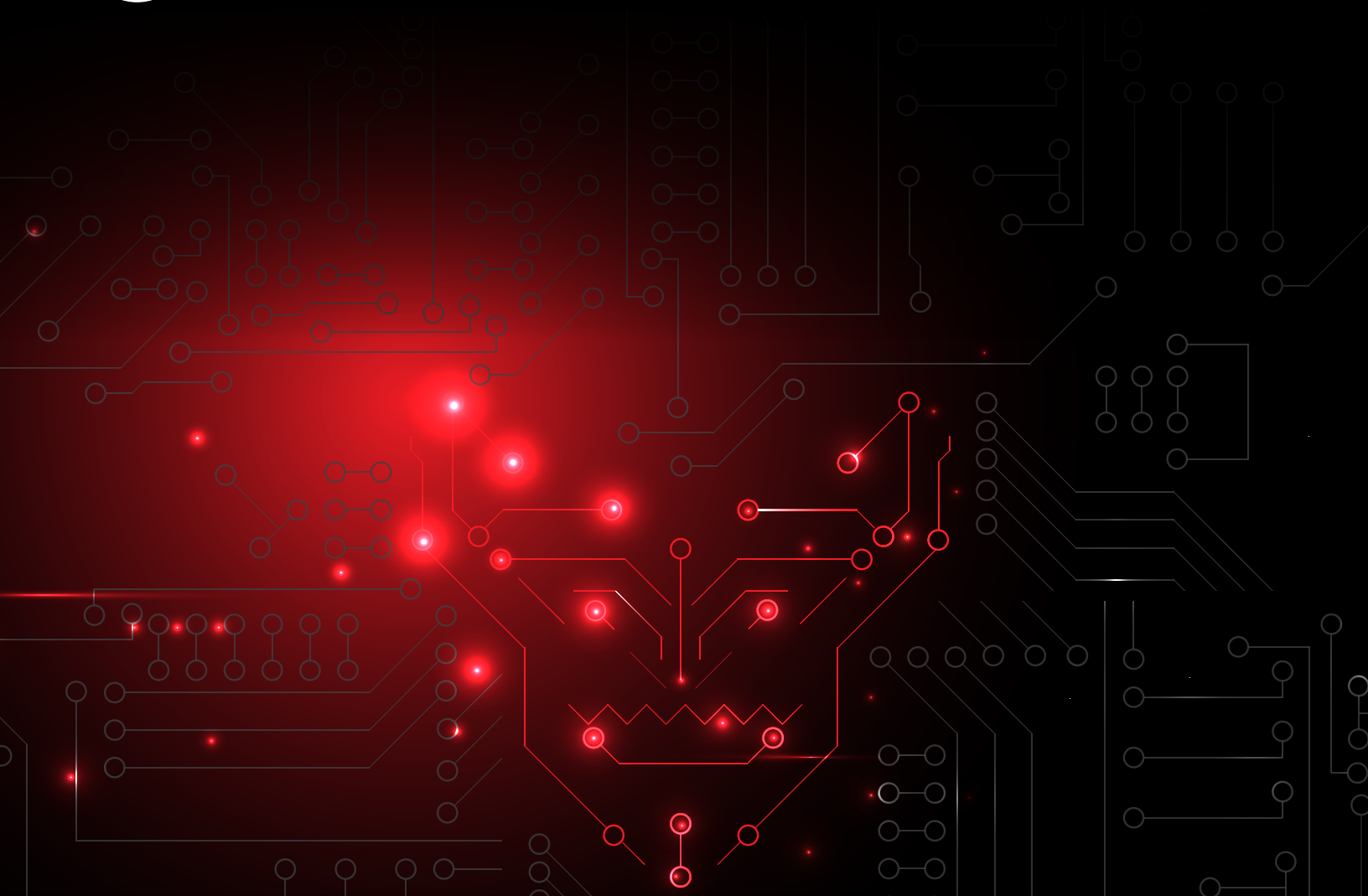
T82
THE CLAROTY RESEARCH TEAM

RESEARCH PAPER



EVIL PLC ATTACK: WEAPONIZING PLCS

Team82, Claroty Research Team

Mashav Sapir, Uri Katz, Noam Moshe, Sharon Brizinov, Amir Preminger

CLAROTY

TABLE OF CONTENTS

TEAM82 RESEARCH

EXECUTIVE SUMMARY

Programmable logic controllers (PLCs) are indispensable industrial devices that control manufacturing processes in every critical infrastructure sector. Because of their position within automation, threat actors covet access to PLCs; several industrial control system malware strains, from Stuxnet to Incontroller/ Pipedream, have targeted PLCs.

But what if the PLC wasn't the prey, and instead was the predator?

This paper describes a novel attack that weaponizes popular programmable logic controllers in order to exploit engineering workstations and further invade OT and enterprise networks. We're calling this the Evil PLC Attack.

The attack targets engineers working every day on industrial networks, configuring and troubleshooting PLCs to ensure the safety and reliability of processes across critical industries such as utilities, electricity, water and wastewater, heavy industry, manufacturing, and automotive, among others.

The Evil PLC Attack research resulted in working proof-of-concept exploits against seven market-leading automation companies, including Rockwell Automation, Schneider Electric, GE, B&R, Xinje, OVARRO, and Emerson.

This paper will describe in depth, not only how engineers diagnose PLC issues, write, and transfer bytecode to PLCs for execution, but also how Team82 conceptualized, developed, and implemented numerous novel techniques to successfully use a PLC to achieve code execution on the engineer's machine.

Below is a list of affected vendors and products, as well as links to their respective advisories and remediations (or mitigations).

AFFECTED VENDORS

Vendor	Platform	EWS	Protocol	CVE
OVARRO	TBOX	TwinSoft	Custom Modbus (Port 502/TCP)	Advisory CVE-2021-22650
B&R (ABB)	X20 System	Automation Studio	ANSL (Port 11169/TCP) INA2000 (Port 11159/UDP)	Advisory CVE-2021-22289
Schneider Electric	Modicon (M340, M580)	EcoStruxure Control Expert (Unity Pro)	Modbus/UMAS (Port 502/TCP)	Advisory CVE-2022-26507
General Electric (GE)	MarkVIe	ToolBoxST	SDI (Port 5311/TCP)	Advisory CVE-2021-44477 CVE-2018-16202
Rockwell Automation	Micro Control Systems	Connected Components Workbench (CCW)	CIP (Port 44818/TCP)	Advisory CVE-2021-27475 CVE-2021-27471 CVE-2021-27473
Emerson	PACSystems	PAC Machine Edition	SRTP (Port 18245/TCP)	Advisory CVE-2022-2788
XINJE	XDPPro	XD PLC Program Tool	Modbus UDP (Port 502/UDP)	Advisory CVE-2021-34605 CVE-2021-34606

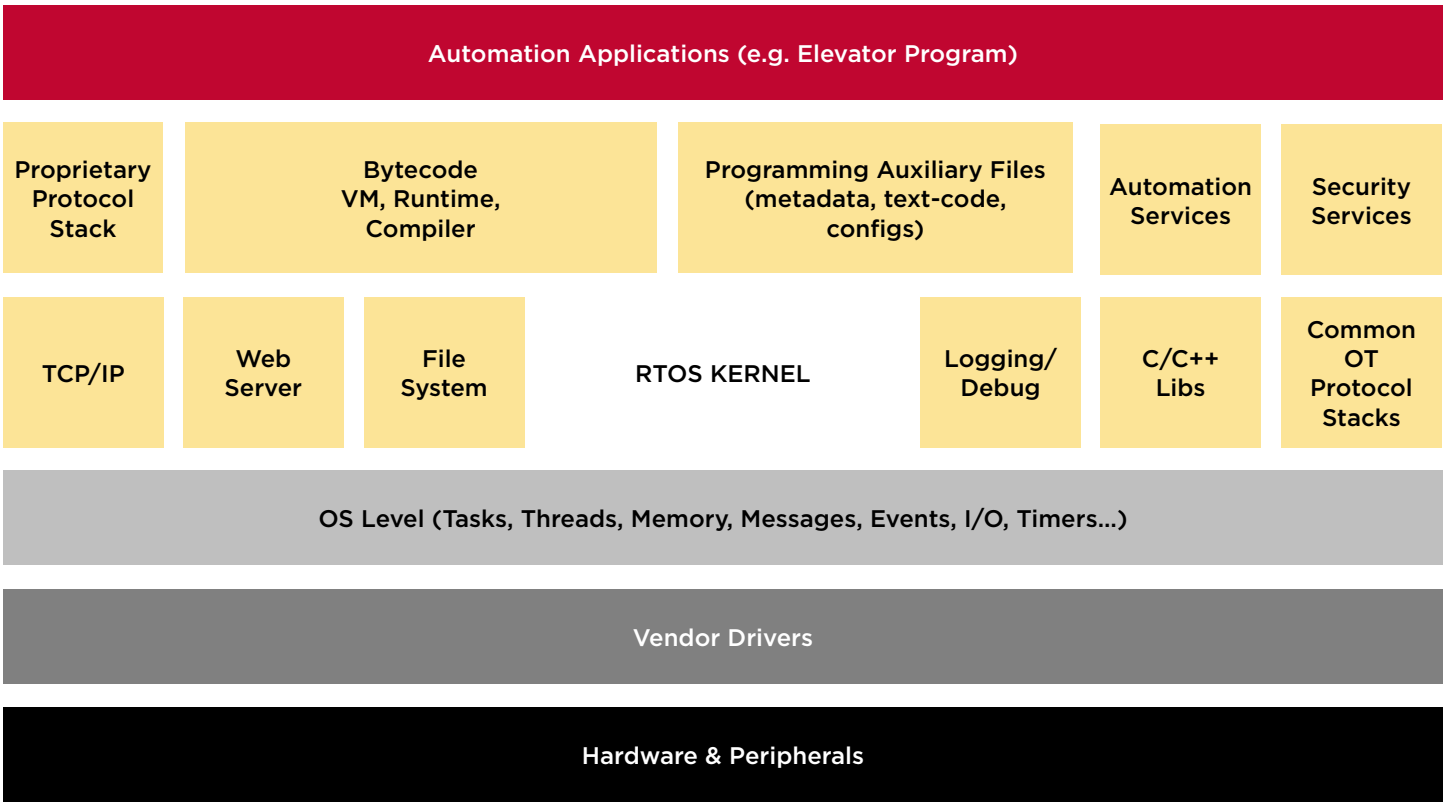
TECHNICAL ANALYSIS

The Relationship Between PLCs and Engineering Workstations

A PLC is a robust computer used to control a machine, small automation process, or an entire production line. It receives data from sensors or input devices, processes this data, and triggers outputs based on the currently loaded code logic and parameters. In addition to orchestrating an automation process, the PLC is also used for monitoring and recording run-time data. A PLC can also automatically start and stop processes, generate alarms if a machine malfunctions, and more.

PLC CPU architecture ranges from NXP ColdFire, to ARM, MIPS, PowerPC, or x86. The operating system, or firmware, is usually based on a commercial real time operating system (RTOS) with vendor-specific modifications; some examples are QNX, uCOS, VxWorks, and others. Newer PLC generations even run the full-scale Linux kernel with some slight modifications.

A PLC's architecture is designed only to control, support, maintain, and monitor an automation process. From the hardware level to the firmware user-mode application, all parts should work together to achieve the goal of executing the code logic the engineer developed and deployed to the PLC.

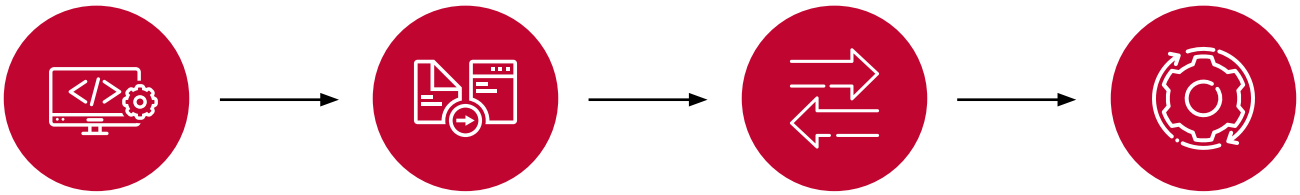


A generic PLC architecture.

As its name suggests, modern controllers are fully programmable and allow engineers to change their logic and behavior by simply rewriting it. To do so, an engineer would use specialized software often called engineering workstation to write and deploy the code to the PLC. For example, Connected Components Workbench is the engineering workstation sold by Rockwell Automation to configure the Micro Control System product line (e.g. Micro820 PLC), and General Electric (GE) ToolBoxST is the software used to control the MarkVI DCS controller series.

Engineering workstation software gives engineers and technicians the tools they need to diagnose, control and maintain the PLCs. Using the engineering workstations it is possible to perform health checks on the PLC, view the current state of all its components including memory variables and physical aspects of the I/O, do firmware upgrades, and modify the PLC code logic.

At its core, an engineering workstation is a fully working integrated development environment (IDE) and compiler for PLC programs. The full process of executing logic on the PLC consists of four main steps.



DEVELOP

PLC code, or textual code, written in either LD, ST, or FBD languages.

COMPILE

Programs are compiled into PLC-compatible bytecode or bindary code.

TRANSFER

Compiled bytecode is transferred to the PLC.

EXECUTE

Once bytecode is successfully delivered to the PLC, the logic is repeatedly executed.

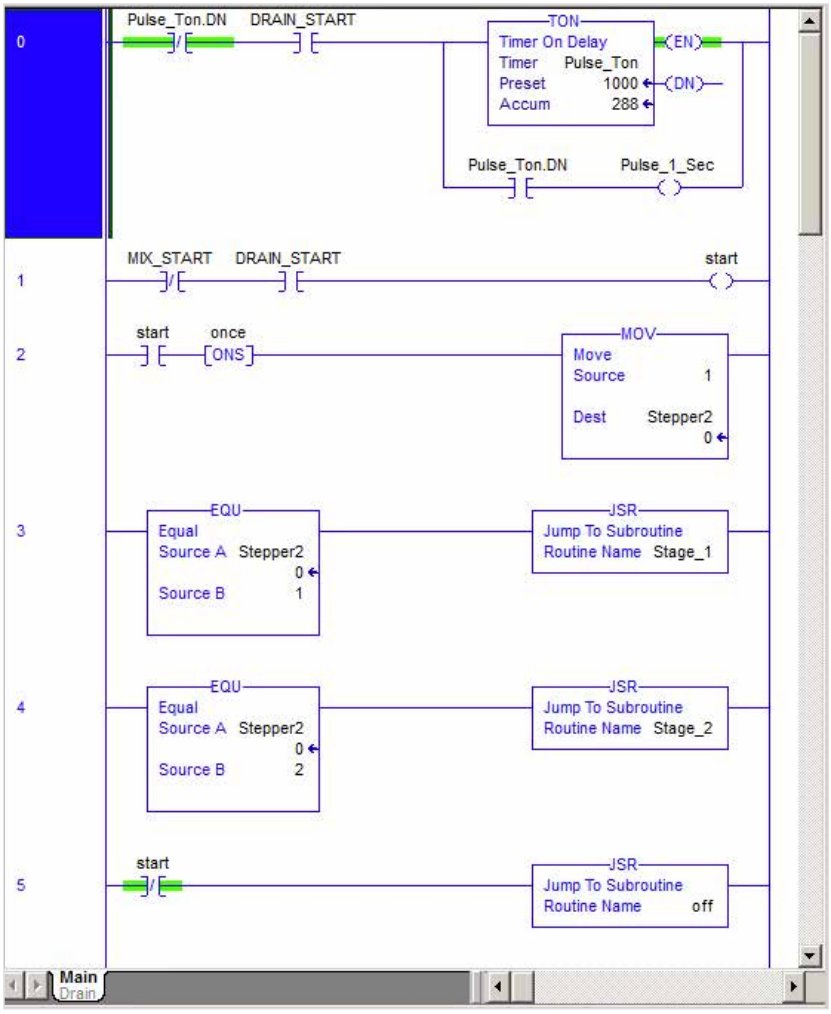
1. **Develop:** The engineer will use the IDE capabilities of the engineering workstation to develop a new PLC program in one of the prominent automation programming languages such as Ladder Diagram (LD), Structure Text (ST), or Function Block Diagram (FBD).
2. **Compile:** When finished, the engineer will want to download the new logic to the controller. To achieve this, the engineering workstation software will compile the program to a PLC-compatible bytecode depending on the firmware and architecture of the target PLC.
3. **Transfer:** Next, the engineering workstation will communicate with the PLC via its proprietary protocol and transfer the compiled bytecode. This process is often called a “Download Procedure,” “Download Logic,” or “Download Configuration.” The download terminology refers to the viewpoint of the PLC in this process (the PLC downloads the code).
4. **Execute:** Finally, after the bytecode has been delivered successfully to the PLC, the logic will be executed natively on the PLC’s CPU. To support this, usually the PLC firmware has a virtual machine de coder that transforms the intermediate bytecode to multiple native machine code instructions.

The high-level textual code language is usually compatible with IEC-61131-3, the open international standard for programmable logic controllers.
The standard determines the objects and resources needed for a PLC to execute logic, including:

- **Language data-types:** BOOL — 1 bit, BYTE — 8 bit, WORD — 16 bit, DWORD — 32 bit, LWORD — 64 bit, etc.
- **Variables:** global/local scope, I/O mapping, etc.
- **Program Organization Unit (POU):** standard lib/ custom functions, function blocks, program, etc.

Such programming languages include graphical and textual programming languages, for example:

- Graphical
  - Ladder Diagram (LD)
  - Function Block Diagram (FBD)
  - Sequential Function Chart (SFC)
- Textual
  - Structured Text (ST)
  - Instruction List (IL)



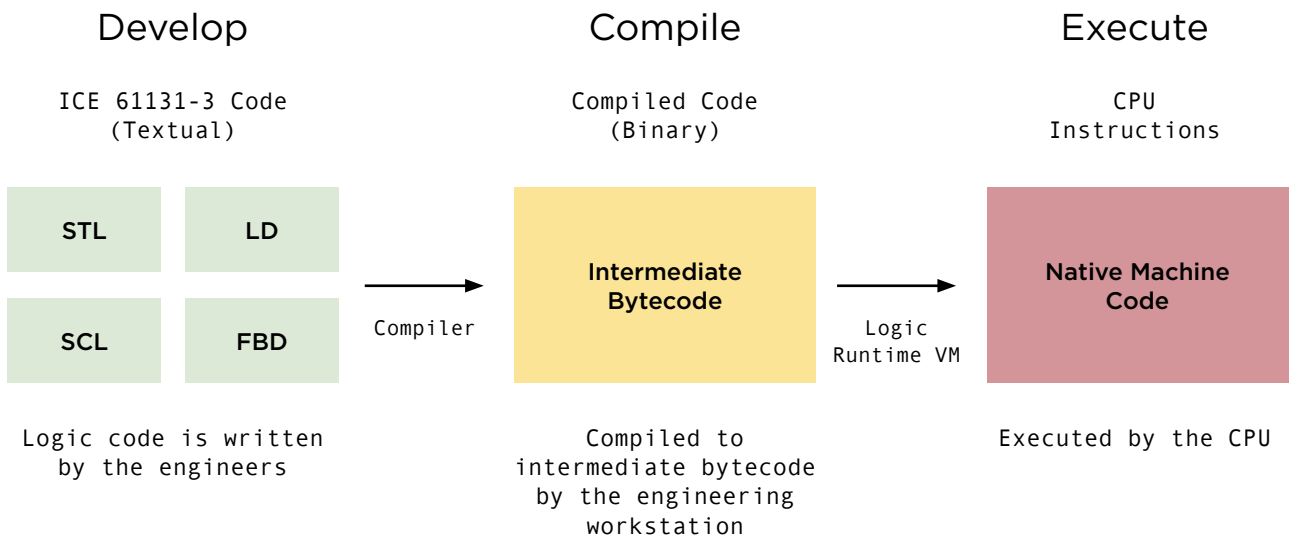
Example of a Ladder Diagram-based program we built in Rockwell Automation's RSLogix5000 engineering workstation (RSLogix5000 is not affected by Evil PLC Attack). The program controls a simple drinking machine automation process.

As mentioned above, the engineering workstation will compile the user program to an intermediate bytecode that will be transferred later to the PLC. The bytecode will be handled by the PLC runtime virtual machine (VM) that will decode each instruction and jump to a specific routine that handles it. Eventually a native machine code instruction will be executed by the CPU.

From a high-level perspective we can summarize the entire PLC-programming code transformation as follows:

```
SOR 0x1E6AB8
NOP 0x3D1988
XIC 0x0 #[0x60274]
OTL 0x3 #[0x6045C]
IRD
OTU 0x3 #[0x6045C]
CMT 0x1E6AB8
SOR 0x1E6AD4
NOP 0x1757EB
XIC 0x0 #[0x60478]
OTL 0x1 #[0x6045C]
IRD
OTU 0x1 #[0x6045C]
CMT 0x1E6AD4
SOR 0x1E6AF0
NOP 0x5B56A2
XIC 0x0 #[0x603B0]
OTL 0x4 #[0x6045C]
IRD
OTU 0x4 #[0x6045C]
CMT 0x1E6AF0
SOR 0x1E6B0C
NOP 0x50D8DD
XIC 0x0 #[0x601F0]
OTL 0x2 #[0x6045C]
IRD
OTU 0x2 #[0x6045C]
CMT 0x1E6B0C
SOR 0x1E6B28
NOP 0x7A530C
XIC 0x0 #[0x603CC]
OTL 0x0 #[0x6045C]
IRD
OTU 0x0 #[0x6045C]
CMT 0x1E6B28
SOR 0x0
```

The user program will be compiled to an intermediate bytecode that can be represented as simple assembly-like instructions. In this image, you can see our Control-Logix Rockwell Automation disassembler in action (ControlLogix is not affected by Evil PLC Attack).



PLC code logic transformation: From IEC 61131-3 compatible plain-text code to native machine code execution.

Upload and Download Procedures

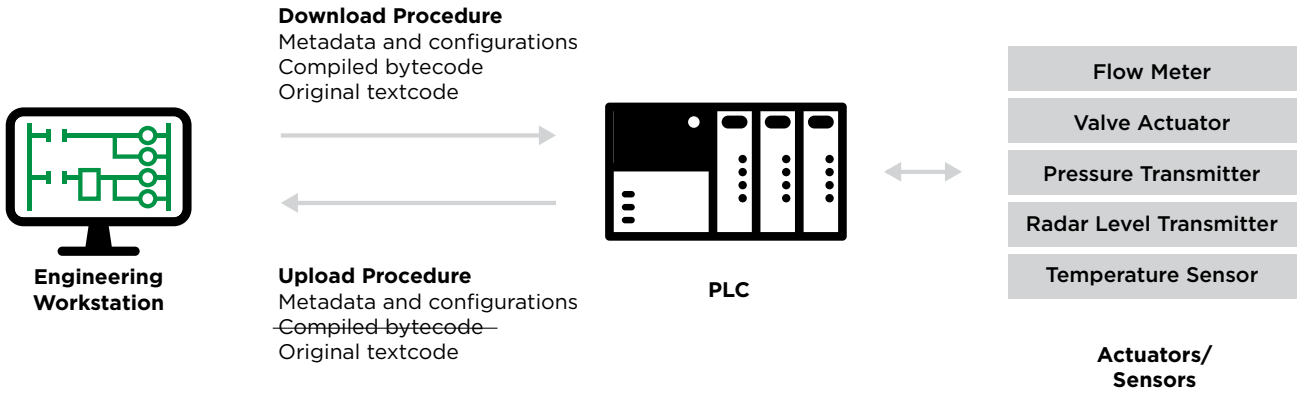
Along with the code logic bytecode, which is the most important piece of information from the PLC's perspective, the engineering workstation would also transfer auxiliary pieces of information, including:

- **Metadata:** project and program names, symbols data, dates (compilation, transfer), information about the engineering workstation, and more.
- **Configurations:** hardware/network settings, memory maps and tags, I/O configuration, variable definitions, parameters, and more.
- **Original Text Code:** source-code the engineer developed (plain-text code or binary serialized representation of the logic).

In many cases, PLCs do not process the metadata or the original text code. This type of data is stored on the PLC to support the ability of the engineer to retrieve a working project from the PLC without needing a local copy beforehand. The transfer and storage of the data on the PLC is called the download procedure.

To retrieve a working copy of the current PLC logic, an engineer will perform an upload operation that would read stored data from the PLC. The data includes metadata the engineering workstation software requires, including numerous data objects rather than just the compiled program the PLC requires to operate. This functionality is often used for maintenance and diagnostics purposes, but can also be considered a backup in case the engineer does not have a copy of running logic.

The fact that the PLC stores additional types of data that are used by the engineering software and not the PLC itself, led us to explore the ability to modify the unused data stored on the PLC to manipulate the engineering software. This topic yielded the advanced PLC hiding logic techniques and the Evil PLC attack technique discussed in this paper.



Understanding download and upload procedures from a PLC's point of view.

This architectural design is common practice in the industry and shared across most if not all ICS vendors. It allows engineers to switch between different engineering workstations and quickly perform upload procedures to retrieve the currently running program by getting the actual source code from the PLC. Therefore, each engineer can connect to the PLC, get the most updated version of the source code and configuration, and continue development from there.

This design choice creates an interesting situation whereby in order to continue code development for an automation process, some engineers use the upload procedure instead of using static project files, since the most recent versions of the code logic and configurations are stored on the PLC.

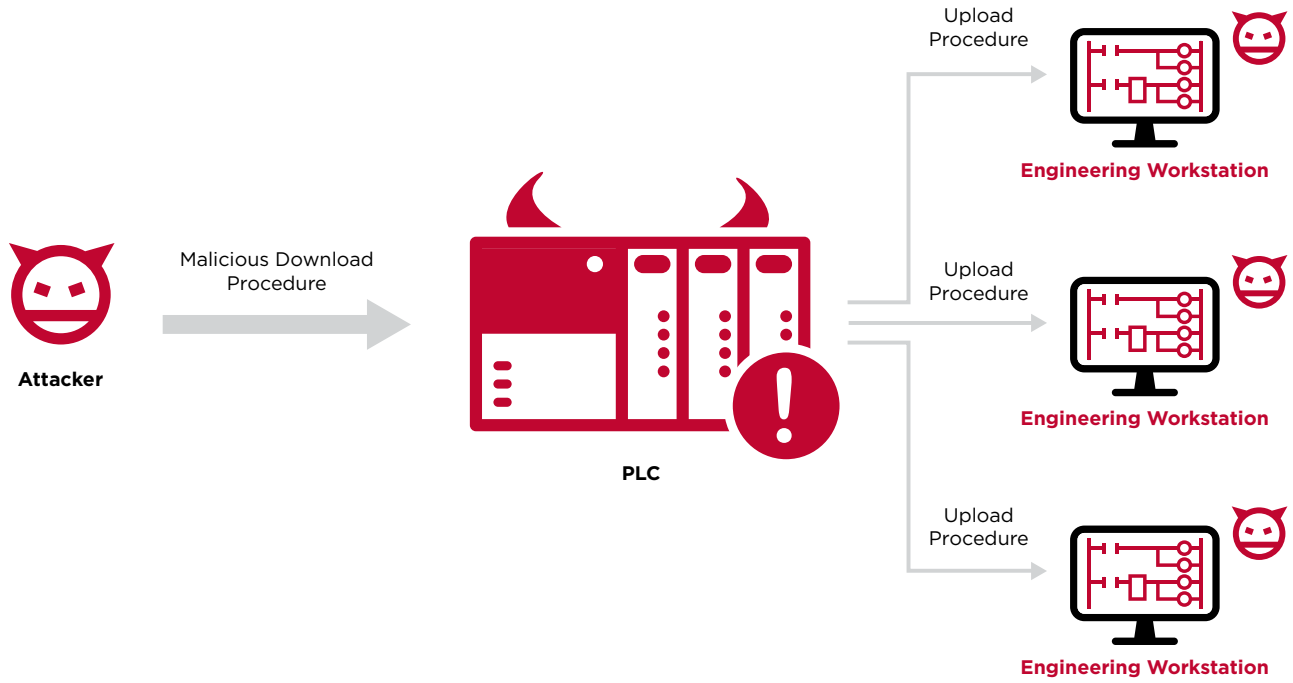
Evil PLC Attack

Most attack scenarios that involve a PLC revolve around accessing and exploiting the controller. PLCs are attractive targets for threat actors because a typical industrial network may have dozens of PLCs performing different operations; an attacker wishing to physically disrupt a process, for example, would need to perform a long enumeration of PLCs in order to find the precise one to attack.

In our research we decided to look for a different approach by focusing on the PLC as the tool rather than the target. In this case, we want to leverage the PLC in order to access its maintainer, the engineering workstation. Once owned, the engineering workstation would be the best source for process-related information and would have access to all the other PLCs on the network. With this access and information, the attacker can easily alter the logic on any PLC.

The quickest approach to luring an engineer to connect to an infected PLC would be for the attacker to cause a malfunction or a fault on the PLC. That will compel the engineer to connect using the engineering workstation software as a troubleshooting tool.

Through our research, we tried to execute this new attack vector against multiple leading ICS platforms. We found various vulnerabilities in each platform that allowed us to weaponize the PLC in a way that when an upload procedure is performed, our specifically crafted auxiliary pieces of data would cause the engineering workstation to execute our malicious code.



We believe that Evil PLC Attack is a new attack technique. This technique weaponizes the PLC with data that isn't necessarily part of a normal static/offline project file, and enables code execution upon an engineering connection/upload procedure. Through this attack vector, the goal is not the PLC, such as it was, for example, with the notorious Stuxnet malware that stealthily changed PLC logic to cause physical damage. Instead, we want to use the PLC as a pivot point to attack the engineers who program and diagnose it and gain deeper access to the OT network.

It's important to note that all the vulnerabilities we found were on the engineering workstation software side and not in the PLC firmware. In most cases, the vulnerabilities exist because the software fully trusted data coming from the PLC without performing extensive security checks.

Attack Scenarios

When looking at our Evil PLC attack vector, we reach a conclusion that it could be utilized in both offensive and defensive fashions. With that in mind, we devised three different attack scenarios where this new attack vector could be used:

- **Weaponizing PLCs to Achieve Initial Access:** Attackers could use weaponized PLCs in order to gain an initial foothold on internal networks, or even for lateral movement.
- **Attacking Traveling Integrators:** Attackers could target system integrators and contractors as a means of entry to many different organizations and sites around the world.
- **Weaponizing PLCs as a Honeypot:** Defenders could use honeypot PLCs to attract and attack possible attackers, thus deterring and frustrating would-be attackers.

Weaponizing PLCs to Achieve Initial Access

Currently, there are hundreds of thousands of ICS devices exposed to the internet, as determined by most public internet scanning services, including Shodan and Censys. These internet-facing devices usually lack security and allow anyone to access them, modify parameters, and even alter their behavior and logic via download procedures.

Protocol	Censys Query	# of Exposed Devices	Top Countries
Modbus	services.service_name= `MODBUS`	36,387	United States, South Korea, France
Niagara Fox	services.port = 911 or services. port = 4911 and niagara	4,175	United States, China, Moldova
BACnet	services.service_name= `BACNET`	13,973	United States, Canada, France
Siemens S7	(services.port = 102) and services.service_name=`S7`	7,308	Japan, Germany, Italy
DNP3	services.service_name= `DNP3`	832	United States, Poland, China
Ethernet/IP	services.service_name= `EIP`	7,231	United States, Canada, Spain

Examples of internet-exposed device statistics from Censys search engine.

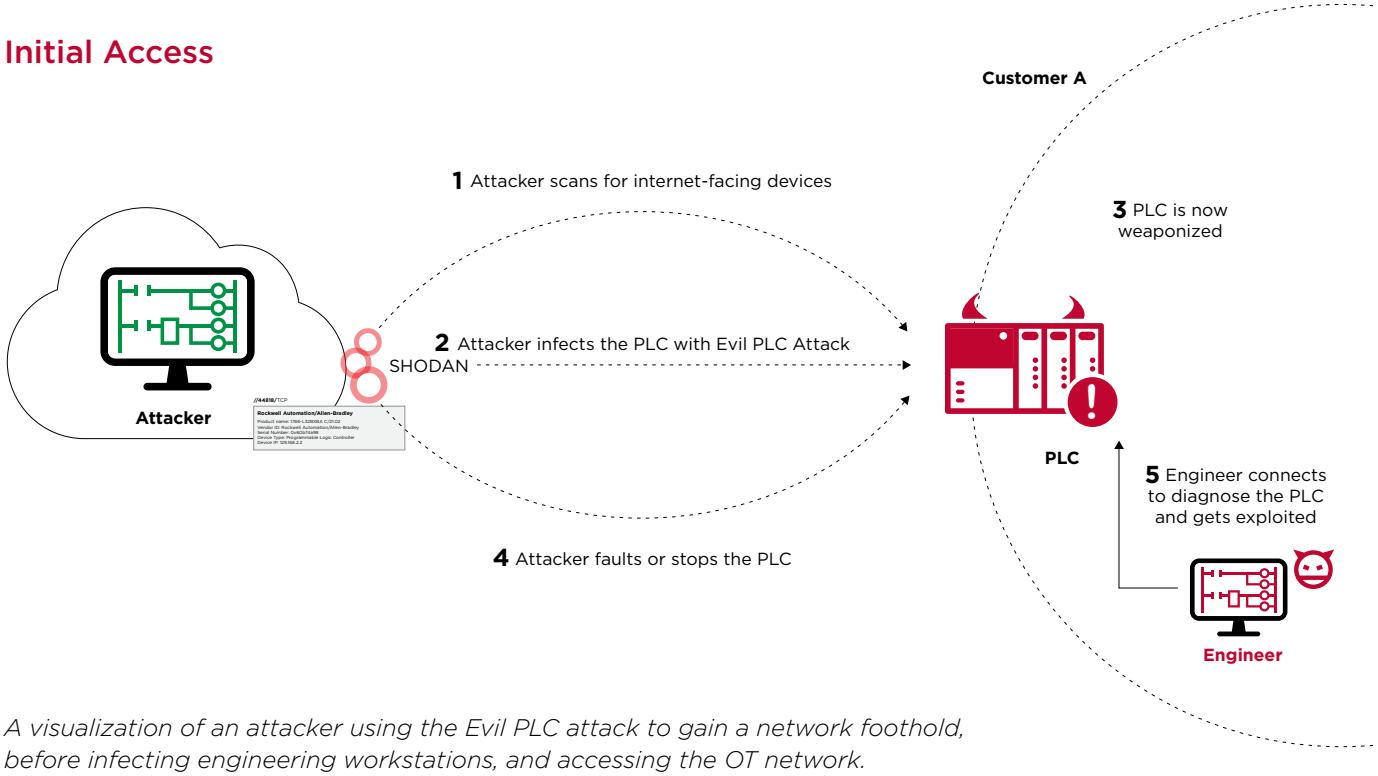
PLCs are classic entry points to critical infrastructure when left publicly accessible. This point has been discussed multiple times in CISA alerts and vulnerability advisories, asking asset owners to secure access to PLCs due to their unsecure nature. Examples of such alerts/advisories are CISA Alert AA20-205A and Shields Up. The key takeaway: In order to perform an attack on the majority of PLCs, the only tool needed is the commercial engineering workstation software provided by the PLC venor.

In the last few years, we've seen examples of such opportunistic attackers and learned their process of exploitation. First, they identify internet-facing PLCs, connect to them using commercial engineering workstation software, and upload the current project, which includes code and settings from the PLC.

Then, the attackers will modify the logic of the project, and perform a download procedure to change the PLC logic with their modifications. One example was the 2020 attack on Israel's water supply, where attackers exploited accessible PLCs and attempted to flood the water supply with chlorine.

Our research suggests that attackers could use the internet-facing PLCs as a pivot point to infiltrate the entire OT network. Instead of simply connecting to the exposed PLCs and modifying the logic, attackers could arm these PLCs and deliberately cause a fault that will lure an engineer to them. The engineer, as a method of diagnostics, will perform an upload procedure that will compromise their machine. The attackers now have their foothold on the OT network.

Initial Access

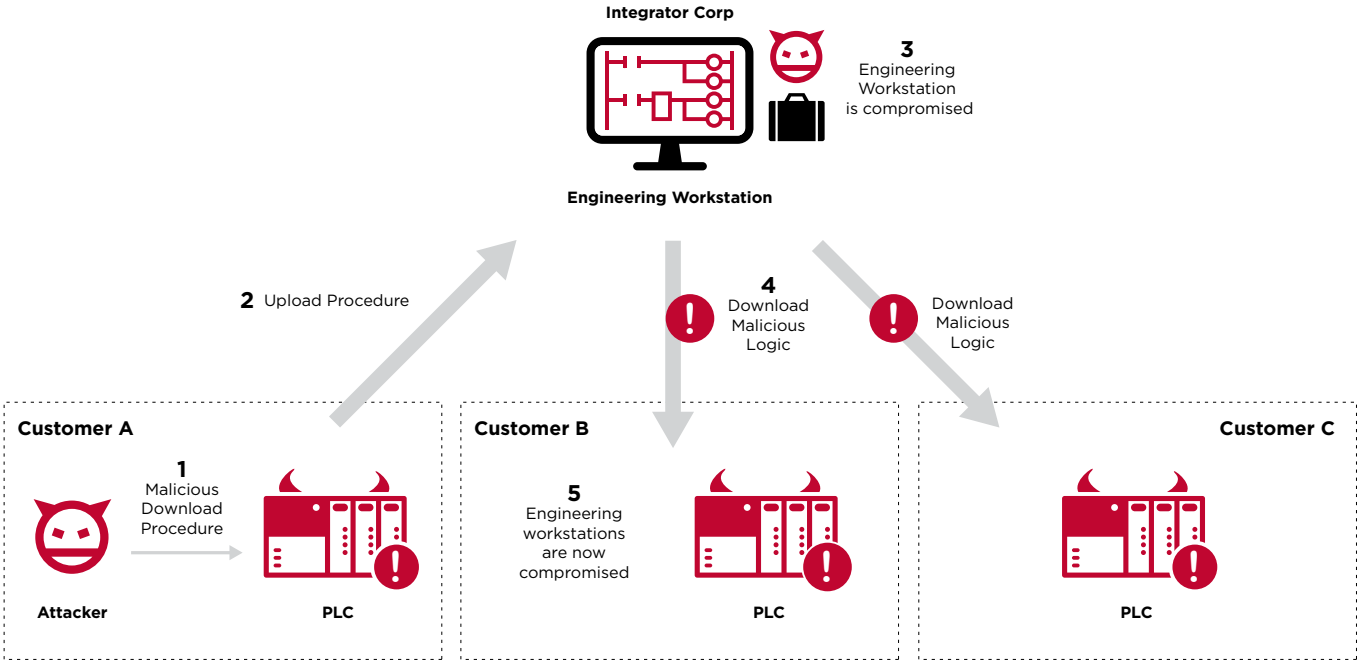


A visualization of an attacker using the Evil PLC attack to gain a network foothold, before infecting engineering workstations, and accessing the OT network.

Traveling Integrators: Another use case for this new attack vector becomes clear when examining modern OT management procedures. In many cases, third-party engineers and contractors manage and interact with many different OT networks and PLCs. With that in mind, attackers could use those system integrators as a pivot point, expanding their reach drastically.

The attack would look like this: An attacker would locate a PLC in a remote, less secure facility that is known to be managed by a system integrator or contractor. The attacker will then weaponize the PLC and deliberately cause a fault on the PLC. By doing so, the victim engineer will be lured to the PLC in order to diagnose it. Through the diagnosis process, the integrator will do an upload procedure and have their machine compromised. After gaining access to the integrator's machine, which by design is able to access many other PLCs, attackers could in turn attack and even weaponize newly accessible PLCs inside other organizations, broadening their control even further.

Traveling Integrators

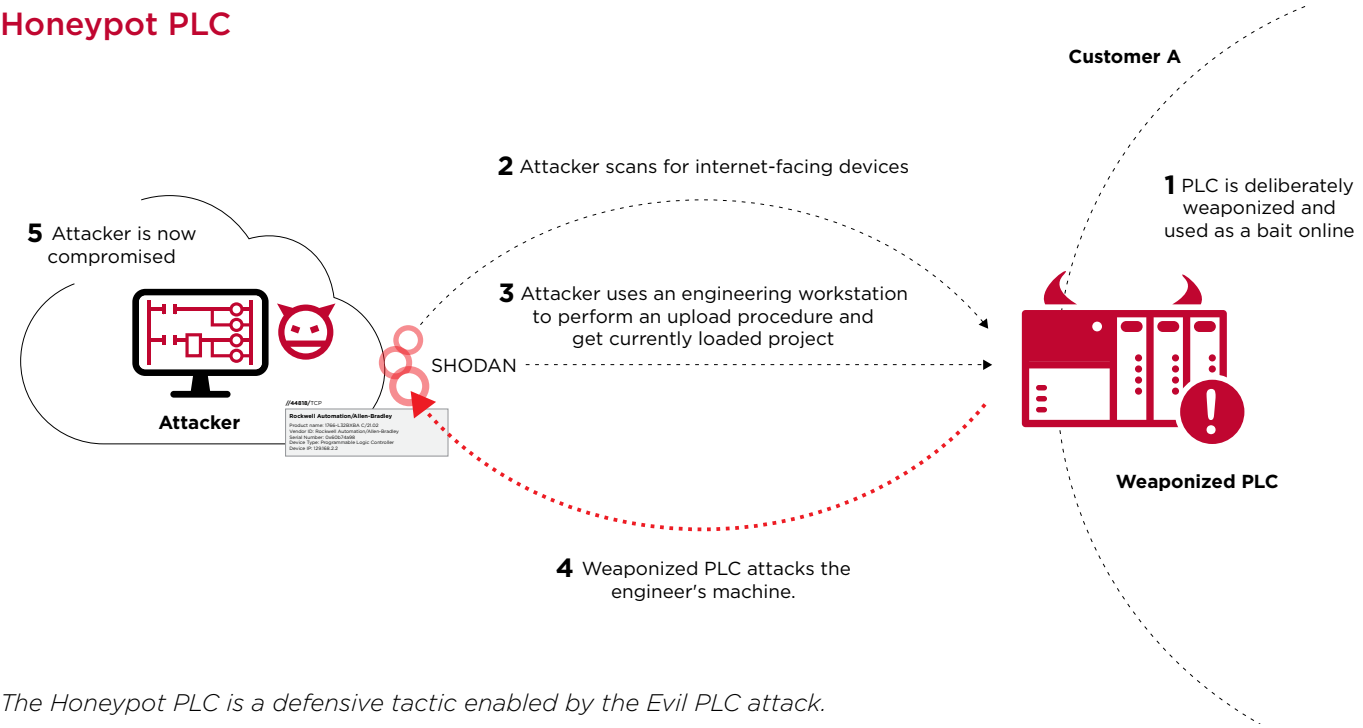


A second Evil PLC attack vector involves targeting integrators and third parties using vulnerable engineering workstation software.

Weaponizing PLCs as a Honeypot:

From a defensive perspective, this new attack vector could be used to lay traps for possible attackers. By leveraging the fact that the attacker also uses the same commercial tools as the engineers, defenders can purposely set up publicly facing weaponized PLCs, and allow attackers to interact with them. These PLCs will act as a honeypot, attracting attackers to interact with them. However, if an attacker falls into the trap and performs an upload from the decoy PLC as part of the enumeration process, our weaponized code will execute on the attacker's machine. This method can be used to detect attacks in the early stage of enumeration and might also deter attackers from targeting internet-facing PLCs since they will need to secure themselves against the target they planned to attack.

Honeypot PLC



The Honeypot PLC is a defensive tactic enabled by the Evil PLC attack.

Research and Methodology

In the scenarios we will explain in this paper, we assume the attacker or defender already has network access to the PLC and the ability to perform a download procedure. Attackers may gain a foothold on an OT product in a number of ways that have already been demonstrated, including being legacy and "insecure by design", the existence of hardcoded keys (e.g. OVARRO TBox, Rockwell Automation ControlLogix), authentication bypass exploits (e.g. Schneider Electric M221), and other broken-cryptography attacks that eventually allow access to the PLCs.

We decided to focus on the following seven targets:

- **OVARRO:** TBox platform
- **B&R** (by ABB Group): X20 System platform
- **Schneider Electric:** Modicon platform (mainly M340, M580)
- **General Electric (GE):** Mark VIe platform
- **Rockwell Automation:** Micro800 Control Systems platform
- **Emerson:** PACSystems platform
- **Xinje:** XD Series platform

The testbed setup in Team82's lab.

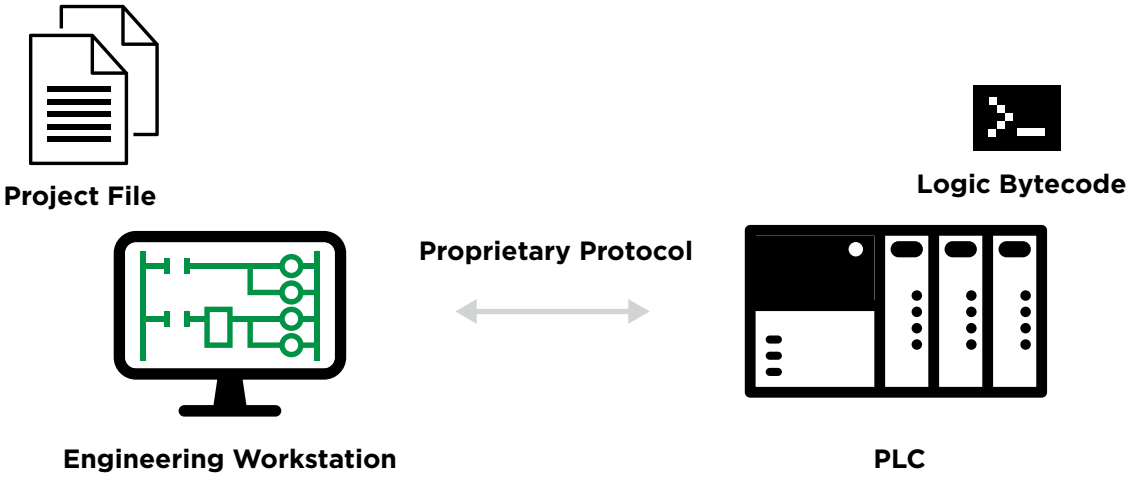
For each target/platform we tried to understand the whole download/upload mechanism by reverse engineering the firmware and the engineering workstation software. Our goal was to find discrepancies between what the PLC is using and what engineering workstation is using. If we were to find such inconsistencies, we could weaponize the PLC through a malicious download procedure to store a specifically crafted piece of data that won't affect the PLC, but when parsed by the engineering platform it will trigger and exploit a vulnerability.

The research process of our ***Evil PLC Attack*** is as follows:

1. **Setup:** Setting up a testbed environment with a target PLC, compatible engineering workstation, and I/O field devices.
2. **Building "Hello World:"** Reading PLC manuals, watching instructional videos, and building a benign program to control simple processes.
3. **Project File:** Explore what is being stored in a project file (metadata, configurations, textcode) and how the data is serialized.
4. **Reverse Engineering:** Exploring the PLC hardware and firmware in addition to the engineering workstation software.
5. **Upload/Download Procedures:** Understand the mechanics of the upload/download procedures, and what data is transferred through the proprietary protocol.
6. **Protocol Analysis:** Analyze the proprietary protocol and its functionality, and build a fully featured client.
7. **Find Discrepancies:** Understand the differences between what information is transferred and stored in the PLC, without being parsed or used.
8. **Hunt for Vulnerabilities:** Research all the parsing code flows of all pieces of information that the engineering workstation transfers to the PLC that are not used/modified on the PLC.
9. **Weaponize:** Using the client, implement a malicious download procedure that stores specifically crafted data on the PLC.
10. **Exploit:** Engineer connects to the PLC and performs an upload procedure. The engineering workstation parses the specifically crafted data we implemented. The parsing flow triggers the vulnerability and executes our code.

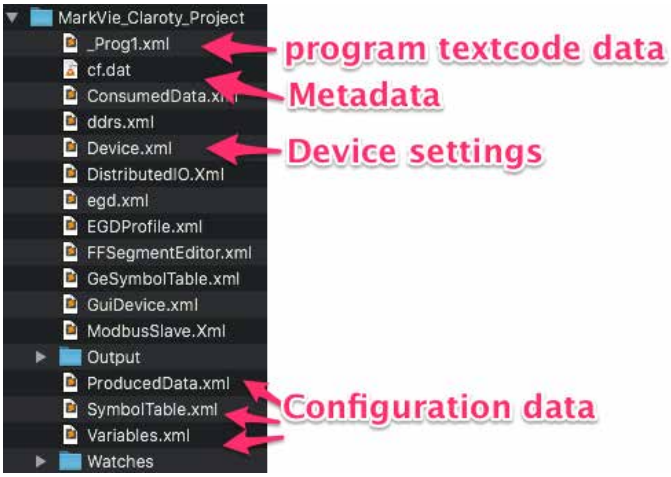
For each target vendor platform, the full ecosystem that we needed to research consist of:

- The engineering workstation software
- Engineering workstation project file
- Proprietary protocol (PLC ⟷ Engineering workstation)
- PLC firmware
- Logic code (bytecode, textcode)



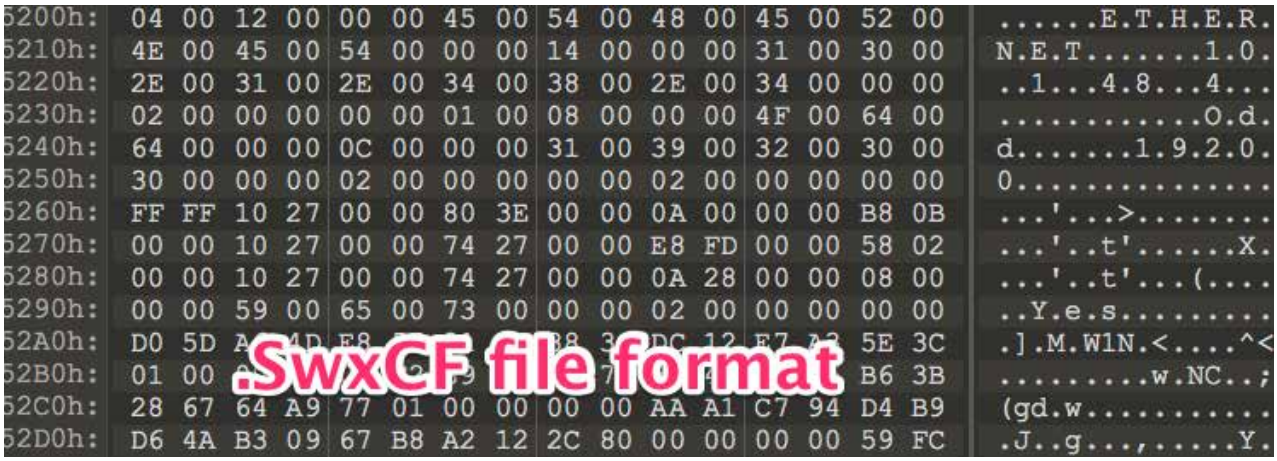
Process components studied by Team82 in order to develop the Evil PLC attack.

In many cases, the project files were simple archive-based files with relatively easy-to-understand structures; in most cases we started from there. For example, the GE MarkVIe project file contains textual XML files with easy-to-understand naming conventions. It helped us to understand the full scope of the types of data the engineering workstation is generating and transferring to the PLC.



A GE ToolBoxST project file.

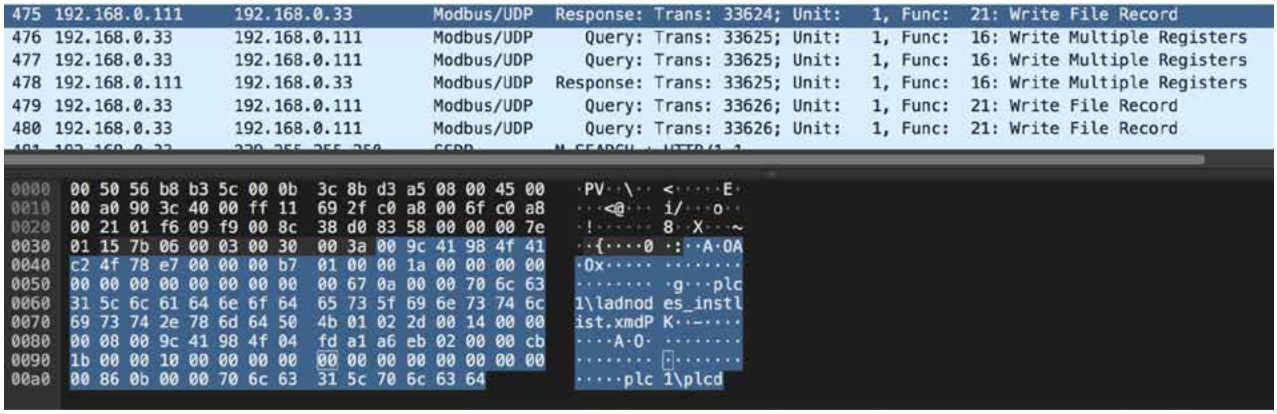
However, some project files are stored in a proprietary format that we needed to analyze and reverse engineer. For example, the Emerson engineering workstation, PAC Machine Edition, is using a proprietary `.SwxCF` project file format that is binary-based MFC serialized data. This project file stores hundreds of serialized C++ objects required for a complete operational project. To complete our research, we wrote parsers for all the file formats and extracted the source code logic, metadata, and configuration data of the projects.



Screenshot of a Emerson PAC Machine Edition .SwxCF project file. It contains binary-based MFC serialized objects.

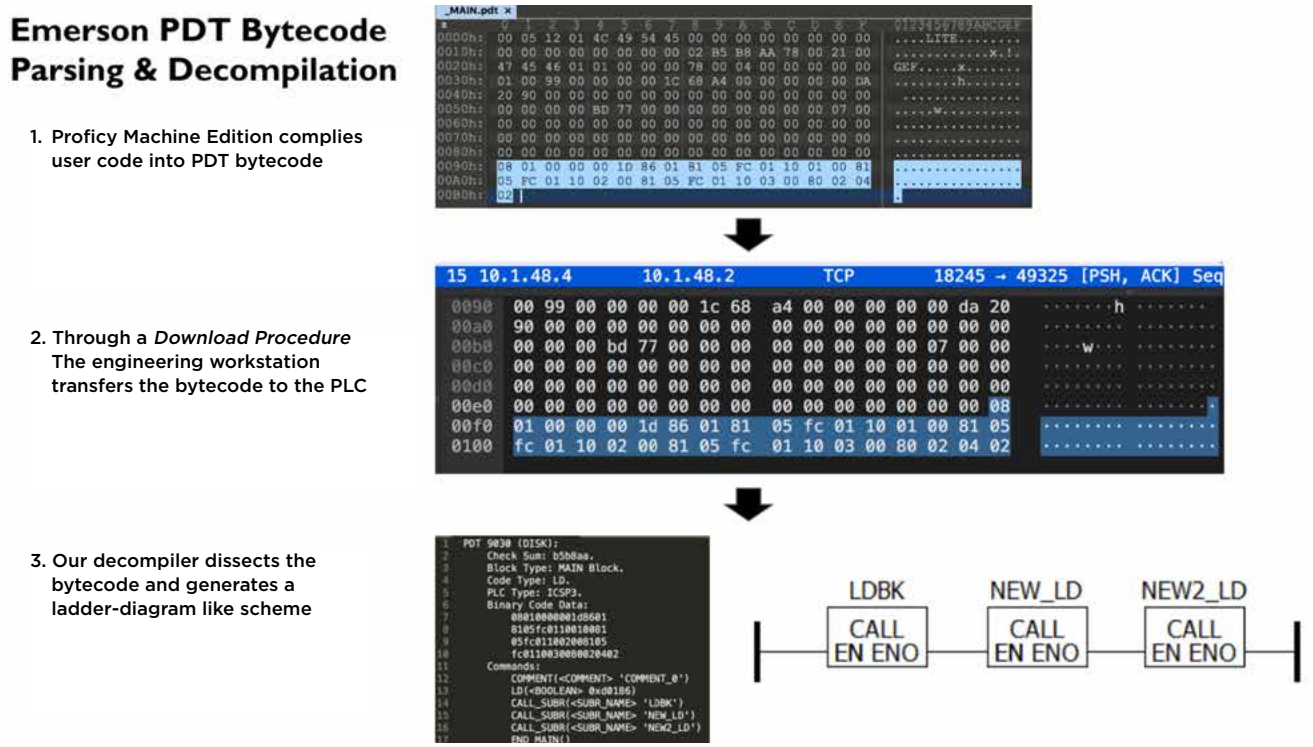
Another important aspect we needed to investigate was the proprietary protocol that each vendor developed and implemented. Using this protocol, the engineering workstation software can communicate with compatible PLCs and perform various actions including getting its status, perform Supervisory Control And Data Acquisition (SCADA) operations, perform firmware upgrades, and most importantly for us, perform upload/download procedures to modify or obtain the currently running logic. Some protocols were easy to analyze while others were complex. Eventually we developed parsers and clients for all the proprietary protocols in the scope of this research project.

For example, the Xinje XD engineering workstation, PLC Program Tool, is using a UDP Modbus-based protocol to communicate with the PLC. Most of the functionality is implemented via Modbus read/write file records and/or multiple registers Modbus function codes.



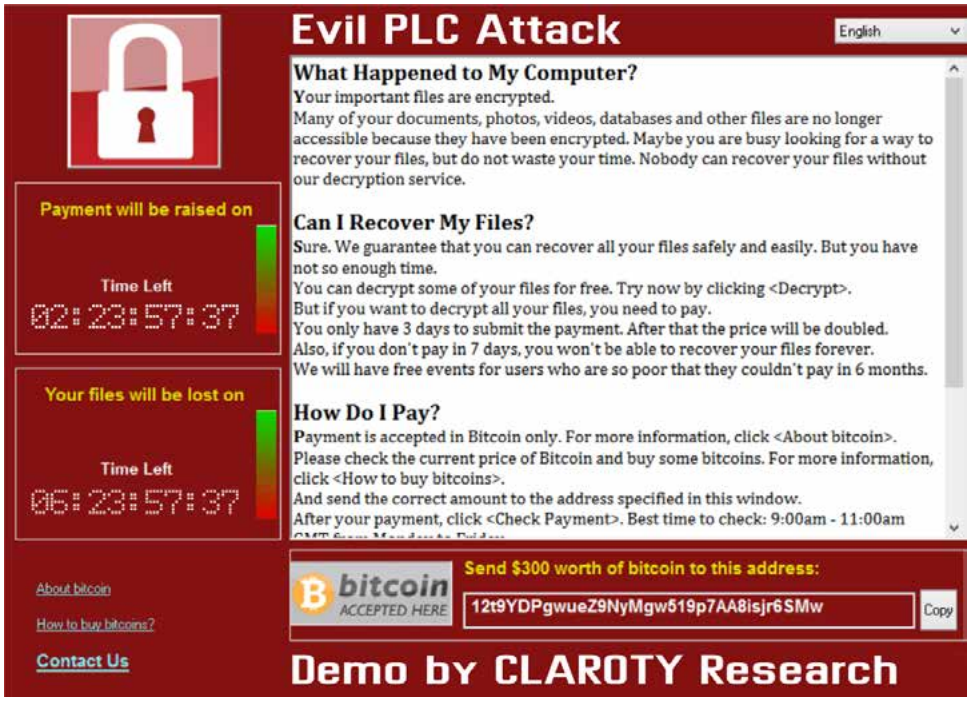
A Wireshark screenshot of Xinje protocol network traffic captured during an upload procedure.

In addition to the project file and the proprietary protocol, we researched the textcode (source code) and the bytecode (compiled code). This was an important aspect of our work because it allowed us to perform full-scale download/upload procedures and helped us automate and gain control over the PLC logic. Although it wasn't necessary in our research, in some cases we developed a partial/fully working compiler and decompiler for the proprietary PLC bytecode.



High level process of our Emerson PDT bytecode decompiler.

Finally, for our exploit payload and demonstration purposes, we prepared a fake WannaCry ransomware that simply presents the user with a scary looking popup ransom note.



Our proof-of-concept demo payload.

Findings

Using our thorough research methods, we were able to find seven previously unreported vulnerabilities that allowed us to weaponize the affected PLCs and attack engineering workstations whenever an upload procedure occurred. All of the findings were reported to the affected vendors in accordance with Team82's coordinated disclosure policy.

Let's look at each of the seven targets we researched and how the Evil PLC attack impacts each:

OVARRO TBox Platform

OVARRO describes its TBox platform as follows:

"Our range of TBox RTUs open up new automation possibilities, simplifying systems engineering and enabling critical industries across the globe to remotely control and monitor their applications. TBox allows users to access networks with their mobile devices and PCs - anytime and anywhere. All TBox devices and connected assets are protected by a state-of-the-art cyber security suite with authentication, encryption, firewall, SSL/TLS, HTTPS, SMTPS, SFTP/FTPS and VPN" (source)

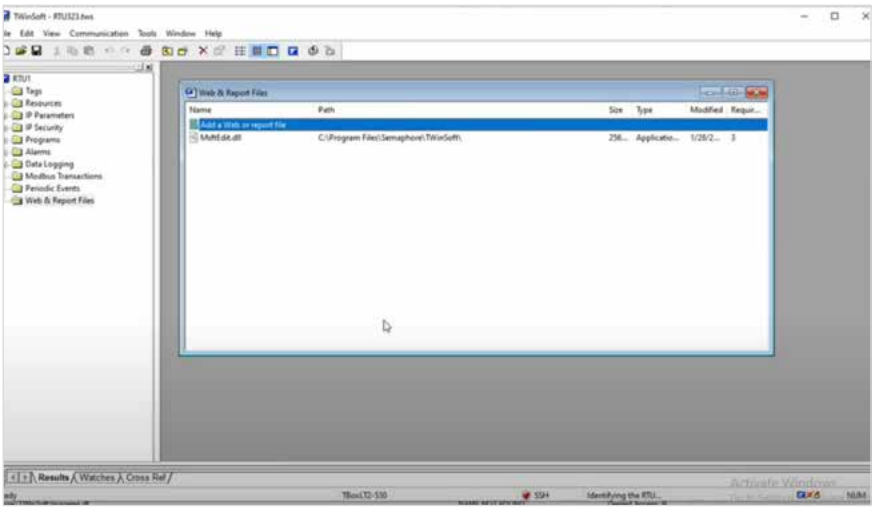
The TBox Remote Terminal Unit (RTU) platform provides HMI and PLC capabilities on a single platform with versatile connectivity options. The devices are widely deployed in areas that have limited resources and require remote connectivity such as water utilities, oil and gas, and other critical infrastructure sectors.

TBox's controller is based on an ARM architecture and runs Linux as its operating system. From a security research perspective, it was very convenient to work with a Linux kernel operating system due to the extensive set of features it offers.

To program the TBox controller, one needs to use the TWinSoft engineering workstation software. The C/C++-based engineering workstation allows operators to configure, control, and program the device. All the communication between TWinSoft and the controller is done via a custom Modbus protocol over TCP port 502. For example, to write a file to the disk, one can simply use the Modbus write functionality. Eventually through our research we found multiple vulnerabilities that were disclosed to the vendor.

We discovered that a remote attacker could update the source project file (TPG) stored within TBox with a malicious project file exploiting a vulnerability in TWinSoft as soon as the engineer performs an upload procedure. OVARRO assigned CVE-2021-22650 to this vulnerability and fixed it in TWinSoft version 12.5 or later. OVARRO also published an advisory TBOX-SA-2021-0004 discussing this vulnerability.

OVARRO TBox LT2-530 RTU



Ovarro TWinSoft Engineering Workstation

Evil PLC Attack Demo: Ovarro TBox Platform.

B&R (ABB) x20 Platform

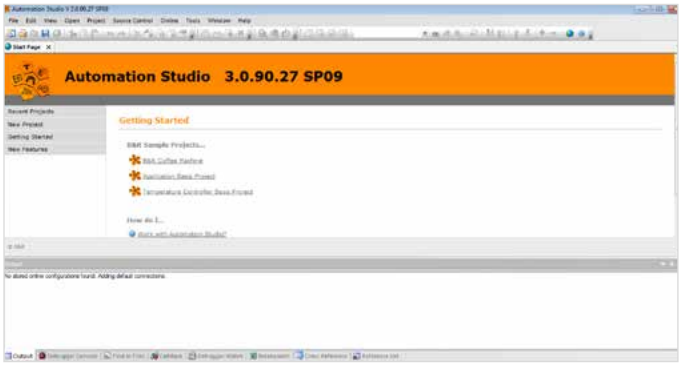
B&R (by ABB group) describes the x20 platform as follows:

"With their compact and powerful products, X20 control systems offer the perfect solution for handling any task large or small. B&R I/O products can be easily added to the controllers; the x20 "slice" system offers the greatest possible flexibility here." (source).

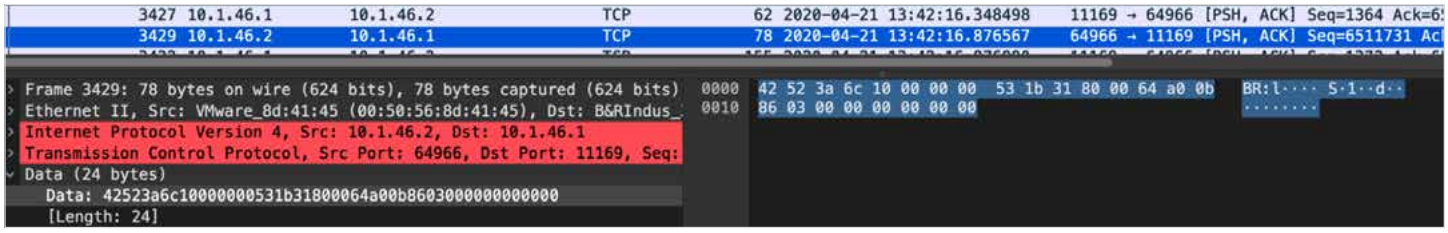
B&R X20 series offers a modular and compact PLC configuration. It is based on an Intel ATOM CPU (X86) or ARM architecture, and runs VxWorks as its RTOS firmware. The x20 PLCs are deployed across many European manufacturing plants, critical infrastructure sectors, and maritime environments.

In order to program the controller, engineers use the C/C++ based Automation Studio (AS) tools, which allow them to write projects to the PLC, change its settings, and perform general maintenance. The X20 systems communicate with Automation Studio using proprietary protocols developed by B&R. Newer Automation Studio versions use the ANSL protocol over TCP port 11169, and older versions use the protocol INA2000 over TCP/UDP port 11159 (newer versions can also fall back to INA2000 in case ANSL session can't be established for some reason).

B&R X20CP1585 PLC



B&R Automation Studio Engineering Workstation



A B&R ANSL packet.

We found that if the PLC has not been sufficiently secured, an attacker could manipulate the stored project information and inject a file with a path traversal name. Alternatively, a remote attacker may use spoofing techniques to make B&R Automation Studio connect to an attacker-controlled device with manipulated project files. When performing a project upload procedure in B&R Automation Studio, such crafted projects will be loaded and opened in the security context of Automation Studio. This may result in remote code execution, information disclosure and denial of service of the system running B&R Automation Studio. B&R assigned CVE-2021-22289 and published an advisory on this topic.

Evil PLC Attack Demo: B&R X20 Platform.

Schneider Electric Modicon M340/M580 Platform

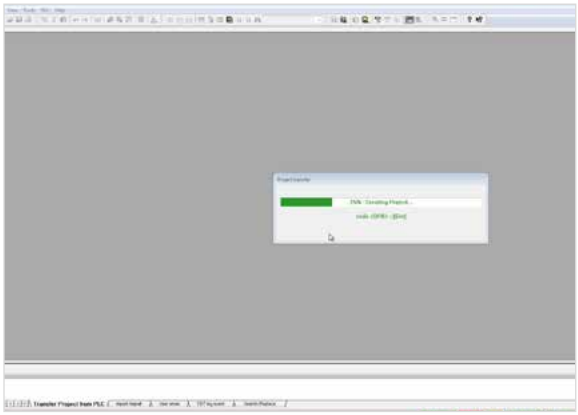
Schneider Electric describes the Modicon M340 platform as follow:

"Modicon M340 mid-range PAC (Programmable Automation Controller) offers compactness, flexibility, scalability, and robustness for the process industry and a wide range of demanding automation applications" (source)

Schneider Electric Modicon M580 (left) and M340 (right) PLCs.

The M340/580 platforms are based on VxWorks RTOS running on ARM processors and are the replacement for the old Quantum series. They are common in multiple verticals ranging from oil and gas, manufacturing, and commercial building management systems (BMS).

The engineering workstation software used to manage, configure, and program the PLC is called EcoStruxure Control Expert (previously Unity Pro). The C/C++ software suite is IEC programming software for Modicon PACs platforms. It helps engineers with programming, debugging, and operating software for Modicon M340, M580, M580S, Premium, Momentum and Quantum ranges.



Schneider Electric Unity Pro Engineering Workstation

EcoStruxure Control Expert communicates with compatible PLCs over a newly developed extension to the Modbus protocol called UMAS. UMAS uses Modbus reserved function code 90 (0x5A) and has security features built-in. For example, some elevated operations such as writing to physical memory addresses (UMAS function 0x29) require authenticating the session using a session key.

In our research, we were able to find a heap-based buffer overflow that exists in XML decompression function `DecodeTreeBlock` in AT&T Labs Xmill 0.7 (CVE-2022-26507). The Xmil is a serialization process used to convert ASCII based XML data to binary streams to be stored in the Schneider Electric package bundle that is transferred and stored on the M340/580 PLCs.

Using a download procedure, we were able to inject to the PLC weaponized XMILL-compressed data. When the engineer performs an upload procedure, the engineering workstation reads the weaponized XMILL-compressed data from the PLC and decompresses it using the vulnerable decompress function. This process triggers the heap-based buffer overflow that leads to a RCE.

Since being reported by Claroty, Schneider Electric has made security ehnancements to address all Xmill and Xdemill vulnerabilities. Subsequently an advisory has been released on how to fix these vulnerability.

Evil PLC Attack Demo: Schneider Electric Modicon M340/M580 Platform.

GE MarkVIe Platform

General Electric (GE) Gas and Power describes the Mark VIe platform as follow:

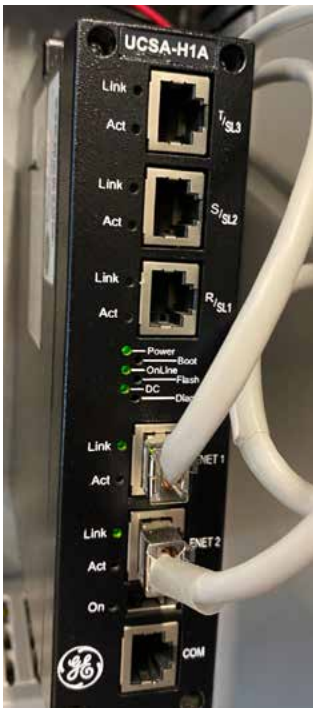
"The Mark VIe distributed control system (DCS) is a flexible platform for multiple applications. It features high-speed, networked I/O for simplex, dual, and triple redundant systems. The Mark VIe also offers SIL3 capable safety system under one common operator and engineering suite of tools. Mix and match redundancy of controllers, networks and IO allows users to meet the specific needs of each application while helping reduce their cost." (source)

The Mark VIe ecosystem is common within the gas and power industries, including thermal, wind, hydro, oil & gas, and nuclear facilities. According to the GE website, there are more than 5,000 thermal turbines, 40,000 wind turbines, and 300 plant DCS installations across the globe.

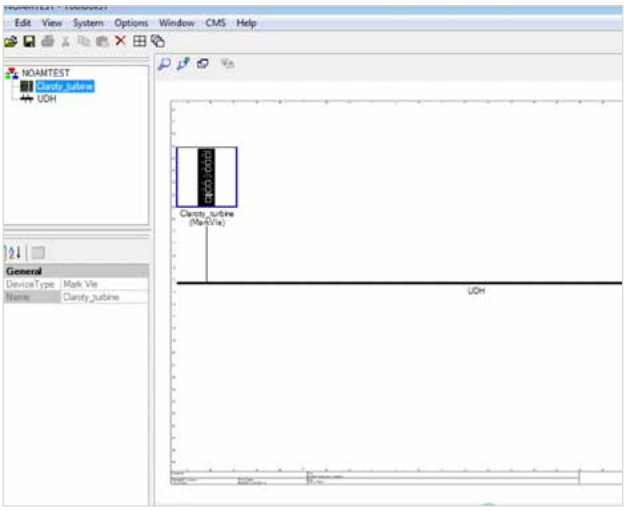
The MarkVIe controller is based on PowerPC (Big Endian) architecture and runs QNX Neutrino as its RTOS firmware. The controller is maintained and controlled by an engineering workstation software named ToolBoxST. The engineering workstation is built in .NET C# and communicates with the controller via the SDI protocol over TCP port 5311.

In our research, we found a ZipSlip vulnerability in ToolBoxST. We were able to weaponize a specific archive on the controller with a special DLL we prepared. Later, when read by the engineering workstation in an upload procedure, we achieve code execution.

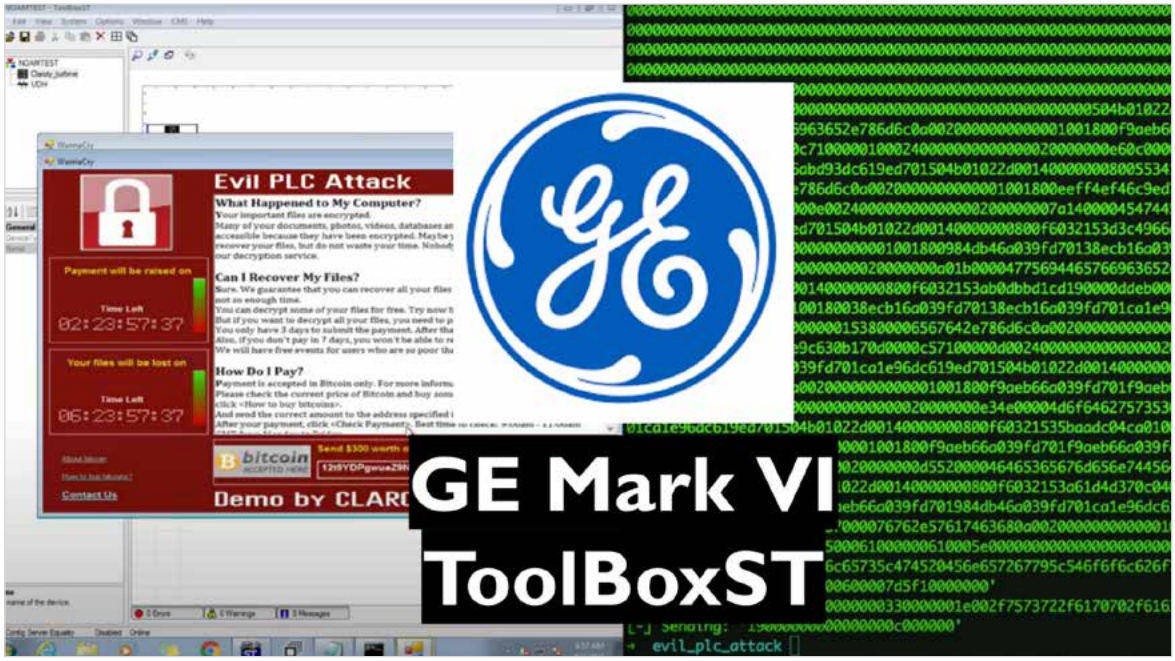
GE fixed the issues we reported in ToolBoxST Version 7.8.0 and ICS-CERT published advisory ICSA-22-025-01 on this topic.



GE Mark VIe Controller (model: IS220UCSAH1A)



GE ToolBoxST Engineering Workstation



Evil PLC Attack Demo: GE MarkVI Platform.

Rockwell Automation Micro Control Systems Platforms

Rockwell Automation describes the Micro800 Series as follows:

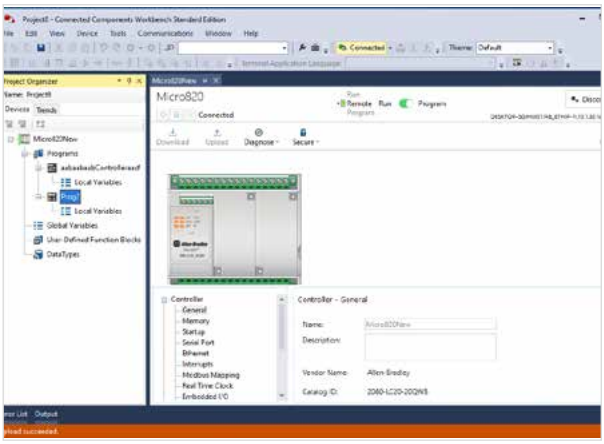
"Micro800 control systems offer a scalable and cost-effective micro control solution for small to large standalone applications. These controllers are optimized for cost and performance for specific applications with customization and flexibility in mind. You can buy only the functionality you need and use plug-in modules to match your application requirements. Plus, boost productivity with reduced design time across the control system using one programming software." (source).

Rockwell Automation Micro Control Systems is a PLC platform series that includes Micro810, Micro820, Micro830, Micro850, and Micro870. Micro800 PLCs are often used in smart manufacturing automation processes.

Rockwell's Connected Components Workbench (CCW) is a .NET C# based engineering workstation program used to configure and communicate with Micro800 PLCs. It can also be used to configure PowerFlex drives, PanelView 800 graphic terminals, and more. CCW enables quick asset discovery by integrating Rockwell's RSLinx Communications Servers used to quickly discover CIP-enabled devices. Therefore, a user can quickly identify relevant devices in the subnet and upload their configurations to view and/or edit them.

Micro800 PLCs, like many other Rockwell devices, communicate with the engineering workstation mainly with the Common Industrial Protocol (CIP) over EtherNet/IP, over TCP port 44818. In addition to the standard classes and services defined as a part of CIP, CCW uses non-standard commands and services. For example, during the upload procedure much of the data is read from the device by using the regular CIP File operations class 0x37 but also using CIP class 0x350 with service 0x4b.

Rockwell Automation Micro820 PLC



Rockwell Automation Connected Components Workbench (CCW) Engineering Workstation

105	10.1.30.14	10.1.30.88	CIP	118	2022-07-06 09:31:48.776610	Success: Class (0x350) - Service (0x4b)
106	10.1.30.88	10.1.30.14	CIP	121	2022-07-06 09:31:48.786586	Class (0x350) - Service (0x4b)
107	10.1.30.14	10.1.30.88	CIP	133	2022-07-06 09:31:48.826622	Success: Class (0x350) - Service (0x4b)
108	10.1.30.88	10.1.30.14	CIP	121	2022-07-06 09:31:48.837713	Class (0x350) - Service (0x4b)
109	10.1.30.14	10.1.30.88	CIP	119	2022-07-06 09:31:48.875779	Success: Class (0x350) - Service (0x4b)
110	10.1.30.88	10.1.30.14	CIP	120	2022-07-06 09:31:48.882157	Class (0x350) - Service (0x4b)
111	10.1.30.14	10.1.30.88	CIP	123	2022-07-06 09:31:48.925728	Success: Class (0x350) - Service (0x4b)
112	10.1.30.88	10.1.30.14	CIP	127	2022-07-06 09:31:48.932811	Class (0x350) - Service (0x4b)
113	10.1.30.14	10.1.30.88	CIP	182	2022-07-06 09:31:48.975857	Success: Class (0x350) - Service (0x4b)
114	10.1.30.88	10.1.30.14	CIP	127	2022-07-06 09:31:48.984862	Class (0x350) - Service (0x4b)
115	10.1.30.14	10.1.30.88	CIP	182	2022-07-06 09:31:49.025372	Success: Class (0x350) - Service (0x4b)

A Rockwell Automation CIP session performing a project upload procedure.

In our research, we found multiple vulnerabilities in CCW, among them, an unsafe .NET deserialization vulnerability. Using the vulnerability we discovered, we managed to weaponize a Micro820 PLC using a download procedure by embedding a specifically crafted serialized .NET object in the downloaded project. The PLC doesn't process the weaponized component and therefore the PLC operation is not compromised. However, after the PLC is weaponized, any user running a vulnerable CCW version who performs an upload procedure from the PLC will read the weaponized serialized object, which will result in arbitrary code execution.

Rockwell Automation assigned CVE-2021-27475 and fixed the issue we reported in version 13.00.00 of Connected Components Workbench. ICS-CERT published advisory ICSA-21-133-01 on this topic in addition to other vulnerabilities we reported in CCW.

Evil PLC Attack Demo: Rockwell Automation Micro Systems Platform.

Emerson PACSystems Platform

Emerson describes PACSystems as follows:

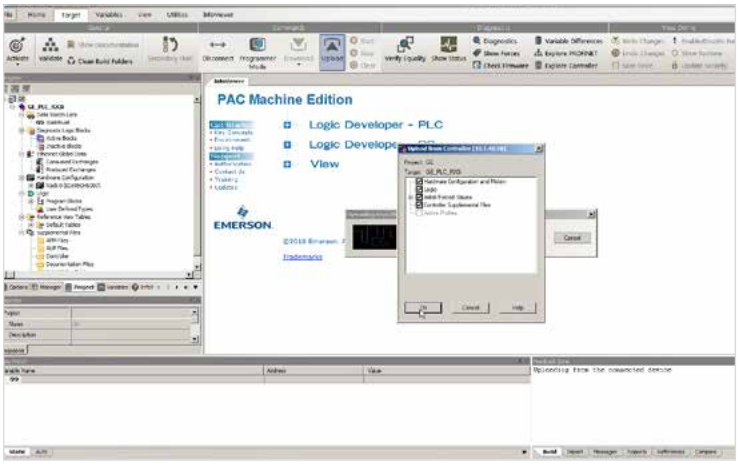
"Emerson's PACSystems™ adds a new dimension to the industrial control and automation landscape, allowing production optimization and monitoring capabilities in the widest range of process and discrete industrial environments. No matter how fast your operation is growing, PACSystems brings control intelligence with a real-world approach closer to the industrial edge." (source).

The PACSystems product family has recently been moved from GE to Emerson and is now developed under the Emerson brand. PACSystems consist of multiple product families including Rx3i, Rx7i, and others. The Rx3i series of PLCs are well known with a long track record in the industry. Rx3i PLCs are widely used in multiple industries ranging from water utilities, pharmaceuticals, and discrete manufacturing. The Rx3i CPUs are based on x86 processors ranging from Intel Celeron/Atom to AMD G-Series CPUs and run the VxWorks RTOS.

The engineering software used to configure PACSystems compatible devices is called PAC Machine Edition (PME). The C/C++ based software provides PACSystems users an integrated environment to configure and maintain control applications. It supports a wide range of devices such as HMIs, PLCs, VFDs, servos, and edge devices related to the PACSystems product family.

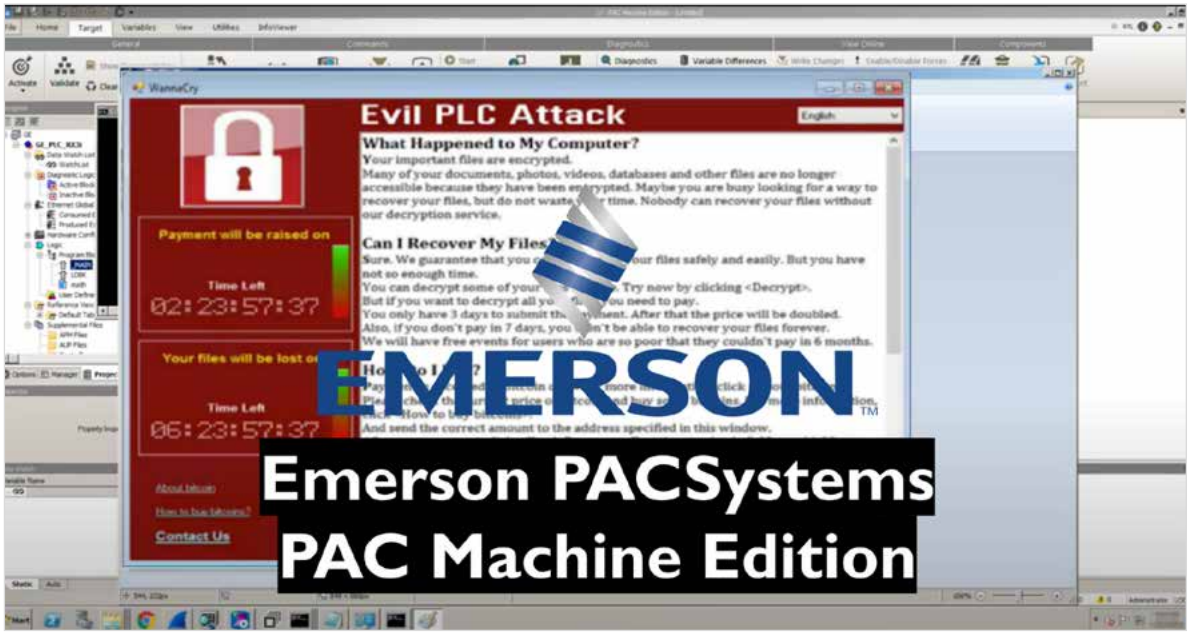
In our research we were able to find a vulnerability in the handling of certain objects transferred from the PLC to the PME software when performing an upload procedure. The specific vulnerable object is a ZIP file which is extracted when the configuration is read. In the extraction process, the software does not protect against a ZIPSlip attack.

Emerson Rx3i PLC



Evil PLC Attack Demo: Emerson PACSystems Platform.

Using the ZIPSlip vulnerability, an attacker can store a file anywhere on the PME machine. To achieve code execution we decided to leverage the vulnerability to overwrite one of the PME's own DLLs that are loaded after the upload procedure ends. Emerson assigned CVE-2022-2788 for this vulnerability and published an advisory on this topic.



Emerson PAC Machine Edition

Xinje XDPPro

Xinje describes the XD Series of PLCs as follows:

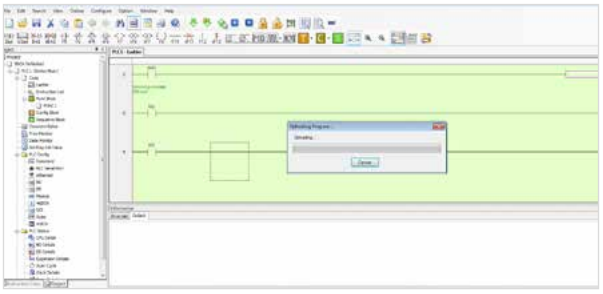
"XD series PLC have diverse CPU units and expansions with powerful functions." (source).

Xinje PLCs are common in the Asia-Pacific region, and are mainly used by companies based in China, usually for process control automation.

The Xinje XD series devices are expandable PLCs manufactured by Xinjie. The ARM-based PLCs are managed using the matching PLC Program Tool. For example a Xinje XD PLC will be managed by the Xinje XD/E Series PLC Program Tool. The .NET C# based PLC Program Tool allows users to easily upload a project by supporting device discovery and project upload.

The engineering workstation communicates with the PLC via Modbus UDP over port 502. The program uses standard Modbus function codes which include read coils and read file record during upload, write file record, and write multiple registers during download.

Xinje XD/E PLC



Xinje XD PLC Programming Tool Engineering Workstation

5	192.168.0.33	192.168.0.111	Modbus/…	54	2019-12-24 11:14:40.978576	Query: Trans: 3401; Unit: 1, Func: 3: Read Holding Registers
6	192.168.0.33	192.168.0.111	Modbus/…	60	2019-12-24 11:14:40.978761	Query: Trans: 3401; Unit: 1, Func: 3: Read Holding Registers
7	192.168.0.111	192.168.0.33	Modbus/…	64	2019-12-24 11:14:41.000994	Response: Trans: 3401; Unit: 1, Func: 3: Read Holding Registers
8	192.168.0.33	192.168.0.111	Modbus/…	54	2019-12-24 11:14:41.001420	Query: Trans: 3402; Unit: 1, Func: 1: Read Coils
9	192.168.0.33	192.168.0.111	Modbus/…	60	2019-12-24 11:14:41.001633	Query: Trans: 3402; Unit: 1, Func: 1: Read Coils
10	192.168.0.111	192.168.0.33	Modbus/…	64	2019-12-24 11:14:41.023925	Response: Trans: 3402; Unit: 1, Func: 1: Read Coils
11	192.168.0.33	192.168.0.111	Modbus/…	54	2019-12-24 11:14:41.024561	Query: Trans: 3403; Unit: 1, Func: 3: Read Holding Registers
12	192.168.0.33	192.168.0.111	Modbus/…	60	2019-12-24 11:14:41.024680	Query: Trans: 3403; Unit: 1, Func: 3: Read Holding Registers
13	192.168.0.111	192.168.0.33	Modbus/…	64	2019-12-24 11:14:41.047113	Response: Trans: 3403; Unit: 1, Func: 3: Read Holding Registers
14	192.168.0.33	192.168.0.111	Modbus/…	54	2019-12-24 11:14:41.047661	Query: Trans: 3404; Unit: 1, Func: 1: Read Coils
15	192.168.0.33	192.168.0.111	Modbus/…	60	2019-12-24 11:14:41.047824	Query: Trans: 3404; Unit: 1, Func: 1: Read Coils
16	192.168.0.111	192.168.0.33	Modbus/…	64	2019-12-24 11:14:41.069077	Response: Trans: 3404; Unit: 1, Func: 1: Read Coils
17	192.168.0.33	192.168.0.111	Modbus/…	54	2019-12-24 11:14:41.113550	Query: Trans: 3405; Unit: 1, Func: 1: Read Coils
18	192.168.0.33	192.168.0.111	Modbus/…	60	2019-12-24 11:14:41.113727	Query: Trans: 3405; Unit: 1, Func: 1: Read Coils
19	192.168.0.111	192.168.0.33	Modbus/…	64	2019-12-24 11:14:41.135205	Response: Trans: 3405; Unit: 1, Func: 1: Read Coils

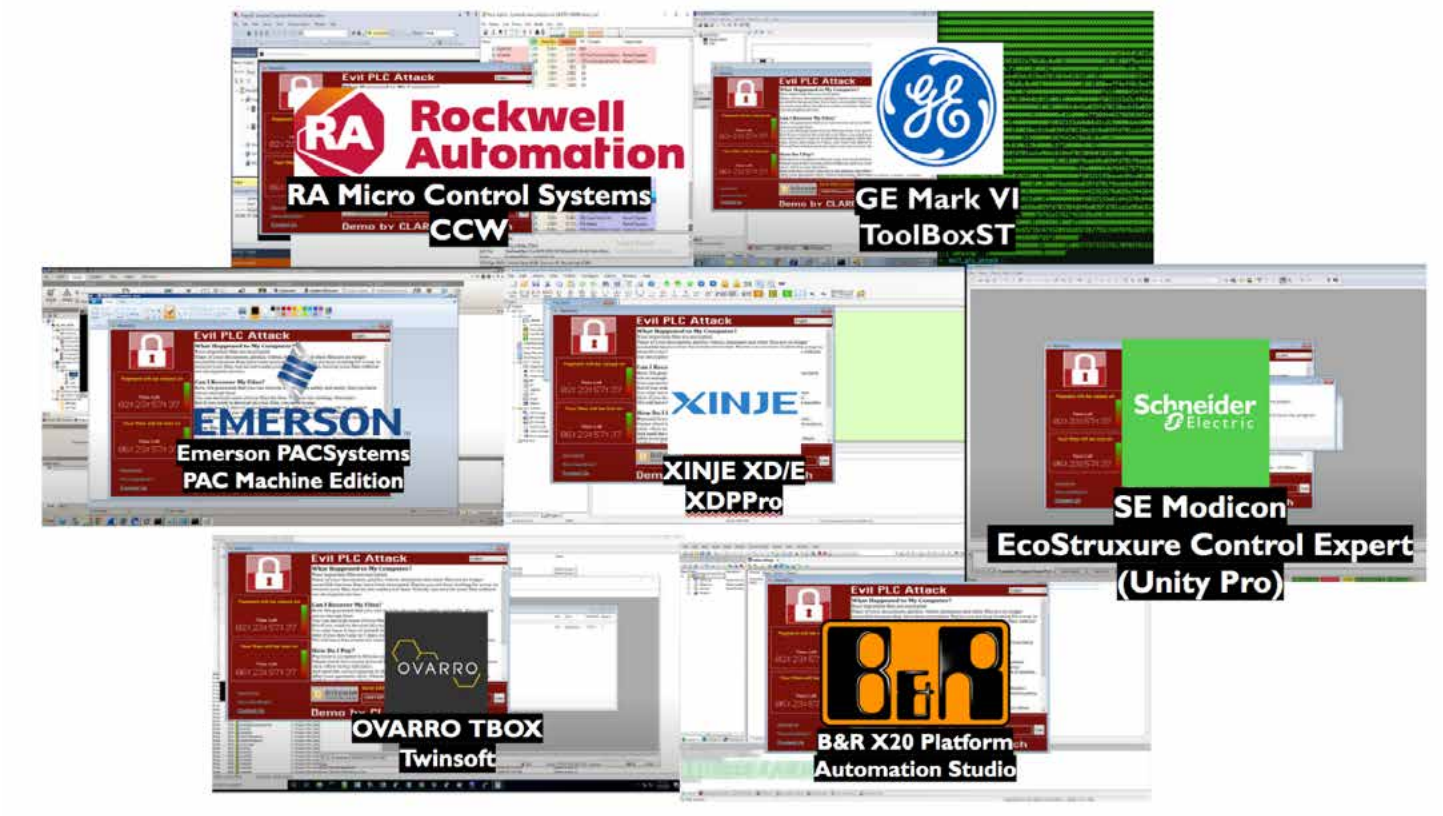
A Xinjie Modbus upload procedure.

During our research, we managed to take advantage of the static project file vulnerabilities we previously discovered (From Project File to Code Execution- Exploiting Vulnerabilities in Xinje PLC Program Tool) to achieve code execution via project upload. This time, instead of weaponizing an offline project file, we used download procedures to weaponize the PLC with a ZIPSlip-vulnerable archive, and then achieve code execution when the project is uploaded to an engineering workstation.

Xinje XD Programing Tool version 3.5.1 is affected, and likely other versions. We began our disclosure efforts in August 2020. More than a year later, Xinje acknowledged our disclosure in September 2021. However, Xinje at that time refused to cooperate with us and asked us to stop communicating with them. Eventually we assigned CVE-2021-34605 and CVE-2021-34606 and published a blog post discussing the vulnerabilities and our disclosure efforts.

Evil PLC Attack Demo: Xinje XD Platform

Summary



Vendor	Platform	Arch	Tested Model (CPU Module)	RTOS Firmware	Engineering Workstation	Protocol	Root Cause	CVE
OVARRO	TBOX	ARM	TBOX LT2-530	Linux	TwinSoft	Custom Modbus (Port 502/ TCP)	ZipSlip / Path Traversal	Advisory CVE-2021-22650
B&R (ABB)	X20 System	ARM/x86	X20CP1585	VxWorks	Automation Studio	ANSL (Port 11169/ TCP) INA2000 (Port 11159/ TCP or UDP)	ZipSlip / Path Traversal	Advisory CVE-2021-22289
Schneider Electric	Modicon (M340, M580)	ARM	M340, M580	VxWorks	EcoStruxure Control Expert (Unity Pro)	Modbus/ UMAS (Port 502/ TCP)	Memory Corruption (Heap overflow)	Advisory CVE-2022-26507
General Electric (GE)	MarkVIe	PPC (BE)	Mark VIe IS220UCSAH1A	QNX Neutrino	ToolBoxST	SDI (Port 5311/ TCP)	ZipSlip / Path Traversal	Advisory CVE-2021-44477 CVE-2018-16202
Rockwell Automation	Micro Control Systems	ColdFire	Micro820	ThreadX	Connected Components Workbench (CCW)	CIP (Port 44818/ TCP)	Unsafe Deserialization	Advisory CVE-2021-27475 CVE-2021-27471 CVE-2021-27473
Emerson	PACSystems	x86	Rx3i	VxWorks	PAC Machine Edition	SRTP (Port 18245/ TCP)	ZipSlip / Path Traversal	Advisory CVE-2022-2788
Xinje	XDPPro	ARM	XD/E PLC	VxWorks	XD PLC Program Tool	Modbus UDP (Port 502/UDP)	ZipSlip / Path Traversal	Advisory CVE-2021-34605 CVE-2021-34606

SHOWCASES

Let's demonstrate three examples of our proof-of-concept exploits in action.

Example No. 1: GE Mark VIe

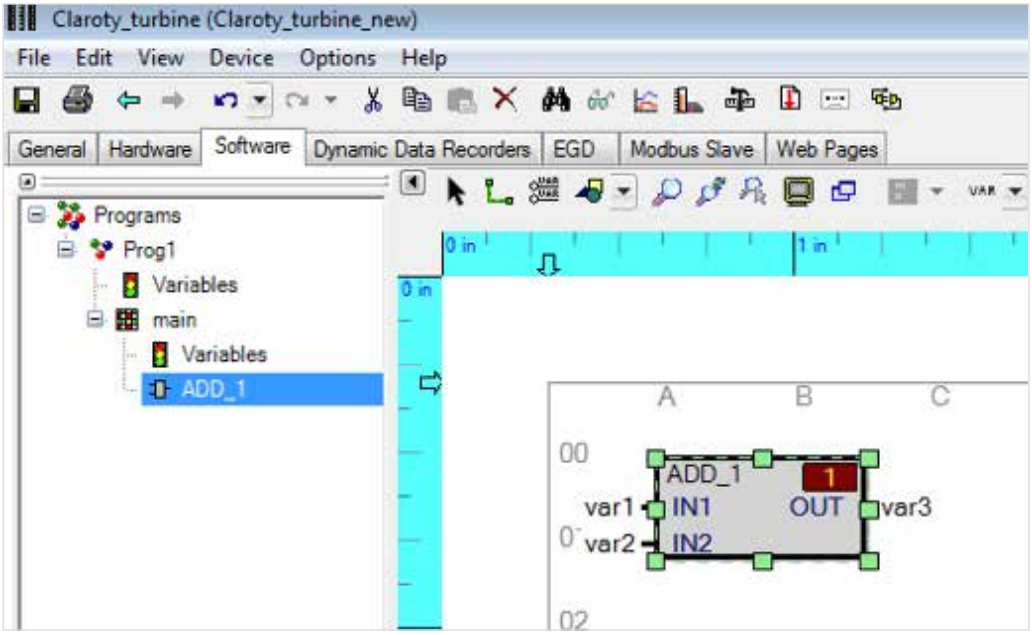
Studying the Platform

GE MarkVIe is a flexible control system used in a variety of applications. MarkVIe and related controls use ToolboxST as a software platform for programming, configuring I/O, trending data, and analyzing diagnostics. At the controller level and at the facility level, it allows you to effectively manage equipment assets with reliable, time-synchronized data.

The ToolBoxST engineering workstation and the MarkVIe controller communicate via the SDI protocol over TCP port 5311. SDI is GE proprietary protocol for system configuration, maintenance, and real time data and alarm management. The protocol is fairly simple and consists of a 12-byte header and payload data. When a client connects to a Mark VIe controller they are faced with a challenge-response mechanism (SDI Command 0x2b). Since the algorithm for the challenge-response is fixed, we were able to implement our own client.

SDI Packet

Using the engineering workstation, ToolBoxST, engineers can develop new programs. For example, we created a new main program with a single function block—ADD—which takes two REAL tag variables as inputs, adds them together and writes back to a third output variable.



ToolboxST main program with a single ADD function block.

We then compiled the program and performed a download procedure in which both the compiled code and the textual code are transferred into the controller. In this case, the MarkVIe controller runs a UNIX based RTOS operating system called QNX. Owned by BlackBerry, QNX is a commercial Unix-like real-time operating system with a microkernel, aimed primarily at the embedded systems market. Therefore, it was quite easy to explore the file-system and see what has changed under `/usr/app/active` or `/usr/app/A`.

For our code manipulation research, there were three important files:

- `device.zip`
  - An archive with XML files that describe the current running project. This also includes a high-level representation (XML) of the current loaded program.
- `device.zip.crc`
  - CRC check for device.zip archive.
- `program{NAME}.pcode`
  - pcode property binary format to store the program bytecode. This bytecode will be translated and executed by the controller.

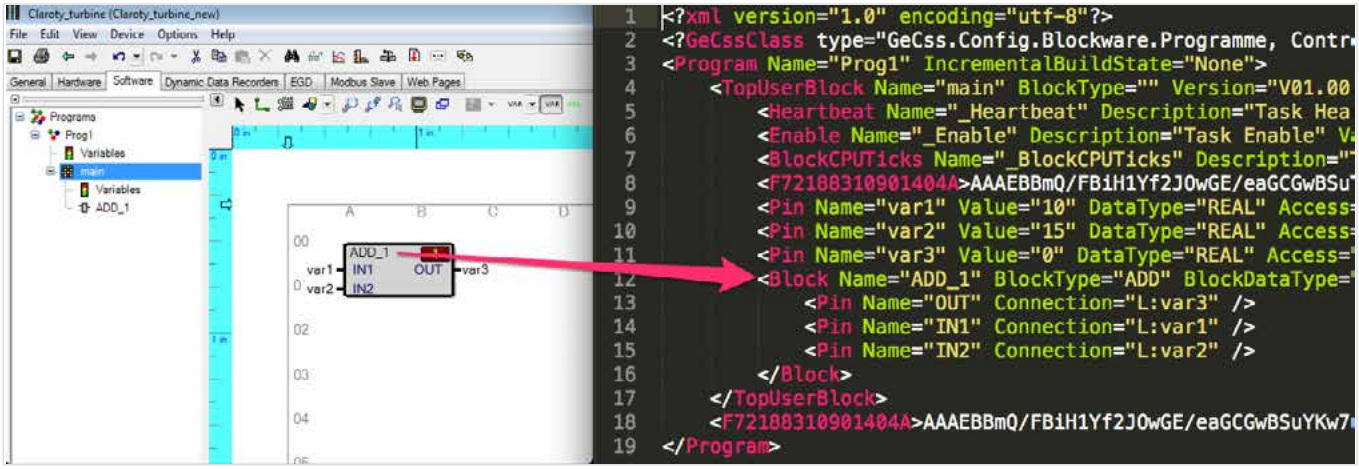
Filename	Filesize	Filetype
..		
alarms.pcode	101	pcode-file
commandeventlog.pcode	12157	pcode-file
ddr.pcode	82	pcode-file
device.zip	23644	ZIP archive
device.zip.crc	4	crc-file
dhcpd.conf	337	conf-file
egd-gesymboltable.cxml	6575	cxml-file
egd-guidevice.xml	1395	XML
egd-outputs.xml	3847	XML
egd-profile.xml	976	XML
egd-symboltable.xml	12530	XML
hart.xml	108	XML
hosts	180	File
ntp.conf	115	conf-file
ntpionet.conf	115	conf-file
nvram.pcode	69	pcode-file
platform.pcode	67	pcode-file
profibus.xml	112	XML
programprog1.pcode	556	pcode-file
redsymboltable.pcode	57	pcode-file
requisitioninfo.xml	314	XML
scheduler.pcode	499	pcode-file
ssif.pcode	63	pcode-file
sysinit.pcode	125	pcode-file
system.pcode	545	pcode-file
vardef.pcode	512	pcode-file

MarkVIe controller file system - `/usr/app/A` after a successful download procedure

For now, we care only about the textual and binary code of the current running program:

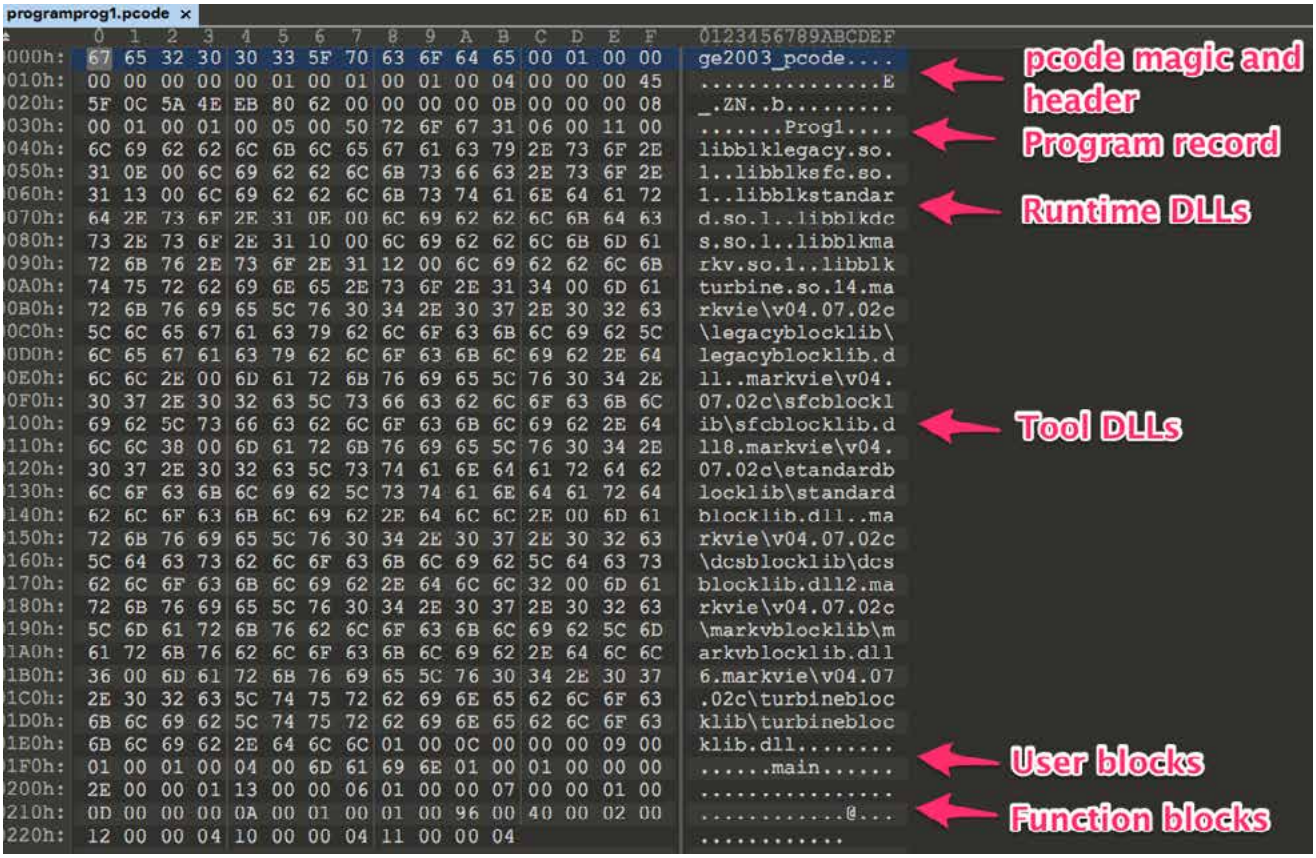
- Binary code: `programprog1.pcode`
  - Used by the controller, MarkVIe, to execute the logic code (bytecode) that was programmed and compiled on the engineering workstation machine. The controller does not use the XML files because they are being used for display purposes only.
- Textual code: `device.zip` → `_Prog1.xml`
  - Used by the engineering workstation, ToolboxST, to show the engineer what code is running on the controller. The engineering workstation is not using the binary code.

The textual code, which is XML-based, is easy to understand. We can see a clear description of the function block including its in and out pins.



XML files generated based on the engineer's code logic (function blocks).

The pcode file that contains the bytecode is a proprietary binary format that represents the .NET assembly objects observed in the XML. In other words, the output of the compilation process is an intermediate language bytecode that will eventually get executed on the controller. By reverse engineering the `GeCss.Util.BinaryPcodeReader` class and the relevant `WritePcode` functions (`Blockware`), we were able to create our own compiler/decompiler to construct our own pcode files with program records.



`programprog1.pcode` file content.

Finding the Vulnerable Path: (ICSA-19-281-02)

To recap, ToolBoxST is used to configure and maintain MarkVIe and other controllers. One of the options it allows is an upload procedure that will retrieve information from the controller and create a new project based on the retrieved information. We discovered that one of the obtained files is a ZIP file named `device.zip` (located at `/usr/app/active/device.zip` on the device) which usually contains XML files with information about the project such as the name of the project, programs, tags, etc.

After ToolBoxST obtains this file from the controller via a SDI protocol upload procedure (SDI command 0x25), it extracts all the files from the zip file without any validation or sanitizing on the file names. We found that ToolBoxST is vulnerable to a ZipSlip attack because an outdated version of Ionic .NET ZIP library is used to handle this process.

Zip files include a directory tree, usually used to create sub-directories within the extracted zip directory. However, the path of a sub-directory can also be "..". Such paths are not sanitized by the zip implementation used in ToolBoxST. As a result, an attacker can chain several such paths in a row to get to the drive root (e.g. "`C:\`"). Then, by adding the path of the expected folder the files from the project file will be extracted to arbitrary locations on the machine's file system. This is also known as a ZipSlip attack.

- **Example:** `..\..\..\..\..\..\..\..\..\..\..\..\..\..\..\..\temp\poc.txt`

The code flow that leads to this vulnerability is as follows. First, below we have the `DoUpload` function with the vulnerable code flow marked:

```
UploadFromControllerProgress
89          internal bool DoUpload()
90          {
91              this.m_UploadedFileName = FileUtils.GetTempFileName();
92              File.Delete(this.m_UploadedFileName);
93              try
94              {
95                  this.Descriptor.Name = Path.GetFileNameWithoutExtension(this.m_UploadedFileName);
96                  this.m_TempDevice = (IMarkVIeSDevice)this.Sys.AddDevice(this.Descriptor);
97                  this.m_TempDevice.DeviceConnection.IPAddress = this.Wizard.UploadFromIpAddress;
98                  ICommandReturn ret = this.m_TempDevice.DeviceConnection.Cmd.ReadFile(this.m_PathToDeviceZip, this.m_UploadedFileName,
                      this.m_UploadProgressBar);
99                  if (!ret.OK)
100                 {
101                     if (ret.Ex != null)
102                     {
103                         ExceptionMessageBox.Show(this, ret.Ex);
104                     }
105                     else
106                     {
107                         MessageBox.Show(this, ret.Message, Application.ProductName, MessageBoxButtons.OK, MessageBoxIcon.Hand);
108                     }
109                     this.Sys.DeleteDevice(this.m_TempDevice.Name);
110                     this.Sys.PurgeDeleteFolder(this.m_TempDevice.Name);
111                     this.m_TempDevice = null;
112                     return false;
113                 }
114                 string tempName = FileUtils.GetTempDirectory();
115                 FileUtils.DeleteReadonlyDirectory(tempName);
116                 FileUtils.RestoreFolderContents(this.m_UploadedFileName, tempName, false, new FileUtils.RestoreFolderContentIncludeFileHandler
                      (this.OnCheckForDevZipName), false);
117                 XmlDocument xmlDoc = new XmlDocument();
118                 xmlDoc.Load(Path.Combine(tempName, "Device.xml"));
119                 XmlElement root = xmlDoc.DocumentElement;
120                 XmlAttribute attrib = root.Attributes["Name"];
```

When the upload finishes, the `OnFinish` function is called to extract the `device.zip` file with the project information, below.

```
public override bool OnFinish()
{
    try
    {
        using (new WaitCursor(this.Wizard))
        {
            IComponentDescriptor uploadDescriptor = this.Descriptor;
            uploadDescriptor.Name = this.m_UploadedDeviceName;
            this.m_UploadedDevice = (IMarkVIeSDevice)this.Sys.AddDevice(this.Descriptor);
            this.m_UploadedDevice.EgdConfig.PublishEgdToEgdCfgServerSkip = true;
            this.m_UploadedDevice.Save();
            this.m_UploadedDevice.EgdConfig.PublishEgdToEgdCfgServerSkip = false;
            this.m_UploadedDevice.SharedIONetUploadFromControllerInProgress = true;
            this.ExtractDeviceFromZipfileIntoDeviceDirectory(this.m_UploadedFileName);
            this.Wizard.UploadedFromDevice = true;
            this.Wizard.Device = this.m_UploadedDevice;
            this.Wizard.Workspace.SetModified(true);
        }
    }
```

Finally, the code that extracts the content of the ZIP file. As you can see the .NET Ionic library is used to extract the content of the ZIP, below.

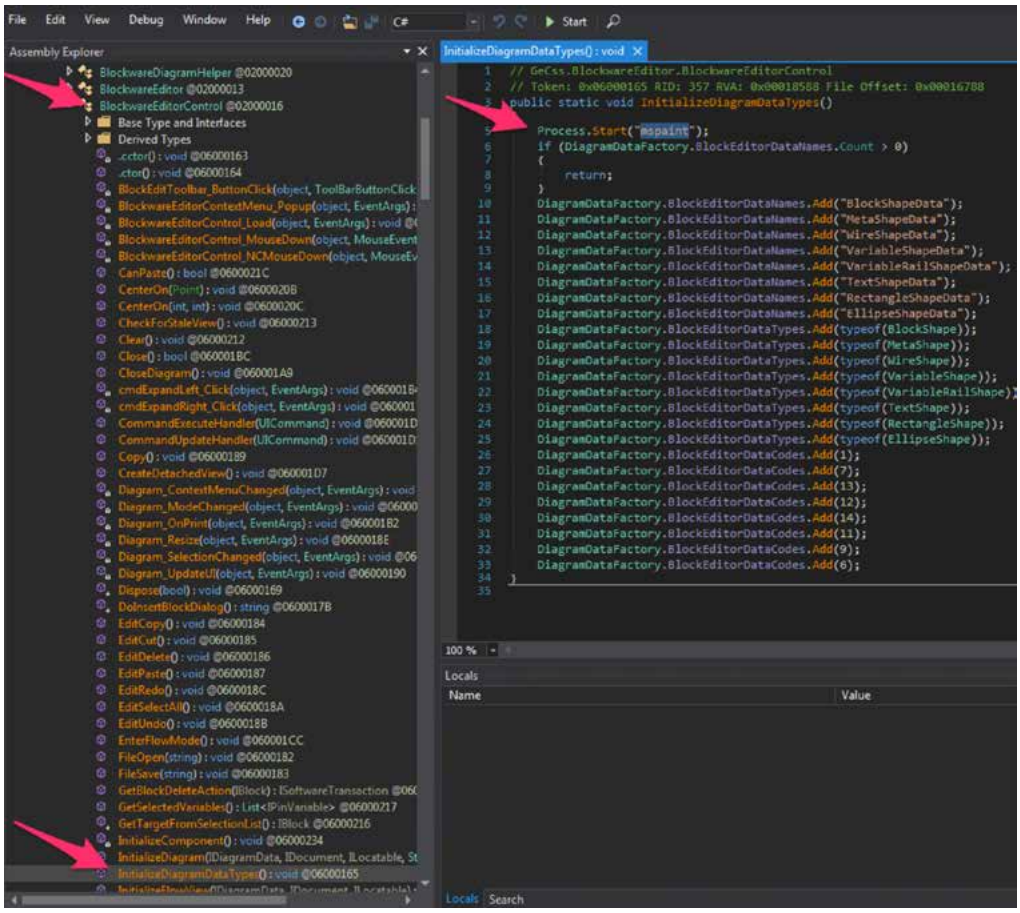
```
public static Exception[] RestoreFolderContents(string srcFilePath, string destDirectory, bool ignoreExceptions,
  FileUtils.RestoreFolderContentIncludeFileHandler fileIncludeHandler, FileUtils.RestoreFolderContentIncludeFileHandle
  postFileExtractHandler, bool skipZeroSizeFiles)
{
    if (fileIncludeHandler == null)
    {
        return FileUtils.RestoreFolderContents(srcFilePath, destDirectory, ignoreExceptions, skipZeroSizeFiles);
    }
    ExtractExistingFileAction extAction = ExtractExistingFileAction.OverwriteSilently;
    List<Exception> exceptions = new List<Exception>();
    Exception[] result;
    using (Ionic.Zip.ZipFile zipFile = new Ionic.Zip.ZipFile(srcFilePath))
    {
        foreach (Ionic.Zip.ZipEntry zipEntry in zipFile)
        {
            try
            {
                if (!skipZeroSizeFiles || zipEntry.UncompressedSize != 0L)
                {
                    string targetDir = destDirectory;
                    if (fileIncludeHandler(zipEntry.FileName, zipEntry.UncompressedSize, ref targetDir))
                    {
                        zipEntry.Extract(targetDir, extAction);
                        if (postFileExtractHandler != null)
                        {
                            targetDir = destDirectory;
                            postFileExtractHandler(zipEntry.FileName, zipEntry.UncompressedSize, ref targetDir);
```

The vulnerable extract code in the engineering workstation, ToolBoxST.

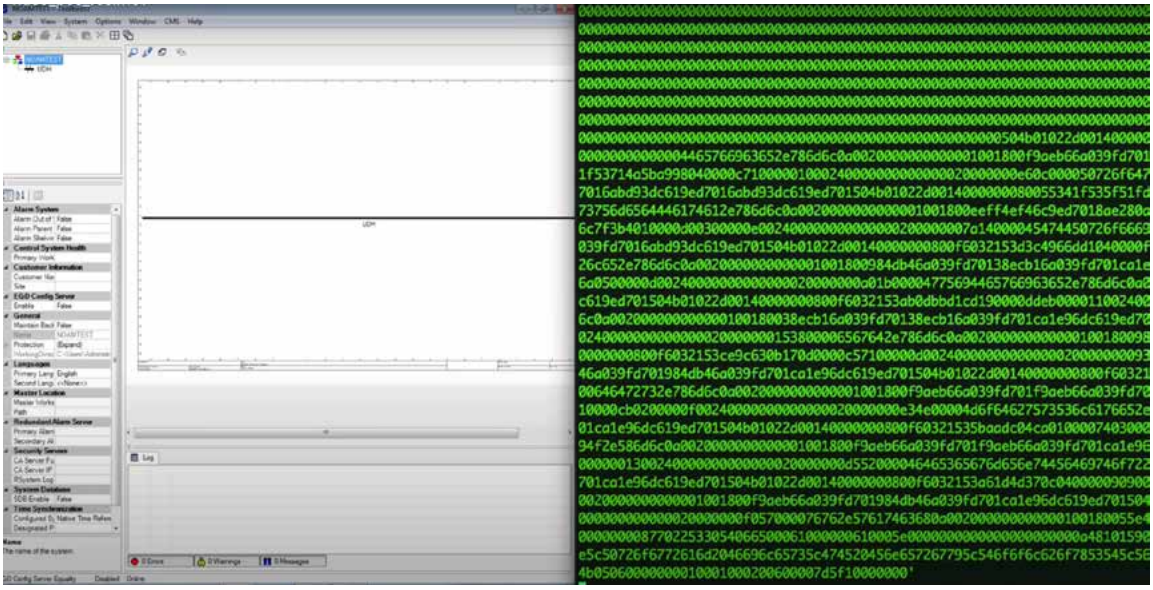
To get immediate code execution, we need to find and override a DLL that is loaded after the upload procedure. We chose `BlockwareEditor.dll` because we noticed that the `C:\Program Files\GE Energy\ToolboxST\V04.07.05C\BlockwareEditor.dll` DLL is loaded right after the upload procedure ends. We patched it so when this DLL gets called, our injected code will get executed.

Demo

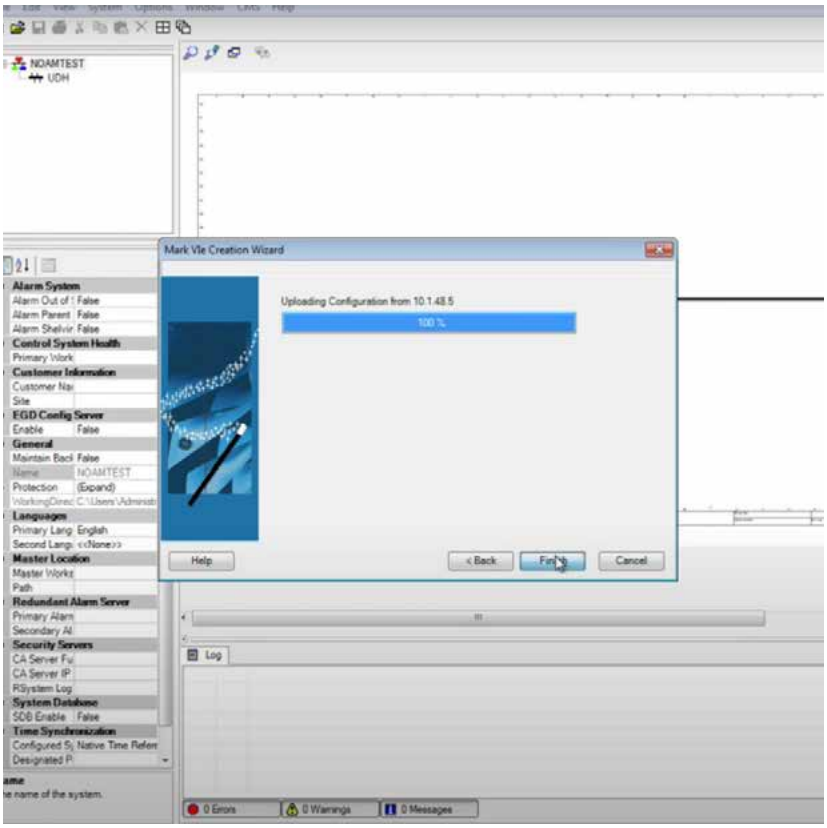
We first prepared the payload: Patching CIL code in `BlockwareEditor.dll` is a good target because the DLL is loaded immediately after an upload procedure. We injected our CIL code to one of the functions we knew would get executed, below.



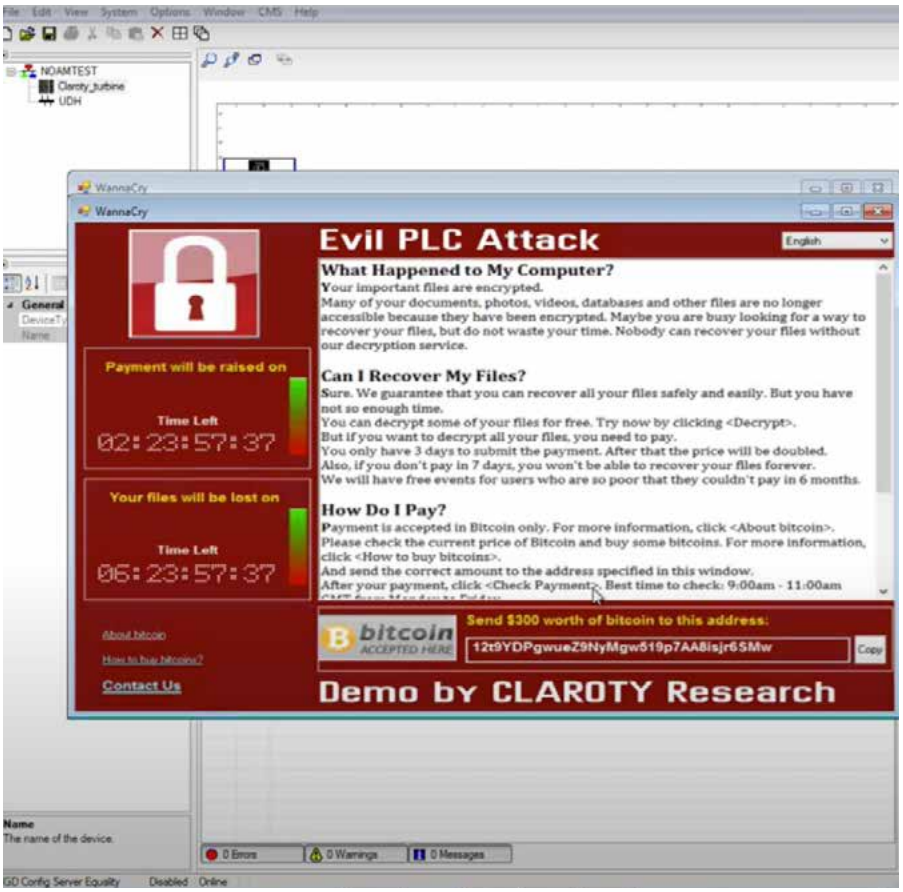
Next, using SDI command 0x25 we performed an upload procedure and retrieved the `/usr/app/active/device.zip` file from the controller. We patched the archive file, and added our malicious DLL with a special traversal path name. We fixed some CRCs and transferred the archive back to the controller using SDI commands 0x18, 0x19 to start the download procedure and command 0x0d to write the malicious archive. Now the controller is weaponized, below.



We “waited” (in our lab) for an engineer to perform an upload procedure through ToolBoxST.
When the upload is complete, ToolBoxST parses the infected archive file and our embedded file triggers the vulnerability.



The exploit was successful and the engineer's machine is now infected with our fake ransomware.



Example No. 2: Schneider Electric M580

Studying the Platform

Schneider Electric's M580 is a popular series of PLCs that are used in many different industrial sectors from water and wastewater facilities, oil & gas, food and beverage production, and even mineral and metal refinement.

In order to interact with M580 PLCs, engineers use Schneider Electric EcoStruxure Control Expert engineering software, formerly called Unity Pro. Through this program, engineers can interact with M580 PLCs, change their configurations, settings and even program the PLC itself.

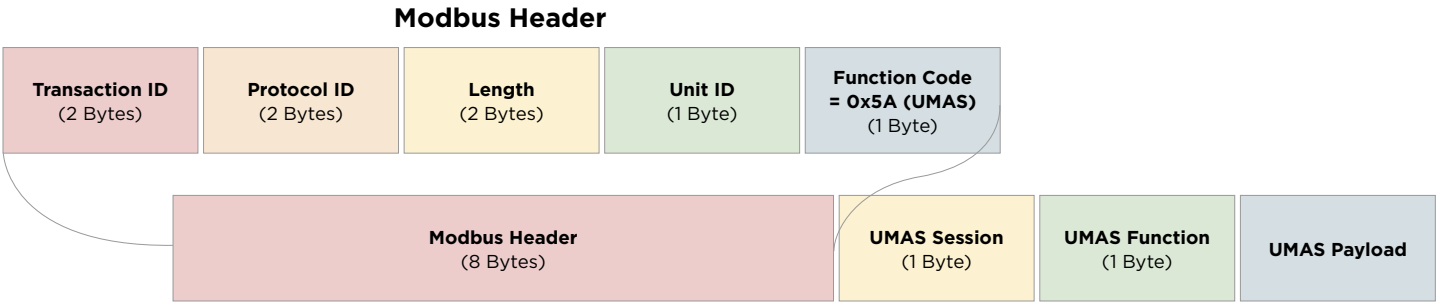
When analyzing how the engineering software communicates with M580 PLCs, we discovered that Schneider Electric chose to use a customized proprietary protocol that extends the Modbus protocol. Schneider's implementation uses a custom Modbus function code, 0x5A, followed by an internal function code used to communicate with supported devices. This new protocol is called UMAS

UMAS

In order to add capabilities and features, UMAS uses an internal function code following the 0x5A Modbus function code, to distinguish between different UMAS operations. Using this second function code, a UMAS client can perform a range of different requests and operations, ranging from performing queries about the device itself (e.g. 0x4 Get PLC Status), reading/writing data to the device (e.g. 0x20 Read Memory Block, 0x21 Write Memory Block), and more. In addition, UMAS also offers security mechanisms such as session-based authentication (a feature missing from Modbus), requiring clients to authenticate using their application password to start a session before performing certain operations.

For example, in order to write to an M580 PLC's physical memory, a client must use the UMAS function code 0x29; however this function code requires elevation, requiring the client to supply a valid session ID. The UMAS client must first acquire a session ID through the use of two function codes—UMAS function 0x6e for supplying the user password hash and UMAS 0x10 to reserve a session key. Only after retrieving the key will they be able to perform the request.

UMAS Packet



Schneider Electric UMAS Packet Structure.

Project Structure

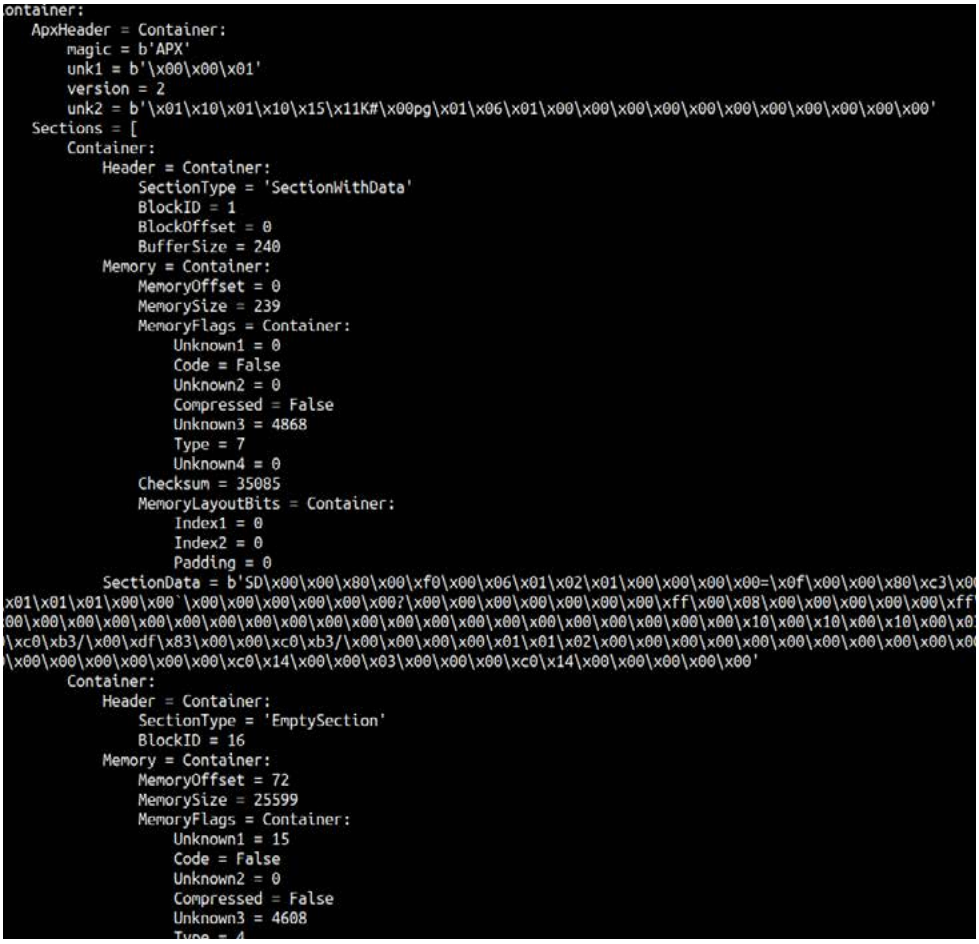
When looking at the way projects are built in Schneider Electric EcoStruxure Control Expert engineering software, we can see that a project is actually compressed data with the suffix of `*.stu` or `*.sta`. When decompressing the project file, we learn that it contains two internal file types:

- **Metadata Files (*.ctx):** Files containing metadata about the project
- **Project Files (*.db, *.apx):** Files containing the actual configurations, settings, and logic code.

```
[root@localhost tmp]# 7z l project.stu

7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,6 CPUs Intel

Scanning the drive for archives:
1 file, 1020978 bytes (998 KiB)

Listing archive: project.stu

--
Path = project.stu
Type = zip
Physical Size = 1020978

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2015-10-15 14:26:14 ....A       180224        39421  ASPROG.db
2015-10-15 14:26:14 ....A       131072        23247  ASROOT.db
2015-10-15 14:26:16 ....A       110592        19615  ATM.DB
2015-10-15 14:20:06 D....            0            0  BinAppli
2015-10-15 14:20:12 ....A       173950        39536  BinAppli/Station.apx
2015-10-15 14:26:16 ....A       135168        25782  CfgMidBase.db
2015-10-15 14:26:16 ....A       315392        41966  CfgPre.db
2015-10-15 14:26:16 ....A       253952        46147  ConfProject.db
2015-10-15 14:26:16 ....A       122880        25975  DCM.DB
2015-10-15 14:26:16 ....A       110592        20251  FDTDTMMgr.db
2015-10-15 14:26:16 ....A          508          211  FLAGS.CTX
2015-10-15 14:26:16 ....A       114688        23828  IOScreen.db
2015-10-15 14:26:16 ....A       180224        34190  MGRCOMBase.db
2015-10-15 14:26:16 ....A        81920        18039  MotionManager.ODB
2015-10-15 14:26:12 ....A         1568          328  OMCS.CTX
2015-10-15 14:26:16 ....A       192512        41007  PathBase.db
2015-10-15 14:26:16 ....A         9482         1451  Project_Settings.xso
2015-10-15 14:26:14 ....A      1048576       335368  SLM.db
2015-10-15 14:26:12 ....A         1296          526  STATION.CTX
2015-10-15 14:26:14 ....A       376832        87821  TypeManager.ODB
2015-10-15 14:26:14 ....A       630784        76790  VariableManager.db
2015-10-15 14:26:16 ....A       610304       117265  XRefManager.db
------------------- ----- ------------ ------------  ------------------------
2015-10-15 14:26:16           4782516      1018764  21 files, 1 folders
```

Contents of the compressed project file in Schneider Electric EcoStruxure Control Expert.

When looking for possible attack surface, we started looking at the `*.apx` files. We learned that those files are actually binary files composed of many different subsections, each with a header containing section type, ID, offset and size, and the section data itself. Different sections can contain different types of data such as code sections including compressed textcode and compressed bytecode of the PLC logic.



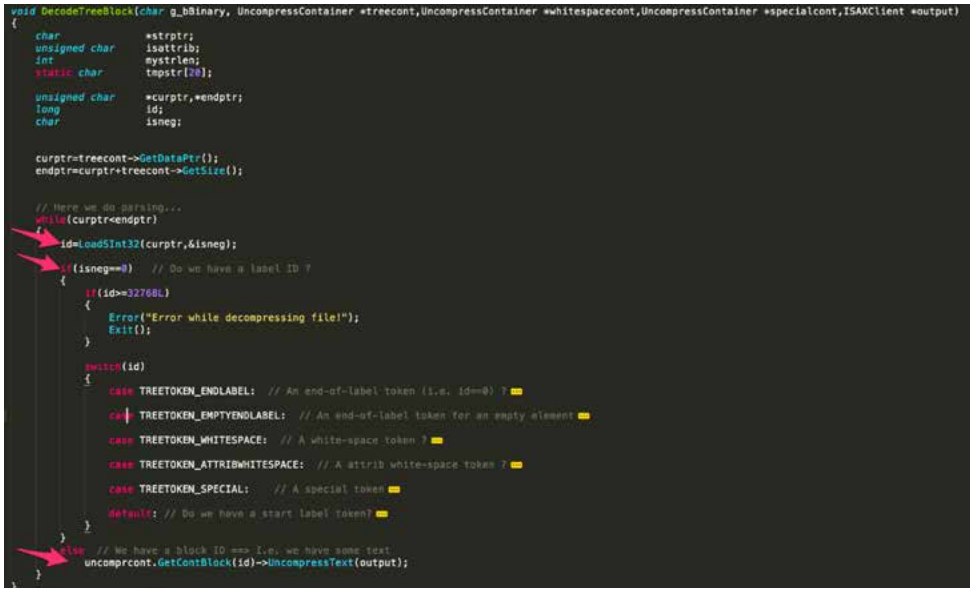
*A parsed *.apx file, showcasing its internal sections' header and data.*

Finding The Vulnerability: Out-Of-Bounds Access

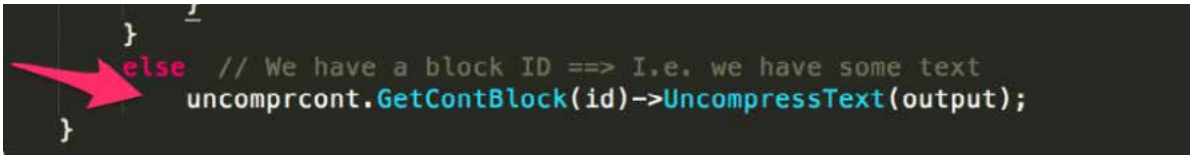
When an engineer performs an upload procedure, the engineering workstation pulls the project files saved on the PLC. After decompressing the files, the program continues to parse the project files, some of which are `*.apx` files which are composed of many different section types. One section type in particular caught our interest: compressed XMLs.

Whenever the engineering workstation encounters this section type, it needs to decompress and parse it as XML, and to do so Schneider Electric chose to use an open source tool called Xmill and Xdemill, which handles the XML compression and decompression for the program. The Xmill and Xdemill tools offer efficient methods of compression for XML, transforming XML-based data from a textual representation to compressed binary format, offering both great compression ratio and speed. The Xmill library is based on research and an academic paper written in collaboration with the University of Pennsylvania. Because the Xmill project is open source, we determined that we should start our research looking into Xmill and Xdemill.

When looking for the main parsing logic of those tools, we reached the `DecodeTreeBlock` function in the `Decode.cpp` file. This function parses the given compressed XML until it reaches the end of the file it received, below.



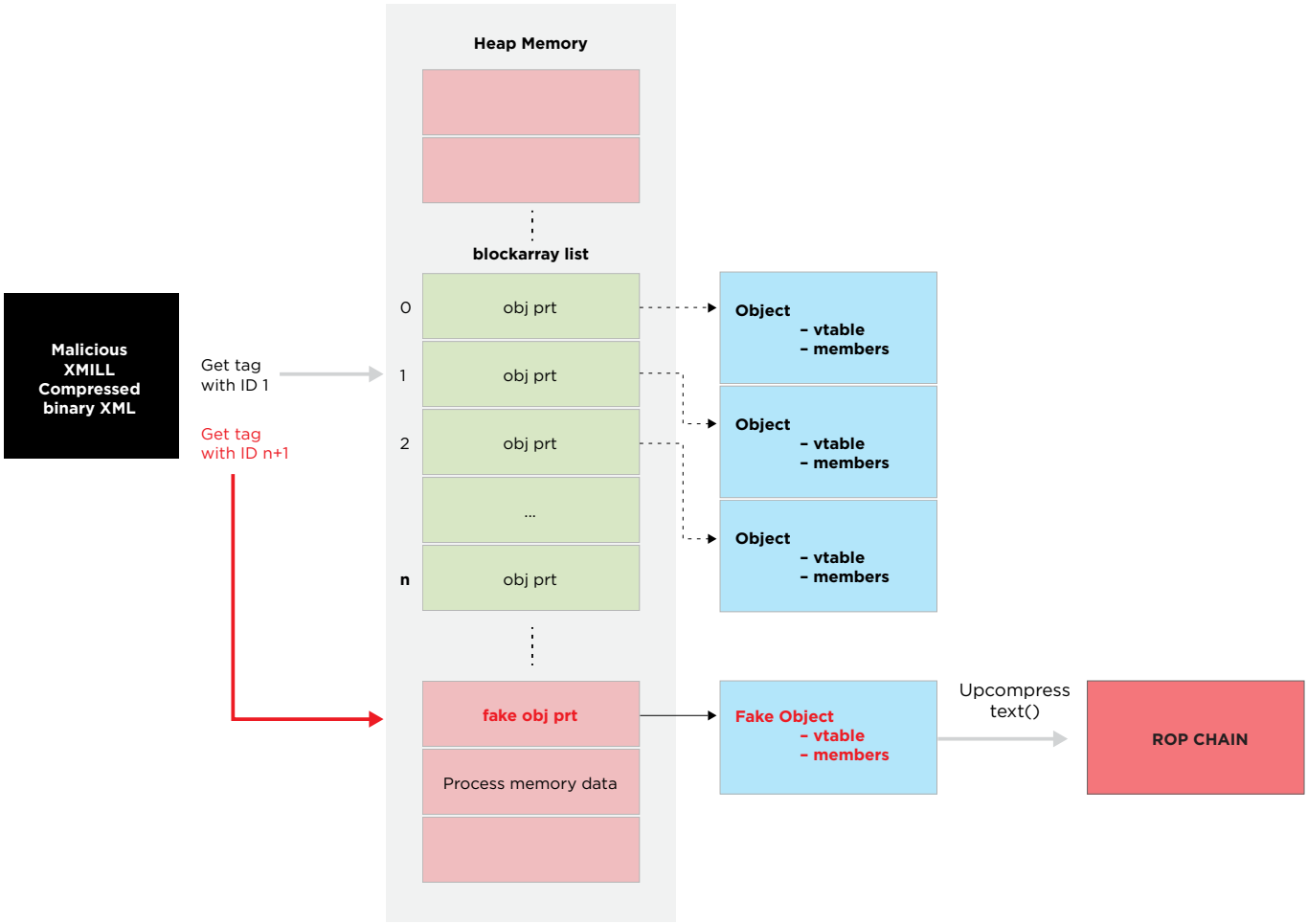
In order to parse each XML tag, the program reads an ID from the given compressed XML stream using the `LoadSInt32` function, and then continues parsing the data according to the ID it parsed. From the parsed ID, the program extracts another variable named `is_neg`, which indicates whether the tag the program will parse is an XML label or inner-text, which leads the program to different parsing code paths. If this flag is raised, the program executes the function, `UncompressText`.



Then, the program will use the ID it parsed in order to retrieve the correct object pointer from a list of object pointers (called `blockarray`), using the ID as the object's offset in the array. After retrieving the object, the program will use it to invoke the `UncompressText` function. As it turns out, Xdemill fails to check that the ID it reads is actually within the `blockarray` bounds.

Abusing this, and the fact that the ID is taken from the compressed XML blob which is under our control, we were able to supply an ID that is longer than the length of the `blockarray` list. Later on, the program will use this ID as an offset in the `blockarray` list, however since we supplied a higher number than the list's length, the program will access memory that is not part of the list, instead accessing "random" memory. This gave us an out-of-bounds access vulnerability in the Xdemill program, and we moved forward onto creating an exploit for this vulnerability. We started to analyze the program's memory structure, looking for a way to control the memory that the program will access whenever we supply a big ID value.

By controlling this object's pointer, we will be able to craft a fabricated object that will execute our code whenever the program tries to execute the `uncompressText` function using this object. Luckily enough, when analyzing the memory near the `blockarray` list, we found that it contains pointers to the uncompressed XML data that is under our control as we encode the compressed XML data and perform download procedure to set it on the PLC.



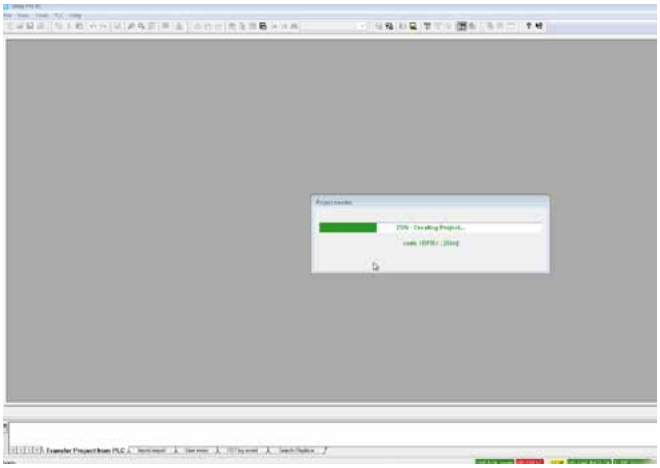
CVE-2022-26507 high level exploit chain diagram.

This meant that we had a pointer that pointed to data that is fully under our control. Using this, we could create a fake object in memory with a fake function VTable from which the program will try to invoke a function. Using this primitive, we constructed such an object, so when the program tries to access it, a small ROP chain will be executed, resulting in remote code execution.

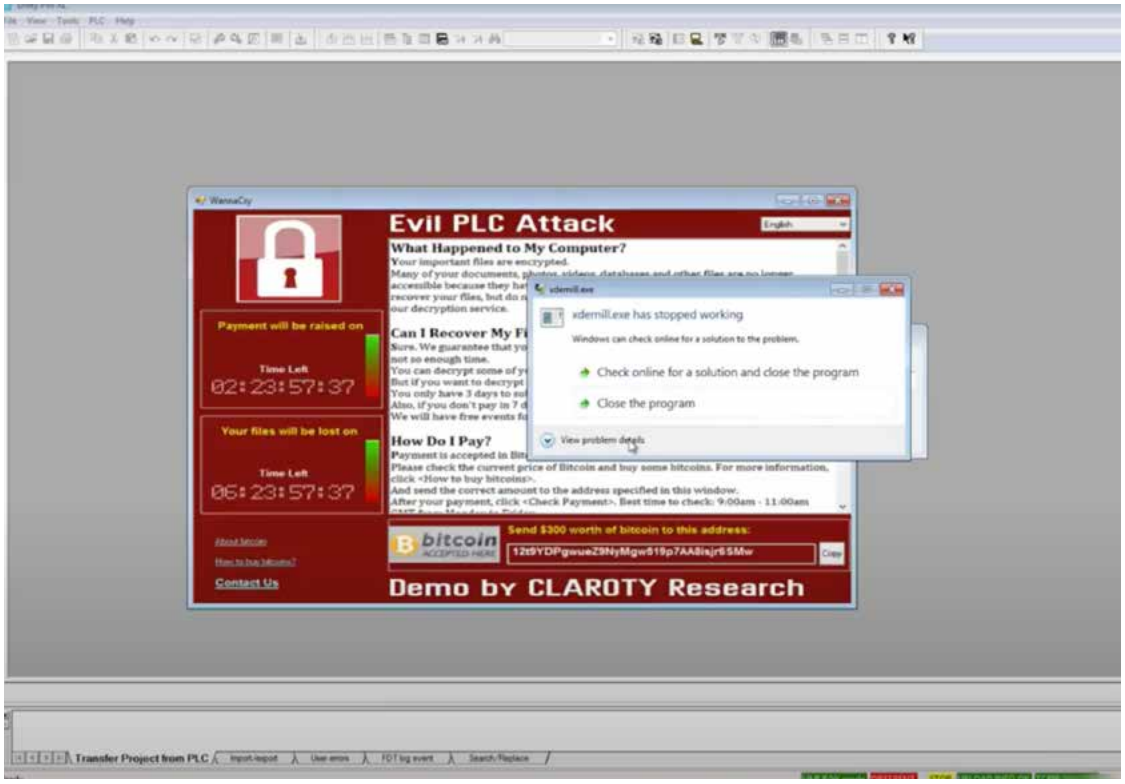
Demo

After creating the malicious *.apx file that will trigger our code execution vulnerability, we implanted it into a legitimate project file. We then use the UMAS protocol in order to download the malicious project file onto a M580 PLC, weaponizing it.

We then "waited" (in our lab) for an engineer to perform an upload procedure through Schneider Electric EcoStruxure Control Expert. When the upload is complete, EcoStruxure Control Expert will parse the infected project file, and the malicious *.apx file in particular, which will trigger our vulnerability, below.



The exploit was successful and the engineer's machine is now infected with our fake ransomware, below.

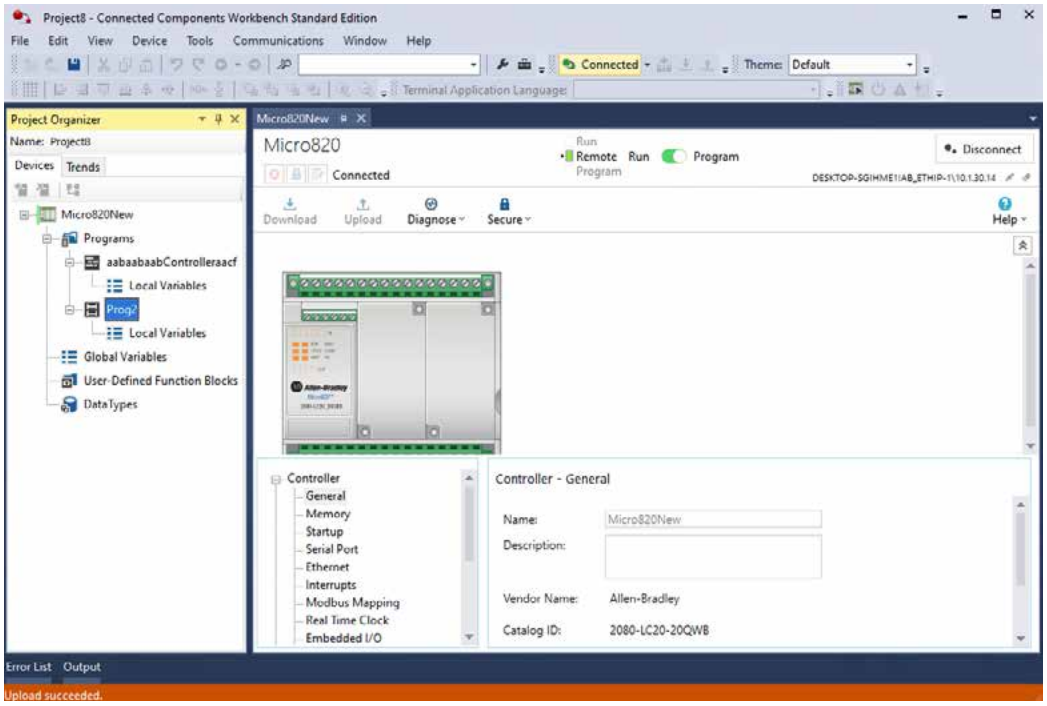


Example No. 3: Rockwell Automation Micro800

Studying the Platform

Rockwell Automation Micro800 is a versatile and modular series of PLCs, usually seen in smart manufacturing and process automation. In order to interact with and program the Micro800 series of PLCs, engineers use the Connected Components Workbench (CCW) software, which offers engineers the ability to configure their Micro800 devices, program them, and even to create integrations between the PLCs and an HMI screen.

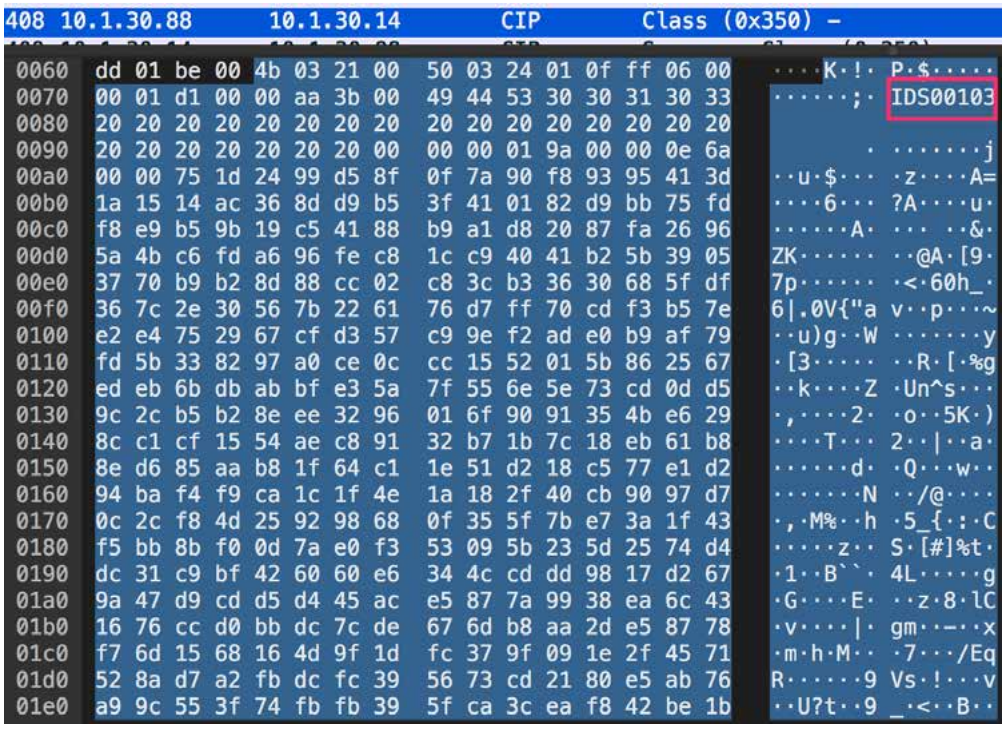
Using CCW, engineers can create a project for the PLC that contains basic configurations, metadata about the project, and a series of programs that could be programmed to the PLC using Ladder Diagram (LD), Structured Text (ST) or Function Block Diagram (FBD).



The CCW program, allowing engineers to program and interact with Micro820 PLC devices.

In order to perform a procedure such as a project download, the CCW engineering workstation communicates with Micro800 PLCs using the Common Industrial Protocol (CIP) protocol. The CIP protocol is an object-oriented protocol at its core, where each I/O device exports a set of objects, defining an interface for communications. For example, a device could export a file object, allowing remote users to perform various actions such as file read and write. Using this core concept, CIP-enabled devices could use a predefined set of common objects, or add proprietary ones to suit their specific use case.

Using the CIP protocol, CCW performs various actions and interacts with the Micro800 series of PLCs. To transfer data CCW uses mostly CIP classes File Object (0x37) and vendor-specific object 0x350. Using both class objects, the CCW software can set or get file data information from/to the PLC.



A CCW download procedure using CIP class 0x350 to transfer data to the PLC.

Project Structure

Before looking for a vulnerability in the upload procedure, we first need to fully understand what is transferred in the upload/download procedures, and the many parts that assemble a project. When looking at the network traffic when performing an upload procedure, we see that the PLC sends the engineering workstation a 7z compressed file, which in our case is named `IDS00103`, along with a few other files. When looking at the content of this file, we see all of the files that assemble the project:

```
Path = /private/tmp/a/Micro820_IDS00103
Type = 7z
Physical Size = 38157
Headers Size = 417
Method = LZMA2:192k
Solid = +
Blocks = 2

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2020-06-22 14:51:07 D....            0            0  Micro820
2020-06-22 14:51:07 D....            0            0  Micro820/Micro820
2020-06-22 14:51:07 ....A         1696        29423  2080LC2020QWBI.target
2020-06-22 14:51:07 ....A        24448               Controller.isaxml
2020-06-22 14:51:07 ....A        79204               Micro820/Micro820.annex
2020-06-22 14:51:07 ....A          561               Micro820/Micro820.isaxml
2020-06-22 14:51:07 ....A        68976               Micro820/Micro820/Micro820.annex
2020-06-22 14:51:07 ....A         9968               Micro820/Micro820/Micro820.isaxml
2020-06-22 14:51:07 ....A         1916               Micro820/Micro820/Prog1.annex
2020-06-22 14:51:07 ....A          422               Micro820/Micro820/Prog1.isaxml
2020-06-22 14:51:07 ....A         2525               Micro820/Micro820/Prog2.annex
2020-06-22 14:51:07 ....A          500               Micro820/Micro820/Prog2.isaxml
2020-06-22 14:51:07 ....A          149               Summary.xml
2020-06-22 14:51:08 ....A        61261         8317  persist1.ccwx
------------------- ----- ------------ ------------  ------------------------
2020-06-22 14:51:08              251626        37740  12 files, 2 folders
```

Content of the compressed file transferred during the upload procedure.

After performing the upload procedure, we see that the PLC sent the engineering workstation three types of files:

- **Metadata Files:** files that provide metadata information about the project and the PLC.
- **Engineering Workstation Project Files (*.annex):** files that contain all information relevant to the engineering workstation project. These files are parsed by the engineering workstation alone, and are used in the upload procedure by the workstation to create the project.
- **Compiled Programs (*.otc):** files that contain bytecode in the intermediate language that the PLC uses. These files are actually the program that the engineer created, and are later on executed by the PLC using its runtime. Those files are not sent as part of the compressed file, instead being sent separately.

Finding The Vulnerability: Unsafe Deserialization

In order to understand the possible attack surface on the engineering process, we started examining what type of information is stored in the project files the engineering workstation pulls from the PLC. As it turns out, a project is composed of many XML-based files holding information about the project itself and its sub-components. This includes the project's programs and their code blocks, the list of variables and tags those programs use, and general settings about the project and the PLC. This information is saved in the XML as tags of a corresponding type, telling the engineering workstation what the project structure and data is. For example, a program's variable will be represented in those XML by a variable tag, having its name, type and other details represented with an attribute in this tag.

When looking further into the content of said files, one file type caught our attention in particular: the `.annex` files. When we read the file content, we found out that those files contained .NET serialized data.

```
/p/t/a/M/M/Prog1.annex
<?xml version="1.0" encoding="utf-8"?>
<Root>
  <ElementDatas />
  <AnnexedDatas>
    <AnnexedData><![CDATA[AAEAAAD/////
AQAAAAAAAAAMAgAAAFlJU2FHUkFGLkFjZi5EYXRhSW1wb3J0LCBWZXJzaW9uPTYuNy4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49NTUZMjRmOGFjN2MyMGFhNgUBAAAAJUlTYUdSQUYuQWNmLkRhdGFJbXBvcnQuQWNmQW5uZXhlZERhdGEDAAAACl9zZXJ2aWNlSUQFX2RhdGEJX2ZpbGVEYXRhAwIEC1N5c3RlbS5HdWlkM0lTYUdSQUYuQWNmLkRhdGFJbXBvcnQuRmlsZVBlcnNpc3RlbmNlLklGaWxlQ29udGVudAIAAAACAAAABP3///
8LU3lzdGVtLkd1aWQLAAAAAl9hAl9iAl9jAl9kAl9lAl9mAl9nAl9oAl9pAl9qAl9rAAAAAAAAAAAAAAAIBwcCAgICAgICAlz6NagPtGJAv5eAZIW6cRoJBAAAAAo
PBAAAAFABAAACAAEAAAD/////
AQAAAAAAAAAMAgAAAFtJU2FHUkFGLkNDVy5FbWJGaWxlcy5Wc3gsIFZlcnNpb249Ni43LjAuMCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZXlUb2tlbj01NTYyN
GY4YWM3YzIwYWE2BQEAAAAqSVNhR1JBRi5DQ1cuRW1iRmlsZXMuVnN4LkNDV0VtYkZpbGVDb250ZW50BAAAAB48RmlsZUV4dGVuc2lvbj5rX19CYWNraW5nRmllbG
QZPEZpbGVOYW1lPmtfX0JhY2tpbmdGaWVsZB88QmFja3VwRmlsZU5hbWU+
a19fQmFja2luZ0ZpZWxkGDxDb250ZW50PmtfX0JhY2tpbmdGaWVsZAEBAQcCAgAAAAYDAAAABC5vdGMGBAAAAAlQcm9nMS5vdGMKCQUAAAAPBQAAAAUAAAACMA0KD
QoLCw==]]></AnnexedData>
    <AnnexedData ServiceID="bf5b1617-9a63-419f-92d5-bbb889f64d19">
      <FileContent Name="Prog1.AcfMlge" Extension=".AcfMlge" IsXml="true">
        <Root Version="7">
  <LanguageContainerStyle CommentTextColor="Green" CommentFont="Courier New, 10pt" PonctuationTextColor="Black"
PonctuationFont="Courier New, 10pt" IdentifierTextColor="Black" IdentifierFont="Courier New, 10pt" OperatorTextColor="Black"
OperatorFont="Courier New, 10pt" ReservedWordTextColor="Fuchsia" ReservedWordFont="Courier New, 10pt"
PouTextColor="BlueViolet" PouFont="Courier New, 10pt" NumberTextColor="Firebrick" NumberFont="Courier New, 10pt"
StringTextColor="Gray" StringFont="Courier New, 10pt" EditorTextAreaBackgroundColor="White" EditorFont="Courier New, 10pt"
Index="0" />
</Root>
      </FileContent>
    </AnnexedData>
  </AnnexedDatas>
</Root>
```

*Content of an *.annex file.*

We immediately noticed that the content of the `AnnexedData` contained a serialized .NET object, because of its very unique signature. This made us believe that a deserialization vulnerability is possible on this platform.

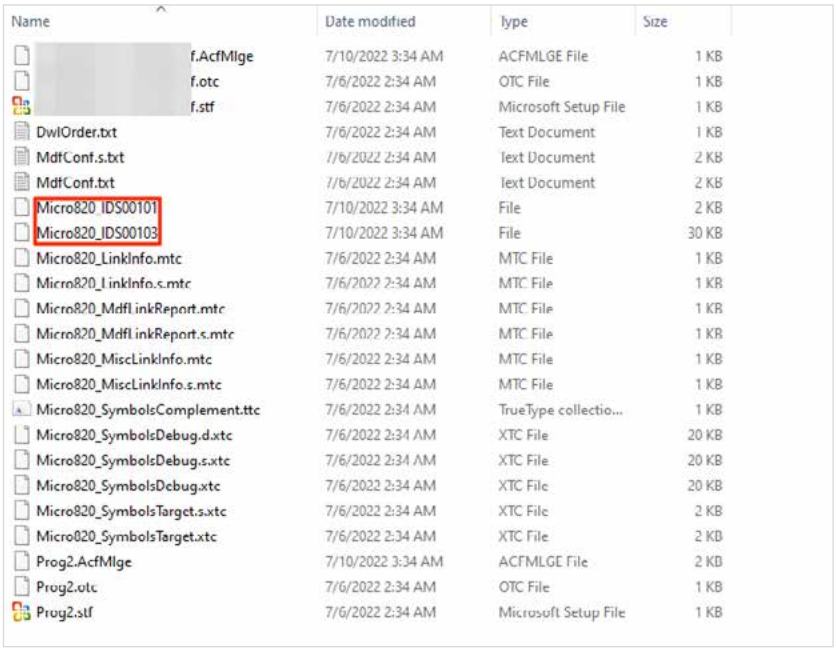
Before we are able to understand the vulnerability, we must have a basic understanding of what is a serialized object, and what is the serialization and deserialization process.

A serialized object is a binary or textual representation of the in-memory state of an object, allowing programs to transfer an object from one process to another, or to store an object for later use. Serialization is the process of taking the state of an object and transforming it into an independent representation that could be understood to recreate this object, while deserialization is the exact opposite, it is the process of transforming this object representation and recreating the object.

While serializing and deserializing objects sounds pretty simple and secure, in reality it's anything but. Many vulnerabilities exist that abuse the ability to force a program to create arbitrary objects that are made possible because of the deserialization process, and many applications fall victim to this vulnerability type. It is important to never deserialize information from an untrusted source, and to limit the classes that can be created by the deserialization process, only allowing for a set of intended classes to be deserialized.

Demo

Before weaponizing the PLC, we first need to perform an upload procedure in order to retrieve the current project that was set up on the PLC. This netted us with a ready-to-use project, that was the base for our vulnerability.



Name	Date modified	Type	Size
[redacted].AcfMlge	7/10/2022 3:34 AM	ACFMLGE File	1 KB
[redacted].otc	7/6/2022 2:34 AM	OTC File	1 KB
[redacted].stf	7/6/2022 2:34 AM	Microsoft Setup File	1 KB
DwlOrder.txt	7/6/2022 2:34 AM	Text Document	1 KB
MdfConf.s.txt	7/6/2022 2:34 AM	Text Document	2 KB
MdfConf.txt	7/6/2022 2:34 AM	Text Document	2 KB
Micro820_IDS00101	7/10/2022 3:34 AM	File	2 KB
Micro820_IDS00103	7/10/2022 3:34 AM	File	30 KB
Micro820_LinkInfo.mtc	7/6/2022 2:34 AM	MTC File	1 KB
Micro820_LinkInfo.s.mtc	7/6/2022 2:34 AM	MTC File	1 KB
Micro820_MdfLinkReport.mtc	7/6/2022 2:34 AM	MTC File	1 KB
Micro820_MdfLinkReport.s.mtc	7/6/2022 2:34 AM	MTC File	1 KB
Micro820_MiscLinkInfo.mtc	7/6/2022 2:34 AM	MTC File	1 KB
Micro820_MiscLinkInfo.s.mtc	7/6/2022 2:34 AM	MTC File	1 KB
Micro820_SymbolsComplement.ttc	7/6/2022 2:34 AM	TrueType collectio...	1 KB
Micro820_SymbolsDebug.d.xtc	7/6/2022 2:34 AM	XTC File	20 KB
Micro820_SymbolsDebug.s.xtc	7/6/2022 2:34 AM	XTC File	20 KB
Micro820_SymbolsDebug.xtc	7/6/2022 2:34 AM	XTC File	20 KB
Micro820_SymbolsTarget.s.xtc	7/6/2022 2:34 AM	XTC File	2 KB
Micro820_SymbolsTarget.xtc	7/6/2022 2:34 AM	XTC File	2 KB
Prog2.AcfMlge	7/10/2022 3:34 AM	ACFMLGE File	2 KB
Prog2.otc	7/6/2022 2:34 AM	OTC File	1 KB
Prog2.stf	7/6/2022 2:34 AM	Microsoft Setup File	1 KB

The project base we gained by performing an upload procedure.

We then prepared our payload: a modified serialized object that will result in arbitrary code execution whenever it will be deserialized. In order to create this object, we used the ysoserial.net library, which allowed us to create the malicious serialized object, for example:

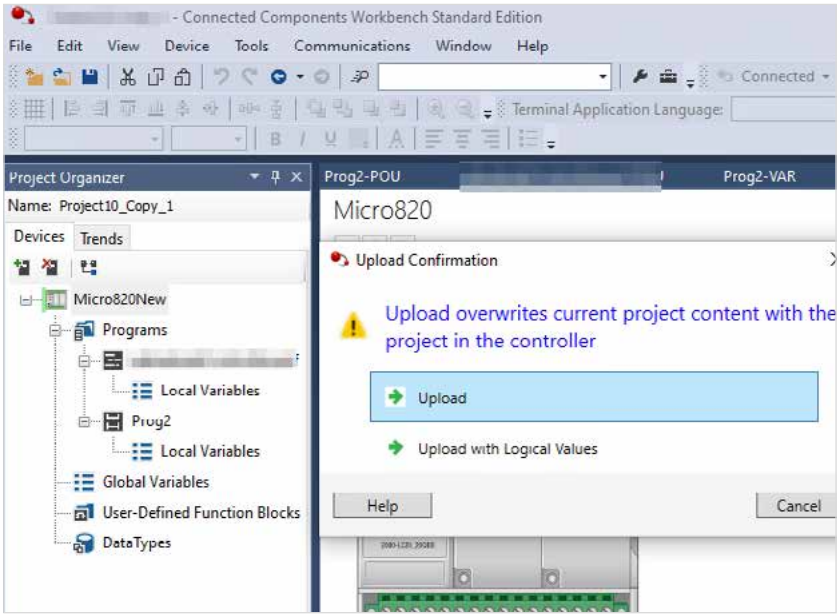
```
Windows PowerShell
PS C:\temp\ysoserial-1.34\Release> ./ysoserial.exe -f BinaryFormatter -g ClaimsIdentity -o base64 -c "mspaint"
AAEAAAD/////AQAAAAAAAAAEAQAAACVTeXN0ZW0uU2VjdXJpdHkuQ2xhaW1zLkNsYWltc0lkZW50aXR5AQAAABJtX3NlcmlhbGl6ZWRDbGFpbX
UFBRUFBQUQvLy8vL0FRQUFBQUFBQUFBTUFnQUFBRjVOYVdOeWIzTnZablF1VUc5M1pYSlRhR1ZzYkM1RlpHbDBiM0lzSUZabGNuTnBiMjQ5TXk
dnUTNWc2RIVnlaVDF1WlhWMGNtRnNMQ0JRZFdKc2FXTkxaWGxVYjJ0bGJqMHpNV0ptTXpnMU5tRmtNelkwWlRNMUJRRUFBQUJDVFdsamNtOXpi
zVmhiRk4wZFdScGJ5NVVaWGGwTGtadmNtMWhkSFJwYm1jdVZHVjRkRVp2Y20xaGRIUnBibWRTZFc1UWNtOXdaWEowYVdWekFRQUFBQTlHYjNKb
UW5KMWMyZ0JBZ0FBQUFZREFBQUF0UVU4UDNodGJDQjJaWEp6YVc5dVBTSXhMakFpSUdWdVkyOWthVzVuUFNKMWRHWXRPQ0kvUGcwS1BFOWlhbV
lFjbTkyYVdSbGNpQk5aWFJvYjJST1lXMWxQU0pUZEdGeWRDSWdTWE5KYm1sMGFXRnNURzlowkVWdVlXSnNaV1E5SWtaaGJITmxJaUI0Yld4dWN
RvdkwzTmphR1Z0WVhNdWJXbGpjbTl6YjJaMExtTnZiUzkzYVc1bWVDOHlNREEyTDNoaGJXd3ZjSEpsYzJWdWRHRjBhVzl1SWlCNGJXeHVjenB6
MVzVoYldWemNHRmpaVHBUZVhOMFpXMHVSR2xoWjI1dmMzUnBZM003WVhOelpXMWliSGs5VTNsemRHVnRJaUI0Yld4dWN6cDRQU0pvZEhSd09pQ
aGN5NXRhV055YjNOdlpuUXVZMjl0TDNkcGJtWjRMekl3TURZdmVHRnRiQ0krRFFvZ0lEeFBZbXBsWTNSRVlYUmhVSEp2ZG1sa1pYSXVUMkpxW1
0Z1WTJVK0RRb2dJQ0FnUEhOa09sQnliMk5sYzNNK0RRb2dJQ0FnSUNBOGMyUTZVSEp2WTJWemN5NVRkR0Z5ZEVsdVptOCtEUW9nSUNBZ0lDQWd
NtOWpaWE56VTNSaGNuUkpibVp2SUVGeVozVnRaVzUwY3owaUwyTWdiWE53WVdsdWRDSWdVM1JoYm1SaGNtUkZjbkp2Y2twdVkyOWthVzVuUFNK
zZlNJZ1UzUmhibVJoY21SUGRYUndkWFJGYm1OdlpHbHVaejBpZTNnNlRuVnNiSDBpSUZWelpYSk9ZVzFsUFNJaUlGQmhjM04zYjNKa1BTSjdlR
SWdSRzl0WVdsdVBTSWlJRXh2WVdSVmMyVnlVSEp2Wm1sc1pUMGlSbUZzYzJVaUlFWnBiR1ZPWVcxbFBTSmpiV1FpSUM4K0RRb2dJQ0FnSUNBOE
jJObGMzTXVVM1JoY25SSmJtWnZQZzBLSUNBZ0lEd3ZjMlE2VUhkdlkyVnpjejROQ2lBZ1BDOVBZbXBsWTNSRVlYUmhVSEp2ZG1sa1pYSXVUMkp
pkR0Z1WTJVK0RRbzhMMDlpYW1WamRFUmhkR0ZRY205MmFXUmxjajRMCw==
PS C:\temp\ysoserial-1.34\Release>
```

Then, we replaced the regular serialized object that the program intended to deserialize with our malicious object, packed it in the project file, and performed a download procedure, weaponizing the PLC with our malicious project.

```
<?xml version="1.0" encoding="utf-8"?>
<Root>
  <ElementDatas />
  <AnnexedDatas>
    <AnnexedData><![CDATA[AAEAAAD/////AQAAAAAAAAAMAgAAAfAFasvZCV1lkjj1ASDF5a23aVasw0vVasd4ADasdDASDvzxcCVAS54323F
    <AnnexedData ServiceID="bf5b1617-9a63-419f-92d5-bbb889f64d19">
      <FileContent Name="Prog2.AcfMlge" Extension=".AcfMlge" IsXml="true">
        <Root Version="7">
  <LanguageContainerStyle DisplayGrid="false" FunctionBlockInstanceName="true" VariableDisplayMode="NameAndAlias"
    <ShapeStyle Type="ISaGRAF.Workbench.Mlge.LanguageContainer.LD.Shape.WbLdRungShape" Assembly="ISaGRAF.Workbench
  </LanguageContainerStyle>
</Root>
      </FileContent>
    </AnnexedData>
  </AnnexedDatas>
</Root>
```

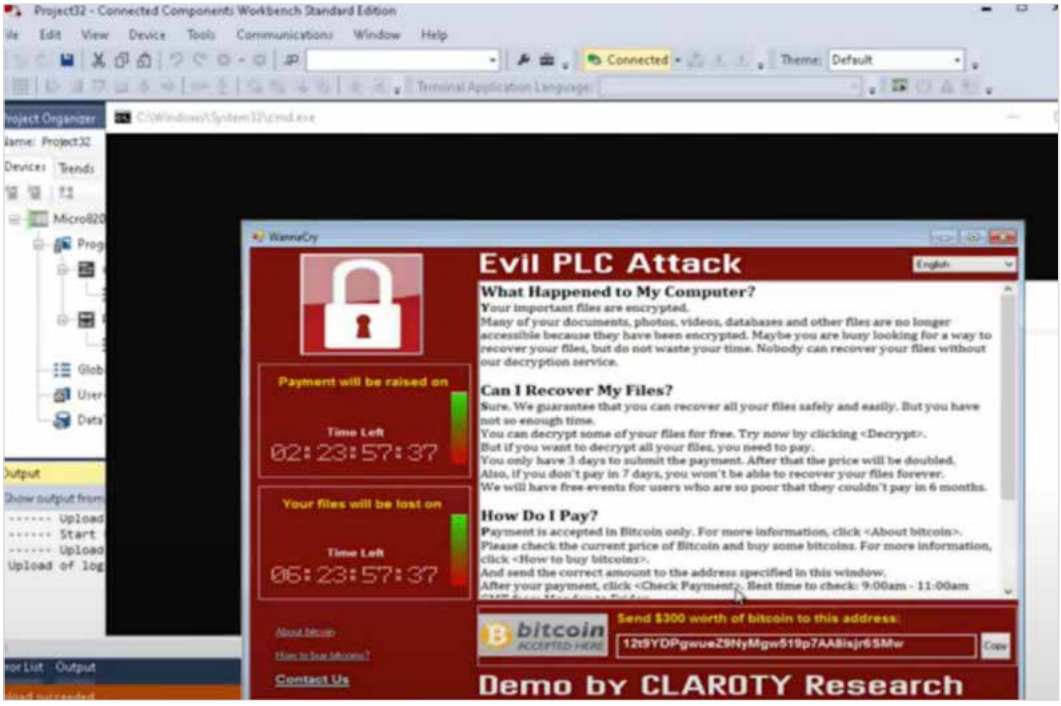
*The weaponized *.annex file, containing our modified serialized object payload.*

The only thing left to do is wait for an unsuspecting engineer to perform an upload procedure through CCW. When the upload is complete, CCW parses the infected `*.annex` file and our embedded serialized object will be deserialized, resulting in code execution.



An engineer performing an upload procedure on our weaponized PLC, which will result in arbitrary code execution.

The exploit was successful and the engineer's machine is now infected with our fake ransomware, below.



Mitigations

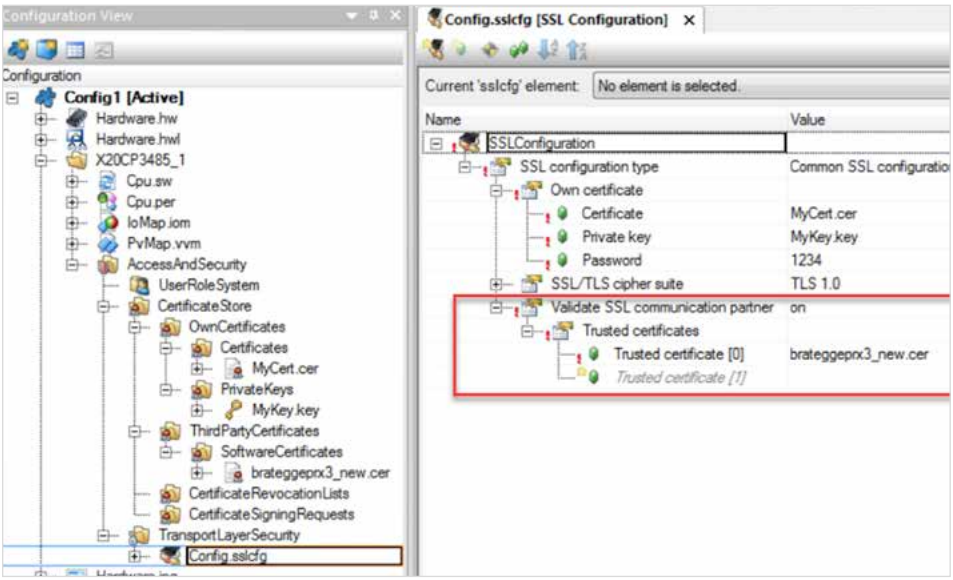
All of the vulnerabilities described in this paper were reported to the affected vendors in accordance with Team82's Coordinated Disclosure policy. Most vendors issued fixes, patches, or mitigation plans against the Evil PLC Attack.

That said, getting to 100% patching level, especially in critical infrastructure, is not easy and therefore requires additional mitigation steps to reduce the risk of the Evil PLC Attack. Here are a few we recommend:

In the Evil PLC attack technique, weaponizing is the first step and therefore, we recommend limiting physical and network access to PLCs as much as possible. There is no question that such devices shouldn't be accessible externally or exposed online. But also internal access should be limited to authorized engineers and operators only.

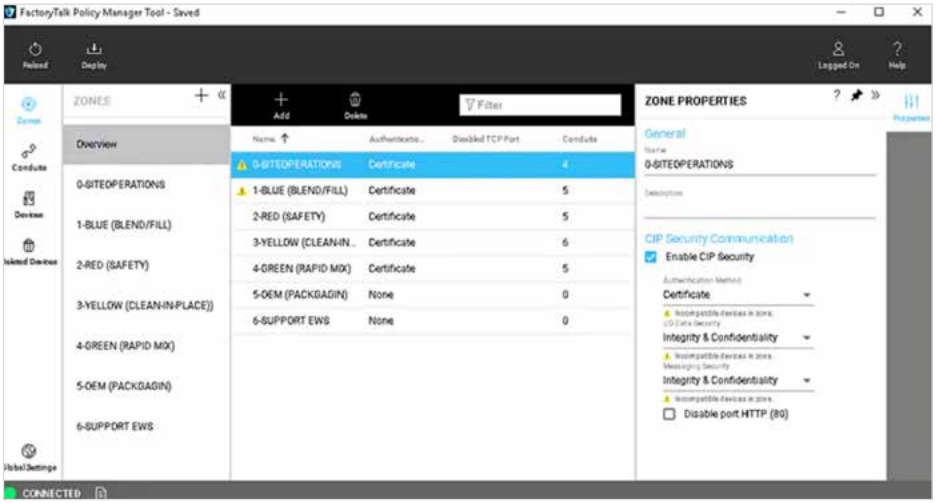
The process of securing the connection to your PLCs is long, tedious, and when implemented incorrectly even ineffective, so we recommend implementing the following:

- **Network Segmentation and Hygiene:** The first step in securing the connection to your PLCs is limiting access by strictly segmenting your network. Allow access to your PLCs only to a small set of engineering workstations, which reduces the attack surface in your network considerably.
- **Use Client Authentication:** It is crucial to configure the PLC to use a client authentication mechanism to validate the identity of the client, the engineering workstation. Currently, some vendors implement such communication protocols, where instead of allowing any engineering workstation to communicate with the PLC, only a specific and predefined set of engineering workstations are able interact with the PLC, by requiring the engineering workstation to present the PLC with a certificate. For example, B&R by ABB group supports TLS client authentication which can be configured in its Automation Server engineering workstation.



B&R Automation Studio SSL Configuration

- **Even Better, PKI:** A more robust solution is to use a full Public Key Infrastructure (PKI) system to validate and encrypt all traffic between the client, engineering workstation, and the server, the PLC. Mutual authentication, more commonly referred to as mutual TLS, helps to significantly reduce the risk of someone hacking your OT assets. However, the reality is that PKI is not yet implemented in many ICS product lines. Having said that, there are some vendors that already offer full-blown PKI systems including:
  - Rockwell Automation CIP Security
  - Siemens TIA v17
  - GE ToolsBoxST Secure Mode



Rockwell Automation FactoryTalk Policy Manager - CIP Security Management Platform

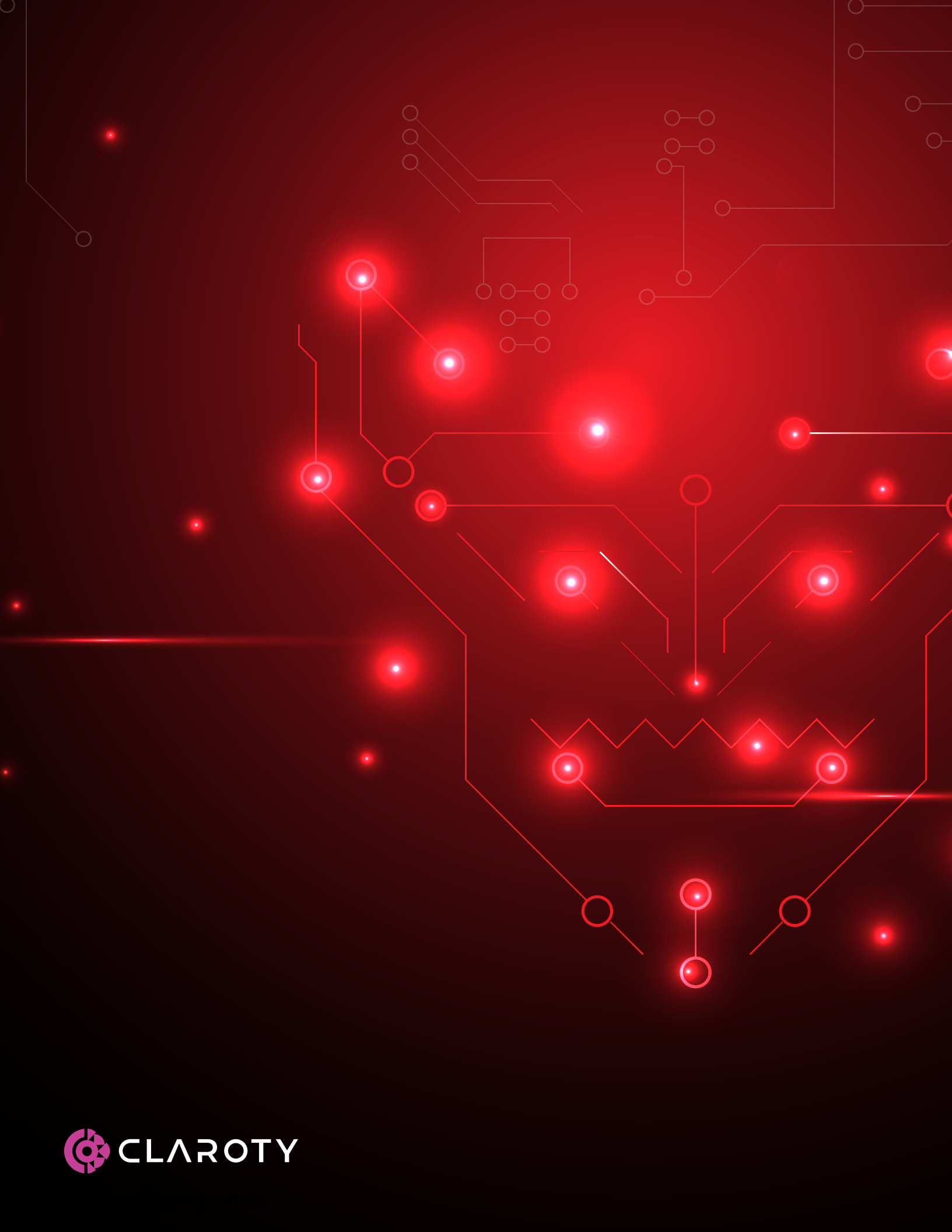
- **Network Traffic Monitoring:** The Evil PLC attack vector involves performing download/upload procedures to/from a PLC. As such, it is important to monitor your OT network traffic and detect these types of events in particular, and if such a procedure would occur in an unexpected situation, it could indicate an exploitation attempt.
- **Stay Up To Date:** As attackers and defenders alike research this new attack vector further, more vulnerabilities like the ones shown above will be discovered, and OT vendors will patch those vulnerabilities. It is important to stay up to date with your OT software, which will protect you from attacks exploiting those one-day vulnerabilities.

SUMMARY

We developed a unique technique that we named Evil PLC Attack, and through the process of developing this attack vector, we found a number of vulnerabilities in products from marketing-leading ICS vendors including Rockwell Automation, Schneider Electric, GE, B&R, Xinje, OVARRO and Emerson. Through this attack vector, a PLC is weaponized to the extent that when an engineer attempts to configure or troubleshoot it, the engineer's machine is compromised.

Adversaries may compromise and infect a PLC to implant malicious data and/or project files such that when an engineering workstation will perform an upload procedure, the PLC will transfer the malicious data to the engineering workstation that will parse it, trigger the vulnerability, and code will get executed on their machine.

Our work improves the security of PLCs, which are key elements of all automated industrial processes. Specifically, we helped further lock down the integrity of data uploads and downloads used by engineers to ensure the safety of processes across numerous critical industries.


CLAROTY