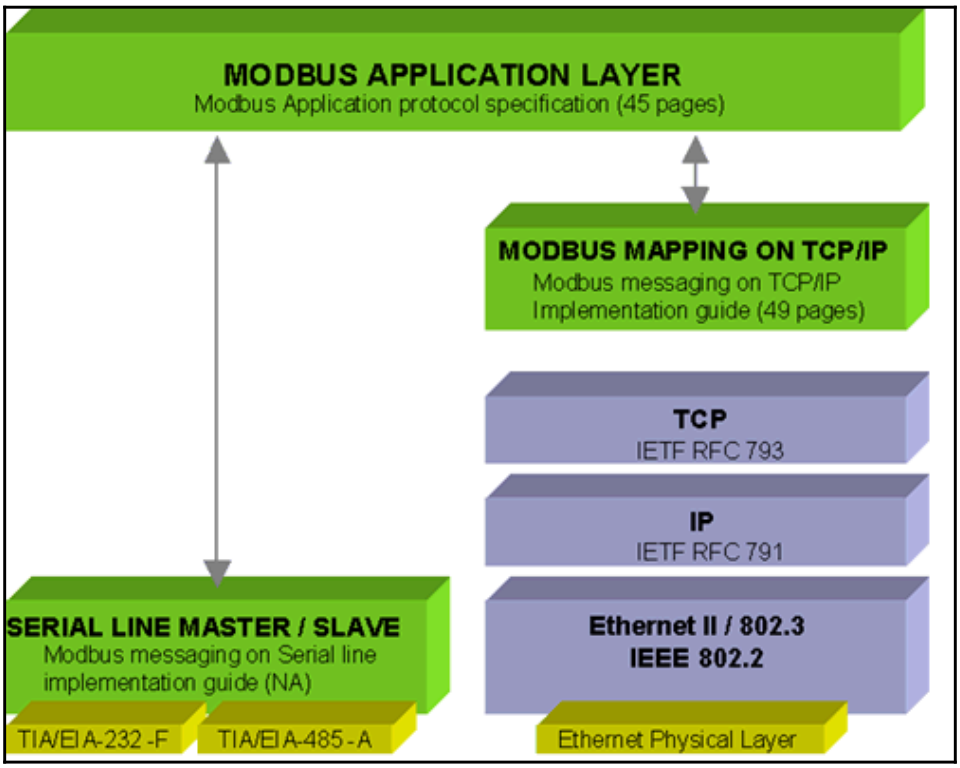
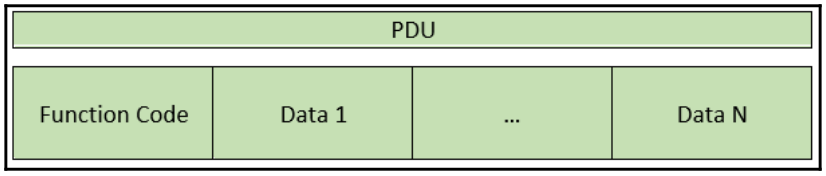
Modbus and Modbus TCP/IP

Introduced in 1979, Modbus has been the de facto standard ever since. Modbus is an application layer messaging protocol. Placed at level 7 of the OSI model, it provides client/server communication between devices connected via different types of communication buses or communication media. Modbus is the most widely used ICS protocol, mainly because it's a proven and reliable protocol, simple to implement, and open to use without any royalties:



On the left-hand side in the preceding figure, we see Modbus communicating over serial (RS-232 or RS-485). The same application layer protocol is used for communicating over Ethernet, as shown on the right-hand side.

The Modbus protocol is built upon a request and reply model. It uses **Function Code** in combination with a data section. The **Function Code** specifies which service is requested for or responded to and the data section provides the data that applies to the function. The **Function Code** and the data sections are specified in the **Protocol Data Unit** (**PDU**) of the Modbus packet frame:

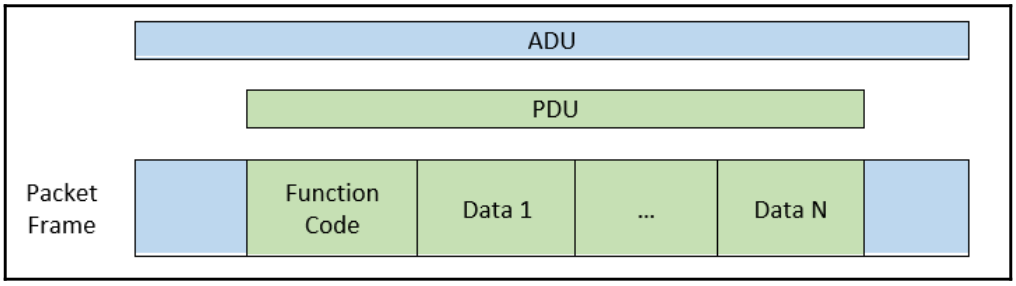


The following is a list of Modbus function codes and their description:

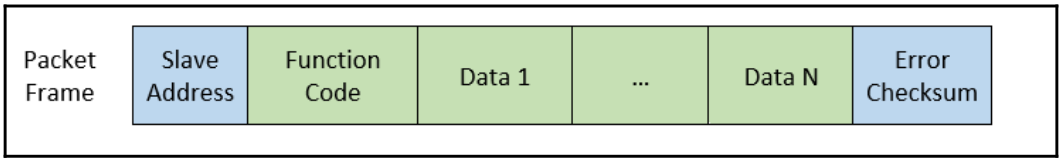
Function	**Description**
`FC=01`	Read Coil Status
`FC=02`	Read Input Status
`FC=03`	Read Multiple Holding Registers
`FC=04`	Read Input Registers
`FC=05`	Write Single Coil
`FC=06`	Write Single Holding Register
`FC=07`	Read Exception Status
`FC=08`	Diagnostics
`FC=11`	Get Comm Event Counter (Serial Line only)
`FC=12`	Get Comm Event Log (Serial Line only)
`FC=14`	Read Device Identification
`FC=15`	Write Multiple Coils
`FC=16`	Write Multiple Holding Registers
`FC=17`	Report Slave ID
`FC=20`	Read File Record
`FC=21`	Write File Record
`FC=22`	Mask Write Register

`FC=23`	Read/Write Multiple Registers
`FC=24`	Read FIFO Queue
`FC=43`	Read Device Identification

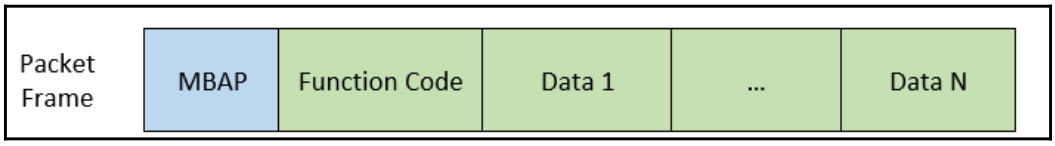
The PDU is the same for every communication media. So, a PDU can be found whether using serial or Ethernet. The way Modbus adapts to different media is by means of an **Application Data Unit**, or **ADU**:



The ADU varies in structure depending on what communication medium is used. The ADU for serial communications, for example, consists of an address header, the PDU, and an **Error Checksum** trailer:



On the other hand, with Modbus TCP/IP, the addressing is done by the IP and TCP layers of the Ethernet packet and the ADU frame consists of a **Modbus Application** (**MBAP**) header and the PDU while omitting the **Error Checksum** trailer.



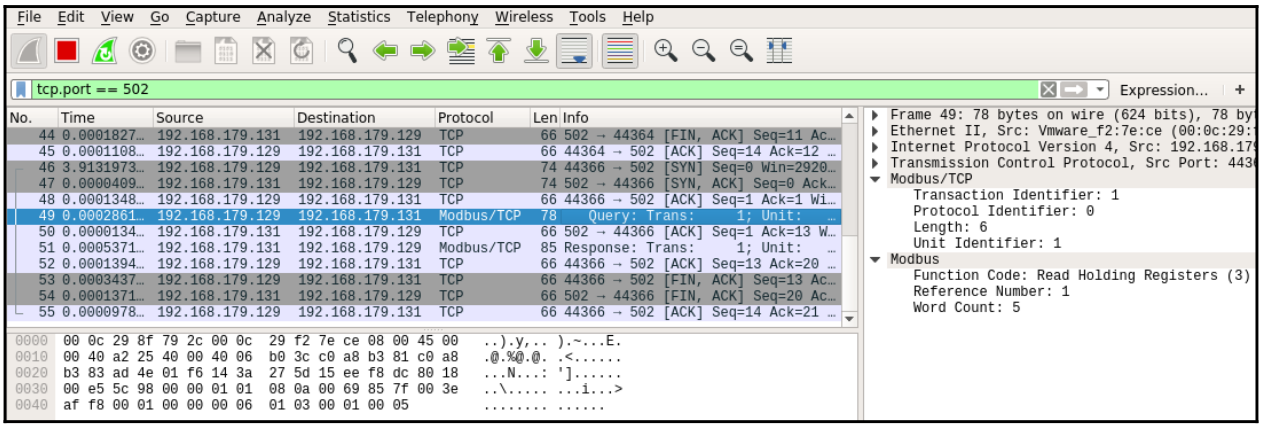
The MBAP includes the following:

- A **Transaction ID** and 2 bytes set by the client to uniquely identify each request. These bytes are echoed by the server since its responses may not be received in the same order as the requests.
- A **Protocol ID** and 2 bytes set by the client. Always equals `00 00`.
- The **Length** and 2 bytes identifying the number of bytes in the message to follow.
- The **Unit ID** and 1 byte set by the client and echoed by the server for the identification of a remote slave connected on a serial line or on some other communication medium:

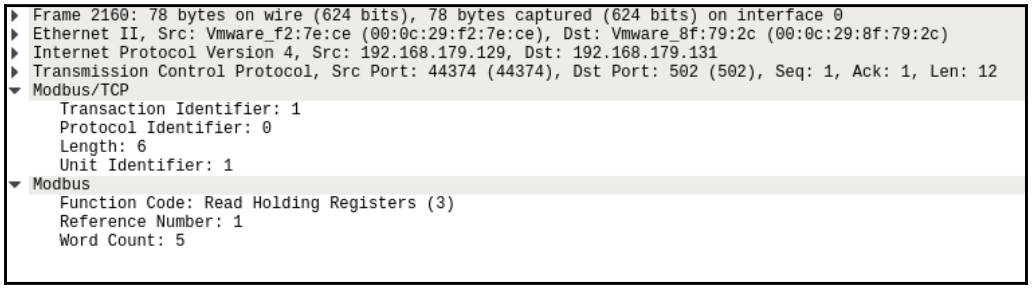
MBAP				PDU		
Transaction ID	Protocol ID	Length	Unit ID	Function Code	Data Start	Data Count
00 01	00 00	00 06	01	03	00 01	00 05

The packet capture was taken with the help of the excellent Wireshark tool, freely downloadable from `https://www.wireshark.org`.

With this basic information covered, we can start dissecting Modbus network packets such as the one shown next:



In the preceding packet, a client device at `192.168.179.129` sends a query to a Modbus server at `192.168.179.131` to request for the values of 5 **Memory Words** (**MW**):



Let's look at the individual layers within this packet. It is not the intention of this book to make you an expert in the inner workings of the Internet Protocol; many books have been written on that subject. But for those of you that are new to the subject or need a refresher on the material, I recommend that you read the book *TCP/IP Guide – A Comprehensive, Illustrated Internet Protocols Reference* by *Charles M. Kozierok*.

Having said that, within a network packet, the Ethernet layer shows the physical source and destination addresses of the packet. The physical address of a device is its **Media Access Control** or the **MAC** address that is burned into the **Network Interface Card** (**NIC**) of the device. Switches make decisions where to forward Ethernet frames to, based on this MAC address. A MAC address normally doesn't change for a device and it is not routable, which means that it is meaningless outside of the local network.

The next layer, the Internet Protocol layer, shows the logical source and destination addresses of the packet by means of IP addresses. An IP address is manually assigned to a device or obtained via DHCP or BOOTP. Within a local network, an IP address will need to be converted into a MAC address before it can be forwarded by a switch. This conversion is done via a protocol called **Address Resolution Protocol**, or **ARP**. If a device wants to know the physical address of an IP address it wants to communicate with, it sends an ARP packet that is a broadcast packet, received by all devices on the local network that basically asks the question who is the device with IP address `xxx.xxx.xxx.xxx`? Please send me your MAC address.

The response of this query will be stored in the device's ARP table, which is a means to temporarily remember the IP address to MAC address relation so the question doesn't have to be repeated for every packet. If the IP address falls outside the local network subnet range, the device will send the packet to the default gateway (router) for that subnet, if one is defined. Routers make decisions based on IP addresses, so adding an IP address to a packet makes it routable so that the client and the server can be on different subnets or completely different networks on opposite sides of the world. Here, Ethernet and the Internet Protocol are tasked with getting the packet to the right targeted address, and the **Transmission Control Protocol** (**TCP**) is responsible for setting up a connection between the server and client's targeted application. The TCP destination port `502`, as shown in the captured packet shown earlier, is the port the Modbus server application runs on. The source port is a randomly chosen value and is used in combination with other details by the TCP protocol to track TCP sessions.

Next, the packet shows the **Modbus/TCP** layer. This is the ADU that Modbus uses to communicate between devices. Wireshark does a great job at dissecting the individual fields of the Modbus protocol. We can see the four fields of the MBAP header within the ADU:

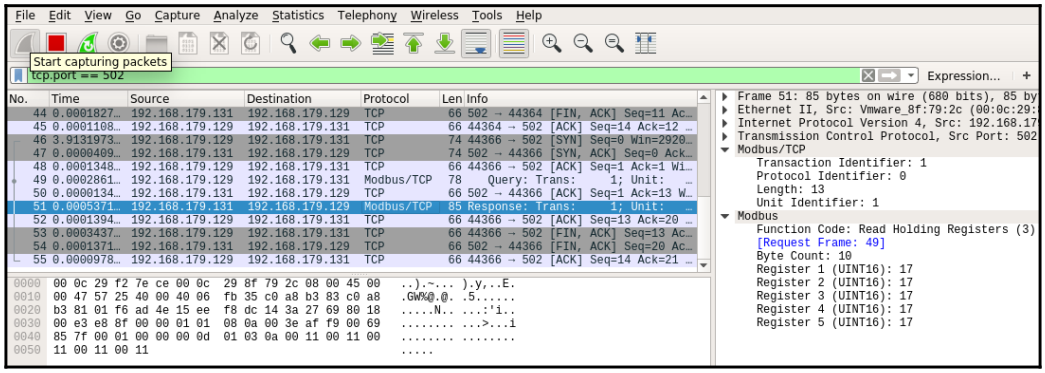
- **Transaction identifier**: `1`
- **Protocol identifier**: `0`
- **Length**: `6`
- **Unit identifier**: `1`

It also shows the fields for the PDU of the Modbus frame, revealing the following:

- **Function code**: `3` (Read Holding Registers)
- **Reference number**: `1` (start at holding register 1)
- **Word count** - `5` (read 5 holding registers)

The implementation of the data part of the PDU is dependent on the requested function. In this case, with function `3`, Read Holding Registers, the data part of the PDU is interpreted as words (16-bit integers). For function `1`, Read Coils, the data will be interpreted as bits, and for other functions, the data part of the PDU may contain additional information all together in order to support the function request.

In the following packet capture we can observer the Modbus server responding with the requested data:



Notice that the server response repeats the function code in the PDU and then uses the data section for the answer.

Now let's start having some fun with this protocol.

Breaking Modbus

For the following exercises, I am using a lab setup that includes two virtual machines, one running a copy of Ubuntu Linux with IP address `192.168.179.131` assigned and the other running a copy of Kali Linux with IP address `192.168.179.129` assigned. The Ubuntu Linux virtual machine is used to run a Modbus stack, implemented in Python. The Kali virtual machine will be our attacker. I choose Kali Linux because it is a free pentesting distro that comes preloaded with a slew of hacking tools.

To get the Modbus server running on the Ubuntu VM, open Command Prompt and install the `pyModbus` module with the following commands:

```
$ sudo apt-get install python-pip      # In case pip did not get installed
$ sudo pip install pyModbus
```

Next, we will write a small script to start an asynchronous Modbus server. Open the text editor of your choice and type in the following script:

```
#!/usr/bin/env python
'''
Asynchronous Modbus Server Built in Python using the pyModbus module
'''

# Import the libraries we need
from pymodbus.server.async import StartTcpServer
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext

# Create a datastore and populate it with test data
store = ModbusSlaveContext(
    di = ModbusSequentialDataBlock(0, [17]*100),    # Discrete Inputs
initializer
    co = ModbusSequentialDataBlock(0, [17]*100),    # Coils initializer
    hr = ModbusSequentialDataBlock(0, [17]*100),    # Holding Register
initializer
    ir = ModbusSequentialDataBlock(0, [17]*100))    # Input Registers
initializer
context = ModbusServerContext(slaves=store, single=True)

# Populate the Modbus server information fields, these get returned as
#  response to identity queries
identity = ModbusDeviceIdentification()
identity.VendorName  = 'PyModbus Inc.'
identity.ProductCode = 'PM'
identity.VendorUrl   = 'https://github.com/riptideio/pyModbus'
identity.ProductName = 'Modbus Server'
identity.ModelName   = 'PyModbus'
identity.MajorMinorRevision = '1.0'

# Start the listening server
print "Starting Modbus server..."
StartTcpServer(context, identity=identity, address=("0.0.0.0", 502))
```

Save the script as `Modbus_server.py` and then start it, as follows:

```
$ sudo python Modbus_server.py
Starting Modbus server...
```

The Modbus server application is now running on the Ubuntu VM. We can start sending queries.

There are several clients out there that can request data from a Modbus server. One such client is `modbus-cli`, written by Tallak Tveide and available for download from GitHub at `https://github.com/tallakt/Modbus-cli` or installable as a RubyGems, which we are going to do on our Kali VM. Open a Terminal and type in the following:

```
$ sudo gem install modbus-cli

Successfully installed Modbus-cli-0.0.13
Parsing documentation for Modbus-cli-0.0.13
Done installing documentation for Modbus-cli after 0 seconds
1 gem installed
```

The `modbus-cli` is a very simple but effective tool with just a few parameters required to get it going:

```
$ sudo modbus -h
Usage:
    modbus [OPTIONS] SUBCOMMAND [ARG] ...

Parameters:
    SUBCOMMAND                    subcommand
    [ARG] ...                     subcommand arguments

Subcommands:
    read                          read from the device
    write                         write to the device
    dump                          copy contents of read file to the device

Options:
    -h, --help                    print help
```

As an example, the following command will go out and read the status of Coils `1` through:

```
# modbus read 192.168.179.131 %M1 5
%M1         1
%M2         1
%M3         1
%M4         1
%M5         1
```

Modbus users will recognize the syntax for the requested register, `%M`, which means memory bit.

Other options are as follows:

Data type	Data size	Schneider address	Modicon address	Parameter
Words (default, unsigned)	16 bits	`%MW1`	400001	`--word`
Integer (signed)	16 bits	`%MW1`	400001	`--int`
Floating point	32 bits	`%MF1`	400001	`--float`
Double words	32 bits	`%MD1`	400001	`--dword`
Boolean (coils)	1 bit	`%M1`	400001	N/A

The `Modbus` command supports the addressing areas `1...99999` for coils and `400001...499999` for the rest using Modicon addresses. Using Schneider addresses, the `%M` addresses are in a separate memory from `%MW` values, but `%MW`, `%MD`, and `%MF` all reside in a shared memory, so `%MW0` and `%MW1` share the memory with `%MF0`.

The command creates the following request packet:

```
▸ Frame 1701: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface 0
▸ Ethernet II, Src: Vmware_f2:7e:ce (00:0c:29:f2:7e:ce), Dst: Vmware_8f:79:2c (00:0c:29:8f:79:2c
▸ Internet Protocol Version 4, Src: 192.168.179.129, Dst: 192.168.179.131
▸ Transmission Control Protocol, Src Port: 44546, Dst Port: 502, Seq: 1, Ack: 1, Len: 12
▾ Modbus/TCP
    Transaction Identifier: 1
    Protocol Identifier: 0
    Length: 6
    Unit Identifier: 1
▾ Modbus
    .000 0001 = Function Code: Read Coils (1)
    Reference Number: 1
    Bit Count: 5
```

Other commands that can be issued include the following:

```
Read the first 5 input registers
# modbus read 192.168.179.131 1 5

Read 10 integer registers starting at address 400001
# modbus read --word 192.168.179.131 400001 10
```

Writing is also possible with the help of following command:

```
# modbus write 192.168.179.131 1 0
This command writes a 0 to the input, verified with
# modbus read 192.168.179.131 1 5
1           0
2           1
```

```
3           1
4           1
5           1
```

The possibilities are extensive. Feel free to play around with them and experiment. The biggest takeaway from this exercise should be that no form of authentication or authorization has been required for any of these commands. What this means is that anyone who knows the address for the Modbus-enabled device (PLC) can read and write its memory and I/O banks.

Finding the Modbus enabled devices is a task that can be achieved with **Nmap** (**Network Mapper**).

NMAP is a network/port scanner, originally written by Gordon Lyon and available for download from `https://nmap.org/download.html`.

Nmap can scan a (local) network for hosts that are up and then scan those hosts for open ports. Let's try that. On our Kali VM, run the following command:

```
# nmap -sP 192.168.179.0/24
```

This will perform a ping scan (`-sP`) of the `192.168.179.0` subnet and report back on any live hosts found:

```
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-13 04:32 EDT
Nmap scan report for 192.168.179.131
Host is up (0.000086s latency).
MAC Address: 00:50:56:E5:C5:C7 (VMware)
Nmap scan report for 192.168.179.129
Host is up.
Nmap done: 256 IP addresses (2 hosts up) scanned in 27.94 seconds
```

The results show two live hosts, the Kali VM at `192.168.179.129` and the Ubuntu VM at `192.168.179.131`. Let's see what ports are open on the Ubuntu machine. Enter the following command to have Nmap scan for open ports:

```
# nmap -A 192.168.179.131
(-A: Enable OS detection, version detection, script scanning, and
traceroute)

Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-13 04:38 EDT
Nmap scan report for 192.168.179.131
Host is up (0.00013s latency).
All 1000 scanned ports on 192.168.179.131 are closed
```

```
MAC Address: 00:0C:29:8F:79:2C (VMware)
Too many fingerprints match this host to give specific OS details
Network Distance: 1 hop
TRACEROUTE
HOP RTT     ADDRESS
1   0.13 ms 192.168.179.131

OS and Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 15.00 seconds
```

Hmm, looks like all ports are closed, but note that Nmap scans only a select (`1000`) ports by default. We can scan all TCP ports by adding `-p-` to the command, which instructs Nmap to scan all TCP ports (`0-65535`). This command will take considerably longer to run and will be very noisy on the network. Not particularly stealthy, but we are on our own turf, so we can get away with this. In real life, you would want to limit yourself and either scan in smaller sub-sections, scan slower, or take educated guesses on port ranges:

```
# nmap -A 192.168.179.131 -p-

Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-13 04:43 EDT
Nmap scan report for 192.168.179.131
Host is up (0.00013s latency).
Not shown: 65534 closed ports
PORT    STATE SERVICE VERSION
502/tcp open  mbap
MAC Address: 00:0C:29:8F:79:2C (VMware)
Device type: general purpose
Running: Linux 3.X|4.X
OS CPE: cpe:/o:linux:linux_kernel:3 cpe:/o:linux:linux_kernel:4
OS details: Linux 3.2 - 4.6
Network Distance: 1 hop

TRACEROUTE
HOP RTT     ADDRESS
1   0.13 ms 192.168.179.131

OS and Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 153.91 seconds
```

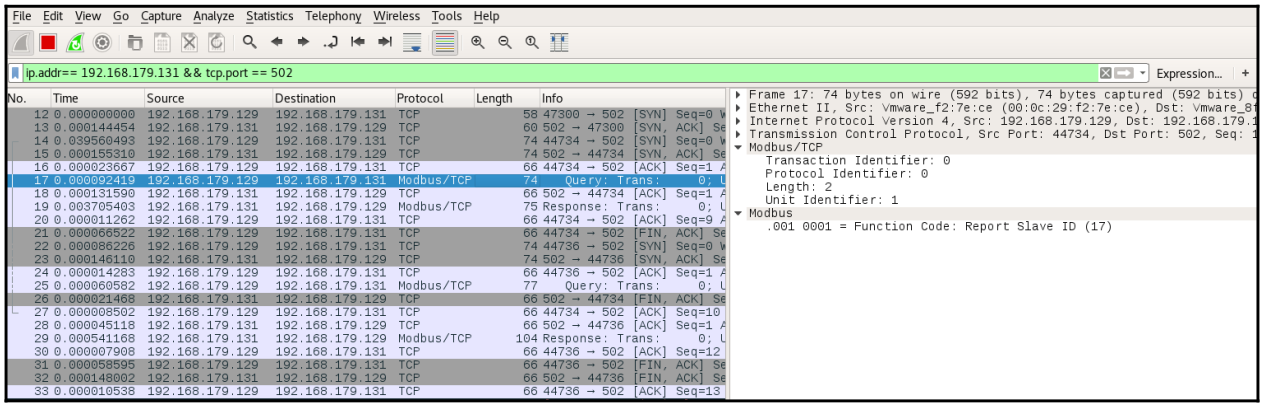
This time around, Nmap picked up the open Modbus port. It identified it as hosting the `mbap` service, which is, if you remember, the ADU header part. But the fun doesn't stop here. Nmap has the ability to run scripts via an NSE script parsing engine. This way, the functionality of Nmap can be extended to include a range of functionality. One such script is `modbus-discover.nse`. This script will interrogate the Modbus server to give up its device information:

```
# nmap 192.168.179.131 -p 502 --script modbus-discover.nse
Starting Nmap 7.40 ( https://nmap.org ) at 2017-05-13 04:51 EDT
Nmap scan report for 192.168.179.131
Host is up (0.00012s latency).
PORT    STATE SERVICE
502/tcp open  Modbus
| Modbus-discover:
|   sid 0x1:
|     error: SLAVE DEVICE FAILURE
|_    Device identification: PyModbus Inc. PM 1.0
MAC Address: 00:0C:29:8F:79:2C (VMware)

Nmap done: 1 IP address (1 host up) scanned in 13.29 seconds
```

Notice something familiar about the `Device identification`? It's the information we gave the Modbus server when we initiated it on the Ubuntu VM:

```
identity.VendorName  = 'PyModbus Inc.'
identity.ProductCode = 'PM'
identity.MajorMinorRevision = '1.0'
```

What made this identification possible are the following two request packets, sent by Nmap to the Modbus server application:



The first request uses function code `17` to request the **Slave ID**:

```
▸ Frame 17: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
▸ Ethernet II, Src: Vmware_f2:7e:ce (00:0c:29:f2:7e:ce), Dst: Vmware_8f:79:2c (00:0c:29:8f:79:2c)
▸ Internet Protocol Version 4, Src: 192.168.179.129, Dst: 192.168.179.131
▸ Transmission Control Protocol, Src Port: 44734, Dst Port: 502, Seq: 1, Ack: 1, Len: 8
▾ Modbus/TCP
    Transaction Identifier: 0
    Protocol Identifier: 0
    Length: 2
    Unit Identifier: 1
▾ Modbus
    .001 0001 = Function Code: Report Slave ID (17)
```

The server responds as follows:

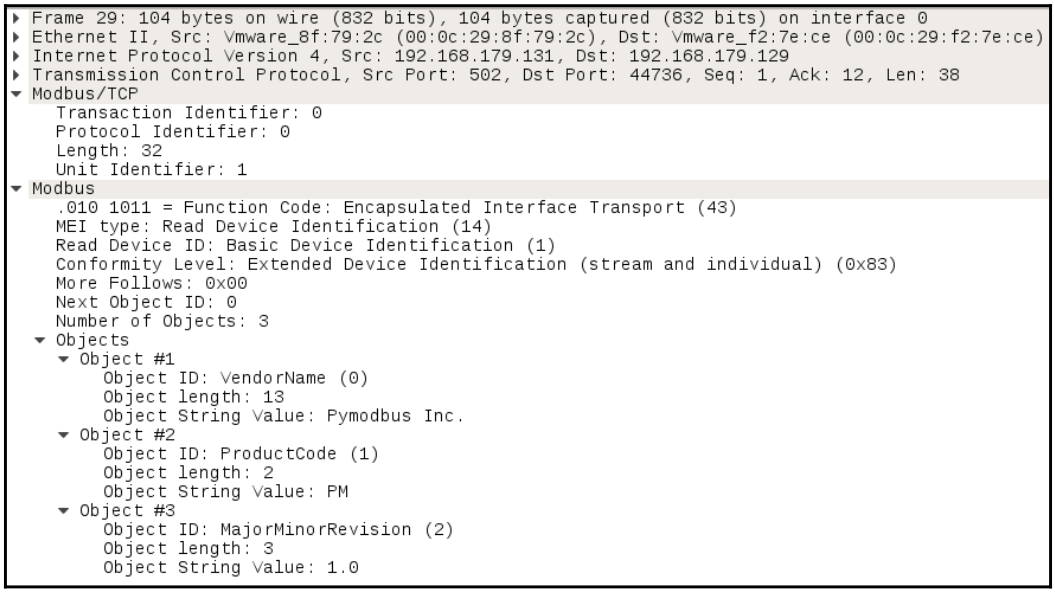
```
▸ Frame 19: 75 bytes on wire (600 bits), 75 bytes captured (600 bits) on interface 0
▸ Ethernet II, Src: Vmware_8f:79:2c (00:0c:29:8f:79:2c), Dst: Vmware_f2:7e:ce (00:0c:29:f2:7e:ce)
▸ Internet Protocol Version 4, Src: 192.168.179.131, Dst: 192.168.179.129
▸ Transmission Control Protocol, Src Port: 502, Dst Port: 44734, Seq: 1, Ack: 9, Len: 9
▾ Modbus/TCP
    Transaction Identifier: 0
    Protocol Identifier: 0
    Length: 3
    Unit Identifier: 1
▾ Function 17:  Report Slave ID.  Exception: Slave device failure
    .001 0001 = Function Code: Report Slave ID (17)
    Exception Code: Slave device failure (4)
```

The server gave an exception response because the Device ID that the `pyModbus` server uses isn't a well-known one.

For the second request, `modbus-cli` uses function code `43` and subcommands `14`, `1`, and `0` to query the server for `VendorName`, `ProductCode`, and `Revision` numbers:

```
▸ Frame 25: 77 bytes on wire (616 bits), 77 bytes captured (616 bits) on interface 0
▸ Ethernet II, Src: Vmware_f2:7e:ce (00:0c:29:f2:7e:ce), Dst: Vmware_8f:79:2c (00:0c:29:8f:79:2c)
▸ Internet Protocol Version 4, Src: 192.168.179.129, Dst: 192.168.179.131
▸ Transmission Control Protocol, Src Port: 44736, Dst Port: 502, Seq: 1, Ack: 1, Len: 11
▾ Modbus/TCP
    Transaction Identifier: 0
    Protocol Identifier: 0
    Length: 5
    Unit Identifier: 1
▾ 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)
```

The Modbus server happily replies to this:



So what? Is what you may think. Well, keep in mind that this information can be obtained by anyone who knows the IP address for the device in question. Armed with the right information, an attacker can now find potential known vulnerabilities for the product. Take, for example, the following Nmap scan output:

```
PORT    STATE SERVICE
502/tcp open  Modbus
| Modbus-discover:
|   sid 0x62:
|     Slave ID data: ScadaTEC
|     Device identification: ModbusTagServer V4.0.1.1
|_
```

Running the device name through a database like the **ICS-CERT** at `https://ics-cert.us-cert.gov/` reveals some potential vulnerabilities to exploit:

There is even a published exploit for this particular vulnerability. Run the following command on your Kali VM:

```
# searchsploit ModbusTagServer
```

The `searchsploit` command queries a local copy of the `exploitdb` (`https://www.exploit-db.com/`) database and returns the location of a published exploit if available. In this case, `searchsploit` returned the following:

```
------------------------------------------------------------ ---------------------------------
 Exploit Title                                              |  Path
                                                            | (/usr/share/exploitdb/platforms/)
------------------------------------------------------------ ---------------------------------
ScadaTEC ModbusTagServer & ScadaPhone - '.zip' Buffer Overf | windows/local/17817.php
------------------------------------------------------------ ---------------------------------
```

```
# cat /usr/share/exploitdb/platforms/windows/local/17817.php
<?php
/*
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ScadaTEC ModbusTagServer & ScadaPhone (.zip) buffer overflow exploit (0day)
Date: 09/09/2011
Author: mr_me (@net__ninja)
Vendor: http://www.scadatec.com/
ScadaPhone Version: <= 5.3.11.1230
ModbusTagServer Version: <= 4.1.1.81
Tested on: Windows XP SP3 NX=AlwaysOn/OptIn
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Notes:
- The ScadaPhone exploit is a DEP bypass under windows XP sp3 only
- The ModbusTagServer exploit does not bypass dep
- To trigger this vulnerability, you must 'load' a project from a zip file.
Feel free to improve it if you want. Example usage:
[mr_me@neptune scadatec]$ php zip.php -t scadaphone
[mr_me@neptune scadatec]$ nc -v 192.168.114.141 4444
Connection to 192.168.114.141 4444 port [tcp/krb524] succeeded!
…
<output cut for brevity>
```

Using Python and Scapy to communicate over Modbus

Modbus-cli is not the only way to query the Modbus server. For this exercise, I will be introducing one of my favorite tools, or more precisely a Python framework, Scapy. Freely available at `http://www.secdev.org/projects/scapy/` but also pre-installed on Kali, Scapy is a powerful interactive packet manipulation tool. It can build, forge, and decode packets for a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. It can handle most classic tasks, such as scanning, tracerouting, probing, unit tests, attacks, or network discovery. It also performs a lot of other specific tasks that most other cookie-cut tools can't do, such as sending invalid frames, injecting your own 802.11 frames, and so on.

We will cover the basics for Scapy here, but for a more in-depth tutorial, you should head over to `http://www.secdev.org/projects/scapy/doc/usage.html#interactive-tutorial`.

As mentioned, Scapy comes preinstalled with Kali Linux, so let's open a Terminal on our Kali VM and start Scapy:

```
# scapy
Welcome to Scapy
>>>
```

Here is an example of building an Ethernet frame (network packet) from scratch:

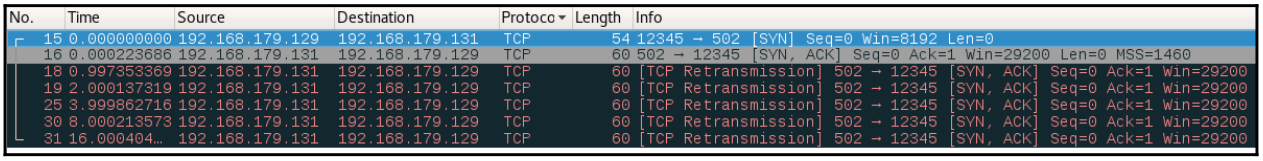
```
>>> ip = IP(src='192.168.179.129', dst='192.168.179.131')
>>> tcp = TCP(sport=12345, dport=502, flags='S')
>>> pkt = ip/tcp
>>> pkt.show()
###[ IP ]###
```

```
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= tcp
  chksum= None
  src= 192.168.179.129
  dst= 192.168.179.131
  \options\
###[ TCP ]###
     sport= 12345
     dport= 502
     seq= 0
     ack= 0
     dataofs= None
     reserved= 0
     flags= SA
     window= 8192
     chksum= None
     urgptr= 0
     options= {}
>>>
```

Now we can send the packet with the `send()` command. (At this point the python Modbus server should be running on the Ubuntu VM):

```
>>> send(pkt)
.
Sent 1 packets.
```

If we look at the packet capture for this action, we can see that the packet we just created got send to the Ubuntu VM (`192.168.179.131`) and got a response as well. As a note, the retransmissions are happening because we started the TCP three-way handshake with setting the `SYN` flag, which caused the Ubuntu server to respond with an `SYN/ACK` packet, and we are neglecting to complete with a final `ACK` packet:



No.	Time	Source	Destination	Protoco	Length	Info
15	0.000000000	192.168.179.129	192.168.179.131	TCP	54	12345 → 502 [SYN] Seq=0 Win=8192 Len=0
16	0.000223686	192.168.179.131	192.168.179.129	TCP	60	502 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460
18	0.997353369	192.168.179.131	192.168.179.129	TCP	60	[TCP Retransmission] 502 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=29200
19	2.000137319	192.168.179.131	192.168.179.129	TCP	60	[TCP Retransmission] 502 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=29200
25	3.999862716	192.168.179.131	192.168.179.129	TCP	60	[TCP Retransmission] 502 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=29200
30	8.000213573	192.168.179.131	192.168.179.129	TCP	60	[TCP Retransmission] 502 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=29200
31	16.000404…	192.168.179.131	192.168.179.129	TCP	60	[TCP Retransmission] 502 → 12345 [SYN, ACK] Seq=0 Ack=1 Win=29200

Instead of using Wireshark to capture the response, we can have Scapy grab it with the `sr1()` command:

```
>>> answer = sr1(pkt)
Begin emission:
.Finished to send 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
>>> answer.show()
###[ IP ]###
  version= 4L
  ihl= 5L
  tos= 0x0
  len= 44
  id= 0
  flags= DF
  frag= 0L
  ttl= 64
  proto= tcp
  chksum= 0x5276
  src= 192.168.179.131
  dst= 192.168.179.129
  \options\
###[ TCP ]###
     sport= 502
     dport= 12345
     seq= 4164488570
     ack= 1
     dataofs= 6L
     reserved= 0L
     flags= SA
     window= 29200
     chksum= 0x5cc
     urgptr= 0
     options= [('MSS', 1460)]
###[ Padding ]###
        load= '\x00\x00'
>>>
```

Notice how the server responds to our request packet with a response that has both the `SYN` and `ACK` TCP flags set, indicating that the port is open. If your response packet shows the flags set to RA (Reset/Ack), indicating the port is closed, verify that the python Modbus server is started on the Ubuntu VM.

Scapy can perform the fuzzing of protocols with a single command:

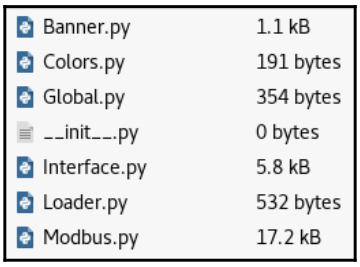
```
>>> send(ip/fuzz(TCP(dport=502)),loop=1)
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.........................................^C^
Sent 780 packets.
```

This will send out packets with a normal IP layer and a TCP layer where all the fields except the destination port (set by `dport=502`) are fuzzed. You can let this command run until it triggers an exception in the destination application, at which point, you can inspect the details behind the exception.

Even though Scapy comes with a tremendous amount of support for all kinds of protocols, there is no default support for the Modbus protocol. So, for example, this isn't supported by default:

```
>>> pkt = ip/tcp/ModbusADU()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
```

Luckily, Python can be extended with the use of modules, so we are going to use some modules created by *enddo* for his Modbus offensive framework, smod, downloadable from `https://github.com/enddo/smod`. After downloading the project archive, extract the contents of the `smod-master/System/Core` directory into a newly created directory `/usr/lib/python2.7/dist-packages/Modbus/` on the Kali Linux VM:



With these files in place, we can now import the necessary modules that contain the required classes to craft and dissect Modbus packets:

```
>>> from Modbus.Modbus import *
>>> pkt = ip/tcp/ModbusADU()
>>> pkt.show()
###[ IP ]###
  version= 4
```

```
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= tcp
  chksum= None
  src= 192.168.179.129
  dst= 192.168.179.131
  \options\
###[ TCP ]###
     sport= 12345
     dport= 502
     seq= 0
     ack= 0
     dataofs= None
     reserved= 0
     flags= S
     window= 8192
     chksum= None
     urgptr= 0
     options= {}
###[ ModbusADU ]###
        transId= 0x1
        protoId= 0x0
        len= None
        unitId= 0x0
>>>
```

Remember that depending on what function code we are sending, the PDU layer will vary. So let's see what we have available to our disposal. Type in the following command, ending with a double tab:

```
>>> pkt = ip/tcp/ModbusADU()/ModbusPDU
## TAB-TAB
ModbusPDU01_Read_Coils
ModbusPDU04_Read_Input_Registers_Exception
ModbusPDU0F_Write_Multiple_Coils_Answer
ModbusPDU01_Read_Coils_Answer
ModbusPDU05_Write_Single_Coil
ModbusPDU0F_Write_Multiple_Coils_Exception
ModbusPDU01_Read_Coils_Exception
ModbusPDU05_Write_Single_Coil_Answer
ModbusPDU10_Write_Multiple_Registers
ModbusPDU02_Read_Discrete_Inputs
ModbusPDU05_Write_Single_Coil_Exception
```

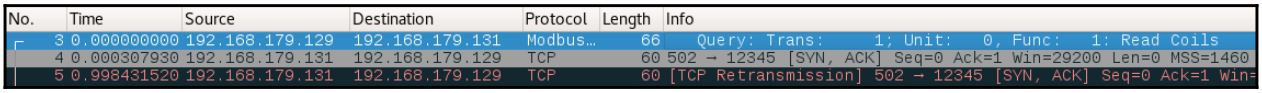
```
ModbusPDU10_Write_Multiple_Registers_Answer
ModbusPDU02_Read_Discrete_Inputs_Answer
ModbusPDU06_Write_Single_Register
ModbusPDU10_Write_Multiple_Registers_Exception
ModbusPDU02_Read_Discrete_Inputs_Exception
ModbusPDU06_Write_Single_Register_Answer
ModbusPDU11_Report_Slave_Id
ModbusPDU03_Read_Holding_Registers
ModbusPDU06_Write_Single_Register_Exception
ModbusPDU11_Report_Slave_Id_Answer
ModbusPDU03_Read_Holding_Registers_Answer
ModbusPDU07_Read_Exception_Status
ModbusPDU11_Report_Slave_Id_Exception
ModbusPDU03_Read_Holding_Registers_Exception
ModbusPDU07_Read_Exception_Status_Answer
ModbusPDU_Read_Generic
ModbusPDU04_Read_Input_Registers
ModbusPDU07_Read_Exception_Status_Exception
>>> pkt = ip/tcp/ModbusADU()/ModbusPDU
```

These are the various PDU formats that the smod framework modules added to Scapy. Let's start with a simple one and select the one for function code `1`, `ModbusPDU01_Read_Coils`:

```
>>> pkt = ip/tcp/ModbusADU()/ModbusPDU01_Read_Coils()
>>> pkt[ModbusADU].show()
###[ ModbusADU ]###
  transId= 0x1
  protoId= 0x0
  len= None
  unitId= 0x0
###[ Read Coils Request ]###
     funcCode= 0x1
     startAddr= 0x0
     quantity= 0x1
```

Let's send this packet and see what happens:

```
>>> send(pkt)
```



Looking at the individual layers of the packet, it seems that our Scapy fabricated packet did the job. All packet fields are filled in with the correct data to get the packet out of the NIC and into the right direction:

```
▸ Frame 3: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
▸ Ethernet II, Src: Vmware_f2:7e:ce (00:0c:29:f2:7e:ce), Dst: Vmware_8f:79:2c (00:0c:29:8f:79:2c)
▾ Internet Protocol Version 4, Src: 192.168.179.129, Dst: 192.168.179.131
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 52
    Identification: 0x0001 (1)
  ▸ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x926d [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.179.129
    Destination: 192.168.179.131
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▾ Transmission Control Protocol, Src Port: 12345, Dst Port: 502, Seq: 0, Len: 12
    Source Port: 12345
    Destination Port: 502
    [Stream index: 0]
    [TCP Segment Len: 12]
    Sequence number: 0    (relative sequence number)
    [Next sequence number: 13    (relative sequence number)]
    Acknowledgment number: 0
    Header Length: 20 bytes
  ▾ Flags: 0x002 (SYN)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...0 .... = Acknowledgment: Not set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    ▾ .... .... ..1. = Syn: Set
      ▾ [Expert Info (Chat/Sequence): Connection establish request (SYN): server port 502]
          [Connection establish request (SYN): server port 502]
          [Severity level: Chat]
          [Group: Sequence]
      .... .... ...0 = Fin: Not set
      [TCP Flags: ··········S·]
    Window size value: 8192
    [Calculated window size: 8192]
    Checksum: 0x7548 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  ▾ [SEQ/ACK analysis]
      [Bytes in flight: 13]
      [Bytes sent since last PSH flag: 12]
    [PDU Size: 12]
▾ Modbus/TCP
    Transaction Identifier: 1
    Protocol Identifier: 0
    Length: 6
    Unit Identifier: 0
▾ Modbus
    .000 0001 = Function Code: Read Coils (1)
    Reference Number: 0
    Bit Count: 1
```

The reason we are not getting a response is that the Modbus protocol needs an established TCP connection to work. We just send out a random packet with the `SYN` flag set. If you look at the packet after ours, the Ubuntu network stack is responding with an `SYN/ACK` packet, the second step in the three-way handshake that establishes a TCP connection. We will need to add connection support to our Modbus packet forgery efforts.

For that purpose, I wrote the following script, which will establish a connection, send the Modbus request packet, and display the response. Open the text editor of your choice and type in the following script:

```
from scapy.all import *
from Modbus.Modbus import *
import time

# Defining the script variables
srcIP   = '192.168.179.129'
srcPort = random.randint(1024, 65535)
dstIP   = '192.168.179.131'
dstPort = 502
seqNr   = random.randint(444, 8765432)
ackNr   = 0
transID = random.randint(44,44444)

def updateSeqAndAckNrs(sendPkt, recvdPkt):
    # Keeping track of tcp sequence and acknowledge numbers
    global seqNr
    global ackNr
    seqNr = seqNr + len(sendPkt[TCP].payload)
    ackNr = ackNr + len(recvdPkt[TCP].payload)

def sendAck():
    # Create the acknowledge packet
    ip      = IP(src=srcIP, dst=dstIP)
    ACK     = TCP(sport=srcPort, dport=dstPort, flags='A',
                  seq=seqNr, ack=ackNr)
    pktACK  = ip / ACK

    # Send acknowledge packet
    send(pktACK)

def tcpHandshake():
    # Establish a connection with the server by means of the tcp
    # three-way handshake
    # Note: linux might send an RST for forged SYN packets.Disable it by
executing:
    # > iptables -A OUTPUT -p tcp --tcp-flags RST RST -s <src_ip> -j DROP
    global seqNr
```

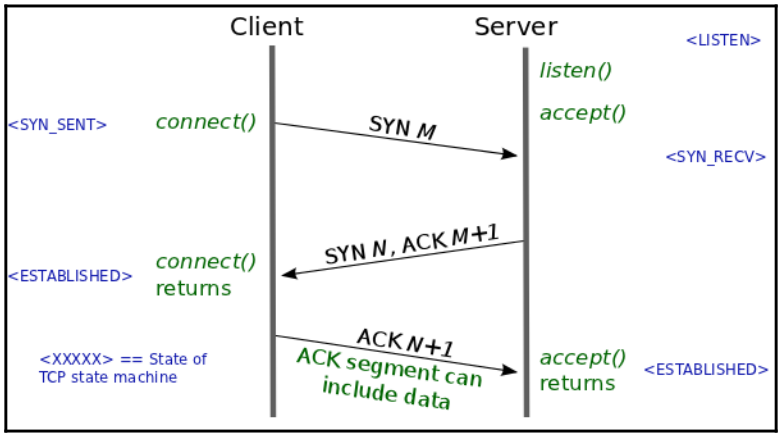
```
    global ackNr

    # Create SYN packet
    ip      = IP(src=srcIP, dst=dstIP)
    SYN     = TCP(sport=srcPort, dport=dstPort, flags='S',
                  seq=seqNr, ack=ackNr)
    pktSYN  = ip / SYN

    # send SYN packet and receive SYN/ACK packet
    pktSYNACK = sr1(pktSYN)

    # Create the ACK packet
    ackNr   = pktSYNACK.seq + 1
    seqNr   = seqNr + 1
    ACK     = TCP(sport=srcPort, dport=dstPort, flags='A', seq=seqNr,
ack=ackNr)
    send(ip / ACK)
    return ip/ACK

def endConnection():
    # Create the rst packet
    ip = IP(src=srcIP, dst=dstIP)
    RST = TCP(sport=srcPort, dport=dstPort, flags='RA',
              seq=seqNr, ack=ackNr)
    pktRST = ip / RST

    # Send acknowledge packet
    send(pktRST)

def connectedSend(pkt):
    # Update packet's sequence and acknowledge numbers
    # before sending
    pkt[TCP].flags  = 'PA'
    pkt[TCP].seq    = seqNr
    pkt[TCP].ack    = ackNr
    send(pkt)

# First we establish a connection. The packet returned by the
# function contains the connection parameters
ConnectionPkt = tcpHandshake()

# With the connection packet as a base, create a Modbus
# request packet to read coils
ModbusPkt = ConnectionPkt/ModbusADU()/ModbusPDU01_Read_Coils()

# Set the function code, start and stop registers and define
# the Unit ID
ModbusPkt[ModbusADU].unitId = 1
```

```
ModbusPkt[ModbusPDU01_Read_Coils].funcCode = 1
ModbusPkt[ModbusPDU01_Read_Coils].quantity = 5

# As an example, send the Modbus packet 5 times, updating
# the transaction ID for each iteration
for i in range(1, 6):
    # Create a unique transaction ID
    ModbusPkt[ModbusADU].transId = transID + i*3
    ModbusPkt[ModbusPDU01_Read_Coils].startAddr = random.randint(0, 65535)

    # Send the packet
    connectedSend(ModbusPkt)

    # Wait for response packets and filter out the Modbus response packet
    Results = sniff(count=1, filter='tcp[tcpflags] & (tcp-push|tcp-ack) !=
0')
    ResponsePkt = Results[0]
    updateSeqAndAckNrs(ModbusPkt, ResponsePkt)
    ResponsePkt.show()
    sendAck()

endConnection()
```

Let's look at that script in sections. After importing the necessary modules and defining the variables and constants used by the script, it defines the following functions:

- The `updateSeqAndAckNrs` function updates the TCP session related **Sequence and Acknowledgment** counters, which effectively keeps the TCP connection alive and in sync. If these fields are off in a packet, the packet is discarded at the receiving end as is not considered part of the connection.
- The `sendAck` function is a helper function to acknowledge a received packet from the sending application.
- The `endConnection` function does just what it says; it ends the connection by sending an RST packet. It's not the most elegant method in the world, but it's very effective.
- The `connectedSend` function can send a packet within the scope of a TCP connection. It does that by updating the sequence and acknowledge fields to the current ones for the connection, setting the TCP flags, and then sending the packet on its way.

- The `tcpHandshake` function sets up a connection with the server by going through the TCP three-way handshake, used for connection establishment.

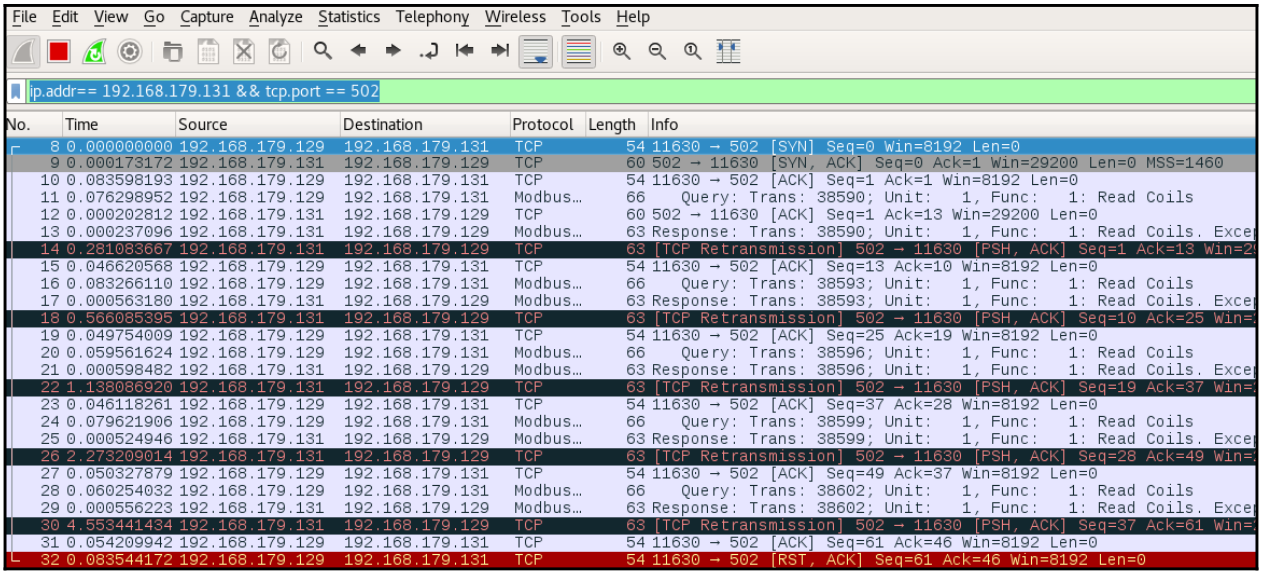


The TCP three-way handshake starts with the client sending a TCP packet with the `SYN` flag set. The server then responds with an `SYN/ACK` packet and the client seals the deal by responding to that with an `ACK` packet. When the handshake is done, both the clients have synchronized their sequence and acknowledge numbers and are ready to communicate until either side closes the connection by sending a `FIN` or `RST` packet.

Looking at the main function of the script, we see that after establishing the connection via `ConnectionPkt = tcpHandshake()`, the script uses the returned connected packet, extends it with `ModbusADU`, and `ModbusDU` layers and sets the `ModbusADU` and `ModbusPDU` variables, such as function code and data fields:

```
ModbusPkt = ConnectionPkt/ModbusADU()/ModbusPDU01_Read_Coils()
ModbusPkt[ModbusADU].unitId = 1
ModbusPkt[ModbusPDU01_Read_Coils].funcCode = 1
ModbusPkt[ModbusPDU01_Read_Coils].startAddr = 1
ModbusPkt[ModbusPDU01_Read_Coils].quantity = 5
```

At this point, the entire Modbus packet is available with Python, which means that any field, any value is changeable and open for fuzzing, iterations, and manipulation. It is up to the programmer's imagination what to do with this functionality. As an example, the script sends five request packets to the Modbus server with a random start address. Here is the result of our hard work:

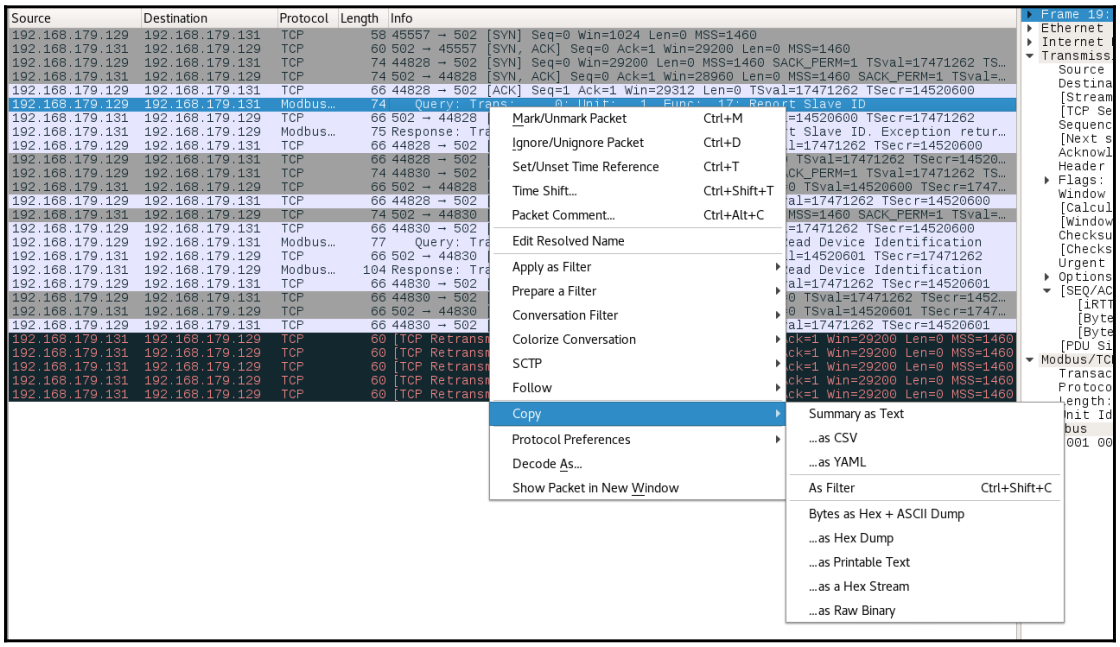


Replaying captured Modbus packets

As a last exercise before we close the book on Modbus, I am going to show how Scapy can use a captured packet as a starting point. If you recall the Nmap script, `Modbus-discover.nse` will send packets containing the function codes that request the Modbus server for information. Let's run the script again and capture the packets with Wireshark:

```
# nmap 192.168.179.131 -p 502 --script Modbus-discover.nse
```

It is the first request packet that we are going to use; right-click on it and go to **Copy** | **...as a Hex Stream**:



Now start Scapy in a Terminal and run the following command:

```
>>> from Modbus.Modbus import *
>>> import binascii # Necessary library to convert the copied hex stream
>>> raw_pkt = binascii.unhexlify(‘
At this point, right-mouse click on the terminal and paste the copied
packet hex string into the terminal
>>> raw_pkt =
binascii.unhexlify('000c298f792c000c29f27ece08004500003ce2e7400040066f7ec0a
8b381c0a8b383af1c01f6e6d975fc2a2d1419801800e5e88400000101080a010a971e00dd91
180000000000020111')
This command converts the copied hex stream version of the packet to a
binary form
```

Continue with the following command:

```
>>> Modbus_pkt = Ether(raw_pkt)  # Convert raw binary blob to Scapy packet
starting at EtherNet
>>> Modbus_pkt.show()
###[ Ethernet ]###
  dst= 00:0c:29:8f:79:2c
  src= 00:0c:29:f2:7e:ce
  type= 0x800
```

```
###[ IP ]###
     version= 4L
     ihl= 5L
     tos= 0x0
     len= 60
     id= 58087
     flags= DF
     frag= 0L
     ttl= 64
     proto= tcp
     chksum= 0x6f7e
     src= 192.168.179.129
     dst= 192.168.179.131
     \options\
###[ TCP ]###
        sport= 44828
        dport= 502
        seq= 3873011196
        ack= 707597337
        dataofs= 8L
        reserved= 0L
        flags= PA
        window= 229
        chksum= 0xe884
        urgptr= 0
        options= [('NOP', None), ('NOP', None), ('Timestamp', (17471262,
14520600))]
###[ ModbusADU ]###
           transId= 0x0
           protoId= 0x0
           len= 0x2
           unitId= 0x1
###[ Report Slave Id ]###
              funcCode= 0x11
```

Note how the captured packet is neatly identified as a Modbus request, with ADU and PDU dissected. At this point, we can change any field within the packet and send it over a TCP connection with the Modbus server.

This was a long haul. We covered a lot of ground and I touched on a lot of subjects. The takeaway of all this is that the one true vulnerability of the Modbus protocol and quite frankly, most—if not all industrial protocols—is that it is open. The Modbus request and response packets are sent unencrypted and are easy to intercept, modify, and replay. The reason the protocol is implemented this way is that it was originally designed to run on low speed, proprietary media, which wasn't supposed to be shared by non-Modbus devices or protocols.