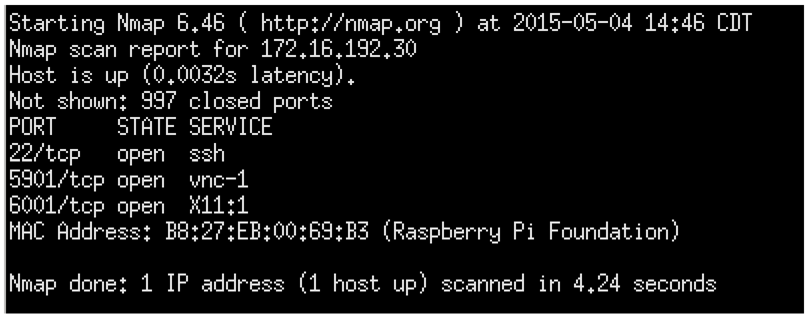
Title: Turning Traffic Bad

Subtitle: Using the network and some bad logic to gridlock a city

Authors: Joe Barrett and Steve Walker, Foreground Security

Recently, your authors had the opportunity to take SANS ICS515 as taught by Rob Lee and received copies of the Cybati kit for simulating control systems. While the class was very educational and a lot of fun, we both preferred to put on 'evil' hats and take a closer look at vulnerabilities in the traffic light systems as opposed to concentrating on network monitoring. Our goal was to look at the individual HMI, PLC, and light system as part of a larger scenario and come up with potential avenues that an attacker might attempt to use to compromise the system for their own goal. One such potential use case is to turn the lights all green to cause a grid lock, as seen in [this feature movie clip](https://www.youtube.com/watch?v=ybYQsVnh9QA).

To begin with, we scanned the HMI system (our Cybati VM) as well as the PLC (the Raspberry Pi board attached to the Cybati kit), looking for potential avenues of attack that would translate to the real world.

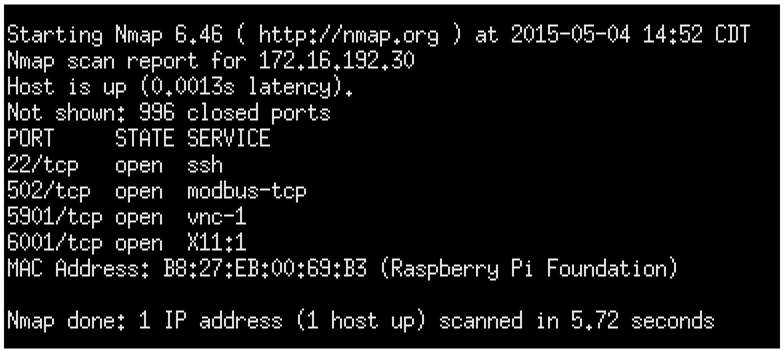


```
Starting Nmap 6.46 ( http://nmap.org ) at 2015-05-04 14:46 CDT
Nmap scan report for 172.16.192.30
Host is up (0.0032s latency).
Not shown: 997 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
5901/tcp open  vnc-1
6001/tcp open  X11:1
MAC Address: B8:27:EB:00:69:B3 (Raspberry Pi Foundation)

Nmap done: 1 IP address (1 host up) scanned in 4.24 seconds
```

Figure 1 - Exposed Ports Before Configuring Logic

Before we load the control system logic into it, we can see that the open ports are SSH, VNC, and X11. While we would prefer not to see VNC and X11 on an Internet connected host, that's a topic for a different blog post. So we went ahead and fired up RexDraw and loaded the sample control system logic to see how the scenario changed. As you can see, once we had the logic loaded, the Raspberry Pi PLC was now listening on Modbus/TCP for incoming connections.



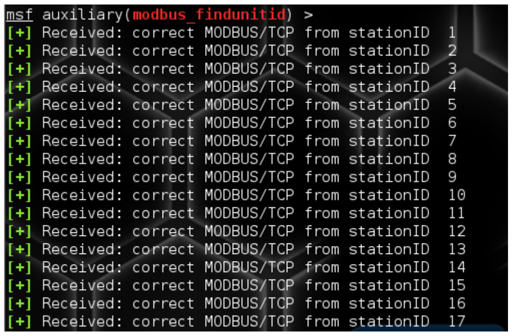
```
Starting Nmap 6.46 ( http://nmap.org ) at 2015-05-04 14:52 CDT
Nmap scan report for 172.16.192.30
Host is up (0.0013s latency).
Not shown: 996 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
502/tcp  open  modbus-tcp
5901/tcp open  vnc-1
6001/tcp open  X11:1
MAC Address: B8:27:EB:00:69:B3 (Raspberry Pi Foundation)

Nmap done: 1 IP address (1 host up) scanned in 5.72 seconds
```

Figure 2 - Exposed Ports After Configuring Logic

Attacking Modbus

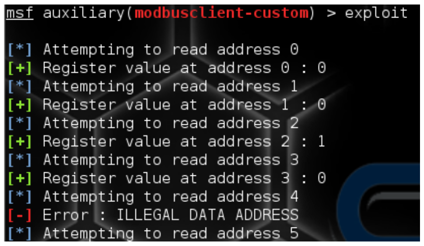
Through our experience, we have seen numerous examples of Modbus exposed to the business network. It is a great port to demonstrate this capability through, as attackers can perform a variety of actions directly on the PLCs using Modbus, while defenders can monitor these connections to quickly identify abnormal activity. To figure out what we could accomplish with Modbus, we began by using some open source tools to investigate the PLC.



```
msf auxiliary(modbus_findunitid) >
[+] Received: correct MODBUS/TCP from stationID  1
[+] Received: correct MODBUS/TCP from stationID  2
[+] Received: correct MODBUS/TCP from stationID  3
[+] Received: correct MODBUS/TCP from stationID  4
[+] Received: correct MODBUS/TCP from stationID  5
[+] Received: correct MODBUS/TCP from stationID  6
[+] Received: correct MODBUS/TCP from stationID  7
[+] Received: correct MODBUS/TCP from stationID  8
[+] Received: correct MODBUS/TCP from stationID  9
[+] Received: correct MODBUS/TCP from stationID  10
[+] Received: correct MODBUS/TCP from stationID  11
[+] Received: correct MODBUS/TCP from stationID  12
[+] Received: correct MODBUS/TCP from stationID  13
[+] Received: correct MODBUS/TCP from stationID  14
[+] Received: correct MODBUS/TCP from stationID  15
[+] Received: correct MODBUS/TCP from stationID  16
[+] Received: correct MODBUS/TCP from stationID  17
```

Figure 3 - Metasploit's modbus_findunitid Plugin - Not Terribly Useful

After striking out using some of the basic tools, we created a modified version of Metasploit's modbusclient plugin that would enable us to read a large range of registers on the PLC at once. We started by trying to read registers 0 through 10 as a sample space.



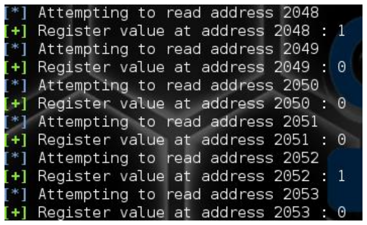
```
msf auxiliary(modbusclient-custom) > exploit

[*] Attempting to read address 0
[+] Register value at address 0 : 0
[*] Attempting to read address 1
[+] Register value at address 1 : 0
[*] Attempting to read address 2
[+] Register value at address 2 : 1
[*] Attempting to read address 3
[+] Register value at address 3 : 0
[*] Attempting to read address 4
[-] Error : ILLEGAL DATA ADDRESS
[*] Attempting to read address 5
```

Figure 4 - Custom Modbus Client Reading a Range of Registers

Through our knowledge of the HMI, we knew that these four registers corresponded to the four operations possible (Force E-W Intersection, Force N-S Intersection, Automatic, and Blink). The value of '1' at Register 2 corresponded with the 'Automatic' operation mode that we knew the PLC was currently configured to utilize.

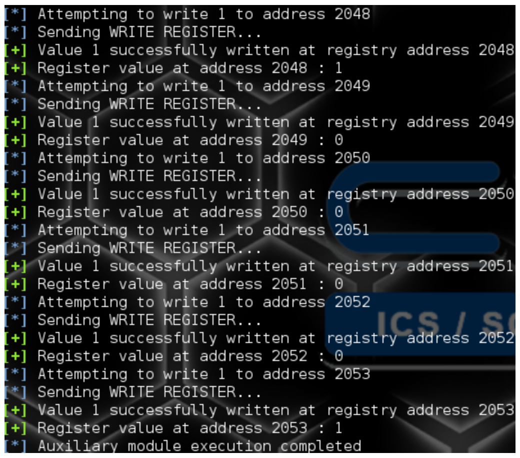
Metasploit also gave us the capability to write values back into the Modbus registers, so we spent some time forcing the various HMI operations over the network. While this was entertaining and demonstrated some of the risk from exposing Modbus/TCP, we wanted to go farther with our ability to cause harm to the traffic system. We continued to scan the range of all possible registers in case there were other options we were missing.



```
[*] Attempting to read address 2048
[+] Register value at address 2048 : 1
[*] Attempting to read address 2049
[+] Register value at address 2049 : 0
[*] Attempting to read address 2050
[+] Register value at address 2050 : 0
[*] Attempting to read address 2051
[+] Register value at address 2051 : 0
[*] Attempting to read address 2052
[+] Register value at address 2052 : 1
[*] Attempting to read address 2053
[+] Register value at address 2053 : 0
```

Figure 5 - Discovering Further Registers

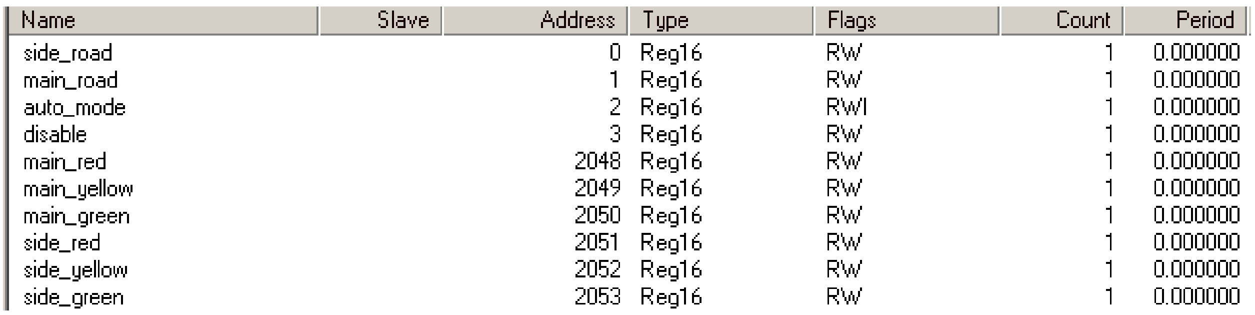
We discovered an additional set of readable registers (2048 through 2053) that appeared to correspond to the six lights (main green, main yellow, main red, side green, side yellow, and side red). We were able to watch in real-time as those values updated in conjunction with the lights on the Cybati board, so we knew that we were starting down the right path. We then attempted to write values back into those registers using the same technique that we had used above to manipulate the HMI functions.



```
[*] Attempting to write 1 to address 2048
[*] Sending WRITE REGISTER...
[+] Value 1 successfully written at registry address 2048
[+] Register value at address 2048 : 1
[*] Attempting to write 1 to address 2049
[*] Sending WRITE REGISTER...
[+] Value 1 successfully written at registry address 2049
[+] Register value at address 2049 : 0
[*] Attempting to write 1 to address 2050
[*] Sending WRITE REGISTER...
[+] Value 1 successfully written at registry address 2050
[+] Register value at address 2050 : 0
[*] Attempting to write 1 to address 2051
[*] Sending WRITE REGISTER...
[+] Value 1 successfully written at registry address 2051
[+] Register value at address 2051 : 0
[*] Attempting to write 1 to address 2052
[*] Sending WRITE REGISTER...
[+] Value 1 successfully written at registry address 2052
[+] Register value at address 2052 : 0
[*] Attempting to write 1 to address 2053
[*] Sending WRITE REGISTER...
[+] Value 1 successfully written at registry address 2053
[+] Register value at address 2053 : 1
[*] Auxiliary module execution completed
```

Figure 6 - Attempting to Write to Light Registers

As can be seen, our attempts to write '1' values to enable the lights were unsuccessful. Even though Metasploit returned successfully after writing the value to a register, a subsequent read operation revealed that the value had not actually changed. Our first thought was that the registers themselves were marked as 'Read-Only' somehow, so we began to investigate the logic files.



Name	Slave	Address	Type	Flags	Count	Period
side_road		0	Reg16	RW	1	0.000000
main_road		1	Reg16	RW	1	0.000000
auto_mode		2	Reg16	RWI	1	0.000000
disable		3	Reg16	RW	1	0.000000
main_red		2048	Reg16	RW	1	0.000000
main_yellow		2049	Reg16	RW	1	0.000000
main_green		2050	Reg16	RW	1	0.000000
side_red		2051	Reg16	RW	1	0.000000
side_yellow		2052	Reg16	RW	1	0.000000
side_green		2053	Reg16	RW	1	0.000000

Figure 7 - Register Configuration in RexDraw

All of the registers were configured as read-write registers, so our attempts to modify the values should have worked, so we began to investigate further. As it turns out, we had to look at the actual program logic to determine why our modifications failed.

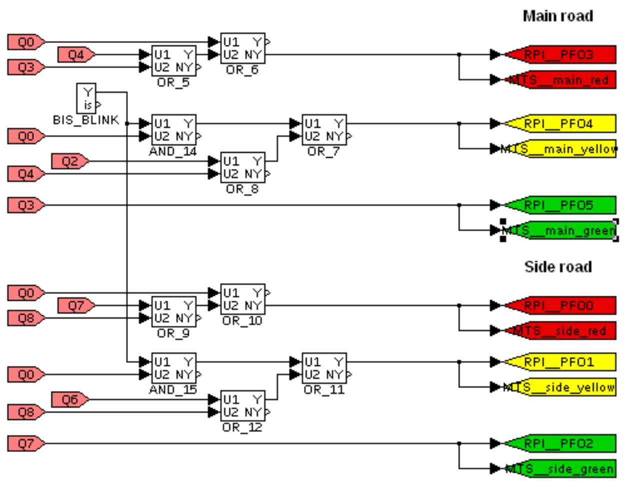


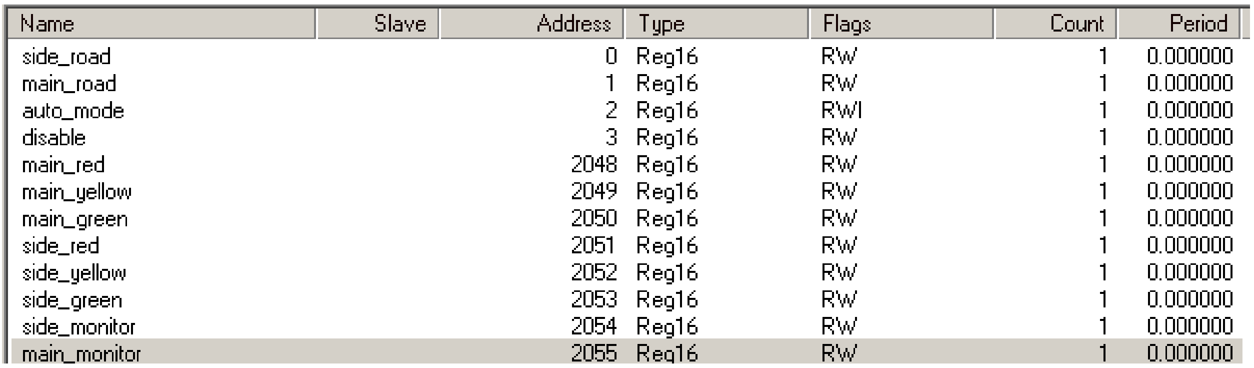
Figure 8 - Input and Output Configuration for Lights

Looking at the program logic in RexDraw, the only times that the MTS__side_green and MTS__main_green registers are used (which correspond to the Modbus/TCP registers we attempted to write to) are as output values. This perfectly explains the behavior that we had seen when attempting to write to the registers; we put data in, but it was immediately overridden with the current state of the lights and our written value had no impact. At this point, we decided that our attackers would need to do something a little bit more dangerous in order to achieve their desired end goal – they would have to modify the program logic on the engineer's workstation and hope to remain unnoticed.

Making Evil Traffic

To meet this goal, we first had to ensure that our evil program logic performed all of the valid operations exactly as expected, so that a system operator never noticed anything unusual until we activated our evil routine. Our second requirement was to ensure that our modifications to the program logic were as minimal as possible and blended in with the legitimate logic so that our changes were not obvious upon cursory examination. Finally, we had to ensure that we could easily trigger our 'attack scenario' over the network upon command to accomplish our evil goals.

First, we created two new Modbus/TCP registers that were in sequence with the existing registers for the traffic lights. We came up with a name that would seem fairly innocuous at first glance and configured the registers identical to the existing ones. A more accurate name for what we are implementing would be an 'override' register.



Name	Slave	Address	Type	Flags	Count	Period
side_road		0	Reg16	RW	1	0.000000
main_road		1	Reg16	RW	1	0.000000
auto_mode		2	Reg16	RWI	1	0.000000
disable		3	Reg16	RW	1	0.000000
main_red		2048	Reg16	RW	1	0.000000
main_yellow		2049	Reg16	RW	1	0.000000
main_green		2050	Reg16	RW	1	0.000000
side_red		2051	Reg16	RW	1	0.000000
side_yellow		2052	Reg16	RW	1	0.000000
side_green		2053	Reg16	RW	1	0.000000
side_monitor		2054	Reg16	RW	1	0.000000
main_monitor		2055	Reg16	RW	1	0.000000

Figure 9 - Adding the Two "monitor" Registers

Once the registers were created, we needed to modify the program logic to include them as inputs to the traffic lights. To accomplish this, we needed to create a few more "OR" blocks and include our new registers on the input side of the diagram which we also modified so that the colors were similar to the nearby logic blocks. In this small example, the difference remains obvious. However, if the viewer were unfamiliar with the original program logic, it would take some time to discover what was added and why.

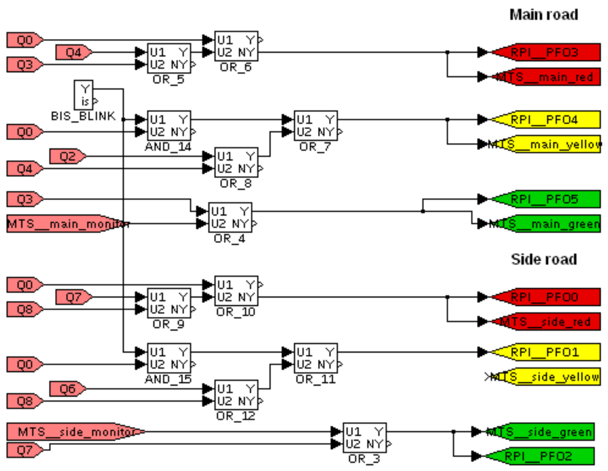


Figure 10 - New Program Logic

With this new program logic in place and uploaded to our PLC, the traffic lights continue to operate exactly as expected. If an operator were to view the HMI for the PLC, it again shows the traffic lights exactly as it should and all operations (force E-W intersection, force N-S intersection, automatic, and blink) continue to operate as normal.

However, when our attacker needs to generate a plot-saving city-wide traffic jam, we can gain access to the control system network and send our malicious commands to the Modbus/TCP interfaces on the various PLCs to force our traffic lights to all turn green. Due to the way that the program logic is structured and the fact that we wanted to keep the number of modifications minimal, our command forces the lights to green but the red and yellow lights continue to cycle as normal. This creates a somewhat confusing scenario where a driver looking at a light will see both the green and red lights illuminated at times, but chaos and confusion is our goal. Plus, when the light cycles a minute later, the only light they'll see is green. Similarly, the traffic operators viewing the HMI will also see the multiple active lights on their console.

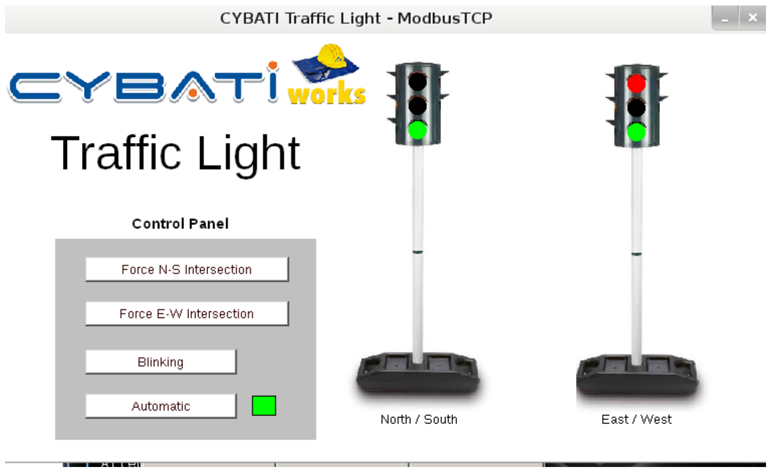


Figure 11 - HMI During Malicious Operation

To recover from this malicious event, the attacker needs to reset the two "monitor" or override registers to 0 (clearing the "OR" blocks to allow the normal logic tree to complete) or else the defenders need to upload the original program logic onto the PLCs. Until either event takes place, the traffic lights will be forced green regardless of any action the system operators take on the HMI.

Conclusions

Based on this example attack scenario, we wanted to highlight a few notes for defenders. A defender could take several steps to detect, prevent, or recover from this attack. Each of these concepts were covered during this defensive focused course. Putting these defensive concepts into practice is the most important take-away from this scenario.

First, as recovery from an incident is the priority, the most direct way to recover from the attack would be to upload known-good logic files from offline back-ups rather than try to decipher an attacker's obfuscated malware or logic file (that comes later). And since you have certified logic files archived (right)? you are prepared for this event!

Second, ensuring your network architecture is structured appropriately will help to detect attacks and defend your PLCs. Building your control system network in such a way as to limit the access to those PLCs will help create virtual choke points in your network environment. Once you have this in place, you can easily augment with network security monitoring to watch for anomalous traffic.

In our attack scenario, a network defender who was looking for unusual network traffic should have easily been able to spot the write commands to the PLCs. How often are write commands sent in your control system environment on a regular basis? We'd wager it's probably not often or it's only in response to documented events. A basic intrusion detection system monitoring rule would have flagged the attacker's initial testing and the eventual triggering of the new override registers.

Finally, we'd like to point out that this attack scenario could potentially have been prevented if the defenders used change management or file hashing to monitor their project files. We're assuming that the attacker didn't directly upload the code onto the PLC (you're capturing event logs from your PLCs to look for those events, right? Of course you are!) and instead modified an at-rest project file and let the system operators upload it into the control system as part of routine work. If the defenders had an automated system to watch the file hashes of the project files, they would know in real-time whenever a modification was made – and modifications should only be made in response to legitimate change requests. Any unauthorized modification is either an attacker or someone making changes they're not supposed to – and that can cause just as much damage as the real bad guys.

Thanks to Rob Lee for teaching ICS515 and indulging us while we played with the Cybati kits as he taught and thanks to Matthew Luallen for creating the Cybati kits that were so much fun to mess with!