# Timing Covert Channel Lab

2025-03-28

## Table of contents



In the last lecture, you covered different ways that someone might set up and take advantage of a Covert Channel. In the last lab, you set up one such Storage covert channel using the file permissions of an FTP server.

In this lab, we shall set up a Timing Covert Channel. We will set up a server that will transmit text overtly, but transmit a covert message hidden in the timings between each successive character of the overt message.

As an example, a delay of 0.025s between parts of the message could be translated to a 0 while a delay of 0.1s can be translated as a 1.

In order to transfer a long covert message, we will need a much longer overt message. Using the suggestion above, transmitting a single covert character (7 or 8 bits) would require the transmission of 8 or 9 characters. As mentioned earlier, this isn't too unreasonable because Covert channels are typically low bandwidth (i.e. can only realistically send small messages).

To end the message we have a couple of options to consider. We could just repeat the covert message over and over and trust the recipient to know when they have received the entire message. However, this would make our channel easy to notice and analyse. A more surreptitious option would be to add a special tag at the end of our covert message to signify that the hidden message has been received in its entirety…perhaps the string "EOF"

# 1 Setting up the server

## 1.1 Sending a basic message

Let's create a basic TCP server using python's socket library.

```
import socket

# define a host and a port
HOST = "127.0.0.1"  # This is the ip address for a local host
PORT = 1337     # Choose a port number that is not typically
            # used by other traffic.

# Now to create a Socket object and connect it to the specific host and
# port identified by our constants above

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s. bind((HOST, PORT))
s.listen(0) # the argument determines how many connection requests
        # will be added to the queue. 0 means that it will not
        # add any new request to the queue and will reject any
        # incoming requests if it is already dealing with one.

print(f"Server is listening on {HOST}:{PORT}")

while True:
    # Note that if you wanted your server to be able to deal with more
    # than one client at a time, this portion would be threaded. For our
    # demonstration, you won't need to create a threaded server.

    c, addr = s.accept()    # store client details when connection
                # comes in. "c" is the client socket,
                # and "addr" is the client address.

    print(f"Connection from {addr} established")

    # the b in the sendall command below ensures that the message is
    # sent as bytes, which is the form that sendall requires.
    c.sendall(b"Hello from the other side. I must have called a thousand times")

    # make sure to close the connection
```

```
    c.close()
```

> **Tip**
>
> If you haven't done any socket programming before, this is a good place to stop and test your server. Its always fun to see how your code could work over a network.
> Later on in this lab, we'll design a programmatic client that will connect to our server. But you don't need one for now. You can use built-in tools in the terminal that allow you to connect to a client/port.
>
> 1. Using **telnet**
>
> ```
> telnet 127.0.0.1 1337   # general form is telnet server_ip server_port
> ```
>
> Telnet is used for plain text communication and is unencrypted so don't use it for sensitive data.
>
> 2. Using **netcat**
>
> ```
> nc 127.0.0.1 1337
> ```
>
> Hit `Ctrl + C` or type `exit` to exit. Netcat is used to test and debug network services.

The code above is a very simple server example. We need to tweak it to send one character at a time with a delay between each character where the delay is actually our covert channel.

In order to do this, we shall have to make a couple of changes to our code above starting with importing the `sleep` command from the `time` library.

```
from time import sleep
```

To make our code easier to adjust, we should probably put some time constants at the top of our code too.

```
ZERO = 0.025    # How long to wait for a zero
ONE = 0.1   # How long to wait for a one
```

Our algorithm for sending the message is also going to change a little bit. Instead of using `sendall`, we shall use `send` which allows us to send smaller chunks of the message.

Our loop for sending the covert message will look something like this

```
msg = "Some message..."
for i in msg:
    c.send(i)
    if (<zero>):
        sleep(ZERO)
    else:
        sleep(ONE)
c.send("EOF")
c.close()
```

## 1.2 Creating our covert message

We have most of the components to put our entire code together. However, we still have the question of how we know whether to transmit a 0 or 1 for our covert message.

To answer that question, we'll need to assemble our covert message

```
covert = "secret" + "EOF"
```

then convert it to binary

```
from binascii import hexlify

covert = "secret" + "EOF"
covert_bin = ""
for i in covert:
    # convert each character to a full byte
    # hexlify converts ASCII to hex
    # int converts the hex to a decimal integer
    # bin provides its binary representation (with a 0b
    # prefix that must be removed)
    # that's the [2:] (return the string from the third
    # character on)
    # zfill left-pads the bit string with 0s to ensure a
    # full byte
    covert_bin += bin(int(hexlify(i), 16))[2:].zfill(8)
```

As an example, if `covert == "ABC"`, then `covert_bin == "010000010100001001000011"`

So our algorithm from earlier will look something like this.

```
import time

ZERO = 0.025
ONE = 0.1

msg = "Some message..."
n = 0

for i in msg:
    c.send(i)
    if (covert_bin[n] == "0"):
        time.sleep(ZERO)
    else:
        time.sleep(ONE)
    n = (n + 1) % len(covert_bin)
c.send("EOF")
c.close()
```

## 2 Setting up the Client

> **Tip**
>
> Historically, timing has not been super reliable on Windows. We mention this so that you are not shocked if the timings are not what you expect when on Windows. Occassionally, this unreliability has also extended to Linux VMs that are hosted on Windows Host machines.

### 2.1 Receiving a Basic Message

Let's create a basic client using python's socket library

```
import socket

# Define server address and port
HOST = '127.0.0.1'  # Must match the server's address
PORT = 1337         # Must match the server's port

# Create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
try:
    # Connect to the server
    s.connect((HOST, PORT))
    print("Connected to the server.")

    # Receive data from the server
    data = s.recv(1024)  # Receive up to 1024 bytes
    print("Received from server:", data.decode('utf-8'))  # Decode bytes to string

finally:
    s.close()  # Close the connection
```

## 2.2 Receiving many small messages

The code above should be able to log into your server and receive the overt message that is being sent from it. However, it *might* not exhibit the behaviour we want. This is because, depending on your system, there is a chance that this client will wait till a certain amount of information has been received as a chunk before displaying it.

To deal with this, we shall have to tweak our algorithm above to 1. write out what it gets as soon as it gets it. This will require us switching from `print` to `sys.stdout.write` 2. Flush out any information it is storing in the buffer once its written using `sys.stdout.flush` 3. Keep on receiving this information until it gets an "EOF"

```
import sys
...
data = s.recv(4096) # increased the number of bytes
while (data.rstrip("\n") != "EOF"):
    sys.stdout.write(data)
    sys.stdout.flush()
    data = s.recv(4096) # receive the next bit of information
...
s.close()
```

## 2.3 Receiving the covert message

We now need to figure out a way to measure the time between each received message and interprete that as either a 0 or a 1. To do this, we'll need to use the `time` library and take multiple time measurements during our code's execution.

```
from time import time
...
covert_bin = "" # start with an empty string
...
t0 = time() # take the time before you receive anything
data = s.recv(4096)
t1 = time() # take the time after you receive something
delta = round(t1 - t0, 3) # calculate the difference

# and then translate that difference into either a 1 or a 0. Add whatever
# it is to the covert message string
if (delta >= ONE):
    covert_bin += "1"
else:
    covert_bin += "0"
```

## 2.4 Understanding our Covert Message

We should have a string of 1s and 0s that make up our covert message. To understand it, we need to convert it to ASCII.

```
covert = ""
i = 0

while (i < len(covert_bin)):
    # process one byte at a time
    b = covert_bin[i:i + 8]
    # convert it to ASCII
    n = int("0b{}".format(b), 2)

    try:
        covert += unhexlify("{0:x}".format(n))
    # in the event that we either made a mistake in our interpretation
    # of a single bit, or we received an incorect bit, the entire
    # character is shot and we might not be able to find its ASCII
    # interpretation. In that case, just put a "?" so that we know there
    # was an error with this specific character.
    except TypeError:
        covert += "?"
```

```
    # stop at the string "EOF"
    i += 8
```

> **Note**
>
> The entire **client** section of this lab has been intentionally vague. All the parts are there but they are not organized into a comprehensive solution. It will be up to you to figure out how to arrange them in order to get a piece of code that will log into the server, receive the overt message, compile the covert message, and interpret the covert message. *Good luck*