EMULATING INDUSTRIAL CONTROL SYSTEM FIELD DEVICES USING GUMSTIX TECHNOLOGY

THESIS

Dustin J. Berman

AFIT/GCO/ENG/12-13

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT/GCO/ENG/12-13

EMULATING INDUSTRIAL CONTROL SYSTEM FIELD DEVICES USING GUMSTIX TECHNOLOGY

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Dustin J. Berman

June 2012

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCO/ENG/12-13

EMULATING INDUSTRIAL CONTROL SYSTEM FIELD DEVICES USING GUMSTIX TECHNOLOGY

Dustin J. Berman

Approved:

Maj Jonathan W. Butts, PhD (Chairman)

21 May 2012
Date

Barry Mullins
Dr. Barry E. Mullins, PhD (Member)

21 May 12
Date

Mr. Juan Lopez Jr. (Member)

21 May 2012
Date

AFIT/GCO/ENG/12-13

Abstract

Industrial Control Systems (ICS) have an inherent lack of security and situational awareness capabilities at the field device level. Yet these systems comprise a significant portion of the nation's critical infrastructure. Currently, there is little insight into the characterization of attacks on ICS. Stuxnet provided an initial look at the type of tactics that can be employed to create physical damage via cyber means. The question still remains, however, as to the extent of malware and attacks that are targeting the critical infrastructure, along with the various methods employed to target systems associated with the ICS environment.

This research presents a device using Gumstix technology that emulates an ICS field device. The emulation device is low-cost, adaptable to myriad ICS environments and provides logging capabilities at the field device level. The device was evaluated to ensure conformity to RFC standards through the use of Triangle MicroWorks and that the operating characteristics are consistent with actual field devices. The device was also evaluated in that the device can respond as a PLC to common fingerprinting techniques. The device was able to respond according to RFC standards and does respond as a valid PLC to common fingerprinting techniques.

Acknowledgments

Many thanks go out to my wife who put up with the late nights and early mornings. I would also like to thank my advisor Maj Jonathan Butts who dedicated much of his time to help me accomplish both my academic and personal goals. I would also like to thank Mr. Juan Lopez Jr. for recommending additional ways to scope my thesis and also want to thank Dr. Barry Mullins for helping me learn a lot of what I know today.

Dustin J. Berman

Table of Contents

Page

Abstract iv

Acknowledgments v

Table of Contents vi

List of Figures ix

List of Tables xi

I. Introduction 1

1.1 Problem Definition 1
1.2 Goals 2
1.3 Scope and Limitations 3
1.4 Organization 3

II. Background 5

2.1 Overview 5
2.2 Industrial Control Systems Background 5
2.3 Modbus Protocol 8
2.4 Critical Infrastructure Protection 11
2.5 Industrial Control Systems Security 13
2.6 ICS Security Mechanisms 15
2.7 Network Attack 17
2.7.1 Fingerprinting 17
2.8 Emulation 19
2.8.1 Honeypot Overview 20
2.8.2 Advantages of Honeypots 21
2.8.3 Disadvantages of Honeypots 22
2.8.4 Honeypot Attributes 22
2.8.5 Honeypot Technology in IT 23
2.9 Honeypots in ICS 27
2.9.1 Current Honeypots in ICS 28
2.9.2 Emulation Requirements 29
2.10 Summary 30

III. Methodology 31

3.1 Problem Definition 31
3.1.1 Goals and Hypothesis 31

3.1.2 Approach 31
3.2 Environment...32
3.3 Evaluation Technique...34
3.3.1 Functionality Test through Modbus Traffic Emulation 34
3.3.2 Fingerprinting Test Cases.. 36
3.3.3 Invalid Traffic Test Cases 40
3.3.4 Logging Capabilities 40
3.3.5 Qualitative Evaluation... 41
3.4 Methodology Summary...42

IV. Analysis and Results...43

4.1 Development of emulated PLC...43
4.1.1 Architecture.. 43
4.1.2 Implementation Details 44
4.2 Emulated PLC Initialization Checks...45
4.3 Results...46
4.3.1 Functionality Test through Modbus Traffic Emulation 46
4.3.2 Fingerprinting Techniques 49
4.3.3 Invalid ICS Traffic 52
4.3.4 Logging Capabilities 52
4.3.5 Qualitative Evaluation... 55
4.4 Analysis...55
4.5 Results Summary 56

V. Conclusions and Recommendations 58

5.1 Conclusions...58
5.2 Future Work 59
5.2.1 Further Protocol Development 59
5.2.2 Levels of Implementation.. 59
5.2.3 Response Time 59
5.2.4 Traffic Loss 60
5.2.5 Ladder Logic and Firmware Implementation.. 60
5.2.6 Serial Implementation 60
5.2.7 Ethernet Header Manufacturing Tags 61
5.3 Concluding Remarks...61

Appendix A: Setting Up Emulated PLC...62

Appendix B: Canary.py code...68

Appendix C: Emulated PLC Test Case Supporting Figures 88

C.1 Functionality Test Through Modbus Traffic Emulation 88
C.2 Fingerprinting Port Scan Test Case 94

C.3 Fingerprinting Banner Grab Test Case 99
C.4 Invalid Traffic Test Case 101

Appendix D: List of Acronyms..104

Bibliography 106

List of Figures

Page

Figure 2.1: ICS network configuration . 7

Figure 2.2: Modbus serial message format. 8

Figure 2.3: Modbus TCP message format. 9

Figure 2.4: Network setup for Honeynet . 24

Figure 2.5: Typical Sebek deployment . 25

Figure 2.6: Sample configuration of Honeyd . 26

Figure 2.7: SCADA Honeynet configuration . 29

Figure 3.1: Network diagram. 32

Figure 3.2: Representative Gumstix device. 33

Figure 4.1: Read coil response Wireshark dissection. 47

Figure 4.2: Triangle MicroWorks response statistics. 47

Figure 4.3: Read, Write, Read dissected in Triangle MicroWorks 48

Figure 4.4: Read coil response Wireshark dissection. 50

Figure 4.5: Nmap MAC address resolution. 50

Figure 4.6: Triangle MicroWorks response statistics for banner grab. 51

Figure 4.7: Response to banner grab in Triangle MicroWorks. 51

Figure 4.8: Syslog entries of interactions with the emulated PLC. 53

Figure 4.9: Syslog entries from read, write, read test. 53

Figure 4.10: Syslog entries for banner grab. 54

Figure 4.11: Syslog entry from invalid TCP checksum. 55

Figure C.1: Traffic captured on HMI running Triangle MicroWorks 88

Figure C.2: Traffic captured on emulated PLC running Triangle MicroWorks 89

Figure C.3: Traffic captured on HMI running Modbus Poll.. 90

Figure C.4: Traffic captured on emulated PLC running Modbus Poll 91

Figure C.5: Traffic captured on HMI during read, write, read test.................................. 92

Figure C.6: Traffic captured on emulated PLC during read, write, read test. 93

Figure C.7: Nmap Intense Scan All TCP Ports, Emulated PLC 94

Figure C.8: Nmap Intense Scan All TCP Ports, CompactLogix 1769............................. 95

Figure C.9: Nmap Intense Scan All TCP Ports, MicroLogix 1100 96

Figure C.10: Nmap Operating System Scan on Ethernet/IP port ControlLogix 1769...... 97

Figure C.11: Nmap Operating System Scan on Ethernet/IP port MicroLogix 1100 98

Figure C.12: Traffic captured on HMI from banner grabbing test. 99

Figure C.13: Traffic captured on emulated PLC from banner grabbing test. 100

Figure C.14: Network capture of invalid checksum sent to MicroLogix 1100 101

Figure C.15: Network capture of invalid checksum sent to ControlLogix 1769............ 102

Figure C.16: Network capture of invalid checksum sent to emulated PLC.................... 103

List of Tables

Page

Table 2.1: Sector-Specific Agencies and assigned CIKR sectors.................................... 12

Table 4.1: Modbus TCP traffic tests. 46

Table 4.2: Response to valid/invalid TCP checksum. 52

EMULATING INDUSTRIAL CONTROL SYSTEM FIELD DEVICES USING GUMSTIX TECHNOLOGY

I. Introduction

Industrial control systems (ICS) constitute a significant portion of the nation's critical infrastructure. The power grid, transportation, oil and gas, and public works sectors rely heavily on the proper operation of control systems. A major disruption of any of these systems may result in devastating consequences. The limitations in ICS security have resulted in numerous failures, both targeted and un-targeted. In 2003, the Sobig virus infected computers at the Amtrak dispatching headquarters, causing signaling systems to shut down and halt ten trains between Pennsylvania and South Carolina [29]. The Slammer worm penetrated a computer at an Ohio nuclear plant in 2003, causing the safety monitoring system to be disabled for nearly five hours [33]. Most notably, the recent Stuxnet virus targeted specific operating characteristics to create direct physical consequences [22].

1.1 Problem Definition

Attacks on networking systems follow a general pattern and can be categorized according to the following sequential steps: reconnaissance, scanning, gaining access, maintaining access and covering tracks [37]. During reconnaissance and scanning, an attacker or malware obtains information about the targeted system. Security mechanisms (e.g., intrusion detection systems, antivirus and honeypots) are employed in traditional Information Technology (IT) systems to detect these malicious actions and provide early indicators of potential impending attacks. For ICS, however, security mechanisms designed specifically for the ICS environment are presently minimal. Indeed, there is lack of monitoring capabilities at the field device level. The field device level includes programmable logic controllers (PLCs) that control and monitor the physical operating parameters. Security monitoring at these end points needs to be improved to detect malicious actions and provide early indicators of potential impending attacks.

End system devices such as PLCs lack the inherent processing power, memory or system capabilities to incorporate security programs. Additionally, the highly dispersed nature of ICS operations requires extensive costs to retrofit security solutions. Finally, the lack of logging capabilities inhibits forensic ability to characterize attack tactics that are targeted towards the ICS environment.

1.2 Goals

The goal of this research is to include attack detection within ICS at the field device level. Specifically, this research develops an inexpensive, configurable, and portable emulation device that provides logging capabilities. The solution provides a low-cost security device that can be readily configured for implementation across many ICS environments. The PLC emulation device can be employed as an early detection sensor, introduces logging capabilities at the field device level, and can help characterize cyber attacks against ICS systems.

The emulated PLC should respond in accordance to RFC standards with any user that may try to interact with it directly. Additionally, the device must be capable of handling invalid traffic and respond to common fingerprinting techniques in a manner that will emulate an operational PLC. The device must handle all forms of traffic and be able to record any interaction with the emulated PLC. It is also expected to evade being fingerprinted as a Linux device while logging any interactions.

1.3 Scope and Limitations

The scope of this research focuses on emulation of Modbus TCP communication of a PLC. The six most common functions used in Modbus TCP traffic are emulated on the Gumstix technology to show that the emulation can be achieved. It is expected that further functions of the Modbus TCP specification can be added in future development iterations. It is also expected that additional protocols and services much like the Modbus protocol can be implemented in future development.

The research is limited by the inability to access a full ICS operational system. The test environment, however, is derived such that findings are expected to be consistent with an operational ICS. Additionally, a common method to fingerprint ICS devices is through Ethernet header manufacturer tags. These tags are represented as Ethernet trailers in many common packet dissection platforms, such as Wireshark. Due to lack of access to operational PLCs that implement the Ethernet tags, this technique is not evaluated. It is expected that in future research the tags can be analyzed and readily implemented.

1.4 Organization

Chapter 2 presents background information about ICS, ICS security and critical infrastructure protection. The Modbus protocol is described along with related research for ICS security measures. Finally, material on emulation in both the IT and ICS sector is presented.

Chapter 3 presents the methodology used in this research. This chapter describes the tests created to validate the emulation of the PLC device. Tests are compiled to cover a variety of traffic scenarios a fielded PLC encounters.

Chapter 4 presents the results from the tests described in Chapter 3. The results from these tests are presented based on each test scenario. The results demonstrate how the emulated device responds as an operational PLC would respond to common fingerprinting techniques used in ICS networks.

Chapter 5 presents the conclusions and future work for this research. This section is followed by several appendices with material to both support the results given in Chapter 4 and to allow other to reproduce the emulated PLC on a Gumstix Overo board.

II. Background

2.1 Overview

This section discusses topics associated with industrial control systems (ICS), critical infrastructure protection, emulation, and related research. The United States has seen a significant and steady increase in cyber attacks on both traditional information technology (IT) networks and ICS [20]. Some of these systems are crucial to our national critical infrastructure, and greater efforts and attention are being focused on securing ICS systems. Recommendations by the National Office for Cyberspace include working with regulatory agencies to develop governing policies for ICS and also work to secure government-owned critical infrastructures.

As an example of the emerging threat, Stuxnet demonstrated the damage that can be caused by malware that targets ICS. Stuxnet showed that an attack on ICS networks is possible and the effects of such an attack can be detrimental. Stuxnet was able to manipulate physical devices connected to the PLC to operate outside their normal parameters, sabotaging these devices [10]. Similar attacks are more likely as organizations increasingly connect their ICS networks to their corporate networks, providing additional attack vectors into ICS networks.

2.2 Industrial Control Systems Background

ICS manage, direct and monitor the behavior of large-scale, distributed systems in the critical infrastructure sectors. ICS use central monitoring stations, typically with a human machine interface (HMI) for an operator in the loop, to control and monitor remote processes [2]. ICS networks control critical infrastructure such as gas and oil pipelines, electric transmission, manufacturing, and many other critical infrastructures.

Figure 2.1 shows the devices typical to an ICS network and their traditional configuration. A Primary Control Center controls and monitors the overall operations. Within the Primary Control Center is the human machine interface (HMI) station, Control Server (Master Terminal Unit), Data Historian and Engineering Workstations.

The HMI provides the data to an operator in a Graphical User Interface (GUI). The GUI allows the operator to interact with the field devices in such a manner that the data is easily interpreted. The Control Server, many times referred to as the Master Terminal Unit (MTU), presents data to the HMI while also transmitting data from the HMI to field devices [19]. The Data Historian stores all the data that is reported to the MTU; this data is used by the engineers at their workstations to determine the efficiency of the network and billing purposes.

Figure 2.1 also shows a Backup Control Center which is a replica of the Primary Control Center that can assume control in case of a potential power outage or natural disaster in the region. The Primary and Backup Control Centers communicate through radio signals and Ethernet-based communications to remote stations via specialized protocols. The Remote Stations consist of field devices and the actuators and sensors that are connected to the field devices.

PLCs are field devices that communicate with the monitoring stations and convert digital control messages into physical actions such as opening and closing valves and breakers, collecting data from sensor systems, and monitoring the local environment for alarm conditions. There are several proprietary and open source protocols designed specifically for communications in ICS networks including: Modbus, DNP3, ProfiNet, Ethernet IP and many others [24], [40], [34].

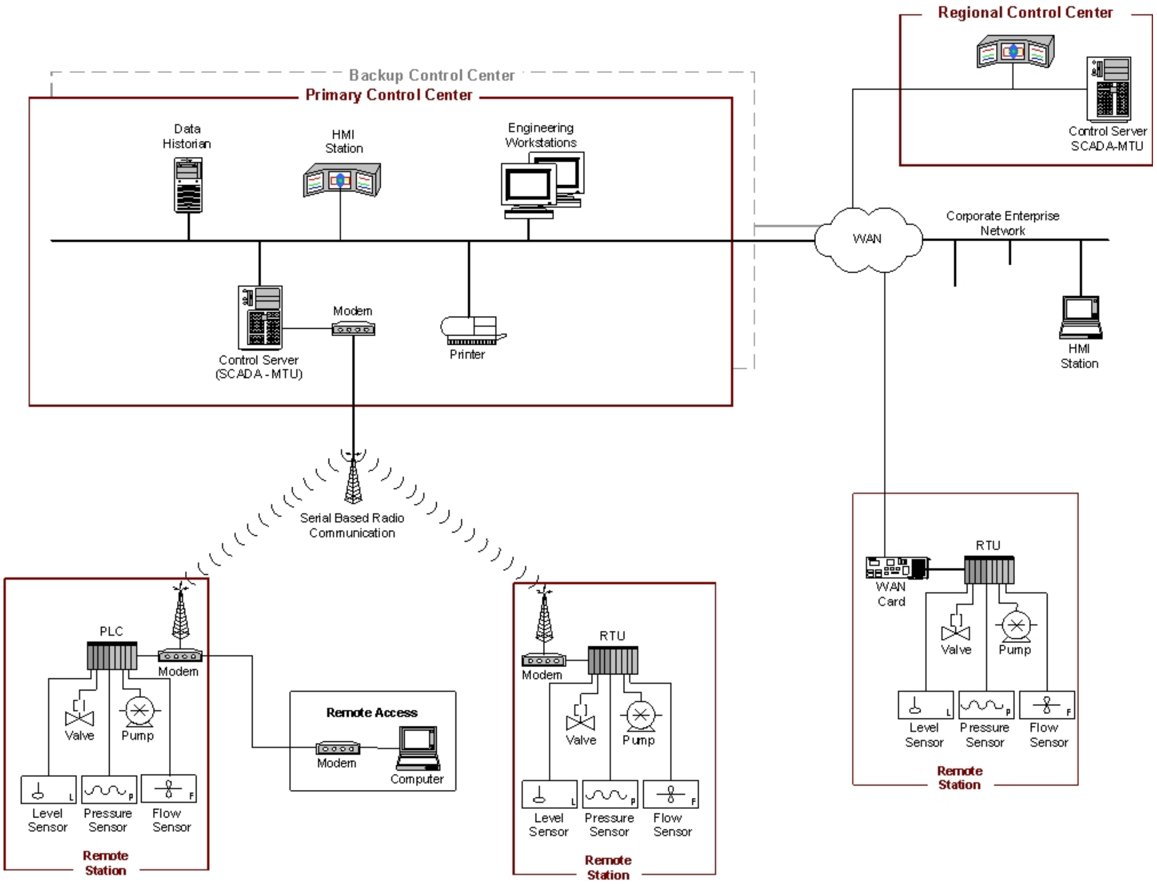


Figure 2.1: ICS network configuration [39].

The Regional Control Center is used in larger ICS networks to handle a subsection of the network (e.g., power generation facility in a power company). These control centers consist of a HMI and a MTU for local control of the network subsection. The ICS network is commonly connected to the Corporate Enterprise Network to allow authorized employees access to an HMI station, many times with read only access, to view the current status of the network.

The communication architecture for control systems uses a hierarchical, request-response paradigm for message transmission between a master control device and remote field devices. The master sends request messages to the outlying field device to gather data or to specify control actions. The field device collects discrete and analog sensor data and maintains actuator settings specified by the master. Response messages are generated by the field device after direct requests from the master. Additionally, the field device may notify the master when alarm conditions are detected.

2.3 Modbus Protocol

Modbus, designed in 1979, is one of the widest implemented communication protocols in the industrial control system environment [26]. Originally designed for serial communication, messages are transmitted between a master and field devices. The Modbus message format, depicted in Figure 2.2, contains three fields: outstation slave address, Modbus application protocol data unit (PDU), and an error checking field (CRC) [24]. The slave address identifies the intended recipient, with each device on the network assigned a unique identifier. The application PDU is comprised of a one byte function code specifying desired actions and up to 252 bytes for function parameters. The CRC

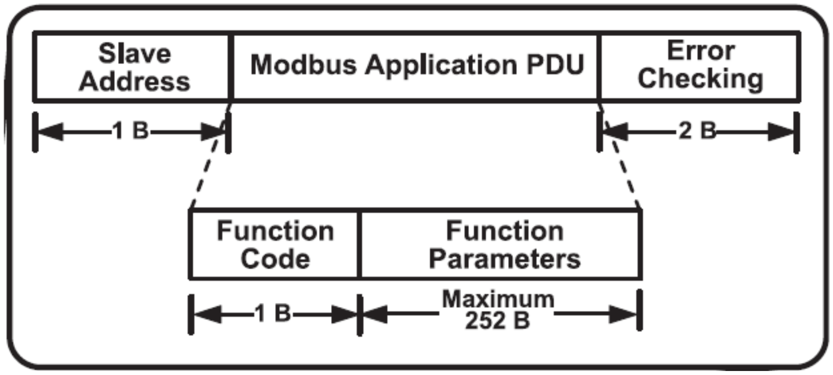


Figure 2.2: Modbus serial message format.

field is used to identify integrity errors that occur during message exchange.

To leverage the benefits and cost savings of LAN/WAN technologies, Modbus was modified for transmission to accommodate TCP/IP channels. Indeed, Modbus TCP extends the serial implementation by enabling a master to have multiple outstanding transactions, and an outstation to engage in concurrent communications with multiple masters [25]. In addition to the original serial message data fields, a Modbus application protocol (MBAP) header is added to facilitate IP communication. Figure 2.3 shows the message format of a Modbus TCP packet. The MBAP header contains a transaction ID, protocol designator, data length and unit id number. The Modbus data frame is encapsulated as a TCP payload and transmitted using Internet Assigned Numbers Authority (IANA) designated port 502.

The Modbus specification identifies a common set of function codes. The basic function codes implemented in the majority of systems are listed below, with the hex representation identified in parenthesis. Note that individual implementation schemes may use additional function codes designated by the standard for individual system configuration.

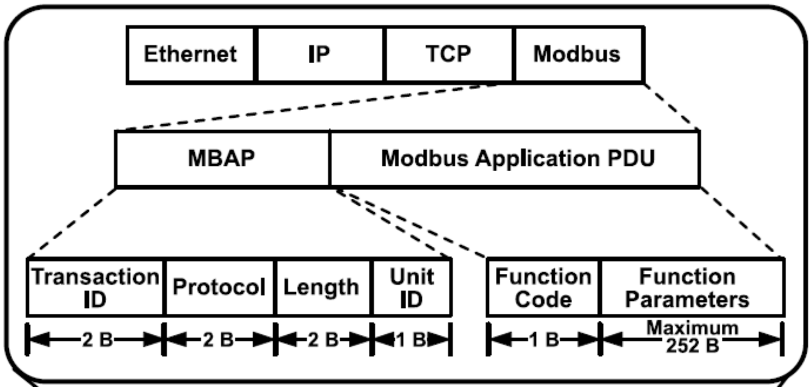


Figure 2.3: Modbus TCP message format.

- (0x01) Read Coils - This function code is used to read from 1 to 2000 contiguous status of coils in a remote device.

- (0x02) Read Discrete Inputs - This function code is used to read from 1 to 2000 contiguous status of discrete inputs in a remote device.

- (0x03) Read Holding Registers - This function code is used to read the contents of a contiguous block of holding registers in a remote device.

- (0x04) Read Input Registers - This function code is used to read from 1 to 125 contiguous input registers in a remote device.

- (0x05) Write Single Coil - This function code is used to write a single output to either ON or OFF in a remote device.

- (0x06) Write Single Register - This function code is used to write a single holding register in a remote device.

- (0x0F) Write Multiple Coils - This function code is used to force each coil in a sequence of coils to either ON or OFF in a remote device.

- (0x10) Write Multiple Registers - This function code is used to write a block of contiguous registers (1 to 123 registers) in a remote device.

- (0x14) Read File Record - This function code is used to perform a file record read.

- (0x15) Write File Record - This function code is used to perform a file record write.

- (0x16) Mask Write Register - This function code is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents.

- (0x17) Read/Write Multiple Registers - This function code performs a combination of one read operation and one write operation in a single MODBUS transaction.

- (0x18) Read FIFO Queue - This function code allows the read the contents of a First-In-First-Out (FIFO) queue of register in a remote device.

- (0x2B) Encapsulated Interface Transport (EIT) - The MODBUS Encapsulated Interface (MEI)Transport is a mechanism for tunneling service requests and method invocations, as well as their returns, inside MODBUS PDUs.

Consider, for example, communication from a master device to a PLC to close a valve in an oil and gas pipeline. The master generates a request message that specifies a write action with opcode 05 for address 01 containing data value FF to close the control valve. The PLC performs the action and responds with a reply message containing opcode 05 and address 01 to indicate completion of the action. Subsequent read requests from the master returns a value indicating the valve is closed.

2.4 Critical Infrastructure Protection

Critical infrastructure protection (CIP) relates to the preparedness to an incident involving critical infrastructure. In testimony to Congress by Gregory C. Wilshusen, Director, Information Security Issues, defines CI as:

"Critical infrastructures are systems and assets, whether physical or virtual, so vital to our nation that their incapacity or destruction would have a debilitation impact on national security, economic well-being, public health or safety, or any combination of these" [11].

Critical Infrastructure is divided into 18 sectors based on Homeland Security Presidential Directive 7 (HSPD-7) [13]. In HSPD-7, the President designates the Secretary of Homeland Security as the "principal Federal official to lead Critical Infrastructure and Key Resources (CIKR) protection efforts" and assigns responsibilities to Federal Sector-Specific Agencies (SSAs). The list of sectors and their corresponding SSAs are provided in Table 2.1. This directive provides the criteria for establishing additional sectors of protection in the future. Many of these sectors are very complex and interconnected in such a way that if one of these sectors is disrupted it could cause disruption in other sectors. An example of this is if an attacker is able to prevent the transmission of power to other facilities such as a manufacturing plant, production at that plant halts. A similar condition occurred in 2003 when a fault in the power grid caused an estimated 55 million people to lose power [30]. As a result, boil water advisories went into effect, train service in the region shut down, airports in the region shut down, many oil refineries on the east coast had to shut down, cellular communications was disrupted due to cell towers backup generators shut off, and many large factories had to stop or slow productions because of supply problems. Although the effects were not caused by a malicious actor, the scenario demonstrates the impact to critical services.

Table 2.1: Sector-Specific Agencies and assigned CIKR sectors [5][6][28].

Sector Specific Agency	Critical Infrastructure and Key Resources Sector
Department of Agriculture Department of Health and Human Services	Agriculture and Food
Department of Defense	Defense Industrial Base
Department of Energy	Energy
Department of Health and Human Services	Healthcare and Public Health
Department of the Interior	National Monuments and Icons
Department of the Treasury	Banking and Finance
Environmental Protection Agency	Water
Department of Homeland Security *Office of Infrastructure Protection*	Chemical Commercial Facilities Critical Manufacturing Dams Emergency Services Nuclear Reactors, Materials and Waste
Office of Cybersecurity and Communications	Information Technology Communications
Transportation Security Administration	Postal and Shipping
Transportation Security Administration *United States Coast Guard*	Transportation Systems
Immigration and Customs Enforcement *Federal Protective Service*	Government Facilities

2.5 Industrial Control Systems Security

Control systems offer unique security challenges [17]. A primary benefit of control systems is that remote and isolated locations can be monitored centrally without the need for onsite personnel. From a security standpoint, however, this provides entry points into the system that have minimal physical safeguards. Additionally, the trend to interconnect devices using networking technologies introduces entry points, often via the Internet.

The limitations in ICS security have resulted in numerous failures, both targeted and un-targeted. In 2003, the Sobig virus infected computers at the Amtrak dispatching headquarters, causing signaling systems to shut down and halt ten trains between Pennsylvania and South Carolina [29]. The Slammer worm penetrated a computer at an Ohio nuclear plant in 2003, causing the safety monitoring system to be disabled for nearly five hours [33]. At the Browns Ferry nuclear power plant in 2006, a “Data Storm” spike in traffic caused a PLC to crash, resulting in the failure of recirculation pumps and forcing a manual reactor shutdown [42]. Most notably, the recent Stuxnet virus targeted specific operating characteristics to create direct physical consequences [22].

ICS networks are connected to the Internet despite known risks. Leverett discovered 10,358 ICS related devices connected to the Internet through a search over a two year period from 2009-2011 [43]. Leverett used a total of 33 queries to find over 10,000 devices using an open source search engine, SHODAN. He also used Google’s geocoding service to locate the devices by the latitude and longitude using the country, city name, and country code. Of the devices discovered, only 17 percent implemented any type of authentication [21].

ICS networks require constant integrity and availability. ICS network engineers typically have not considered confidentiality because the networks were primarily air gapped. Integrity is important because valid traffic is necessary to ensure that a device is operating within normal operating parameters. Availability is also very important because the systems are responsible for critical services that require optimal uptime.

A primary shortfall in ICS security is the lack of ability to monitor and detect malicious events at the field device level. PLCs have little memory, hard drive space, or processing power and are not designed to execute additional applications. As a result, there are minimal security mechanisms designed specifically for the ICS environment. The lack of early attack indicators and logging capabilities impedes identification of attacks and the ability to perform forensics if a system is compromised.

Encountered in many ICS networks are legacy system devices. It is not abnormal for a system to be in use for 30 years in a traditional ICS network [19]. Many of these systems must be in operation 100% of the time, so they cannot be taken offline for system upgrades even when a security hole has been discovered. These systems are typically left vulnerable for many years without replacement or upgrades.

ICS security has to deal with the challenge of bridging the gap between Information Technology (IT) experts who know the traditional security solutions and the engineers that configure ICS networks. IT experts are typically concerned about security in the enterprise networks and the engineers are concerned about system availability and functionality [19].

2.6 ICS Security Mechanisms

There is a need to develop and implement a more robust security mechanism for ICS networks. Digital Bond has developed an IDS signature package for four different control system protocols [7]. The signatures are able to defend against known attacks, malformed protocol requests and rarely used commands. There is a need to discover previously unknown attacks to create additional signatures for the IDS. Note that the IDS is designed for the perimeter network layer and not the field device layer.

Another research team, Morris and Pavurapu [27], established a bump-in-the-wire device that is placed in a network to encrypt, analyze, and log each network packet. This device is able to defend against response injection, command injection, and denial of service on a control system. This inline system introduces the risk of compromising availability of the systems it is protecting if it goes offline.

Many of the ICS security developments work to incorporate in a layer of encryption. Balitanas et al. [1] look to add in a crossed-cipher scheme to increase security through encryption with reduced delay in the system compared to IT encryption techniques. The authors note that there are significant challenges when implementing cryptography because of the time requirements of ICS systems and the time delay added by encryption, decryption and processing time. Unfortunately, this solution would have to be built into all devices and adds additional latency counter to requirements of real time environments.

Other secure ICS architectures are described by Pal et al. [31]. The authors take into account the limitations of limited computational capacity, limited space capacity, low bandwidth and real-time processing. The architectures discussed each have their own advantages of key storage and distribution among field devices. Each of the architectures requires a different number of keys to be stored at the field device layer depending if the field devices need to communicate between each other.

There are currently inline solutions that have been created such as the EtherGuard Encryptor developed by Ultra Electronics [41]. These products offer a way to help increase security; however, if these devices are inline and fail they will disrupt the availability of the overall system. Additionally, these devices introduce latency in the network traffic that may be detrimental to the need to operate in a real-time environment [19].

Stouffer's *Guide to ICS Security* [39] recommends integrating security into networks through network segregation. The first recommendation is to keep the ICS system air-gapped from the corporate network. However, Leverett [21] has shown that many networks are connected to the Internet with or without the network administrator's knowledge. Stouffer also presents multiple firewall models to create network segregation if the network must be connected to the Internet. These models include: a dual-homed machine (i.e., a system connected to both the controlled ICS network and the corporate network), firewall between the corporate and ICS networks, a firewall and router between the corporate and ICS networks, a firewall with a demilitarized zone (DMZ) between the corporate and ICS networks and two firewalls between corporate and ICS networks. Stouffer notes that firewalls are not the best solution but that the firewalls do provide an effective baseline level of security.

Remote forensics on ICS networks has been demonstrated by Chavez et al. [3] when they showed that Encase Enterprise can be used to perform forensics on ICS networks. Test results from this research demonstrate the feasibility of conducting forensics on a field device without disrupting normal operations.

2.7 Network Attack

Skoudis describes a five step process for attacking a network: Reconnaissance, Scanning, Gaining Access, Maintaining Access and Covering Tracks and Hiding [37]. Reconnaissance requires discovering as much about a target as possible. Attackers use common fingerprinting mechanisms to find the machine they are attempting to compromise. SHODAN, for example, allows an intruder to perform reconnaissance to find a device vulnerable to an attack. The next step, scanning, occurs when an attacker knows IP addresses of targeted systems and involves scanning to find potential vulnerabilities. A common tool used for scanning systems to find more information about the device is Nmap. Nmap determines which ports are open and potentially vulnerable to attack. The next step, gaining access, is when the attacker uses an exploit against a vulnerability to gain access to the system. There are many exploit databases or tools that an attacker can consult to get a description of an exploit or to launch an automated attack. Once the attacker has access they use a Trojan Horse or add a backdoor on the system to maintain the access. Once the attacker knows that they are able to maintain access on a system, they cover their tracks by installing rootkits, modifying logs, creating hidden files and establishing cover channels.

2.7.1 Fingerprinting

Fingerprinting is a standard technique used to identify the OS running on the target system. In control systems, fingerprinting is used to find the make and model of field devices [9]. In the reconnaissance and scanning phases of network attack, fingerprinting of ICS field devices is performed in a variety of ways. There are four identifiable elements on most field devices: known set of open ports through port scans, known behavior of services through banner grabbing, Ethernet header manufacturer tags and known MAC address space. Through the combination and correlation of these items, a fingerprint can be produced for a field device.

2.7.1.1 Port Scan

Most PLCs operate on a select set of proprietary ports. Allen Bradley devices, for example, run a proprietary protocol, Ethernet IP, over port 44818. If the device is scanned and port 44818 is determined open, an attacker could conclude that the device has a likelihood of being an Allen Bradley device. If a port scan identifies a particular set of open ports, it is likely that the device is from a specific vendor. Devices that communicate Modbus TCP have TCP port 502 open. Once an attacker discovers open ports they are able to further fingerprint the device using banner grabbing techniques.

2.7.1.2 Banner Grabbing

A device can be correctly identified through banner grabbing via known responses on open ports. The SHODAN system, used by Leverett during his research, was able to compile a list of responses from banner grabs against open ports on devices. Banner grabbing on web servers is very common because many times information obtained corresponds to a company that manufactures the device.

Modbus designated port 502 is also susceptible to banner grabbing. The Modbus TCP protocol makes it mandatory to incorporate the Encapsulated Interface Transport function with the Modbus Encapsulated Interface type Read Device Identification. This function allows any Modbus TCP connection to read very critical information about the device. The mandatory objects that must be defined are vendor name, product code and major/minor revision. The information returned from those three objects identify the exact device and firmware.

2.7.1.3 Ethernet Manufacturer Tags

Digital Bond discusses how some devices have manufacturer specific tags in the Ethernet header of response packets from field devices [9]. This field is placed as a Ethernet trailer used to designate that the traffic is to a specific device.

2.7.1.4 MAC address

The last piece that can be used to fingerprint a field device is the MAC address space of the vendor. Each manufacturer of Ethernet enabled devices is assigned a MAC address range which can be used to determine the vendor of the device if the fingerprinting is done on a local segment.

2.8 Emulation

Emulation is software or hardware that allows one system to imitate the behavior of another system. This phenomenon is very common in the IT sector with the development of honeypots. A honeypot is a closely monitored computing resource that is intended to be probed, attacked or compromised. More precisely, a honeypot is “an information system resource whose value lies in unauthorized or illicit use of the resource” [38]. Honeypot technology has been around for many years on the Internet but only recently has it been introduced in the ICS community. Honeypots were first discussed, in 1990, with the book Clifford Stoll’s *The Cuckoo’s Egg* and Bill Cheswick’s “An Evening With Berferd”; the first honeypot was deployed in 1997 [38].

Honeypots are an effective way to detect intruders and to gather malware samples to create signatures to prevent future attacks. Honeypots add value to the security of a system by detecting and logging threats and allowing mitigations of such attacks. In a honeypot, an inbound connection implies the system is being scanned or attacked. This is the case because honeypots are intended to be dormant with no legitamate traffic sent to the devices. Outbound connections usually represent a compromise of the system because honeypots are configured not to send traffic on the network.

2.8.1 Honeypot Overview

There are two types of honeypots: production and research honeypots. Each honeypot type operates in the same manner but are used for different objectives. Research honeypots are used to gather malware for further analysis and creation of detection signatures. Production honeypots add to the overall security posture of an organization by detecting attacks and mitigating the risk of attackers [38]. Mitigating the risk is done through many different means such as blocking inbound connections from the specific IP address.

Honeypots mainly consists of two variations: low-interaction and high-interaction honeypots. Low-interaction honeypots consists of emulated services and operating systems which provide targets. These honeypots are easily fingerprinted as they only emulate the basic services. High-interaction honeypots provide real systems applications and services for the attacker to interact with. High-interaction honeypots are difficult to set up because they need to be secure enough so an attacker cannot use the machine to attack other machines in the network [38].

2.8.2 Advantages of Honeypots

Honeypots afford advantages including valuable data, resources, simplicity and return on investment [15]. The first advantage is the value in the data collected. Honeypots only collect data when interacted with, making the data much more manageable to analyze than traditional network logging systems. Additionally, honeypots reduce the amount of false positives because any interactions indicate unauthorized traffic. Honeypots are able to detect many more attacks because any activity in the honeypot is an irregularity which makes novel attacks easier to identify. This is more effective than alternatives that use signatures which require previous identification of the attack.

Honeypots require minimal resources for employment. Honeypots can be set up on aging computers because they have little interactions and typically do not have to deal with resource exhaustion. Even large networks only require one or two systems to monitor any kind of attack on the network.

Honeypots are also very simplistic. One does not have to keep up with signature sets or rule sets; someone just needs to place the honeypot somewhere in the network and then wait for the attack. Some honeypots are more complex but all follow the same simple premise: if something interacts with the honeypot it is ilegitimate communication and needs to be examined [38].

The last advantage to honeypots is the return on investment. Honeypots are cost effective because of minimal resource requirements. Honeypots also demonstrate that if it has been attacked that someone has been able to infiltrate the network [38].

2.8.3 Disadvantages of Honeypots

While honeypots have many advantages, they also have disadvantages. The first disadvantage is that they have a narrow field of view. Honeypots only see what traffic is directed at the honeypot. If the attack is never directed at the honeypot it will never detect the attack.

Honeypots are also susceptible to fingerprinting. Fingerprinting occurs when an attacker can identify the true identity of the honeypot because of certain characteristics or behaviors [38]. If the attacker can correctly identify the honeypot, he can avoid it when attacking the network. While uncommon, fingerprinting can also be done if the programming of the honeypot has misspelled a word somewhere which alerts an attacker when the response is sent back to the attacker.

The last disadvantage is the risk that a honeypot introduces into the network. While the amount of risk each type of honeypot introduces into the environment is different, the risk is still present. Once a honeypot has been attacked and compromised it can be used to attack, infiltrate or harm other computer systems in the organization or other organizations [15].

2.8.4 Honeypot Attributes

There are three fundamental requirements of honeypots: data control, data capture and data analysis. The first, data control, is used for mitigation of risk through the containment of the activity of the attacker. This is accomplished by controlling what an attacker is able to do once on the honeypot. Note that it is important to make sure that once the honeypot is compromised, another system cannot be compromised by it. A common way to do this is with a fail-safe that prevents all outbound connections from the honeypot once compromised. Honeypots should also alert when a system has been compromised to notify an administrator of the event.

Data capture is the next fundamental requirement which includes logging and auditing functions. The most common way to collect the data is with a layer two bridge that collects any traffic that has been directed to or from the honeypot. Note that nothing should be stored on the local honeypot machine to prevent fingerprinting information for the honeypot to the attacker.

The last requirement is data analysis which is the synthesis of information gathered from the honeypot. If multiple honeypots are implemented across a large network spanning multiple time zones the information needs to be standardized and have synchronized time stamping to correlate the data. This is important for analyzing the attack methods to ensure continuity between collection methods [15], [16].

2.8.5 Honeypot Technology in IT

There are many different solutions developed for the IT sector. The following set of solutions are indicative of current honeypots in IT. The first solution in the IT sector is honeynets. A honeynet is a “high-interaction honeypot designed to capture extensive information on threats “[16]. A honeynet is an architecture with multiple, networked honeypots. Each of the honeypots in the network can be different systems ranging from Windows workstations to IIS web servers to Cisco routers. Honeynets rely on the same basic principles that honeypots follow in that they are not productions systems so that any communication with these systems is considered malicious. In the paper "Know Your Enemy: Honeynets [16]" The Honeynet Project states:

> "In many ways a honeynet is like a fishbowl. You create an environment where you can watch everything happening inside it. However, instead of putting rocks, coral, and sea weed in your fish bowl, you put Linux DNS servers, HP printers, and Juniper routers in your honeynet architecture. Just as a fish interacts with the elements in your fishbowl, intruders interact with your honeypots"[16].

Figure 2.4 shows an example of a typical network configuration of a honeynet. The most critical component to a honeynet is a Honeywall.

A Honeywall is a transparent bridge that is setup to enable data capture, data control and data analysis. Honeywall is configured with three interfaces, two for the

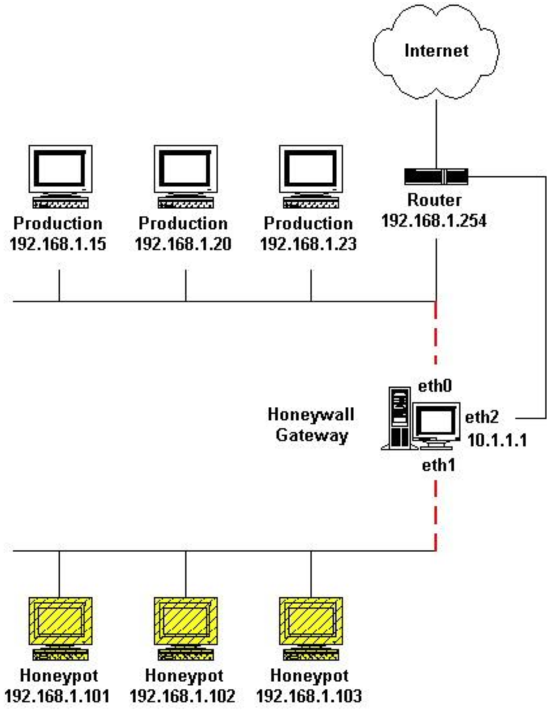


Figure 2.4: Network setup for Honeynet [16].

transparent bridge and one for management. A transparent bridge has no IP address so all the traffic passes promiscuously through the device. The third interface is configured for th management network to enable remote control of the Honeywall. Honeywall limits malware damage by implementing a fencelist (i.e., a list of IP's for non-target computers which honeypots on the LAN cannot communicate with). Honeywall uses snort-inline [23] as an intrusion protection system to prevent attackers from sending known exploits to other machines once the machine is compromised. The number of connections out is typically filtered from the Honeywall to prevent too much activity once the box has been compromised. Honeywall also uses the monitoring system Sebek [35].

Sebek is a client-server data capture tool which closely monitors and logs all user activity. Sebek replaces several common system calls which can then observe all accessed data [35]. Sebek is a kernel-level rootkit which hooks and replaces common calls. Sebek has the following capabilities: record keystrokes of a session that is using encryption, recover files copied with SCP, capture passwords used to log in to remote

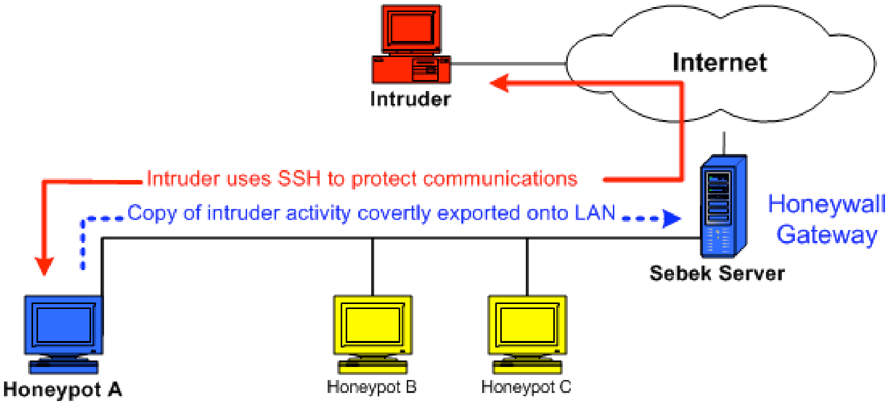

Intruder
Internet
Intruder uses SSH to protect communications
Copy of intruder activity covertly exported onto LAN
Honeywall
Gateway
Sebek Server
Honeypot A
Honeypot B
Honeypot C

Figure 2.5: Typical Sebek deployment [14].

system, recover passwords used to enable Burneye protected binaries and accomplish many other forensics related tasks [14]. In Figure 2.5, the client module is installed on the honeypot A. The attacker's activity captured by the honeypot is then sent to the network and passively collected by the server (Honeywall Gateway). Sebek data is not stored on the target, but rather transmitted via UDP to the sniffing honeywall or designated log server. Packets are masked from the attacker, even if a sniffer is run on the target through the use of a special Kernel module created to interact directly with the network device driver instead of using the TCP/IP stack [14].

Honeyd is another common honeypot solution. Honeyd is an open source low-interaction virtual honeypot. Honeyd has the capability to simulate thousands of virtual systems on one single physical system. Figure 2.6 shows a sample configuration of honeyd. Honeyd is able to provide arbitrary services, via a configuration file, that interact with an attacker. Honeyd simulates each operating system at the TCP/IP level which provides honeyd the ability to deceive Nmap into believing the virtual honeypot is an actual operating system.

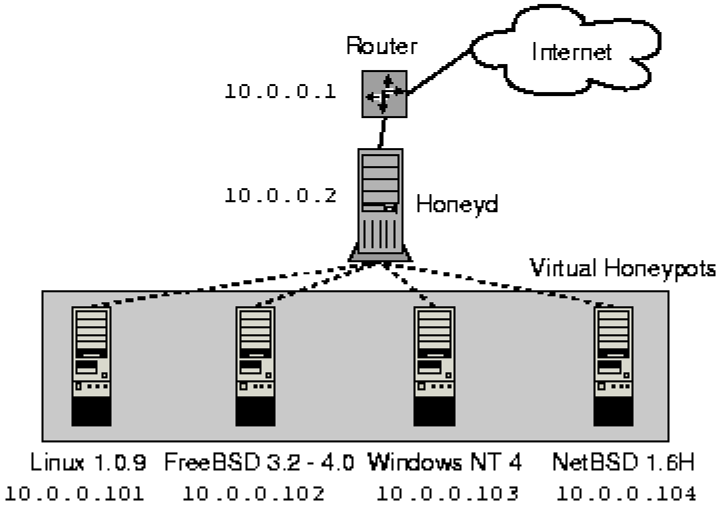


Figure 2.6: Sample configuration of Honeyd [35].

2.9 Honeypots in ICS

Honeypots are useful in ICS networks to help improve the overall security posture. Today there is no nominal way to detect malware running on PLCs. Consider the case of Stuxnet which was on the PLCs for a year before being detected [10]. Indeed, a honeypot for ICS would help identify malware currently in ICS networks and an ability to study any future malware.

Honeypots aid in the overall security posture through prevention, detection and response. Honeypots help with prevention by acting as an early warning of an attack. A honeypot generates an alert for any connection allowing an administrator to block the IP address and prevent the user from attacking any other machines. Some honeypots use deception or deterrence to prevent attackers from further attempts to attack the system. Deception involves making the attacker waste time on a honeypot that has no value and deter them from trying to attack production machines. Deterrence is used when the honeypot is coded to inform the attacker that the box they are interacting with is a honeypot in an attempt to dissuade them from attacking the network any further.

Honeypots also add to ICS security posture through detection. Honeypots are an effective way at detecting attacks through reducing false positives, false negatives and through data aggregation. The last way that honeypots add to the overall ICS security posture is through response. Honeypots collect all the data to and from the system so the data necessary to respond to an incident can be retrieved by the incident responder. The honeypot can also be taken offline for further analysis without affecting production systems [38].

There are many IT solutions currently developed; however, these solutions are not readily applicable to ICS networks. The IT honeypots are not effective because the cost of solutions to disseminate across ICS is too high. Additionally, solutions that place a honeypot in line with a production device creates a point of failure which could disrupt the availability that is critical to ICS networks. Current IT solutions are also not applicable to ICS networks due to the nonstandard communication protocols.

2.9.1 Current Honeypots in ICS

Even with the current landscape and challenges, some solutions have been proposed for ICS networks. The first solution is a SCADA Honeynet that was started in 2004 utilizing Honeyd, simulating a limited set of services from a popular PLC [32]. The goals of this project were to create a framework to simulate a variety of industrial networks on a single Linux host running honeyd (e.g., minimal Modbus TCP functions, FTP, Telnet, and web server). These servers are only basic simulations and offer a limited number of functions to interact with. The work is no longer maintained; however, a follow on was initiated by Digital Bond.

The work by Digital Bond utilizes two separate virtual machines. One of the virtual machines is a modified Honeywall which implements Digital Bond SCADA IDS signatures to detect malicious attacks against the second virtual machine [8]. This is an efficient tool that can also be used in line with a physical device as well. Note that this introduces latency and could fail causing communication to the physical device to fail. The second virtual machine is a simulated PLC that exposes a number of services to an attacker [8]. Digital Bond implements Modbus TCP protocol, FTP server, Telnet, HTTP and SNMP servers. These services are much like the other project in that they only

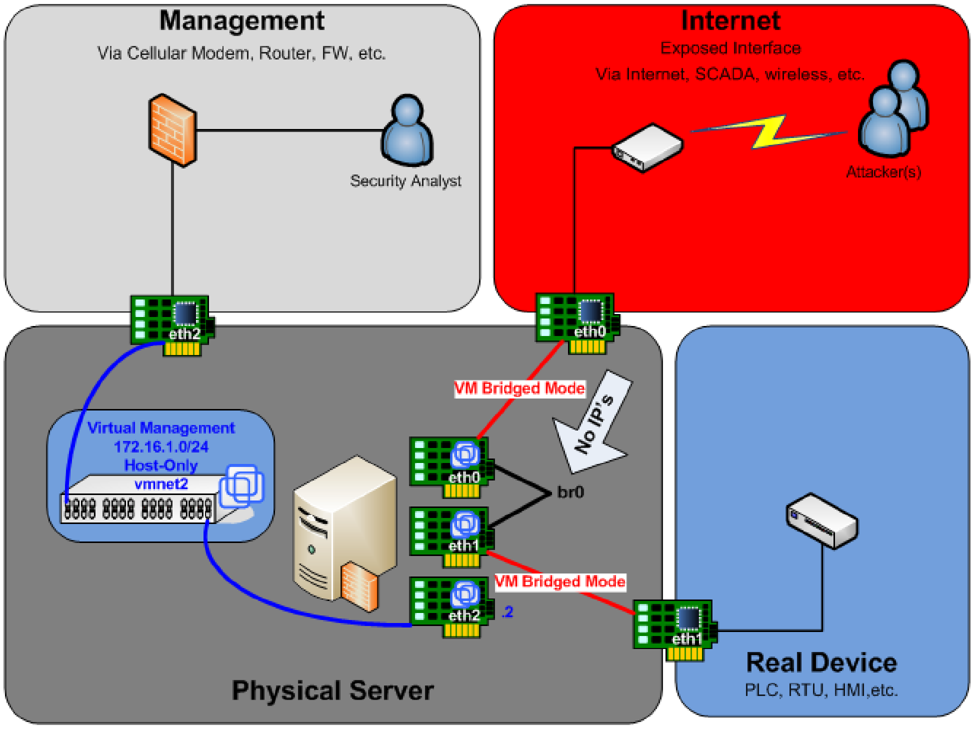


Figure 2.7: SCADA Honeynet configuration [8].

simulate the banner for the different protocols and minimal basic functions. Figure 2.7 shows the configuration of the two virtual machines from Digital Bond.

These solutions are both efficient solutions but are only designed for a particular PLC or particular protocol. The solutions require more modularity to allow expansion into the majority of protocols and devices in ICS networks.

2.9.2 Emulation Requirements

ICS honeypots have extra challenges associated because of the variety of ICS networks. There are many manufacturers of PLCs, differing protocols, and system-specific configurations for ICS networks. This makes it challenging for a single honeypot solution to emulate a variety of systems. Additionally, each PLC has different field devices ranging from sensors to valves. As a result, each PLC has a different configuration to control each of these field devices.

When a honeypot detects a new attack, an analyst can analyze it to create a signature to input into the IDS to prevent the attack in the future [18]. This idea is restated by the Department of Energy when they provided the “21 Steps to Improve Cyber Security of SCADA Networks.” Number eight in the list is to implement internal and external IDS and establish 24-hour-a-day incident monitoring [4].

2.10 Summary

This chapter explains ICS, critical infrastructure protection, emulation and the current technology surrounding honeypots. It details current ICS honeypots and short comings with the current technology. ICS honeypots need to be modularized and allow easy reconfiguration. This chapter demonstrate the necessity for additional security in ICS networks and how current IT solutions are not capable of protecting the vastly different ICS networks. The next chapter discusses the methodology used to evaluate the effectiveness of the ICS honeypot created as part of this research.

III. Methodology

This chapter discusses the methodology for evaluating an emulated PLC to determine if the device responds to basic network traffic and can avoid common fingerprinting techniques. Successful emulation of a PLC utilizing Modbus TCP traffic is contingent upon the device (1) correctly responding to standard traffic, (2) avoiding being fingerprinted as a Linux machine using common ICS fingerprinting techniques and (3) correctly handling invalid traffic.

3.1 Problem Definition

3.1.1 Goals and Hypothesis

The goal of this research is to include attack detection within Industrial Control Systems (ICS) at the field device level through development of an inexpensive, configurable and portable emulation device that contains logging capabilities.

It is expected that the emulated PLC responds according to RFC standards with any user that may try to interact with it directly. It is expected that the device responds to all traffic sent to the device in a valid manner and be able to log all interactions with the emulated PLC. It is also expected to respond as an operational PLC to common ICS fingerprinting techniques (i.e., Port Scan, MAC Address, and Banner Grabbing).

3.1.2 Approach

This research determines the effectiveness of the emulated PLC at emulating an operational PLC. Allen Bradley PLCs are used as a baseline for fingerprinting tests and invalid traffic tests while the Modbus TCP RFC is used as the baseline for standard traffic response tests. The emulated PLC is evaluated to see how responses compare with the baseline and if the interaction is logged. The emulated PLC and Allen Bradley PLCs are subjected to a variety of tests outlined in Section 3.3 to determine if the devices respond in the appropriate manner with the corresponding baseline. Analysis of the results is examined to determine the effectiveness of the emulated PLC at detecting traffic on ICS. Additionally, a qualitative analysis using Air Force ICS assessors is used to provide a notional evaluation of the effectiveness of emulating an operational PLC.

3.2 Environment

Figure 3.1 shows the environment used for the following experiments. The HMI is a Windows 7 64 bit SP1 machine running Triangle MicroWorks Protocol Test Harness, Nmap and Wireshark. Triangle MicroWorks Protocol Test Harness is a package that has been designed to test PLC devices to determine if they conform to protocol standards. The HMI has a 500GB hard drive with 4GB of memory. The PLCs are a factory install of an Allen Bradley Micrologix 1100 and an Allen Bradley CompactLogix 1769. The

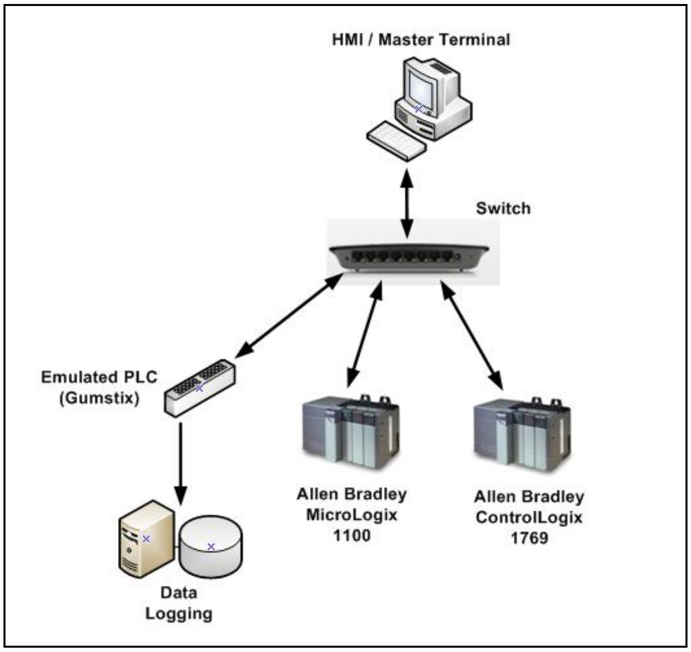


Figure 3.1: Network diagram.

emulated PLC is an Overo Earth COM Gumstix. Gumstix is a mini computer, not surprisingly, about the size of a stick of gum (see the bottom of Figure 3.2). It runs a Linux based platform using the Open Embedded framework and costs approximately $200 [12]. The Gumstix board has an ARM Cortex-A8 CPU, 512MB of flash memory and 512MB of RAM with a microSD card slot to be used as non-volatile storage. In the case of this research, an 8GB microSD card is used. Gumstix computers leverage expansion boards to extend IO capabilities to a range of operations (e.g., GPS, bluetooth, and 802.11 wireless). For this research, the Tobi-Duo expansion board (shown in the top of Figure 3.2) is incorporated to provide a dual NIC configuration allowing a primary NIC for ICS communication and another NIC for out-of-band logging. The Gumstix Overo CPU board snaps onto the Tobi-Duo expansion board for quick connection. The operating system is Linux 2.6.34 built and installed on the device. Appendix A provides

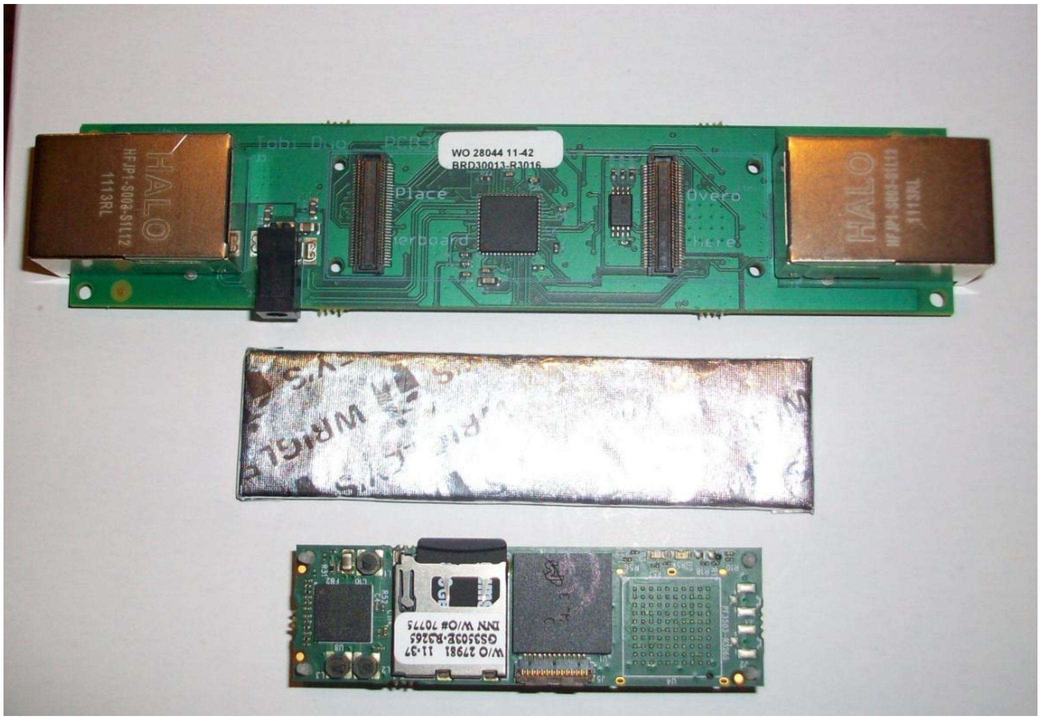


Figure 3.2: Representative Gumstix device.

the steps required to build the emulated PLC. The logging device is an Ubuntu 11.10 Machine with a 120GB hard drive with 2GB of memory running syslog server capturing the logging entries sent from the emulated PLC. The systems are connected with CAT5e cable which supports up to 100MB/s connection. Figure 3.1 shows that both the emulated PLC and the Allen Bradley PLCs communicate directly with the HMI. The figure depicts that the emulated PLC can sit in a network next to any vendor specific device (e.g., Siemens, Omron, Allen Bradley). For testing purposes, however, the PLCs that are connected to the switch are an Allen Bradley MicroLogix 1100 and an Allen Bradley ControlLogix 1769. The emulated PLC also communicates with the data logging device through logs sent out the secondary NIC.

3.3 Evaluation Technique

3.3.1 Functionality Test through Modbus Traffic Emulation

The Modbus traffic test cases are used to verify the ability of the emulated PLC to communicate in accordance with Modbus RFC standards. Although there are numerous Modbus TCP standard function codes, the most commonly used include:

- Read Coil
- Write Coil
- Read Discrete Inputs
- Read Holding Registers
- Write Holding Register
- Read Input Registers

Each of the function codes listed above is sent in accordance with Triangle MicroWorks and Modbus Poll evaluation process to the emulated PLC in order to verify proper responses. The commands are sent in the order shown above with thirty seconds in between each packet being transmitted. Note that the focus of these tests is to evaluate operational functionality; analysis on traffic rate limits is recommended for future work. The following two software packages are used to test the emulated PLC against the traffic standard:

1. Triangle MicroWorks Protocol Test Harness
2. Modbus Poll

Triangle MicroWorks Protocol Test Harness is a software package created to test if a device adheres to the Request for Comments (RFC) for a given protocol. This checks the response packets bytes to make sure that the packet is a valid packet. If the packet is valid the Test Harness logs each response received, and if the packet is invalid the Test Harness times out waiting for a valid response. Modbus Poll is a free software package created to communicate with Modbus enabled devices. This software is also used to communicate with the emulated PLC to see if the responses are considered valid.

The following steps outline how to complete each test case. First, the emulated PLC is attached to the switch and the logging device using CAT5 cables. After the device is turned on and both Ethernet NICs have been initialized then, SSH is initiated from the logging device to the emulated PLC. The command ‘ifconfig’ is run to determine the IP address of the emulated PLC. Next, the command ‘ps –ef’ is run and then viewed to make sure that both tcpdump and the python script are initiated. On the HMI, Wireshark is started to capture the network traffic to and from the emulated PLC. The software indicated in the test case (i.e., Triangle MicroWorks or Modbus Poll) is then started, and each of the six most popular Modbus TCP commands listed above are initiated by the software. After the six commands conclude, the software and Wireshark are stopped and the capture saved for analysis. An SSH session is then initialized to the emulated PLC and the pcap created from tcpdump is retrieved. The syslog on the logging machine is also saved. The emulated PLC is then restarted to make a consistent starting point for each test.

The emulated PLC is successful if the program used for testing is able to receive a valid response from the device. The traffic is analyzed to see if the Modbus Wireshark dissector is able to determine that the traffic being sent from the emulated PLC is Modbus TCP. The test is not successful if the device does not respond in an expected manner.

The emulated PLC must keep state as part of the functionality such that if a coil is turned from off to on, subsequent reads indicate the coil is now turned on. This test is successful if a second Read Coil response shows the coil has transitioned status. The test is unsuccessful if the response to the second Read Coil shows that the coil did not transition state.

3.3.2 Fingerprinting Test Cases

3.3.2.1 Port Scan Test Case

The intent of the emulated PLC is to act as an operational PLC and avoid being fingerprinted as a Linux device. The most common way to detect a device is through port scanning with a tool such as Nmap. Nmap is run to scan the device for open ports but can also attempt to determine the operating system (OS) that the device is running. In the Port Scan Test Cases the following devices are scanned:

1. Emulated PLC
2. Allen Bradley MicroLogix 1100

3. Allen Bradley ControlLogic 1769

These scans determine if it is possible to fingerprint the device as an emulated PLC through scanning and examining the results against the scans of two Allen Bradley devices that are configured to communicate over Ethernet/IP (port 44818). The Allen Bradley devices do not communicate over Modbus TCP so the devices do not have port 502 open as the emulated PLC; however, they do have a standard ICS communication protocol (Ethernet/IP port 44818) that is open in the same manner as the Modbus port for the emulated PLC.

The following outlines how to complete each test case. First the emulated PLC is connected to the switch and the logging device. Both Allen Bradley devices are connected to the switch with Ethernet cables. After the emulated PLC is turned on and both Ethernet NICs have initialized then SSH is initiated from the logging device to the emulated PLC. The command ‘ifconfig’ is run to determine the IP address of the emulated PLC. Next, the command ‘ps -ef’ is run and then the results are viewed to make sure that both tcpdump and the python script are initiated. On the HMI, Wireshark is initialized on the Ethernet port to capture the network traffic to and from the emulated PLC and Allen Bradley Devices. Nmap is also initialized on the HMI. An Intense Scan including all TCP ports is done against each of the devices listed above. After the Nmap scan has completed Wireshark is stopped. The Wireshark and Nmap captures are saved for analysis. After the scan of the emulated PLC an SSH session is initialized to the emulated PLC and the pcap created from tcpdump is retrieved. The syslog on the logging machine is also saved. Another scan of the Allen Bradley devices is also accomplished to run the OS detection scan against just the ICS communication port and one closed port (e.g. <IP address> -O -p 44818-44819). The emulated PLC does not currently implement a web server so the second Nmap scan provides a closer representation of what the emulated PLC results should look like. Future work includes implementation of a web server on the emulated PLC to provide a closer representation of modern PLCs.

The Nmap scan on the emulated PLC is successful if only the ICS communication port, TCP port 502, appears to be open. If the OS scan on the emulated PLC results are the same as that of the OS scan specifically targeting the open ICS communication port on the Allen Bradley devices then the test is considered successful. The test is not successful if the OS on the emulated PLC is not the same as the Allen Bradley devices.

3.3.2.2 MAC Address Resolution Test Case

MAC Address Resolution is a common fingerprinting technique common to ICS field devices. The emulated PLC is tested to verify that the MAC address resolves to a known ICS vendor. The Wireshark traffic from both the Modbus Traffic Emulation test cases and the Nmap scan are examined to see if the MAC address resolves to a known ICS vendor. The results of the Nmap scan are also examined to see if the MAC address resolves to a known ICS vendor.

The MAC Address Resolution succeeds if the MAC addresses in all the Wireshark captures are the same and resolve to a known ICS vendor. The MAC address must also resolve in the Nmap scan to succeed. The MAC Address Resolution fails if in any of the captures the MAC address does not indicate a known ICS vendor MAC address.

3.3.2.3 Banner Grabbing Test Case

Banner Grabbing is another fingerprinting technique common to ICS field devices. This test case tests that the emulated PLC can respond to banner grabbing request on the Modbus TCP port. The following states how to configure the emulated PLC for this test case. First the emulated PLC is connected to the switch and the logging device. After the device is turned on and both Ethernet NICs have been initialized then SSH is initiated from the logging computer to the emulated PLC. The command 'ifconfig' is run to determine the IP address of the emulated PLC. Next, the command 'ps -ef' is run and then the results are viewed to make sure that both tcpdump and the python script have initiated. On the HMI, Wireshark is started on the Ethernet port to capture the network traffic to and from the emulated PLC. Triangle MicroWorks Protocol Test Harness is then started on the HMI. The Modbus TCP command Device ID is then sent to banner grab the information from the emulated PLC. This command helps to correctly identify a device because the response sends information such as the vender name and product name. Note that this command is a Modbus command and therefore cannot be run against either of the Allen Bradley devices. After the command has completed, Triangle MicroWorks and Wireshark are stopped. The Wireshark capture is saved for analysis. An SSH session is initialized to the emulated PLC and the .pcap created from tcpdump is retrieved. The syslog on the logging machine is also saved.

The banner grabbing on the emulated PLC is successful if Triangle MicroWorks is able to receive a valid response from the emulated PLC. Successful emulation of this command gives another way that an attack can fingerprint the emulated PLC as an operational PLC. The test is not successful if the device does not respond in a manner that Triangle MicroWorks is expecting.

3.3.3 Invalid Traffic Test Cases

This test is used to make sure that when an invalid packet is received by the emulated PLC it responds in the same manner as an operational PLC. This test incorporates the emulated PLC and both the Allen Bradley devices.

The devices are configured as in Section 3.3.2.1 Port Scan. The only difference is that Scapy is started on the HMI to allow for the creation of an invalid TCP packet to be sent to each of the devices. Scapy sends an invalid SYN packet with a NULL TCP checksum and then waits for a response. After 60 seconds Scapy then sends a valid SYN packet verifying that the device is responsive. After Scapy has completed sending the packets Wireshark is stopped. The Wireshark capture is saved for analysis. After the invalid traffic to the emulated PLC has completed an SSH session is initialized to the emulated PLC and the .pcap created from tcpdump is retrieved. The syslog is also saved for analysis.

The invalid traffic test case on the emulated PLC is successful if the response from the emulated PLC matches that of the Allen Bradley devices. The test is not successful if the device does not respond in a similar manner to that of the Allen Bradley devices. This test is emulating a standard IT practice of fingerprinting a device through responses received to certain invalid packets.

3.3.4 Logging Capabilities

Logging on the emulated PLC is important to be able to capture any interaction that an attack may have with the emulated PLC. The logging of the emulated PLC is tested in the Modbus traffic emulation, port scan, banner grabbing and invalid traffic tests as described above. For each of the tests the number of packets captured on the emulated PLC are compared to the number of packets captured on the HMI. The syslog on the logging device is also checked for each test to see if each of the commands sent to the emulated PLC are correctly logged.

The logging of the emulated PLC is successful if both of the following conditions are met: (1) the number of packets captured on the emulated PLC match the number of packets sent from the HMI and (2) each of the commands sent to the emulated PLC are correctly logged.

3.3.5 Qualitative Evaluation

A qualitative analysis is conducted through work with a member of the Air National Guard's 262nd Network Warfare Squadron. The 262nd is based at McChord Air Force Base outside Tacoma, Washington and attracts people from many tech companies such as Microsoft, Cisco Systems and Adobe Systems. A member of this unit analyzed the emulated PLC for fingerprinting techniques. The 262nd does ICS assessments on Air Force networks which gives them the capability of comparing the emulated PLC to operational devices.

The emulated PLC is provided to one of the members of the Air National Guard's 262nd. The member uses their available testing environment and techniques for the evaluation. The evaluation is considered successful if the member reports that the emulated PLC is not distinguishable from an operational device. Note that specific techniques used to evaluate this device are not considered important; the goal is to see if an experienced assessor determines the emulated PLC is consistent with an operational PLC.

3.4 Methodology Summary

This chapter provided the methodology for evaluating the emulated PLC. The Modbus traffic emulation of the emulated PLC is examined with: valid Modbus traffic, fingerprinting techniques and invalid traffic. Modbus traffic is used to examine if the emulated PLC functionality is the same as that of an operational PLC. Fingerprinting techniques are used to study the case of an attacker scanning the emulated PLC. The invalid traffic is used to see if the emulated PLC is able to respond in the same manner as a PLC by other typical IT methods of fingerprinting. These tests are all accomplished to simulate possible network traffic a device may receive when emulating a PLC. During each of the tests, the logging capabilities are verified for capture on the emulated PLC as well as remote logging capabilities. The qualitative analysis is conducted by an ICS security expert who is able to give an evaluation on how well the emulated PLC emulates an operational PLC.

IV. Analysis and Results

Results of this research are organized as the following: Section 4.1 discusses the development of the emulated PLC. Section 4.2 describes the emulated PLC initialization checks prior to running the test. Section 4.3 presents the results from the functionality checks with valid Modbus Traffic, fingerprinting techniques, invalid TCP traffic, logging capabilities and the qualitative evaluation given by a subject matter expert. Section 4.4 is the analysis of the results given in Section 4.3, and Section 4.5 summarizes all the results.

4.1 Development of emulated PLC

4.1.1 Architecture

Device implementations for ICS field devices more closely resemble cell phones than traditional information technology platforms. Indeed, there are myriad vendors, model numbers, configurations, chipsets and different operating systems/firmware for each associated device [36]. While the devices are quite unique in platform characteristics, PLCs that implement Modbus TCP conform to the RFC protocol specifications in order to enable inter-device communication. Although the emulated PLC does not contain input/output functionality for analog and digital signals to control physical devices, it can be programmed to respond appropriately to Modbus communications. For this research, the emulated PLC was programmed according to RFC specifications to incorporate common function codes identified in Chapter 3.

The emulated PLC is incorporated into ICS operations similar to other PLC field devices. Note that the emulated PLC can be readily modified to emulate various PLC vendors (e.g., Siemens, Omron, and Allen-Bradley). PLC identification is determined by

MAC addresses assigned to the various vendors. The emulated PLC can be set to respond with any MAC address such that if correlated, it resembles the associated vendor product. Additionally, the emulated PLC incorporates an out-of-band logging capability to record and report on specified criteria (e.g., unexpected traffic patterns, attempt to read/write unauthorized parameters and unexpected function codes).

4.1.2 Implementation Details

The program that provides the emulation capability was developed using Python with the Scapy 2.2.0 module. Additionally, tcpdump is implemented on the ICS-facing NIC to allow capture of network traffic and storage as pcap files. Syslog is used to send alerts and traffic files to a remote logging device via the other NIC. The emulated PLC is readily configurable to respond to operating parameters in the same manner as an operational PLC. The emulated PLC maintains system state in the event that function parameters are modified. For example, if a message is received to write to a single coil (e.g., close a valve) and a subsequent message requests a read for the same parameter (e.g., status of the valve), the emulation device will respond with the updated state (e.g., valve closed). Additionally, if an unrecognized function code or transaction message is received, the emulated PLC responds with an appropriate unrecognized error code. If traffic is received on a port other than the designated TCP port 502 (Modbus), the PLC emulation device responds with a simple RESET ACK; however, the action generates a logging event. Such traffic may be indicative of a port scan.

For logging purposes, any received traffic that is not consistent with pre-defined parameters generates an event. Consider, for example, the configuration of the PLC emulation device to continually respond to read requests for various defined parameters.

The PLC emulation device is expecting to receive the message traffic precisely as specified; any traffic not conforming generates an event. Alternatively, the PLC emulation device can be deployed to a segment without a specific configuration for expected message traffic. In this scenario, the PLC emulation device serves as a traditional honeypot and can indicate attempts to scan the network for ICS devices.

Emulating a traditional PLC for open ports is accomplished by implementing iptables to make all ports appear closed on the ICS-facing NIC, with the exception of port 502. The Scapy module generates packets from the PLC emulation device to craft messages consistent with Modbus standards. Additionally, a startup script is included that changes the MAC address to a specified value to correspond with a PLC vendor. Banner grabbing is implemented for the Modbus TCP communication service emulated on TCP port 502. For purposes of this research, replicating additional services to respond to banner grabbing or other identified fingerprinting techniques are not implemented. Future work for the PLC emulation device consists of developing such functionality.

4.2 Emulated PLC Initialization Checks

Prior to sending traffic to the devices in the test cases, the emulated PLC is checked to validate that the necessary services are running. The services required are as follows:

- PLC emulation (canary.py)
- Tcpdump
- Network communications
- Syslog

The detailed steps for verifying services are as follows:

1. SSH connection is initiated from the logging device to the emulated PLC
    a. Run the command ps-ef
        i. Check the results and verify python and tcpdump are running
2. A command window is opened on the HMI
    a. The command ping <IP address of emulated PLC> is initiated
    b. The command window is checked for a valid response
    c. The syslog on the logging device is then checked to confirm that the packet is recorded

For each test the emulated PLC successfully initialized and the processes correctly started.

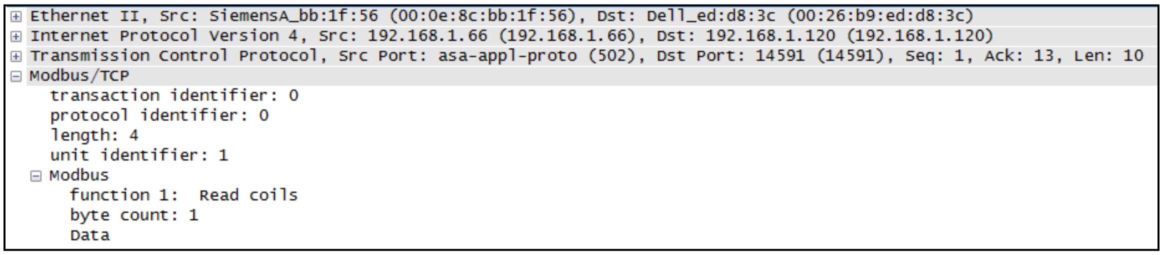
4.3 Results

4.3.1 Functionality Test through Modbus Traffic Emulation

The tests from the two software programs emulating Modbus TCP traffic demonstrate the emulated PLC conforms to RFC standards. Table 4.1 shows that for each software package all six commands initiated by the test harness received valid responses.

Table 4.1: Modbus TCP traffic tests.

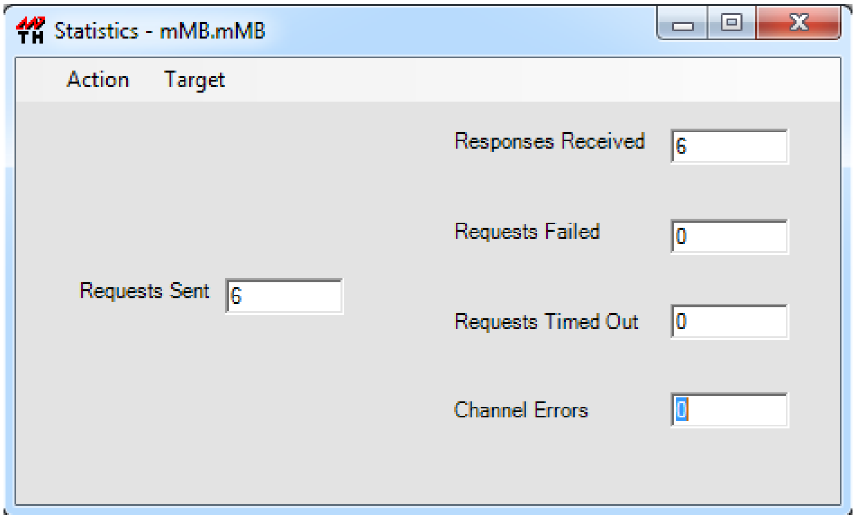
Software	Messages Sent	Valid Responses	Invalid Responses
Triangle MicroWorks Protocol Test Harness	6	6	0
Modbus Poll	6	6	0



```
⊞ Ethernet II, Src: SiemensA_bb:1f:56 (00:0e:8c:bb:1f:56), Dst: Dell_ed:d8:3c (00:26:b9:ed:d8:3c)
⊞ Internet Protocol Version 4, Src: 192.168.1.66 (192.168.1.66), Dst: 192.168.1.120 (192.168.1.120)
⊞ Transmission Control Protocol, Src Port: asa-appl-proto (502), Dst Port: 14591 (14591), Seq: 1, Ack: 13, Len: 10
⊟ Modbus/TCP
    transaction identifier: 0
    protocol identifier: 0
    length: 4
    unit identifier: 1
  ⊟ Modbus
      function 1:  Read coils
      byte count: 1
      Data
```

Figure 4.1: Read coil response Wireshark dissection.

The findings indicate that the responses of the emulated PLC conform to RFC standards for Modbus TCP. Figure 4.1 shows the read coil response from the emulated PLC correctly dissected in Wireshark, indicating a valid response. The correct dissection of the response in Wireshark further demonstrates that the response adheres to the RFC standard for Modbus TCP. Figure 4.2 shows the statistics on Triangle MicroWorks denoting that all six responses are valid. The Responses Received field identifies when the response is valid and conforms to RFC standards. Requests Failed, Requests Timed Out, and Channel Errors indicate erroneous or invalid responses.



Responses Received: 6

Requests Failed: 0

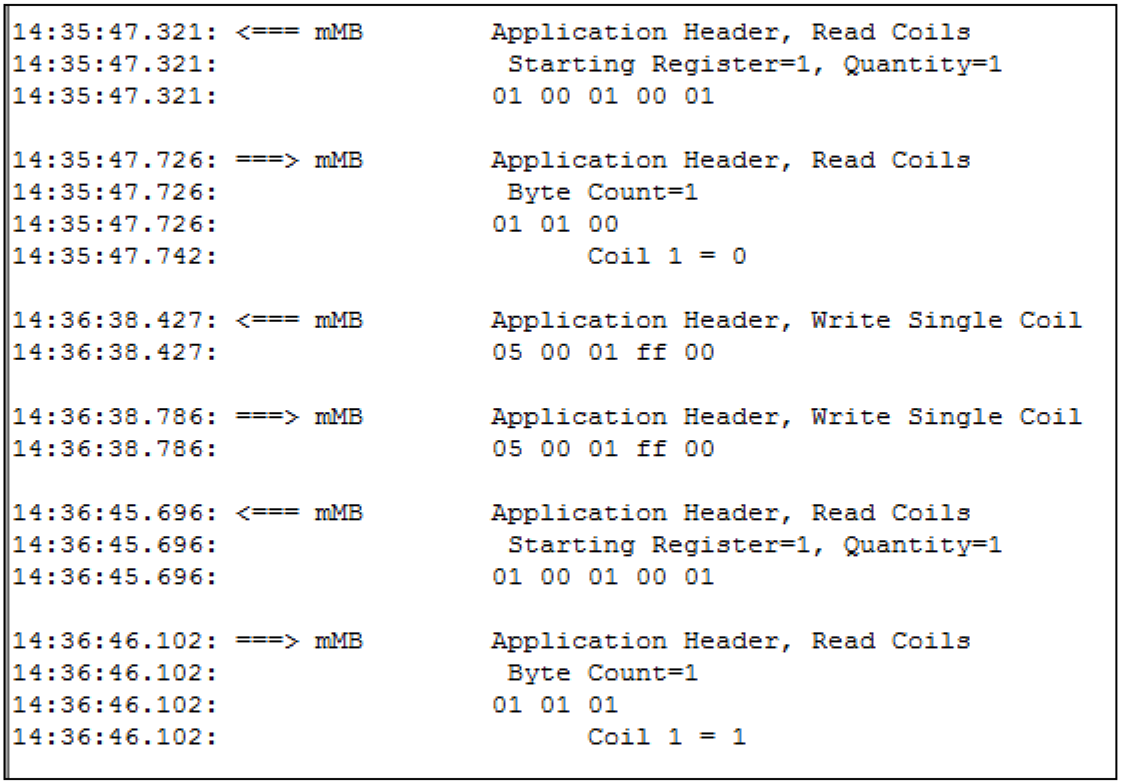
Requests Sent: 6

Requests Timed Out: 0

Channel Errors: 0

Figure 4.2: Triangle MicroWorks response statistics.

The other tested functionality is the ability of the emulated PLC to maintain appropriate state. Figure 4.3 shows the request and response of the traffic dissected in Triangle MicroWorks. The first read shows that coil 1 is set to *off* (0). The coil is then turned *on* (ff) and the status displays *on* (1) for the subsequent read. The findings demonstrate the ability of the emulated PLC to update and maintain state.



```
14:35:47.321: <=== mMB        Application Header, Read Coils
14:35:47.321:                  Starting Register=1, Quantity=1
14:35:47.321:                 01 00 01 00 01

14:35:47.726: ===> mMB        Application Header, Read Coils
14:35:47.726:                  Byte Count=1
14:35:47.726:                 01 01 00
14:35:47.742:                       Coil 1 = 0

14:36:38.427: <=== mMB        Application Header, Write Single Coil
14:36:38.427:                 05 00 01 ff 00

14:36:38.786: ===> mMB        Application Header, Write Single Coil
14:36:38.786:                 05 00 01 ff 00

14:36:45.696: <=== mMB        Application Header, Read Coils
14:36:45.696:                  Starting Register=1, Quantity=1
14:36:45.696:                 01 00 01 00 01

14:36:46.102: ===> mMB        Application Header, Read Coils
14:36:46.102:                  Byte Count=1
14:36:46.102:                 01 01 01
14:36:46.102:                       Coil 1 = 1
```

Figure 4.3: Read, Write, Read dissected in Triangle MicroWorks

The emulated PLC provides functionality for six Modbus commands according to the RFC standards. The test used to evaluate the functionality of the six commands is consistent with industry standards used to evaluate an operational PLC before deploying to the field. While only six commands are implemented in this iteration, incorporating additional commands is trivial.

4.3.2 Fingerprinting Techniques

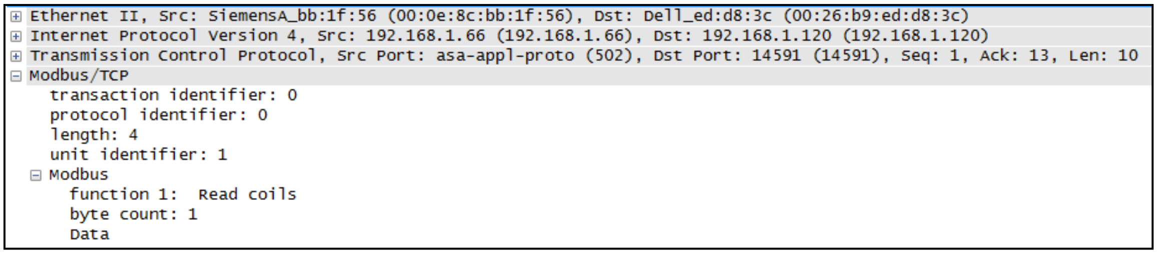
4.3.2.1 Port Scan

The port scan using the software package Nmap identifies open ports and provides OS detection. When the emulated PLC is scanned with the *intense scan all TCP ports* the results show TCP port 502 open and all other ports closed. The results are indicative of a PLC device communicating Modbus on port 502. Note that the Allen Bradley devices use Ethernet/IP as opposed to Modbus for ICS communication. Scan results for both Allen Bradley devices similarly show a native ICS communication protocol Ethernet/IP on port 44818.

For baseline purposes, an Nmap scan was performed against the Overo Gumstix with the Linux image and resulted in ports 22 and 111 open and an OS detection result of Tomato 1.27 (Linux 2.4.20). The Overo Gumstix was then configured to the emulated PLC. The OS detection results is ‘none’ because the results failed to match any operating systems in the Nmap OS database. A similar scan for the Allen Bradley devices also resulted in ‘none’. The scanning methodology is indicative of an attacker scanning a device to identify open communication ports and attempting to identify the OS to fingerprint a device prior to launching an exploit. The findings demonstrate that an attacker scanning the emulated PLC with Nmap would infer the device to be an actual PLC due to the manner the emulated PLC responds consistent with a PLC communicating Modbus.

4.3.2.2 MAC Address Resolution

The MAC Address resolution test demonstrates the ability to mimic manufacturer device identifiers. Figure 4.4 shows that the MAC address of the emulated PLC resolves

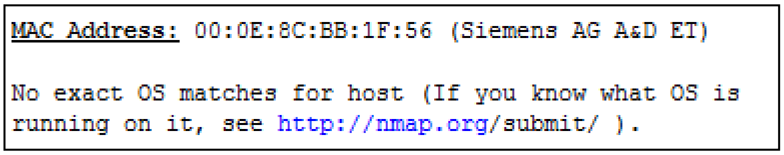


```
⊞ Ethernet II, Src: SiemensA_bb:1f:56 (00:0e:8c:bb:1f:56), Dst: Dell_ed:d8:3c (00:26:b9:ed:d8:3c)
⊞ Internet Protocol Version 4, Src: 192.168.1.66 (192.168.1.66), Dst: 192.168.1.120 (192.168.1.120)
⊞ Transmission Control Protocol, Src Port: asa-appl-proto (502), Dst Port: 14591 (14591), Seq: 1, Ack: 13, Len: 10
⊟ Modbus/TCP
    transaction identifier: 0
    protocol identifier: 0
    length: 4
    unit identifier: 1
  ⊟ Modbus
      function 1:  Read coils
      byte count: 1
      Data
```

Figure 4.4: Read coil response Wireshark dissection.

to a Siemens PLC when examined via Wireshark. This is consistent with all traffic captures associated with the emulated PLC; in each test the MAC address appropriately resolved to a Siemens PLC.

During the Nmap scan of the emulated PLC, the MAC address resolution, shown in Figure 4.5, also resolves to a Siemens Automation device. With the OS scan coming back as negative an attacker examining the MAC address resolution and open ports likely concludes the device is indeed a PLC.

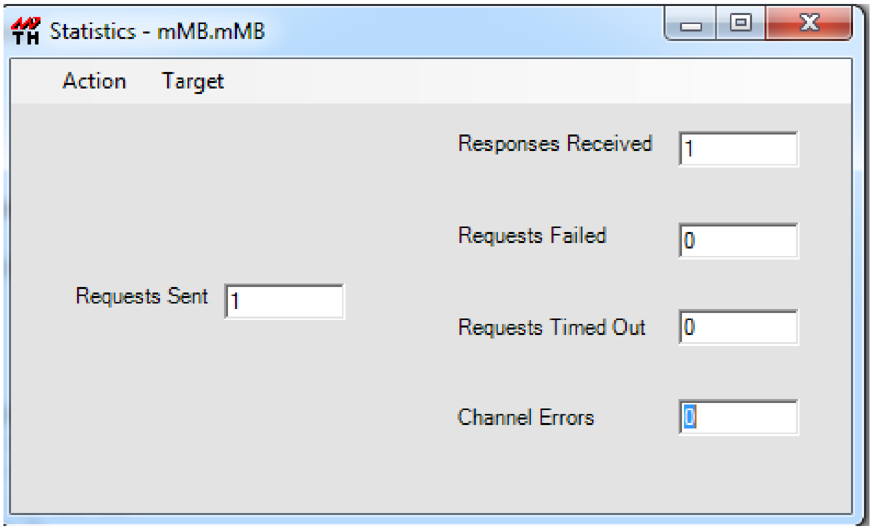


```
MAC Address: 00:0E:8C:BB:1F:56 (Siemens AG A&D ET)

No exact OS matches for host (If you know what OS is
running on it, see http://nmap.org/submit/ ).
```

Figure 4.5: Nmap MAC address resolution.

4.3.2.3 Banner Grabbing

The next common fingerprinting technique in ICS field devices is banner grabbing. The Modbus TCP function code for Encapsulated Interface Transport-Device ID x43 x14, allows a user to retrieve information about a PLC, such as the vendor name and product code. For the banner grabbing test, Triangle MicroWorks Protocol Test Harness initiates the protocol messages. Figure 4.6 demonstrates a successful response from the emulated PLC conforming to the RFC standard. Figure 4.7 provides the values in detail for each of the three objects returned in response to the command. Device object zero is the vendor name, object 1 is the product code and object 2 is the major minor revision. The Conformity Level means that the information is basic information about the PLC and the next object ID is only used if the information cannot be encapsulated in one packet. The emulated PLC for the banner grabbing test is configured to appear to be an Allen Bradley, showing the adaptability of the emulated PLC to emulate multiple device types. The results show the device is an Allen Bradley MicroLogix 1500 V1.12.1.



Responses Received 1

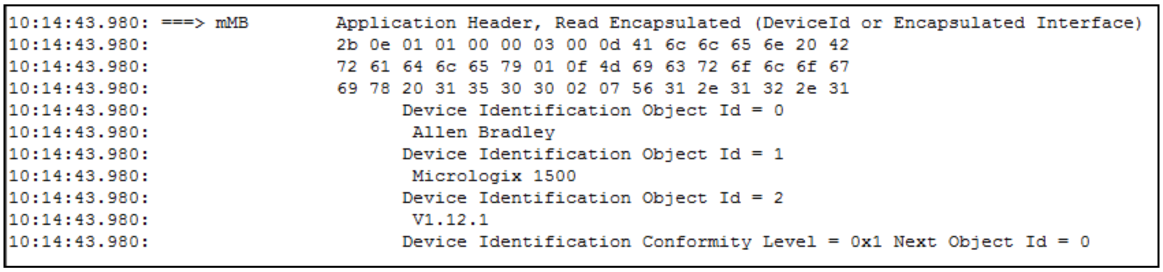
Requests Failed 0

Requests Sent 1

Requests Timed Out 0

Channel Errors 0

Figure 4.6: Triangle MicroWorks response statistics for banner grab.



```
10:14:43.980: ===> mMB        Application Header, Read Encapsulated (DeviceId or Encapsulated Interface)
10:14:43.980:                 2b 0e 01 01 00 00 03 00 0d 41 6c 6c 65 6e 20 42
10:14:43.980:                 72 61 64 6c 65 79 01 0f 4d 69 63 72 6f 6c 6f 67
10:14:43.980:                 69 78 20 31 35 30 30 02 07 56 31 2e 31 32 2e 31
10:14:43.980:                       Device Identification Object Id = 0
10:14:43.980:                        Allen Bradley
10:14:43.980:                       Device Identification Object Id = 1
10:14:43.980:                        Micrologix 1500
10:14:43.980:                       Device Identification Object Id = 2
10:14:43.980:                        V1.12.1
10:14:43.980:                       Device Identification Conformity Level = 0x1 Next Object Id = 0
```

Figure 4.7: Response to banner grab in Triangle MicroWorks.

Note that the banner grabbing is only tested for the Modbus protocol implementation. As services are added to the emulated PLC (e.g., web servers), the device requires evaluation of banner grabbing techniques for the service added.

4.3.3 Invalid ICS Traffic

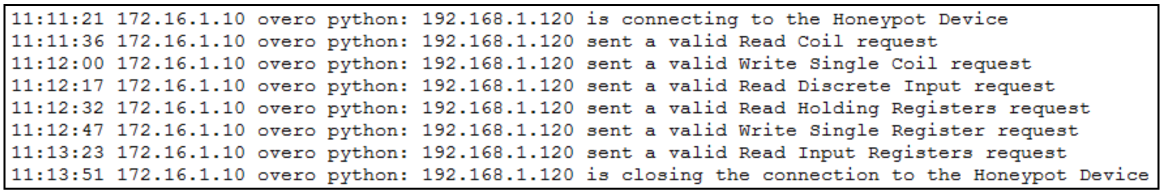
The emulated PLC was evaluated for the ability to handle invalid ICS traffic. A SYN packet with an invalid checksum was sent to each of the PLC devices. As demonstrated in Table 4.2, each device appropriately dropped the invalid packet. Appendix C.3 provides screen captures of the traffic in Wireshark for each device. Note that there is no response to any of the request packets. The second packet in each capture is a valid SYN packet followed by a response from each of the PLCs. The results demonstrate that the device is functioning and checks for valid TCP checksums.

Table 4.2: Response to valid/invalid TCP checksum.

Device	Response to Invalid Checksum	Response to Valid Checksum
Emulated PLC	No	Yes
Allen Bradley ControlLogix 1739	No	Yes
Allen Bradley MicroLogix 1100	No	Yes

4.3.4 Logging Capabilities

The logging capability is designed to record any interaction with the emulated PLC. The traffic captured on the emulated PLC using tcpdump and logged on a remote logging device. During the valid Modbus TCP traffic, the connection with the emulated PLC is logged, all six commands are logged and the connection tear down with the emulated PLC is logged. Figure 4.8 shows the syslog entry connection and commands from the Triangle MicroWorks functionality test.

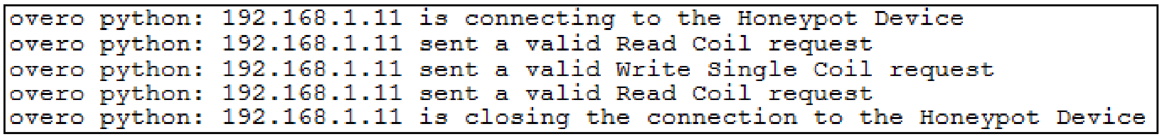


```
11:11:21 172.16.1.10 overo python: 192.168.1.120 is connecting to the Honeypot Device
11:11:36 172.16.1.10 overo python: 192.168.1.120 sent a valid Read Coil request
11:12:00 172.16.1.10 overo python: 192.168.1.120 sent a valid Write Single Coil request
11:12:17 172.16.1.10 overo python: 192.168.1.120 sent a valid Read Discrete Input request
11:12:32 172.16.1.10 overo python: 192.168.1.120 sent a valid Read Holding Registers request
11:12:47 172.16.1.10 overo python: 192.168.1.120 sent a valid Write Single Register request
11:13:23 172.16.1.10 overo python: 192.168.1.120 sent a valid Read Input Registers request
11:13:51 172.16.1.10 overo python: 192.168.1.120 is closing the connection to the Honeypot Device
```

Figure 4.8: Syslog entries of interactions with the emulated PLC.

Appendix C.1 provides screenshots of all the packets captured for the HMI and on the emulated PLC. Both figures show that there are 23 packets transmitted during the test case for both Triangle MicroWorks and Modbus Poll, demonstrating the ability to correctly log the interactions.

During the state functionality test, the emulated PLC and the HMI captured 14 packets. Appendix C.1 provides screenshots of both Wireshark captures displaying the packets communicated across the channel. Figure 4.9 below shows that the logging device is able to log all the traffic sent to it during the test. The connection and tear down, both the read commands, and the write command are all logged.



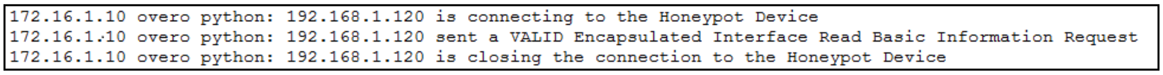
```
overo python: 192.168.1.11 is connecting to the Honeypot Device
overo python: 192.168.1.11 sent a valid Read Coil request
overo python: 192.168.1.11 sent a valid Write Single Coil request
overo python: 192.168.1.11 sent a valid Read Coil request
overo python: 192.168.1.11 is closing the connection to the Honeypot Device
```

Figure 4.9: Syslog entries from read, write, read test.

Logging is also examined during the port scan against the emulated device. The number of packets captured on the emulated PLC is compared to the number of packets captured on the HMI. During the intense scan of the emulated PLC, tcpdump fails to log all the packets. The amount of packets that are captured on the HMI is 147,295 packets, compared to only 44,838 packets captured on the emulated PLC. The device is able to respond to all operational traffic; however, it is not able to log all messages. The tcpdump records packets for approximately eleven seconds then ceases to log packets for approximately fifteen minutes while it responds to the network traffic load. While not every packet was checked for a response during the test a look through the Wireshark capture on the HMI appeared as though the emulated PLC is able to respond to all the packets being sent to the device. It appears as though the CPU cycles during this period of time are all allocated for the response to network traffic and the logging is not given any of these cycles. This is a shortfall in that an attack could flood the device and exploit the system without the events being logged. The syslog is also checked to see if the connections to the device are logged. The syslog, much like that of tcpdump, fails to log all connections and packets to the device. There are only 165 packets captured in the syslog and one connection to the emulated PLC recorded.

Logging was also evaluated in the banner grabbing test. The traffic comparison between the number of packets captured on the HMI and the number of packets captured on the emulated PLC (8 packets) are equal for the command sent from Triangle MicroWorks. The Wireshark captures from the HMI and emulated PLC are shown in Appendix C.3.

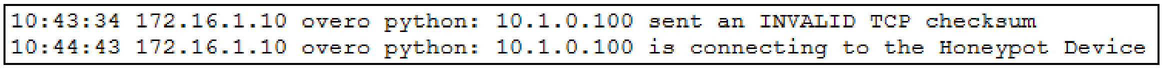
The verification that interactions with the emulated PLC are logged on a remote device is also evaluated. The connection with the emulated PLC, the request command and the connection tear down are all identified and logged. Figure 4.10 shows the connection and command sent from Triangle MicroWorks as logged in the syslog.



```
172.16.1.10 overo python: 192.168.1.120 is connecting to the Honeypot Device
172.16.1.10 overo python: 192.168.1.120 sent a VALID Encapsulated Interface Read Basic Information Request
172.16.1.10 overo python: 192.168.1.120 is closing the connection to the Honeypot Device
```

Figure 4.10: Syslog entries for banner grab.

The invalid traffic logging is also evaluated to see if the emulated PLC correctly logged all interactions. The number of packets captured on the emulated PLC correlates with the number of packets captured on the HMI (4 packets). Appendix C.4 provides screenshots of the captured traffic. The verification that any interaction with the emulated PLC is logged on a remote logging device is checked. In this case, the failed connection with the emulated PLC is logged and the successful completion of the SYN with the valid TCP checksum is also shown in the logs. Figure 4.11 below shows the connection and command sent from Triangle MicroWorks as logged in the syslog.



```
10:43:34 172.16.1.10 overo python: 10.1.0.100 sent an INVALID TCP checksum
10:44:43 172.16.1.10 overo python: 10.1.0.100 is connecting to the Honeypot Device
```

Figure 4.11: Syslog entry from invalid TCP checksum.

4.3.5 Qualitative Evaluation

An ICS Subject Matter Expert (SME) from the 262nd Air National Guard unit evaluated the emulated PLC using assessment techniques. The emulated PLC responded in a manner consistent with an operational PLC during evaluation. The individual stated that based on the Modbus characteristics, operational parameters and interactive sessions, the emulated PLC would have been considered an operational PLC typically encountered during an ICS assessment. The findings indicate that an attacker attempting to exploit a PLC target would not readily discern the differences between the emulated PLC and an operational PLC. The ICS SME from the 262nd recommended inclusion of a web server in the next iteration, as this is the service most used for remote access and exploitation by malicious actors.

4.4 Analysis

The emulated PLC successfully emulates the six Modbus TCP commands based on the RFC as tested with Triangle MicroWorks Protocol Test Harness. The emulated PLC also maintains system state as expected in an operational PLC. The emulated PLC is responsive as an operational PLC instead of a Linux machine to three of the four fingerprinting methods common to ICS. The MAC address of the emulated PLC is easily configured to appear as a Siemens Automation PLC. Port scans for the PLC demonstrate Modbus TCP server process and the OS detection is not able to successfully fingerprint the device. The emulated PLC is able to successfully respond to banner grabbing techniques used to fingerprint a device running a Modbus server. The emulated PLC also successfully responds to invalid traffic in the same manner as other PLCs. Finally, evaluation of the emulated PLC by a subject matter expert demonstrates the ability to appear as a legitimate operational PLC to an external individual using ICS assessment techniques.

The emulation of the PLC is successful; however, a shortfall is identified with the logging functionality. The logging functionality failed to properly log all traffic during intense port scan. The logging is able to catch up before the scan is finished, however, that could miss valuable information during an attack. The logging capability requires further evaluation. The other services during the intense traffic load also need to be examined to evaluate if the performance of other services is degraded during this time. Regardless, the demonstrated ability is an improvement over current logging capabilities at the field device level.

4.5 Results Summary

The emulated PLC successfully passes fingerprinting techniques used to classify the device as a PLC. The emulated PLC successfully responds to Modbus TCP traffic and maintains the proper system state. The device also responds to invalid traffic in the same manner as legitimate PLCs. Although interaction with the emulated PLC was appropriately logged, further evaluation is required to determine traffic and bandwidth limits.

V. Conclusions and Recommendations

5.1 Conclusions

This research introduces a novel approach to help secure ICS. The PLC emulation device offers many capabilities associated with employment in the operational ICS environment. The device helps identify reconnaissance and exploitation attempts against an operational ICS. During scanning, an attacker attempts to identify available systems on the network. Once identified, an attacker may attempt to manipulate parameters to alter system functionality. In each instance, the PLC emulation device identifies the attempted actions and logs the events.

In addition to identifying attempted exploitation, the PLC emulation device offers situational awareness. Often times, asset owners have only awareness of network traffic and operating characteristics as reported at the HMI. The PLC emulation device characterizes network traffic patterns and identifies erroneous communications. Indeed, the device helps provide holistic awareness of the system and can be used as an early detection against propagating malware that is targeting ICS. Finally, the logging capability provides insight into attack characteristics. By deploying PLC emulation devices across a wide range of ICS, logging can be evaluated to determine attacker tactics and techniques.

Although the PLC emulation device offers security protections against an external attacker and malware, it is important to note that it may not be as effective against trusted insiders. Because insiders have explicit knowledge of ICS operations, awareness that the PLC emulation device is employed may result in the attacker avoiding communication with the device. Regardless, the approach demonstrates utility for increasing the security posture for ICS. Indeed, use of the emulated PLC device affords a capability that is inexpensive, configurable, portable, and offers event logging.

5.2 Future Work

5.2.1 Further Protocol Development

Currently the emulated PLC only emulates a portion of the Modbus TCP protocol. Follow on work includes development of the additional functions in the Modbus TCP standard to create a more robust solution. Further development also includes adding additional ICS communication protocols such as DNP3 and EtherNet/IP to make the device.

5.2.2 Levels of Implementation

Follow on work for the emulated PLC includes expanding the current level of services offered by the device. Currently the device successfully emulates the protocol level and application level for Modbus TCP. The stack level is partially emulated through the use of iptables and configuring responses in Scapy to respond in a similar manner to ICS devices. Future work is to fully implement the stack level of a PLC. Scapy can be used to fully implement a response for all iterations of packets. Working on additional application level programs such as a web server allow for enhanced PLC emulation.

5.2.3 Response Time

Future work requires comparison of the response time for the current emulated PLC to that of a real PLC. With the knowledge of honeypots in the IT sector, response time is used to determine if an attacker is communicating with a legitimate computer or if it is a honeypot.

5.2.4 Traffic Loss

During this research there was traffic that failed to be logged during heavy traffic loads. Determining the reason for traffic loss and a solution to better handle the traffic when it increases is important. Successfully capturing all packets is needed to help determine the attack characteristics in ICS networks.

5.2.5 Ladder Logic and Firmware Implementation

The implementation of ladder logic allows enhanced emulation of a PLC. If ladder logic is implemented, the devices the values on the device would be constantly changing to help trick an attacker. If an attacker was scanning the network waiting to see how the values are changing then the device could emulate the fluctuation of a pressure sensor reading changing frequently.

Likewise is the ability to allow firmware updates to a device. While it would not update any actual firmware, emulating the traffic to and from the emulated PLC would make an attacker assume that he is interacting with a real PLC. Once the firmware update has completed, the emulation will then save that state so if later the same attacker attempted to scan for current firmware it would appear as though the firmware is the new version.

5.2.6 Serial Implementation

Implementing an emulated device that communicates over serial lines instead of Ethernet TCP/IP cables would be the next capture interface. Many devices still communicate over serial lines and adding the capability enables the device to broaden the array of devices that it can emulate. Much of the communication that is discussed in current ICS systems is the communication over Ethernet because it is readily accessible. Serial communication would have to communicate through an HMI that could be configured as a honeypot.

5.2.7 Ethernet Header Manufacturing Tags

Ethernet header manufacturing tags are another way to commonly fingerprint an ICS device and this needs to be evaluated as future work. This also can be implemented with the use of Scapy easily once the knowledge of the header tags in acquired.

5.3 Concluding Remarks

The primary goal of this research is to develop an inexpensive, configurable and portable emulation device that contains logging capabilities. In order to properly emulate a PLC, the emulated PLC device avoids common fingerprinting techniques specific to ICS devices. This research develops such a device that is able to be expanded upon and deployed to a live environment to better characterize and identify attacks on ICS networks.

Appendix A: Setting Up Emulated PLC

I. BUILDING OVERO OPEN EMBEDDED IMAGE

Guide: http://gumstix.org/software-development/open-embedded/61-using-the-open-embedded-build-system.html

1) Build a new machine with the Ubuntu 10.10 x86 ISO file to act as the development laptop.
    a. http://releases.ubuntu.com/10.10/ubuntu-10.10-desktop-i386.iso
2) Once booted, use the Update Manager to update the default packages. Do not upgrade to Ubuntu 11.04 or other versions.
3) Open the synaptic package manager and select the following packages for install:
    a. git
    b. subversion
    c. gcc
    d. build-essential
    e. help2man
    f. diffstat
    g. texi2html
    h. texinfo
    i. libncurses5-dev
    j. cvs
    k. gawk
    l. python2.7-dev
    m. python-pysqlite2
    n. unzip
    o. chrpath
    p. ccache
4) sudo dpkg-reconfigure dash
    a. Answer **No** when asked whether you want to install dash as /bin/sh.
5) mkdir -p ~/overo-oe
6) cd ~/overo-oe
7) git clone git://gitorious.org/gumstix-oe/mainline.git org.openembedded.dev
8) cd org.openembedded.dev
9) git checkout --track -b overo-2011.03 origin/overo-2011.03
10) cd ~/overo-oe
11) git clone git://git.openembedded.org/bitbake bitbake
12) cd bitbake
13) git checkout 1.12.0
14) cd ~/overo-oe
15) cp -r org.openembedded.dev/contrib/gumstix/build .
16) cp ~/.bashrc ~/bashrc.bak
17) cat ~/overo-oe/build/profile >> ~/.bashrc
18) Close the Terminal window and open a new one.
19) gedit ~/overo-oe/org.openembedded.dev/recipes/images/omap3-console-image.bb

   a. Add iptables to the TOOLS_INSTALL section
   b. Save and close the window

20) bitbake omap3-console-image
21) The Overo file system is built at: ~/overo-oe/tmp/deploy/glibc/images/overo/omap3-console-image-overo.tar.bz2
22) The Overo OE Linux Kernel is built at: ~/overo-oe/tmp/deploy/glibc/images/overo/uImage-overo.bin

II. RECONFIGURING THE OVERO KERNEL TO INCLUDE IPTABLES SUPPORT

Guide: http://gumstix.8.n6.nabble.com/iptables-on-Overo-td663707.html

1) On the development laptop:
2) cd ~/overo-oe
3) mkdir -p ./user.collection/recipes
4) cp -r ./org.openembedded.dev/recipes/linux /home/<user>/overo-oe/user.collection/recipes
   a. (bitbake looks at user.collection first. org.embedded.dev holds the original copy)
5) cd ~/overo-oe/tmp/work/overo-angstrom-linux-gnueabi/linux-omap3<kernel version>/git
6) make menuconfig ARCH=arm
   a.

```
→ Networking Support
        → Networking Options
            [*] Network Packet Filtering (netfilter)
            → Network Packet Filtering (netfilter)
                → Core Netfilter Configuration
                ENABLE [M] all options in this menu
                → IP: Netfilter Configuration
                [*] proc/sysctl compatibility with old connection tracking
                ENABLE [M] all other menu options
                →IPv6: Netfilter Configuration
                ENABLE [M] all options in this menu
```

   b. Exit
   c. Save: Yes
7) ls –al
   a. Check that date was made today
8) cp ./.config ~/overo-oe/user.collection/recipes/linux/linux-omap3/overo/defconfig
9) cd ~/overo
10) bitbake –c clean linux-omap3
11) bitbake –c build linux-omap3
12) The Overo OE Linux Kernel is built at: ~/overo-oe/tmp/deploy/glibc/images/overo/uImage-overo.bin

III. PARTITIONING BOOTABLE SD CARD FOR OVERO IMAGE

Guide: http://gumstix.org/create-a-bootable-microsd-card.html
Guide: http://gumstix.org/how-to/70-writing-images-to-flash.html

1) df
2) umount /media/…
3) umount /…

IV. DEPLOYING OVERO IMAGE

1) On the development laptop:
2) Delete the current file structure, if any, on the EXT3 partition of the micro SD card
    a. sudo nautilus
    b. Edit > Preferences > Behavior > Check Include a Delete command that bypasses Trash
    c. Select rootfs
    d. Select all files > Right Click > Delete
3) Copy the contents of ~/overo-oe/tmp/deploy/glibc/images/overo/omap3-console-image-overo.tar.bz2 into the rootfs partition of the micro SD card.
4) On the micro SD card FAT partition:
    a. Delete uImage
    b. Copy uImage-<kernel version>-overo.bin into /
    c. Rename uImage-<kernel version>-overo.bin to uImage

V. BOOTING OVERO IMAGE CONSOLE

1) Power off the Overo board.
2) Insert the newly created micro SD card into the micro SD slot of the Overo board.
3) Connect a USB cable between the “Console” mini USB B port on the Overo board and the development laptop with ckermit installed.
4) On the development laptop create a file called overo_serial.cfg

```
set line /dev/ttyUSB0  (Note: 0 might changed)
set flow-control none
set carrier-watch off
set speed 115200
set reliable
fast
set prefixing all
set file type bin
set rec pack 4096
set send pack 4096
set window 5
 connect
```

5) Open a terminal and type:
    a. kermit

        i. take overo_serial.cfg

6) Power on the Overo board. You should see the boot sequence displayed on the terminal.
7) Break the boot sequence when prompted then type:
    a. nand erase 240000 20000
    b. reset
8) Enter “root” as the username to log in.
9) To exit kermit:
    a. ctrl-/-c
    b. Type: exit

VI. COMPILING SCAPY FOR OVERO IMAGE

1) On the development laptop
    a. Go to www.secdev.org/projects/scapy
    b. Scroll down to the section labeled Download
    c. Download Scapy’s latest revision
    d. Unzip the folder to the desktop
    e. scp –r <foldername> <IP address of gumstix>:/home/root
    f. You will also want to move over the canary.py script
    g. scp –r canary.py <IP address if gumstix>:/home/root
2) On the gumstix now type the following commands
    a. opkg update
    b. opkg install python-core
    c. opkg install python-modules
    d. mkdir /usr/include/python2.6
3) Go back to the development laptop
    a. sudo scp /usr/include/python2.6/pyconfig.h <IP address>:/usr/include/python2.6
4) On the gumstix
    a. cd <foldername of scapy files>
    b. python setup.py install

VII. COMPILING TCPDUMP FOR OVERO IMAGE

1) On the development laptop:
    a. bitbake tcpdump
2) Packages will be built in: /overo-oe/tmp/deploy/glibc/ipk/armv7a
3) Copy the packages onto the Overo EXT3 partition
    a. sudo scp ./tcpdump_<version number>.ipk <overo IP address>:/home/root
4) On the Overo console, install the package
    a. opkg install ./tcpdump_<version number>.ipk

VIII. COMPILING BITSTRING FOR OVERO IMAGE

1) On the development laptop
    a. Go to http://code.google.com/p/python-bitstring/downloads/list
    b. Download bitstring latest revision

    c. Unzip the folder to the desktop
    d. scp –r <foldername> <IP address of gumstix>:/home/root

2) On the gumstix
    a. cd <foldername of bitstring files>
    b. python setup.py install

IX. REMOVING UNWANTED PACKAGES

1) update-rc.d –f ntpd remove
2) update-rc.d –f avahi-daemon remove
3) update-rc.d –f portmap remove

X. CONFIGURING STARTUP SCRIPT

1) Create a file in the /etc/init.d directory called canary.sh with the contents below

```
#!/bin/bash

/etc/init.d/networking start

ifconfig eth0 hw ether 00:0e:8c:bb:1f:56
ifconfig eth0 up
dhclient eth0

ifconfig eth1 up
ifconfig eth1 172.16.1.10

/etc/init.d/sshd start

iptables -A OUTPUT -p tcp --sport 502 -j DROP
iptables -A INPUT -p tcp --sport 502 -j ACCEPT
iptables -A INPUT -j DROP -p tcp --sport 22
iptables -A INPUT -j DROP -p tcp --sport 111
iptables -A INPUT -j LOG --log-level 6 -m pkttype --pkt-type host -i eth0
iptables -A INPUT -j REJECT --reject-with tcp-reset -i eth0
iptables -A FORWARD -j REJECT -i eth0

nohup tcpdump -s 0 -i eth0 -C 10 -w /tmp/capture.pcap &

nohup python /home/root/canary.py &
```

2) update-rc.d canary.sh defaults 100

X. CONFIGURING SSH TO RUN ON ETH1 ONLY

1) cd /etc/ssh
2) vi sshd_config
    a. Add the lines (These lines may be commented and you just need to uncomment them.)

        i. Port 22
        ii. AddressFamily inet
        iii. ListenAddress 172.16.1.10
    b. Restart ssh (/etc/init.d/sshd restart)

XI. CONFIGURING THE OVERO BOARD TO WORK WITH THE TOBI DUO

1) This is only needed if you have used the Tobi board to set up the Overo Board
2) Place the Overo Board on the Tobi Duo Expansion Board and power on the board.
3) Once the board has come online (Detected by the blue light on the CPU stops flashing) unplug the board and place the board back on the Tobi Expansion board.
4) Turn on the Overo
5) vi /etc/udev/rules.d/70-persistant-net.rules
6) There should now be three net device () lines in this file eth0 – eth2.
7) Edit the eth1 line so that NAME=”eth0”
8) Edit the eth2 line so that NAME=”eth1”
9) If you restart the Overo with the Tobi-Duo extension you should now be able to SSH to 172.16.1.10.

XII. CONFIGURING THE SYSLOG SERVER

1) Edit the /etc/init.d/sysklogd file.
2) Find the line SYSLOGD=“” and replace it with SYSLOGD=”-rm 0”
3) You will also need to edit the /etc/syslog.conf file.
4) There is a line that starts with *.=info;…… -/var/log/messages
    a. After *.=info; add kern.!=info;
5) After this line also add in the line kern.=info /var/log/canary.log
6) Restart the syslog service
    a. /etc/init.d/syslog restart

XIII. CONFIGURING SYSLOG ON THE GUMSTIX

1) Edit the file /etc/syslog-ng.conf
2) In the destination section add in the following line.
    a. Destination logging {udp(“172.16.1.11” port(514));};
3) Further down in the log section add the following line.
    a. log { source(src); destination(logging); };
4) Restart the syslog service
    a. /etc/init.d/syslog restart

Appendix B: Canary.py code

```
#! /usr/bin/python

### Dustin Berman
### AFIT/ENG
### Masters of Cyber Operations, June 2012

### File Information
### canary.py
### Emulates a PLC with the following commands: Read Coil, Read Discrete Inputs, Read Holding
Registers, Read Input Registers, Write Single Coil, Write Single Register
### This will also log any connections to the syslog

# Imports
import logging, platform, random
import syslog
from struct import *
from bitstring import BitArray

# Designed with Scapy 2.2.0
logging.getLogger("scapy").setLevel(1)
from scapy.all import *

# Display the version of Python and Scapy being used
print "Python %s\tScapy %s" % (platform.python_version(), conf.version)

#Global Variables
numcoils = 100
numdinputs = 100
numinputregisters = 100
numholdregisters = 100
coil = ['0']*numcoils
dinputs = ['0']*numdinputs
inputregister = ['\x00\x00']*numinputregisters
holdregister = ['\x00\x00']*numholdregisters
vendorname = "Allen Bradley"
productcode = "Micrologix 1500"
majorminorrevision = "V1.12.1"
ipid = random.randint(1,65535)

# Dictionaries
# Need to add in all function codes here
function_code_enum = {1:"Read Coil", 2:"Read Discrete Inputs", 3:"Read Holding Registers", 4:"Read
Input Registers", 5:"Write Single Coil", 6:"Write Single Register", 43:"Encapsulated Interface Transport"}
function_code = {"Read Coil":1, "Read Discrete Inputs":2, "Read Holding Registers":3, "Read Input
Registers":4, "Write Single Coil":5, "Write Single Register":6, "Encapsulated Interface Transport":43}

# Modbus Header
class Modbus(Packet):
        name = "Modbus"
        fields_desc = [ShortField("transaction", 0),
                ShortField("protocol", 0),
                ShortField("length", 0),
                ByteField("unit", 0),
                ByteEnumField("function", 1, function_code_enum)
                ]

        # This will determine how to dissect the rest of the packet
        def guess_payload_class(self, payload):
                if self.function == function_code['Read Coil']:
                    return ReadCoil
                elif self.function == function_code['Read Discrete Inputs']:
                    return ReadDiscreteInputs
                elif self.function == function_code['Read Holding Registers']:
                    return ReadHoldingRegisters
                elif self.function == function_code['Read Input Registers']:
                    return ReadInputRegisters
                elif self.function == function_code['Write Single Coil']:
                    return WriteSingleCoil
                elif self.function == function_code['Write Single Register']:
                    return WriteSingleRegister
                elif self.function == function_code['Encapsulated Interface Transport']:
                    return EncapsulatedInterfaceRequest
                else:
                    return Packet.guess_payload_class(self,payload)

# Read Coil Payload
class ReadCoil(Packet):
        name= "ReadCoil"
        fields_desc = [ShortField("startcoil", 0),
                ShortField("quantitycoils", 0)
                ]

# Read Coil Response Payload
class ReadCoilResponse(Packet):
        name= "ReadCoilResponse"
        fields_desc = [ByteField("bytecount", 0),
                StrField("status", "")
                ]

# Read Discrete Inputs Payload
class ReadDiscreteInputs(Packet):
        name= "ReadDiscreteInputs"
        fields_desc = [ShortField("startinput", 0),
                ShortField("quantityinputs", 0)
                ]

# Read Discrete Inputs Response Payloads
class ReadDiscreteInputsResponse(Packet):
        name= "ReadDiscreteInputsResponse"
        fields_desc = [ByteField("bytecount", 0),
                StrField("status", "")
                ]

# Read Holding Registers Payload
class ReadHoldingRegisters(Packet):
        name= "ReadHoldingRegisters"
```

```
        fields_desc = [ShortField("startaddress", 0),
                ShortField("quantityregs", 0)
                ]

# Read Holding Registers Response Payload
class ReadHoldingRegistersResponse(Packet):
        name= "ReadHoldingRegistersResponse"
        fields_desc = [ByteField("bytecount", 0),
                StrField("status", "")
                ]

# Read Input Registers Payload
class ReadInputRegisters(Packet):
        name= "ReadInputRegisters"
        fields_desc = [ShortField("startaddress", 0),
                ShortField("quantityregs", 0)
                ]

# Read Input Registers Response Payload
class ReadInputRegistersResponse(Packet):
        name= "ReadHoldingRegistersResponse"
        fields_desc = [ByteField("bytecount", 0),
                StrField("status", "")
                ]

# Write Single Coil Payload
class WriteSingleCoil(Packet):
        name = "WriteSingleCoil"
        fields_desc = [ShortField("coilnumber", 0),
                ByteField("state", 0),
                ByteField("padding", 0)
                ]

# Write Single Register Payload
class WriteSingleRegister(Packet):
        name= "WriteSingleRegister"
        fields_desc = [ShortField("regaddress", 0),
                ShortField("regvalue", 0)
                ]

# Encapsulated Interface Transport Request Payload
class EncapsulatedInterfaceRequest(Packet):
        name= "EncapsulatedInterfaceRequest"
        fields_desc = [ByteField("meitype", 0),
                ByteField("deviceid", 1),
                ByteField("objectid", 0)
                ]

# Encapsulated Interface Transport Response Payload
class EncapsulatedInterfaceResponse(Packet):
        name= "EncapsulatedInterfaceResponse"
        fields_desc = [ByteField("meitype", 0),
                ByteField("deviceid", 1),
                ByteField("conformity", 1),
        ByteField("morefollows", 0),
        ByteField("objectid", 0),
        ByteField("numobjects",0)
        ]

# Encapsulated Interface Transport Object Payload
class EncapsulatedInterfaceObject(Packet):
    name= "EncapsulatedInterfaceObject"
    fields_desc = [ByteField("objectid", 0),
        ByteField("objectlength", 0),
        StrField("objectvalue","")
        ]

# Error Payload
class Error(Packet):
    name= "Error"
    fields_desc = [ByteField("code", 1)
        ]

# Bind Layers
bind_layers(TCP, Modbus, sport = 502)
bind_layers(TCP, Modbus, dport = 502)

# Responding to a SYN request
def responsesyn(packet):
    global ipid
    #Write the connection to the syslog
    syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' is connecting to the Honeypot Device')

    #Build a packet to send back
    response = Ether()/IP()/TCP()
    response[Ether].src = packet[Ether].dst
    response[Ether].dst = packet[Ether].src
    response[IP].src = packet[IP].dst
    response[IP].dst = packet[IP].src
    ipid = random.randint(1,65535)
    response[IP].id = ipid
    ipid = ipid + 1
    response[IP].flags = 2
    response[TCP].sport = packet[TCP].dport
    response[TCP].dport = packet[TCP].sport
    response[TCP].seq = random.randint(1,4294967295)
    response[TCP].ack = packet[TCP].seq + 1
    response[TCP].flags = 'SA'
    if packet[TCP].window == 1 or packet[TCP].window == 63 or packet[TCP].window == 4 or
packet[TCP].window == 16:
        response[TCP].options = [('MSS', 1460), ('NOP', None), ('WScale', 0), ('NOP', None),
('NOP', None), ('Timestamp', (0, 4294967295))]
    elif packet[TCP].window == 512:
        response[TCP].options = [('MSS', 1460),('NOP', None), ('NOP', None), ('Timestamp', (0,
4294967295))]
    else:
        response[TCP].options = packet[TCP].options
    del(response[IP].chksum)
    del(response[TCP].chksum)
    del(response[IP].len)
    #sendp will recalculate the checksums and IP length before sending the packet.
    sendp(response, loop=0)

# Responding to a SYN request
def responserstack(packet):
    global ipid
    #Write the connection to the syslog
    syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' is closing the connection to the Honeypot
Device')

    #Build a packet to send back
    response = Ether()/IP()/TCP()
    response[Ether].src = packet[Ether].dst
    response[Ether].dst = packet[Ether].src
    response[IP].src = packet[IP].dst
    response[IP].dst = packet[IP].src
    response[IP].id = ipid
    ipid = ipid + 1
    response[IP].flags = 2
    response[TCP].sport = packet[TCP].dport
    response[TCP].dport = packet[TCP].sport
    response[TCP].seq = packet[TCP].ack
    response[TCP].ack = packet[TCP].seq + 1
    response[TCP].flags = 'RA'
    response[TCP].options = packet[TCP].options
    del(response[IP].chksum)
    del(response[TCP].chksum)
    del(response[IP].len)
    #sendp will recalculate the checksums and IP length before sending the packet.
    sendp(response, loop=0)

# Building a response to Read Coil
def responsereadcoil(packet):
    global ipid
    # Build a packet to send back
    response = Ether()/IP()/TCP()

    if packet.haslayer(ReadCoil):

        # This checks to see if the request was valid in the number of coils it requested.
        if packet[ReadCoil].quantitycoils > numcoils or packet[ReadCoil].quantitycoils < 1:
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Read
Coil request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 3
        # This checks to see if the starting address is valid and the starting address + Quatity of
Outputs is valid
        elif (packet[ReadCoil].startcoil < 0 or packet[ReadCoil].startcoil > numcoils) or
((packet[ReadCoil].startcoil + packet[ReadCoil].quantitycoils) > numcoils):
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Read
Coil request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 2

        # Else the Read Coil was a valid command
        else:
            response = response/Modbus()/ReadCoilResponse()
            # Write the valid request to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent a valid Read Coil
request')

            bytecount = (packet[ReadCoil].quantitycoils/8)
            partbytecount = packet[ReadCoil].quantitycoils%8
            if partbytecount != 0:
                bytecount = bytecount + 1
            response[ReadCoilResponse].bytecount = bytecount
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = response[ReadCoilResponse].bytecount + 3
            leadzero = (8 - partbytecount)%8
            status = []
            output = ''

            # This is + 7 because each high order bit is the highest output address.
            # (Look at Modbus Application Protocol Specification V1.1b at www.Modbus-
IDA.org)
            current = packet[ReadCoil].startcoil + 7

            # This will build each byte to be sent back to the master.
            for x in range(0, bytecount):
                if x == (bytecount-1):
                    for z in range(current,current-leadzero, -1):
                        status.append('0')
                    for k in range(current-leadzero, current-8, -1):
                        status.append(coil[k])
                    current = current + 8
                    status = ''.join(status)
                    # Below will take the bits and create a byte to be added to the
output string.
                    temp = BitArray(bin=status)
                    value = temp.uint
                    string = pack('!h', value)
                    output = output + string[1]
                    status = []
                else:
                    for y in range(current, current-8, -1):
                        status.append(coil[y])
                    current = current + 8
                    status = ".join(status)
                    # Below will take the bits and create a byte to be added to the
output string.
                    temp = BitArray(bin=status)
                    value = temp.uint
                    string = pack('!h', value)
                    output = output + string[1]
                    status = []

            response[ReadCoilResponse].status = output
        response[Ether].src = packet[Ether].dst
        response[Ether].dst = packet[Ether].src
        response[IP].flags = 0
        response[IP].ttl = 64
        response[IP].id = ipid
        ipid = ipid + 1
        response[IP].src = packet[IP].dst
        response[IP].dst = packet[IP].src
        response[TCP].flags = 'PA'
        response[TCP].sport = packet[TCP].dport
        response[TCP].dport = packet[TCP].sport
        response[TCP].seq = packet[TCP].ack
        response[TCP].ack = packet[TCP].seq + 12
        response[TCP].window = 4096
        del(response[IP].chksum)
        del(response[TCP].chksum)
        del(response[IP].len)
        #sendp will recalculate the checksums and IP length before sending the packet.
        sendp(response, loop=0)

    else:
        responseerror(packet)

# Building a response to Read Discrete Inputs
def responsereaddiscreteinputs(packet):
    global ipid
    # Build a packet to send back
    response = Ether()/IP()/TCP()

    if packet.haslayer(ReadDiscreteInputs):
        # This checks to see if the request was valid in the number of inputs it requested.
        if packet[ReadDiscreteInputs].quantityinputs > numdinputs or
packet[ReadDiscreteInputs].quantityinputs < 1:
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Read
Discrete Input request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 3

        # This checks to see if the starting address is valid and the starting address + Quatity of
Outputs is valid
        elif (packet[ReadDiscreteInputs].startinput < 0 or packet[ReadDiscreteInputs].startinput
>= numdinputs) or ((packet[ReadDiscreteInputs].startinput + packet[ReadDiscreteInputs].quantityinputs)
>= numdinputs):
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Read
Discrete Input request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 2

        # Else the Read Discrete Inputs was a valid command
        else:
            response = response/Modbus()/ReadDiscreteInputsResponse()
            # Write the valid request to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent a valid Read Discrete
Input request')

            bytecount = (packet[ReadDiscreteInputs].quantityinputs/8)
            partbytecount = packet[ReadDiscreteInputs].quantityinputs%8
            if partbytecount != 0:
                bytecount = bytecount + 1
            response[ReadDiscreteInputsResponse].bytecount = bytecount
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].function = packet[Modbus].function
            response[Modbus].length = response[ReadDiscreteInputsResponse].bytecount +
3
            leadzero = (8 - partbytecount)%8
            status = []
            output = ''

            # This is + 7 because each high order bit is the highest output address.
            # (Look at Modbus Application Protocol Specification V1.1b at www.Modbus-
IDA.org)
            current = packet[ReadDiscreteInputs].startinput + 7

            # This will build each byte to be sent back to the master.
            for x in range(0, bytecount):
                if x == (bytecount-1):
                    for z in range(current,current-leadzero, -1):
                        status.append('0')
                    for k in range(current-leadzero, current-8, -1):
                        status.append(dinputs[k])
                    current = current + 8
                    # Below will take the bits and create a byte to be added to the
output string.
                    status = ''.join(status)
                    temp = BitArray(bin=status)
                    value = temp.uint
                    string = pack('!h', value)
                    output = output + string[1]
                    status = []
                else:
                    for y in range(current, current-8, -1):
                        status.append(dinputs[y])
                    current = current + 8
                    # Below will take the bits and create a byte to be added to the
output string.
                    status = ''.join(status)
                    temp = BitArray(bin=status)
                    value = temp.uint
                    string = pack('!h', value)
                    output = output + string[1]
                    status = []

            response[ReadDiscreteInputsResponse].status = output

        response[Ether].src = packet[Ether].dst
        response[Ether].dst = packet[Ether].src
        response[IP].flags = 0
        response[IP].ttl = 64
        response[IP].id = ipid
        ipid = ipid + 1
        response[IP].src = packet[IP].dst
        response[IP].dst = packet[IP].src
        response[TCP].flags = 'PA'
        response[TCP].sport = packet[TCP].dport
        response[TCP].dport = packet[TCP].sport
        response[TCP].seq = packet[TCP].ack
        response[TCP].ack = packet[TCP].seq + 12
        response[TCP].window = 4096
        del(response[IP].chksum)
        del(response[TCP].chksum)
        del(response[IP].len)
        #sendp will recalculate the checksums and IP length before sending the packet.
        sendp(response, loop=0)

    else:
        responseerror(packet)

# Building a response to Read Holding Registers
def responsereadregisters(packet):
    global ipid
    # Build a packet to send back
    response = Ether()/IP()/TCP()

    if packet.haslayer(ReadHoldingRegisters):
        # This checks to see if the request was valid in the number of registers it requested.
        if packet[ReadHoldingRegisters].quantityregs < 1 or
packet[ReadHoldingRegisters].quantityregs > numholdregisters:
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Read
Holding Registers request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 3

        # This checks to see if the starting address is valid and the starting address + Quatity of
Outputs is valid
        elif (packet[ReadHoldingRegisters].startaddress < 0 or
packet[ReadHoldingRegisters].startaddress >= numholdregisters) or
((packet[ReadHoldingRegisters].startaddress + packet[ReadHoldingRegisters].quantityregs) >=
numholdregisters):
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Read
Holding Registers request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 2

        # Else the Read Holding Register Request is valid
        else:
            response = response/Modbus()/ReadHoldingRegistersResponse()
            # Write the valid request to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent a valid Read Holding
Registers request')

            response[ReadHoldingRegistersResponse].bytecount =
packet[ReadHoldingRegisters].quantityregs * 2
            response[Modbus].length =
response[ReadHoldingRegistersResponse].bytecount + 3
            response[Modbus].function = packet[Modbus].function
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].transaction = packet[Modbus].transaction

            # This will loop and add all the values requested to the status.
            for x in range(packet[ReadHoldingRegisters].startaddress,
(packet[ReadHoldingRegisters].startaddress + packet[ReadHoldingRegisters].quantityregs)):
                response[ReadHoldingRegistersResponse].status =
response[ReadHoldingRegistersResponse].status + holdregister[x]

        response[Ether].src = packet[Ether].dst
        response[Ether].dst = packet[Ether].src
        response[IP].flags = 0
        response[IP].ttl = 64
        response[IP].id = ipid
        ipid = ipid + 1
        response[IP].src = packet[IP].dst
        response[IP].dst = packet[IP].src
        response[TCP].flags = 'PA'
        response[TCP].sport = packet[TCP].dport
        response[TCP].dport = packet[TCP].sport
        response[TCP].seq = packet[TCP].ack
        response[TCP].ack = packet[TCP].seq + 12
        response[TCP].window = 4096
        del(response[IP].chksum)
        del(response[TCP].chksum)
        del(response[IP].len)
        #sendp will recalculate the checksums and IP length before sending the packet.
        sendp(response, loop=0)

    else:
        responseerror(packet)

# Building a response to Read Input Registers
def responsereadinputregisters(packet):
    global ipid
    # Build a packet to send back
    response = Ether()/IP()/TCP()

    if packet.haslayer(ReadInputRegisters):
        # This checks to see if the request was valid in the number of registers it requested.
        if packet[ReadInputRegisters].quantityregs < 1 or
packet[ReadInputRegisters].quantityregs > numinputregisters:
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Read
Input Registers request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 3

        # This checks to see if the starting address is valid and the starting address + Quatity of
Outputs is valid
        elif (packet[ReadInputRegisters].startaddress < 0 or
packet[ReadInputRegisters].startaddress >= numinputregisters) or
((packet[ReadInputRegisters].startaddress + packet[ReadInputRegisters].quantityregs) >=
numinputregisters):
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Read
Input Registers request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 2

        # Else the Read Input Registers is valid
        else:
            response = response/Modbus()/ReadInputRegistersResponse()
            # Write the valid request to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent a valid Read Input
Registers request')

            response[ReadInputRegistersResponse].bytecount =
packet[ReadInputRegisters].quantityregs * 2
            response[Modbus].length = response[ReadInputRegistersResponse].bytecount +
3
            response[Modbus].function = packet[Modbus].function
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].transaction = packet[Modbus].transaction

            # This will loop and add all the values requested to the status.
            for x in range(packet[ReadInputRegisters].startaddress,
(packet[ReadInputRegisters].startaddress + packet[ReadInputRegisters].quantityregs)):
                response[ReadInputRegistersResponse].status =
response[ReadInputRegistersResponse].status + inputregister[x]

        response[Ether].src = packet[Ether].dst
        response[Ether].dst = packet[Ether].src
        response[IP].flags = 0
        response[IP].ttl = 64
        response[IP].id = ipid
        ipid = ipid + 1
        response[IP].src = packet[IP].dst
        response[IP].dst = packet[IP].src
        response[TCP].flags = 'PA'
        response[TCP].sport = packet[TCP].dport
        response[TCP].dport = packet[TCP].sport
        response[TCP].seq = packet[TCP].ack
        response[TCP].ack = packet[TCP].seq + 12
        response[TCP].window = 4096
        del(response[IP].chksum)
        del(response[TCP].chksum)
        del(response[IP].len)
        #sendp will recalculate the checksums and IP length before sending the packet.
        sendp(response, loop=0)

    else:
        responseerror(packet)

# Building a response to Write Single Coil
def responsewritecoil(packet):
    global ipid
    # Build a packet to send back
    response = Ether()/IP()/TCP()
    if packet.haslayer(WriteSingleCoil):
        # This checks to see if the request value was valid.
        if not (packet[WriteSingleCoil].state == 0 or packet[WriteSingleCoil].state == 255):
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Write
Single Coil request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 3

        # This checks to see if the coil number is valid
        elif packet[WriteSingleCoil].coilnumber < 0 or packet[WriteSingleCoil].coilnumber >=
numcoils:
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Write
Single Coil request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 2

        # Else the Write Single Coil request is valid
        else:
            response = response/Modbus()/WriteSingleCoil()
            # Write the valid request to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent a valid Write Single
Coil request')
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = packet[Modbus].length
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function
            response[WriteSingleCoil].state = packet[WriteSingleCoil].state
            response[WriteSingleCoil].coilnumber = packet[WriteSingleCoil].coilnumber

            # If the value is 255 switch the value to 1 else if it is 0 switch the value to 0.
            if packet[WriteSingleCoil].state == 255:
                coil[packet[WriteSingleCoil].coilnumber] = '1'
            elif packet[WriteSingleCoil].state == 0:
                coil[packet[WriteSingleCoil].coilnumber] = '0'
        response[Ether].src = packet[Ether].dst
        response[Ether].dst = packet[Ether].src
        response[IP].flags = 0
        response[IP].ttl = 64
        response[IP].id = ipid
        ipid = ipid + 1
        response[IP].src = packet[IP].dst
        response[IP].dst = packet[IP].src
        response[TCP].flags = 'PA'
        response[TCP].sport = packet[TCP].dport
        response[TCP].dport = packet[TCP].sport
        response[TCP].seq = packet[TCP].ack
        response[TCP].ack = packet[TCP].seq + 12
        response[TCP].window = 4096
        del(response[IP].chksum)
        del(response[TCP].chksum)
        del(response[IP].len)
        #sendp will recalculate the checksums and IP length before sending the packet.
        sendp(response, loop=0)

    else:
        responseerror(packet)

# Building a response to Write Single Register
def responsewriteregister(packet):
    global ipid
    # Build a packet to send back
    response = Ether()/IP()/TCP()

    if packet.haslayer(WriteSingleRegister):
        # This checks to see if the request value was valid.
        if packet[WriteSingleRegister].regvalue < 0 or packet[WriteSingleRegister].regvalue >
65535:
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Write
Single Register request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 3

        # This checks to see if the register address is valid.
        elif packet[WriteSingleRegister].regaddress < 0 or
packet[WriteSingleRegister].regaddress >= numholdregisters:
            # Write the error to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID Write
Single Register request')

            response = response/Modbus()/Error()
            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = 3
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function + 128
            response[Error].code = 2

        # Else the Write Single Register request is valid
        else:
            response = response/Modbus()/WriteSingleRegister()
            # Write the valid request to the syslog
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent a valid Write Single
Register request')

            response[Modbus].transaction = packet[Modbus].transaction
            response[Modbus].length = packet[Modbus].length
            response[Modbus].unit = packet[Modbus].unit
            response[Modbus].function = packet[Modbus].function
            response[WriteSingleRegister].regaddress =
packet[WriteSingleRegister].regaddress
            response[WriteSingleRegister].regvalue = packet[WriteSingleRegister].regvalue
            # This updates the value of the register to be changed.
            holdregister[packet[WriteSingleRegister].regaddress] = pack('!h',
packet[WriteSingleRegister].regvalue)

        response[Ether].src = packet[Ether].dst
        response[Ether].dst = packet[Ether].src
        response[IP].flags = 0
        response[IP].ttl = 64
        response[IP].id = ipid
        ipid = ipid + 1
        response[IP].src = packet[IP].dst
        response[IP].dst = packet[IP].src
        response[TCP].flags = 'PA'
        response[TCP].sport = packet[TCP].dport
        response[TCP].dport = packet[TCP].sport
        response[TCP].seq = packet[TCP].ack
        response[TCP].ack = packet[TCP].seq + 12
        response[TCP].window = 4096
        del(response[IP].chksum)
        del(response[TCP].chksum)
        del(response[IP].len)
        #sendp will recalculate the checksums and IP length before sending the packet.
        sendp(response, loop=0)

    else:
        responseerror(packet)

def responseencapsulatedinterface(packet):
    global ipid
    # Build a packet to send back
    response = Ether()/IP()/TCP()
    if packet.haslayer(EncapsulatedInterfaceRequest):

        # This checks to see if the request was valid MEI type
        if packet[EncapsulatedInterfaceRequest].meitype == 14:
            # Basic Device Identification Stream
            if packet[EncapsulatedInterfaceRequest].deviceid == 1:
                # This must start with a 0
                if packet[EncapsulatedInterfaceRequest].objectid == 0:
                    syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent a
VALID Encapsulated Interface Read Basic Information Request')
                    response =
response/Modbus()/EncapsulatedInterfaceResponse()
```

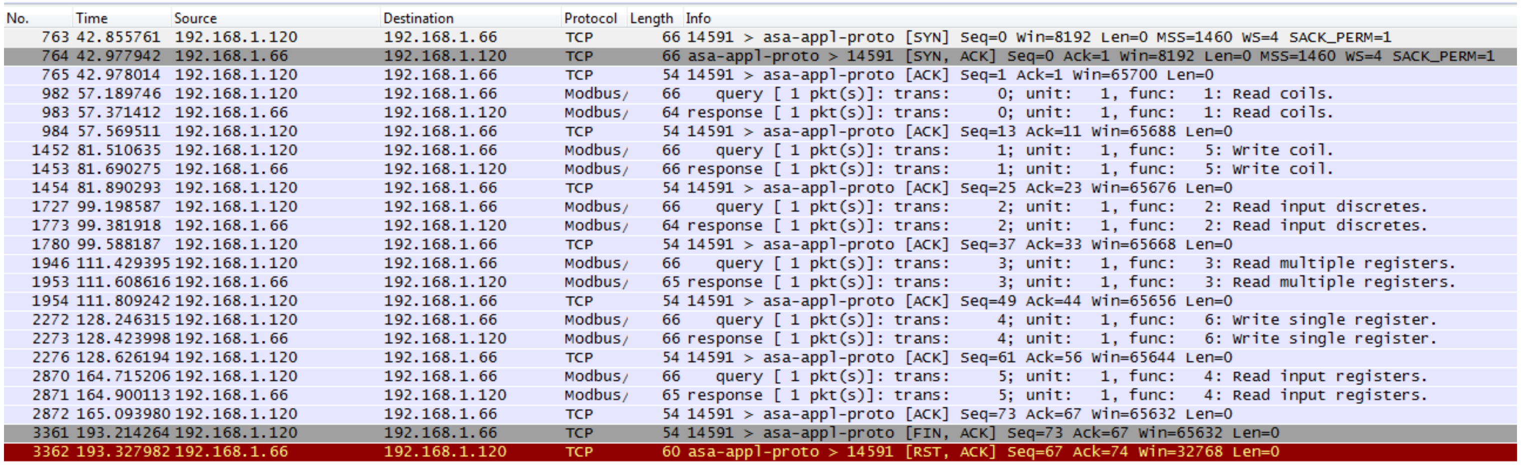
```
                    response[EncapsulatedInterfaceResponse].meitype =
packet[EncapsulatedInterfaceRequest].meitype
                    response[EncapsulatedInterfaceResponse].deviceid =
packet[EncapsulatedInterfaceRequest].deviceid
                    response[EncapsulatedInterfaceResponse].conformity = 1
                    response[EncapsulatedInterfaceResponse].morefollows = 0
                    response[EncapsulatedInterfaceResponse].objectid = 0
                    response[EncapsulatedInterfaceResponse].numobjects = 3
                    length = len(response[EncapsulatedInterfaceResponse])

                    #Building the objects to send in the packet
                    object1 = EncapsulatedInterfaceObject()
                    object1[EncapsulatedInterfaceObject].objectid = 0
                    object1[EncapsulatedInterfaceObject].objectvalue =
vendorname
                    object1[EncapsulatedInterfaceObject].objectlength =
len(object1[EncapsulatedInterfaceObject].objectvalue)
                    object2 = EncapsulatedInterfaceObject()
                    object2[EncapsulatedInterfaceObject].objectid = 1
                    object2[EncapsulatedInterfaceObject].objectvalue =
productcode
                    object2[EncapsulatedInterfaceObject].objectlength =
len(object2[EncapsulatedInterfaceObject].objectvalue)
                    object3 = EncapsulatedInterfaceObject()
                    object3[EncapsulatedInterfaceObject].objectid = 2
                    object3[EncapsulatedInterfaceObject].objectvalue =
majorminorrevision
                    object3[EncapsulatedInterfaceObject].objectlength =
len(object3[EncapsulatedInterfaceObject].objectvalue)
                    response = response/object1/object2/object3
                    length = length + len(object1) + len(object2) + len(object3) +
2
                    response[Modbus].length = length
                else:
                    syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an
INVALID Object ID code ' + str(packet[EncapsulatedInterfaceRequest].objectid))

                    response = response/Modbus()/Error()
                    response[Modbus].transaction = packet[Modbus].transaction
                    response[Modbus].length = 3
                    response[Modbus].unit = packet[Modbus].unit
                    response[Modbus].function = packet[Modbus].function + 128
                    response[Error].code = 2

            # One Specific Identification Object
            elif packet[EncapsulatedInterfaceRequest].deviceid == 4:
                if packet[EncapsulatedInterfaceRequest].objectid >= 0 and
packet[EncapsulatedInterfaceRequest].objectid < 3:
                    response =
response/Modbus()/EncapsulatedInterfaceResponse()/EncapsulatedInterfaceObject()
                    syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent a
VALID Encapsulated Interface Read Single Device Object Request')
                    response[EncapsulatedInterfaceResponse].meitype =
packet[EncapsulatedInterfaceRequest].meitype
```

```
                    response[EncapsulatedInterfaceResponse].deviceid =
packet[EncapsulatedInterfaceRequest].deviceid
                    response[EncapsulatedInterfaceResponse].conformity = 129
                    response[EncapsulatedInterfaceResponse].morefollows = 0
                    response[EncapsulatedInterfaceResponse].objectid = 0
                    response[EncapsulatedInterfaceResponse].numobjects = 1
                    response[EncapsulatedInterfaceObject].objectid =
packet[EncapsulatedInterfaceRequest].objectid
                    if packet[EncapsulatedInterfaceRequest].objectid == 0:
                        response[EncapsulatedInterfaceObject].objectvalue =
vendorname
                    elif packet[EncapsulatedInterfaceRequest].objectid == 1:
                        response[EncapsulatedInterfaceObject].objectvalue =
productcode
                    elif packet[EncapsulatedInterfaceRequest].objectid == 2:
                        response[EncapsulatedInterfaceObject].objectvalue =
majorminorrevision
                    response[EncapsulatedInterfaceObject].objectlength =
len(response[EncapsulatedInterfaceObject].objectvalue)
                    response[Modbus].length =
len(response[EncapsulatedInterfaceResponse]) + len(response[EncapsulatedInterfaceObject])
                    response[EncapsulatedInterfaceObject].objectid =
packet[EncapsulatedInterfaceRequest].objectid

                else:
                    syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an
INVALID Object ID code ' + str(packet[EncapsulatedInterfaceRequest].objectid))

                    response = response/Modbus()/Error()
                    response[Modbus].transaction = packet[Modbus].transaction
                    response[Modbus].length = 3
                    response[Modbus].unit = packet[Modbus].unit
                    response[Modbus].function = packet[Modbus].function + 128
                    response[Error].code = 2

            # The Read Device Code is not supported so respond with an error
            else:
                syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an
INVALID Read Device ID code ' + str(packet[EncapsulatedInterfaceRequest].deviceid))

                response = response/Modbus()/Error()
                response[Modbus].transaction = packet[Modbus].transaction
                response[Modbus].length = 3
                response[Modbus].unit = packet[Modbus].unit
                response[Modbus].function = packet[Modbus].function + 128
                response[Error].code = 3

        # The MEI Type is not supported
        else:
            syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID MEI
Type ' + str(packet[EncapsulatedInterfaceRequest].meitype))
            responseerror(packet)

        response[Modbus].transaction = packet[Modbus].transaction
        response[Modbus].unit = packet[Modbus].unit
        response[Modbus].function = packet[Modbus].function
        response[Ether].src = packet[Ether].dst
        response[Ether].dst = packet[Ether].src
        response[IP].flags = 0
        response[IP].ttl = 64
        response[IP].id = ipid
        ipid = ipid + 1
        response[IP].src = packet[IP].dst
        response[IP].dst = packet[IP].src
        response[TCP].flags = 'PA'
        response[TCP].sport = packet[TCP].dport
        response[TCP].dport = packet[TCP].sport
        response[TCP].seq = packet[TCP].ack
        response[TCP].ack = packet[TCP].seq + len(packet[TCP].payload)
        response[TCP].window = 4096
        del(response[IP].chksum)
        del(response[TCP].chksum)
        del(response[IP].len)
        #sendp will recalculate the checksums and IP length before sending the packet.
        sendp(response, loop=0)

    else:
        responseerror(packet)

def responseerror(packet):
    global ipid
    # Build a packet to send back
    response = Ether()/IP()/TCP()/Modbus()/Error()

    # Write the error to the syslog
    syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an INVALID function code ' +
str(packet[Modbus].function))

    response[Ether].src = packet[Ether].dst
    response[Ether].dst = packet[Ether].src
    response[IP].flags = 0
    response[IP].ttl = 64
    response[IP].id = ipid
    ipid = ipid + 1
    response[IP].src = packet[IP].dst
    response[IP].dst = packet[IP].src
    response[TCP].flags = 'PA'
    response[TCP].sport = packet[TCP].dport
    response[TCP].dport = packet[TCP].sport
    response[TCP].seq = packet[TCP].ack
    response[TCP].ack = packet[TCP].seq + len(packet[TCP].payload)
    response[TCP].window = 4096
    response[Modbus].transaction = packet[Modbus].transaction
    response[Modbus].length = 3
    response[Modbus].unit = packet[Modbus].unit

    # This will change the high order bit to one unless it is already a 1.
    if packet[Modbus].function > 127:
        response[Modbus].function = packet[Modbus].function
    else:
        response[Modbus].function = packet[Modbus].function + 128
    del(response[IP].chksum)
    del(response[TCP].chksum)
    del(response[IP].len)
    #sendp will recalculate the checksums and IP length before sending the packet.
    sendp(response, loop=0)


def response(packet):
    if packet.haslayer(TCP):
        if packet[TCP].dport == 502:
            originalChecksum=packet[TCP].chksum
            originalIPChecksum=packet[IP].chksum
            del packet[IP].chksum
            del packet[TCP].chksum
            packet=Ether(str(packet))
            recomputedIPChecksum=packet[IP].chksum
            recomputedChecksum=packet[TCP].chksum
            if originalChecksum == recomputedChecksum and originalIPChecksum ==
recomputedIPChecksum:
                if packet.haslayer(Modbus) and packet[Modbus].protocol == 0:
                    if packet[Modbus].function == function_code['Read Coil']:
                        responsereadcoil(packet)
                    elif packet[Modbus].function == function_code['Read
Discrete Inputs']:
                        responsereaddiscreteinputs(packet)
                    elif packet[Modbus].function == function_code['Read Holding
Registers']:
                        responsereadregisters(packet)
                    elif packet[Modbus].function == function_code['Read Input
Registers']:
                        responsereadinputregisters(packet)
                    elif packet[Modbus].function == function_code['Write Single
Coil']:
                        responsewritecoil(packet)
                    elif packet[Modbus].function == function_code['Write Single
Register']:
                        responsewriteregister(packet)
                    elif packet[Modbus].function == function_code['Encapsulated
Interface Transport']:
                        responseencapsulatedinterface(packet)
                    else:
                        responseerror(packet)
                elif packet[TCP].flags == 2:
                    #send a SYN ACK response
                    responsesyn(packet)
                elif packet[TCP].flags == 17:
                    #Closing down the connection to a FIN ACK
                    responserstack(packet)
            elif not originalChecksum == recomputedChecksum:
                syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an
INVALID TCP checksum')
```

```
            else:
                syslog.syslog(syslog.LOG_ALERT, packet[IP].src + ' sent an
INVALID IP checksum')
    return

## This is the Main Script
sniff(iface="eth0", store = 0, prn=lambda x: response(x))
```

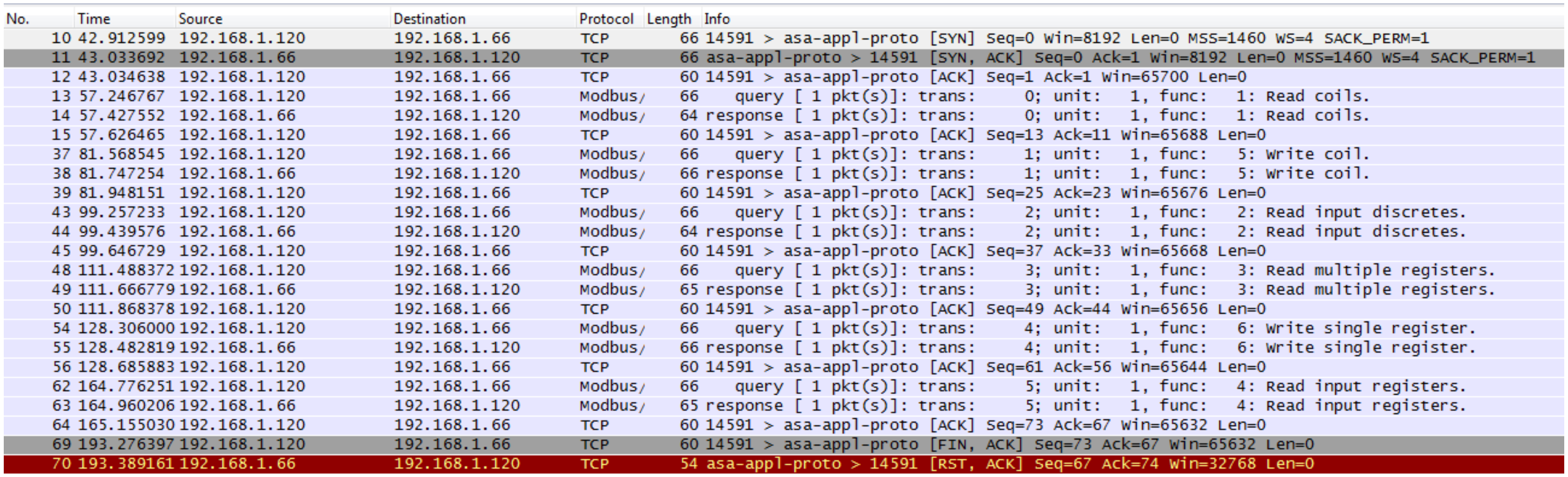
Appendix C: Emulated PLC Test Case Supporting Figures

C.1 Functionality Test Through Modbus Traffic Emulation



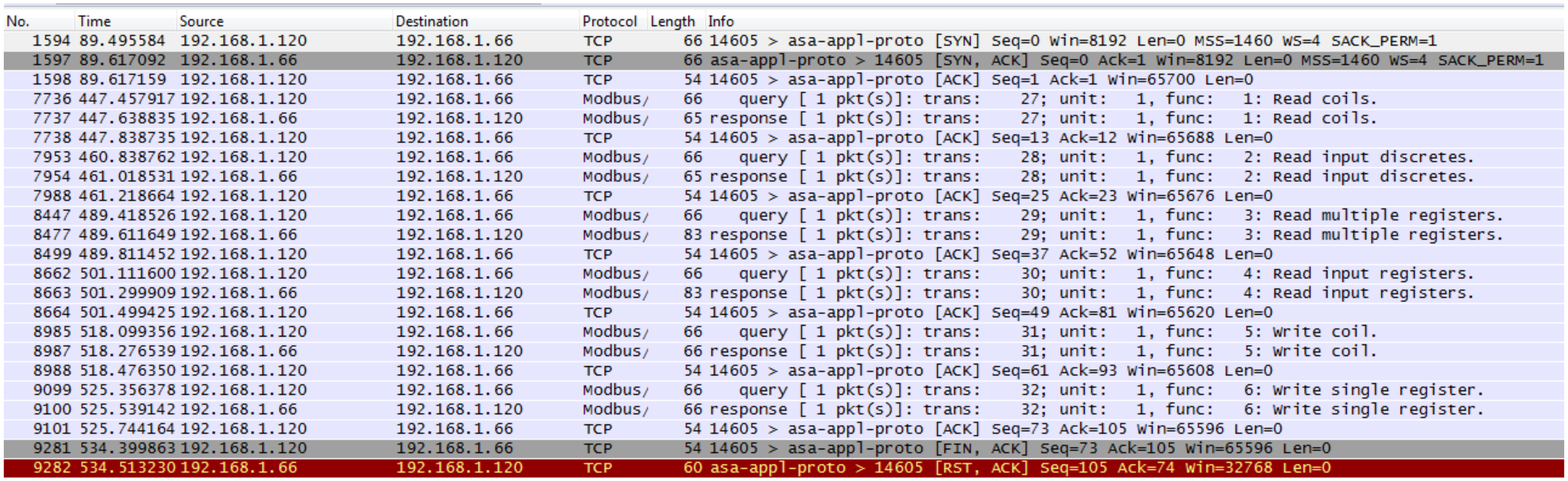
No.	Time	Source	Destination	Protocol	Length	Info
763	42.855761	192.168.1.120	192.168.1.66	TCP	66	14591 > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
764	42.977942	192.168.1.66	192.168.1.120	TCP	66	asa-appl-proto > 14591 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
765	42.978014	192.168.1.120	192.168.1.66	TCP	54	14591 > asa-appl-proto [ACK] Seq=1 Ack=1 Win=65700 Len=0
982	57.189746	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 0; unit: 1, func: 1: Read coils.
983	57.371412	192.168.1.66	192.168.1.120	Modbus/	64	response [1 pkt(s)]: trans: 0; unit: 1, func: 1: Read coils.
984	57.569511	192.168.1.120	192.168.1.66	TCP	54	14591 > asa-appl-proto [ACK] Seq=13 Ack=11 Win=65688 Len=0
1452	81.510635	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 1; unit: 1, func: 5: Write coil.
1453	81.690275	192.168.1.66	192.168.1.120	Modbus/	66	response [1 pkt(s)]: trans: 1; unit: 1, func: 5: Write coil.
1454	81.890293	192.168.1.120	192.168.1.66	TCP	54	14591 > asa-appl-proto [ACK] Seq=25 Ack=23 Win=65676 Len=0
1727	99.198587	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 2; unit: 1, func: 2: Read input discretes.
1773	99.381918	192.168.1.66	192.168.1.120	Modbus/	64	response [1 pkt(s)]: trans: 2; unit: 1, func: 2: Read input discretes.
1780	99.588187	192.168.1.120	192.168.1.66	TCP	54	14591 > asa-appl-proto [ACK] Seq=37 Ack=33 Win=65668 Len=0
1946	111.429395	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 3; unit: 1, func: 3: Read multiple registers.
1953	111.608616	192.168.1.66	192.168.1.120	Modbus/	65	response [1 pkt(s)]: trans: 3; unit: 1, func: 3: Read multiple registers.
1954	111.809242	192.168.1.120	192.168.1.66	TCP	54	14591 > asa-appl-proto [ACK] Seq=49 Ack=44 Win=65656 Len=0
2272	128.246315	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 4; unit: 1, func: 6: Write single register.
2273	128.423998	192.168.1.66	192.168.1.120	Modbus/	66	response [1 pkt(s)]: trans: 4; unit: 1, func: 6: Write single register.
2276	128.626194	192.168.1.120	192.168.1.66	TCP	54	14591 > asa-appl-proto [ACK] Seq=61 Ack=56 Win=65644 Len=0
2870	164.715206	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 5; unit: 1, func: 4: Read input registers.
2871	164.900113	192.168.1.66	192.168.1.120	Modbus/	65	response [1 pkt(s)]: trans: 5; unit: 1, func: 4: Read input registers.
2872	165.093980	192.168.1.120	192.168.1.66	TCP	54	14591 > asa-appl-proto [ACK] Seq=73 Ack=67 Win=65632 Len=0
3361	193.214264	192.168.1.120	192.168.1.66	TCP	54	14591 > asa-appl-proto [FIN, ACK] Seq=73 Ack=67 Win=65632 Len=0
3362	193.327982	192.168.1.66	192.168.1.120	TCP	60	asa-appl-proto > 14591 [RST, ACK] Seq=67 Ack=74 Win=32768 Len=0

Figure C.1: Traffic captured on HMI running Triangle MicroWorks



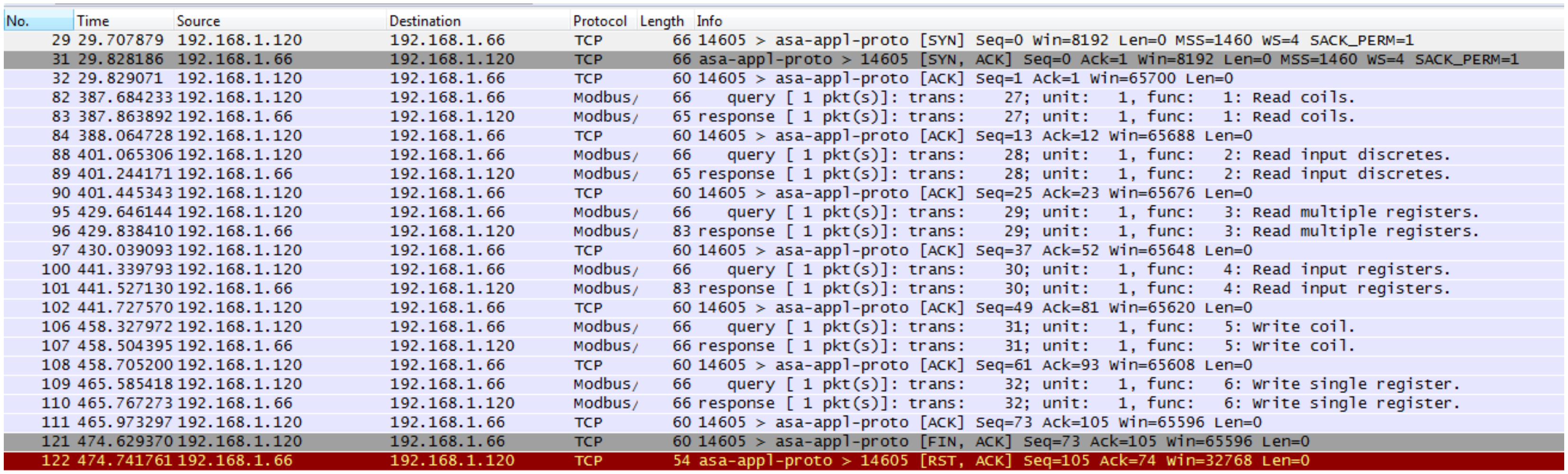
No.	Time	Source	Destination	Protocol	Length	Info
10	42.912599	192.168.1.120	192.168.1.66	TCP	66	14591 > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
11	43.033692	192.168.1.66	192.168.1.120	TCP	66	asa-appl-proto > 14591 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
12	43.034638	192.168.1.120	192.168.1.66	TCP	60	14591 > asa-appl-proto [ACK] Seq=1 Ack=1 Win=65700 Len=0
13	57.246767	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 0; unit: 1, func: 1: Read coils.
14	57.427552	192.168.1.66	192.168.1.120	Modbus/	64	response [1 pkt(s)]: trans: 0; unit: 1, func: 1: Read coils.
15	57.626465	192.168.1.120	192.168.1.66	TCP	60	14591 > asa-appl-proto [ACK] Seq=13 Ack=11 Win=65688 Len=0
37	81.568545	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 1; unit: 1, func: 5: Write coil.
38	81.747254	192.168.1.66	192.168.1.120	Modbus/	66	response [1 pkt(s)]: trans: 1; unit: 1, func: 5: Write coil.
39	81.948151	192.168.1.120	192.168.1.66	TCP	60	14591 > asa-appl-proto [ACK] Seq=25 Ack=23 Win=65676 Len=0
43	99.257233	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 2; unit: 1, func: 2: Read input discretes.
44	99.439576	192.168.1.66	192.168.1.120	Modbus/	64	response [1 pkt(s)]: trans: 2; unit: 1, func: 2: Read input discretes.
45	99.646729	192.168.1.120	192.168.1.66	TCP	60	14591 > asa-appl-proto [ACK] Seq=37 Ack=33 Win=65668 Len=0
48	111.488372	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 3; unit: 1, func: 3: Read multiple registers.
49	111.666779	192.168.1.66	192.168.1.120	Modbus/	65	response [1 pkt(s)]: trans: 3; unit: 1, func: 3: Read multiple registers.
50	111.868378	192.168.1.120	192.168.1.66	TCP	60	14591 > asa-appl-proto [ACK] Seq=49 Ack=44 Win=65656 Len=0
54	128.306000	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 4; unit: 1, func: 6: Write single register.
55	128.482819	192.168.1.66	192.168.1.120	Modbus/	66	response [1 pkt(s)]: trans: 4; unit: 1, func: 6: Write single register.
56	128.685883	192.168.1.120	192.168.1.66	TCP	60	14591 > asa-appl-proto [ACK] Seq=61 Ack=56 Win=65644 Len=0
62	164.776251	192.168.1.120	192.168.1.66	Modbus/	66	query [1 pkt(s)]: trans: 5; unit: 1, func: 4: Read input registers.
63	164.960206	192.168.1.66	192.168.1.120	Modbus/	65	response [1 pkt(s)]: trans: 5; unit: 1, func: 4: Read input registers.
64	165.155030	192.168.1.120	192.168.1.66	TCP	60	14591 > asa-appl-proto [ACK] Seq=73 Ack=67 Win=65632 Len=0
69	193.276397	192.168.1.120	192.168.1.66	TCP	60	14591 > asa-appl-proto [FIN, ACK] Seq=73 Ack=67 Win=65632 Len=0
70	193.389161	192.168.1.66	192.168.1.120	TCP	54	asa-appl-proto > 14591 [RST, ACK] Seq=67 Ack=74 Win=32768 Len=0

Figure C.2: Traffic captured on emulated PLC running Triangle MicroWorks



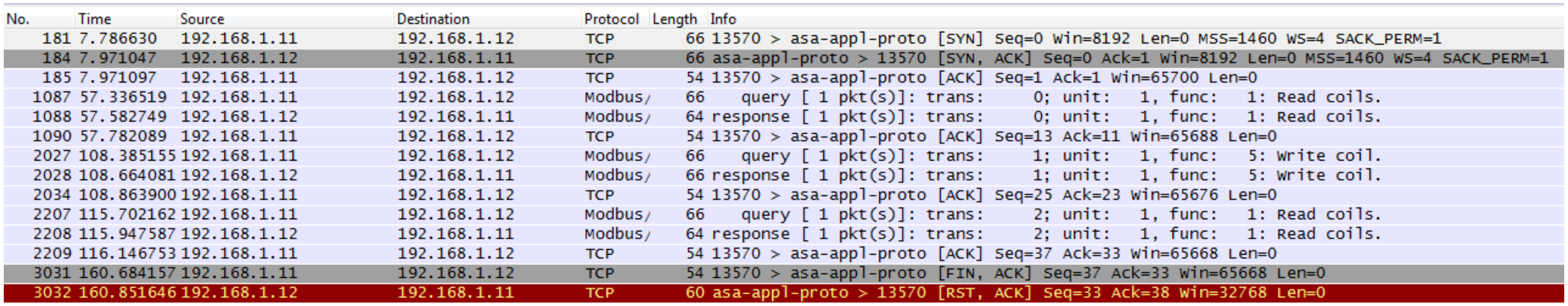
```
No.   Time       Source        Destination    Protocol Length Info
 1594 89.495584  192.168.1.120 192.168.1.66   TCP      66 14605 > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
 1597 89.617092  192.168.1.66  192.168.1.120  TCP      66 asa-appl-proto > 14605 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
 1598 89.617159  192.168.1.120 192.168.1.66   TCP      54 14605 > asa-appl-proto [ACK] Seq=1 Ack=1 Win=65700 Len=0
 7736 447.457917 192.168.1.120 192.168.1.66   Modbus/  66    query [ 1 pkt(s)]: trans:    27; unit:   1, func:   1: Read coils.
 7737 447.638835 192.168.1.66  192.168.1.120  Modbus/  65 response [ 1 pkt(s)]: trans:    27; unit:   1, func:   1: Read coils.
 7738 447.838735 192.168.1.120 192.168.1.66   TCP      54 14605 > asa-appl-proto [ACK] Seq=13 Ack=12 Win=65688 Len=0
 7953 460.838762 192.168.1.120 192.168.1.66   Modbus/  66    query [ 1 pkt(s)]: trans:    28; unit:   1, func:   2: Read input discretes.
 7954 461.018531 192.168.1.66  192.168.1.120  Modbus/  65 response [ 1 pkt(s)]: trans:    28; unit:   1, func:   2: Read input discretes.
 7988 461.218664 192.168.1.120 192.168.1.66   TCP      54 14605 > asa-appl-proto [ACK] Seq=25 Ack=23 Win=65676 Len=0
 8447 489.418526 192.168.1.120 192.168.1.66   Modbus/  66    query [ 1 pkt(s)]: trans:    29; unit:   1, func:   3: Read multiple registers.
 8477 489.611649 192.168.1.66  192.168.1.120  Modbus/  83 response [ 1 pkt(s)]: trans:    29; unit:   1, func:   3: Read multiple registers.
 8499 489.811452 192.168.1.120 192.168.1.66   TCP      54 14605 > asa-appl-proto [ACK] Seq=37 Ack=52 Win=65648 Len=0
 8662 501.111600 192.168.1.120 192.168.1.66   Modbus/  66    query [ 1 pkt(s)]: trans:    30; unit:   1, func:   4: Read input registers.
 8663 501.299909 192.168.1.66  192.168.1.120  Modbus/  83 response [ 1 pkt(s)]: trans:    30; unit:   1, func:   4: Read input registers.
 8664 501.499425 192.168.1.120 192.168.1.66   TCP      54 14605 > asa-appl-proto [ACK] Seq=49 Ack=81 Win=65620 Len=0
 8985 518.099356 192.168.1.120 192.168.1.66   Modbus/  66    query [ 1 pkt(s)]: trans:    31; unit:   1, func:   5: Write coil.
 8987 518.276539 192.168.1.66  192.168.1.120  Modbus/  66 response [ 1 pkt(s)]: trans:    31; unit:   1, func:   5: Write coil.
 8988 518.476350 192.168.1.120 192.168.1.66   TCP      54 14605 > asa-appl-proto [ACK] Seq=61 Ack=93 Win=65608 Len=0
 9099 525.356378 192.168.1.120 192.168.1.66   Modbus/  66    query [ 1 pkt(s)]: trans:    32; unit:   1, func:   6: Write single register.
 9100 525.539142 192.168.1.66  192.168.1.120  Modbus/  66 response [ 1 pkt(s)]: trans:    32; unit:   1, func:   6: Write single register.
 9101 525.744164 192.168.1.120 192.168.1.66   TCP      54 14605 > asa-appl-proto [ACK] Seq=73 Ack=105 Win=65596 Len=0
 9281 534.399863 192.168.1.120 192.168.1.66   TCP      54 14605 > asa-appl-proto [FIN, ACK] Seq=73 Ack=105 Win=65596 Len=0
 9282 534.513230 192.168.1.66  192.168.1.120  TCP      60 asa-appl-proto > 14605 [RST, ACK] Seq=105 Ack=74 Win=32768 Len=0
```

Figure C.3: Traffic captured on HMI running Modbus Poll



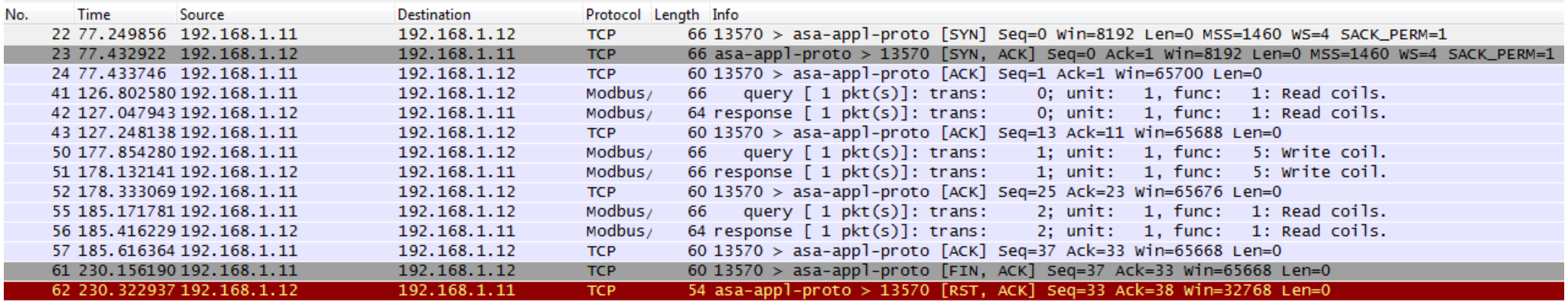
```
No.  Time        Source         Destination    Protocol Length Info
 29 29.707879  192.168.1.120  192.168.1.66   TCP       66 14605 > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
 31 29.828186  192.168.1.66   192.168.1.120  TCP       66 asa-appl-proto > 14605 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
 32 29.829071  192.168.1.120  192.168.1.66   TCP       60 14605 > asa-appl-proto [ACK] Seq=1 Ack=1 Win=65700 Len=0
 82 387.684233 192.168.1.120  192.168.1.66   Modbus/   66    query [ 1 pkt(s)]: trans:    27; unit:   1, func:   1: Read coils.
 83 387.863892 192.168.1.66   192.168.1.120  Modbus/   65 response [ 1 pkt(s)]: trans:    27; unit:   1, func:   1: Read coils.
 84 388.064728 192.168.1.120  192.168.1.66   TCP       60 14605 > asa-appl-proto [ACK] Seq=13 Ack=12 Win=65688 Len=0
 88 401.065306 192.168.1.120  192.168.1.66   Modbus/   66    query [ 1 pkt(s)]: trans:    28; unit:   1, func:   2: Read input discretes.
 89 401.244171 192.168.1.66   192.168.1.120  Modbus/   65 response [ 1 pkt(s)]: trans:    28; unit:   1, func:   2: Read input discretes.
 90 401.445343 192.168.1.120  192.168.1.66   TCP       60 14605 > asa-appl-proto [ACK] Seq=25 Ack=23 Win=65676 Len=0
 95 429.646144 192.168.1.120  192.168.1.66   Modbus/   66    query [ 1 pkt(s)]: trans:    29; unit:   1, func:   3: Read multiple registers.
 96 429.838410 192.168.1.66   192.168.1.120  Modbus/   83 response [ 1 pkt(s)]: trans:    29; unit:   1, func:   3: Read multiple registers.
 97 430.039093 192.168.1.120  192.168.1.66   TCP       60 14605 > asa-appl-proto [ACK] Seq=37 Ack=52 Win=65648 Len=0
100 441.339793 192.168.1.120  192.168.1.66   Modbus/   66    query [ 1 pkt(s)]: trans:    30; unit:   1, func:   4: Read input registers.
101 441.527130 192.168.1.66   192.168.1.120  Modbus/   83 response [ 1 pkt(s)]: trans:    30; unit:   1, func:   4: Read input registers.
102 441.727570 192.168.1.120  192.168.1.66   TCP       60 14605 > asa-appl-proto [ACK] Seq=49 Ack=81 Win=65620 Len=0
106 458.327972 192.168.1.120  192.168.1.66   Modbus/   66    query [ 1 pkt(s)]: trans:    31; unit:   1, func:   5: Write coil.
107 458.504395 192.168.1.66   192.168.1.120  Modbus/   66 response [ 1 pkt(s)]: trans:    31; unit:   1, func:   5: Write coil.
108 458.705200 192.168.1.120  192.168.1.66   TCP       60 14605 > asa-appl-proto [ACK] Seq=61 Ack=93 Win=65608 Len=0
109 465.585418 192.168.1.120  192.168.1.66   Modbus/   66    query [ 1 pkt(s)]: trans:    32; unit:   1, func:   6: Write single register.
110 465.767273 192.168.1.66   192.168.1.120  Modbus/   66 response [ 1 pkt(s)]: trans:    32; unit:   1, func:   6: Write single register.
111 465.973297 192.168.1.120  192.168.1.66   TCP       60 14605 > asa-appl-proto [ACK] Seq=73 Ack=105 Win=65596 Len=0
121 474.629370 192.168.1.120  192.168.1.66   TCP       60 14605 > asa-appl-proto [FIN, ACK] Seq=73 Ack=105 Win=65596 Len=0
122 474.741761 192.168.1.66   192.168.1.120  TCP       54 asa-appl-proto > 14605 [RST, ACK] Seq=105 Ack=74 Win=32768 Len=0
```

Figure C.4: Traffic captured on emulated PLC running Modbus Poll



No.	Time	Source	Destination	Protocol	Length	Info
181	7.786630	192.168.1.11	192.168.1.12	TCP	66	13570 > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
184	7.971047	192.168.1.12	192.168.1.11	TCP	66	asa-appl-proto > 13570 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
185	7.971097	192.168.1.11	192.168.1.12	TCP	54	13570 > asa-appl-proto [ACK] Seq=1 Ack=1 Win=65700 Len=0
1087	57.336519	192.168.1.11	192.168.1.12	Modbus/	66	query [1 pkt(s)]: trans: 0; unit: 1, func: 1: Read coils.
1088	57.582749	192.168.1.12	192.168.1.11	Modbus/	64	response [1 pkt(s)]: trans: 0; unit: 1, func: 1: Read coils.
1090	57.782089	192.168.1.11	192.168.1.12	TCP	54	13570 > asa-appl-proto [ACK] Seq=13 Ack=11 Win=65688 Len=0
2027	108.385155	192.168.1.11	192.168.1.12	Modbus/	66	query [1 pkt(s)]: trans: 1; unit: 1, func: 5: Write coil.
2028	108.664081	192.168.1.12	192.168.1.11	Modbus/	66	response [1 pkt(s)]: trans: 1; unit: 1, func: 5: Write coil.
2034	108.863900	192.168.1.11	192.168.1.12	TCP	54	13570 > asa-appl-proto [ACK] Seq=25 Ack=23 Win=65676 Len=0
2207	115.702162	192.168.1.11	192.168.1.12	Modbus/	66	query [1 pkt(s)]: trans: 2; unit: 1, func: 1: Read coils.
2208	115.947587	192.168.1.12	192.168.1.11	Modbus/	64	response [1 pkt(s)]: trans: 2; unit: 1, func: 1: Read coils.
2209	116.146753	192.168.1.11	192.168.1.12	TCP	54	13570 > asa-appl-proto [ACK] Seq=37 Ack=33 Win=65668 Len=0
3031	160.684157	192.168.1.11	192.168.1.12	TCP	54	13570 > asa-appl-proto [FIN, ACK] Seq=37 Ack=33 Win=65668 Len=0
3032	160.851646	192.168.1.12	192.168.1.11	TCP	60	asa-appl-proto > 13570 [RST, ACK] Seq=33 Ack=38 Win=32768 Len=0

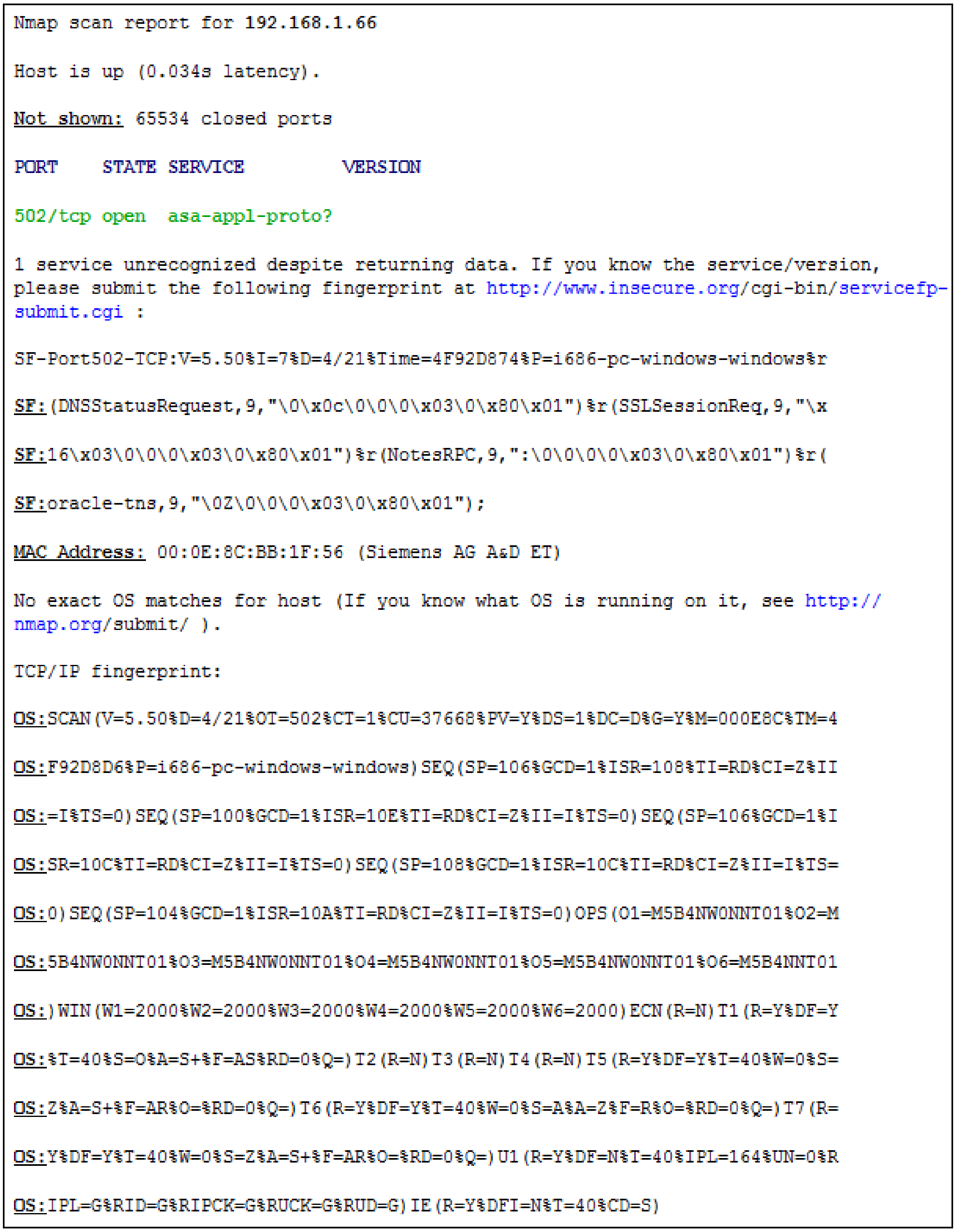
Figure C.5: Traffic captured on HMI during read, write, read test.



No.	Time	Source	Destination	Protocol	Length	Info
22	77.249856	192.168.1.11	192.168.1.12	TCP	66	13570 > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
23	77.432922	192.168.1.12	192.168.1.11	TCP	66	asa-appl-proto > 13570 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
24	77.433746	192.168.1.11	192.168.1.12	TCP	60	13570 > asa-appl-proto [ACK] Seq=1 Ack=1 Win=65700 Len=0
41	126.802580	192.168.1.11	192.168.1.12	Modbus/	66	query [1 pkt(s)]: trans: 0; unit: 1, func: 1: Read coils.
42	127.047943	192.168.1.12	192.168.1.11	Modbus/	64	response [1 pkt(s)]: trans: 0; unit: 1, func: 1: Read coils.
43	127.248138	192.168.1.11	192.168.1.12	TCP	60	13570 > asa-appl-proto [ACK] Seq=13 Ack=11 Win=65688 Len=0
50	177.854280	192.168.1.11	192.168.1.12	Modbus/	66	query [1 pkt(s)]: trans: 1; unit: 1, func: 5: Write coil.
51	178.132141	192.168.1.12	192.168.1.11	Modbus/	66	response [1 pkt(s)]: trans: 1; unit: 1, func: 5: Write coil.
52	178.333069	192.168.1.11	192.168.1.12	TCP	60	13570 > asa-appl-proto [ACK] Seq=25 Ack=23 Win=65676 Len=0
55	185.171781	192.168.1.11	192.168.1.12	Modbus/	66	query [1 pkt(s)]: trans: 2; unit: 1, func: 1: Read coils.
56	185.416229	192.168.1.12	192.168.1.11	Modbus/	64	response [1 pkt(s)]: trans: 2; unit: 1, func: 1: Read coils.
57	185.616364	192.168.1.11	192.168.1.12	TCP	60	13570 > asa-appl-proto [ACK] Seq=37 Ack=33 Win=65668 Len=0
61	230.156190	192.168.1.11	192.168.1.12	TCP	60	13570 > asa-appl-proto [FIN, ACK] Seq=37 Ack=33 Win=65668 Len=0
62	230.322937	192.168.1.12	192.168.1.11	TCP	54	asa-appl-proto > 13570 [RST, ACK] Seq=33 Ack=38 Win=32768 Len=0

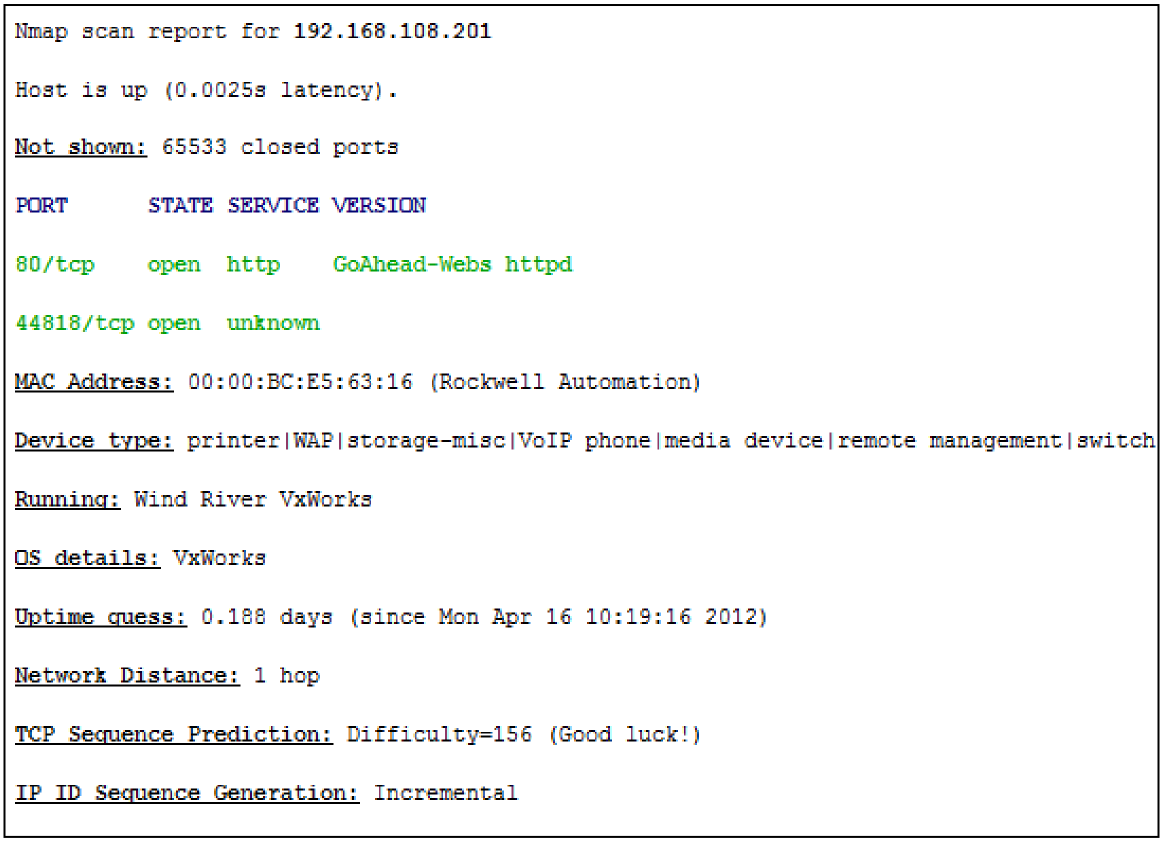
Figure C.6: Traffic captured on emulated PLC during read, write, read test.

C.2 Fingerprinting Port Scan Test Case



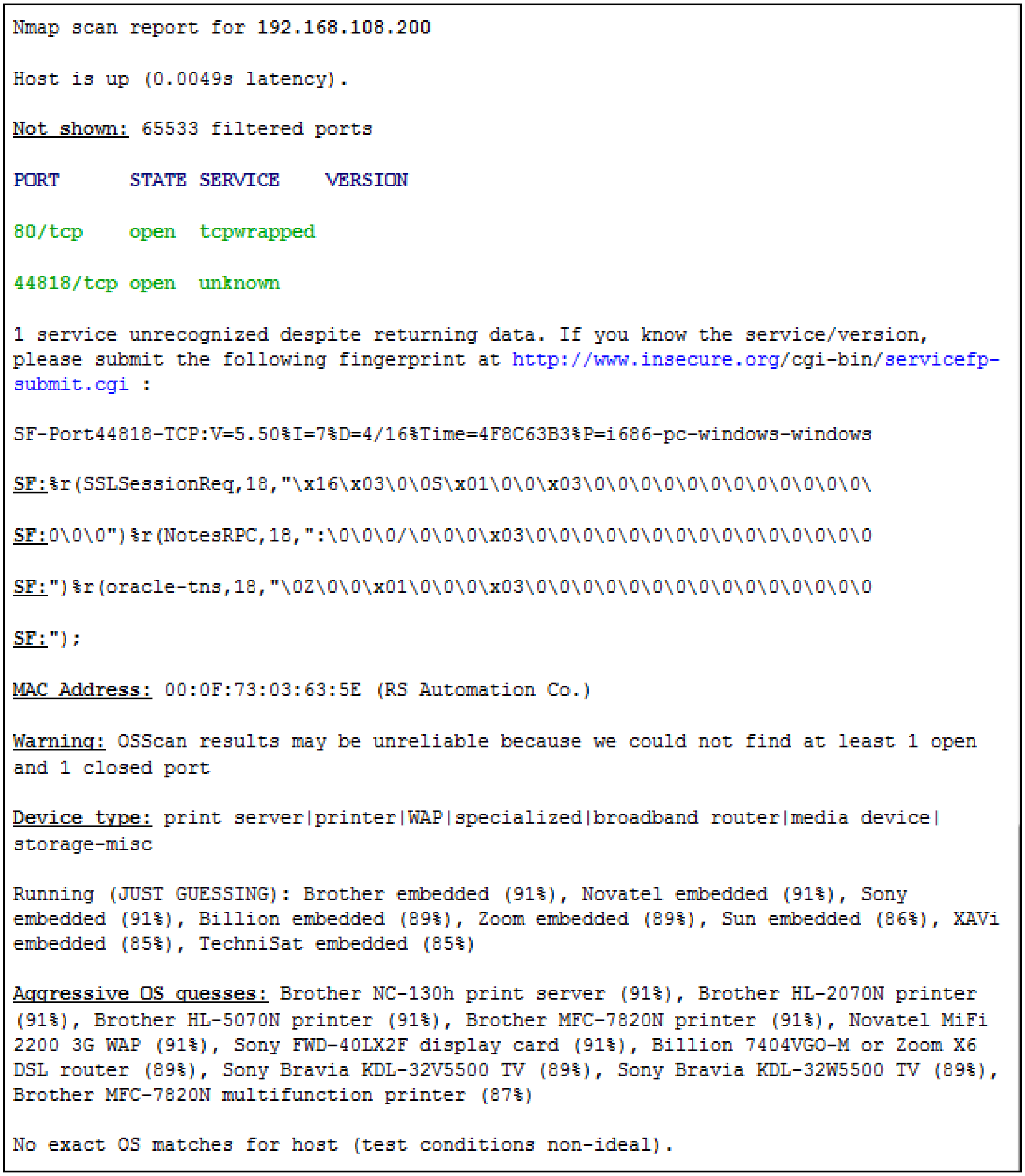
```
Nmap scan report for 192.168.1.66

Host is up (0.034s latency).

Not shown: 65534 closed ports

PORT     STATE SERVICE          VERSION

502/tcp open  asa-appl-proto?

1 service unrecognized despite returning data. If you know the service/version,
please submit the following fingerprint at http://www.insecure.org/cgi-bin/servicefp-
submit.cgi :

SF-Port502-TCP:V=5.50%I=7%D=4/21%Time=4F92D874%P=i686-pc-windows-windows%r

SF:(DNSStatusRequest,9,"\0\x0c\0\0\0\x03\0\x80\x01")%r(SSLSessionReq,9,"\x

SF:16\x03\0\0\0\x03\0\x80\x01")%r(NotesRPC,9,":\0\0\0\0\x03\0\x80\x01")%r(

SF:oracle-tns,9,"\0Z\0\0\0\x03\0\x80\x01");

MAC Address: 00:0E:8C:BB:1F:56 (Siemens AG A&D ET)

No exact OS matches for host (If you know what OS is running on it, see http://
nmap.org/submit/ ).

TCP/IP fingerprint:

OS:SCAN(V=5.50%D=4/21%OT=502%CT=1%CU=37668%PV=Y%DS=1%DC=D%G=Y%M=000E8C%TM=4

OS:F92D8D6%P=i686-pc-windows-windows)SEQ(SP=106%GCD=1%ISR=108%TI=RD%CI=Z%II

OS:=I%TS=0)SEQ(SP=100%GCD=1%ISR=10E%TI=RD%CI=Z%II=I%TS=0)SEQ(SP=106%GCD=1%I

OS:SR=10C%TI=RD%CI=Z%II=I%TS=0)SEQ(SP=108%GCD=1%ISR=10C%TI=RD%CI=Z%II=I%TS=

OS:0)SEQ(SP=104%GCD=1%ISR=10A%TI=RD%CI=Z%II=I%TS=0)OPS(O1=M5B4NW0NNT01%O2=M

OS:5B4NW0NNT01%O3=M5B4NW0NNT01%O4=M5B4NW0NNT01%O5=M5B4NW0NNT01%O6=M5B4NNT01

OS:)WIN(W1=2000%W2=2000%W3=2000%W4=2000%W5=2000%W6=2000)ECN(R=N)T1(R=Y%DF=Y

OS:%T=40%S=O%A=S+%F=AS%RD=0%Q=)T2(R=N)T3(R=N)T4(R=N)T5(R=Y%DF=Y%T=40%W=0%S=

OS:Z%A=S+%F=AR%O=%RD=0%Q=)T6(R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)T7(R=

OS:Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)U1(R=Y%DF=N%T=40%IPL=164%UN=0%R

OS:IPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)IE(R=Y%DFI=N%T=40%CD=S)
```

Figure C.7: Nmap Intense Scan All TCP Ports, Emulated PLC



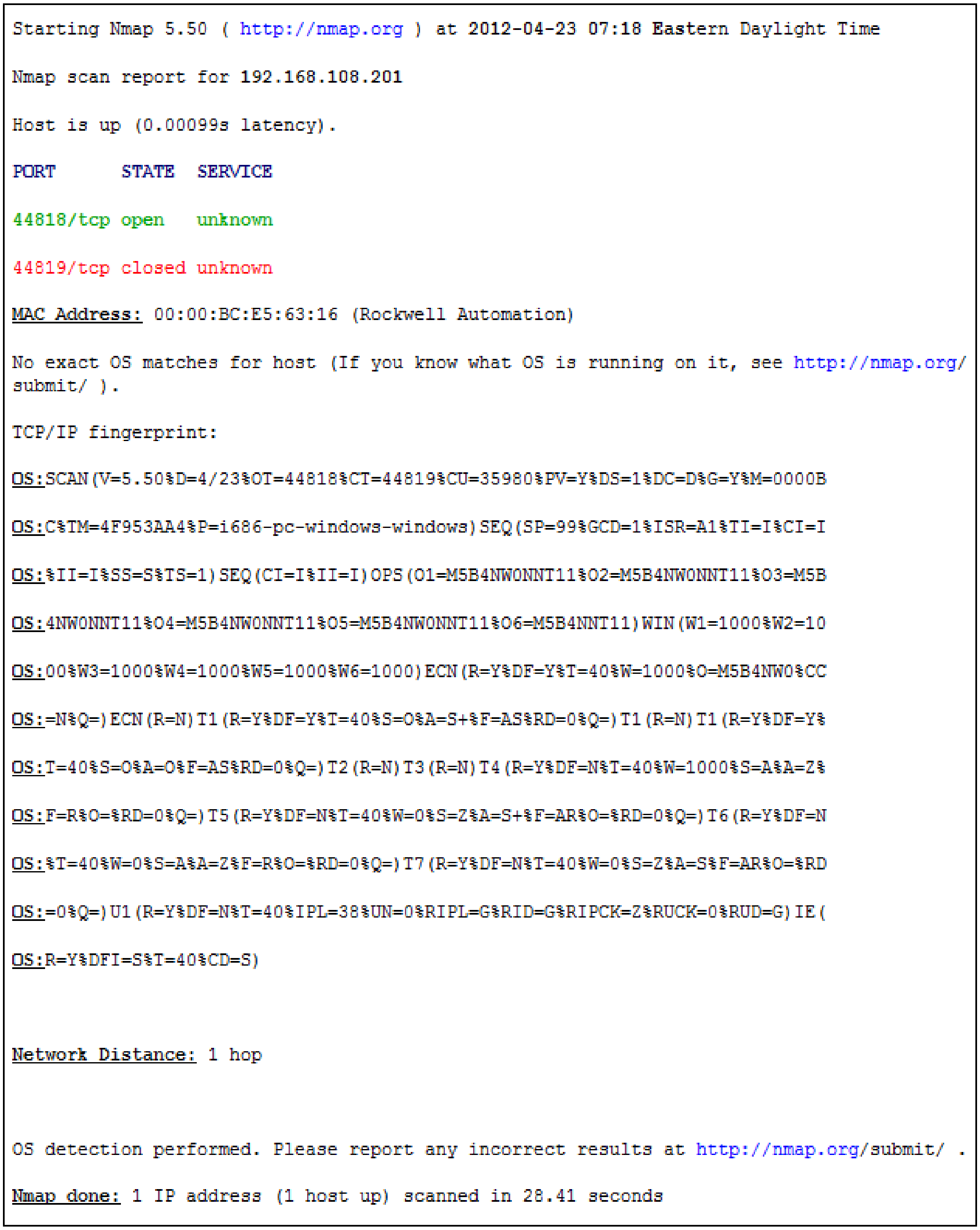
```
Nmap scan report for 192.168.108.201

Host is up (0.0025s latency).

Not shown: 65533 closed ports

PORT      STATE SERVICE VERSION

80/tcp    open  http    GoAhead-Webs httpd

44818/tcp open  unknown

MAC Address: 00:00:BC:E5:63:16 (Rockwell Automation)

Device type: printer|WAP|storage-misc|VoIP phone|media device|remote management|switch

Running: Wind River VxWorks

OS details: VxWorks

Uptime guess: 0.188 days (since Mon Apr 16 10:19:16 2012)

Network Distance: 1 hop

TCP Sequence Prediction: Difficulty=156 (Good luck!)

IP ID Sequence Generation: Incremental
```

Figure C.8: Nmap Intense Scan All TCP Ports, CompactLogix 1769



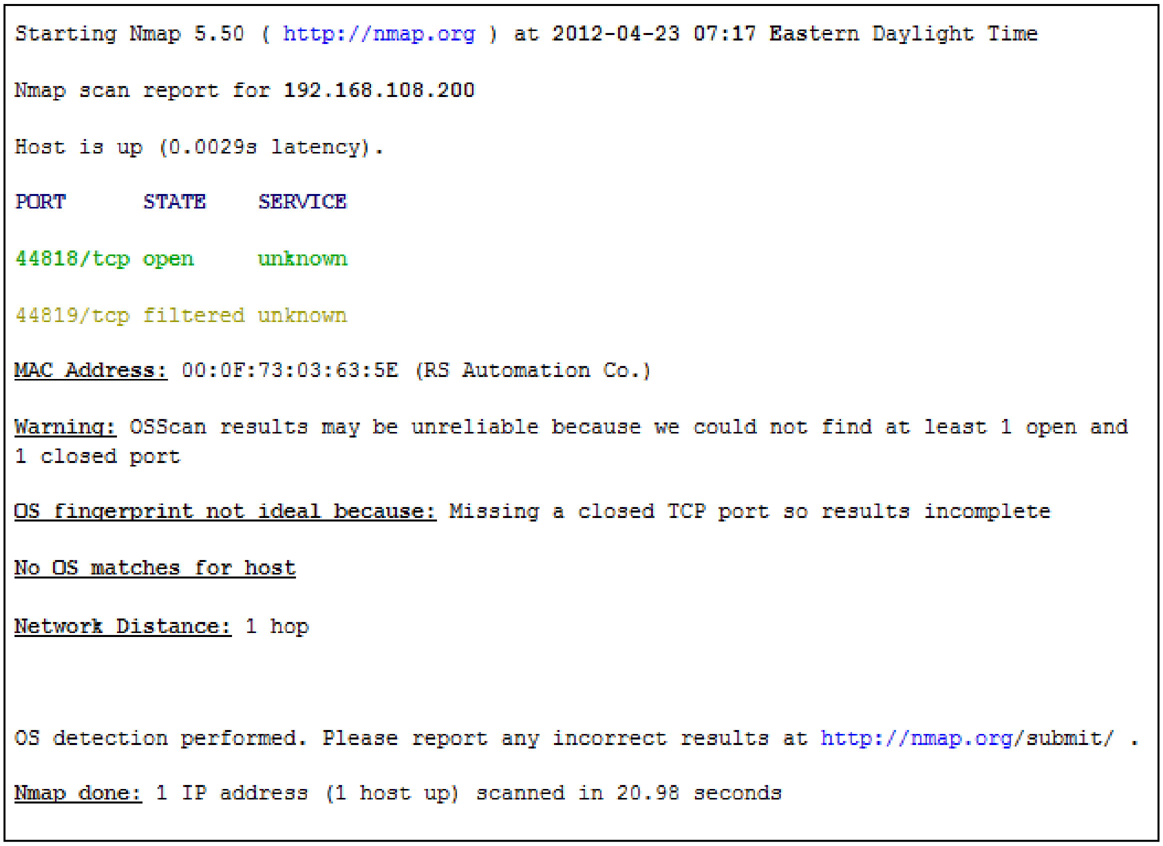
```
Nmap scan report for 192.168.108.200

Host is up (0.0049s latency).

Not shown: 65533 filtered ports

PORT      STATE SERVICE    VERSION

80/tcp    open  tcpwrapped

44818/tcp open  unknown

1 service unrecognized despite returning data. If you know the service/version,
please submit the following fingerprint at http://www.insecure.org/cgi-bin/servicefp-
submit.cgi :

SF-Port44818-TCP:V=5.50%I=7%D=4/16%Time=4F8C63B3%P=i686-pc-windows-windows

SF:%r(SSLSessionReq,18,"\x16\x03\0\0S\x01\0\0\x03\0\0\0\0\0\0\0\0\0\0\0\

SF:0\0\0")%r(NotesRPC,18,":\0\0\0/\0\0\0\x03\0\0\0\0\0\0\0\0\0\0\0\0\0\0

SF:")%r(oracle-tns,18,"\0Z\0\0\x01\0\0\0\x03\0\0\0\0\0\0\0\0\0\0\0\0\0\0

SF:");

MAC Address: 00:0F:73:03:63:5E (RS Automation Co.)

Warning: OSScan results may be unreliable because we could not find at least 1 open
and 1 closed port

Device type: print server|printer|WAP|specialized|broadband router|media device|
storage-misc

Running (JUST GUESSING): Brother embedded (91%), Novatel embedded (91%), Sony
embedded (91%), Billion embedded (89%), Zoom embedded (89%), Sun embedded (86%), XAVi
embedded (85%), TechniSat embedded (85%)

Aggressive OS guesses: Brother NC-130h print server (91%), Brother HL-2070N printer
(91%), Brother HL-5070N printer (91%), Brother MFC-7820N printer (91%), Novatel MiFi
2200 3G WAP (91%), Sony FWD-40LX2F display card (91%), Billion 7404VGO-M or Zoom X6
DSL router (89%), Sony Bravia KDL-32V5500 TV (89%), Sony Bravia KDL-32W5500 TV (89%),
Brother MFC-7820N multifunction printer (87%)

No exact OS matches for host (test conditions non-ideal).
```

Figure C.9: Nmap Intense Scan All TCP Ports, MicroLogix 1100



```
Starting Nmap 5.50 ( http://nmap.org ) at 2012-04-23 07:18 Eastern Daylight Time

Nmap scan report for 192.168.108.201

Host is up (0.00099s latency).

PORT      STATE  SERVICE

44818/tcp open   unknown

44819/tcp closed unknown

MAC Address: 00:00:BC:E5:63:16 (Rockwell Automation)

No exact OS matches for host (If you know what OS is running on it, see http://nmap.org/
submit/ ).

TCP/IP fingerprint:

OS:SCAN(V=5.50%D=4/23%OT=44818%CT=44819%CU=35980%PV=Y%DS=1%DC=D%G=Y%M=0000B

OS:C%TM=4F953AA4%P=i686-pc-windows-windows)SEQ(SP=99%GCD=1%ISR=A1%TI=I%CI=I

OS:%II=I%SS=S%TS=1)SEQ(CI=I%II=I)OPS(O1=M5B4NW0NNT11%O2=M5B4NW0NNT11%O3=M5B

OS:4NW0NNT11%O4=M5B4NW0NNT11%O5=M5B4NW0NNT11%O6=M5B4NNT11)WIN(W1=1000%W2=10

OS:00%W3=1000%W4=1000%W5=1000%W6=1000)ECN(R=Y%DF=Y%T=40%W=1000%O=M5B4NW0%CC

OS:=N%Q=)ECN(R=N)T1(R=Y%DF=Y%T=40%S=O%A=S+%F=AS%RD=0%Q=)T1(R=N)T1(R=Y%DF=Y%

OS:T=40%S=O%A=O%F=AS%RD=0%Q=)T2(R=N)T3(R=N)T4(R=Y%DF=N%T=40%W=1000%S=A%A=Z%

OS:F=R%O=%RD=0%Q=)T5(R=Y%DF=N%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)T6(R=Y%DF=N

OS:%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)T7(R=Y%DF=N%T=40%W=0%S=Z%A=S%F=AR%O=%RD

OS:=0%Q=)U1(R=Y%DF=N%T=40%IPL=38%UN=0%RIPL=G%RID=G%RIPCK=Z%RUCK=0%RUD=G)IE(

OS:R=Y%DFI=S%T=40%CD=S)


Network Distance: 1 hop


OS detection performed. Please report any incorrect results at http://nmap.org/submit/ .

Nmap done: 1 IP address (1 host up) scanned in 28.41 seconds
```

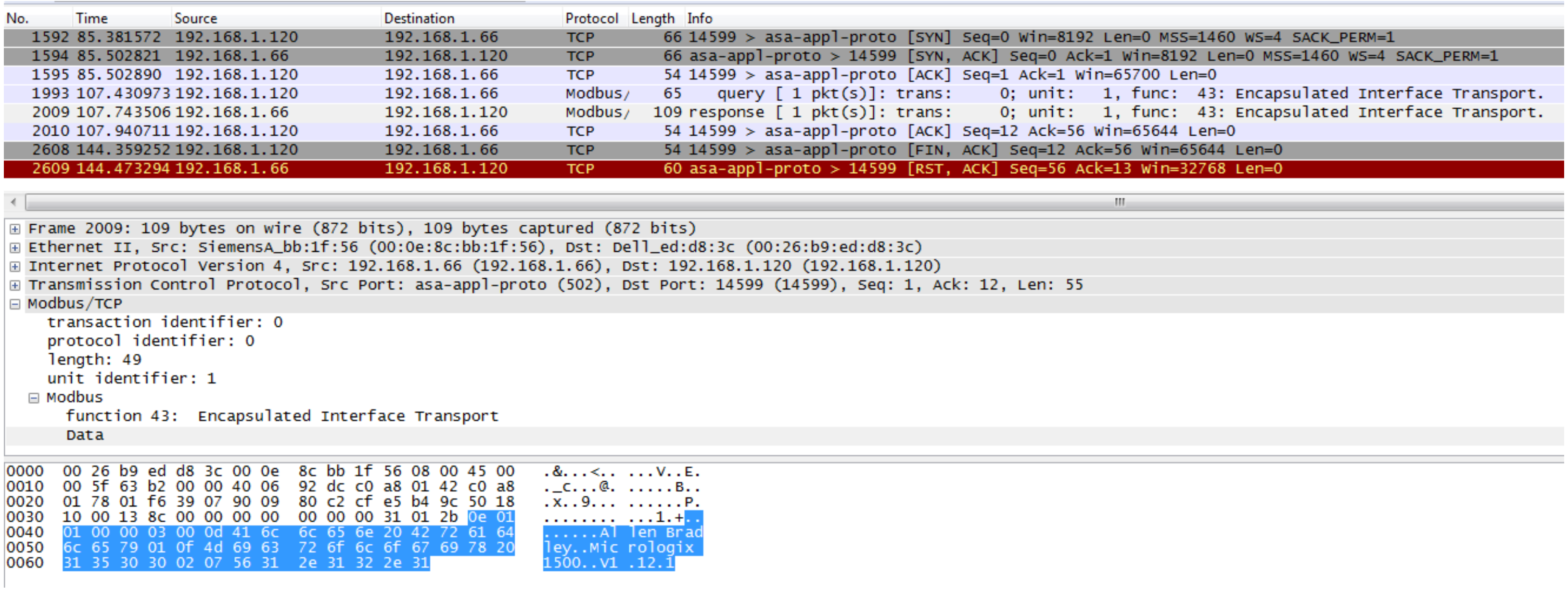
Figure C.10: Nmap Operating System Scan on Ethernet/IP port ControlLogix 1769



```
Starting Nmap 5.50 ( http://nmap.org ) at 2012-04-23 07:17 Eastern Daylight Time

Nmap scan report for 192.168.108.200

Host is up (0.0029s latency).

PORT      STATE    SERVICE

44818/tcp open     unknown

44819/tcp filtered unknown

MAC Address: 00:0F:73:03:63:5E (RS Automation Co.)

Warning: OSScan results may be unreliable because we could not find at least 1 open and
1 closed port

OS fingerprint not ideal because: Missing a closed TCP port so results incomplete

No OS matches for host

Network Distance: 1 hop



OS detection performed. Please report any incorrect results at http://nmap.org/submit/ .

Nmap done: 1 IP address (1 host up) scanned in 20.98 seconds
```

Figure C.11: Nmap Operating System Scan on Ethernet/IP port MicroLogix 1100

C.3 Fingerprinting Banner Grab Test Case

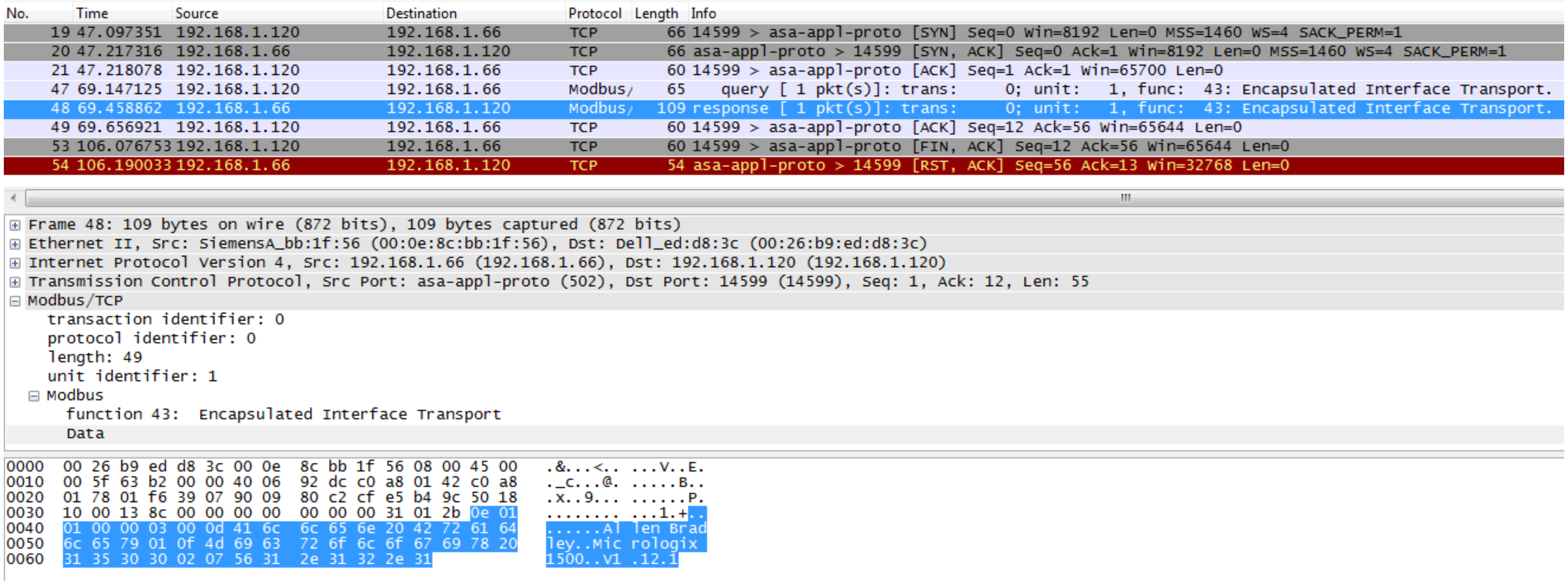


```
No.  Time       Source         Destination    Protocol Length Info
1592 85.381572  192.168.1.120  192.168.1.66   TCP       66 14599 > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
1594 85.502821  192.168.1.66   192.168.1.120  TCP       66 asa-appl-proto > 14599 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
1595 85.502890  192.168.1.120  192.168.1.66   TCP       54 14599 > asa-appl-proto [ACK] Seq=1 Ack=1 Win=65700 Len=0
1993 107.430973 192.168.1.120  192.168.1.66   Modbus/   65    query [ 1 pkt(s)]: trans:     0; unit:   1, func:  43: Encapsulated Interface Transport.
2009 107.743506 192.168.1.66   192.168.1.120  Modbus/  109 response [ 1 pkt(s)]: trans:     0; unit:   1, func:  43: Encapsulated Interface Transport.
2010 107.940711 192.168.1.120  192.168.1.66   TCP       54 14599 > asa-appl-proto [ACK] Seq=12 Ack=56 Win=65644 Len=0
2608 144.359252 192.168.1.120  192.168.1.66   TCP       54 14599 > asa-appl-proto [FIN, ACK] Seq=12 Ack=56 Win=65644 Len=0
2609 144.473294 192.168.1.66   192.168.1.120  TCP       60 asa-appl-proto > 14599 [RST, ACK] Seq=56 Ack=13 Win=32768 Len=0
```

```
⊞ Frame 2009: 109 bytes on wire (872 bits), 109 bytes captured (872 bits)
⊞ Ethernet II, Src: SiemensA_bb:1f:56 (00:0e:8c:bb:1f:56), Dst: Dell_ed:d8:3c (00:26:b9:ed:d8:3c)
⊞ Internet Protocol Version 4, Src: 192.168.1.66 (192.168.1.66), Dst: 192.168.1.120 (192.168.1.120)
⊞ Transmission Control Protocol, Src Port: asa-appl-proto (502), Dst Port: 14599 (14599), Seq: 1, Ack: 12, Len: 55
⊟ Modbus/TCP
    transaction identifier: 0
    protocol identifier: 0
    length: 49
    unit identifier: 1
  ⊟ Modbus
      function 43:  Encapsulated Interface Transport
      Data
```

```
0000  00 26 b9 ed d8 3c 00 0e  8c bb 1f 56 08 00 45 00   .&...<.. ...V..E.
0010  00 5f 63 b2 00 00 40 06  92 dc c0 a8 01 42 c0 a8   ._c...@. .....B..
0020  01 78 01 f6 39 07 90 09  80 c2 cf e5 b4 9c 50 18   .x..9... ......P.
0030  10 00 13 8c 00 00 00 00  00 00 00 31 01 2b 0e 01   ........ ...1.+..
0040  01 00 00 03 00 0d 41 6c  6c 65 6e 20 42 72 61 64   ......Al len Brad
0050  6c 65 79 01 0f 4d 69 63  72 6f 6c 6f 67 69 78 20   ley..Mic rologix 
0060  31 35 30 30 02 07 56 31  2e 31 32 2e 31            1500..V1 .12.1
```

Figure C.12: Traffic captured on HMI from banner grabbing test.



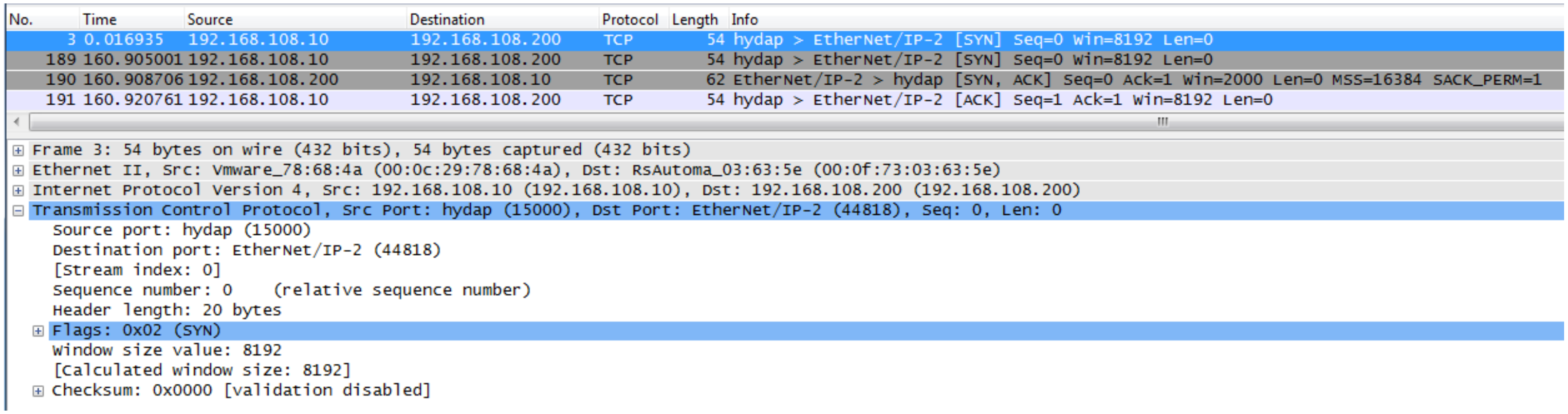
```
No.  Time        Source         Destination    Protocol Length Info
  19 47.097351   192.168.1.120  192.168.1.66   TCP       66 14599 > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
  20 47.217316   192.168.1.66   192.168.1.120  TCP       66 asa-appl-proto > 14599 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
  21 47.218078   192.168.1.120  192.168.1.66   TCP       60 14599 > asa-appl-proto [ACK] Seq=1 Ack=1 Win=65700 Len=0
  47 69.147125   192.168.1.120  192.168.1.66   Modbus/   65    query [ 1 pkt(s)]: trans:     0; unit:    1, func:   43: Encapsulated Interface Transport.
  48 69.458862   192.168.1.66   192.168.1.120  Modbus/  109 response [ 1 pkt(s)]: trans:     0; unit:    1, func:   43: Encapsulated Interface Transport.
  49 69.656921   192.168.1.120  192.168.1.66   TCP       60 14599 > asa-appl-proto [ACK] Seq=12 Ack=56 Win=65644 Len=0
  53 106.076753  192.168.1.120  192.168.1.66   TCP       60 14599 > asa-appl-proto [FIN, ACK] Seq=12 Ack=56 Win=65644 Len=0
  54 106.190033  192.168.1.66   192.168.1.120  TCP       54 asa-appl-proto > 14599 [RST, ACK] Seq=56 Ack=13 Win=32768 Len=0
```

```
⊞ Frame 48: 109 bytes on wire (872 bits), 109 bytes captured (872 bits)
⊞ Ethernet II, Src: SiemensA_bb:1f:56 (00:0e:8c:bb:1f:56), Dst: Dell_ed:d8:3c (00:26:b9:ed:d8:3c)
⊞ Internet Protocol Version 4, Src: 192.168.1.66 (192.168.1.66), Dst: 192.168.1.120 (192.168.1.120)
⊞ Transmission Control Protocol, Src Port: asa-appl-proto (502), Dst Port: 14599 (14599), Seq: 1, Ack: 12, Len: 55
⊟ Modbus/TCP
    transaction identifier: 0
    protocol identifier: 0
    length: 49
    unit identifier: 1
  ⊟ Modbus
      function 43:  Encapsulated Interface Transport
      Data
```

```
0000   00 26 b9 ed d8 3c 00 0e  8c bb 1f 56 08 00 45 00   .&...<.. ...V..E.
0010   00 5f 63 b2 00 00 40 06  92 dc c0 a8 01 42 c0 a8   ._c...@. .....B..
0020   01 78 01 f6 39 07 90 09  80 c2 cf e5 b4 9c 50 18   .x..9... ......P.
0030   10 00 13 8c 00 00 00 00  00 00 00 31 01 2b 0e 01   ........ ...1.+..
0040   01 00 00 03 00 0d 41 6c  6c 65 6e 20 42 72 61 64   ......Al len Brad
0050   6c 65 79 01 0f 4d 69 63  72 6f 6c 6f 67 69 78 20   ley..Mic rologix 
0060   31 35 30 30 02 07 56 31  2e 31 32 2e 31            1500..V1 .12.1
```

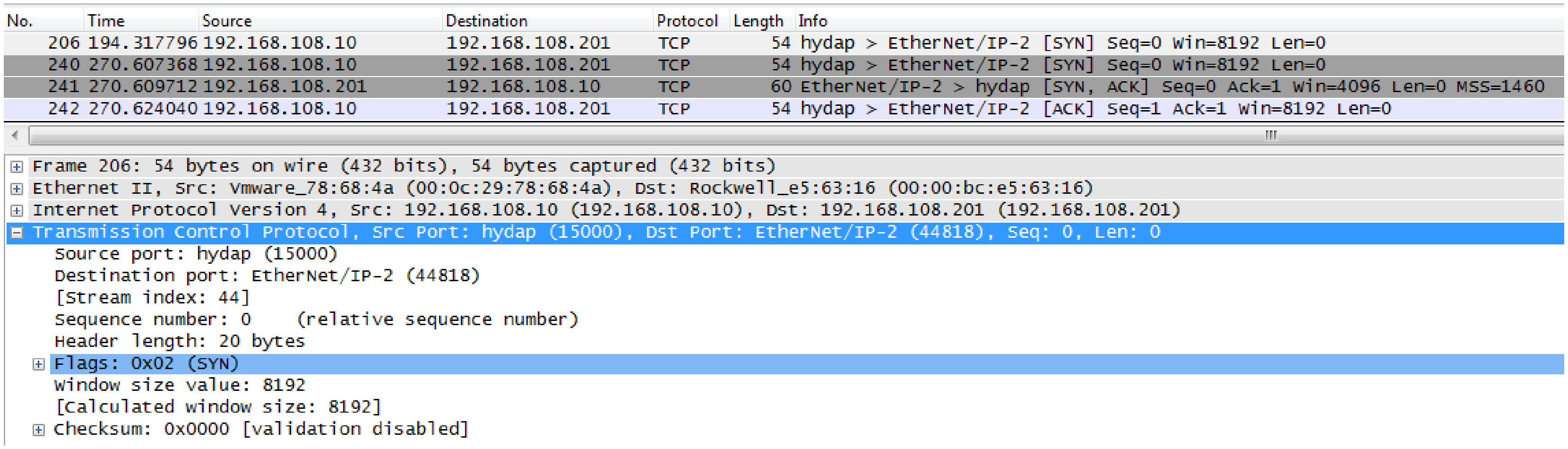
Figure C.13: Traffic captured on emulated PLC from banner grabbing test.

C.4 Invalid Traffic Test Case



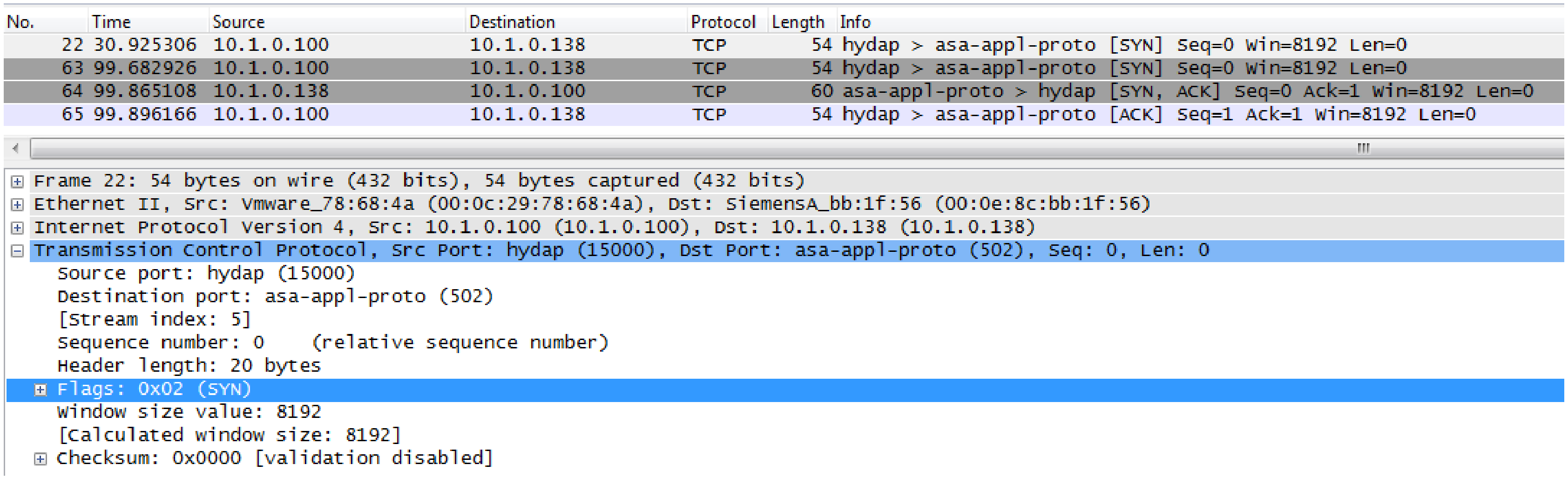
```
No.  Time         Source           Destination      Protocol Length Info
   3 0.016935     192.168.108.10   192.168.108.200  TCP      54 hydap > EtherNet/IP-2 [SYN] Seq=0 Win=8192 Len=0
 189 160.905001   192.168.108.10   192.168.108.200  TCP      54 hydap > EtherNet/IP-2 [SYN] Seq=0 Win=8192 Len=0
 190 160.908706   192.168.108.200  192.168.108.10   TCP      62 EtherNet/IP-2 > hydap [SYN, ACK] Seq=0 Ack=1 Win=2000 Len=0 MSS=16384 SACK_PERM=1
 191 160.920761   192.168.108.10   192.168.108.200  TCP      54 hydap > EtherNet/IP-2 [ACK] Seq=1 Ack=1 Win=8192 Len=0

⊞ Frame 3: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
⊞ Ethernet II, Src: Vmware_78:68:4a (00:0c:29:78:68:4a), Dst: RsAutoma_03:63:5e (00:0f:73:03:63:5e)
⊞ Internet Protocol Version 4, Src: 192.168.108.10 (192.168.108.10), Dst: 192.168.108.200 (192.168.108.200)
⊟ Transmission Control Protocol, Src Port: hydap (15000), Dst Port: EtherNet/IP-2 (44818), Seq: 0, Len: 0
    Source port: hydap (15000)
    Destination port: EtherNet/IP-2 (44818)
    [Stream index: 0]
    Sequence number: 0    (relative sequence number)
    Header length: 20 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size value: 8192
    [Calculated window size: 8192]
  ⊞ Checksum: 0x0000 [validation disabled]
```

Figure C.14: Network capture of invalid checksum sent to MicroLogix 1100



```
No.  Time       Source          Destination     Protocol Length Info
 206 194.317796 192.168.108.10  192.168.108.201 TCP          54 hydap > EtherNet/IP-2 [SYN] Seq=0 Win=8192 Len=0
 240 270.607368 192.168.108.10  192.168.108.201 TCP          54 hydap > EtherNet/IP-2 [SYN] Seq=0 Win=8192 Len=0
 241 270.609712 192.168.108.201 192.168.108.10  TCP          60 EtherNet/IP-2 > hydap [SYN, ACK] Seq=0 Ack=1 Win=4096 Len=0 MSS=1460
 242 270.624040 192.168.108.10  192.168.108.201 TCP          54 hydap > EtherNet/IP-2 [ACK] Seq=1 Ack=1 Win=8192 Len=0

⊞ Frame 206: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
⊞ Ethernet II, Src: Vmware_78:68:4a (00:0c:29:78:68:4a), Dst: Rockwell_e5:63:16 (00:00:bc:e5:63:16)
⊞ Internet Protocol Version 4, Src: 192.168.108.10 (192.168.108.10), Dst: 192.168.108.201 (192.168.108.201)
⊟ Transmission Control Protocol, Src Port: hydap (15000), Dst Port: EtherNet/IP-2 (44818), Seq: 0, Len: 0
    Source port: hydap (15000)
    Destination port: EtherNet/IP-2 (44818)
    [Stream index: 44]
    Sequence number: 0     (relative sequence number)
    Header length: 20 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size value: 8192
    [Calculated window size: 8192]
  ⊞ Checksum: 0x0000 [validation disabled]
```

Figure C.15: Network capture of invalid checksum sent to ControlLogix 1769



```
No.   Time       Source       Destination   Protocol Length Info
   22 30.925306  10.1.0.100   10.1.0.138    TCP        54 hydap > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0
   63 99.682926  10.1.0.100   10.1.0.138    TCP        54 hydap > asa-appl-proto [SYN] Seq=0 Win=8192 Len=0
   64 99.865108  10.1.0.138   10.1.0.100    TCP        60 asa-appl-proto > hydap [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0
   65 99.896166  10.1.0.100   10.1.0.138    TCP        54 hydap > asa-appl-proto [ACK] Seq=1 Ack=1 Win=8192 Len=0
```

```
⊞ Frame 22: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
⊞ Ethernet II, Src: Vmware_78:68:4a (00:0c:29:78:68:4a), Dst: SiemensA_bb:1f:56 (00:0e:8c:bb:1f:56)
⊞ Internet Protocol Version 4, Src: 10.1.0.100 (10.1.0.100), Dst: 10.1.0.138 (10.1.0.138)
⊟ Transmission Control Protocol, Src Port: hydap (15000), Dst Port: asa-appl-proto (502), Seq: 0, Len: 0
    Source port: hydap (15000)
    Destination port: asa-appl-proto (502)
    [Stream index: 5]
    Sequence number: 0    (relative sequence number)
    Header length: 20 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size value: 8192
    [Calculated window size: 8192]
  ⊞ Checksum: 0x0000 [validation disabled]
```

Figure C.16: Network capture of invalid checksum sent to emulated PLC

Appendix D: List of Acronyms

CIKR	Critical Infrastructure and Key Resources
CIP	Critical Infrastructure Protection
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DHS	Department of Homeland Security
DMZ	Demilitarized Zone
EIT	Encapsulated Interface Transport
FIFO	First-In-First-Out
GUI	Graphical User Interface
HMI	Human Machine Interface
HSPD	Homeland Security Presidential Directive
IANA	Internet Assigned Numbers Authority
ICS	Industrial Control System
IDS	Intrusion Detection System
IT	Information Technology
MBAP	Modbus Application Protocol
MEI	Modbus Encapsulated Interface Transport
MTU	Master Terminal Unit
OS	Operating System
PDU	Protocol Data Unit

PLC	Programmable Logic Controller
RFC	Request For Comment
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
SSA	Sector Specific Agencies

Bibliography

[1] Balitanas, Maricel, Rosslin John Robles, Ronnie Caytiles, Yvette Gelogo, Tai-hoon Kim. “Protecting IP-based SCADA System with Crossed-Cipher Scheme.” 2011 International Conference on Ubiquitous Computing and Multimedia Applications. pp. 123 – 126, 2011.

[2] Boyer, S., *SCADA: Supervisory Control and Data Acquisition*, The Instrumentation, Systems, and Automation Society, Research Triangle Park, North Carolina, 2004.

[3] Chavez, A., Cassidy, R. F., Trent, J. and Urrea, J. (2008) "Remote Forensic Analysis of Process Control Systems". *Critical Infrastructure Protection*, Vol 253, pp. 223-235.

[4] Department of Energy. *21 Steps to Improve Cyber Security of SCADA Networks*.

[5] Department of Homeland Security, *National Infrastructure Protection Plan*, 2009

[6] Department of Homeland Security. *Recommended Practice: Improving Industrial Control Systems Cybersecurity with Defense-In-Depth Strategies*. October 2009

[7] Digital Bond. (2011). “Quickdraw SCADA IDS”. http://www.digitalbond.com/tools/quickdraw/

[8] Digital Bond. (2011). “Scada honeynet”. http://www.digitalbond.com/tools/scada-honeynet/

[9] Digital Bond (2012). “Field Device Fingerprinting”. http://www.digitalbond.com/scadapedia/security-controls/field-device-fingerprinting/

[10] Falliere, Nicolas, Liam O Murchu, and Eric Chien. “W32.Stuxnet Dossier”. Symantec, version 1.4 edition, February 2011

[11] GAO. *Cybersecurity: Continued Attention Needed to Protect Our Nation’s Critical Infrastructure*. GAO-11-865T, 2011.

[12] Gumstix (www.gumstix.com), 2012.

[13] Homeland Security Presidential Directive 7 (HSPD 7) — Critical Infrastructure Identification, Prioritization, and Protection (HSPD-7) (Dec. 17, 2003)

[14] The Honeynet Project. (2003). “Know your enemy: Sebek”. http://old.honeynet.org/papers/sebek.pdf

[15] The Honeynet Project. (2004). *Know your enemy: Learning about security threats*. (2nd ed.). Addison-Wesley.

[16] The Honeynet Project. (2006). “Know your enemy: Honeynets”. http://old.honeynet.org/papers/honeynet/

[17] Idaho National Laboratory, Control systems cyber security: Defense in depth strategies, External Report INL/EXT-06-11478, Idaho Falls, Idaho, May 2006.

[18] Igure, Vinay, Sean Laughter, Ronald Williams, “Security Issues in SCADA networks”. Computers & Security, v 25, pp 498-506, num 7, 2006.

[19] Krutz, R. L. (2006). *Securing SCADA systems*. Indianapolis, IN: Wiley Pub.

[20] Langevin, R., R. McCaul, et al. (2008). *Securing Cyberspace for the 44th Presidency.*

[21] Leverett, E.P. “Quantitatively Assessing and Visualising Industrial System Attack Surfaces". 2011.

[22] Matrosov, A., E. Rodionov, D. Harley and J. Malcho, Stuxnet under the microscope revision 1.31, ESET North America, 2010.

[23] Metcalf, William, Victor Julien. “Snort_Inline” http://snort-inline.sourceforge.net/oldhome.html

[24] Modbus IDA, “Modbus Application Protocol Specification v1.1a”, North Grafton, Massachusetts (www.modbus.org/specs.php), 2004.

[25] Modbus IDA, “Modbus Messaging on TCP/IP Implementation Guide v1.0a”, North Grafton, Massachusetts (www.modbus.org/specs.php), 2004.

[26] Modbus.org, “Modbus over Serial Line Specification and Implementation Guide v1.0”, North Grafton, Massachusetts (www.modbus.org/specs.php), 2002.

[27] Morris, Thomas and Kalyan Pavurapu. A retrofit network transaction data logger and intrusion detection system for transmission and distribution substations. In Power and Energy (PECon), 2010 IEEE International Conference on, pages 958–963, 2010.

[28] Moteff, John and Paul Parfomak (2004). *Critical infrastructure and key assets: definition and identification*.

[29] Niland, M. Computer virus brings down train signals, *InformationWeek*, August, 2003.

[30] Northeast blackout of 2003. March 2, 2012. n. pag. http://en.wikipedia.org/wiki/Northeast_blackout_of_2003

[31] Pal, Om, Sharda Saiwan, Peeyush Jain, Zia Saquib, and Dhiren Patel. Cryptographic key management for SCADA system: An architectural framework. In Advances in Computing, Control, & Telecommunication Technologies, 2009. ACT '09. International Conference on, pages 169–174, 2009.

[32] Pothamsetty, V., & Franz, M. (2005). "Scada honeynet project: Building honeypots for industrial networks". http://scadahoneynet.sourceforge.net/

[33] Poulsen, K. Slammer worm crashed Ohio nuke plant network, *SecurityFocus*, August, 2003.

[34] PROFINET International: Overview. March 1, 2012. n. pag. http://www.profibus.com/technology/profinet/overview/

[35] Provos, Niels, and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. 4th. Laflin: Addison-Wesley, 2010. Print.

[36] Schwartz, M., J. Mulder, J. Trent and W. Atkins, Control system devices: Architectures and supply channels overview, SANDIA Report SAND2010-5183, Sandia National Laboratories, Albuquerque, New Mexico, 2010

[37] Skoudis, Ed, and Tom Liston (2009). *Counter hack : a step-by-step guide to computer attacks and effective defenses.* Upper Saddle River, N.J.; London: Prentice Hall PTR ; Pearson Education [distributor].

[38] Spitzner, Lance (2002). *Honeypots: Tracking hackers*. (First ed.). Addison-Wesley.

[39] Stouffer, Keith, Falco, Joe, and Scarfone, Karen, NIST SP 800-82, *Guide to Industrial Control Systems (ICS) Security*, 2011

[40] Triangle MicroWorks, "DNP3 Overview", Raleigh, North Carolina (www.trianglemicroworks.com/documents/DNP3_Overview.pdf), 2002

[41] Ultra Electronics (2011), "EtherGuard Encryptor 3e-636S-2". http://www.ultra-3eti.com/products/cyberfence/etherguard_encryptor_3e-636s-2/

[42] United States Nuclear Regulatory Commission Office of Nuclear Reactor Regulation, Effects of ethernet-based, non-safety related controls on the safe and continued operation of nuclear power stations, NRC Information Notice 2007-15, April, 2007.

[43] Zetter, Kim. “10K Reasons to Worry About Critical Infrastructure”. http://www.wired.com/threatlevel/2012/01/10000-control-systems-online/, 2011. Retrieved on March 28, 2012.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

1. REPORT DATE *(DD-MM-YYYY)* 14 Jun 2012	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From – To)* Sep 2010 – Jun 2012

4. TITLE AND SUBTITLE Emulating Industrial Control System Field Devices Using Gumstix Technology	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER HSHQDC-11-X-00089
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Berman, Dustin, CIV	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCO/ENG/12-13
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Homeland Security ICS-CERT POC: Eric Cornelius, DHS ICS-CERT Technical Lead ATTN: NPPD/CS&C/NCSD/US-CERT Mailstop: 0635, 245 Murray Lane, SW, Bldg 140, Washington, DC 20528 email: ics-cert@dhs.gov phon: 1-877-776-7585	**10. SPONSOR/MONITOR'S ACRONYM(S)** DHS ICS_CERT **11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

This material is declared a work of te U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Industrial Control Systems (ICS) have an inherent lack of security and situational awareness capabilities at the field device level. Yet these systems comprise a significant portion of the nation's critical infrastructure. Currently, there is little insight into the characterization of attacks on ICS. Stuxnet provided an initial look at the type of tactics that can be employed to create physical damage via cyber means. The question still remains, however, as to the extent of malware and attacks that are targeting the critical infrastructure, along with the various methods employed to target systems associated with the ICS environment. This research presents a device using Gumstix technology that emulates an ICS field device. The emulation device is low-cost, adaptable to myriad ICS environments and provides logging capabilities at the field device level. The device was evaluated to ensure conformity to RFC standards and that the operating characteristics are consistent with actual field devices.

15. SUBJECT TERMS

Supervisory Control and Data Acquisition, Industrial Control System, Programmable Logic Controller, Emulation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Maj Jonathan W. Butts
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	122	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, x 4332 (Jonathan.Butts@afit.edu)