Network Security

Assignment 1, Thursday, September 3, 2015, version 1.0

Handing in your answers: Submission via Blackboard (`http://blackboard.ru.nl`)

Deadline: Thursday, September 10, 23:59:59 (midnight)

DON'T PANIC

Yes. This assignment really is 11 pages long. However, since you will be learning a new programming language and a new programming paradigm, a lot of it is filled with code examples and explanation. Even so, there is a chance that this is simply too much work to fit into two hours of Werkcollege and four hours of unsupervised group work, which is the amount of work you should put into this 3EC course (on average). If at any point beyond that you feel that you cannot spend any more time on this exercise, you can stop there without fear of getting an NSI, so it will not influence your chances of passing this course. However, in that case we would like you to provide us with a short note on what you were stuck on, what parts took a lot of time, and how long you have worked on the exercise. This way we can tune it for future students.

Having said that, I wish you good luck on this exercise, and hope it takes you a lot less time to complete it than it took me to create it. If you have any questions, send me an e-mail or drop by my office in Mercator 1 (M1.2.13).

– Pol

In this course you will learn about network security. Fundamental assumptions in network security are that an attacker is able to eavesdrop on and/or manipulate network traffic. Therefore, in this course you will be eavesdropping and manipulating network traffic. For this purpose, you will need to write some software. We assume basic programming skills, as taught in the courses Imperatief Programmeren 1 & 2, or equivalent knowledge. No previous knowledge in network programming is required. The programming language of choice for this course is Python, because it makes it relatively easy to write networking tools in a few lines of code. Since you have not been taught Python, this first exercise is designed to familiarize you with the language and with basic network programming.

The Python programming language comes in two flavours: Python 2 and Python 3. As the names imply, Python 2 is the older version of the language, whereas Python 3 is the shiny new iteration. Python 2 is no longer actively developed, but does receive bugfixes. Python 3 is considered the future of the language. There are some incompatibilities between the two, and this poses a problem when using libraries that are not yet ported to Python 3. It is considered good practice to write any *new* software in Python 3, so we will start working with Python 3. We will handle any incompatibilities with Python 2 if and when they become relevant.

For many of the exercises in this course you will need a unix-like environment (Linux, BSD, Mac OS) with root access on the machine you are working on. We are unable to provide such an environment for every student, so we ask everyone with their own laptop to use that. In particular, students do not have root rights on the terminals in the Huygens building. If you do not have a system on which you can run Linux with root access natively, you can use the Debian-based virtual machine we have built from within Windows or Linux. If you are unable to use this, or have other problems in obtaining a Linux system with root rights, contact us as soon as possible.

The exercises were written on an Arch Linux-environment for Python 3.4.1, but should work on other Linux and Python versions as well. Students are expected to be able to deal with platform-specific differences arising from using another operating system or Python version (as you will be expected to in the real world), but if you need assistance contact us as soon as possible by e-mail.

At the end of this exercise, you will have sent some data to a separate program running on the same machine, programmed a basic network sniffer in Python, and analyzed the structure of some different types of network packets.

If you wish to learn more about Python, or need documentation for these exercises, the official Python website has both a tutorial (`https://docs.python.org/3.4/tutorial/`) of which chapters 3–5 will be very helpful, and official documentation on the standard libraries including the socket modules (`https://docs.python.org/3.4/index.html`). Note that you can change the version in the top left corner to fit the version you are using.

1. (a) *Hello, World!* is the most boring assignment in the history of programming education, closely followed by the fibonacci numbers for functional programming. Since you have not done any network programming in your first year, we will be skipping all that boring stuff and just dive straight into network sockets.

   Python can be executed in an interactive shell, or from source files. Start an interactive Python shell by issuing the command `python3` on the command line. The Python interpreter should start, and you should see an input line prefixed by `>>>`.

   On a separate command line (*not* in a Python shell), start the command `nc -l -p 42424`. On some versions of netcat it is necessary to omit the `-p`, making it `nc -l 42424`. This opens a listening TCP socket on port 42424. If you get the error ”Address already in use”, pick a different port between 1024 and 65535.

   In the interactive Python shell, type the following command:

```
>>> import socket
```

   This is the way to import a library in Python. You now have access to all the socket operations, as documented on https://docs.python.org/3/library/socket.html. A socket is an endpoint in network communication. Since there are many different network protocols, there are many different socket types encapsulating them at different layers.

   We will create a TCP socket to connect and send some data to the netcat process in the second shell. Issue the following command:

```
>>> s = socket.create_connection(("localhost", 42424))
```

   This asks the socket library to create a connection to the endpoint defined by the tuple (”localhost”, 42424). This is the hostname `localhost`, which will be your local machine, at port 42424. Then we assign the resulting socket object to `s`.

   Note that we do not specify a type for `s`. Python is a dynamically typed language, using a form of typing called “duck typing”, which can be explained as “if it walks like a duck, swims like a duck, and quacks like a duck, then it’s probably a duck”. Duck typing is about objects behaving in a certain way, rather than being of a certain type. This is checked dynamically at runtime, and causes runtime errors if an object without the expected behaviour is used. We will not discuss the advantages and disadvantages of this approach in this exercise.

   Now that we have a connected socket, we can send some data to the other end of that connection. Issue the command:

```
>>> s.sendall("Egg, bacon, sausage and spam\n")
```

   Note you will get a TypeError, saying that `str` does not support the buffer interface. This is an example of duck typing. A normal string in Python 3 is of type `str`, and is a sequence of unicode code points making up the string[1]. The function `socket.sendall`, however, expects a type that can behave as a buffer, and `str` cannot. We must convert the string to something that can be passed into sendall. We do this by encoding the string into a character set, which will result in a `bytes` object. We will use the UTF-8 character set. UTF-8 is the most common nowadays, and has the advantage that it is backwards compatible with the original 127-character ASCII set. Issue the following command:

```
>>> buf = "Egg, bacon, sausage and spam\n".encode("utf-8")
```

   The string `"Egg, bacon, sausage and spam\n"` is a string literal, but is also of the type str. Therefore, we can execute the function `str.encode` directly on the literal, resulting in the encoded bytestring we assign to buf.

[1] Text encoding has been a headache for computing for a long time. Not every character in every language used fits in the basic 128-character set of ASCII. When the top 128 characters became available, a lot of different (incompatible) encodings popped up defining different characters for the top 128 values. Unicode is both the best thing ever happening to digitized text and a giant headache for programmers. We have a huge amount of space to accomodate written characters both in- and outside the classic ASCII range, but creating programs that manipulate basic text has become a lot harder to do correctly. We will try to avoid these problems as much as we can. What is important to understand is that an encoded string is a sequence of bytes which can be interpreted as a sequence of unicode “codepoints”, which in turn can become readable text. A codepoint as such does not have a defined representation in bytes, and thus cannot be sent over the network without encoding it first. The UTF-8 encoding has become the de facto preferred encoding.

Now issue

```
>>> s.sendall(buf)
```

You should now see the string `Egg, bacon, sausage and spam` appear in the netcat process. Let's clean up the connection correctly:

```
>>> s.close()
```

You will see that the netcat process terminates. After all, its communication partner has decided to close the connection so there is no reason to stay around - to open a new connection, you will have to start a new netcat process. Close the Python shell with ˆd (ctrl + d).

Let's put all this together in a script that we can execute. Create a new directory somewhere called `exercise1`. Inside this directory, create a new file called `netcat.py`. Open it in an editor.

At the top of the file, write the shebang-line `#!/usr/bin/env python3`. This tells the shell with which program to execute this file. Skip a line, then write the `import socket` statement. Skip another line, then just write the commands used above, skipping the ones that caused an error.

Note that Python scope is determined by whitespace, not by curly brackets ({ }) as you have seen in C(++) and Java. Therefore, do not use any leading tabs or spaces when writing this script.

Now start a new netcat process in a different terminal (`nc -l -p 42424`), and either execute the script by issuing `python3 netcat.py`, or by making it executable (`chmod +x netcat.py`) and executing it directly (`./netcat.py`). The behaviour should be the same as when you entered the commands in the interactive shell.

(b) Now let's build the other side of the process, in a somewhat nicer fashion. In the same directory, create a file called `netcat-l.py`. Open this file in an editor, and add the same shebang-line as before. Also make sure to `import socket`.

Then write the following code (omitting the line numbers, they are only for guidance):

```
host = "localhost"
port = 42424
backlog = 5
size = 1024
```

This sets some variables we want to use later. Next, write:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

This defines `s` as a TCP/IP socket. However, it is not yet doing anything. Let's tell it to listen for incoming connections:

```
s.bind((host, port))
s.listen(backlog)
```

This binds the socket to the address belonging to the hostname "localhost", as determined by resolving "localhost" (which will usually just be in the file `/etc/hosts` as 127.0.0.1), and the port 42424, and tells it to start listening for incoming connections.

But this does not do anything yet, because any connection made is not being handled. When a connection comes in, we must `accept` it, and handle it using the socket that `accept` provides:

```
conn, clientaddress = s.accept()
```

Here we see another Python construction you may not be familiar with: multiple return values. It is actually just a single return value: a tuple with a new (connected) socket and the address of the connecting client. This is immediately unpacked and the parts are assigned to `conn` and `clientaddress`.

We now have an active connection to handle. Let's receive some data.

```
data = conn.recv(size)
```

We receive a maximum of 1024 bytes of data, and get a `bytes` object called `data`. Since the socket is a TCP/IP socket, all the headers around the data are handled for us and we only get the payload.

Now let's decide what to do with those bytes. They can mean anything, but let's assume they are a UTF-8 encoded string.

```python
if data:
    datastring = data.decode("utf-8")
    print(datastring)
```

This if-statement checks to see if `data` is non-empty. A buffer-like object, in Python, will evaluate to `True` if it contains anything, and to `False` if it is empty. This is also useful in for-loops, as we will see later.

Then, we decode the data into a unicode string. This happens within the scope of the if-statement. As mentioned earlier, Python scope is determined by whitespace. By convention, four spaces are used for one level of scoping. *Never* mix tabs and spaces in a Python program. Finally, we print the string to the standard output.

Let's close the client connection and listening socket.

```python
conn.close()
s.close()
```

Note that we drop out of the scope for the if-statement, we don't care whether we received any data before closing the connection.

Save the file and run it. Then run `netcat.py` from the previous exercise. You should see the expected string appear, then both processes end.

(c) In its current form, the program is a bit limited. It can only accept a single connection, and it will only accept one message with a maximum size of 1024 bytes. To fix this, we need loops.

Let's first fix `netcat.py` so that it actually sends more than 1024 bytes. Now of course we are not going to type in these bytes by hand, we'll use a loop.

Recall that netcat.py currently looks like this:

```python
#!/usr/bin/env python3

import socket

s = socket.create_connection(("localhost", 42424))
buf = "Egg, bacon, sausage and spam\n".encode("utf-8")
s.sendall(buf)
s.close()
```

Now let's build a loop which repeats the word "spam" 1000 times, and includes the number of repetitions:

```python
stringbuf = ""
for i in range(0, 1000):
    stringbuf = stringbuf + "spam " + str(i) + "\n"
buf = stringbuf.encode("utf-8")
```

Of course, take out the other assignment to `buf`, and feel free to change the program so that it uses variables instead of hardcoded constants for things like the port number and address.

Now run both `netcat-l.py` and `netcat.py`. You'll notice that after the 125th repetition, the program stops. This is the byte limitation in action. Notice that neither program stops with an error: as far as `netcat-l.py` is concerned, it only needs to read 1024 bytes and doesn't care if there's any received data remaining to be read, and `netcat.py` has successfully sent all its data and doesn't know that the program at the other end is silently discarding it. This highlights the importance of protocols.

So let's also add a loop to `netcat-l.py`:

```python
conn, clientaddress = s.accept()
data = b""
newdata = conn.recv(size)
while newdata:
    data += newdata
    newdata = conn.recv(size)
if data:
    datastring = data.decode("utf-8")
    print(datastring)
conn.close()
```

First, we create an empty `bytes` object called data. Then we receive the first block of data, with a maximum size of 1024. This while-loop will check whether any new data has been received, and if so, append it to the current databuffer and wait for more data until the other side closes the connection, or some other error occurs.

Run the programs, and check whether they now perform as you would expect.

So this solves the size limitation, now let's make it so the program can handle multiple connections in sequence. Once again, we will need a loop. Let's say we want it to handle 3 connections before stopping. Here is one of the ways to write a for-loop to do that:

```python
s.listen(backlog)
for i in [1, 2, 3]:
    conn, clientaddress = s.accept()
    data = b""
    newdata = conn.recv(size)
    while newdata:
        data += newdata
        newdata = conn.recv(size)
    if data:
        datastring = data.decode("utf-8")
        print(datastring)
    conn.close()
s.close()
```

Using `for i in range(0, 3)` would be equivalent (and, arguably, better, especially when dealing with a large number of iterations, because `range` (and in Python 2, `xrange`) does not build the entire list of numbers in memory before starting iteration).

You should now be able to run `netcat-l.py`, and then run `netcat.py` 3 times before `netcat-l.py` will terminate.

(d) The nested loop does not look very elegant. And maybe we want to reuse this handling of the connection in another piece of the code. So let's make that a function:

```python
#!/usr/bin/env python3

import socket

host = "localhost"
port = 42424
backlog = 5
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(backlog)
for i in [1, 2, 3]:
    conn, clientaddress = s.accept()
```

```
14     handle(conn)
15 s.close()
16
17
18 def handle(passedconn):
19     data = b""
20     newdata = passedconn.recv(size)
21     while newdata:
22         data += newdata
23         newdata = passedconn.recv(size)
24     if data:
25         datastring = data.decode("utf-8")
26         print(datastring)
27     passedconn.close()
```

As you can see, defining functions is easy. `netcat-l.py` will start as normal. However, when you run `netcat.py`, you will get the error

```
Traceback (most recent call last):
  File "./netcat-l.py", line 15, in <module>
    handle(conn)
NameError: name 'handle' is not defined
```

Since Python is parsed top-down, when the code at line 14 gets executed, the function has not been defined yet. So put the function on line 5, before all the other code. If you run it then, the program should once again behave as expected.

Note that the name of the variable in `def handle(passedconn):` could also just have been `conn`.

There is one convention on Python programs that makes it easier to reuse code. As `netcat-l.py` is now, any time that file is loaded by the Python interpreter the code on lines 17–27 gets run. Maybe we want to load the file to reuse some functions in there, or make a module-file which can also be run directly for testing purposes. Because of this, a convention has formed to put all code which is only necessary if the file is run as a program in a function, e.g. `def main():`, and then call that function only if the file is run as a program:

```
17
18 def main():
19     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
20     s.bind((host, port))
21     s.listen(backlog)
22     for i in [1, 2, 3]:
23         conn, clientaddress = s.accept()
24         handle(conn)
25     s.close()
26
27
28 host = "localhost"
29 port = 42424
30 backlog = 5
31 size = 1024
32
33 if __name__ == "__main__":
34     main()
```

(e) Finally, let's say we don't really like spam. We want to filter out the spam, and only print the numbers. How do we get stuff without spam?

There is of course more than one way to do it, but we're using Python, not Perl, so there is one obvious way to do it. Since most of you are Dutch, it might even obvious at first glance[2].

[2] see PEP 20, "The Zen of Python", http://legacy.python.org/dev/peps/pep-0020/

The obvious way to filter out the spam is by using the function `str.replace` (https://docs.python.org/3.4/library/stdtypes.html#string-methods) on the datastring after decoding (line 13 of `netcat-l.py`) and passing it "spam " as the old and "" as the new string. Note that strings are immutable, so `replace` does not modify the original string, rather, it returns a new string. Do this and test that it works.

However, this is easy and does not teach you anything about lists and list slicing, so we will also do it by transforming the string into a list of strings, then removing the first five characters of each string, then printing them in a loop.

First, change line 14 of netcat-l.py:

```
12      if data:
13          datastring = data.decode("utf-8")
14          print(handlestring(datastring, len("spam "), "\n"))
```

Next, define a new function below `handle` called `handlestring`. It should take three arguments: the string to handle, the number of characters to cut off at the start of each substring, and the character to use to split the string into substrings. An example of how this function might be implemented:

```
18  def handlestring(datastring, length, delimiter):
19      stringlist = datastring.split(sep=delimiter)
20      filteredlist = []
21      for string in stringlist:
22          filteredlist.append(string[length:])
23      filteredstring = delimiter.join(filteredlist)
24      return filteredstring
```

Let's go through this line by line. Line 19 splits the string using the function `str.split` using the delimiter as named argument. When a function takes multiple arguments, and some of them have defaults, then the way to specify overrides for only some of the arguments is by providing them with their names.

On line 20, we create a new empty list to take the filtered strings.

Lines 21 and 22 loop over all the strings in `stringlist`, which are now of the form "spam 152". Notice they do not contain the newline, since that has been removed in the split. On line 22, we first slice the string we're handling. The syntax for slicing is explained at https://docs.python.org/3.4/tutorial/introduction.html#strings. Slicing works on any sequence type, so it works on e.g. `str`, `bytes`, `list`, and `tuple`. Here, we start our slice at the sixth character (index 5) and take everything after that. Then we append the resulting string to `filteredlist`.

On line 23 we then join the strings in `filteredlist` together, putting the delimiter back between each string. Had we wanted to use a different delimiter, e.g. a dash, the call would have been `"-".join(filteredlist)`.

Finally, on line 24, we return the resulting string.

Try this and see whether it works. Feel free to come up with your own solution.

2. As mentioned in the lecture, using TCP for this message passing is overkill. Copy the folder `exercise1` and all its contents to the folder `exercise2`. Then, in this new folder, change `netcat.py` and `netcat-l.py` to use UDP. You will have to consult the documentation for the socket library (https://docs.python.org/3.4/library/socket.html#socket.socket.recv) to understand what functions to use. Some hints:

    - UDP is the User *Datagram* Protocol.
    - UDP sockets do not have connections. For the sender, this means no connection is made, data is just sent to the location where it should end up. Checking to see whether all data has been sent is a good idea.
    - For the listener, the lack of connections means not listening for them, not accepting them, and not using a different socket for handling them. You do still have to bind, though.

- Receiving UDP data reads an entire packet of UDP data, regardless of the buffer size you use. Any additional data is *discarded*. There are ways to detect this but right now the best solution is to just use the theoretic maximum size of a UDP data packet.
- You can also not wait to handle the string until the received data is empty, since there is no connection to signal that there will be no more data. Since all the characters we are sending fit in one byte in UTF-8, you can safely just decode everything you receive immediately.

Your program does not have to behave exactly as the TCP version, e.g. you do not have to automatically stop after receiving 3 messages.

It is acceptable, even expected, that to build the solution for this exercise you will use other Internet resources than the Python documentation. Feel free to search. However, do *not* post the literal question to e.g. stackoverflow, this is considered homework fraud, and is also frowned upon by the relevant communities. If you cannot figure it out using available resources, skip this exercise for now and send an e-mail to Pol about it.

3. For the final two exercises, you will build your own network sniffer. Create a folder `exercise3` and create a file `sniffer.py`. Build the basic structure of a program in this file, with the shebang line, the socket import, the `main` function and the corresponding program name check.

   (a) Now, in the `main` function, create a socket with the following options: family is `AF_INET`, type is `SOCK_RAW`, and protocol is `IPPROTO_UDP`. Then write an endless loop in which you use the `recvfrom` function to receive packets with the maximum buffer size 65565, and print them directly to stdout using something like `print(s.recvfrom(65565))`.

   Make the script executable, and execute it with root rights. It needs root rights because we are using raw sockets. The output you will see should be something like

```
(b'E\x02\x00\x8e\xe7\xc7\x00...', ('192.0.2.5', 0))
(b'E\x02\x00\x8b\xe7\xc8\x00...', ('192.0.2.5', 0))
...
```

   These are dumps of the packets as `bytes` objects shown in hexadecimal notation, as well as their source address / port combination. These are all the UDP/IP packets being sent to your machine.

   So let's build a function to parse and print the UDP header. The UDP protocol is, as are almost all Internet protocols, defined in a Request For Comments (RFC): RFC 798, which can be found at https://www.ietf.org/rfc/rfc768.txt. According to this document, the UDP header looks like this:

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
```

   So that's 16 bits source port, 16 bits destination port, 16 bits length and 16 bits checksum, for a total of 64 bits (8 bytes).

   So let's slice this header from the UDP packet:

```python
def parse_udp(packet):
    header_start = 0
    header_length = 8
    header = packet[header_start:header_start + header_length]
```

   Now there are multiple ways to translate these 8 bytes back to meaningful data, but in Python the way to do this is to use the `struct` module, which performs conversions between Python values and C-like byte structures represented as Python `bytes` objects. The function to use is `struct.unpack`, which takes a format string as defined by https://docs.python.org/3.4/library/struct.html#format-strings. Add an import for struct on line 4, and add this to `parse_udp`:

```
    (source_port, dest_port,
     data_length, checksum) = struct.unpack("!HHHH", header)
```

The format string specifies the struct is in network byte order ("!"), then specifies that there are four unsigned short integers to read. The `unpack` function translates these and returns a four-tuple with these values.

If this causes a `TypeError: 'tuple' does not support the buffer interface`, you should check the documentation for the `socket.recvfrom` function and make sure that you handle its return values correctly.

Now let's print them:

```
    print("Source Port: {}\nDestination Port: {}\n"
          "Data length: {}\nChecksum: {}\n".format(
              source_port, dest_port, data_length, checksum))
```

For documentation on how this string formatting works, see https://docs.python.org/3.4/library/string.html#format-examples and its surrounding documentation.

Finally, in the `main` function, instead of printing the packet, call the `parse_udp` function.

(b) However, if you now run this code you will notice that the numbers don't really make sense. E.g. the source port for every packet is the same, the destination port keeps changing, and the data length keeps incrementing by 1 for each packet.

Turns out that you do not only get the UDP header, but the entire packet including IP header. So we need to take this into account when finding the UDP header. The Internet Protocol is defined in RFC 791: https://www.ietf.org/rfc/rfc791.txt. According to this document, the IP header looks like this:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The IHL field contains the Internet Header Length in number of 32-bit words, and the minimum length is 5. So the minimum number of bits for a header is 160, or 20 bytes. To know where to find the UDP header we need to at least parse the IHL field. Notice that this field is a 4-bit field embedded in an 8-bit field, so we will need to fiddle with some bits to get the correct value. The struct-functions only work on whole bytes.

```
def parse_ip(packet):
    header_length_in_bytes = (packet[0] & 0x0F) * 4
    header = packet[:header_length_in_bytes]
    data = packet[header_length_in_bytes:]
    return header_length_in_bytes, header, data
```

Now there are two ways of dealing with this nesting of packets. We can strip of the outer layer at each parsing function, returning all the required information in tuples; or we can pass an offset to each parsing function telling it where to find its data. In this case I have chosen the former, but there is an equally valid argument for doing the latter. Feel free to choose another way of doing this.

Since at the moment we don't need much information from the IP header I return the header length I parsed, the unparsed header, and the data. To keep this consistent throughout the program, change the `parse_udp` function to this:

```
def parse_udp(packet):
    header_length = 8
    header = packet[:header_length]
    data = packet[header_length:]
    (source_port, dest_port,
     data_length, checksum) = struct.unpack("!HHHH", header)
    return source_port, dest_port, data_length, checksum, data
```

Amend the `main` function so that you first call the `parse_ip` function on the packet, then the `parse_udp` function on the returned data. Print all the data you receive from `parse_udp` in a similar fashion as before, but now include the data as well. You can decide for yourself how to print the data, whether as a single string or as a pretty printed hexdump.

(c) The size of `data` returned by `parse_udp` is not the same as the value for `data_length`. Take a look at the RFC for UDP and explain why, in a file called `udp-length-mismatch-explanation`.

(d) Using the definition of the IP header as explained above, and the documentation of the `struct` module (https://docs.python.org/3.4/library/struct.html#module-struct), amend the function `parse_ip` so that it also parses and returns the Total Length, Protocol, Source Address and Destination Address fields, instead of the unparsed header.

For the parsing of these fields you can either slice them out of the header before parsing, or build a format string with pad bytes for the bytes you are not interested in. If you take the latter approach, beware that the function `struct.unpack` requires the header to be the exact size as specified by the format string, and since an IP header is of variable size, you must use `struct.unpack_from`. You may assume the IP header is at least 20 bytes long, since the RFC states that that is the minimum size.

Note that since IP addresses should be written in dotted decimal notation, you may want to search the `socket` documentation for a way to easily convert them between a 32 bit integer representation and dotted decimal notation. For this you will have to parse them as `bytes` objects and then convert them to strings. Doing this conversion for 4 bytes requires you to specify "4s" in the format string at that location.

Also amend the main function so that it prints these values.

4. Copy the folder `exercise3` and all its contents to `exercise4`. Continue working in this folder. Edit `sniffer.py`.

(a) We want to go one level lower, and get everything which is delivered through ethernet. So that includes outgoing packets, and other protocols like TCP and ICMP. For this to happen, we need to change the socket. Currently, the socket is created with these options: family is `AF_INET`, type is `SOCK_RAW`, and protocol is `IPPROTO_UDP`. These need to be changed: family becomes `AF_PACKET`, type remains `SOCK_RAW`, and protocol becomes `socket.ntohs(0x0003)`.

However, when you test this you will quickly realize that your program no longer functions, because now you also get packets for protocols other than UDP, and you get the entire ethernet frame header as well.

An ethernet frame header looks like this:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Ethernet destination address (first 32 bits)           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Ethernet dest (last 16 bits)  |Ethernet source (first 16 bits)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Ethernet source address (last 32 bits)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      802.1Q tag (optional)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Type code              |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

If the optional 802.1Q tag is not present, the type code immediately follows the ethernet source address, at bytes 13 and 14. If, however, this location holds the value 0x8100, an 802.1Q tag is present. The type code should then be read after this 4-byte tag, at bytes 17 and 18.

A type code of 0x0800 specifies that the Ethernet frame contains Internet Protocol data. There are numerous type codes, a partial overview can be found on wikipedia: http://en.wikipedia.org/wiki/EtherType. E.g. a type code of 0x0806 specifies ARP.

Note that there is no padding after the type code.

Create a function `parse_ethernet` which parses the ethernet destination address, source address, and type code. The type code should be checked for the 802.1Q specifier, and if it is present the function should retrieve the type code at the alternative location.

(b) Now, since you have changed the socket definition, your calls to `recvfrom` will provide you with the entire ethernet frame. Amend the `main` function to parse it using the function you just created.

Then, after parsing it, check whether the type code is equal to the IP type code, 0x0800. If it is, you can continue parsing the ethernet payload as an IP packet, otherwise, discard the payload and parse the next frame (the Python keyword to continue with the next iteration of a loop is simply `continue`).

Finally, after parsing the IP header, you can check whether the packet is a UDP packet or not. The IP protocol number for UDP is 17. If the protocol number is 17, parse the packet as UDP. Finally, print all the information you have gathered: ethernet source, ethernet destination (these are MAC addresses), IP header length and IP total length, IP protocol, IP source and destination addresses, UDP source port and destination port, UDP payload length, and finally the actual UDP data.

It might be helpful for debugging to print a message whenever you encounter a frame or IP packet with a different protocol number.

(c) As a completely optional extra, at this point it is also interesting to look at the `source` returnvalue from `socket.recvfrom` again, since it will now tell you several new, useful things about the ethernet frames you receive. See if you can figure out (using comparison or by reading the documentation) what this information is.

(d) As a completely optional extra, you may create parsing functions for other IP-borne protocols such as TCP (http://www.ietf.org/rfc/rfc793.txt) or ICMP (http://www.ietf.org/rfc/rfc792), or other ethernet-borne protocols such as ARP (http://www.ietf.org/rfc/rfc826). Just make sure to select them in the correct way when parsing.

5. Place the directories `exercise1`, `exercise2`, `exercise3`, and `exercise4` and all their contents in a directory called `netsec-assignment1-STUDENTNUMBER1-STUDENTNUMBER2` (replace `STUDENTNUMBER1` and `STUDENTNUMBER2` by your respective student numbers, and accomodate for extra / fewer student numbers). Make a `tar.gz` archive of the whole `netsec-assignment1-STUDENTNUMBER1-STUDENTNUMBER2` directory and submit this archive in Blackboard.