RdS Cyber Report

The aim of the project is to develop a simulated testbed for an OT system, with the goal of implementing a secure communication channel using both symmetric and asymmetric encryption. The testbed will consist of some PLCs, an Operator Panel and, eventually, sensors and actuators connected to the PLCs. The communication between the operator panel and the PLCs will occur through the Modbus TCP/IP communication protocol, which represents a widely used standard in industrial systems.

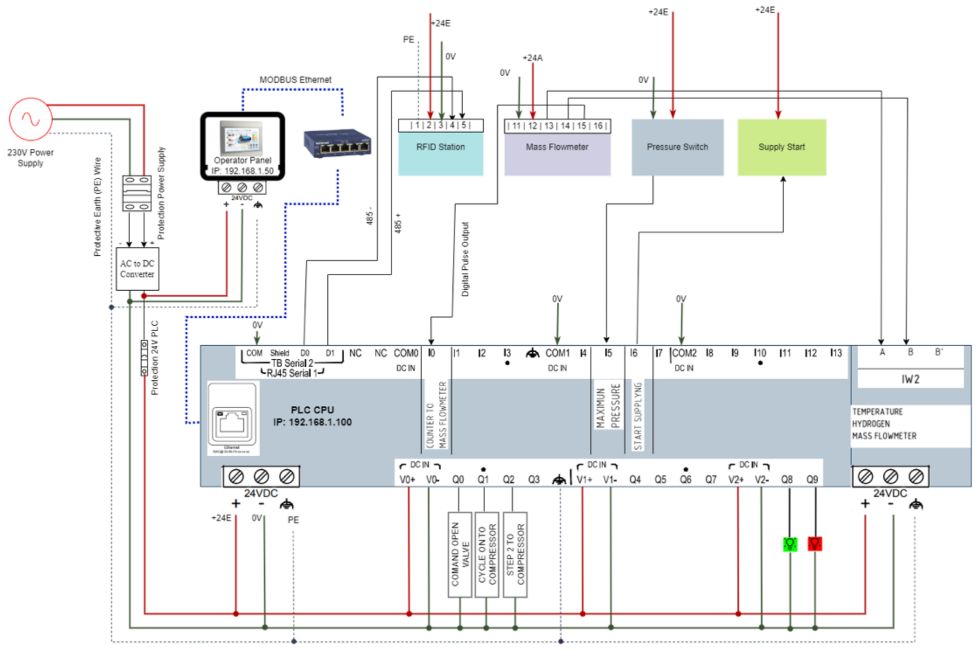


Figure 1: OT system schematics

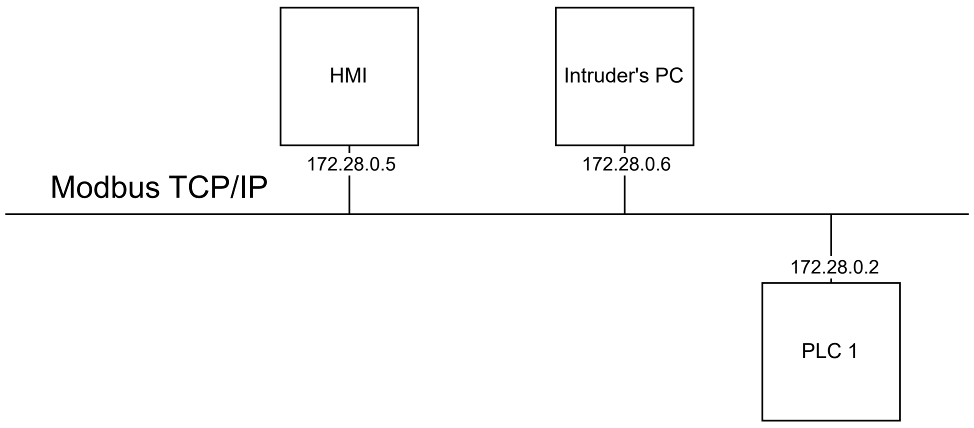
Simulation

To simulate the testbed, Docker was chosen as the preferred tool due to its versatility and efficiency in managing containerized environments. It simplifies the process of creating, deploying, and managing applications by encapsulating them into lightweight, portable containers that can run consistently across different environments.

In addition, given that a network must be simulated, Docker Compose, an orchestration tool for Docker, was utilized. It is a very powerful tool that allows for the definition and management of multi-container Docker applications through a YAML file, typically named docker-compose.yml. Within this file, it's possible to define various containers representing the industrial components such as operator panels, PLCs, or in general nodes of our network.

Each container within the Docker Compose file can be configured with its own dependencies and, more important, network settings, enabling the creation of an isolated Docker environment where different components can interact with each other as they would in a real industrial setting.

The simplest network that can be simulated comprises an operator panel (HMI), a PLC, and an "intruder" executing attacks. In this setup, the operator panel serves as the interface through which operators interact with the industrial system, while the PLC controls the industrial machinery and outputs the measures to the operator panel.



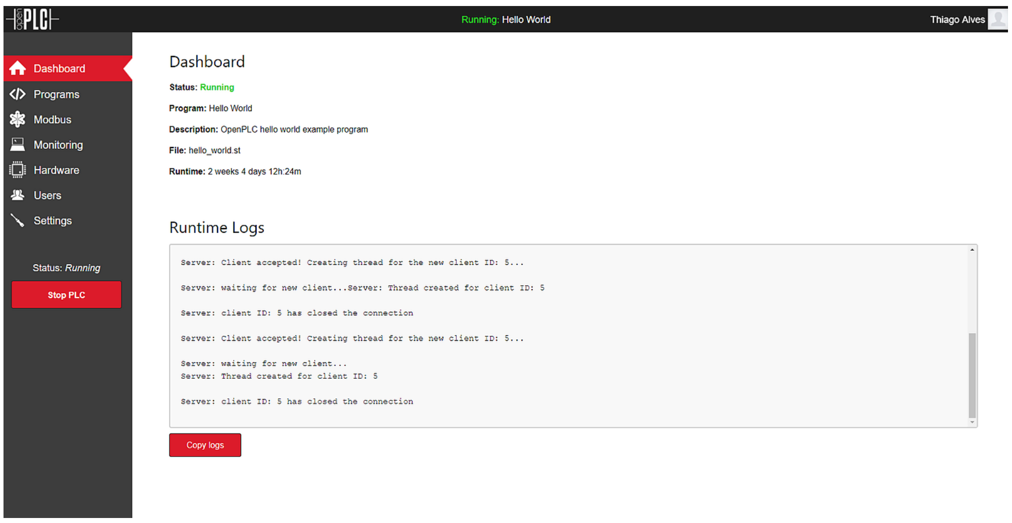
The operator panel can be represented by a container on which various software can be executed, such as a SCADA system or a Python script capable of reading and writing to the PLC's pins. SCADA systems are widely used in industrial automation, and there's plenty of open-source solutions for building it, (for example ScadaBR). Alternatively, a custom Python script can be developed to interface with the PLC, leveraging the pymodbus library that offer several functions to communicate through Modbus.

The PLC will be simulated with OpenPLC Runtime, an open-source project that allows to run PLC programs developed through OpenPLC Editor with one of IEC's standard programming languages: Ladder Logic, Function Block Diagram, Sequential Function Charts, Structured Text and Instruction List.

The intruder can use sniffing tools, such as Wireshark, to capture all Modbus messages exchanged between the HMI and the PLC. In fact, Modbus does not employ any encryption to maintain confidentiality, so all messages are sent in plain text. Exploiting this vulnerability of Modbus, the intruder can carry out various attacks: Reconnaissance attacks, Man-in-the-Middle attacks, Replay attacks and Denial of Service attacks.

OpenPLC Runtime Overview

OpenPLC Runtime features a built-in webserver that allows to configure several parameters of the runtime and to monitor the code execution. However, micro implementations of the OpenPLC Runtime, such as those tailored for microcontrollers and Arduino boards, lack this web server functionality. Instead, configuration adjustments for the micro runtime are managed directly through the OpenPLC Editor upload dialog.


Running: Hello World

Thiago Alves

Dashboard

Status: Running

Program: Hello World

Description: OpenPLC hello world example program

File: hello_world.st

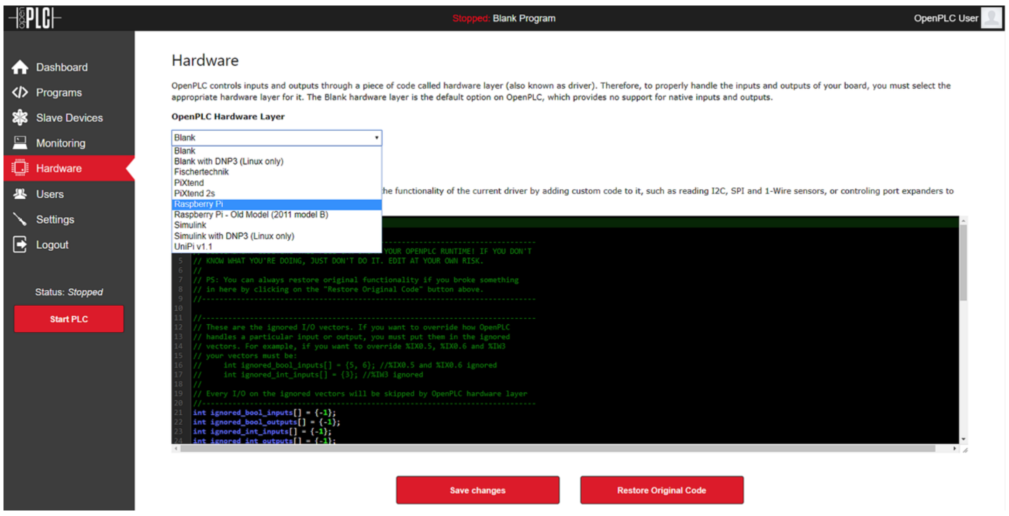
Runtime: 2 weeks 4 days 12h 24m

Runtime Logs

```
Server: Client accepted! Creating thread for the new client ID: 5...

Server: waiting for new client...Server: Thread created for client ID: 5

Server: client ID: 5 has closed the connection

Server: Client accepted! Creating thread for the new client ID: 5...

Server: waiting for new client...
Server: Thread created for client ID: 5

Server: client ID: 5 has closed the connection
```

The main runtime webserver is available at PLC's IP address on port 8080. Upon accessing it, an authentication page should initially appear to ensure its security. Without authentication, anyone connected to the same network as the PLC would have been able to connect to the web server and modify the code executed by the PLC, or even stop it from working. The default username and password are openplc and openplc, this means that the first thing to do after logging in for the first time is to change the default username and password, just going to the Users menu on the left and clicking on the OpenPLC User to change the user informations.

OpenPLC is able, thanks to what is called "the Hardware Layer", to control the GPIO pins of the hardware it is running on. By default, the Blank hardware layer is selected, since the software is being executed on Linux. However, if it is run on a Raspberry Pi for instance, by selecting the appropriate Hardware Layer, it is possible to map the pins defined in the PLC code with those present on the Raspberry Pi board. (GPIO pins documentation: https://autonomylogic.com/docs/2-4-physical-addressing/)


Stopped: Blank Program

OpenPLC User

Hardware

OpenPLC controls inputs and outputs through a piece of code called hardware layer (also known as driver). Therefore, to properly handle the inputs and outputs of your board, you must select the appropriate hardware layer for it. The Blank hardware layer is the default option on OpenPLC, which provides no support for native inputs and outputs.

OpenPLC Hardware Layer

Blank

Blank
Blank with DNP3 (Linux only)
Fischertechnik
PiXtend
PiXtend 2s
Raspberry Pi
Raspberry Pi - Old Model (2011 model B)
Simulink
Simulink with DNP3 (Linux only)
UniPi v1.1

he functionality of the current driver by adding custom code to it, such as reading I2C, SPI and 1-Wire sensors, or controling port expanders to

```
//-----------------------------------------------------------------------------
                                       YOUR OPENPLC RUNTIME! IF YOU DON'T
// KNOW WHAT YOU'RE DOING, JUST DON'T DO IT. EDIT AT YOUR OWN RISK.
//
// PS: You can always restore original functionality if you broke something
// in here by clicking on the "Restore Original Code" button above.
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// These are the ignored I/O vectors. If you want to override how OpenPLC
// handles a particular input or output, you must put them in the ignored
// vectors. For example, if you want to override %IX0.5, %IX0.6 and %IW3
// your vectors must be:
//     int ignored_bool_inputs[] = {5, 6}; //%IX0.5 and %IX0.6 ignored
//     int ignored_int_inputs[] = {3}; //%IW3 ignored
//
// Every I/O on the ignored vectors will be skipped by OpenPLC hardware layer
//-----------------------------------------------------------------------------
int ignored_bool_inputs[] = {-1};
int ignored_bool_outputs[] = {-1};
int ignored_int_inputs[] = {-1};
int ignored_int_outputs[] = {-1};
```

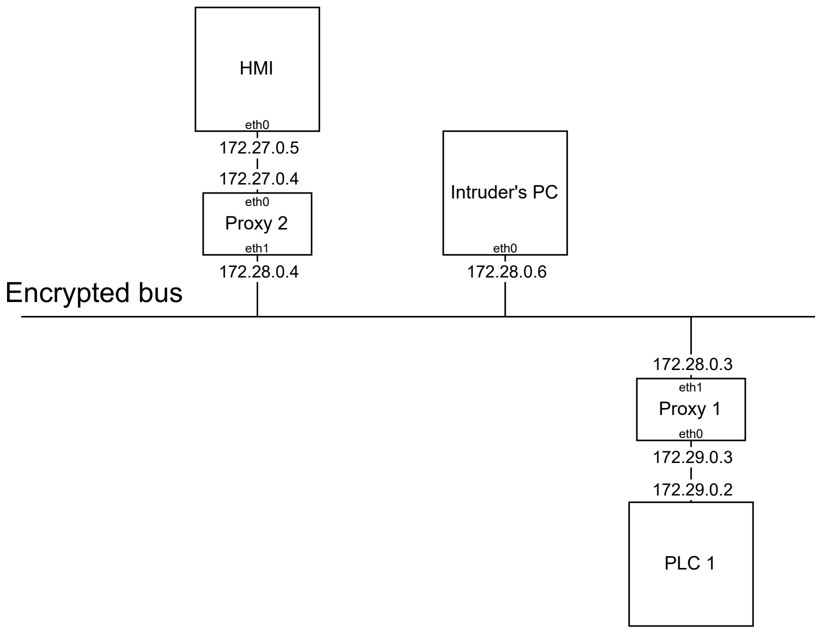
To upload a new program on OpenPLC, navigate to the Programs section located in the left menu. Within this section, a list displays all recently uploaded programs. It's possible to revert to a previously uploaded program, clicking on the desired program in the list and confirming on the subsequent page. For uploading a new program, select "Choose File", then proceed by selecting the .st file and clicking on "Upload Program". On the window that appears it’s possible to fill out some information about the program just to know what it is about next time it will be loaded again. Finally, clicking on “Upload program” the .st file will be loaded into OpenPLC.

Once the file is uploaded, the app will direct back to the dashboard screen and the OpenPLC status will change to “compiling”. The logs about the compilation process are displayed on the runtime logs box. Once the compilation process is finished, the status will change to “running” and the new program will be run. If there was an error on the program, the status will change to “stopped” and the errors will be displayed on the runtime logs box.

Cryptographic protocol

As previously mentioned, the PLC communicates with other devices via Modbus, and the goal of the project is to secure this protocol. There is also a variant called Modbus/TCP Security which, by the name, is a security focused variant of the Mobdbus/TCP protocol utilizing Transport Layer Security, but not all the PLC, especially the older ones support it, so there’s the need to implement from scratch a cryptographic protocol that can by compatible with all kinds of existing PLC.

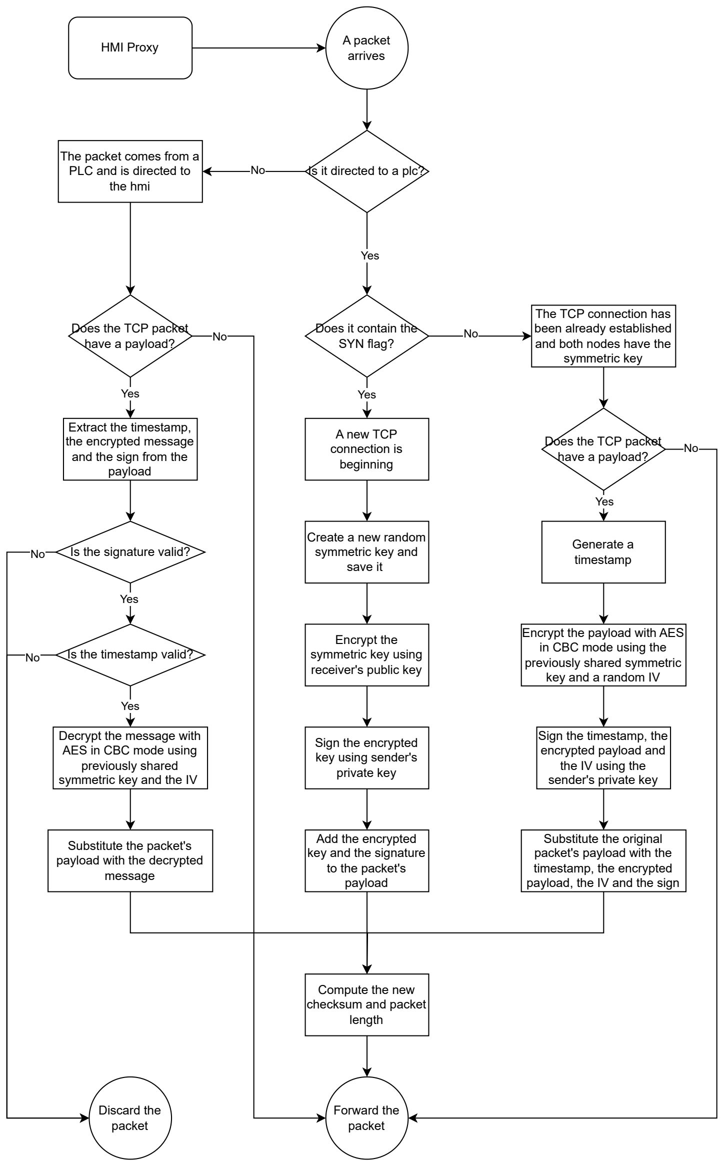
Since common industrial PLCs do not easily support cryptographic methods, it was decided to place a transparent proxy in front of each device on the network. This proxy is capable of intercepting incoming packets and encrypting or decrypting them before forwarding them to the destination node.



With this solution, there will be no need to modify the software running on the PLCs and HMIs. They will operate under the impression that they are communicating directly with each other using Modbus, with no intermediary nodes involved.

In order to add the proxies to the network, it's sufficient to define them in the docker-compose file. Specifically, these are Linux containers running a Python code responsible for intercepting and forwarding packets. However, before allowing the HMI to communicate with the PLC through the proxies, it's necessary to correctly configure the routing tables of each node in the network. For instance, in Proxy 2, one must configure it to forward the packet to Proxy 1 in order to reach the PLC at IP address 172.29.0.2.

The Python code executed in the proxies will be very similar between the two, utilizing the Netfilter Queue, which is a mechanism in Linux that allows users to intercept incoming packets and to send them to a queue where they will be processed and forwarded or dropped. After intercepting a packet, it is analyzed using a Python library called Scapy, which allows for easy packet decoding and modification. The Modbus message is embedded in the packet as the payload of TCP layer and will thus be read by Scapy as raw data. This is, of course, the 'sensitive' data that require encryption. Assuming that the public keys of each node in the main network have been preloaded onto all devices, the following protocol has been implemented (HMI's proxy version):



HMI Proxy
A packet arrives
Is it directed to a plc?
No
Yes
The packet comes from a PLC and is directed to the hmi
Does the TCP packet have a payload?
Yes
No
Extract the timestamp, the encrypted message and the sign from the payload
Is the signature valid?
Is the timestamp valid?
Decrypt the message with AES in CBC mode using previously shared symmetric key and the IV
Substitute the packet's payload with the decrypted message
Does it contain the SYN flag?
A new TCP connection is beginning
Create a new random symmetric key and save it
Encrypt the symmetric key using receiver's public key
Sign the encrypted key using sender's private key
Add the encrypted key and the signature to the packet's payload
The TCP connection has been already established and both nodes have the symmetric key
Generate a timestamp
Encrypt the payload with AES in CBC mode using the previously shared symmetric key and a random IV
Sign the timestamp, the encrypted payload and the IV using the sender's private key
Substitute the original packet's payload with the timestamp, the encrypted payload, the IV and the sign
Compute the new checksum and packet length
Discard the packet
Forward the packet

Following the above protocol, confidentiality is ensured by using symmetric encryption to encrypt the Modbus message, employing a new key for each session. Additionally, the use of RSA keys enables authentication and integrity of the entire packet payload. Finally, the timestamp serves to prevent replay attacks.