

Exercise 6

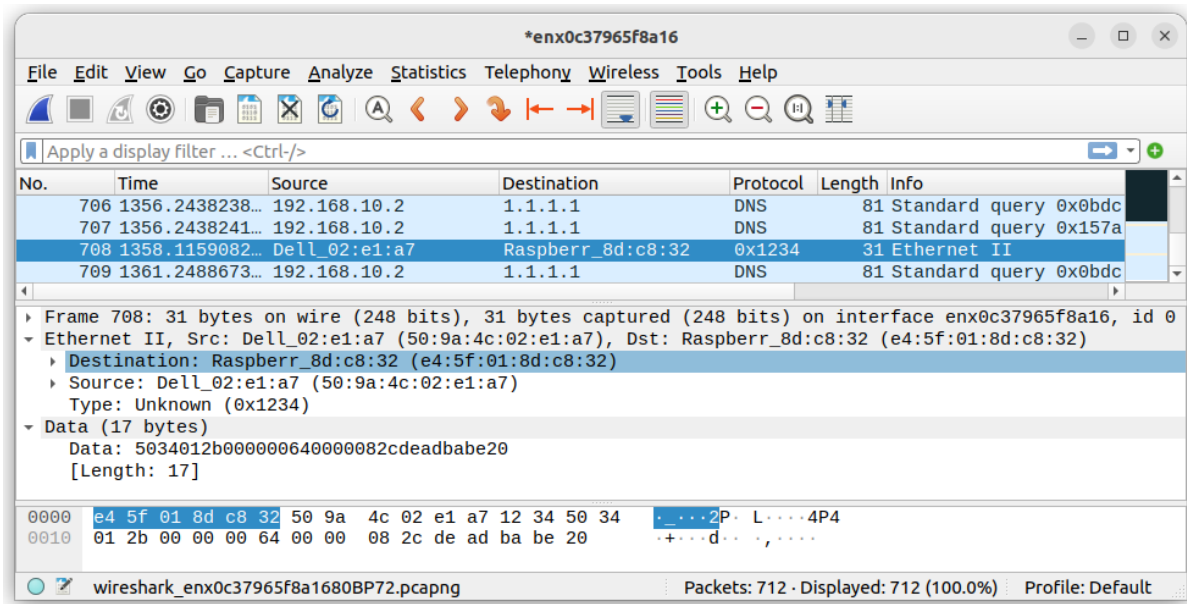
Overview

In this exercise we will implement some form of encryption on the packets being sent on a network, using a combination of the P4 and python programming languages. The network consists of a Linux machine and a Raspberry Pi connected by an ethernet cable.

Packet contents

Firstly, let's think about what we actually want to encrypt. P4 operates primarily on packet headers, so in this case we will put data in them and encrypt that rather than the payload. But how much of the header can we encrypt and still have it be transmitted to the correct recipient?

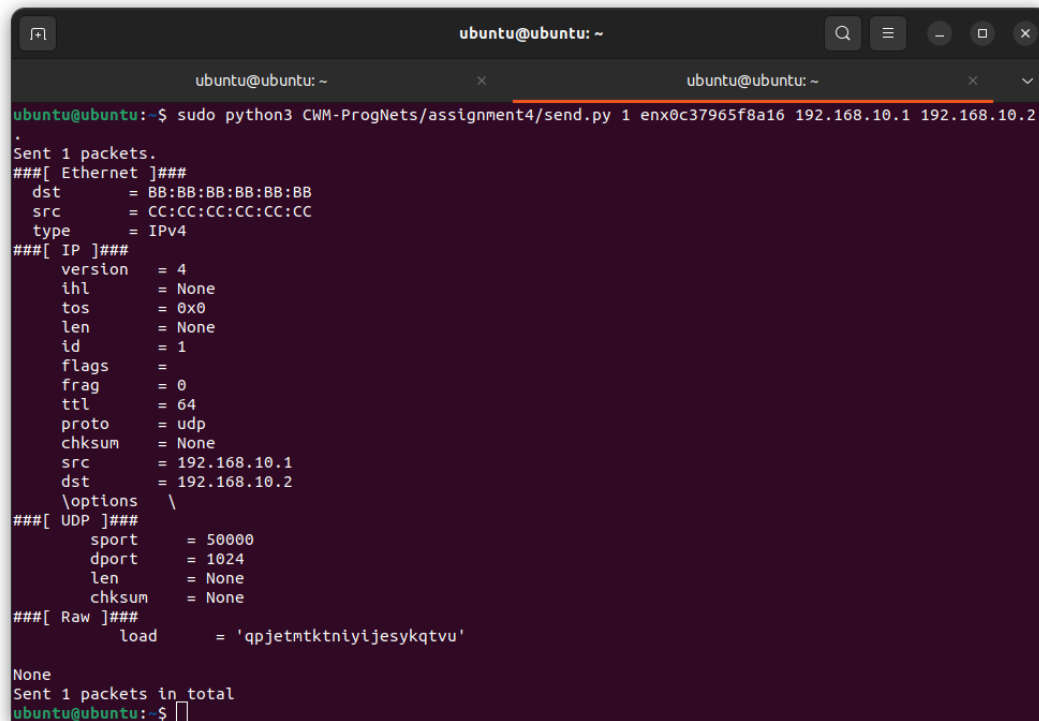
Exercise 5 uses packets at the data layer (OSI L2) and so the header consists of two ethernet addresses and their type, stacked onto the custom *P4calc* header that was written for the calculator to work.



```
ubuntu@ubuntu:~$ sudo python3 CWM-ProgNets/assignment5/calc.py
> 100 +02092
100 +02092
###[ Ethernet ]###
  dst      = e4:5f:01:8d:c8:32
  src      = 50:9a:4c:02:e1:a7
  type     = 0x1234
###[ P4calc ]###
  p        = 'P'
  four     = '4'
  version  = 0x1
  op       = '+'
  operand_a = 100
  operand_b = 2092
  result   = 3735927486
###[ Raw ]###
  load     = ' '
```

A packet sent using the code from Exercise 5 (calc.py). The result is an arbitrary default value here since the packet has not arrived at the Raspberry Pi

In Exercise 4, the packet is sent using UDP, a transport layer (OSI L4) protocol. Here we see that there are multiple layers of header stacking: ethernet onto IP onto UDP onto the payload.

A terminal window on an Ubuntu system showing the output of a Python script. The script sends a packet to the interface 'enx0c37965f8a16' from 192.168.10.1 to 192.168.10.2. The output displays the packet structure in a hierarchical format, showing the Ethernet II header, the IP header, and the UDP header, followed by the raw payload.

```
ubuntu@ubuntu: ~  
ubuntu@ubuntu: ~  
ubuntu@ubuntu: ~  
ubuntu@ubuntu:~$ sudo python3 CWM-ProgNets/assignment4/send.py 1 enx0c37965f8a16 192.168.10.1 192.168.10.2  
.  
Sent 1 packets.  
###[ Ethernet ]###  
  dst      = BB:BB:BB:BB:BB:BB  
  src      = CC:CC:CC:CC:CC:CC  
  type     = IPv4  
###[ IP ]###  
  version  = 4  
  ihl      = None  
  tos      = 0x0  
  len      = None  
  id       = 1  
  flags    =  
  frag     = 0  
  ttl      = 64  
  proto    = udp  
  chksum   = None  
  src      = 192.168.10.1  
  dst      = 192.168.10.2  
  \options \  
###[ UDP ]###  
  sport    = 50000  
  dport    = 1024  
  len      = None  
  chksum   = None  
###[ Raw ]###  
  load     = 'qpjetmtktniyjesyqtvu'  
  
None  
Sent 1 packets in total  
ubuntu@ubuntu:~$
```

A packet sent using the code from Exercise 4 (send.py)

In both cases the connection relies on a working ethernet header, so we won't encrypt that in this case, or the packet will never reach its correct destination. In the case of Exercise 4, the connection relies on all 3 of the headers (L2, L3 and L4), so we cannot encrypt them either (but we can change some of them, such as in the case of VPNs). Therefore, we will leave them as is.

For simplicity's sake, we will send packets at the data layer, since the network graph consists of only one edge (the ethernet cable) and hence there is no use for routing, which would happen at the control layer (OSI L3). This means that our packets will have a structure similar to that of those in Exercise 5:

```

###[ Ethernet ]###
  dst      = e4:5f:01:8d:c8:32
  src      = 50:9a:4c:02:e1:a7
  type     = 0x1234
###[ SecureHeader ]###
  encrypted_data = 'message'
  ...
###[ Raw ]###
  load       = ' '

```

Mockup of the structure of the packet with encrypted header

Now we have to decide what our *SecureHeader* looks like. Let's send an encrypted message consisting of a short string of ASCII characters. We could also indicate whether the packet is in fact an encrypted message, since a given packet might not actually have been encrypted. For now, we will use a plaintext tag at the top of *SecureHeader*, with a command word to indicate that the message is to be decrypted (or encrypted).

Perhaps in the future, this could be implemented by a checksum for more discreet communications, since it is a bit silly to announce to someone that your message is for decryption. Or we could have the bits associated with the tag decrypt to something meaningful.

```

* The Protocol header looks like this:
*
*      <--tag-->      <--cmd-->
*      0          1      2          3
* +-----+-----+-----+-----+
* |  P          4          $          e/d  |
* +-----+-----+-----+-----+
* |   data                                           |
* +-----+-----+-----+-----+
* |   data                                           |
* +-----+-----+-----+-----+
* |   etc.                                           |
* +-----+-----+-----+-----+
*
*
* P is ASCII Letter 'P' (0x50)
* 4 is ASCII Letter '4' (0x34)
* $ is ASCII Letter '$' (0x24)
* e/d are ASCII Letter 'e' (0x65) or 'd' (0x64)

```

The structure of SecureHeader

Simple encryption with XOR

Let's consider the properties of the XOR function. Here is its truth table:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

We see that bit B is inverted when bit A is 1. Therefore, for a given string of bits C, we can selectively invert its bits based on a string of bits D, and an attacker will not know which bits were inverted at first inspection. In this case, D is the key to the hidden message.

We might now ask, how do we retrieve C from the encrypted string? We can use D again to find which bits were inverted and re-invert them. This is equivalent to applying XOR D again, so XOR is self-inverting!

Implementation in python and P4

The overall implementation looks as follows:

- The python side runs a CLI which converts user inputs into packets
- The user message is stored in a custom *SecureHeader* protocol along with the instruction for the network device
- The P4 side takes the packet and parses the instruction for the command
- It uses the command as the key for a table match-action pipeline
- Depending on the command it applies an XOR with the relevant key, or reflects the packet
- The python side prints the packet contents and checks that it was processed correctly
- The python side also has a help command which prints short documentation for the commands

[illegible]

A screenshot of the output of the program when “This is a message” is sent for encryption. This output is annotated with extra information for debugging

Evaluation

In the end, we did not get to implement as much as we wanted. In particular, I struggled with handling the different data types in python, since bytes, hexadecimal numbers, and integers are all formatted differently. I would have liked to add more commands to the CLI and perhaps implement the checksum or something of the like so that more of the packet would be encrypted. But, it did work, so I would say that this mini project was at least partially successful.