Kaspersky ICS CERT

kaspersky

Security research: CODESYS Runtime, a PLC control framework

Alexander Nochvay

18.09.2019 Version 1.0

Contents

Research on the security of technologies used by automation system developers that can potentially be applied at industrial facilities across the globe is a high-priority area of work for the Kaspersky Industrial Systems Emergency Response Team (Kaspersky ICS CERT).

This article continues the discussion of research on popular OEM technologies that are implemented in the products of a large number of vendors. Vulnerabilities in such technologies are highly likely to affect the security of many, if not all, products that use them. In some cases, this means hundreds of products that are used in industrial environments and in critical infrastructure facilities. This is the case with CODESYS Runtime®, a framework by CODESYS designed for developing and executing industrial control system software. The trademark rights are held by 3S-Smart Software Solutions – a member of the CODESYS Group.

According to the developer's official information, CODESYS Runtime has already been adapted for more than 350 devices from different vendors, used in the energy sector, industrial manufacturing, internet of things and industrial internet of things systems, etc. It should also be noted that the actual adoption figures are much higher, since many vendors' PLCs that use the CODESYS Runtime framework are missing from the official list. The number of such devices continues to grow: there were only 140 of them in 2016. We won't be surprised if this trend continues into the future.

Fragments of technical information were removed at the request of the CODESYS Group. Request more information from security@codesys.com.

The framework

Today, the use of ready-made software code in a product is a rule rather than an exception. This enables the developers of a new product to avoid 'reinventing the wheel', helping reduce development time.

The degree to which third-party code affects a software product that incorporates it, and the degree to which it affects the security of a system where that product us used, can vary.

Third-party code is often used to implement a specific function or set of functions, such as rendering images or the user interface, sending files to the printer or saving data in a database. We have conducted security research and identified vulnerabilities in third-party code before. For example, in 2017, in part of SafeNet Sentinel, a hardware-based solution designed to control licensing agreement compliance and protect applications from being 'cracked', and in 2018, in the OPC UA library by OPC Foundation.

The situation with the CODESYS framework is different: vendors of CODESYS-based PLCs adapt the framework for their hardware and, if necessary, develop additional modules using services provided by CODESYS. PLC end-users (i.e., engineers) use the CODESYS development environment to develop the code of industrial process automation programs. And the execution flow of the additional modules developed and the industrial process automation program is controlled on PLCs by the versions of CODESYS Runtime adapted for those PLCs.

The fact that the framework controls the execution flow of the program means that using the framework imposes restrictions at the architecture level – at the product design stage. In other words, the framework is a sophisticated mechanism that is already in place, and the user's code must be designed to be a cog in that mechanism.

In terms of ensuring security when using a framework, the developer must address the following questions:

- What's inside the framework?
- How does it work?
- How do I make my software secure if there is a vulnerability in the framework, rather than in my code?

This paper is devoted to research on the security of CODESYS Runtime. In it, we address the first two of the above questions: what happens inside the framework and how it works. We also demonstrate a technique for identifying vulnerabilities without being able to analyze the source code.

CODESYS Runtime: description of the object of research

Before discussing the results of research on an object's security, it is essential to clarify what that object is. We developed the technical description of CODESYS Runtime provided in this chapter in the process of analyzing the framework.

Product bundle offered by the CODESYS Group

The CODESYS Group develops two main products:

1. CODESYS Development System, a development environment;
2. CODESYS Runtime, an execution environment.

The two products work together. The CODESYS Development System is an IDE used to develop software for controlling devices on which CODESYS Runtime runs. The development environment includes numerous tools designed to simplify the development and testing process.

In the context of our research, it is important that the CODESYS Development System is a customizable development environment. Solutions based on it include IDE SoMachine by Schneider Electric, TwinCAT by Beckhoff Automation, IdraWorks by Bosch, Wagilo Pro by WAGO, IDEs under the name of CODESYS Development System by Owen, STW Technic, and prolog-plc, as well as other IDEs.

To program a controller using the CODESYS Development System IDE, CODESYS Runtime should be running on the controller. For CODESYS Runtime to run correctly on a specific device, it has to be adapted to the operating system and hardware selected. According to information on the CODESYS official website, CODESYS developers themselves have only adapted CODESYS Runtime for 15 devices. However, distributors have adapted CODESYS Runtime for over 350 devices.

CODESYS Runtime adaptations include versions for:

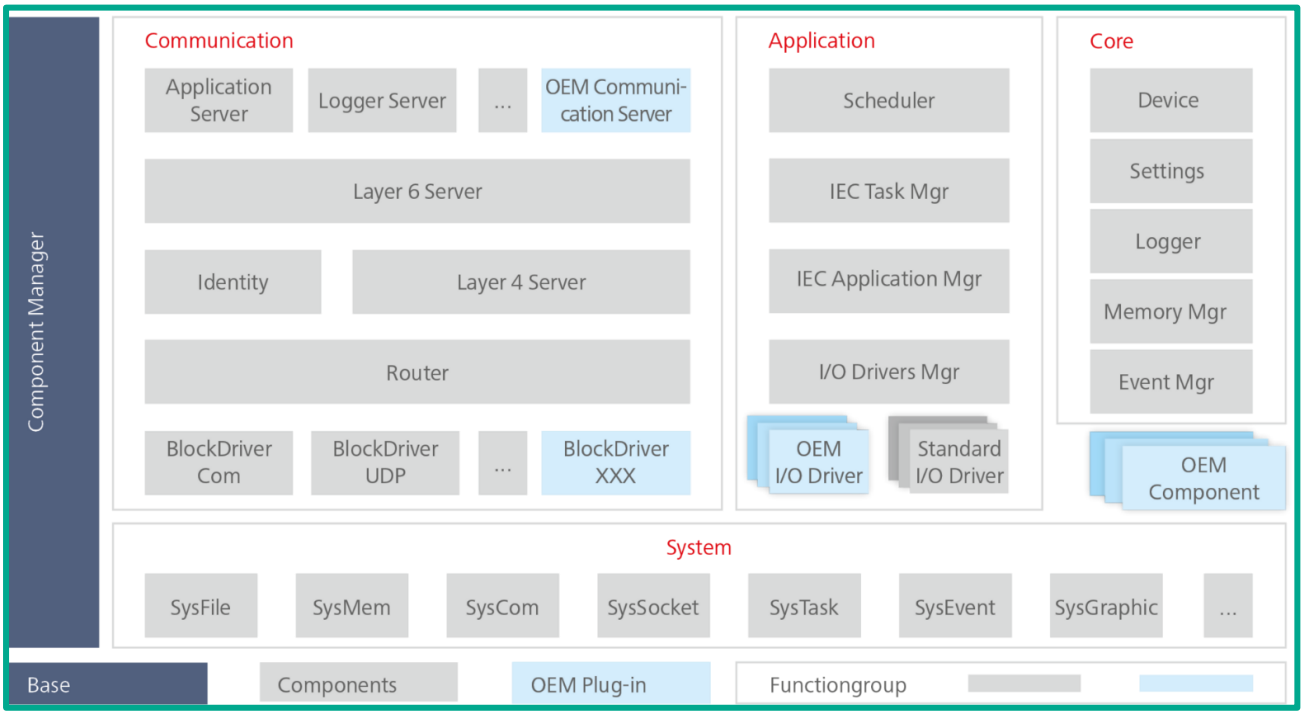
- Single-board Linux-based computers, such as Raspberry Pi, UniPi, and BeagleBone;
- Windows and Linux based Soft PLC installations;
- PLCs by ASEM S.p.A, exceet electronics AG, Hitachi Europe GmbH, Hans Turck GmbH & Co. KG, elrest Automationssysteme GmbH, Janz Tec AG, Kendrion Kuhnke Automation GmbH, Beijer Electronics, ifm electronic gmbh, Nidec Control Techniques Limited, Advantech Europe B.V, WAGO Kontakttechnik GmbH & Co. KG, KEB Automation KG, Berghof Automation GmbH, and many other vendors.

Architecture

Components

CODESYS Runtime is based on a component-oriented architecture. This means that each logical or functional part of CODESYS Runtime is divided into one or more components or modules.

Each component is responsible for a specific task in a specific functional area, such as logging, network communication, communication over serial cables, core load balancing, program debugging, etc.



CODESYS Runtime component-oriented architecture
Source: https://www.codesys.com/fileadmin/data/Downloads/Broschueren/CODESYS-Runtime-en.pdf

The following modules can be identified as the main CODESYS Runtime components:

1. **Component Manager or CM** – the component that launches and initializes all other components on a system;
2. **System Components** – the group of components that define communication with the operating system and the hardware. Components in this group are responsible for communicating with physical ports and with the file system, for dynamic and static memory allocation, etc.
3. **Communication Components** – the group of components for communicating with the outside world, e.g., over the network or serial cables;
4. **Application components** – components responsible for controlling the PLC program;
5. **Core components** – components for controlling the PLC and its state.

A developer has several ways of extending CODESYS Runtime:

1. Replacing existing components;
2. Writing custom components;
3. Write custom components designed to extend the functionality of existing components.

Component structure

CODESYS components are dynamic libraries (the equivalent of *.dll in Windows and *.so in Linux). All components are loaded by the **Component Manager** component.

CODESYS Runtime can be built either statically or dynamically.

If CODESYS Runtime is built statically, the components' code is contained in the executable file itself.

If CODESYS Runtime is built dynamically, a list of components to be loaded is specified in the configuration file and the component files are located separately from the executable file.

The structure of a component file was not an object of this study, but it can be said with confidence that the structure includes:

- The component's program code;
- The component's name, the author's name, component version and description;
- Various checksums and magic numbers.

Structure of communication interfaces

For a researcher, the structure of component files is not as interesting as how CODESYS Runtime communicates with external components and how internal components communicate with each other.

Each component must include the following functions: initialization function, export function, import function, event handling function, and a function to create and remove its instance. A component must also have a unique numeric identifier.

The functions that delete and create a component's instance are optional. They turned out to be empty for most components. For this reason, they will not be covered by this paper. The remaining functions are discussed below.

Implementation of the initialization function

The initialization function is analogous to an entry point for PE and ELF files; the only difference is that it does not start the component's actual operation. The function is called by the Component Manager.

```
01 Removed at the vendor's request
02: {
03:   Removed at the vendor's request
04:   Removed at the vendor's request
05:   Removed at the vendor's request
06:   Removed at the vendor's request
07:   Removed at the vendor's request
08:   Removed at the vendor's request
09:   Removed at the vendor's request
[...]
17:   Removed at the vendor's request
18: }
```

Decompiled code of the inutialization function of CmpBlkDrvUdp component from the Communication group

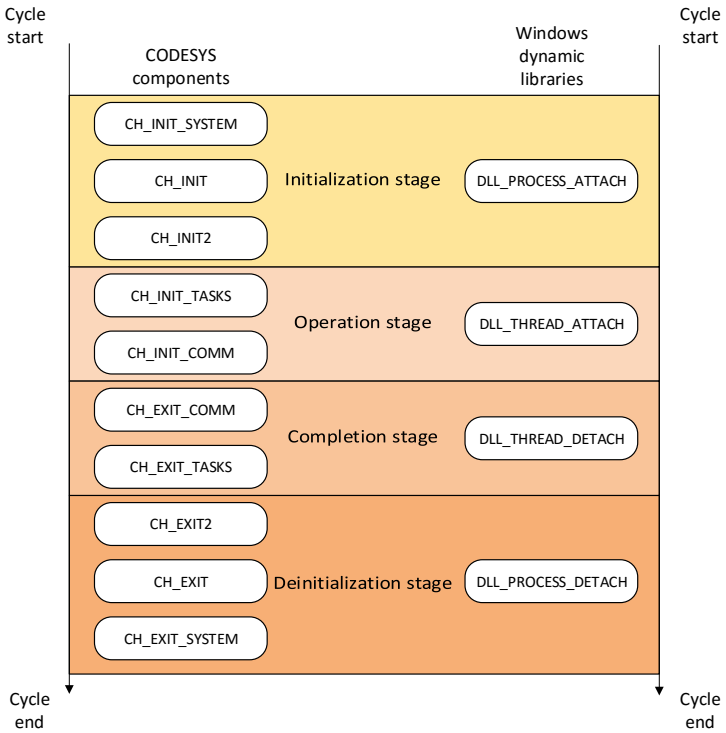
The function **ModuleCmpBlkDrvUdp_entry** is an initialization function. The function takes the structure INIT_STRUCT as an argument. The function is usually called using the **Component Manager** to populate the structure. The initialization function populates all the fields in the structure, including all the functions mentioned above and the component identifier, which is equal to 7 for the CmpBlkDrvUdp component.

Implementation of the event handling function

The next function of interest is the event handling function. For the CmpBlkDrvUdp component, this is the **ModuleCmpBlkDrvUdp_hook** function**.** It determines what the **Component Manager** requires it to do based on the event ID received.

Main event identifiers:

- CH_INIT_SYSTEM – ID 1. If a component is in the **System Components** group, it must be initialized;
- CH_INIT – ID 2. Components must initialize all local variables;
- CH_INIT2 – ID 3. The component must initialize;
- CH_INIT_TASKS – ID 5. The component can execute its threads;
- CH_INIT_COMM – ID 6. The component can start communication;
- CH_EXIT_COMM – ID 10. The component must close all communication channels;
- CH_EXIT_TASKS – ID 11. The component must stop and terminate all threads of execution created by it;
- CH_EXIT2 – ID 13. The component must save all data before calling CH_EXIT;
- CH_EXIT – ID 14. The component must release memory;
- CH_EXIT_SYSTEM – ID 15. If the component is in the **System Components** group, it must release memory;
- CH_COMM_CYCLE – ID 20. It is called in every cycle and is used for the execution threads created.



CODESYS component lifecycle compared to Windows Dynamic Link Library lifecycle

Like events created when calling a DLL in Windows, events handled by a CODESYS component are created in ‘mirror’ pairs: [CH_INIT_SYSTEM – CH_EXIT_SYSTEM], [CH_INIT – CH_EXIT], etc.

```
001: Removed at the vendor’s request
002: {
003:   Removed at the vendor’s request
004:   Removed at the vendor’s request
005:   Removed at the vendor’s request
006:   Removed at the vendor’s request
007:   Removed at the vendor’s request
008:
009:   Removed at the vendor’s request
010:   {
011:     Removed at the vendor’s request : // Initialization of the component’s variables
012:       Removed at the vendor’s request
013:       Removed at the vendor’s request
014:       Removed at the vendor’s request
015:       Removed at the vendor’s request
016:       Removed at the vendor’s request
017:       Removed at the vendor’s request
018:       Removed at the vendor’s request
019:       Removed at the vendor’s request
020:       Removed at the vendor’s request
021:       Removed at the vendor’s request
022:       Removed at the vendor’s request
023:       Removed at the vendor’s request
024:       Removed at the vendor’s request
025:     Removed at the vendor’s request : // Initialization of the component. Setting values from the
configuration file
026:       Removed at the vendor’s request
027:       Removed at the vendor’s request
028:       Removed at the vendor’s request
029:       Removed at the vendor’s request
030:         Removed at the vendor’s request
031:       Removed at the vendor’s request
032:       Removed at the vendor’s request
033:       Removed at the vendor’s request
034:       {
035:         iRemoved at the vendor’s request
036:           Removed at the vendor’s request
037:         Removed at the vendor’s request
038:           Removed at the vendor’s request
039:         Removed at the vendor’s request
040:       }
041:       Removed at the vendor’s request
042:       {
043:         Removed at the vendor’s request
044:         Removed at the vendor’s request
045:         {
046:           Removed at the vendor’s request
047:           Removed at the vendor’s request
048:             Removed at the vendor’s request
049:           Removed at the vendor’s request
050:           Removed at the vendor’s request
051:           Removed at the vendor’s request
052:         }
053:       }
054:       Removed at the vendor’s request
055:     Removed at the vendor’s request  // Creating a communication thread and starting communication
056:       Removed at the vendor’s request
057:       Removed at the vendor’s request
058:       {
059:         Removed at the vendor’s request
060:           Removed at the vendor’s request
061:         Removed at the vendor’s request
062:       }
063:       Removed at the vendor’s request
064:     Removed at the vendor’s request : // Completing communication
065:       Removed at the vendor’s request
066:       Removed at the vendor’s request
```

```
067:        {
068:          Removed at the vendor's request
069:          Removed at the vendor's request
070:          Removed at the vendor's request
071:        }
072:        Removed at the vendor's request
073:      Removed at the vendor's request:  // Saving data and releasing it
074:        Removed at the vendor's request
075:        {
076:          Removed at the vendor's request
077:          {
078:            Removed at the vendor's request
079:              Removed at the vendor's request
080:            Removed at the vendor's request
081:          }
082:          Removed at the vendor's request
083:          Removed at the vendor's request
084:        }
085:        Removed at the vendor's request
086:        {
087:          Removed at the vendor's request
088:          Removed at the vendor's request
089:        }
090:        Removed at the vendor's request
091:      Removed at the vendor's request  // Updating the communication socket
092:        Removed at the vendor's request
093:        {
094:          Removed at the vendor's request
095:          Removed at the vendor's request
096:          {
097:            Removed at the vendor's request
098:            Removed at the vendor's request
099:          }
100:          Removed at the vendor's request
101:        }
102:        Removed at the vendor's request
103:      Removed at the vendor's request :
104:        Removed at the vendor's request
105:    }
106: }
```

Decompiled code of the ModuleCmpBlkDrvUdp_hook function, demomstrating the handling of event by the CmpBlkDrvUdp component

Using the function **ModuleCmpBlkDrvUdp_hook** as an example, we can see the following:

- Components can ignore prescribed event handling rules and use one handler to handle several different events, as implemented in the function used in this example: when handling the event **CH_INIT_COMM**, a thread is both initialized and executed, although the event **CH_INIT_TASKS** is designed to perform the latter task;
- Components do not necessarily have to handle all events. Specifically, components do not have to handle both 'symmetrical' events if one of the two 'mirror' events has already been handled. For example, the component **CmpBlkDrvUdp** does not handle the event **CH_EXIT,** although it handled the event **CH_INIT**.

Implementation of import and export functions

The export function and the import function use a mechanism that provides the same overall capabilities as exported and imported functions in Windows and Linux dynamic libraries. The main way in which import and export functions of CODESYS components are different from Windows and Linux libraries is that these functions register exported functions and initialize pointers that point to imported functions.

```
01: Removed at the vendor's request
02: {
03:     Removed at the vendor's request
04:     Removed at the vendor's request
05:     Removed at the vendor's request
06:     Removed at the vendor's request
07: }
08:
09: Removed at the vendor's request
10: Removed at the vendor's request
11: Removed at the vendor's request
12: Removed at the vendor's request
13:
14: Removed at the vendor's request
15: {
16:   Removed at the vendor's request
17:
18:   Removed at the vendor's request
19:
20:   Removed at the vendor's request
21: }
22:
```

Pseudocode for the export function of the CmpRasPi component (available only in CODESYS Runtime for Raspberry PI)

The function **CMRegisterAPI** takes a pointer which points to an array populated with exported functions as its first argument and the component identifier as its last argument. Here is an example of an exported function: line 10 of the code fragment shown above contains the structure exported_function populated with the following values: pointer which points to the function sub_84e5bc0, function name "raspiyuv", hash 0xF81Fd05, and version 0x3050400.

Thus, the CmpRasPi component provides all other CODESYS components and application programs with an API that enables them to communicate with the camera module on the Raspberry PI device. An example of the use of the API is shown below in the sample project Camera.project for CODESYS Control for Raspberry Pi.

```
1: PROGRAM PLC_PRG
2: VAR
3:   xTakePicture: BOOL;
4: END_VAR
5:
6: IF xTakePicture THEN
7:   Raspberry_Pi_Camera.Still('-o Picture.jpg');
8:   xTakePicture := FALSE;
9: END_IF
```

Camera.project program code

Each component's import function attempts to find functions exported by other components and record their addresses.

```
01: Removed at the vendor's request
02: {
03:   Removed at the vendor's request
04:   Removed at the vendor's request
05:   Removed at the vendor's request
06:   Removed at the vendor's request
07:   Removed at the vendor's request
08:   Removed at the vendor's request
09:   Removed at the vendor's request
10:   Removed at the vendor's request
11: [...]
12:
```

Import function in the CmpApp module

The function CMGetAPI2 looks for a function that was registered by another component. The first argument is the function's name, the second is the value in which to save the function pointer obtained, the third is the expected hash of the function, if the hash is passed, and the last argument is the expected version.

Prior to this, all these functions were registered by the SysTarget component.

```
01: Removed at the vendor's request
02: Removed at the vendor's request
03: Removed at the vendor's request
04: Removed at the vendor's request
05: Removed at the vendor's request
06: Removed at the vendor's request
07: Removed at the vendor's request
08: Removed at the vendor's request
09: Removed at the vendor's request
10: Removed at the vendor's request
11: Removed at the vendor's request
12: Removed at the vendor's request
13: Removed at the vendor's request
14: Removed at the vendor's request
15: Removed at the vendor's request
16: Removed at the vendor's request
```

Fragment of the exported functions array

The mechanism of importing and exporting functions provides developers with core functionality for creating their own components or extending the capabilities of existing components.

Component configuration

Since the component configuration mechanism demonstrates how part of the CODESYS Runtime architecture operates, it is discussed here as a conclusion to the chapter on the architecture of CODEYS Runtime.

A CODESYS Runtime user can control components via an .ini configuration file. An .ini configuration file is a text file containing keys and parameters used to configure components.

```
[...]
28: [CmpWebServer]
29: ConnectionType=0
30:
31: [CmpOpenSSL]
[...]
```

Fragment of configuration file for CODESYS Control for Raspberry Pi

The **Component Manager** initializes all components. System components, such as CmpMemPool, CmpLog, CmpSettings, SysFile, etc., are the first to be initialized.

```
********* CoDeSysControl DEMO VERSION - runs 2 hours*********
[...]
======================================================================
1526222855: Cmp=CM, Class=1, Error=0, Info=4, pszInfo= CODESYS Control V3
1526222855: Cmp=CM, Class=1, Error=0, Info=5, pszInfo= Copyright (c) 3S - Smart Software Solutions GmbH
1526222855: Cmp=CM, Class=1, Error=0, Info=6, pszInfo= <version>3.5.12.0</version> <builddate>Dec 18
2017</builddate>
======================================================================
1526222855: Cmp=CM, Class=1, Error=0, Info=10, pszInfo= System: <cmp>CM</cmp>, <id>0x00000001</id>
<ver>3.5.12.0</ver>
1526222855: Cmp=CM, Class=1, Error=0, Info=10, pszInfo= System: <cmp>CmpMemPool</cmp>, <id>0x0000001e</id>
<ver>3.5.12.0</ver>
1526222855: Cmp=CM, Class=1, Error=0, Info=10, pszInfo= System: <cmp>CmpLog</cmp>, <id>0x00000013</id>
<ver>3.5.12.0</ver>
```

```
1526222855: Cmp=CM, Class=1, Error=0, Info=10, pszInfo= System: <cmp>CmpSettings</cmp>,
<id>0x0000001a</id> <ver>3.5.12.0</ver>
1526222855: Cmp=CM, Class=1, Error=0, Info=10, pszInfo= System: <cmp>SysFile</cmp>, <id>0x00000104</id>
<ver>3.5.12.0</ver>
1526222855: Cmp=CM, Class=1, Error=0, Info=10, pszInfo= System: <cmp>SysTimer</cmp>, <id>0x00000116</id>
<ver>3.5.12.0</ver>
1526222855: Cmp=CM, Class=1, Error=0, Info=10, pszInfo= System: <cmp>SysTimeRtc</cmp>, <id>0x00000127</id>
<ver>3.5.12.0</ver>
[...]
```

Fragment of CODESYS Control for Raspberry Pi startup log

One of the system components, CmpSettings, is of interest because its export function registers APIs that are used by all other components to get parameters from the configuration file.

```
01: Removed at the vendor's request
02: Removed at the vendor's request
03: Removed at the vendor's request
04: Removed at the vendor's request
05: Removed at the vendor's request
06: Removed at the vendor's request
07: Removed at the vendor's request
08: Removed at the vendor's request
09: Removed at the vendor's request
10: Removed at the vendor's request
11: Removed at the vendor's request
12: Removed at the vendor's request
13: Removed at the vendor's request
14: Removed at the vendor's request
15: Removed at the vendor's request
```

Fragment of the exported functions array of the CmpSettings component

The functions **SettgGetIntValue** and **SettgGetStringValue** are used by most components to determine their operating parameters. Using cross-references from the calls of these functions, it can be determined which components can be configured via the configuration file and which keys should be included in the configuration file.

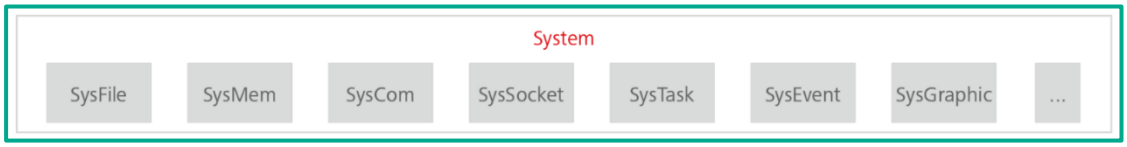
By searching for calls of the SettgGetIntValue function using cross-references, it is possible to find the key DemoTimeUnlimited for configuring the ComponentManager component:

```
01: Removed at the vendor's request
02: {
03:   Removed at the vendor's request
04:
05:   Removed at the vendor's request
06:   Removed at the vendor's request
07:   Removed at the vendor's request
08:   Removed at the vendor's request
09:   Removed at the vendor's request
10:   Removed at the vendor's request
11:   Removed at the vendor's request
12:     Removed at the vendor's request
13:   Removed at the vendor's request
14: }
```

Configuration keys for the ComponentManager component

Adaptation

Support for adapting CODESYS Runtime for any hardware and operating system is certainly its main feature. Developers of products that use the framework are responsible for adapting CODESYS Runtime to the needs and requirements of the specific application, including the industrial process type. The adapted CODESYS Runtime framework should be able to communicate with hardware interfaces and the Ethernet, release and allocate memory, work with the timer, events, inter-thread communication, etc.



System components from the CODESYS Runtime component-oriented architecture

The adaptation of system components is performed by exporting functions required by other components (this process was described in the previous chapter).

Upon analyzing several variants of CODESYS Runtime, we determined that there are a total of 25 system components. The main system components are listed below:

```
SysTimer, SysTimeRtc, SysTime, SysTask, SysTarget, SysSocket, SysShm, SysSemProcess,
SysSemCount, SysSem, SysReadWriteLock, SysProcess, SysOut, SysMutex, SysMsgQ,
SysModule, SysMem, SysInternalLib, SysFile, SysExcept, SysEvent, SysEthernet, SysDir,
SysCpuHandling, SysCom
```

The main system components

After ensuring that the system components operate properly, the developer should create custom CODESYS Runtime modules for PLCs with the specific functionality required.

Implementation

The first version of CODESYS Control for Raspberry Pi was released in December 2016. In June 2018, a version for Linux (CODESYS Control for Linux SL) was released. There is also a CODESYS Control emulator for Windows, which is part of the CODESYS Development System software package. All these implementations are analogous to the CODESYS Control for Raspberry Pi implementation and have similar or identical implementation elements.

In this chapter, we discuss the implementation of CODESYS Runtime using CODESYS Control for Raspberry Pi and CODESYS Control for Linux SL as examples.

Installer file

The CODESYS Development System transfers a CODESYS Control installer to the Raspberry Pi device using the SSH client. The installer is a .deb (Debian binary package) file.

```
# dpkg -c codesyscontrol_arm_raspberry_V3.5.12.0.deb
drwxr-xr-x root/root         0 2017-12-18 09:22 ./
drwxr-xr-x root/root         0 2017-12-18 09:22 ./var/
drwxr-xr-x root/root         0 2017-12-18 09:22 ./var/opt/
drwxr-xr-x root/root         0 2017-12-18 09:22 ./var/opt/codesys/
drwxr-xr-x root/root         0 2017-12-18 09:22 ./var/opt/codesys/backup/
drwxr-xr-x root/root         0 2017-12-18 09:22 ./var/opt/codesys/cmact_licenses/
-rwxr-xr-x root/root      2640 2017-12-18 09:22 ./var/opt/codesys/bacstacd.ini
-rw-r--r-- root/root     20736 2017-12-18 09:22 ./var/opt/codesys/3SLicense.wbb
drwxr-xr-x root/root         0 2017-12-18 09:22 ./var/opt/codesys/restore/
drwxr-xr-x root/root         0 2017-10-09 14:16 ./etc/
-rw-r--r-- root/root       216 2017-10-09 14:16 ./etc/CODESYSControl_User.cfg
drwxr-xr-x root/root         0 2017-12-18 09:22 ./etc/init.d/
-rw-r--r-- root/root      3355 2017-12-18 09:22 ./etc/init.d/codesyscontrol
-rw-r--r-- root/root       158 2017-10-09 14:16 ./etc/3S.dat
-rw-r--r-- root/root       943 2017-10-09 14:16 ./etc/CODESYSControl.cfg
drwxr-xr-x root/root         0 2017-12-18 09:22 ./opt/
drwxr-xr-x root/root         0 2017-12-18 09:22 ./opt/codesys/
drwxr-xr-x root/root         0 2017-12-18 09:22 ./opt/codesys/bin/
-rwxr-xr-x root/root   7330296 2017-12-18 09:22 ./opt/codesys/bin/codesyscontrol.bin
drwxr-xr-x root/root         0 2017-12-18 09:22 ./opt/codesys/scripts/
```

Contents of the .deb file

The main elements of the .deb file are the configuration file and the executable file.

Configuration file

The configuration file includes a huge number of different configuration parameters for CODESYS Control. The following conclusions can be made based on the contents of the file:

- CODESYS Control for Raspberry Pi can work as a web server;
- CODESYS Control for Raspberry Pi uses OpenSSL;
- There are logging parameters for the CmpLog component;
- Parameters of the CmpSettings component can include references to other files;
- For the SysProcess component, there is the key Command.%d, the value of whose parameter is the same as the name of the shutdown system utility in Linux OS.

```
01: # cat etc/CODESYSControl_User.cfg
02: [SysCom]
03: ;Linux.Devicefile=/dev/ttyS
04:
05: [CmpBlkDrvCom]
06: ;Com.0.Name=MyCom
07: ;Com.0.Baudrate=115200
08: ;Com.0.Port=3
09: ;Com.0.EnableAutoAddressing=1
10:
11: [SysProcess]
12: Command.0=shutdown
13:
14: [CmpApp]
15: Bootproject.RetainMismatch.Init=1
01: # cat etc/CODESYSControl.cfg
02: [SysFile]
03: FilePath.1=/etc/, 3S.dat
04: PlcLogicPrefix=0
05:
06: [CmpLog]
07: Logger.0.Name=/tmp/codesyscontrol.log
08: Logger.0.Filter=0x0000000F
09: Logger.0.Enable=1
10: Logger.0.MaxEntries=1000
11: Logger.0.MaxFileSize=1000000
12: Logger.0.MaxFiles=1
13: Logger.0.Backend.0.ClassId=0x00000104 ;writes logger messages in a file
14: Logger.0.Type=0x314 ;Set the timestamp to RTC
15:
16: [CmpSettings]
17: FileReference.0=SysFileMap.cfg, SysFileMap
18: FileReference.1=/etc/CODESYSControl_User.cfg
19:
20: [SysExcept]
21: Linux.DisableFpuOverflowException=1
22: Linux.DisableFpuUnderflowException=1
23: Linux.DisableFpuInvalidOperationException=1
24:
25: [CmpBACnet]
26: IniFile=bacstacd.ini
27:
28: [CmpWebServer]
29: ConnectionType=0
30:
31: [CmpOpenSSL]
32: WebServer.Cert=server.cer
33: WebServer.PrivateKey=server.key
34: WebServer.CipherList=HIGH
35:
36: [SysMem]
37: Linux.Memlock=0
38:
39: [CmpCodeMeter]
40: InitLicenseFile.0=3SLicense.wbb
41:
```

```
42: [SysEthernet]
43: Linux.ProtocolFilter=3
44:
45: [CmpSchedule]
46: ProcessorLoad.Enable=1
47: ProcessorLoad.Maximum=95
48: ProcessorLoad.Interval=5000
```

Contents of the configuration file for CODESYS Control For Raspberry Pi v3.5.14.10

Executable file

File protection parameters

An initial analysis of executable files usually includes checking the compilation options for security parameter settings. The checksec tool shows the following security parameter values for executable files.

```
# ./checksec.sh/checksec -o csv -f codesyscontrol_armv6l_raspberry.bin
No RELRO,No Canary found,NX disabled,No PIE,RPATH,No RUNPATH,No SYMTABLES,No
Fortify,0,23,codesyscontrol_armv6l_raspberry.bin

# ./checksec.sh/checksec -o csv -f codesyscontrol_armv7l_raspberry.bin
No RELRO,No Canary found,NX disabled,No PIE,RPATH,No RUNPATH,No SYMTABLES,No
Foritfy,0,23,codesyscontrol_armv7l_raspberry.bin
```

Result of checking CODESYS Control for Raspberry Pi v3.5.14.10 executable files with the checksec utility

Results produced by the utility demonstrate that the executable files of CODESYS Control for Raspberry Pi v3.5.14.10 were compiled without additional protection that might make exploiting binary vulnerabilities more difficult.

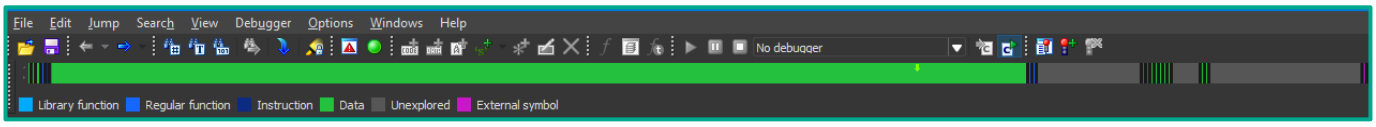
The situation with the compilation of the CODESYS Control for Linux SL file is slightly better, because the file has the option PIE enabled.

```
# ./checksec.sh/checksec -o csv -f codesyscontrol.bin
Partial RELRO,No Canary found,NX disabled,PIE enabled,No RPATH,No RUNPATH,No
SYMTABLES,No Fortify,0,23,codesyscontrol.bin
```

Result of running the checksec utility on the executable files of CODESYS Control for Linux SL v3.5.14.10

The state of the executable file

Static analysis of the executable file using the IDA Pro tool shows that 99% of the file is data (shown in green) rather than machine code:



State of the packed executable file of CODESYS Runtime For Raspberry Pi

This state is typical of executable files whose machine code is packed. However, all executable files must have an entry point. For the executable file of CODESYS Runtime for Raspberry Pi, the entry point is the start function, so this function can be the one with which to start an analysis.

```
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
```

```
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
Removed at the vendor's request
```

Assembler code of the start function

The **start** function's code is recognized normally and IDA Pro suggests that the function **sub_86a0840**, a.k.a. the **main** function, also contains valid program code.

```
1: Removed at the vendor's request
2: {
3:   Removed at the vendor's request
4:   Removed at the vendor's request
5:   Removed at the vendor's request
6:   Removed at the vendor's request
7:   Removed at the vendor's request
8: }
```

Decompiled pseudocode of the function sub_86a0840

The **main** function stores the number of command-line arguments used and a pointer pointing to their values in global variables (lines 3:4). Next, it calls the **mprotect** function (line 5), which changes the access parameters for the memory area. The first argument is a pointer pointing to the **start** function, which is also the beginning of the .text segment. The second argument is the size of the memory whose access parameter will be changed. The memory size should also point to the end of the segment. The last argument is the memory access parameters replacing the original parameters. It is equal to 7, i.e., the sum of the values of the parameters PROT_READ | PROT_WRITE | PROT_EXEC.

In other words, line 5 prepares a memory area in which program code is to be unpacked and executed. After that, the next function is called (line 6) and a pointer to the memory area in which the variable **dword_86A0460** is stored is passed to it as an argument. The pointer points to the original main function after it has been unpacked.

Thus, for the file to be further analyzed, it needs to be unpacked.

Running process

CODESYS Runtime for Raspberry Pi and for Linux traces its process, i.e., CODESYS Runtime for Raspberry Pi and For Linux debugs itself. This mechanism is used for two purposes: to intercept system calls (syscalls) and to implement primitive anti-debugging protection: it is impossible to connect to a running CODESYS Runtime process using third-party debugging tools, such as gdb, IDA Pro, radare, or strace.

Tracing

```
01: # strace -f ./codesyscontrol.bin
02: execve("./codesyscontrol.bin", ["./codesyscontrol.bin"], [/* 18 vars */]) = 0
03: brk(NULL)                               = 0x90ce000
04: uname({sysname="Linux", nodename="raspberrypi", ...}) = 0
05: [...]
06: mprotect(0x8050000, 6495192, PROT_READ|PROT_WRITE|PROT_EXEC) = 0
07: cacheflush(0x8050000, 0x8681bd8, 0, 0x8681bd8, 0x8681828) = 0
08: open("/home/pi/", O_RDONLY)             = 3
09: rt_sigaction(SIGTERM, {sa_handler=0x8050a40, sa_mask=[], sa_flags=SA_RESTORER,
sa_restorer=0x76d436b0}, NULL, 8) = 0
10: rt_sigaction(SIGINT, {sa_handler=0x8050a40, sa_mask=[], sa_flags=SA_RESTORER, sa_restorer=0x76d436b0},
NULL, 8) = 0
11: rt_sigaction(SIGPIPE, {sa_handler=SIG_IGN, sa_mask=[], sa_flags=SA_RESTORER, sa_restorer=0x76d436b0},
NULL, 8) = 0
12: rt_sigaction(SIGABRT, {sa_handler=SIG_IGN, sa_mask=[], sa_flags=SA_RESTORER, sa_restorer=0x76d436b0},
NULL, 8) = 0
13: clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x76faabf8) = 4290
14: strace: Process 4290 attached
15: [pid  4290] set_robust_list(0x76faac00, 12) = 0
16: [pid  4290] getppid( <unfinished ...>
17: [pid  4289] ptrace(PTRACE_CONT, 4290, NULL, SIG_0 <unfinished ...>
18: [pid  4290] <... getppid resumed> )      = 4289
19: [pid  4290] getsid(4289)                 = 3663
20: [pid  4290] ptrace(PTRACE_TRACEME)       = -1 EPERM (Operation not permitted)
21: [pid  4290] getpid()                     = 4290
22: [pid  4290] kill(4290, SIGKILL)          = ?
23: [pid  4289] <... ptrace resumed> )       = -1 ESRCH (No such process)
24: [pid  4289] wait4(-1,  <unfinished ...>
25: [pid  4290] +++ killed by SIGKILL +++
26: <... wait4 resumed> [{WIFSIGNALED(s) && WTERMSIG(s) == SIGKILL}], 0, NULL) = 4290
27: --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_KILLED, si_pid=4290, si_uid=0, si_status=SIGKILL,
si_utime=0, si_stime=1} ---
28: ptrace(PTRACE_CONT, 4290, NULL, SIG_0)  = -1 ESRCH (No such process)
29: getpid()                                 = 4289
30: kill(4289, SIGKILL)                      = ?
31: +++ killed by SIGKILL +++
32: Killed
33:
```

Launching the strace utility with the key –f with the executable file of CODESYS Control For Raspberry Pi v3.5.14.00

It can be seen in the log for executing the strace utility with the key -f that CODESYS Runtime changes memory access parameters (line 06), which, as discussed in the previous section, is necessary to unpack program code.

Next, the clone syscall creates a child process (line 13). The parent process has the identifier 4289. The newly created child process is assigned the identifier 4290. Since the -f key is used, strace attempts to trace child processes, causing a notification that a child process has been attached to be shown in line 14.

After that, the parent process attempts to resume the stopped child process by calling the ptrace function with the argument PTRACE_CONT (line 17). Meanwhile, the child process executes ptrace with the argument PTRACE_TRACEME (line 20), indicating by this that the process should be traced by the parent process.

However, the result of executing the function indicates that the process cannot be traced by the parent process. Due to this, the child process terminates (lines 21:22). After that, the parent process receives a response from the ptrace function (line 23) and determines that the child process no longer exists on the system. Next, the parent process makes one more attempt to call the child process (line 28) and, after failing again to find it, terminates (lines 29:30).

At this point, the strace utility terminates.

Debugging

A similar situation arises when attempting to run the executable file in the gdb debugger.

```
01: # gdb ./codesyscontrol.bin
02: GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
03: Copyright (C) 2016 Free Software Foundation, Inc.
04: License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
05: This is free software: you are free to change and redistribute it.
06: There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
07: and "show warranty" for details.
08: This GDB was configured as "arm-linux-gnueabihf".
09: Type "show configuration" for configuration details.
10: For bug reporting instructions, please see:
11: <http://www.gnu.org/software/gdb/bugs/>.
12: Find the GDB manual and other documentation resources online at:
13: <http://www.gnu.org/software/gdb/documentation/>.
14: For help, type "help".
15: Type "apropos word" to search for commands related to "word"...
16: Reading symbols from ./codesyscontrol.bin...(no debugging symbols found)...done.
17: (gdb) set follow-fork-mode child
18: (gdb) run
19: Starting program: /home/pi/ggasss/codesyscontrol.bin
20: [Thread debugging using libthread_db enabled]
21: Using host libthread_db library "/lib/arm-linux-gnueabihf/libthread_db.so.1".
22: [New process 5379]
23: [Thread debugging using libthread_db enabled]
24: Using host libthread_db library "/lib/arm-linux-gnueabihf/libthread_db.so.1".
25:
26: Program terminated with signal SIGKILL, Killed.
27: The program no longer exists.
```

Starting the gdb debugger with the executable file of CODESYS Control For Raspberry Pi v3.5.14.00

To debug the child process, the relevant mode should be set for gdb: set follow-fork-mode child (line 17). After that, CODESYS Runtime is executed (line 18). Next, a child process is created (line 22) and, after some time, the program terminates (lines 26:27).

Consequently, to analyze the executable file, after being unpacked it needs to be brought to a state in which it can be debugged.

It should be noted that a successfully created file tracing process can be emulated by specifying a library containing the functions fork, ptrace, getppid and getsid as the LD_PRELOAD environment variable. However, at this stage this would not be particularly effective.

Threads

CODESYS Runtime is a multithreaded application. In addition to the running process being cloned and tracing the child process, the child process creates an enormous number of threads. Linux system utilities ps and htop get a list of threads created by the process.

```
01: # ps aux | grep -i codesyscontrol
02: root      5404 10.0  0.7  11184  7448 pts/0    S    04:25   0:02 ./codesyscontrol.bin
03: root      5405  5.6  1.3  14852 13172 pts/0    Sl1  04:25   0:01 ./codesyscontrol.bin
04: root      5419  0.0  0.0   4372   540 pts/0    S+   04:25   0:00 grep --color=auto -i codesyscontrol
05:
06: # htop -p 5405
07:
08:   PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
09:  5405 root       20   0 14852 13404  2684 S  4.7  1.4  0:54.62 │  └─ ./codesyscontrol.bin
10:  5416 root       20   0 14852 13404  2684 S  0.0  1.4  0:00.32 │     ├─ BlkDrvTcp
11:  5415 root       20   0 14852 13404  2684 S  0.0  1.4  0:00.37 │     ├─ BlkDrvUdp
12:  5414 root       20   0 14852 13404  2684 S  0.0  1.4  0:00.00 │     ├─ GwCommDrvTcp
13:  5413 root       20   0 14852 13404  2684 S  0.7  1.4  0:01.07 │     ├─ OPCUAServer
14:  5412 root       20   0 14852 13404  2684 S  0.7  1.4  0:00.19 │     ├─ WebServerCloseC
```

```
15:  5411 root      -70   0 14852 13404  2684 S  0.0  1.4  0:00.00 |     ├─ CAAEventTask
16:  5410 root      -95   0 14852 13404  2684 S  2.7  1.4  0:29.29 |     ├─ Schedule
17:  5409 root      -69   0 14852 13404  2684 S  0.0  1.4  0:00.00 |     ├─ SchedException
18:  5408 root       20   0 14852 13404  2684 S  1.3  1.4  0:11.77 |     └─ SchedProcessorL
```

Getting a list of threads created by the process of the executable file of CODESYS Control for Raspberry Pi v3.5.14.00

After running the ps utility and filtering the result with the grep utility, it can be seen that the child process has the identifier 5405 (line 03).

Running the htop command for process 5405 (line 06) produces a list of threads created by the child process (lines 09:18).

Some of the component names from the communication group and the name of the OPC UA industrial protocol can be recognized in thread names (e.g., the BlkDrvTcp component and the BlkDrvUdp component).

Network communications

Based on information provided by the netstat utility, CODESYS Runtime listens on the following ports:

```
1: # netstat -ntupl | grep -i codesys
2: tcp        0      0 0.0.0.0:11740           0.0.0.0:*               LISTEN      5405/./codesyscontr
3: tcp        0      0 0.0.0.0:1217            0.0.0.0:*               LISTEN      5405/./codesyscontr
4: tcp        0      0 127.0.0.1:4840          0.0.0.0:*               LISTEN      5405/./codesyscontr
5: tcp        0      0 192.168.0.92:4840       0.0.0.0:*               LISTEN      5405/./codesyscontr
6: udp        0      0 192.168.0.255:1740      0.0.0.0:*                           5405/./codesyscontr
7: udp        0      0 192.168.0.92:1740       0.0.0.0:*                           5405/./codesyscontr
```

List of listening ports opened by the process of the executable file of CODESYS Control for Raspberry Pi v3.5.14.00

CODESYS Runtime listens both on TCP and on UDP ports. TCP port 11740 (line 2) is used for TCP communication between CODESYS Runtime and the CODESYS Development System.

UDP port 1740 (line 7) is used for the same purpose, the difference being that the communication is carried out over the UDP protocol.

CODESYS Runtime also listens on a broadcast address on UDP port 1740 (line 6). The purpose of listening on broadcast addresses on the client side is usually to enable servers to discover these clients, i.e. as a discovery service. TCP port 4840 (lines 4:5) is used as an OPC UA discovery service.

Information from public sources

Searching for information in public sources is an integral part of research work. We found:

- *Removed at the vendor's request*. Contains a technical overview of the CODESYS Control architecture.
- *Removed at the vendor's request*. Contains a large amount of information that is useful for static analysis, such as the purpose of different functions and their arguments.
- CODESYS Control adaptation manual (*Removed at the vendor's request*). Describes a basic approach to porting CODESYS Runtime to a device without an OS.

Unfortunately, all the information we were able to find dates back to late 2015. However, it needed to be analyzed: although the document versions were outdated, they provided numerous clues that helped us find answers to questions which came up while researching the protocol used for communication between the CODESYS Development System and CODESYS Runtime.

Investigating the CODESYS PDU protocol stack

This chapter is devoted to investigating the stack of protocols of CODESYS PDU (Packet Data Unit). This protocol stack is used for communication between CODESYS network nodes, including CODESYS Development System and CODESYS Runtime.

The CODESYS PDU protocol stack is based on the ISO/OSI model. Like the ISO/OSI model, each layer in the CODESYS PDU protocol is responsible for its own area of operations. To fully understand the operations of the CODESYS PDU protocol, each of its layers must be studied in detail.

Note: The CODESYS PDU protocol stack was investigated using the "black box" method, so most of the names of fields and layers used herein are based on their purpose. Therefore, the names used in the subsequent description may differ from those that are used in public documentation or that were termed by other researchers.

For example, in one of the researched documents, the following names are used for various layers of the CODESYS PDU protocol:

- For the first layer: "datagram layer", "Layer 2" or "block driver" (hereinafter referred to as the Block Driver layer)
- For the second layer: "network layer", "Layer 3" or "router" (hereinafter referred to as the Datagram layer)
- For the third layer: "protocol layer", "Layer 4" or "channel management" (hereinafter referred to as the Channel layer)
- For the fourth layer: "application layer", "layer 7" or "application services" (hereinafter referred to as the Services layer)

Basic description of the protocol

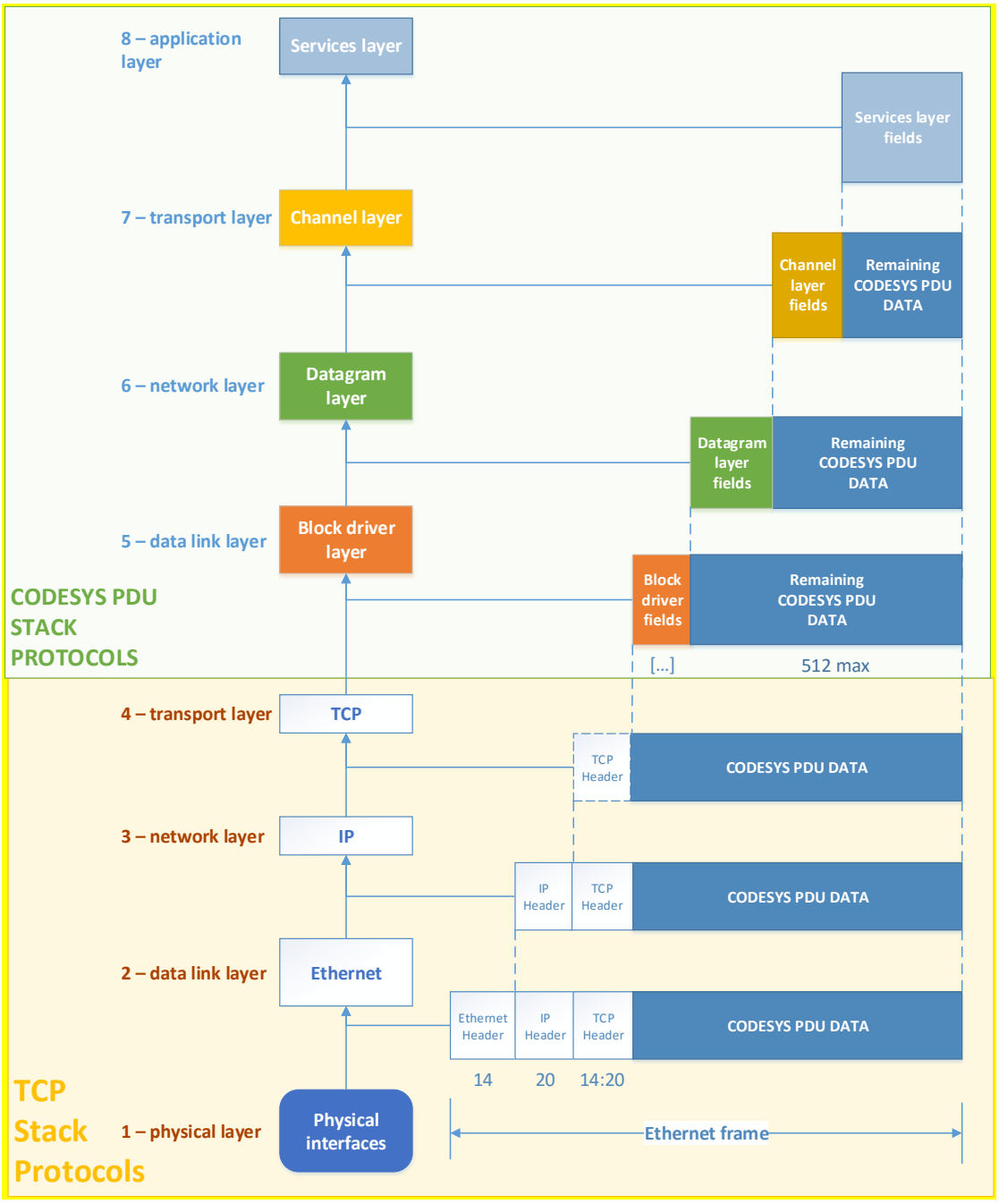
CODESYS PDU (Packet Data Unit) is a protocol stack consisting of four different layers:

- Block Driver layer
- Datagram layer
- Channel layer
- Services layer

The order of bytes in this protocol stack is little endian, but can be changed to big endian if necessary. Protocol operation is synchronous or asynchronous depending on the protocol layer.

Use of the CODESYS PDU protocol is not limited to network communication. It is also used for communication over USB, the CAN bus, and serial ports. The CODESYS Runtime environment always uses the capabilities of the operating system for which it was adapted. Therefore, the resultant information packet will contain the generated data of CODESYS Runtime and the data generated by OS drivers for the specific physical interface.

For example, the resultant CODESYS PDU packet sent through a network interface over TCP will contain two protocol stacks: TCP and CODESYS PDU.

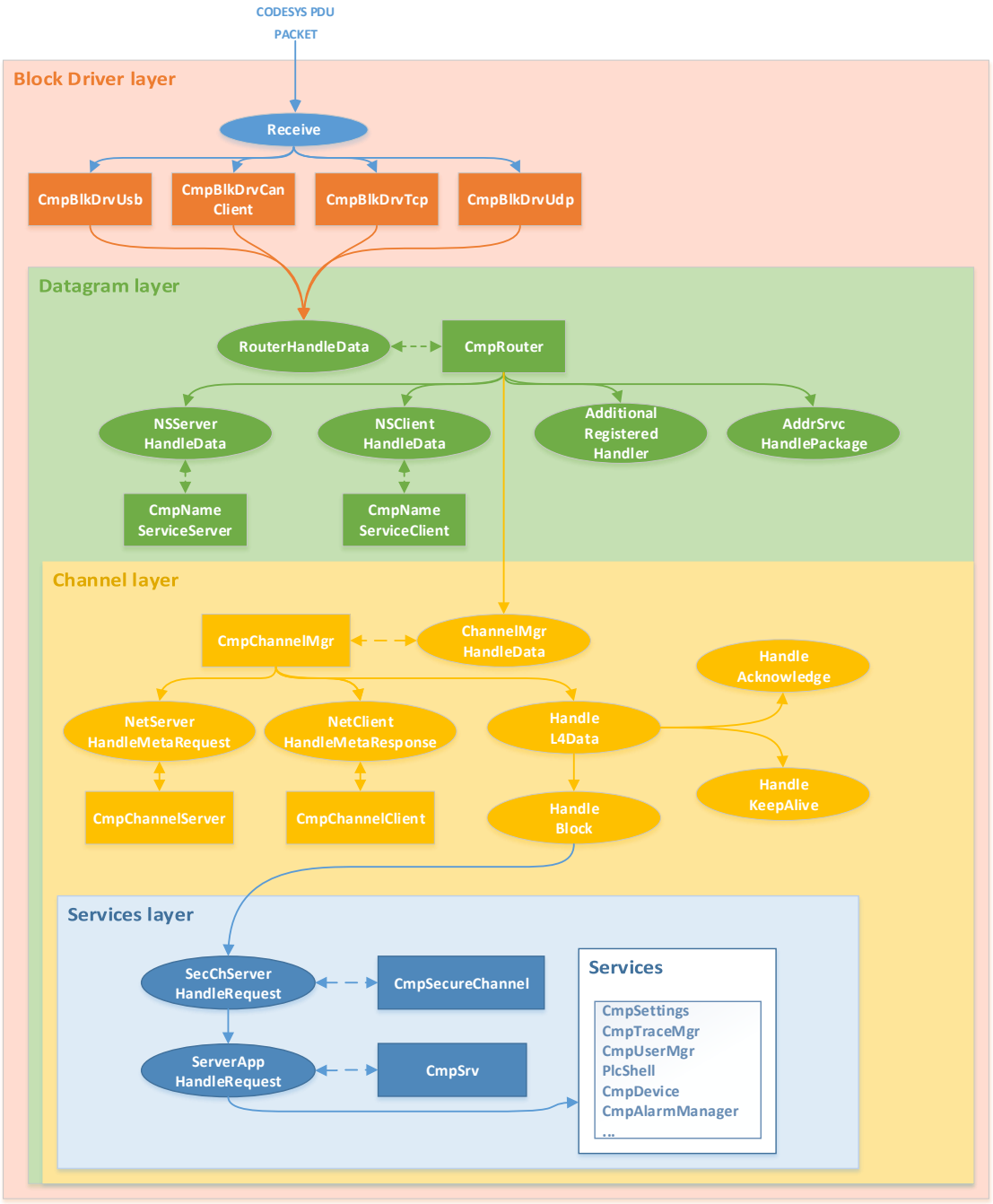


Example use of the TCP and CODESYS PDU protocol stacks in one packet

The capability for communication over physical interfaces is implemented by components of the Communication – Block Drivers group. In addition, any developer may develop their own Block Driver and use the CODESYS PDU protocol within it.

This protocol is based on the ISO/OSI model. CODESYS PDU fully excluded the physical layer from this model, and the session layer and presentation layer were merged with the application layer. Each specific layer is processed by one component or multiple components from one group.

Below is a schematic representation of how components are involved in parsing an inbound packet generated based on the CODESYS PDU protocol.



Schematic representation of how a CODESYS PDU packet is parsed by components

The cumulative operation of these components determines the capacity of the CODESYS PDU protocol stack.

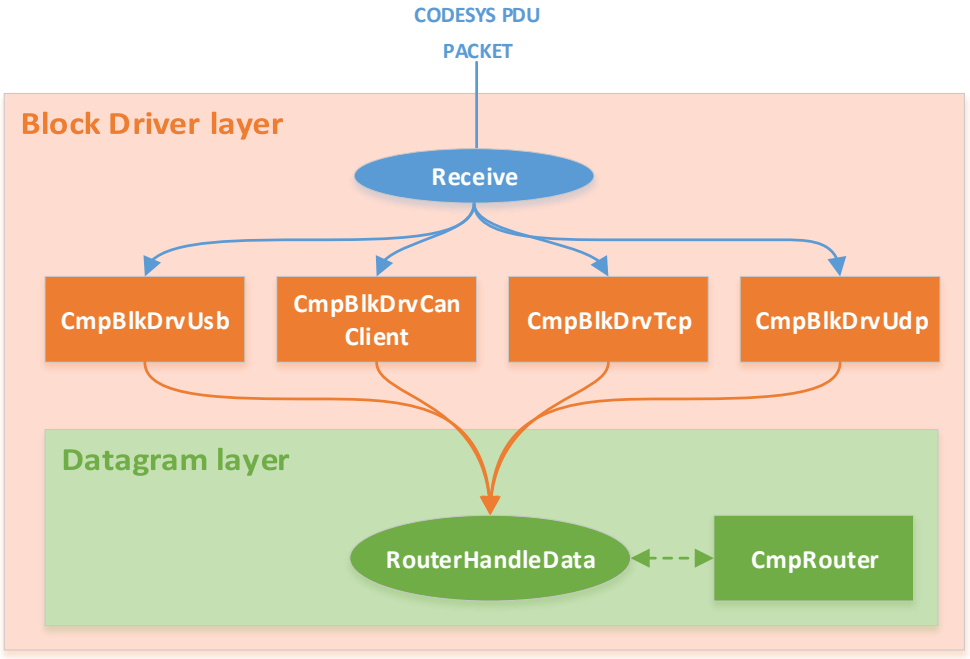
This document examines each layer in the CODESYS PDU protocol stack.

Analysis of the protocol stack

Block Driver layer

All useful operation of CODESYS Runtime is the cumulative work of its components. The components can expand the capabilities of each other. This also works for components that parse the received packet generated over the PDU protocol.

The main task of components from the Block Drivers group is to create the capability to communicate over a physical or software interface. Any Block Drivers component is an "input point" for receiving an information packet and the point from which it is transmitted. Therefore, these components can add additional fields in the protocol prior to sending a packet.



Schematic representation of how a CODESYS PDU packet is parsed by components at the Block Driver layer

For example, this is how the Block Driver **CmpBlkDrvTcp** component works. This component implements communication over the TCP protocol. In each message, **CmpBlkDrvTcp** adds two fields, each of which is a 4-byte number:

```
> Transmission Control Protocol, Src Port: 49171, Dst Port: 11740, Seq: 93, Ack
> CoDeSys V3 Protocol
```

```
0000  08 00 27 a5 f2 66 08 00  27 90 85 bf 08 00 45 00   ··'··f·· '·····E·
0010  00 80 00 cc 40 00 80 06  77 e2 c0 a8 00 21 c0 a8   ····@··· w····!··
0020  00 58 c0 13 2d dc 99 0c  36 6a 9d 4a 13 5e 50 18   ·X··-··· 6j·J·^P·
0030  3f e1 2c fc 00 00 00 01  17 e8 58 00 00 00 c5 6b   ?·,····· ··X····k
0040  40 40 00 53 2d dc c0 a8  00 58 2d df c0 a8 00 21   @@·S-··· ·X-····!
0050  80 00 00 00 00 00 01 81  10 00 01 00 00 00 00 00   ········ ········
0060  00 00 24 00 00 00 db c5  6d 9e 55 cd 10 00 01 00   ··$····· m·U·····
0070  01 00 00 00 00 00 10 00  00 00 00 00 00 00 01 8c   ········ ········
0080  80 00 06 10 00 00 00 00  00 00 00 00 00 00        ········ ·····
```

Color			
fields	magic	length	CODESYS PDU
value	Const 0xe8170100	88 (0x58)	[...]

Example use of two additional fields at the Block Driver layer

- **magic** refers to the magic number. The constant number **0xe8170100** is inserted and verified by the **CmpBlkDrvTcp** component each time the component receives a network packet.
- **length** is the cumulative number of bytes in the packet, include the sizes of the **magic** and **length** fields (both fields have a size of 4 bytes each).

Below is a tracing of a call of the **Receive()** function, which belongs to the **CmpBlkDrvTcp** component. The **Receive()** function processes the **magic** and **length** fields.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
001: Removed at the vendor's request
002: {
[...]
061:              Removed at the vendor's request
[...]
068:              Removed at the vendor's request
069:                               Removed at the vendor's request
070:                               Removed at the vendor's request
071:                               Removed at the vendor's request
072:                               Removed at the vendor's request
073:                               Removed at the vendor's request
[...]
094:              Removed at the vendor's request
095:              Removed at the vendor's request
096:              {
099:                Removed at the vendor's request
100:                {
[...]
101:                Removed at the vendor's request
102:                {
[...]
144:                Removed at the vendor's request
[...]
148:                Removed at the vendor's request
149:                                 Removed at the vendor's request
150:                                 Removed at the vendor's request
151:                                 Removed at the vendor's request
152:                                 Removed at the vendor's request
153:                                 Removed at the vendor's request
[...]
179:                    Removed at the vendor's request
180:                    Removed at the vendor's request
[...]

196:        Removed at the vendor's request
[...]
206: }
```

Decompiled pseudocode of the Receive function of the CmpBlkDrvTcp component

Receiving all data from the network for subsequent processing by the **CmpBlkDrvTcp** component occurs in two steps:

1. At the first step, the component obtains the first 8 bytes (line 068) from the data received over the network through the **SysSockRecv** function, which was exported by the system component **SysSocket.** The maximum number of bytes that can be received is transmitted in the third argument of the **SysSockRecv** function. Then the first 4 bytes are compared with the magic constant (line 099). The second 4 bytes are compared with the number 520. The number 520 was obtained by adding the maximum possible size of a packet generated over the CODESYS PDU protocol (512 bytes) to the cumulative size of the **magic** and **length** fields (8 bytes).
2. The remaining data is extracted at the second step. It is expected that the data size will be equal to the difference between the value of the **length** field and the cumulative size of the **magic** and **length** fields (line 144).

Then the **CmpBlkDrvTcp** component transfers management of the **RouterHandleData** function (line 196), which is registered by the **CmpRouter** component, to the Datagram layer.

Please bear in mind that the **magic** and **length** fields will be absent when communicating over the UDP protocol.

```
> User Datagram Protocol, Src Port: 1743, Dst Port: 1743
> CoDeSys V3 Protocol
```

```
0000  ff ff ff ff ff ff 08 00  27 90 85 bf 08 00 45 00
0010  00 34 00 9e 00 00 80 11  b7 aa c0 a8 00 21 c0 a8
0020  00 ff 06 cf 06 cf 00 20  2f ac c5 74 40 03 00 30
0030  26 6e 03 21 80 00 00 00  00 00 02 c2 00 04 8d f5
0040  00 00
```

Color	
fields	CODESYS PDU
value	[...]

Example absence of additional fields at the Block Driver layer

The **UdpReceiveBlock()** function of the **CmpBlkDrvUdp** component is analogous to the **Receive()** function of the **CmpBlkDrvTcp** component.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
001: Removed at the vendor's request
002: {
019:
[...]
047:            Removed at the vendor's request
[...]
055:      Removed at the vendor's request
056:      {
[...]
064:            Removed at the vendor's request
[...]
071:      }
072:      Removed at the vendor's request
```

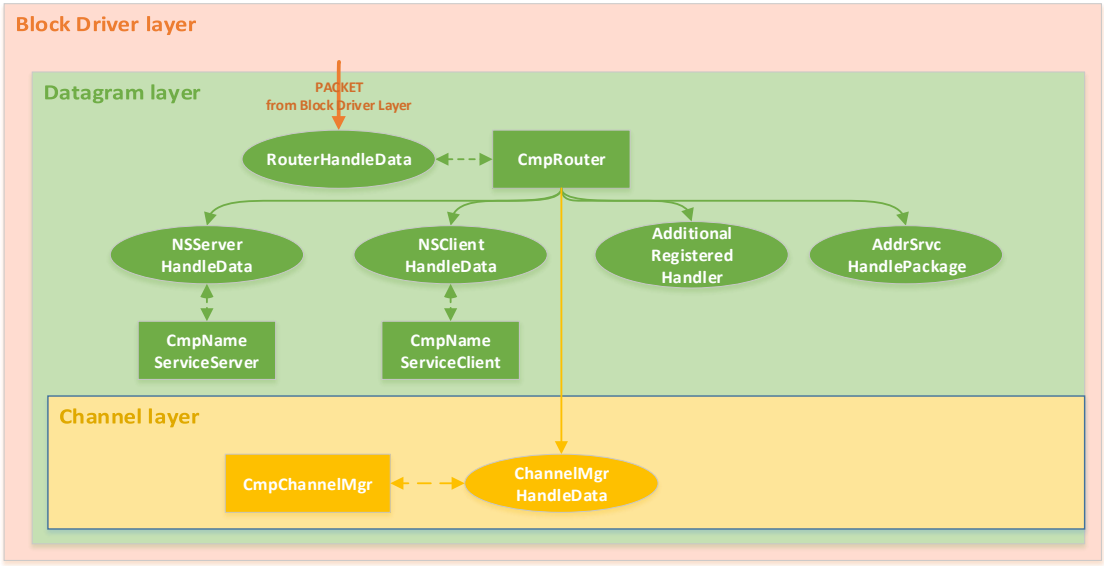
```
073:      {
[...]
080:          Removed at the vendor's request
081:          Removed at the vendor's request
083:      }
084:    }
[...]
111:      Removed at the vendor's request
[...]
```

Decompiled pseudocode of the UdpReceiveBlock function of the CmpBlkDrvUdp component

The **UdpReceiveBlock()** function does not perform any verifications of the received data. Moreover, the **CmpBlkDrvUdp** component has another special feature. Namely, the **UdpReceiveBlock()** function listens for broadcast messages (line 047). If such data was not detected, the component attempts to count the data that was sent specifically to it (line 64). If data was received in one of these cases, the **CmpBlDrvUdp** component calls the **RouterHandleData** function for further processing (line 111).

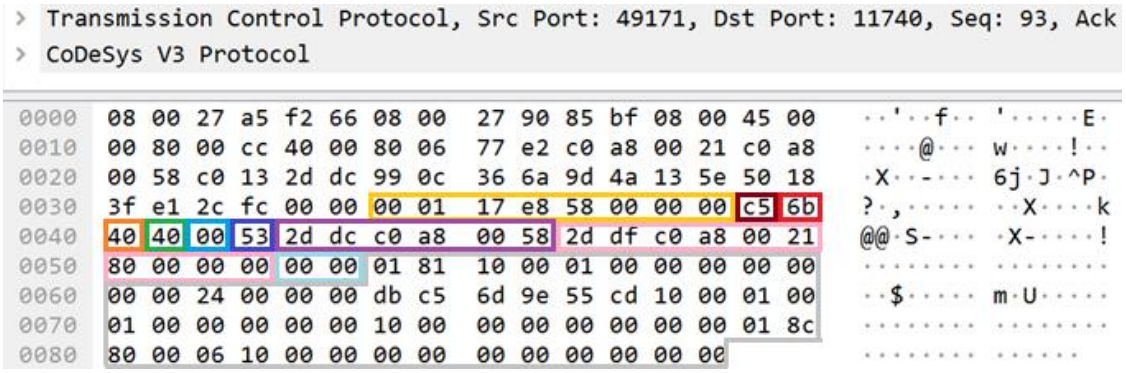
Datagram layer

The Datagram layer is the next layer in the CODESYS PDU protocol stack. The main purpose of this layer is to route packets, detect nodes in the CODESYS network, and transmit data to the next layer. The main component in this layer is **CmpRouter**. The **CmpNameServiceClient** and **CmpNameServiceServer** components are auxiliary components.



Schematic representation of how a CODESYS PDU packet is parsed by components at the Datagram layer

Components of the Block Drivers group are required to call the **RouterHandleData** function, which operates at the Datagram layer. In function call arguments, components transmit the received data.



Color								
fields	Blk_driver_fields (CmpBlkDrvTcp)	magic	hop_info_byte		packet_info			
			hop_count (5 bit)	header_length (3 bit)	priority (2 bit)	signal (1 bit)	type_address (1 bit)	length_data_block (4 bit)
value	[...]	197 (0xc5)	0xd	0x3	0x1 (NORMAL)	0x0 (NO_SIGNAL)	0x0 (DIRECT)	0x0

Color								
fields	service_id	message_id	lengths		sender		Receiver	
			receiver_length	sender_length	port	address	port	Address
value	0x40	0x00	0x5	0x3	11740 (2ddc)	192.168.0.88 (c0a80058)	11743 (2ddf)	192.168.0.33 (c0a80021) 800000

Color		
Fields	Padding	Remaining data
value	0x0000	[...]

Utilized fields at the Datagram layer

In terms of traffic, this function processes the following fields and data:

- **magic** refers to the magic number of the packet generated over the CODESYS PDU protocol. The size of this field is one byte, and it is inserted by the **CmpRouter** component.
- **hop_info** refers to the bit structure, which consists of two fields: 5-bit **hop_count** field and 3-bit **header_length** field:
  a. The **hop_count** field is responsible for the possible number of transmissions of a packet received over the network. Each time one CODESYS network node receives a packet and redirects it to another CODESYS network node, it decrements the value of the **hop_count** field. If a node received a packet but is not its final recipient and the value of the **hop_count** field is equal to 0, the node will discard this packet. Essentially, this field protects a network that has CODESYS nodes from an endless forwarding of a packet.
  b. The **header_length** field indicates the number of bytes until the next field with the data size (**lengths)**. When the value of the **header_length** field is added to its position in the packet, it is expected that the position on the **lengths** field will be obtained**.**

- **packet_info** refers to packet settings. This field also represents the bit structure.
  a. The first two bits are the **priority** field. It designates the priority of the processed packet. The following numerical values are used to designate priority: 0 – low, 1 – normal, 2 – high, 3 – emergency
  c. The following **signal** bit is used by the **CmpRouter** component as the returned packet processing status in which you can indicate errors.
  d. The **type_address** field indicates the type of transmitted address. This field is necessary so that the **CmpRouter** component can understand the contents of the **sender** and **receiver** fields. There are two values for the **type_address** field: 0 – full address, 1 – relative address
  e. The last field **length_data_block** indicates the maximum size of data that can be accepted by a recipient.
- **service_id** refers to the ID of the service. Indicates which specific server must process the received data. CODESYS Runtime contains and identifies the following services:
  a. The service with an ID of **1** for a request and **2** for a response is the address service. This service is used to detect nodes that are "alive" in the network and to build an information network from these nodes. A node in this network serves as a participant with CODESYS Runtime or CODESYS Development System running.
  b. The service with an ID of **3** for a request and **4** for a response is the name service. This service is used to receive information about a node.
  c. The service with an ID of **64 (0x40)** for both a request and response is the channel service. This service is used for querying the server and the communication channel manager.
- **message_id** refers to the ID of the message. This value is indicated by the sender and is used to identify the message. Normally **CmpRouter** sends a 4-bit value of the current time as the message ID. This provides for a unique message ID.
- **lengths** refers to the sizes of the **receiver and sender** fields. The **lengths** field is a bit structure in which the most significant 4 bits contain a value corresponding to half of the number of bytes in the **receiver** field, while the least significant 4 bits contain a value corresponding to half of the number of bytes in the **sender** field. In other words, the number of bytes in the **receiver** field and in the **sender** field will be two times more than those specified in the lengths field. For example, the value of the most significant 4 bits of the **lengths** (0x53) field is equal to 5 for the examined packet. This means that the total number of bytes for the **receiver** field will be 10.
- **sender** refers to the address for which the message is intended.
- **receiver** refers to the address to which the response to the message must be sent.
- The **padding** field is added to the end of the packet. This field is optional.

The **sender** and **receiver** fields have their own data format that depends on the utilized Block Driver component. For example, **CmpBlkDrvTcp** expects the full network address of the node and number of the network port in these fields. In other words, the bytes of the receiver field (**2ddcc0a80058)** actually contain port 11740 (**2ddc**) and recipient address 192.168.0.88 (**c0a80058**).

CmpBlkDrvUdp uses a different format. It uses a relative address instead of a full address, and it uses one byte instead of two bytes for the port value. This byte for the port indicates the port index. **CmpBlkDrvUdp** identifies four port indexes: 0, 1, 2, 3. Each index corresponds to one UDP port: 0 – 1740, 1 – 1741, 2 – 1742, 3 – 1743. The relative network address is the last byte in numerical format of the physical address.

```
> User Datagram Protocol, Src Port: 1743, Dst Port: 1740
> CoDeSys V3 Protocol
```

```
0000  08 00 27 a5 f2 66 08 00  27 90 85 bf 08 00 45 00   ··'··f·· '·····E·
0010  00 64 15 df 40 00 80 11  62 e0 c0 a8 00 21 c0 a8   ·d··@··· b····!··
0020  00 58 06 cf 06 cc 00 50  53 60 c5 6b 40 40 00 31   ·X·····P S`·k@@·1
0030  00 58 03 21 80 00 00 00  00 00 01 81 24 00 01 00   ·X·!···· ····$···
0040  00 00 00 00 00 00 24 00  00 00 13 fd 2b 3a 55 cd   ······$· ····+:U·
0050  10 00 01 00 01 00 00 00  00 00 10 00 00 00 00 00   ···· ···· ····
0060  00 00 01 8c 80 00 06 10  00 00 05 00 00 00 00 0d   ···· ···· ····
0070  05 03                                             ··
```

Color								
fields	lengths		Sender		receiver		padding (optional)	Remaining data
	receiver_ length	sender_ length	port_ index	relative address	port_ index	relative_ address		
value	0x1	0x3	0	88 (0x58)	3	33 (0x21) 800000	0x0000	[...]

Example contents of the Sender and Receiver fields when using the CmpBlkDrvUdp component

Therefore, the bytes value of the **sender** field (**0058**) will contain the port 1740 (the value of the **port_index** field is equal to **0x0**) and the address 192.168.0.88 (**0x58** is the last byte of the address, while the first three bytes are extracted from the interface address). The recipient will await a response at port 1743 (the value of the **port_index** field is equal to **0x3**) and at the address 192.168.0.33 (**0x21**).

The purpose and format of all other data in the packet depends on the service (**service_id**) for which this packet is intended. If this ID is equal to **1**, **2**, **3** or **4**, the handler remains the **CmpRouter** component or its auxiliary components **CmpNameServiceClient** and **CmpNameServiceServer**. If the ID is equal to **64 (0x40)**, all other data is transmitted to the **CmpChannelManager** component.

Based on the **sender** field, the **CmpRouter** component determines if the packet is intended for itself or if it needs to be forwarded to another node. In the first case, the **CmpRouter** component first decrements the value of **hop_count** in the packet, then sends it as is to the node specified in the **sender** field. In the second case, the **CmpRouter** component processes the packet and returns the result to the address specified in the **receiver** field. A packet that was intended for the node is processed by the **HandleLocally** function.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
001: Removed at the vendor's request
002: {
[...]
044:    Removed at the vendor's request
045:    Removed at the vendor's request
046:    {
[...]
051:      Removed at the vendor's request
```

```
[...]
060:    }
061:    Removed at the vendor's request
062:    {
063:      Removed at the vendor's request
064:      {
[...]
069:        Removed at the vendor's request
[...]
078:      }
079:      Removed at the vendor's request
080:      {
[...]
086:        Removed at the vendor's request
[...]
096:        Removed at the vendor's request
097:      }
098:    }
099:    Removed at the vendor's request
100:    {
101:      Removed at the vendor's request
102:        Removed at the vendor's request
103:      Removed at the vendor's request
111:      [...]
119:      Removed at the vendor's request
120:    }
121:    Removed at the vendor's request
122:    {
123:      Removed at the vendor's request
124:      {
125:        Removed at the vendor's request
126:        Removed at the vendor's request
127:          Removed at the vendor's request
[...]
142:      }
143:    }
144:    Removed at the vendor's request
145: }
```

Decompiled pseudocode of the HandleLocally function of the CmpRouter component

The **HandleLocally** function uses the value of the **service_id** field to determine which specific handler must be queried:

- For a value that is equal to **1** or **2**: the **AddrSrvcHandlePackage** function (line 103). This is a handler of the address service.
- For a value that is equal to **3**: the **NSServerHandleData** function (line 51). This is the handler of the name service, which processes incoming requests. In other words, it operates as a server.
- For a value that is equal to **4**: the **NSClientHandleData** function (line 69). This is a handler of the name service, which processes the results of completed requests. In other words, it operates as a client.
- For a value that is equal to **0x40**: the **ChannelMgrHandleData** function (line 86). This is a handler of the channel service.

If a suitable handler was not found for the received **service_id**, a search is performed among the additional handlers registered by the **RouterRegisterProtocolHandler** function. If one is found, it is queried (lines 125:127).

```
01: Removed at the vendor's request
02: {
[...]
14:    Removed at the vendor's request
15:    {
16:      Removed at the vendor's request
17:      {
18:        Removed at the vendor's request
```

```
19:       {
20:         Removed at the vendor's request
21:       }
22:       Removed at the vendor's request
23:       {
24:         Removed at the vendor's request
25:         Removed at the vendor's request
26:         Removed at the vendor's request
[...]
39:   Removed at the vendor's request
40: }
```

Decompiled pseudocode of the RouterRegisterProtocolHandler function

The **RouterRegisterProtocolHandler** function takes the **service_id** and **handler** as arguments (line 1). This function does not let you register the handler (line 16) for the following IDs: **1**, **2**, **3**, **4** and **64 (0x40)**. If a handler is not set for a specific ID (line 18), the handler will be added to the global dictionary of handlers (**s_protocolHandlers**), where the key for the handler is the first argument of **service_id** (line 24).

Next we will examine handlers of CODESYS Runtime system services, namely the handlers of the address service and handlers of the name service.

The handler of the channel service is examined in the chapter titled "Channel layer".

Address service

The handler of the address service uses the **service_id** to identify possible commands: the "request" command with **service_id** 1 and the "response" command with **service_id** 2. The **AddrSrvcHandlePackage** function is the handler.

```
Call trace:
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request

Pseudocode
01: Removed at the vendor's request
02: {
[...]
23:   Removed at the vendor's request
24:   {
25:     Removed at the vendor's request
26:       Removed at the vendor's request
27:   }
28:   Removed at the vendor's request
29:   {
30:     Removed at the vendor's request
31:     Removed at the vendor's request
32:     {
33:       Removed at the vendor's request
34:       {
[...]
75:             Removed at the vendor's request
76:         }
77:       }
78:     }
[...]
98: }
```

Decompiled pseudocode of the AddrSrvcHandlePackage function

The "request" command with an ID of 1 (line 23) is sent by a CODESYS network node to notify the other nodes about its existence. This request can be continually monitored in the traffic over a broadcast address: it is sent at the same frequency by all CODESYS network nodes to a broadcast address.

```
9227 971.844759   192.168.0.92    192.168.0.255   CODESYSV3   60 1740 → 1740 Len=8
9228 971.844930   192.168.0.92    192.168.0.255   CODESYSV3   60 1740 → 1741 Len=8
9229 971.844931   192.168.0.92    192.168.0.255   CODESYSV3   60 1740 → 1742 Len=8
9230 971.845001   192.168.0.92    192.168.0.255   CODESYSV3   60 1740 → 1743 Len=8
9268 978.456825   192.168.0.88    192.168.0.255   CODESYSV3   60 1740 → 1740 Len=12
9269 978.456868   192.168.0.88    192.168.0.255   CODESYSV3   60 1740 → 1741 Len=12
9270 978.456868   192.168.0.88    192.168.0.255   CODESYSV3   60 1740 → 1742 Len=12
9271 978.456868   192.168.0.88    192.168.0.255   CODESYSV3   60 1740 → 1743 Len=12
9579 1012.630681  192.168.0.33    192.168.0.255   CODESYSV3   50 1743 → 1740 Len=8
9580 1012.630796  192.168.0.33    192.168.0.255   CODESYSV3   50 1743 → 1741 Len=8
9581 1012.630859  192.168.0.33    192.168.0.255   CODESYSV3   50 1743 → 1742 Len=8
9582 1012.630942  192.168.0.33    192.168.0.255   CODESYSV3   50 1743 → 1743 Len=8
```

Broadcast notifications of CODESYS network nodes

If there is a parent node among the CODESYS network nodes, it learns about the existence of the node that sent the request. This same request can be sent by the parent node. If such a request is received by child objects, they notify the parent node about their existence. Additional fields are not used for this request.

The "response" command with an ID of 2 (line 28) is usually sent by a parent node. This command is used to build a CODESYS information network. It uses the following fields:

Fields	size (byte)	Type
version_major	1	Uint8_t
version_minor	1	Uint8_t
Unused	1	Uint8_t
address_len	1	Uint8_t
address	30	Uint8_t[]
subnet_id	4	Uin32_t
subnet_params	1	Uint8_t
parent_subnet_params	1	Uint8_t
Unused	2	Uint16_t

- **version_major** is a field that is used to designate the version of the CODESYS information network that will be generated. For the CODESYS PDU protocol, the value of **version_major** is always equal to 1 (line 75 of the decompiled pseudocode of the **AddrSrvcHandlePackage** function).
- **version_minor** is the field indicating the utilized version of the command. It determines the additional fields in the command and in the response. For example, if the field has a positive value, the **parent_subnet_params** field will be used in the packet.
- **address_len** indicates the number of bytes of the **parent_address** field that needs to be processed. The total number of bytes is multiplied by two.
- **address** refers to the address of the parent node.
- **subnet_id** refers to the ID of the generated subnet.
- **subnet_params** and **parent_subnet_params** are the settings of the current subnet and the subnet of the parent node.

The CODESYS network node that receives such a message sets the source of this message as the parent node (provided that no parent node was previously specified).

Name service

Messages that are intended for the name service are processed by the auxiliary components **CmpNameServiceClient** and **CmpNameServiceServer**. Messages of the name service are also divided into a "request" command and "response" command. Incoming "requests" from other nodes are processed by the **CmpNameServiceServer** component. Requests that were sent by the CODESYS Runtime node are processed as "responses" by the **CmpNameServiceClient** component.

The common header of name service messages (**name_service_header**) uses the following fields:

Fields	size (byte)	Type
Subcmd	2	Uint16_t
Version	2	Uint16_t
Message_id	4	Uint32_t
Message_data	n	Uint8_t[n]

- **subcmd** refers to the ID of the command that the name service needs to execute.
- **version** refers to the command version number. This field determines the availability of additional fields in the **message_data** field that are typical for the specified version of the command.
- **message_id** refers to the ID of the message. This value is returned in a response. A numerical identifier of the current time is used to create the value of this field.
- **message_data** refers to command fields whose format is determined by the command **(subcmd)**.

The **CmpNameServiceServer** component exports the **NSServerHandleData** function**.** The **NSServerHandleData** function serves as the handler of requests sent to the name service from other nodes.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
01: Removed at the vendor's request
02: {
[...]
09:   Removed at the vendor's request
10:     Removed at the vendor's request
11:   Removed at the vendor's request
12:     Removed at the vendor's request
13:   Removed at the vendor's request
14:   Removed at the vendor's request
15:   {
[...]
18:     Removed at the vendor's request
```

```
19:    }
20:    Removed at the vendor's request
21:    {
[...]
24:      Removed at the vendor's request
25:    }
26:    Removed at the vendor's request
27:    {
28:      Removed at the vendor's request
29:    }
30:    Removed at the vendor's request
31: }
```

Decompiled pseudocode of the NSServerHandleData function of the CmpNameServiceServer component

The **NsServerHandleData** handler determines the two possible IDs of the **subcmd** field, and queries the corresponding auxiliary handler for each of them:

- For the ID **0xc202** (line 20), this is the **HandleResolveAddrReq** handler (line 18). This is a request to receive information about a node. A response to this request uses the value of **service_id 4** and will be processed by the **NsClientHandleData** function.
- For the ID **0xc201** (line 14), this is the **HandleResolveNameReq** handler (line 18). This request is analogous to a request with the ID **0xc202**, with the only difference being that the node name is transmitted in the body of the request. If the transmitted node name does not match the name of the node that received the request, the node ignores the request. A response to this request uses the value of **service_id 4** and will be processed by the **NsClientHandleData** function.

The **CmpNameServiceClient** component exports the **NSClientHandleData** function that serves as the handler of responses to name service requests received from nodes. The response to the request uses the common message header **(name_service_header)**. Depending on the value specified in the **version** field, the response may differ. For a **version** field that is equal to **0x103**, the response will contain the following fields:

Fields	size (byte)	Type
max_channels	2	Uint16_t
byte_order	1	Uint8_t
Unknown	1	Uint8_t
node_name_length	2	Uint16_t
device_name_length	2	Uint16_t
vendor_name_length	2	Uint16_t
target_type	4	Uint32_t
target_id	4	Uint32_t
target_version	4	Uint32_t
Node_name	node_name_length	Uint8_t[node_name_length]
device_name	device_name_length	Uint8_t[device_name_length]
vendor_name	vendor_name_length	Uint8_t[vendor_name_length]

- **max_channels** refers to the number of simultaneously supported communication channels. This number is regulated by the settings of the **CmpChannelMgr** component. This same communication channel is used at the Channel layer of the CODESYS PDU protocol stack.
- **byte_order** indicates the byte order used in the protocol. As mentioned earlier, CODESYS PDU uses the little endian byte order by default. However, the byte order

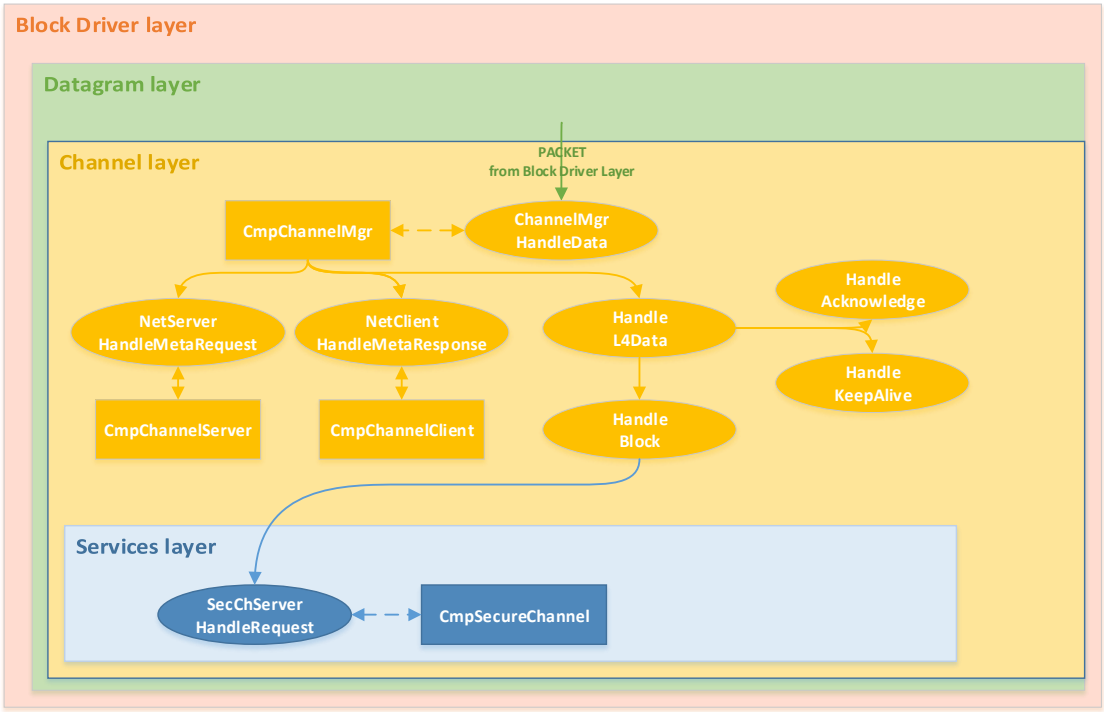
can be changed. A value of the **byte_order** field equal to 1 indicates the use of little endian byte order.

- ***Unknown*** means that we were unable to determine the purpose of the field during our research.
- **node_name_length** refers to the size of the **node_name** field.
- **device_name_length** refers to the size of the **device_name** field.
- **vendor_name_length** refers to the size of the **vendor_name** field.
- **target_type** refers to the type of device.
- **target_id** refers to the device ID.
- **target_version** refers to the version of the device.
- **node_name** refers to the network name of the device.
- **device_name** refers to the name of the device.
- **vendor_name** refers to the name of the organization that developed the device or implemented CODESYS Runtime into the device.

If the value **0x400** was indicated in the **version** field of the request, the response will contain the following fields: address of the parent node, license number, and type of Block Driver component.

Channel layer

The channel layer is the next layer in the CODESYS PDU protocol stack.



Schematic representation of how a CODESYS PDU packet is parsed by components at the Channel layer

A channel is a mechanism of communication between nodes of a CODESYS network that guarantees synchronization of communication, verification of the transmitted data integrity, notification of message delivery, and transmission of a large amount of data.

The main component at this layer is the **CmpChannelMgr** component (Component Channel Manager). This component is the communication channel manager. It tracks synchronization

of communication between nodes and the integrity of received data, or transfers management to the channel server (the **CmpChannelServer** component**)** or a client of communication channels (the **CmpChannelClient** component).

The **CmpChannelServer** component is a channel server. It is responsible for the following:

- Creating an accumulation buffer for received and sent messages
- Creating and closing communication channels
- Delivering information about a communication channel
- Closing channels whose time to live expired or that haven't been accessed for a long time

The **CmpChannelClient** component is a client of channels. It generates the necessary requests and handles the processing of responses from the channel server.

The **CmpChannelMgr** component exports the **ChannelMgrHandleData** function that is queried by the **CmpRouter** component if the value of the **service_id** field is equal to 0x40 at the Datagram layer.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
001: Removed at the vendor's request
[...]
079:     Removed at the vendor's request
080:     {
[...]
086:       Removed at the vendor's request
[...]
096:       Removed at the vendor's request
097:     }
[...]
```

Locations for transferring management of the ChannelMgrHandleData function

A common header (**channel_common_header**) is used for the channel layer. It contains the following fields:

Fields	size (byte)	Type
package_type	1	Uint8_t
Flags	1	Uint8_t
packet_data	n	Uint8_t[n]

- The **package_type** field determines the type of packet. If the most significant bit is set in the value of this field, the packet is a command for the channel server. If the most significant bit is absent, the packet is intended for the channel manager.
- The **flags** field has a varying purpose depending on the **package_type** field**.**
- The **packet_data** field contains the other packet data that is determined by the **packet_type** field**.**

The **ChannelMgrHandleData** function exported by the **CmpChannelMgr** component is the handler at the Channel layer. Operations of this function can be divided into three categories: operations with the channel server, operations with the channel client, and operations with the channel manager.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
01: Removed at the vendor's request
02: {
[...]
20:         Removed at the vendor's request
21:         {
22:           Removed at the vendor's request
23:           {
[...]
27:             Removed at the vendor's request
28:           }
29:         }
30:         Removed at the vendor's request
31:         {
[...]
35:           Removed at the vendor's request
36:         }
37:       }
  [...]
43:         Removed at the vendor's request
44:       }
45:       Removed at the vendor's request
[...]
48: }
```

Fragment of the decompiled pseudocode of the ChannelMgrHandleData function of the CmpChannelMgr component

Operations with the channel server occur in the **NetServerHandleMetaRequest** handler**.**

A response from the channel server is processed by the channel client in the **NetClientHandleMetaResponse** handler (line 35) and (line 27).

Operations with the channel manager occur in the **HandleL4Data** function (line 43).

Commands for the communication channel server

We first examine the group of commands for working directly with the communication channel server. This group contains commands for opening and closing the channel, and for receiving information about the communication channel server.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
```

```
Pseucode:
01: Removed at the vendor's request
02: {
[...]
27:           Removed at the vendor's request
[...]
35:         Removed at the vendor's request
36:       }
[...]
48: }
```

Fragment of the decompiled pseudocode of the ChannelMgrHandleData function

The command handler for the communication channel server is the **NetServerHandleMetaRequest** function**,** and the command handler for a client is the **NetClientHandleMetaResponse** function. The first function processes incoming requests, which means it implements the server side. The second function processes responses to requests, which means it implements the client side. Both of these functions will be examined below.

```
Call trace:
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request

Pseudocode:
01: Removed at the vendor's request
02: {
03:   Removed at the vendor's request
04:
05:   Removed at the vendor's request
06:     Removed at the vendor's request
07:   Removed at the vendor's request
08:   Removed at the vendor's request
09:   {
10:     Removed at the vendor's request
11:       Removed at the vendor's request
12:       Removed at the vendor's request
13:     Removed at the vendor's request
14:       Removed at the vendor's request
15:       Removed at the vendor's request
16:     Removed at the vendor's request
17:       Removed at the vendor's request
18:       Removed at the vendor's request
19:   }
20:   Removed at the vendor's request
21: }
```

Fragment of the decompiled pseudocode of the NetServerHandleMetaRequest function

The **NetServerHandleMetaRequest** function (line 01) identifies three possible IDs of a command (**command_id)** to be processed:

- **0xC2** (line 16) for the **GET_INFO** command. The **HandleInfoReq** function (line 17) serves as the command handler.
- **0xC3** (line 10) for the **OPEN_CHANNEL** command. The **HandleOpenChannelReq** function (line 11) serves as the command handler.
- **0xC4** for the **CLOSE_CHANNEL** command. The **HandleCloseChannelReq** function (line 14) serves as the command handler.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
23: Removed at the vendor's request
24: {
25:   Removed at the vendor's request
26:
27:   Removed at the vendor's request
28:     Removed at the vendor's request
29:   Removed at the vendor's request
30:   Removed at the vendor's request
31:   {
32:     Removed at the vendor's request
33:   }
34:   Removed at the vendor's request
35:   {
36:     Removed at the vendor's request
37:   }
38:   Removed at the vendor's request
39: }
```

Fragment of the decompiled pseudocode of the NetClientHandleMetaResponse function

The client function **NetClientHandleMetaResponse** (line 23) identifies only two possible commands for a client:

- **0xC3** (line 30) for the **OPEN_CHANNEL** command. The **HandleOpenChannelResp** function (line 32) serves as the command handler.
- **0xC4** (line 34) for the **CLOSE_CHANNEL** command. The **HandleCloseChannelResp** function (line 36) serves as the command handler.

A channel layer message that is intended for the server and client of a channel has its own header (**channel_header**). The following fields are used in the **channel_header**:

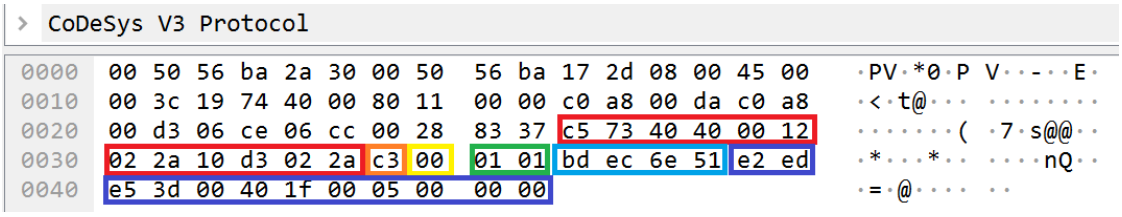
Fields	size (byte)	Type
command_id	1	Uint8_t
Flags	1	Uint8_t
version	2	Uint16_t
checksum	4	Uint32_t
command_data	n	Uint8_t[n]

- The **command_id** field designates the ID of the command and indicates whether the message is a request or a response to a request. A set 7th bit indicates that the message is a response. The other first 6 bits designate the command ID. The following IDs of commands are available for the channel server:
  - **0xc2 (GET_INFO)** refers to an information command for obtaining the number of simultaneously supported channels on the node.
  - **0xc3 (GET_CHANNEL)** refers to a request to create a communication channel between nodes.

- **0xc4 (CLOSE_CHANNEL)** refers to a request to close a communication channel between nodes.

- During our research, we did not detect the use of a value set in the **flags** field**.**
- The **version** field indicates the ID of the command version. Depending on this field, additional fields may be used in the body of the command message.
- The remaining data of a command is determined by the command (**command_id).**

For example, for a request to open a communication channel, the server function **NetServerHandleMetaRequest** processes the following fields and data:

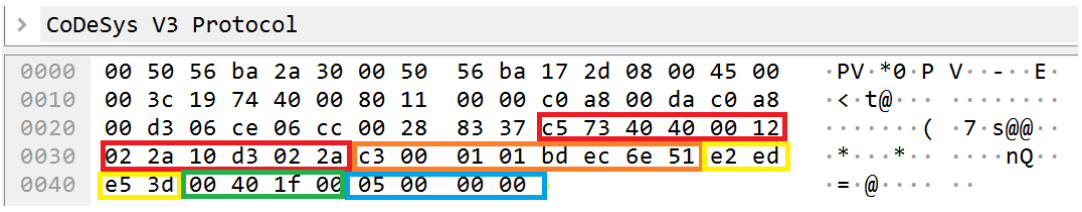


Color						
fields	datagram_ layer_ fields	command_id	Flags	version	checksum	command_data
value	[...]	195 (0xc3) GET_CHANNEL	0	0x101	0x516eecbd	[...]

Utilized fields at the Channel layer

- **command_id** with the ID **0xc3** means that the message is a request to open a communication channel **(GET_CHANNEL).**
- The **flags** field will be ignored when the **GET_CHANNEL** command is processed**.**
- The **version** field determines the availability of additional fields in a message. For the current value, two additional fields will be used.
- **checksum** refers to the packet checksum. The CRC32 algorithm is used for the checksum.
- **command_data** is a field that is examined below.

The following fields and data is used for a **GET_CHANNEL** command request:

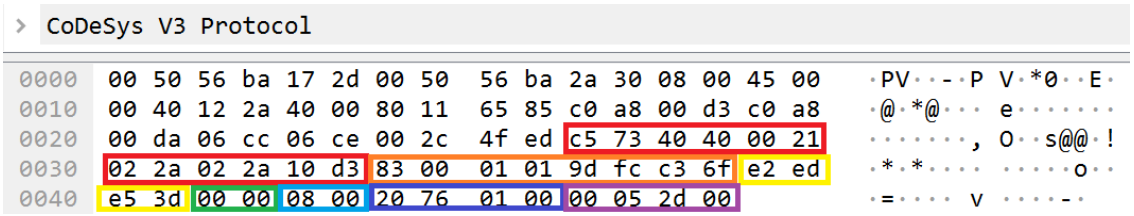


Color					
fields	Datagram_ layer_ fields	channel_header	message_id	receiver_buffer_size	Unknown
value	[...]	[...]	0x3de5ede2	0x1f4000	0x5

Utilized fields of a request with the GET_CHANNEL command

- The **datagram_layer_fields** field refers to fields of the Datagram layer.
- The **channel_header** field refers to the header of a command sent to the channel server.
- **Message_id** refers to the ID of the message. A 4-bit representation of the current time is normally used as the value of this field.
- **Receiver_buffer_size** refers to the maximum permissible amount of data that can be accumulated by the recipient in the communication channel.

In response to this request, the **command_id** field of the **channel_header** sets the 7th bit. The **command_data** fields in the response to the request will be as follows:



Color							
fields	Datagram_ layer_ fields	channel_header	message_id	Reason	channel_id	Reveiver_ buffer_ size	Unknown
value	[...]	[...]	0xe2ede53d	0x00 (OK)	0x08	0x20760100	0x52d00

Utilized fields of a response to the GET_CHANNEL command

- The **datagram_layer_fields** field refers to fields of the Datagram layer.
- The **channel_header** field is the header of a command sent to a channel client.
- **Message_id** refers to the returned message ID. This value is equivalent to the value that was received in the request.
- **Reason** refers to the command processing status.
- **Channel_id** refers to the ID of the open communication channel.
- **Receiver_buffer_size** refers to the maximum permissible amount of data that can be accumulated by the recipient in the communication channel.

Other commands use their own set of fields in the **command_data** field. One exclusion is the **GET_INFO** command**.** To receive the result of this command, all you have to do is send a filled **channel_header** that specifies the ID of the **GET_INFO** command. The response will contain one field:

Fields	size (byte)	Type
Max_channels	4	Uint32_t

- The **Max_channels** field contains the maximum number of simultaneously supported communication channels.

A request to close a channel (**CLOSE_CHANNEL**) uses the following fields in **command_data**:

Fields	size (byte)	Type
channel_id	2	Uint16_t
Reason	2	Uint16_t

- **channel_id** refers to the ID of the channel that needs to be closed.
- **reason** refers to the reason for closing the channel.

The node that received this request does not return any response.

Commands for the communication channel manager

The second examined group of commands works directly with the communication channel manager. An open communication channel is required for querying any command from this group. This group contains commands for transferring data to the next layer, and notifications about its receipt and support for the created communication channel.

Commands for the communication channel manager use the common header (**channel_manager_header)** with the following fields:

Fields	size (byte)	Type
package_type	1	Uint8_t
Flags	1	Uint8_t
packet_data	N	Uint8_t[n]

- **packet_type** refers to the ID of the packet type. The following types of packets and their purposes are identified:
  - **BLK** refers to the transfer of data for the next layer in the protocol stack.
  - **ACK** refers to a notification about data receipt.
  - **KEEPALIVE** refers to keeping a communication channel active.
- **Flags** are additional settings or indicators that are specific to the packet type (**packet_type**).
- **packet_data** refers to the specific data for the packet type (**packet_type**).

The **HandleL4Data** function processes all commands of the examined group.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
01: Removed at the vendor's request
02: {
[...]
17:
20:    Removed at the vendor's request
21:    Removed at the vendor's request
22:    Removed at the vendor's request
23:    Removed at the vendor's request
24:    {
25:      Removed at the vendor's request // ACK
26:        Removed at the vendor's request
27:        Removed at the vendor's request
28:          Removed at the vendor's request
29:        Removed at the vendor's request
30:        Removed at the vendor's request
```

```
31:      Removed at the vendor's request // KEEPALIVE
32:        Removed at the vendor's request
33:        Removed at the vendor's request
34:      Removed at the vendor's request // BLK
35:        Removed at the vendor's request
36:        Removed at the vendor's request
37:          Removed at the vendor's request
38:        Removed at the vendor's request
39:        Removed at the vendor's request
[...]
43:    }
[...]
62:    Removed at the vendor's request
63:    {
[...]
69:      Removed at the vendor's request
72:      Removed at the vendor's request
73:    }
74:    Removed at the vendor's request
75:    {
81:      Removed at the vendor's request
82:    }
[...]
94: } }
```

Fragment of the decompiled pseudocode of the HandleL4Data function

The **HandleL4Data** function (line 01) identifies three possible IDs of **packet_type** for processing **BLK (0x1)**, **ACK (0x2)**, **KEEPALIVE (0x3)**.

Each packet type as specific type of data body. For instance, the **BLK** packet type uses the following fields in a message body:

> CoDeSys V3 Protocol

```
0000  00 50 56 ba 2a 30 00 50  56 ba 17 2d 08 00 45 00   ·PV·*0·P V··-··E·
0010  00 98 2c 5c 40 00 80 11  00 00 c0 a8 00 da c0 a8   ··,\@··· ········
0020  00 d3 06 ce 06 cc 00 84  83 93 c5 73 40 40 00 12   ········ ···s@@··
0030  02 2a 10 d3 02 2a 01 81  20 00 02 00 00 00 01 00   ·*···*·· ·· ·····
0040  00 00 5c 00 00 00 51 40  52 8d 55 cd 10 00 01 00   ··\···Q@ R·U·····
0050  02 00 11 00 00 00 48 00  00 00 00 00 00 00 22 84   ····· H· ····· "·
0060  80 00 01 00 00 00 23 84  80 00 15 14 4e 11 81 01   ····· #· ···· N···
0070  b4 00 10 0e 41 64 6d 69  6e 69 73 74 72 61 74 6f   ···· Admi nistrato
0080  72 00 11 a0 80 00 ce 01  29 3b 20 5f 36 12 18 42   r····· ); _6··B
0090  46 58 f9 75 70 68 4c 54  68 75 77 3f 70 68 76 44   FX·uphLT huw?phvD
00a0  72 2a 87 55 62 52                                  r*·UbR
```

Color									
fields	Datagram_ layer_ fields	packet_type	flags	channel_id	Blk_id	Ack_id	Remaining_ data_ size	Checksum	Remaining_ data
value	[...]	0x01 (BLK)	0x81	32 (0x20)	0x02	0x1	0x5c	0x89	[...]

Utilized fields for the BLK packet type at the Channel layer

- **Packet_type** refers to the **BLK (0x1)** packet type, which indicates data transfer.
- **Flags** refers to packet flags for the **BLK** packet type. The specified value 0x81 means the following:
  - The node that received this packet serves as the server (most significant bit), and the data of the request is the first in the transmission (least significant bit).

  - If the least significant bit is not set, the message contains a continuation of the data of the last packet. The least significant bit of this packet indicates that the packet is sending data to the next layer for the first time.
- **Channel_id** refers to the ID of the open channel used for data transfer.
- **Blk_id** refers to the ID of the current BLK message. This ID is incremented each time by the side that initiated the start of communication over the channel.
- **Ack_id** refers to the ID of the last ACK message. This ID is changed by the responding side each time. After receiving the last packet for data transfer to a service at the application layer, the responding side changes the value of this ID to **Blk_id.**
- **Remaining_data_size** refers to the size of the expected data contained by the **remaining_data** field.
- **Checksum** is the checksum of the data contained in the **remaining_data** field. The CRC32 algorithm is used to calculate the checksum.

If a **BLK** packet contained data whose size does not exceed the maximum size of a CODESYS PDU packet (512 bytes), the response will contain a modified value of the **flags** field in which the most significant bit will not be set, meaning that the recipient is now the client. The value of the **ack_id** field will be changed to the value of the **blk_id** field.

Despite the fact that the maximum size of a CODESYS PDU packet is 512 bytes, very large-sized data can be transmitted over the CODESYS PDU protocol. This is possible through the accumulation of incoming data on the receiving side. The receiving side understands that data needs to be accumulated due to the values in the **flags** field**.** The **checksum** and **remaining_data_size** fields indicate when the packet contains the last data for a command**.**

The **ACK** message type is used for notifying the sending side that a portion of the data was received and the next portion of data is anticipated. This message uses the following fields:

Fields	size (byte)	Type
Channel_id	2	Uint16_t
Blk_id	4	Uint32_t

- **Channel_id** is the ID of the channel used to receive the BLK packet.
- **Blk_id** refers to the value of the **Blk_id** field from the BLK packet that was received by the receiving side.

The **KEEPALIVE** message type is used to keep the open communication channel active. If the channel manager is not receiving messages, it will soon close the channel. The message timeout before the channel is closed is regulated by the component settings.

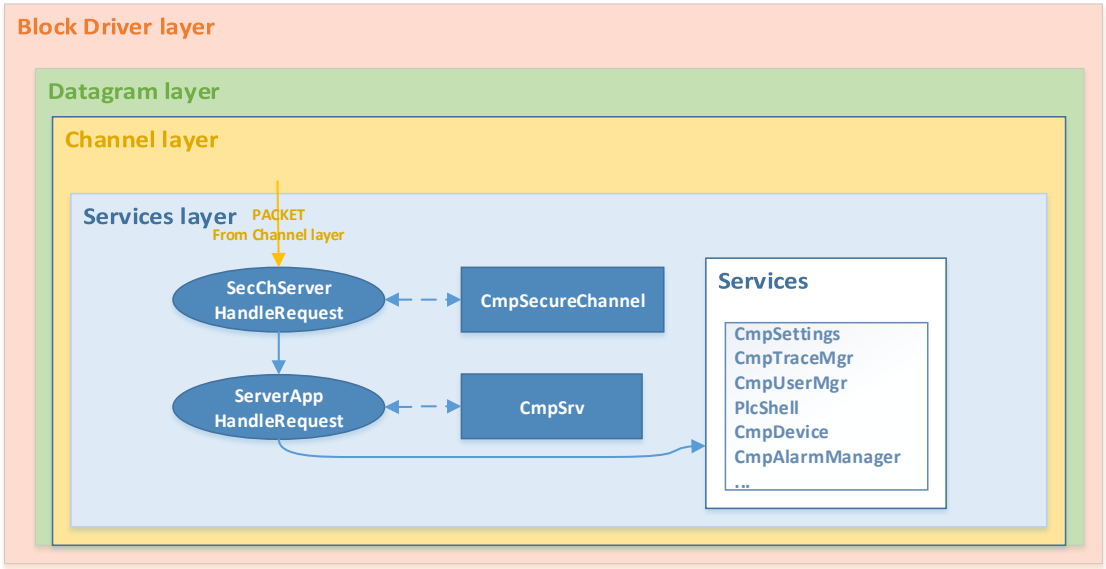
The **KEEPALIVE** message type uses one field:

Fields	size (byte)	Type
Channel_id	2	Uint16_t

- **Channel_id** is the ID of the channel whose time needs to be extended.

Services layer

The next layer in the CODESYS PDU protocol stack is the Services layer.



Schematic representation of how a CODESYS PDU packet is parsed by components at the Services layer

The Services layer represents a combination of several layers of the ISO/OSI model: session layer, presentation layer, and application layer. The main task of this layer is to query the requested service and transmit its operating settings. Additional tasks of the Services layer include encoding, decoding, encrypting, and decrypting data transmitted at this layer. Another additional task is support of sessionization on a device.

The latest implementations of CODESYS Runtime support data encryption at the Services layer. The **CmpSecureChannel** component encrypts and decrypts data at this layer. This occurs in the **SecChServerHandleRequest** function that is exported by it. If data was successfully decrypted or if it was not initially encrypted, it is transmitted to the **ServerAppHandleRequest** function that was exported by the **CmpSrv** component.

If there is no **CmpSecureChannel** component, the **CmpChannelServer** component independently transfers management to the **ServerAppHandleRequest** function.

The format of the message header (**protocol_header**) for an encrypted or unencrypted message is as follows:

Fields	size (byte)	Type
protocol_id	2	Uint16_t
header_size	2	Uint16_t
cmd_group	2	Uint16_t
subcmd	2	Uint16_t
session_id	4	Uint32_t
content_size	4	Uint32_t
additional_data	4	Uint32_t
protocol_data	content_size	Uint8_t[content_size]

- **protocol_id** refers to the ID of the utilized protocol. This ID indicates which protocol handler modified the data and which protocol should be used to transmit data to services. There are two system IDs of the protocol:
  - **HeaderTagProtocol** with the ID **0xcd55.** This protocol ID indicates that data of the **protocol_data** field contains tags.
  - **SecureProtocol** with the ID **0x7557** refers to the protocol for secure data transfer. This ID indicates that data of the **protocol_data** field needs to be decrypted.
- **Header_size** refers to the size of the **protocol_header**. The value of this field does not contain the sizes of previous fields and the current field.
- **service_group** refers to the ID of the queried service. If the most significant bit is set in the ID, this means that the message is a response from a service. Based on the service ID, the following components are identified as a service:
  - **CmpAlarmManager – 0x18;**
  - **CmpApp – 0x2;**
  - **CmpAppBP – 0x12;**
  - **CmpAppForce – 0x13;**
  - **CmpCodeMeter – 0x1d;**
  - **CmpCoreDump – 0x1f;**
  - **CmpDevice – 0x1;**
  - **CmpFileTransfer – 0x8;**
  - **CmpIecVarAccess – 0x9;**
  - **CmpIoMgr – 0xb;**
  - **CmpLog – 0x5;**
  - **CmpMonitor – 0x1b;**
  - **CmpOpenSSL – 0x22;**
  - **CmpSettings – 0x6;**
  - **CmpTraceMgr – 0xf;**
  - **CmpTraceMgr – 0xf;**
  - **CmpUserMgr – 0xc;**
  - **CmpVisuServer – 0x4;**
  - **PlcShell – 0x11;**
  - **SysEthernet – 0x7.**
- **service_id** refers to the ID of the command. This ID determines what exactly the service must do.
- **session_id** refers to the ID of the session. It contains the value of the received session or empty session. This value is checked by protocol handlers and by most commands that require elevated user privileges.
- **content_size** refers to the size of data in the **protocol_data** field**.**
- **additional_data** refers to the field used for additional data.
- **protocol_data** refers to data generated over the utilized protocol (**protocol_id**).

If a message was encrypted by the **SecureProtocol**, almost all fields of the **protocol_header** will contain zero bytes. An exception would be the **header_size** and **content_size** fields**,** which operate normally, and the **protocol_data** field that contains an encrypted **protocol_header**. After the **protocol_data** field is decrypted, the decrypted **protocol_header** will be processed by the **HeaderTagProcol** protocol handler.

If the message was not encrypted and the **HeaderTagProcol** protocol was used**,** the **protocol_data** field will contain **tags**.

A user can register their handler for **protocol_id** using the **ServerRegisterProtocolHandler** function that is exported by the **CmpSrv** component:

```
01: Removed at the vendor's request
02: {
[...]
06:
07:   Removed at the vendor's request
08:   Removed at the vendor's request
09:     Removed at the vendor's request
10:   Removed at the vendor's request
11:   {
12:     Removed at the vendor's request
13:       Removed at the vendor's request
14:   }
15:   Removed at the vendor's request
16:   {
17:     Removed at the vendor's request
18:     {
19:       Removed at the vendor's request
20:       Removed at the vendor's request
21:     }
22:   }
23:   Removed at the vendor's request
24:     Removed at the vendor's request
25:   Removed at the vendor's request
26:   Removed at the vendor's request
27:   Removed at the vendor's request
28:     ++Removed at the vendor's request
29:   Removed at the vendor's request
30: }
```

Decompiled pseudocode of the ServerRegisterProtocolHandler function

The **ServerRegisterProtocolHandler** function is quite simple, and its algorithm consists of the following:

1. At lines 10 through 12, the function compares each of the registered handlers for the **protocol_id** field with the handler that is expected to be registered. If this handler is detected among the registered handlers, the function returns the corresponding status (line 13).
2. Then it attempts to find an unoccupied cell for registering the handler (lines 15:21).
3. It registers the new handler in an unoccupied cell (line 25:26).

For the **service_id** field, one component can register only one handler simultaneously. To register a handler for **service_id**, the **ServerRegisterServiceHandler** function must be queried**.** Its algorithm is analogous to the algorithm of the **ServerRegisterProtocolHandler** function. Therefore, we will not examine it here.

For example, for a non-encrypted request to complete authentication, the **ServerAppHandleRequest** function processes a packet as follows:

```
› CoDeSys V3 Protocol

0000  00 50 56 ba 2a 30 00 50  56 ba 17 2d 08 00 45 00   ·PV·*0·P V··-··E·
0010  00 98 2c 5c 40 00 80 11  00 00 c0 a8 00 da c0 a8   ··,\@··· ········
0020  00 d3 06 ce 06 cc 00 84  83 93 c5 73 40 40 00 12   ········ ···s@@··
0030  02 2a 10 d3 02 2a 01 81  20 00 02 00 00 00 01 00   ·*···*·· ········
0040  00 00 5c 00 00 00 51 40  52 8d 55 cd 10 00 01 00   ··\···Q@ R·U·····
0050  02 00 11 00 00 00 48 00  00 00 00 00 00 00 22 84   ······H· ······"·
0060  80 00 01 00 00 00 23 84  80 00 15 14 4e 11 81 01   ·····#· ····N···
0070  b4 00 10 0e 41 64 6d 69  6e 69 73 74 72 61 74 6f   ····Admi nistrato
0080  72 00 11 a0 80 00 ce 01  29 3b 20 5f 36 12 18 42   r······ ); _6··B
0090  46 58 f9 75 70 68 4c 54  68 75 77 3f 70 68 76 44   FX·uphLT huw?phvD
00a0  72 2a 87 55 62 52                                  r*·UbR
```

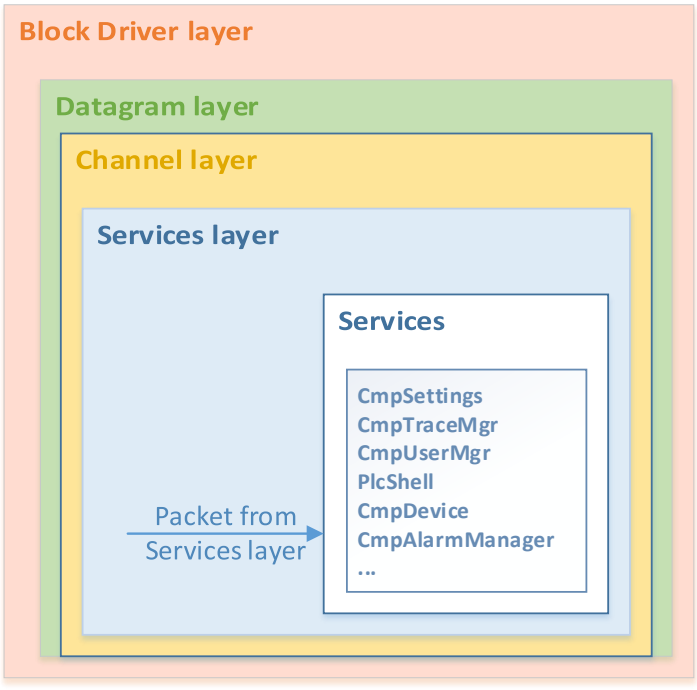
Color						
Fields	datagram_layer_fields	channel_layer	protocol_id	header_size	service_group	service_id
Value	[...]	[...]	0xcd55	0x10	0x1 (**CmpDevice**)	0x2 (**AUTH**)
Color						
Fields	Session_id	protocol_data_size	Additional_data			protocol_data
Value	0x11	0x48	0x0			[...]

Example contents of fields at the Services layer

- **Protocol_id** contains the ID **0xcd55**. This means that the **HeaderTagProtocol** protocol was used**:** data of the **protocol_data** field is not encrypted, and the **service_group** and **service_id** fields contain the values of the requested service.
- **Header_size** contains the value 0x10. This means that the size of the utilized header (**protocol_header**) is 16 (0x10) bytes.
- **Service_group** indicates the ID of the service that was registered by the **CmpDevice** component**.**
- **Service_id** indicates the ID of the requested command for the service. The value 2 indicates that the service registered by the **CmpDevice** component needs to execute the **AUTH** command**.**
- **Protocol_data_size** indicates that the data size of the **protocol_data** field is 72 (0x48) bytes.
- The **additional_data** field was not used.

Tags

The last examined layer in the CODESYS PDU protocol stack is tags.



Schematic representation of how a CODESYS PDU packet is parsed by components at the Services layer

Tags refer to the interface for transmitting settings for services. The service on the client side knows how data needs to be formulated so that the service on the server side correctly receives the settings of this data.

Types of tags

Tags are transmitted in the **protocol_data** field of the **protocol_header.** They can be of two types: a data tag or a parent tag.

Both types of tags have an identical structure, but use different sizes for the first two elements of the structure of fields:

Fields	Parent tag		Data tag	
	size (byte)	Type	Size (byte)	Type
tag_id	2	Uint16_t	1	Uint8_t
tag_size	2	Uint16_t	1	Uint8_t
additional_data	0:10	Uint8_t[0:9]	0:10	Uint8_t[0:9]
tag_data	Tag_size	Uint8_t[tag_size]	Tag_size	Uint8_t[tag_size]

- **tag_id** refers to the tag ID. The IDs of a parent tag and data tag are distinguished by the value of the most significant bit. If the value of the most significant bit is set, this means that the tag is a parent tag and all other data is typical for a parent tag. Otherwise the tag is a data tag, and its data contains the final settings for the service.
- **tag_size** refers to the size of the data. This field determines the amount of data in the **tag_data** field. In addition, the value of the most significant bit of the **tag_size** field determines the availability of the additional field **additional_data:** if the value of the most significant bit is set, the **additional_data** field is available.
- **additional_data** is an additional field. It has a dynamic size, which cannot be larger than 10 bytes. The end of this field is determined by a zero byte.
- **tag_data** refers to data of the parent tag or data of the data tag.

Data extracted by a service from a data tag is converted into a specific type of data. For example, a tag containing 4 bytes in the **tag_data** field may be converted to one of the numerical data types by the service. The variable type is not transmitted in the tag structure. However, may times it was observed that CODESYS Runtime services transmit a group of tags from which one tag may contain a value, the second tag – type ID, and the third – size. These bundles of tags are usually combined under one parent tag.

A parent tag is used for linking several types into one logical element. An example of this type of linking was presented above. Another example is the linking of tags with a user name and password that are used for authentication. A data tag containing a user name and a data tag containing a password may be combined into one parent tag.

Processing tags

Let's examine how a specific service handler extracts settings from received tags based on an example packet containing tags for user authentication. In this case, tags will be analyzed as follows:

```
> CoDeSys V3 Protocol
```

```
0000  00 50 56 ba 2a 30 00 50  56 ba 17 2d 08 00 45 00   ·PV·*0·P V··-··E·
0010  00 98 2c 5c 40 00 80 11  00 00 c0 a8 00 da c0 a8   ··,\@··· ········
0020  00 d3 06 ce 06 cc 00 84  83 93 c5 73 40 40 00 12   ········ ···s@@··
0030  02 2a 10 d3 02 2a 01 81  20 00 02 00 00 00 01 00   ·*···*·· ········
0040  00 00 5c 00 00 00 51 40  52 8d 55 cd 10 00 01 00   ··\···Q@ R·U·····
0050  02 00 11 00 00 00 48 00  00 00 00 00 00 00 22 84   ······H· ······"·
0060  80 00 01 00 00 00 23 84  80 00 15 14 4e 11 81 01   ······#· ····N···
0070  b4 00 10 0e 41 64 6d 69  6e 69 73 74 72 61 74 6f   ····Admi nistrato
0080  72 00 11 a0 80 00 ce 01  29 3b 20 5f 36 12 18 42   r······ ); _6··B
0090  46 58 f9 75 70 68 4c 54  68 75 77 3f 70 68 76 44   FX·uphLT huw?phvD
00a0  72 2a 87 55 62 52                                  r*·UbR
```

Color								
fields	datagram_ layer_ fields	channel_ layer_ fields	Services_ layer_ fields	Data_tag_1	Data_tag_2	Parent_tag_1	Data_tag_3	Data_tag_4

Example parsing of the contents of the protocol_data field for tags in an authentication request

There is a **protocol_data** field inside the **services_layer_fields** header. If this field is broken down into tags, it has 4 data tags and 1 parent tag.

The hierarchy of tags will be as follows:

- Data_tag_1
- Data_tag_2
- Parent_tag_1
    - Data_tag_3
    - Data_tag_4

Data_tag_1 and Data_tag_2 are at the same level as Parent_tag_1.

Data_tag_3 and Data_tag_4 are within the parent tag.

Interaction with tags occurs through the utilization of a multitude of API functions, which include functions for interaction with incoming tags with the BTagReader (Binary Tag Reader) prefix, and functions for interaction with outgoing tags with the BtagWriter (Binary Tag Writer) prefix.

Main API functions for interaction with tags:

For incoming tags

1. **BTagReaderInit** refers to initialization of the structure for reading received data.
   The data read structure stores the following elements: data, current position in the data, possible end position of data, and data size. In most cases, all functions for interaction with incoming tags use only the pointer to the current position in data.
   This is how it moves through tags in pure data.
2. **BTagReaderGetTagId** refers to getting the tag ID.
3. **BTagReaderGetContent** refers to getting the tag data.
4. **BTagReaderMoveNext** refers to moving to the next tag.
5. **BTagReaderSkipContent** refers to moving to the end of data of the current tag.

For outgoing tags

1. **BTagWriterInit** refers to initialization of the structure for writing outgoing data. The data write structure stores the following elements: initially received data, pointer to the end of data, and current size of data. All functions for interaction with outgoing tags change all elements of the structure.
2. **BTagWriterStartTag** refers to opening a new tag for outgoing data. Opening a tag must be accompanied by a query of the function for closing the tag – **BTagWriterEndTag**.
3. **BTagWriterAppendBlob** refers to adding data for a created tag.
4. **BTagWriterEndTag** refers to closing the created data tag.
5. **BTagWriterFinish** refers to finishing writing tags as outgoing data. Essentially, this function verifies that all added tags have a valid structure and were closed by the **BTagWriterEndTag** function.

The packet has a **services_layer_fields** header in which the value of the **service_group field** is equal to 1. This ID is registered by the **CmpDevice** component. During its initialization, the component registers the **DeviceServiceHandler** function as a service by settings its ID equal to 1:

```
Call trace:
       Removed at the vendor's request

Pseudocode:
1: Removed at the vendor's request
2: {
3:   Removed at the vendor's request
4: }
```

Decompiled pseudocode of the DeviceSrvInitComm function of the CmpSrv component

In this packet, the command specified in the command ID field (**service_id)** is equal to 2. The **DeviceServiceHandler** function identifies 9 commands that include a command with the ID 2 (line 254):

```
001: Removed at the vendor's request
002: {
[...]
133:   Removed at the vendor's request
134:   {
135:     Removed at the vendor's request
[...]
254:     Removed at the vendor's request
[...]
399:     Removed at the vendor's request
[...]
411:     Removed at the vendor's request
[...]
473:     Removed at the vendor's request
[...]
548:     Removed at the vendor's request
[...]
611:     Removed at the vendor's request
[...]
624:     Removed at the vendor's request
[...]
695:     Removed at the vendor's request
[...]
763: }
```

Fragment of the decompiled pseudocode of the DeviceServiceHandler function

The **DeviceServiceHandler** function executes a command with the ID 2 in three steps:

1. Extracts the settings and sets the local values.
2. Executes commands with the local values and the received settings.
3. Returns the result of command execution.

```
001: Removed at the vendor's request
002: {
[...]
129:   Removed at the vendor's request
130:   Removed at the vendor's request
131:   Removed at the vendor's request
132:   Removed at the vendor's request
133:   Removed at the vendor's request
134:   {
[...]
254:     Removed at the vendor's request
[...]
266:       Removed at the vendor's request
267:       Removed at the vendor's request
268:       {
269:         Removed at the vendor's request
270:         Removed at the vendor's request
271:         {
272:           Removed at the vendor's request
273:             Removed at the vendor's request
274:             Removed at the vendor's request
275:           Removed at the vendor's request
276:             Removed at the vendor's request
277:             Removed at the vendor's request
278:             {
279:               Removed at the vendor's request
280:               Removed at the vendor's request
281:               {
282:                 Removed at the vendor's request
283:               }
284:               Removed at the vendor's request
285:               {
286:                 Removed at the vendor's request
287:                 Removed at the vendor's request
288:               }
289:               Removed at the vendor's request
290:               {
291:                 Removed at the vendor's request
292:               }
293:               Removed at the vendor's request
294:               Removed at the vendor's request
295:               Removed at the vendor's request
296:             }
297:             Removed at the vendor's request
298:           Removed at the vendor's request
299:             Removed at the vendor's request
300:             Removed at the vendor's request
301:           Removed at the vendor's request
302:             Removed at the vendor's request
303:             Removed at the vendor's request
304:         }
305:         Removed at the vendor's request
306:         Removed at the vendor's request
307:         Removed at the vendor's request
308:       }
[...]
315:       Removed at the vendor's request
316:       {
317:         Removed at the vendor's request
318:         Removed at the vendor's request
[...]
327:       }
330:       Removed at the vendor's request
331:       Removed at the vendor's request
332:       {
```

```
333:          Removed at the vendor's request
334:        }
335:        Removed at the vendor's request
336:        {
337:          Removed at the vendor's request
338:          Removed at the vendor's request
339:          {
[...]
341:            Removed at the vendor's request
342:            Removed at the vendor's request
343:            Removed at the vendor's request
344:            Removed at the vendor's request
345:            Removed at the vendor's request
346:            Removed at the vendor's request
347:            Removed at the vendor's request
348:            Removed at the vendor's request
349:          }
[...]
357:          Removed at the vendor's request
[...]
365:          Removed at the vendor's request
366:        }
367:        Removed at the vendor's request
[...]
376:        Removed at the vendor's request
377:        Removed at the vendor's request
378:        Removed at the vendor's request
379:        Removed at the vendor's request
380:        Removed at the vendor's request
[...]
389:        Removed at the vendor's request
390:        Removed at the vendor's request
391:        Removed at the vendor's request
392:        Removed at the vendor's request
393:        Removed at the vendor's request
394:        Removed at the vendor's request
395:        Removed at the vendor's request
396:        Removed at the vendor's request
397:        Removed at the vendor's request
398:        Removed at the vendor's request
[...]
761:    }
762:    Removed at the vendor's request
763: }
```

Fragment of the decompiled pseudocode of the DeviceServiceHandler function

The algorithm for a command with the ID 2 for each step is as follows:

Settings extraction step

1. Line 131 involves initialization of the structure for writing outgoing data (**writer**), will be used at the results return step. Line 132 involves initialization of the structure for reading incoming data (**reader**), which is used at the current step.
2. There is an attempt to recognize tags within incoming data (line 266). If successful, the tag ID is extracted from the first tag (line 269).
3. Depending on the tag ID received at the previous step, tag data is written to the corresponding variables. For the ID 0x23 (line 272), data is written to the **pulChallenge** variable (line 273). For the ID 0x22 (line 298), data is written to the **pulCrypeType** variable (line 299). If a tag ID is equal to 0x81 (line 275), the tag is a parent tag and it is searched for data tags (line 279) with the ID of 16 (line 280). Data from the found tags is written to the **user_name** variable (line 282). From the found tag with the ID 17 (line 284), data is written to the **encrypted_password** variable (line 286).
4. Line 315 involves verifying that the variables necessary for executing the command have been filled with data from tags.

Command execution step

1. The received **encrypted_password**, **pulCrypeType** and **pulChallenge** variables are used in the function for decrypting the **UserMgrDecryptPassword** (line 318). The decrypted password will be written to the **decrypted_password** variable.
2. The **user_name** variable will be used in the function for checking the existence of **FindUser** users. This function searches for an entry regarding users in the database.
3. If an entry is found about a user that has a name from the **user_name** variable, the user's password is checked to see if it matches the decrypted password that was saved to the **decrypted_password** variable.
4. If the decrypted password matches the password from the database, the current user is checked for permissions to a "Device" object (line 357).
5. If the user has permissions to a "Device" object, the generated session ID (**ulSessionId** variable) is assigned to the current user (line 365) and to the utilized communication channel (line 367).

Results return step

1. A tag with the ID 0x82 (line 376) is opened for outgoing data. This tag is a parent tag, and within it the tags with an ID of 0x20 (opening at line 377 and closing at line 379), 0x24 (opening at line 389 and closing at line 391) and 0x21 (opening at line 392 and closing at line 394) are sequentially opened and closed.
2. The following values are written to data tags: with the ID 0x20 – value of the command execution status (line 378); with the ID 0x24 – value of device settings (line 390); with the ID 0x21 – value of the generated session (line 393).
3. At line 395, the parent tag with the ID 0x82 is closed and writing of outgoing data is finished (line 396).

If a user entry is not found, permissions are insufficient, passwords do not match, or data is incorrect, the corresponding response is generated at the results return step.

In response to a request for CODESYS Runtime authentication with the correct user authentication data, the following packet is returned:

```
> CoDeSys V3 Protocol
0000  00 50 56 ba 17 2d 00 50  56 ba 2a 30 08 00 45 00   ·PV··-·P V·*0··E·
0010  00 68 1e 03 40 00 80 11  59 84 c0 a8 00 d3 c0 a8   ·h··@··· Y·······
0020  00 da 06 cc 06 ce 00 54  17 b6 c5 73 40 40 00 21   ·······T ···s@@·!
0030  02 2a 02 2a 10 d3 01 01  1c 00 02 00 00 00 02 00   ·*·*···· ········
0040  00 00 2c 00 00 00 70 63  90 23 55 cd 10 00 81 00   ··,···pc ·#U·····
0050  02 00 11 00 00 00 18 00  00 00 00 00 00 00 82 01   ········ ········
0060  94 00 20 02 00 00 24 84  80 00 0f 00 00 00 21 84   ·· ···$· ·····!·
0070  80 00 12 f9 3e 9f                                 ····>·
```

Color fields	datagram_ layer_ fields	channel_ layer_ fields	Services_ layer_ fields	Parent_tag	Tag with status	Tag with settings	Tag with **session_id**

Example parsing of the contents of the protocol_data field for tags in a response to an authentication request

Consequently, we can unequivocally determine which tags with which IDs will be used to transmit specific settings for services. The main settings in this response will be contained in a tag with the ID 0x21. This tag will contain the session ID that will later be used as the value of the **session_id** field at the Services layer.

Detected vulnerabilities and potential attacks

After we created the conditions for conducting a static and dynamic analysis and peeled back the layers of the protocol, we were able to search for vulnerabilities.

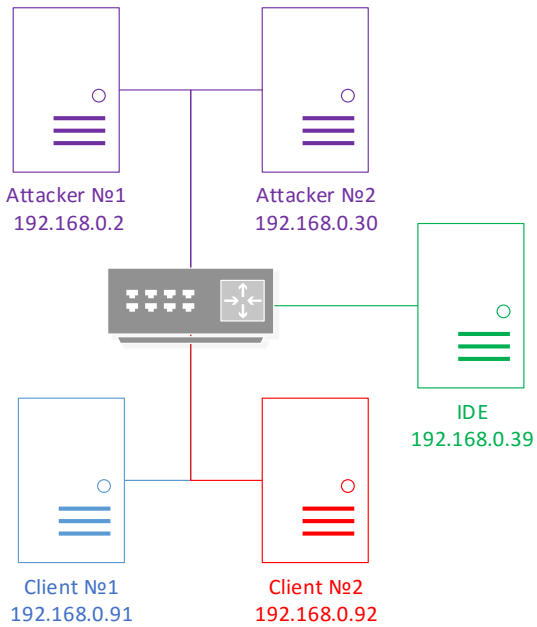
This chapter will examine several detected logical vulnerabilities that could be exploited to take over a device on which CODESYS Runtime software is installed.

Based on the results of our research, we also plan to publish an article devoted to an automatic search of network binary vulnerabilities in conditions when the source code is not available.

Description of the testing bench

To demonstrate the exploitation of vulnerabilities, we will use a testing bench consisting of the following network nodes:

1. Attacking computers with network addresses 192.168.0.2 and 192.168.0.30. The attacking nodes are designated as Attacker №1 and Attacker №2.
2. Raspberry Pi device running CODESYS Control For Raspberry PI that has the network address 192.168.0.91. The abbreviated name Client №1 will be used hereinafter for this node.
3. Raspberry Pi device running CODESYS Control For Raspberry PI that has the network address 192.168.0.92. The abbreviated name Client №2 will be used hereinafter for this node.
4. Computer with CODESYS Development System installed and the network address 192.168.0.39. The abbreviated name IDE will be used for this node.



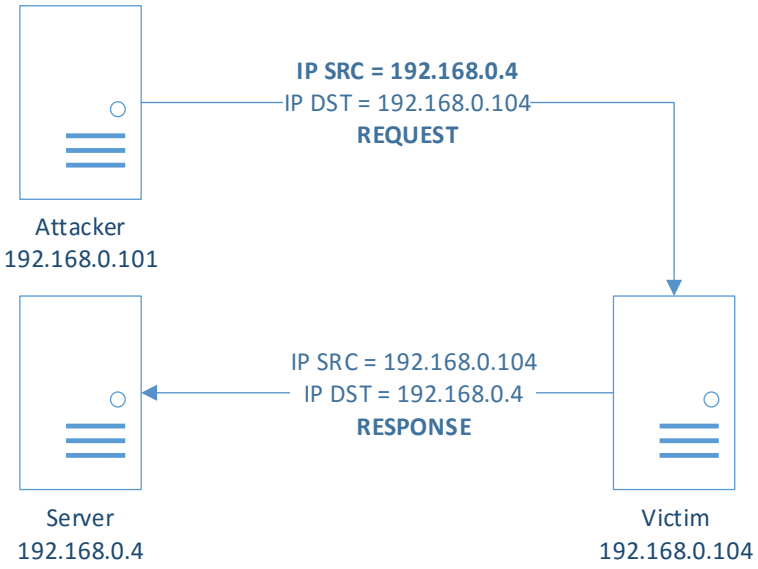
Test bench schematic

Attacks at the Datagram layer

The CODESYS PDU protocol is based on the ISO/OSI model. Therefore, we can presume that, along with the concept of the ISO/OSI model and its stack of protocols, the CODESYS PDU protocol also inherited the shortcomings of this model and the security threats corresponding to these shortcomings.

IP spoofing

Each protocol in the ISO/OSI model has its own set of threats. IP spoofing refers to an attack on the ISO/OSI model at the network layer consisting of forging the address of the message source (IP SRC) for the purpose of concealing the sender's address. The victims who receive such a packet will process the request and return a response to the source address indicated in the field (IP SRC).



Schematic of an IP spoofing attack for the ISO/OSI model

The sender's address is concealed for the purpose of deceiving security systems and hindering discovery of the attack.

Attacks analogous to IP spoofing can be conducted over the CODESYS PDU protocol.

The following two attacks on the CODESYS PDU protocol will be examined:

1. Attack aimed at concealing the address of the message source.
2. Attack aimed at taking control over the existing channel of communication between nodes of the CODESYS network.

Attack aimed at concealing the address of the message source

The **CmpRouter** component processes fields at the Datagram layer. The header of the datagram layer contains fields for addressing, packet parameterization, channel layer service ID, and others. Among the addressing fields is the **receiver** field, which indicates to which address a response to a request should be sent.

```
> Frame 8: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interfa
> Ethernet II, Src: PcsCompu_90:85:bf (08:00:27:90:85:bf), Dst: Raspberr_92:3a:
> Internet Protocol Version 4, Src: 192.168.0.39, Dst: 192.168.0.92
> User Datagram Protocol, Src Port: 1742, Dst Port: 1740
> CoDeSys V3 Protocol
```

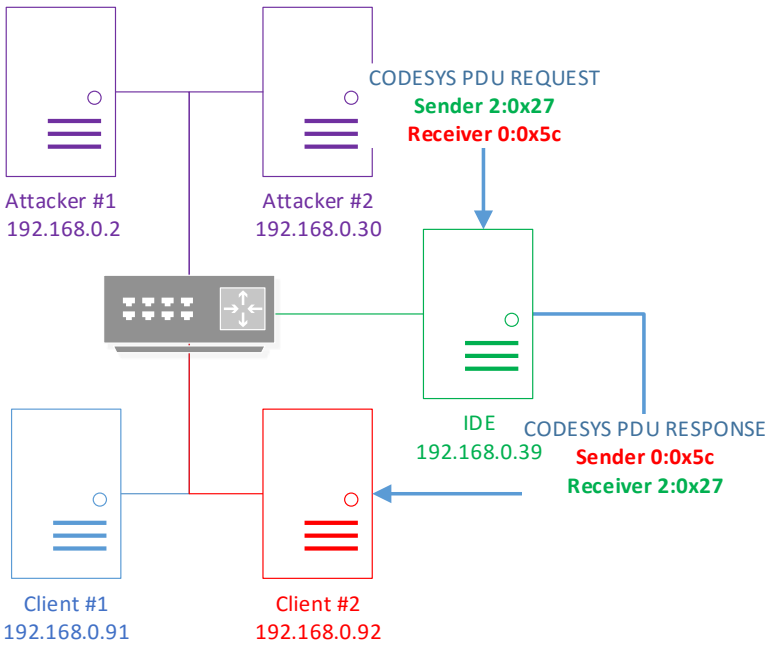
```
0000  b8 27 eb 92 3a ff 08 00  27 90 85 bf 08 00 45 00   ·'··:··· '·····E·
0010  00 3c 02 03 00 00 80 11  b6 da c0 a8 00 27 c0 a8   ·<····· ·····'··
0020  00 5c 06 ce 06 cc 00 28  a1 f8 c5 73 40 40 00 11   ·\·····( ···s@@··
0030  00 5c 02 27 00 00 c3 00  01 01 df db 21 ab a5 ed   ·\·'···· ····!···
0040  37 39 00 40 1f 00 04 00  00 00                     79·@···· ··
```

Color				
fields	Sender		**Receiver**	
	Port index	Address	Port index	Address
value	0 (1740)	0x5c (192.168.0.92)	2 (1742)	0x27 (192.168.0.39)

Location of the receiver field in the header of the Datagram layer

As evident from the example data stream presented above, the **receiver** field contains the value of the last bit of the numerical representation of the IDE node address (0x27) and the value of the port index equal to 2. Both of these values match the numerical value of the message source address (the src field, which is equal to 192.168.0.39) and the value of the port from which the request was sent (the src port field, which is equal to 1742).

By changing the value in the **receiver** field, an attacker can implement a classic IP spoofing attack.



Schematic of a classic IP spoofing attack over the CODESYS PDU protocol

The CODESYS PDU protocol has another architectural vulnerability that could be exploited to implement an advanced IP spoofing attack. It is based on routing, which is one of the responsibilities of the **CmpRouter** component.

```
> Frame 8: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interfac
> Ethernet II, Src: PcsCompu_90:85:bf (08:00:27:90:85:bf), Dst: Raspberr_92:3a:f
> Internet Protocol Version 4, Src: 192.168.0.39, Dst: 192.168.0.92
> User Datagram Protocol, Src Port: 1742, Dst Port: 1740
> CoDeSys V3 Protocol
```

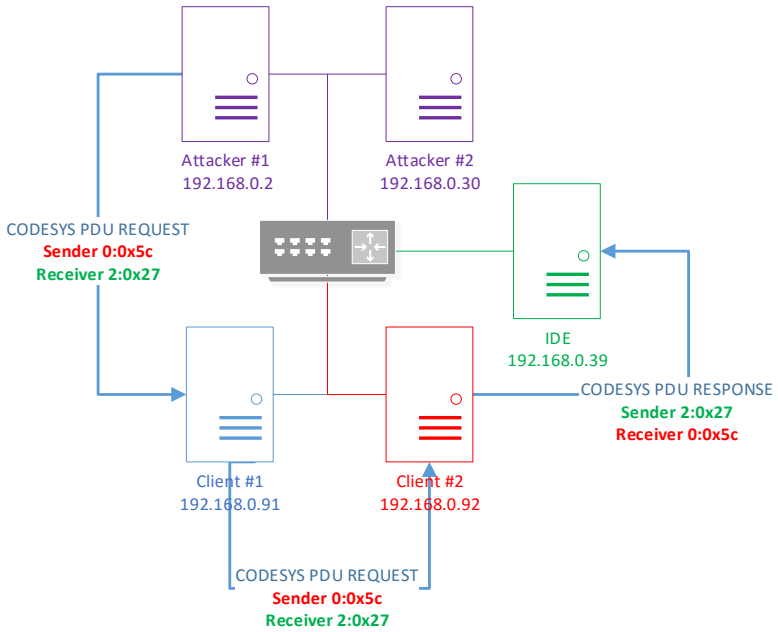
```
0000  b8 27 eb 92 3a ff 08 00  27 90 85 bf 08 00 45 00   ·'··:··· '·····E·
0010  00 3c 02 03 00 00 80 11  b6 da c0 a8 00 27 c0 a8   ·<······ ·····'··
0020  00 5c 06 ce 06 cc 00 28  a1 f8 c5 73 40 40 00 11   ·\·····( ···s@@··
0030  00 5c 02 27 00 00 c3 00  01 01 df db 21 ab a5 ed   ·\·'···· ····!···
0040  37 39 00 40 1f 00 04 00  00 00                     79·@···· ··
```

Color				
fields	Sender		Receiver	
	Port index	Address	Port index	Address
value	0 (1740)	0x5c (192.168.0.92)	2 (1742)	0x27 (192.168.0.39)

Location of the sender field in a header of the Datagram layer

The **sender** field indicates the address of the node for which the packet is intended. The **CmpRouter** component redirects a received CODESYS PDU packet to a different node in the network corresponding to the one indicated in the **sender** field if the value of the **sender** field does not match the address of the node that received the packet.

By manipulating the **sender** field, an attacker can modify the IP spoofing attack by adding an intermediate node. The intermediate node will serve as a proxy for redirecting a malicious packet to other nodes in the CODESYS network.



Schematic of a modified IP spoofing attack over the CODESYS PDU protocol

The finishing stroke of the IP spoofing attack over the CODESYS PDU protocol will be the concealed receipt of a response to the redirected request. The receipt of the response can be concealed by specifying a broadcast address as the response recipient in the message.

```
> Frame 1444: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
> Ethernet II, Src: D-LinkIn_ad:fb:c0 (9c:d6:43:ad:fb:c0), Dst: Raspberr_92:3a:ff (b8:27:eb:92:3a:ff)
> Internet Protocol Version 4, Src: 192.168.0.30, Dst: 192.168.0.92
> User Datagram Protocol, Src Port: 1740, Dst Port: 1740
> CoDeSys V3 Protocol
```

```
0000  b8 27 eb 92 3a ff 9c d6  43 ad fb c0 08 00 45 00   ·'··:··· C·····E·
0010  00 3c f6 09 00 00 80 11  00 00 c0 a8 00 1e c0 a8   ·<·····  ········
0020  00 5c 06 cc 06 cc 00 28  82 04 c5 73 40 40 00 11   ·\·····( ···s@@··
0030  00 5c 00 ff 00 00 c3 00  01 01 bf 19 68 af 00 00   ·\·····  ····h···
0040  00 00 00 40 1f 00 04 00  00 00                     ···@····  ··
```

Color fields	Sender		Receiver	
	Port index	Address	Port index	Address
value	0 (1740)	0x5c (192.168.0.92)	0 (1740)	0xff (192.168.0.255)

Example packet in which a broadcast address is indicated as the value for the receiver field in the Datagram layer header

After receiving a request in which the last byte of a broadcast address (0xff) is indicated as the response recipient in the **receiver** field, the node will return a response to the broadcast address.

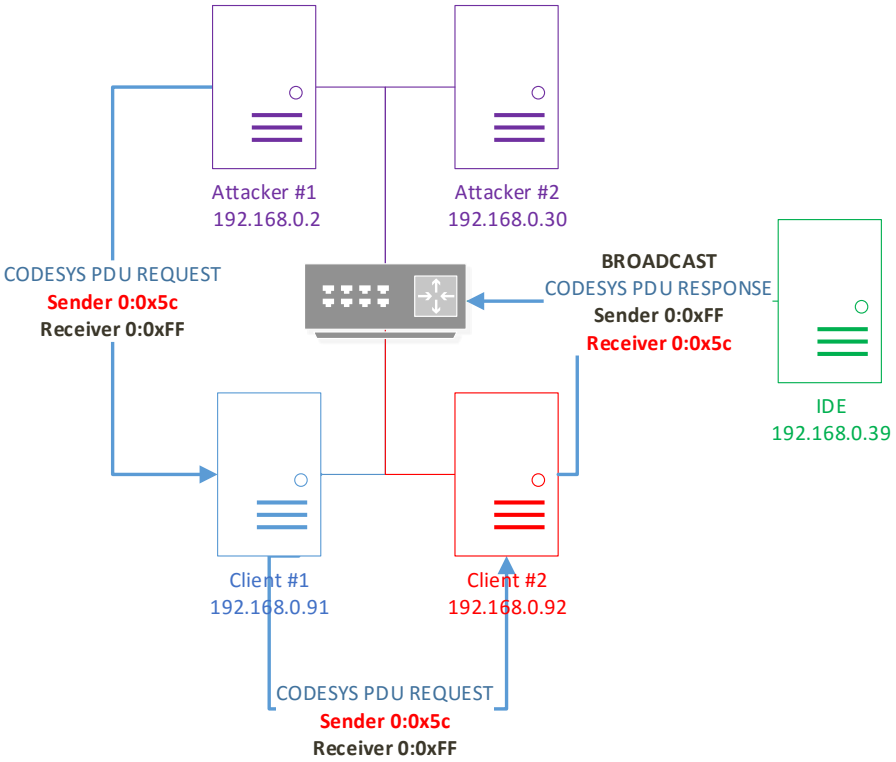
```
1444 235.811028   192.168.0.30    192.168.0.92      CODESY…   74 1740 → 1740 Len=32
1445 235.813098   192.168.0.92    192.168.0.255     CODESY…   78 1740 → 1740 Len=36
1446 235.816995   192.168.0.30    192.168.0.92      CODESY…  110 Device.GetTargetIdent
```

```
> Frame 1445: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface 0
> Ethernet II, Src: Raspberr_92:3a:ff (b8:27:eb:92:3a:ff), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 192.168.0.92, Dst: 192.168.0.255
> User Datagram Protocol, Src Port: 1740, Dst Port: 1740
> CoDeSys V3 Protocol
```

```
0000  ff ff ff ff ff ff b8 27  eb 92 3a ff 08 00 45 00   ·······' ··:···E·
0010  00 40 5d ce 40 00 40 11  5a 33 c0 a8 00 5c c0 a8   ·@]·@·@· Z3···\··
0020  00 ff 06 cc 06 cc 00 2c  1b f7 c5 73 40 40 00 11   ·······, ···s@@··
0030  00 ff 00 5c 00 00 83 00  01 01 2f da 45 dd 00 00   ···\····  ··/·E···
0040  00 00 00 00 08 00 20 76  01 00 00 04 29 08        ······ v ····)·
```

Example of a node sending a response to a broadcast address

The attacker can conceal the address of its node by receiving responses of the victim's nodes from the broadcast node.



Schematic representation of an IP spoofing attack over the CODESYS PDU protocol with concealed receipt of a response to a request

By automating the exploitation of detected vulnerabilities and implementing the described scheme of attack, an attacker could make it more difficult for analysts to investigate an attack.

Attack aimed at taking control of an existing channel of communication between nodes of the CODESYS network

In light of the fact that the **CmpRouter** component performs its functions based only on data in the CODESYS PDU packet, an attacker can infiltrate the existing communications between nodes. However, because each layer of the CODESYS PDU protocol is dependent on the previous layer, control of the last layer (Services layer) can be taken over only by taking control at all previous layers.

To interact at the Channel layer, network participants have to establish a communication channel. To query most services at the Services layer, one node must complete authentication on the other node and receive a session ID. The value of the channel ID is transmitted in the header at the Channel layer. The value of the session ID is transmitted in the header of the Services layer.

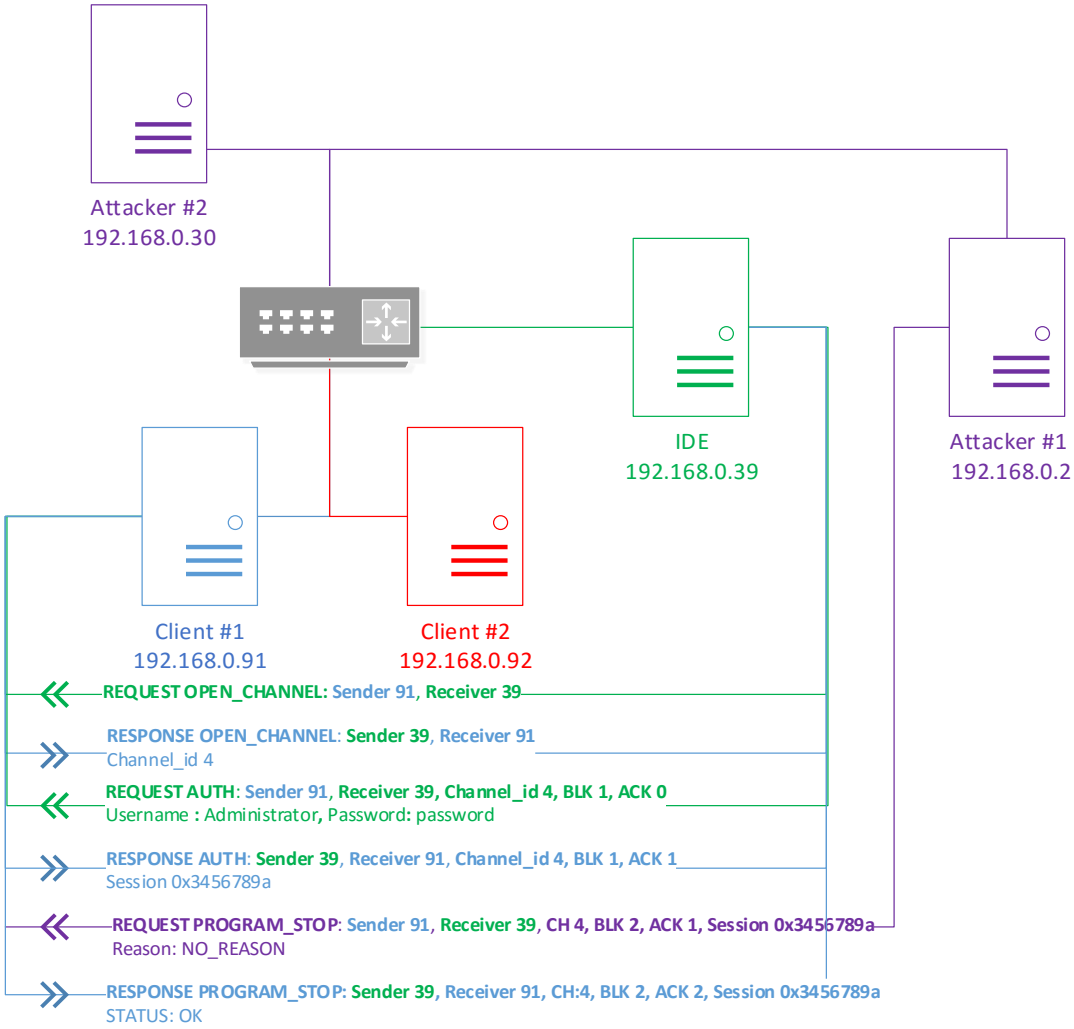
Taking control in the CODESYS PDU protocol can be divided into several tasks:

1. Receiving addresses of communication participants
2. Receiving the channel ID
3. Receiving the BLK and ACK ID
4. Receiving the session ID

The first task can be accomplished using the standard capabilities of the CODESYS PDU protocol. The third is accomplished by finding out IDs.

To accomplish the second and fourth tasks, you must know and exploit several detected vulnerabilities.

If an attacker has detected the vulnerabilities that we found and can automate an attack, he could implement the following attack scenario:



Scenario of attack aimed at intercepting control

The attack algorithm may be as follows:

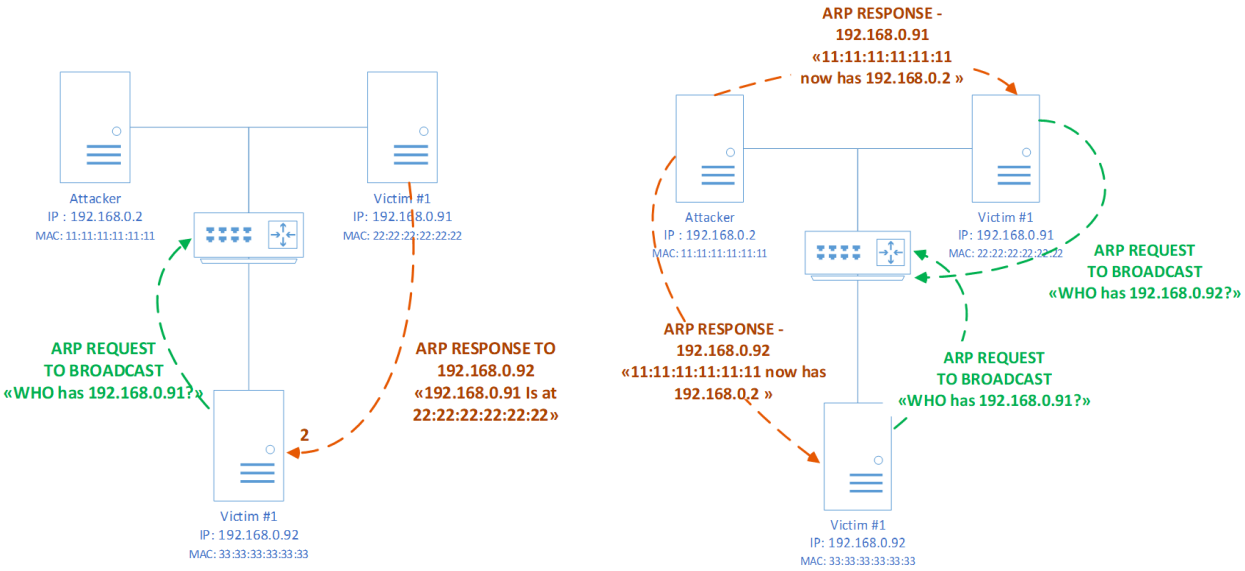
1. **OPEN_CHANNEL request:**
   The IDE node requests open channels on the Client #1 node.
2. **OPEN_CHANNEL response**:
   The Client #1 node opens a communication channel for the IDE node and sends it the channel ID (channel_id) equal to 4.
3. **AUTH request**:
   The IDE node is authenticated on the Client #1 node by sending the password and user name.
4. **AUTH response**:
   The Client #1 node searches the database for an entry on the received user name. After finding the entry and comparing the password, Client #1 returns a session ID equal to 0x3456789a to the IDE node.

5. **PROGRAM_STOP request**:
   An attacker from the Attacker #1 node sends a message with the command PROGRAM_STOP to the Client #1 node. This packet must indicate an incremental BLK ID from the last message of the IDE node and an ACK ID analogous to the ID in the last message of the IDE node. The IDs of the session (session_id) and channel (channel_id) match the IDs used in the communication between the nodes.
6. **PROGRAM_STOP response**:
   The Client #1 node processes the request containing the PROGRAM_STOP command received from the Attacker #1 node as if the request came from the IDE node. This is because the received ID of the communication channel (channel_id) matches the existing ID of the communication channel between the IDE node and Client #1 node, the received BLK and ACK IDs are correct for the utilized channel, and the session ID is available. As a result, the Client #1 node returns a positive response about stopping the program.

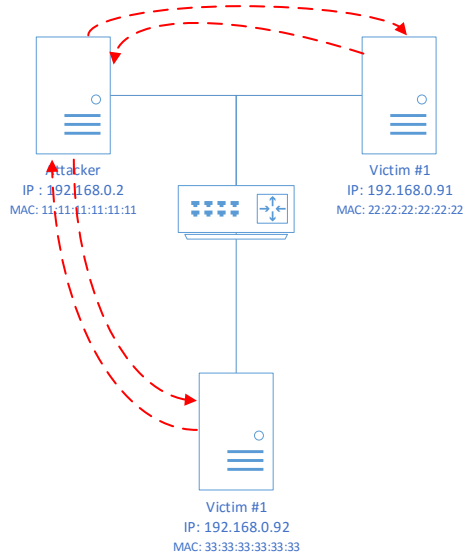
Setting an arbitrary parent node

Network traffic interception is one of the threats for the data link layer of the ISO/OSI model. An attack that implements the threat of network traffic interception is called a "Man-in-the-middle" (MITM) attack.

In the ARP protocol, which operates at the data link layer of the ISO/OSI model, one of the implementations of a man-in-the-middle attack is ARP poisoning. This attack consists of modifying the ARP table on the victim's computer by sending specially generated ARP responses to network nodes of victims. After modifications are made to the ARP table, all outbound and inbound traffic of the victim will pass through the network address of the attacker.



Schematic of an ARP poisoning attack over the ARP protocol. Step 1 is modification of the ARP table



Schematic of an ARP poisoning attack over the ARP protocol. Step 2 is the modified route of network traffic

CODESYS Runtime does not have an ARP table. However, it has a mechanism for changing the route of a CODESYS network for which the **CmpRouter** component is responsible.

The address of any node in a CODESYS network consists of the addresses of all previous parent nodes and its own address as the terminal address. A CODESYS network node generates and remembers its full address after receiving a series of requests for the address service. After generating an address, the node will send all outgoing packets to the source of the request, assuming that it is the parent node.

```
    1 0.000000    192.168.0.39    192.168.0.255    CODESYSV3    62 1742 → 1743 Len=20
    2 0.003809    192.168.0.92    192.168.0.39     CODESYSV3   297 1740 → 1742 Len=255
    3 20.131552   192.168.0.30    192.168.0.255    CODESYSV3   119 1740 → 1740 Len=77
    4 60.022305   192.168.0.39    192.168.0.255    CODESYSV3    62 1742 → 1743 Len=20
    5 60.026594   192.168.0.92    192.168.0.30     CODESYSV3   297 1740 → 1740 Len=255
```

```
> Frame 1: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface 0
> Ethernet II, Src: PcsCompu_90:85:bf (08:00:27:90:85:bf), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 192.168.0.39, Dst: 192.168.0.255
> User Datagram Protocol, Src Port: 1742, Dst Port: 1743
> CoDeSys V3 Protocol
```

```
0000  ff ff ff ff ff ff 08 00  27 90 85 bf 08 00 45 00   ........ '.....E.
0010  00 30 00 e3 00 00 80 11  b7 63 c0 a8 00 27 c0 a8   .0...... .c...'..
0020  00 ff 06 ce 06 cf 00 1c  5e 66 c5 74 40 03 00 10   ........ ^f.t@...
0030  63 d3 02 27 00 00 02 c2  00 04 a2 f3 00 00        c..'.... ......
```

Color				
Network address	192.168.0.255 (Broadcast)	192.168.0.39	192.168.0.92	192.168.0.30
Host		IDE	Client #1	Attacker #2

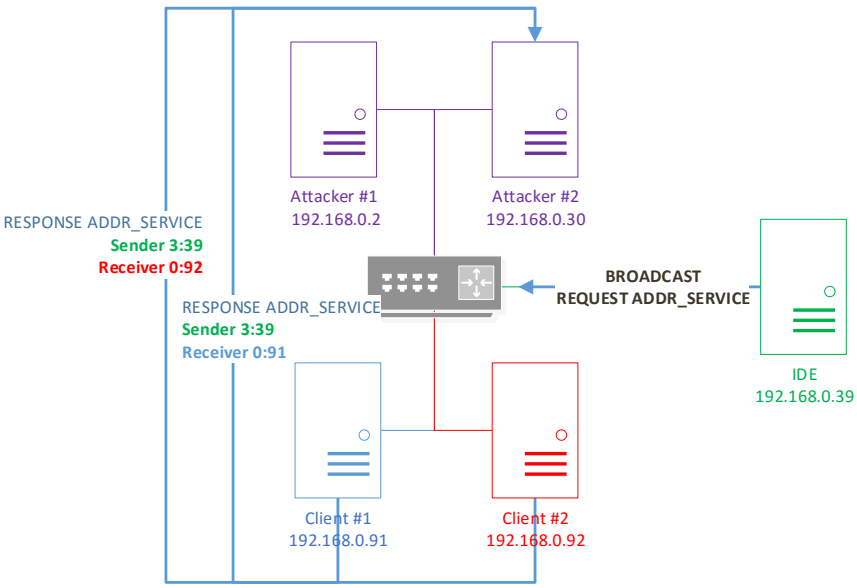
Implementing an attack by setting an arbitrary parent node

The figure above shows 5 packets and the contents of the last packet. Below is a description of each of the packets and the sequence of actions taken by CODESYS network nodes on our test bench:

1. The IDE node sends packet №1 to a **broadcast address**. This packet contains a request to receive information about nodes in the network. The request is intended for the name service.
2. The Client #1 node receives packet №1 from a **broadcast address**. When processing the request, the node extracts from the **receiver** field the address of the node to which a response must be sent. The value of the **receiver** field indicates the address of the IDE node. Therefore, the Client #1 node responds with packet №2 sent to the IDE node. This packet contains information about the Client #1 node.
3. Attacker #2 sends packet №3 to the **broadcast address**. This packet contains a request for the address service to change the parent address (service_id 2). Packet №3 is received by the Client #1 node. After processing packet №3, the Client #1 node modifies the route for its network interface and sets the address of Attacker #2 as the parent node.
4. The IDE node sends a request (packet №4) to the **broadcast address** to receive information about nodes in the network. Packet №4 is identical to packet №1.
5. The Client #1 node receives packet №4 from the **broadcast address**. When processing this request, the node generates a response and sends it in packet №5. This packet is sent to the address of the parent node that was assumed by the Attacker #2 node instead of the address of the IDE node specified in the **receiver** field.

Despite the fact that the Client #1 node received both requests to receive information from the same address but it sent responses to different addresses, the contents of these responses are identical. The values of the **sender** and **receiver** fields were also unchanged. In other words, the node with the modified route and the set parent node awaits delivery of the packet sent from its parent node, therefore the node does not change the values of the **receiver** and **sender** fields.

This means that an attacker can change the route of CODESYS network traffic without any privileges at the node running CODESYS Runtime. By making his machine the parent node, the attacker can infiltrate already existing or future traffic by implementing a man-in-the-middle attack.



Schematic of a modified route of network traffic after setting an arbitrary parent node

Vulnerability in the channel layer. Predictability of the channel ID

Initially we assumed that the value of a created communication channel ID is always incremented by four from the value of the last communication channel ID. This was indicated by the log returned by the program:

```
1528045861: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>4</channelid> connected
1528045863: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>4</channelid> closed by request, <error>0</error>
1528045864: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>8</channelid> connected
1528045864: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>8</channelid> closed by request, <error>0</error>
1528045865: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>12</channelid> connected
1528045865: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>12</channelid> closed by request, <error>0</error>
1528045865: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>16</channelid> connected
1528045865: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>16</channelid> closed by request, <error>0</error>
1528045866: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>20</channelid> connected
1528045866: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>20</channelid> closed by request, <error>0</error>
1528045866: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>24</channelid> connected
1528045867: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>24</channelid> closed by request, <error>0</error>
1528045867: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>28</channelid> connected
1528045867: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>28</channelid> closed by request, <error>0</error>
1528045868: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>32</channelid> connected
1528045868: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>32</channelid> closed by request, <error>0</error>
1528045868: Cmp=CmpChannelServer, Class=16, Error=0, Info=0, pszInfo= Channel <channelid>36</channelid> connected
```

Fragment of the CODESYS Runtime software log

To confirm this hypothesis, we researched the **HandleOpenChannelReq** function, which serves as the handler at the channel layer for the command to open channels (**OPEN_CHANNEL**). This function belongs to the **CmpChannelServer** component.

```
Call trace:
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
001: Removed at the vendor's request
002: {
[...]
064:    Removed at the vendor's request
065:    {
[...]
086:    Removed at the vendor's request
087:    {
[...]
094:        Removed at the vendor's request
095:        Removed at the vendor's request
096:          Removed at the vendor's request
097:        Removed at the vendor's request
[...]
128:        Removed at the vendor's request
129:        Removed at the vendor's request
130:        Removed at the vendor's request
131:        Removed at the vendor's request
132:        Removed at the vendor's request
133:        Removed at the vendor's request
134:        Removed at the vendor's request
135:      }
[...]
148:    Removed at the vendor's request
149:    Removed at the vendor's request
150:    Removed at the vendor's request
151:    Removed at the vendor's request
152:    Removed at the vendor's request
```

```
153:   Removed at the vendor's request
154:   {
155:     Removed at the vendor's request
156:     Removed at the vendor's request
157:   }
158: }
```

Decompiled pseudocode of the HandleOpenChannelReq function of the CmpChannelServer component

While researching the code of the **HandleOpenChannelReq** function, we discovered that each new ID of the channel actually depends on the value of the previous communication channel ID. However, it is incremented not by a fixed value equal to four, but by the number of simultaneously supported communication channels (line 94).

The event processing function of the **CmpChannelServer** component assigns the number of simultaneously supported channels in the global variable **s_iMaxServerChannels**.

```
001: Removed at the vendor's request
002: {
[...]
008:
009:   Removed at the vendor's request
010:   {
011:     Removed at the vendor's request
[...]
023:          Removed at the vendor's request
024:          Removed at the vendor's request
025:          Removed at the vendor's request
026:            Removed at the vendor's request
027:          Removed at the vendor's request
028:          Removed at the vendor's request
029:          {
030:            Removed at the vendor's request
031:            Removed at the vendor's request
032:            {
033:              Removed at the vendor's request
034:              Removed at the vendor's request
035:            }
036:          }
037:          Removed at the vendor's request
038:          Removed at the vendor's request
039:          Removed at the vendor's request
040:          Removed at the vendor's request
041:          Removed at the vendor's request
042:          Removed at the vendor's request
[...]
124:   Removed at the vendor's request
125: }
```

Disassembled code of the CmpChannelServer_hook function

Line 24 of the pseudocode of the **CmpChannelServer_hook** function shows that the value of the global variable **s_iMaxServerChannels** is regulated by the settings of the configuration file. If a section named CmpChannelServer and a setting named MaxChannels are missing, the default value, which is equal to four, is set in the **s_iMaxServerChannels** variable.

An attacker can obtain the value of the **s_iMaxServerChannels** variable by contacting the communication channel manager using the **GET_INFO** command at the Channel layer or using any command for the name service at the Datagram layer. Privileges are not required for execution of these commands.

```
Call trace:
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
       Removed at the vendor's request
```

```
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request
        Removed at the vendor's request

Pseudocode:
01: Removed at the vendor's request
02: {
03:   Removed at the vendor's request
04:
05:   Removed at the vendor's request
06:   Removed at the vendor's request
07:   Removed at the vendor's request
08:   Removed at the vendor's request
09:   Removed at the vendor's request
10: }
```

Disassembled pseudocode of the HandleInfoReq function

The **HandleInfoReq** function processes the **GET_INFO** command for the communication channel manager at the Channel layer. The global variable **s_iMaxServerChannels** will be written to the last two bytes of the response (line 08).

Depending on the settings used in the configuration file, CODESYS Runtime may fail to process information requests at the Datagram layer and at the Channel layer. However, an attacker can always send multiple paired requests to open and close a channel. The difference between two consecutively received IDs of channels will unequivocally identify the number of simultaneously supported channels on a CODESYS Runtime node.

Vulnerabilities of the Services layer

Vulnerabilities in the authentication system

This chapter will examine two detected vulnerabilities in the authentication system, namely the vulnerability in session ID generation and in password decryption. Exploitation of the first vulnerability can lead to prediction of a session ID. Exploitation of the second vulnerability can lead to decryption of an intercepted password.

Vulnerability in the predictability of session ID generation

For most services, the node must complete authentication. A request to complete authentication is processed by the **DeviceServiceHandler** function, which is registered by the **CmpDevice** component as a service with the ID of 1. The **DeviceServiceHandler** function was examined as an example of tag processing in the chapter titled "Processing tags".

When the **DeviceServiceHandler** function processes an incoming request containing a command whose ID is equal to 2, the **DeviceServiceHandler** function transfers management to the **ServerGenerateSessionId** function (examined below).

```
001: Removed at the vendor's request
002: {
[...]
129:   Removed at the vendor's request
130:   Removed at the vendor's request
131:   Removed at the vendor's request
132:   Removed at the vendor's request
133:   Removed at the vendor's request
134:   {
[...]
254:     Removed at the vendor's request
```

```
[...]
262:       Removed at the vendor's request
263:       Removed at the vendor's request
[...]
365:         Removed at the vendor's request
366:        }
367:        Removed at the vendor's request
[...]
389:        Removed at the vendor's request
390:        Removed at the vendor's request
391:        Removed at the vendor's request
392:        Removed at the vendor's request
393:        Removed at the vendor's request
394:        Removed at the vendor's request
395:        Removed at the vendor's request
396:        Removed at the vendor's request
397:        Removed at the vendor's request
398:        Removed at the vendor's request
[...]
761:    }
762:    Removed at the vendor's request
763: }
```

Fragment of the decompiled pseudocode of the DeviceServiceHandler function of the CmpDevice component

A response to a successful authentication request contains a data tag with the ID 0x21 (line 392). The data of this tag contains a generated ID for the created session (line 393). The session ID itself is created within the **HandleLoginSessionId** function even before the received authentication data is verified (line 263). The generated ID is returned in the **ulSessionId** variable.

```
01: Removed at the vendor's request
02: {
[...]
09:    Removed at the vendor's request
10:    Removed at the vendor's request
11:    Removed at the vendor's request
12:    {
13:      Removed at the vendor's request
14:      Removed at the vendor's request
15:    }
16:    Removed at the vendor's request
17:    {
18:      *Removed at the vendor's request
19:      Removed at the vendor's request
20:    }
21:    Removed at the vendor's request
22:    {
23:      *Removed at the vendor's request
24:      Removed at the vendor's request
25:    }
26:    Removed at the vendor's request
27:    Removed at the vendor's request
28: }
```

Fragment of the decompiled pseudocode of the HandleLoginSessionId function

The **ServerGenerateSessionId** function (line 13), which generates the session ID, is queried within the **HandleLoginSessionId** function. This function is queried under the condition that the received session ID is equal to one of the following numbers: 0x0, 0x11 or 0x815. An additional condition for creating a session ID is that a session ID must not already exist for the received channel ID (line 10).

```
01: Removed at the vendor's request
02: {
03:    Removed at the vendor's request
04:
```

```
05:   Removed at the vendor's request
06:   Removed at the vendor's request
07: }
08:
09: Removed at the vendor's request
10: {
11:   Removed at the vendor's request
[...]
15:   Removed at the vendor's request
16:     Removed at the vendor's request
17:   Removed at the vendor's request
18:   Removed at the vendor's request
19:   *Removed at the vendor's request
20:   *Removed at the vendor's request
21:   Removed at the vendor's request
22: }
```

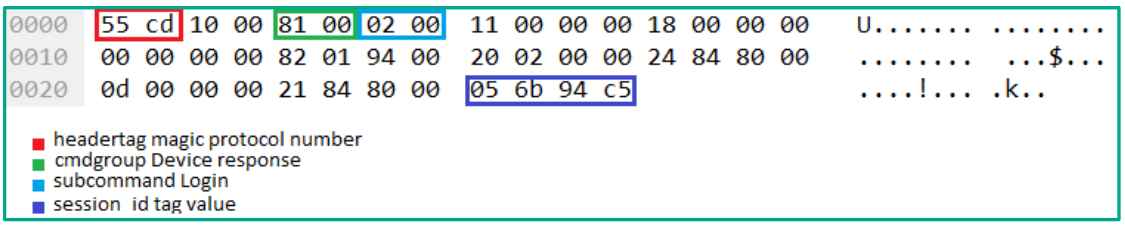
Fragments of the decompiled pseudocode of the SysTimeGetMs function of the SysTimer component and fragments of the decompiled pseudocode of the ServerGenerateSessionId function of the CmpSrv component

The **ServerGenerateSessionId** function is exported by the **CmpSrv** component. Its algorithm is described below:

1. Verify the presence of an argument handle (lines 15 and 16).
2. Receive the current time by calling the SysTimeGetMs function (line 17).
3. Initialize the pseudo-random number generator. Set the value received at the preceding step as the seed value for the generator (line 18).
4. Add the received time value to the random number and write it to the value for the argument handle (line 19).
5. Set the most significant bit in the value for the argument handle (line 20).

This generator algorithm uses a pseudo-random number generator. This means that the generated random number depends on the set seed value and may be restored. An attacker can find out session IDs by decrementing the seed values and recreating the session ID for the modified seed. This can be done within an acceptable time interval, even though the attack will occur remotely.

The second weakness of this algorithm is the use of the **SysTimeGetMs** function**.** This function is exported by the system component **SysTimer**. To correctly run CODESYS Runtime, the developer must implement all system components. This means that the provided implementation of the **SysTimeGetMs** function may differ from the implementation of this function in a different CODESYS Runtime. In terms of security, this implementation of the function that generates a session ID imposes additional responsibility on the developer that will adapt the system components, including the **SysTimer** component.



Color				
fields	protocol_id	service_group	service_id	Session id
Value	0xcd55 (HeaderTagProtocol)	0x81 (CmpDevice, Response)	0x2 (Login)	0xc5946b05

Example fragment of a packet containing a generated session ID (session_id)

For instance, in the analogous example of a response to an authentication request presented in the chapter titled "Processing tags", the fragment of the response contains a data tag with the session ID setting, and a seed with the value 356267299 was used to generate a session ID value equal to 0xc5946b05.

Vulnerabilities in password encryption

In the **DeviceServiceHandler** function registered as a service by the **CmpDevice** component, multiple vulnerabilities in the password encryption mechanism were detected. These vulnerabilities can be exploited to decrypt an intercepted password.

A request to complete authentication is processed by the **DeviceServiceHandler** function, which is registered by the **CmpDevice** component as a service with the ID of 1. The **DeviceServiceHandler** function was examined as an example of tag processing in the chapter titled "Processing tags".

To complete authentication in CODESYS Development System, a request must be sent for the **DeviceServiceHandler** service containing a command with an ID equal to 2. The data sent for authentication will include the encrypted password that will be decrypted during processing by the service.

```
001: Removed at the vendor's request
002: {
[...]
130:   Removed at the vendor's request
[...]
133:   Removed at the vendor's request
134:   {
[...]
254:     Removed at the vendor's request
[...]
266:       Removed at the vendor's request
267:       Removed at the vendor's request
268:       {
269:         Removed at the vendor's request
270:         Removed at the vendor's request
271:         {
272:           Removed at the vendor's request
273:             Removed at the vendor's request
274:             Removed at the vendor's request
275:           Removed at the vendor's request
276:             Removed at the vendor's request
277:             Removed at the vendor's request
278:             {
279:               Removed at the vendor's request
280:               Removed at the vendor's request
281:               {
282:                 Removed at the vendor's request
283:               }
284:               Removed at the vendor's request
285:               {
286:                 Removed at the vendor's request
287:                 Removed at the vendor's request
288:               }
289:               Removed at the vendor's request
290:               {
291:                 Removed at the vendor's request
292:               }
293:               Removed at the vendor's request
294:               Removed at the vendor's request
295:               Removed at the vendor's request
296:             }
297:             Removed at the vendor's request
298:           Removed at the vendor's request
299:             Removed at the vendor's request
300:             Removed at the vendor's request
```

```
301:            Removed at the vendor's request
302:              Removed at the vendor's request
303:              Removed at the vendor's request
304:          }
305:          Removed at the vendor's request
306:          Removed at the vendor's request
307:          Removed at the vendor's request
308:        }
[...]
315:        Removed at the vendor's request
316:        {
317:          Removed at the vendor's request
318:          Removed at the vendor's request
[...]
327:        }
[...]
761:    }
762:    Removed at the vendor's request
763: }
```

Fragment of decompiled code of the DeviceServiceHandler function of the CmpDevice component

The received password is decrypted in the **UserMgrDecryptPassword** function (line 318). This function uses the following values as arguments:

1. **encrypted_password** – value of the encrypted password that is extracted from the data tag with the ID 17 (line 286).
2. **pulCrypeType** – ID of the encryption algorithm that was used to encrypt the transmitted password. The value of the ID is extracted from the data tag with the ID 0x22 (line 299).
3. **pulChallenge** – random number value (nonce) involved in password decryption. The value of the random number is extracted from the data tag with the ID 0x23.

```
01: Removed at the vendor's request
02: {
[...]
16:    Removed at the vendor's request
17:      Removed at the vendor's request
18:    Removed at the vendor's request
19:    Removed at the vendor's request
20:    Removed at the vendor's request
21:    Removed at the vendor's request
22:    {
23:      Removed at the vendor's request
24:      {
25:        Removed at the vendor's request
26:        Removed at the vendor's request
27:        Removed at the vendor's request
28:        Removed at the vendor's request
29:        Removed at the vendor's request
30:        Removed at the vendor's request
31:        {
32:          Removed at the vendor's request
33:          Removed at the vendor's request
34:          Removed at the vendor's request
35:            Removed at the vendor's request
36:          Removed at the vendor's request
37:            Removed at the vendor's request
38:        }
39:        Removed at the vendor's request
40:      }
[...]
```

Fragment of decompiled pseudocode of the UserMgrDecryptPassword function

The **UserMgrDecryptPassword** function performs the following actions:

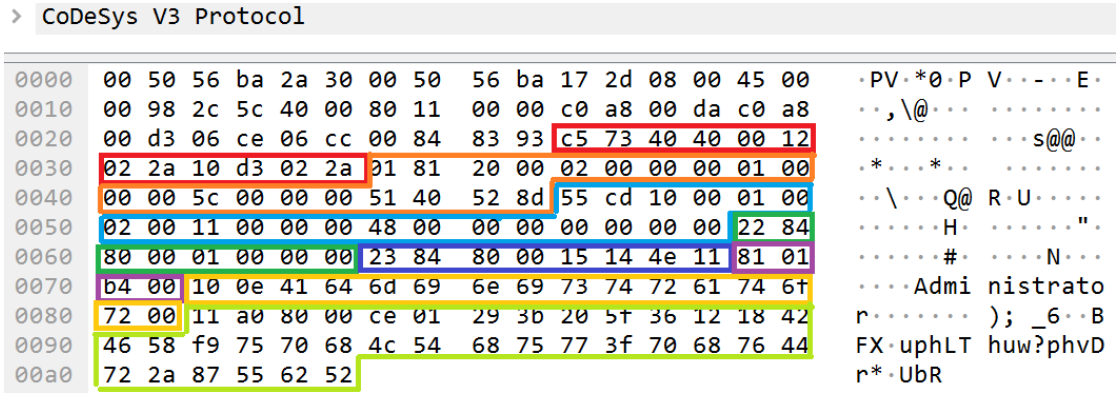
1. Compares the values of the **pulCrypeType** argument with 1 (line 16). If these values are different, the function is not processed further. In other words, CODESYS Runtime provides only one encryption algorithm for a transmitted password, and the **pulCrypeType** value must be transmitted each time during authentication, despite the lack of alternatives for selecting the password encryption algorithm.
2. Writes a fixed key to the local **key** variable (line 18).
3. From the 4-byte value of the **pulChallenge** argument, writes only the least significant bit to the 4-byte **aChallenge** array (line 25). Zero values are set for the remaining three bytes of the array (lines 26-28).
4. Then the function uses three indexes: **index** – password index, **index_key** – key index, and **challenge_index** – random number index. The password index identifies the end of decryption of an encrypted password. When the password index is equal to the size of the encrypted password, the **UserMgrDecryptPassword** function terminates and returns management to the **DeviceServiceHandler** function (line 32). The remaining two indexes (random number index and key index) will be zeroed each time the size of their variables (**aChallenge** and **key**) are equal to the value of the index (line 34 for the **key** variable and line 36 for the **aChallenge** variable).
5. A character-by-character decryption of a password is provided below. Each decrypted character is obtained by adding one byte taken from the **ulChallenge** array for the random number index (**challenge_index)** and one byte taken from the **key** variable for the key index (**index_key)**. For the obtained number, an XOR operation is performed with the byte taken from the **encrypted_key** password argument for the encrypted password index (line 32).

If the **UserMgrDecryptPassword** function is successfully performed, the obtained password will be decrypted and written to the **decrypted_password** argument.

From the described algorithm of the function, three vulnerabilities can immediately be emphasized:

1. **Using a weak random number.** Despite the fact that the **DeviceServiceHandler** function registered by the **CmpDevice** component for the authentication command extracts a 4-byte numerical value from received data tags, only one of the four bytes is used to decrypt the obtained password. Because the encryption algorithm is symmetrical, only one byte is involved in password encryption as well.
   Use of a random one-byte value for password protection does not increase the security of a transmitted password because this value can be obtained by brute force within a very short period of time.
2. **Using an arbitrary value as the random number**. The examined algorithm of the **DeviceServiceHandler** function shows that the service authentication command of the **CmpDevice** component uses the obtained numerical parameter from the authenticated node as the random number (**aChallenge**) for password decryption.
   In other words, despite the fact that the side performing the authentication sends a random number that must be used in password encryption when it receives an authentication request, the side that is being authenticated can encrypt the password with its own number and transmit this number for decryption.
   This means that an attacker who has intercepted the authentication data one time can resend it without modifications to complete authorization.
3. **Using a fixed key**. The examined decryption algorithm uses a fixed key for password encryption and decryption. This means that an attacker who has received the encrypted authentication data can always decrypt an intercepted password.

Based on the example of a packet containing an authentication request examined in the chapter titled "Processing tags", we will show how a transmitted password can be partially decrypted.



Color								
fields	datagram_ layer_ fields	channel_ layer_ fields	Services_ layer_ fields	Data_tag_1	Data_tag_2	Parent_tag_1	Data_tag_3	Data_tag_4

Packet containing an authentication request

The encrypted password is transmitted in the data tag with the ID 0x11, which is designated as **Data_tag_4.** You can partially restore a password by extracting data of the encrypted password from the data tag and perform the XOR operation with bytes of the key for the same indexes for each byte of data.

```
1: encrypted_password =
"\xce\x01\x29\x3b\x20\x5f\x36\x12\x18\x42\x46\x58\xf9\x75\x70\x68\x4c\x54\x68\x75\x77\x3f\x70\x68\x76\x44\
x72\x2a\x87\x55\x62\x52"
2: KEY = "zeDR96EfU#27vuph7Thub?phaDr*rUbR"
3: for c, s in enumerate(encrypted_password):
4:     print chr(ord(KEY[c]) ^ ord(encrypted_password[c])),
5:
6: d m i  i s t M a t o
```

Script in Python for partial decryption of a password

A partially restored password contains the letters d, m, I, I, s, t, a, t, o (line 6). These characters are included in the "Administrator" string, which is the default password for an Administrator.

Vulnerability of application code

One of the tasks performed by CODESYS Runtime is loading, managing, and executing applications. CODESYS Development System compiles an application for CODESYS Runtime and loads it over the CODESYS PDU protocol. A loaded application is a stream of binary data.

During our research, we did not focus on research of the structure of a compiled application. (It should be noted that the U.S. Office of Naval Research supported research work that was conducted to analyze the contents of a compiled application for CODESYS Runtime. This work resulted in the development of the ICSREF framework.)

When studying the contents of packets sent from CODESYS Development System to CODESYS Runtime while an application is being loaded, we detected places in the binary stream where we could inject **arbitrary machine code** – shellcode.

Two functions turned out to be our backdoors: the function for initialization of global variables (named **Global INIT** in the cited work) and the program start function (named **PLC_PRG** in the cited work).

```
Header:
1: PROGRAM PLC_PRG
2: VAR
3:         magic: DWORD:= 16#DEADBEEF;
4: END_VAR

Body:
5: magic := magic + 16#BEEF;
```

Source code of the PLC_PRG program

To confirm the capability of injecting **arbitrary machine code for the purpose of executing it in the binary stream of an application**, we used CODESYS Development System to compile the program and remotely downloaded it to a Raspberry PI device running CODESYS Runtime. In the variable declaration block, the **magic** variable was declared with the value **0xDEADBEEF** (line 3). The program body block indicates that the value of the **magic** variable must be permanently stored with the value **0xBEEF** and the obtained result must be written to the **magic** variable (line 5).

```
00c0  00 0a 0b 00 00 ea 6c 40  99 e5 08 50 a0 e3 04 50
00d0  85 e0 05 50 d9 e7 0a 00  55 e3 05 00 00 1a 01 40
00e0  a0 e3 0c 40 ca e5 6c 40  99 e5 01 40 84 e2 6c 40
00f0  89 e5 ff ff ff ea 30 42  bd e8 08 d0 8d e2 00 84
0100  bd e8 00 00 00 60 a0 01  c0 00 21 06 03 00 28 15
0110  01 00 22 b4 80 00 30 00  00 00 00 44 2d e9 0d a0
0120  a0 e1 30 00 2d e9 18 b0  9f e5 00 40 9b e5 0c 50
0130  9f e5 05 40 84 e0 00 40  8b e5 30 00 bd e8 00 84
0140  bd e8 ef be 00 00 70 38  00 00 a0 01 f0 08 21 06
0150  03 00 58 15 01 00 22 e4  88 00 60 04 00 00 00 44
0160  2d e9 0d a0 a0 e1 20 d0  4d e2 71 00 2d e9 00 40
0170  a0 e3 10 40 ca e5 00 40  a0 e3 20 40 0a e5 00 40
0180  a0 e3 1c 40 0a e5 00 40  a0 e3 18 40 0a e5 1f 40
0190  a0 e3 14 40 4a e5 1c 40  a0 e3 13 40 4a e5 1f 40
01a0  a0 e3 12 40 4a e5 1e 40  a0 e3 11 40 4a e5 1f 40
01b0  a0 e3 10 40 4a e5 1e 40  a0 e3 0f 40 4a e5 1f 40
01c0  a0 e3 0e 40 4a e5 1f 40  a0 e3 0d 40 4a e5 1e 40
01d0  a0 e3 0c 40 4a e5 1f 40  a0 e3 0b 40 4a e5 1e 40
01e0  a0 e3 0a 40 4a e5 1f 40  a0 e3 09 40 4a e5 0c 40
01f0  9a e5 00 50 94 e5 08 50  0a e5 10 d0 4d e2 08 40
0200  9a e5 00 40 8d e5 a8 43  9f e5 04 40 8d e5 9c b3
0210  9f e5 00 40 9b e5 0d 00  a0 e1 04 a0 2d e5 88 a3
0220  9f e5 04 a0 2d e5
```

```
00c0  00 00 60 4b 00 00 40 5f  00 00 00 00 00 60 a0 01
00d0  c8 00 21 06 03 00 e0 89  01 00 22 bc 80 00 38 00
00e0  00 00 00 44 2d e9 0d a0  a0 e1 08 d0 4d e2 10 08
00f0  2d e9 00 40 a0 e3 09 40  ca e5 00 40 a0 e3 08 40
0100  0a e5 00 40 a0 e3 04 40  4a e5 10 08 bd e8 08 d0
0110  8d e2 00 84 bd e8 00 00  00 60 a0 01 c8 00 21 06
0120  03 00 18 8a 01 00 22 bc  80 00 38 00 00 00 00 44
0130  2d e9 0d a0 a0 e1 08 d0  4d e2 10 08 2d e9 00 40
0140  a0 e3 09 40 ca e5 00 40  a0 e3 08 40 0a e5 00 40
0150  a0 e3 04 40 4a e5 10 08  bd e8 08 d0 8d e2 00 84
0160  bd e8 00 00 00 60 a0 01  d8 00 21 06 03 00 50 8a
0170  01 00 22 cc 80 00 48 00  00 00 00 44 2d e9 0d a0
0180  a0 e1 08 d0 4d e2 10 08  2d e9 00 40 a0 e3 09 40
0190  ca e5 00 40 a0 e3 08 40  0a e5 00 40 a0 e3 04 40
01a0  4a e5 14 40 9f e5 0c b0  9f e5 00 40 8b e5 10 08
01b0  bd e8 08 d0 8d e2 00 84  bd e8 70 38 00 00 ef be
01c0  ad de a0 01 d8 00 21 06  03 00 98 8a 01 00 22 cc
01d0  80 00 48 00 00 00 00 44  2d e9 0d a0 a0 e1 08 d0
01e0  4d e2 10 08 2d e9 00 40  a0 e3 09 40 ca e5 00 40
01f0  a0 e3 08 40 0a e5 00 40  a0 e3 04 40 4a e5 00 40
0200  a0 e3 0c b0 9f e5 00 40  8b e5 10 08 bd e8 08 d0
0210  8d e2 00 84 bd e8 7c 38  00 00 00 00 00 60 a0 01
0220  98 05 21 06 03 00
```

Fragment of traffic when loading an application with references to using EF BE bytes

After downloading the compiled program to the Raspberry Pi device, the traffic was searched for EF BE bytes. These bytes are contained in the numbers **0xDEADBEEF** and **0xBEEF** when the bytes are ordered from least significant to most significant (little endian). Only two references of these bytes were detected in traffic (highlighted in red). It should be noted that in one packet, AD DE bytes (highlighted in blue) were detected next to the sought EF BE bytes.

Then the entire loaded stream of binary data was analyzed for the presence of machine instructions of the ARM processor on which the Raspberry Pi is running. While searching for EF BE bytes, we detected an instruction querying the number **0xDEADBEEF** and an instruction querying the number **0xBEEF**.

We will examine the instructions of the first query.

```
01: 00 00 00 60        ANDVS           R0, R0, R0
02: A0 01 D8 00        SBCEQS          R0, R8, R0,LSR#3
03: 21 06 03 00        ANDEQ           R0, R3, R1,LSR#12
```

```
04: 50 8A 01 00       ANDEQ           R8, R1, R0,ASR R10
05: 22 CC 80 00       ADDEQ           R12, R0, R2,LSR#24
06: 48 00 00 00       ANDEQ           R0, R0, R8,ASR#32
07: 00 44 2D E9       STMFD           SP!, {R10,LR}
08: 0D A0 A0 E1       MOV             R10, SP
09: 08 D0 4D E2       SUB             SP, SP, #8
10: 10 08 2D E9       STMFD           SP!, {R4,R11}
11: 00 40 A0 E3       MOV             R4, #0
12: 09 40 CA E5       STRB            R4, [R10,#9]
13: 00 40 A0 E3       MOV             R4, #0
14: 08 40 0A E5       STR             R4, [R10,#-8]
15: 00 40 A0 E3       MOV             R4, #0
16: 04 40 4A E5       STRB            R4, [R10,#-4]
17: 14 40 9F E5       LDR             R4, =0xDEADBEEF
18: 0C B0 9F E5       LDR             R11, =0x3870
19: 00 40 8B E5       STR             R4, [R11]
20: 10 08 BD E8       LDMFD           SP!, {R4,R11}
21: 08 D0 8D E2       ADD             SP, SP, #8
22: 00 84 BD E8       LDMFD           SP!, {R10,PC}
```

Detected assembler instructions for an ARM processor querying the number 0xDEADBEEF

The constant **0xDEADBEEF** is written to register R4 at line 17. The constant **0x3870** is written to register R11 at line 18. At line 19, the value of the R4 register (**0xDEADBEEF**) is written at the address of the value of the R11 register (**0x3870**).

```
01: 00 00 00 60       ANDVS           R0, R0, R0
02: A0 01 C0 00       SBCEQ           R0, R0, R0,LSR#3
03: 21 06 03 00       ANDEQ           R0, R3, R1,LSR#12
04: 28 15 01 00       ANDEQ           R1, R1, R8,LSR#10
05: 22 B4 80 00       ADDEQ           R11, R0, R2,LSR#8
06: 30 00 00 00       ANDEQ           R0, R0, R0,LSR R0
07: 00 44 2D E9       STMFD           SP!, {R10,LR}
08: 0D A0 A0 E1       MOV             R10, SP
09: 30 00 2D E9       STMFD           SP!, {R4,R5}
10: 18 B0 9F E5       LDR             R11, =0x3870
11: 00 40 9B E5       LDR             R4, [R11]
12: 0C 50 9F E5       LDR             R5, =0xBEEF
13: 05 40 84 E0       ADD             R4, R4, R5
14: 00 40 8B E5       STR             R4, [R11]
15: 30 00 BD E8       LDMFD           SP!, {R4,R5}
16: 00 84 BD E8       LDMFD           SP!, {R10,PC}
```

Detected assembler instructions for an ARM processor querying the number 0xBEEF

In contrast to the machine code of the first query, the machine code of the second query works in reverse. First the constant **0x3870** is written to the R11 register (line 10). Then the memory cell contents based on the address of the constant **0x3870** (line 11) is written to the R4 register. Then the constant **0xBEEF** is written to the R5 register (line 12). The value of the R5 register (**0xBEEF**) is added to the value at the address of **0x3870** (line 13). The resulting sum of the saved value from memory and the constant **0xBEEF** is written to the R4 register, whose value is then written to the memory cell at the address of **0x3870**.

Based on the examined queries, we can assume that the address of the declared global **magic** variable is located at the address of **0x3870**. The value **0xDEADBEEF** is written to the **magic** variable in the examined machine code of the first detected query. In the second detected query, the number **0xBEEF** is added to the **magic** variable. The result of this addition is again written to the address of the global **magic** variable (**0x3870**). Both fragments of machine code correspond to the source code of the program **PLC_PRG**.

```
00c0  00 0a 0b 00 00 ea 6c 40  99 e5 08 50 a0 e3 04 50
00d0  85 e0 05 50 d9 e7 0a 00  55 e3 05 00 00 1a 01 40
00e0  a0 e3 0c 40 ca e5 6c 40  99 e5 01 40 84 e2 6c 40
00f0  89 e5 ff ff ff ea 30 42  bd e8 08 d0 8d e2 00 84
0100  bd e8 00 00 00 60 a0 01  c0 00 21 06 03 00 28 15
0110  01 00 22 b4 80 00 30 00  00 00 00 44 2d e9 0d a0
0120  a0 e1 30 00 2d e9 18 b0  9f e5 00 40 9b e5 0c 50
0130  9f e5 05 40 84 e0 00 40  8b e5 30 00 bd e8 00 84
0140  bd e8 ef be 00 00 70 38  00 00 a0 01 f0 08 21 06
0150  03 00 58 15 01 00 22 e4  88 00 60 04 00 00 00 44
0160  2d e9 0d a0 a0 e1 20 d0  4d e2 71 00 2d e9 00 40
0170  a0 e3 10 40 ca e5 00 40  a0 e3 20 40 0a e5 00 40
0180  a0 e3 1c 40 0a e5 00 40  a0 e3 18 40 0a e5 1f 40
0190  a0 e3 14 40 4a e5 1c 40  a0 e3 13 40 4a e5 1f 40
01a0  a0 e3 12 40 4a e5 1e 40  a0 e3 11 40 4a e5 1f 40
01b0  a0 e3 10 40 4a e5 1e 40  a0 e3 0f 40 4a e5 1f 40
01c0  a0 e3 0e 40 4a e5 1f 40  a0 e3 0d 40 4a e5 1e 40
01d0  a0 e3 0c 40 4a e5 1f 40  a0 e3 0b 40 4a e5 1e 40
01e0  a0 e3 0a 40 4a e5 1f 40  a0 e3 09 40 4a e5 0c 40
01f0  9a e5 00 50 94 e5 08 50  0a e5 10 d0 4d e2 08 40
0200  9a e5 00 40 8d e5 a8 43  9f e5 04 40 8d e5 9c b3
0210  9f e5 00 40 9b e5 0d 00  a0 e1 04 a0 2d e5 88 a3
0220  9f e5 04 a0 2d e5
```

```
00c0  00 00 60 4b 00 00 40 5f  00 00 00 00 00 60 a0 01
00d0  c8 00 21 06 03 00 e0 89  01 00 22 bc 80 00 38 00
00e0  00 00 00 44 2d e9 0d a0  a0 e1 08 d0 4d e2 10 08
00f0  2d e9 00 40 a0 e3 09 40  ca e5 00 40 a0 e3 08 40
0100  0a e5 00 40 a0 e3 04 40  4a e5 10 08 bd e8 08 d0
0110  8d e2 00 84 bd e8 00 00  00 60 a0 01 c8 00 21 06
0120  03 00 18 8a 01 00 22 bc  80 00 38 00 00 00 00 44
0130  2d e9 0d a0 a0 e1 08 d0  4d e2 10 08 2d e9 00 40
0140  a0 e3 09 40 ca e5 00 40  a0 e3 08 40 0a e5 00 40
0150  a0 e3 04 40 4a e5 10 08  bd e8 08 d0 8d e2 00 84
0160  bd e8 00 00 00 60 a0 01  d8 00 21 06 03 00 50 8a
0170  01 00 22 cc 80 00 48 00  00 00 00 44 2d e9 0d a0
0180  a0 e1 08 d0 4d e2 10 08  2d e9 00 40 a0 e3 09 40
0190  ca e5 00 40 a0 e3 08 40  0a e5 00 40 a0 e3 04 40
01a0  4a e5 14 40 9f e5 0c b0  9f e5 00 40 8b e5 10 08
01b0  bd e8 08 d0 8d e2 00 84  bd e8 70 38 00 00 ef be
01c0  ad de a0 01 d8 00 21 06  03 00 98 8a 01 00 22 cc
01d0  80 00 48 00 00 00 00 44  2d e9 0d a0 a0 e1 08 d0
01e0  4d e2 10 08 2d e9 00 40  a0 e3 09 40 ca e5 00 40
01f0  a0 e3 08 40 0a e5 00 40  a0 e3 04 40 4a e5 00 40
0200  a0 e3 0c b0 9f e5 00 40  8b e5 10 08 bd e8 08 d0
0210  8d e2 00 84 bd e8 7c 38  00 00 00 00 00 60 a0 01
0220  98 05 21 06 03 00
```

Fragment of traffic when loading an application with assembler instructions

Machine instructions of an ARM processor are also transmitted over the CODESYS PDU protocol. In the example of the second query, the instruction **ADD R4, R4, R5** (line 13) corresponds to bytes 05 40 84 E0 (highlighted in green in the packet), and the next instruction **STR R4, [R11]** (line 14) corresponds to bytes 00 40 8B E5 (highlighted in orange).

Therefore, an attacker can inject his own machine code and execute it on a target device. It should be mentioned that the system daemons of CODESYS Runtime on a Raspberry Pi device are run as a background process (daemon) under the user name root, which has the highest level of privileges in Linux OS. Therefore, executable machine code will also run with the highest privileges in the system. The CODESYS Runtime emulator in Windows OS is also run with the highest permissions in the system: SYSTEM permissions.

In conclusion

CODESYS Runtime is a sophisticated and powerful tool designed for developing PLC programs and controlling PLCs. At the same time, it implements an effective architectural approach that enables the capabilities of CODESYS Runtime itself to be expanded. The protocol used for communication between the development environment, i.e., the CODESYS Development System, and the execution environment, i.e., CODESYS Runtime, is multi-layer, dynamic and, most importantly, proprietary.

Ultimately, the use of proprietary protocols negatively affects PLC vendors, because they have limited knowledge of the software they use and have to entrust the initial protection of their PLCs blindly to the framework's developers.

CODESYS has made many steps to improve the security of their products. However, as our research of the security of CODESYS Runtime has demonstrated, these steps are insufficient.

While analyzing the security of CODESYS Runtime, we identified 15 vulnerabilities and reported them to the software vendor. Of those vulnerabilities reported, the vendor was already aware of 5 (i.e., duplicates), 2 were referred to as architectural features by the CODESYS security group and the remaining 8 were fixed. The vulnerabilities that were fixed had CVSS base scores of 5.4 to 9.

It is worth noting that CODESYS subsequently did fix one of the vulnerabilities they had called architectural features.

Our research was based on the black box method, which means that we had originally had no information on CODESYS Runtime. Any information we have has been obtained from public sources and from technical research.

After analyzing the data transferred over the CODESYS protocol and correlating CODESYS Runtime software code with the data received over the network, we were able to identify four vulnerabilities in the authentication mechanism – in the Services layer (which is the last of four layers in the CODESYS protocol stack). These vulnerabilities were assigned the following IDs: KLCERT-18-037 (CVE-2018-20025) and KLCERT-19-031 (CVE-2019-9013). By exploiting these vulnerabilities in the authentication system, an attacker could be able to decrypt the password being sent, implement an attack in which encrypted authentication data is reused without being modified and predict the session ID.

CODESYS developers built their protocol on the TCP/IP protocol stack. As a result of this, CODESYS has inherited some of the issues characteristic of TCP/IP: we identified vulnerabilities in the Datagram layer (the second of four layers) and the Channel layer (the third of four layers), the possible existence of which in the TCP/IP protocol stack was reported back in 1989 (see Security Problems in the TCP/IP Protocol Suite).

In the Datagram layer of the CODESYS protocol stack, we determined that an attack identical to IP spoofing was possible. This vulnerability was assigned the ID KLCERT-18-036 (CVE-2018-20026). By automating the exploitation of this vulnerability, attackers could disguise their activity on the network for a long time, manipulating devices with CODESYS Runtime running on them and making these devices send malicious packets to each other.

We also discovered that an attack similar to the infamous ARP spoofing attack was possible against the Datagram layer: the routing mechanism used on the CODESYS network makes it possible to build an information network based on the tree topology from CODESYS Runtime nodes. If the parent node can be changed without authentication, this makes man-in-the-middle attacks possible. Thus, an attacker can use the protocol's capabilities at the datagram

level to inform a CODESYS Runtime host that it has become that host's new parent, which will result in the host sending all of its outgoing traffic via the new parent.

The last specimen in the classical vulnerability zoo is the absence of a sandbox for the program downloaded to the device. In the process of analyzing the protocol, it was determined that some fragments of the program downloaded to the device are machine instructions. The hypothesis that arbitrary code (shellcode) can be injected instead of these instructions was confirmed. The vulnerability was assigned the ID KLCERT-18-035 (CVE-2018-10612).

Since the CODESYS Runtime daemon in Linux and the CODESYS Runtime service in Windows run with the highest privileges (root and SYSTEM, respectively), arbitrary code will also run with the highest privileges. This means that the attacker will not need to perform any additional manipulations in the system or exploit any further vulnerabilities to achieve the highest privilege level.

As information security experts know from their many years of experience, the 'security by obscurity' approach is not the best strategy for protecting information. This is certainly true of undocumented, proprietary network communication protocols. Any such protocol will eventually be analyzed and its vulnerabilities identified. Unfortunately, in many cases threat actors will do this sooner than white-hat researchers, if only because they have a much stronger motivation.

We believe that all the vulnerabilities described in this article, and possibly others as well, could have been found by the community of information security experts and enthusiasts at the early stages of the protocol's design, development and use. If the protocol specification had been available to potential users, its vulnerabilities could have been identified during its discussion and analysis and would not have affected products by hundreds of developers installed at tens or even hundreds of thousands of industrial facilities.

At the current stage, such work takes much greater effort and requires much more specialist knowledge, which, unfortunately, may be inaccessible to many developers who use CODESYS in their solutions. Perhaps, in this respect, the development approach selected was not optimal.

The CODESYS security group responded to information on the vulnerabilities identified promptly and responsibly.

We sincerely thank the CODESYS team for their cooperation.

```
pi@raspberrypi:~ $ ./opt/codesys/bin/codesyscontrol.bin -vvvvvvv
CODESYS Control V3.5.12.0 for ARM - build Dec 18 2017
type:4102 id:0x00000010 name:CODESYS Control for Raspberry Pi SL vendor: 3S - Smart
Software Solutions GmbH
buildinformation: <none>

 _____________
< ... bye >
 -------------
        \    ^__^
         \   (--)\_______
             (__)\       )\/\
                 ||----w |
                 ||     ||
```

Kaspersky Industrial Control Systems Cyber Emergency Response Team (Kaspersky ICS CERT) is a global project of Kaspersky aimed at coordinating the efforts of automation system vendors, industrial facility owners and operators, and IT security researchers to protect industrial enterprises from cyberattacks. Kaspersky ICS CERT devotes its efforts primarily to identifying potential and existing threats that target industrial automation systems and the industrial internet of things.

Kaspersky ICS CERT ics-cert@kaspersky.com