Report for the CPS Final Project

Andrea Auletta
andrea.auletta@studenti.unipd.it

Niccolò Zenaro
niccolo.zenaro@studenti.unipd.it

Reference paper:
Attack taxonomies for the Modbus protocols [1]

February 16, 2025

Contents

Abstract

Modbus is a commonly used protocol in Supervisory Control And Data Acquisition (*SCADA*) environments for monitoring, control and data acquisition. Despite its wide popularity, Modbus is not secure because when it was developed and adopted (1979) security was not considered to be a concern in isolated Industrial Control Systems (*ICS*), thus is not designed to be secure like modern IT networks. Among the various attacks, 4 different taxonomies can be identified to facilitate formal risk analysis efforts for clarifying the nature and the scope of the security threats on Modbus systems and networks.

1 Introduction and objectives

1.1 Modbus

The Modicon Communication Bus (*Modbus*) protocol operates in a master-slave or server-client based model. The master devices initiates the queries while the slave devices respond to all such queries. Masters can either send a broadcast message to all the slaves or individually poll a specific device. All the experiments run in this work, like in the original paper [1] are focused on TCP/IP implementation, while the Modbus protocol can also be implemented on top of several communication networks, such as serial or UDP.

Modbus TCP messages are wrapped in TCP/IP header and transmitted over an ethernet-based Modbus network between different devices. The *Modbus Slaves* listen for incoming TCP connections on port 503, that was selected by us, and once a connection has been established the Protocol Data Unit (*PDUs*) are exchanged and encapsulated in TCP messages. The protocol itself can be broken down into six sections:

- Transaction Identifier: a 2-byte field that is used to correlate request and responses; it is easily predictable due to poor randomization.
- Protocol identifier: a 2-byte field that for Modbus is always set to 0.
- Length: a 2-byte field that indicates the length of remaining bytes in the payload.
- Unit Identifier: 1 byte that is used to identify the specific slave at an IP address.

- Function Code: is a 1-byte field that indicates the action requested my master. For example, reading and writing coils, registers or holding registers.
- Data: this field has variable length, with values associated with the various function codes.

1.2 Attack identification

In general, attacks on Modbus systems and networks can exploit protocol's specifications, i.e. they are common to all Modbus systems/networks that conform to the protocol specifications. The attack identification methodology used by the authors [1] involves an analysis of each protocol, that leads to four groups or threat categories: *interception*, *interruption*, *modification*, *fabrication*. The main targets for the Modbus protocol are the master, the field device, the serial communication links and messages. Attacks were implemented based on the possibility of the system to have a Modbus sniffer and a packet injector, that also could block, modify or fabricate arbitrary Modbus messages or sequences of messages.

Fifteen attacks that exploit TCP protocols have been recognized, and as said require access to the master device, network communication path or field device. The most serious attacks are those that disable or bypass the master unit and seize control of field devices, this type of attacks affect the integrity and the availability of messages or of the network; on the other hand attacks on confidentiality involve obtaining information on the network or on slave devices by simply reading messages.

1.3 Objectives

In this work we tried to replicate the various attacks defined by the authors, specifically we emulate one attack for each taxonomy identified in the paper. We built our own Modbus simulator with python libraries and then we produced python scripts for each attack. More precisely, the *interception*, *interruption*, *modification* and *fabrication* identified in the original paper are described and implemented as follows:

- Interception: *Passive reconnaissance* involves passively reading Modbus messages or network traffic, intercepting the messages and reading field device data.

- Interruption: *TCP FIN flood* is an interruption attack that aims to launch spoofed TCP packets with the FIN flag set after a legitimate message from Modbus server to Modbus client, in order to close the TCP connection or cause significant delays.

- Modification: *Response Delay* involves delaying a response message so that the master receives out-of-date information from slaves, is done by sniffing and modifying field device, sending the modified packet with a delay.

- Fabrication: *Rogue Interloper* is a sort of man-in-the-middle attack where a MITM device can sniff and fabricate messages.

2 System setup

This work was entirely done with python scripts, crafting master and slaves with `PyModbus` and working on TCP level with `Scapy`. Although these are two of the most famous python labraries for dealing with TCP Modbus packets, we found out that their documentation is often incomplete and not accurate, with most of the information difficult to retrieve.

Wireshark was employed for sniffing the entire connection on the `Loopback interface` and on `port 503`, in order to obtain a feedback on what actually happened on our Modbus TCP network.

3 Experiments, results and discussions

In this section we describe the experiments done to emulate the attacks. All the scripts are available in the `code` folder of the Repository. We assume that the attacker knows IP address and the port where the devices are listening. First of all, in all the experiments we have a master and a slave except for the Fabrication where we need only the server. The slave is a simple Modbus server that listens on port 503 and the master is a Modbus client that sends requests to the slave. So run the `slave.py` script on a terminal to start the server listening. In the file `master.py` there are three different function used to read, write coils and registers:

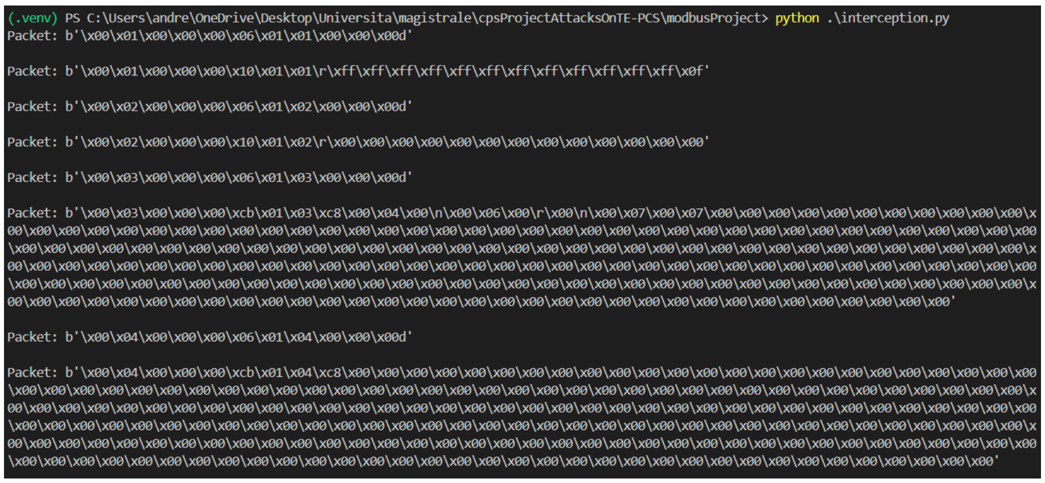
- master_read(): reads the data from all the registers of the slave;

- master_connected(time_wait): the master will be connected for time_wait seconds to the slave, it will read coils and will be disconnected time_wait seconds;
- master_write(): writes values in the holding registers of the slave.

One of these functions is called based on the attack type (change it in the main function of the file). The master has to be run in a different terminal.

3.1 Interception attack

The goal of this attack is to intercept a sent message from the master to the slave. In the master, the function `master_read()` is called. Before calling the master, run the `interception.py` script that will sniff the packet and print it. The attacker will listen to the network and, when it finds the packet with the right characteristics, it will print it. In the image 1 we can see that



```
(.venv) PS C:\Users\andre\OneDrive\Desktop\Universita\magistrale\cpsProjectAttacksOnTE-PCS\modbusProject> python .\interception.py
Packet: b'\x00\x01\x00\x00\x00\x06\x01\x01\x00\x00\x00d'

Packet: b'\x00\x01\x00\x00\x00\x10\x01\x01\r\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x0f'

Packet: b'\x00\x02\x00\x00\x00\x06\x01\x02\x00\x00\x00d'

Packet: b'\x00\x02\x00\x00\x00\x10\x01\x02\r\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

Packet: b'\x00\x03\x00\x00\x00\x06\x01\x03\x00\x00\x00d'

Packet: b'\x00\x03\x00\x00\x00\xcb\x01\x03\xc8\x00\x04\x00\n\x00\x06\x00\r\x00\n\x00\x07\x00\x07\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

Packet: b'\x00\x04\x00\x00\x00\x06\x01\x04\x00\x00\x00d'

Packet: b'\x00\x04\x00\x00\x00\xcb\x01\x04\xc8\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

Figure 1: Results of the interception attack

there are 4 operations requested by the master and 4 responses from the slave. Here there is a conversion of the first packet present in the image:

1. Transaction ID = x00x01 = 1;
2. Protocol ID = x00x00 = 0;
3. Length = x00x06 = 6;

4. Unit ID = x01 = 1;

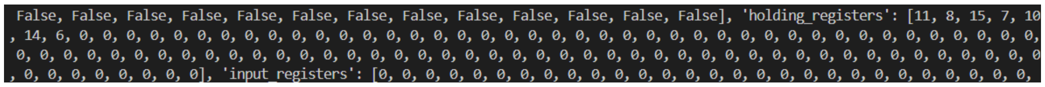
5. Function Code = x01 = 1 (read coils);

6. Data = x00x00x00;

We found the function codes and their assignd functions in the following site codes

3.2 Fabrication attack

The goal of this attack is to fabricate a fake message and send it to the server in order to change the value of a register, more precisely we substituted the value of the register 1 with 1337 of the holding registers. For this attack, after starting the slave, run the `fabrication.py` script that will send the packet to the server. In the following image 2 we can see the register of interest before the attack: A packet with the following characteristics is sent



```
 False, False, False, False, False, False, False, False, False, False, False, False, False], 'holding_registers': [11, 8, 15, 7, 10
, 14, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0], 'input_registers': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Figure 2: Register before the fabrication attack

to the server:

1. Transaction ID = x12x34 (chosen randomly);

2. Protocol ID = x00x00 = 0;

3. Length = x00x06 = 6;

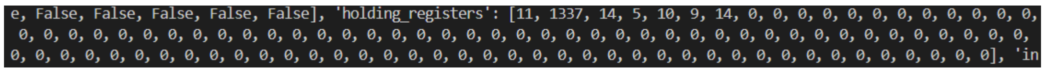
4. Unit ID = x01 = 1;

5. Function Code = x06 = 6 (write single register);

6. Register Address = x00x01;

7. Value = x05x39 = 1337;

And this is the result 3:

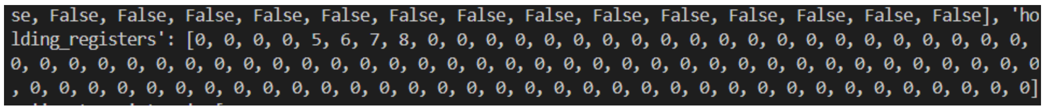


```
e, False, False, False, False, False], 'holding_registers': [11, 1337, 14, 5, 10, 9, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'in
```

Figure 3: Register after the fabrication attack

3.3 Modification attack

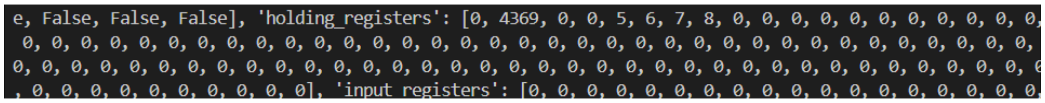
The aim of this attack is to intercept the packet sent by the master to write in multiple registers and modify the value of one of them, and also delaying the response. When a packet with the function code 16 is sent, the attacker will intercept it and modify the value of the register and send it to the server. For this attack, after starting the slave, run the master with the function `master_write()` and `master_read()` to see the registers as in figure. Then run the `modification.py` script and with the master read the registers again to see the modification. In the following image 4 we can see the register of interest before the attack:



```
se, False, False, False, False, False, False, False, False, False, False, False, False, False, False], 'ho
lding_registers': [0, 0, 0, 0, 5, 6, 7, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 4: Register before the modification attack

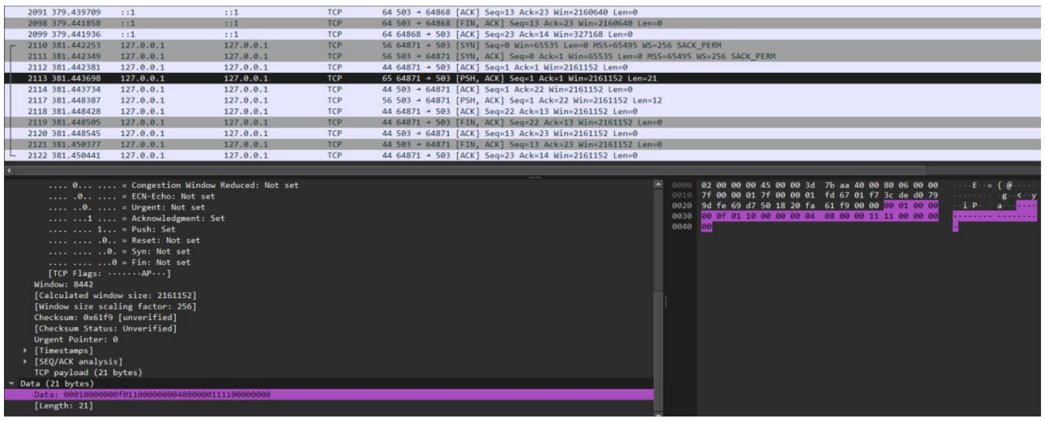
After the modification, the value of the register is changed as shown in the following image 5:



```
e, False, False, False], 'holding_registers': [0, 4369, 0, 0, 5, 6, 7, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'input registers': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Figure 5: Register after the modification attack

In this case the original packet sent by the master is not blocked, but the aim is achieved anyway because the value of the register is changed starting from the packet sent by the master.
As we can see in the image 6 the packet is sent to the server with two seconds of delay (id 2099-2110) and in the bottom-rigth section, higlighted in purple, there are the values x11x11 that have been modified by the attacker.



```
2091 379.439709  ::1        ::1        TCP  64 503 → 64868 [ACK] Seq=13 Ack=23 Win=2160640 Len=0
2098 379.441858  ::1        ::1        TCP  64 503 → 64868 [FIN, ACK] Seq=13 Ack=23 Win=2160640 Len=0
2099 379.441936  ::1        ::1        TCP  64 64868 → 503 [ACK] Seq=23 Ack=14 Win=327168 Len=0
2110 381.442253  127.0.0.1  127.0.0.1  TCP  56 64871 → 503 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
2111 381.442349  127.0.0.1  127.0.0.1  TCP  56 503 → 64871 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
2112 381.442381  127.0.0.1  127.0.0.1  TCP  44 64871 → 503 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
2113 381.443698  127.0.0.1  127.0.0.1  TCP  65 64871 → 503 [PSH, ACK] Seq=1 Ack=1 Win=2161152 Len=21
2114 381.443734  127.0.0.1  127.0.0.1  TCP  44 503 → 64871 [ACK] Seq=1 Ack=22 Win=2161152 Len=0
2117 381.448387  127.0.0.1  127.0.0.1  TCP  56 503 → 64871 [PSH, ACK] Seq=1 Ack=22 Win=2161152 Len=12
2118 381.448428  127.0.0.1  127.0.0.1  TCP  44 64871 → 503 [ACK] Seq=22 Ack=13 Win=2161152 Len=0
2119 381.448505  127.0.0.1  127.0.0.1  TCP  44 64871 → 503 [FIN, ACK] Seq=22 Ack=13 Win=2161152 Len=0
2120 381.448545  127.0.0.1  127.0.0.1  TCP  44 503 → 64871 [ACK] Seq=13 Ack=23 Win=2161152 Len=0
2121 381.450377  127.0.0.1  127.0.0.1  TCP  44 503 → 64871 [FIN, ACK] Seq=13 Ack=23 Win=2161152 Len=0
2122 381.450441  127.0.0.1  127.0.0.1  TCP  44 64871 → 503 [ACK] Seq=23 Ack=14 Win=2161152 Len=0
```

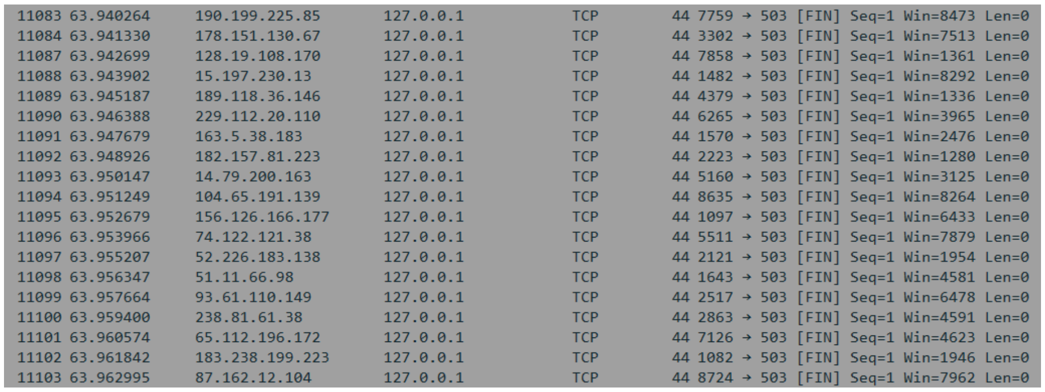
```
      .... 0... .... = Congestion Window Reduced: Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 1... = Push: Set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·······AP···]
    Window: 8442
    [Calculated window size: 2161152]
    [Window size scaling factor: 256]
    Checksum: 0x61f9 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ [Timestamps]
  ▸ [SEQ/ACK analysis]
    TCP payload (21 bytes)
▾ Data (21 bytes)
    Data: 00010000000f011000000004080000111100000000
    [Length: 21]
```

```
0000  02 00 00 00 45 00 00 3d  7b aa 40 00 80 06 00 00   ····E··={·@·····
0010  7f 00 00 01 7f 00 00 01  fd 67 01 f7 3c de d0 79   ·········g··<··y
0020  9d fe 69 d7 50 18 20 fa  61 f9 00 00 00 01 00 00   ··i·P· ·a·······
0030  00 0f 01 10 00 00 00 04  08 00 00 11 11 00 00 00   ················
0040  00                                                 ·
```

Figure 6: Packet exchanged in the modification attack

3.4 Interruption attack

The aim of this attack is to interrupt the communication between the master and the slave by sending a flood of TCP FIN packets. The idea is to send so many packets that the server will be overloaded and will not be able to respond to the master. After starting the slave, run the `interruption.py` script and then the master with the function `master_connected()`. The idea was to create a TCP FIN packet (which is the flag that indicates the end of the connection) and send it to the server with multiple threads in order to create also multiple requests at the same time. This attack didn't work as expected, because the server was able to respond to the master anyway.



```
11083 63.940264   190.199.225.85    127.0.0.1   TCP   44 7759 → 503 [FIN] Seq=1 Win=8473 Len=0
11084 63.941330   178.151.130.67    127.0.0.1   TCP   44 3302 → 503 [FIN] Seq=1 Win=7513 Len=0
11087 63.942699   128.19.108.170    127.0.0.1   TCP   44 7858 → 503 [FIN] Seq=1 Win=1361 Len=0
11088 63.943902   15.197.230.13     127.0.0.1   TCP   44 1482 → 503 [FIN] Seq=1 Win=8292 Len=0
11089 63.945187   189.118.36.146    127.0.0.1   TCP   44 4379 → 503 [FIN] Seq=1 Win=1336 Len=0
11090 63.946388   229.112.20.110    127.0.0.1   TCP   44 6265 → 503 [FIN] Seq=1 Win=3965 Len=0
11091 63.947679   163.5.38.183      127.0.0.1   TCP   44 1570 → 503 [FIN] Seq=1 Win=2476 Len=0
11092 63.948926   182.157.81.223    127.0.0.1   TCP   44 2223 → 503 [FIN] Seq=1 Win=1280 Len=0
11093 63.950147   14.79.200.163     127.0.0.1   TCP   44 5160 → 503 [FIN] Seq=1 Win=3125 Len=0
11094 63.951249   104.65.191.139    127.0.0.1   TCP   44 8635 → 503 [FIN] Seq=1 Win=8264 Len=0
11095 63.952679   156.126.166.177   127.0.0.1   TCP   44 1097 → 503 [FIN] Seq=1 Win=6433 Len=0
11096 63.953966   74.122.121.38     127.0.0.1   TCP   44 5511 → 503 [FIN] Seq=1 Win=7879 Len=0
11097 63.955207   52.226.183.138    127.0.0.1   TCP   44 2121 → 503 [FIN] Seq=1 Win=1954 Len=0
11098 63.956347   51.11.66.98       127.0.0.1   TCP   44 1643 → 503 [FIN] Seq=1 Win=4581 Len=0
11099 63.957664   93.61.110.149     127.0.0.1   TCP   44 2517 → 503 [FIN] Seq=1 Win=6478 Len=0
11100 63.959400   238.81.61.38      127.0.0.1   TCP   44 2863 → 503 [FIN] Seq=1 Win=4591 Len=0
11101 63.960574   65.112.196.172    127.0.0.1   TCP   44 7126 → 503 [FIN] Seq=1 Win=4623 Len=0
11102 63.961842   183.238.199.223   127.0.0.1   TCP   44 1082 → 503 [FIN] Seq=1 Win=1946 Len=0
11103 63.962995   87.162.12.104     127.0.0.1   TCP   44 8724 → 503 [FIN] Seq=1 Win=7962 Len=0
```

Figure 7: Flood of TCP FIN packets

In the image 7 we can see a part of the flood of TCP FIN packets sent to the server, and effectively not all of them are responded by the server. However, if we run the master the server is still able to connect and respond to the request of the master. We supposed that the server is able to manage the requests because of the implementation of the library `pymodbus`: we haven't found a way to determine if there is a maximum number of requests that the server can handle simultaneously, so it could work until the resources are saturated.

4 Conclusions

In this work we have implemented four attacks on Modbus TCP networks, as described in the original paper [1]. Unfortunaltely, the interruption attack didn't work as expected. We tried also other kinds of interruption attacks but we left in the final report only the most significant one. The other attacks worked as expected and we were able to intercept, modify and fabricate. This work allowed us understand how vulnerable the Modbus protocol is.

References

[1] Peter Huitsing et al. "Attack taxonomies for the Modbus protocols". In: *International Journal of Critical Infrastructure Protection* 1 (2008), pp. 37–44.