



# Dissecting OpenPLC's Modbus TCP Communications With Wireshark

By [Rodrigo Cantera](#) 23/01/2021 [Attacking the ICS, Virtual Industrial Cybersecurity Lab](#) [0 Comments](#)

The first step in every attack consists of a reconnaissance of the environment in order to identify vulnerabilities and attack vectors. Usually, attackers will use Nmap to discover the devices and open ports in the network, but in this case, we will start with a passive network listening method via Wireshark.

**+** Prerequisites (click to extend)

## Dissecting OpenPLC Communications

### Traffic Capture

In order to capture traffic between OpenPLC and FactoryIO, it is necessary to open a terminal in the attacker machine and turn on Wireshark:

A notepad about Industrial Cybersecurity

## CATEGORIES

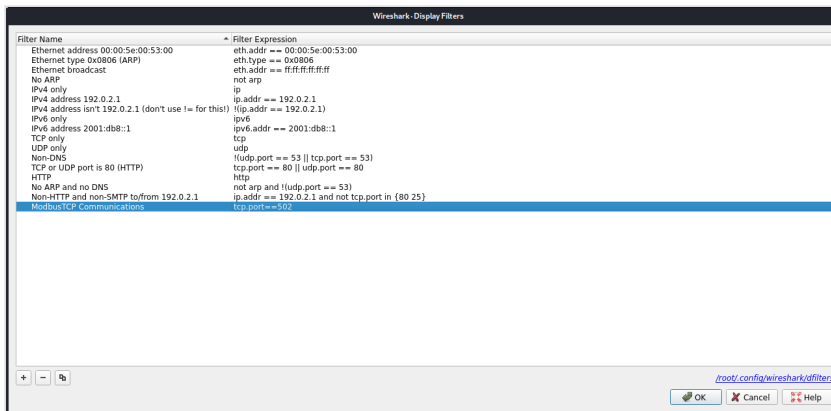
- Virtual Industrial Cybersecurity Lab (8)
  - Attacking the ICS (2)
  - Design and Deployment (6)

## 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 –

```
sudo wireshark
```

Once Wireshark is opened, it is recommended to add a new filter that can be used to improve the data visualization by showing only traffic against port 502 (default port for Modbus TCP).



Modbus TCP filter in Wireshark

Now, we can start listening to the network traffic with Wireshark, turn on the OpenPLC by clicking the "Start PLC" button on the web interface, and you will start to see some traffic being captured. After some seconds, there should be enough data captured in order to analyze OpenPLC's communication loop.

#### Note:

If no traffic is being detected, make sure the network adapter of the attacker machine is [configured properly](#) to allow promiscuous mode.

Part 4: Programming the Control Logic

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



No.	Time	Source	Destination	Protocol	Length	Info
87	7.494287997	192.168.88.201	192.168.88.100	TCP	74	50816 -> 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=4889281191 TSecr=0 WS=128
88	7.494288161	192.168.88.100	192.168.88.201	TCP	66	502 -> 50816 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256
89	7.494288161	192.168.88.201	192.168.88.100	TCP	66	50816 -> 502 [ACK] Seq=1 Ack=1 Win=4256 Len=0
90	7.497687390	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 1; Unit: 1, Func: 2: Read Discrete Inputs
91	7.498695096	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 1; Unit: 1, Func: 2: Read Discrete Inputs
92	7.498695345	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=13 Ack=11 Win=64256 Len=0
93	7.498695506	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 2; Unit: 1, Func: 15: Write Multiple Coils
94	7.500954526	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 2; Unit: 1, Func: 15: Write Multiple Coils
95	7.500786839	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=27 Ack=23 Win=64256 Len=0
96	7.601212539	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 3; Unit: 1, Func: 2: Read Discrete Inputs
97	7.601319648	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 3; Unit: 1, Func: 2: Read Discrete Inputs
98	7.603225607	192.168.88.201	192.168.88.100	Modbus	60	50816 -> 502 [ACK] Seq=39 Ack=33 Win=64256 Len=0
99	7.603225749	192.168.88.201	192.168.88.100	Modbus	68	Query: Trans: 4; Unit: 1, Func: 15: Write Multiple Coils
100	7.605292206	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 4; Unit: 1, Func: 15: Write Multiple Coils
101	7.605292528	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=53 Ack=45 Win=64256 Len=0
102	7.707226503	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 5; Unit: 1, Func: 2: Read Discrete Inputs
103	7.708087117	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 5; Unit: 1, Func: 2: Read Discrete Inputs
104	7.707226503	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=65 Ack=55 Win=64256 Len=0
105	7.707226607	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 6; Unit: 1, Func: 15: Write Multiple Coils
106	7.708961877	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 6; Unit: 1, Func: 15: Write Multiple Coils
107	7.709187385	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=79 Ack=67 Win=64256 Len=0
108	7.809629501	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 7; Unit: 1, Func: 2: Read Discrete Inputs
109	7.811075279	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 7; Unit: 1, Func: 2: Read Discrete Inputs
110	7.811031646	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=91 Ack=77 Win=64256 Len=0
111	7.812095661	192.168.88.201	192.168.88.100	Modbus	68	Query: Trans: 8; Unit: 1, Func: 15: Write Multiple Coils
112	7.813953213	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 8; Unit: 1, Func: 15: Write Multiple Coils
113	7.813795627	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=103 Ack=89 Win=64256 Len=0
114	7.915287687	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 9; Unit: 1, Func: 2: Read Discrete Inputs
115	7.917235661	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 9; Unit: 1, Func: 2: Read Discrete Inputs
116	7.917689809	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=117 Ack=99 Win=64256 Len=0
117	7.917689813	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 10; Unit: 1, Func: 15: Write Multiple Coils
118	7.919655568	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 10; Unit: 1, Func: 15: Write Multiple Coils
119	7.919368689	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=131 Ack=111 Win=64256 Len=0
121	8.013093164	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 11; Unit: 1, Func: 2: Read Discrete Inputs
122	8.012910890	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 11; Unit: 1, Func: 2: Read Discrete Inputs
123	8.022134134	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=143 Ack=121 Win=64256 Len=0
124	8.022134263	192.168.88.201	192.168.88.100	Modbus	68	Query: Trans: 12; Unit: 1, Func: 15: Write Multiple Coils
125	8.026929699	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 12; Unit: 1, Func: 15: Write Multiple Coils
131	8.027226262	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=157 Ack=133 Win=64256 Len=0
132	8.128999106	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 13; Unit: 1, Func: 2: Read Discrete Inputs
133	8.128915477	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 13; Unit: 1, Func: 2: Read Discrete Inputs
134	8.129157906	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=169 Ack=143 Win=64256 Len=0
135	8.129159020	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 14; Unit: 1, Func: 15: Write Multiple Coils
136	8.130997279	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 14; Unit: 1, Func: 15: Write Multiple Coils
137	8.172893637	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=183 Ack=155 Win=64256 Len=0
143	8.221762523	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 15; Unit: 1, Func: 2: Read Discrete Inputs
144	8.222592392	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 15; Unit: 1, Func: 2: Read Discrete Inputs

## Network traffic capture with Wireshark

Stop the data capture and save it somewhere, so it can be accessed later on if needed.

## Traffic Analysis

Now let's start analyzing the traffic captured in order to understand the communication loop used by OpenPLC. This information can be very relevant for an attacker as it may point out some vulnerabilities or attack vectors to exploit.

## Analyzing the Communication Loop

No.	Time	Source	Destination	Protocol	Length	Info
87	7.494287997	192.168.88.201	192.168.88.100	TCP	74	50816 -> 502 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=4889281191 TSecr=0 WS=128
88	7.494288161	192.168.88.100	192.168.88.201	TCP	66	502 -> 50816 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256
89	7.494288161	192.168.88.201	192.168.88.100	TCP	66	50816 -> 502 [ACK] Seq=1 Ack=1 Win=4256 Len=0
90	7.497687390	192.168.88.201	192.168.88.100	Modbus/TCP	66	Query: Trans: 1; Unit: 1, Func: 2: Read Discrete Inputs
91	7.498695096	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 1; Unit: 1, Func: 2: Read Discrete Inputs
92	7.498695345	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=13 Ack=11 Win=64256 Len=0
93	7.498695506	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 2; Unit: 1, Func: 15: Write Multiple Coils
94	7.500954526	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 2; Unit: 1, Func: 15: Write Multiple Coils
95	7.500786839	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=27 Ack=23 Win=64256 Len=0
96	7.601212539	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 3; Unit: 1, Func: 2: Read Discrete Inputs
97	7.601319648	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 3; Unit: 1, Func: 2: Read Discrete Inputs
98	7.603225607	192.168.88.201	192.168.88.100	Modbus	60	50816 -> 502 [ACK] Seq=39 Ack=33 Win=64256 Len=0
99	7.603225749	192.168.88.201	192.168.88.100	Modbus/TCP	68	Query: Trans: 4; Unit: 1, Func: 15: Write Multiple Coils
100	7.605292206	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 4; Unit: 1, Func: 15: Write Multiple Coils
101	7.605292528	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=53 Ack=45 Win=64256 Len=0
102	7.707226503	192.168.88.201	192.168.88.100	Modbus/TCP	66	Query: Trans: 5; Unit: 1, Func: 2: Read Discrete Inputs
103	7.708087117	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 5; Unit: 1, Func: 2: Read Discrete Inputs
104	7.707226503	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=65 Ack=55 Win=64256 Len=0
105	7.707226607	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 6; Unit: 1, Func: 15: Write Multiple Coils
106	7.708961877	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 6; Unit: 1, Func: 15: Write Multiple Coils
107	7.709187385	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=79 Ack=67 Win=64256 Len=0
108	7.809629501	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 7; Unit: 1, Func: 2: Read Discrete Inputs
109	7.811075279	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 7; Unit: 1, Func: 2: Read Discrete Inputs
110	7.811031646	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=91 Ack=77 Win=64256 Len=0
111	7.812095661	192.168.88.201	192.168.88.100	Modbus	68	Query: Trans: 8; Unit: 1, Func: 15: Write Multiple Coils
112	7.813953213	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 8; Unit: 1, Func: 15: Write Multiple Coils
113	7.813795627	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=103 Ack=89 Win=64256 Len=0
114	7.915287687	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 9; Unit: 1, Func: 2: Read Discrete Inputs
115	7.917235661	192.168.88.100	192.168.88.201	Modbus/TCP	64	Response: Trans: 9; Unit: 1, Func: 2: Read Discrete Inputs
116	7.917689809	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=117 Ack=99 Win=64256 Len=0
117	7.917689813	192.168.88.201	192.168.88.100	Modbus	66	Query: Trans: 10; Unit: 1, Func: 15: Write Multiple Coils
118	7.919655568	192.168.88.100	192.168.88.201	Modbus/TCP	66	Response: Trans: 10; Unit: 1, Func: 15: Write Multiple Coils
119	7.919368689	192.168.88.201	192.168.88.100	TCP	60	50816 -> 502 [ACK] Seq=131 Ack=111 Win=64256 Len=0

## OpenPLC's communication loop

By direct observation of the traffic capture, one can see that:

The first communication between OpenPLC and FactoryIO is a [3-way handshake](#):

- OpenPLC -> FactoryIO: SYN
- FactoryIO -> OpenPLC: SYN/ACK
- OpenPLC -> FactoryIO: ACK

From that point, there is a communication loop that repeats indefinitely using a [typical Modbus TCP](#) query-response structure:

- Read Discrete Inputs
  - **Query** OpenPLC -> FactoryIO: query the status of the sensors
  - **Response** FactoryIO -> OpenPLC: responds with the status of the sensors
  - **ACK** OpenPLC -> FactoryIO: confirms that the response was received correctly
- Write Multiple Coils
  - **Query** OpenPLC -> FactoryIO: asks to modify the status of the actuators
  - **Response** FactoryIO -> OpenPLC: responds that the actuators were modified correctly
  - **ACK** OpenPLC -> FactoryIO: confirms that the response was received correctly

Note:

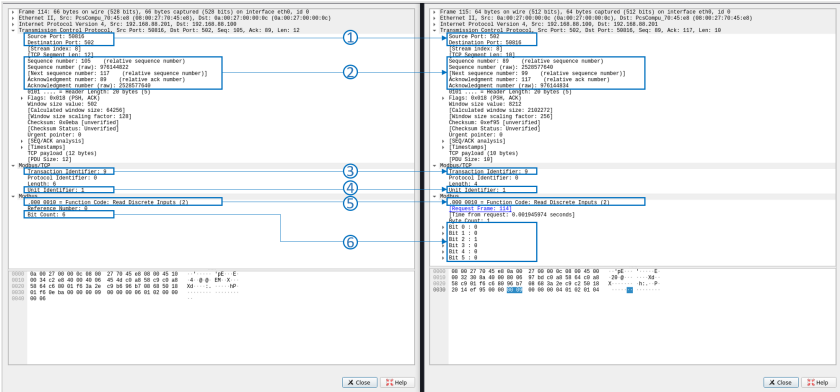
The time between communication loops should approximately match the polling value configured under the "Settings" tab in OpenPLC (100ms in this case).

If the control logic for the PLC would use registers (instead of only discrete inputs and coils) it should be expected to find also some query-response pairs to check and update the registers.

# Analyzing Modbus Query-Response Messages

With the general communication logic figured out, it is time to step down a level and look at each individual Modbus TCP message in order to understand its structure:

## Read Inputs



Request (left) and response (right) for Read Inputs function

1. Analyzing the TCP layer, it is possible to identify the ports used by both devices in the communications. It is interesting to observe that Factorylo uses the defined 502 port while OpenPLC is using a random one
2. Between the Modbus query and the response, there is no ACK. They are sequential
3. Modbus TCP uses the same Transaction ID for both the query and the response
4. The message intended and responded by the RTU with the ID 1 (in this case there is only 1 RTU, but in real environments, there may be more of them)
5. The function used is [number 2](#) (000 0010 in binary), which matches standard Modbus function codes
6. The query requests to read 6 inputs (Bit Count) starting at 0 (Reference Number). The reason for this is that both the [Driver in FactoryIO](#), and [OpenPLC's slave device](#) are configured to have a total of 6

inputs. The response to the query shows the state of each input (1=on, 0=off), but it is not possible to know to which sensor is assigned (safety door, box detection, etc.)

Write Coils

The image displays a Wireshark packet capture of a Modbus TCP communication. The left pane shows a 'Request' packet (1) and the right pane shows a 'Response' packet (2). Both packets are of type 'Modbus TCP' and are part of a sequence. The packets are captured on the 'eth0' interface. The left pane shows the 'Request' packet (1) and the right pane shows the 'Response' packet (2). The packets are captured on the 'eth0' interface. The left pane shows the 'Request' packet (1) and the right pane shows the 'Response' packet (2). The packets are captured on the 'eth0' interface.

Request (left) and response (right) for Write Coils function

1. Analyzing the TCP layer, it is possible to identify the ports used by both devices in the communications. It is interesting to observe that Factorylo uses the defined 502 port while OpenPLC is using a random one
2. Between the Modbus query and the response, there is no ACK. They are sequential
3. Modbus TCP uses the same Transaction ID for both the query and the response
4. The message is intended to and responded by the RTU with the ID 1 (in this case there is only 1 RTU, but in real environments, there may be more of them)
5. The function used is **number 15** (000 0111 in binary), which matches standard Mobus function codes
6. The query requests to write 5 coils (Bit Count) starting at 0 (Reference Number). The reason for this is that both the **Driver in FactoryIO**, and **OpenPLC's slave device** are configured to have a total of 5 coils (outputs)
7. The requested status for the coils will be 00. There are other values found for this data on different

frames of the capture (14, 16, 0d, etc.) which signals the usage of a hex to binary translation in order to reflect the target status for the actuators.

```

Frame 3194: 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: 50816, Dst Port: 502, Seq: 11323, Ack: 9581, Len: 14
  Modbus/TCP
  Modbus
    .000 1111 = Function Code: Write Multiple Coils (15)
    Reference Number: 0
    Bit Count: 5
    Byte Count: 1
    Data: 0d
Frame 3195: 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: 50816, Dst Port: 502, Seq: 11167, Ack: 9449, Len: 14
  Modbus/TCP
  Modbus
    .000 1111 = Function Code: Write Multiple Coils (15)
    Reference Number: 0
    Bit Count: 5
    Byte Count: 1
    Data: 14
Frame 3196: 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: 50816, Dst Port: 502, Seq: 11115, Ack: 9405, Len: 14
  Modbus/TCP
  Modbus
    .000 1111 = Function Code: Write Multiple Coils (15)
    Reference Number: 0
    Bit Count: 5
    Byte Count: 1
    Data: 16

```

Examples of several data values found in the Write Coils query

Hex	Bin (5bits)	QX100.0	QX100.1	QX100.2	QX100.3	QX100.4
00	00000	OFF	OFF	OFF	OFF	OFF
14	10100	ON	OFF	ON	OFF	OFF
16	01110	ON	OFF	ON	ON	OFF
0d	01101	OFF	ON	ON	OFF	ON

## Write Coils function data translation from Hex to RTU outputs

## Observation Summary

After al the analysis, it is possible to make the following observations:

- The connection between OpenPLC and FactoryIO starts with a 3-way handshake
- Messages use the common TCP protocol ACK Number and Sequence Number
- Each Modbus response triggers an ACK from the destination system

- It is possible to know the destination RTU of a message by checking the "Unit Identifier" of the Modbus TCP payload
- Modbus messages use a sequential number (Transaction Identifier) that increase with the queries but not with the responses
- It is possible to know the intent of a query/response because OpenPLC uses standard Modbus functions
- It is possible to know the current state of the sensors by analyzing the payload of the Read Inputs response
- It is possible to know the target state of the actuators by analyzing the payload of the Write Coils query
- The Write Coils uses Hex to Binary translation to reflect the target status of the outputs
- It is not possible to know which sensor or actuator is associated with a certain value just by analyzing the payload
- All of this is possible because ModbusTCP is not an encrypted protocol

All of this information may appear harmless now but will be very valuable later on in order to decide on an effective attack vector.

