Trellix ATR Industrial Control System Simulation

Executive Summary

The industrial revolution began between the 18th and the 19th century and never stopped growing. Its goal was to reduce costs by replacing humans with machines and trading factories for industrial processes.

Nowadays, we are surrounded by technology, from our simple desktop or smartphone to complex processes such as energy controls or nuclear power plants. These critical assets are extremely sensitive because they represent the pillars of the economies of our countries. They provide the comfort and safety of people, as well as money for industry.

Industrial control systems (ICS) are at the heart of our factories; they are essentially a way of interacting between the digital and physical worlds, crossing the border between data and physical actions. For a decade, industrialthreats have continued to be more violent and more impactful. This began in 2008 with Stuxnet, the first known industrial malware to manipulate the monitored data of a nuclear power plant. Since then, there have been many other threats, such as Industroyer in 2016, which targeted a power plant in Ukraine and eventually took down the electricity supply for more than 200,000 people and Triton, the first industrial malware targeting a safety system used to protect human lives in 2017.

Due to the complexity of industrial systems and their high cost, information on either is not easily available. To target such systems, attackers must invest time, people and money.

Most of the time, industrial attacks can have a political and economic impact; they can also be used to discredit governments and manipulate public opinion. Such attacks create critical situations that represent major threats. For us, as defenders, it is our role to inform and educate our audience to the industrial risks we are facing.

Last year, Paul Rascagneres and Patrick DeSantis from Talos published a great article about an ICS platform they released. Inspired by this effort, McAfee Advanced Threat Research forked and adapted this platform to create a similar project. This blog will outline the project goals and show how to build your own ICS simulation. We will also explain some of the challenges the industrial field might face, including the Modbus protocol, one of the most used industrial protocols, and present some use cases.

How Does the Modbus Protocol Work?

According to Wikipedia, "Modbus is a serial communications protocol originally published by Modicon (now Schneider Electric) in 1979 for use with its programmable logic controllers (PLCs). Modbus has become a standard communication protocol and is now a commonly available means of connecting industrial electronic devices".

It allows for the monitoring of industrial processes and devices such as valves, engines and thermometers, among many others, and works by following the client/server model.

Industrial processes are usually connected to a supervisory control and data acquisition (SCADA) system which allows data visualization. To communicate with TCP/IP, the reserved port 502 for Modbus has been assigned.

To store data, Modbus uses the following tables within the protocol implementation:

Tables	**Access**	**Object Size**
Coil	Read-write	1 bit
Discrete input	Read-only	1 bit
Input register	Read-only	16 bits
Holding register	Read-write	16 bits

The coil and discrete inputs store 1-bit values (a Boolean value that is either on or off) and the registers store numerical 16-bit values. For each type of data there is one read/write and one read-only table. There are no tables for the 32-bit data size due to the legacy design of Modbus, however it can be stored by combining 2 registers.

Coil and Discrete Inputs can store one bit of data as this is a Boolean value – it is either on or off.

Registers can store an integer, meaning they can be used to store more complex data such as temperature, pressure or speed for example.

Modbus uses numerical function codes that tell the PLC whether to read or write to a specific table. Each function code relates to a specific data table address range. An extract of some function codes has been provided in the table on the next page:

Function Code	Action	Description
Read Coils	0x01	Used to read from 1 to 2000 contiguous statuses of coils in a remote device
Read Discrete Input	0x02	Used to read from 1 to 2000 contiguous statuses of discrete inputs in a remote device
Read Holding Registers	0x03	Used to read the contents of a contiguous block of holding registers in a remote device
Read Input Registers	0x04	Used to read from 1 to 125 contiguous input registers in a remote device
Write Single Coils	0x05	Used to write a single output to either ON or OFF in a remote device
Write Single Register	0x06	Used to write a single holding register in a remote device
Read Exception Status	0x07	Used to read the contents of eight exception status outputs in a remote device
Diagnostics	0x08	Provides a series of tests for checking the communication system between a client device and a server, or for checking various internal error conditions within a server
Write Multiple Coils	0x0F	Used to force each coil in a sequence of coils to either ON or OFF in a remote device
Write Multiple Registers	0x10	Used to write a block of contiguous registers (1 to 123 registers) in a remote device
Read/Write Multiple Registers	0x17	Performs a combination of one read operation and one write operation in a single MODBUSs transaction where the write operation is performed before the read
Read Device Information	0x2B/0x0E	Allows reading the identification and additional information relative to the physical and functional description of a remote device only

As Modbus is a protocol that does not define exactly how the data should be stored in the registers, different vendors can use different ways to store and transmit it in the registers transmitting, for example, the higher bits first then the lower bits.

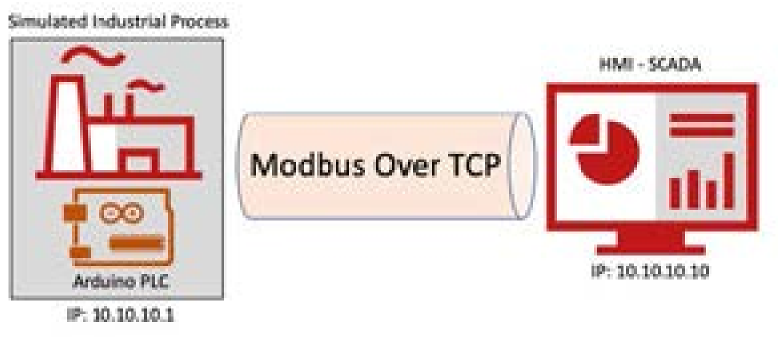
Importantly, the device that receives the data must know the order to receive and read the data.

With that brief overview of the Modbus protocol we can move forward to the next section to build our own industrial system simulation.

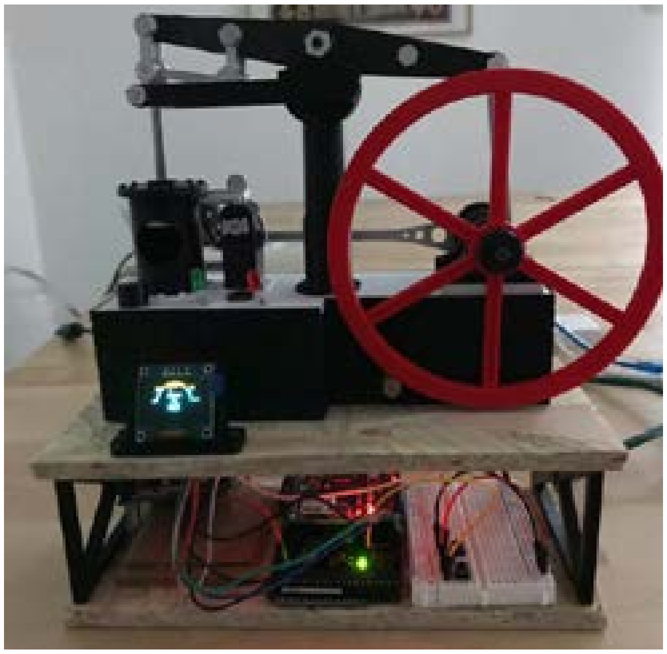
Project Overview

The goal of this project is to provide a functional industrial model that can simulate process monitoring based on the Modbus protocol. We 3D-printed a Beam engine and connected it to an Arduino that will be the PLC. The system is connected through the internet to a Human Machine Interface (HMI) that monitors the engine and some other indicators such as LEDs, temperature and a siren. An on-board OLED screen shows the speed of the engine.

The following diagram illustrates the connection between the components:



The complete setup looks like this:



The project is 3D-printed with this model (https://www.thingiverse.com/thing:1350988).

Hardware Details

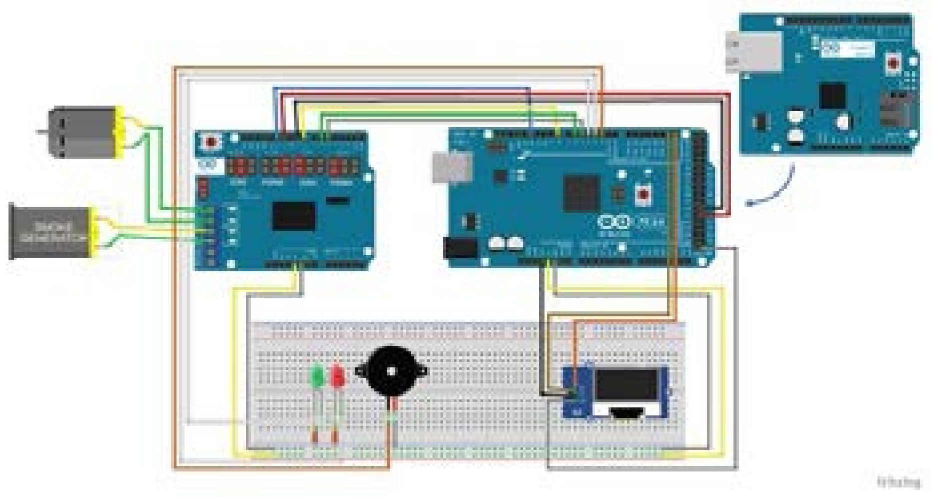
To build this platform we used the following components:

- Arduino Mega: the core module that uses Modbus (https://store.arduino.cc/mega-2560-r3)

- Ethernet shield: for TCP/IP connection and monitoring (https://www.arduino.cc/en/Main/ArduinoEthernetShieldV1)
- **Motor Shield**: to control the motors (https://www.velleman.eu/products/view/?id=412538)
- **An OLED screen**: to shows the speed of the engine (https://www.amazon.com/Diymall-Yellow-Arduino-Display-Raspberry/dp/B00O2LLT30)
- **LEDs**: to monitor the status (green if OK, red if something wrong)
- **A buzzer**: for the siren
- **A steam generator:** to simulate overheating (https://www.graupner.com/Super-steam-generator-6-V/2324/)

The Arduino and the ethernet shield are powered via a USB port. The VM03 shield is powered externally by a 12-volt adapter.

The following diagram shows the interconnection between each component:



NB: in the above diagram the DC motor generates electricity that can perturbate the ethernet connection. To mitigate that we added 3 capacitors directly connected to the DC motor.

Software Details

The code on the Arduino is based on an implementation of the Modbus library (https://github.com/luizcantoni/mudbus) and supports Modbus over TCP. The data monitored with Modbus in this model includes:

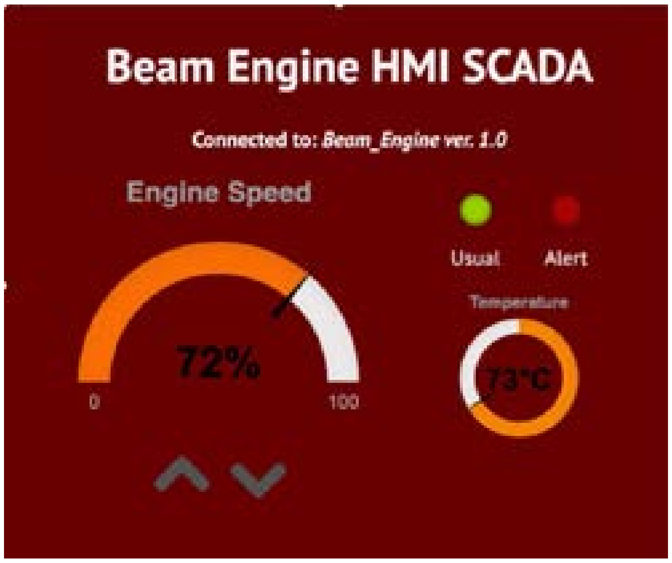
- **Device Information**: it can be retrieved with the function code 0x2b (43).

- **Speed Engine:** this is the speed of the motor. This data is stored in register 6.
- **Gauge value:** this is the speed represented as a percentage. This data is stored in register 7.
- **Temperature**: this is an arbitrary value generated with the motor speed value. The quicker the motor runs, the higher the temperature is. This data is stored in register 10.
- **LED status:** there are currently 2 LEDs, a green one to indicate that everything is working great and a red one to indicate that something is going wrong. The green LED status is stored in coil 0, the red LED in coil 1.

Human Machine Interface (HMI)

The HMI is coded with Flask and Pymodbus. It monitors the Beam Engine and shows the data in a graphical way. It gets the device information as well as the value of the registers. It is also possible to increase or decrease the speed of the engine. The LEDs are also monitored; if something goes wrong, the red LED starts to blink.

Below is a screenshot of the HMI.



Overall Demo

To present the full demonstration of the Beam Engine we created a video that shows it in action: https://youtu.be/vqlCuM_lg3E.

Use Cases

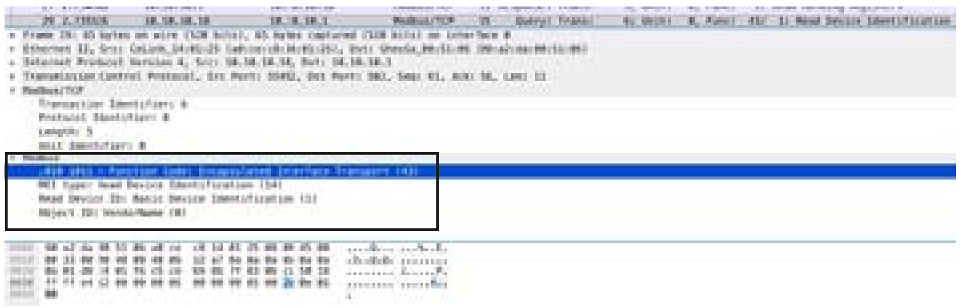
Network Dissection and Attack

To understand how a protocol works it can be useful to sniff the traffic between the HMI and the PLC. Interestingly, Modbus is a clear text protocol without any authentication.

The TCP frame is composed with the following fields:

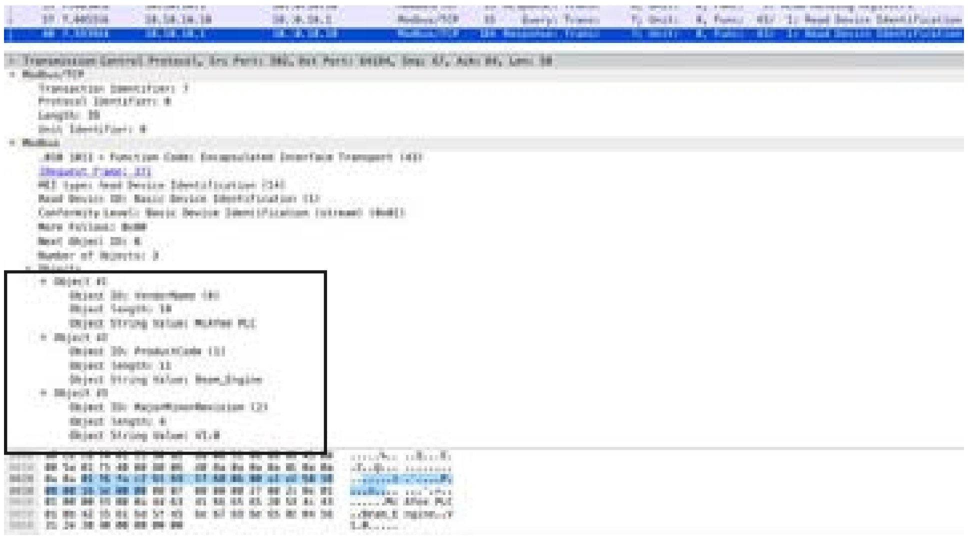
Transaction Identifier	Synchronizing communication
Protocol Identifier	0 for Modbus TCP
Length	Length of the packet
Unit Identifier	Identifier of the slave
Function Code	Function to execute

It is then possible to understand the protocol with Wireshark. In the below screenshot we requested that the PLC give us the information about the device with the function code 0x2B (43):



```
Frame 29: 65 bytes on wire (520 bits), 65 bytes captured (520 bits) on interface 0
Ethernet II, Src: [illegible], Dst: [illegible]
Internet Protocol Version 4, Src: 10.10.10.10, Dst: 10.10.10.1
Transmission Control Protocol, Src Port: [illegible], Dst Port: 502, Seq: [illegible], Ack: [illegible], Len: 11
Modbus/TCP
    Transaction Identifier: 6
    Protocol Identifier: 0
    Length: 5
    Unit Identifier: 0
Modbus
    .010 1011 = Function Code: Encapsulated Interface Transport (43)
    MEI type: Read Device Identification (14)
    Read Device ID: Basic Device Identification (1)
    Object ID: VendorName (0)
```

Response from the PLC:

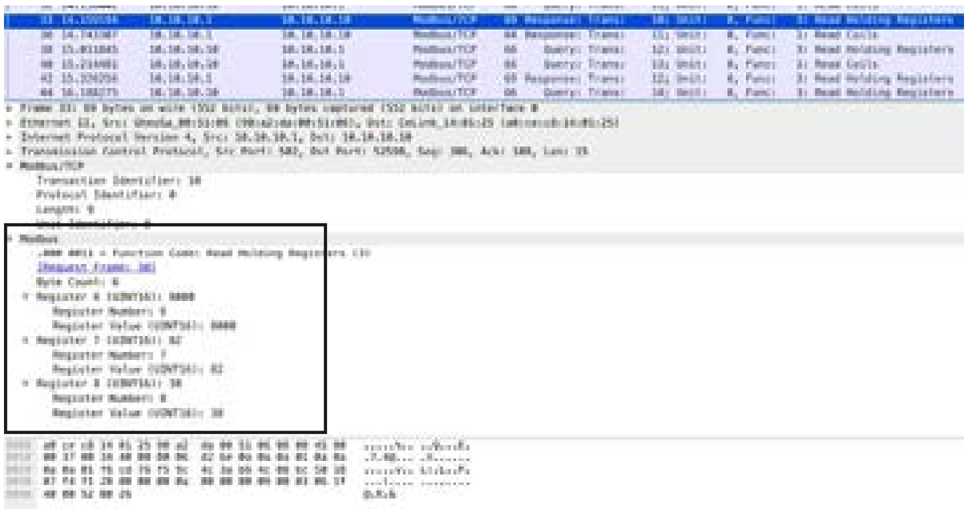


```
Transmission Control Protocol, Src Port: 502, Dst Port: [illegible], Seq: [illegible], Ack: [illegible], Len: 38
Modbus/TCP
    Transaction Identifier: 7
    Protocol Identifier: 0
    Length: 39
    Unit Identifier: 0
Modbus
    .010 1011 = Function Code: Encapsulated Interface Transport (43)
    [Request Frame: 37]
    MEI type: Read Device Identification (14)
    Read Device ID: Basic Device Identification (1)
    Conformity Level: Basic Device Identification (stream) (0x01)
    More Follows: 0x00
    Next Object ID: 0
    Number of Objects: 3
    Objects
        Object #1
            Object ID: VendorName (0)
            Object length: 10
            Object String Value: McAfee PLC
        Object #2
            Object ID: ProductCode (1)
            Object length: 11
            Object String Value: Beam_Engine
        Object #3
            Object ID: MajorMinorRevision (2)
            Object length: 4
            Object String Value: V1.0
```

Here we can retrieve the following information:

- VendorName: McAfee PLC
- ProductCode: Beam Engine
- MajorMinorRevision: 1.0

It is also possible to retrieve information about the register with the function code 0x03 (Read Holding Register):

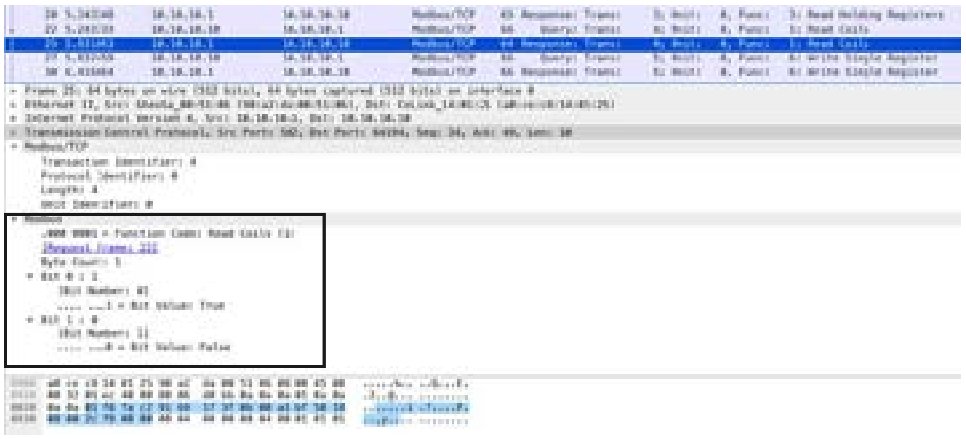


```
[illegible]
▸ Modbus/TCP
    Transaction Identifier: 10
    Protocol Identifier: 0
    Length: 9
    Unit Identifier: 0
▾ Modbus
    .000 0011 = Function Code: Read Holding Registers (3)
    [Request Frame: 38]
    Byte Count: 6
  ▾ Register 6 (UINT16): 0000
      Register Number: 6
      Register Value (UINT16): 0000
  ▾ Register 7 (UINT16): 82
      Register Number: 7
      Register Value (UINT16): 82
  ▾ Register 8 (UINT16): 30
      Register Number: 8
      Register Value (UINT16): 30
[illegible]
```

In this part we can see information about these registers:

- **Register 6**: Speed value
- **Register 7**: Gauge value
- **Register 8**: Temperature

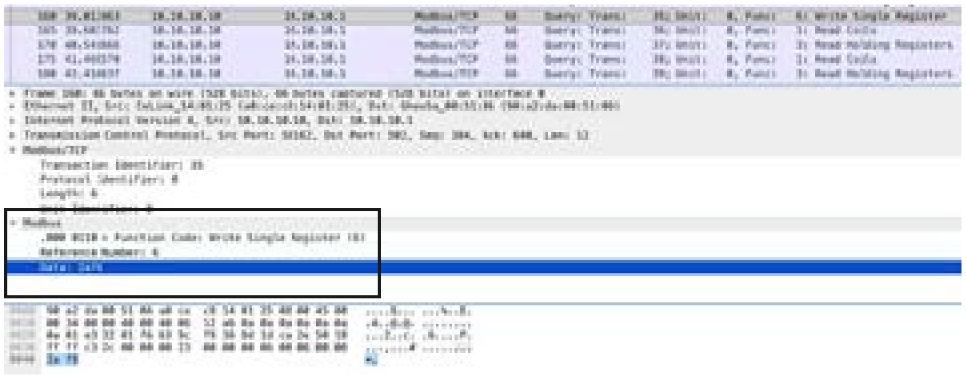
It is also possible to retrieve the content from the coils with the function code 0x01 (Read Coils)



```
[illegible]
▸ Modbus/TCP
    Transaction Identifier: 4
    Protocol Identifier: 0
    Length: 4
    Unit Identifier: 0
▾ Modbus
    .000 0001 = Function Code: Read Coils (1)
    [Request Frame: 22]
    Byte Count: 1
  ▾ Bit 0 : 1
      (Bit Number: 0)
      .... ...1 = Bit Value: True
  ▾ Bit 1 : 0
      (Bit Number: 1)
      .... ...0 = Bit Value: False
[illegible]
```

Here we have coil number 0 that stores the value "True," corresponding to the green LED and coil number 1 that stores the value "False".

If we increase the speed of the engine through the HMI, we send the function code 0x06 (Write Single Register) and we send the data 0x2af8 (11000) which corresponds to the speed value of the engine.



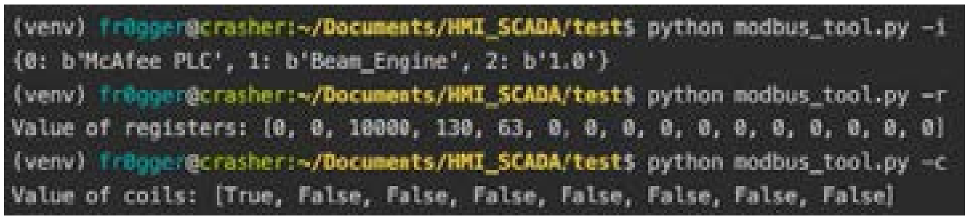
```
[illegible]  10.10.10.10  10.10.10.1  Modbus/TCP  66  Query: Trans:  [illegible] Unit:  0, Func:  6: Write Single Register
[illegible]  10.10.10.10  10.10.10.1  Modbus/TCP  66  Query: Trans:  [illegible] Unit:  0, Func:  1: Read Coils
[illegible]  10.10.10.10  10.10.10.1  Modbus/TCP  66  Query: Trans:  [illegible] Unit:  0, Func:  3: Read Holding Registers
[illegible]  10.10.10.10  10.10.10.1  Modbus/TCP  66  Query: Trans:  [illegible] Unit:  0, Func:  1: Read Coils
[illegible]  10.10.10.10  10.10.10.1  Modbus/TCP  66  Query: Trans:  [illegible] Unit:  0, Func:  3: Read Holding Registers
> Frame [illegible]: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
> Ethernet II, Src: [illegible]
> Internet Protocol Version 4, Src: 10.10.10.10, Dst: 10.10.10.1
> Transmission Control Protocol, Src Port: [illegible], Dst Port: 502, Seq: [illegible], Ack: [illegible], Len: 12
v Modbus/TCP
    Transaction Identifier: [illegible]
    Protocol Identifier: 0
    Length: 6
    Unit Identifier: 0
v Modbus
    .000 0110 = Function Code: Write Single Register (6)
    Reference Number: 6
    Data: 2af8
```

Basic Modbus Attack

Now that we know a little more about how Modbus works, we can try to perform a basic attack. As Modbus is an unauthenticated protocol it is possible to connect to it to modify the data.

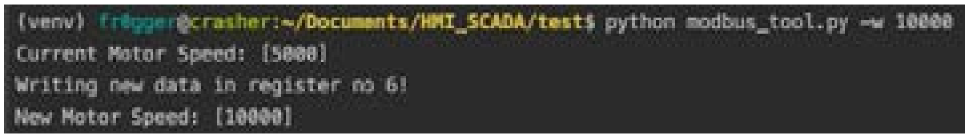
With Pymodbus it is possible to interact directly with the PLC. We wrote a simple tool that can grab information about the PLC such as:

- Device info
- Register values
- Coil values



```
(venv) fr0gger@crasher:~/Documents/HMI_SCADA/test$ python modbus_tool.py -i
{0: b'McAfee PLC', 1: b'Beam_Engine', 2: b'1.0'}
(venv) fr0gger@crasher:~/Documents/HMI_SCADA/test$ python modbus_tool.py -r
Value of registers: [0, 0, 10000, 130, 63, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
(venv) fr0gger@crasher:~/Documents/HMI_SCADA/test$ python modbus_tool.py -c
Value of coils: [True, False, False, False, False, False, False, False]
```

It is also possible to insert data to modify the speed engine with the option "-w".



```
(venv) fr0gger@crasher:~/Documents/HMI_SCADA/test$ python modbus_tool.py -w 10000
Current Motor Speed: [5000]
Writing new data in register no 6!
New Motor Speed: [10000]
```

A Metasploit module that allows the connection and the modification of the PLC via Modbus also exists. It can be found here: https://github.com/rapid7/metasploitframework/blob/master/modules/auxiliary/scanner/scada/modbusclient.rb.

With such a scenario, we could make a man-in-the-middle attack that could manipulate the data and show fake data to the SCADA interface.

Conclusion

Threats that target industrial devices are highly critical because they have an impact on the physical world. Many industrial systems are poorly protected or inadvertently connected to the Internet or even unmonitored, assuming security due to network segmentation. To appreciate the risk of connected industrial systems, it is crucial to understand how such devices work and the risks we are facing so we can adapt the protection.

Most of these devices are expensive and not always accessible to the practitioners. This 3D-printed ICS model is a quick and inexpensive way to demonstrate and practice on a simulated ICS network. We show in this blog how easy it is to manipulate Modbus packets to interfere with a PLC. You can easily improve this model by adding your own values and configuration.

Similar attacks can be and have been reproduced on real PLCs and SCADA systems, making it imperative that industrial systems have critical security measures in place.

References

http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

https://blog.talosintelligence.com/2019/02/oil-pumpjack.html