UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA

DIMES

COMPUTER ENGINEERING FOR THE INTERNET OF THINGS

Project Report

MITM Attack on a fire alarm system utilizing MQTT protocol

for the exam

IoT Security

Professor

Prof. Antonio Guerrieri
Prof. Michele Ianni

Student

Mario Pingitore - 225638

February 2024

Academic Year 2023-2024

Summary

1. Introduction

The **Internet of Things (IOT)**, even if largely applied nowadays in a lot of scenarios, lack a fixed definition and, in fact, we can try to describe this technology based on the different way the stakeholders approach it. If we think about **Things**, we of course focus our attention on devices and their "smart" behavior, so they can for example have sensors and actuators to perceive and react to the environment in which they are deployed, they can have process ability to make computation and can have some sort of tags (like RFID), to be identifiable in a larger infrastructure. If we think about **Internet**, we focus instead on the network characteristics that connect these tiny, low power devices.
However, in more modern application of the technology, the concept of IoT is expanded, so that almost everything can be seen as "Things", for example people, physical and digital objects and also representation of objects; the interconnection between these things has to be seen only as a way to create a framework in which everything is connected and works together to build a large ecosystem.
Having such large domain, the IoT sure has a lot of application scenarios, like Industry 4.0, Medical and Healthcare, Quality control, Logistic and supply chain optimization, House Automation, Smart Cities, Safety and Security.
In this report will be exploited a simple IoT implementation, a small IoT WSN which consists of a smart device that performs some readings from the environment in which they are located, send the data to server application to be processed and the result is forwarded to another device equipped with a display as actuator, to show the mentioned result.
The report is structured as it follows:

- Chapter 2: the hardware components of the architecture such as board, sensors and so on, are presented and discussed;
- Chapter 3: the software technologies and solutions utilized for the logic of the system for the testing, are analyzed in detail;
- Chapter 4: all the tools and the software utilized by the attacker are described;
- Chapter 5: the system is presented and discussed in its entirety, from the code used in the device and the backend logic;
- Chapter 6: the various steps of the MITM attack are described in detail;
- Chapter 7: possible security solution and how the attack is conduct when such measures are in place;
- Chapter 8: a collection of the results obtained, and the problems encountered, but also ideas and suggestions for future improvements.

2. Hardware

In this chapter will be presented all the hardware components utilized in the building of the architecture. The main protagonists are certainly the two boards utilized, the Arduino and the ESP32. For this project, the two boards are physically connected to the PC to be powered, but it's possible to equip them with rechargeable batteries through apposite modules, to ensure autonomous execution.

2.1 Boards

2.1.1 Arduino UNO R3 (ELEGOO)

The Board utilized is a dupe version of the Arduino UNO R3 and such as that, it is a microcontroller board based on the ATmega328P chip (that can be easily replaced, as it is not soldered on the board). It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header and a reset button. It contains everything needed to support the microcontroller; it just needs to be connected to a computer with a USB cable or power it with an AC-to-DC adapter or battery to get started. It features 1kb of EEPROM, a memory which is not erased when powered off. The board can also be connected and integrated with a lot of sensors or modules, to design almost every kind of amateur, and maybe semi-professional, project.

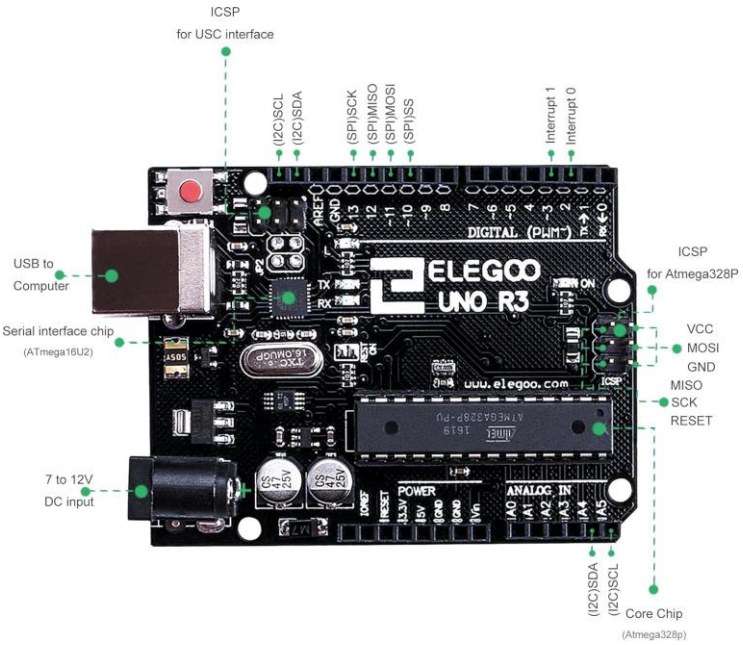


Figure 1:The Elegoo Uno R3

2.1.2 ESP32

ESP32 is a low-cost System on Chip (SoC) Microcontroller from Espressif Systems. It is the successor to the ESP8266 SoC and comes in both single-core and dual-core variations of the Tensilica's 32-bit Xtensa LX6 Microprocessor with integrated Wi-Fi and Bluetooth. The good thing about ESP32 is its integrated RF components like Power Amplifier, Low-Noise Receive Amplifier, Antenna Switch, Filters and RF Balun. This makes designing hardware around ESP32 very easy as very few external components are required.
It features the following characteristics:

- I/O Pins: 34
- Interfaces: SPI, I2C, I2S, CAN, UART
- Protocols Wi-Fi: 802.11 b/g/n (802.11n up to 150 Mbps)
- Frequencies Wi-Fi: 2.4 GHz - 2.5 GHz
- Bluetooth: V4.2 - BLE e classic Bluetooth
- Wireless Antenna: PCB
- SoC: ESP32-WROOM 32
- Clock Frequency: 80MHz / 240MHz
- RAM: 512kB
- External flash memory: 4MB

Figure 2: The ESP32

2.2 Sensors and modules

2.2.1 DHT11 Module

The DHT11 is a basic, ultra low-cost digital temperature and humidity sensor. It uses a capacitive humidity sensor and a thermistor to measure the surrounding air and spits out a digital signal on the data pin (no analog input pins needed). It's fairly simple to use but requires careful timing to grab data. In fact, the only real downside of this sensor is that you can only get new data from it once every 2 seconds, so when using its library, sensor readings can be up to 2 seconds old.
Compared to the DHT22, this sensor is less precise, less accurate, and works in a smaller range of temperature/humidity, but its smaller and less expensive. Following, the technical detail of the sensor:

- 3 to 5V power and I/O
- 2.5mA max current use during conversion (while requesting data)
- Good for 20-80% humidity readings with 5% accuracy
- Good for 0-50°C temperature readings ±2°C accuracy
- No more than 1 Hz sampling rate (once every second)
- Body size 15.5mm x 12mm x 5.5mm

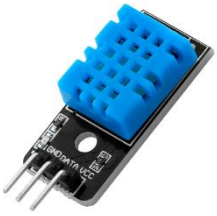


Figure 3: The DHT11 module

The module utilized in the project is already equipped with the 10k resistor necessary for the utilization of the sensor and features 3 pins instead of four: output, power and ground.

2.2.2 LCD Display

The display utilized in this project is a 16x2 LCD display compatible with the Hitachi HD44780 driver. The LCDs have a parallel interface, meaning that the microcontroller has to manipulate several interface pins at once to control the display. The interface consists of the following pins:

- A **register select (RS)** pin that controls where in the LCD's memory we're writing data to. We can select either the data register, which holds what goes on the screen, or an instruction register, which is where the LCD's controller looks for instructions on what to do next.
- A **Read/Write (R/W)** pin that selects reading mode or writing mode
- An **Enable pin** that enables writing to the registers
- **8 data pins (D0 -D7).** The states of these pins (high or low) are the bits that we are writing to a register when we write, or the values we are reading when you read.

Figure 4: The 16x2 LCD display

There's also a display contrast pin (Vo), power supply pins (+5V and GND) and LED Backlight (Bklt+ and BKlt-) pins that we can use to power the LCD, control the display contrast, and turn on and off the LED backlight, respectively.

The process of controlling the display involves putting the data that form the image of what you want to display into the data registers, then putting instructions in the instruction register.

3. Software and communication protocols for the system

3.1 Arduino IDE

The Arduino Integrated Development Environment - or Arduino Software (IDE) - contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus and was used to write the code for both the boards. Programs written using Arduino Software (IDE) are called sketches and saved with the file extension *.ino*. It allows, other than write the program, to upload it on the board and to visualize the serial output through a serial monitor. It's also possible to utilize many kinds of **Libraries**, to provide extra functionality for use in sketches, e.g. working with hardware or manipulating data; to utilize them is just necessary to insert one or more **#include** statement on top of the sketch, one for each library. In this specific project, I have used the legacy libraries **DHT Sensor Library** (by Adafruit) and the **LiquidCrystal Library**, to respectively work, as the name implies, with the DHT module and with the LCD Display.

3.2 Node-RED

Developed in 2013 by IBM's Emerging Technology Services team and now a part of the OpenJS Foundation, **Node-RED** is a powerful general-purpose development tool that exploits flow-based paradigm (FBP) for visual programming applied to Web-of-Things applications, allowing to connect together hardware devices, APIs and Web Services. At the base of the flow programming there is the **node**, that is an atomic element that can be seen as a sort of *black box*: it receives some data in input, it does something with that data and then it passes that data on in single or multiple output.
The network is responsible for the flow of data between the nodes. Nodes are triggered by either receiving a message from the previous node in a flow, or by waiting for some external event, such as an incoming HTTP request, a timer or GPIO hardware change. They process that message, or event, and then may send a message to the next nodes in the **flow**.
Node-RED provides a very rich and operational basic set of nodes, each one specialized in a single task and graphically diversified from the others with different colors and icons. This set includes nodes belonging to these typologies:

- ***common***, that is the basic core and contains some of the most used nodes, like *inject* or *debug*;
- ***function***, that provides useful nodes for manipulating data, like the function node in which it's possible to write custom code;
- ***network***, where we find all the networking related node, such as nodes for TCP or HTTP connection;
- ***sequence***, that deals with temporal flows of data;
- ***parser***, that transform data and convert them in various formats, like json or xml;
- ***storage***, containing nodes that allows the developer to work with files.

Moreover, the palette of nodes can be easily extended by installing new nodes and modules created by the community and the flows you create can be easily shared as JSON files or saved for re-use.
Every flow can be composed by other sub-flow or by simple nodes, connected between each other through **wires**: thanks to this connection, the nodes can exchange data in the form of **messages**

(msg), simple JavaScript objects that can be tailored by the developers and can have any set of properties, including the *topic*, that contains the subject of the message, and the *payload*, that contain the most useful information, given that it contains the message itself.
To store information and variables that can be shared among nodes without wiring them and using a message, you can use the **context**, which is available in three modalities:

- *Node*: the variables are only visible to the node that set them;
- *Flow*: the variables are visible to all nodes on the same flow (or tab in the editor);
- *Global*: the variables are visible to all nodes in the workspace.

By default, Node-RED uses an in-memory Context store so values do not get saved across restarts, but it can also be configured to use a file-system based store to make the values persistent.
Node-RED consists of a Node.js based runtime loaded on a web browser that allows to access the flow editor. Within the browser we create the application by dragging nodes from your palette into a workspace and start to wire them together. With a single click, the application is deployed.
The light-weight runtime is built on Node.js, taking full advantage of its event-driven, non-blocking model: this makes it ideal to run at the edge of the network on low-cost hardware such as the Raspberry Pi or any other board, as well as in the Cloud.
Node-RED is distributed for a wide variety of platforms and is among the most valid tools available for web-based IoT app, since it allows to develop projects in a fast and concise way, even if the lack of a real framework for testing and debugging, can sometimes become an issue for the developers.
The software was used to make the server application.

4. Software and tools of the attacker

4.1 Kali Linux

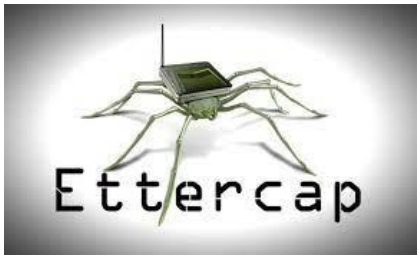
The operate the attack, a virtual machine containing the Kali Linux distribution has been utilized (specifically, the pre-built VM provided by the official site, available on the following link -> https://www.kali.org/get-kali/#kali-virtual-machines) .
Kali Linux (formerly known as BackTrack Linux) is an open-source, Debian-based Linux distribution aimed at advanced Penetration Testing and Security Auditing. It does this by providing common tools, configurations, and automations which allows the user to focus on the task that needs to be completed, not the surrounding activity.
Kali Linux contains industry specific modifications as well as several hundred tools targeted towards various Information Security tasks, such as Penetration Testing, Security Research, Computer Forensics, Reverse Engineering, Vulnerability Management and Red Team Testing. The distribution offers the following features:

- **More than 600 penetration testing tools included:** Among them, the ones utilized in the project, which will be discussed in the following paragraph;
- **Free:** Kali Linux, like BackTrack, is completely free of charge and always will be;
- **Open source Git tree:** All of the source code which goes into Kali Linux is available for anyone who wants to tweak or rebuild packages to suit their specific needs;
- **FHS compliant:** Kali adheres to the Filesystem Hierarchy Standard, allowing Linux users to easily locate binaries, support files, libraries, etc;
- **Wide-ranging wireless device support:** A regular sticking point with Linux distributions has been support for wireless interfaces;
- **Custom kernel, patched for injection:** As penetration testers, the development team often needs to do wireless assessments, so the kernel has the latest injection patches included;
- **Developed in a secure environment:** The Kali Linux team is made up of a small group of individuals who are the only ones trusted to commit packages and interact with the repositories, all of which is done using multiple secure protocols;
- **GPG signed packages and repositories:** Every package in Kali Linux is signed by each individual developer who built and committed it, and the repositories subsequently sign the packages as well;
- **Multi-language support:** Although penetration tools tend to be written in English, Kali includes true multilingual support, allowing more users to operate in their native language and locate the tools they need for the job;
- **Completely customizable:** Easy to customize Kali Linux all the way down to the kernel;

- **ARMEL and ARMHF support:** Since ARM-based single-board systems like Raspberry PI, are becoming more and more prevalent and inexpensive, the Kali's ARM support provide fully working installations for both ARMEL and ARMHF systems. Kali Linux is available on a wide range of ARM devices and has ARM repositories integrated with the mainline distribution so tools for ARM are updated in conjunction with the rest of the distribution.

4.2 Tools

4.2.1 Ettercap



Ettercap is an open-source tool that can be used to support man-in-the-middle attacks on networks. The application can capture packets and then write them back onto the network and also enables the diversion and alteration of data virtually in real-time. Ettercap can also be used for the protocol analysis necessary to analyse network traffic.

The tool provides a Graphical User Interface (UI) as well as a command line interface. While it can support network traffic analysis, the most frequent use of Ettercap is to set up man-in-the-middle attacks using ARP poisoning. The program also supports both active and passive deep analysis of many protocols and includes many features for network and host analysis. Many "sniffing" modes are available – this includes MAC based, IP based, ARP based (full duplex), and PublicARP based (half duplex). Ettercap can also detect a switched local area network (LAN) and use the OS fingerprints to determine the total geometry of the LAN. The tool is already present in the pre-built Kali Linux VM.

4.2.2 hping3

hping3 is a network tool able to send custom ICMP/UDP/TCP packets and to display target replies. It handles fragmentation and arbitrary packet body and size and can be used to transfer files under supported protocols. Using hping3, it's possible to test firewall rules, perform (spoofed) port scanning, test network performance using different protocols, do path MTU discovery, perform traceroute-like actions under different protocols, fingerprint remote operating systems, audit TCP/IP stacks, etc. hping3 is scriptable using the Tcl language.
Also this tool is already provided in the Kali Linux VM.

4.2.3 Wireshark

Wireshark is a widely used, open-source network analyser that can capture and display real-time details of network traffic. It is particularly useful for troubleshooting network issues, analysing network protocols and ensuring network security and can offer an in-depth view into network activities, diagnose network performance issues or identify potential security threats.

Wireshark seeks to simplify and enhance the process of network traffic analysis. Each function is designed to offer unique insights and control over network activities. Here are some of its core features:

- **Packet capture (PCAP):** Converts network traffic into a human-readable format, making it easier to understand and diagnose concerns;
- **Real-time analysis:** Provides a live view of network traffic, offering immediate insights into ongoing network activities;
- **Filtering capabilities:** Enables users to focus on specific types of network traffic, making analysis more efficient and targeted;
- **Graphical user interface (GUI):** Designed for ease of use, ensures that both beginners and experts can navigate and analyze data effectively.

The application can be used to examinate the details of traffic at a variety of levels, ranging from connection-level information to the bits constituting a single packet.
PCAP can provide a network administrator with information about individual packets, including transmit time, source, destination, protocol type and header data. This information can be useful for evaluating security events and troubleshooting network security device issues.
Wireshark's capabilities extend beyond just monitoring to address other network administration tasks:

- **Network troubleshooting:** Pinpoints and resolves network issues with the comprehensive data Wireshark provides;
- **Security analysis:** Detects and analyzes potential security threats in the network;
- **Performance analysis:** Monitors and optimizes network performance to ensure smooth operations;
- **Protocol analysis:** Gains insights into the behavior of individual protocols within the network.

5. Implementation of the system

Until now, the report showed which were the hardware component of the project and the software applications and tools utilized to build the whole architecture. From now on, the practical realization of the project will be discussed in greater detail, started focusing on the aim of the project, its workflow and the allowed user interactions. Next, the functionalities implemented and the hardware architecture upon which the project is built will be discussed and, of course, which network protocol were used to create the communication between the devices and the server application. In the end, the report will show explanation about the code of the two devices as well as the backend logic written on Node-RED.

5.1 Workflow of the system

Before starting with the workflow of the system, it's better to further clarify what is the aim of the assignment. In this project, the user will utilize an ESP32 device equipped with a DHT11 module to sense data from the environment and will send them to a Node-RED server application, which will transform and process the data and will send the result to the Arduino that, equipped with the LCD display, will show the result of the processing operations. In particular, the ESP32 device, will sense temperature in the environment every ten seconds and will simply forward them to Node-Red through the MQTT protocol. Upon receiving the data, the server will do some operation based on which three different outputs can be returned:

- **Alarm 1 – Abnormal value:** in this case, a very high value is received and so the corresponding alarm code will be sent to the Arduino;
- **Alarm 2 – Spike of temperature:** the system store 5 consecutive readings from the ESP32 and, if between a reading and another there is a difference of at least 10 °C, the respective alarm code will be sent;
- **Alarm 3 – Increasing temperature:** the system utilizes again the 5 consecutive readings and, if the temperature increases of at least 2 °C between each reading, a corresponding alarm code is sent;
- **0 – OK!:** If no one of these cases happen, then no problem is raised and so the only data sent to the Arduino will be the detected temperature.

The entire process is fully automated once the two devices are activated, and the server application is up and running.

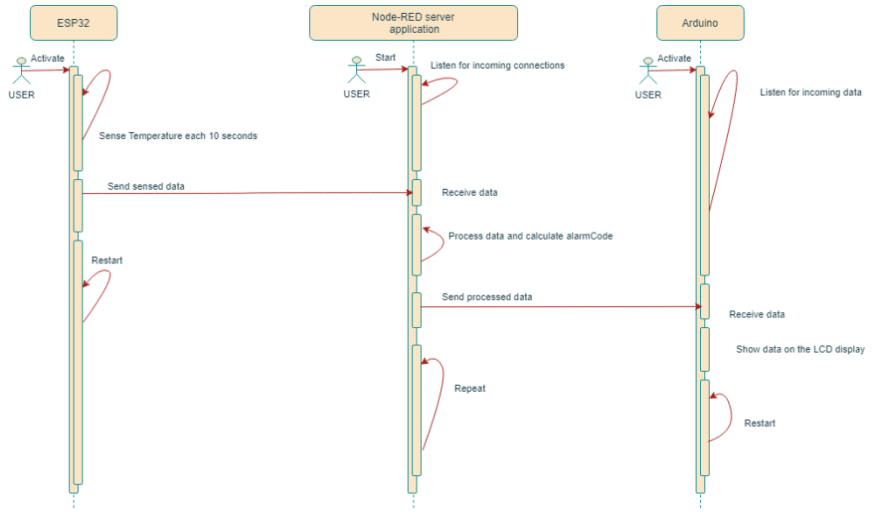


Figure 5: Sequence Diagram of the system

5.2 Hardware architecture

Regarding the hardware architecture of the system, as mentioned before, we have an ESP-32 equipped with a DHT sensor module and an Arduino device provided with an LCD display. The first device is used only to sense the temperature and send it and the other one only to show the detected value and, possibly, different alarm code. The ESP32 is connected to the Node-RED server application through the MQTT protocol, which in turn is connected to the Arduino through a serial communication. The server application is launched from PC and is connected to the same router to which the ESP32 is connected. The architecture is summarized in the following picture.

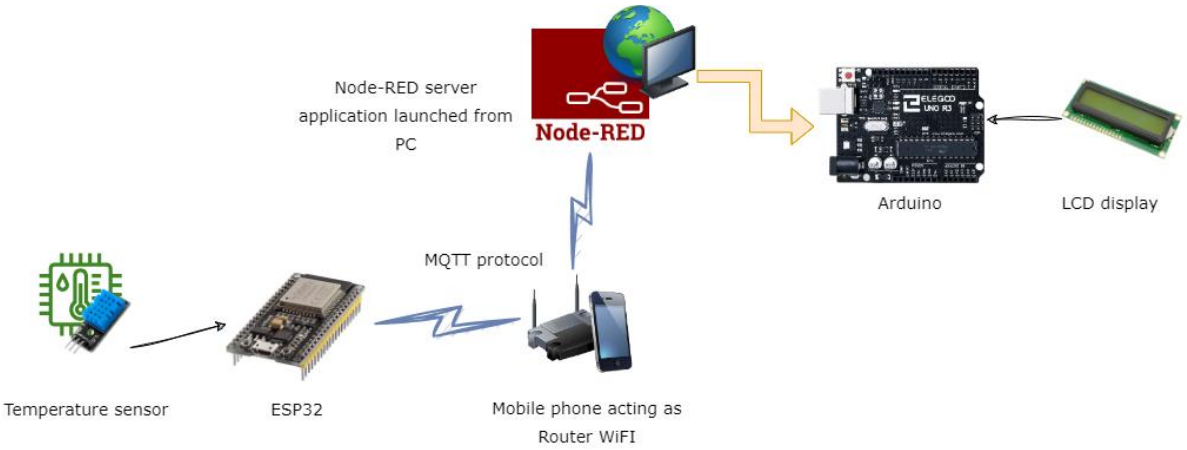


Figure 6: Architecture of the system

5.3 Network protocol

5.3.1 MQTT protocol

The **MQTT** (Message Queue Telemetry Transport) protocol is the most widely used communication protocol for IoT. Invented more than 20 years ago by Andy Stanford-Clark from IBM and Arlen Nipper, it started to become popular in the early '00 with the massive use of sensors, interconnected devices and M2M communications. Its success is due to two main factors:

1. ease of use;
2. lightweight and efficient exploitation of the bandwidth.

The protocol is based on the architectural pattern **publish-subscribe** (see Fig. 7), which implements an asynchronous communication system between two or more clients, which are named *publisher* or *subscriber* based on their role, and a central server in charge of routing all the messages. In order to better understand the MQTT protocol and the pattern to which it's designed, it is appropriate to introduce some terms:

- **Topic**: It's an endpoint to which various clients connect. Publishers and subscribers can communicate each other because they're linked to the same topic, represented by an UTF-8 string. Topics are located in the central server, the *broker*;
- **Publisher**: It's a client able to send (*publish*) messages to the selected topic. Fundamental, for the publisher, is knowing the name of the topic and the address of the broker that hosts the topic;
- **Subscriber**: The other type of client considered in this protocol is the receiver of the messages created by the publisher. The subscriber must connect to a broker and "subscribe" to a specific topic. Whenever a publisher sends a message to that topic, the subscriber receives it;

- **Broker**: Is the central server involved in managing topics and communications between publishers and subscribers, forwarding all the messages written by the publishers to the subscribers that are "listening" to that specific topic. Since it is the central point of communication, the broker is often designed to be hosted by a distributed system, in order to avoid problems like *bottleneck* or *single point failure*. Currently there are lots of broker software designed to work locally. *Mosquitto* is probably the most famous of this category, creating a LAN network among few interconnected devices, or remotely, with cloud brokers that are able to manage a huge number of devices contemporarily.

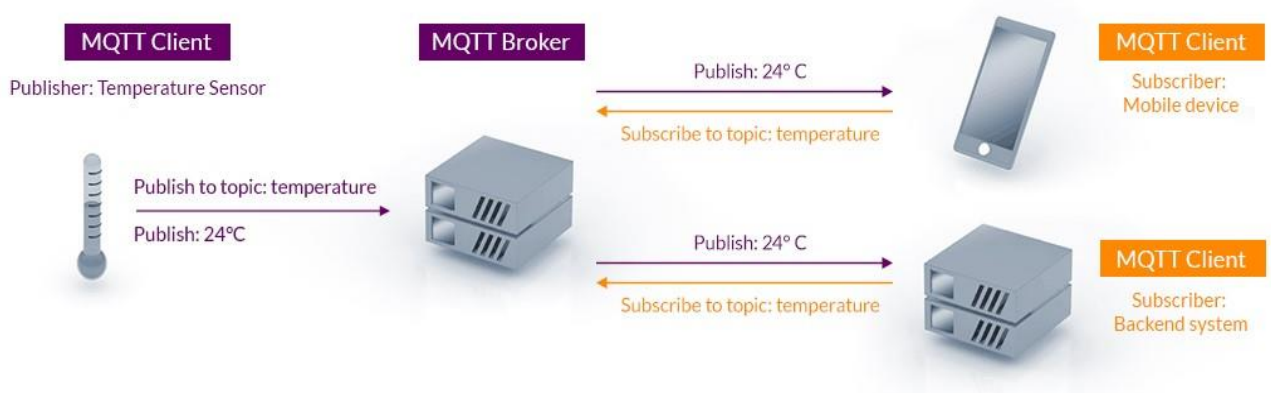


Figure 7: MQTT protocol

MQTT connections are sent by default to the port TCP/UDP 1883, which, however, doesn't grant any security to users (and this is still now one of the weakest points of the MQTT protocol). To make up for it, some providers offer the possibility to also connect to the broker by using other ports, such as 8883 or 8884, that support *TLS/SSL* encryption and authentication.
MQTT supports 3 different levels of **Quality of Service**, both for "publisher to broker" communications and for "broker to subscriber" ones:

- ***Qos 0*** *(At most once)*. The sender sends a QoS 0 PUBLISH packet, containing the protocol header and the payload (the actual content of the message), to the receiver, that doesn't send back any type of acknowledgement. There's no way, for the sender, to understand if the message has been received;
- ***QoS 1*** *(At least once)*. The sender sends a QoS 1 PUBLISH packet containing the message, that is temporary stored in memory. The receiver will provide to send a PUBACK as soon as it receives the PUBLISH packet. If no PUBACK is received within a reasonable amount of time, the sender resends the same message. In this way a sender is sure that the receiver gets the message once, but is also possible for the latter to receive the message multiple times;
- ***QoS 2*** *(Exactly once)*. It implements a 4-parts handshake between the sender and the receiver, managing to be the safest but also the slowest quality of service level. The sender sends a QoS 2 PUBLISH packet and stores the message in the memory, waiting for the response of the receiver. The latter, as soon as it receives the packet, sends back a PUBREC packet that acknowledges the previous PUBLISH one. If the sender doesn't receive this packet within a reasonable amount of time, it sends back the PUBLISH with a DUP (duplicate) flag until it receives an acknowledgement. As soon as the sender receives the PUBREC, that it can safely discard the previously stored PUBLISH packet, storing in its place the just received PUBREC and sending back a PUBREL. After the receiver gets this packet, then it can discard all the stored states and answer with a PUBCOMP. This is important to avoid processing the message a second time. With this QoS level, both parts are sure that the message was received without having duplicates.

Anyway, none of the three cases can directly ensure to a publisher that some subscriber has actually read the message, maybe because there are no active subscribers when the message is created, so its content is lost. In order to avoid it, publishers can use the **retain** flag applied to the message. When brokers receive a message with the retain flag set to true, they store it inside the topic and send it every time a new subscriber joins the topic. Another useful feature of this protocol is the so-called "**Last Will and Testament**", that allows to handle sudden and unexpected disconnections of some client from the broker.
The MQTT protocol has been chosen for this project because it's a widely supported one, especially among low-powered devices like ESP32 boards being also very power-efficient.

5.4 Software code in the boards

In the following subchapters, the code flashed on the two devices will be analyzed, and every method or function will be explained in detail. As said before, the boards have been programmed by using Arduino IDE. This way of programming boards consist in developing two mandatory methods, *setup()* and *loop(),* that necessarily have to include all the functions and operation that the device have to execute. Both sketches are composed by these main parts, in order from the top of the file:

- Inclusion of used libraries, in the first code lines;
- Definition of constants and creation of global variables;
- Definition of all the methods invoked in the *setup()* and *loop()* methods and the two methods themselves.

All the code snippets here shown have truncated comments for space reasons, but the latter are of course present in their entirety in the code attached to this report.

5.4.1 ESP32-DHT

For first, we include the required library to work with the MQTT protocol and to utilize the DHT module. After that, the type of DHT module is defined, as well as the ESP32 pin utilized for the output. After these preliminary operations, the client object is defined by providing name and password of the network to which the device is connected, the address and credentials of the MQTT broker, the client name and the port utilized. In the *setup()* method, the serial communication and the DHT object are initialized and some standard MQTT client features are set. In the *onConnectionEstablished()*, the client subscribe to the requested topic (building/controlTemperature) and to the wildcard topic; the payload is then printed on serial output to check if everything went according to the plan. Finally, in the *loop()* method , we read the data from the DHT module and check if anything has gone wrong; after that, we print on serial output the sensed temperature and publish it on the required topic. A delay of 10 second is established between each loop.

```ino
//import of the libraries
#include "EspMQTTClient.h"
#include "DHT.h"

//definition of the variables
#define DHTPIN 21
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);

//definition of the objects
EspMQTTClient client(
  "Xiaomi 13T Pro", //WiFI name
  "lullabi9(", //WiFi password
  "192.168.4.204",  // MQTT Broker server ip
  //"piccolo",   // MQTT Username
  //"pimpo", // MQTT Password
  "ESP32DHT", // Client name that uniquely identify your device
  1883 // The MQTT port, default to 1883. this line can be omitted
);


void setup()
{
  Serial.begin(9600);
  dht.begin();
  // Optional functionalities of EspMQTTClient
  client.enableDebuggingMessages(); // Enable debugging messages sent to se
  client.enableHTTPWebUpdater(); // Enable the web updater
  client.enableOTA(); // Enable OTA (Over The Air) updates
  client.enableLastWillMessage("ESP32DHT/lastwill", "I am going offline");
  Serial.println(client.isConnected());
}
```

```ino
// This function is called once everything is connected (Wifi and MQTT)
void onConnectionEstablished()
{
  // Subscribe to "building/controlTemperature" and display received message to
  client.subscribe("building/controlTemperature", [](const String & payload) {
    Serial.println(payload);
  });

  // Subscribe to "building/controlTemperature/#" and display received message t
  client.subscribe("building/controlTemperature/wildcard/#",
                   [](const String & topic, const String & payload) {
    Serial.println("(From wildcard) topic: " + topic + ", payload: " + payload);
  });
}


void loop()
{
  client.loop();
  // Read relative humidity
  float hum = dht.readHumidity();
  // Read temperature as Celsius (the default)
  float temp = dht.readTemperature();
  // Read temperature as Fahrenheit (isFahrenheit = true)
  float far = dht.readTemperature(true);
  if (isnan(hum) || isnan(temp) || isnan(far)) { //If some value can be retrieve
    Serial.println(F("Failed to read from DHT sensor!"));
    return;
  }

  Serial.print(F("%  Temperature: "));
  Serial.print(temp);
  String stringTemperature = String(temp);
  client.publish("building/controlTemperature", stringTemperature); // It's poss
  delay(10000);  //set delay to 10 second between each reading
}
```

Figure 8: The ESP32-DHT code

5.4.2 ARDUINO-LCD

First thing first, the library necessary to utilize the LCD display is imported, and the lcd object is defined, as well as the Arduino pin that will be utilized. In the *setup()* method, the lcd object and the serial communication are initialized. In the *loop()* method, we receive the string from Node-RED containing the alarmCode and the sensed temperature in the following format "alarmCode-temperature-". From this string we create two different substring, one for each value. At this point, based on the different code received from the server, different messages will be shown on the display, together with the temperature, which in turn is always showed:

- **Code 1:** Abnormal Temperature;
- **Code 2:** Spike of Temperature;
- **Code 3:** Increasing Temperature;
- **Code 0:** Normal reading with no alarm.

```ino
//import libraries
#include <LiquidCrystal.h>

//define object and pin utilized
LiquidCrystal lcd(7, 8, 9, 10, 11, 12);


void setup()
{
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  Serial.begin(9600);
}


void loop()
{
  while(Serial.available()>0){
    //receive from node-RED a string containing the alarmCode and the temperature
    String reading=Serial.readString();
    //generate two different substring for the two values
    int firstIndex = reading.indexOf("-");
    String firstSubstring = reading.substring(0, firstIndex);
    int alarmCode=firstSubstring.toInt();
    int secondIndex = reading.indexOf("-", firstIndex + 1);
    String temperature = reading.substring(firstIndex + 1, secondIndex);

    Serial.println(alarmCode);
    Serial.println(temperature);
    long starttime = millis(); //timer to utilize in the display
    long endtime = starttime;
    switch(alarmCode){ //based on the alarm code, different messages are displayed
      case 1:   //Abnormal Temperature
        while ((endtime - starttime) <=5000){// do this loop for up to 5000mS
          lcd.print("    " + temperature + " " +  ((char)223) + "C" );
          lcd.setCursor(0, 1);
          lcd.print("     ALARM!     ");
          delay(500);
          lcd.clear();
          delay(500);
          lcd.print("ABNORMAL");
          lcd.setCursor(0, 1);
          lcd.print("TEMPERATURE");
          delay(500);
          lcd.clear();
          delay(500);
          endtime = millis();
```

```ino
      case 2:   //Increasing Temperature
        while ((endtime - starttime) <=5000){// do this loop for up to 5000mS
          lcd.print("    " + temperature + " " +  ((char)223) + "C" );
          lcd.setCursor(0, 1);
          lcd.print("     ALARM!     ");
          delay(500);
          lcd.clear();
          delay(500);
          lcd.print("INCREASING");
          lcd.setCursor(0, 1);
          lcd.print("TEMPERATURE");
          delay(500);
          lcd.clear();
          delay(500);
          endtime = millis();
        }
        break;
      case 3:   //Spike of Temperature
        while ((endtime - starttime) <=5000){// do this loop for up to 5000mS
          lcd.print("    " + temperature + " " +  ((char)223) + "C" );
          lcd.setCursor(0, 1);
          lcd.print("     ALARM!     ");
          delay(500);
          lcd.clear();
          delay(500);
          lcd.print("SPIKE OF");
          lcd.setCursor(0, 1);
          lcd.print("TEMPERATURE");
          delay(500);
          lcd.clear();
          delay(500);
          endtime = millis();
        }
        break;
      case 0:   //Normal reading with no alarm
        while ((endtime - starttime) <=2000){// do this loop for up to 2000mS
          lcd.print("    " + temperature + " " +  ((char)223) + "C" );
          lcd.setCursor(0, 1);
          lcd.print("      !OK!    ");
          endtime = millis();
        }
        break;
    }
  }
```

Figure 9: The ARDUINO-LCD code.

In case of alarm, the display will flicker and show the corresponding alarm message and the received temperature, while in case of normal reading, the display will only be showing the temperature.

5.5 Backend logic in Node-RED

The server application build on the Node-RED framework is very simple. The first node is an incoming MQTT node that, subscribed to the requested topic, will receive all the publish performed on the latter. The *Receive Readings* is a function node that will take the received temperature and will store it in two different manners: first, it will store the value by pushing it in an array (*storedReadings*) and lastly, it will store it in a dedicated variable, *lastReading*. Both of them have flow context visibility and will be useful for the operation performed on the next node of the flow.

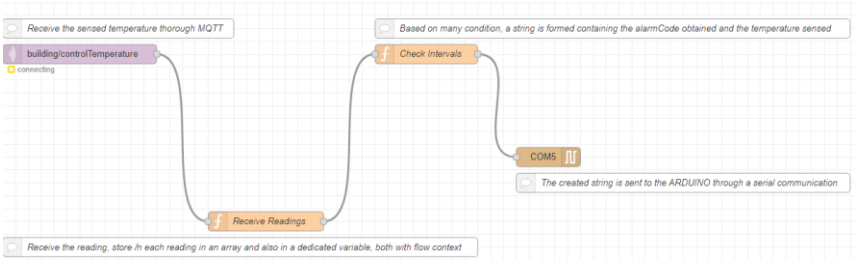


Figure 10: The main flow of the Node-RED server application.

The node is called *Check Intervals* and, based on the data received, will create various alarm code:

- If the value received is abnormally high -> **Code 1**
- If checking the last five readings, there is a sudden spike of temperature -> **Code 2**
- If checking the last five readings, the temperature keeps increasing between each reading -> **Code 3**
- If none of these scenarios happen -> **Code 0** (normal reading)

At the end, the string containing the alarmCode and the temperature is created and is sent to the Arduino through the last node, that enables serial communication through the specified COM port.

```
if (typeof flow.get('storedReadings')=== "undefined"){
    var storedReadings=[];
}else{
    storedReadings=flow.get("storedReadings");

}
var lastReading = msg.payload;
storedReadings.push(lastReading);
flow.set("storedReadings", storedReadings);
flow.set("lastReading", lastReading);

msg.payload=storedReadings;
return msg;
```

```
var readings=msg.payload;
var lastReading=flow.get("lastReading");
var count=0;
if(lastReading>40){ //if a very hight temperature is sensed -> 1
    msg.payload = "1" + "-" + lastReading.toString() + "-";
    readings.length=0;
    flow.set("storedReadings", readings);
}else{
    if (readings.length == 5) { //check the last sequential readings
        for (var i = 1; i <= readings.length; i++) {
            if (readings[i] - readings[i - 1] >= 10) { //if there is a s
                msg.payload = "2" + "-" + lastReading.toString() + "-";
                break;
            }else if (readings[i] - readings[i - 1] >= 2) { // count if
                count++;
            } else{
                msg.payload = "0" + "-" + lastReading.toString() + "-";
            }
        }
        if (count == 4){ // if the temperature has increased for all the
            msg.payload = "3" + "-" + lastReading.toString() + "-";
        }
        readings.length=0;
        flow.set("storedReadings", readings);
    }else{
        msg.payload = "0" + "-" + lastReading.toString() + "-";
    }
}
return msg;
```

Figure 11: The Receive Readings (left) and Check Intervals (right) nodes code.

6. Implementation of the attack

The attack is performed through the Kali Linux pre-built VM and consist of the following steps.

6.1 ARP Poisoning

The ARP Poisoning on the connection between the ESP32 and the Node-RED server application is performed through Ettercap. In this case, it was used the GUI version, instead of the CLI version, to which admin privileges must be granted to allow the execution of the tool. Initially, the network interface must be selected (in case of a VM is *eth0*) and then a unified sniffing operation is performed at startup. The next step is to scan the network for host, through the corresponding button and, after this, through another button is possible to visualize every *host* connected to the network, with its IP and MAC address. At this point, the hacker have to select the device which wants to attack (in this case the ESP32) as *Target 1* and, sequentially, the host corresponding to the router as *Target 2*. After doing so, by clicking on *"ARP poisoning…"* on the *MITM menu* (and deselecting every optional parameter), it's possible to start the attack. By doing so, ARP messages are sent over the network in order to link the MAC address of the attacker with the IP address of the device that is targeted and this way the attacker can intercept, edit and delete every message sent to the legitimate MAC address.

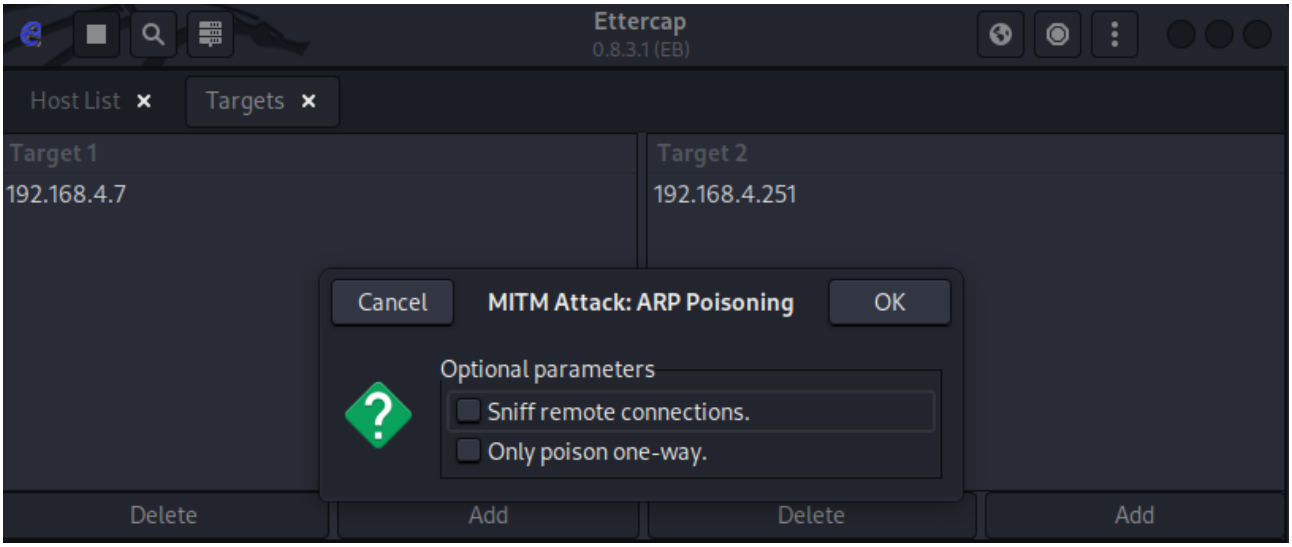


Figure 12: Selection of the target and start of the ARP poisoning on Ettercap

6.2 Packets capture

Once the ARP Poisoning is in progress, it's possible to utilize the Wireshark application to analyze the packet traffic present on the network adapter. Applying a filter on the protocol used, it's possible to capture only the packets that are transmitted utilizing the MQTT protocol. Now the attacker can select for example a publish message packet and see its full details, not only parameters like the QoS level, but also the topic in which the message is published and the content of the message itself. By doing so, the attacker can understand better how the entire system works and what kind of message he has to produce in order to trigger an evacuation alarm. Moreover, the attacker can also visualize information regarding the TCP and IP level of the packet, and he can discover the source IP address and port of the packet, which will be useful in the next step of the attack.

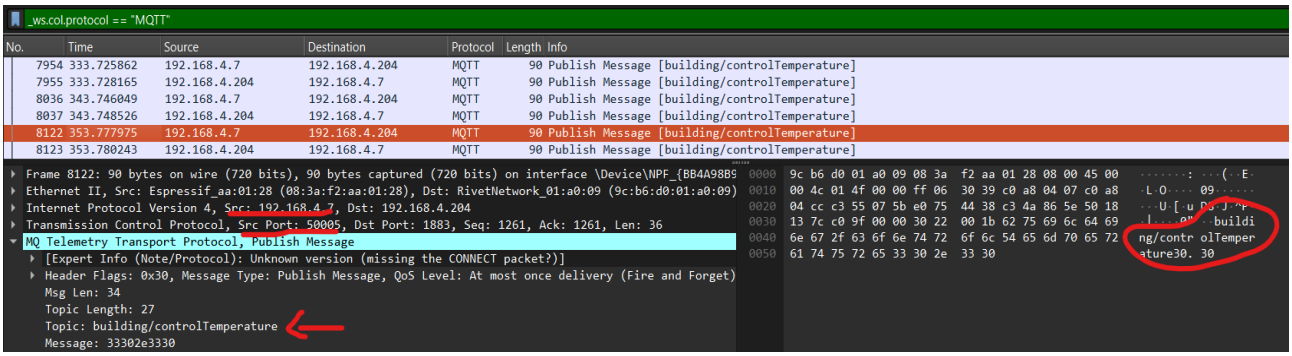


Figure 13: Network analysis through Wireshark

6.3 DoS attack

Now the attacker must be able to prevent the ESP32 device to send messages to the Node-RED application, so that he can instead send a fake message. To do so, the *hping3* tool, already present in the Kali Linux distribution, is utilized. From the terminal, by granting Admin privileges, it's possible with a single command to attack the targeted device. The command performs the attack to the specified IP address and port, which are the ones discovered in the precious step, by flooding the target with uncountable packets from various random sources, so that the network is saturated and, consequentially, any kind of communication through that port for that specific IP address is blocked. To reach more quickly the saturation, it's also possible to launch the attack with the same command from more than one terminal.

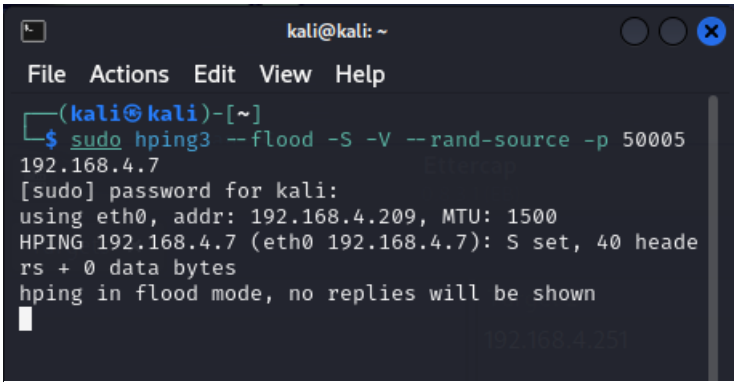
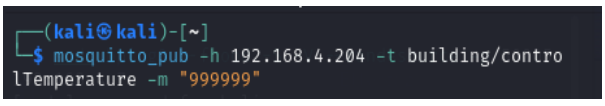


Figure 14: DoS attack through hping3

6.4 Sending the fake message

Once that the attacker is sure that the connection between the device and the server is crashed (it's possible to see through Wireshark that no more MQTT packets are sent), the attacker can send, through terminal or using a dedicated bash script, a fake MQTT message utilizing the topic discovered during the ARP Poisoning and containing an abnormal temperature value, so that an evacuation alarm will be triggered.



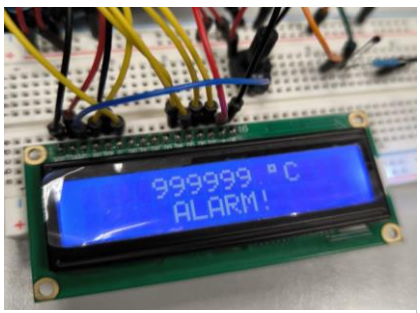


Figure 15: Sending of the fake message with consequent trigger of the evacuation alarm.

7 Possible security solutions

The most immediate way to make the system more secure is to implement the authentication in MQTT, so that username-password pair must be required in order to publish and subscribe to the topics. This way, even sniffing the packet and finding the topic to which publish, the broker will prevent the operation because it's from an unauthorized user.

```
┌──(kali㉿kali)-[~]
└─$ mosquitto_pub -h 192.168.4.204 -t building/controlTemp
erature -m "999999"
Connection error: Connection Refused: not authorised.
Error: The connection was refused.
```

Figure 16: Blocked connection in case of unauthorized user

It's also possible to attempt a brute force attack to try to discover the required MQTT credential. To do so, it's possible to utilize another tool pre-installed in Kali Linux, which is *Metasploit.* The tool, however, requires a dictionary from which extract all possible combination. To create the dictionary, another pre-installed tool can be used, which is called *crunch.* The tool creates a dictionary based on the command from which is launched: in the first attempt the aim was to create a dictionary with the following command:

crunch 7 7 aeioupswrdabc123 -o dictionary.txt

This command creates word of 7 letters by combining the all the vocals, the first three letters of the alphabet, the first three numbers and the letters which compose the word "password", which is common knowledge that is one of the most used passwords. The result was a dictionary.txt file of more than 1 GB but, during the bruteforce attack, the repeated try caused the entire pc to freeze and the only solution was to restart the computer. So, for project needs, a new credential was created (username->abc and password->abc) and a new dictionary of 6MB was generated with the following command:

crunch 4 4 abc123 -o shortDictionary.txt

After this, the attack through Metasploit is launched, thanks to the built-in utilities of the tool that allow to attack directly a MQTT connection. As it's possible to see, it's only needed to provide the dictionary file, the target host (that is known thanks to the Wireshark analysis during the ARP poisoning) and, as a result, an uncountable number of CONNACK packets are sent to the broker with all the possible combination which are in the dictionary file as credentials: after almost an hour, the tool found the right username-password combination.

```
msf6 auxiliary(scanner/mqtt/connect) > set RHOST 192.168.4.204
RHOST ⇒ 192.168.4.204
msf6 auxiliary(scanner/mqtt/connect) > set VERBOSE false
VERBOSE ⇒ false
msf6 auxiliary(scanner/mqtt/connect) > set PASS_FILE shortDictionary.txt
PASS_FILE ⇒ shortDictionary.txt
msf6 auxiliary(scanner/mqtt/connect) > set STOP_ON_SUCCESS true
STOP_ON_SUCCESS ⇒ true
msf6 auxiliary(scanner/mqtt/connect) > set PASS_FILE shortDictionary.txt
PASS_FILE ⇒ shortDictionary.txt
msf6 auxiliary(scanner/mqtt/connect) > exploit

[+] 192.168.4.204:1883    - MQTT Login Successful: abc/abc
[*] 192.168.4.204:1883    - Scanned 1 of 1 hosts (100% complete)
[*] Auxiliary module execution completed
```

Figure 17: Bruteforce attack through Metasploit.

8 Conclusions

The aim of the project was to realize a Man-In-The-Middle attack to a very simple system utilizing the MQTT protocol. As seen in the report, it's very easy for a hacker to attack such a system, regardless of whether the broker requires authentication to publish or subscribe to topics.
Anyway, the brute force attack is neither the optimal solution for the attacker, due to the enormous resources that are needed to execute it. During the project simulation, in fact, even if the resource were limited by having the VM acting as the attacker, it goes without saying that, for trying to find long and complicated passwords, a lot of power as well as a lot of time are needed.
Nonetheless, the credentials can still be discovered, so the best way to protect such a connection, will be to use a TLS encryption of the data: by setting the same certificate on both the device and the broker, the user of the system can be sure that, even if an attacker is able to sniff the packets on the network, it's almost impossible to decrypt them. For this reason, no information can be obtained from the sniffed packets and so the attacker would be unable to replicate the behavior of the system and thus to conduct the MITM attack.