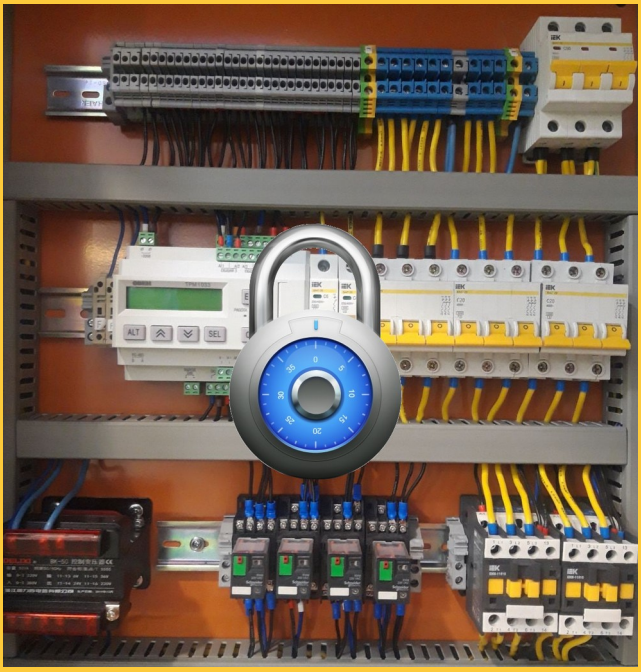
Exercise 10

Penetration Testing
Reconnaissance



Dr Diarmuid Ó Briain

Version: 3.0

Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

Dr Diarmuid Ó Briain

Table of Contents

1 The Modbus Testbed....5
2 The Modbus Server....6
2.1 Initialisation and Configuration....6
2.2 Binding and Listening....6
2.3 Entering the Service Loop....6
2.4 The Modbus Data Types....7
3 The Modbus Client Connection....8
3.1 Establishing the Connection....8
3.2 The Data Retrieval Loop....8
3.3 Displaying the Results....9
3.4 Termination....9
4 Python Virtual Environment....10
4.1 On computers running GNU/Linux....10
4.2 On computers running Microsoft Windows....11
5 Running the Server....12
6 Running the Modbus Client....13
6.1 Reading the current Modbus Server values....15
6.2 Read the Modbus Server values, write new values & re-read....16
7 Testing with Wireshark....18
8 Nmap interrogation of the Modbus Server....20
8.1 The Correct Nmap Approach....22
9 Between two separate computers....24

Illustration Index

Figure 1: The testbed....5
Figure 2: Wireshark capture (before Decode)....21
Figure 3: Wireshark capture (after Decode)....22
Figure 4: Traffic between two computers....29

Index of Tables

Table 1: Modbus Functions....9

This page is intentionally blank

1 The Modbus Testbed

You have been provided with the files to run a small Modbus Testbed with a Modbus Server (Modbus Slave), representing a control device with Discrete Input Contacts and Output Coils as well as an Analogue Input Register and Output Holding Register. With the Modbus Client (Modbus Master) communications can be made with the Modbus Server over the network. These transactions can be monitored and as Modbus is a non-encrypted protocol, it is possible to manipulate the traffic as a Man in the Middle (MitM) attack.

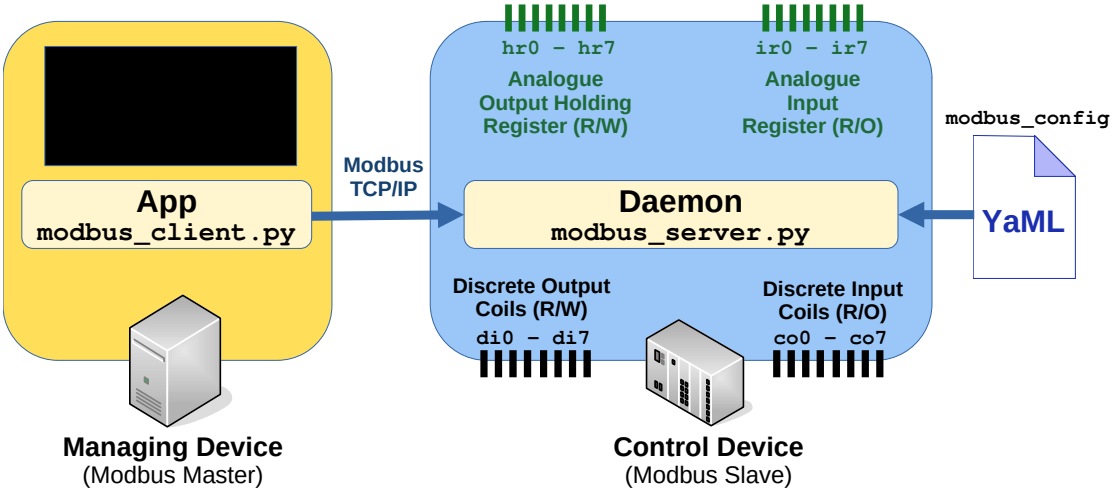


Figure 1: The testbed

The Modbus Testbed in Figure 1 consists of a control device (Modbus Slave), which will be represented by a given program called **`modbus_server.py`**. This control device has four sets of datablocks, namely:

Discrete Input Contacts (Function Code 2)

- A set of eight, input, Read-Only coils (or bits). These contacts are either deactivated (False/0) or activated (True/1).

Discrete Output Coils (Function Codes 1, 5, 15)

- A set of eight, output, Read/Write coils (or bits). These coils are either deactivated (False/0) or activated (True/1).

Analogue Input Register (Function Code 4)

- A set of eight, 16-bit, input, Read-Only registers. These registers contain 16-bit integer values (typically ranging from 0 to 65535, though your current test data uses 20).

Analogue Output Holding Register (Function Codes 3, 6, 16)

- A set of eight, 16-bit, output, Read/Write registers. These registers contain 16-bit integer values (typically ranging from 0 to 65535, though your current test data uses 10 or 8).

A managing device, **`modbus_client.py`** i(Modbus Master) interacts with the control device registers and coils.

2 The Modbus Server

When the Modbus Server, **`modbus_server.py`**, is ran the following series of events occur.

2.1 Initialisation and Configuration

The moment the program starts, it performs necessary initialisations:

- **Load Parameters:** The Modbus Server will bind to a socket consisting of either a given IP address and TCP Port values or the default values 127.0.0.1:5002.
- **Establish Data Map:** The program reads in Register and coil values from the YAML file and with these it creates a data model.

2.2 Binding and Listening

The program then instructs the operating system to bind a network socket (IP address and TCP Port). Once bound, the Modbus Server enters a listening state, actively waiting for incoming TCP connection requests (SYN packets) from any device on the network that wants to communicate using the Modbus protocol.

2.3 Entering the Service Loop

The Modbus Server program enters a service loop, waiting for a Modbus Client. When a remote Modbus Client connects, the Modbus Server accepts the connection, establishing a dedicated communication channel (Socket). For every subsequent packet received on that channel, the Modbus Server:

- **Parses the Modbus Protocol Data Unit (PDU)**: It identifies the function code (e.g., read coils, write registers) and the starting address.
- **Executes the Request**: It retrieves or updates the corresponding data in its internal map.
- **Sends a Response**: It constructs a valid Modbus response packet and sends the data back to the Modbus Client.

The Modbus Server remains in this state, binding, listening, and serving requests, until it is manually stopped (typically with **`Ctrl+C`**), or an error forces it to shut down. The terminal display messages confirming the Modbus Server is running.

2.4 The Modbus Data Types

The four primary data types in Modbus are: two types of Coils (single-bit data) and two types of Registers (16-bit word data). The key difference between the types is whether they are Read-Only or Read/Write.

- **Coils**: are the simplest data type in Modbus, representing a Boolean (ON/OFF or True/False) state. They are typically used for discrete I/O (Input/Output).
- **Registers**: are used for non-Boolean data, such as numeric values. A Modbus register is always 16 bits (two bytes), which can represent an integer value from 0 to 65,535 (or a signed integer/part of a 32-bit float).

3 The Modbus Client Connection

Once the Modbus Client, **`./modbus_client.py`**, is ran with a Socket (IP address and TCP Port), it attempts to connect to the Modbus Server over the network. Assuming the IP Address and TCP Port are correctly matched to the Modbus Server socket the following occurs:

3.1 Establishing the Connection

The Modbus Client sends a SYN (SYNchronise) packet to the Modbus Server. If a Modbus Server is actively listening it accepts the connection, sending back a SYN-ACK (SYNchronise-ACKnowledge) and a full TCP connection is established.

3.2 The Data Retrieval Loop

With the connection active, and depending on what flags were set (**`-a, -r, -c`**) will dictates the action. The program starts to interrogate the Modbus Server. It sends a sequence of Modbus function code requests to the Modbus Server. These codes are listed in Table 1. These requests target:

- **Holding Registers:** Requests to read data points that can be both read from and written to.
- **Input Registers:** Requests to read data points that are read-only (like sensor values).
- **Coils (Digital Outputs):** Requests to read boolean (ON/OFF) states that can be controlled.
- **Discrete Inputs (Digital Inputs):** Requests to read boolean (ON/OFF) states that are read-only.

For each request, the Modbus Client:

- **Formats a PDU:** Encapsulates the desired function code and register address/quantity into a Modbus TCP Application Header.
- **Sends the PDU:** Transmits the message over the established TCP connection.

1. **Server Processing:** The Modbus Server receives, parses the request, fetches the corresponding data from its internal memory map, and formats a response PDU.
2. **Client Receives:** The Modbus Client receives the response packet, validates its contents, and extracts the register values.

Function Code	Hex Code	Operation	Data Type Accessed	Purpose
1	0x01	Read	Coils (Read/Write Bit)	Reads the ON/OFF status of one or more digital outputs.
2	0x02	Read	Discrete Inputs (Read-Only Bit)	Reads the ON/OFF status of one or more digital inputs (sensors).
3	0x03	Read	Holding Registers (Read/Write 16-bit Word)	Reads the numeric value of one or more configuration or setpoint registers.
4	0x04	Read	Input Registers (Read-Only 16-bit Word)	Reads the numeric value of one or more analogue input measurements (e.g., temperature).
5	0x05	Write Single	Coil	Writes a single ON/OFF state (True/False) to a specific digital output.
6	0x06	Write Single	Holding Register	Writes a single 16-bit numeric value to a configuration register.
15	0x0F	Write Multiple	Coils	Writes the ON/OFF status to a contiguous block of digital outputs.
16	0x10	Write Multiple	Holding Registers	Writes a block of 16-bit numeric values to configuration registers.

Table 1: Modbus Functions

3.3 Displaying the Results

As each response is processed, the Modbus Client program prints the retrieved data to Standard Out (STDOUT), the terminal. Output includes:

- The type of register being read.
- The raw hexadecimal value.
- The interpreted decimal or float value (depending on how the program handles data conversion).

This loop continues until the Modbus Client has polled all the addresses defined by the selected flag logic.

3.4 Termination

Finally, once all automatic reads are complete, the Modbus Client program performs a graceful shutdown. It closes the TCP socket, releases resources, and the program exits, returning control to the terminal. It completes with a **`Client disconnected`** message.

4 Python Virtual Environment

4.1 On computers running GNU/Linux

Create a directory for the testbed and copy the files into it.

```
~$ mkdir ~/pymodbus
~$ cd pymodbus
~/pymodbus$ cp ~/Downloads/pymodbus.zip .
~/pymodbus$ unzip pymodbus.zip
Archive:  pymodbus.zip
  inflating: modbus_testbed.pdf
  inflating: modbus_client.py
  inflating: modbus_config.yaml
  inflating: modbus_server.py
  inflating: requirements.txt
~/pymodbus$ ls
modbus_server.py
modbus_client.py
modbus_testbed.pdf
requirements.txt
```

Make the files executable.

```
~/pymodbus$ chmod +x *.py
```

Create a Python virtual environment called **.pymodbus**.

```
~/pymodbus$ python3 -m venv ~/.pymodbus
```

Activate the virtual environment.

```
~/pymodbus$ source ~/.pymodbus/bin/activate
(.pymodbus) ~/pymodbus$
```

Install the relevant modules.

```
(.pymodbus) ~/pymodbus$ python3 -m pip install -r requirements.txt
```

4.2 On computers running Microsoft Windows

Microsoft Windows does not have Python natively as GNU/Linux does. Download and install Python from the following directory first **https://www.python.org/downloads/**

After that installation on Microsoft Windows follows a similar path as on GNU/Linux.

```
C:\Users\an.other> mkdir pymodbus
```

Place the **pymodbus.zip** file in this directory and extract it.

```
C:\Users\an.other> cd pymodbus

C:\Users\an.other\pymodbus> dir
 Volume in drive C is Windows
 Volume Serial Number is DE28-E03F

 Directory of C:\Users\an.other\pymodbus

01/11/2025  08:22    <DIR>          .
01/11/2025  08:18    <DIR>          ..
01/11/2025  01:08            18,221 modbus_client.py
31/10/2025  18:06               586 modbus_config.yaml
30/03/2025  22:38           102,458 modbus_testbed.pdf
01/11/2025  00:16             9,947 modbus_server.py
01/11/2025  08:22           106,670 pymodbus.zip
01/11/2025  00:08                47 requirements.txt
               6 File(s)        237,929 bytes
               2 Dir(s)  434,781,093,888 bytes free

C:\Users\an.other\pymodbus>
```

Note that on GNU/Linux the Python command is **python3** as there was previously a python2 version on that operating system. On Microsoft Windows the Python version 3 command is simply **python**.

```
C:\Users\an.other\pymodbus> python -m venv ..\.pymodbus

C:\Users\an.other\pymodbus> ..\.pymodbus\Scripts\activate

(.pymodbus) C:\Users\an.other\pymodbus> python -m pip install -r
requirements.txt
```

Activity:

- Install Python if necessary and establish the virtual environment.

5 Running the Server

Run the Modbus Server with the **–help** or **–h** switches to see the options.

```
(.pymodbus)~/pymodbus$ ./modbus_server.py --help
usage: modbus_server.py [-h] [-i IP] [-p PORT]

Modbus Server

options:
  -h, --help            show this help message and exit
  -i IP, --ip IP        IP address to bind the server to
                        (default: 127.0.0.1)
  -p PORT, --port PORT  Port to bind the server to
                        (default: 5002)
```

Select a working IP address on the Modbus Server or for basic tests ignore the **--ip**, **-i** switch and the default will be employed.

```
(.pymod)~$ ./modbus_server.py
```

```
--- INITIAL MODBUS STATE (Loaded from config) ---
```

Addr	Coil (0x01, R/W)	DI (0x02, R/O)	HR (0x03, R/W)	IR (0x04, R/O)
0	False	True	0	100
1	False	False	0	200
2	False	True	0	300
3	False	False	0	400
4	False	True	0	500
5	False	False	0	600
6	False	True	0	700
7	False	False	0	800

```
----------------------------------------------------------------
INFO: Starting Modbus TCP server on 127.0.0.1:5002 (Unit ID: 1)...
INFO: Server listening.
```

Upon running the Modbus Server the default data is read in from the **modbus_config.yaml** file and the coils, contacts and registers are populated with the block data which is displayed in the terminal.

Note that on Microsoft Windows the command is ran as follows:

```
(.pymodbus) C:\Users\an.other\pymodbus> python modbus_server.py
```

6 Running the Modbus Client

Open a new terminal and change to the **pymodbus** directory.

```
~$ cd pymodbus
```

Activate the Python virtual environment.

```
~$ source ~/.pymodbus/bin/activate
(.pymodbus) ~$
```

Run the **modbus_client.py** with the **--help** or **-h** flags to see the options.

```
(.pymodbus) ~$ ./modbus_client.py --help
usage: modbus_client.py [-h] [-i IP] [-p PORT] [-u UNIT]
                        [-d | -w] [-c | -r] [-s]

Modbus Client

options:
  -h, --help            show this help message and exit
  -i IP, --ip IP        IP address of the server (default: 127.0.0.1)
  -p PORT, --port PORT  Port of the server (default: 5002)
  -u UNIT, --unit UNIT  Modbus Unit ID (Slave ID) (default: 1)
  -d, --read            Perform a READ-ONLY request of the current state
                        (Default).
  -w, --write           Perform a WRITE TEST: Read initial state, WRITE to
                        Coils/HRs, Read final state, and display both tables.
  -c, --coil-only       Limit the operation to Coils (0x01) and Discrete
                        Inputs (0x02) only.
  -r, --register-only   Limit the operation to Holding Registers (0x03) and
                        Input Registers (0x04) only.
  -s, --server          Also read and display server device identification
                        information only.
```

Select the bound IP address on the Modbus Server, the Modbus Server port and the Modbus Unit ID or for basic tests ignore and the default will be employed.

Note that on Microsoft Windows the command is ran as follows:

```
(.pymodbus) C:\Users\an.other\pymodbus> python modbus_client.py
```

Reading the Server Information. The **--server** or **-s** switch reads the Modbus Server information.

```
(.pymodbus) ~$ ./modbus_client.py --server
[INFO] No action flag (-d or -w) provided. Defaulting to READ-ONLY
(-d) mode.
Connecting to Modbus Server at 127.0.0.1:5002 (Unit ID: 1)...
Client connected.

[INFO] Server identification requested (-s). Overriding all other
actions.

--- Modbus Server Device Identification ---
```

Field	Value
Vendor Name	SETU Modbus Laboratory
Product Code	PM
Major/Minor Revision	1.0
Product Name	N/A
Model Name	N/A
Vendor URL	N/A

```
Client disconnected.
```

6.1 Reading the current Modbus Server values

Run the **modbus_client.py** without flags. This reads the values currently in the Modbus Server .

```
(.pymodbus) ~$ ./modbus_client.py

[INFO] No action flag (-d or -w) provided. Defaulting to READ-ONLY (-d) mode.
Connecting to Modbus Server at 127.0.0.1:5002 (Unit ID: 1)...
Client connected.

--- ACTION: READ-ONLY OF CURRENT MODBUS STATE ---

--- CURRENT MODBUS REGISTER STATE (All Types Read) ---
```

Addr	Coil (0x01, R/W)	DI (0x02, R/O)	HR (0x03, R/W)	IR (0x04, R/O)
0	False	True	0	100
1	False	False	0	200
2	False	True	0	300
3	False	False	0	400
4	False	True	0	500
5	False	False	0	600
6	False	True	0	700
7	False	False	0	800

```
------------------------------------------------------------

Client disconnected.
```

6.2 Read the Modbus Server values, write new values & re-read

Selecting the **--write** or **-d** switches causes the Modbus Client to write new values to the Modbus Server. It will check that the new values are uploaded by performing a read afterwards. All is displayed to the terminal.

```
(.pymodbus)~/pymodbus$ ./modbus_client.py --write
Connecting to Modbus Server at 127.0.0.1:5002 (Unit ID: 1)...
Client connected.

--- ACTION: WRITE TEST SUITE (Read Initial, Write, Read Final) ---

--- INITIAL MODBUS REGISTER STATE (Read before write) ---
```

Addr	Coil (0x01, R/W)	DI (0x02, R/O)	HR (0x03, R/W)	IR (0x04, R/O)
0	False	True	0	100
1	False	False	0	200
2	False	True	0	300
3	False	False	0	400
4	False	True	0	500
5	False	False	0	600
6	False	True	0	700
7	False	False	0	800

```
------------------------------------------------------------

[INFO] Attempting to WRITE Coils (0-7) with values: [True, True,
True, True, True, True, True, True]
[VERIFY] Coils read back MATCHES written values.

[INFO] Attempting to WRITE Holding Registers (0-7) with values:
[88, 88, 22, 55, 55, 121, 99, 55]
[VERIFY] Holding Registers read back MATCHES written values.
```

```
--- FINAL MODBUS REGISTER STATE (Read after write) ---
```

Addr	Coil (0x01, R/W)	DI (0x02, R/O)	HR (0x03, R/W)	IR (0x04, R/O)
0	True	True	88	100
1	True	False	88	200
2	True	True	22	300
3	True	False	55	400
4	True	True	55	500
5	True	False	121	600
6	True	True	99	700
7	True	False	55	800

```
------------------------------------------------------------

Client disconnected.
```

Activity:

- Run the Modbus Server in default mode in a terminal or command prompt.
- Run various Modbus Client commands, read and write read until you are familiar with the interface.

7 Testing with Wireshark

Run **`wireshark`** and select **`Loopback: lo`** to capture if the defaults have been selected when running the **`modbus_server.py`**, otherwise select the appropriate interface. In the filter line add a filter to select the traffic of interest, **`tcp.port == 5002`** for the default settings in the programs or the appropriate ports as required.

Run the **`modbus_client.py`**.

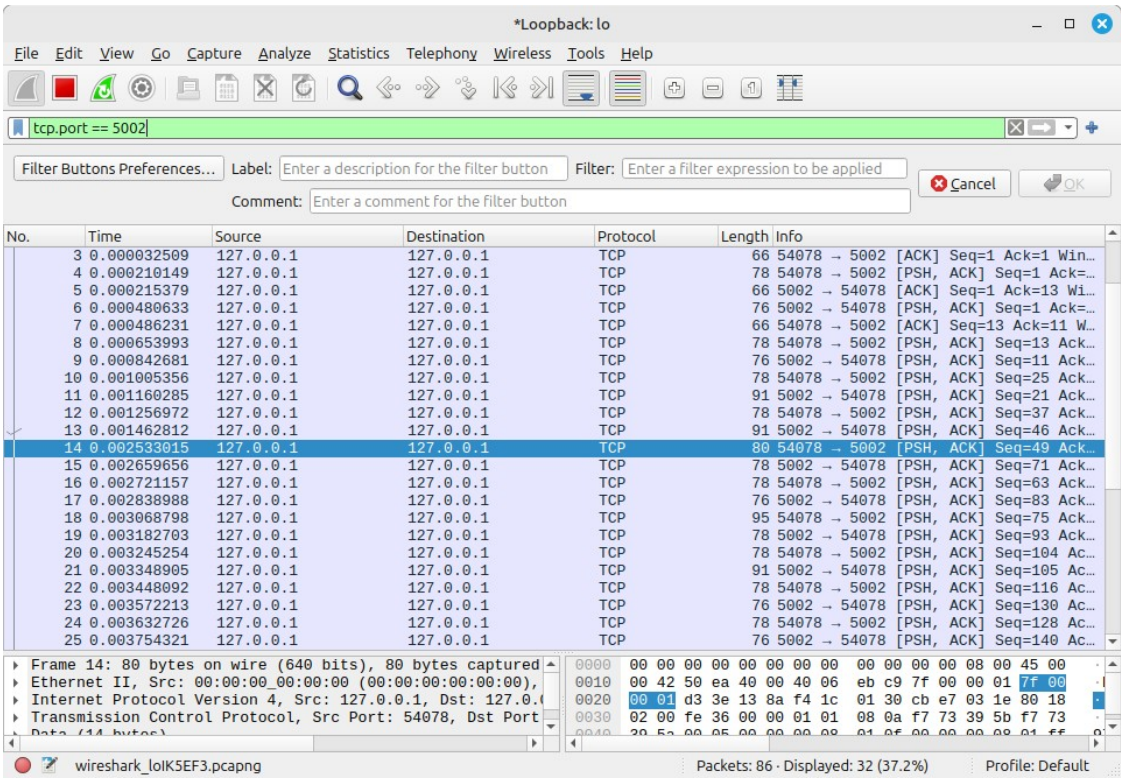


Figure 2: Wireshark capture (before Decode)

Wireshark captures and displays the TCP packets with a port number of **`5002`**.

Right click on any of the packets displayed and select "**`Decode as`**". In the popup window click on **`(none)`** under **`Current`** and select **`Modbus/TCP`** from the drop down menu. Now click **`OK`**. The protocol will change to **`Modbus/TCP`** and the decode window will display the Modbus detail. In Figure 3 the highlighted packet shows a Modbus function 4 activity, **`Read Input Registers`**.

Go through each packet to understand exactly what was requested by **`modbus_client.py`**.

Now use the **`--write`** switch and see the different **`Modbus/TCP`** packets.

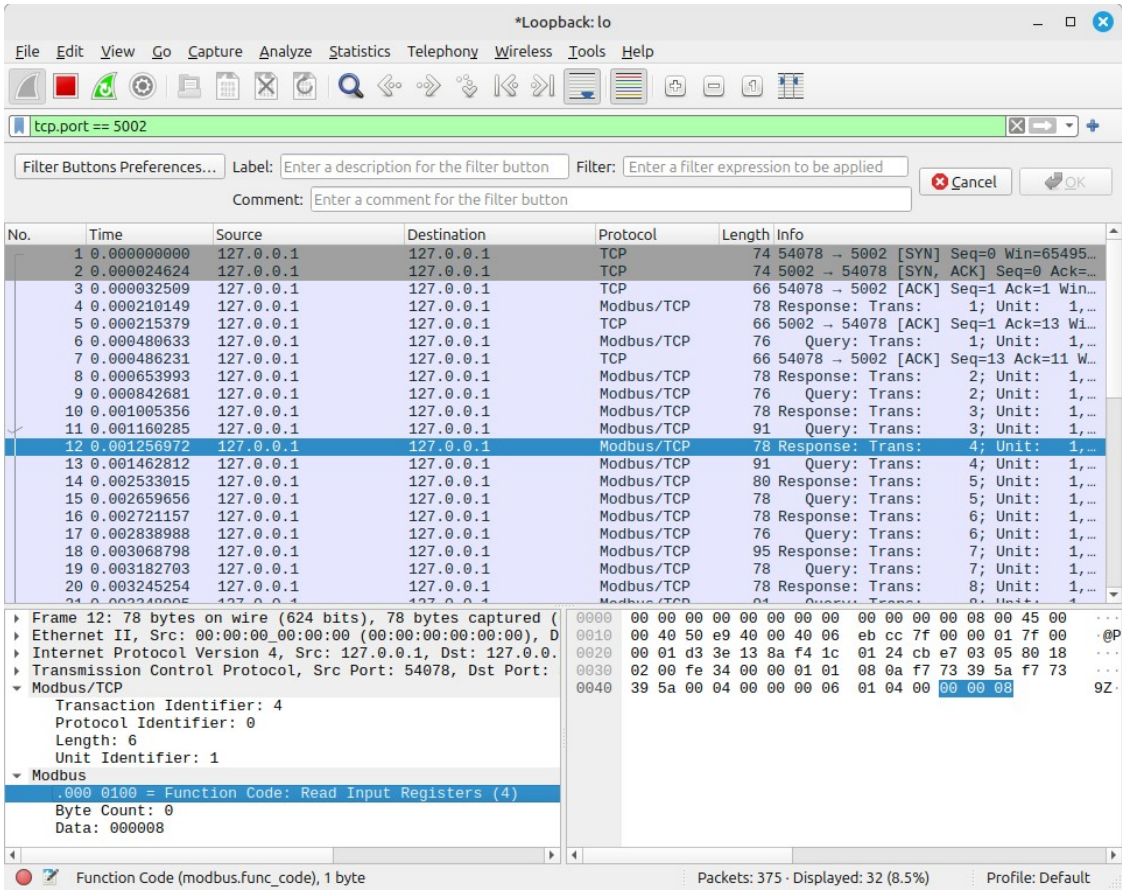


Figure 3: Wireshark capture (after Decode)

Activity:

- Run the Modbus Server in default mode.
- Monitor the traffic on the loopback interface of the computer. Run some Modbus Client commands and monitor the packets.

8 Nmap interrogation of the Modbus Server

Using the Network check the status of the specific port 5002 on the local machine.

```
~$ nmap -p 5002 127.0.0.1
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-11-01 01:11 GMT
Nmap scan report for view-localhost (127.0.0.1)
Host is up (0.00012s latency).

PORT     STATE SERVICE
5002/tcp open  rfe

Nmap done: 1 IP address (1 host up) scanned in 0.07 seconds
```

It is returning Remote File Exchange (RFE) which is a protocol used for remote file operations. It is not a widely used protocol today, but historically, it was assigned to port 5002 by the Internet Assigned Numbers Authority (IANA) and hence the reason Nmap is displaying that protocol and not Modbus.

Now try Nmap with the **-sV** switch which enables version detection. Nmap sends probes to the open port to try and identify the service type (which should hopefully be reported as Modbus/TCP).

```
~$ nmap -p 5002 -sV 127.0.0.1
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-11-01 01:12 GMT
Nmap scan report for view-localhost (127.0.0.1)
Host is up (0.000060s latency).

PORT     STATE SERVICE VERSION
5002/tcp open  rfe?
1 service unrecognized despite returning data. If you know the
service/version, please submit the following fingerprint at
https://nmap.org/cgi-bin/submit.cgi?new-service :
SF-Port5002-TCP:V=7.94SVN%I=7%D=11/1%Time=69055ECF%P=x86_64-pc-
linux-gnu%r
SF:(LDAPSearchReq,9,"0\x84\0\0\0\x03\x02\x81\x04");

Service detection performed. Please report any incorrect results
at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 162.46 seconds
```

Additionally on the Modbus Server.

```
INFO: Starting Modbus TCP server on 127.0.0.1:5002 (Unit ID: 1)...
INFO: Server listening.
ERROR: Datastore unable to fulfill request: 2; Traceback (most recent call last):
                                                                File "/home/user/.pymodbus/lib/python3.12/site-packages/pymodbus/server/requesthandler.py", line 89, in handle_request
    context = self.server.context[self.last_pdu.dev_id]
              ~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^
KeyError: 2

>>>>> extra:  unexpected data: 0x0 0xc 0x0 0x0 0x10 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
>>>>> recv: 0x48 0x45 0x4c 0x50 0xd 0xa extra data:
>>>>> extra:  unexpected data: 0x48 0x45 0x4c 0x50 0xd 0xa
>>>>> recv: 0x16 0x3 0x0 0x0 0x53 0x1 0x0 0x0 0x4f 0x3 0x0 0x3f 0x47 0xd7 0xf7 0xba 0x2c 0xee 0xea 0xb2 0x60 0x7e 0xf3 0x0 0xfd 0x82 0x7b 0xb9 0xd5 0x96 0xc8 0x77 0x9b 0xe6 0xc4 0xdb 0x3c 0x3d 0xdb 0x6f 0xef 0x10 0x6e 0x0 0x0 0x28 0x0 0x16 0x0 0x13 0x0 0xa 0x0 0x66 0x0 0x5 0x0 0x4 0x0 0x65 0x0 0x64 0x0 0x63 0x0 0x62 0x0 0x61 0x0 0x60 0x0 0x15 0x0 0x12 0x0 0x9 0x0 0x14 0x0 0x11 0x0 0x8 0x0 0x6 0x0 0x3 0x1 0x0 extra data:
>>>>> extra:  unexpected data: 0x16 0x3 0x0 0x0 0x53 0x1 0x0 0x0 0x4f 0x3 0x0
```

Nmap's version detection is not compatible with the Modbus/TCP protocol. It also demonstrates why it is always advised to only passively scan on Operational Technology (OT) networks.

The **Modbus/TCP** server is designed to receive a very specific format of data: a Modbus Application Protocol (MBAP) header followed by a Modbus Protocol Data Unit (PDU).

When **-sV, Nmap** is ran it tries to identify the service by sending various generic application probes, often including text-based or well-known protocol handshake attempts (such as HTTP, SSH, or SSL/TLS handshakes). The Modbus Server receives unexpected, non-Modbus data, such as the plain text **HELP\r\n** and the other binary sequences (**0x48 0x45 0x4c 0x50...**). The lines showing unexpected data and the hex dumps (**>>>>> recv: 0x48 0x45 0x4c 0x50 0xd 0xa...**) are the Modbus Server trying to process these invalid packets. The Modbus Server then attempts to parse this garbage data as a Modbus request, which results in internal errors and exceptions in the Python Modbus library. The Modbus Server is returning that it received a connection, but the data sent was not Modbus, and it broke the internal state machine.

8.1 The Correct Nmap Approach

To generate a realistic scenario it is necessary to run the Modbus Server on its native protocol port of 502. To do this the program will need to be ran with root privileges and using the python3 instance in the virtual environment.

```
(.pymodbus)~/pymodbus$ sudo ../.pymodbus/bin/python3
modbus_server.py -p 502
[sudo] password for user: xxxxx

--- INITIAL MODBUS STATE (Loaded from config) ---
```

Addr	Coil (0x01, R/W)	DI (0x02, R/O)	HR (0x03, R/W)	IR (0x04, R/O)
0	False	True	0	100
1	False	False	0	200
2	False	True	0	300
3	False	False	0	400
4	False	True	0	500
5	False	False	0	600
6	False	True	0	700
7	False	False	0	800

```
------------------------------------------------------------
INFO: Starting Modbus TCP server on 127.0.0.1:502 (Unit ID: 1)...
INFO: Server listening.
```

Next step is to run Nmap with the **modbus-discover.nse** script which is essentially a small, automated Modbus Client built into Nmap. It runs a series of standard queries designed to map out the device's configuration and available data, going beyond a simple open-port check. It sends a Modbus command using Function Code 43 / Sub-function 1 (Read Device Identification) which retrieves mandatory identification objects (Vendor Name, Product Code, Revision) and optional ones (Product Name, Model Name, Vendor URL). In the example below the script correctly read and displayed the three configured values from the Modbus Server's data store.

```
~$ nmap --script modbus-discover.nse -p 1-5000 127.0.0.1
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-11-01 01:41 GMT
Nmap scan report for view-localhost (127.0.0.1)
Host is up (0.00010s latency).
Not shown: 4995 closed tcp ports (conn-refused)
PORT     STATE SERVICE
22/tcp   open  ssh
111/tcp  open  rpcbind
502/tcp  open  modbus
| modbus-discover:
|   sid 0x1:
|     Slave ID data: SETU Modbus Laboratory-PM-1.0\xFF
|_    Device identification: SETU Modbus Laboratory PM 1.0
631/tcp  open  ipp
2049/tcp open  nfs

Nmap done: 1 IP address (1 host up) scanned in 0.22 seconds
```

Nmap correctly identifies the Modbus Server running on port 502. It also retrieves some identification information from the Modbus Device Identification information.

This behaviour highlights why the third command approach is the only reliable way to check a Modbus Server:

Command	Why it works / fails
`nmap -p 502 127.0.0.1`	**Works**. It performs a basic SYN scan to check if the port is open without sending any application-layer data, so the Modbus Server is never confused.
`nmap -p 502 -sV 127.0.0.1`	**Fails**. It sends invalid protocol probes (as shown by your terminal output), causing the Modbus Server to throw exceptions.
`nmap -p 502 --script modbus-discover.nse 127.0.0.1`	**Works**. It uses a specialised script that speaks valid Modbus/TCP commands (like Function Code 43 for device identification), so the Modbus Server understands and responds correctly.

For Modbus, always use the dedicated Nmap Scripting Engine (NSE) script (**`--script modbus-discover.nse`**) or stick to the basic port scan. Avoid the generic **`-sV`** option.

Activity:

- Try all three Nmap commands on the testbed.

9 Between two separate computers

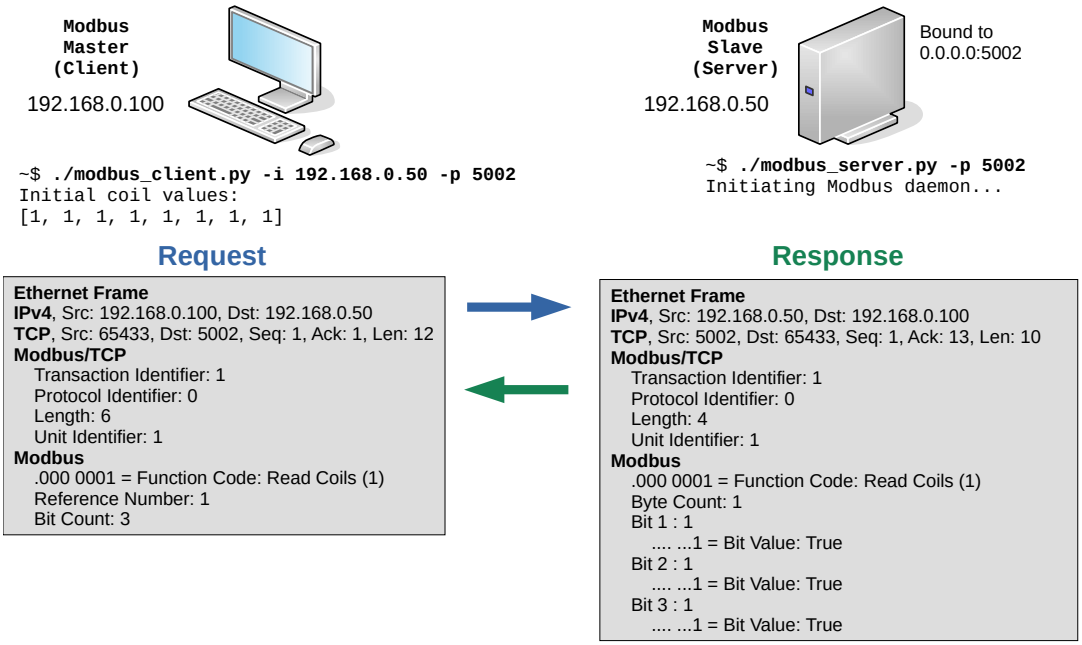


Figure 4: Traffic between two computers

To demonstrate functionality between two separate computers. Run the Modbus Server on the GNU/Linux computer and the Modbus Client on the Microsoft Windows computer.

On the GNU/Linux computer. Note the IP address it is bound to **`0.0.0.0:5002`**, this means bind to all IP addresses that are active on interfaces on the computer.

```
(.pymodbus)~/pymodbus$ ./modbus_server.py –i 0.0.0.0 –p 5002
```

On the Microsoft Windows computer.

```
(.pymodbus) C:\Users\an.other\pymodbus> python modbus_client.py –i
192.168.0.102 –p 5002 –w
```

A connection will establish in the same manner as in the earlier examples; however, in this instance it will occur across the network.

Activity:

- Run Wireshark on both computers, monitor the traffic on each and map corresponding packets.