On Dynamic Malware Payloads Aimed at Programmable Logic Controllers

Stephen McLaughlin
smclaugh@cse.psu.edu

Abstract

With the discovery of the Stuxnet attack, increasing attention is being paid to the potential for malware to target Programmable Logic Controllers (PLCs). Despite much speculation about threats from PLC malware, the popular opinion is that automated attacks against PLCs are not practical without having *a priori* knowledge of the target physical process. In this paper, we explore the problem of designing PLC malware that can generate a *dynamic payload* based on observations of the process taken from inside the control system. This significantly lowers the bar for attacks against PLCs. We evaluate how PLC malware may infer the structure of the physical plant and how it can use this information to construct a dynamic payload to achieve an adversary's end goal. We find that at the very least, a dynamic payload can be constructed that causes unsafe behavior for an arbitrary process definition.

1 Introduction

A process control system is a computerized means for regulating the behavior of a physical process called the *plant*. Examples of such processes are the production of industrial goods, the delivery of electrical power, and the automation of transportation infrastructure. While control systems come in several forms such as Supervisory Control and Data Acquisition (SCADA) [20] and Distributed Control Systems (DCS), they virtually all use programmable logic controllers (PLCs) (or in some cases automation controllers) to interface directly with the physical equipment contained in the plant.

It is well understood that PLCs are computers, and thus are susceptible to the same classes of attacks as traditional IT systems [4]. The ramifications of these attacks are however different as PLCs are responsible for the correct and safe operation of physical processes. With the discovery of the Stuxnet attack [21], increasing attention is being paid to the potential for malicious code to be uploaded directly to PLCs, giving an adversary control over the physical process. As there has been much speculation about what malware may be capable of doing with PLCs [9], we aim to refine the notions and problems involved in this new frontier for malicious code.

In this paper, we consider the ways in which a PLC may be attacked directly by an autonomous program. Of main concern is the ability of such a program to generate malicious payload (PLC code) that disrupts the process, endangering workers and incurring financial loss. To do this, the malware first gathers clues from within the control system regarding the nature of the process, the layout of the physical plant, or both. These clues are then used to generate a payload that can be uploaded to the PLC and executed. We will describe several such types of clues that reveal how the process is intended to work, what types of equipment are used in the plant, and what operations may be considered unsafe.

It is assumed that PLC malware must act autonomously, without human assistance, for several reasons. First, as was the case with the Stuxnet attack, an air gap may exist between public networks and the target PLC. Such a gap must be spanned either by propagation through removal storage such a USB drive [6], or with the aid of an insider. We note that for non-critical facilities, this is less of an issue as Internet-enabled PLCs are now entering the market [18, 5]. A second reason that PLC malware must act autonomously is that it significantly reduces the sophistication needed on the part of its author. By assuming that PLC malware may be written in a generic form by a few skilled individuals and widely disseminated to unskilled insiders or script kiddies, we obtain a stronger adversary model for when we later consider mitigations against PLC malware. Finally, as will be shown later, one scenario for the use of PLC malware is the indiscriminate attacking of any PLC onto which it happens to propagate. This obviously cannot be done with any specific goal in mind, and thus an attack must be derived on the fly.

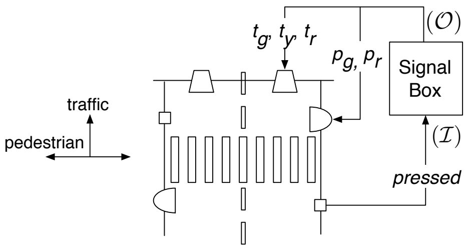

Figure 1: Pedestrian crossing with signal box inputs and outputs labeled.

The problem of autonomously generating a PLC payload is broken down into several steps. First, the malware author must specify a goal to be carried out in one or more target control systems. Once the PLC malware is inside the control system, it must at least read the PLC's code and data memory to obtain clues about the process structure and operations. Examples of such clues that will be covered later are the process fieldbus IDs of devices in the plant and the safety interlocks that prevent the process from entering unsafe states. If these clues are sufficient to carry out the goal, a payload is generated and executed on the PLC. Each of these steps will now be detailed after a summary of PLC functionality.

2 Logic Controllers

PLCs are the real-time systems that closely monitor and control plant devices to keep the process functioning correctly. Program execution on PLCs differs substantially from on general-purpose computers. A PLC program, often referred to as the *logic*, is executed within a loop many times per second. Each execution is called a *scan cycle*. During each scan cycle, a set of input variables I are read from the sensors in the plant and processed by the logic to produce a set of output variables O that dictate the behavior of each physical device. The logic may also maintain a set of internal state variables $\mathcal{C}$ and a set of timer variables $\mathcal{T}$. Often, the addresses referenced by variables reveal which class it belongs to. In Siemens S7 PLCs for example, separate memory areas are used for I, O, $\mathcal{C}$, and $\mathcal{T}$.

While PLCs may be programmed in a number of different languages, e.g. relay ladder logic, most PLC programs can be represented as a set of Boolean expressions Φ. As this is a commonly used intermediate representation for logic verification [8, 15], we adopt it here as well. Each expression in Φ is of the form $y_i \leftarrow \phi_i$, where $y_i \in O \cup \mathcal{C} \cup \mathcal{T}$ is the result of evaluation and $Var(\phi_i) \subseteq I \cup O \cup \mathcal{C} \cup \mathcal{T}$ is the set of variables in the expression ϕ_i. With the exception of timers, all expressions are evaluated using the values of variables at the beginning of the scan cycle. Thus, if one Boolean expression depends on the result of another, the value of the result from the previous scan cycle is used. In the case of timers, the time at the exact moment of evaluation is used.

For illustrative purposes throughout this paper, we use a simplified traffic light system for a pedestrian street crossing, based on the example in [11]. (See Figure 1.) In this system, the only input variable is the button used to request pedestrian crossing (*pressed*). The output variables control the signals for traffic green, yellow, and red (t_g, t_y, t_r) and pedestrian green and red (p_g, p_r). An example of an expression in Φ for this system is:

$$p_g \leftarrow pressed \wedge t_r$$

which says that the pedestrian green light should be active only if the crossing button has been pressed and traffic has a red light ($p_g, t_r \in O$ and $pressed \in I$). It is a standard practice that each output variable is only assigned once in the logic. Thus, $\neg(pressed \wedge t_r) \rightarrow \neg p_g$.

A PLC is typically programmed from commodity systems over a serial connection. The programming machine in question is called the Master Terminal Unit or MTU. (Note that the MTU need not always be connected to the PLC, as this may sometimes only be the case when uploading new logic.) Along with uploading code to PLCs, MTUs may also upload configuration parameters, and collect plant statistics from the PLC. The MTU also represents the main entry point for PLC malware. Typically, the only security present between the MTU and PLC is a password-based authentication to the PLC before uploading new code. One can imagine the myriad ways in which malicious code on the MTU could bypass such a mechanism.

2.1 PLC Malware

Recent concerns surrounding PLC malware were spawned from the emergence of the Stuxnet virus [21, 6].[1] Without dwelling on the alleged purpose of the virus, we briefly describe Stuxnet's internal mechanics as detailed in [6]. Stuxnet had a sophisticated infection process which used code signed by two valid certificates, multivector propagation (including via USB), and Microsoft Windows zero-day exploits. The goal of Stuxnet's propagation is to reach the nodes in the SCADA system that are directly connected to the PLCs operating the target plant, i.e. the MTUs. Once executing on an MTU, Stuxnet uploads its payload of static code blocks to the PLC. At this point, the process is under malicious control.

[1] There are several previous examples of malware having affected control systems unintentionally [12].

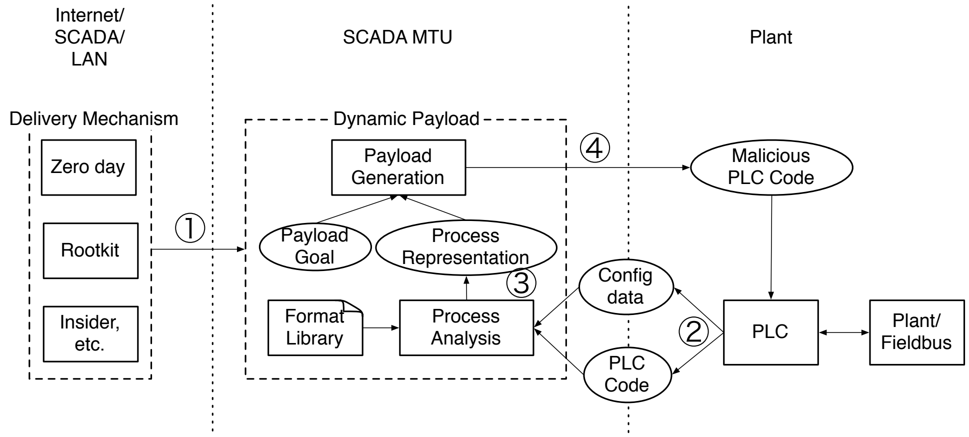

Figure 2: The basic steps for constructing a dynamic payload based on observations taken from within a process control system.

Because the payload was precompiled, it is believed that Stuxnet's authors had previous knowledge of the exact layout of the target process and plant. Thus, it is unlikely that its attack would succeed against any other facility besides the intended one. This need for *a priori* knowledge of the target is assumed to be the main mitigating factor against the more common occurrence of PLC malware. It is this belief that motivates our inspection of dynamic payload generation for PLC malware.

3 Dynamic Payloads

Figure 2 shows the basic steps PLC malware take to dynamically construct a payload against an unknown process. As with any malware, it must first *infect* (1) one or more hosts before executing its *payload*. Infection may occur via viral propagation, Trojan horse, insiders, or any other attack vector. PLC malware ultimately tries to infect a host that can reach a PLC. Because the details of the process are unknown at this point, a payload cannot yet be directly uploaded. Instead, the PLC's memory contents are read (2) for a step called *process analysis*, which produces a canonical *process representation* (3). This may require the use of a *format library* to decode proprietary binary formats. The process representation is then used by the subsequent *payload generation* step to create a payload that will achieve the *payload goal* in the plant (4). If payload generation is successful, then a payload tailored to the specific process may be uploaded to the PLC and executed.

For the remainder of this section, we describe techniques by which each of the above steps may be achieved.

3.1 Payload Goals

A payload goal specifies the behavior that the adversary wishes to cause in the plant. It may be as simple as "Open all breakers in the electrical substation," or as complex as "Identify all incompatible regions of track and signal two trains to enter a conflicting route." It may also be very broad in scope, e.g. "Identify and violate all safety checks maintained by the PLC." Regardless of the exact goal, the dynamic payload will ultimately be a sequence of one or more assignments to output variables that achieves the goal in the plant. Thus, the payload goal can be thought of as a template for the dynamic payload, with the specifics being filled in by the steps of process analysis and payload generation.

3.2 Process Analysis

The job of process analysis is to convert the logic and data read from a PLC into a canonical process representation. A complete process representation should contain both a canonicalization of the PLC code, and the mapping from input and output variables to their corresponding sensors and devices in the plant. While the code can always be obtained by reading the PLC's function blocks, it may not be possible to obtain or infer the device mapping. The challenges associated with each task are described as follows.

Recovering the Boolean equations. The first step towards obtaining the process representation is to recover the set of Boolean equations Φ that represents the logic. Similar to the procedure for reverse engineering a typical computer program, the native code will have to be disassembled into mnemonics and then transformed to the higher-level representation. The disassembling will require one module in the format library for each native binary execution format.

A good candidate for mnemonic language is the IEC 61131-3 Instruction List (IL) language [3]. IL is an assembly language for accumulator-based architectures that is supported at least partially by most popular vendors including Siemens, Rockwell Automation, and ABB. Because IL is accumulator based, it can be converted to a set of Boolean equations via a symbolic execution tracking the logic accumulator. As a proof of concept, we wrote a simple program to recover Boolean equations from Siemens' version of IL (called statement list) in under 200 lines of Standard ML code. Previous investigations have shown that the mapping for disassembly can be reverse engineered with concentrated manual effort [6].

Discovering Plant Devices. Given the capabilities of modern high-end PLCs, it is in some cases possible to extract a description of the plant directly from a PLC's configuration data. A prime example of this is the emerging use of *process fieldbuses* such as Profibus and Profinet, which is quickly becoming a dominant solution for industrial Ethernet [1]. In these protocols, each piece of plant equipment identifies itself by a unique ID number that specifies its vendor and model. By collecting the device IDs, PLC malware may infer higher-level aspects of plant behavior such as which safety properties are most critical. Fieldbus device information may be acquired in two ways: by querying system configuration data in the PLC or by executing a small fieldbus scan on the PLC. The former is stored in *system data blocks* in Siemens PLC's and the latter can be achieved by uploading the scanner program to the PLC from the infected MTU.

The device IDs themselves reveal no semantic information about the nature of a device or its purpose. However, there are Internet databases, e.g. on the Profibus website, that contain pairings of device ID with additional device information. These databases may be scraped by PLC malware authors. One additional complication arises from the fact that device information is collected via a specialized network protocol. Because fieldbus-enabled devices are not connected directly to the PLC's input and output ports, PLC malware must piggyback on existing fieldbus interface code to send commands to devices.

Inferring Plant Device Types. While the target PLC may not support fieldbus communication with devices, it may still be possible to infer the *types* of devices in the plant. This will however require some additional hints to the malware from its author regarding the nature of the plant. For example, if the target plant is a rail yard, then it is known that the two most common devices are signals and switches. One common safety requirement for rail yard automation is that a signal be activated several seconds before a switch is activated [10]. Pairs of signals and switches may be identified by searching for logic variables that are connected by such timing delays. Similar ordering relationships exist in the manufacturing processes, and the operation of switchgear in electrical substations [17].

3.3 Payload Generation

The payload generation step produces a PLC program that will execute the payload goal in a specific plant. It should also be recognized if this is not possible given the available information about the process. Payload generation may result with infeasible if either the process description contains insufficient information about the process to instantiate the payload goal, or if the goal is not relevant to the structure of the process.

Inferring Safety Interlocks. A *safety interlock* is a check in a PLC's logic that attempts to ensure the plant never enters some unsafe state. For example, an interlock may be used to make sure that an electrical substation only performs one switching operation at a time [17]. From the perspective of PLC malware, the safety interlocks offer a definition of how to make the plant perform unsafe operations. It may obtain the set of interlocks by analysis of the set of Boolean equations in the process representation.

Formally, safety interlocks my be stated as relationships among the variables in O. We can illustrate a basic interlock using the pedestrian crossing. The output variables p_r and p_g control the pedestrian's red and green lights respectively, and t_r and t_g control the red and green lights for traffic. A light is turned on only when its variable is true. Without even examining the implementation, one can imagine some interlocking properties. For example the state $p_g \wedge t_g$ should never occur, or else traffic might conflict with pedestrians crossing. To ensure that this does not happen, an interlock statement such as $p_g \rightarrow \neg t_g$ should be added to the logic. The presence of such a statement is immediately indicative to PLC malware that sending the signals $p_g = 1$ and $t_g = 1$ to the plant will cause an unsafe state, regardless of the meaning of the variables p_g and t_g.

Of course, the property $\neg(p_g \wedge t_g)$ may be implicit in the implementation. For example, it may have been verified via formal methods that all executions preserve the desired property. For such implicit safety properties, it is not guaranteed that PLC malware can always infer an unsafe input to the plant (especially when the number of variables in the property becomes large). A significant body of work exists on property verification for control systems [8, 15, 7, 11, 10], which may be leveraged to find implicit safety properties. It is however important

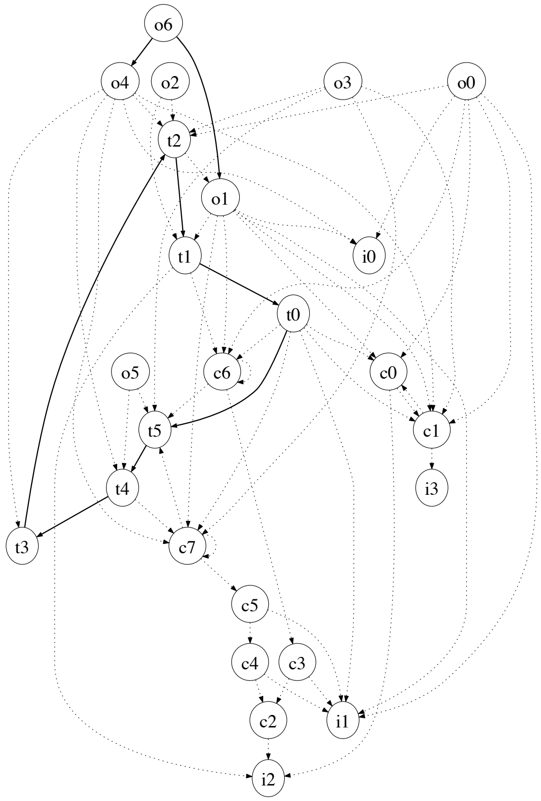

Figure 3: Dependency graph for traffic light control.

to differentiate between the problem of verifying a property and finding one. Even if the verification problem is tractable, the search space of possible properties can be quite large. Because the verification procedure can be quite difficult in practice, most safety properties are explicitly encoded in the process. Though it is worth noting that an equivalent rewriting of a process that contains only implicit properties could be an effective measure to thwart some dynamic malware payloads.

Inferring Plant Structure and Purpose. Before delivering a payload, PLC malware may want to test that the plant is of a specific class. Causing anomalous but harmless behavior due to misunderstanding of the purpose of the plant is likely to cause suspicion. The plant structure refers to the relationships (e.g. dependencies) between plant devices. One tool for achieving this is the dependency graph. Much like dependency graphs are used to statically find flows between variables in programs, they can be used to identify the flow of work between devices in a plant. Questions that a dependency graph can answer include which sensor inputs affect which devices, and the ordering of devices in the process' sequence of events.

An example of a process dependency graph for a full-featured traffic light control system (taken from [19]) is shown in Figure 3. We assume that it is only known if a variable is an input, output, state, or timer, labeled as i*n*, o*n*, c*n*, and t*n* respectively. There are at least two items of interest in the dependency graph. First, the six timers form a cycle, indicating that the process follows a set sequence of events in repetition. This means that the process is inherently sequential in nature as opposed to event driven. Second, the output variable o6 (top) depends on two other output variables, o1 and o4, and is not a dependency for any other variable. This suggests that o6 is interlocked into o1 and o4 as a terminal condition in the process. Indeed, inspection of the ladder logic reveals that o6 triggers an alarm condition when the two opposing green lights controlled by o1 and o4 are simultaneously active. Thus, the assignment ($o1 \leftarrow 1$, $o4 \leftarrow 1$, $o6 \leftarrow 0$) will cause an unsafe state in the plant by disabling the alert signal when the green lights are conflicting. We have found that this same pattern reveals alarm states in ladder logic programs for industrial processes.

Compiling The Payload. The malicious payload is a piece of control logic that ultimately assigns values to output variables in order to disrupt proper plant behavior as described by the adversarial goal. In the absence of a specific goal, a measure such as violating all safety interlocks may also prove destructive. If the goal contains assignments to devices for which no variables have been discovered or inferred, then a payload cannot be compiled. Otherwise, a set of Boolean assignments is created, and assembled back into the PLC's native format. (The format libraries for this step are available from most vendors.) As was the case with Stuxnet, the malicious assignment may be embedded within the valid logic to remain stealthy for some time before executing.

4 Related Work

Significant work has been done in the automation of exploit discovery and execution. Penetration testing frameworks such as Metasploit [14] and Canvas [2] use collections of known exploits to test the vulnerability of entire networks. Increasingly, these frameworks and tools are becoming applicable to process control systems. For example, The White Phosphorus and SCADA+ extension to the Canvas framework contain modules specifically for attacking SCADA networks [13]. Additionally, researchers have consistently identified vulnerabilities in SCADA systems. One study found over the course of ten years that SCADA systems exhibit all of the vulnerabilities found in IT systems ranging from arbitrary code execution, to cross site scripting, and denial of service [16]. Just as recently, a researcher released exploits for 34 new vulnerabilities in popular SCADA products [22].

The framework for attacking PLCs presented in this paper is complementary to these efforts. Often, compromising the SCADA network to reach a PLC is a necessary prerequisite for the dynamic payload approach described here. The existing attack frameworks however, lack mechanisms to dynamically derive PLC attacks, which is a significantly different problem.

5 Summary

We have described the problem of designing PLC malware that automatically generates a malicious payload against a process control system, with little or no prior knowledge. Open problems that we have identified include the definition of payload goals, the disassembly and reverse engineering of logic programs, the discovery and inference of plant devices and the relationships between these devices, and the construction of a malicious payload to achieve an adversarial goal in the plant. Each of these problems has been fit into a framework of payload generation that significantly lowers the bar of expertise for those wishing to disrupt process control systems. Along with the development of each of these steps, future work in this area must include practical measures that can be taken to secure PLCs against dynamic malicious payloads.

References

[1] IMS Research Estimates Top Position for PROFINET. `http://www.profibus.com/news-press/detail-view/article/ims-research-estimates-top-position-for-profinet/`, 2010.

[2] Immunity Canvas. `http://www.immunitysec.com/products-canvas.shtml`, 2011.

[3] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the Automatic Verification of PLC Programs Written in Instruction List. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 4, 2000.

[4] A. A. Cárdenas, S. Amin, and S. Sastry. Research Challenges for the Security of Control Systems. In *Proceedings of the 3rd conference on Hot topics in security*, 2008.

[5] Control Technology Corp. Blue Fusion: Model 5200 Controller. `http://www.ctc-control.com/products/5200/5200.asp`.

[6] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. `www.symantec.com/connect/blogs/w32stuxnet-dossier`, 2010.

[7] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi. Model Checking Interlocking Control Tables. In E. Schnieder and G. Tarnai, editors, *FORMS/FORMAT 2010*. 2011.

[8] N. G. Ferreira and P. S. M. Silva. Automatic Verification of Safety Rules for a Subway Control Software. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF)*, 2004.

[9] T. Gjelten. Stuxnet Computer Worm Has Vast Repercussions. `http://www.npr.org`, 2010.

[10] J. Groote, S. van Vlijmen, and J. Koorn. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. In *Computer Assurance, 1995. Proceedings of the Tenth Annual Conference on Systems Integrity, Software Safety and Process Security'*, 1995.

[11] P. James and M. Roggenbach. SAT-based Model Checking of Train Control Systems. In *Proceedings of the CALCO Young Researchers Workshop*, 2009.

[12] B. Krebs. Cyber incident blamed for nuclear power plant shutdown. `http://www.washingtonpost.com`, 2008.

[13] J. Langill. White Phosphorus Exploit Pack Ver 1.11 Released for Immunity Canvas. `http://scadahacker.blogspot.com/2011/04/white-phosphorus-exploit-pack-ver-111.html`, 2011.

[14] D. Mayor, K. K. Mookhey, J. Cervini, and F. Roslan. *Metasploit Tookit: For Penetration Testing, Exploit Devevlopment, and Vulnerability Research.* Syngress, 2007.

[15] T. Park and P. I. Barton. Formal Verification of Sequence Controllers. *Computers & Chemical Engineering*.

[16] J. Pollet. Electricity for free? the dirty underbelly of scada and smart meters. In *Proceedings of Black Hat USA 2010*, 2010.

[17] M. Sachdev, P. Dhakal, and T. Sidhu. A Computer-Aided Technique for Generating Substation Interlocking Schemes. *Power Delivery, IEEE Transactions on*, 15(2), 2000.

[18] Triangle Research International. Connecting Super PLCs to the Internet. `http://www.tri-plc.com/internetconnect.htm`.

[19] L. Yang and C. XianFeng. Design of Traffic Lights Controlling System Based on PLC and Configuration Technology. In *International Conference on Multimedia Information Networking and Security*, 2009.

[20] T. Yardley. SCADA: Issues, Vulnerabilities, and Future Directions. *;login*, 34(6), 2008.

[21] K. Zetter. Clues Suggest Stuxnet Virus Was Built for Subtle Nuclear Sabotage. `http://www.wired.com/threatlevel/2010/11/stuxnet-clues/`, 2010.

[22] K. Zetter. Attack code for scada vulnerabilities released online. `http://www.wired.com/threatlevel/2011/03/scada-vulnerabilities/`, 2011.