12-2022

Digital forensics for Investigating Control-logic Attacks in Industrial Control Systems

Nauman Zubair
University of New Orleans, New Orleans, nzubair@uno.edu

Digital forensics for Investigating Control-logic Attacks in Industrial Control Systems

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
Engineering and Applied Science
Computer Science

by

Nauman Zubair

B.E. University Of New Orleans

December, 2022

Abstract

Programmable logic controllers (PLC) are required to handle physical processes and thus crucial in critical infrastructures like power grids, nuclear facilities, and gas pipelines. Attacks on PLCs can have disastrous consequences, considering attacks like Stuxnet and TRISIS. Those attacks are examples of exploits where the attacker aims to inject into a target PLC malicious control logic, which engineering software compiles as a reliable code. When investigating a security incident, acquiring memory can provide valuable insight such as runtime system activities and memory-based artifacts which may contain the attacker's footprints. The existing memory acquisition tools for PLCs require a hardware-level debugging port or network protocol-based approaches, which are not practical in the real world or provide partial acquisition of memory.

This research work provides an overview of different attacks on PLCs. This work shows what embodies these three different approaches. These novel approaches leaves PLCs vulnerable that can unleash mayhem in the physical world.

The first approach describes denial of engineering operations (DEO) attacks in industrial control systems, referred to as a denial of decompilation (DoD) attack. The DoD attack involves obfuscating and installing a (malicious) control logic into a programmable logic controller (PLC) to fail the decompilation function in engineering software required to maintain control logic in PLCs. The existing seminal work on the DEO attacks exploits engineering software's improper input validation vulnerability. On the other hand, the DoD attack targets a fundamental design principle in compiling and decompiling control logic

in engineering software, thereby affecting the engineering software of multiple vendors. We evaluate the DoD attack on two major PLC manufacturers' PLCs, i.e., Schneider Electric Modicon M221 and Siemens S7-300. We show that simple obfuscation techniques on control logic are sufficient to compromise the decompilation function in their engineering software, i.e., SoMachine Basic and TIA Portal, respectively.

The second approach propose two control-logic attacks and a new memory acquisition framework for PLCs. The first attack modifies in-memory firmware such that the attacker takes control of a PLC's built-in functions. The second attack involves obfuscating and installing a malicious control logic into a target PLC to fail the decompilation process in engineering software. The proposed memory acquisition framework remotely acquires a PLC's volatile memory while the PLC is controlling a physical process. The main idea is to inject a harmless code that essentially copies the protected memory fragments to protocol-mapped memory space, which is acquirable over the network. Since the proposed memory acquisition allows access to the entire memory, we can also show the evidence of the attacks.

The third approach propose an attack which doesn't involve alteration or injection of PLC's control logic. Return Oriented Programming(ROP) is an exploiting technique which can perform sophisticated attacks by utilizing the existing code in the memory of the PLC. This attack doesn't involves injecting code which makes this technique unique and hard to discover. This work is the first attempt to introduce ROP attack technique successfully on PLC without disrupting the control logic cycle.

We evaluate the proposed methods on a gas pipeline testbed to demonstrate the attacks and how a forensic investigator can identify the attacks and other critical forensic artifacts using the proposed memory acquisition method.

Acknowledgements

I would like to express my deepest gratitude to my professor Dr. Yoo, for his invaluable patience, feedback and continuous support for my PhD study and research. His guidance was essential to my success. I couldn't have imagined a better advisor and mentor.

I would also like to thank the Computer Science Chair Dr. Abdelguerfi for his support. I would also like to thank my parents and my siblings for sticking with me through this journey.

Finally I want to thank Allah, for letting me through the difficult times. You are the reason for my success.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Industrial control systems (ICS) are used to control physical processes in critical infrastructures such as power grids, nuclear facilities, and gas pipelines and water treatment systems [5, 29] as shown in Figure 1.1. Since they are increasingly connected to corporate networks and the broader Internet for economic gain, they become vulnerable to cyberattacks. Within ICS, programmable logic controllers (PLCs) are embedded devices directly connected with physical processes at field sites using I/O devices such as sensors and actuators as shown in Figure 1.2 . Programmable logic controllers (PLC) are among the most commonly used controller types, which criminals and state-backed actors often target [19, 31, 17, 7, 34]. In ICS security incidents, forensic investigation on suspect PLCs is crucial to answering many questions about cyberattacks [2, 3]. They are often the main target of adversaries to sabotage physical processes. Specifically, PLCs run a special program called *control logic* that implements a logic to control the underlying physical process. At a control center, engineering (programming) software is used to program and compile a control logic and then, download and upload it to/from a PLC. Attackers modify the control logic in PLCs to manipulate the control over a physical process [18, 43].

Engineering operation in industrial control systems (ICS) is defined as a continuous cycle that develops and updates the logic of controllers, such as programmable logic controllers (PLC), in response to changing operational requirements [38]. Vendor-provided programming software, called *engineering software*, is used to program a control logic, which defines how a PLC controls a physical process, and then to download or upload it to/from a remote PLC over the network. Previous security incidents (e.g., Stuxnet, Triton) and academic studies (such as [26] and [48]) have shown that the control logic of a PLC is vulnerable to malicious modification. In case of a security incident, forensic investigators use engineering software to obtain the control logic from a suspicious PLC.

1.2 Work Outline

This work has discussed three completely different kind of attacks. These attacks utilize different strategies to harm the integrity of the PLC. The three attacks are as follows: Denial Of Engineering (DEO) operation, PLC Memory Extractor (PEM) and Return Oriented Programming (ROP). DEO attack utilize code injection in the PLC, PEM attack comprises of manipulating the functionality of the methods and lastly, ROP uses the existing memory of the PLC to form gadgets and use the gadgets to form malicious code.

1.3 Objective of the research

1.3.1 DEO

We have three attack scenarios which are called denial of engineering operations (DEO). The DEO attacks make use of the actual control logic by avoiding the capability of the programming software from an infected PLC. In the most common attack, the attacker hides the infection by removing the code which makes the programming software show the original uninfected code to the inquirer. In the second attack, the attacker adds noise to the

control logic instructions in packages which crashed the software. The first two attacks both employ the man in the middle approach but the third is different as it doesn't use the said approach. In this attack, the attacker uses a hostile control logic that runs but crashes the software when trying to take control from the PLC. The firmware is intact in these attacks so the infected control logic can be extracted for forensic analysis. We use the Ladder logic which is pretty commonly used language for PLC.

The seminal work of Senthivel *et al.* [38] presented three ICS attack scenarios, referred to as denial-of-engineering (DEO) attacks, that undermine the capability of engineering software to obtain the control-logic programs from infected PLCs. The most stealthiest DEO attack installs a well-crafted control-logic program that runs on a PLC successfully but prevents engineering software from acquiring the control logic from the PLC. However, their approach relies on an implementation vulnerability and thus is not widely applicable.

This work presents a new approach to the DEO attack, referred to as DoD Attack, which does not rely on implementation vulnerabilities. Instead, it exploits, through control-logic obfuscation, a fundamental design principle in compiling and decompiling control logic in engineering software. We evaluate DEO on two major PLC manufacturers' PLCs and engineering software—i.e., Schneider Electric Modicon M221, and Siemens S7-300. In addition, we empirically show that DEO can also evade existing control-logic detection (i.e., [49]).

1.3.2 Memory Acquisition and Analysis

Memory forensics has evolved in the IT domain over the past decade because of its unique role in providing a view of the system's runtime state and memory-based artifacts. Recent trends in malware development where malware routinely leaves no traces on non-volatile storage also emphasize the importance of volatile memory analysis [11]. Current approaches in acquiring a PLC's volatile memory mostly use an ICS protocol to read memory over a network [47, 37, 16, 49]. However, these approaches are limited in terms of the amount of memory they can acquire because not every memory address is mapped to an ICS protocol's

address space [35]. There is another direction that uses a debug port such as JTAG, but it requires physical access and disrupts a PLC's normal operation.

This work proposes a new approach for PLC memory acquisition. We present PEM (**P**LC m**EM**ory extractor), a nondisruptive remote memory acquisition framework for PLCs. The main idea is infecting the control logic of a PLC with a harmless memory duplicator which copies the memory contents unreachable from an ICS protocol to a memory region that is reachable through the protocol. We also propose a new control-logic attack that modifies in-memory firmware to be more stealthy and persistent compared to existing attacks. Further, we present a case study of PEM in investigating the control-logic attack on a gas pipeline testbed. The main contributions of this approach are:

1. We propose a forensic framework, PEM, to remotely acquire the entire memory of a PLC without interrupting the PLC's normal operation.

2. We present a control-logic attack that modifies the in-memory firmware of a PLC over a network.

3. Using PEM, we present a case study of investigating the control-logic attack on a gas pipeline testbed with the Schneider Electric Modicon M221 PLC.

Stuxnet [18], first discovered in 2010, manipulates the control logic of Siemens S7-300 PLCs to damage nuclear facilities in Iran. The PLCs control the rotative velocity of centrifuges through variable sequence drives to enrich Uranium-235. Stuxnet modifies the control logic of the PLCs to manipulate the motor speed periodically from 1,410 Hz to 2 Hz to 1,064 Hz and then over again, which resulted in unrecoverable damages in about 1,000 centrifuges of the facility. Similarly,

TRISIS [43] targets the control logic of safety instrumented systems (SIS), also referred to as safety-PLCs, installed in a Saudi Arabian oil company. SIS is a PLC with a strong emphasis on reliability through redundant components such as multiple circuits/processors

and watchdog capabilities for self-diagnostic routines. TRISIS modifies the control logic of SIS to prevent it from safely shutting down the connected physical process.

Stuxnet [18] and TRISIS [43] represent the state-of-the-art real-world ICS malware. However, they only modify control logic to inject malicious control logic code. In this approach, we demonstrate that PLC can execute any arbitrary (malicious) code that is injected remotely and does not have to be a control logic. This capability opens up a completely new set of attack vectors on PLCs that do not restrict to control logic. As proof of this offensive capability, we present three new remote code execution attacks on PLCs: 1) PLC function hijacking, 2) remote PLC memory acquisition, and 3) control-logic code obfuscation. These attacks exploit the inherent design vulnerabilities of current PLCs used in industry settings to execute arbitrary (malicious) code. The vulnerabilities include remote read/write access to a large PLC memory address space using PLC communication protocols, no protection of PLC code and data structures for malicious modifications, and no prevention of data execution thereby, allowing PLC to execute code/data in any part of PLC memory.

PLC function hijacking attack is a remote firmware attack that modifies a jump table loaded on on-chip RAM by the PLC firmware, thereby redirecting a normal PLC function call into the injected malicious payload. *Remote memory acquisition attack* injects a small piece of code to the protocol-mapped memory space that copies the internal memory (which is not remotely accessible) into the area which can be read through the PLC protocol. This technique may also be used by forensic investigators for good, but it is not the focus of this work. We further show that the acquired memory data can reveal to the attacker the original control logic and information about the data points used by the control logic. These attacks do not modify the original control logic. Thus, when the engineering software retrieves the control logic from the PLC, it will always receive the original logic.

We evaluate the attacks on Schneider Electric's Modicon M221 PLC successfully. For PLC function hijacking, we design a malicious timer that nullifies the normal timer's counting feature in control logic. For remote PLC memory acquisition, we analyze the on-chip RAM

and external RAM data acquired from a PLC to identify digital artifacts in memory. Lastly, for the control-logic obfuscation attack, we evade `Shade` [49], which is a shadow-memory system to identify control logic code in PLC memory.

Contributions The contribution of this approach is summarized as follows:

- We demonstrate that the remote code execution is possible on PLCs by exploiting their inherent design vulnerabilities.
- We present three novel remote code execution attacks on PLCs, i.e., PLC function hijacking, remote PLC memory acquisition, and control logic code obfuscation.
- We evaluate the attacks successfully on a real-world PLC device, Schneider Electric's Modicon M221 PLC that is used in industry settings.

Memory forensics has evolved in the IT domain over the past decade because of its unique role in providing a view of the system's runtime state and memory-based artifacts. Recent trends in malware development where malware routinely leaves no traces on non-volatile storage also emphasize the importance of volatile memory analysis [11]. Current approaches in acquiring a PLC's volatile memory mostly use an ICS protocol to read memory over a network [47, 37, 16, 49]. However, these approaches are limited in terms of the amount of memory they can acquire because not every memory address is mapped to an ICS protocol's address space [35]. There is another direction that uses a debug port such as JTAG, but it requires physical access and disrupts a PLC's normal operation.

1.3.3 Return Oriented Programming Attack

Return Oriented Programming (ROP) have never been utilized as a pragmatic ICS attack prior to this work. This attack technique is stealthy and sophisticated to stay undetected with current ICS security. ROP attack process requires attacker to select malicious packets ending with return instructions, this malicious packets followed by return instruction are

known as gadgets. Since the attacker changes an application's code operations sequence, she does not have to inject a new malicious code to launch an attack [10, 41, 44].

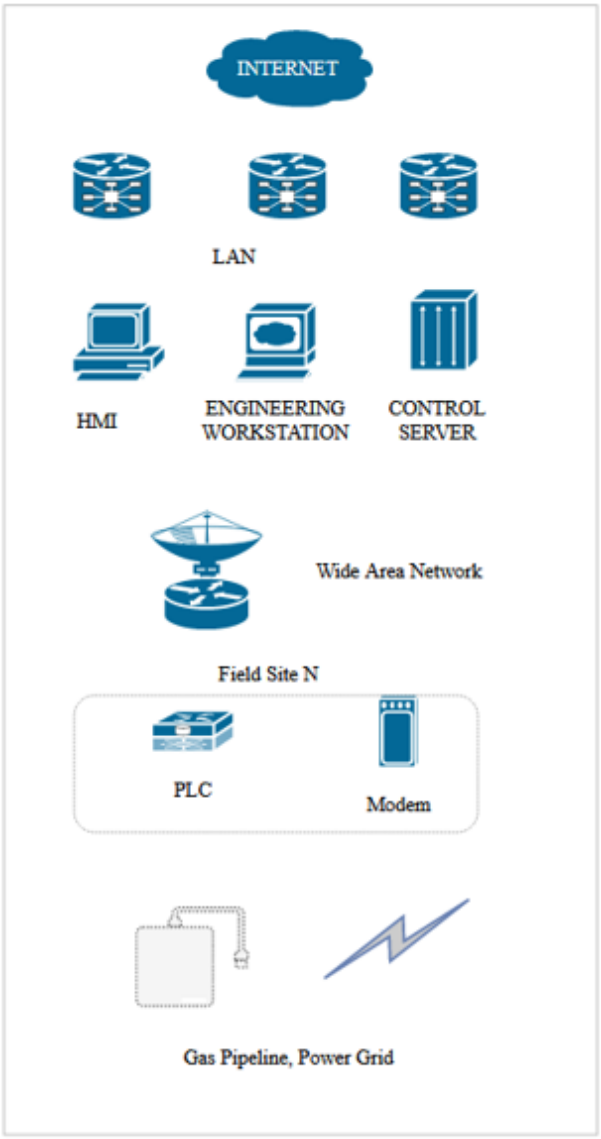


Figure 1.1: Industrial Control System

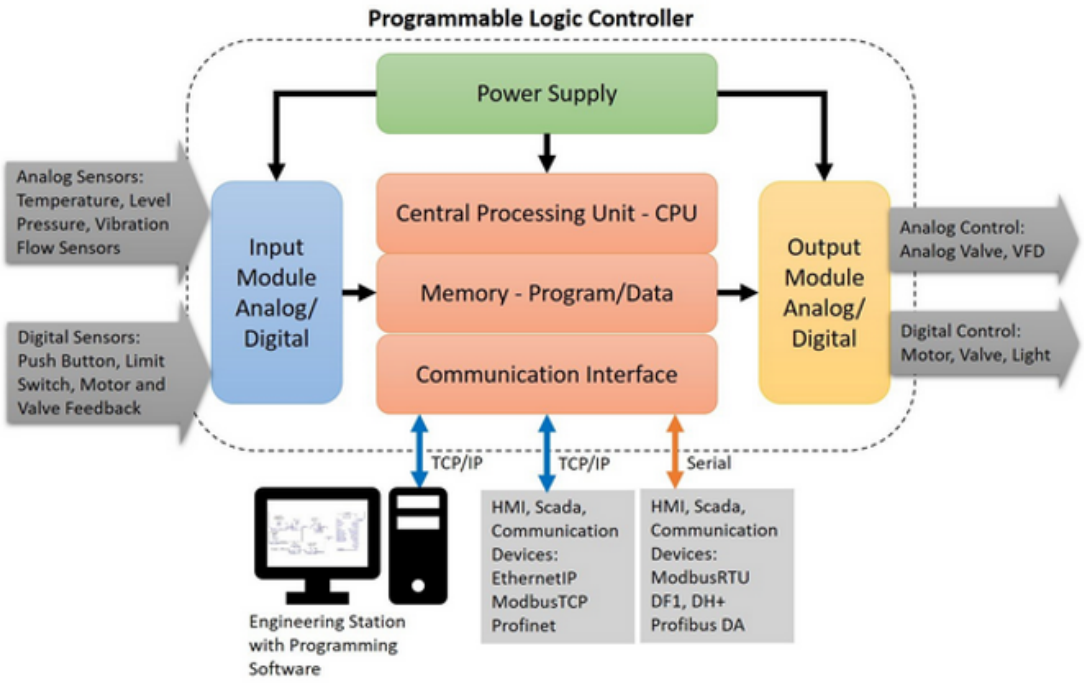


Figure 1.2: Programming Logic Controller Architecture

Chapter 2

Literature Review

2.1 PLC, Engineering Software and Control Logic

PLCs are widely used in industries and critical infrastructures. They are embedded devices that are designed to operate in harsh environments and are the main target of a cyberattack to sabotage a physical process [20, 39, 4, 26]. A PLC connects with multiple input/output devices (e.g., switches, push-buttons, solenoid valve, etc.) to control a physical process such as gas pipeline, water pump, traffic light signals.

The engineering workstation runs an engineering software to write control logic for PLCs. This proprietary programming software is offered by ICS vendors to configure, program, and perform maintenance on their PLCs. For instance, SoMachine Basic, RsLogix 500, and CX-Programmer are used for the PLCs of Schneider Electric, Allen-Bradley, and Omron, respectively. Most modern PLCs are equipped with an Ethernet interface to download and upload a compiled control logic to/from a PLC over the network using the engineering software referred to as PLC programming channel.

2.2 Control Logic Attacks

Sethivel *et al.* [39] shows three new attack scenarios, referred to as denial of engineering operations (DEO) attacks that subvert the capability of a vendor-supplied PLC programming software to acquire the control logic from a PLC remotely. These attacks demonstrate that forensic investigators cannot use the PLC programming software as reliable forensic data acquisition tools. However, the DEO attacks rely on man-in-the-middle attack capability (i.e., employing arp spoofing) and software bugs in the engineering software.

Kalle *et al.* [26] present a full attack-chain on the control logic of a PLC. The attack-chain dynamically generate a malicious control logic based on the original logic retrieved from a target PLC. It employs a control logic decompiler which decompiles binary logic code into an instruction list code, then modifies the decompiled code into a malicious one using simple rule-based modification. The malicious code is compiled and downloaded into the PLC. It hides the infection of control logic from the control center application using a virtual-PLC located at a man-in-the-middle position. Although it utilizes its own compiler, the compilation method is exactly the same as the vendor-provided engineering software. Therefore, a forensic investigator can examine the attacker's logic code using the engineering software.

Yoo and Ahmed [48] propose two evasion techniques called data execution and fragmentation & noise padding, against deep packet inspection on PLC protocols. Data execution evades protocol-header based logic code detection by executing logic code in the data section of PLC memory. When the logic code is transferred over the network, its packet header indicates that it is data, not code, deceiving detection rules on headers. Fragmentation & noise padding is an extreme padding attack where attackers craft packets with only one or two bytes of logic code per packet, and pads the rest with noise or normal-looking data. They also propose a network-based defense technique against the evasion attacks, which employs a shadow-memory that maintains a local copy of PLC memory [49]. While the defense technique is effective in defeating the evasion attacks, it fails to detect obfuscated control logic

code as shown in Section 3.2.1.

Garcia *et al.* [20] present a PLC rootkit that infects a PLC at the firmware level. The firmware of a PLC provides interfaces between control logic code and the hardware resource of the PLC (e.g., input and output ports). Since a firmware-level rootkit can directly control the hardware resource below the control logic code, it is the most stealthy and dangerous type of attack on PLCs. They infect the PLC firmware through a JTAG interface, which requires attacker's local access to a target PLC. On the other hand, in Section 3.2.2, we show that an attacker can remotely manipulate a jump table loaded from the firmware, thereby hijacking a PLC function call.

Return oriented programming(ROP) for control logic is never explored before as a practical ICS attack. It is one of the sneakiest attack to avoid current security measures in ICS environment. ROP is an exploit technique in which the attacker utilizes the PLC memory to coin gadgets. Gadgets are valid instructions that are followed by return instruction. These gadgets can be combined to form instructions to perform malicious operations. The attack code can be manipulated to the attacker's need. These gadgets are all found in the memory to execute the attack [40, 10, 44]

Shacham [40] first presented return oriented programming as a return-to-libc attack without function calls on x86 architecture. He showed how to combine instruction sequences from memory to generate gadget chains that perform arbitrary operation.

Carlini *et al.* [9] presented two return oriented programming attack methods that break defenses such as ROPpecker and kBouncer. They use three main building blocks to bypass detection by these tools. 1) Call-Preceded ROP (in which they show that ROP attacks are possible even when defenses make sure that return instructions always targets an instruction that immediately follows some call), 2) Evasion attacks (in which the gadgets classified as normal by defenses can be used), and 3) History flushing (in which the history of all signs of ROP attack is cleansed to evade detection).

Weidler *et al.* [45] presents ROP on microcontrollers. He presented a gadget set that was

capable of erasing flash memory and then re-programming this region and a Turing complete gadget set that was capable of performing arbitrary computation.

Jaloyan *et al.* [25] presents ROP attack on RISC architecture. They show that gadgets on this architecture are rich enough to mount complex attacks. They give examples and proof of concept gadget chains. They also present ROP gadget finder algorithm that finds their new class of gadgets.

Homescu *et al.* [22] present a turing complete gadget set using gadgets that are restricted to only 2 or 3 bytes. He calls them microgadgets and argue that the small size increase the likelihood of finding all required gadgets. They also present an efficient scanner that finds these gadgets in Linux distributions to show that several of them contain these microgadgets that attacker can use to perform arbitrary computation. Traditional use of ROP is software exploitation. Some effort has been made in using ROP for benign purposes as well. Our work is different in a way that we do not really exploit the PLC but the physical process that it controls. The PLC functions and behaves as normal while the physical process that it controls behaves maliciously.

Chapter 3

Methodology

3.1 Background and Related Work

3.1.1 Programmable Logic Controllers

Programmable logic controllers (PLC) are embedded systems used in various industries to control physical processes. Engineers at a control center can program over a network a PLC at a remote field site using *engineering software*. They use the software to write a PLC program, called *control logic*, compile and transfer it to a PLC. Then, the PLC runs the control logic in an infinite loop to control its underlying physical process. Engineering software can also retrieve the control logic from the PLC over a network, and show the decompiled source code using its built-in decompiler.

3.1.2 Roles of Engineering Software in Forensic Investigation

Engineering software plays a critical role in digital forensics and incident response against ICS cyberattacks. Attackers' control logic codifies their intention to the physical world; hence a high priority needs for understanding their (malicious) control logic in incident response. The built-in decompiler of engineering software allows forensic investigators (or control operators) examine the acquired control logic in a high-level PLC programming language (e.g., ladder

logic, functional block diagram).

Unlike forensic investigations on typical IT systems, decompilation is an essential process to understand the semantics of the control-logic binary since the embedded systems in ICS are much more heterogeneous and exclusive than common IT systems. In most cases, if not all, the file formats of control-logic binaries are proprietary. In addition, the library/system functions called in the control-logic code and the information of a PLC's memory-mapped I/O are generally unknown. Of course, the investigators can consult with PLC manufacturers, but it will significantly delay response time, causing extended damage to the physical world.

3.2 Denial of Engineering Operation (DEO) Attacks

The seminal work of Senthivel *et al.* [38] presented three attack scenarios, called DEO attacks, to subvert engineering software's capabilities to acquire and decompile the actual control logic from a suspicious PLC. The first two attacks employ a man-in-the-middle (MITM) approach to intercept and modify the network traffic when the control logic of a PLC is being transferred over the network, while the third attack (DEO attack-3) does not require the MITM capability.

DEO attack-3 is more difficult to detect or respond to than the other two attacks since it allows an attacker to leave the network after installing a malicious control logic into a target PLC. Specifically, the attack creates a well-crafted control-logic program that 1) runs on a PLC successfully 2) but disables engineering software's decompilation function. Senthivel *et al.* [38]'s approach on DEO attack-3 creates a malformed control-logic program that exploits an inconsistency of input validation between engineering software and a PLC. For example, in their study, Rockwell Automation's RSLogix 500 (the engineering software) refused to decompile a control-logic bytecode when integrity checks failed, while the Allen-Bradley MicroLogix 1400 PLC ran the logic successfully.

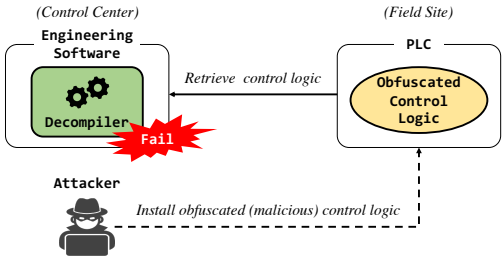


Figure 3.1: An overview of the denial-of-decompilation (DoD) attack in ICS

Their approach, however, needs to find a malformed instance that can trigger the discrepancy between engineering software's and the PLC's input validation, which is by no means a trivial task; in some cases, such inconsistency may not exist at all. In addition, PLC manufacturers can quickly fix the problem; likewise, investigators can easily correct the mismatched integrity-check values in a suspect control logic.

3.2.1 DEO Attack using Control Logic Obfuscation

This section proposes a new approach for the DEO attack, referred to as DoD Attack, that exploits a fundamental design principle of engineering software's compilation/decompilation. Figure 3.1 shows an overview of DEO attack. An attacker installs an obfuscated (malicious) control logic into a PLC and leaves the network. Then, when a forensics investigator attempts to acquire the control logic from the PLC using engineering software, the attempt fails since the decompilation function of the software is not defined under the obfuscated control logic.

Attack Scenarios:

Attack 1: Fooling the engineering software. When PLC is operating, engineering software can monitor the performance of the PLC to ensure that the ladder logic is behaving correctly. There are scenarios where an attacker can perform Man in the middle attack. In first case, when a new control logic is being downloaded to the PLC from the engineering software. Attacker can replace the machine code of that logic. The second case is when a control logic is already downloaded to PLC and ladder logic uses timers. When an engineering software sends signal to PLC and the timer becomes active. The counter for timer is sent to PLC

every time it changes through packets of machine code and as soon as the counter of the timer reaches the preset the desire logic takes place. If that counter of the timer is altered during the exchange the attacker can manipulate the outcome.

Attack 2: Crashing the PLC programming software. In this scenario attacker implements man in the middle attack to acquire the traffic between the programming software running on engineering workstation and the targeted PLC. The attacker acts as a middle person through which all the traffic passes through. This attack takes place when the engineering software is acquiring the ladder logic machine code from the PLC. Attacker replaces the machine code acquired from the PLC and replaces it with obfuscated code, which engineering software fails to compile and comprehend that cause the engineering software to crash.

Attack 3: Feeding an artificial payload to PLC. In this attack situation, the attacker infects the PLC with an obfuscated payload. This attack creates a window for attack one and two to be performed. There are two scenarios in which payload can affect the process. In first scenario the PLC is uploaded with ladder logic with well-crafted binary altered payload which compiles successfully in PLC. However, when the engineering software attempts to run it. It fails to understand the binary modification made by the attacker and the software crashes. This attack is the sneakiest because the attacker doesn't have to be present during exchange as compared to the two attacks mentioned above.

IL (Instruction List)	Assembly RX	Compiled Machine Code 1
LD I0.1	BTST 1, r12	7C 1C
ST Q0.0	BMC 0, r13	FD E0 2D
LD I0.2	BTST 2, r12	7C 2C
ST Q0.1	BMC 1, r13	FD E1 2D
	RTS	02

Table 3.1: Original Machine Code

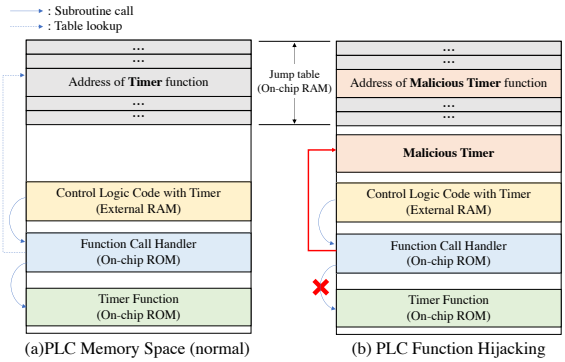


Figure 3.2: PLC function hijacking

Assembly RX Obstruction	Obfuscation Code
MOV.L 0, R1	66 01
BTST 1, R12	7C 1C
BMC 1, R1	FD E1 21
CMP 1, R1	61 11
BMGE 0, R13	FD E0 8D
MOV.L 0, R1	66 01
BTST 2, R12	7C 2C
BMC 1, R1	FD E1 21
CMP 1, R1	61 11
BMGE 1, R13	FD E1 8D
RTS	02

Table 3.2: Example of Obfuscation Code

The second scenario of this attack is to infect PLC with payload which modifies part of firmware which is responsible for timer. Instead of obfuscating the ladder logic, which causes interpretation of the ladder logic on the engineering software side, this scenario causes timer to perform erratically. This attack once performed can infect all the ladder logic codes running on that PLC. It is sneaky in a way where attacker can choose whether the timer is always is set to "On" or "Off". The attack can only be countered by resetting the firmware which Is executed by restarting the PLC

3.2.2 PLC Function Hijacking

Stuxnet [18] infects the control logic of Siemens S7-300 PLCs and hides the infection through its rootkit component (s7otbxdx.dll) installed on the compromised Windows systems in control centers. The rootkit intercepts all the communication between the STEP7 engineering software and an infected PLC, thereby modifying the data sent to or responded from the PLC. However, since the rootkit operates on the side of control center applications (e.g., engineering workstations and HMIs), the operators of the PLC can detect the infection from undamaged systems.

On the other hand, PLC rootkits such as HARVEY [20] can be very stealthy by infecting the firmware of a PLC. Since the firmware has full control over the PLC including the execution of control logic, the communications with control center applications, and input/output hardware, malicious firmware can directly manipulate the underlying physical process without infecting the control logic, which makes it difficult for the operators to aware the attack from control centers. As with a rootkit on common IT systems exploiting a kernel-process has a root privilege and be able to hide itself from system monitoring mechanisms, a firmware-level rootkit has privileged access to the PLC device including the ability to control the connected physical process covertly.

While the firmware of a PLC is a favorable target for an attacker who wants to achieve a great stealthiness, modifying the firmware *remotely* has been considered a difficult task in practice. In most cases, PLC firmware update is only possible through local access (e.g., USB interfaces, SD cards), unlike control logic updates, which can be done over the network. Also, while attackers can find vulnerabilities on firmware update processes (e.g., insecure checksum validation) [8], sometimes firmware update on a PLC is protected by cryptographic means such as digital signature (e.g., signing the firmware image with a vendor's private key). Although Garcia *et al.* [20] show that attackers can bypass the protection and infect PLC firmware through a debugging interface such as JTAG, it requires the attacker's physical access to a target device.

We present a new firmware attack on a PLC that can be conducted *remotely* through the PLC programming channel. The firmware code in a PLC is directly executed from EEPROM. The address space mapped to EEPROM is read-only, which prevents the injected code (through the programming channel) from modifying the firmware code. However, the firmware includes a jump table which contains a PLC's function pointers and loads the table on the on-chip RAM at boot time. If the RAM area is set as read-write, the injected code modifies a jump table entry to redirect a normal PLC function call to the malicious payload injected together.

Figure 3.2(a) shows a simplified address space of a PLC and the control flow of a PLC function call. Some control logic instruction such as `timers` or `counters` makes a function call to the actual implementation of the instruction in the firmware code (On-chip ROM). For example, a timer instruction makes a call to a function call handler on the firmware. Then, the handler looks up the jump table to find the right entry containing the corresponding function pointer and jumps to the function.

If attackers modify the function pointer in the table, they can hijack the normal function call and redirect it to malicious code as shown in Figure 3.2(b). They inject code through a target PLC's programming channel. The injected code dumps a malicious payload into a safe memory region which is not overwritten at control logic updates and modifies a target entry of the jump table. Then, whenever control logic makes a call to the affected function, it will execute the malicious payload. Since the function call handling is done by the firmware below the control-logic level, the tampering is not detectable from the control center applications and remains effective after control logic updates. Although the infection does not survive power cycling because the manipulation is made on the RAM area, it can be almost permanent in practice since power cycling a PLC is rarely done in typical ICS environments.

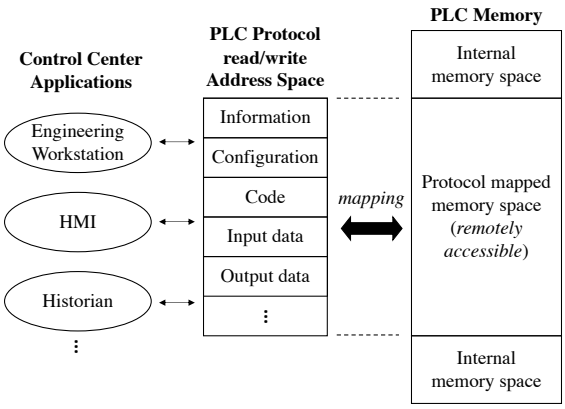


Figure 3.3: Mapping between PLC protocol address space and memory space

3.2.3 Remote PLC Memory Acquisition

PLC memory data is a critical digital artifact for both attackers and forensic investigators. To the attackers, it provides valuable information about the control logic of a target PLC, which enables them to generate malicious control logic precisely. It can also reveal vulnerabilities and possible attack vectors on the PLC. For the forensics investigators, given a suspicious PLC in ICS cyberattack incidents, acquiring the PLC memory data including RAM and EEPROM is necessary to identify any malicious modification on the PLC.

The existing memory acquisition techniques require local access to a PLC and utilize a debugging interface such as JTAG or UART [6]. However, PLC vendors often hide the debugging interfaces or physically remove them from a PCB at the end of the production process. Chip-off techniques may be used to desolder the flash memory chip and read it using a chip programmer [46]. These methods could destroy the device under investigation, and more importantly, inevitably interrupt a PLC's operation on the connected physical process.

3.3 PEM

This section presents a memory acquisition framework for PLCs, PEM (**P**LC m**EM**ory extractor), which remotely acquires the memory of a PLC in operation by appending memory copying code (called *a duplicator*) to the PLC's control logic code.

3.3.1 Main Idea

We establish three requirements for PLC memory acquisition, considering the characteristics of the industrial control systems (e.g., assuring 24/7 availability, operating physical processes, controllers at remote field sites).

1. *Nondisruptive*. Memory acquisition must be accomplished while a target PLC is controlling a physical process. This requirement is crucial in most ICS environments where very high level of availability is expected.

2. *Remote*. PLC memory should be acquired over a network. PLCs are often spread over geographically dispersed remote field sites in a large ICS setting, such as SCADA systems of power grids. In this case, the remote forensic capability will enable a faster incident response to cyberattacks.

3. *Complete*. A forensic investigator should be able to acquire the entire memory of a PLC.

Existing approaches meet only one or two requirements. For example, utilizing a debug port can be complete but disruptive and requires physical access to a PLC. Using an ICS protocol is nondisruptive and remote but not complete. Completeness is important since an attacker's footprint may reside only in the memory region that is not mapped to an ICS protocol's address space (we will discuss this kind of attack in Section 3.3.3).

In this work, we propose a new approach that meets all three requirements. The main idea is quite simple. It infects the control logic of a PLC with a harmless memory duplicator which copies the memory contents unreachable from an ICS protocol to a memory region that is mapped to the protocol's address space. In each scan cycle, the duplicator will copy a block of memory from *non-protocol-mapped space* to *protocol-mapped space*, which then can be readable over a network using the ICS protocol. The original control logic of a PLC runs *before* the appended duplicator. Therefore, the PLC can still control its underlying physical process during memory acquisition.

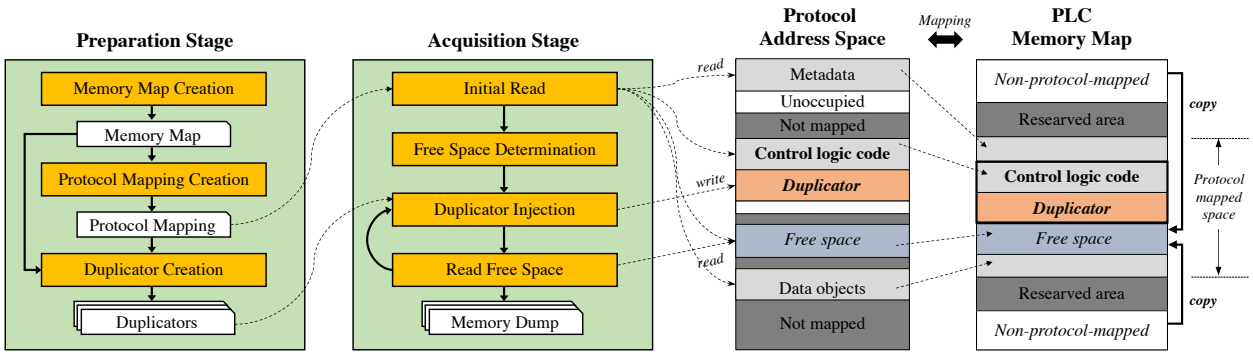


Figure 3.4: The design overview of PEM

Requirements First, the proposed approach assumes the memory duplication code can access the whole memory space of a PLC to achieve completeness. This assumption can be justified in current industrial settings. Many embedded devices like PLCs lack hardware supports for memory protection (e.g., MMU, MPU) [1]. Even when a PLC supports a hardware feature such as Memory Protection Unit (MPU), in practice, access control policies are often not fine-grained enough, running a PLC program in a privileged mode [17, 12]. Secondly, we assume that we can modify a PLC's control logic without requiring the PLC to stop executing the logic. Major PLC manufacturers such as Schneider Electric and Rockwell Automation support online editing in their PLCs (e.g., Modicon M221, MicroLogix, and ControlLogix) that allows changing a running control logic.

3.3.2 The Design of PEM

Figure 3.4 shows the design overview of PEM. It downloads one or more *duplicators* into a PLC through an ICS protocol. A duplicator copies a certain amount of non-protocol-mapped memory into an unoccupied protocol-mapped space named *free space*.

A metadata region in protocol-mapped space can also be used as free space. The metadata of control logic includes a programmer's comments and symbols created for data objects. For example, we found about 6 KB of metadata region in a PLC (Schneider Electric Modicon M221) that runs control logic for a gas pipeline system. The region includes a zip file containing an XML document of the control logic's metadata. Since the metadata does

not affect the execution of control logic, we can overwrite it once acquired. The free space can be pre-determined (if a specific space is never used in a particular PLC model) in the preparation stage of PEM or dynamically decided after scanning the protocol address space of a target PLC.

Preparation stage The preparation stage of PEM consists of the following three steps. Although each step requires manual engineering effort, they need to be done only once for a given PLC model.

1) Memory map creation. This step prepares the memory map of a PLC's microcontroller. Usually, we can obtain a processor's memory map from hardware manuals or datasheets provided on its vendor's website. Without available datasheets, we will need to create one from scratch using a debugging interface. In this case, we can utilize the recent study [35] that presents a systematical method to create the memory map of a PLC through the JTAG interface.

2) Protocol mapping creation. The protocol mapping information (i.e., the mapping between an ICS protocol's address space and a PLC's memory space) can be determined by looking at how the addresses of data objects in source code (e.g., a ladder diagram) are translated in the compiled machine code; and how they are translated in the address fields of protocol messages. This process may require reverse engineering the protocol to infer its message formats [13, 14] if it is proprietary. Along the process, we can determine which protocol addresses are not mapped (i.e., PEM does not have to read those addresses during acquisition) and which are never occupied (i.e., ideal for the free space).

3) Duplicator creation. Given the two pieces of information (i.e., memory map and protocol mapping), duplicators can be generated for each memory block in the non-protocol-mapped space. Duplicators may copy different sizes of data. The free space size determines the maximum block size that can be copied by one duplicator. As we will see in Section 4.2, a duplicator's block size affects a PLC's scan time (i.e., the larger the size of the block to

be copied, the longer the scan time.). An assembly language would be a preferred language in writing duplicators. The actual calling convention used in a PLC's firmware may not be the same as the one used in a high-level language compiler provided by a microprocessor/microcontroller vendor (e.g., `rx-elf-gcc` [36]). For example, the firmware could use a different set of callee-saved registers; thus, a compiler output may not preserve some register values it should have.

Acquisition stage PEM performs remote memory acquisition through the following steps.

1) Initial Read. The first step is reading the entire protocol-mapped space through a target PLC's communication protocol. If different protocol addresses are mapped to the same memory address, acquired data will be duplicated. PEM can use the protocol mapping generated in the preparation stage to avoid duplicate memory acquisition.

2) Free space determination. If a free space has not been determined in the preparation stage, PEM scans the acquired protocol-mapped memory to find an unoccupied region. For example, it can search a large chunk (e.g., above 1 KB) of consecutive `0x00`, which may indicate an unused memory region. When a free space has been pre-determined in the preparation stage, it can simply check that the space is indeed safe to use. If a new free space is selected, duplicators' destination addresses and block sizes need to be fixed.

3) Duplicator injection. Next, it downloads a duplicator to the PLC using the protocol's write request messages. It appends the duplicator into the control logic, overwriting the logic code's return instruction. Then, the duplicator will be executed in each scan cycle after the control logic, copying a (non-protocol-mapped) memory block into the free space.

4) Read free space. Then, it reads the free space using the protocol's read request messages. After that, it repeats steps 3 and 4 until there are no duplicators left to be injected. Note that we do not need a separate channel for synchronization between a duplicator's copying operation and PEM's read operation over a network; since PLCs handle communication requests after executing control logic in their scan cycle.

3.3.3 Control-Logic Attack

This section presents a new control-logic attack that modifies in-memory firmware instead of a user-defined PLC program to be more stealthy and persistent. We also discuss that PEM can acquire the evidence of the attack while the existing memory acquisition approaches cannot.

3.3.4 Existing Attacks

Control-logic modification attacks (or control-logic attacks) are the attacks that change the way a PLC controls a physical process [42]. Existing control-logic attacks [19, 26, 38, 48, 33, 32] mainly focus on modifying a PLC program written by a user. However, any modifications in a user-defined PLC program can be easily detected since an ICS protocol can access the PLC program (i.e., the PLC program resides in a memory region that is mapped to the protocol's address space). Therefore, a forensic investigator can use engineering software to retrieve the modified PLC program from a suspect PLC and analyze it. Similarly, an ICS operator can easily overwrite the PLC program modified by an attacker using engineering software to recover the PLC's normal operation.

On the other hand, firmware modification attacks on PLCs [8, 21] can be more stealthy and persistent. Engineering software can retrieve a PLC program from a PLC but usually does not support reading the firmware over a network (i.e., the firmware areas in memory are not mapped to an ICS protocol's address space). While the firmware of a PLC is a favorable target for an attacker who wants to achieve great stealthiness, modifying the firmware *remotely* has been considered a difficult task in practice. In most cases, PLC firmware update is only possible through local access (e.g., USB interfaces, SD cards). Moreover, the firmware update is generally protected by cryptographic means such as a digital signature (i.e., signing the firmware image with a vendor's private key). Although Garcia *et al.* [21] show that an attacker can bypass the protection and infect PLC firmware through the JTAG interface, the attack requires physical access to a target device.

3.3.5 Proposed Attack

Main idea The proposed attack targets in-memory firmware that can be modified by a PLC program injected via an ICS protocol; hence it is a *remote* attack. Firmware code is usually executed directly from EEPROM (or flash memory), and the address space mapped to EEPROM is read-only, preventing an injected PLC program from modifying the firmware code. However, a portion of the firmware is loaded into RAM (i.e., in-memory firmware) at boot time. For example, the firmware of a PLC can have a *jump table* loaded into RAM, which contains the pointers to the PLC's built-in functions (e.g., timers, counters, PID control). If a PLC program has read-write access to that memory area, an attacker can inject a malicious PLC program that modifies a table entry in RAM to redirect a built-in function call to a malicious function.

Requirements The following three are the requirements for the proposed attack.

1. An attacker can command a target PLC over a network. Typically, this can be achieved by compromising engineering workstations at control centers.

2. A PLC program has read-write access to the RAM area where the jump table is loaded that contains the addresses of a PLC's built-in functions.

3. There is non-protocol-mapped space in memory that can be both writable/executable. This is required as a malicious function will be written and executed in that space unreachable from an ICS protocol.

Attack method The attack appends malicious code into the end of a running PLC program. The malicious code consists of three parts: an injector, a payload (a malicious implementation of a target function), and the code for jump table modification.

The injector copies the payload into a non-protocol-mapped area for persistence. Generally, a PLC's protocol-mapped space is overwritten when a new PLC program is downloaded

into a PLC, whereas the non-protocol-mapped space is not affected. The target address where the payload to be injected can be pre-determined before the attack. Then, the malicious code modifies a target function's table entry to the payload's address. After that, the attack overwrites the appended malicious code with `0x00` to clean up the attack footprint from the protocol-mapped space not to be detected from control center applications.

Detection Existing memory acquisition methods cannot acquire the evidence of the attack. Using a debug port requires power cycling, thereby losing the evidence of tampering in volatile memory. Note that the attack only modifies in-memory firmware (i.e., in RAM), not the firmware in EEPROM, so its existence disappears after power cycling. On the other hand, previous ICS protocol-based approaches cannot read the infected memory data in non-protocol-mapped space. On the contrary, PEM can effectively detect the attack since it reads the entire memory without requiring hardware interference with a suspect PLC.

3.3.6 Example: Hijacking Timer

This section presents an example attack implementation on Schneider Electric Modicon TM221CE16R (referred to as Modicon M221) that runs on Renesas RX630 microcontroller. In this example, the PLC's built-in *timer* function is hijacked.

Timer in control logic Figure 3.5 shows an example of control logic written in ladder diagram (LD), which is one of the most popular PLC programming languages. It is a graphical language that looks similar to the circuit diagram of relay logic, with two vertical rails and one or more horizontal *rungs* between them. Each rung has zero or more input instructions (i.e., logical checkers) on the left and has one or more output instructions (i.e., logical actuators) on the right. In each scan cycle, a PLC executes ladder logic from the top to the bottom rung and from left to right within a rung. The evaluation result (either true or false) of a rung's logical expression (consisting of input instructions) affects the behaviors of the rung's output instructions.

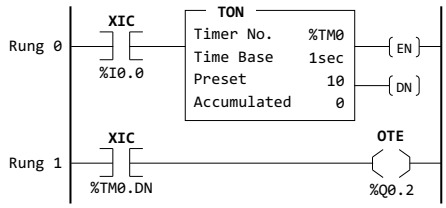


Figure 3.5: Example ladder logic diagram with timer

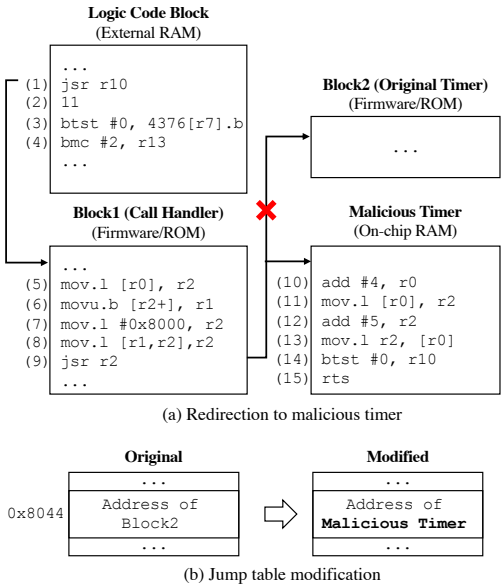


Figure 3.6: Hijacking a PLC timer function

The example ladder logic has three different instructions: XIC, TON, and OTE. XIC (examine-if-closed) is an input instruction that examines a bit and evaluates as true if the bit value is one and vice-versa. The XIC on Rung 0 checks %I0.0 (input port 0 on slot 0), and if the bit is one, then executes TON (timer-on-delay), the timer instruction. Since the timer's time base is one second and its preset is 10, it counts for 10 seconds. When the accumulated count reaches the preset, it sets the DN (done) bit. On Rung 1, the XIC examines the DN bit of the timer (%TM0). If it is set, the OTE (output energize) instruction is executed, which sets %Q0.2 (output port 2 on slot 0).

Normal control flow In Figure 3.6(a), `Logic Code Block` shows a snippet of the RX assembly code compiled by the PLC's engineering software (i.e., SoMachine Basic). The timer instruction makes a subroutine call to `Block1` (a call handler) by `(1) jsr r10` (the

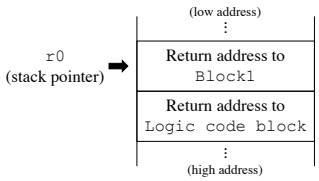


Figure 3.7: The runtime stack

R10 register maintains the address of `Block1`, which is 0xFFF3E1EF). Then, `Block1` refers to the jump table to find the address of the built-in timer function. The jump table is loaded at a fixed address (0x8000) on the PLC's on-chip RAM region (i.e., the in-memory firmware region). `Block1` calculates the address of the corresponding table entry (`(5)-(8)`) based on the one-byte index value next to the subroutine call in `Logic Code Block` (i.e., `(2) 11`). Since the index value is 0x11, the address of the table entry is calculated as 0x8044. `Block1` makes a subroutine call to `Block2` (the built-in timer function) (`(9) jsr r2`) of which address is stored at 0x8044, then `Block2` performs the necessary task for the timer logic. When the control flow returns to `Logic Code Block`, `(3) btst #0, 4376[r7].b` is executed, which tests the DN (done) bit of the timer and set the zero and carry flags of the CPU as follows:

$$Z\ (Zero\ flag)\ = \sim (([r7 + 4376]\ >>\ 0)\ \&\ 1)$$

$$C\ (Carry\ flag)\ =\ (([r7 + 4376]\ >>\ 0)\ \&\ 1)$$

The bit 0 (i.e., bit position 0) at the address `[r7+4376]` stores the timer's DN bit. If the bit is set, the `btst` instruction sets the carry flag and vice-versa. When the carry flag is set, `(4) bmc #2, r13` sets the bit 2 of R13 (the R13 register is mapped to the PLC's output ports), actuating the output device connected to the PLC's output port 2.

Malicious timer In this example, we hijack the PLC's timer function. We download our malicious code into the PLC, which injects `Malicious Timer` (i.e., the payload) into an on-chip RAM area (which is non-protocol-mapped space). After placing the payload, the malicious code modifies the address stored at 0x8044 (i.e., the jump table entry for timer)

to the payload's address.

Figure 3.6(a) `Malicious Timer` shows a simplified assembly code of the actual payload. When `Block1` calls it, the return address to `Block1` is pushed on the stack (Figure 3.7 shows the state of the stack right after executing `(9) jsr r2`). The return address to `Logic Code Block` (i.e., the address of `(2) 11`) has been pushed on the stack.

`Malicious Timer` nullifies a timer's counting logic by always setting the carry flag and skipping the instruction that tests the timer's DN bit (`(3) btst #0, 4376[r7].b`). The instructions from `(10)` to `(13)` modify the stack pointer and the return address to `Logic Code Block` on the stack, so that when `Malicious Timer` returns, it directly returns to `(4) bmc #2, r13`. Since `(14) btst #0, r10` always sets the carry flag (bit 0 of the R10 register is always one), `(4) bmc #2, r13` always sets bit 2 of R13 regardless of the actual state of the timer's DN bit, thereby immediately actuating the connected output device ignoring the intended time delays.

3.3.7 Mitigation

This section discuss some mitigation strategies to the remote code execution on PLCs.

Operate with RUN mode PLCs often support three different operation modes which is configured using a physical method (e.g., hardware key). In `RUN` mode, one can not overwrite the control logic section in the memory. In `PROGRAM` mode, a PLC can be programmed by engineering software. And in `REMOTE` mode, engineer can remotely change the mode of the PLC. One of the best practice to keep the PLC protected is to operate the PLC in `RUN` mode during normal operation. However, PLCs often run in `REMOTE` mode for convenience in practice. In addition, since it does not prevent an attacker inject code into the data section in the memory, it is vulnerable to data execution attack [48].

Secure authentication protocol Most PLCs use password-based authentication protocol to authenticate users who wants to communicate with them. However, the PLC vendors

typically define their own authentication protocols and do not disclose how they work, which often makes the authentication protocol weak [23, 15, 24].

Control logic code signing Code signing is a mechanism of authenticating the authors of executables or scripts based on cryptographic measures. It is widely used in IT systems to authenticate the code publisher and provide the integrity of the code. Cryptographic hardware modules such as HSM can be used to safely keep the private key . To the best of our knowledge, code signing is not used in PLCs. Although a code signing system can be breached if not properly designed [27, 28], it can improve the security of PLCs against unauthorized remote code execution threats.

3.3.8 Compilation and Decompilation in Engineering Software

We can define a compilation as a multi-valued function τ, given the source language L_1 and the target language L_2. For each string (source program) $x \in L_1$,

$$\tau : L_1 \mapsto \mathcal{P}(L_2); \tau(x) = \{y \in L_2 : sem(x) = sem(y)\}$$

where $sem(x)$ represents the semantics of the program x. Note that the compilation is multi-valued since usually some statement of a high-level (source) language can have several different realizations in a low-level (target) language. However, if we consider a particular compiler f, the compilation is single-valued, because the compiler selects exactly one out of the many possible low-level implementations of the source program.

$$f : L_1 \mapsto L_2; f(x) = ys.t. y \in L_2 \wedge sem(x) = sem(y)$$

In general, a compilation is not injective because more than one source programs may be translated into the same target program. Therefore, in typical IT systems, a decompilation (reconstructing a source program from a target program) is not the inverse of a compilation;

compilation and decompilation are independently designed and do not necessarily make the same design decisions for their mappings.

On the other hand, the compilation in engineering software is injective—namely, two different control-logic source programs are always translated into two different target programs. There is an evident advantage in this design principle. Control operators often need to examine or debug the control logic running in a PLC, which requires the *original* source program. We can make this possible in two ways. The first way is to transfer the binary and source code together when updating the control logic of a PLC. However, this approach wastes limited PLC memory resources to store the source code. Moreover, the source code can be exposed during transmission since most ICS protocols do not support encryption. The second way is to make the compilation function *invertible*; given a compiler f, we define a decompiler g such that:

$$g : f(L_1) \mapsto L_1; g(f(x)) = x for all x \in L_1$$

We exploit this design principle in compilation and decompilation to achieve our attack goal (i.e., install into a PLC a control logic that cannot be decompiled in engineering software). Since the domain of a decompiler g is restricted to $f(L_1)$, which is a strict subset of L_2, given a source program $x \in L_1$, we can find a target program y such that $y \in \tau(x)$ but $y \notin f(L_1)$. In other word, the target program y has the same semantics as the source program x, but it can never be generated by the particular compiler f. The function g cannot decompile the target program since it is not defined for y.

3.3.9 Control-logic Obfuscation

We can find such a target program y ($y \in \tau(x)$ but $y \notin f(L_1)$) through obfuscation, which is a common practice in malware development in the IT domain. We can define an obfuscation

as a multi-valued function δ:

$$\delta : f(L_1) \mapsto \mathcal{P}(L_2); \delta(f(x)) = \{y : y \in \tau(x) \wedge y \neq f(x)\}$$

In other words, given a target program $f(x)$, which is the output of a particular compiler f on an input source program x, the obfuscation produces a set of morph y, which is a target program whose semantics is the same as $f(x)$ (and so as x), but whose realization is different. Note that a morph $y \in \delta(f(x))$ could be a member of $f(L_1)$ by chance, meaning y is defined under a decompiler g. Then, $g(y)$ will produce x' such that $sem(x) = sem(x')$ and $x \neq x'$ (x and x' cannot be the same since the compilation function is injective). We can test whether a morph can be used for using the decompiler g (in practice, we use engineering software for the test). If the decompiler generates an error, we can use it, otherwise select another morph from $\delta(f(x))$.

To implement a particular obfuscator $\delta'(f(x)) \subseteq \delta(f(x))$, we can borrow common obfuscation strategies that have been extensively studied in the IT domain. However, the purposes are somewhat different. In the IT domain, attackers often obfuscate their malware greatly, and performance is a low priority. On the other hand, a PLC and its physical process operate within the real-time constraint; thus, complex obfuscation techniques (e.g., emulation-based, return-oriented programming), which can significantly increase the execution time, may not be suitable. We argue that simple obfuscation is enough for our purpose to hinder incident response due to the reasons mentioned in Section 3.1—it is challenging to analyze control-logic binary even without obfuscation (due to the heterogeneity and exclusiveness) if a decompiler is unusable. The following section presents two case studies that perform through simple instruction-level obfuscation on two major manufacturers' PLCs.

3.4 Return Oriented Programming

We describe the first-ever ROP attack against PLCs in this research. We demonstrate how ROP can be used to attack a PLC control logic that interferes with the operation of the physically connected process. We read the memory and obtain its dumps using the ICS protocol. The RX Renesas toolchain is then used to disassemble the dumps and locate gadgets in the memory. Lastly, we change the stack pointer to point at the gadgets we wish to run.

We followed the following steps to scrutinize the memory to extract gadgets and perform ROP

1. Using the ICS protocol, obtain the PLC's memory
2. Implement a script that creates combination of different machine code to create different gadgets
3. Handpicked gadgets that will has malicious effect on PLC.

3.5 Implementation

In this section we discussed the steps to execute ROP attack

PLC's memory acquisition First step is to acquire memory of the PLC through the attack as discussed in the 3.3 . This method utilizes ICS network protocol to access the free space location. This step is crucial in investigating the memory

Acquired memory analysis The acquired memory is in assembly code. Using the reasas toolchain, the binaries of the memory is formed by disassembling the assembly code. Analysis of the memory of control logic portion of different programs helped us to determine the stack pointer location. This stack pointer location was carried in R10 register through the processes of the PLC.

Finding the gadgets and generating a gadget database We then use the same approach followed by [40] to find gadgets except that we look for the opcode `'0x02'` instead of `'0xc3'` to find the return instruction. We not only look for gadgets that exist as a result of the code-generation choices of the compiler but the ones that exist otherwise as well. For instance, the disassembly of the on-chip ROM region in M221 PLC shows the machine code `"e5 12 08 02"` which translates to **"mov.l 32[r1], 8[r2]"** as a result of the compiler code-generation choice. However, the last two bytes of this machine code `"08 02"`can also form the instruction sequence, **"bra.s 0x8 rts"** which we can use as a gadget. Algorithm **??** shows how we find gadgets in the memory dump. First we look for `'0x02'` byte in the binary file. Then at this byte, we step back and look for the maximum number of bytes that can make a valid instruction. There can be multiple possible instructions that can exist before a return such Add, Sub, Jump, etc. We record all the possible instruction sequences and keep repeating the process of finding instruction sequences until no more instruction sequence is found. This way we get a combination of gadgets from just one return instruction.

3.5.1 The Design of PEM

Finding useful gadgets from the gadget database Once we have all the gadgets available, we look for useful ones that can help us accomplish our task.

Generating a gadget chain As a final step, we can combine gadgets to form a chain of gadgets for the attacker to manipulate PLC with the desire malicious code.

```
1: maxIntSize ← 0x08
2: gadgets ← []
3: for each offset of 0x02 do
4:     failCount ← 0
5:     startAddr ← the offset -1
6:     stopAddr ← the offset +1
7:     while failCount ≤ maxIntSize & startAddr ≥ 0 do
8:         res ← the result of the following command:
       rx-elf-objdump –D –b binary –m rx <filename>
       –start-address =start_addr –stop-address =stop_addr
9:         startAddr ← startAddr − 1
10:        if res ends with 'rts' and does not contain the string
           "*unknown" then
11:            gadgets ← res
12:            failCount ← 0
13:        else:
14:            failCount ← failCount +1
15:        end if
16:    end while
17: end for
```

Figure 3.8: Algorithm for finding valid ROP gadgets from a memory dump

Chapter 4

Results and Discussion

4.1 Experimental Evaluation of DEO

We have evaluated DEO attack (i.e., the obfuscation-based DEO attack) on two different manufacturers' PLCs: Schneider Electric Modicon M221 and Siemens S7-300. We utilized the well-known instruction-level obfuscation strategies such as garbage-code insertion, equivalent-instructions substitutions [30]. Given a control-logic source program, we first compiled it using engineering software, and extracted the control-logic binary. Then, we disassembled the binary into an assembly code to which obfuscation was applied, and assembled back to machine code. The obfuscated control-logic binary was transferred and installed into a PLC, then we checked that the control logic ran successfully in the PLC while engineering software failed to decompile when attempting to acquire the control logic from the PLC. In addition, although it is not the primary goal of DEO attack, we conducted a separate experiment to see whether the obfuscated control logic can also evade [49] which is a ML-based control-logic detection.

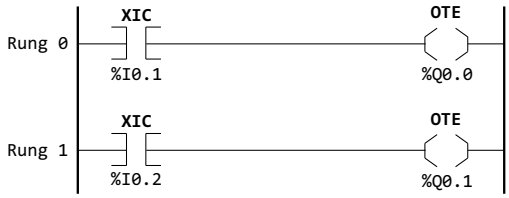


Figure 4.1: An example control-logic source (Modicon M221)

Table 4.1: Original and obfuscated programs (M221)

Original	Morph-1	Morph-2
BTST #1, r12 BMC #0, r13 BTST #2, r12 BMC #1, r13 RTS	TST #2, r12 BMNE #0, r13 TST #4, r12 BMNE #1, r13 RTS	MOV.L #0, r1 BTST #1, r12 BMC #1, r1 CMP #1, r1 BMGE #0, r13 MOV.L #0, r1 BTST #2, r12 BMC #1, r1 CMP #1, r1 BMGE #1, r13 RTS

4.1.1 Subverting Decompilation in Engineering Software

Attack on Modicon M221 PLC Our first subject were the Schneider Electric Modicon M221 PLC (firmware v1.6.0.1) which runs on a Renesas RX630 microcontroller, and SoMachine Basic (v1.6), the engineering software. We utilized a toolchain[1] provided by Renesas to perform assembling/disassembling for the RX architecture. The obfuscated control logic was transferred into the PLC using a rouge clients [48].

Figure 4.1 shows a control-logic source program. The `XIC` (examine-if-closed) instruction on the first line (Rung 0) examines a PLC's digital input `%I0.1` (input port 1 on slot 0), and if the bit is set, then the connected `OTE` instruction is executed, which sets the digital output `%Q0.0` (output port 0 on slot 0). Then, the next line (Rung 1) is executed in a similar way, and then over again for each scan cycle.

Table 4.1 represents the original target program (produced by SoMachine Basic) for the example source program, and its morphs generated through instruction-level obfuscations. In the original program, `BTST #1,r12` examines bit 1 (i.e., the second to the LSB) of the `r12` register which reflects the digital inputs of the PLC; namely, it tests `%I0.1`. The `BTST`

[1]https://gcc-renesas.com/

instruction has two operands—`BTST src,src2`. If `src2` is a register, then it sets the carry flag as following:

$$Carry\ flag\ =\ ((\ src2\ >>\ (\ src\ \&\ 31\ ))\ \&\ 1\ )$$

Thus, `BTST #1,r12` sets the carry flag only if bit 1 of `r12` is set. The next instruction `BMC #0,r13` sets bit 0 of `r13` if the carry flag is set. Since the bits of `r13` are mapped to the PLC's digital outputs, the instruction basically actuates the output device connected to the output port 0 on the PLC's slot 0, when the carry flag is set, and vice versa. Lastly, `RTS` is a return instruction.

Morph 1—It was generated through equivalent-instructions substitution. The `TST` instruction, replacing `BTST` in the original code, performs a logical AND operation on its two operands and sets the zero flag if the result is zero, otherwise clears. Then, we can substitute `BMC` with `BMNE` which sets a bit if the zero flag is 0, otherwise clears the bit. To sum up, an instruction sequence (`BTST`, `BMC`) can be substituted with an equivalent sequence (`TST`, `BMNE`) in the RX machine code.

Morph 2—It represents a bit more complicated obfuscation. First, `MOV.L #0, r1` clears `r1` because we will use `r1` later. Then, the next `BTST` instruction checks bit 1 of the input register (`r12`) and modifies the carry flag accordingly as in the original program. However, the logic executed by a single `BMC` instruction in the original program was stretched over three instructions—`BMC`, `CMP`, and `BMGE`.

Both morphs ran successfully on the Modicon M221 PLC, but SoMachine Basic failed to decompile them.

Attack on S7-300 PLC Our second subject were Siemens S7-300 (firmware v3.2.17) and TIA Portal (v16). We used Radare2 with a library[2] to disassemble MC7 bytecode (which is the target language for S7-300) into the STL language (which is an assembly-like language corresponding to MC7 bytecode). The Snap7 library and its python wrapper[3] were used to

[2]https://github.com/wargio/libmc7
[3]https://github.com/gijzelaerr/python-snap7

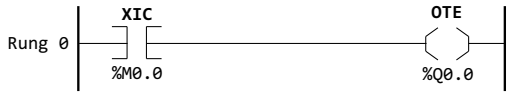


Figure 4.2: An example control-logic source (S7-300)

Table 4.2: Original and obfuscated programs (S7-300)

Original			Morph-1		
Offset	MC7	STL	Offset	MC7	STL
0x24 0x26 0x28	8000 d880 6500	A M 0.0 = Q 0.0 BE	0x24 0x28 0x2a 0x2c	700b00 02 8000 d880 6500	JU 0x28 A M 0.0 = Q 0.0 BE

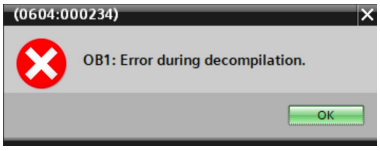


Figure 4.3: An decompilation error message from TIA Portal

download the obfuscated code into the PLC.

Figure 4.2 shows an example control-logic source. Table 4.2 describes the original target program (produced by TIA Portal) for the example source program, and a morph generated through garbage-code insertion. In this example, the garbage code is the jump-unconditional (JU) instruction that we used for merely jumping to the next instruction; namely, it plays like a no-operation (NOP) instruction. The obfuscated code ran well on S7-300 while TIA Portal generated an error message (refer to Figure 4.3) when attempting to retrieve the control logic from the PLC.

4.1.2 Evading ML-based Control-logic Detection

We also evaluated the control-logic obfuscation against [49], which detects network packets containing control-logic code using a ML-based approach. Since does not support Siemens S7-300, we conducted an experiment only for Modicon M221. Our test dataset contained 14 different original programs and their corresponding morphs. We utilized two obfuscation strategies; 1) inserting `NOP` instructions between each assembly instruction; 2) substituting

(TST, BMNE) for (BTST, BMC).

We configured using a support-vector machine (SVM) with the radial basis function (RBF) kernel. Among the 42 different features studied in [49], we used the L4gram feature, which represents the longest continuous match of 4-grams that are present in a pre-generated bloom filter. The original programs were transferred in 578 packets, of which 469 packets were detected (86.17% accuracy) by Shade. On the other hand, the morphs were transferred in 1700 packets, of which only 61 packets were detected (3.59% accuracy). This experiment result indicates that control-logic obfuscation can also be used to evade the machine-learning models that are trained using the compilation output produced by engineering software. paragraphExperiment result The attack has been conducted on Modicon M221 connected to LED indicator lights that runs simple control logic with a timer. The attack has successfully hijacked the calls to the original timer function, turning on an LED light without a time delay specified in control logic.

4.2 Case Study: Investigation of a control-logic attack on a gas pipeline testbed using PEM

This section presents a case study of forensic investigation into a control-logic attack on a gas pipeline testbed. The attack hijacks a built-in *comparison* function of a PLC controlling the gas pipeline. To investigate the attack, we use PEM to acquire the infected PLC's memory and analyze it to identify the attack and other important information. Further, we evaluate the performance of PEM in the testbed environment.

4.2.1 A gas pipeline testbed

Gas pipeline Gas pipeline systems are used for safely transferring natural gas over long distances, often at high pressure (typically 200-1500 psi). In this case study, we use a testbed that simulates a gas pipeline utilizing compressed air. Figure 4.4 shows a top-view

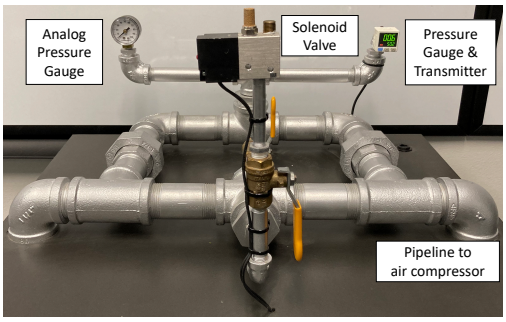


Figure 4.4: A top-view of the gas pipeline testbed

of the testbed. An air compressor installed under the pipeline (not shown in the figure) feeds compressed air to the pipe. A Schneider Electric Modicon M221 (TM221CE16R) PLC receives through an attached I/O module (TM3AM6) analog input signals from a pressure gauge/transmitter (Panasonic DP-102A-N-P). Then, through its two relay-type digital output ports, the PLC controls open/close of a solenoid valve (Grainger P251SS-024-D) and on/off of the air compressor, to maintain the pressure in the pipeline at a desired level. The PLC's control logic is written in the ladder logic language, consisting of 16 rungs. In short, the logic opens the valve when the measured pressure is greater than 400 KPa. It stops the air compressor if the pressure exceeds 600 KPa and starts again when the pressure drops to 200 KPa.

Modicon M221 PLC (TM221CE16R) The PLC has a 32-bit microcontroller (Renesas RX630) that operates at 100 MHz. The microcontroller includes 1.5 MB of main flash memory, 32 KB of data flash memory, and 128 KB of (on-chip) SRAM. It also has a memory protection unit (MPU), but control logic runs with privileged mode, thereby allowing both the proposed attack and PEM. Modicon M221 is equipped with additional 512 KB of external SRAM (Renesas R1LV0414DSB). And it supports up to 2 GB of optional SD card (which needs to be formatted using either FAT or FAT32), but not utilized in the testbed's PLC. The PLC has 9 digital inputs (24V), 2 analog inputs (0-10V) and 7 relay-type digital outputs (5-125 V DC / 5-250 V AC). An I/O expansion module (TM3AM6) having 4 analog inputs and 2 analog outputs is attached to the PLC at the testbed. In addition, the PLC supports four communication interfaces: USB 2.0, RS232/RS485, and Ethernet (100BASE-TX). In

the testbed, the PLC is connected with a HMI and an engineering workstation through an Ethernet switch. Modicon M221 uses a proprietary ICS protocol that is encapsulated by the Modbus protocol. Although there is no publicly available official document about the protocol, it has been partially reverse engineered [26, 48]. Lastly, the firmware version used is v1.6.0.1.

4.2.2 Control-logic Attack Scenario

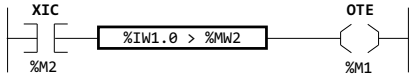
We present a targeted attack specially designed for a gas pipeline system. We assume that the attacker knows the target PLC model and firmware version so that she can prepare a pre-compiled malicious code before the attack.

A closed-loop system determines a control action by the difference between measured values and setpoints (or thresholds). To maintain the desired pressure in the pipeline, the control logic of a PLC inevitably *compares* an analog input value (i.e., a measured pressure value) to predefined setpoints. For example, our gas pipeline system compares the measured pressure with three thresholds: 200 KPa (low pressure), 400 KPa (high pressure), and 600 KPa (high-high pressure). If the measured pressure is greater than 400 KPa, the PLC opens the valve, and if it exceeds 600 KPa, the air compressor stops to lower the pressure.

Comparison operator in control logic In Modicon M221, comparison operators are implemented as built-in functions, which can be hijacked, as we saw in Section 3.3.3. If a comparison operator in control logic does not produce correct answers, the control actions from the logic execution would be wrong.

Figure 4.5(a) shows a ladder logic diagram containing a greater-than (>) operator. The logic evaluates the logical expression, (XIC %M2) AND (%IW1.0 > %MW2), where %M2, %IW1.0, and %MW2 are all data objects of control logic [4]. If it is true, %M1 is energized. Figure 4.5(b) represents the compiled binary code. Line 1 tests bit 2 at the address [R7 +

[4] %M represents memory bit, %MW for memory word, and %IW for analog input.



(a) A ladder logic diagram with a comparison operator

```
line  binary code       comments
   1: f6 72 00 00     ; btst #2, 0[r7].b
   2: 23 0b           ; bnc.b 0xf
   3: 7f 1a           ; jsr r10
   4: 27              ; (arg) table entry number
   5: 16              ; (arg) left type
   6: 14              ; (arg) right type
   7: ac 91           ; (arg) left address
   8: 04 81           ; (arg) right address
   9: fc e6 72 00 00  ; bmc #1, 0[r7].b
  10: 02              ; rts
```

(b) The compiled binary code

Figure 4.5: A ladder logic diagram and its binary code with a comparison operator

```
line  binary code  assembly
   1: 62 40      ; add #4, r0
   2: ec 02      ; mov.l [r0], r2
   3: 62 72      ; add #7, r2
   4: e3 02      ; mov.l r2, [r0]
   5: 7c 07      ; btst #0, r7
   6: 02         ; rts
```

Figure 4.6: The binary code of a malicious '>' operator

0], and sets the carry flag if it is one, otherwise clears the flag (R7 points the base address of the block of data objects, which is 0x07018000. Memory bits are the first objects in the block, so the bit 2 at 0x07018000 corresponds to %M2).

Line 2 jumps to line 9, if the carry flag is zero, otherwise it falls through to line 3, which calls a subroutine. As we saw in Section 3.3.3, the subroutine pointed by R10 is a call handler, and the next byte on line 4 is an index number to the jump table. Lines 4-8 are not machine code, but rather arguments used by the call handler and the called function. The table index 0x27 is for the greater-than operator. The next two bytes tell us about the left and right operand types of the operator: 0x16 means analog input (%IW) and 0x14 means memory word (%MW). Line 7 and 8 specify the lower 16-bit addresses of each operand: the left operand's address is 0x070191ac (i.e., the address of %IW1.0) and the right operand's address is 0x07018104 (i.e., the address of %MW2). Line 9 sets bit 1 at the address [r7 + 0] (i.e., %M1) if the carry flag is set, otherwise it clears the bit. Lastly, line 10 represents the return instruction, marking the end of the control logic code.

Malicious comparison operator This attack hijacks the greater-than operator by modifying its table entry values at the memory address 0x809c (0x8000 + 0x27 * 4). It changes the value to 0x1EB00 where the *malicious* greater-than operator is injected. The injection location is selected since there is an unoccupied space of size 432 bytes between 0x1EA80 0x1EC2F.

Figure 4.6 shows the binary code of the malicious operator. Line 1 adds four to R0 (the stack pointer), making the malicious operator directly return to the logic code block, skipping the call handler. Line 2-4 adds seven to the return address to skip the seven bytes of arguments and return to the bmc (bit move conditional) instruction (refer to line 9 of Figure 4.5(b)). Line 5 always clears the carry flag since R7 fixed to 0x07108000, so its bit 0 is always zero. When the control flow returns to the bmc instruction, the OTE instruction's memory bit is always de-energized (i.e., set to zero) since the carry flag is always zero. Namely, greater-than operators in control logic are always evaluated as false, meaning the comparisons of the measured pressure with setpoints will not work—only 11 bytes of code can physically destroy the gas pipeline system.

Attack impact The two comparison operations—'measured pressure > 400 KPa' and 'measured pressure > 600 KPa'—always return false. Therefore, the PLC never opens the valve nor stops the air compressor, making the pressure in the pipeline to rapidly exceed 600 KPa. An operator at a control center reads the control logic from a suspect PLC, but she would find it is just the original normal logic. Then, the operator downloads a new control logic to reconfigure the PLC, but nothing is changed (since it only overwrites protocol-mapped space). In our experiment, although the galvanized steel pipeline is built to overwhelm the system's air compressor, we manually shut down the testbed when the pressure exceeded 600 KPa by pulling off its power plug to prevent any possible damage to the system.

Table 4.3: The memory map of Modicon M221

Start	End	Size	Description
0x00000000	0x0001FFFF	128KB	On-chip RAM
0x00080000	0x000FFFFF	512KB	Peripheral I/O
0x00100000	0x00107FFF	32KB	On-chip ROM
0x007F8000	0x007F9FFF	8KB	FCU-RAM
0x007FC000	0x007FC4FF	1280B	Peripheral I/O
0x007FFC00	0x007FFFFF	1KB	Peripheral I/O
0x07000000	0x0707FFFF	512KB	External RAM
0xFEFFE000	0xFEFFFFFF	8KB	On-chip ROM
0xFF7FC000	0xFF7FFFFF	16KB	On-chip ROM
0xFFE80000	0xFFFFFFFF	1.5MB	On-chip ROM

4.2.3 Memory Acquisition from a Suspect PLC

In this section, we acquire and analyze the suspect PLC's memory using PEM. The malicious comparison operator and modified jump table are only in the RAM of the suspect PLC. If we reboot the PLC, the trace of attack will disappear. After manually turning off the air compressor (to avoid any explosion), we use PEM to acquire the memory.

Implementation To use PEM, we create a memory map of the PLC from the datasheets of Renesas RX 630 microcontroller (Part No. R5F5630DCDBG) and Schneider Electric Modicon M221 (TM221CE16R). Table 4.3 shows the memory map of the PLC.

Control logic code and its data objects are placed in the 512 KB of external RAM (0x07000000–0x0707FFFF), which we found mapped to the protocol address space (i.e., protocol-mapped space). We read the protocol-mapped space of the PLC with varying control logic in the preparation stage of PEM, which revealed that 0x7030000–0x7040000 is never occupied. Therefore, we use that pre-determined space as a free space for PEM.

We implement PEM in Python (the current implementation supports only Modicon M221). Total 40 duplicators are created with different source addresses and block sizes (the maximum block size is 64 KB, which equals the free space size) in the preparation stage.

Figure 4.7 shows the 49-byte size of binary code of a duplicator, which copies 64 KB

```
offset   binary code          assembly
     0:  7e a1                push.l r1
     2:  7e a2                push.l r2
     4:  7e a3                push.l r3
     6:  7e a8                push.l r8
     8:  7e aa                push.l r10
     a:  66 03                mov.l  #0, r3
     c:  fb 82 00 00 03 07    mov.l  #0x7030000, r8
    12:  fb aa 00 40          mov.l  #0x4000, r10
    16:  66 01                mov.l  #0, r1
    18:  ec 32                mov.l  [r3], r2
    1a:  e3 82                mov.l  r2, [r8]
    1c:  62 43                add    #4, r3
    1e:  62 48                add    #4, r8
    20:  62 11                add    #1, r1
    22:  47 a1                cmp    r10, r1
    24:  2b f4                ble.b  0x18
    26:  7e ba                pop    r10
    28:  7e b8                pop    r8
    2a:  7e b3                pop    r3
    2c:  7e b2                pop    r2
    2e:  7e b1                pop    r1
    30:  02                   rts
```

Figure 4.7: A duplicator of PEM

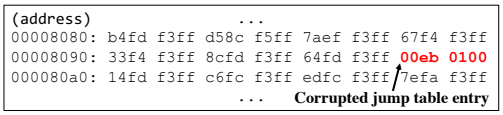


Figure 4.8: The corrupted jump table

of On-chip RAM (0x00000000–0x0000FFFF) to the free space (0x07030000–0x07040000). We wrote the code in the RX assembly language and converted it into an executable file (ELF format) using the `rx-elf-as` tool, then extracted only the code section using `rx-elf-objcopy` [36].

The target PLC's control logic (controlling the gas pipeline) is 343-byte size located at 0x0701E26C. Therefore, duplicators are injected in turn at 0x071E3C2 (0x0701E26C + 342), overwriting the logic code's last byte (0x02), which is the return instruction. After injecting a duplicator, the size of code increases to 391 bytes (i.e., 342-byte of original logic + 49-byte of duplicator) with a new return instruction at 0x0701E3F2 (0x0701E26C + 390).

Memory acquisition result PEM acquires a total of 2,820,352 bytes (about 2.69 MB) memory dump. Out of that, we read 512 KB (i.e., external RAM) in the initial read step while the other memory was copied by the duplicators to the free space first and read over a network.

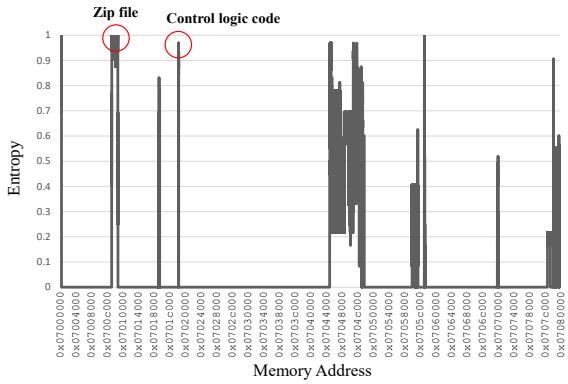


Figure 4.9: Entropy analysis on memory dump

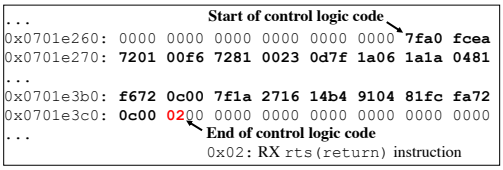


Figure 4.10: Control logic code in the memory dump

4.2.4 Verification and Analysis of Acquired Data

Attack identification We found that the jump table at 0xFFF5A3F4 (ROM) is loaded at 0x8000 (on-chip RAM) at boot time from the memory dump.

Figure 4.8 shows a portion of the jump table entries in the on-chip RAM region (i.e., in-memory firmware region). The jump table starts at the address 0x8000, and each entry is a 4-byte function pointer. We can recognize that the entry at 0x809c points 0x1EB00 (where the malicious payload was placed), which is in the on-chip RAM region, whereas all the other function pointers point to some addresses in the ROM region (0xFFE80000-0xFFFFFFFF). That is enough to raise suspicion, and we can further examine memory contents at 0x1EB00, identifying the malicious payload.

Control logic extraction Control logic is one of the most critical forensic artifacts in investigating ICS security incidents. We can understand attackers' intentions by analyzing their malicious control logic, thereby planning a better response strategy.

As forensic investigators, our purpose is to extract control logic from the memory dump and analyze it. Since control logic is in the protocol-mapped space, we can generally upload

it from a PLC and see the decompiled source code using engineering software. However, existing studies [38, 48] show that an attacker can subvert the engineering operations of the software. In that case, an investigator needs to extract and analyze the control logic from the PLC memory dump.

To find control logic code from the memory dump, we employ a simple entropy analysis, expecting the executable code has higher entropy than non-executable data. Figure 4.9 shows the entropy on the memory dump. We calculate Shannon entropy for each 16-consecutive byte string and divide it by the maximum entropy ($\log_2 16$) to normalize it. Then, we try decompiling start from each byte string of which entropy is greater than 0.7, using the `Eupheus` decompiler [26]. Figure 4.10 shows a portion of the memory dump where the logic code is found.

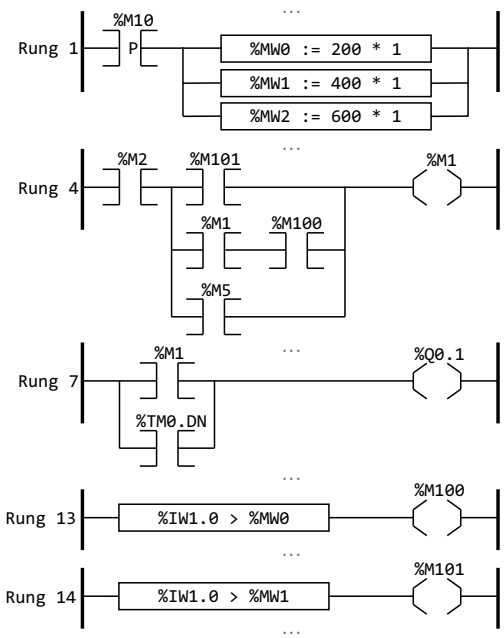


Figure 4.11: Decompiled control logic in ladder logic diagram

Decompilation of the control logic Figure 4.11 shows the decompiled ladder logic diagram. The decompiled logic code alone may not tell much about the controlled physical process. It shows the basic structure of the logic, but what exactly the logic controls are difficult to infer without the semantics of data objects (e.g., what %IW1.0 represents in the

physical process).

From the external RAM area, we have found signatures indicating a zip file: `\x50\x4b\x03\x04` (local file header), `\x50\x4b\x01\x02` (central directory file header), `\x50\x4b\x05\x06` (end of central directory record). Since the structure of a zip file is well known (refer to Figure 4.12), we can successfully extract the zip file and decompress it into an XML file.

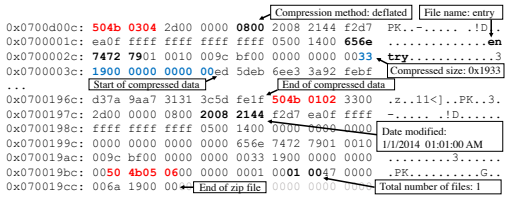


Figure 4.12: A zip file in the memory dump

We find that the XML file describes the semantics of data objects used in the control logic (refer to Figure 4.13). For example, %Q0.0 and %Q0.1 represent AIR_PUMP_RUN and SOLENOID_OPEN respectively. Using this information, we can infer that the ladder logic controls a physical process with an air-pump and a solenoid valve. Further, we can say that the logic controls a physical process that tries to maintain a gas pressure, based on the comments on the data objects.

We briefly explain how the logic (shown in Figure 4.11) controls open/close of the solenoid valve. On Rung 1, it first sets %MW0 (low pressure setpoint), %MW1 (high pressure setpoint), and %MW2 (high-high pressure setpoint) to 200, 400, and 600 respectively. Then, on Rung 4, %M1 (SOLENOID_ON) is set only if %M2 (PUMP_ON) is set and one of the following conditions is satisfied: 1) %MW101 is set (it is set if %IW1.0, the measured pressure represented between 4–20mA analog signal, is higher than %MW1 on Rung 14), 2) %M1 and %M100 are set (%M100 is set if %IW1.0 is higher than %MW0 on Rung 13), and 3) %M5 (FORCE_ON) is set. In other words, when the measured pressure exceeds the high pressure setpoint, it opens the valve and remains open until the pressure drops below the low pressure setpoint. An operator can also force the valve open in manual operation by setting the %M5 bit to one through an HMI or engineering software. On Rung 7, it actuates

```
<?xml version="1.0"?>
<MetaDataEntity xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<DeviceMetadata>
    <Q> <Address>%Q0.0</Address> <Index>0</Index>
        <Symbol>AIR_PUMP_RUN</Symbol> </Q>
    <Q> <Address>%Q0.1</Address> <Index>1</Index>
        <Symbol>SOLENOID_OPEN</Symbol> </Q>
                                    ...
    <IW> <Address>%IW1.0</Address> <Index>0</Index>
         <Symbol>PRESSURE_SIGNAL</Symbol>
         <Type> <Value>3</Value> <Name>Type_4_20mA</Name> </Type>
                                    ...
    <MB> <Index>1</Index> <Symbol>SOLENOID_ON</Symbol>
    <Comment>Solenoid Open Command</Comment> </MB>
    <MB> <Index>2</Index> <Symbol>PUMP_ON</Symbol>
    <Comment>Air Pump Run Command</Comment> </MB>
                                    ...
    <MB> <Index>5</Index> <Symbol>FORCE_ON</Symbol>
    <Comment>1</Comment> </MB>
                                    ...
    <MB> <Index>10</Index> <Symbol>FIRST_SCAN</Symbol> </MB>
                                    ...
    <MW> <Index>0</Index> <Symbol>HMI_PRESS_SP_LO</Symbol>
    <Comment>Low Pressure Setpoint set from HMI</Comment> </MW>
    <MW> <Index>1</Index> <Symbol>HMI_PRESS_SP_HI</Symbol>
    <Comment>High Pressure Setpoint set from HMI</Comment> </MW>
    <MW> <Index>2</Index> <Symbol>HMI_PRESS_SP_HIHI</Symbol>
    <Comment>High-High Pressure Setpoint set from HMI</Comment> </MW>
                                    ...
    <T> <Index>0</Index> <Symbol>AIR_BLEED_TIMER</Symbol>
    <Comment>Air Bleed Timer</Comment> <Type>TP</Type>
    <Preset>5</Preset> <Base>OneSecond</Base> </T>
                                    ...
```

Figure 4.13: Decompressed zip file contents

%Q0.1 (the PLC's output port 1) connected to the solenoid valve, only if either %M1 or %TM0.DN (the DN bit of Timer 0) is set.

Other Information Extracted We have extracted ASCII strings from the memory dump using the GNU `strings` tool. In particular, from the on-chip RAM region (in-memory firmware), we have identified some network information of the PLC (i.e., IP address, subnet mask, gateway's IP, MAC address, local DNS server) along with the PLC model name, the firmware version, and the project name of control logic. From the XML file, we have found user information (who may have written the control logic), including last and first names

and a phone number.

4.2.5 Performance Evaluation

Scan time To evaluate how PEM affects a PLC's regular operation, we measured the PLC's scan time. In a clean state, the PLC's scan time in controlling the gas pipeline process is between 331-333µs . When PEM appends a duplicator with 64KB of block size to the control logic, the scan time is increased up to 30.1ms. However, the gas pipeline system still works well without any noticeable difference. Since the time required in mechanical operations of the valve and the air compressor is dominant, about 29.8ms of overhead in the PLC's processing time can be negligible. However, we can try to reduce the overhead if a physical process is very time-critical. We can get 674-675µs of scan time when we reduce the block size of a duplicator to 256 bytes. Note that the total number of duplicators increases as their block sizes decrease.

Memory acquisition time We measure the elapsed time for acquiring memory over a network. The machine running PEM is a 64-bit Ubuntu 20.04 VM with two vCPU cores (i7-6920HQ/2.90GHz) and 4 GB RAM. The average acquisition time is 2.03 seconds per a 64 KB of memory block.

4.3 Case Study of ROP Attacks on Physical Processes

We utilized a similar gas pipeline model as in 4.2. The ladder logic implemented in gas pipeline utilizes comparison operator. The comparison operator ensures that the gas pressure remains in the ideal threshold region. If the pressure in the pipeline reduces to the lower threshold it opens the valve and if the threshold exceeds the higher threshold, it closes the valve and vice versa. As depicted in Figure 13, the comparison operator either leaves the circuit open or close depending on the current pressure. When the pressure exceeds, it leaves the circuit open and it turns off the OTE 4.4, which portrays a valve.

Testing ROP 1: In our attack we used ROP 1 (that uses just one gadget) which renders the comparison useless by setting the desired output value, in this case %Q0.2 to always on. In this case study, we observe this attack performed successfully and had to turn off the PLC to prevent any damage to gas pipeline.

Scan time and program size In a clean state, the PLC's scan time in controlling the gas pipeline process is 331-333µs. When we append SPMC which in this case is 11 bytes. The original control logic is 343 bytes in size so a small increase of 11 bytes does not affect the scan time.

Number and size of gadgets : We use one gadget that is 3 bytes in size for this case study.

Chapter 5

Conclusion

This work describes three different approaches that have detrimental effect on PLC.It talks about denial-of-engineering-operation (DEO) attacks. Instead of relying on implementation vulnerabilities (e.g., improper input validation), the proposed approach exploits a fundamental design principle in compiling and decompiling the control logic of a PLC by control-logic obfuscation, making our method more generally applicable. We have evaluated our approach on two major vendors' PLCs. Experiments using instruction-level obfuscation have shown that our approach is effective. In addition, we have conducted a separate experiment to see whether control-logic obfuscation can evade existing ML-based control-logic detection. The experiment result indicates that control-logic obfuscation is also effective in avoiding detection when transferred over the network. Based on the findings in this study, we argue that current ICS forensic capabilities relying on engineering software are incomplete, and the ICS community needs to develop more robust strategies and tools to respond to cyberattacks employing control-logic obfuscation.

Secondly we talk about PEM, a remote memory acquisition framework for PLCs, which can extract the entire memory over a network while a target PLC is controlling a physical process. We also present a new control-logic attack that remotely modifies in-memory firmware to hijack a PLC's built-in system functions. The effectiveness of PEM in investi-

gating the control-logic attack is demonstrated by a case study over the Schneider Electric Modicon M221 PLC installed in a gas pipeline testbed.

Lastly we talk about ROP attacks, a malicious attack which uses control of the call stack to execute sophisticated machine instruction. This eliminates the requirement for direct code injection because every instruction that is executed comes from executable memory regions within the original program.

Bibliography

[1] Ali Abbasi et al. "Challenges in Designing Exploit Mitigations for Deeply Embedded Systems". In: *Proceedings of 2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 31–46.

[2] I. Ahmed et al. "Programmable Logic Controller Forensics". In: *IEEE Security Privacy* 15.6 (Nov. 2017), pp. 18–24. ISSN: 1558-4046.

[3] I. Ahmed et al. "SCADA Systems: Challenges for Forensic Investigators". In: *Computer* 45.12 (Dec. 2012), pp. 44–51. ISSN: 0018-9162.

[4] I. Ahmed et al. "SCADA Systems: Challenges for Forensic Investigators". In: *Computer* 45.12 (Dec. 2012), pp. 44–51. ISSN: 0018-9162. DOI: 10.1109/MC.2012.325.

[5] Irfan Ahmed et al. "A SCADA System Testbed for Cybersecurity and Forensic Research and Pedagogy". In: *Proceedings of the 2nd Annual Industrial Control System Security (ICSS) Workshop*. 2016, pp. 1–9.

[6] Irfan Ahmed et al. "Programmable Logic Controller Forensics". In: *IEEE Security Privacy* 15.6 (Nov. 2017), pp. 18–24. ISSN: 1540-7993. DOI: 10.1109/MSP.2017.4251102.

[7] Adeen Ayub, Hyunguk Yoo, and Irfan Ahmed. "Empirical Study of PLC Authentication Protocols in Industrial Control Systems". In: *Proceedings of the 15th IEEE Workshop on Offensive Technologies (WOOT)*. IEEE, 2021.

[8] Zachry Basnight et al. “Firmware modification attacks on programmable logic controllers”. In: *International Journal of Critical Infrastructure Protection* 6.2 (2013), pp. 76–84.

[9] Nicholas Carlini and David Wagner. “{ROP} is still dangerous: Breaking modern defenses”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 385–399.

[10] Nicholas Carlini and David Wagner. “ROP is Still Dangerous: Breaking Modern Defenses”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 385–399. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[11] Andrew Case and Golden G Richard III. “Memory Forensics: The Path Forward”. In: *Digital Investigation* 20 (2017), pp. 23–33.

[12] Abraham A Clements et al. “Protecting Bare-Metal Embedded Systems with Privilege Overlays”. In: *Proceedings of 2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 289–303.

[13] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. “Discoverer: Automatic Protocol Reverse Engineering from Network Traces.” In: *Proceedings of 2007 USENIX Security Symposium*. 2007, pp. 1–14.

[14] Weidong Cui et al. “Tupni: Automatic reverse engineering of input formats”. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. 2008, pp. 391–402.

[15] *CVE-2017-14466*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14466. Online accessed May-2020. 2017.

[16] George Denton et al. “Leveraging the SRTP Protocol for Over-the-Network Memory Acquisition of a GE Fanuc Series 90-30”. In: *Digital Investigation* 22 (2017), S26–S38.

[17] Alessandro Di Pinto, Younes Dragoni, and Andrea Carcano. "TRITON: The First ICS Cyber Attack on Safety Instrument Systems". In: *Proceedings of 2018 Black Hat USA*. 2018.

[18] Nicolas Falliere, Liam O Murchu, and Eric Chien. "W32.stuxnet dossier". In: *White paper, Symantec Corp., Security Response* 5.6 (2011), p. 29.

[19] Nicolas Falliere, Liam O Murchu, and Eric Chien. *W32.Stuxnet Dossier Version 1.4*. Tech. rep. 6. Symantec, 2011, p. 29.

[20] Luis Garcia et al. "Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit". In: *Network and Distributed System Security Symposium (NDSS)*. 2017.

[21] Luis Garcia et al. "Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit." In: *NDSS*. 2017.

[22] Andrei Homescu et al. "Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming." In: *WOOT* 12 (2012), pp. 64–76.

[23] *ICS Advisory (ICSA-18-240-01)-Modicon M221*. `https://ics-cert.us-cert.gov/advisories/ICSA-18-240-01`. Online accessed May-2020. 2018.

[24] *ICS Advisory (ICSA-20-070-06) Rockwell Automation MicroLogix Controllers and RSLogix 500 Software*. `https://www.us-cert.gov/ics/advisories/icsa-20-070-06`. Online accessed May-2020. 2020.

[25] Georges-Axel Jaloyan et al. "Return-oriented programming on RISC-V". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 471–480.

[26] Sushma Kalle et al. "CLIK on PLCs! Attacking control logic with decompilation and virtual PLC". In: *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*. 2019.

[27] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. "Certified malware: Measuring breaches of trust in the windows code-signing pki". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1435–1448.

[28] Platon Kotzias et al. "Certified PUP: abuse in authenticode code signing". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 465–478.

[29] Nishchal Singh Kush et al. "Gap analysis of intrusion detection in smart grids". In: *Proceedings of the 2nd International Cyber Resilience Conference*. Australia: secau-Security Research Centre, 2011, pp. 38–46.

[30] Per Larsen et al. "SoK: Automated software diversity". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 276–291.

[31] Robert M Lee, Michael J Assante, and Tim Conway. *Analysis of the Cyber Attack on the Ukrainian Power Grid*. Tech. rep. SANS Industrial Control Systems, 2016.

[32] Stephen McLaughlin and Patrick McDaniel. "SABOT: Specification-based payload generation for programmable logic controllers". In: *Proceedings of 2012 ACM Conference on Computer and Communications Security (CCS)*. 2012, pp. 439–449.

[33] Stephen E McLaughlin. "On Dynamic Malware Payloads Aimed at Programmable Logic Controllers." In: *HotSec*. 2011.

[34] Syed Qasim et al. "Attacking IEC-61131 Logic Engine in Programmable Logic Controllers". In: *Proceedings of the 15th Annual IFIP WG 11.10 International Conference on Critical Infrastructure Protection*. Springer. 2021.

[35] Muhammad Haris Rais et al. "JTAG-based PLC memory acquisition framework for industrial control systems". In: *Forensic Science International: Digital Investigation* 37 (2021), p. 301196.

[36] *Renesas GNU Tools*. `https://gcc-renesas.com/`. [Online; accessed 09-May-2021]. 2021.

[37] Saranyan Senthivel, Irfan Ahmed, and Vassil Roussev. "SCADA Network Forensics of the PCCC Protocol". In: *Digital Investigation* 22 (2017), S57–S65.

[38] Saranyan Senthivel et al. "Denial of Engineering Operations Attacks in Industrial Control Systems". In: *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2018, pp. 319–329.

[39] Saranyan Senthivel et al. "Denial of Engineering Operations Attacks in Industrial Control Systems". In: *Proceeding of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2018. ISBN: 978-1-4503-5632-9.

[40] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. URL: https://doi.org/10.1145/1315245.1315313.

[41] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)". In: *Proceedings of CCS 2007*. Ed. by Sabrina De Capitani di Vimercati and Paul Syverson. ACM Press, Oct. 2007, pp. 552–61.

[42] Ruimin Sun et al. "SoK: Attacks on Industrial Control Logic and Formal Verification-Based Defenses". In: *Proceedings of 2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021.

[43] *TRISIS Malware: Analysis of Safety System Targeted Malware*. https://dragos.com/blog/trisis/TRISIS-01.pdf. [Online; accessed 25-May-2020]. 2017.

[44] Nathanael R. Weidler et al. "Return-oriented programming on a resource constrained device". In: *Sustainable Computing: Informatics and Systems* 22 (2019), pp. 244–256. ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2018.10.002. URL: https://www.sciencedirect.com/science/article/pii/S2210537917303931.

[45] Nathanael R. Weidler et al. “Return-oriented programming on a resource constrained device”. In: *Sustainable Computing: Informatics and Systems* 22 (2019), pp. 244–256. ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2018.10.002. URL: https://www.sciencedirect.com/science/article/pii/S2210537917303931.

[46] Svein Willassen. “Forensic analysis of mobile phone internal memory”. In: *IFIP International Conference on Digital Forensics*. Springer. 2005, pp. 191–204.

[47] Tina Wu and Jason RC Nurse. “Exploring the Use of PLC Debugging Tools for Digital Forensic Investigations on SCADA Systems”. In: *Journal of Digital Forensics, Security and Law* 10.4 (2015), p. 7.

[48] Hyunguk Yoo and Irfan Ahmed. “Control logic injection attacks on industrial control systems”. In: *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer. 2019, pp. 33–48.

[49] Hyunguk Yoo et al. “Overshadow PLC to detect remote control-logic injection attacks”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2019, pp. 109–132.

Vita

Nauman Zubair was born in Karachi, Pakistan. He came to US to obtain his bachelor's in electrical engineering. During his time in undergrad, he competed in IEEE robotics region 5 competition. He was able to achieve first place. That is where his interest of programming stemmed from. He decided to obtain PhD in computer Science in Fall 2018. He has been doing research under Dr Hyunguk Yoo in the cybersecurity lab. His research focus has been on Digital forensics of Programmable Logic Controllers(PLC), where he found security loopholes in PLC architecture which also contributed to his thesis. He aspires to work as a Software Engineer focusing on security and development.