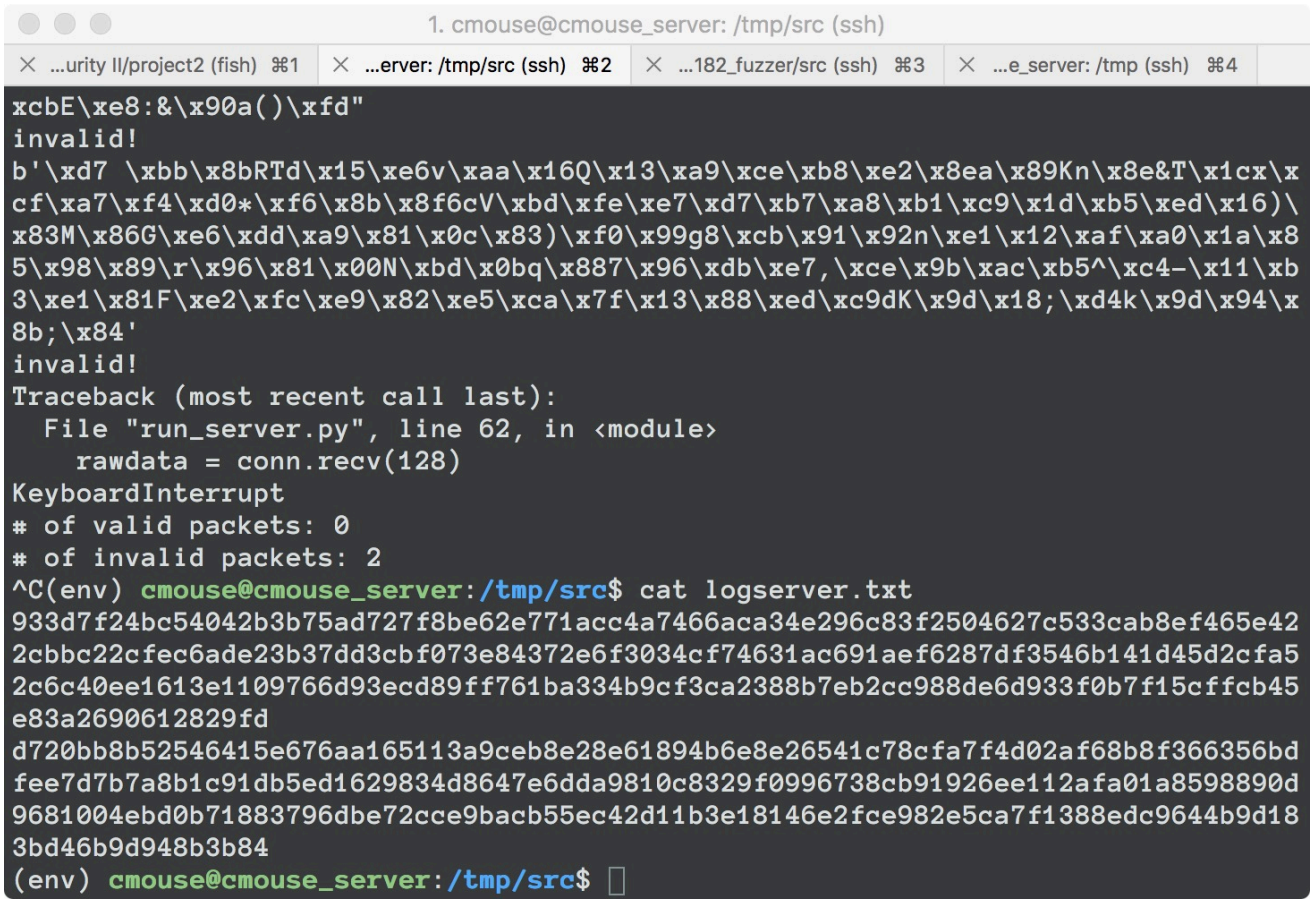
Fuzzer command: `sudo python3 fuzz.py -A -N 2 -D 160.39.6.141 -DP 1349 -S 160.39.6.46 -SP 1350 -Z log.txt`

Server command: `sudo python3 run_server.py -L logserver.txt`

Fuzzer results:



Server results:



As we can see, both the server and the fuzzer logs the packets received/sent in the specified log file. Thus the logging functionality is working correctly!