« Intro to PLCs, ICS and SCADA Defending Against Apache Web Server DDoS Attacks »

The Modbus Protocol

Modbus is a serial communication protocol. It is the most widespread used protocol within ICS.

It works in a Master / Slave mode. This means the **Master** has the pull the information from a **Slave** at regular times.

Modbus is a clear text protocol with no authentication.

Although it was initially developed for serial communication it is now often used over TCP. Other versions of Modbus (used in serial communication) are for example Modbus RTU and Modbus ASCII. For serial communication, Modbus ASCII and Modbus RTU are incompatible (meaning you have to use one or the other but not both on a network).

Every Modbus variant has to choose a frame format:

- Modbus TCP (no checksum as lower network layers should include a checksum);
- Modbus RTU (uses binary encoding and a CRC error check);
- Modbus ASCII (uses ASCII characters);

You can have only one Master on a “Modbus” network and maximum 247 slaves, each with a unique slave ID. In the serial world, the devices have to been connected in a daisy-chain manner, not in a star topology.

In TCP we often refer to the Master as the **Client** and to the Slave as the **Server**.

I based my previous post with an Intro to PLCs, ICS and SCADA on a Black Hat 2014 presentation by Arnaud Soullié in a Industrial Control Systems : Pentesting PLCs 101. This post is based on the same video, together with some of my findings when I did the labs.

Modbus TCP

The TCP frame format consists of

- **Transaction identifier** : to synchronize communication between devices
- **Protocol identifier** : always 0 for Modbus TCP
- **Length field** : identifies the remaining length of the packet
- **Unit identifier** : the address of the slave (most of the time 255 because we already use the TCP/IP addresses as identifier)
- **Function code** : the function to execute
  - Most functions allow to read or write data from/to a PLC
    - **3** : Read Multiple Holding Registers
    - **1** : Read Coils
    - **5** : Write Single Coil
    - …
  - Diagnostics functions
  - Some undocumented functions
- **Data bytes or command**

Storing information

There are two types of places where information can be stored : **coils** and **registers**. Each of these datastore types has two different types of registers : a **read/write** and a **read only**. Each of these datastore types is a reference to a memory address.

Simply put :

- a **coil** is used for storing simple booleans (1 bit). It is read/write and starts from **00001** to **09999**;
- a **discrete input** is a read only type for booleans, starting from **10001** to **19999**;
- an **input register** is a read only type for longer values (16 bits), starting from **30001** to **39999**;
- a **holding register** is a read/write type for longer values (16 bits), starting from **40001** to **49999**;

Be aware that, depending on the hardware implementation, sometimes the registers start at 0 and sometimes they start at 1.

Unit identifiers

A word on Modbus unit devices. In most cases you don’t need a unit id because you already addressed the correct unit via its IP address. In some cases however you will run into a situation where multiple devices are connected to one IP address (for example ‘bridges’). In that case the unit id might have to be set to 255.

The **unit id of 0** can be seen as a **broadcast** address. Messages sent to 0 can be accepted by all slaves. If you setup a Modbus client remember that it can not have unit id 0!

Modbus traffic

You can use ModbusPal to simulate the behavior of a Modbus slave. It is a Java application that allows you to play with different slaves (registers and coils). You can then query the Modbus instance with MBTGET. MBTGET is a simple modbus/TCP client write in pure Perl.

There are a couple of alternatives that you can use to play with Modbus. For example

- Modbus poll (a Master application on Windows)
- CAS Modbus Scanner (a Master application on Windows)
- ModScan (a Master application on Windows)
- modbus-tk (a Slave emulator for Linux)
- Conpot (a Modbus -ICS- honeypot for Linux)

For my setup I used ModbusPal (Slave) on a Kali VM host and MBTGET (Master) on a Linux VM host.

```
Modbus Slave : 192.168.171.182
Modbus Master : 192.168.171.139
```

Analyzing Modbus traffic

The network captures are done with the use of **vmnet-sniffer** to get the traffic between different virtual machines running on OSX.

```
sudo "/Applications/VMware Fusion.app/Contents/Library/vmnet-sniffer" -w
modbus.pcap vmnet8
```

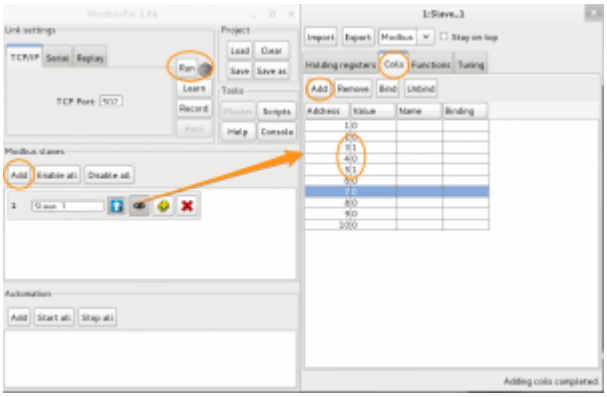
Later on you can then read the pcap files with Wireshark. Modbus TCP traffic runs on **tcp/502**.

Setting up ModbusPal

First we have to setup ModusPal to emulate a Modbus slave. After downloading ModbusPal you can run it with

```
java –jar ModbusPal.jar
```

Add a slave, edit the slave and add some coils.



Ideally you also alter the value of some of the coils. Remember these are booleans, so the value is either 0 or 1. Then click **Run** to start the slave.

MBTGET

Now switch over to your Linux client with MBTGET installed. The usage of MBTGET is fairly easy:

```
usage : mbtget [-hvdsf] [-2c]
                [-u unit_id] [-a address] [-n number_value]
                [-r[12347]] [-w5 bit_value] [-w6 word_value]
                [-p port] [-t timeout] serveur
```

You have to use **-r1** to read coils and with **-r3** you can read holding registers.

Querying coils in Modbus

The first traffic capture is querying the coils in our slave. As a reminder, the network captures are done with vmnet-sniffer and then opened in Wireshark. I use this Modbus command

```
mbtget -r1 -u 1 -n 8 192.168.171.182
```

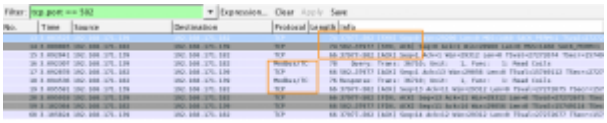
It will read 8 registers from unit id 1 from the slave at 192.168.171.182. The output is

```
values:
  1 (ad 00000):     0
  2 (ad 00001):     0
  3 (ad 00002):     1
  4 (ad 00003):     0
  5 (ad 00004):     1
  6 (ad 00005):     0
  7 (ad 00006):     0
  8 (ad 00007):     0
```

In Wireshark I filter the traffic to Modbus only with

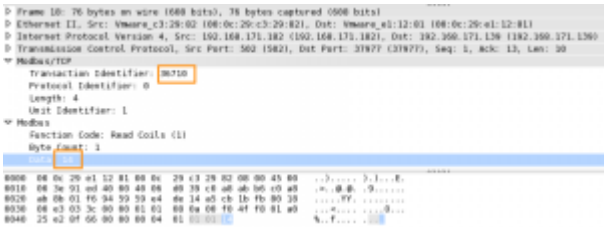
```
tcp.port == 502
```

In the network capture you can first observe the TCP 3-way handshake followed by the first Modbus packet.



Let's have a look at the Modbus packet. Wireshark has a decoder for Modbus (at least for captures done via TCP, for serial captures you have to set mbrtu in the user DLT) which makes it easier to look at the data. The network capture shows that we requested to read 8 bit (the **-n 8**) from coils (the **-r1**) in the unit id 1 (**-u 1**)

The next packet is the Modbus reply packet. In the reply packet you can see that the **Transaction Identifier** (36710) is the same as in the previous request. This is the way that Modbus synchronizes the communication. The reply also contains the requested function (F1 – read coils) and the unit identifier (1). The most interesting part is the **data**, or the payload.



The data is 14. This is 14 in hexadecimal. The coil values are booleans or binary values. So we have to convert the 14hex to a binary value.

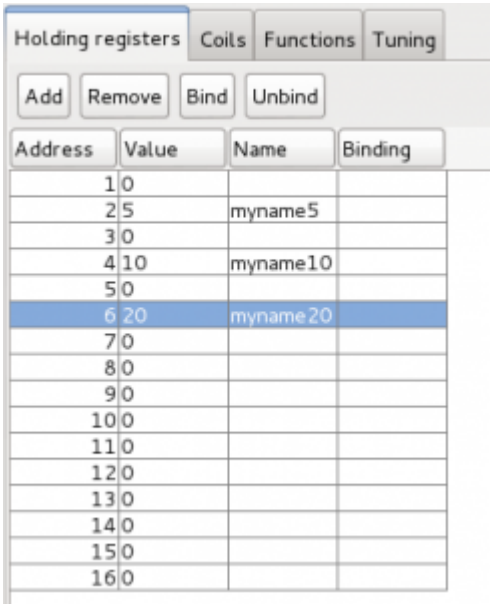
```
1 = 0001
4 = 0100
```

So in binary this becomes 00010100.

This binary values corresponds with how we set the coils previously in ModbusPal. The third and fifth registers were set to 1.

Retrieving holding registers in Modbus

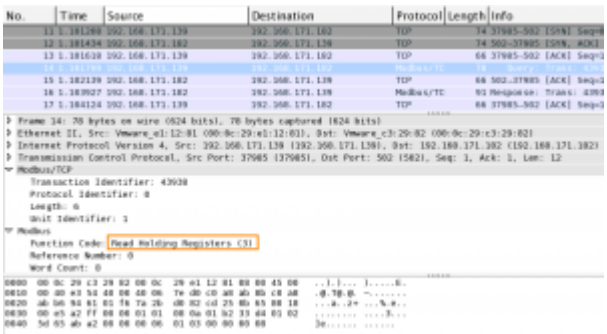
For the next capture I set the value for three holding registers.



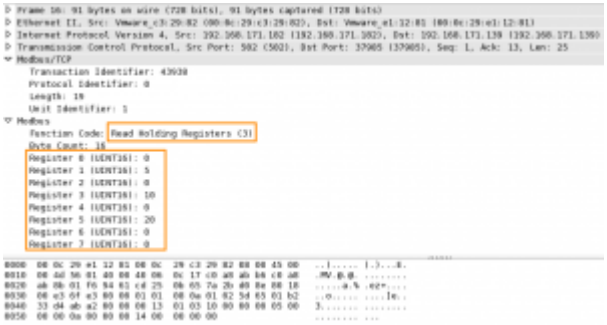
and then query the Slave for its values with

```
mbtget -r3 -u 1 -n 8 192.168.171.182

values:
  1 (ad 00000):     0
  2 (ad 00001):     5
  3 (ad 00002):     0
  4 (ad 00003):    10
  5 (ad 00004):     0
  6 (ad 00005):    20
  7 (ad 00006):     0
  8 (ad 00007):     0
```

In the traffic capture you can now see that the requested function is **Read Holding Registers (F3)** with a length of 8.



The Modbus reply holds the requested function call (3) together with the results of the 8 (starting from 0) registers.



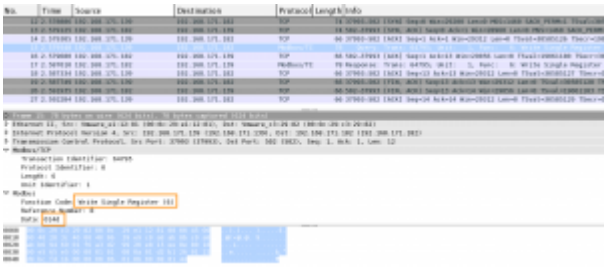
Writing to holding registers in Modbus

Now let's have a look what happens if we write to a holding register.

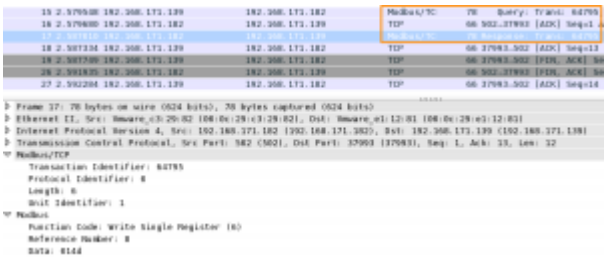
```
mbtget -w6 333 -u 1 -a 8 192.168.171.182

word write ok
```

The ModbusPal interface will show that the register 9 (mbtget starts counting at 0, ModbusPal at 1) holds the value 333.

The packet capture shows a familiar output. First the 3-way handshake and then a Modbus packet. This packet contains the function request **Write Single Register** together with the reference number (8) and the payload (data, 014d).



The response packet (verify the sequence number to make sure you are looking at the correct combination of request/response) again contains the requested function (6) together with the submitted payload.



Nmap modbus-discover

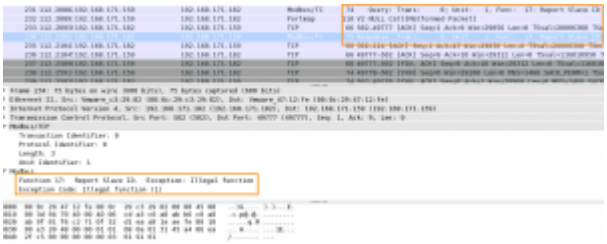
You can search for Modbus devices with nmap. What's more, there's a script that gives you more information on the Modbus device.

```
sudo nmap -p 502 -sV --script modbus-discover.nse 192.168.171.182
```

If you take a look at the source of the script you can see that it tries to discover the available device IDs.

```
for sid = 1, 246 do
  stdnse.debug3("Sending command with sid = %d", sid)
  local rsid = form_rsid(sid, 0x11, "")
```

Notice the **0x11**. The hex value 0x11 corresponds with 17 decimal. The Modbus function code **17** is a diagnostics function to **Report Slave ID**. If you open a packet capture from when nmap was running you will notice the same request.

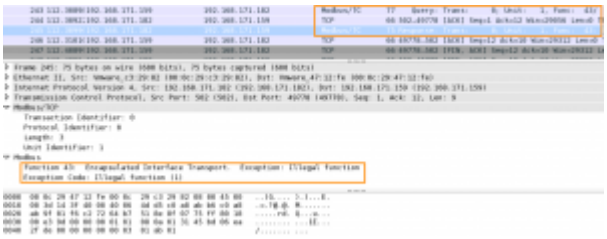


The reply from ModbusPal indicates that this function request is not supported.

The next part of the NSE discovery script sends another request.

```
discover_device_id_recursive = function(host, port, sid, start_id,
objects_table)
  local rsid = form_rsid(sid, 0x2B, "\x0E\x01" .. bin.pack('C', start_id))
  local status, result = comm.exchange(host, port, rsid)
```

Again notice the payload **0x2B**. The hex value 0x2B corresponds with 43 decimal. The Modbus function code **43** is also a diagnostics function to **Read Device Identification**. This is confirmed in the pcap capture.



Online PCAP captures

If you want to practice your skills with reading Modbus PCAP captures then have a look at pcapr.net and query for Modbus.

Conclusion

Modbus TCP traffic is not that hard to read and understand. The biggest challenge that you will probably face is capturing the traffic, especially if it concerns serial communication. Serial Modbus communication is no different than Modbus TCP communication, so once you have the capture and make Wireshark understand the communication it is easy to analyze.

Remember that this is a protocol with little to none security built-in. This makes it easier to capture and read but also more difficult to protect.

Follow @cudeso

Nothing to see here - yet

When they Tweet, their Tweets will show up here.

View on Twitter