

Development Team Project: Secure System Architecture

Group 2: Raquel Martinez Diez, Marzio Hruschka, Sebastian Lewle, Kalina Mohonee

Table of Contents

- Project Description 2
 - Vulnerabilities and Mitigations 4
 - Security Requirements 4
 - Implementation Challenges 4
 - Conclusion 5
- APPENDIX A: Vulnerability Risk Assessment 6
- APPENDIX B: System Structure..... 7
 - Folder Structure..... 7
- APPENDIX C: Execution Instructions via Docker..... 8
- APPENDIX D: Project Testing 9
 - Quality Assurance..... 9
 - Latency Test..... 11
- Reference List..... 12

Project Description

The goal of this project is to implement a system prototype for a Smart Home system integrating the security requirements identified on the Attack-Defence Tree analysis previously performed.

The prototype is composed of one client device that manages several thermostat controller nodes. The MQTT (Message Queuing Telemetry Transport) message broker has been used to manage communication between nodes. The MQTT has become the standard protocol for the Internet of Things (IoT) due to its simple and lightweight clients that require minimal resources (Yuan, 2021). MQTT aims to be a protocol for resource-constrained devices and low-bandwidth, high-latency, unreliable networks (MQTT, 2021). Furthermore, MQTT includes security features such as data-in-transit encryption using TLS and strong client authentication that covers most of the security requirements found in our analysis.

The prototype has been developed based on the following SysML diagrams:

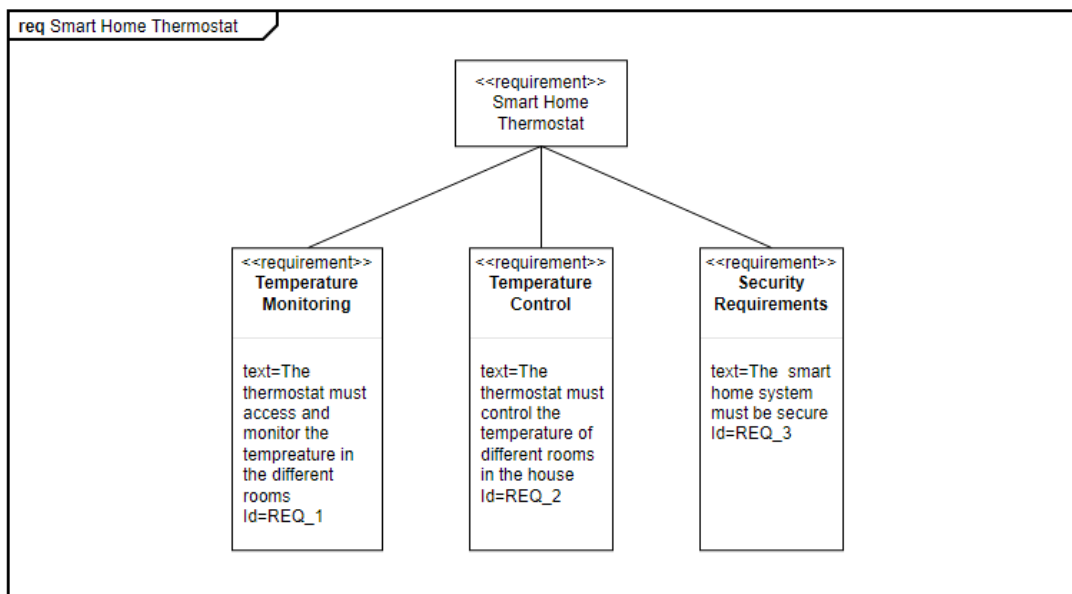


Figure1. SysML Requirements diagram for Smart Home Thermostat

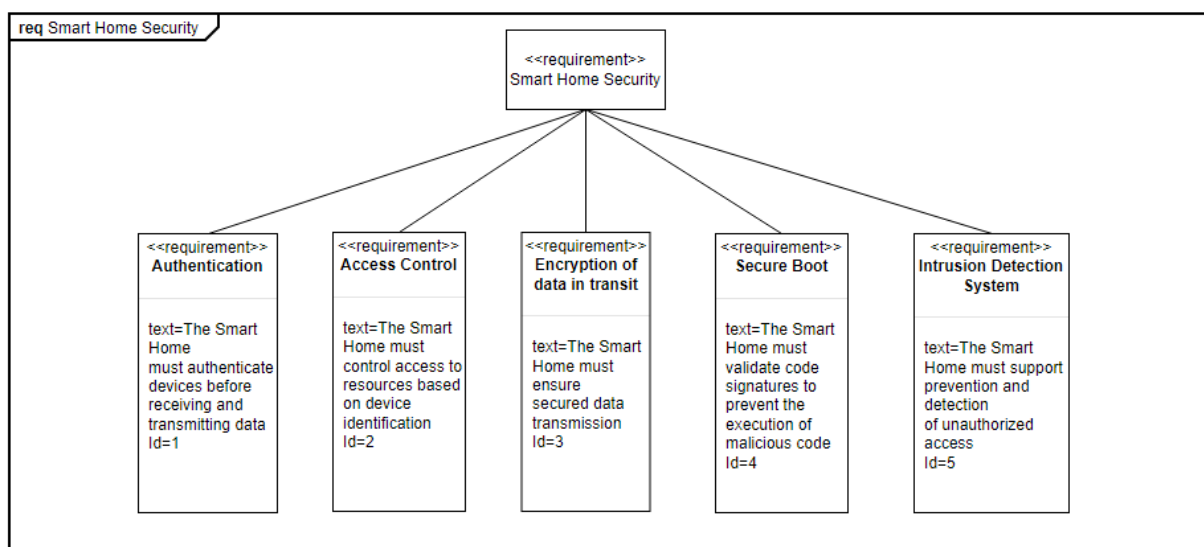
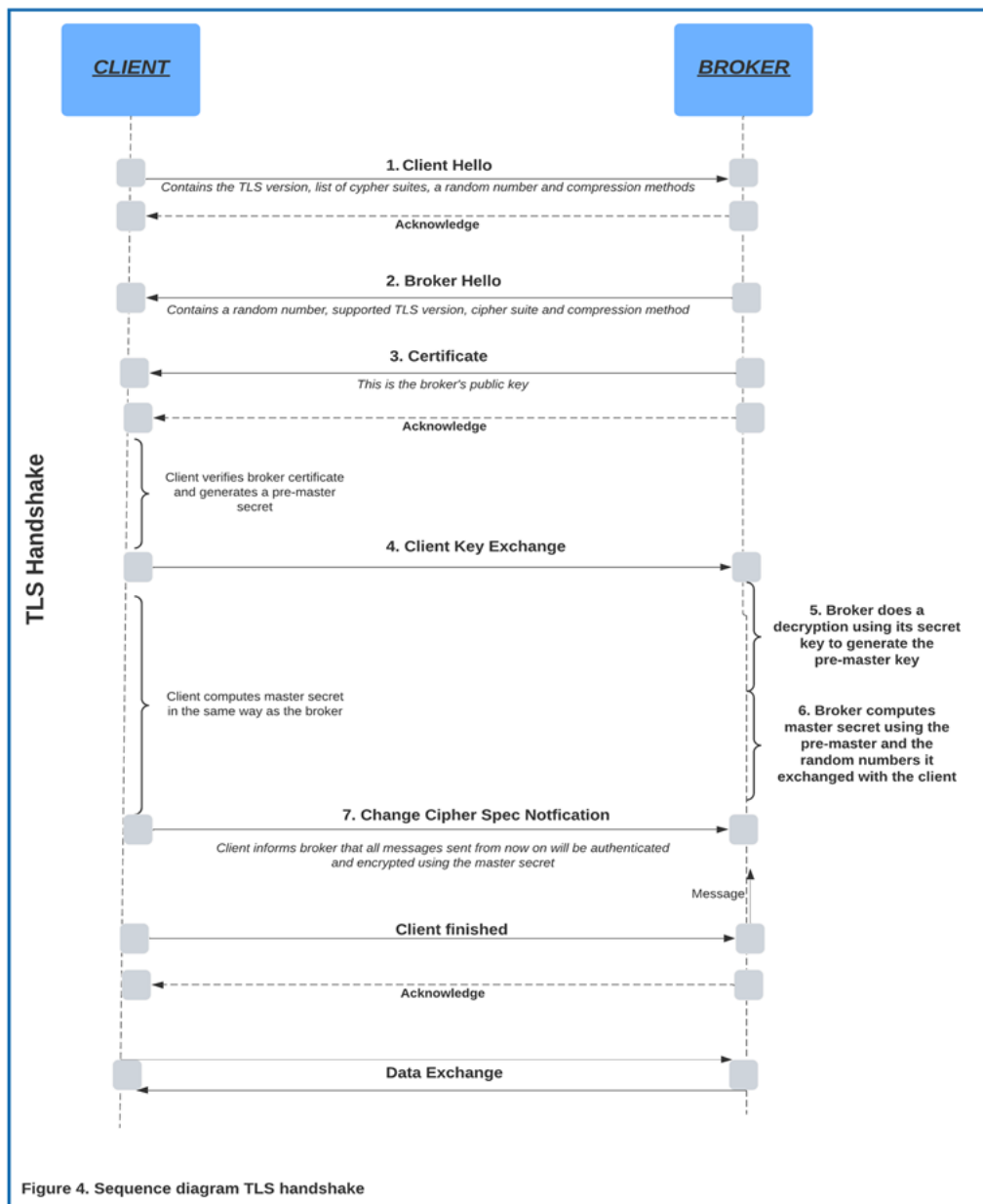
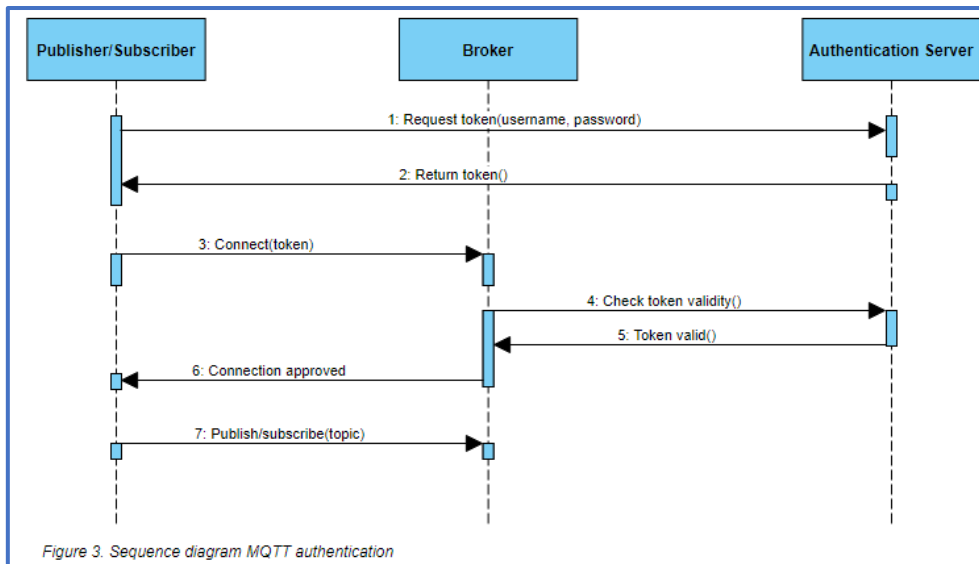


Figure 2. SysML diagram for Security Requirements



Vulnerabilities and Mitigations

The vulnerability analysis presented in the Design Document has been extended to include a quantitative assessment based on likelihood and impact scales. This allows us to create a ranking and prioritize the implementation of the mitigation measures (APPENDIX A: Vulnerability Risk Assessment).

In addition to our analysis, the OWASP top 10 vulnerabilities of IoT (OWASP, 2018) were considered when identifying security controls.

Security Requirements

These are the Security Controls that have been implemented in the prototype:

- The encryption of data in transit has been implemented by enabling TLS certificates on the MQTT Broker side.
- Authentication of devices before receiving and transmitting data done with username and password.
- To implement the secure boot feature or verification of code signatures, an RSA public/private key pair has been created and the private key has been used to sign the files. The signatures for the device and controller python files are stored in the respective config folders. Upon execution of the code, the source code is checked against its signature to ensure that the file has not been tampered with.
- All connections with the MQTT broker have been audited through a topic and are stored locally on the Broker. The purpose of this audit log is to serve as input data for a monitoring and analysis tool or Intrusion Detection System (IDS).
- Ports have been whitelisted at the Broker level to reduce the IP range that is allowed to connect to the Broker minimizing the probability of a Denial of Service (DoS) attack.
- The principle of least privilege has been configured in the Broker.
- Protection of user credentials has been implemented by using symmetric encryption keys to generate and validate credentials.

Implementation Challenges

Docker containers have been used in the prototype to configure light virtual machine containers to simulate a distributed system (Docker, N.D.). Common distributed system challenges include latency and message loss, our prototype also has IoT limitations like bandwidth and processing power (Gerber & Romeo, 2020). The table below demonstrates what measures MQTT has taken to lessen these challenges:

| Challenges | Mitigations |
|-------------------|---|
| Latency | MQTT protocol's performance evaluations confirm good rates in response time across different security levels (MQTT, MQTTS) (Liu & Al-Masri, 2021; Wang, 2018). |
| Reliability | MQTT ensures reliability by supporting session persistence (client establishes new connection after loss). |
| Lost Messages | MQTT allows configuring Quality of Service (QoS) levels for the messages sent to the Broker. According to our system structure and needs, QoS has been configured to use QoS1. QoS2 has been disregarded due to network overhead (The HiveMQ Team, 2015). |
| Power Consumption | MQTT messages are small to optimize power consumption and network bandwidth. The message header size is 2 bytes, and the payload is limited to 256 megabytes (Bernstein et al, 2021; Tracy, 2016). |

Conclusion

Due to the time constrain of the project, during the prototype development we have concentrated our efforts on the implementation of the security requirements and therefore, some of the functional requirements like the temperature control have been omitted. Security features for future work include access control mechanism, monitoring and analysis tool and device security updates management.

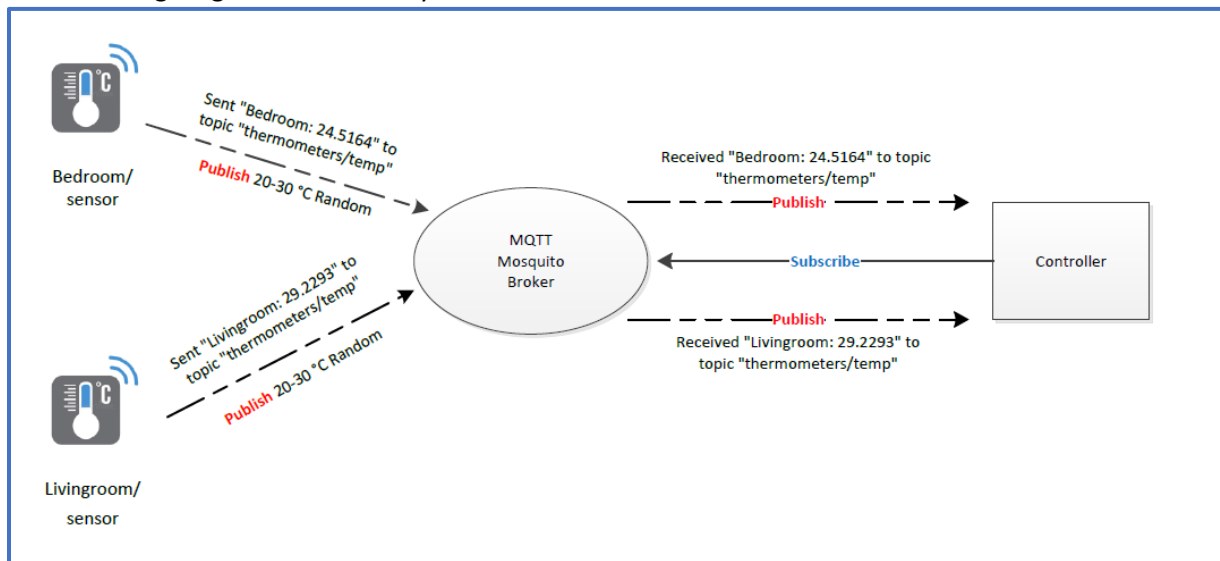
APPENDIX A: Vulnerability Risk Assessment

The main goal of this assessment is to assign a risk score to each vulnerability identified in the AD Tree analysis based on the technical capability needed to exploit the vulnerability (Likelihood) and the damage caused by the exploit (Impact) (Sutton, 2014). The following table includes a list of vulnerabilities found and the risk evaluation along with the mitigations identified for each vulnerability:

| Vulnerability | Likelih. Score | Impact Score | Risk Score | Mitigation Measures |
|---|----------------|--------------|------------|--|
| Man-in-the-Middle attack | 3 | 3 | 6 | <ul style="list-style-type: none"> Encryption data in transit using cryptographic algorithms with symmetric or asymmetric keys |
| Breach of Device or Infrastructure | 2 | 3 | 5 | <ul style="list-style-type: none"> Authentication of devices before receiving and transmitting data Secure Boot code signatures |
| Distributed or permanent denial of service (DDoS or PDoS) | 2 | 3 | 5 | <ul style="list-style-type: none"> Data monitoring and analysis tools can be used to detect possible security violations or threats. Whitelist ports at Broker level |
| Weak or standard passwords | 3 | 1 | 4 | <ul style="list-style-type: none"> Strong password policy and user credentials encryption |
| Elevation of privilege | 2 | 2 | 4 | <ul style="list-style-type: none"> Implement the Least Privilege principle to prevent elevation of privilege |
| Comprised Encryption Keys | 1 | 1 | 2 | <ul style="list-style-type: none"> Security management of IoT by rapid over the air (OTA) device key(s) replacement |
| Data Tampering | 1 | 1 | 2 | <ul style="list-style-type: none"> Access control mechanism |

| | | Impact/Consequence | | |
|------------|------------------------|----------------------------|----------------------------|----------------------------|
| | | Slightly harmful (1) | Harmful (2) | Extremely Harmful (3) |
| Likelihood | Highly unlikely (1) | Low risk (Score 2) | Low risk (Score 3) | Moderate risk (Score 4) |
| | Unlikely (2) | Low risk (Score 3) | Moderate risk (Score 4) | High Risk (Score 5) |
| | Likely (3) | Moderate risk (Score 4) | High Risk (Score 5) | High Risk (Score 6) |

The following diagram shows the system structure:



The MQTT broker has been installed on an AWS EC2 instance and the domain “mosquitto.ssa-project.xyz” has been pointed to the instance's public IP address. Using the certificate authority Let's Encrypt TLS, certificates were generated for the domain and are used to encrypt the data sent via MQTT. Effectively this is done leveraging MQTTS (Message Queuing Telemetry Transport Secured), which is the TLS secured version of the MQTT protocol. There are two ports open:

- **1883** used for MQTT
- **1884** used for MQTTS

For the proof-of-concept implementation, the system fully utilises the MQTTS protocol.

As an additional layer of security, user authentication has been enabled at the Broker level. Whenever a device wants to connect to the Broker it has to supply a valid username and password combination.

The prototype has been developed in Python and the files are executed locally. Docker containers have been configured to run the client device and the IoT controller (Python files) on separate instances to simulate distributed systems.

Folder Structure

The project includes the four main folders:

Controller: Includes all files necessary to run the controller (subscriber) of the IoT system. Prints back device temperatures and Broker logs to the user.

Device: Includes all files necessary to run the thermometer device (publisher) of the IoT system. Publishes test data for a given device to display back to the controller.

Latency-test: Includes a test script to measure the latency between the publishing of the messages and the reception at the subscriber.

Setup: Setup includes files to understand the initial setup of the project. In an actual deployment, this folder would not exist. In total, the folder includes four scripts:

- `credential_encryption.py`: the python script used to encrypt the clear credentials initially and outputs the `credentials.bin` file as seen in the 'config' folder.

- [fernet_key_gen.py](#): the python script used to generate a new fernet master key and outputs the key.bin file as seen in the 'config' folder.
- [signature_key_gen.py](#): Python script to generate a new RSA public and private key pair to sign and verify files.
- [file_signing.py](#): Python script used to demonstrate the generation of the signature of a python file using the private key.
- [broker_credentials_clear.txt](#): displays the Broker credentials prior to being encrypted (format: "host:port:user:password").

APPENDIX C: Execution Instructions via Docker

To make the execution as seamless as possible, we created Dockerfiles for both the Client Device and the Controller. This allows for building Docker Images for both and launching one or more of each as separate Container Instances.

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux, Mac OS and Windows.

We have created a DockerHub repository to store the container images and from where they can be pulled. The repository can be found here: <https://hub.docker.com/r/marzioh/ssa-project/tags>.

To pull both the controller and device images, please use the following commands:

```
$ docker pull marzioh/ssa-project:device
```

```
$ docker pull marzioh/ssa-project:controller
```

Documentation: <https://docs.docker.com/engine/reference/commandline/pull/>

Once the images are present in your Docker, you will be able to run a new container using the images. To do so, please ensure that you run the container in interactive mode, to be able to interact with the python user inputs:

```
$ docker run -it marzioh/ssa-project:device
```

```
$ docker run -it marzioh/ssa-project:controller
```

Documentation: <https://docs.docker.com/engine/reference/commandline/run/>

By executing the “run” command multiple times on the same image, you can spin up multiple containers for the same category. This makes sense for when you want to test multiple devices running on the same network and all communicating with a single controller.

APPENDIX D: Project Testing

Quality Assurance

Throughout the project, three levels of testing were utilised to ensure the quality and functional integrity of the system:

- Unit Testing
- Continuous Review via Linters
- Functional Test Plan (End-to-End)

A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. In most programming languages, that is a function, a subroutine, a method or property. In the Smart Home System, this would mean the outputs of the various functions are tested to confirm that they fulfil expectations in terms of typing and values. Ideally, for the complete system implementation, a framework such as "unittest" or "Robot" could be used to create automated test cases that verify the outputs on-demand. For instance, whenever a major code update is pushed, the unit tests could then be run before merging the code to confirm that previous functions are not affected by the update.

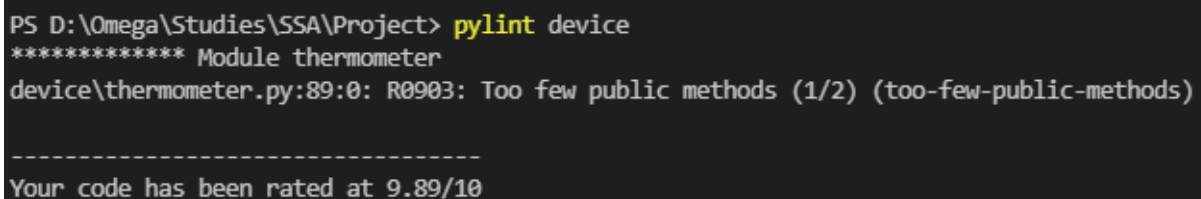
In the case of the proof-of-concept implementation, we have instead opted to go with print-based unit tests due to the tight timeline. While developing and adjusting code, print statements were used to verify the outputs of functions and ensure that the individual units of the System are working as expected.

Continuous Source Code review is done to ensure the structure, style, complexity, syntax, and security aspect of the source code as a whole is guaranteed. This can be done with the help of static code analysis tools called Linters. For this project, we decided to go with two different Linters to cover most bases:

Pylint for checking the syntax, complexity, styling and referencing of the code basis.

Bandit for checking security due to its capability of flagging vulnerabilities and their severeness.

By following the Pylint feedback, we raised our initial code score from around 6 to 9+ / 10 points for both the client and device modules. The only remaining issue was unused arguments which were set by the `on_connect()` function from Paho MQTT and our Thermometer device class being very slim (only PoC) and thus having too few public methods to call.



```
PS D:\Omega\Studies\SSA\Project> pylint device
***** Module thermometer
device\thermometer.py:89:0: R0903: Too few public methods (1/2) (too-few-public-methods)

-----
Your code has been rated at 9.89/10
```

Figure 5: Pylint Final Score for Device Source Code

```

PS D:\Omega\Studies\SSA\Project> pylint controller
***** Module log
controller\log.py:14:15: W0613: Unused argument 'client' (unused-argument)
controller\log.py:14:23: W0613: Unused argument 'userdata' (unused-argument)
***** Module temperatures
controller\temperatures.py:15:15: W0613: Unused argument 'client' (unused-argument)
controller\temperatures.py:15:23: W0613: Unused argument 'userdata' (unused-argument)

-----
Your code has been rated at 9.66/10 (previous run: 9.58/10, +0.08)

```

Figure 6: Pylint Final Score for Controller Source Code

Bandit flagged one remaining low severity security issue namely that we use a pseudo-random generator for our device temperature allocation with the `random.uniform()` function, and that this is not suitable for security/cryptographic purposes. However, we only use the uniform function to allocate random temperatures to our thermometer devices as test data and therefore using pseudo-random number generators are appropriate. Other than that, no issues were flagged.

```

PS D:\Omega\Studies\SSA\Project> bandit -r ./
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.8.10
Run started:2022-03-04 11:40:44.346202

Test results:
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.
Severity: Low Confidence: High
CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
Location: ./device\thermometer.py:134:23
More Info: https://bandit.readthedocs.io/en/1.7.4/blacklists/blacklist\_calls.html#b311-random
133     time.sleep(1)
134     thermometer.temp = uniform(20, 30) # assign random float value between 20 and 30 as dev temp
135     thermometer.publish_temp() # publish the temperature to the MQTT topic

-----

Code scanned:
    Total lines of code: 343
    Total lines skipped (#nosec): 0

Run metrics:
    Total issues (by severity):
        Undefined: 0
        Low: 1
        Medium: 0
        High: 0
    Total issues (by confidence):
        Undefined: 0
        Low: 0
        Medium: 0
        High: 1
Files skipped (0):

```

Figure 7: Bandit's Final Run

The last type of testing was executed before submitting the final project. Functional Testing ensures that all end-to-end use cases and user scenarios that will be done via the System are included and

working. First, we established a Functional Test Plan containing a full checklist of test cases for the individual roles and the system in general. We then proceeded with manually testing each of the test scenarios to ensure that the system as a whole is working as per the use-case specifications. The complete Functional Test Plan can be found below. Doing so gave certainty that all use cases the system set out to cater towards were fulfilled.

The full end-to-end Functional Test Plan can be found here:

https://github.com/MarzioHr/SSA_Project/blob/main/Functional%20Test%20Plan.pdf

Latency Test

Another point of interest to us was the latency of the MQTT protocol itself, as well as the implications of using MQTTS (TLS secured) over the standard protocol. For this reason, we have created a latency measurement script that captures the high-resolution timestamp at the point of publishing a message and calculates the latency upon receiving the message at the subscriber client. This script can be found in the “latency-test” folder.

Interestingly, we have found that there wasn’t a big difference latency-wise when comparing both protocols. Sending a message via MQTTS with our set-up broker took about 0.0001 seconds longer than using the non-TLS secured version of the protocol. The results for our test runs can be found here:

```
Test Results
-----
Host: mosquitto.ssa-project.xyz
Port: 1883 | MQTT
Total Cycles: 100

Average Speed per Message: 0.10207982978435477
```

Figure 8: Latency Test for MQTT

```
Test Results
-----
Host: mosquitto.ssa-project.xyz
Port: 1884 | MQTTS
Total Cycles: 100

Average Speed per Message: 0.10216959799178923
```

Figure 9: Latency Test for MQTTS

The reason for this is that with MQTT, a client only needs to establish a connection once per session unlike protocols such as HTTP. During the handshake, more resources (especially CPU) are required with MQTTS, however, once the clients are connected, the overhead is negligible.

Reference List

Berstein, C., Brush, K. & Gillis, A. (2021) MQTT (MQ Telemetry Transport) Definition. Available from: <https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport> [Accessed 5 March 2022]

Docker (N.D.) Use containers to Build, Share and Run your applications. Available from: <https://www.docker.com/resources/what-container> [Accessed 28 February 2022]

Gerber, A. & Romeo, J. (2020) Connecting all the things in the Internet of Things. <https://developer.ibm.com/articles/iot-lp101-connectivity-network-protocols/> [Accessed 26 February 2022]

Liu, Y. & Al-Masri, E. (2021) Evaluating the Reliability of MQTT with Comparative Analysis. *IEEE 4th International Conference on Knowledge Innovation and Invention (ICKII) 2021*: 24-29. Available from: <https://ieeexplore.ieee.org/document/9574783> [Accessed 3 March 2022]

MQTT (N.D) MQTT - The Standard for IoT Messaging. Available from: <https://mqtt.org/> [Accessed 5 March 2022]

OWASP (2018) OWASP Top 10 Internet of Things. Available from: <https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf> [Accessed 28 February 2022]

Sutton, D. (2014) Information Risk Management. 2nd ed. Swindon, UK: BCS Learning & Development Limited.

The HiveMQ Team (2015) Quality of Service 0,1 & 2 - MQTT Essentials: Part 6. Available from: <https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/> [Accessed 27 February 2022]

Tracy, P (2016) MQTT protocol minimizes network bandwidth for the internet of things. Available from: <https://www.rcrwireless.com/20161129/fundamentals/mqtt-internet-of-things-tag31-tag99> [Accessed 26 February 2022]

Yuan, M (2021) Getting to know MQTT. Available from: <https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/> [Accessed 4 March 2022]

Wang, C (2018) HTTP vs. MQTT: A tale of two IoT protocols. Available from: <https://cloud.google.com/blog/products/iot-devices/http-vs-mqtt-a-tale-of-two-iot-protocols> [Accessed 26 February 2022]