# Internet-facing PLCs - A New Back Orifice

Johannes Klick, Stephan Lau, Daniel Marzin, Jan-Ole Malchow, Volker Roth
Freie Universität Berlin - Secure Identity Research Group
<firstname>.<lastname>@fu-berlin.de

***Abstract*—Industrial control systems (ICS) are integral components of production and control processes. Our modern infrastructure heavily relies on them. Unfortunately, from a security perspective, thousands of PLCs are deployed in an Internet-facing fashion. Security features are largely absent in PLCs. If they are present then they are often ignored or disabled because security is often at odds with operations. As a consequence, it is often possible to load arbitrary code onto an Internet-facing PLC. Besides being a grave problem in its own right, it is possible to leverage PLCs as network gateways into production networks and perhaps even the corporate IT network. In this paper, we analyze and discuss this threat vector and we demonstrate that exploiting it is feasible. For demonstration purposes, we developed a prototypical port scanner and a SOCKS proxy that runs in a PLC. The scanner and proxy are written in the PLC's native programming language, the *Statement List* (STL). Our implementation yields insights into what kinds of actions adversaries can perform easily and which actions are not easily implemented on a PLC.**

## I. Introduction

Industrial control systems (ICS) are integral components of production and control tasks. Modern infrastructure heavily relies on them. The introduction of the *Smart Manufacturing (Industry 4.0)* technology stack further increases the dependency on industrial control systems [1]. Modern infrastructure is already under attack and offers a broad attack surface, ranging from simple XSS vulnerabilities [2, 3] to major design flaws in protocols [4, 5].

The canonical example of an attack on an industrial control system is the infamous Stuxnet worm that targeted an Iranian uranium enrichment facility. However, adversaries increasingly target ordinary production systems [6]. A recent example is the forced shutdown of a blast furnace in a German steelworks in 2014. The attackers reportedly gained access to the pertinent control systems via the steelwork's business network [7]. This is a typical attack vector because business networks serve humans and humans are susceptible to spear phishing.

Arguably, spear phishing is easy to carry out when accompanied with research and social engineering. However, in far too many cases, even easier ways exist into industrial control systems. Published scan data shows that thousands of ICS components, for example, programmable logic controllers (PLCs), are directly reachable from the Internet [8, 9, 10]. While only one PLC of a production facility may be reachable in this fashion, the PLC may connect to internal networks with many more PLCs. This is what we call the "deep" industrial network. In this paper, we investigate how adversaries can leverage exposed PLCs to extend their access from the Internet to the deep industrial network.

The approach we take is to turn PLCs into gateways (we focus on Siemens PLCs). This is enabled by a notorious lack of proper means of authentication in PLCs. A knowledgeable adversary with access to a PLC can download and upload code to it, as long as the code consists of MC7 bytecode, which is the native form of PLC code. We explored the runtime environment of PLCs and found that it is possible to implement several network services using uploaded MC7 code. In particular, we implemented

- a SNMP scanner for Siemens PLCs, and
- a fully fledged SOCKS proxy for Siemens PLCs

entirely in *Statement List* (STL), which compiles to MC7 byte code. Our scanner and proxy can be deployed on a PLC without service interruption to the original PLC program, which makes it unlikely that unsuspecting operators will notice the infection. In order to demonstrate and analyze deep industrial network intrusion, we developed a proof of concept tool called *PLCinject*. Based on our proof of concept, we analyzed whether the augmentation of the original code with our PLC malware led to measurable effects that might help detecting such augmentations. We looked at timing effects, specifically. We found that augmented code is distinguishable from unaugmented code, that is, statistically significant timing differences exist. The difference is minor in absolute terms, that is, the augmentation does not likely affect a production process and hence it will not be noticable unless network operators actively monitor for malicious access. The downside is that operators of industrial networks must include PLCs in their vulnerability assessment procedures and they must actively monitor internal networks for malicious network traffic that originates from their own PLCs. Moreover, adversaries can leverage our approach to attack a company's business network from the industrial network. This means that network administrators must guard their business networks from the front and the back. The remainder of this paper is organized as follows. We begin with a discussion of work related to ours in §II. In §III, we give technical background for readers unfamiliar wth industrial control systems. We describe our attack and intrusion methods in §IV. In §VI, we discuss mitigations and §VII concludes the paper.

## II. Related Work

Various attacks on PLCs have been published. Most attacks target the operating systems of PLCs. In contrast we leverage the abilities of logic programs running on the PLCs. As such we do not use any unintended functionality. In the following, we compare our approach to well-known (code) releases and published attacks that manipulate logic code. One of the most

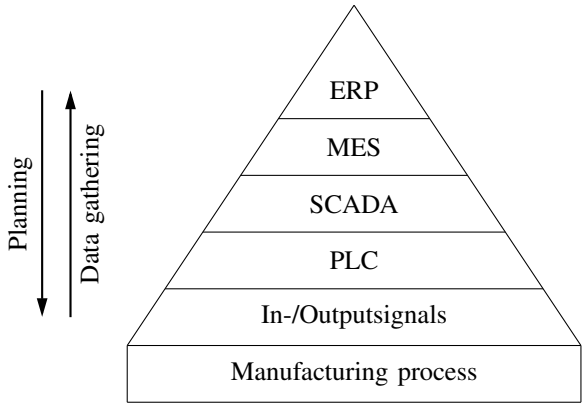



**Figure III.1:** *Automation pyramid, adopted from [15]*

cited SCADA attack descriptions is Beresfords' 2011 Black Hat USA release [5]. He demonstrated how credentials can be extracted from remote memory dumps. In addition he shows how to start and stop PLCs through replay attacks. In contrast to our work he does not alter the logic program on the PLC. In 2011 Langner released "A timebomb with fourteen bytes" [11] wherein he describes how to inject rogue logic code into PLCs. He borrows the same code prepending technology as we do, from Stuxnet. He conceptualizes how to take control away from the original code. In contrast, our program runs in parallel to the original code with the goal to not interfere with the original code's execution. An attack similar to Langners' was presented at Black Hat USA 2013 by Meixell and Forner [12]. In their release they describe different ways of exploiting PLCs. Among those are ways to remove safety checks from logic code. Again, our approach differs as we add new functionality while preserving original functionality. To our best knowledge, the first academic paper on PLC malware was published by McLaughlin in 2011 [13]. In this work he proposes a basic mechanism for dynamic payload generation. He presents an approach based on symbolic execution that recovers boolean logic from PLC logic code. From this, he tries to determine unsafe states for the PLC and generates code to trigger one of these states. In 2012 McLaughlin published a followup paper [14], which extends his approach in a way that automatically maps the code to a predefined model by means of model checking. With his model, he can specify a desired behaviour and automatically generate attack code. In his work McLaughlin focuses on manipulating the control flow of a PLC. We, in contrast, use the PLC as a gateway to the network and leave its original functions untouched.

## III. Industrial Control Systems

Figure III.1 illustrates the structure of a typical company that uses automation systems. Industrial control systems consist of several layers. At the top are enterprise resource planning (ERP) systems, which hold the data about currently available resources and production capacities. Manufacturing execution systems (MES) are able to manage multiple factories or plants and receives tasks from ERP systems. The systems below the MES are located in the factory. Supervision, control and data acquisition (SCADA) systems control production lines. They provide data about the current production state and they provide means for intervention. The devices holding the logic for production processes are called programmable logic controllers (PLC). We explain them in more detail in section III-A. Human machine interfaces (HMI) display the current progress and allow operators to interact with the production process.

### A. PLC Hardware

A PLC consists of a central processing unit (CPU) which is attached to a number of digital and analog inputs and outputs. A PLC program stored on the integrated memory or on a external Multi Media Card (MMC) defines how the inputs and outputs are controlled. A special feature of a PLC is the guaranty of a defined executions time to control time critical processes. For communication or special purpose applications the functionality of a CPU can be extended with modules. The Siemens S7-314C-2 PN/DP we use in our experiments has 24 digital inputs, 16 digital outputs, 5 analog inputs, 2 analog outputs and a MMC slot. It is equipped with 192 KByte of internal memory, 64 KByte can be used for permanent storage. Additionally, the PLC has one RS485 and two RJ45 sockets [16].

### B. PLC Execution Environment

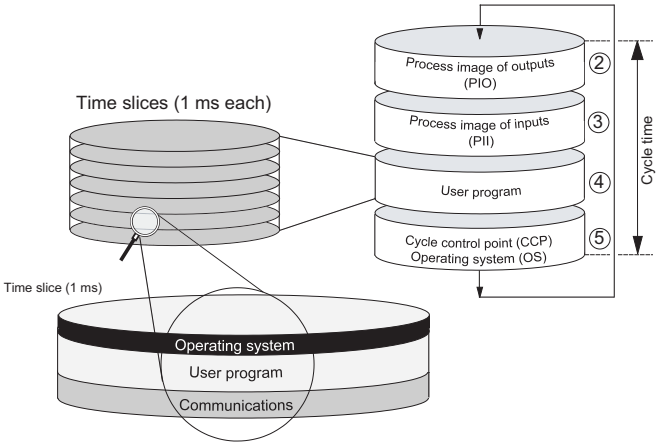



**Figure III.2:** *Overview of program execution, extracted from [17]*

Siemens PLCs run a real time operating system (OS), which initiates the cycle time monitoring. Afterwards the OS cycles through four steps (see figure III.2). In the first step the CPU copies the values of the process image of outputs to the output modules. In the second step the CPU reads the status of the input modules and updates the process image of input values. In the third step the user program is executed in time slices with a duration of 1 ms. Each time slice is divided into three parts, which are executed sequentially: The operating system, the user program and the communication. The number of time slices depends on the current user program. By default the time should be not longer than 150 ms. An engineer can configure a different value. If the defined time expires, an interrupt routine is called. In the common case the CPU returns to the start of the cycle and restarts the cycle time monitoring [17].

### C. Software

Siemens provides their *Total Integrated Automation* (TIA) portal software to engineers for the purpose of developing PLC programs. It consists of two main components. The STEP 7 as development environment for PLCs and WinCC to configure HMIs. Engineers are able to program PLC in *Ladder Diagram* (LAD), *Function Block Diagram* (FBD), *structured control language* (SCL) and *Statement List* (STL). In contrast to the text-based SCL and assembler-like STL the LAD and FBD languages are graphical. PLC programs are divided into units of organization blocks (OB), functions (FC), function blocks (FB), data blocks (DB), system functions (SFC), system function blocks (SFB) and system data blocks (SDB). OBs, FCs and FBs contain the actual code while DBs provide storage for data structures and SDBs current PLC configurations. For internal data storage addressing the prefix M for memory is used.

### D. PLC Programs

A PLC program consists of at least one organization block called OB 1, which is comparable to the *main()* function in a traditional C program. It will be called by the operating system. There exist more organization blocks for special purposes, for example, OB 100. This block is called once when the PLC starts and is used usually for the initialization of the system. Engineers can encapsulate code by using functions and function blocks. The only difference is an additional DB as a parameter to calling a FB. The SFCs and SFBs are built into the PLC. The code can not be inspected. The STEP 7 software knows which SFCs and SFBs are available based on hardware configuration steps. The following examples give an overview of the the programming languages SCL, LAD and STL. Each example shows the same configuration of three inputs and one output. First, the CPU performs a logical AND operation of inputs `0.0` and `0.1`. Next, it calculates a logical OR operation of the outcome and the input `0.2`. The result is written to output `0.0` which sets the logical values to the connected wire in the next cycle. The first example represents the described program in STL. This is done in four lines of assembler-like instructions. Each line defines one instruction.

```
1 A     %I0.0
2 A     %I0.1
3 O     %I0.2
4 =     %Q0.0
```

The next example shows the same program in the text-based language SCL. This program can be expressed in one line.

```
1 %Q0.0 := (%I0.0 AND %I0.1) OR %I0.2;
```

The graphical example needs the help of the STEP 7. Inputs and outputs are positioned through *drag & drop* on the wire. New connections can be made on predefined positions by selecting the wire-tool from the toolbar above. Figure III.3 shows the graphical representation of our example program.

The following description can also be found in the Siemens manual delivered with the PLC [18]. The CPU has several registers used for execution and current state. For binary operations the *status word* register is important. All binary operations influence this register. The CPU uses for calculations

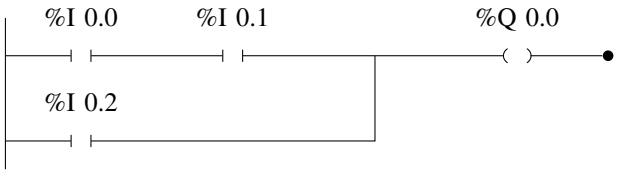



**Figure III.3:** *Function block diagram example*

up to four accumulator registers of 32 bits width. They are organized like a stack. It is possible to address independently each byte of the top register. Before a new value is loaded into the *accumulator one* the current value is copied to *accumulator two*. For adding two numbers the values have to be loaded successively into the accumulator register before the *+D* operation is called. The result is written back into *accumulator one*. In STL the program would look like as following.

```
1 L     DW#16#1  // ACCU1=1
2 L     DW#16#2  // ACCU1=2,ACCU2=1
3 +D             // ACCU1=ACCU1+ACCU2
```

Code which is used multiple times in the program should be implemented as functions. These functions can be called from every point in the code. The *CALL* instruction allows to jump into the defined function. The necessary parameters are defined in the called function header and have to be specified below every *CALL* instruction.

```
1 CALL FC1
2      VAR1 := 1
3      VAR2 := W#16#A
```

As mentioned earlier the only difference between function blocks and functions is a reference to the corresponding data block. In many cases the program needs storage which is assigned to a specific function to read constants or save process values. It is unusual to put constants direct in the code, because the code have to be recompiled after every change. In contrast data blocks can be manipulated easily even remotely. A function block call looks like as following.

```
1 CALL FB1, %DB1
2      VAR1 := 1
3      VAR2 := W#16#A
```

Both function types can define different parameters: `IN`, `OUT`, `IN_OUT`, `TEMP` and `RET_VAL`. The FB `STAT` parameters are stored in its data block, which is passed as an additional argument. The `TEMP` type declares local variables which only are available in the function. The other types are self explanatory.

### E. Binary Representation of PLC Program

Every code written in any language is compiled into MC 7. The opcode length of MC 7 instructions is variable and the encoding of parameters differs on many instructions. The binary representation of the example program from the section before looks as following.

```
1  00100000        A       %I0.0
2  00110000        A       %I0.1
3  01120000        O       %I0.2
4  41100000        =       %Q0.0
```

### F. Network Protocol

The Siemens PLCs uses the proprietary S7Comm protocol for transferring blocks. It is a remote procedure call (RPC) protocol based on TCP/IP and ISO over TCP. Figure III.4 illustrates the encapsulation of the protocols. The protocol provides the following functionality:

- System State List (SSL) request
- List available blocks
- Read/write data
- Block info request
- Up-/download block
- Transfer block into filesystem
- Start, stop and memory reset
- Debugging

The executing of one of these function requires an initialized connection. After a regular TCP handshake the ISO over TCP setup is proceeded to negotiate the PDU size. In the S7Comm protocol the client has to provide additionally to his preferred PDU size the rack and slot of the CPU (see connection setup in figure III.5). The CPU responses with its preferred PDU size and both agree to continue with the minimum of both values. After this initialization the client is able to invoke the functions on the CPU. Figure III.5 shows the packet order of a download block function including the transfer into the filesystem. The PLC controls the download process after receiving the *download request*. The number of *download block* requests depends on the length of the block and the PDU size. The end is signaled with the *download end* request. The PLC waits after receiving the acknowledgement for further requests. Finally the transferred block should be persisted by calling the *plc control* request. With the destination filesystem `P` as parameter the CPU stores the block and executes it.

The upload process is similar. The engineering work station (EWS) requests for the upload of a specific block and waits for the acknowledgement. After receiving the acknowledgement without errors the EWS starts requesting the block. The responses contain the data of the block. The EWS repeats the procedure as long as the whole block is transferred. The end is signaled with an *upload end* request. The transferred blocks are structured and consists of header, data part and footer. The table III.1 shows the structure of the known bytes. The footer contains information about the parameters used for calling the function. Not every byte of the header and footer are known well, but we have identified the necessary areas to understand the content.

## IV. Attack Description

The search engine SHODAN shows that thousands of industrial control systems are direct accessible via the Internet [8, 10]. As shown in chapter III it is possible to download and upload the PLC program code. This enables attacker to manipulate the logic code of the PLCs that reads inputs and outputs. Furthermore the PLC offers a system library [21] which contains functions to establish arbitrary TCP / UDP communication. An attacker can use the full TCP/UDP support to scan the local production network behind the internet-facing PLC. Furthermore he can leverage this PLC as a gateway to reach all the other production or network devices.

Like Stuxnet we prepend the attacker's code to the existing logic code of the PLC. The malicious code will be executed at the very beginning of organization block 1 (OB) in addition to the normal control code. That is why the PLC will not be disturbed in its function. The easiest way is to download

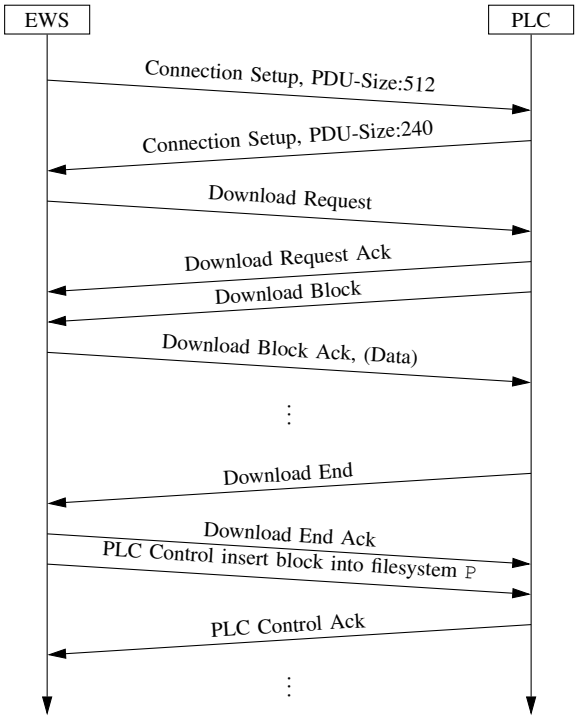



**Figure III.5:** *Download block sequence diagram*

**Table III.1:** *Block structure, adopted from code [20]*

| Description | Bytes | Offset |
|---|---|---|
| Block signature | 2 | 0 |
| Block version | 1 | 2 |
| Block attribute | 1 | 3 |
| Block language | 1 | 4 |
| Block type | 1 | 5 |
| Block number | 2 | 6 |
| Block length | 4 | 8 |
| Block password | 4 | 12 |
| Block last modified date | 6 | 16 |
| Block interface last modified date | 6 | 22 |
| Block interface length | 2 | 28 |
| Block Segment table length | 2 | 30 |
| Block local data length | 2 | 32 |
| Block data length | 2 | 34 |
| Data (MC 7 / DB) | x | 36 |
| Block signature | 1 | 36+x |
| Block number | 2 | 37+x |
| Block interface length | 2 | 39+x |
| Block interface blocks count | 2 | 41+x |
| Block interface | y | 43+x |
| ... | | |

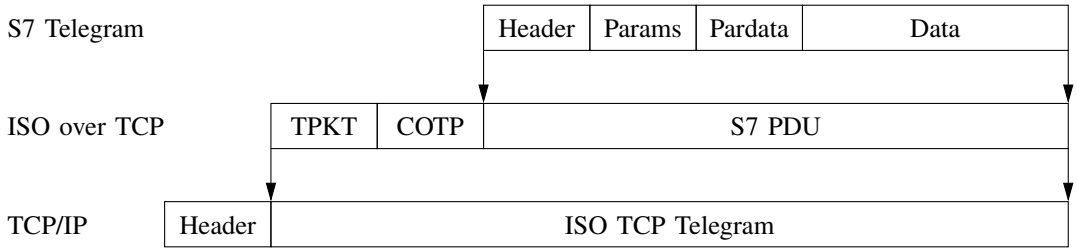
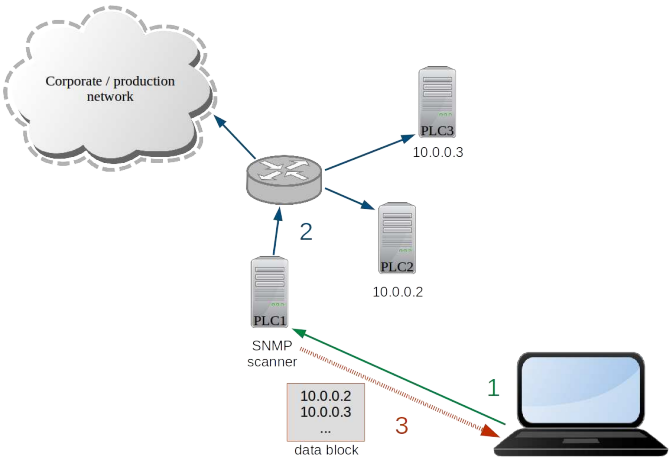
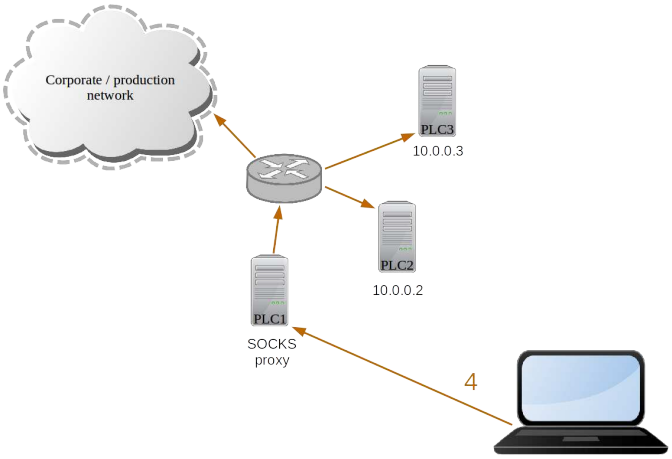



**Figure III.4:** *Packet encapsulation, adopted from [19]*



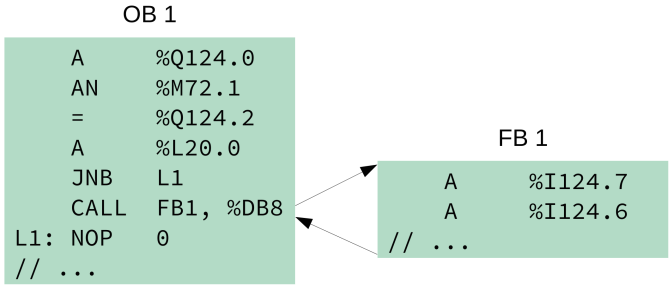

**(a)** *Attacker abuses the PLC to scan the local network for SNMP devices*



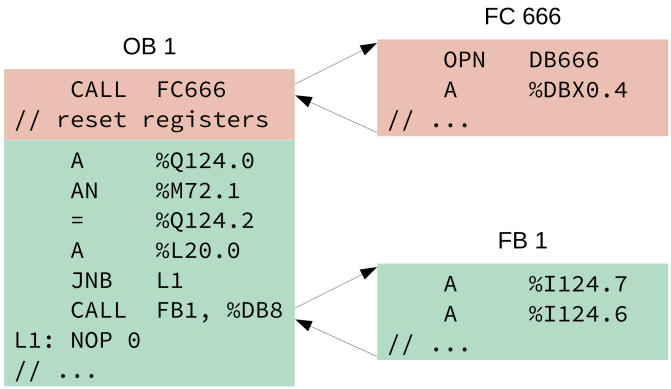

**(b)** *Now he can use the PLC as a gateway into the local network*

**Figure IV.1:** *Attack cycle*

the OB1 of Siemens PLCs and add a *CALL* instruction to an arbitrary function under our control, in our example a function called *FC 666*. Then the *patched OB1*, *FC 666* and additional blocks will be uploaded to the PLC. Picture IV.2 illustrates the code injection process. With the next execution cycle of the PLC the new uploaded program including the attacker's code will be executed without any kind of service disruption. This process enables the attacker to run any additional malicious code on the PLC. With this paper we publish a tool called *PLCinject* that will automate this process [22]. Having this capabilities an attacker is able to execute the attack cycle as shown in figure IV.1. In step one the attacker injects a *SNMP Scanner* that runs in addition to the normal control code of the PLC. After a full SNMP scan of the local network (step two) the attacker can download the scan results from the PLC (step three). The attacker has now an overview of the network behind the Internet-facing PLC. The attacker removes the SNMP scanner and injects a *SOCKS Proxy* to the PLC logic program (step four). This enables the attacker to reach all PLCs in the local production network via the compromised PLC which acts as a SOCKS proxy. In the next two sections we




**(a)** *Original program.*




**(b)** *Patched program. The red blocks are added by PLCinject.*

**Figure IV.2:** *Scheme of patching the PLCs program.*

are going to explain the implementation of the SNMP scanner and the SOCKS proxy. We will not explain every operation and system function in detail. For a complete description of

those we refer to [18] and [21].

### A. SNMP Scanner

Siemens PLCs can not be used as a TCP port scanner because the used TCP connection function *TCON* cannot be aborted until the function has established an connection. Furthermore it is only possible to run eight TCP connection in parallel on a Siemens S7-300 PLC. Consequently the PLC is only able to perform a TCP scan until eight unsuccessful connection attempts occured. This limitation do not apply to stateless UDP connections. That is why we use the UDP based *Simple Network Management Protocol (SNMP)*. SNMP version 1 is defined in RFC 1157 [23] and was developed for monitoring and controlling network devices. A lot of network devices and most of the Siemens Simatic PLCs have SNMP enabled by default. Siemens PLCs are very communicative in case of enabled SNMP. By reading the *SNMP sysDesc object* with the *OID 1.3.6.1.2.1.1.1*, the Siemens PLC will transmit its *product type, product model number, hardware and firmware version* as shown in the following SNMP response:

```
Siemens, SIMATIC S7, CPU314C-2 PN/DP,
6ES7 314-6EH04-0AB0 , HW: 4, FW: V3.3.10.
```

The system description is very useful for matching discovered PLCs against vulnerability and exploit databases. The firmware of PLCs is not very often patched. There are mainly two reasons: On the one hand a PLC firmware patch will interrupt the production process which causes a negative monetary impact. On the other hand a firmware patch of the PLC can lead to a loss of the production certification or other kind of quality assurance that is important for the customers of the manufacturing company. That is why the probability to find a Siemens PLC with a known vulnerability is very high. The SNMP scanner can be broken down into the following steps:

1) Get local IP and subnet
2) Calculate IPs of the subnet
3) Set up UDP connection
4) Send SNMP request
5) Receive SNMP responses
6) Save responses in a DB
7) Stop scanning and disconnect UDP connection

As described in the chapter III the programming of a PLC is quite different from normal programming with e.g. the C language on a X86 system. Each PLC program is cyclically executed. That is it is needed save the state of the program after each step with condition variables. For reasons of comprehensibility we will only explain steps one to three.

Figure IV.3 shows a code snippet of step one that calls the *RDSYSST* function. The *RDSYSST* function reads the internal System State List (SSL) of the Siemens PLC to obtain the PLC's local IP. SSL requests are normally used for diagnostic purposes. Line 14 and 15 will end the function in the case that the *RDSYST* function is busy. Figure IV.4 shows how the program calculates the first local IP. This is done by bitwise logic AND operation of the PLC's local IP address with its subnet mask, which returns the start address of the local network adress range (line 24 - 30). Now the SNMP scanner needs to

```
0001 get_ip : NOP 1
0002
0003 // read ip from system state list (SZL)
0004       CALL  RDSYSST
0005          REQ         :=TRUE
0006          SZL_ID      :=W#16#0037
0007          INDEX       :=W#16#0000
0008          RET_VAL     :=#sysst_ret
0009          BUSY        :=#sysst_busy
0010          SZL_HEADER :="DB".szlheader.SZL_HEADER
0011          DR          :="DB".ip_info
0012
0013 // wait until SZL read finished
0014       A      #sysst_busy
0015       BEC
0016
0017       SET
0018       S      #got_ip
```

**Figure IV.3:** *Get PLCs local IP*

```
0020 // calc first ip of local network
0021 // L "DB".ip_info.local_ip
0022       OPN    "DB"
0023       L      %DBD406
0024 // L "DB".ip_info.subnet
0025       L      %DBD410
0026       AD
0027 // T "DB".ADDRESS.rem_ip_addr
0028       T      %DBD64
0029
0030 // get number of hosts from subnet
0031 // L "DB".ip_info.subnet
0032       L      %DBD410
0033       L      DW#16#FFFFFFFF
0034       XOD
0035       T      #num_hosts
```

**Figure IV.4:** *Calculate the local nets first IP and the maximal number of hosts*

know how often it must increment the IP address to cover the whole local subnet. Therefore we XOR the subnet mask with `0xFFFFFFFF` (line 35 - 39). The result is the number of IP addresses in the subnet. Figure IV.5 shows how to set up an

```
0001       CALL  TCON , "TCON_DB_SCAN"
0002          REQ     :=#connect
0003          ID      :=1
0004          DONE    :=#con_done
0005          BUSY    :=#con_busy
0006          ERROR   :=#con_error
0007          STATUS  :=#con_status
0008          CONNECT :="DB".TCON_PAR_SCAN
0009
0010       AN     #connected
0011       =      #connect
```

**Figure IV.5:** *Setup a UDP connection*

UDP connection in STL. At first we need to call the *TCON* function with special parameters in our *TCON_PAR_SCAN* data block. In case of UDP the *TCON* function does not set up a connection, this will only be done in the case of TCP because it is connection oriented in contrast to UDP. But calling the

TCON parameter once is not enough. The connection function will start to work when the *#connect* variable raises from 0 to 1 between two calls of the function. That is why we programmed a toggle function after the first appearance of the connect function (line 10 - 11). This will change the *#connect* value from false to true after *TCON* has been called the first time in a cycle. The *TCON* function will detect a raising signal edge on its call in the next cycle and will then be executed. The next step is to send the UDP based SNMP packets and receive them. This will be done by calling the functions *TUSEND* and *TURCV*. After the SNMP scan has been completed all data will be stored in data block which can be downloaded by the attacker (step 3).

### B. SOCKS 5 Proxy

Once the attacker has discovered all SNMP devices, including the local PLCs, the next step is to connect to them. This can be accomplished by using the accessible PLC as a gateway into the local network. To achieve this we chose to implement a SOCKS 5 proxy on the PLC. This has two main reasons. At first the SOCKS protocol is quite lightweight and easy to implement. Furthermore all applications can use this kind of proxy, either they are SOCKS aware and thus can be configured to use one or you use a so-called proxifier to add SOCKS support to arbitrary programs. The *SOCKS 5* protocol is defined in *RFC 1928* [24]. An error-free TCP connection to a target through the proxy consists of the following steps:

1) The client connects via TCP to the SOCKS server and sends a list of supported authentication methods.
2) The server replies with one selected authentication method.
3) Depending on the selected authentication method the appropriate sub-negotiation is entered.
4) The client sends a connect request with the targets IP.
5) The server sets up the connection and replies. All subsequent packets are tunnelled between client and target.
6) The client closes the TCP connection.

Our implementation offers the minimal necessary functionality. It supports no authentication, so we can skip step 3. Also we do not support proper error handling. In the end only TCP connects with IPv4 addresses are supported. Once the client connected, we expect this message flow:

1) Client offers authentication methods: any message, typically `0x05 <authcount-n>` (1 byte) `<authlist>` (n bytes).
2) Server chooses authentication method: `0x05 0x00` (perform no authentication).
3) Client wants to connect to target: `0x05 0x01 0x00 0x01 <ip>` (4 bytes) `<port>` (2 bytes).
4) Server confirms connection: `0x05 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00`.
5) Client and target can now communicate through the connection with the server.

As previously mentioned, programs on the PLC are cyclically executed. This is why we use a simple state machine to handle the SOCKS protocol. Therefore we number each state and use a jump list to execute the corresponding code block, see figure IV.6. A state transition is achieved by incrementing

```
0002        JL    lend
0003        JU    bind                  // state == 0
0004        JU    negotiate             // state == 1
0005        JU    authenticate          // state == 2
0006        JU    connect_request       // state == 3
0007        JU    connect               // state == 4
0008        JU    connect_confirm       // state == 5
0009        JU    proxy                 // state == 6
0010        JU    reset                 // state == 7
0011 lend:  JU    end
```

**Figure IV.6:** *Jump list for the states of SOCKS 5*

the state number which is persisted in a data block.

```
0005        CALL  TRCV , "TRCV_client_DB"
0006           EN_R     :=TRUE
0007           ID       :=W#16#0001
0008           LEN      :=0
0009           NDR      :=#rcv_ndr
0010           BUSY     :=#rcv_busy
0011           ERROR    :=#rcv_error
0012           STATUS   :=
0013           RCVD_LEN :=
0014           DATA     :="buffers".rcv
0015
0016        A     #rcv_ndr
0017        AN    #rcv_busy
0018        AN    #rcv_error
0019        JC    next_state
```

**Figure IV.7:** *Receive the clients authentication negotiation*

```
0008        CALL  TSEND , "TSEND_client_DB"
0009           REQ    :=#authenticate
0010           ID     :=W#16#0001
0011           LEN    :=2
0012           DONE   :=#snd_done
0013           BUSY   :=#snd_busy
0014           ERROR  :=#snd_error
0015           STATUS :=
0016           DATA   :="buffers".snd
0017
0018        AN    #authenticate
0019        S     #authenticate
0020        JC    authenticate
0021
0022        A     #snd_done
0023        AN    #snd_error
0024        AN    #snd_busy
0025        JC    next_state
```

**Figure IV.8:** *Respond with no authentication necessary*

It follows a description of each state and its actions:

*bind:* On first start the program has to bind and listen to SOCKS port 1080. This is accomplished by using the system function *TCON* in passive mode. We stay in this state until a partner is connecting to this port.

*negotiate:* We wait until the client sends any message. This is done with the function *TRCV* which is enabled with the *EN_R* argument, see figure IV.7.

*authenticate:* After the first message we send a reply which indicates the client to perform no authentication. For this purpose we use the *TSEND* system function. In contrast to *TRCV* this function is edge controlled which means the parameter *REQ* has to change from *FALSE* to *TRUE* between consecutive calls to activate sending. As shown in figure IV.8 we toggle a flag and call *TSEND* twice with a rising edge on *REQ*.

*connect_request:* Then we expect the client to send a connection set up message containing target IP and port number which is stored for the next state.

*connect:* We set up the connection to the target with *TCON*.

*connect_confirm:* When the connection to the target is established, we send the confirmation message to the client.

*proxy:* Now we simply tunnel the connections between client and target. All data received from the client with *TRCV* is stored in a buffer which is reused to feed the *TSEND* function for sending data to the client. The same principle applies to the opposite direction, but we have to consider that sending messages can take a couple of cycles. Therefore a second buffer is used to ensure that no messages are mixed or lost. A disconnect is signaled with the error flag of *TRCV*. When this occurs we will send the last received data and then we go to the next state.

*reset:* In this state we close all connections with *TDISCON* and reset all persisted flags to its initial values.

## V. Evaluation

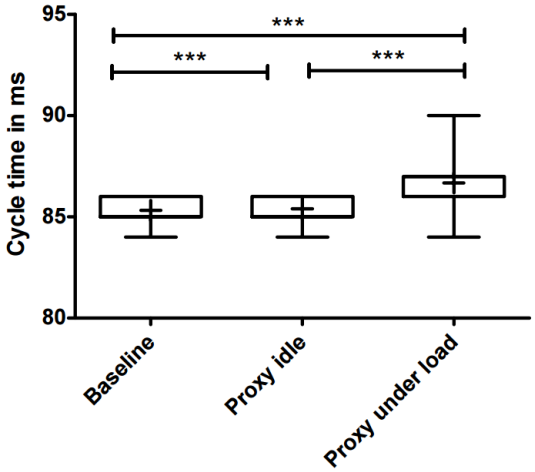



**Figure V.1:** *Shows the data distribution of the measured scan cycles for the three scenarios. Data are represented as boxplots with mean and were analyzed with the use of the Kruskal-Wallis test and the Dunn's Multiple Comparison Test. Significant differences are shown in the graph ($p < 0.0001$ = ***). All data were statistically analyzed with Prism software, version 5.0 (Graph Pad Inc).*

We analyzed the differences of the execution cycle times of the following scenarios: (a) a simple control program as a baseline, (b) its malicious version with the prepended SOCKS proxy in idle mode and (c) under load. Idle mode means that the proxy has been added to the control code but no proxy connection has been established. The *Baseline* program copies bytewise the input memory to the output memory 20 times which results in 81920 copy instructions. For the measurement, we added small code snippets which store the last cycle time in a data block. Siemens PLCs store the time of last execution cycle in a local variable of OB1 called *OB1_PREV_CYCLE*. We measured 2046 cycles in each scenario. All three scenarios do not exhibit normal distributions. We used the *Kruskal–Wallis* and the *Dunn's Multiple Comparison Test* for statistical significance analysis. The results are shown in Figure V.1. Execution time differed significantly in all three scenarios. Table V.1 shows the mean difference of the *Baseline* and the *Proxy under load* program, which is only 1.35 ms. The maximum transfer rate of the SOCKS proxy prepended to the *Baseline* program was about 40 KB/s. If the SOCKS proxy runs alone on the PLC it is able to transfer up to 730 KB/s. All network measurements have used a direct 100 Mbit/s Ethernet connection to the PLC. Finally, we tested the described attack cycle in our laboratory. In addition to regular traffic, we verified that we were able to tunnel an exploit for the DoS vulnerability *CVE-2015-2177* via the SOCKS tunnel using the *tsocks* library. The exploit worked as expected via the SOCKS tunnel.

**Table V.1:** *Statistical analysis of the three scenarios*

| | **Baseline** | **Proxy idle** | **Proxy under load** |
|---|---|---|---|
| Mean | 85.32 | 85.40 | 86.67 |
| Sdt. Deviation | 0.4927 | 0.5003 | 0.5239 |
| Sdt. Error | 0.01089 | 0.01106 | 0.01158 |

## VI. Discussion

Our attacks have limitations. In order to ensure that the PLC is always responsive, the execution time of the main program is monitored by a watchdog which kills the main program if the execution time becoes too long. The additional SNMP Scanner or Proxy code that we upload, together with the original programm, should not exceed the overall maximum execution time of 150 ms. An injection of the scanner or proxy is unlikely to trigger this timeout because the mean additional execution time of the proxy under load is 1.35 ms which is small compared to 150 ms. Furthermore, time-outs can be avoided by resetting the time counter after the execution of the injected program with the system function *RE_TRIGR* [21]. The easiest way to mitigate the described attack is to keep the PLC offline or to use a virtual private network (VPN) instead. If this is not possible protection-level 3 should be activated on the Siemens PLC. This enables a password-based read and write protection for the PLC. Without the right password the attacker can not modify the PLCs program. Based on our experience, this feature is rarely used in practice. Another applicable protection mechanism would be a firewall with deep packet inspection which is aware of industrial control protocols and thus can block potential malicious accesses such as attempts to reprogram the PLC.

## VII. Conclusion

We have shown a new threat vector that enables an external attacker to leverage a PLC as a SNMP scanner and network gateway to the internal production network. This makes it possible to access control systems behind an Internet-facing PLC. Our measurements indicate that the attack code, which runs de facto parallel to the original control program, causes a statistically significant but negligible increase of the execution cycle time. This makes a service disruption of the PLC unlikely and increases the chances that an attack remains undetected. Prior work on scanning the Internet for ICS only adressed risks due to control systems that are connected to the Internet directly. Our investigation shows that risks assessments must take PLCs into account that are connected only indirectly to the Internet. As a consequence, the target set of Internet-reachable industrial control systems is probably larger than expected and includes the "deep" industrial control network.

## References


[1] S. Heng, "Industry 4.0 upgrading of germany's industrial capabilities on the horizon," *Deutsche Bank Research*, 2014.

[2] NIST, "CVE-2014-2908," Apr. 2014. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-2908

[3] ——, "CVE-2014-2246," Mar. 2014. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-2246

[4] ——, "CVE-2012-3037," May 2012. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3037

[5] D. Beresford, "Exploiting Siemens Simatic S7 PLCs," *Black Hat USA*, 2011.

[6] N. Cybersecurity and C. I. C. (NCCIC), "Ics-cert monitor," Sep. 2014.

[7] Bundesamt für Sicherheit in der Informationstechnik, "Die Lage der IT-Sicherheit in Deutschland 2014," 2015.

[8] Industrial Control Systems Cyber Emergency Response Team, "Alert (ICS-ALERT-12-046-01A) Increasing Threat to Industrial Control Systems (Update A)," Available from ICS-CERT, ICS-ALERT-12-046-01A., Oct. 2012. [Online]. Available: https://ics-cert.us-cert.gov/alerts/ICS-ALERT-12-046-01A

[9] J.-O. Malchow and J. Klick, *Sicherheit in vernetzten Systemen: 21. DFN-Workshop*. Paulsen, C., 2014, ch. Erreichbarkeit von digitalen Steuergeräten - ein Lagebild, pp. C2–C19.

[10] B. Radvanovsky, "Project shine: 1,000,000 internet-connected scada and ics systems and counting," *Tofino Security*, 2013.

[11] R. Langner. (2011) A time bomb with fourteen bytes. [Online]. Available: http://www.langner.com/en/2011/07/21/a-time-bomb-with-fourteen-bytes/

[12] B. Meixell and E. Forner, "Out of Control: Demonstrating SCADA Exploitation," *Black Hat USA*, 2013.

[13] S. E. McLaughlin, "On dynamic malware payloads aimed at programmable logic controllers." in *HotSec*, 2011.

[14] S. McLaughlin and P. McDaniel, "Sabot: specification-based payload generation for programmable logic controllers," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 439–449.

[15] Wikipedia, "Automation Pyramid (content taken)." [Online]. Available: https://de.wikipedia.org/wiki/Automatisierungspyramide

[16] Siemens, "S7 314C-2PN/DP Technical Details." [Online]. Available: https://support.industry.siemens.com/cs/pd/495261?pdti=td&pnid=13754&lc=de-WW

[17] ——, "S7-300 CPU 31xC and CPU 31x: Technical specifications." [Online]. Available: https://cache.industry.siemens.com/dl/files/906/12996906/att_70325/v1/s7300_cpu_31xc_and_cpu_31x_manual_en-US_en-US.pdf

[18] ——. (2011) S7-300 Instruction list S7-300 CPUs and ET 200 CPUs . [Online]. Available: https://cache.industry.siemens.com/dl/files/679/31977679/att_81622/v1/s7300_parameter_manual_en-US_en-US.pdf

[19] SNAP7, "S7 Protocol." [Online]. Available: http://snap7.sourceforge.net/siemens_comm.html#s7_protocol

[20] J. Kühner, "DotNetSiemensPLCToolBoxLibrary." [Online]. Available: https://github.com/jogibear9988/DotNetSiemensPLCToolBoxLibrary

[21] Siemens. (2006) System Software for S7-300/400 System and Standard Functions Volume 1/2. [Online]. Available: https://cache.industry.siemens.com/dl/files/574/1214574/att_44504/v1/SFC_e.pdf

[22] D. Marzin, S. Lau, and J. Klick, "PLCinject Tool." [Online]. Available: https://github.com/SCADACS/PLCinject

[23] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)," RFC 1157 (Historic), Internet Engineering Task Force, May 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1157.txt

[24] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "SOCKS Protocol Version 5," RFC 1928 (Proposed Standard), Internet Engineering Task Force, Mar. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1928.txt