Internship Project Documentation

Giuseppe Cervone
giuseppe.cervone@edu.unife.it

1 Introduction

The aim of my internship project with Alstom Group is to start by developing what in cybersecurity and IoT is called a data diode, a unidirectional communication device for data exchange. The data that will be exchanged is diagnostic data caught by network monitoring program Zabbix. Zabbix will catch data from devices inside the protected network with a server running on a Raspberry Pi 3 and recreate the same Zabbix setup with the protected network values on the unprotected network. This Zabbix server will also run on a Raspberry Pi 3. These two devices are also the base behind the diode itself. The importance behind this project is filling a market void created by industrial data diodes, which are very expensive devices. In use cases like this one, where the protected -¿ unprotected network links to make are plenty, implementing a high amount of them is a very costly move, but with software implementations (firewalls) or hardware implementations (serial or optical communication built to be unidirectional) it is possible to have cheaper data diodes that are almost as functional for most use case scenario. As mentioned earlier, the setup includes two Raspberry Pi model 3 computers, and a TTL-232R-3V3 cable with a specific configuration which we will highlight later.

2 Devices

2.1 Raspberry Pi

As far as choosing the best Raspberry Pi model for the job, I have chosen to use the Raspberry Pi 3. For this type of project, finding a machine that can be used without cooling, that has enough RAM to handle possible big data chunks and that has a fast enough CPU at a low enough cost is key. Needing for it not to be cooled removes the Raspberry Pi 4 from the equation, since it prefers having a cooling solution and the increased clock speeds won't help with serial communication limitations, on the other hand the Raspberry Pi 2 already comes with 1Gb Ram, but misses the clock speed needed for multiple programs to run at a good enough speed. This particular machine should be able to handle the data size we are planning to share (around 1Gb/day). The machines are configured as follows:

- Raspberry Pi 3 Model B Ver1.2:
  - Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
  - 1GB RAM
  - BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
  - 100 Base Ethernet

  - 40-pin extended pinout with GPIO support
  - 4 USB 2 ports
  - 4 Pole stereo output and composite video port
  - Full size HDMI
  - CSI camera port for connecting a Raspberry Pi camera
  - DSI display port for connecting a Raspberry Pi touchscreen display
  - Micro SD port for loading your operating system and storing data
  - Upgraded switched Micro USB power source up to 2.5A
  - Raspberry Pi OS Lite (latest version)
  - 16GB microSD card

Using the RPI-Imager tool I installed the latest Raspberry PI OS image. Upon first boot up, remember that RPi login is username: pi, password: raspberry. As Raspberry Pi OS Lite is an OS without a desktop environment, we don't have a systems setting manager with a GUI, luckily we can use the utility *raspi-config* to enable a series of features useful for testing software. Using the command *sudo raspi-config* we can open the utility, once in the main menu we change these specific settings:

- In the system options:
  - Enable WIFI for SSH purposes.
  - Change username and password if needed.
- In the interface options:
  - Enable SSH. Using SSH is suggested for development on these units, as they are installs without a DE/WM(desktop environment/window manager) configuration, and thus it's best to use your preferred terminal emulator.
  - Enable Serial interface, remembering to disable login shell but enable serial interface.
- Disable bluetooth:
  - We won't be using bluetooth for the project, so we disable it, making the final product more secure.
  - We disable bluetooth by adding *dtoverlay=disable-bt* in */boot/config.txt*.
- Troubleshooting
  - Check that UART is enabled in */boot/config.txt* by adding *enable_UART=1*.
  - Check that serial console is disabled by removing *console=serial0,115200* or *console=ttyS0,115200* in */boot/cmdline.txt*.
  - Check for permissions in the dialout group.
  - Check that */dev/serial0* doesn't have a getty console running on it. In case it does, it can be disabled by using the commands: *sudo systemctl stop serial-getty@ttyS0.service* and *sudo systemctl disable serial-getty@ttyS0.service*.
- Additional software:
  - Run *sudo apt-get install python3* to install Python in it's latest version.
  - Run *sudo apt-get install python3-pip* to install Python's package manager.

- Run pip install pyserial to install the serial library that we will be using to have the machines comunicate.

- Run *sudo apt-get upgrade* to make sure all packages are up to date.

2.2 Serial cable

As mentioned earlier, the serial communication will go through a TTL-232R-3V3 cable, which is USB-to-serial, with +3.3V TTL levels UART signals. The cable has 6-pins on one end, and USB on the other. With the cable plugged in we run the command *ls /dev/tty** on both RPi machines so we have an idea of what port we use to send or receive data on each RPi in the serial communication, for the port that's sending data it will be */dev/ttyS0*, while for the other port it should be */dev/ttyUSB0*. Raspian will always map the serial port to the alias */dev/serial0*, so it is suggested to use the alias in programming as it doesn't differentiate between Raspberry Pi Models. Here below in Figure 1 there is an image showing how the cable is meant to be plugged in, while in the Appendix 1 there is the full extract from the documentation of the cable, showing the functionality of all the cable pins in detail.

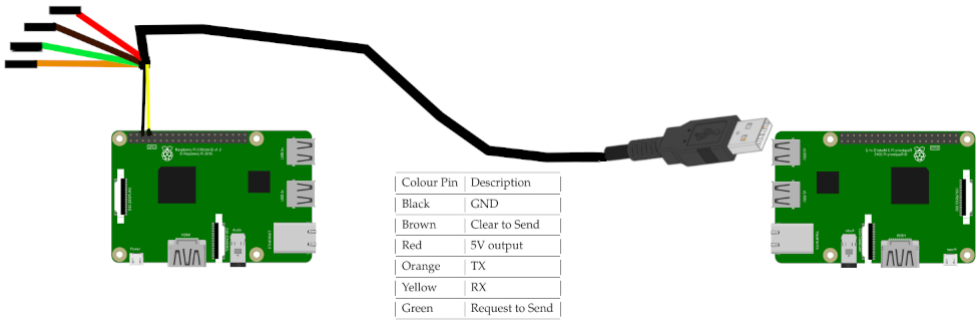

Figure 1: Pinout diagram

The cable pinout is peculiar as most pins are not used. The pins that get used are the ground pin and the pin that sends data over to the other RPi machine. Yellow pin is plugged in GPIO14, ground can be plugged in any of the RPi ground pins.

2.3 How is it a diode?

The importance behind this project is creating a diode-type communication, thus a unilateral comunication from a protected network to less protected or unprotected network. Below shown in Figure 2 how the two RPi interface within the network of Alstom devices, that is the first piece that shows how this setup has diode communication. The second piece is the pinout shown in Figure 1 in the chapter above, we can see that in the RPi placed in the private network, the serial pins that are plugged in are GND and TX, thus no data will be received through the cable by the RPi in the private network (RX pin is not plugged). These two factors create unidirectional communication similar to that of a diode.

3 Program versions

3.1 Diagnostic test

The objective of this test is to check if the two RPi are connected correctly. In Appendix 2 we have example code and example output that can be used to troubleshoot and check that the

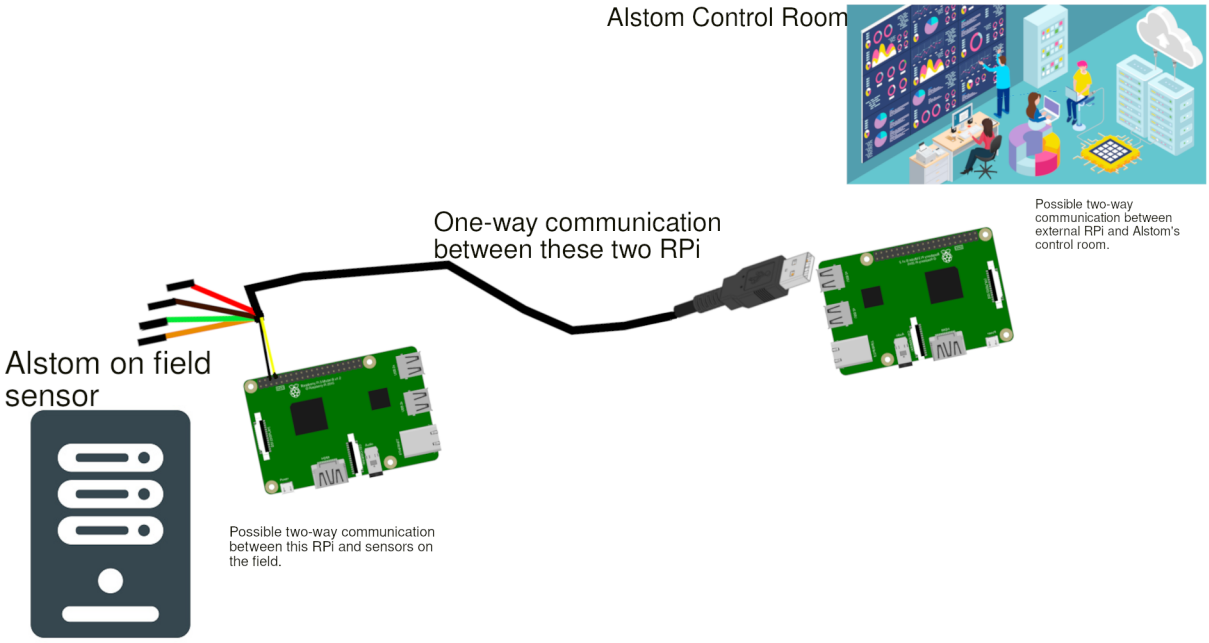

Figure 2: Network diagram

cable has been plugged in correctly. In case this program won't run correctly, refer back to the troubleshooting section earlier. Make sure to run the code *sndtest.py* on the machine where the GPIO pins are being used, and *rcvtest.py* has to be run on the RPi where the USB is plugged in. The scripts are started by running the command *python3 rcvtest.py*, and then run *python3 sndtest.py* on each machine.

3.2 Data diode 0.1

In this initial implementation we switch from sending text to sending an actual file via serial communication. The improvements from this version to the next will be improved baudrate, meaning faster file transfer. In the next version SHA256 checksum will also be implemented to make sure file transfer is working correctly. Future considered improvements are to include a handshake system via CTS/RTS signals and cleaner code in general. The code can be found in Appendix 3.

3.3 Data diode 0.2

This version of the program has SHA256 integration to validate file transfer and improved transfer speeds thanks to a much better baudrate. The limitations with this version of the program is that because of limitations of USB2.0 and limitations of the UART communication protocol, 115200 baudrate seems to be the maximum baudrate for communication without introducing errors, meaning that improving transfer speeds can't be done by increasing the baudrate, but improvements have to be found in other points of the code. A comparative of baudrates can be seen in Table 1, where examples of how increasing baudrates helps with transfer speeds, and also a small example of whether calculating SHA256 while sending the file slows down communication, it will still be wise to split file transfer and calculations. Another important detail to note from the table is the approximate time for 1Gb file transfer, which given the current times might be a little too slow. In appendix 4 there are code exam-

ples for this version of the program as well.

File size (mb)	Time (s)	SHA256
1(9600baud)	1065.9	Yes
1	88.8	No
1	88.8	Yes
5	448	Yes
10	892.7	Yes
20	1785.4	Yes
1000	89250	Yes

Table 1: Transfer times comparative

3.4 Data diode 0.3

The aim with version 0.3 of the program is to initially format the code so as to make it much more readable. As far as technical improvements, this code had the massive flaw of wanting to send any type of file, but but using a transfer protocol built for text files, while at the same time opening the files in byte read as that's the only way of sending data through the serial port. Thus we will include byte read with a chunk size to be defined, intead of *readline()*, so as to make less calls to the read and write methods, also we will be splitting up the SHA256 algorithm from the copying of file contents. Program will be upgraded to version 1.0 when it will be fully optimized for the final app. Code is in the appendix.

4 Network monitoring

4.1 Why do we need network monitoring

Having created a working data diode clone, we move on to the network monitoring part of the project. We need network monitoring software, so as to have a way to create the data that we then transfer via diode. Obviously the data will be, in the final version, a breakdown of the private network diagnostic values caught by the network monitor we choose. This data will be used to create a simulated version of the private network on the public network Raspberry Pi. We will have thus to pick a network manager able to export and import data. *PRTG Network Manager* is considered the best tool for this type of job, but it is only made to run on Windows server, and it is a closed source solution, so ARM based open source solution is preferred. Well known market solutions for network monitoring are the following:

- Cacti
- Nagios
- Zabbix

These solutions are all open source, all working on ARM and are all based on a LAMP stack. LAMP is an acronym for Linux, Apache, MySQL and PHP, but these programs are all flexible enough to be able to interchange Apache with Nginx or MySQL with database management systems such as PostgreSQL.

4.1.1 Comparison between network monitoring software

Starting from the premise that all these softwares allow for export of configuration and data, in picking the best network monitoring software for the job I will also have to evaluate other factors including the feasability of recreating the simulation of the protected network on the unprotected network, Starting with Cacti, it's a stable solution since it has been in development since 2001. This leads sadly to it having less modern features compared to some of the other network managers, and this lack of features leads to Cacti being used less in the user space. Even if some of these features won't be used for our project, picking a program that has lots of users will make debugging for errors easier, as the community will have much more people around it. Cacti has a very straightforward way of importing data into it, but it has very little documentation on the idea of recreating a device which is not actually on the current network, so I decided to look at other options. Nagios is the second options that I analyzed, it is stable, widely used, modern and has a big user space, sadly it has a very limited free version, and it is very comparable to Zabbix, which instead is free. Zabbix is the option I ended up going with for all the reasons mentioned above inlcuing some other factors such as the fact that their Raspberry Pi install guide is also very detailed, it includes the utility *zabbix_sender* which we can use to import data, and has another fan-made utility called *Zabbix Agent Simulator* which we can use to create devices on a network where we don't have one. Being able to do exactly that, and import our data from the private network on those devices should give us the exact solution we are looking for.

4.2 How does Zabbix work?

Zabbix as a network monitoring tool has a series of main components, these are hosts (or agents), items, triggers. Hosts are the devices which we want to monitor,

4.3 Installation guide

Initially I'll install Zabbix on the private Raspberry Pi for testing purposes, after having a working Zabbix on the private Raspberry Pi, I will work with the API and the other tools I mentioned to simulate setups. The install guide I followed is the one on the *zabbix.com/download* websites, which has a detailed guide for exactly our RPi setup. Beware that the MySQL version in the install guide could be outdated, so I suggest updating to the newest version running the command *sudo apt-get install mysql* before creating the database host. I also suggest *mysql_secure_installation* as away to correctly set-up the MySQL installation on the machine.

5 Appendix

5.1 Appendix 1: Documentation extract

4.2 TTL-232R-5V and TTL-232R-3V3 Cable Signal Descriptions

Header Pin Number	Name	Type	Colour	Description
1	GND	GND	Black	Device ground supply pin.
2	CTS#	Input	Brown	Clear to Send Control input / Handshake signal.
3	VCC	Output	Red	+5V output,
4	TXD	Output	Orange	Transmit Asynchronous Data output.
5	RXD	Input	Yellow	Receive Asynchronous Data input.
6	RTS#	Output	Green	Request To Send Control Output / Handshake signal.

Table 4.1 TTL-232R-5V and TTL-232R-3V3 Cable Signal Descriptions

5.2 Appendix 2: Diagnostic output and code example

Listing 1: sndtest.py

```
#!/usr/bin/env python
import time
import serial

ser = serial.Serial(
        port='/dev/serial0',
        baudrate = 9600,
        parity=serial.PARITY_NONE,
        stopbits=serial.STOPBITS_ONE,
        bytesize=serial.EIGHTBITS,
        timeout=1
)
counter=0

while True:
        ser.write(b'Write counter: %d \n'%(counter))
        time.sleep(1)
        counter += 1
```

Listing 2: rcvtest.py

```
#!/usr/bin/env python
import time
import serial

ser = serial.Serial(
        port='/dev/ttyUSB0',
        baudrate = 9600,
        parity=serial.PARITY_NONE,
        stopbits=serial.STOPBITS_ONE,
        bytesize=serial.EIGHTBITS,
        timeout=1
)

while True:
        x=ser.readline()
        print(x)
```

Listing 3: output.txt

```
b'Write counter: 0\n'
b'Write counter: 1\n'
b'Write counter: 2\n'
b'Write counter: 3\n'
b'Write counter: 4\n'
b'Write counter: 5\n'
```

5.3 Appendix 3: Data diode 0.1 code example

Listing 4: sndfile0.1.py

```
import serial
import time
import os
from hurry.filesize import size,alternative

ser = serial.Serial(
        port = "/dev/serial0",
        baudrate = 9600,
        parity = serial.PARITY_NONE,
        stopbits = serial.STOPBITS_ONE,
        bytesize = serial.EIGHTBITS,
        timeout=1
        )
#Initialize serial port

filesize = size(os.path.getsize('log.txt'), system=alternative)
#gets filesize in a nice manner to read,
#will be used for performance purposes

print("Starting transfer... Time started")
start_time= time.time()
#Here we open the file, and keep on reading until there's
#lines, we then send <<EOF>> to signal to the other
#machine that file transfer is over.

f = open("log.txt","rb")
line = f.readline()
while line:
    ser.write(line)
    line = f.readline()

ser.write(b"<<EOF>>\n")

f.close()

end_time = time.time() - start_time

print('Transfer completed... Filesize:',filesize,'. Time:',end_time,'seconds.')
```

Listing 5: rcvfile0.1.py

```
import serial
import time

ser = serial.Serial(
        port = "/dev/ttyUSB0",
        baudrate = 9600,
        parity = serial.PARITY_NONE,
```

```
        stopbits = serial.STOPBITS_ONE,
        bytesize = serial.EIGHTBITS,
        timeout = 1
        )

eof = b'<<EOF>>\n'

f = open("log.txt", "wb")

x = ser.readline()

while x != eof:
    f.write(x)
    x = ser.readline()
f.close()
```

5.4 Appendix 4: Data diode 0.2 code example

Listing 6: sndfile0.2.py

```
import serial
import time
import os
from hurry.filesize import size,alternative
import hashlib

ser = serial.Serial(
        port = "/dev/serial0",
        baudrate = 115200,
        parity = serial.PARITY_NONE,
        stopbits = serial.STOPBITS_ONE,
        bytesize = serial.EIGHTBITS,
        timeout=1
        )
#Initialize serial port

filesize = size(os.path.getsize('log.txt'), system=alternative)
#gets filesize in a nice manner to read, will be used for performance purposes

sha256_hash = hashlib.sha256()
#we will calculate the SHA256 of the file to verify integrity.

f = open("log.txt","rb")
print("Starting transfer... Time started")
start_time= time.time()
#Here we open the file, and keep on reading until there's lines
#we then send <<EOF>> to signal to the other machine that file transfer is over.
ser.flushOutput()
line = f.readline()
while line:
    sha256_hash.update(line)
    ser.write(line)
    line = f.readline()
end_time = time.time() - start_time
f.close()
ser.write(b"<<EOF>>\n")

sha256_hash = sha256_hash.digest() #digest transforms into string
ser.write(sha256_hash)


print('Transfer completed... Filesize:',filesize,'. Time:',end_time,'seconds.')
```

Listing 7: rcvfile0.2.py

```
import serial
import time
import hashlib
```

```
ser = serial.Serial(
        port = "/dev/ttyUSB0",
        baudrate = 115200,
        parity = serial.PARITY_NONE,
        stopbits = serial.STOPBITS_ONE,
        bytesize = serial.EIGHTBITS,
        timeout = 1
        )
#initialize serial port

eof = b"<<EOF>>\n"
#define EOF, signals to the receiver that the file is over

sha256_hash = hashlib.sha256()
#we will calculate the SHA256 of the file
#to verify it's integrity after transfering

f = open("log.txt", "wb")
#open file to write into
ser.flushInput()
x = ser.readline()
while x != eof:
#until file comes, decode, if x=eof then check if file transfered correctly
    sha256_hash.update(x)
    #decoding is done in bytes, so we make sha before making it in bits
    f.write(x)
    x = ser.readline()
f.close()

sha256_send = ser.read(32)
if sha256_send != sha256_hash.digest():
    print("File Transfer failed")
else:
    print("File Transfer success")
```