Automatic Network Generator Interface

Michael Seth Heinzman

April 29, 2023

GitHub Repository https://github.com/MichaelHeinzman/Automatic-NPG-Interface

Abstract

This project aims to create an Automatic Network Packet Generator Interface that allows a user to create and send network packets on a given network. The interface will enable users to create network packets (TCP, ICMP, UDP, ARP, and DNS) using a combination of Scapy, PyQt, and PyTest libraries in the Python programming language. The project is composed of three sections, the Network package which includes Scapy functionality for creating and sending packets, the PyQt User Interface section which handles the displaying of the interface to the user, and the Testing section where one can see how to test the Network package.

1 Introduction

The increase in network technology has created a dynamic area where testing and analysis are a must. Network administrators, security professionals, and researchers need to be able to test various scenarios to ensure that a network is functioning to the expected level and that the data on a network is secure. One way to test the network's functionality, security, and performance is to generate and send custom packets to the network. This approach allows users the ability to test different situations and analyze the network's responses to different traffic types. In this project, I aimed to develop an Automatic Network Packet Generator Interface that takes advantage of different Python libraries to send different network packets (ICMP, TCP, UDP, ARP, DNS). The interface will allow users to easily generate and send custom packets to test a network's functionality, security, and performance. I will use the Python programming language and the Scapy, PyQt, and PyTest libraries to achieve this goal. Scapy is a powerful tool that allows for packet manipulation to send, sniff, dissect, and forge network packets. PyTest is a popular testing framework that allows one to write test cases and automate the testing process in a Python project. PyQt allows one to create a cross-platform graphical user interface. The Automatic Network Packet Generator Interface will provide an intuitive way to select, customize, and send packets across the network according to the user's requirements. The Scapy library will be combined with the PyQt graphical interface to accomplish this. PyTest will be used to automate testing through GitHub actions. Overall, the Automatic Network Packet Generator Interface is a tool that will be used for network testing and analysis. It will allow users to generate and send custom network packets, which will be used to test a network's functionality, security, and performance. This interface will be user-friendly and allow for easy manipulation of packets.

2 Network Package with Scapy

The first step in creating an Automatic Network Generator tool was to develop a package that included code to generate, send, create, and export packets with Scapy and Python. The package was named Network in my project and it includes seven Python files: `packet_generator`, `packet_creator`, `packet_sending`, `packets`, `tcp_handshake`, `packet_info`, and `export`. Each one has a specific use case which will be explained thoroughly below.

2.1 Packet Generator

The `packet_generator.py` file contains one function which can be called by the user interface. It handles the creation and sending of packets. This function is called `generate_packets`, it has an input and an output. The function input is packet_info or a dictionary of the necessary information about a packet that will allow the packet to be created and sent with no errors. This packet_info dictionary will be explained more thoroughly in the packet_info.py file explanation. The output of the generate_packets function is the output of the send_packet function: answered, unanswered, and sent packets. This is so the PyQt user interface can display all of the information that went on during the process of sending and receiving the packets. The code for the packet_generator.py file is below.

```python

__all__ = ['generate_packets']

from .packet_creator import create_packet
from .packet_sending import send_packet
from .error_handling import handle_error

# Handles creating and sending packets.
# Packet Generation based on packet info.
@handle_error
def generate_packets (packet_info):
    new_packet = create_packet(packet_info)
    return send_packet(new_packet, packet_info)
```

2.2 Packet Information

I created a file to represent the different packet information that one would need to use the functions that include packet_info as their input. This file contains examples of ARP, TCP, UDP, ICMP, and DNS packets in dictionary form. They directly correlate with the values you need to pass to Scapy to create the packets. It's important to note that information can be left out for different packet types since Scapy handles some values automatically. Here are the packet information dictionaries that are passed to generate_packets and other functions. It's important to note I replaced all ip addresses with 127.0.0.1 for privacy, but for testing, I inserted my own network addresses.

```python

TCP_IP_PACKET = {
    "type": "IP",
    "version": 4,
    "ihl": 5,
    "tos": 0,
    "len": 64,
    "id": 54321,
    "flags": 0,
    "frag": 0,
    "ttl": 64,
    "proto": 6,
    "chksum": None,
    "srcIP": "127.0.0.1",
    "dstIP": "127.0.0.1",
    "options": None,
    "payload": "This is a TCP packet",
    "number": 1,
    "sport": 0,
    "dport": 80,
    "tcp_type": "S"
}

UDP_IP_PACKET ={
    "type": "IP",
    "version": 4,
    "ihl": 5,
    "tos": 0,
    "len": 64,
    "id": 12345,
    "flags": 0,
    "frag": 0,
    "ttl": 64,
    "proto": 17,
    "chksum": None,
    "srcIP": "127.0.0.1",
    "dstIP": "127.0.0.1",
    "options": None,
    "sport": 0,
    "dport": 80,
    "payload": "This␣is␣a␣UDP␣packet",
    "number": 2
}

ICMP_IP_PACKET = {
    "type": "IP",
    "version": 4,
    "ihl": 5,
    "tos": 0,
    "len": 84,
    "id": 54321,
    "flags": 0,
    "frag": 0,
    "ttl": 64,
    "proto": 1,
    "chksum": None,
    "srcIP": "127.0.0.1",
    "dstIP": "127.0.0.1",
    "options": None,
    "sport": 0,
    "dport": 80,
    "payload": "This␣is␣an␣ICMP␣packet.",
    "number": 1
}

ARP_PACKET_INFO = {
    "type": "ARP",
    "hwtype": 1,
    "ptype": 2048,
    "hwlen": 6,
    "plen": 4,
    "op": 1,
    "hwsrc": "",
    "psrc": "127.0.0.1",
    "hwdst": "ff:ff:ff:ff:ff:ff",
    "pdst": "127.0.0.1",
    "number": 10
}

DNS_PACKET_INFO = {
    "srcIP": "127.0.0.1",
    "dstIP": "",
    "type": "DNS",
    "id": 0xAAA,
    "qr": 0,
    "opcode": 0,
    "aa": 0,
    "tc": 0,
    "rd": 1,
    "ra": 0,
    "z": 0,
    "ad": 0,
    "cd": 0,
    "rcode": 0,
    "qdcount": 1,
    "ancount": 0,
    "nscount": 0,
    "arcount": 0,
    "qd": [],
    "an": [],
    "ns": [],
    "ar": [],
    "qname": "www.example.com",
    "number": 0,
}
```

Each packet information dictionary includes a type and number field. The type field indicates the type of packet that the packet creator reads to understand what type of packet it needs to create. The number field indicates the number of packets a user wants to send that have this specific information. The rest of the fields indicate different Scapy parameters that can be found in their documentation.

2.3 Packet Creator

After the `generate_packets` function is called, it begins the packet creation process by calling the `create_packet` function. This function is located inside the file `packet_creator.py` which is designated for receiving packet information and choosing the right process to create the given packet from the information. It receives the packet information as input and checks the packet type to see what packet function should be called. For now, the three main packet functions that can be called are ARP, IP, and DNS. If the packet information fails to match the type to a packet creation function, then it will throw an error and return None. The code is below.

```python

__all__ = ['create_packet']

from .packets import create_ARP, create_IP,create_DNS
from .error_handling import handle_error

# Creates a packet based on type and data.
@handle_error
def create_packet(packet_info):
    switcher = {
        "ARP": create_ARP,
        "IP": create_IP,
        "DNS": create_DNS,
    }

    type = packet_info.get("type", "")
    generate_packet = switcher.get(type)

    if not generate_packet:
        raise ValueError(f"Unsupported packet type: {type}")

    new_packet = generate_packet(packet_info) or None

    return new_packet
```

I specifically made the function able to handle new packet types with little code changes, at most one would have to add another mapping to a create function in the switcher dictionary. This allows for the packet creation function to handle multiple types of packets without needing to change code and handle the logic for each specific packet type.

2.4 Packets

This file contains multiple functions that handle the creation of each packet with Scapy. These functions are the ones shown in the `packet_creator.py` file: `create_ARP`,`create_IP`, and `create_DNS`. Each function takes in a packet information dictionary where the necessary values are retrieved and used along with the Scapy protocols to create and return a packet. The code for the `packets.py` file is below.

```python

__all__ = ['create_ARP', 'create_IP', 'create_DNS']

from scapy.all import  get_if_hwaddr, get_if_addr, get_working_if, Raw,
    RandShort
from scapy.layers.inet import IP,ICMP, TCP, UDP
from scapy.layers.l2 import *
from scapy.layers.dns import DNS, DNSQR
from .error_handling import handle_error


# Creates an ARP response and request packet.
@handle_error
def create_ARP(packet_info):
    # All fields can be added to IP, I removed them since I couldn't test them
         all.
    pdst = packet_info.get('pdst', '192.168.1.1')
    hwdst = packet_info.get('hwdst', 'ff:ff:ff:ff:ff:ff')
    srcIP = packet_info.get('srcIP', get_if_addr(get_working_if()))
    hwsrc = packet_info.get('hwsrc', get_if_hwaddr(get_working_if()))

    # Wraps the created ARP packet in an Ether packet.
    arp_packet = Ether(dst=hwdst)/ARP(op=1,pdst=pdst, psrc=srcIP, hwsrc=hwsrc)
    return arp_packet

@handle_error
def create_IP(packet_info):
    # Gets the necessary information from packet_info.
    # All fields can be added to IP, I removed them since I couldn't test them
         all.
    src_IP = packet_info.get("srcIP", get_if_addr(get_working_if()))
    dst_IP = packet_info.get("dstIP")
    ttl = packet_info.get("ttl", 64)
    payload = packet_info.get("payload", '')
    options = packet_info.get("options")
    if options is None:
        options = {}

    # create the IP packet object
    ip_packet = IP(src=src_IP, dst=dst_IP, ttl=ttl)

    # check the value of "proto" and add the appropriate layer
    proto = packet_info.get("proto", 6)
    if proto == 1:
        ip_packet /= ICMP()
    elif proto == 6:
        ip_packet /= TCP(sport=packet_info.get("sport", 0), dport=packet_info.
            get("dport", 80),flags=packet_info.get("tcp_type", "S"),seq=
            packet_info.get("seq", None), ack=packet_info.get("ack", None))/
            Raw(load=payload)
    elif proto == 17:
        ip_packet /= UDP(sport=packet_info.get("sport", 12345), dport=
            packet_info.get("dport", 54321))/Raw(load=payload)

    # calculate the checksum
    ip_packet.chksum = None
    ip_packet = IP(bytes(ip_packet))

    return ip_packet


# Create a DNS packet.
@handle_error
def create_DNS (packet_info):
    # All fields can be added to IP, I removed them since I couldn't test them
        all.
    qname = packet_info.get("qname", 'example.com')

    dns_packet = IP(dst='8.8.8.8')/UDP(sport=RandShort(),dport=53)/DNS(rd=1,
       qd=DNSQR(qname=qname))

    return dns_packet
```

2.5 Sending Packets

The sending of packets is handled by the `packet_sending.py` file which includes three different functions: `send_packet`, `check_packet_type_assign_send_method`, and `get_default_interface_name`.

The `send_packet` function receives a created packet which was created from the `packets.py` file, it also receives the `packet_info` parameter which is the packet information dictionary given. The decision to include the packet information dictionary in the sending of the packet was to handle future cases where information that isn't included in a created Scapy packet would be needed in sending the packet. For example, a packet created from Scapy does not include how many you want to send. However, the packet information dictionary includes the field 'number' which is designated for this purpose. Therefore, I decided it was necessary to include the packet information dictionary as a parameter for future-proofing rather than just passing the number of packets to be sent. The `send_packet` function's output is a tuple that contains answered, unanswered, and sent packets. This tuple will be used by the PyQt user interface and displayed to the user, it will also be processed into a result that can be used to export the information to a PCAP file. The `send_packet` function also prints out the sent, answered, and unanswered packets to the console where PyQt code captures this output and displays it to the user.

The `check_packet_type_assign_send_method` is called by the send packet function to decide what type of send function from Scapy to use given the packet information and packet type. For example, Scapy has multiple ways to send packets and some network packets use different functions. I thought in order to future-proof the process, a function to return the correct Scapy send method given a packet was due. This function contains a dictionary of Scapy send functions that decide which function to return based on the packet type in the packet information dictionary it receives as a parameter.

The `get_default_inteface_name` function handles the retrieving of the default network interface on the user's system. The network interface is needed to send packets correctly using Scapy, so this function is essential for sending the network packets. It uses the Netifaces library to retrieve a list of network interfaces on the device and filters through them until the IPv4 address '0.0.0.0' is found.

```


__all__ = ['send_packet']

import netifaces
from scapy.all import *
from .error_handling import handle_error

# Sends multiple of the same packet.
@handle_error
def send_packet(packet, packet_info, iface=None):
    if packet is None or packet_info is None:
```

```python
        return None

    number_of_packets = packet_info.get("number", 1)

    if number_of_packets is None:
        return None

    send_method = check_packet_type_assign_send_method(packet, packet_info)
    iface = get_default_interface_name()
    result = send_method(packet * number_of_packets, timeout=10, iface=iface,
        filter=None, verbose=0, chainCC=0, retry=0, multi=0)
    time.sleep(.1)

    # Extract answered and unanswered packets from the result variable
    answered, unanswered = result

    # Print sent packets
    sent_packets = packet * number_of_packets
    print(f"Sent␣{number_of_packets}␣packets:")
    for i in range(number_of_packets):
        print(sent_packets[i].summary())

    # Print answered packets
    print(f"\nReceived␣{len(answered)}␣packets,␣got␣{len(answered)}␣answers:")
    for pkt in answered:
        if isinstance(pkt, tuple):
            # if the packet is a tuple, it contains the packet and the answer
            pkt, ans = pkt
            pkt.time = ans.time
        print(pkt.summary())

    # Print unanswered packets
    if len(unanswered) > 0:
        print(f"\n{len(unanswered)}␣packets␣were␣not␣answered:")
        for pkt in unanswered:
            print(pkt.summary())
    else:
        print("\nAll␣packets␣were␣answered.")

    print("\n")

    # Create a new tuple to include the sent packets
    result = (answered, unanswered, sent_packets)
    return result

@handle_error
def check_packet_type_assign_send_method(packet, packet_info):
    if packet is None or packet_info is None:
        return None

    packet_type = packet_info.get("type")

    send_method_dict = {
        "IP": sr,
        "ARP-who-has": srp,
        "DNS": sr,
    }

    send_method = send_method_dict.get(packet_type, sr)

    return send_method

```

```python
def get_default_interface_name():
    """
    Returns the name of the default network interface on the system.
    """
    # Get the addresses of all available network interfaces
    interfaces = netifaces.interfaces()

    # Find the default interface, which has a 0.0.0.0 IP address
    for iface in interfaces:
        addresses = netifaces.ifaddresses(iface)
        if netifaces.AF_INET in addresses:
            ipv4_addresses = addresses[netifaces.AF_INET]
            for addr in ipv4_addresses:
                if addr['addr'] == '0.0.0.0':
                    return iface

    # If the default interface was not found, return None
    return None
```

2.6 TCP Handshake

The TCP handshake is an important process in sending network packets and testing. Using the code I already created, I developed a function to generate TCP packets and simulate a handshake transaction between two devices. The `tcp_handshake.py` file contains a function called `tcp_handshake` that receives a packet information dictionary as its input. This dictionary is then used to call the `generate_packets` function discussed earlier, where the result is stored in the variable `syn_ack`. This variable is the response from a device after receiving the packet sent by `generate_packets` which represents an SYN-ACK response. This result is then manipulated into an ACK packet and sent to the device that responded. Thus, the simulation of the TCP handshake is completed.

```python
__all__ = ['tcp_handshake']
from network.packet_generator import generate_packets
from scapy.layers.inet import TCP,IP
from .error_handling import handle_error

@handle_error
def tcp_handshake(packet):
    print("Starting␣TCP␣Handshake␣\n")
    # SYN packet
    if packet["tcp_type"] == "S":
        # SYN ACK Response (Has multiple packets if sent multiple of the same
           packet.)
        syn_ack = generate_packets(packet)
        if not syn_ack[0]:
            return syn_ack

        # SYN ACK Packet.
        syn_ack_packet = syn_ack[0][0]
        ack_packet_info = {
            "type": "IP",
            "proto": 6,
            "srcIP": syn_ack_packet[0][IP].src,
            "dstIP": syn_ack_packet[0][IP].dst,
            "sport": syn_ack_packet[0][TCP].sport,
            "dport": syn_ack_packet[0][TCP].dport,
            "seq": syn_ack_packet[0][TCP].ack,
            "ack": syn_ack_packet[0][TCP].seq + 1,
            "tcp_type": "A"
        }

        print("Sending ACK TCP Packet... \n")

        # Send ACK and receive result if any.
        ack_result = generate_packets(ack_packet_info)

        print("TCP Handshake Finished \n")

        # Return sent, answered packets.
        return (syn_ack[0] + ack_result[0], syn_ack[1] + ack_result[1],
            syn_ack[2] + ack_result[2])
    else: return generate_packets(packet)
```

It's important to note if I had more time I would create a file to handle all handshakes and export the `ack_packet_info` into the `packet_info.py` file so that the creation of the `tcp_handshake` function only handles the handshake and nothing more.

2.7 Export Packets

The `export.py` file contains four different functions designated for exporting packets into a pcap file: `export_to_wireshark`, `create_pcap_file`, `dowload_pcap_file`, and `move_to_selected_folder`. The `export_to_wireshark` receives a list of packets and creates a PCAP file and opens the file in Wireshark, this function uses both the `create_pcap_file` and `move_to_selected_folder` to help with storing and creating the file where the user wants. The creation of a PCAP file is handled by the `create_pcap_file` function that takes in a list of packets and uses the wrpcap function provided by Scapy to create and store the file given a file path. If the user chooses to download the PCAP file option, instead of exporting to the Wireshark option, the `download_pcap_file` function will be called. Where the creation and saving of a PCAP file will occur.

```
__all__ = ['export_to_wireshark','download_pcap_file']
import os
import shutil
import tkinter as tk
from tkinter import filedialog
from scapy.all import wrpcap


# Handles creating and sending of packets. Packet Generation based on packet
    info.
def export_to_wireshark(result):
    file_path = create_pcap_file(result)
    folder = move_to_selected_folder(file_path)

    # Open the file in Wireshark
    os.startfile(os.path.join(folder, os.path.basename(file_path)))


def create_pcap_file(result):
    file_path = os.path.join(os.getcwd(), "combined_results.pcap")

    # Check if file already exists
    if os.path.isfile(file_path):
        # File already exists, prompt user to overwrite or choose a new
            filename
        overwrite = tk.messagebox.askyesno("File already exists", "A file
            named 'combined_results.pcap' already exists in the current
            directory. Do you want to overwrite it?")
        if not overwrite:
            # User chose not to overwrite, prompt user to choose a new
                filename
            file_path = filedialog.asksaveasfilename(initialfile="
                combined_results.pcap", defaultextension=".pcap", filetypes=[(
                "PCAP␣Files", "*.pcap")])
            if not file_path:
                # User canceled file dialog, return None
                return None

    # Write all the packets to the pcap file
    wrpcap(file_path, result)

    return file_path

def download_pcap_file(result):
    # Get the default file name
    default_file_name = "combined_results.pcap"

    # Open the file dialog box for saving the file
    file_path = filedialog.asksaveasfilename(initialfile=default_file_name,
        defaultextension=".pcap", filetypes=[("PCAP␣Files", "*.pcap")])

    # Check if a file name was entered
    if file_path:
        # Write all the packets to the pcap file
        wrpcap(file_path, result)

        # Move the file to the selected folder
        return file_path
    else:
        # User canceled file dialog, return None
        return None

def move_to_selected_folder(file_path):
    # Create a Tkinter window and hide it
    root = tk.Tk()
    root.withdraw()

    # Get the file name from the file path
    file_name = os.path.basename(file_path)

    # Open a dialog box for the user to enter a new file name
    new_file_name = filedialog.asksaveasfilename(initialfile=file_name)

    # Check if a file name was entered
    if new_file_name:
        # Move the file to the selected directory with the new file name
        shutil.move(file_path, new_file_name)
        return os.path.dirname(new_file_name)
    else:
        return None
```

3 Automated Testing with GitHub Actions

To test the network code and ensure that I was getting the correct results while updating the code, I setup a GitHub action that connected to my repository and ran test cases on the functions I mentioned above. There was not enough time to completely automate the process and to develop code to have multiple randomized test examples for the different functions, but I managed to test the main functions using this feature of GitHub. To implement this, I had to include a test.yml file in my GitHub folder in the project directory. This file contained instructions on how to run the test cases which were developed using PyTest. It explains to GitHub that I want to run the tests on every update of the master branch, and it also explains where the test cases were found which was in the tests folder in the project. The yaml file is below.

```yaml
name: Automated Testing on Windows
on:
  push:
    branches: [master]
  pull_request:
    branches: [master]
  workflow_dispatch:

jobs:
  test:
    runs-on: windows-latest

    steps:
      - uses: actions/checkout@v2

      - name: Set up Python 3.8
        uses: actions/setup-python@v2
        with:
          python-version: 3.8

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Run tests
        working-directory: ./app/src/
        run: pytest tests/
```

3.1 Test Case Example

In the tests folder, there are multiple test files that test different files in the Network package that I explained in section two. I had time to create and test with three files, which represent a simple way of testing sending and the creation of packets. These files include `test_packet_creation.py`, `test_packet_creator.py`, and `test_packet_sending.py`. As the names suggest, these packets test the three files in the Network folder or package: `packets.py`, `packet_creator.py`, and `packet_sending.py`.

3.1.1 Test Packet Creation

The `test_packet_creation.py` tests the creation of individual packets such as ARP, IP, and DNS. These tests call the functions designated for creating those packets with example test data and compare the results to the expected results. If they match, then an error is not asserted and the test case passed. I only did enough test cases to show how it would be done, I was unable to test every type of packet expected, which would require much more complex code than time allowed.

```python
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), ".."))

from network.packets import create_ARP, create_IP, create_DNS
from scapy.all import *

def test_create_ARP():
    packet_info = {'pdst': '192.168.1.1', 'hwdst': 'ff:ff:ff:ff:ff:ff'}
    packet = create_ARP(packet_info)
    assert packet.haslayer(Ether)
    assert packet[Ether].dst == packet_info['hwdst']
    assert packet.haslayer(ARP)
    assert packet[ARP].op == 1
    assert packet[ARP].pdst == packet_info['pdst']
```

```


def test_create_IP():
    packet_info = {'srcIP': '192.168.1.100', 'dstIP': '8.8.8.8', 'ttl': 64,
                   'proto': 17, 'payload': 'hello world'}
    packet = create_IP(packet_info)
    assert packet.haslayer(IP)
    assert packet[IP].src == packet_info['srcIP']
    assert packet[IP].dst == packet_info['dstIP']
    assert packet[IP].ttl == packet_info['ttl']
    assert packet.haslayer(UDP)
    assert packet[UDP].sport == 12345
    assert packet[UDP].dport == 54321


def test_create_DNS():
    packet_info = {'qname': 'example.com'}
    packet = create_DNS(packet_info)
    assert packet.haslayer(IP)
    assert packet[IP].dst == '8.8.8.8'
    assert packet.haslayer(UDP)
    assert packet[UDP].sport > 0
    assert packet[UDP].dport == 53
    assert packet.haslayer(DNS)
    assert packet[DNS].rd == 1
    assert packet.haslayer(DNSQR)
    assert packet[DNSQR].qname == (packet_info["qname"] + ".").encode()
```

3.1.2 Test Packet Creator

The `test_packet_creator.py` tested whether or not the correct packet was created based on the given packet information dictionary given. Since that's the `packet_creator.py` function's job. It passes packet information to the `create_packet` function and determines whether the output is indeed the correct packet.

```
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), ".."))

from network.packet_creator import create_packet
from scapy.all import *

def test_create_packet():
    # Test creating an ARP packet
    packet_info = {
        "type": "ARP",
        "hwsrc": "00:11:22:33:44:55",
        "hwdst": "ff:ff:ff:ff:ff:ff",
        "srcIP": "192.168.1.100",
        'dstIP': '192.168.1.1',  # fixed key name
    }
    packet = create_packet(packet_info)
    assert packet.haslayer(ARP)
    assert packet[Ether].dst == packet_info['hwdst']
    assert packet.haslayer(ARP)
    assert packet[ARP].op == 1
    assert packet[ARP].pdst == packet_info['dstIP']  # fixed key name

    # Test creating an IP packet
    packet_info = {
        "type": "IP",
```

```python
        "srcIP": "192.168.1.100",
        "dstIP": "8.8.8.8",
        "ttl": 64,
        "proto": 17,
        "payload": b"hello world",
    }
    packet = create_packet(packet_info)
    assert packet.haslayer(IP)
    assert packet[IP].src == packet_info["srcIP"]
    assert packet[IP].dst == packet_info["dstIP"]
    assert packet[IP].ttl == packet_info["ttl"]
    assert packet.haslayer(UDP)
    assert packet[UDP].sport == 12345
    assert packet[UDP].dport == 54321

    # Test creating a DNS packet
    packet_info = {
        "type": "DNS",
        "qname": "example.com",
    }
    packet = create_packet(packet_info)
    assert packet.haslayer(IP)
    assert packet[IP].dst == "8.8.8.8"
    assert packet.haslayer(UDP)
    assert packet[UDP].sport > 0
    assert packet[UDP].dport == 53
    assert packet.haslayer(DNS)
    assert packet[DNS].rd == 1
    assert packet.haslayer(DNSQR)
    assert packet[DNSQR].qname == (packet_info["qname"] + ".").encode()
```

3.1.3 Test Packet Sending

The `test_packet_sending.py` checks whether packets were sent correctly and the result was received or not. Since testing network output on a GitHub actions service was difficult to implement, I instead opted to check whether or not I handled null cases and whether or not the correct Scapy send function was called. The code is below.

```python
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), ".."))

from unittest.mock import MagicMock, patch
from scapy.layers.l2 import Ether
from scapy.layers.inet import IP
from scapy.layers.dns import DNS, DNSQR
from network.packet_sending import send_packet,
    check_packet_type_assign_send_method

# Test send_packet function with a single packet
def test_send_packet_single():
    packet = Ether() / IP(dst="8.8.8.8") / DNS(rd=1, qd=DNSQR(qname="www.
        google.com"))
    packet_info = {"type": "DNS", "number": 1}
    result = send_packet(packet, packet_info)
    assert result[0] is not None  # Check that there is at least one answered
        packet
    assert result[1] is not None  # Check that there are no unanswered packets
    assert len(result[2]) == 1  # Check that all packets were sent

# Test send_packet function with multiple packets
```

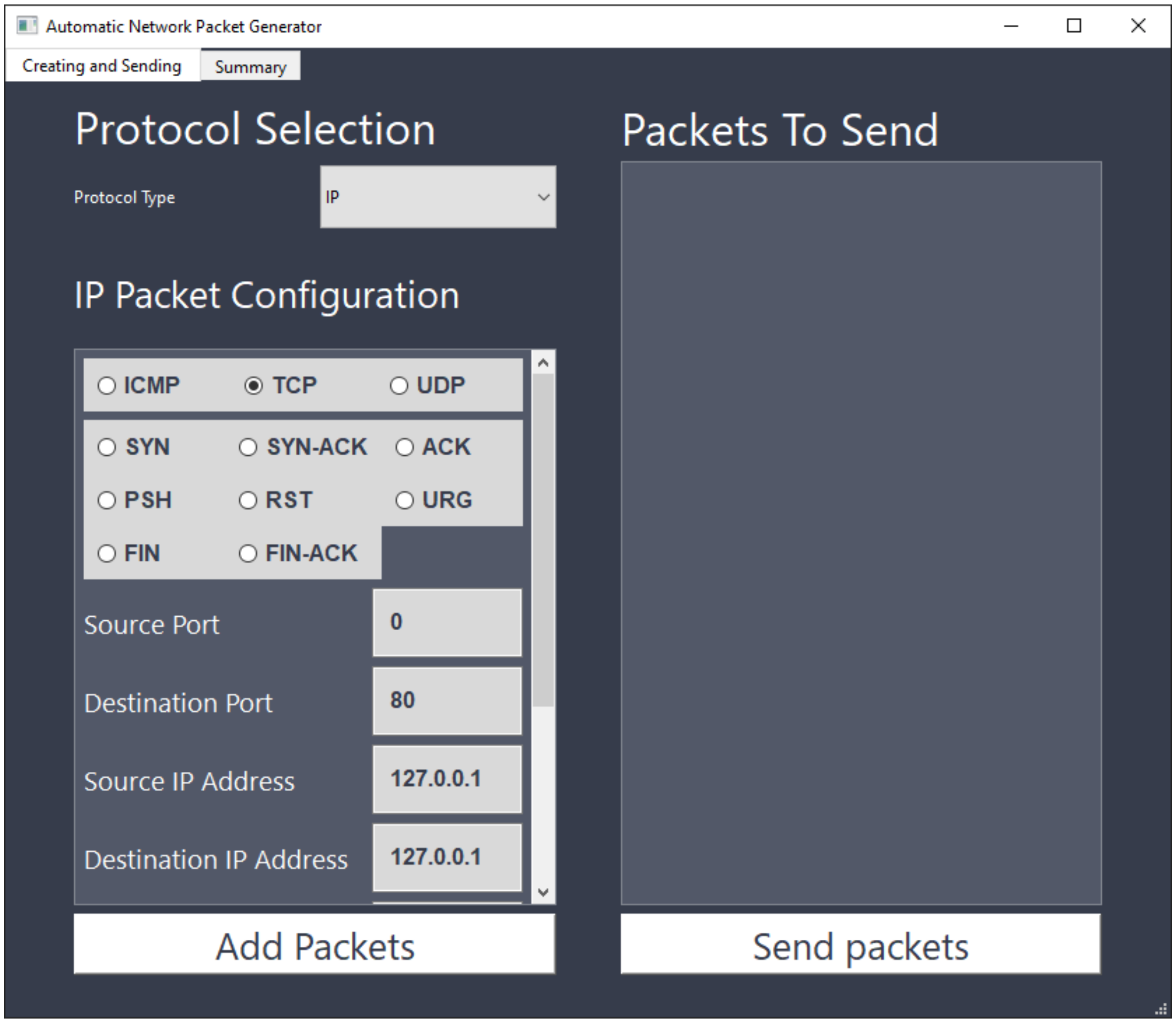
```python
def test_send_packet_multiple():
    packet = Ether() / IP(dst="8.8.8.8") / DNS(rd=1, qd=DNSQR(qname="www.
        google.com"))
    packet_info = {"type": "DNS", "number": 5}
    result = send_packet(packet, packet_info)
    assert result[0] is not None  # Check that there is at least one answered
        packet
    assert result[1] is not None  # Check that there are no unanswered packets
    assert len(result[2]) == 5  # Check that all packets were sent

# Test check_packet_type_assign_send_method function with a valid packet type
def test_check_packet_type_assign_send_method_valid():
    packet = Ether() / IP(dst="8.8.8.8") / DNS(rd=1, qd=DNSQR(qname="www.
        google.com"))
    packet_info = {"type": "DNS", "number": 1}
    send_method = check_packet_type_assign_send_method(packet, packet_info)
    assert send_method.__name__ == "sr"  # Check that the correct send method
        was assigned

# Test check_packet_type_assign_send_method function with an invalid packet
    type
def test_check_packet_type_assign_send_method_invalid():
    packet = Ether() / IP(dst="8.8.8.8") / DNS(rd=1, qd=DNSQR(qname="www.
        google.com"))
    packet_info = {"type": "HTTP", "number": 1}
    send_method, iface = check_packet_type_assign_send_method(packet,
        packet_info)
    assert send_method.__name__ == "sr"  # Check that the default send method
        was assigned
```

4 Developing the User Interface

Since PyQt code is not the main point of this project, I'll explain briefly the user interface and the classes behind them. The interface consists of two pages, the packet-sending page, and the summary page. The packet-sending page consists of options: the ability to select different network packet protocols, edit the packets, add different packets to a list to send later, and send the list of packets. The summary page consists of options that allow the user to see the received and sent packets, the amount of time it took to send every packet and receive an answer, any output that Scapy produced, exported to Wireshark the packets received and sent, and download the received and sent packets to a PCAP file.

4.1 Packet Sending Page

The packet-sending page consists of three different sections and a few subsections. The three main sections are Protocol Selection, Packets To Send, and Configuration. The Protocol Selection is for selecting the different types of protocols: IP, ARP, and DNS to be precise. The Configuration section changes based on the protocol selected, in figure 1 below the protocol is IP so the configuration is set to IP Packet Configuration. The Packets To Send section displays the packets that are to be sent that were added through the use of the additional packets button. When a user selects a protocol, the default data represents the given protocol's packet information dictionary which was explained in section 2.2. In Figure 1, we can see that the selected protocol is IP and the packet type is TCP. In this case, the TCP packet information dictionary will be used as the default data. When the add packets button is clicked, the packet will be sent to the packets to send list and displayed to the user, where the user can open the packet to see if the information is correct. After the send packets button is pressed, the buttons will become disabled and the user will be redirected to the Summary page.



Protocol Type: IP

ICMP [✓ TCP] UDP

Source Port: 0

Destination Port: 80

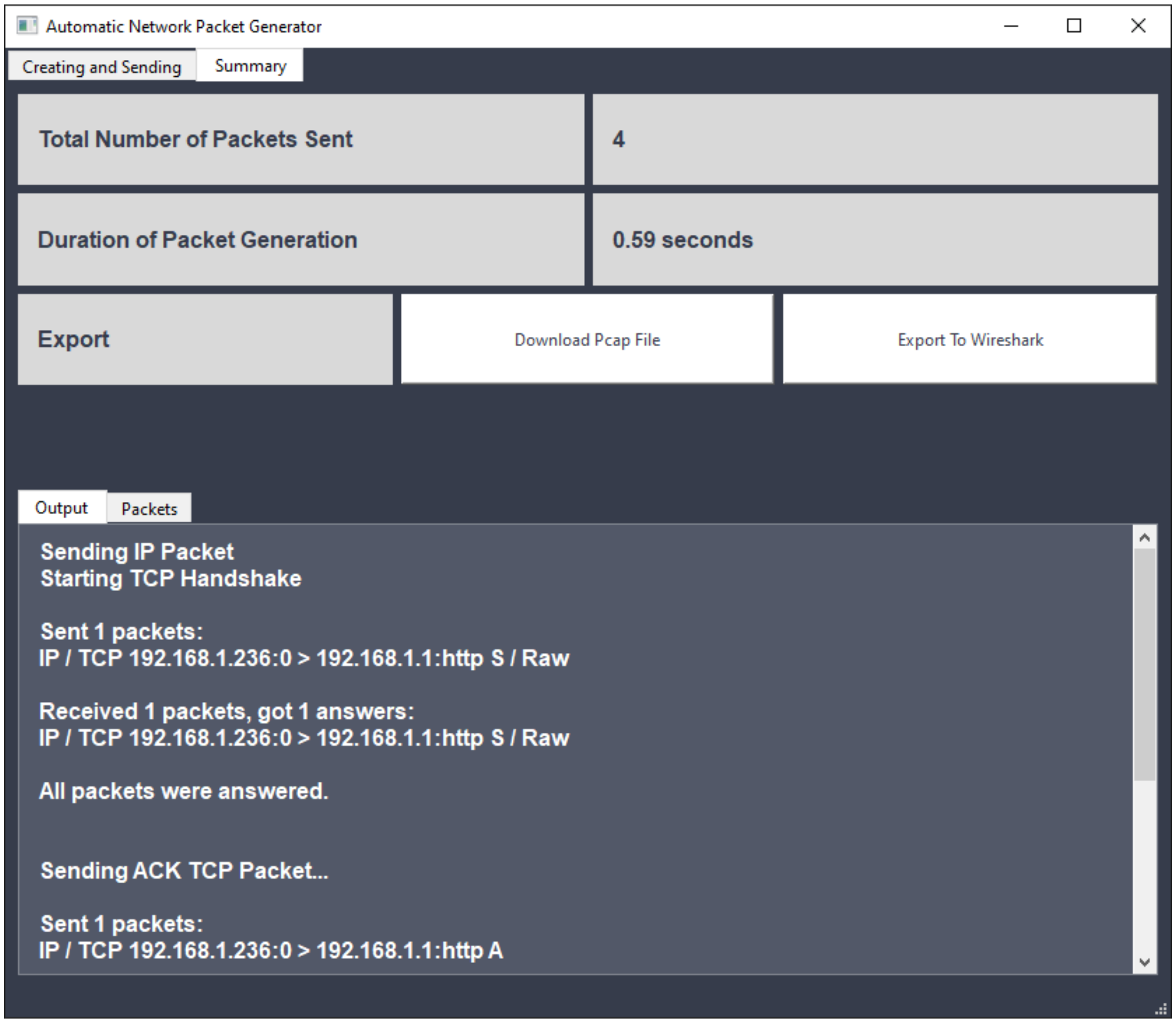
Source IP Address: 127.0.0.1

Destination IP Address: 127.0.0.1

Figure 1: Example image

4.2 Packet Summary Page

The summary page consists of three main sections: Output, Packets, and Details / Export. The output section (which is displayed in Figure 2) consists of updates for the user, these updates include whether the packets were sent, or received, and if there were any unanswered packets. The packets section includes packets displayed in a list in a more readable format than the output section. It displays the packet number, source address, destination, and packet type. In Figure 3, the section of the packet is shown for a TCP Handshake. Once the user has confirmed that the packets were sent and received, they have the option of downloading or opening the PCAP file in Wireshark.



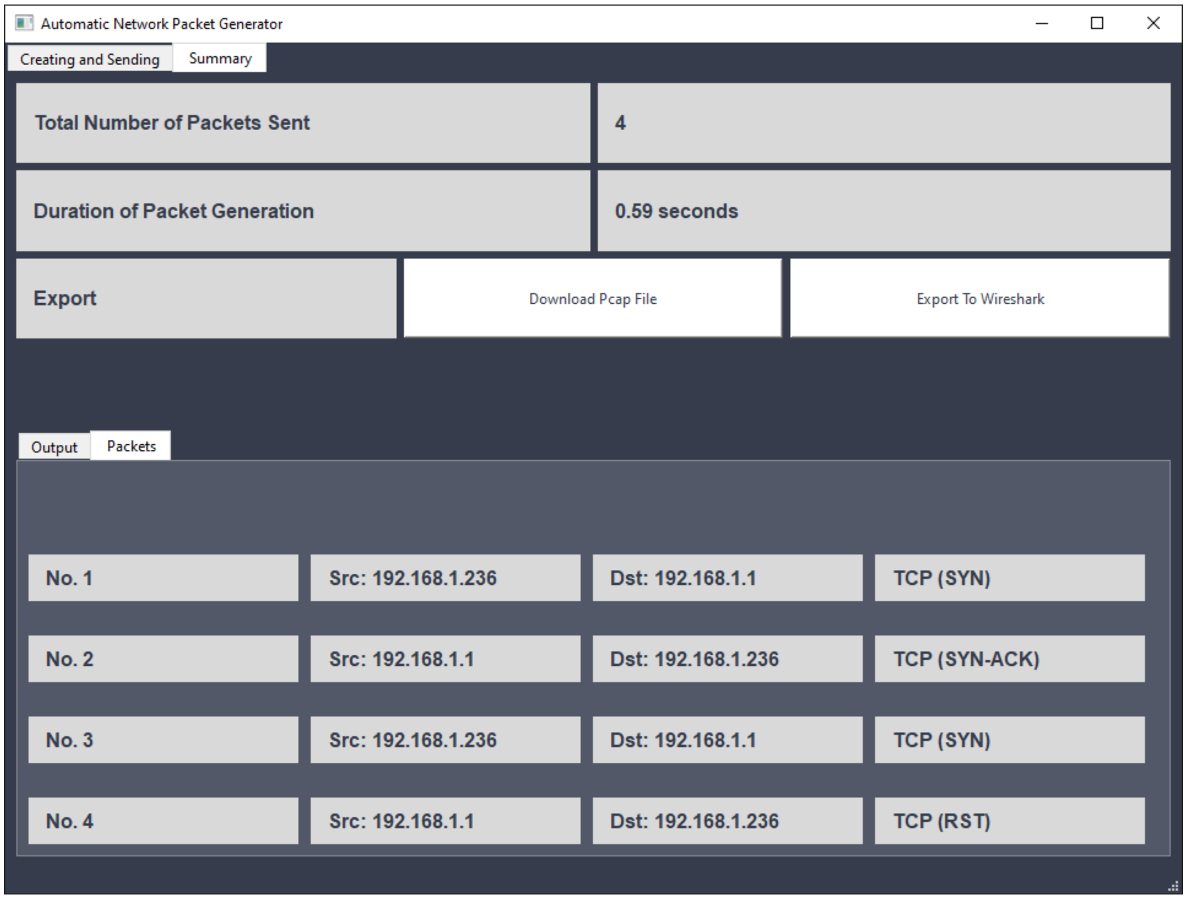
Total Number of Packets Sent: 4

Duration of Packet Generation: 0.59 seconds

Export

```
Sending IP Packet
Starting TCP Handshake

Sent 1 packets:
IP / TCP 192.168.1.236:0 > 192.168.1.1:http S / Raw

Received 1 packets, got 1 answers:
IP / TCP 192.168.1.236:0 > 192.168.1.1:http S / Raw

All packets were answered.


Sending ACK TCP Packet...

Sent 1 packets:
IP / TCP 192.168.1.236:0 > 192.168.1.1:http A
```

Figure 2: Summary Output with TCP Handshake



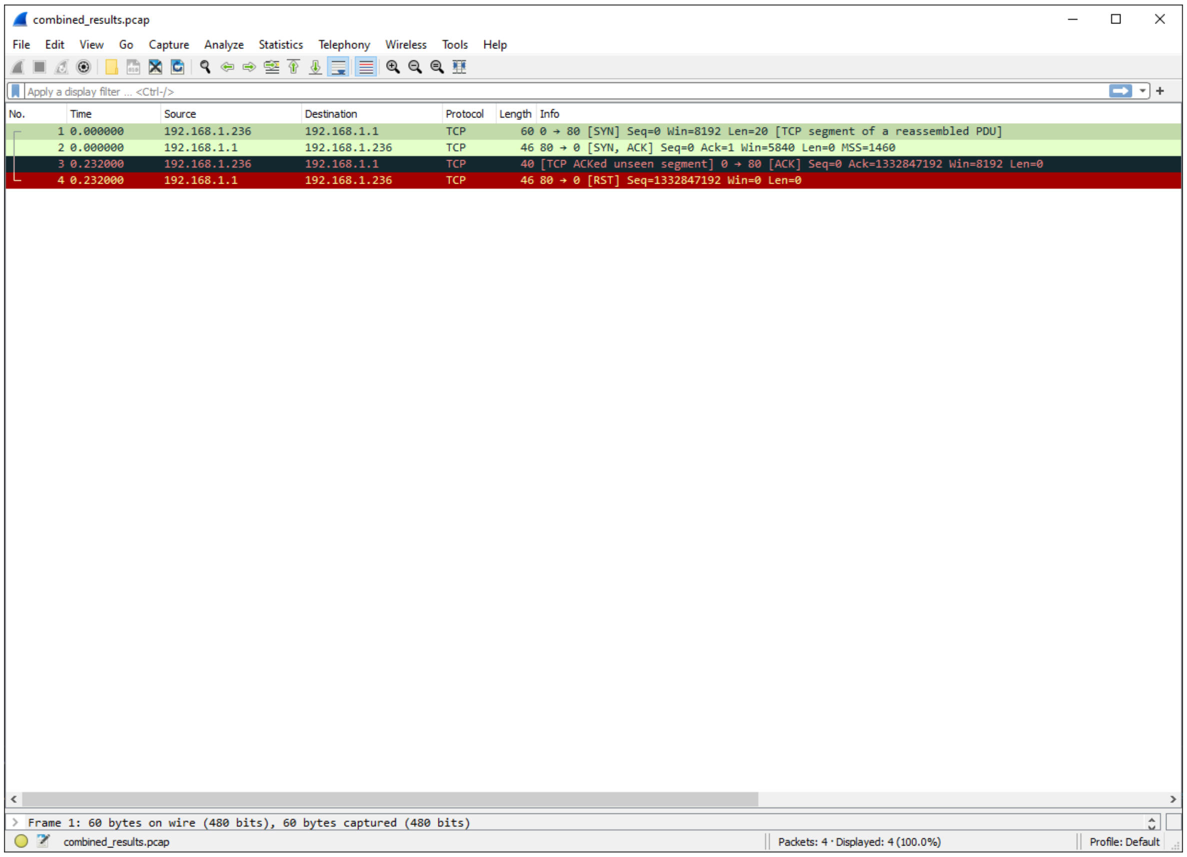
Total Number of Packets Sent	4
Duration of Packet Generation	0.59 seconds
Export	

No. 1	Src: 192.168.1.236	Dst: 192.168.1.1	TCP (SYN)
No. 2	Src: 192.168.1.1	Dst: 192.168.1.236	TCP (SYN-ACK)
No. 3	Src: 192.168.1.236	Dst: 192.168.1.1	TCP (SYN)
No. 4	Src: 192.168.1.1	Dst: 192.168.1.236	TCP (RST)

Figure 3: Summary Packets List with TCP Handshake

4.3 Exporting to Wireshark

The export to Wireshark button handles creating a PCAP file, letting the user save the file where they want, and opening the file in Wireshark. The following figure, or Figure 4 represents what will be shown if the TCP handshake shown before was exported to Wireshark. This export represents the combined results, so each packets sent and received responses. It's important to note that if I had multiple TCP Handshakes happening, or a mix of network packets, these would all be exported to one Wireshark PCAP file.


combined_results.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time      Source         Destination    Protocol Length Info
   1 0.000000  192.168.1.236  192.168.1.1    TCP      60 0 → 80 [SYN] Seq=0 Win=8192 Len=20 [TCP segment of a reassembled PDU]
   2 0.000000  192.168.1.1    192.168.1.236  TCP      46 80 → 0 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460
   3 0.232000  192.168.1.236  192.168.1.1    TCP      40 [TCP ACKed unseen segment] 0 → 80 [ACK] Seq=0 Ack=1332847192 Win=8192 Len=0
   4 0.232000  192.168.1.1    192.168.1.236  TCP      46 80 → 0 [RST] Seq=1332847192 Win=0 Len=0

Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
combined_results.pcap    Packets: 4 · Displayed: 4 (100.0%)    Profile: Default
```

Figure 4: TCP Handshake Exported to Wireshark

5 Conclusion

With an increase in the necessity for network testing and analysis, the development of an Automatic Network Generator Interface was a project that I thought would be useful to some professionals. This project allows a user to easily send network packets across a network and export them to Wireshark for further analysis. Some limitations of this project are the time frame that was dealt to me. I believe with further testing and planning, the project would have turned out significantly better than it was. Especially if it was a team project. I've learned that researching a library's capabilities is a necessity when planning for a large-scale project and that insufficient design and research can lead to unwanted and larger problems in the long run. For example, I did not fully understand the Scapy send functions and their abilities. I spent a week figuring out how to code threads with PyQt when Scapy can handle that automatically by passing the thread into the send function. This lack of research into the capabilities of Scapy was what led to many mistakes in developing this project. Although there were many mistakes and work I could complete in a better way, I believe I came out of this with a better understanding of how networks work and how packets are built and created. Not to mention my programming abilities have increased since this was my first real Python project. Overall, I developed an Automatic Network Generator Interface that allows a user to easily send packets across a network and hopefully provides a guide and warning of potential mistakes to those looking to build their own Automatic Network Generator Interface.

References

[1] The Scapy Project. (n.d.). Scapy Documentation. Retrieved from https://scapy.net/

[2] Eichner, R., & Summerfield, M. (2015). *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming*. Prentice Hall Press.

[3] Okken, B. (2017). *Python Testing with pytest*. Pragmatic Bookshelf.