AGH University of Science and Technology

Faculty of Computer Science, Electronics and Telecommunications

Design Laboratory

Cybersecurity, Reliability and Risk

Topic: Covert channel using TTL variance.

Date	: 27.01.2021
Students	: Tomasz Lejkowski, Szymon Mazurek
Supervisor	: Prof. Dr. Hab. Piotr Chołda

1. Introduction

Given program presents a covert channel exemplary implementation with the usage of TTL field in the IPv4 protocol packet header as a way to send messages. The program consists of two parts: message coder and decoder.

The coder is responsible for simulating the data flow, similar to the one found on the Internet, then encoding a message as an 8bit representation of input ASCII characters. On the basis of the article[1] packets representing low or high bits, are sent in a group of 3 and with the maximum difference between them equal 1, which should give us stealth in the network.

Then, the whole packet flow is saved into a packet capture file (.pcap) file, which should be analyzed by the second part of the software - the decoder. The decoder reads the file, and at first, packets examines the properties of the data flow such as:

- TTL values,
- Amplitude,
- Rate of TTL changes,

and then look for a key indicating the beginning of the covert message transmission. After being found, the message is converted back from the binary representation into ASCII characters and displayed to the user. If no message is found the user is also informed about this fact.

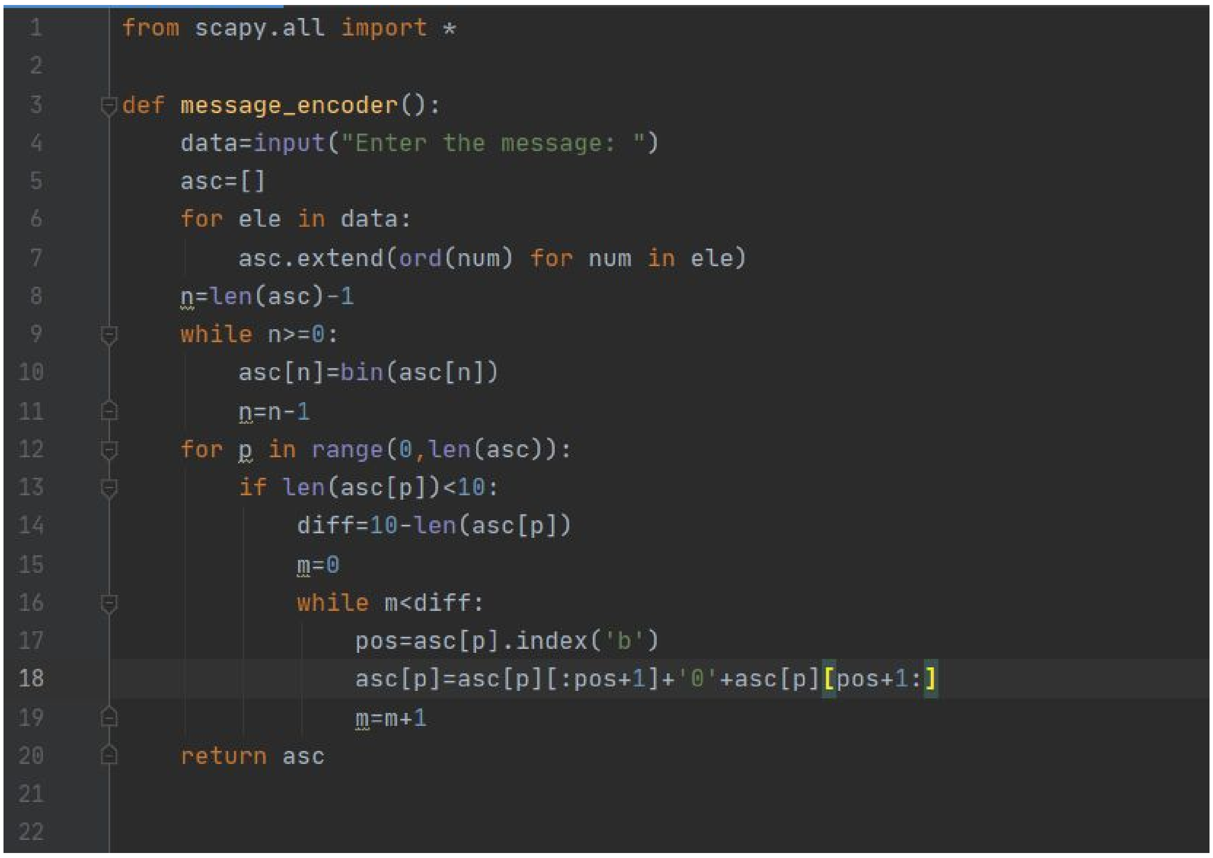
2. Environment

The program was created using Python 3.8 programming language with PyCharm 2020 IDE. To provide proper handling of the packet manipulation ScaPy 2.4.4 library was utilized. For the proper operation of the mentioned library on the Windows 10 operating system, it is necessary to install Npcap 1.10 (packet capture library for Windows). An important note to mention is that using such libraries requires administrator privileges.

3. Encoder

The following description will provide information about the specification of the code of the first part of the program - encoder.

Firstly, the message_encoder function is defined. The user is asked to input the message that will be sent. Next, the function translates every ASCII character into the corresponding integer and writes it into the list. Then every element of the list is converted into binary numbers. Built-in function converting integers to binary numbers removes any ‘0’ at the beginning of the number, therefore the program checks if the binary number contains always 8 bits, and adds zeros in the beginning if it is not true. The function returns a list containing the binary representation of input ASCII characters, each 8 bits long.

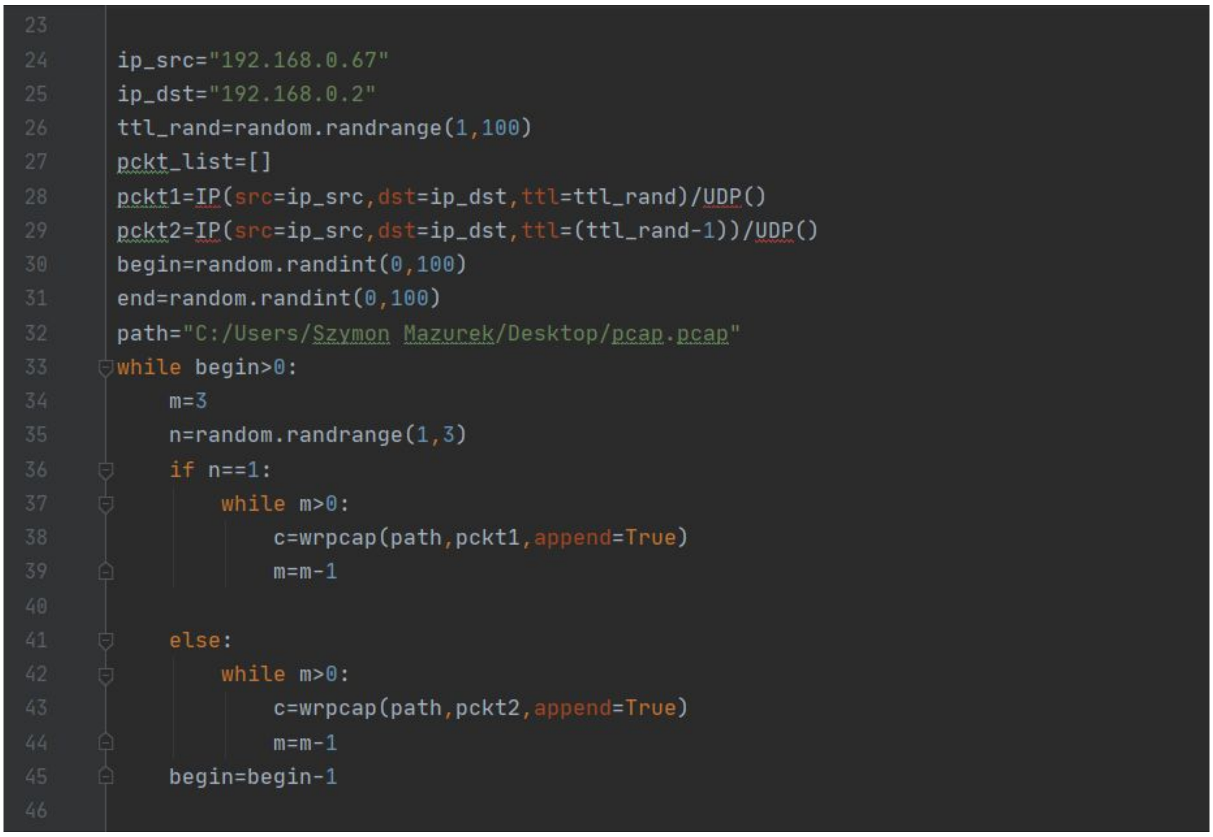


```python
1   from scapy.all import *
2
3   def message_encoder():
4       data=input("Enter the message: ")
5       asc=[]
6       for ele in data:
7           asc.extend(ord(num) for num in ele)
8       n=len(asc)-1
9       while n>=0:
10          asc[n]=bin(asc[n])
11          n=n-1
12      for p in range(0,len(asc)):
13          if len(asc[p])<10:
14              diff=10-len(asc[p])
15              m=0
16              while m<diff:
17                  pos=asc[p].index('b')
18                  asc[p]=asc[p][:pos+1]+'0'+asc[p][pos+1:]
19                  m=m+1
20      return asc
21
22
```

The next part (lines 24-31) are responsible for setting parameters of the simulated data flow:

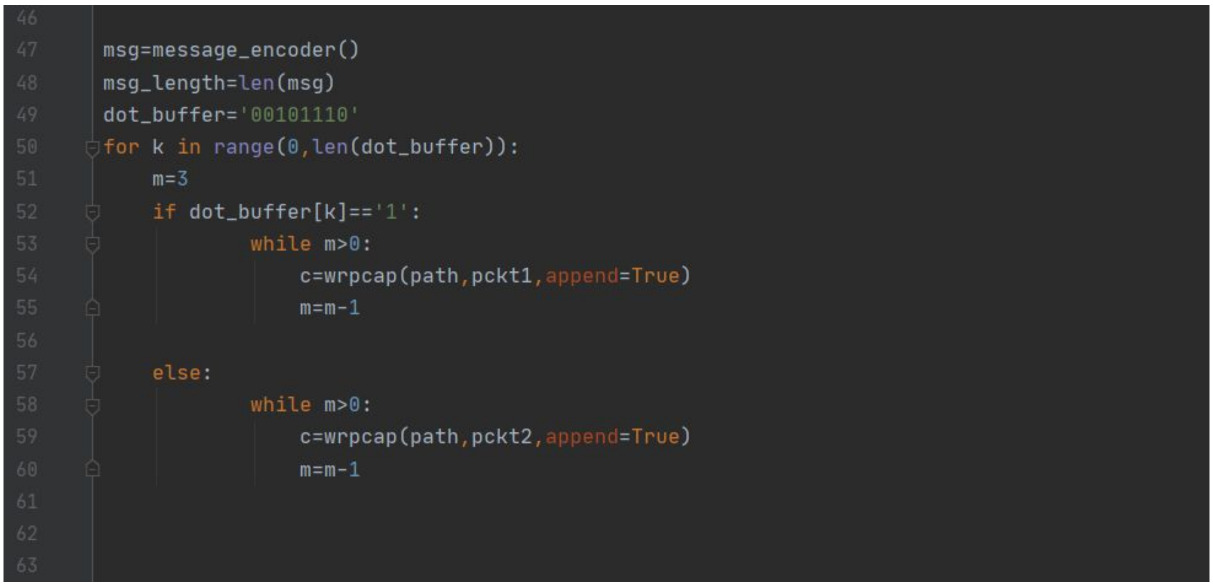
- source and destination IP address
- TTL value of the packets (randomly chosen from numbers between 2 and 99)
- creating packets that will be used during simulating data flow and sending the message
- number of packets sent before and after the main message (to simulate natural packet flow)

Lines 33-45 are responsible for sending the first part of random data flow. TTL changes in this flow every three packets sent. The amount of triples of packets sent is determined by the randomly chosen *begin* value. The packets are appended into the .pcap file (directory of the file should be chosen accordingly to the user’s own preferences and needs, here it is configured for testing purposes)



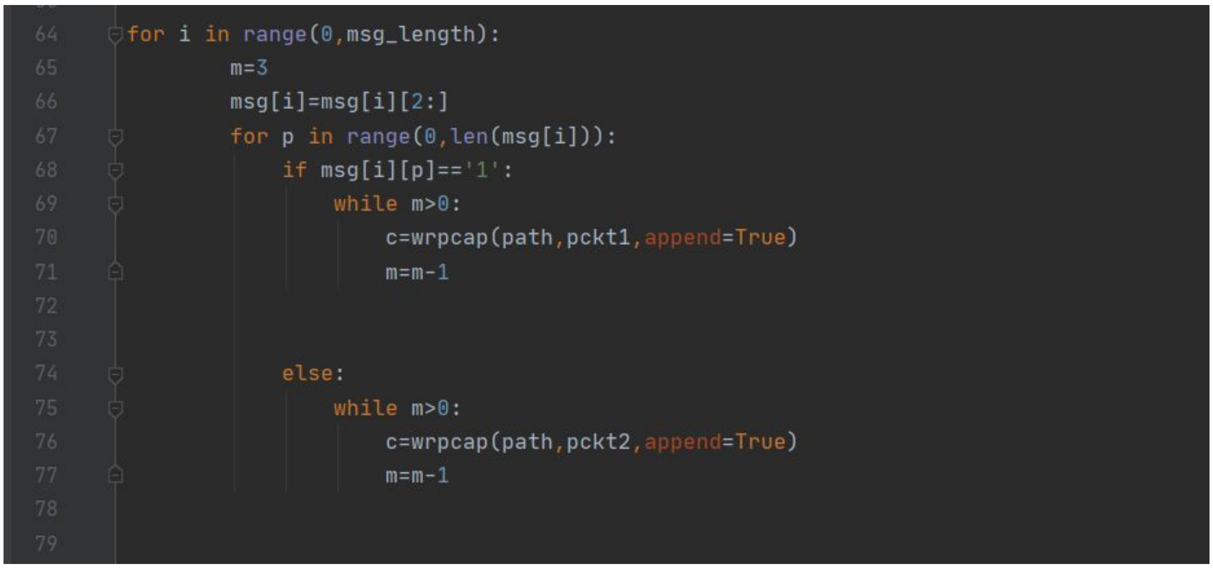
```
23
24  ip_src="192.168.0.67"
25  ip_dst="192.168.0.2"
26  ttl_rand=random.randrange(1,100)
27  pckt_list=[]
28  pckt1=IP(src=ip_src,dst=ip_dst,ttl=ttl_rand)/UDP()
29  pckt2=IP(src=ip_src,dst=ip_dst,ttl=(ttl_rand-1))/UDP()
30  begin=random.randint(0,100)
31  end=random.randint(0,100)
32  path="C:/Users/Szymon Mazurek/Desktop/pcap.pcap"
33  while begin>0:
34      m=3
35      n=random.randrange(1,3)
36      if n==1:
37          while m>0:
38              c=wrpcap(path,pckt1,append=True)
39              m=m-1
40
41      else:
42          while m>0:
43              c=wrpcap(path,pckt2,append=True)
44              m=m-1
45      begin=begin-1
46
```

In line 47, the message_encoder function is called. After the input message is given, the program sends the “.” sign (binary representation in line 49) to signal the beginning of the data transfer. The program sends packets with higher or lower TTL value depending on the corresponding bit value of the message. As previously, packets are sent in groups of three to avoid detection of TTL manipulations.



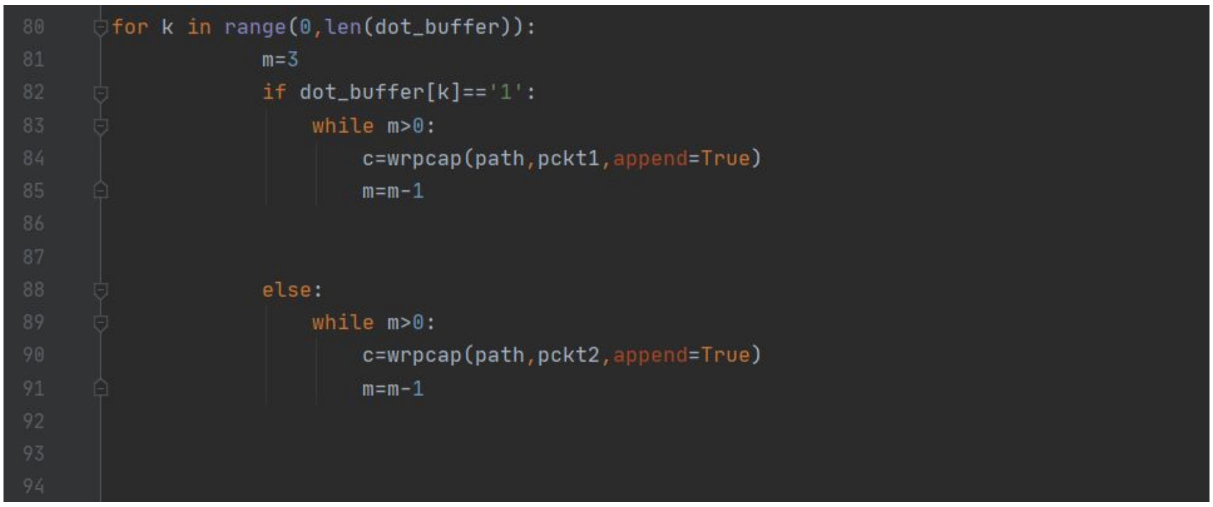
```
46
47  msg=message_encoder()
48  msg_length=len(msg)
49  dot_buffer='00101110'
50  for k in range(0,len(dot_buffer)):
51      m=3
52      if dot_buffer[k]=='1':
53              while m>0:
54                  c=wrpcap(path,pckt1,append=True)
55                  m=m-1
56
57      else:
58              while m>0:
59                  c=wrpcap(path,pckt2,append=True)
60                  m=m-1
61
62
63
```

Lines 64-79 show the sending procedure of the input message. The principle is the same as the one used to send the “.” sign.



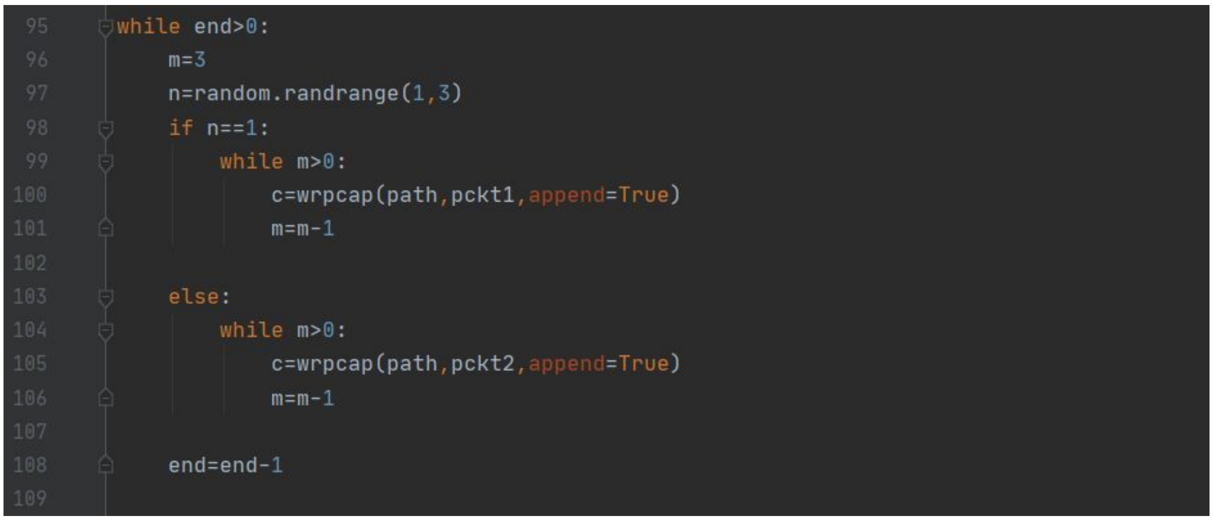
```
64  for i in range(0,msg_length):
65          m=3
66          msg[i]=msg[i][2:]
67          for p in range(0,len(msg[i])):
68              if msg[i][p]=='1':
69                  while m>0:
70                      c=wrpcap(path,pckt1,append=True)
71                      m=m-1
72
73
74              else:
75                  while m>0:
76                      c=wrpcap(path,pckt2,append=True)
77                      m=m-1
78
79
```

After sending the message, the ‘ . ’ key is sent once again to show the end of the message.



```
80  for k in range(0,len(dot_buffer)):
81              m=3
82              if dot_buffer[k]=='1':
83                  while m>0:
84                      c=wrpcap(path,pckt1,append=True)
85                      m=m-1
86
87
88              else:
89                  while m>0:
90                      c=wrpcap(path,pckt2,append=True)
91                      m=m-1
92
93
94
```

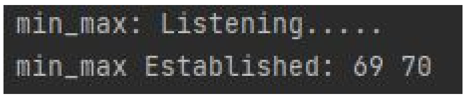
In the end, random packets are created once again to simulate natural data flow, using the same principle as in the beginning part of the data flow simulation.



```python
while end>0:
    m=3
    n=random.randrange(1,3)
    if n==1:
        while m>0:
            c=wrpcap(path,pckt1,append=True)
            m=m-1

    else:
        while m>0:
            c=wrpcap(path,pckt2,append=True)
            m=m-1

    end=end-1

```

4. Decoder

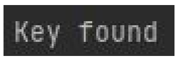
At first, the decoder needs to know what low and high TTL values are, so we listen, on a certain port, first few packets to determine whether we see a change with amplitude 1 and if so, remember those values for decoding. In Fig. 4.1 we see an output, confirming established TTL values.



```
min_max: Listening.....
min_max Established: 69 70
```

Figure 4.1

Using those values we search for an agreed key (Fig.4.2), which is a binary representation of dot. This function also attaches our listening exactly for the packet change, which gives us redundancy. During receiving, without any data loss, we can lose up to 2 packets during the whole transmission this can be manually changed in the decoder and encoder source code.

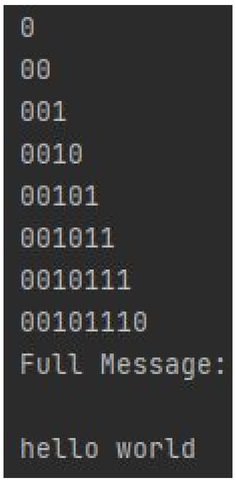


```
Key found
```

Figure 4.2

Just after the dot, we are creating a byte, bit by bit, from single TTL values, then convert that byte into a decimal, and then into ASCII which gives us the output message.

Alongside decoding, we must check for another key, if we find it, that means we end the transmission. In Fig 4.3 we can see the output of the program showing the assembly of the last revived information: that is a dot in binary and showing the full message received in transmission.



```
0
00
001
0010
00101
001011
0010111
00101110
Full Message:

hello world
```

Figure 4.3

I’ve decided not to describe my source code here, that is done as comments in the source code file.

5. Flaws and future work

1. No error correction.
    a. That means if more than the accepted number of packets are dropped, we lose the rest of the message.
    b. If we lose 1 packet, we do not correct our reading position to again acquire a loss of 2 packets.
2. No new TTL adaptation.
    a. When the path of packets will change, we will get completely new TTL values, which will crash the app.
3. Only string input accepted.
    a. On the same concept, it is possible to build other data type transmission.
4. Strictly simplex system.
    a. As of right now, the system is only simplex, we use separate programs to code and decode, there is no implementation of any conversation.
5. No real network transmission.
    a. In this version, we save our simulated network traffic into a file which we analyze on the other side. This was caused strictly by software incompatibilities, as Scapy was unreliable when sniffing on Win10. But the research found that using the Linux platform together with raw socket library will create more reliable header sniffing.

6. Teamwork division

We tried to split our efforts equally, we both did a lot of research about steganography, but Mr. Mazuerek proposed and explained the use of the TTL field. The encoder was strictly done by Szymon Mazurek. Tomasz Lejkowski worked at the decoder and on debugging the program in LAN and loopback environment, but we did not opt to switch to raw sockets and Linux platform but to show working coder and decoder using packet capture file.

7. Sources, bibliography and references

1. S. Zander, P.Branch "Covert channels in the IP time to live field", 2007
2. A.Mileva, B.Panajotov "Covert Channels in TCP/IP Protocol Stack", 2014
3. C.Abad "IP Checksum Covert Channels and Selected Hash Collision", 2001
4. Scapy Philippe Biondi and the Scapy community 2021 scapy.net
5. Npcap https://nmap.org/npcap/