1-1-2013

PLC Code Vulnerabilities Through SCADA Systems

Sidney E. Valentine, Jr.
University of South Carolina

PLC Code Vulnerabilities Through SCADA Systems

By

Sidney E Valentine

Bachelor of Science
West Virginia University Institute of Technology 1995
Master of Science
The University of South Carolina 2000

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2013

Accepted by:

Dr. Csilla Farkas, Major Professor

Dr. Manton Matthews, Committee Member

Dr. John Bowles, Committee Member

Dr. Wenyuan Xu, Committee Member

Dr. Herbert Ginn, Committee Member

Dr. Lacy Ford, Vice Provost and Dean of Graduate Studies

Dedication

I would like to dedicate this work to my wife Amanda, my son Jake and my parents Sid and Ruth Valentine. To my wife and son, you have helped me more in this endeavor than you will ever know. I appreciate the patience and support that each of you have shown in allowing me to pursue this dream. It takes a strong support system to be able to contend with continual hurdles and still come out successful on the other side. Without the both of you, I don't know that this would have been possible.

To mom and dad, it is through your guidance, throughout my life, that has shown me how to continuously persevere, never stop reaching for my goals and succeed in spite of the odds. You have molded me into the person that I am today, and for that I will always be thankful.

Acknowledgments

I would like to thank my advisor, Dr. Csilla Farkas, for allowing me the opportunity to pursue this research and providing countless hours of assistance and guidance. I am truly grateful to have had the opportunity to work with you, as well as learn from you, throughout my graduate studies. Through our time together you have taught me how to be a better researcher and writer; and for that I will always be thankful. I look forward to our collaborations in the future and the continued development of this, and many other, projects. As I take this next step in my professional career, I take it not only with you as a colleague, but as a friend as well.

I would like to thank Dr. Manton Matthews for all of his help from the time I began this journey, until its completion.

I would like to thank Dr. John Bowles, Dr. Wenyuan Xu and Dr. Herbert Ginn for their support through involvement in my dissertation committee. I realize that a dissertation committee takes extensive time and effort, and I truly appreciate all of your willingness to serve.

I would like to thank Mr. Jyron Baxter, through his assistance and 'code development mind' we were able to turn the Static Analysis Tool into a tangible object. I am thankful that I had the opportunity to work with you and am honored to call you a colleague and a friend. I look forward to working with you in the future, and am anxious to see what the future of the Static Analysis Tool holds.

Finally, I would like to thank Randi Baldwin, Barb Ulrich, Jewell Rodgers and Sherri Altizer for their assistance in helping me 'take care of the details' even when that needed to be accomplished from three states away.

Abstract

Supervisory Control and Data Acquisition (SCADA) systems are widely used in automated manufacturing and in all areas of our nation's infrastructure. Applications range from chemical processes and water treatment facilities to oil and gas production and electric power generation and distribution. Current research on SCADA system security focuses on the primary SCADA components and targets network centric attacks. Security risks via attacks against the peripheral devices such as the Programmable Logic Controllers (PLCs) have not been sufficiently addressed.

Our research results address the need to develop PLC applications that are correct, safe and secure. This research provides an analysis of software safety and security threats. We develop countermeasures that are compatible with the existing PLC technologies. We study both intentional and unintentional software errors and propose methods to prevent them. The main contributions of this dissertation are:

- Develop a taxonomy of software errors and attacks in ladder logic
- Model ladder logic vulnerabilities
- Develop security design patterns to avoid software vulnerabilities and incorrect practices
- Implement a proof of concept static analysis tool which detects the vulnerabilities in the PLC code and recommend corresponding design patterns.

Contents

List of Tables

List of Figures

List of Abbreviations

CIP.....Critical Infrastructure Protection

COP.....Copy

CPU.....Central Processing Unit

CTD.....Count Down Counter

CTU.....Count Up Counter

EQU.....Equals

FLL.....File Fill

GEQ.....Greater Than or Equal To

GRT.....Greater Than

HMI.....Human Machine Interface

ICS.....Industrial Control System

IO.....Input / Output

JMP.....Jump

JSR.....Jump to Subroutine

LBL.....Label

LEQ.....Less Than or Equal To

LES.....Less Than

LIM.....Limit

MOV.....Move

NEQ.....Not Equal

OTE.....Output Enable

PC.....Personal Computer

PID.....Proportional-Integral-Derivative

RES.....Reset

RTN.....Return

RTO.....Retentive Timer On

SCADA.....Supervisory Control and Data Acquisition

TOF.....Timer Off

TON.....Timer On

XOR.....Exclusive OR

Chapter 1

Introduction

Lack of peripheral device protection in a SCADA system is a problem in that it is the basis by which most 'control based' attacks on the nations' infrastructure could be carried out. Most current work on industrial control system protection is directed toward the graphical monitoring software, as opposed to the devices from which its data is controlled. Network attacks on these peripheral devices, by design, are not required to go through the personal computer (PC) hosting the SCADA software directly. As most programmable logic controllers (PLC's) are now equipped with Ethernet communications cards, an attacker could access the PLC hardware and its programming tools, directly. The traditional SCADA problem in which an attacker enters the system through the PC housing the SCADA software only adds to the issue of protection, but never fully addresses protection of the system at the operational level. This issue is further compounded when you take into consideration internal attackers as well as external attackers and cross reference those two subgroups against malicious attacks verses unintentional coding errors. In this dissertation, we address multiple fundamental errors in the PLC programming platform and present methods by which to defend against, or correct, these errors. These errors are broken down by attackers knowledge, type of attack, severity of the attack performed or intended, internal or external attackers (to determine practical knowledge of the system) and the degree to which a given attack could be achieved intentionally or unintentionally. Rules will be presented to address these scenarios in an open format which would allow for their implementation regardless of controller type.

The research that we are proposing addresses the issue of industrial control system infrastructure at the programmable device level. This can only be properly addressed if the problem is looked at from multiple perspectives and severity levels, both of which are missing from current research in the field. We plan to develop multiple attack models and scenarios, giving real world coding examples and providing a means to address each. This would allow the current device manufacturers, as well as OEM's, to address these issues prior to being placed in 'live scenarios', thereby leaving their systems open for control level attacks. To accomplish my dissertation research, the following tasks are anticipated to be required:

1. Create a table of errors (vulnerabilities) outlining the knowledge of the attacker, or unintentional error, against the probability of the attack occurring. We provide coding examples of 'entry methods' of the vulnerabilities into the system. We will develop a PLC software security taxonomy to model and conceptualize the vulnerabilities we identify. This taxonomy forms the basis to represent mitigation methods of the detected vulnerabilities.

2. Create severity measurements and a severity chart which will outline the severity of the possible attacks and/or unintentional errors in the PLC system and in SCADA. We will present evaluations and examples of these attacks in a manner similar to the attack descriptions of the Open Web Application Security Project.

3. The results of tasks 1 and 2, along with application specific logic (state-transition diagrams) are used to develop formal models of these vulnerabilities. These models are used to identify these vulnerabilities in PLC code and to develop mitigation strategies.

4. Prevention, detection, and removal of software vulnerabilities:

    - Prevention: software design methodologies leading to best practices guidelines, represented as design patterns

- Detecting vulnerabilities: static analysis tools “screening“ the PLC code for vulnerabilities modeled in step 3. We propose a rule-based code analysis tool that 1) detects known vulnerability and 2) identifies the source of the vulnerability. Our aim is to develop a tool with low false positive and false negative occurances.
- Removal of vulnerability: We will link the detected vulnerabilities with the appropriate PLC security design patterns. This will allow the system developer to modify the code in a manner that removes the vulnerabilities. At this point, we are not proposing an automated system to remove the vulnerability because the main focus of this research is to aid the detection of the software vulnerability and to provide guidelines to the developer.

5. Proof-of-concept implementation.

- State-transition-diagram/rule-based detection
- Input: PLC code that has passed the PLC compilation successfully
- Output: List of vulnerabilities and associated design patterns

Chapter 2

Related Work

Currently, most facilities that use factory automation are turning to SCADA systems to track and control those factory automation devices. This includes not only manufacturing facilities but also those major infrastructure facilities such as power, water and natural gas [23]. By using a SCADA system to track and control these systems, it leaves them extremely vulnerable to both those individuals with malicious intent as well as those that made unintentional mistakes.

It appears that the research on the problem domain as a whole (SCADA technology as it relates, in general, to the public and private sectors) is slowly beginning to make its way to the forefront [44]. During the last decade, we have seen an increased national awareness of critical infrastructure incidents. Assessing and mitigating the cyber security vulnerabilities of SCADA systems are in the focus of academia, government, and industry research. Nicholson et al. [34] give a survey of the security concerns in SCADA systems. The authors present the change of focus in SCADA security, provide and overview of the know attacks and the type of malicious users, current and future threats, and discuss current best practices. This is a result of the potential impact that SCADA technology could have on the national infrastructure arena. The related research that we have found spans multiple areas of interest pertaining to the specific job function of the researcher. These related works, to date, range from utility company consortiums and working groups to government level directives and studies. For example, there currently are working groups that have been created for the various infrastructure sectors of water, electricity and nat-

ural gas [4, 11, 29]. Furthermore, the national agencies such as the US Departments of Energy and Homeland Security each have published white papers and begun initiatives to begin investigations into the problem domain of SCADA systems in general [2,36]. The White House has released "Presidential Directive 63" as well as "The National Strategy to Secure Cyberspace" both of which discuss SCADA systems as a direct threat to national security [24]. The vendor specific publications suggest that increased security may adversely affect their products performance and, therefore, strongly encourage the end user to disable or bypass certain security features [5, 6].

Academia research to strengthen SCADA security falls in two general categories: 1) overview of SCADA security risks and the need for new security technologies to strengthen security [9, 16, 19, 26, 32, 34, 43, 45] and 2) developing new methods to support security analysis and technologies [12, 15, 21, 30, 31, 33, 38, 39]. For example, Cardenas et al. give an overview of the cyber security risk in industrial control systems and emphasize the importance of distinguishing these systems from general purpose IT systems [9]. The authors present detailed overview of government and industry regulation, such as North American Electric Reliability Corporation (NERC) cybersecurity standards for control systems [35] and the NIST Guide to Industrial Control System (ICS) Security [46] to improve SCADA security. Cardenas et al. argue that the knowledge of the physical system enables malicious attackers to change system behavior, therefore indicating control device vulnerabilities. Several unique SCADA security requirements, e.g., real-time requirements, need for continuity of operations, and large number of legacy systems, over traditional information security are discussed in the paper. Miller and Rowe present a comprehensive overview of SCADA and critical infrastructure incidents in [32]. They propose a standardized taxonomy of SCADA incidents to support comparison of known incidents.

Until recently, SCADA security focused on network-based security threats, assuming that preventing unauthorized external access to the SCADA system provides

sufficient security. However, this approach will not prevent attacks exploiting other SCADA components, for example malicious control code for PLC components. In their 2007 publication, pre-Stuxnet, Valentine and Farkas [50] argued that the Programmable Logic Controllers (PLCs) are vulnerable to intentional software-based attacks by malicious users. The authors discuss the inability of PLC code compilers to detect such software errors. In their followup publications [51, 52], the authors provide a taxonomy of coding errors, recommend detection and mitigation methods.

The widely publicized Stuxnet [53] attack has shifted national attention to address software vulnerabilities of control devices [12,30,31,38,45]. Schaefer [45] discusses the disconnection between modern PLCs and the physical world these devices control. In particular, the author points out several important aspects of ladder logic execution that may create unsafe conditions, such as race condition. Olmstead et al. [38] and Minn at al. [31] survey PLC security concerns and provide guidelines to mitigate threats. The authors study software-based vulnerabilities among the important threats against SCADA. Several publications address the need for monitoring software process controllers [12, 33]. While these approaches are useful to detect anomalous activities of malware, they do not prevent the initial execution and propagation of such malware. The closest to our work is the attack presented by McLaughlin and McDaniel [30]. The authors developed an automated tools, called SABOT, that is capable of generating PLC code that, when executes, creates system behavior according to the attackerâĂŹs specification. For this, the tool must have access to the control logic bytecode from the targeted PLC. The authors demonstrate that contrary to general belief that attacks against SCADA systems may be launched by attackers without any specific knowledge about the system. In our work, we address the threat of malicious PLC code uploads by requiring that all PLC code uploads must be evaluated by our static analysis tool. Therefore, the malicious code, generated by SABOT, would be detected and prevented from execution.

The current research appears to focus more on the data monitored by the SCADA system itself and the malicious means available to shut down or slow down the SCADA system when used as a controller interface [8, 17, 18, 23, 40, 41, 46]. Ongoing research needs to consider the topic of SCADA systems from the viewpoint of the negative impact that breaching the SCADA system terminal could have on the PLC ladder logic itself. By addressing this problem you would then begin to develop a basis for the overall protection of automated control systems. This would begin to allow for a layered approach in protection. The SCADA PC would be protected, for example possibly by IPSec, while implementing a second layer directly into the PLC compiler would begin to track changes in the PLC code and look for known vulnerable statements that could cause severe issues in intended functionality. We believe that the available research using this approach is somewhat limited at this point due to a lack of understanding of possible fault scenarios in the ladder logic itself. We believe that by expanding the research to include investigation of protection of the PLC ladder logic through fundamental changes in the way that the logic is compiled and tracked will begin to address a more solid foundation for automation security as a whole.

Currently, there is no existing 'complete' solution to this problem. This explains why many private sector working groups are being formed to better understand the problem of developing safe and secure systems. For example, the beginnings of a document currently in the formation process by the The American Gas Association [4] published an overview and recommendations on secure SCADA communications, policies and plans. This work, similar to other publications, focuses on network level security. It fails to take into consideration the protection of system level components such as the PLC's. We argue that while it is critical that appropriate security safeguards are implemented, they cannot protect against exploitation of code-level vulnerabilities.

The subject of SCADA system security is at the forefront of discussions involving the protection of the national infrastructure. These processes include those elements critical to everyday life such as water, power and natural gas. This can be seen in documentation from various US government agencies. The Homeland Security Act of 2002 [2], specifically addresses the concept of infiltration of a SCADA system from a network level. Furthermore, The US Department of Energy [36], in their working document, "21 Steps to Improve Cyber SCADA Security", proposed guidelines for improving cyber security for SCADA systems. These guidelines are intended to help alleviate some of the most common "hacking" problems related to SCADA system. The main focus of these documents is the accessibility of SCADA components through IT. The guidelines do not address the hardware components used to supply information to the SCADA system, such as the PLC or its subsequent devices. These elements are critical in the creation of any plan which is intended to fully protect automated systems currently and into the future. To further solidify the need for a more complete solution, we refer to an article published by the US Department of Homeland security as recently as October 31, 2012 [49]. This article describes a buffer overflow vulnerability which resulted in a denial of service attack. This article suggests the following solutions:

- "Minimize network exposure for all control system devices. Critical devices should not directly face the internet."
- "Locate control system networks and remote devices behind firewalls, and isolate them from the business network."
- "When remote access is required, use secure methods, such as Virtual Private Networks (VPNs), recognizing that VPN is only as secure as the connected devices."

These solutions are not practical, given the current state of modern control systems. Current control system devices such as PLC's, have on board networking capability and as such have potential to "directly face the internet". To minimize this capability may, in certain instances, minimize the ability of the device to deliver data optimally. To the second point, although all industrial control system users and developers are strongly encouraged to separate the business network functions from the control systems functions, this may not always be practical or possible based on availability of resources. Finally, to address the third and final point, "VPNs are only as secure as the connected devices." PLC's, in particularly legacy systems were not designed as, nor intended to be, secure devices. Therefore, it is imperative that solutions, such as those addressed throughout the remainder of this work, be considered as a probable solution to a practical scenario. This problem is magnified, when you consider that tools are readily available, and generally free, which allow hackers into internet facing control systems [48]. Each of these reports continue to address the problem from a network security perspective. As such, our research expands on the current concerns by addressing the problem of software application security.

2.1 SCADA and PLC Overview

This section will give a general SCADA and PLC overview. Section 2.1.1 begins by giving an introduction to ladder logic. Section 2.1.2 describes the PLC hardware, specifically as it pertains to the hardware configuration types, PLC CPU, and input and output cards. Section 2.1.3 gives an overview of SCADA systems and their relation to PLC's and the automation process.

2.1.1 Overview of Ladder Logic

Ladder logic is the basis behind all PLC programming, regardless of the hardware manufacturer. While each manufacturer may have their own programming tool, fundamentally, each have the same hardware and software requirements. In this section, we will give a brief overview of ladder logic and its components.

Ladder logic programming is based on graphical symbols intended to mirror the hardware which was once solely used in automated processes. The backbone behind the entire concept of this language is the hardware relay. Prior to PLC's, hardware relays were used as the switching mechanism of choice in automated systems. The problem with this approach was the size and adaptability of the processes. If it became necessary to alter the way that a given process functioned, it required additional hardware and generally space for expansion. Figure 2.1 shows a standard hardware relay.

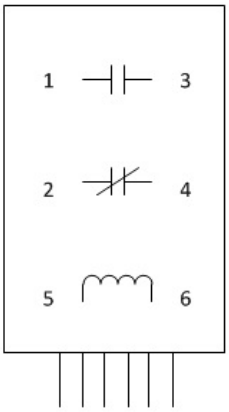


Figure 2.1: Standard Hardware Relay

In the diagram, points (1) and (3) represent the hardwire points for a normally closed contact, points (2) and (4) the hardwire points for a normally closed contact and points (5) and (6) the hardwire points for the activation coil. Each hardware relay generally had one normally open contact, one normally closed contact and one

latching coil. As a general functional overview, a standard relay would operate as follows: the normally open and normally closed contacts are in the original (resting) state until which point the control coil becomes activated. Once this control coil becomes activated, the normally open and normally closed contacts change from their resting state to their activate state. When this occurs a normally open contact becomes closed and similarly, a normally closed contact opens. To bring a relay panel to the level of scalability of a modern day PLC system, it would require roughly 20,000 hardware relays to do the work of one small PLC system. This takes into consideration that the PLC has functionality built in to use internal binary points as well as the hardwired input and output points. In terms of altering process functionality, with the hardware relay system, each relay had to be manually rewired as needed. With the modern day PLC system, the functionality can be changed, including the hardware, based purely on software tools. This ease of alteration becomes the basis behind this work.

As stated, PLC ladder logic code was designed around the concept of these hardware relays. As such, PLC ladder logic software has graphical components for normally open contacts, normally closed contacts and latching mechanisms (coils). Figure 2.2 shows a basic set of PLC components.

Figure 2.2: Standard Set of PLC Components

As PLC systems began to expand, the functional components grew from only using contacts and coils, such that would be found in a relay, to the incorporation of other hardware components such as timer and counter mechanisms. These devices were followed by the addition of mathematical functions, comparative routines and

proportional integral differential (PID) loop controllers. The modern day PLC system can perform any of the functionality of a traditional system with features that exceed those that were available to purely hardware based systems [3]. Figure 2.3 shows a standard ladder logic diagram.

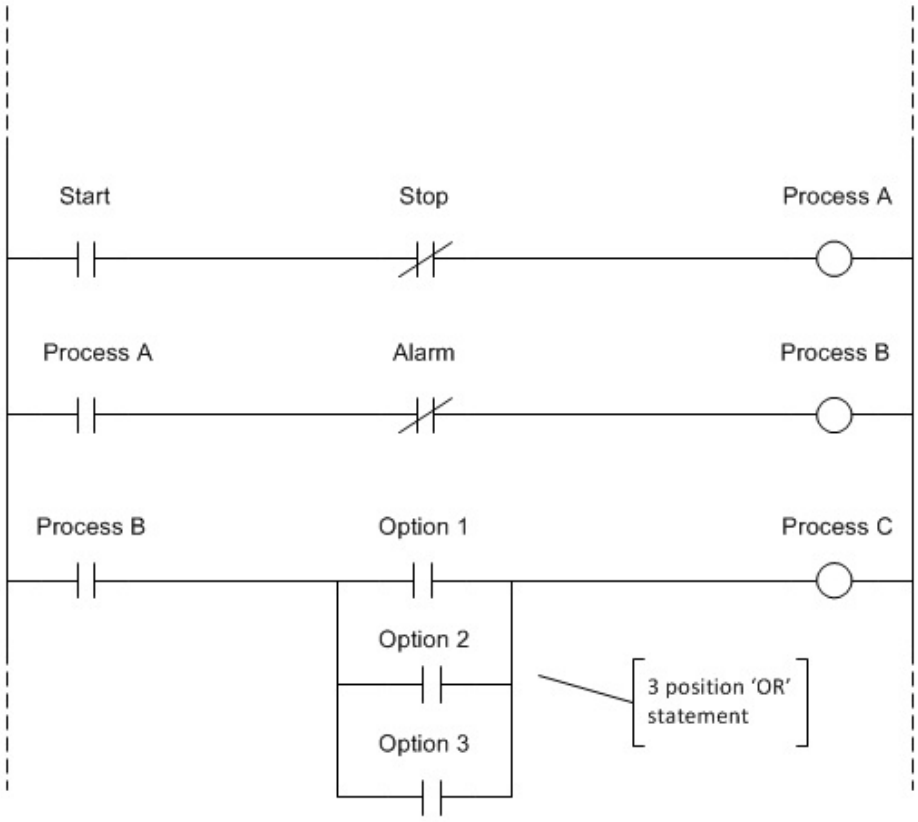


Figure 2.3: Standard Ladder Logic Diagram

This programming style is referred to as ladder logic, due to the fact that each line shown represents one 'rung' on the ladder. The flow of a standard ladder logic program is left to right, top to bottom. This flow will continue until which point a command is encountered that would move the pointer to a different location in the code, such as a jump, return or jump to subroutine. In the example shown, the ladder logic would read as follows: once the start command is activated (closed), the stop command is verified not to be activated (remains closed). If both of these conditions are true, then the 'process A' coil is activated. Once the 'begin process' coil

is activated, it's associated 'process A' coil on the second rung is activated (closed). When this contact has been activated, the alarm command is verified not to be activated (remains closed) and the 'process B' coil is activated. Finally, once the 'process B' coil is activated, it's associated 'process B' contact is activated (closed). At this point, we move to the section of the third rung where we encounter the 'OR' statements. This section of the ladder allows for a decision to be made between options 1, 2 or 3 as to which will allow 'process C' to activate. Note, as we have worked through this example, that the contact and coil operations of PLC ladder logic are identical to the functionality of the traditional hardware relay previously described.

2.1.2 Overview of PLC Hardware

The hardware that makes up a standard PLC unit can be found in one of two configurations / styles: 'rack mount' or 'block'. The major difference between either of these available configurations lies in the ability to alter the input and output devices available to each. We will now give a general overview of each configuration:

2.1.2.1 Available Configuration Types

Block Configuration: When a PLC is considered a 'block configuration', the hardware itself comes as a standard package that is purchased with a preset amount of input and output points, a specific communication protocol and the CPU. This configuration comes as one complete unit and cannot be physically altered. The only means available to expand this configuration is by chaining the devices together using the available communication protocol and adding more blocks to the chain. This would allow the user to expand the number of input and output points, in very small chunks. Figure 2.4 shows a standard block style PLC hardware device.

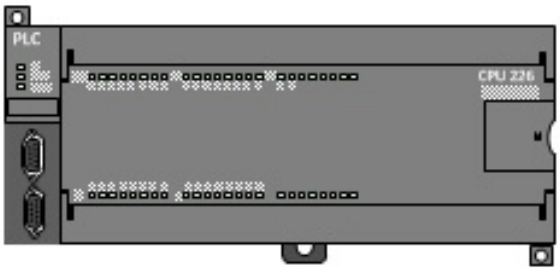


Figure 2.4: Block Style PLC Configuration

Rack Mount Configuration: A 'rack mount configuration', PLC configuration allows the user to select, and interchange, everything in the PLC control system. This includes the CPU type, the number and type of input and output cards and the communications protocol. All of the hardware involved in a rack mount configuration passes data between the input and output cards by way of a slot based chassis. This chassis serves a two fold purpose: 1) to supply power to the entirety of the rack and 2) transmit data across a hardware back plane. Figure 2.5 shows a standard rack mount configuration.

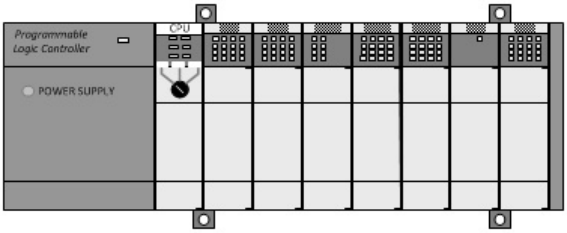


Figure 2.5: Rack Mount PLC Configuration

2.1.2.2 PLC CPU

The PLC CPU unit is the repository for ladder logic program as well as the processor for the information gathered from, and delivered to, the external devices being controlled by the PLC. The CPU contains all of the pertinent information required to fully automate the process in which it is involved. The number of input and output

cards and their types, the communication protocol, and all information contained within the PLC data tables are located in the PLC CPU. There is only one CPU module allowed in the rack mount configurations of the legacy systems currently available by all of the PLC manufacturers (legacy systems represent over 80 percent of those PLC's currently in infrastructure areas). The most current systems allow the designer to incorporate multiple CPU modules in the hardware design of their process. This has the potential of allowing the software (ladder logic) designer to split the process across multiple computing devices making code integrity even more crucial from a security perspective.

2.1.2.3 Input and Output Cards

The external information which is transmitted to and from the PLC occurs over various input and output (I/O) cards. These I/O cards can contain numerous connection points, depending on the need of the user, and can be either digital or analog in relation to the type of information sent and received. We will now give a brief overview of input and output cards and the devices they control.

Input Cards: As noted, PLC input cards can come equipped with multiple connection points, this is solely at the discretion of the developer. A single PLC input card generally comes with its connections points in multiples of 8, with the largest being 64 (8, 16, 24, ... 64). The PLC input cards gather their information from their associated control devices such as temperature sensors, level sensors, proximity switches, and variable frequency drives.

Output Cards: Just as with PLC input cards, PLC output cards can come equipped with multiple connection points. The number available within any manufacturer generally mirror those available for the input cards. The PLC output cards send out control information in the form of analog and digital signals. These signals are used by the various control devices as an activation mechanism or as a set point.

2.1.3 SCADA and Automation System Overview

The PLC is the backbone of the system architecture of an industrial network. The information which is transmitted by the PLC to the SCADA system is collected by the PLC's input and output cards (rack mount configuration) or provided input and output points (block configuration). The SCADA system PC is the information terminal through which the control room operator, and anyone else with intended, or unintended, access can view the real time functionality of the automated system. This terminal is generally connected via Ethernet to the facilities existing network. The SCADA system receives its information directly from the PLC CPU by way of the ethernet connection provided. This could be in the form of a physical communication card, in the case of a rack mount configuration or an internal communication protocol if a block configuration is used. Also, it is now possible to share this information via wireless communications cards as well, which only adds another layer to the security issue.

A standard SCADA system [Figure 2.6] serves as the oversight device which is connected to one or many PLC units throughout a given infrastructure system. The SCADA computer itself is no more than a standard industrial grade computer, running a vendor specific piece of software, which is used to monitor and track the states and conditions of all of the device's connected to its associated controllers. These devices are generally calibrated with the PLC and SCADA system computer upon initial installation and assumed to be accurate thereafter. This assumption is critical to understanding the severity of compromising the PLC. Since the control room operator is taught to rely on data being received by the PLC system and the devices are calibrated using the PLC itself as the calibration method, any individual that gains access to the PLC can potentially directly affect the system and falsify the data that is reported to the SCADA system.

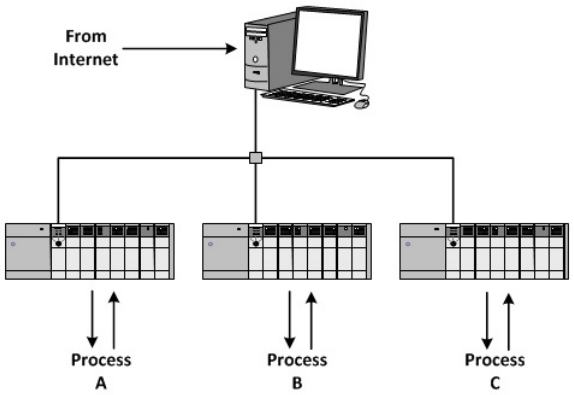


Figure 2.6: Standard SCADA System Configuration

2.2 SCADA and PLC Security

Most of the SCADA system security research addresses security issues raised by network centric operation, such as secure communication, without addressing the security needs in PLC's. As shown in a joint report by the US Department of Energy and the Presidents' Critical Infrastructure Protection Board [36], the current SCADA infrastructure protection focus is on the hardware housing the SCADA software itself and not the programmable devices that are responsible for controlling all of the processes. This report states that "Most older SCADA systems (most systems in use) have no security features whatsoever. SCADA system owners must insist that their system vendor implement security features in the form of product patches or upgrades. Some newer SCADA devices are shipped with basic security features, but these are usually disabled to ensure ease of installation." It continues by stating that "additionally, factory default security settings (such as in computer network firewalls) are often set to provide maximum usability, but minimal security." Current capabilities that permit wireless communications cards in PLC controllers and SCADA systems, makes it necessary to evaluate security needs of each components.

The increased risk of access of malicious users to SCADA components, increases the risk to the PLC code itself. Since the PLC dictates the functionality of the

process, even if functional commands may be given through the SCADA computer, it is crucial that it functions correctly and securely. Unfortunately, attacks against the PLC components, as we will demonstrate in the following sections, are easy to carry out by a sophisticated attacker. For example, just by looking at the ladder logic code it is possible to determine the most likely points of entry into the PLC CPU from an outside source, such as a SCADA system. To the best of our knowledge, there is no related work that addresses the implementation of a best practices guide to correctly writing PLC code which, in itself, could alleviate certain security concerns.

Although, protecting the system on which the SCADA backbone resides may eliminate some of the PLC security threats it does not remove all PLC vulnerabilities. Once access is granted though either the SCADA backbone, or any other network medium, then the entire PLC network is open for an attack. Sophisticated attackers, with working knowledge of the system and ladder logic, may be able to access the PLC system directly. From the PLCs, the attacker can gain access to the SCADA terminal. In 2003, the Davis-Besse Nuclear Power Plant was crashed by a slammer worm that infiltrated the SCADA network. It was stated that the Safety Parameter Display System (SPDS) "monitors the most crucial safety indicators at a plant, like coolant systems, core temperature sensors, and external radiation sensors. Many of those continue to require careful monitoring even while a plant is offline." [42] More recently, a Stuxnet attack was performed on a nuclear reactor station in Iran, this attacked directly targeted the PLC hardware to access and alter the ladder logic used to control the facility.

In a recent ISA article [53] on the Stuxnet attack, it is stated that "prior to Stuxnet, it was believed any cyber attack (targeted or not) would be detected by IT security technologies such as firewalls or intrusion detection systems and defense-in-depth would prevent damage to physical processes. However, previous actual control system cyber incidents (malicious and unintentional) have demonstrated that many

ICS cyber incidents are not readily detectable, and they can cause physical damage even with existing defense-in-depth designs." The article goes on to say "it is important to note the use of the term SCADA, as these same technologies have not been employed on many legacy non-SCADA devices such as programmable logic controllers (PLCs), electronic drives, process sensors, and other field devices. Another implicit assumption in the standards being developed such as ISA99 and the North American Electric Reliability Corporation (NERC) Critical Infrastructure Protection (CIP) standards is they would be comprehensive enough to address cyber attacks against ICSs including sophisticated attacks. The inadequacy of these assumptions against a sophisticated attack such as Stuxnet requires a detailed reassessment of ICS cybersecurity assumptions. Stuxnet is more than data filtration, it is the first rootkit targeted at PLCs. It is essentially a weaponized attack against a process. It has the ability to take advantage of the programming software to upload its own code to the PLC."

In addition to the sophisticated attackers, novice users also represent considerable risks. The basic problems of learning bad habits and applying those bad habits into logic diagrams could cause larger security and functionality concerns. It is imperative that a standard be implemented that would at least provide a mechanism for a novice, or experienced users to be able to verify and validate their programs against a defined set of rules.

The tool and methodology we are proposing would allow the code to be easily verified and validated as often as required. This validation would be against a known set of suspect coding practices. Furthermore, this tool, when aligned with a vulnerabilities ranking mechanism, could allow for early alerts into possible points of entry with which to be concerned.

2.3 Secure Software Verification Methods and Software Code Review Tools

The 24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them [25] outlines best practices for generalized software development. We cross reference these methods to expand and validate our table of vulnerabilities. *SOA Design Patterns* [13] outlines various methods for creating correct design patterns. We use the methods found within this text as a framework for the creation of the design patterns which support secure software development. Furthermore, there are other code security and static analysis tools currently in existence for traditional software development. However, none of these tools are capable of handling ladder logic software. Moreover, the recommended best practices are too general to provide valuable guidance in the complex context of SCADA control systems [27]. Taxonomies of software errors can be used to model PLC ladder logic vulnerabilities. *Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors* [10] suggests a methodology for building a taxonomy to "help developers and security practitioners understand common types of coding errors that lead to vulnerabilities. By organizing these errors into a simple taxonomy, we can teach developers to recognize categories of problems that lead to vulnerabilities and identify existing errors as they build software." We believe a similar solution can be developed which specifically addresses PLC ladder logic software. Our intention is to assist the practitioner in understanding the common types of errors as stated above, while providing a methodology to mitigate these errors. This, in turn, will provide a means to mitigate the security risk created by the errors documented in the vulnerability taxonomy. Furthermore, Landwehr [1]

classified security flaws based on three dimensions, genesis, time of introduction and location. The genesis classification creates two subcategories for flaws, intentional and inadvertent. The Severity Chart that we create provides a similar grouping for the development of the Severity Engine within the Static Analysis Tool: novice and malicious users.

2.4 Limitations of SCADA/PLC Security Research

The current SCADA / PLC security research is limited, due to three critical areas. Lack of understanding of all of the exposed entry points into the automation system, which leads to more SCADA software centric research. The lack of sufficient training by current PLC code developers. This creates a culture of trial and error programming, which is due to a lack of best practices standards. The current research in this field does very little to address the training component and development of a best practices approach to PLC coding. Most importantly, there is a lack of tools which can be implemented and used to test current and future code, both after and during development. This network-centric approach, which is important, has become a network biased approach. That is, the research focuses more on the commonly accepted remote access component and seems to ignore the vulnerability created by the existence of insecure application software [47, 48, 49].

Chapter 3

Proposed PLC Security Framework

3.1 PLC Code Analysis (PLC-SF)

In this work, we address the vulnerabilities in the PLC code itself. The components of our work are shown in Figure 3.1. The input of the PLC Security Framework (PLC-SF) is PLC code that has passed and been accepted by the ladder logic compiler. The Static Analysis Tool, we have developed [51], uses the following three components: PLC Security Vulnerability Taxonomy, Severity Chart and Design Patterns. The output of PLC-SF is list of vulnerabilities and associated design patterns to remove the vulnerabilities.

Currently PLC (ladder logic) code compilers announce any of three states (or a combination or a combination thereof) after the code is compiled. These states are 'compiled without errors', 'compiled with warnings', and 'compiled with errors.' It is assumed by both novice and experienced coders that if the compiler announces 'compiled without errors' that the code is correct. Code that is 'compiled with warnings' may have minor bugs that do not restrict the compilation and the execution of the code by the PLC. Code that is 'compiled with errors' indicates the error, and this code cannot be uploaded to the PLC as long as those error(s) exist. We do not address these errors in this work. However, we anticipate that our Static Analysis Tool will also eliminate some of these errors. The Static Analysis Tool uses the vulnerability taxonomy and the severity chart to detect and rank ladder logic vulnerabilities. It will then compare these vulnerabilities against a set of known design patterns, to

determine a corrective action which would alleviate the vulnerability. We now give a brief description of each component within the Static Analysis Tool.

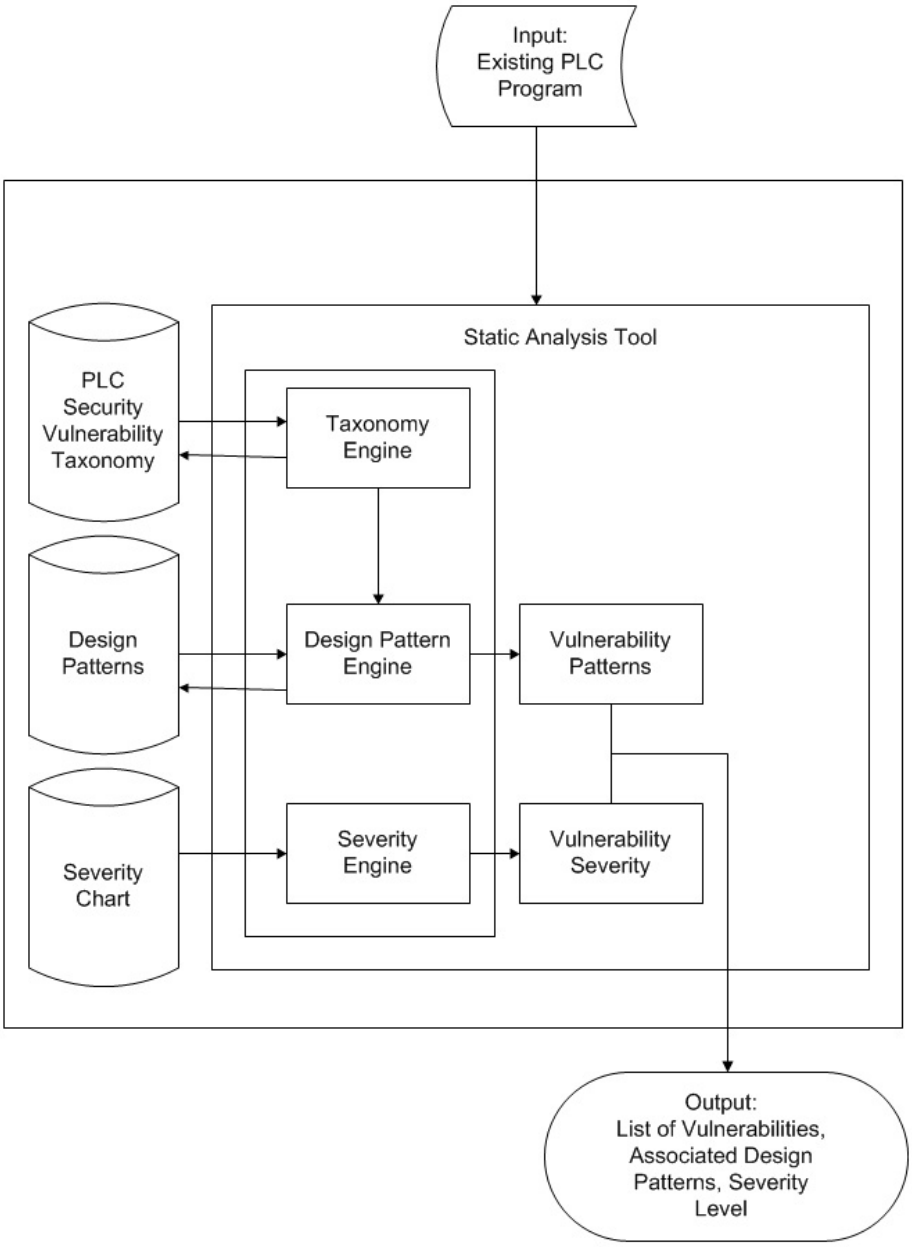


Figure 3.1: Proposed Security Framework PLC-SF

PLC Security Vulnerability Taxonomy: The vulnerability taxonomy is used to conceptualize the vulnerabilities. These vulnerabilities are then depicted using state transition diagrams. Using both the Vulnerability Taxonomy and the associ-

ated state transition diagrams, a Vulnerability Engine was created. This Vulnerability Engine will determine the existence of a vulnerability within the various levels of the taxonomy. The taxonomy that we have created categorizes potential PLC vulnerabilities, as initially depicted in Figures 4.2 through 4.10. The taxonomy is intended to help answer the following questions: "How did this vulnerability occur, and how can it be exploited?" This will then allow for the formulation of detection and prevention methods. Our approach to model PLC vulnerabilities is extensible, representing an initial characterization methodology which can be continually expanded as new vulnerabilities surface. Initially, the taxonomy verifies that, in fact the vulnerability in question is a design-level vulnerability. It is then determined rather the perceived vulnerability is hardware (physical) or software (virtual) based. If it is determined to be hardware based, then the specific subclass is added to the taxonomy, and its physical characteristics mapped. If it is determined to be software-based, then it is determined if a class should be created, or if a subcategory already exists to insert the vulnerability. The Vulnerability Taxonomy will be discussed in detail in Chapter 4, Section 4.4.

Design Patterns: Design patterns show methods of mitigating the vulnerabilities encountered in PLC ladder logic code. We have created design patterns to mitigate the various vulnerabilities listed in the Vulnerability Taxonomy. The design patterns which were modeled, were used to create the Design Pattern Engine. Once a vulnerability is determined to exist, the vulnerability is cross referenced against a list of design patterns in the Design Pattern Engine. The Design Pattern Engine generates design patterns to be given to the user during the output phase of the Static Analysis Tool. The design patterns which were created will be shown and explained in Chapter 5.1.

Table of Vulnerabilities: The table of vulnerabilities groups the vulnerabilities based on the potential consequences of an exploitation. We have assigned a severity

rating to each vulnerability based on the impact of the outcomes to the PLC and SCADA system. Each vulnerability is linked to the table based on the severity level assigned. This will allow the defense to rank the vulnerabilities.

Severity Chart: The Severity Chart links the severity level assigned from the table of vulnerabilities to potential effects in both the PLC and SCADA system platforms. The Severity Chart is the basis on which the Severity Engine is created. The Severity Chart is shown in Table 4.1.

Static Analysis Tool: As previously stated, the Static Analysis Tool takes as its input PLC ladder logic code, determines the existence of a vulnerability within that code, the severity level, or levels, that exist within that vulnerability and the design pattern, or patterns, that can be used to map the best probable solution. This is accomplished by the Static Analysis Tool using three different engines to represent the three distinct internal components. These are the taxonomy, design pattern and severity engines and will be explained in Chapter 6.

List of Vulnerabilities and Associated Design Patterns: The output will consist of a list of the vulnerabilities and their associated design patterns as determined by the static analysis tool. The Static Analysis Tool will determine the existence of the vulnerability, the severity level and the associated design pattern based on the existence of certain strings in regular expressions. This will be explained in detail in Chapter 6.

3.2 Malicious Entry Points

We also studied the interaction between the PLC and other SCADA components. The data input source for the PLC ladder logic are numeric tables that store sensor (device) data. Figure 3.2 demonstrates the standard control flow of a SCADA system. The numeric tables, which consist of binary, floating point, and integer data are the

control repository for the entire system. Erroneous data in these tables may corrupt PLC execution and, therefore, the entire SCADA system. In addition to direct access, there are three ways to modify table entries; data from hardware devices, the PLC ladder logic, and the SCADA PC.

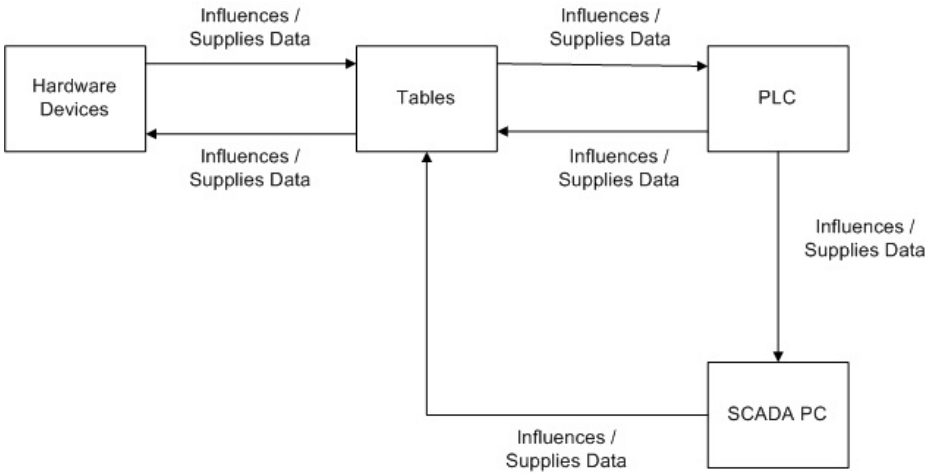


Figure 3.2: SCADA System Control Flow

Hardware devices are devices that monitor or initiate process execution such as variable frequencies drives, proportional-integral-derivative (PID) controllers and Human Machine Interface (HMI) devices. This classification includes all of the physical devices that have the ability to directly receive from, or deliver data to, the PLC. Although wireless communications mediums are available for certain hardware devices and more recent PLC controllers, the communication medium of choice is still the hardwired approach. Using the hardwired approach as our basis of communication, hardware devices can be further classified as those devices which are wired directly to a PLC input or output card.

The PLC Ladder logic, has a direct link to the data tables. The ladder logic code, as well as the data tables are embedded into the CPU on the PLC once they are uploaded. Therefore, there is no external communication necessary to maliciously alter the data, directly or via PLC code, if someone accesses the PLC CPU.

The SCADA PC is used not only to view data but can also be configured to allow the user to input data directly into the PLC, by way of the control tables. This ability is through the database that is resident on the SCADA PC itself. This database is part of the front end of the SCADA package that is installed on the PC. The SCADA PC database, not unlike the data tables on the PLC, not only holds information in the form on numeric data, but also data location information for write and read purposes into the data tables.

Figure 3.3 illustrates the four data entry points that may be exploited by malicious users. Users may exploit these points of entry, directly or indirectly, to modify control data. The dashed lines represent the insecure pathways that exist between each SCADA component. We do not generalize the hardware devices into a specific group that can be accessed directly. This is due to the existence of multiple devices, currently in use in the field, that do not have the capability of direct network access.

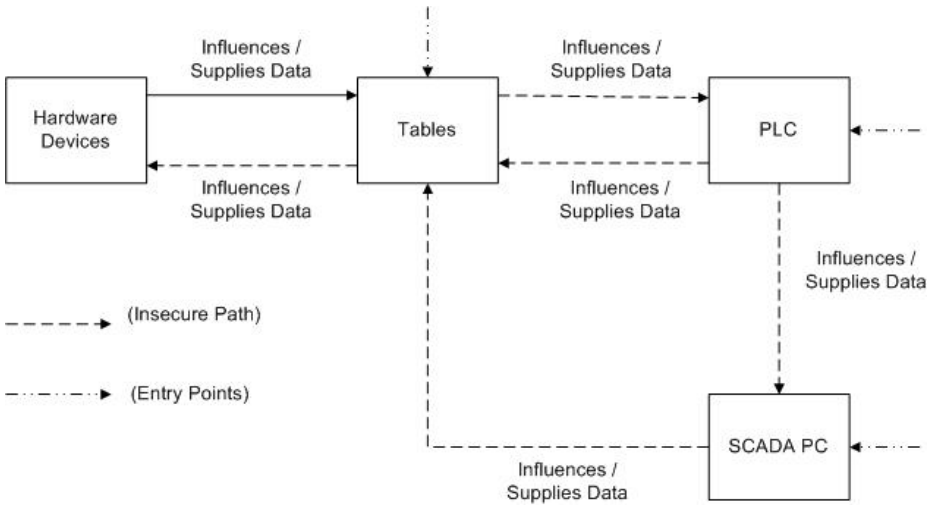


Figure 3.3: SCADA System Control Flow Possible Malicious Entry Points

Chapter 4

Vulnerabilities Analysis

Current ladder logic compilers are not designed to detect security vulnerabilities or subtle logic errors. PLC code, that was compiled without error, may still contain vulnerabilities. These vulnerabilities can be subtle enough that the novice user would not be aware of the possible security risks they represent. Malicious users may exploit these vulnerabilities and cause severe damage. In this chapter, we outline the vulnerability taxonomy and the consequences of their exploitation.

We also associate the vulnerability taxonomy, a severity chart and potential actions that can be carried out by malicious users. Each attack (error) has severity ratings assigned to it, as shown in Table 4.2. Section 4.3 develops a classification mechanism for process critical and nuisance errors as previously defined. The categories shown are broad in scope so as not to be process limiting or process specific. Table 4.3 lists the error type, error classification and opportunities presented to a malicious user through the existence of each error.

4.1 Attack Severity Analysis

This section outlines the attack severity chart as well as the novice and malicious users ability to create each level of severity.

We will present detailed descriptions of the severity classifications and examples of their associated effects. These classifications will allow for the foundation of a best practices guide. Table 4.1 gives a general and functional overview of each of

the severity rankings created [50]. This table outlines the severity level which would be applied under each of the scenarios shown. Each row of the table represents a different level of security, **'A'** being the most severe and **'D'** being the least. Each column represents the effects in the PLC and the SCADA system, respectively.

Table 4.1: Severity Chart

Severity	**Effects in PLC**	**Effects in SCADA**
A	PLC code will not perform the desired tasks	Will not allow for remote operation of the process
B	Serious hindrance to the process	The process may appear to be operating correctly, but given optimal conditions, the machine could be thrown into an unexpected process failure
C	Adversely effects PLC code performance. A minimal cost effect to the project but a "quick fix" is possible.	Data shown on the SCADA screen is most likely false.
D	Effects the credibility of the system, but PLC code is operable.	Incorrect data could randomly be reported causing a lack of confidence in the system and therefore causing the system to be "disregarded" even if the information is relevant.

4.1.1 Severity Classifications

It is critical that each severity level depicts not only the outcomes that can be detrimental to the SCADA system as a whole, but the effect on the individual components as well. These individual components can be any of the automated components which are PLC controlled. We will now give a description of each of severity level.

Severity Level 'A'

- A concern is considered as severity level 'A' if its existence could potentially cause all, or part, of a critical process to become non-functional. Furthermore, residual effects may include malfunctioning of other processes who's outcomes are determined by the process at risk. For example, consider the situation when the mechanism that causes heat to be released from a process is no longer functional. However, the heat continues to be generated into the process. If this situation were not corrected expediently then the device that stores the energy may be damaged, even destroyed.

Severity Level 'B'

- A concern is considered as severity level 'B' if its existence could potentially cause all, or part, of a critical process to perform erratically. This differs from severity level 'A' in terms of the absolution of the result. Severity level 'A' concerns have the potential to cause permanent process failure, whereas level 'B' concerns would cause incremental process interruptions.

Severity Level 'C'

- Severity level 'C' concerns are denoted as quick fixes. The errors are most likely created by 1) a novice user without a good fundamental knowledge of PLC programming components or 2) a malicious user who wishes to cause functional problems.

Severity Level 'D'

- Severity level 'D' concerns involve providing false or misrepresented information to the SCADA terminal itself.

4.2 Examples of Severity Level Effects

In this section, we present examples of each of the severity levels defined in section 4.1.1. We will use the data from table 4.3 as the basis for each.

Example of Severity Level 'A':

Hidden Jumpers: A hidden jumper could involve either a force, an empty branch, or a branch with a normally closed contact that has no trigger coil associated with the contact. Hidden jumpers have the potential to be a severity level 'A' concern in that they could cause all, or part, or a given rung to be inoperable.

Example of Severity Level 'B':

Duplicate Objects: If a duplicate object is installed in the ladder logic, it presents the potential for the occurrence of two distinct issues. First, the duplicate object could fail to let either rung, in which the logic is installed, to activate. Second, duplicate objects can operate on an incremental trigger basis. In this scenario, the logic would randomly select one of the objects to activate.

Example of Severity Level 'C':

Logic Errors: A logic error could involve any element or number of elements within the context of the PLC program itself. We will show this error, in context, using a timer element. The concern lies in the alteration of the timer preset. Initially, the timer preset would be set to a certain value. If a novice or malicious user alters this preset value positively or negatively, severe damage could occur within the process. Assuming the process is programmed based on time delay and not on physical sensor technology, placing one timer out of sequence potentially could alter the entire process.

Example of Severity Level 'D':

Creation of false information: Information that is incorrectly transmitted due to incorrect implementation of specific functions such as timers or math functions.

4.3 Potential Exploitation of Coding Errors

We start the section with a discussion on the effect of the PLC users' knowledge. Then, we present a collection of coding errors and intentional attacks against PLCs and SCADA networks.

4.3.1 Knowledge of the User

As shown in Table 4.2, we outline a set of criteria based on the knowledge of the user. We define a user both in terms of a novice user as well as a malicious user. We acknowledge that a novice user could be malicious and a malicious user could make unintended mistakes just as a novice user. Our contention for these two initial knowledge level classifications is not to disregard the possibilities of crossover, but only to serve as a basis for preliminary data development. Recognizing the advanced level of a malicious user as opposed to a novice user, Table 4.2 makes the assumption that any function that can be performed by a novice user could also be performed by a malicious user. Under this criteria, we will apply the following definitions throughout the remainder of this dissertation: A **novice user** is considered to be an individual that is authorized to work on and view the system, but lacks the correct training or experience. A **Malicious user** is considered as both an authorized user with malicious intent as well as an individual that is not authorized to access the system.

Typically a malicious user is capable of performing all of the functions that would be expected of a novice user. An unauthorized malicious user may also have the capability to hack into the SCADA system, or any computer that would have the programming software required to view, or alter, the PLC ladder logic program. Furthermore, a malicious user would most likely have an advanced knowledge of control systems and their integrated components which were explained in Section 3.2 and shown in Figure 3.3.

Table 4.2: Severity Rating vs. Attacker's Knowledge

Severity	**Novice User**	**Malicious User**
A	Inserts incorrect code. Fails to remove unused table locations prior to compilation. Incorrect IP addressing of PLC network components.	Inserts hidden IO reference points as a potential back door into the program.
B	Lack of knowledge of correct implementation of certain software components. Lack of knowledge of correct implementation of certain hardware components.	Uses advanced knowledge of software and hardware implementation to *correctly place* ladder logic code and IO reference points in *incorrect* locations.
C	Detailed labeling of rungs, components and devices.	Uses the current notation system to incorrectly label rung and component functionality.
D	Incorrectly scales values to be sent to the SCADA display	Everything that a novice user may unintentionally perform.

We give a brief overview of the coding errors and potential exploitation by malicious users. In Table 4.3, these errors are organized into two main categories, process critical errors and nuisance errors.

Process Critical Errors represent those errors that could cause a severe failure in the process operation and can be affected once access is gained to the PLC CPU.

Nuisance Errors represent those errors that would cause minimal process issues and are relatively easy to find.

4.3.2 Process Critical Errors

We have identified the following process critical error categories. We also show examples of how to exploit of some of these errors.

- Duplicate Objects: Objects that have been defined more than once. These objects could include things such as coils, timers and counters. Using our proposed severity rating system, this would have a severity rating of 'A' based on the potential for total process failure.

- Unused Objects: Objects which were defined in the initial database, but were never used in the ladder logic. These pre-loaded variables can be used for random functions. This is given a severity rating of 'A' since the extent to which the unused objects are employed will determine the extent of the consequences.

- Scope and Linkage Errors: Such errors deal with the deletion of, or failure to install, a communication block between two or more separate ladders in a PLC program. This would have a severity rating of 'A' based on the potential for total process failure.

- Logic Errors: Errors that could occur result in state transition, timing, control and data flow issues. This error could be classified as 'A–C', depending on the extent of the logic error and the specific device effected.

- Syntax Errors (or warnings): Warnings that were problematic in compilation, but compilation was not restricted. This code is downloaded to the processor with no more than a warning to the individual downloading to the device. This error is classified as a level 'B' concern due to the fact that initially, no symptoms were present and intermittent failure could occur.

4.3.3 Nuisance Errors

We have identified the following nuisance error categories:

- Hidden Jumpers: These software jumpers effectively bypass a portion of a rung in a ladder logic routine. These are easily hidden to the untrained eye, and are not searchable utilizing the current PLC platforms. This error has the potential to be classified at any level. The severity and consequences depends solely on the location of the jumper.

Table 4.3: Development Error vs. Opportunity

Error Type	**Taxonomy Classification**	**Malicious User Opportunity**
Process Critical / Nuisance	Duplicate objects installed	Alteration of one or more of the duplicate objects
Process Critical	Unused objects	Pre-loaded variables allow for an immediate entry point into the system with no additional requirements on the database
Process Critical	Scope and linkage errors	Installation of jump to subroutine command which would alter the intended file to file interaction
Process Critical	Logic errors	Immediate entry point to logic level components such as timers, counters and arithmetic operations
Process Critical	Syntax Errors	Could cause the system to act intermittently erratic, therefore causing future alarms to be ignored
Process Critical / Nuisance	Hidden Jumpers	Could allow a placement point for system bypass scenario to occur

4.4 Building the Vulnerability Taxonomy

In this section, we we build a vulnerability taxonomy. The purpose of the taxonomy is to aid the process of detecting these vulnerabilities in the PLC code. The taxonomy is intended to be dynamic by design, and was created so that it can be continually expanded upon as future versions of PLC's are created and new errors discovered. Figure 4.1 gives a generalized overview of the Vulnerability Taxonomy. The classifications within each level, and their attributes, will be further explained throughout the remainder of this section.

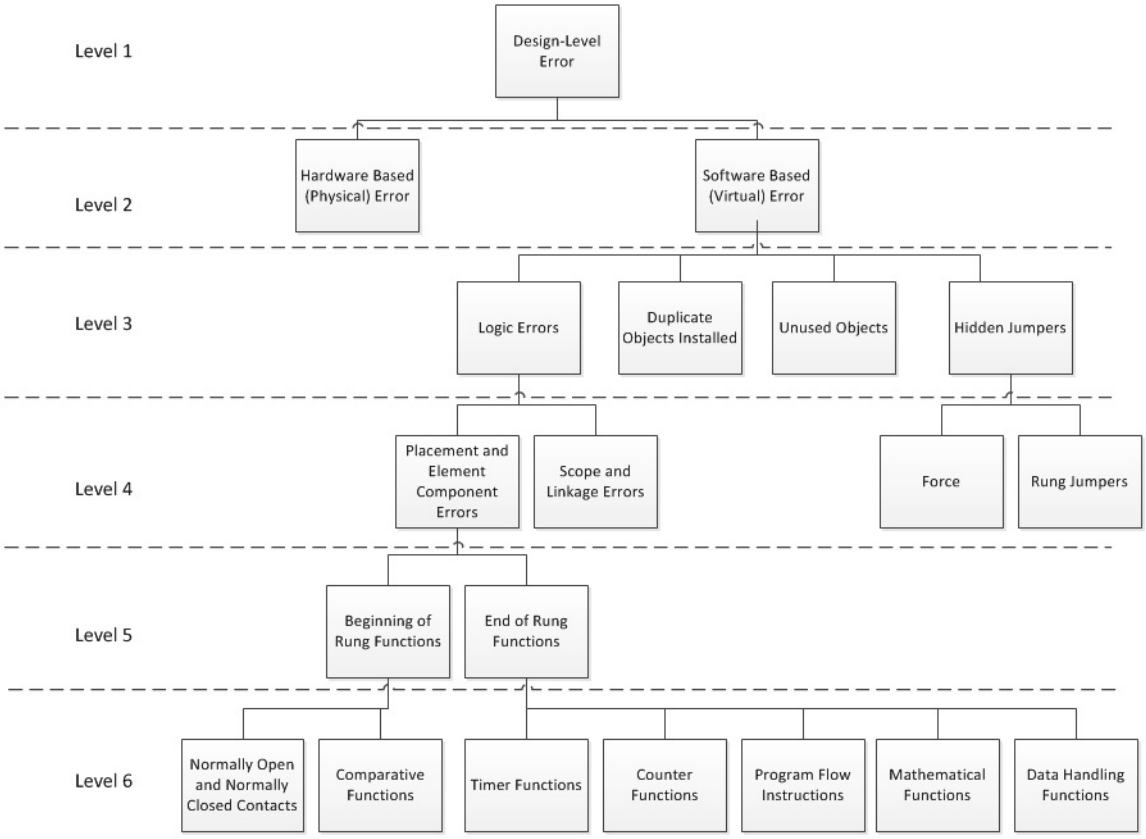


Figure 4.1: Vulnerability Taxonomy

The top level of the taxonomy is shown in Figure 4.2. Figure 4.3 represents the hardware input components to the PLC system with a direct link to the data

tables of the PLC. In this work, we focus on software-based vulnerabilities. Figure 4.4 represents the possible software error classifications. These classifications are further explained in Table 4.3. Figures 4.5 through 4.10 represent each subclass of the software components in which the errors are likely to be found. They also show the security risks which could be encountered under each subclass.

Figure 4.2 represents the highest level of the Vulnerability Taxonomy. The errors that we have chosen to focus on in this dissertation are design-level errors as shown. Design-level errors can be further broken down into the sub-classes of hardware based and software based errors. We will be focusing our research efforts on the software based errors throughout the remainder of this work. The hardware-errors are noted to show that we recognize the existence of these errors and the security concerns that various hardware systems could potentially introduce. We will now define the attributes of each of these areas.

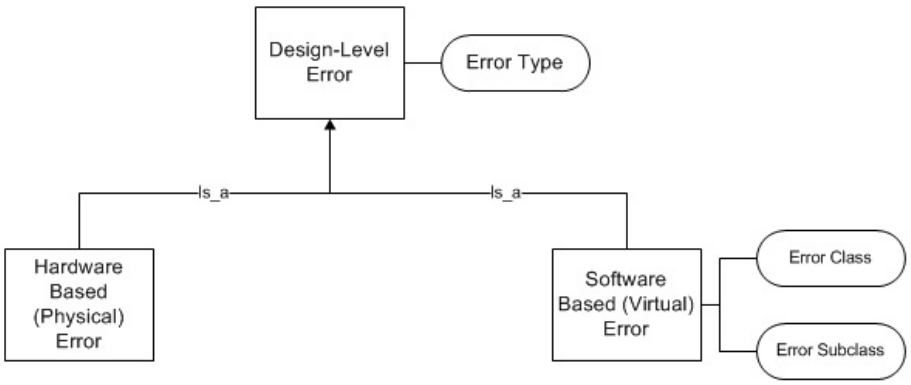


Figure 4.2: Ladder Logic Vulnerability Taxonomy: Design Level Error

Design-Level Errors: The attribute associated with design-level errors is the error type. The possible designations for error type are hardware error or software error, which would lead us to the sub-categories previously mentioned.

Hardware Based (Physical) Errors: The attribute associated with hardware based (physical) errors is device type. The possible designations for device type are

switches, relays, sensors or pushbuttons. These attributes will lead us to our next in the Vulnerability Taxonomy where these attributes will be associated with hardware components individually.

Figure 4.3 represents the hardware components of the second level of the Vulnerability Taxonomy. The hardware based (physical) errors are broken down into the subcategories of switches, relays, sensors and pushbuttons. We will now define the attributes of each of these areas.

Switches: The attribute associated with switching errors is the signal type. The possible designations for signal type are analog or digital, depending on the signal being conveyed through the specific input / output card used.

Relays: The attribute associated with relay errors is relay type. The possible designations for relay type are input or output. This designation depends on whether we are tracking input data or output data for a particular security risk.

Sensors: The attribute associated with sensor errors is sensor type. The possible designations for sensor type are analog or digital, again depending on the type of sensor used.

Pushbuttons: The attribute associated with pushbutton errors is style. The possible designations for style are momentary or latched depending on the type of hardware used.

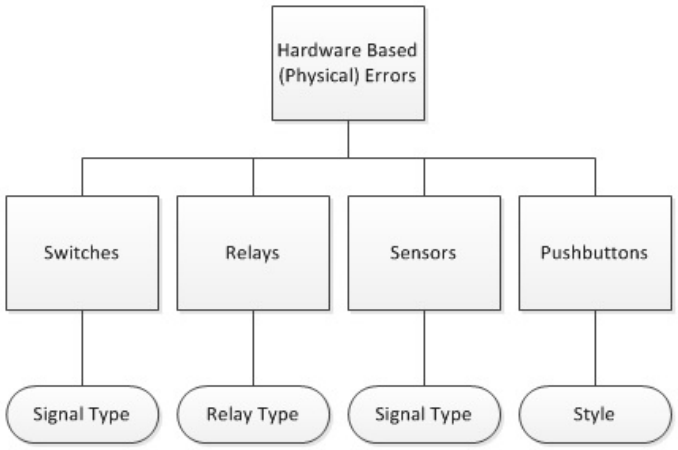


Figure 4.3: Ladder Logic Vulnerability Taxonomy: Hardware

Software Based (Virtual) Errors: The attributes associated with software based (virtual) errors are the error class and error subclass. At this level of the Vulnerability Taxonomy, the possible designation for error class is design-level error. The possible designations for error subclass are logic errors, duplicate objects installed, unused objects, and hidden jumpers.

Figure 4.4 represents the software components of the second level of the Vulnerability Taxonomy.

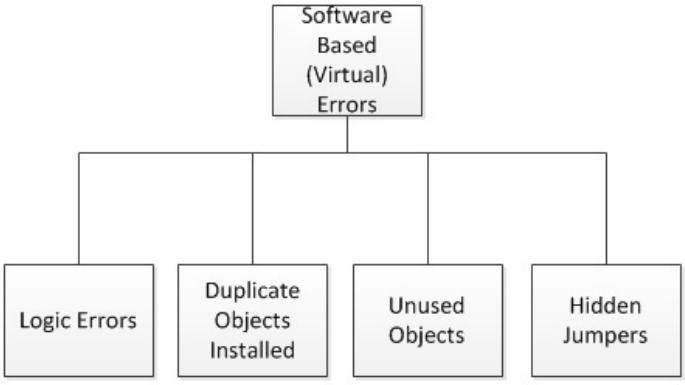


Figure 4.4: Ladder Logic Vulnerability Taxonomy: Software

The software based (virtual) errors are broken down into the subclasses of **logic errors**, **duplicate objects installed**, **unused objects** and **hidden jumpers**. The

single attribute to each of these error classifications will be the error subclass. This attribute, will be used to determine which subclass of each software error type is being addressed. Each of the subclasses of **Software Based (Virtual) Errors** will be addressed in the following sub-sections: sub-section 4.4.1 presents Logic Errors, sub-section 4.4.2 presents Duplicate Objects Installed Errors, sub-section 4.4.3 presents Unused Objects Installed Errors and sub-section 4.4.4 presents Hidden Jumpers Installed Errors. It is also possible that there may be multiple sub-classes referenced simultaneously, at this level, as any given rung of ladder logic has the potential to have multiple security risks which may need to be addressed. We will now define the attributes of each component at the various levels.

4.4.1 Logic Errors

Figure 4.5 represents the logic error subclass. Logic errors are further categorized as **Placement and Element Component Errors** or **Scope and Linkage Errors**. The set of placement and element component errors are broken out as to their location in the PLC ladder logic itself, beginning or end of the rung functions. It should also be noted that between these subclasses, there is a potential of crossover as shown.

Logic Errors: The attribute associated with logic errors is the error subclass. The possible designations of this attribute are placement and element component errors or scope and linkage errors. This attribute will make the determination as to the type of subclass of logic error affected. The component levels of these two classifications are not mutually exclusive. It is possible, for example, that a JSR element could fall into each category, by way of its placement in the ladder logic itself, as well as its potential to contribute to a scope and linkage error.

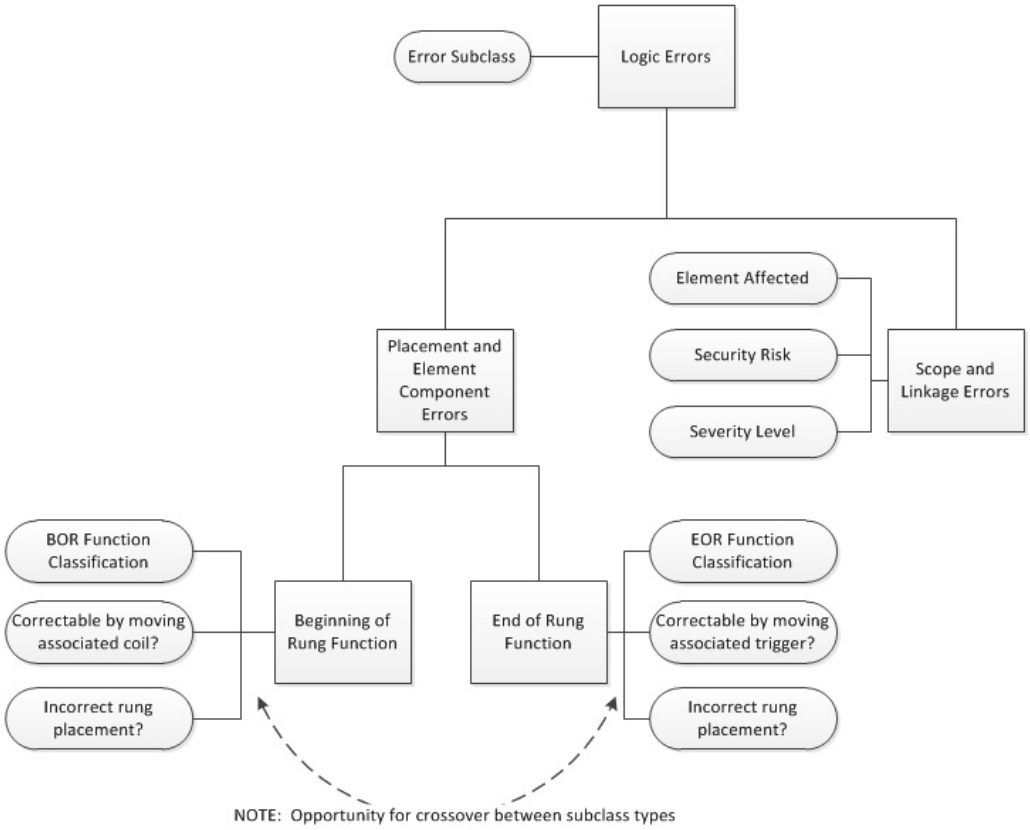


Figure 4.5: Ladder Logic Vulnerability Taxonomy: Logic Errors

Placement and Element Component Errors: The attribute associated with this classification is the PEC error subclass. The possible designations for this attribute are beginning of the rung function or end of the rung function. This attribute will make the determination as to the placement criteria which would be enforced to correct the error and alleviate the security risk.

Scope and Linkage Errors: The attributes associated with this classification are:

- Element affected, which can have the designations of jump to subroutine (JSR), jump (JMP), label (LBL) or return (RTN).
- Security risk, which has the designation of man in the middle attack. These elements, in whole or in part, have the ability to provide a mechanism to open

the door to a man in the middle attack. An attacker could use a misdirected JMP command for example, to insert their own code at the misplaced location and cause multiple errors to occur before the RTN command returned the code to the intended location. This would cause the processor to run the original code up to the point of the JMP, read the inserted code and then continue along the original path, with the possibility of the introduction of a new set of parameters.

- Severity Level, which can have a designation of 'A - D' depending on the type of code alterations inserted into the linked file in question. Each sub-class of Software Based (Virtual) Errors, upon reaching a leaf point, and security risk attribute, will also have the attribute of severity level to associate with the security risk.

Beginning of the Rung Functions: The attributes associated with this classification are:

- BOR Function classification, which can have the designations of normally open or normally closed contacts or comparative functions.
- Correctable by moving the associated coil, which yields a yes or no response.
- Incorrect rung placement, which yields a yes or no response.

Figure 4.6 represents the the first subclass of logic errors which is shown as beginning of the rung functions.

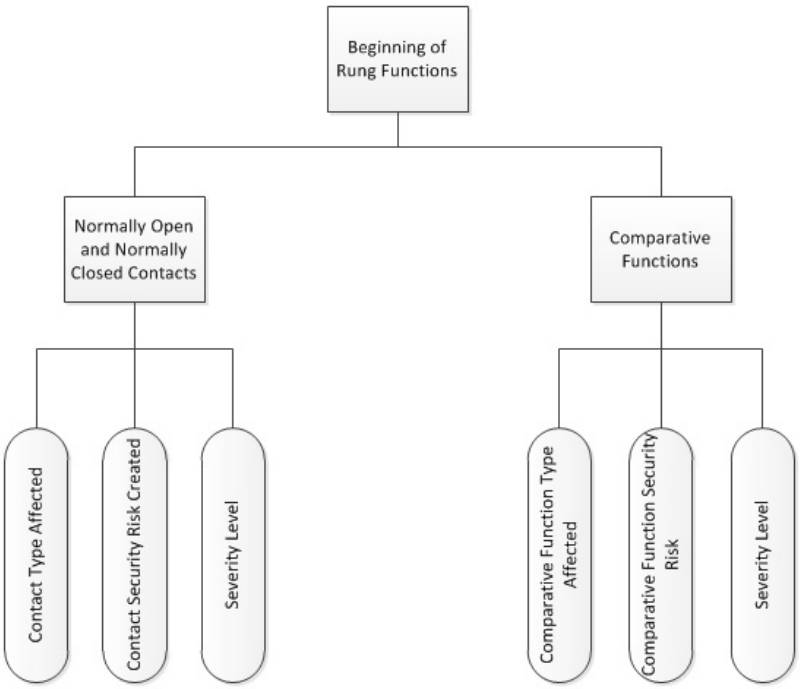


Figure 4.6: Ladder Logic Vulnerability Taxonomy: Beginning of Rung Functions

The beginning of the rung functions are further defined as normally open and normally closed contacts or comparative functions. We will now define the attributes of each beginning of the rung subclass.

Normally Open and Normally Closed Contacts: The attributes associated with normally open and normally closed contacts are contact type affected and contact security risk created. The possible designations of the contact type affected are normally closed or normally open. The possible designations of the contact security risk created are rung blocking or rung bypass.

Comparative Functions: The attributes associated with comparative functions are comparative function type affected and comparative function security risk created. The possible designations of the comparative function type affected are EQU, NEQ, GRT, LES, GEQ, LEQ, LIM. The possible designations of the contact security risk created are rung blocking, rung bypass, delayed start or delayed stop.

End of the Rung Functions: The attributes associated with this classification are:

- EOR function classification, which can have the designations of timer, counter, mathematical or data handling functions, or program flow instructions. This attribute assists in further dissecting the error to a specific element or set of elements. As such, this attribute is non exclusive. It is possible for the static analysis tool to determine the existence of multiple errors within the same rung or set of rungs.
- Correctable by moving the associated trigger, which yields a yes or no response.
- Incorrect rung placement, which yields a yes or no response.

Figure 4.7 represents the the second subclass of logic errors which is shown as end of the rung functions.

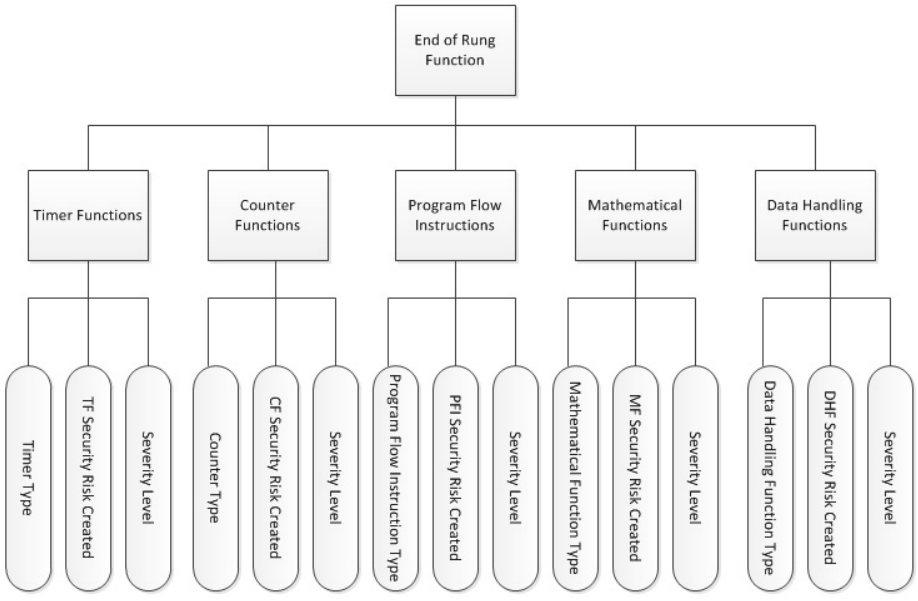


Figure 4.7: Ladder Logic Vulnerability Taxonomy: End of Rung Functions

The end of the rung functions are further defined as timer functions, counter functions, program flow instructions, mathematical functions and data handling functions. We will now define the attributes of each end of the rung subclass.

Timer Functions: The attributes associated with timer functions are the timer type affected, TF security risk created and severity level. The possible designations of the timer type are timer on (TON), timer off (TOF) or retentive timer on (RTO). The possible designations of the TF security risk created are race conditions, premature start, delayed start, premature finish and delayed finish. Severity Level, can have a designation of 'A - D.' Note that with any of these security risks related to timers, it has the ability to affect such things as the heating pattern of a heating process.

Counter Functions: The attributes associated with counter functions are counter type, CF security risk created and severity level. The possible designations of the counter type affected are count up (CTU), count down (CTD) or reset (RES). The possible designations of the CF security risk created are incorrect iterations and quality concerns. Severity Level, can have a designation of 'A - D.'

Program Flow Instructions: The attributes associated with program flow instructions are the program flow instruction type affected, PFI security risk created and severity level. The possible designations of the program flow instruction type are jump (JMP), jump to subroutine (JSR), label (LBL) and return (RET). The possible designation of the PFI security risk created is man in the middle attack. Severity Level, which can have a designation of 'A - D.' These elements, in whole or in part, have the ability to provide a mechanism to open the door to a man in the middle attack. An attacker could use a misdirected JMP command for example, to insert their own code at the mislabeled location and cause multiple errors to occur before the RTN command returned the code to the intended location. This would cause the processor to run the original code up to the point of the JMP, read the inserted code and then continue along the original path, with the possibility of the introduction of

a new set of parameters. Note: Program flow instructions could also be considered as a subclass of scope and linkage errors, depending on how the error was initiated.

Mathematical Functions: The attributes associated with mathematical functions are mathematical function type, MF comparative function security risk created and severity level. The possible designations of the mathematical function type are basic mathematical functions or trigonometric functions. The set of basic mathematical functions is considered to be addition, subtraction, multiplication, division, absolute value and logarithms; while the set of trigonometric functions is considered to be sine, cosine, tangent, arc sine, arc cosine and arc tangent. The possible designations of the MF contact security risk created are numerical (integer, real and floating point) data manipulation and false numerical data received by the SCADA PC. Severity Level, can have a designation of 'A - D.'

Data Handling Functions: The attributes associated with data handling functions are data handling function type, DHF security risk created and severity level. The possible designations of the data handling function type are file fill (FLL), OR, AND, NOT, exclusive or (XOR), move (MOV) and copy (COP). The possible designations of the contact security risk created are binary data manipulation and false binary data received by the SCADA PC. Severity Level, which can have a designation of 'A - D.'

4.4.2 Duplicate Objects Installed

Figure 4.8 represents the subclass of duplicate objects installed. The attributes of the classification of duplicate objects installed are duplicated type, OTE security risk created, JSR security risk created and severity level. We will now define each of these attributes.

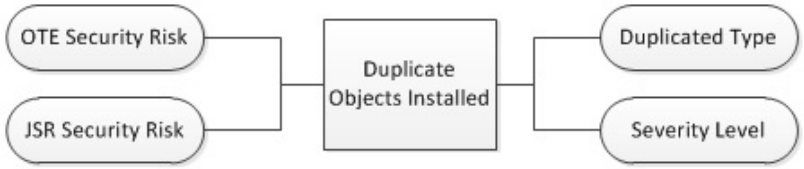


Figure 4.8: Ladder Logic Vulnerability Taxonomy: Duplicate Objects Installed

- Duplicated Type: The possible designations for this attribute are output enable (OTE) or jump to subroutine (JSR).
- OTE Security Risk Created: The possible designations for this attribute are no trigger or false trigger created.
- JSR Security Risk Created: The possible designations for this attribute are no trigger possible or man in the middle attack.
- Severity Level: Severity Level, can have a designation of 'A - D.'

4.4.3 Unused Objects

Figure 4.9 represents the subclass of unused objects. The attributes of the classification of unused objects unused object type, UOT security risk created and severity level. We will now define each of these attributes.

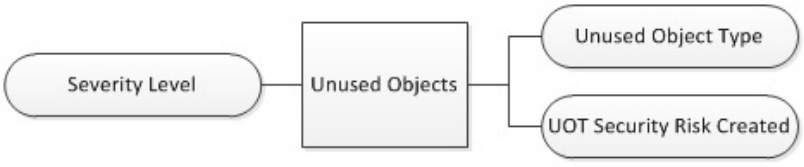


Figure 4.9: Ladder Logic Vulnerability Taxonomy: Unused Objects Installed

- Unused Object Type: The possible designations for this attribute are broad and can be considered any component throughout this taxonomy.

- UOT Security Risk Created: The possible designation for this attribute is open, pre-instantiated entry points. If any if of the components listed throughout this taxonomy are instantiated (created within the data table during the initial design process) and unused in the completed code, they are available for immediate insertion by an attacker upon gaining access to the system with no additional effort required.

- Severity Level: Severity Level, can have a designation of 'A - D.'

4.4.4 Hidden Jumpers

Figure 4.10 represents the subclass of hidden jumpers. Hidden jumpers are further defined as forces or rung jumpers. This distinct is made on the basis of how the hidden jumper is created. A force, is created by activating a mechanism within the PLC CPU which will allow you to override certain elements within the PLC. Rung jumpers are physically written into the ladder logic code itself and allow the user to bypass any number of components in the same rung simultaneously. We will now define the attributes at each of the level in the hidden jumper subclass.

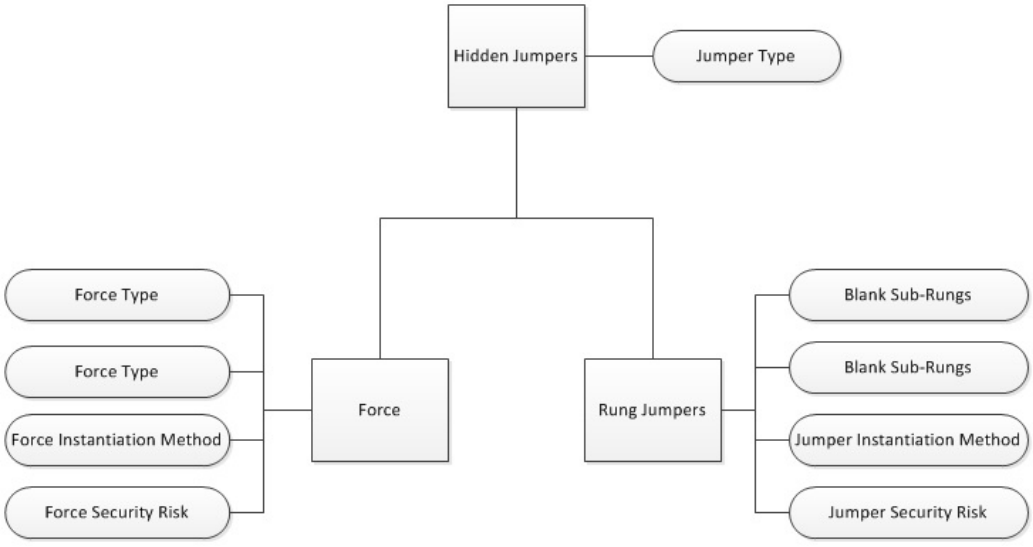


Figure 4.10: Ladder Logic Vulnerability Taxonomy: Hidden Jumpers

Hidden Jumpers: The attribute associated with hidden jumpers is the jumper type. The possible designations of this attribute are force or rung jumpers. This attribute will make the determination as to the type of jumper mechanism which has been created. Theoretically, the component levels of these two classifications are not mutually exclusive. It is possible, for example, to have a forced element condition located within the boundaries of a rung jumper. The static analysis tool would address each of these areas withing the same rung.

Force: The attributes associated with this classification are force type, force instantiation method, force security risk and severity level. The possible designations for each attribute are shown below:

- Force type, can have the designations of force on or force off.
- Force instantiation method, which has the designation of multiple forces detected or single force detected. Knowing the number of forces active, assists the determination of rather the forces were placed as a testing mechanism by the designer, and inadvertently left on, or potentially by a malicious attacker. The fewer forces seen in a given code file, the less likely it was performed by a person with malicious intent.
- Force security risk, which has the designation of 'rung-bypass' or 'element-bypass' depending on the number of forces installed in a single rung. The designation of 'rung-bypass' will be used if a force was used to bypass every element within a given rung. The designation of 'element-bypass will be used if a force was used to bypass a single element within a given rung, when more than one element exists in that rung
- Severity Level, can have a designation of 'A - D.'

Rung Jumpers: The attributes associated with this classification are:

- Blank sub-rungs, which yields a yes or no response.
- Jumper instantiation method, which has the designation of 'blank' or 'contact override'. The designation of blank would be given if a 'yes' response is given to the 'blank sub-rung' attribute. The designation of 'contact override' would be given, if an instance is encountered where there exists a sub-rung which is entirely comprised of contacts which are in a closed state. This also serves as a secondary mechanism to check for the non-existence of an activation coil, as this could be the case if the normally closed contacts never encounter an open state.
- Jumper Security Risk, which has the designations of 'rung-bypass' or 'element-bypass' depending on the type of jumper and degree to which the jumper is installed. The designation of 'rung-bypass' will be used when a rung jumper (branch) is used to bypass every element within a given rung. The designation of 'element-bypass' will be used when a rung-jumper is used only to bypass a single element within a given rung.
- Severity level, which can have a designation of 'A - D.'

4.5 Modeling PLC Vulnerabilities

We have modeled the known vulnerabilities using state transition diagrams. Our approach is motivated by the success of using state transition diagrams to model intrusion patterns [20, 28]. We will create state transition diagrams to represent each vulnerability. The potential errors shown in table 4.3 represent the initial findings of vulnerabilities which we have identified during our research. Using this we generalize

each vulnerability using state transition diagrams. Our approach can be used to model new vulnerabilities that are detected in the future.

4.5.1 State Transition Diagram Analysis: Race Condition

The race condition, as shown in Figure 4.11, occurs as the timer done bit (T4:/DN) opens at the exact same moment that the timer (T4:0) reaches its preset value. This causes the timer to cycle and the process resets, never allowing a shutdown to fully occur. The state transition diagram depicting this race conditions is shown in Figure 4.12. Figure 4.13 shows the corrected state transition diagram for this scenario.

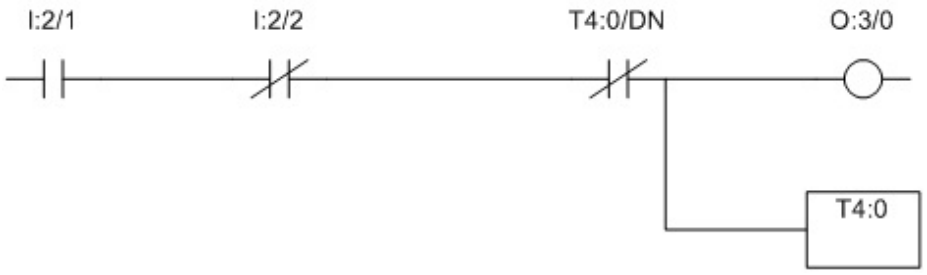


Figure 4.11: Race Condition: Ladder Logic Incorrect

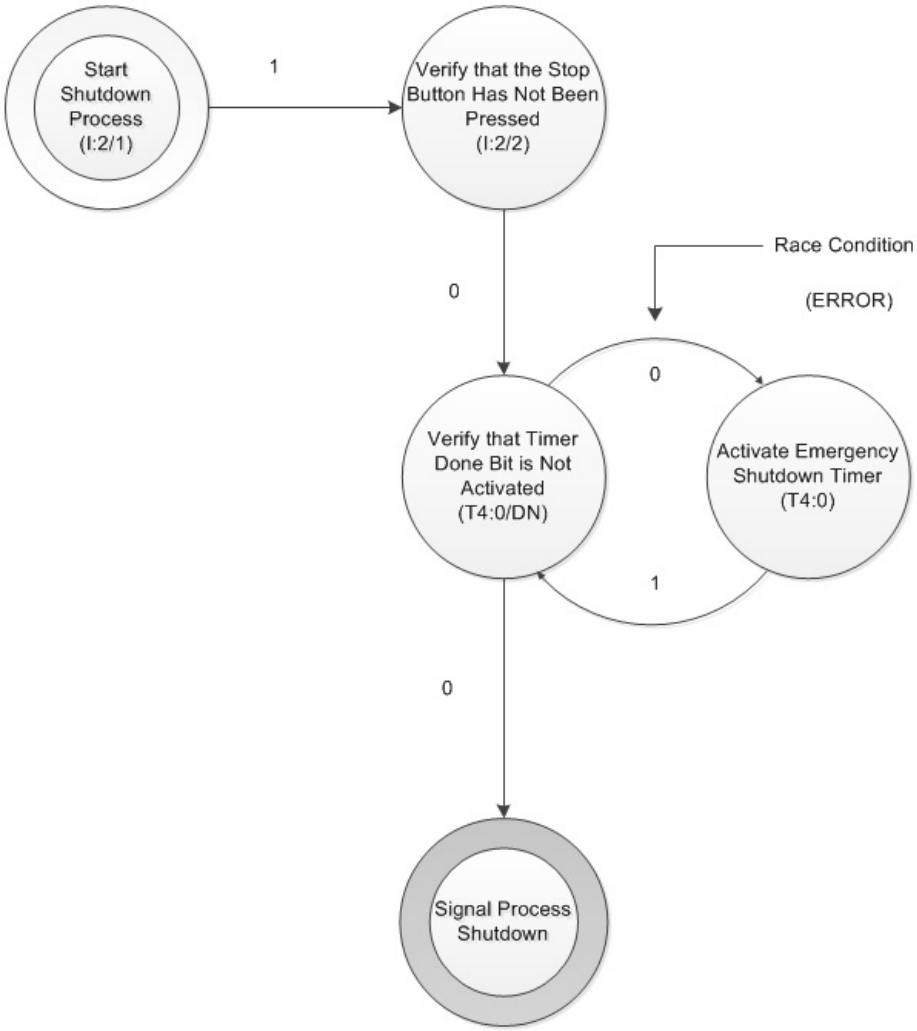


Figure 4.12: State Transition Diagram: Existing Race Condition

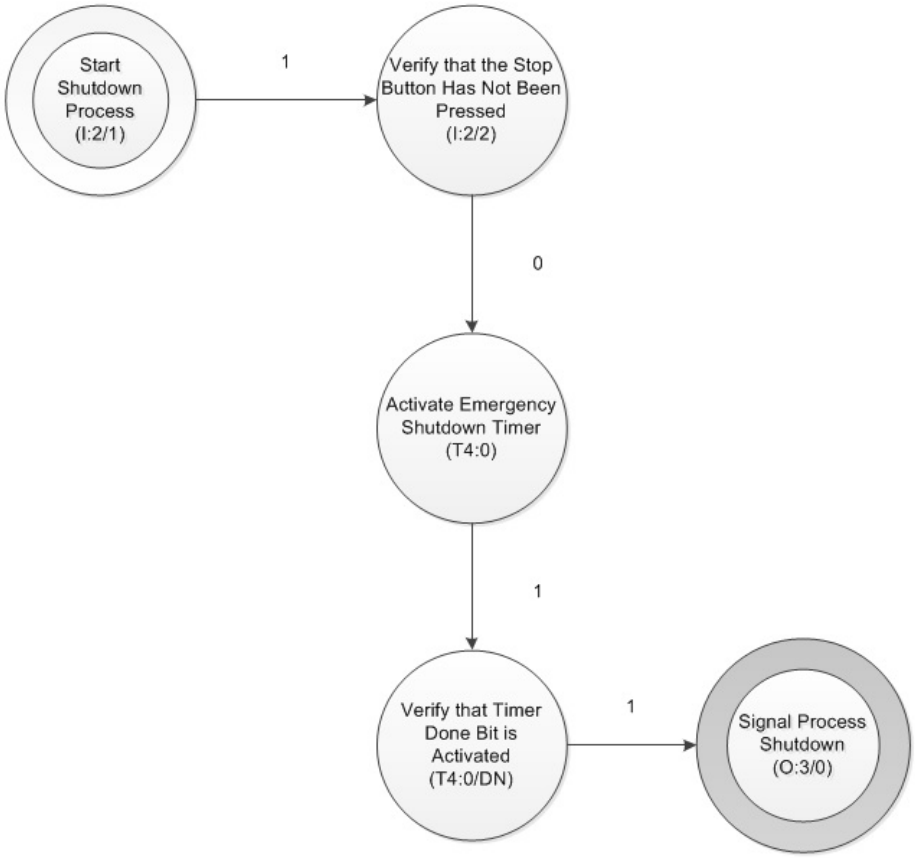


Figure 4.13: State Transition Diagram: Elimination of Race Condition

Figure 4.14 depicts one possible solution to the problem, wherein the shutdown process begins after a preset amount of time; again, as determined by the timer preset value.

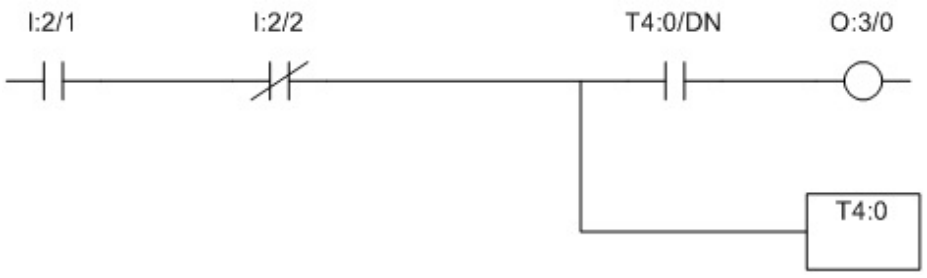


Figure 4.14: Ladder Logic: Elimination of Race Condition

4.5.2 State Transition Diagram Analysis: Comparative Functions

Comparative functions, if incorrectly coded, can insert a security risk such that a malicious user could insert incorrect data into the process through the comparative function. This new data could cause changes in the process sequence or cause the process to be aborted in its entirety. Figure 4.15 shows the state transition diagram which depicts the security risk. Note that if any portion of the comparative function elements are hard coded, the potential for a security risk exists and the altered value would be returned to the system. Figure 4.16 shows the state transition diagram which depicts the correct process flow using a comparative element. The ladder logic structure for this, and the remaining state transition diagrams, are discussed in Chapter 5 during our discussion of the associated design patterns.

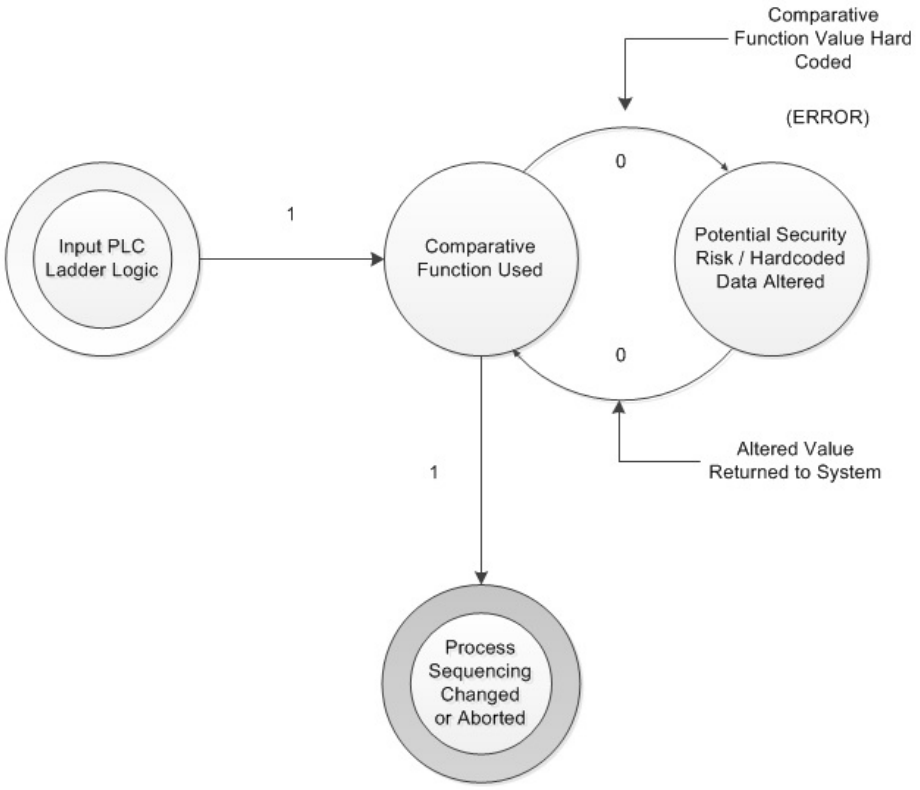


Figure 4.15: State Transition Diagram: Comparative Function Risk

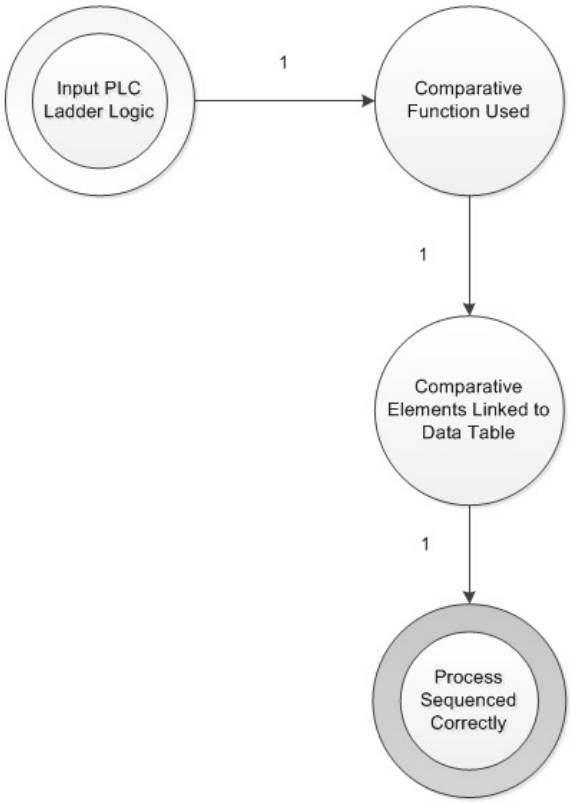


Figure 4.16: State Transition Diagram: Comparative Function Risk Eliminated

4.5.3 State Transition Diagram Analysis: Missing Trigger Coil

A missing trigger coil, which is used to activate a like named contact, will eliminate the activation of a given process. This has the potential security risk of blocking certain embedded safety measures written into the PLC ladder logic program. Figure 4.17 shows the state transition diagram which depicts the security risk. As the loop condition shows, once a given contact is found, the PLC will attempt to pair that contact with its related coil and check the activation status of that coil. If an associated coil cannot be found, the rung in which the contact is located potentially will not activate. Figure 4.18 shows the state transition diagram which depicts the correct process flow in which the contact is correctly paired with its activation coil.

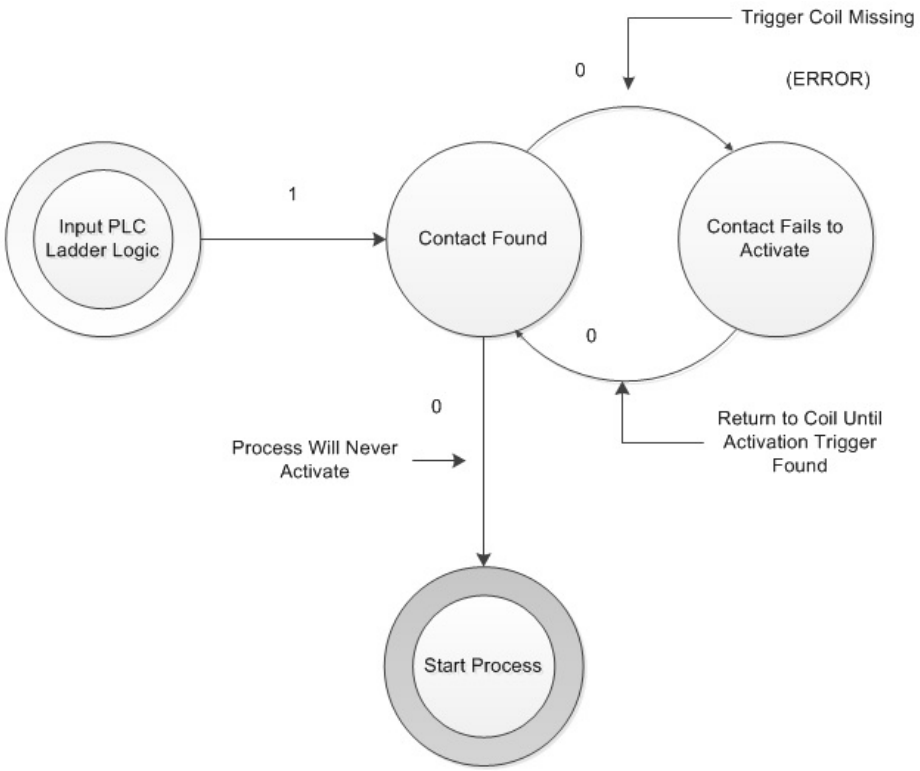


Figure 4.17: State Transition Diagram: Missing Trigger Coil

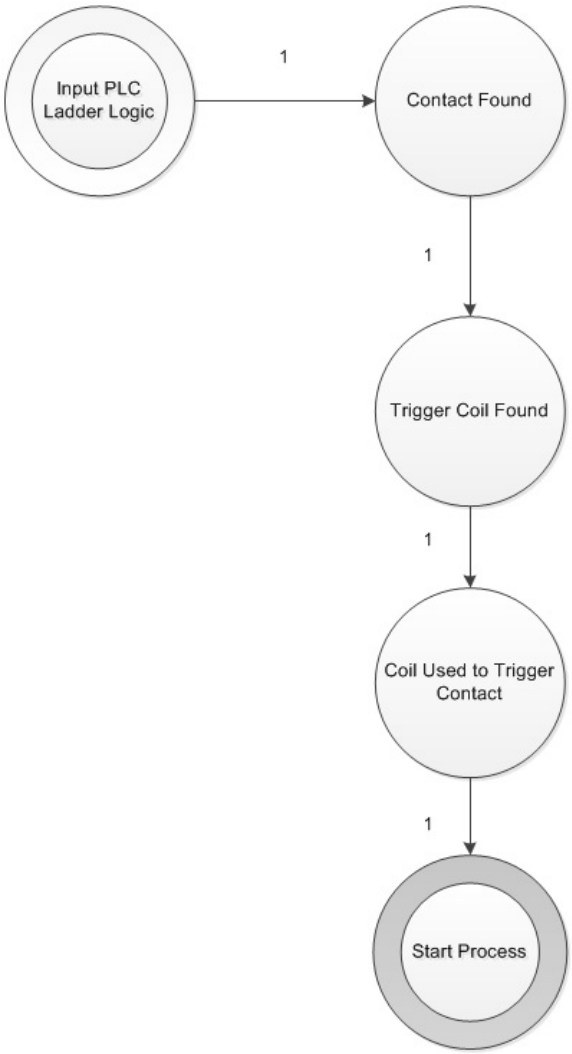


Figure 4.18: State Transition Diagram: Missing Trigger Coil Error Eliminated

4.5.4 State Transition Diagram Analysis: Scope and Linkage Errors

Scope and linkage errors, as they pertain to ladder logic code occur when a file, or part of a file, is accessed due to its unintended availability. This file can currently exist on the PLC itself with the intention of being used for a process variation, or can be inserted by a malicious user. This has the potential security risk of allowing for the insertion of large quantities of malicious data through one entry point. Figure 4.19

shows the state transition diagram which depicts the security risk. In this figure, File A makes an unintended call to file B. File B inserts malicious data into the system by way of code which was instantiated by the attacker. This malicious data is now returned back to file A to continue into the remainder of the process. Figure 4.20 shows the state transition diagram which depicts the correct process flow in which the unintended element is removed.

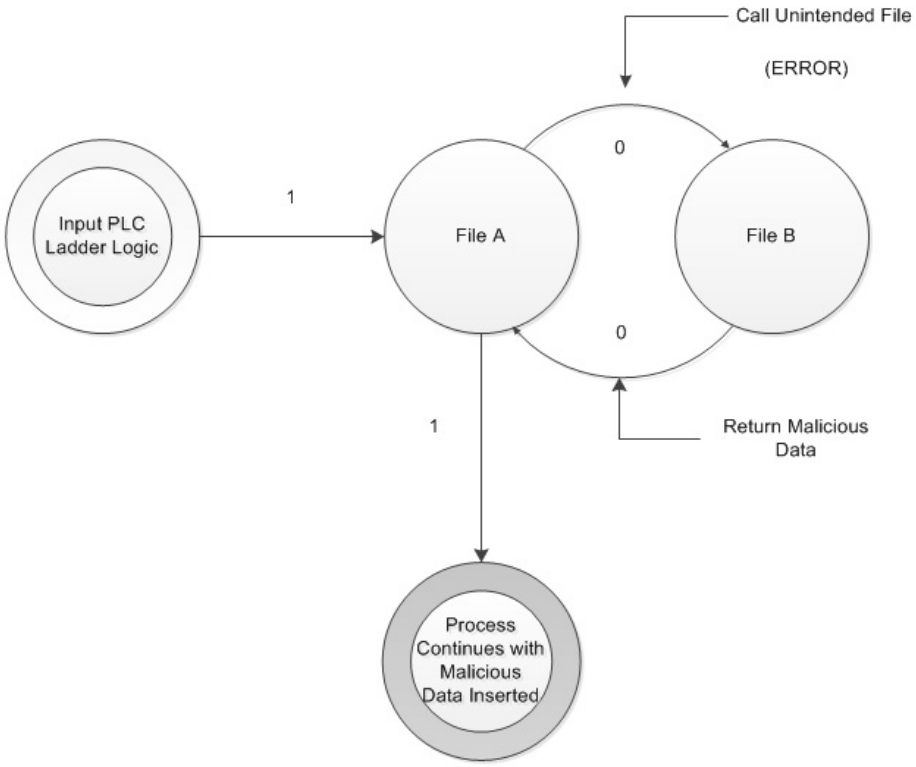


Figure 4.19: State Transition Diagram: Scope and Linkage Risk

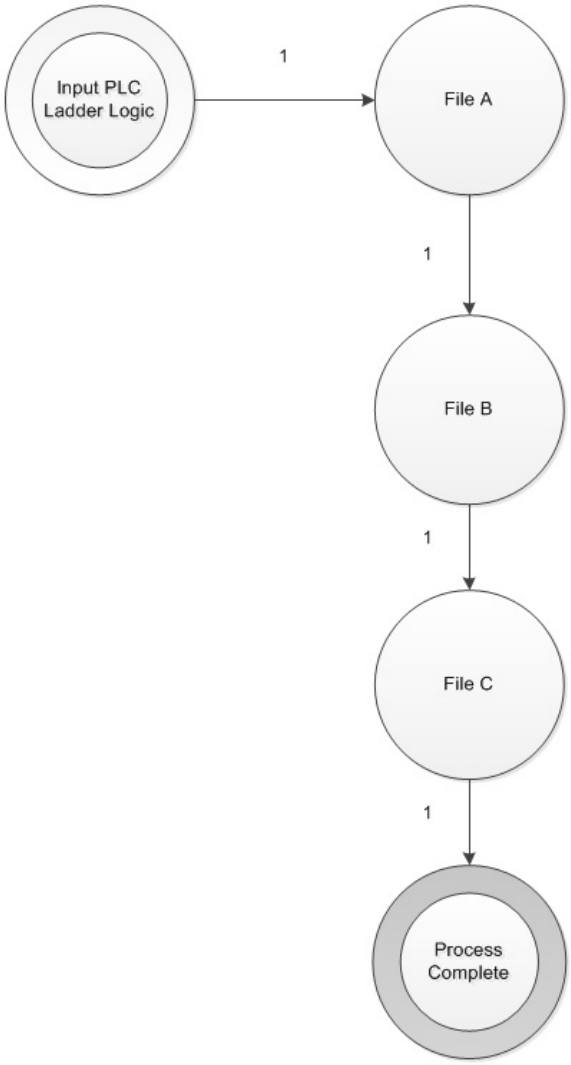


Figure 4.20: State Transition Diagram: Scope and Linkage Risk Eliminated

4.5.5 State Transition Diagram Analysis: Hidden Jumper Inserted

The insertion of what are known as hidden jumpers in ladder logic generally occur in two forms: 1) through the use of override forces which are built into the PLC programming tool itself and 2) through the use of blank sub-rungs, also referred to a branch elements. Hidden jumpers introduce the security risk of allowing sections of rungs to be bypassed and the inadvertent trigger, or refusal of trigger of a given

rungs output coil. Figure 4.21 shows the state transition diagram which depicts the security risk. In this figure, the arrow to the left of the states represents a bypass opportunity for a malicious user to circumvent the intended activation requirements of state two. By circumventing the activation requirements, as stated previously, this allows for the inadvertent triggering, or refusal to trigger, the activation rung. Figure 4.22 shows the state transition diagram which depicts the correct process flow in which the hidden jumper is removed.

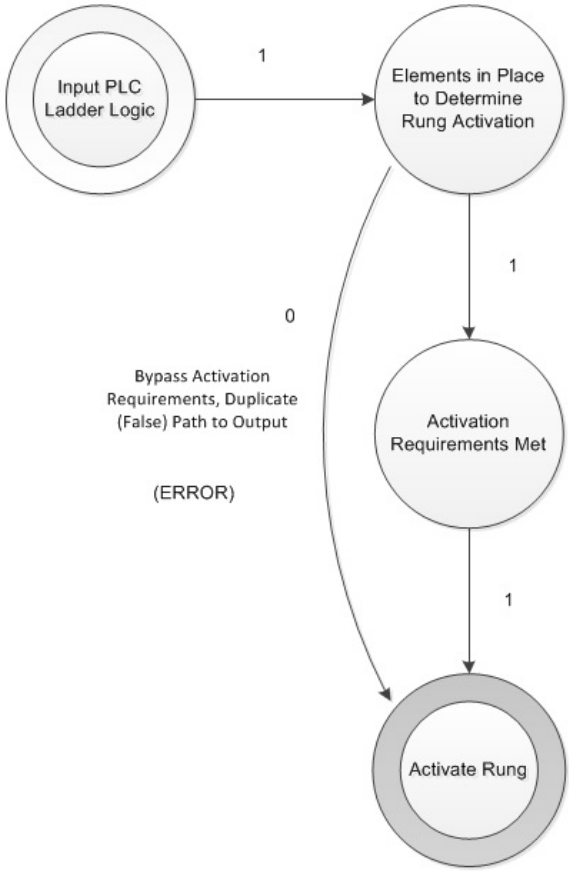


Figure 4.21: State Transition Diagram: Hidden Jumper Risk

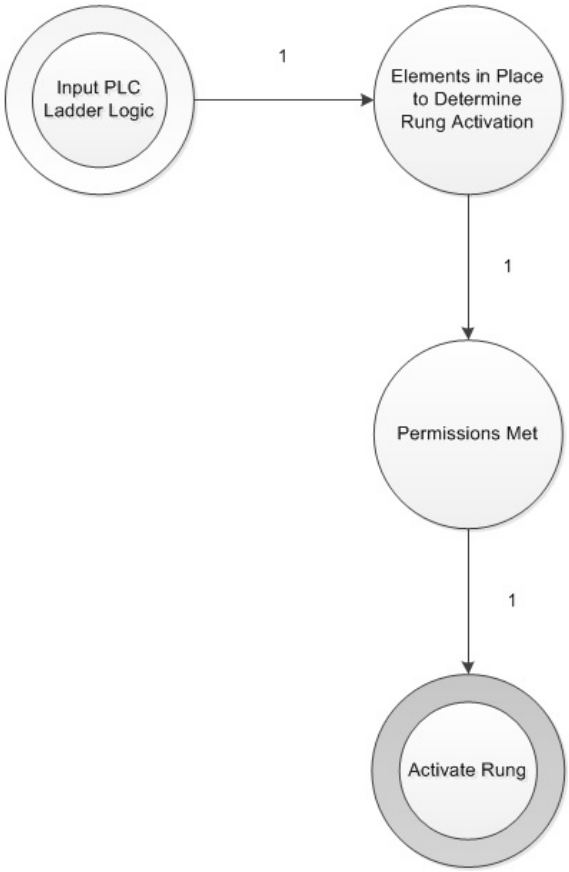


Figure 4.22: State Transition Diagram: Hidden Jumper Risk Eliminated

4.5.6 State Transition Diagram Analysis: Duplicate Object Inserted

Duplicate objects in ladder logic will have one of two detrimental affects on the process: 1) the duplicate object will cause the related contact to never trigger and 2) the duplicate object may cause the system to misfire the contact and start the related rung on an unintended time line. Figure 4.23 shows the state transition diagram which depicts the security risk described in (1) above. In this figure, duplicate trigger coils are found which could be paired with the contact in question, therefore the contact is never triggered and leads to a dead state with each coil. The correct process flow for this duplicate object example is the same as previously shown in Figure 4.18.

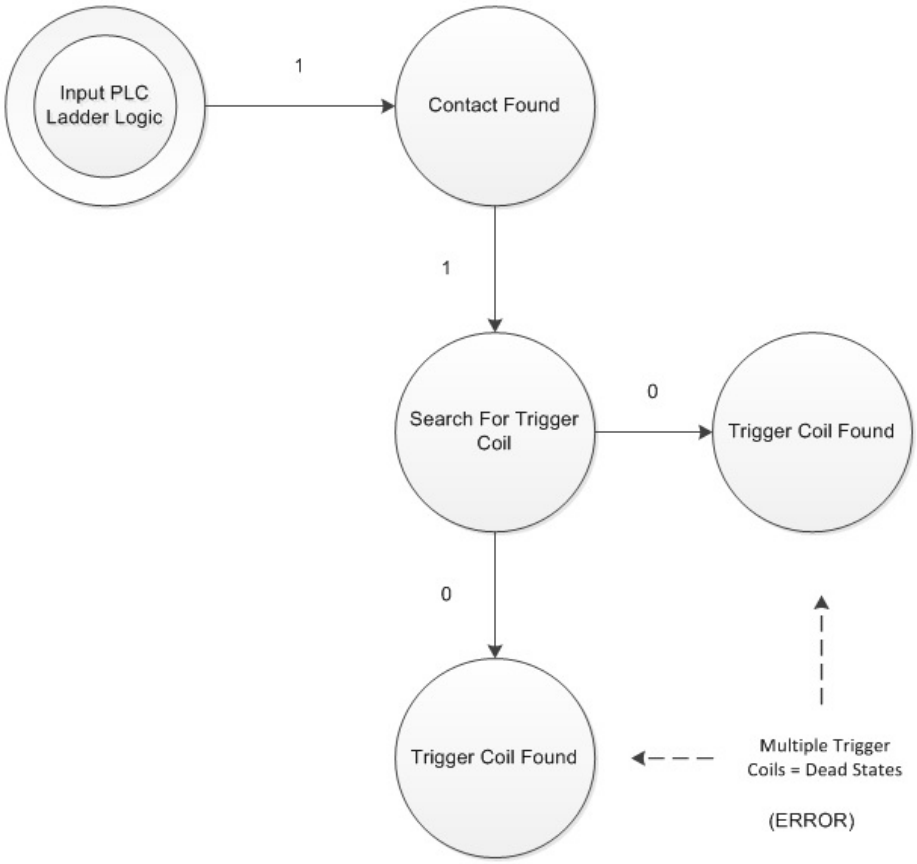


Figure 4.23: State Transition Diagram: Duplicate Object Inserted Risk

We use the state transition models of the PLC code vulnerabilities to evaluate actual PLC code and detect vulnerabilities in this code. We describe our static analysis approach in Chapter 6.

Chapter 5

Supporting Correct Software Development

After the detection and ranking of the vulnerabilities, we provide guidance to the developer to remove the vulnerabilities. For this, we have developed a set of design patterns supporting targeted best practice guidelines for the ladder logic software developer.

It is our intention to guide the developer through the process of eliminating these vulnerabilities. The design patterns will support both novice and experienced users, enabling them to improve their coding [7, 14, 22]. Design patterns have been used successfully for various applications. We apply the same principles in the context of ladder logic. Our goal is to keep these design patterns vendor neutral, not targeting specific industry types or PLC manufacturers. This will allow the adaptation of our approach by all PLC developers.

Section 5.1 gives examples of design patterns for mitigating the various classes and subclasses of the Vulnerability Taxonomy. Section 5.2 discuses the methodology used in the selection of a design pattern (or patterns) to mitigate the software vulnerabilities found. The methodology, which is incorporated into the Static Analysis Tool, allows for the assessment and mitigation of multiple areas within the same rung of logic or multiple rungs of logic. The combination of the design patterns and mitigation methodologies, as created, allow for a greater depth and breath in the overall application of the Static Analysis Tool.

5.1 PLC Security Design Patterns

The following are examples of vulnerability entry points within ladder logic code. We represent the vulnerabilities as a vulnerability pattern. Figure 5.1 shows the relationships between the design patterns associated with the classes of the vulnerability taxonomy.

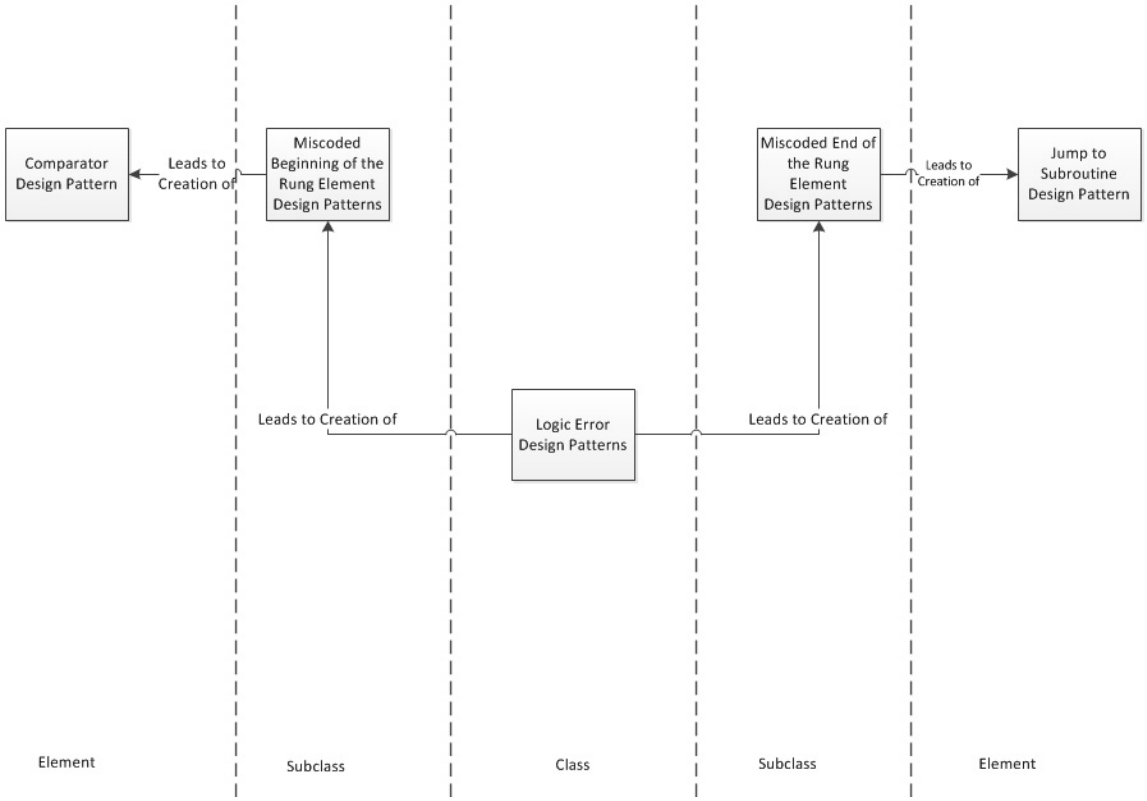


Figure 5.1: Design Pattern Relationships

PLC ladder logic is comprised of multiple switching techniques such as normally open and normally closed switches, timer functions, boolean functions, math routines, branch circuits and jump to subroutine functions. Improper use of these sub-areas have the ability to introduce security risks. Design patterns will be used to give a solution to each specific type of error / security risk outlined in the Vulnerability Taxonomy. We also present the overall connection of the individual patterns. The

Design Pattern Engine, which is incorporated into the Static Analysis Tool, uses information received from the Vulnerability Engine to determine the vulnerability assessed and assign the necessary design pattern to mitigate the vulnerability. The ladder logic code, that has been successfully compiled by the ladder logic compiler, must be loaded into the Static Analysis Tool manually for validation and verification.

In the following sections, we present design patterns to address specific software vulnerabilities. We begin each section with a tabular overview of the problem, the recommended solution and the application of the pattern. The detailed discussion of the pattern follows the table and incorporates a specific example to illustrate the vulnerability as well as the design pattern.

5.1.1 Logic Errors

The classification of logic errors can be split into two sub-classes: 1). Placement and Element Component Errors and 2). Scope and Linkage Errors. These two classifications represent the breath of logic errors encountered within PLC ladder logic code.

Logic errors, if allowed to clear the the design process at compile time, at a minimum introduce reliability issues, but to a greater extent allow for the introduction of security risks. Placement and element component errors can further broken down into the following sub-categories:

- Beginning of rung functions
  - Normally open and normally closed contacts
  - Comparative functions
- End of rung functions
  - Output Enable

- Timer functions
- Counter functions
- Program flow instructions
- Mathematical functions
- Data handling functions

Scope and linkage errors have no sub-classes but specifically represent the following components:

- Jump to subroutine
- Jump
- Label
- Return

The class of logic errors and its sub-classes represent the most common functions and components encountered while writing PLC ladder logic code. As such, these components and functions are the most likely targets of malicious users. This is due both to their high level of availability within the system as well as their potential for crossover between types of platforms, which will allow the malicious user to have a more general knowledge of PLC ladder logic but still accomplish the same goal. We will now show design patterns for sub-classes and components in each of these areas.

5.1.1.1 Placement and Element Component Errors

Table 5.1: Pattern: Comparative Functions Miscoded

Classification	Placement and Element Component Errors
Sub-Class	Beginning of the Rung Elements
Instance	Comparative Functions
Problem	Comparative functions within the PLC ladder logic code, if miscoded, can cause security issues in the form of ladder logic misfiring or availability for numerical data manipulation. This value, by not being driven through a data table location, is considered open and unprotected.
Solution	If the Vulnerability Engine detects a hard-coded value within a comparative function, the user will be required to redirect this value through the data table.
Application	By requiring that the user redirect a hard-coded value through the PLC data table, this will add another level of complexity , and therefore security to the PLC ladder logic.
Impacts	The impact of this solution is in the removal of the vulnerabilities ranging from severity levels A - D.

Table 5.1 shows the instance of **comparative functions**. As shown, a comparative function is part of the **Beginning of Rung Elements** sub-classification, which, subsequently, is part of the **Placement and Component Errors** classification. The table is broken down into the following categories: problem, solution, application and impact. Each of these will now be explained in greater detail.

Problem

As stated, PLC ladder logic elements such as comparators, if miscoded, can allow for the implementation of security issues such as misfiring or numerical data manipulation.

For example:

Assume the following:

- In a given ladder logic sequence, a normally open contact is supposed to trigger the initialization of a high pressure boiler.
- There is a comparative function that states that as long as integer 'A' is less than 'B' then the heating process will continue with no interruption.

If the comparative sequence is coded such that comparative element 'B' is hard coded with a numerical value (as opposed to referencing a data table location), this allows for a security entry point into the system by allowing this hard coded number to be changed directly in the component itself. Figure 5.2 shows the vulnerability pattern and Figure 5.3 shows the ladder logic equivalent.

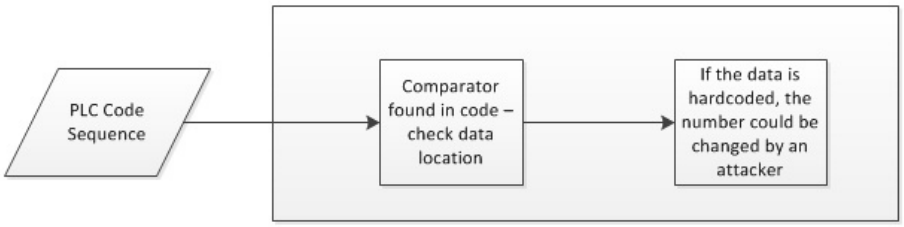
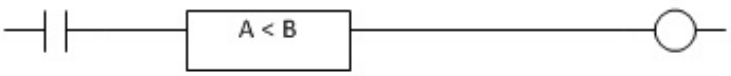


Figure 5.2: Pattern: Hard Coded Value Vulnerability



For comparator:

Element	Data Link	Value
A	I:2 / 7	Determined by analog input point, read from table
B	25	25 (Hard Coded)

Figure 5.3: Comparator with Hard Coded Element

Solution

If the Vulnerability Engine detects that there is a comparative function in the ladder logic, it will note the occurrence and require that the user verify that each element of the comparative function points to a location in the PLC data tables. Figure 5.4 shows the design pattern and Figure 5.5 shows the ladder logic equivalent.

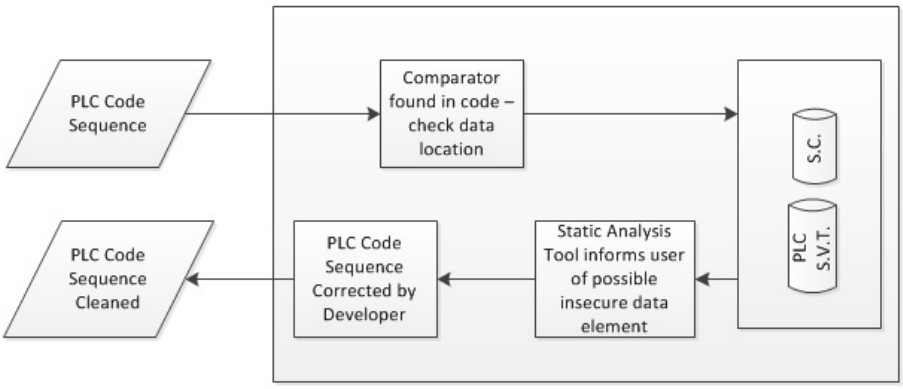
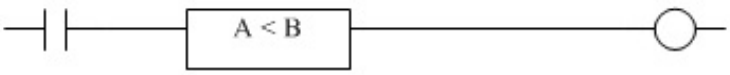


Figure 5.4: Pattern: Elimination of Hardcoded Value Vulnerability



For comparator:		
Element	Data Link	Value
A	I:2 / 7	Determined by analog input point, read from table
B	N7:2	Determined from integer database file

Figure 5.5: Compartor with Data Table Directed Elements

Application

In the example shown, the Design Pattern Engine will be used to provide descriptive input, in the form of noting the occurrence and requiring the solution described. This solution should also be placed in the best practices guide for future reference and training and development purposes.

Impacts

The impact of this solution is in the use of the tables themselves as the defense mechanism, ensuring that no hard coded value can violate the intended functionality.

Table 5.2: Pattern: Trigger Bit Missing

Classification	Placement and Element Component Errors
Sub-Class	Beginning of the Rung Elements
Instance	Normally Closed Contacts, Normally Open Contacts
Problem	Normally open and normally closed contacts are the backbone of all PLC code, regardless of manufacturer or type. These contacts are the integration point between every other type of functional component within the PLC ladder logic code. If one of these contacts (triggers) is installed without a corresponding trigger coil, then the desired outcome of the rung in which the contact exists will never occur.
Solution	Each contact withing the ladder logic code is compared against the available set of outputs currently available. If no trigger coil exists, or more than one, then the Vulnerability Engine would announce the vulnerability and direct the user to the contact that is currently lacking a trigger mechanism.
Application	By assuring that each end of rung element, which affects other like named bits, has a corresponding trigger bit association, we alleviate the scenario which would allow an attacker to place a random trigger within the code due to lack of a matching pair.
Impacts	The impact of this solution is in the removal of the vulnerabilities ranging from severity levels A - C.

Table 5.2 shows the instances of **normally open and normally closed contacts**. As shown, normally open and normally closed contacts are part of the **Beginning of Rung Elements** sub-classification, which, subsequently, is part of the **Placement and Component Errors** classification.

Problem

As stated, the normally open and normally closed contacts are the integration point between every other type of functional component within the ladder logic code. These root level components are necessary for every type of rung that may be created. However, these contacts are only functional, above and beyond their initial state, if a triggering mechanism exists which would allow for a change of their state condition to occur.

For example:

Assume the following:

- In a given ladder logic sequence, an output enable (O:3/4) is supposed to trigger the initialization of an emergency shutdown procedure.
- The fault routine has a normally open contact (O:3/4) associated with the output enable which would trigger the beginning of the emergency shutdown procedure (O:3/5).
- The output enable component (O:3/4) that would trigger this event has been left out, or removed from, the ladder logic code.

This scenario will yield one of two end results to occur:

- The emergency shutdown procedure would be continually activated, due to a lack of a triggering mechanism (in the case of a normally closed contact insertion).
- The emergency shutdown procedure would never activate, due to a lack of a triggering mechanism (in the case of a normally open contact).

Figure 5.6 shows the vulnerability pattern and Figure 5.7 shows the ladder logic equivalent.

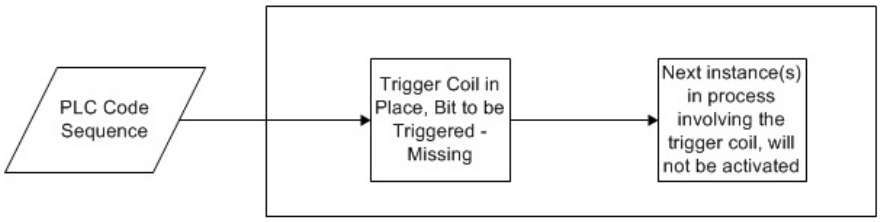


Figure 5.6: Pattern: Missing Trigger Bit Vulnerability

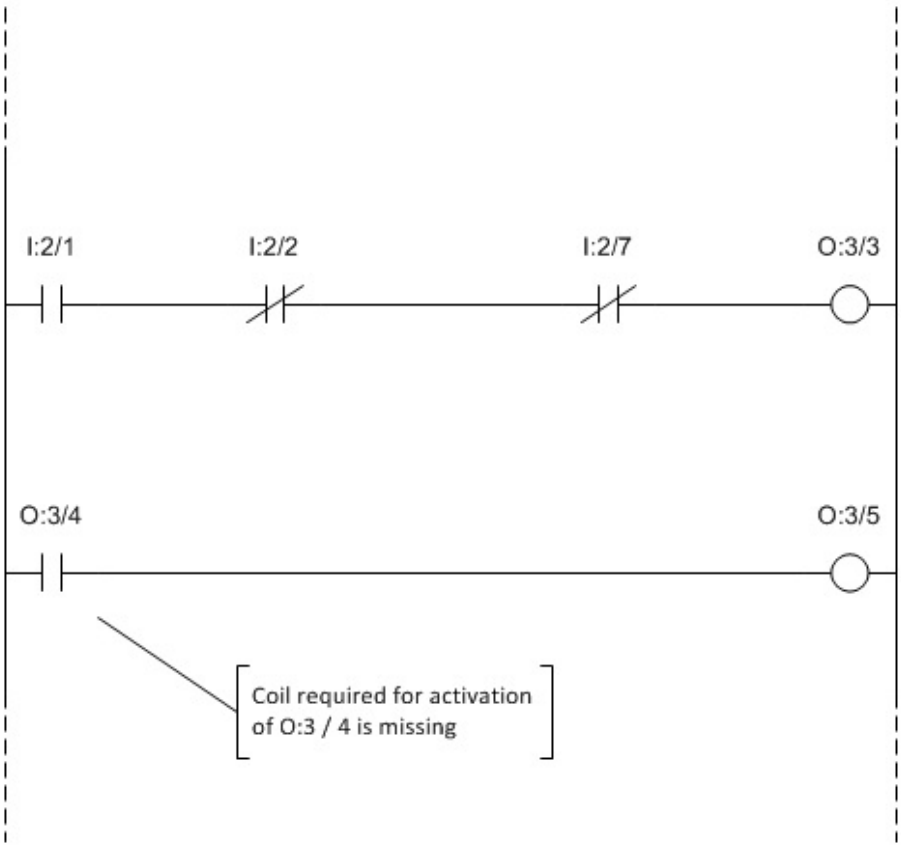


Figure 5.7: Missing Trigger Bit Ladder Logic

Solution

As stated, the Static Analysis Tool will determine if at least one associated contact exists for each trigger coil. If an association doesn't exist, the user is instructed to add the necessary contact to the ladder logic file, in its correct location, or to delete the trigger coil mechanism. An example will be shown to the user, through the Static Analysis Tool as to the intent of the correction. Figure 5.8 shows the design pattern and Figure 5.9 shows the ladder logic equivalent.

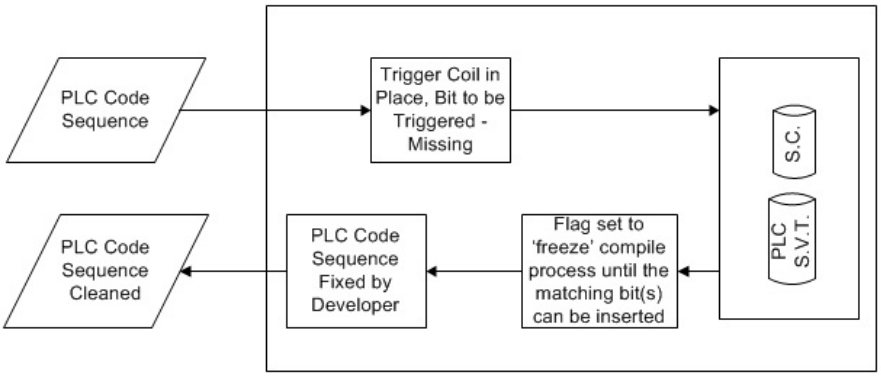


Figure 5.8: Pattern: Elimination of Missing Trigger Bit Vulnerability

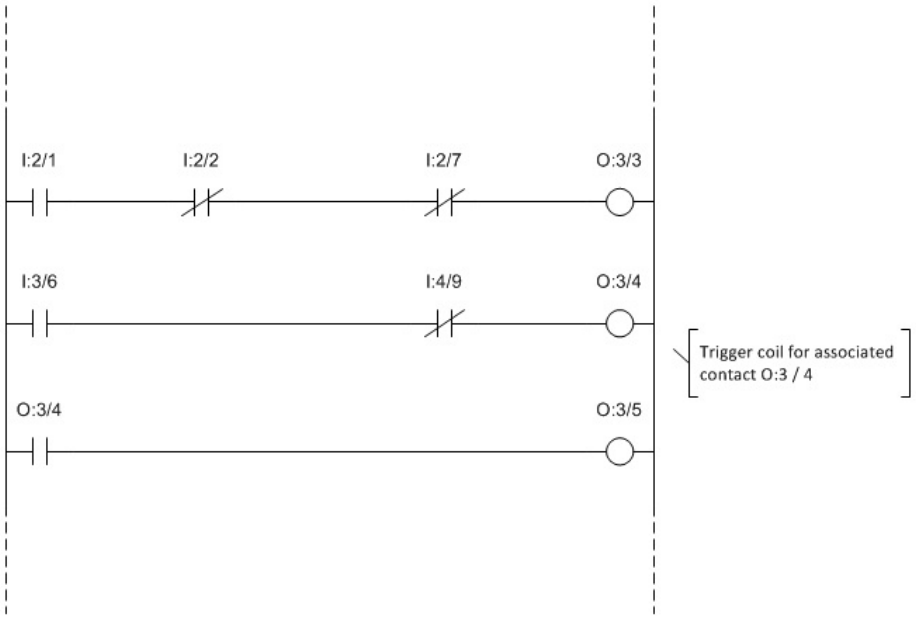


Figure 5.9: Missing Trigger Bit Corrected

Application

In the example shown, the Design Pattern Engine will be used to provide descriptive input, in the form of noting the occurrence and suggesting a placement solution. This will be a suggested placement by way of example, based on the intent of the coil in question. The correct placement location would only be known by the developer.

Impacts

The impact of this solution is in assuring that at one software trigger coil is inserted for every software based contact, preventing potentially catastrophic occurrences ranging from severity levels A - D.

Table 5.3: Pattern: Timer Race Condition

Classification	Placement and Element Component Errors
Sub-Class	Beginning of the Rung Elements
Instance	Normally Closed Contacts, Normally Open Contacts (Timer Done Bits)
Problem	Incorrect placement of a timer done bit element can cause the process involving the timer done bit and the timer itself to go into a race condition.
Solution	As a timer is encountered in a ladder logic rung, the timer done bit will be paired with that timer and a determination will be made as to the correctness of the current placement. This will be accomplished with the Design Pattern Engine of the Static Analysis Tool.
Application	The Vulnerability Engine, upon determining that a race condition exists between the timer contact and its triggering function, will relay that information to the Static Analysis Tool. The Static Analysis Tool will use the Design Pattern Engine to suggest a corrected outcome to eliminate the race condition.
Impacts	The impact of this solution is in the removal of the vulnerabilities ranging from severity levels A - D.

Table 5.3 shows the instances of **normally open and normally closed contacts (timer done bits)**. As shown, normally open and normally closed contacts are part of the **Beginning of Rung Elements** sub-classification, which, subsequently, is part of the **Placement and Component Errors** classification.

Problem

As stated, the normally open and normally closed contacts which are used to create timer done bits such as T4:0/DN, if not properly placed, can cause a race condition to occur. This race condition occurs when the timer done bit becomes a required

element in activating its own triggering mechanism. This would cause a continual oscillation between the active and inactive states of the element in question, thus causing a race condition scenario. Currently, the compiler would allow the ladder logic to be compiled and the continual oscillation of the on/off state, as described above, to occur. Furthermore, this could lead to consequences such as improper event sequencing in a fault routine, or the inability of a process to shut down. An example of this scenario was shown in Chapter 4, Figure 4.11. The vulnerability pattern for the race condition is shown in Figure 5.10.

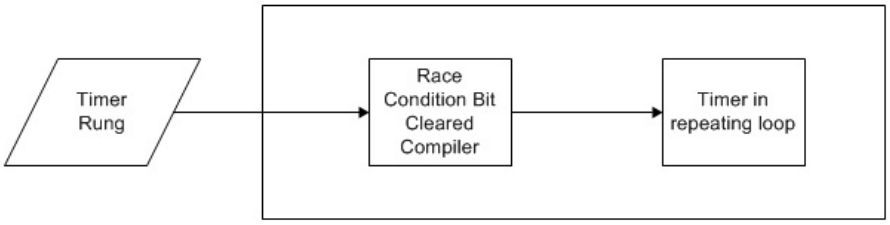


Figure 5.10: Timer Race Condition Vulnerability

Solution

As stated, the Static Analysis Tool, upon determining the existence of the race condition, will alert the developer of the potential security risk and suggest corrective measures. The best correct measure in this scenario is that of moving the timer time bit in such a position that it is not a direct contributor to its own triggering mechanism. Figure 5.11 shows the associated design pattern for the race condition.

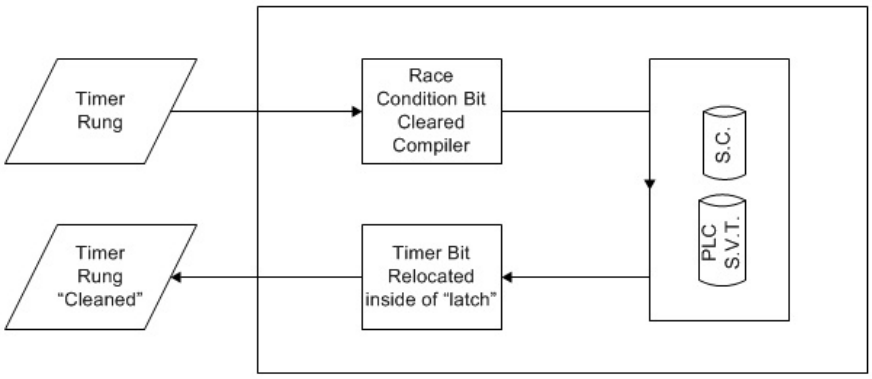


Figure 5.11: Pattern: Timer Race Condition

Application

In the example shown, the Static Analysis Tool will use the Vulnerability Engine in making the determination of the existence of the vulnerability. The Design Pattern Engine will then be used to suggest the correct procedure to follow to alleviate the error and possible security risk.

Impacts

The impact of this solution is the correction of the race condition by using correct placement of components throughout each rung of logic. By alleviating the race condition, this could avert failures in the sequencing of alarm and shutdown processes.

5.1.1.2 Scope and Linkage Errors

Table 5.4: Pattern: Scope and Linkage Errors

Classification	Scope and Linkage Errors
Sub-Class	N/A
Instance	Jump to Subroutine (JSR), Jump (JMP), Label (LBL), Return (RTN)
Problem	Missing or miscoded jump to subroutine (JSR) functions introduce a security risk which could allow a malicious user to introduce an unintended subroutine. In the case of a missing JSR, the JSR could be inserted and a non-intended location referenced. Whereas, in the case of a miscoded JSR, the unintended subroutine could be introduced using the location currently pointed to by the JSR.
Solution	To detect a missing JSR function, it is necessary to determine the number of files available and then query the location that each existing JSR is directed toward. The Static Analysis Tool can detect the number of possible files that exist and then track the pointer locations of each JSR. If a file exists that has no JSR pointing to that file, then the file is currently non-functional and should be removed from the PLC CPU.
Application	By removing the file in question from the PLC CPU, a possible entry point for a man in the middle attack is averted. If this file is necessary for a future use application, this file can be saved to a removable storage device such as a disk or EEPROM card.
Impacts	The impact of this solution is in the removal of the vulnerabilities ranging from severity levels A - D.

Table 5.4 shows the instances of **Jump to Subroutine (JSR), Jump (JMP), Label (LBL) and Return (RTN)**. As shown, these instances are part of the **Scope and Linkage Errors** classification.

Problem

A miscoded JSR function points to an unintended location and could introduce functionality, or data to the system, that is unintended. A JSR function is required to jump to a new file location, but upon reaching the end of that file, the jumper automatically returns to the line following the JSR as if nothing has occurred. A missing or miscoded jump to subroutine (JSR) function introduces a security risk which could allow a malicious user to introduce an unintended subroutine. In the case of a missing JSR, the JSR could be inserted and a non-intended location referenced. Whereas, in the case of a miscoded JSR, the unintended subroutine could be introduced using the location currently pointed to by the JSR. This type of security risk is similar to a man in the middle attack.

For example:

Assume that a process reacts differently depending on various environmental factors such as varying degrees of temperature.

In the spring and summer months a process may function based on a ladder logic subroutine that is tempered for high levels of heat. In the fall and winter months the same process may have been coded to take into consideration freezing temperatures.

To ensure that the ladder logic developer doesn't have to re-write the code each time that there are these environmental issues, most likely two subroutines are written, and one is pointed to based on seasonal changes. This means that each of the subroutines are most likely stored on the same processor throughout the year. This allows for an extra location for a JSR function to point toward at any given time. This is critical in that if this file is first modified and then the pointer redirected, then the new data would be used. Again, very similar in principal to a man in the middle attack. Figure 5.12 shows the vulnerability pattern and Figure 5.13 shows the ladder logic equivalent.

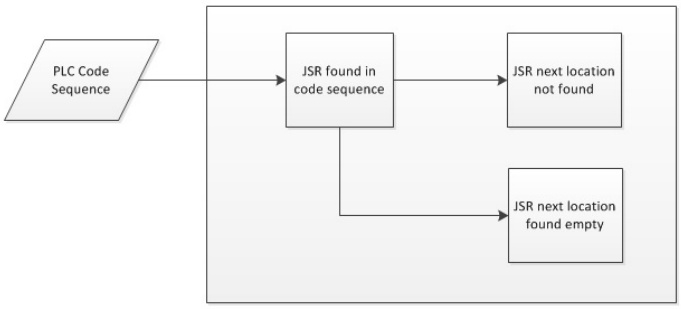


Figure 5.12: Pattern: JSR Vulnerability

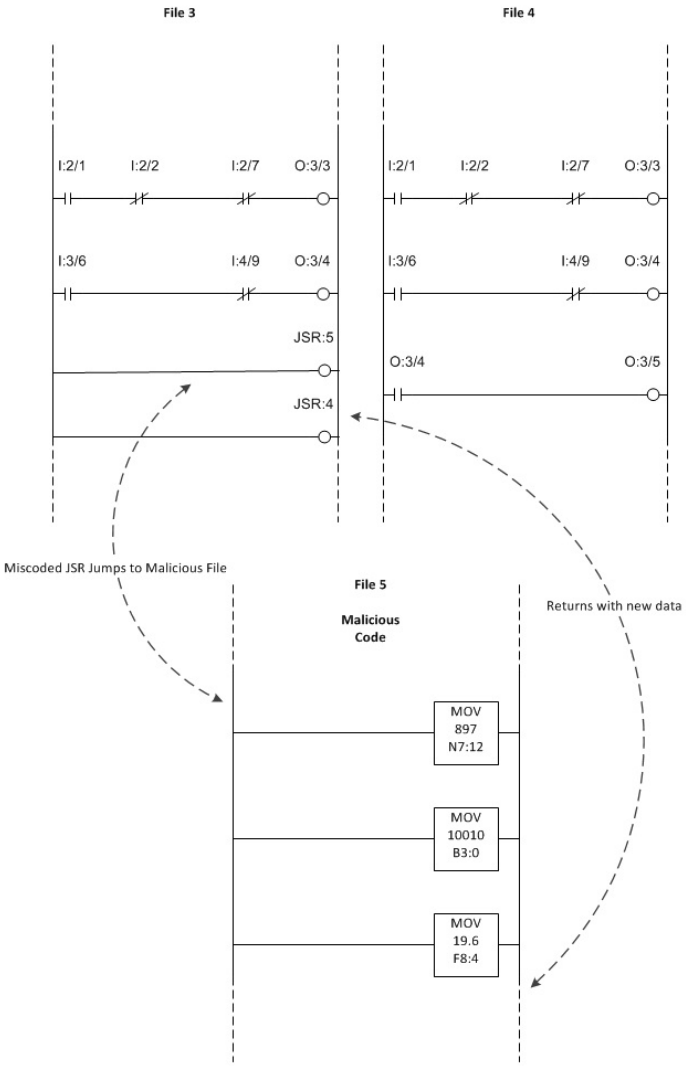


Figure 5.13: JSR Man in the Middle Attack

Solution

The Static Analysis Tool uses a two prong approach in determining the existence of the JSR vulnerability. First, the existence condition is checked to verify that a file exists which correlates to the JSR function. Once it is determined that a file does exist which correlates to the JSR function, the file is then verified to be non-empty. Figure 5.14 shows the design pattern and Figure 5.15 show the ladder logic equivalent.

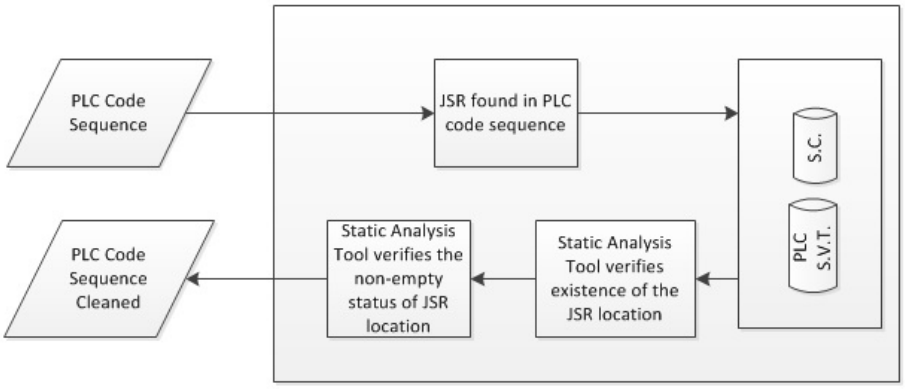


Figure 5.14: Pattern: Elimination of Incorrect JSR

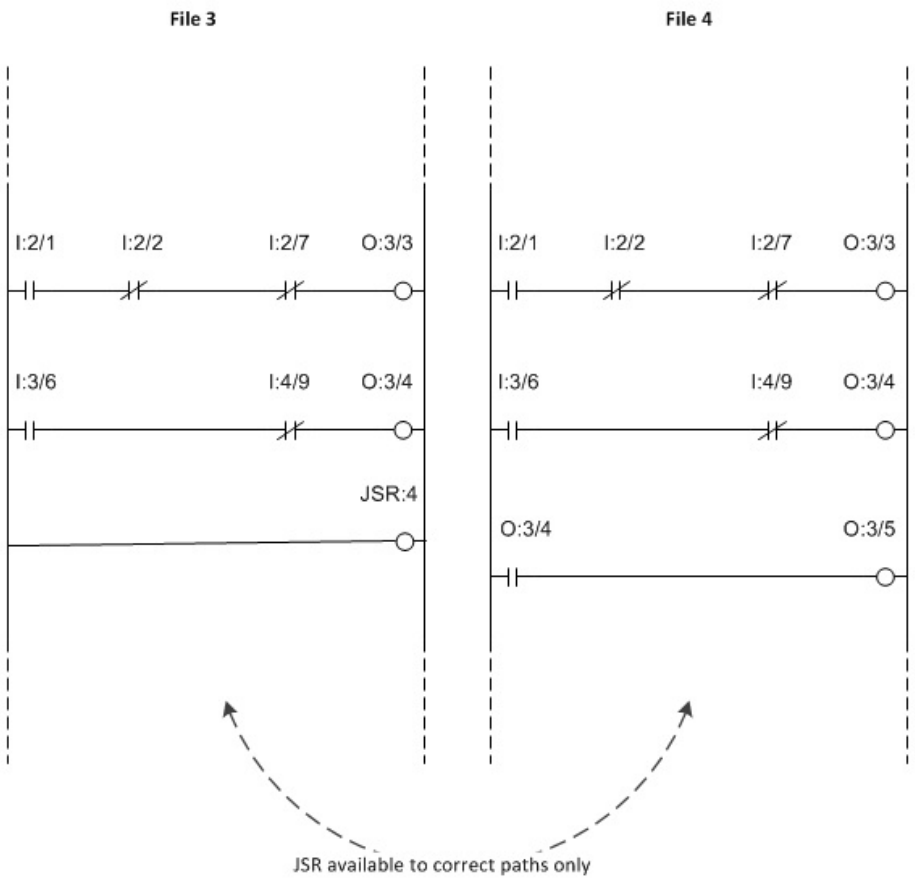


Figure 5.15: PLC Code After Elimination of Security Risk

Application

In the example shown, the Design Pattern Engine will be used to provide the two prong solution to the JSR vulnerability. The user will be instructed to insert the appropriate file being referenced and to ensure that the file is non-empty.

Impacts

As noted in the example, a common usage of the JSR routine is to jump between various functional needs in a given process without the need to add and subtract code as the process needs change. The greatest impact that will be a direct result of the application of our Static Analysis Tool will be the need to store the excess files

in a location separate from the PLC CPU. Although we recognize that there may be an additional security risk in having this information on transportable media, this security concern can be alleviated by keeping the removable storage in a locked cabinet, for example. The removable media risk is not nearly as great as the online attack risk which could occur, as it requires physical access to the facility in question.

5.1.2 Duplicate Objects Installed

Table 5.5: Pattern: Duplicate Objects Installed

Classification	Software Based Errors
Sub-Class	Duplicate Objects Installed
Instance	Timers, Counters, Output Enable, JSR
Problem	Ladder logic code requires a one to many relationship between potential output (trigger functions) and the elements that they trigger. If there become a many to many relationship between these two components, then the security risks which exist are in the areas of potential misfiring of the ladder logic code affected, or denial of access by either trigger mechanism.
Solution	The Vulnerability Engine of the Static Analysis Tool will determine the existence of more than one trigger mechanism within a complete set of PLC ladder logic code. If duplicate triggers are found, the developer is alerted to the existence of the multiple triggering elements and forced to eliminate all but one of these occurrences.
Application	Utilizing the Static Analysis Tool, if duplicate triggers are found, the developer is alerted to the existence of the multiple triggering elements and forced to eliminate all but one of these occurrences.
Impacts	The impact of this solution is in the removal of the vulnerabilities ranging from severity levels A - D.

Table 5.5 shows the instances of **Timers, Counters, Output Enable and JSR**. As shown, these instances are part of the **Duplicate Objects Installed** sub-classification, which, subsequently, is part of the **Software Based Errors** classification.

Problem

As stated, the relationship between a triggering function and the elements (contacts) that are triggered, must be a one to one or one to many relationship. Figure 5.16 shows this relationship constraint.

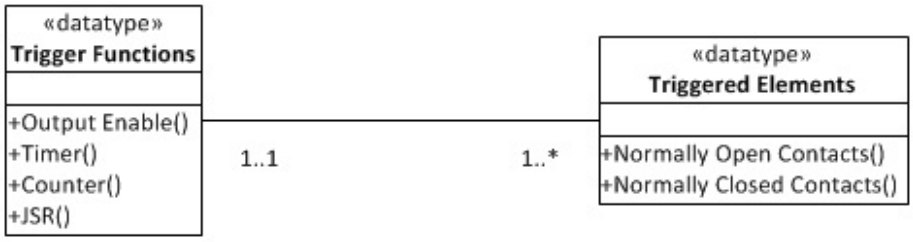


Figure 5.16: Trigger Function to Element Relationship

For example:

Assume that contact O:3/0 is designed such that once it is triggered, it would begin the shutdown process on a high pressure boiler. The output associated with this process is O:3/4.

Further assume that within the complete set of ladder logic files which run this process, there has been a duplicate trigger point installed attempting to trigger O:3/4.

This scenario has the potential to cause a two fold security risk, if the malicious user installs a dual trigger as described.

- The identical trigger, depending on multiple factors such as ladder logic process rate, may trigger the O:3/0 prematurely or later than intended.
- With two identical trigger mechanisms installed, it is also possible that the processor would find itself at an impasse and fail to instantiate either trigger mechanism, therefore failing to shutdown the high pressure boiler as required.

Figure 5.17 shows the vulnerability pattern of duplicate objects installed.

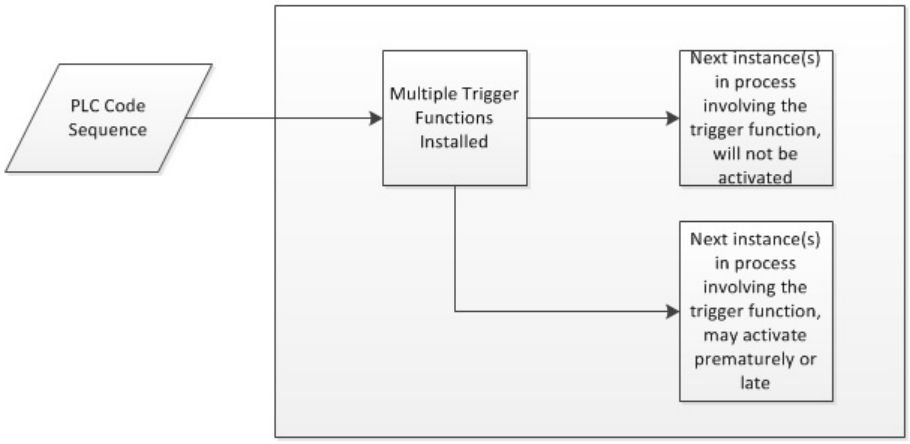


Figure 5.17: Pattern: Duplicate Objects Installed Vulnerability

Solution

The Static Analysis Tool will be used in determining the existence of duplicate objects installed within a PLC ladder logic file. Each time that a non-duplicatable element is found within the PLC ladder logic code, it is compared against like items in the remainder of the code. If a duplicate item is found, the Static Analysis Tool with present the user with the correct design pattern and require that they remove one of the offending duplicate objects. Figure 5.18 shows the correct duplicate object installed design pattern.

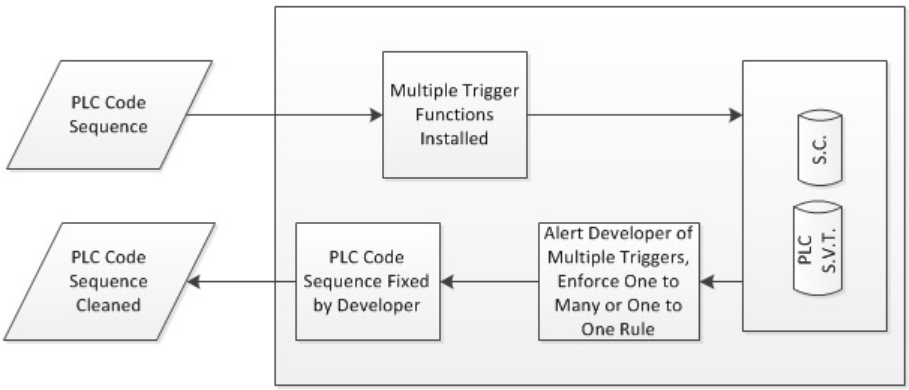


Figure 5.18: Pattern: Elimination of Duplicate Objects

Application

In the example shown, the Design Pattern Engine will be used to enforce the one to one or one to many relationship necessary for the elimination of this security risk.

Impacts

The impact of this solution is in the enforcement of the relationship rules. This enforcement will require that the developer fully understand the various relationships that must be in place which are currently not addressed by the existing compilers. The enforcement of having exactly one trigger coil per like numbered contact will eliminate misfires and freezing of the ladder logic process.

5.1.3 Unused Objects

Table 5.6: Pattern: Unused Objects Instantiated

Classification	Software Based Errors
Sub-Class	Unused Objects
Instance	Any input or output function
Problem	When initially developing ladder logic code, the developer may choose instantiate more variable locations within the data table than is necessary. These unused components, if determined by the attacker, can be used more expeditiously than non-instantiated variables. If a variable already exists, then the effort required to attack the system is already decreased.
Solution	The Vulnerability Engine of the Static Analysis Tool, as currently developed, will use the vulnerability pattern related to placement and element component errors to determine the existence of unused objects. The related design pattern will be extended to include the required elimination of these objects by the developer prior to uploading the ladder logic to the PLC CPU.
Application	By using the vulnerability pattern related to placement and element component errors to recognize the existence of the error (Figure 5.6), and extending the current related design pattern (Figure 5.8), we eliminate this security risk.
Impacts	The impact of this solution is in the removal of the vulnerabilities ranging from severity levels A - D.

The instance classification of Table 5.6 generally refers to any input or output function, as unused objects can occur within any function. As shown, unused objects are part of the **Software Based Errors** classification.

Problem

Pre-instantiated objects within ladder logic code potentially opens multiple paths into the system itself. If an object is already pre-instantiated, and this knowledge becomes available to the attacker, then they are required to perform one less step in terms of breaching security within the system.

Slightly reworking the missing trigger coil example from above:

Assume the following:

- In a given ladder logic sequence, an output enable (O:3/4) is supposed to trigger the initialization of an emergency shutdown procedure.
- The fault routine has a normally open contact (O:3/4) associated with the output enable which would trigger the beginning of the emergency shutdown procedure (O:3/5).
- An attacker hopes to block the emergency shutdown process using a binary contact. The though process for using this contact is that it is easily disguised, as it is internal (not a hardware I/O point) and located within many other contact points.
- When instantiating the initial data point objects for the PLC ladder logic creation, the developer decided to add 32 binary points for current and future use (B3:0/0 - B3:0/31).
- The attacker floods the binary table with an array of 1's. Any of these values that remain unchanged in the table alert the attacker to a possible unused point for his attack.
- The attacker determines that B3:0/17 is an unused point and places a normally open contact for this location into the ladder logic code described.

- As this is an unused object, there is no trigger coil to alter the state of this contact, and the attacker will have successfully blocked the emergency shutdown process, prior to performing an attack on the system.

Figure 5.19 shows the vulnerability pattern and Figure 5.20 shows the ladder logic equivalent.

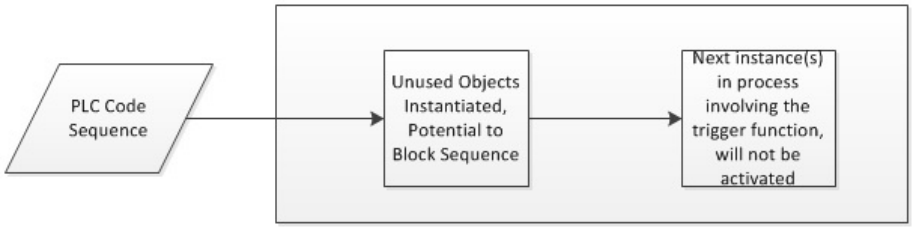


Figure 5.19: Pattern: Unused Objects Installed Vulnerability

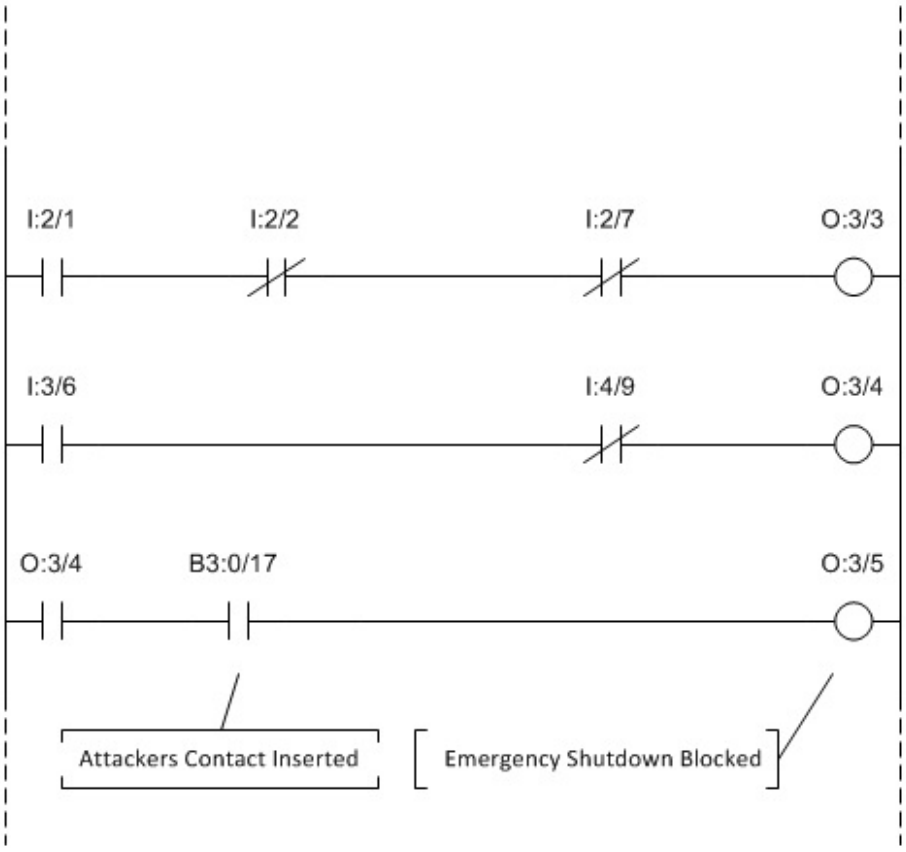


Figure 5.20: Blocking Contact Inserted

Solution

As stated, the Static Analysis Tool used the placement and element component vulnerability and the Static Analysis Tool extended the functionality of the placement and element component design pattern. This allows the system to determine the existence of unused objects without requiring that the data table elements are processed directly by the Static Analysis Tool. Figure 5.21 shows the design pattern.

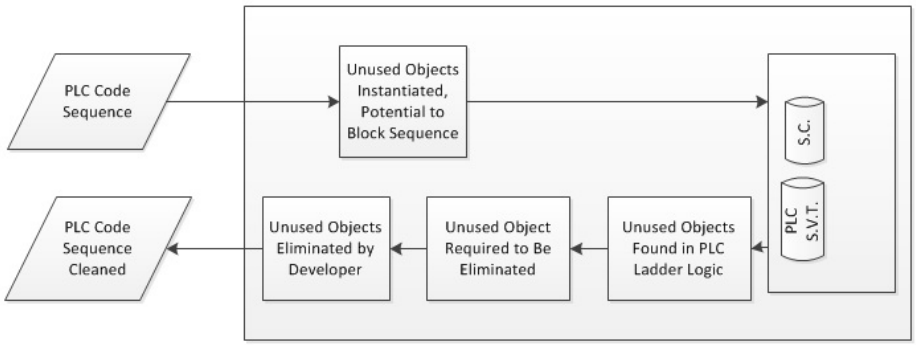


Figure 5.21: Pattern: Elimination of Unused Objects

Application

As in the example shown, the Design Pattern Engine will be used to enforce the requirement of elimination of all unused objects.

Impacts

Through the elimination of unused objects, the static analysis tool will only allow the developer to instantiate objects that are functional within the ladder logic at compile time. By eliminating unused objects, the available ladder logic elements which could be used for malicious intent would have to be instantiated by the attacker, which would possibly require an advanced knowledge of the system.

5.1.4 Hidden Jumpers

Table 5.7: Pattern: Hidden Software Jumpers

Classification	Software Based Errors
Sub-Class	Hidden Jumpers
Instance	Force routines or software jumpers installed (branch routines)
Problem	PLC ladder logic has the ability to introduce software bypass routines known as hidden jumpers, which current compilers do not reject before the program is uploaded to the CPU.
Solution	The Vulnerability Engine of the Static Analysis Tool will determine if hidden jumpers exist, either in the form of forces or blank sub-rungs.
Application	If the Static Analysis Tool detects the existence of hidden jumpers the user will be alerted and instructed to remove the vulnerability prior to the ladder logic being uploaded to the PLC CPU.
Impacts	The impact of this solution is in the removal of the vulnerabilities ranging from severity levels A - D.

Table 5.7 shows the instances of **force routines or software jumpers installed (branch routines)**. As shown, force routines or branch routines are part of the **Hidden Jumpers** sub-classification, which, subsequently, is part of the **Software Based Errors** classification.

Problem

A software force or software jumper, in the form of a blank sub-rung, introduces the security risk of allowing any part, or parts, of the PLC ladder logic to be bypassed or overridden. These hidden jumpers are intended to be force routines for testing purposes and branch routines for multiple OR functionality within the ladder logic itself, when not left as blank sub-rungs.

For example:

Assume that some input (I:2/1) causes the activation of a process output (O:3/3). Further assume that their exists a function (LEQ A, B) that implies that the data associated with the variable A must be less than the data associated with the variable B. Building on our design pattern involving hard coded values (Figure 5.4), assume that the attacker has tried, and failed, to override the current values in the LEQ command. The attacker may choose to insert a software jumper to bypass the LEQ command and therefore activate the next process prematurely. Figure 5.22 shows the vulnerability pattern and Figure 5.23 shows the ladder logic equivalent.

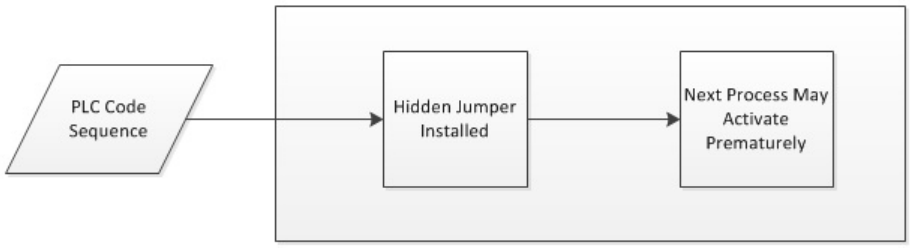


Figure 5.22: Pattern: Hidden Jumper Installed Vulnerability

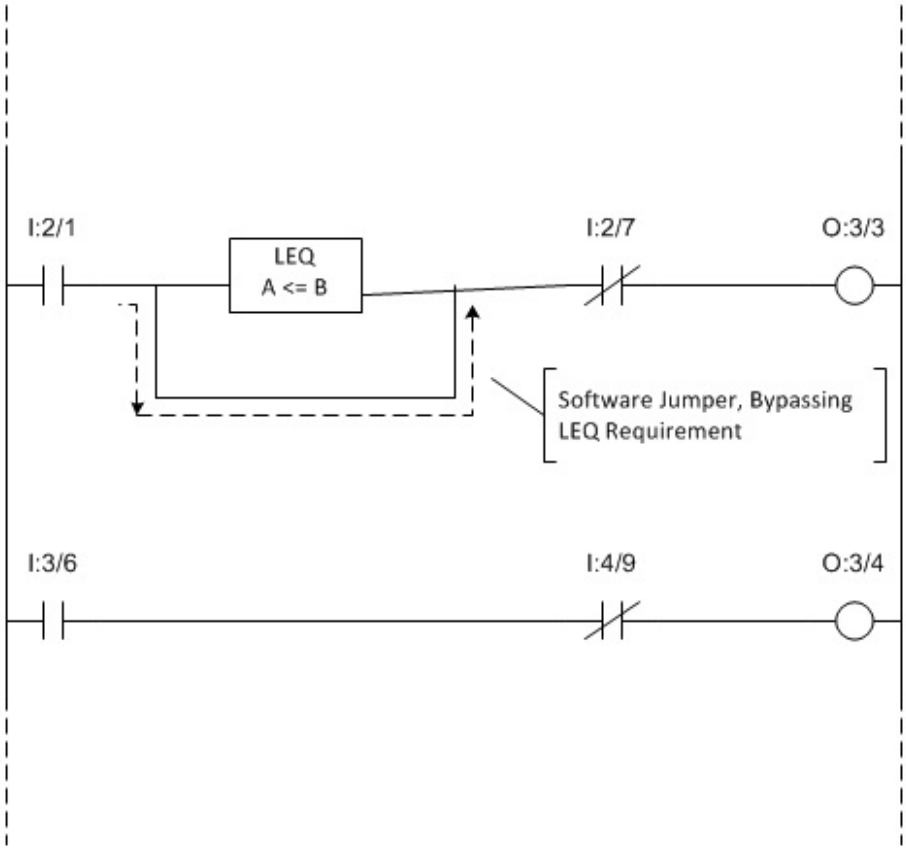


Figure 5.23: Software Jumper Installed

Solution

The Static Analysis Tool, upon detection of the vulnerability, would alert the developer of the existence of the hidden jumpers, require the enforcement of the elimination of all types of hidden jumper and negate the vulnerability. Figure 5.24 shows the vulnerability pattern and Figure 5.25 shows the ladder logic equivalent.

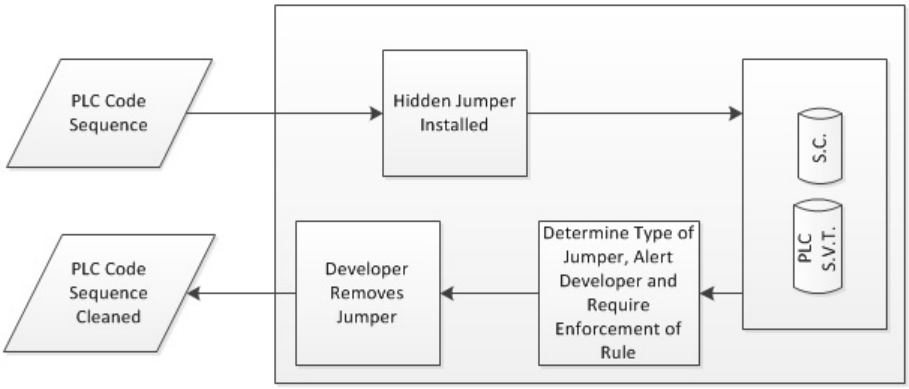


Figure 5.24: Pattern: Elimination of Hidden Jumpers

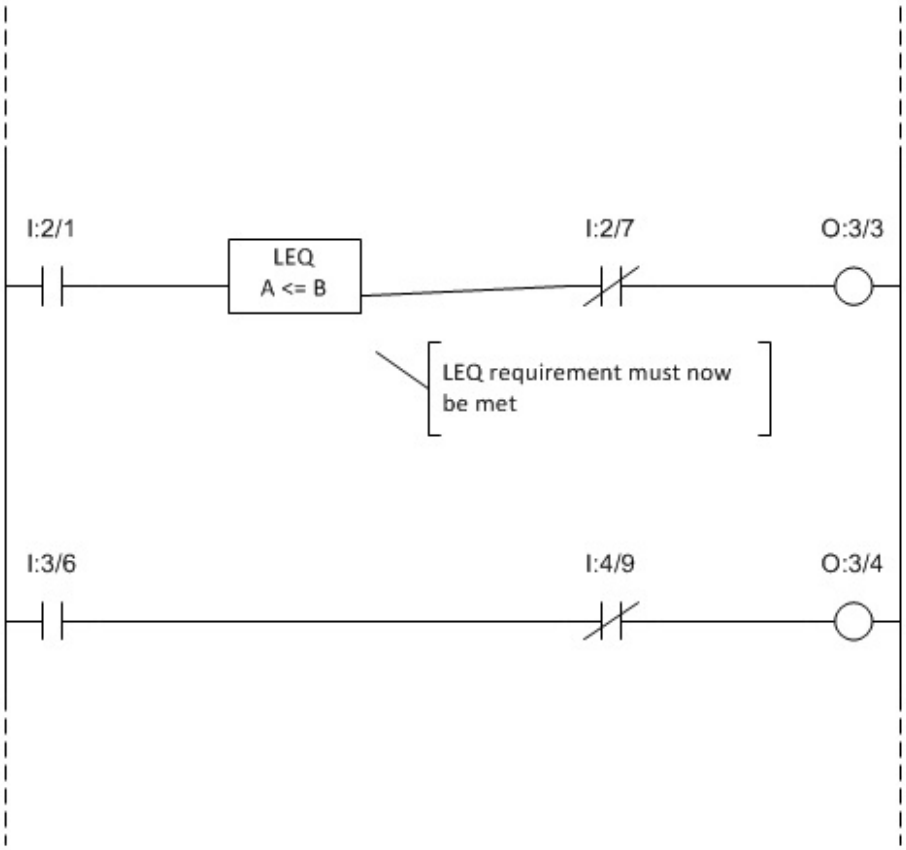


Figure 5.25: Elimination of Software Jumper

Application

The Vulnerability Engine of the Static Analysis Tool detects the existence of any type of hidden jumper. The static analysis tool would need to go a step further to properly negate this vulnerability, by being active on the PLC CPU and continuously reviewing live ladder logic. This would allow for the Static Analysis Tool to be configured such that it could differentiate between forces in test mode and forces installed in live code.

Impacts

As described, hidden jumpers can be software forces or blank sub-rungs located within the programming structure of the ladder logic code. Forces, as they are called, are sometimes used as a testing and troubleshooting tool before and during the commissioning of the system utilizing the PLC and the ladder logic. Forces are not intended to remain active in 'live' code. The design pattern would require the enforcement of the rule banning forces and blank sub-rungs from live code.

5.2 Selection of Design Patterns to Mitigate Software Vulnerabilities

We associate each vulnerability with one or more design patterns. In this section we address the need to select the appropriate design patterns to mitigate the detected problem. We denote the set of design patterns associated with the vulnerability in V as $\mathcal{P}$, such that the application of a pattern in $\mathcal{P}$ on the ladder logic code will remove V.

Given ladder logic code L and the set of detected vulnerabilities, we propose a minimal and complete set of design patterns. We say that $\mathcal{P}$ is complete with respect to a set of vulnerabilities V if for every vulnerability $v_i \in V$ there is a pattern $p_i \in$

$\mathcal{P}$ such that p_i mitigates v_i. We say that $\mathcal{P}$ is minimal with respect to a set of vulnerabilities V if there is no pattern $p_i \in \mathcal{P}$ such that $\mathcal{P} - \{p_i\}$ would still mitigate all vulnerabilities in V.

The security Static Analysis Tool may reach one of two outcomes: No security vulnerability or Compiled with a Possible Vulnerability Existence.

- No Security Vulnerability. The proposed code has been completely tested using the PLC-SF set of tools and has been determined to be error free.
- Compiled with a Possible Vulnerability Existence. After using the PLC-SF set of tools, the compiler would warn of a possible coding issue that may result in a security concern. The severity level of the concern would be noted as well.

Chapter 6

Static Analysis Tool

6.1 Overview of the Static Analysis Tool

We have implemented a security static analysis tool that detects PLC code vulnerabilities and recommends mitigation strategies. It uses the vulnerability taxonomy (Figures 4.2 - 4.10) the severity chart ratings (Table 4.1), and state-transition models to detect PLC vulnerabilities in the PLC code. Each Vulnerability is cross referenced with the design pattern that mitigates the vulnerability. The static analysis tool, in relation to the PLC-SF framework, uses the following as input:

- Ladder logic from an existing PLC program
- The PLC code Vulnerability Taxonomy
- State Transition diagrams for the vulnerabilities
- A list of Design Patterns
- The Severity Chart

Figure 6.1 shows the current work flow of a PLC compiler. As noted, the dashed area represents the insertion of our Static Analysis Tool into this process.

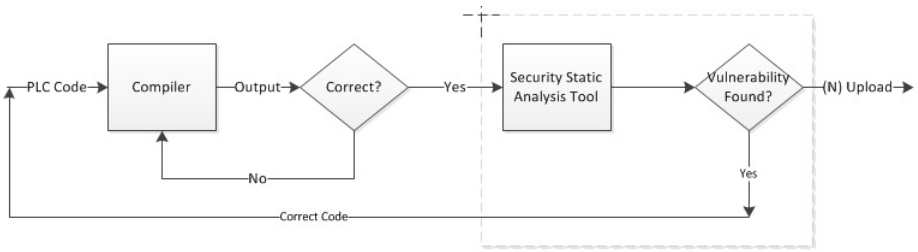


Figure 6.1: PLC Compiler Work Flow with Static Analysis Tool

6.1.1 PLC Code Vulnerability Taxonomy

The Vulnerability Taxonomy is the basis behind which the Taxonomy Engine is created. The Taxonomy Engine uses state transition diagrams as a bridge between the Vulnerability Taxonomy and the Taxonomy Engine itself. With each vulnerability documented in the Vulnerability Taxonomy, a state transition diagram is created to represent that vulnerability. The Taxonomy Engine evaluates the input PLC code by building the stages in the state transition diagrams. If an error is detected the Taxonomy Engine passes the vulnerability information to the design pattern engine and the severity engine.

6.1.2 Matching Design Patterns

In Section 4.5 we mapped the corrective actions with state transition diagrams. These corrective actions are converted to design patterns. The design patterns are intended to provide the user with a proven solution to mitigate the vulnerability. The design pattern engine uses the vulnerability information from the Taxonomy Engine to match corrective actions (design patterns) for each detected vulnerability.

6.1.3 The Severity Chart

The Severity Chart, which is shown in Chapter 4 serves as the initial ranking mechanism of each of the vulnerabilities. Vulnerabilities are evaluated based on their potential impact considering novice or malicious users. The aim of our ranking is that developers can address the most critical problems.

6.1.4 Static Analysis Tool Output

The Static Analysis Tool produces the following output:

- List of Vulnerabilities and their Ranking
- Associated Design Patterns
- Suggested Solution

This level of output, once presented to the user, should provide a basis to make a sound decision to substantiate, as well as mitigate, the security risk determined by the vulnerability engine.

6.2 Static Analysis Tool Implementation Examples

We have implemented a proof of concept static analysis tool. The tool is running on the Windows 7 platform and was coded in C#. In the following we provide screen-shots of the tool evaluating PLC code containing the vulnerabilities we have studied.

6.2.1 Determination of Race Condition

Explanation of Analysis:

Figure 5.10 depicted the vulnerability created if the trigger bit for a timer element is located such that it becomes a condition for its own state transition. This would cause a continual oscillation between the active and inactive states of the element in question, thus causing a race condition scenario. Current compilers would allow the ladder logic to be compiled and the race condition would allow for the continual oscillation of the on/off state, as described above, to occur. Initially, the Vulnerability Engine would determine the existence of this vulnerability. Once the vulnerability is determined, the code is then compared against the severity chart, which ranks the current potential error as to the level of security risk that would be created if no action is taken. Once the vulnerability is determined and the severity assigned the Design Pattern Engine is used to determine the best course of action to mitigate the vulnerability. Figure 5.11 shows the proposed design pattern that would be suggested by the Static Analysis Tool. Using figure 6.2, we will now look at the race condition scenario in more detail.

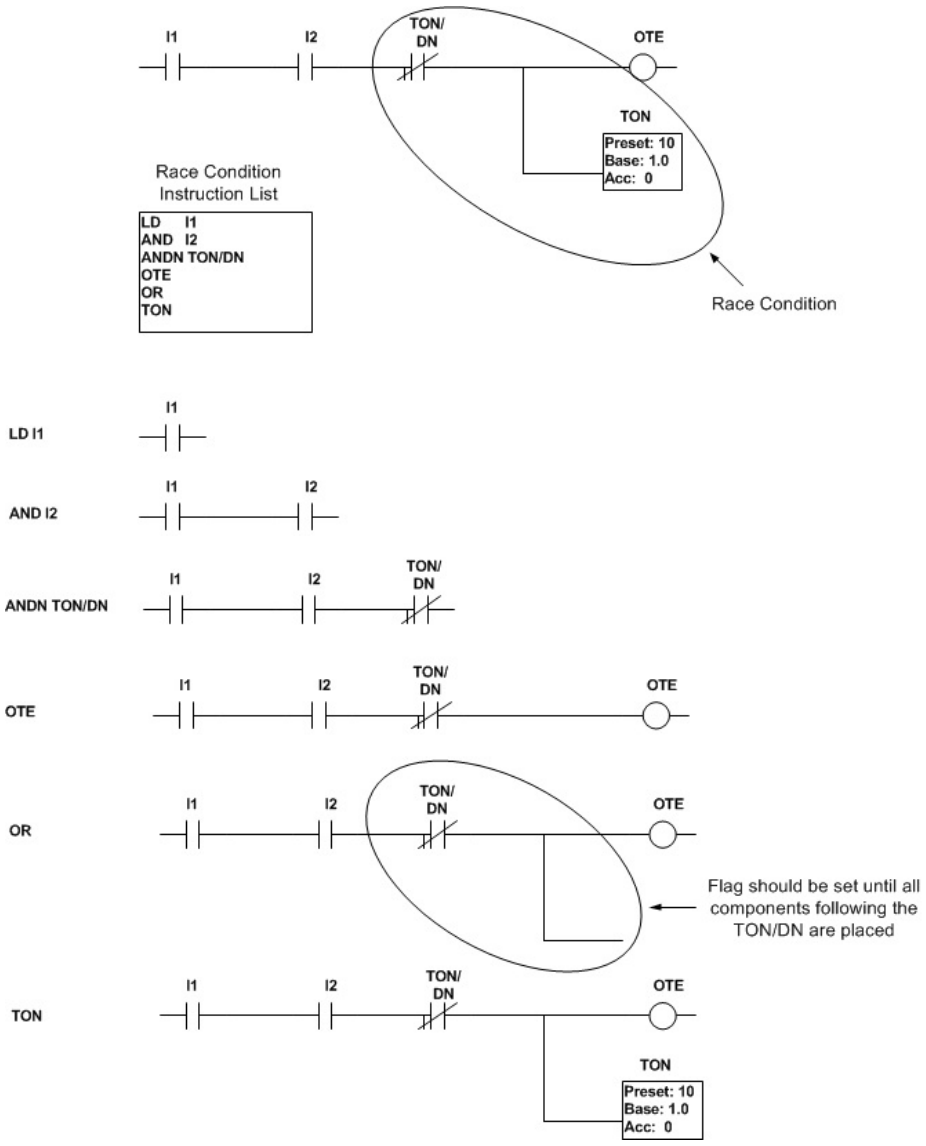


Figure 6.2: Timer Race Condition (Ladder Logic Example)

As each element is added to the specific rung of code, it is compared to the software vulnerability taxonomy to determine if a flag is necessary as processing continues. Note an example of this in line of 5 of this figure. The TON/DN bit is flagged and placed on a 'watch' condition as the remainder of the code is processed. This flag is based on the insertion of the TON/DN bit and its intended function as it relates to the remainder of the rung. At this point, the Static Analysis Tool is uncertain as to the remainder of the rung, so the flag becomes necessary as a cross check mechanism.

Once the OR statement is placed, the Static Analysis Tool checks the current flagged elements against both sides of the OR statement, as well as setting any new flags which may be necessary. As denoted in line 5 (circled region) of this figure, clearly there is a potential race condition between the TON/DN bit used to activate the timer, and the TON (timer coil element) which is used to activate the TON/DN bit.

The software vulnerability taxonomy would denote this error as a **Beginning of Rung Function** subclass which is a subset of the **Placement and Element Component Errors** subclass, which in turn is a subset of the **Logic Errors** classification. Once the error has been determined with the Vulnerability Taxonomy and a severity level assigned by the Severity Engine, the correct design pattern would be determined and applied by the Design Pattern Engine. As stated, the correct design pattern for this scenario Figure 5.11. Using the suggestion determined by the design pattern, the timer bit would be recommended to be placed inside of the latch. Each of the processes described would continue until a probable solution for each vulnerability is determined. This solution would be noted by the compiler and a flag would be set to alert the developer so that corrective action is taken. Once the error is determined and the probable correction noted, the error and this be cataloged. Each cataloged entry would contain information such as the type of error, time and date of error, number of times this type of error has been generated and the severity level of the error itself. This information could then be used not only to solve the immediate occurrence, but also provide a means to begin the development of a best practices guide. The best practices guide could then be continually updated and used as a training tool for new PLC programmers, as well as a design tool for experienced programmers, which would allow them to begin coding toward known, and perceived, security risks.

Screen shots for race condition error:

Figure 6.3 shows the Static Analysis Tool upon initialization. The user will be prompted to browse for the PLC ladder logic code file that they need to have verified and validated.

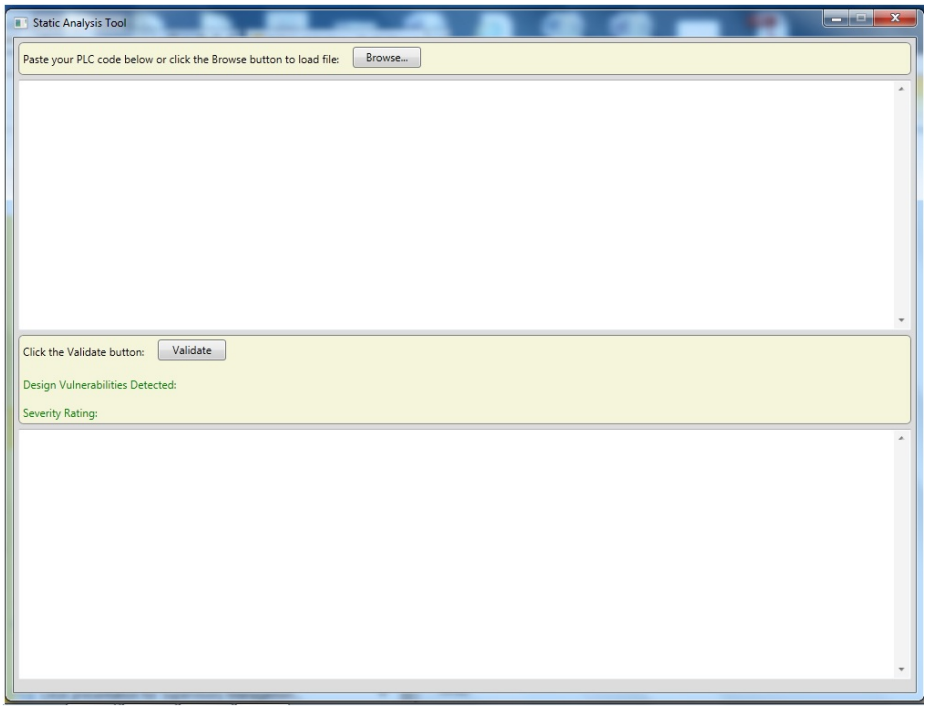


Figure 6.3: Static Analysis Tool: Initialization

Once the file is loaded, as shown in Figure 6.4, the user will select the validate button, and validation process will begin.

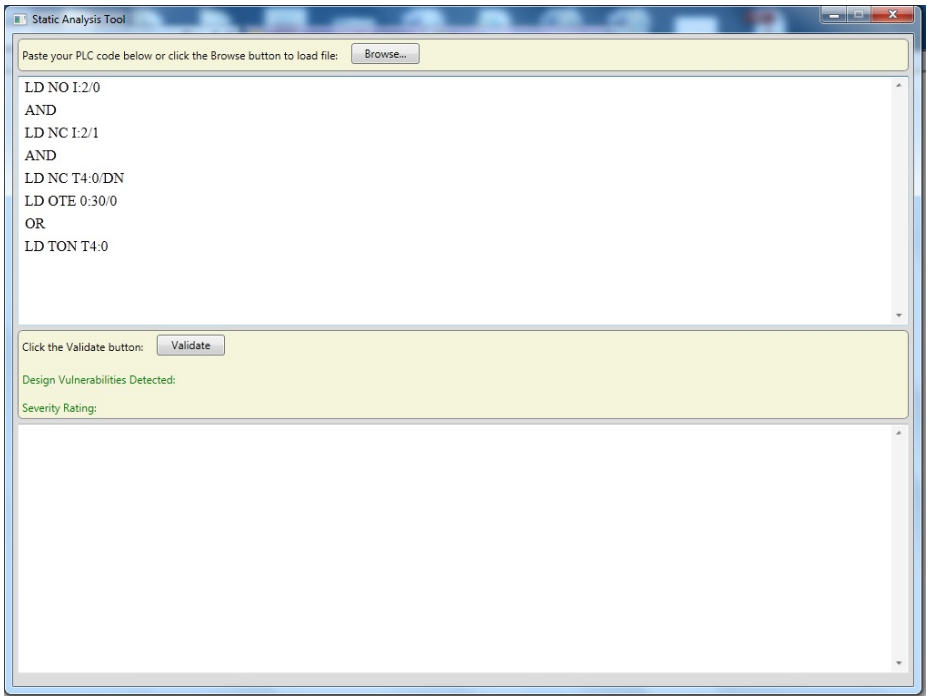


Figure 6.4: Static Analysis Tool: Race Condition Error

After the validation process is completed, the user will be given the following information as shown in Figure 6.5: the design vulnerabilities found, severity rating given, vulnerability pattern, associated design pattern and suggested PLC ladder logic modification.

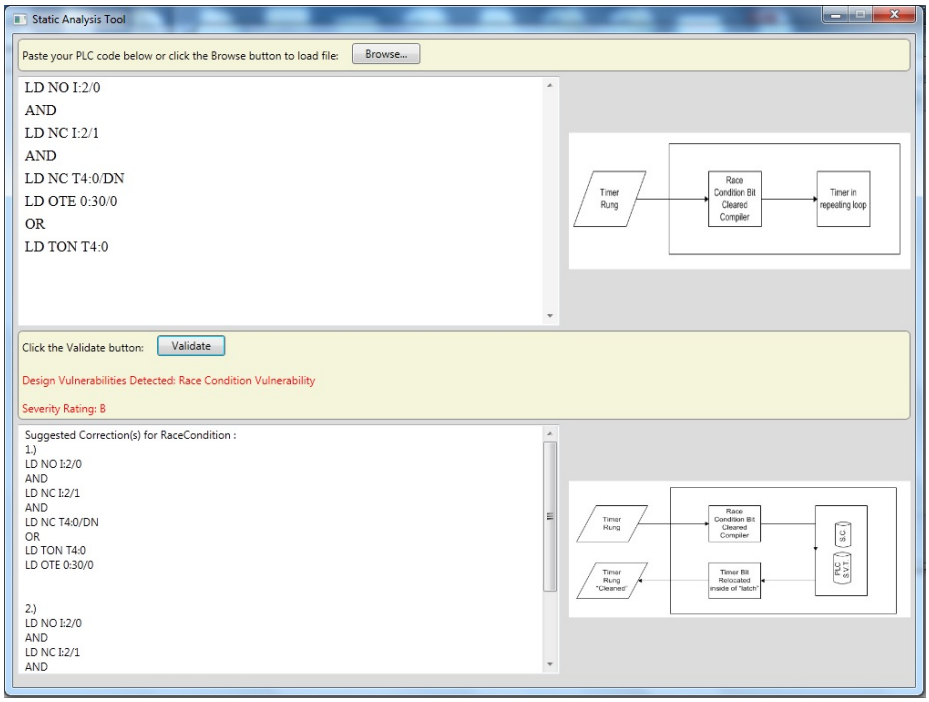


Figure 6.5: Static Analysis Tool: Mitigation of the Race Condition Error

6.2.2 Determination of Missing Trigger Coil

Explanation of Analysis:

Figure 5.6 depicted the vulnerability created if the required trigger bit to continue the execution of a process is non-existent. If the trigger bit (coil) is missing from the ladder logic then the individual contact that requires this bit for change of state purposes would become non-functional. Current compilers would allow the ladder logic to be compiled as written. As previously stated for the race condition example, the Vulnerability Engine would determine the existence of this vulnerability. Once the vulnerability is determined, the code is then compared against the severity chart, which ranks the current potential error as to the level of security risk that would be created if no action is taken. Once the vulnerability is determined and the severity assigned the Design Pattern Engine is used to determine the best course of action to mitigate the vulnerability. Figure 5.8 shows the proposed design pattern that would

be suggested by the Static Analysis Tool. Using figure 6.6, we will now look at the missing trigger bit scenario in more detail. This figure depicts three rungs of ladder logic code. Each rung is comprised of normally open and normally closed contacts as well as output coils. The normally open contacts such as I:x/y need no trigger coils, as these are directly triggered by hardwired input devices as the 'I' indicates. One the third rung, note that contact O:4/7 is circled. This contact, as it is triggered by an output coil, as denoted by the 'O' has the potential to be missing its trigger coil. Therefore, a flag is set at this point until which time that the matching trigger is located. The matching trigger coil is not found after the remainder of the ladder logic has been scanned, therefore a vulnerability has been determined. The software vulnerability taxonomy would denote this error as a **End of Rung Function** subclass which is a subset of the **Placement and Element Component Errors** subclass, which in turn is a subset of the **Logic Errors** classification. Once the error has been determined with the Vulnerability Taxonomy and a severity level assigned by the Severity Engine, the correct design pattern would be determined and applied by the Design Pattern Engine. As stated, the correct design pattern for this scenario Figure 5.8. Using the suggestion determined by the design pattern, the missing trigger bit would be noted and the developer notified to insert the coil where required.

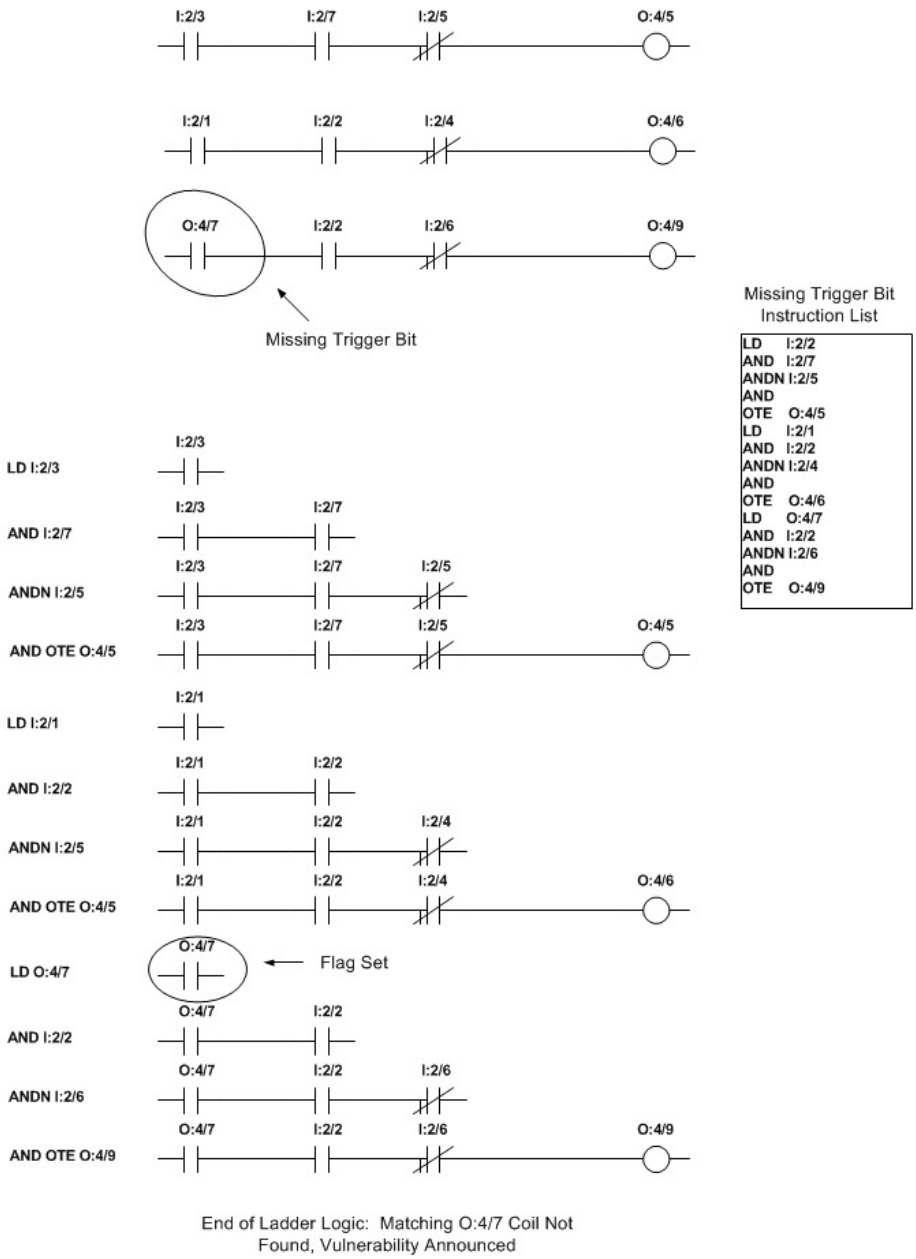


Figure 6.6: Missing Trigger Bit (Ladder Logic Example)

Screen shots for missing trigger bit error:

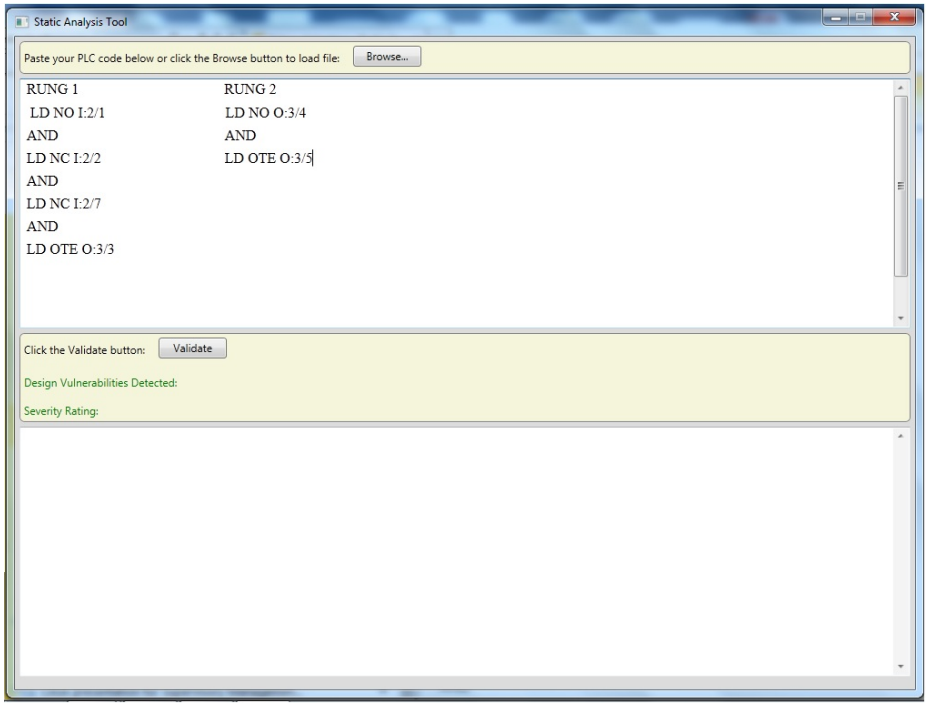


Figure 6.7: Static Analysis Tool: Missing Trigger Bit Error

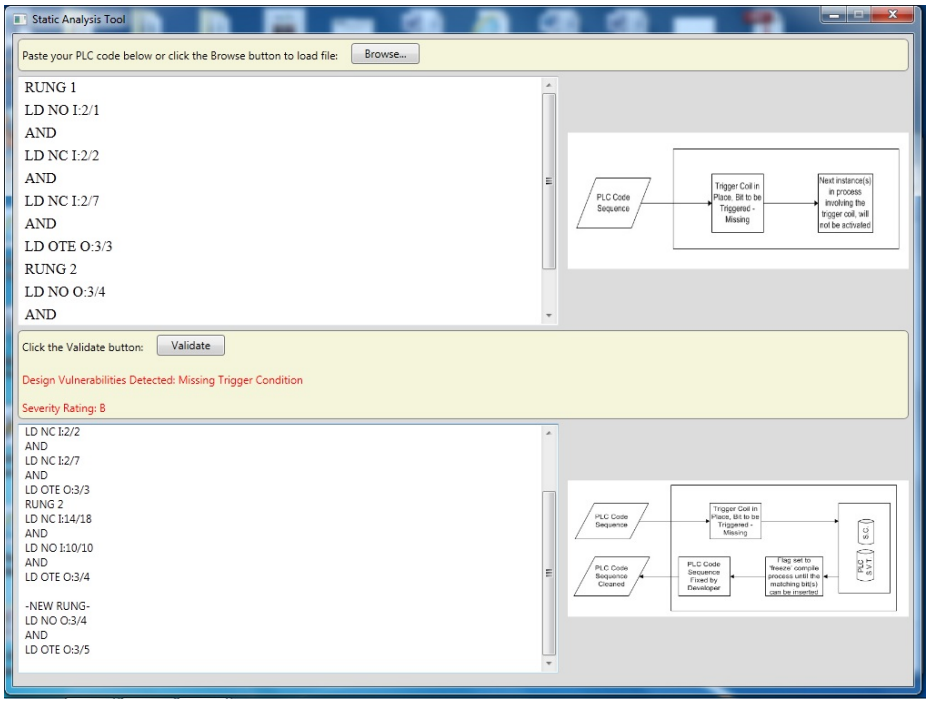


Figure 6.8: Static Analysis Tool: Mitigation of the Missing Trigger Bit Error

6.2.3 Determination of Hidden Jumpers

Explanation of Analysis:

Figure 5.22 depicted the vulnerability created if the required trigger bit to continue the execution of a process is non-existent. If a software jumper were intentionally or unintentionally installed, this would introduce the existence of a bypass condition on a single, or multiple elements, within a given rung of ladder logic code. Current compilers would allow the ladder logic to be compiled as written. As previously stated for the race condition example, the Vulnerability Engine would determine the existence of this vulnerability. Once the vulnerability is determined, the code is then compared against the severity chart, which ranks the current potential error as to the level of security risk that would be created if no action is taken. Once the vulnerability is determined and the severity assigned the Design Pattern Engine is used to determine the best course of action to mitigate the vulnerability. Figure 5.24

shows the proposed design pattern that would be suggested by the Static Analysis Tool. Using figure 6.9, we will now look at the hidden jumper inserted scenario in more detail. This figure depicts two rungs of ladder logic code. Each rung is comprised of normally open and normally closed contacts as well as output coils. This example also shows the presence of a 'OR' circuit in the form of a branch rung. The ladder logic code is accepted token by token and once the presence of the 'Branch' statement is found, a flag is set until the existence or non-existence of a contact within the branch circuit. In the following line, an 'End Branch' command is encountered before a contact was detected. Therefore the software vulnerability taxonomy would denote this error as a **Rung Jumper** subclass which is a subset of the **Hidden Jumpers** classification. Once the error has been determined with the Vulnerability Taxonomy and a severity level assigned by the Severity Engine, the correct design pattern would be determined and applied by the Design Pattern Engine. As stated, the correct design pattern for this scenario Figure 5.24. Using the suggestion determined by the design pattern, the rung jumper (branch) would be removed.

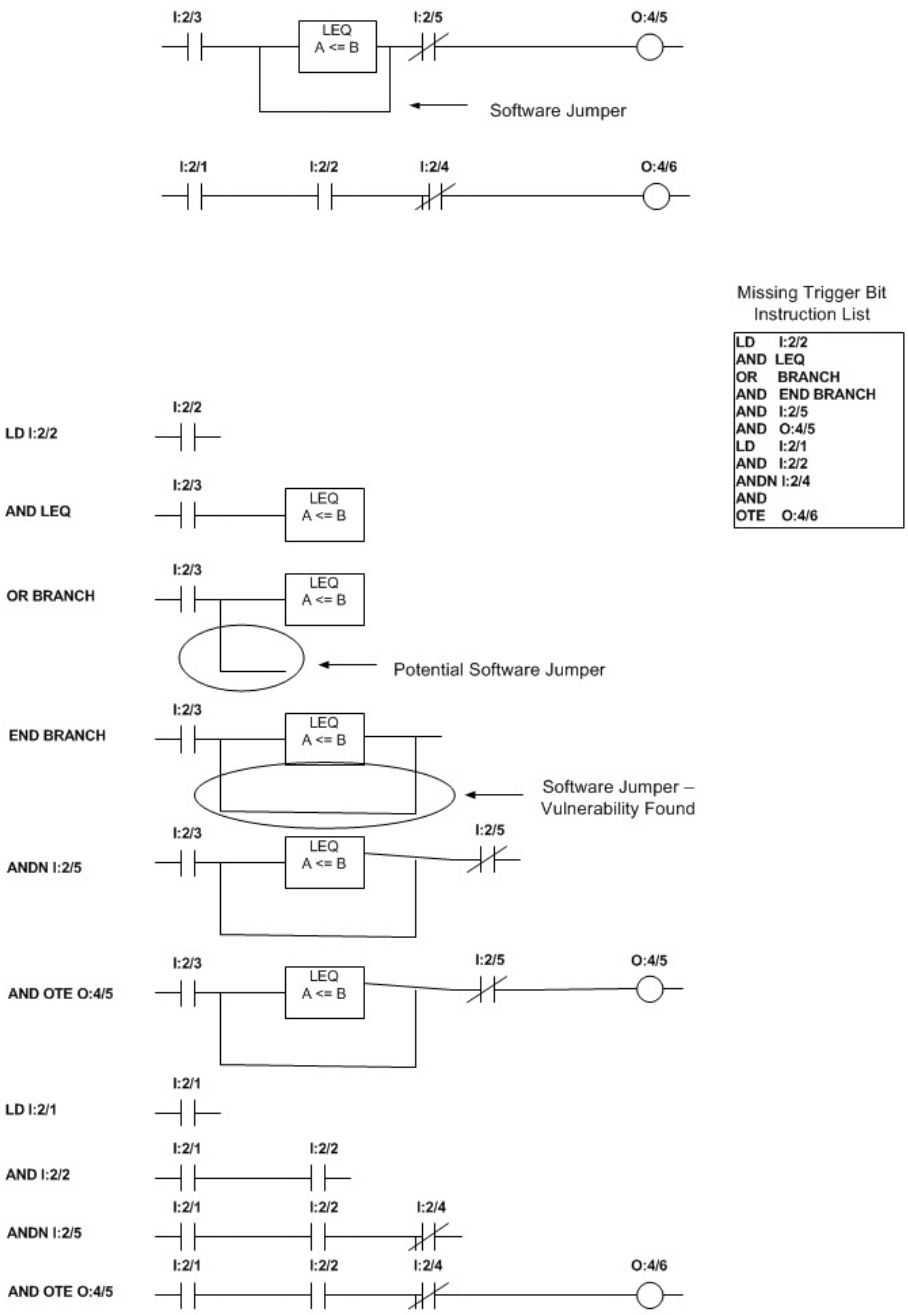


Figure 6.9: Hidden Jumper (Ladder Logic Example)

Screen shots for hidden jumper error:

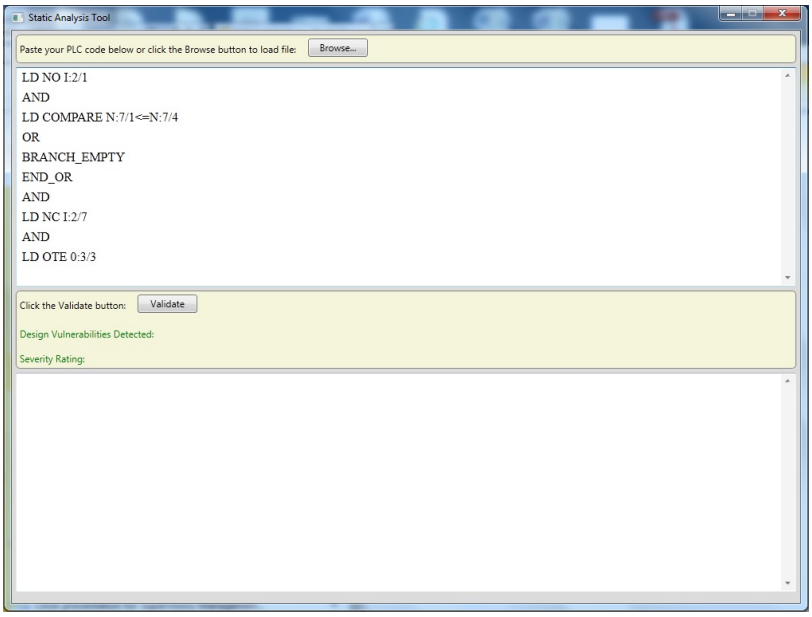


Figure 6.10: Static Analysis Tool: Hidden Jumper Error

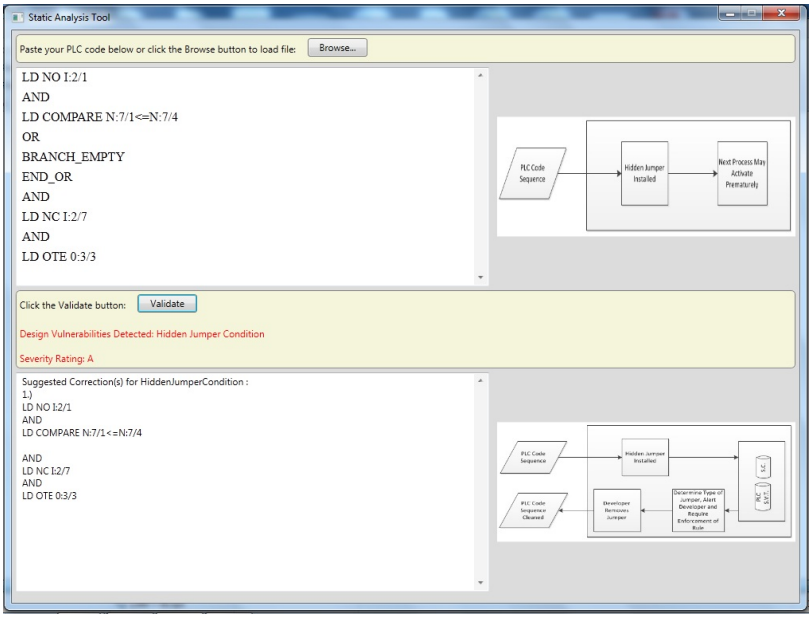


Figure 6.11: Static Analysis Tool: Mitigation of the Hidden Jumper Error

Chapter 7

Conclusions and Future Research

The area of Supervisory Control and Data Acquisition System (SCADA) protection is currently being studied by multiple parties. However, the security vulnerabilities associated with Programmable Logic Controllers (PLC) are a relatively new area of SCADA security research. The SCADA system depends on the data and control supplied by the PLC's. Our research has allowed us to look at the data acquisition system from not only the initial gathering point of the data, but also through the device that ultimately controls the automation processes in their entirety. The examples that we have shown throughout this work instantiate the need to secure not only the SCADA PC's, but also the PLC's to which these PC's are connected.

This research proposes the development of a systematic classification of PLC software vulnerabilities, mitigation strategies, and, ultimately, a static analysis tool which identifies potential threats in ladder logic code and recommend corrective actions to mitigate the threats. The intent of this research is to provide a mechanism which, through its various components, would allow for the protection of the PLC ladder logic code, thereby enforcing the security requirements of the SCADA system in its entirety. The Static Analysis Tool is built around the concepts of: 1) The Vulnerability Taxonomy, 2) Severity Chart and 3) Design Patterns.

Using the Vulnerability Taxonomy as a guide, each vulnerability class was mapped using state transition diagrams. These state transition diagrams allowed for a direct correlation between the information contained within the Vulnerability Taxonomy and the PLC ladder logic code. The Vulnerability Taxonomy is the basis around

which the Taxonomy Engine was created. The Taxonomy Engine takes the PLC ladder logic code which is input into the Static Analysis Tool and determines the existence of vulnerability based on the set of rules provided.

Once it has been determined that a vulnerability exists, the Severity Engine determines the severity of the vulnerability found. For this, we developed a Severity Chart. The severity of each vulnerability is ranked according to the extent of damage possible if a given security breach would occur. The ranking assigned by the Severity Chart are 'A - D', with 'A' being the most severe and 'D' being less severe. The Severity Engine is built around the severity levels found withing the severity chart.

Next, we address the need for mitigating detected vulnerabilities. The Design Pattern Engine selects Design Patterns, i.e. proper coding methods, to mitigate the vulnerabilities. The output provided by the Static Analysis Tool provides a mechanism for the user to determine the existence of a security risk within their PLC code, an understanding of the security severity level(s) involved, and a list of design patterns that will help to alleviate those risks.

Our Vulnerability Taxonomy incorporates both safety and security vulnerabilities. Therefore it supports both novice and experienced coders as well as defends against malicious attackers.

Our future research addresses the following problems:

1. Automate the correction process within the Static Analysis Tool

   Currently, the output of the Static Analysis Tool lists the vulnerability found, severity of the vulnerability, the associated design pattern which could be used to mitigate the vulnerability and sample PLC code showing the proposed correction. We envision automating this entire process such that the end user could agree or disagree with the proposed change. If the user agrees with the proposed change, the change would automatically modify the PLC ladder logic in question. Figure 7.1 shows a 'mock up' of the proposed automated process.

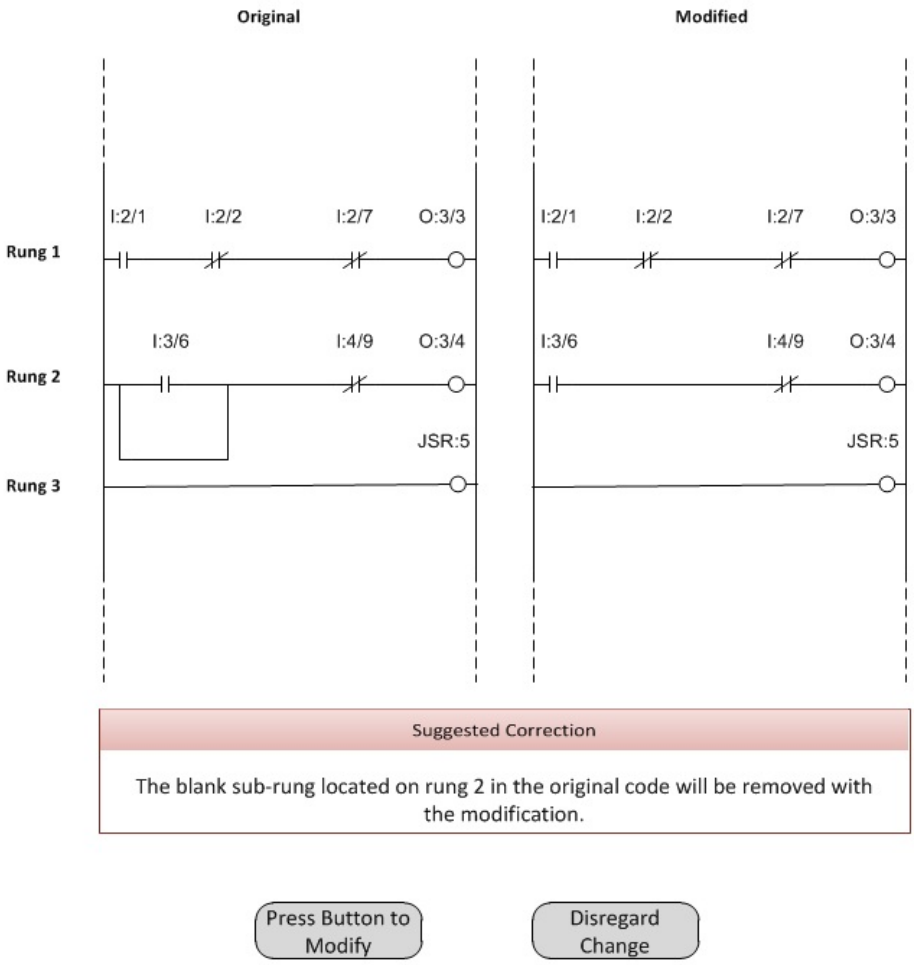


Figure 7.1: Future Automated Static Analysis Tool

2. Optimized design pattern selection

   Currently, the Static Analysis Tool will select each design pattern that provides a mitigation strategy to the vulnerability found. Our future research will attempt to find the most efficient design pattern of the suggestions given.

   Multiple design patterns may exist to mitigate a vulnerability. Enhanced pattern selection would allow the system to select design patterns for a set of vulnerabilities by optimizing a cost function, e.g. number of patterns or cost of redesign of the PLC code.

3. Experimentation

   Through experimentation to be done with groups of students currently enrolled in the Electrical Engineering Technology program at York Technical College, we are planning to investigate the ease of use of our ladder logic Static Analysis Tool. We plan to follow the approach presented in the technical paper from the NIST [37]. This will allow us to determine any necessary corrections or additions necessary prior to working with the individual product vendors to gain access to their proprietary systems.

Bibliography

[1] *A taxonomy of computer program security flaws with examples*, Prentice Hall, 1993.

[2] *Homeland security act of 2002, h.r. 5005*, 2002.

[3] Zulfakar Aspar and Mohamed Khalil-Hani, *Modeling of a ladder logic processor for high performance programmable logic controller*, Tech. report, Universiti Teknologi Malaysia, 2009.

[4] American Gas Association, *Cryptographic protection of scada communications, part 1: Background policies and test plan*, Tech. report, American Gas Association, 2005.

[5] Rockwell Automation, *Error getting factory talk directory schema*, Tech. report, Rockwell Automation, 2003.

[6] Rockwell Automation, *Studio seems unresponsive and generates: The requested access check failed because the token has expired*, Tech. report, Rockwell Automation, 2005.

[7] Eva Blomqvist, Aldo Gangemi, Elena Montiel, Nadejda Kikitina, Valentina Presutti, and Boris Villazon-Terrazas, *D2.5.2 pattern based ontology design: Methodology and software support*, Tech. report, CNR, 2010.

[8] Phil Cambell, Jennifer DePoy, John Dillinger, Jason Stamp, and William Young, *Sustainable security for infrastructure scada*, Tech. report, Sandia National Labs, 2006.

[9] Alvaro A. Cardenas, Saurabh Amin, Zong-Syun Lin, Yu-Lun Huang, Chi-Yen Huang, and Shankar Sastry, *Attacks against process control systems: Risk assessment, detection and response*, Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11), pp. 355–366.

[10] Brian Chess, Gary McGraw, and Katrina Tsipenyuk, *Seven pernicious kingdoms: A taxonomy of software security errors*, Tech. report, Fortify Software, 2005.

[11] North American Electric Reliability Council, *Cyber security standards*, Tech. report, North American Electric Reliability Council, 2005.

[12] Dipankar Mohd Dasgupta, Hassan Ali, Robert Abercrombie, Bob Schlicher, Fredrick Sheldon, and Marco Carvalho, *Secure vm for monitoring industrial process controllers*, Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW '11).

[13] Thomas Erl, *Soa design patterns*, Prentice Hall, 2009.

[14] Eduardo B. Fernandez, M. M. Larrondo-Petrie, Yifeng Shao, and Jie Wu, *On building secure scada systems using security patterns*, Tech. report, Florida Atlantic University, 2009.

[15] Eduardo B. Fernandez, Jie Wu, M.M. Larrondo-Petrie, and Yifeng Shao, *On building secure scada systems using security patterns*, Proceedings of 5th Annual Workshop on Cyber Security and Information Intelligent Research: Cyber Security and Information Intelligence Challenges and Strategies (CSIIRW '09), 2009.

[16] John D. Fernandez and Andres E. Fernandez, *Scada systems: Vulnerabilities and remediation*, Journal of Computing in Small Colleges (2005), 160–168.

[17] National Center for Advanced Secure Systems Research, *Ncassr project: Scada protocol authentication project*, Tech. report, National Center for Advanced Secure Systems Research, 2007.

[18] National Institute for Standards and Technology, *It security for industrial control systems*, Tech. report, National Institute for Standards and Technology, 2002.

[19] Guillermo Francia III, David Thornton, and Thomas Brookshire, *Wireless vulnerability of scada systems*, Proceedings of the 50th Annual Southeast Regional Conference (ACM-SE '12), pp. 331–332.

[20] K Goseva-Popstojanova, Feiyi Wang, Rong Wang, Fengmin Gong, K. Vaidyanathan, K. Trivedi, and B. Muthusamy, *Characterizing intrusion tolerant systems using a state transistion model*, DARPA Information Survivability Conference, vol. 2, pp. 211–221.

[21] Adam Hahn, Ben Kregel, Manimaran Govindarasu, Justin Fitzpatrick, Rafi Adnan, Siddharth Sridhar, and Michael Higdon, *Development of the powercyber scada security testbed*, Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW '10), 2010.

[22] Ronald Heller and Freek Van Teeseling, *Business patterns in ontology design*, Tech. report, Be Informed, 2010.

[23] Mariana Hentea, *Improving security for scada control systems*, Interdisciplinary Journal of Information, Knowledge and Management (2008).

[24] The White House, *Decision directive/nsc-63*, Tech. report, The White House, 1998.

[25] Michael Howard, David LeBlanc, and John Viega, *24 deadly sins of software security: Programming flaws and how to fix them*, McGraw-Hill, 2009.

[26] D.J. Kang, J.J. Lee, B.H. Kim, and D. Hur, *Proposal strategies of key management for data encryption in scada network of electric power systems*, International Journal of Electric Power and Energy Systems **33** (2011), 1521–1526.

[27] Paul A. Karger, Amit Paradkar, and Sam Weber, *A software flaw taxonomy: Aiming tools at security*, Tech. report, IBM Research Division, 2005.

[28] Jia-Ling Lin, *Real-time misuse detection using abstract signatures and adaptable strategies*, 1997.

[29] Michael Loughnane, *Epa needs to determine what barriers prevent water systems from securing known supervisory control and data acquisition (scada) vulnerabilities*, Tech. report, Environmental Protection Agency, 2005.

[30] Stephen McLaughlin and Patrick McDaniel, *Sabot: Specification based payload generation for programmable logic controllers*, Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12), pp. 439–449.

[31] Lucille McMinn, Jonathan Butts, David Robinson, and Billy Rios, *Exploiting the critical infrastructure via nontraditional system inputs*, Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW '11).

[32] Bill Miller and Dale Rowe, *A survey of scada and critical infrastructure incidents*, Proceedings of the 1st Annual Conference on Research in Information Technology (RIIT '12), pp. 51–56.

[33] John C. Munson, Jack L. Meador, and Rick P. Hoover, *Software process control for secure program execution*, Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW '10).

[34] A. Nicholson, S. Webber, S. Dyer, T. Patel, and H. Janicke, *Scada security in the light of cyber-warfare*, Security **31** (2012), 418–436.

[35] North American Reliability Corporation, *Critical infrastructure protection.*

[36] US Department of Energy, *21 steps to improve cyber scada security*, Tech. report, US Department of Energy, 2005.

[37] Vadim Okun, William F. Guthrie, Romain Gaucher, and Paul E. Black, *Effect of static analysis tools of software security: Preliminary investigation*, Tech. report, National Institute of Standards and Technology, 2007.

[38] Summer Olmstead, Joseph Stites, and Ferrol Aderholdt, *A layer cyber security defense strategy for smart grid programmable logic controllers*, Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW '11).

[39] Sandip C. Patel, Ganesh D. Bahatt, and James H. Graham, *Improving the cyber security of scada communication networks*, ACM (2009), 139–142.

[40] May Robin Permann and Kenneth Rohde, *Cyber assessment methods for scada security*, Tech. report, Idaho National Laboratory, 2005.

[41] Venkat Pothamsetty and M. Franz, *Honeynet project: Building honeypots for industrial networks*, Tech. report, Independent, 2004.

[42] Kevin Poulsen, *Slammer worm crashed ohio nuke plant network*, SecurityFocus (2003).

[43] Abdalhossein Rezai, Parviz Keshavarzi, and Zahra Moravej, *Secure scada communication by using a modified key management scheme*, ISA Transactions (2013).

[44] Dibya Sarkar, *Dhs funds control systems research*, Federal Computer Week (2004).

[45] Robert Schaefer, *Relay ladder logic considered harmful*, SIGSOFT Software Engineering Notes (2013), 8–9.

[46] K. Stouffer, J. Falco, and K. Kent, *Guide to supervisory control and data acquisition (scada) and industrial control systems security*, Tech. report, NIST, 2006.

[47] Industrial Control Systems Cyber Emergency Response Team, *Icsa-10-301-01 control system internet accessibility*, Tech. report, US Department of Homeland Security, 2010.

[48] Industrial Control Systems Cyber Emergency Response Team, *Icsa-11-343-01 control system internet accessibility*, Tech. report, US Department of Homeland Security, 2011.

[49] Industrial Control Systems Cyber Emergency Response Team, *Icsa-12-305-01 siemens sipass server buffer overflow*, Tech. report, US Department of Homeland Security, 2012.

[50] Sidney Valentine and Csilla Farkas, *Impact of plc code vulnerabilities on scada systems*, Tech. report, The University of South Carolina, 2007.

[51] Sidney Valentine and Csilla Farkas, *Software security: Application level vulnerabilities in scada systems*, 2011.

[52] Sidney Valentine and Csilla Farkas, *Software safety and security for programmable logic controllers*, Tech. report, The University of South Carolina, 2013.

[53] Joe Weiss, *Stuxnet: Cybersecurity trojan horse*, InTech (2010).