Forensic Science International: Digital Investigation 40 (2022) 301336

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2022 EU - Selected Papers of the Ninth Annual DFRWS Europe Conference

PEM: Remote forensic acquisition of PLC memory in industrial control systems

Nauman Zubair [a], Adeen Ayub [b], Hyunguk Yoo [a, *], Irfan Ahmed [b, **]

[a] University of New Orleans, New Orleans, LA, 70148, USA
[b] Virginia Commonwealth University, Richmond, VA, 23284, USA

ARTICLE INFO

Article history:

Keywords:
Programmable logic controllers
Industrial control systems
Control-logic attacks
Memory forensics

ABSTRACT

Programmable logic controllers (PLC) are special-purpose embedded devices used in various industries for automatic control of physical processes. Cyberattacks on PLCs can unleash mayhem in the physical world. In case of a security breach, volatile memory acquisition is critical in investigating the attack since it provides unique insights into the runtime system activities and memory-based artifacts. However, existing memory acquisition methods for PLCs (i.e., using a hardware-level debugging port and network protocol-based approaches) are either inapplicable in real-world forensic investigations (due to requiring disassembling of a suspect PLC or power cycling) or incomplete (i.e., acquire only partial memory contents). This paper proposes a new memory acquisition framework to remotely acquire a PLC's volatile memory while the PLC is controlling a physical process. The main idea is to inject a harmless memory duplicator into the running control logic of a PLC to copy local memory contents into a protocol-mapped address space, which is then readable over a network. We also present a new control-logic attack that targets in-memory firmware to compromise a PLC's built-in system functions. Since PEM can acquire the entire PLC memory, we show that its memory dump contains evidence of this attack. Further, we present a case study on a gas pipeline testbed to demonstrate the effectiveness of the attack on a physical process and how PEM plays its role in effectively identifying the attack and other important forensic artifacts such as the control logic of a PLC.

1. Introduction

Industrial control systems (ICS) are used to control physical processes in critical infrastructures such as power grids, nuclear facilities, and gas pipelines (Ahmed et al., 2016; Kush et al., 2011). They have controllers at field sites that are connected with physical processes through sensors and actuators. Programmable logic controllers (PLC) are among the most commonly used controller types, which criminals and state-backed actors often target (Falliere et al., 2011; M Lee et al., 2016; Di Pinto et al., 2018; Ayub et al., 2021; Qasim et al., 2021). In ICS security incidents, forensic investigation on suspect PLCs is crucial to answering many questions about cyberattacks (Ahmed et al., 2012, 2017).

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 (Case and Richard, 2017). Current approaches in acquiring a PLC's volatile memory mostly use an ICS protocol to read memory over a network (Wu and Nurse, 2015; Senthivel et al., 2017; Denton et al., 2017; Yoo et al., 2019). 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 (Rais et al., 2021). 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 paper 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

* Corresponding author.
** Corresponding author.
E-mail addresses: hyoo1@uno.edu (H. Yoo), iahmed3@vcu.edu (I. Ahmed).

https://doi.org/10.1016/j.fsidi.2022.301336

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 study 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.

The rest of the paper is organized as follows. Section 2 provides the background of this study. Section 3 presents PEM in detail, followed by Section 4, which describes a new control-logic attack. Section 5 shows a case study on a gas pipeline testbed. Finally, Section 6 concludes the paper.

2. Background & related work

2.1. Programmable logic controllers

Programmable logic controllers (PLC) are special-purpose computers designed for automatic control of physical processes (Qasim et al., 2019, 2020). They have dozens to thousands of digital/analog inputs and outputs wired with sensors and actuators. Although PLCs are often used in larger control systems (e.g., SCADA and DCS) as local controller components, they can also be used as stand-alone controllers in smaller control system configurations.

The *control logic* of a PLC (a.k.a. a PLC program), which defines how a PLC controls a physical process, can be programmed using *engineering software* running on an *engineering workstation* (typically located at a control center). IEC 61131-3 defines five PLC programming languages: ladder diagram (LD), function block diagram (FBD), structured text (ST), instruction list (IL), and sequential function chart (SFC). They are domain-specific languages and hence have specialized features for the industrial control domain. For example, LD represents control logic through graphical diagrams that resemble the circuit diagrams of traditional relays.

Once compiled and downloaded into a PLC, control logic runs in a *scan cycle* consisting of three main steps. First, the PLC reads inputs through connected input devices (e.g., sensors, switches) and updates an input image table in memory. Then, the control logic runs on the input image table, modifying an output image table based on its execution results. Finally, the PLC controls connected output devices (e.g., actuators, lights) by developing output signals according to the output image table. The period of each scan cycle, called *scan time*, is affected by a PLC's computing power and the complexity of control logic[1]. It ranges from a few to hundreds of milliseconds in typical industrial settings.

2.2. PLC memory acquisition

Existing methods to acquire a PLC's volatile memory can be categorized into the following three approaches.

Built-in support. Some PLCs have a built-in feature that allows a user to acquire volatile memory (mainly for debugging purposes) by dumping volatile memory to non-volatile memory (e.g., flash memory) when they crash. However, such a feature is not generally supported. Moreover, it is not capable of dumping memory while a PLC is controlling a physical process (Ahmed et al., 2017), which makes this approach inapplicable in the situations where a PLC under investigation is the only controller connected to a physical process (i.e., no secondary backup PLC is available).

Debug port. A debug port such as JTAG can be used to read the volatile memory of an embedded device (Ahmed et al., 2017; Rais et al., 2021). In this approach, a forensic investigator connects an in-circuit debugger (e.g., SEGGER J-Link (SEGGER J-Link, 2021)) to the JTAG pins (i.e., TCK, TMS, TDI, TDO, and optional TRST) on a printed circuit board (PCB) to access volatile memory (Awad et al., 2019). It is possible to acquire the entire memory, given full implementation of JTAG, although it can be slow. However, most manufacturers hide or even physically remove the JTAG pins from a PCB at the end of the production process (Rais et al., 2021). Moreover, this approach entails hardware interference (e.g., reassembling, soldering, power cycling) to install an in-circuit debugger to JTAG pins, meaning volatile-memory data will be lost during the installation. Therefore, it is not a practical approach to be used in forensic investigations unless in-circuit debuggers are installed to PLCs in advance before a security incident.

ICS communication protocol. Most current forensic efforts focus on using ICS protocols to acquire a PLC's volatile memory (Wu and Nurse, 2015; Senthivel et al., 2017; Denton et al., 2017). ICS protocols (e.g., Modbus, DNP3, IEC 61850, PCCC (Senthivel et al., 2017), S7Comm (Biham et al., 2019)) support various functions that allow a control center application (e.g., engineering software, HMI) to command and monitor controllers at a remote field site over a network. Among the most basic functions are to read and write the memory of a controller. However, this approach is usually unable to acquire the entire memory since ICS protocols can access only limited memory areas that are mapped to the protocols' address space.

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.

3.1. 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

[1] Some PLCs support an optional minimum scan cycle to eliminate the variation of the time period. If this feature is enabled, the PLC may delay until the minimum scan cycle elapses before running the following scan.

that is not mapped to an ICS protocol's address space (we will discuss this kind of attack in Section 4).

In this paper, 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.

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) (Abbasi et al., 2019). 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 (Di Pinto et al., 2018; Clements et al., 2017). 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.

3.2. The design of PEM

Fig. 1 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 (Rais et al., 2021) 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 (Cui et al., 2007, 2008) 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 5, 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` (Renesasools GNU Tools, 2021)). 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

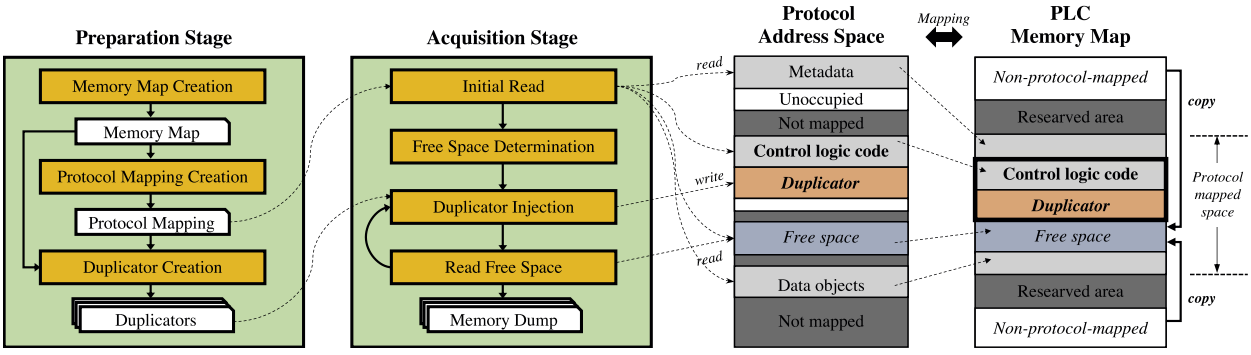


Fig. 1. The design overview of PEM.

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.

4. 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.

4.1. Existing attacks

Control-logic modification attacks (or control-logic attacks) are the attacks that change the way a PLC controls a physical process (Sun et al., 2021). Existing control-logic attacks (Falliere et al., 2011; Kalle et al., 2019; Senthivel et al., 2018; Yoo and Ahmed, 2019; McLaughlin, 2011; McLaughlin and McDaniel, 2012) 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 (Basnight et al., 2013; Garcia et al., 2017) 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. (2017) 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.

4.2. 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.

4.3. 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. Fig. 2 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

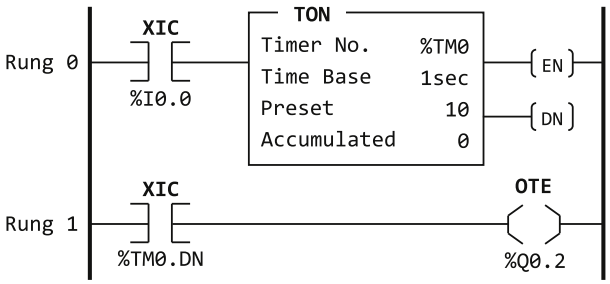
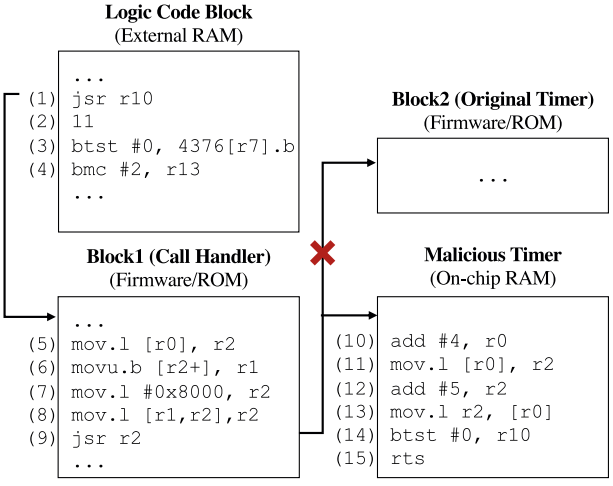


Fig. 2. Example ladder logic diagram with timer.

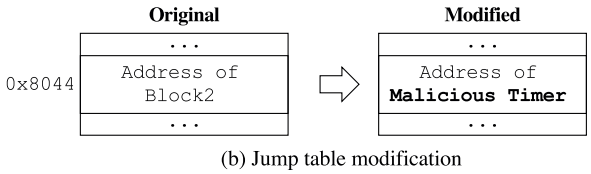
evaluation result (either true or false) of a rung's logical expression (consisting of input instructions) affects the behaviors of the rung's output instructions.

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 1 s and its preset is 10, it counts for 10 s. 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 Fig. 3(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 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



(a) Redirection to malicious timer



(b) Jump table modification

Fig. 3. Hijacking a PLC timer function.

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:

$$Z\ (\textit{Zero flag}) = \sim(([r7+4376] \gg 0)\ \&\ 1)$$

$$C\ (\textit{Carry flag}) = (([r7+4376] \gg 0)\ \&\ 1)$$

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.

Fig. 3(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 (Fig. 4 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.

Experiment 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.

5. 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

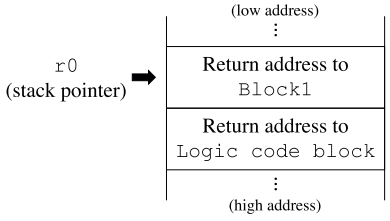


Fig. 4. The runtime stack.

memory and analyze it to identify the attack and other important information. Further, we evaluate the performance of PEM in the testbed environment.

5.1. 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. Fig. 5 shows a top-view 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 (Kalle et al., 2019; Yoo and Ahmed, 2019). Lastly, the firmware version used is v1.6.0.1.

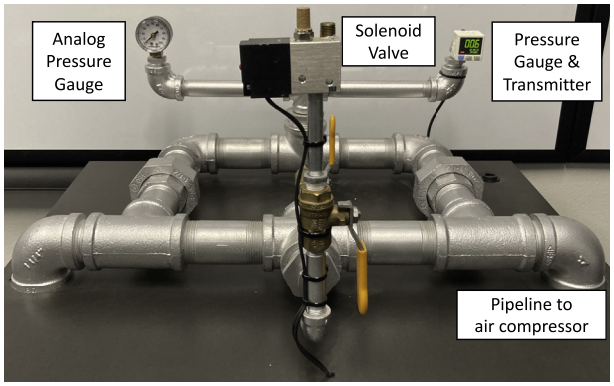


Fig. 5. A top-view of the gas pipeline testbed.

5.2. Control-logic attack scenario

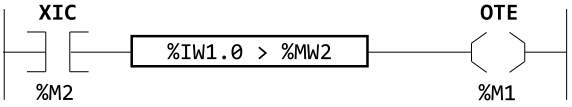
We present a targeted attack specially designed for a gas pipeline system. We assume that the attacker knows the target PLC model and firmware version so that she can prepare a pre-compiled malicious code before the attack.

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 4. If a comparison operator in control logic does not produce correct answers, the control actions from the logic execution would be wrong.

Fig. 6(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[2]. If it is true, %M1 is energized. Fig. 6(b) represents the compiled binary code. Line 1 tests bit 2 at the address [R7 + 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 4, 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



(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

Fig. 6. A ladder logic diagram and its binary code with a comparison operator.

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

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 from 0x1EA80 to 0x1EC2F.

Fig. 7 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 Fig. 6(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.

5.3. 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 1 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.

Fig. 8 shows the 49-byte size of binary code of a duplicator, which copies 64 KB 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` (Renesasools GNU Tools, 2021).

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.

5.4. 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.

Fig. 9 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.

```
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
```

Fig. 7. The binary code of a malicious '>' operator.

Table 1
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

```
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
```

Fig. 8. A duplicator of PEM.

```
(address)                ...
00008080: b4fd f3ff d58c f5ff 7aef f3ff 67f4 f3ff
00008090: 33f4 f3ff 8cfd f3ff 64fd f3ff 00eb 0100
000080a0: 14fd f3ff c6fc f3ff edfc f3ff 7efa f3ff
                         ...   Corrupted jump table entry
```

Fig. 9. The corrupted jump table.

However, existing studies (Senthivel et al., 2018; Yoo and Ahmed, 2019) 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. Fig. 10 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 (Kalle et al., 2019). Fig. 11 shows a portion of the memory dump where the logic code is found.

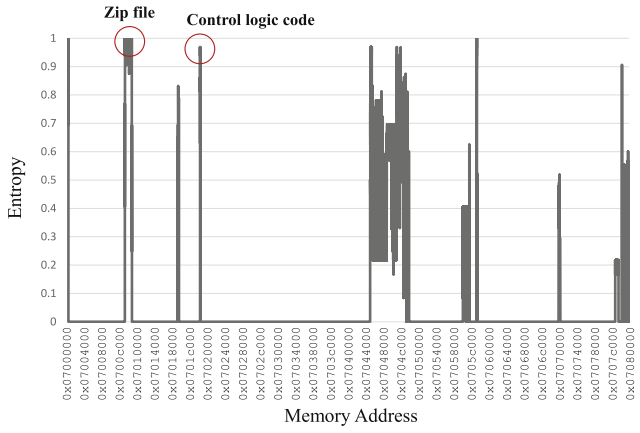


Fig. 10. Entropy analysis on memory dump.

Decompilation of the control logic. Fig. 12 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 Fig. 13), we can successfully extract the zip file and decompress it into an XML file.

We find that the XML file describes the semantics of data objects used in the control logic (refer to Fig. 14). 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 Fig. 12) 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 and 20 mA 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 %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.

5.5. 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 µs and 333 µs. When PEM appends a duplicator with 64 KB of block size to the control logic, the scan time is increased up to 30.1 ms.

```
...                                  Start of control logic code
0x0701e260: 0000 0000 0000 0000 0000 0000 7fa0 fcea
0x0701e270: 7201 00f6 7281 0023 0d7f 1a06 1a1a 0481
...
0x0701e3b0: f672 0c00 7f1a 2716 14b4 9104 81fc fa72
0x0701e3c0: 0c00 0200 0000 0000 0000 0000 0000 0000
...                                  End of control logic code
                                     0x02: RX rts(return) instruction
```

Fig. 11. Control logic code in the memory dump.

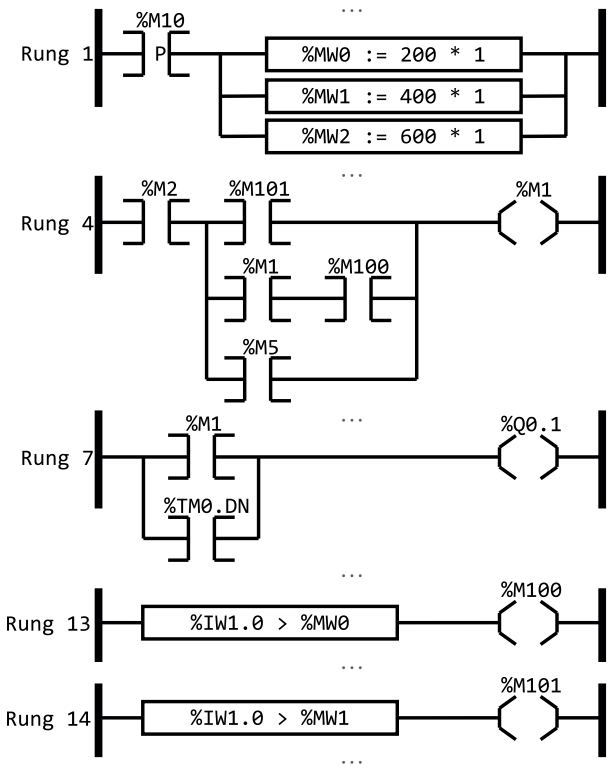


Fig. 12. Decompiled control logic in ladder logic diagram.

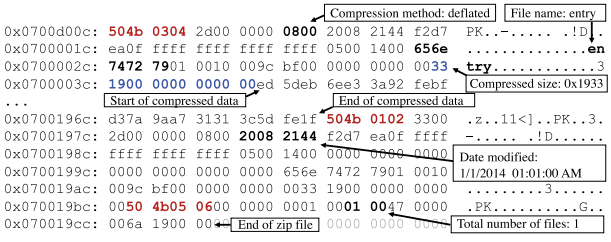


Fig. 13. A zip file in the memory dump.

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 29.8 ms 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 μs–675 μs 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/ 2.90 GHz) and 4 GB RAM. The average acquisition time is 2.03 s per a 64 KB of memory block.

6. Conclusion

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

```
<?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>
                                ...
```

Fig. 14. Decompressed zip file contents.

Acknowledgement

This work was supported, in part, by the Virginia Commonwealth Cyber Initiative, an investment in the advancement of cyber R&D, innovation, and workforce development. For more information, visit www.cyberinitiative.org. In addition, this project has received funding from the Louisiana Board of Regents' Research Competitiveness Subprogram (RCS) under contract number LEQSF(2021-22)-RD-A-27.

References

Abbasi, A., Wetzels, J., Holz, T., Etalle, S., 2019. Challenges in designing exploit mitigations for deeply embedded systems. In: Proceedings of 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 31–46.

Ahmed, I., Obermeier, S., Naedele, M., Richard III, G.G., 2012. SCADA systems: challenges for forensic investigators. Computer 45 (12), 44–51.

Ahmed, I., Roussev, V., Johnson, W., Senthivel, S., Sudhakaran, S., 2016. A SCADA system testbed for cybersecurity and forensic research and pedagogy. In: Proceedings of the 2nd Annual Industrial Control System Security (ICSS) Workshop, pp. 1–9.

Ahmed, I., Obermeier, S., Sudhakaran, S., Roussev, V., 2017. Programmable logic controller forensics. IEEE Secur. Priv. 15 (6), 18–24.

Awad, R.A., Lopez, J., Rogers, M., 2019. Volatile memory extraction-based approach for level 0-1 CPS forensics. In: Proceedings of 2019 IEEE International Symposium on Technologies for Homeland Security. IEEE, pp. 1–6.

Ayub, A., Yoo, H., Ahmed, I., 2021. Empirical study of PLC authentication protocols in industrial control systems. In: Proceedings of the 15th IEEE Workshop on Offensive Technologies (WOOT). IEEE.

Basnight, Z., Butts, J., Lopez Jr., J., Dube, T., 2013. Firmware modification attacks on programmable logic controllers. Int. J. Crit. Infrastruct. Protect. 6 (2), 76–84.

Biham, E., Bitan, S., Carmel, A., Dankner, A., Malin, U., Wool, A., 2019. Rogue7: rogue engineering-station attacks on S7 simatic PLCs. In: Proceedings of 2019 Black. Hat USA.

Case, A., Richard III, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.

Clements, A.A., Almakhdhub, N.S., Saab, K.S., Srivastava, P., Koo, J., Bagchi, S., Payer, M., 2017. Protecting bare-metal embedded systems with privilege overlays. In: Proceedings of 2017 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 289–303.

Cui, W., Kannan, J., Wang, H.J., 2007. Discoverer: automatic protocol reverse engineering from network traces. In: Proceedings of 2007 USENIX Security Symposium, pp. 1–14.

Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L., 2008. Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. CCS), pp. 391–402.
Denton, G., Karpisek, F., Breitinger, F., Baggili, I., 2017. Leveraging the SRTP protocol for over-the-network memory acquisition of a GE Fanuc Series 90-30. Digit. Invest. 22, S26–S38.
Di Pinto, A., Dragoni, Y., Carcano, A., 2018. TRITON: the first ICS cyber attack on safety instrument systems. In: Proceedings of 2018 Black Hat USA.
Falliere, N., Murchu, L.O., Chien, E., 2011. Tech. Rep. W32.Stuxnet Dossier Version 1.4, 6. Symantec.
Garcia, L., Brasser, F., Cintuglu, M.H., Sadeghi, A.-R., Mohammed, O.A., Zonouz, S.A., 2017. Hey, my malware knows physics! Attacking PLCs with physical model aware rootkit. In: Proceedings of 2017 Network and Distributed System Security Symposium. NDSS).
Kalle, S., Ameen, N., Yoo, H., Ahmed, I., 2019. CLIK on PLCs! Attacking control logic with decompilation and virtual PLC. In: Proceedings of 2019 NDSS Workshop on Binary Analysis Research. BAR).
Kush, N.S., Foo, E., Ahmed, E., Ahmed, I., Clark, A., 2011. Gap analysis of intrusion detection in smart grids. In: Proceedings of the 2nd International Cyber Resilience Conference. secau-Security Research Centre, Australia, pp. 38–46.
Lee, R.M., Assante, M.J., Conway, T., 2016. Analysis of the Cyber Attack on the Ukrainian Power Grid. Tech. rep. SANS Industrial Control Systems.
McLaughlin, S.E., 2011. On dynamic malware payloads aimed at programmable logic controllers. In: Proceedings of the 6th Usenix Workshop on Hot Topics in Security. HotSec).
McLaughlin, S., McDaniel, P., 2012. SABOT: specification-based payload generation for programmable logic controllers. In: Proceedings of 2012 ACM Conference on Computer and Communications Security. CCS), pp. 439–449.
Qasim, S.A., Lopez, J., Ahmed, I., 2019. Automated reconstruction of control logic for programmable logic controller forensics. In: Proceedings of 2019 International Conference on Information Security. Springer, pp. 402–422.
Qasim, S.A., Smith, J.M., Ahmed, I., 2020. Control logic forensics framework using built-in decompiler of engineering software in industrial control systems. Forensic Sci. Int.: Digit. Invest. 33, 301013.
Qasim, S., Ayub, A., Johnson, J., Ahmed, I., 2021. 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.
Rais, M.H., Awad, R.A., Lopez Jr., J., Ahmed, I., 2021. JTAG-based PLC memory acquisition framework for industrial control systems. Forensic Sci. Int.: Digit. Invest. 37, 301196.
Renesas GNU Tools, 2021. https://gcc-renesas.com/. (Accessed 9 May 2021).
SEGGER J-Link, 2021. https://www.segger.com/products/debug-probes/j-link/. (Accessed 9 May 2021).
Senthivel, S., Ahmed, I., Roussev, V., 2017. SCADA network forensics of the PCCC protocol. Digit. Invest. 22, S57–S65.
Senthivel, S., Dhungana, S., Yoo, H., Ahmed, I., Roussev, V., 2018. Denial of engineering operations attacks in industrial control systems. In: Proceedings of the 8th ACM Conference on Data and Application Security and Privacy. CODASPY), pp. 319–329.
Sun, R., Mera, A., Lu, L., Choffnes, D., 2021. 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.
Wu, T., Nurse, J.R., 2015. Exploring the use of PLC debugging tools for digital forensic investigations on SCADA systems. J. Digital Forensics, Secur. Law 10 (4), 7.
Yoo, H., Ahmed, I., 2019. Control logic injection attacks on industrial control systems. In: Proceedings of the 34th IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC). Springer, pp. 33–48.
Yoo, H., Kalle, S., Smith, J., Ahmed, I., 2019. Overshadow PLC to detect remote control-logic injection attacks. In: Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). Springer, pp. 109–132.