Cybersecurity Report for Monitoring Battery Percentage of Robotics Machine

TNE20003 Internet and Cybersecurity for Engineering Applications

Ly Vien Minh Khoi
103844366

***Abstract*— The increase in demands for a secure way to automatically monitor the status of a machine evokes a need to implement a monitoring system that relies on using a centralised private server and a message broker protocol to send and receive data. This is done by utilizing MQTT protocol, which enables customise security elements and a publish/subscribe based data transmission where only clients with the same security elements and subscribe to the same topic can see the published data. The data being transmitted stems from the issues of monitoring a robot's battery percentage, while also implementing fail-safe mechanism to prevent the robot from stop working during important tasks. Despite the tightly regulated nature of the protocol, many cybersecurity concerns were raised and inspected, along with countermeasures in attempts to combat these concerns. One of which is the used of cryptography, where both the published and subscribed message use the same private key to encrypt and decrypt. This help prevents to a certain extent the cybersecurity aspects of the system.**

I. INTRODUCTION

In a modern society where technologies become increasingly relevant, the needs to automatically monitor the state of an equipment is more important than ever. This is highlighted in the book Automation and Human Performance [1, pp. 91–115], which although praise human's monitoring for being unexpectedly efficient; due to poor working condition, low wage occupation and lacks experience in the context of highly automated environment, the book points out that manual monitoring can be very unreliable in critical cases. On the other hand, automated monitoring system is more effective overall as suggested by [1, pp. 91–115], which is due to its speed and consistency. The idea is further supported by [2], which demonstrates the evolution of Machine Condition Monitoring, where apart from improving in sensing, diagnosing, analysing, the process has also been able to integrate itself in IoT devices or have its data put on cloud infrastructure. With such developments in the field, it is safe to say that there should be a safe, secure method for these machine monitoring devices to send its data to engineers or interested stakeholders, or to actuators that can use these data to trigger a command. This report discusses one such method, which involves the used of a message broker protocol called MQTT and a private server that restrict access from outside to act as a medium through which different devices can publish or subscribe to a particular topic. The report touches on the cybersecurity issues that evokes by using this method and attempts to resolve those issues using cryptography and tunnelling. It also explains the python code files written to emulate the behaviour of the sensor, analysing the data receiving from it and computing the conditions to trigger a particular action.

II. MONITORING SYSTEM FOR ROBOT'S BATTERY LEVEL

In automation, electronic equipment that uses battery as its main power source becomes increasingly dominant in multiple industries [3]. This, along with the prevalence of machine monitoring process give rises to the needs of monitoring the battery percentage remaining of a particular automated machine.

A. Current Issues

It is suitable to make the automated machine to be a robot that operate in an environment, where it is required to performed multiple highly sophisticated tasks and to operate for a long time while also ensuring that it finished what it has started even after its battery runs out. Hence, the current issues that need to be addressed with regards to the robot's operation is implement a fail-safe mechanism so that the robot will not suddenly cease to function during an important task. The action the robot takes also needs to emulate the chaotic nature of its working environment, where it is impossible to know for certain which tasks the robot will do next.

B. Proposed Solution

To combat the issue of the robot's behaviour during operation, a simulation is developed, in which the robot's next course of action is determined randomly from a set of fixed movement. The robot can perform one movement at time, which is sent to another simulation that emulate the robot's battery percentage. This battery simulation then decides how much battery percentage it will deduct from the robot's battery base on it received robot's movement, each of which will consume a different amount of battery. On the other hand, the needs for a fail-safe mechanism in the robot is resolved by introducing a back-up power source that automatically charges the robot when its battery drop below a certain threshold. In the case when the back-up power source also runs out, there exists a final resort plan to prevent the robot's battery from drying up, which is a command to initiating a power generator placed within the robot. All these components combined to make up a battery monitoring system where the robot's battery is monitored and managed by different power sources. The state of the robot's battery can be accessed by users who has authentication and authorization over the robot.

III. CYBERSECURITY ASPECTS

A. Cybersecurity Concerns

In the current state of the system, there are multiple concerns in the aspect of cybersecurity and data authentication. These issues are best categorised by the security triad: Confidentiality, Integrity, and Availability to

have a better judgement in developing the appropriate security policy for the system.

The first set of concerns are about confidentiality. The information on the battery level of the robot is currently vulnerable to hijacking attempts from malicious parties, who can use them to know whether the robot is low on battery and send attacks to the facilities the robot is operating in.

Data integrity is another set of concerns that needed to be addressed. In specification, having access to the data in the system means being able to change the data send to certain components, thus triggering controllers that take action based on those data (in this case is the power generator). This can lead to serious consequences with regards to property damage or waste of energy to power unnecessary equipment.

Moreover, availability is also a major issue in the security of the system. This can be seen through the fact that remote access to the data over the internet can introduces a myriad of issues, including but not limited to cybersecurity issues. Since more people can get access to the data, the issues arise beforehand becomes even more prevalence. On top of that, the communication between components as well as between components and users can be interrupted through denial-of-service attack, un-config firewall or network unreliability.

B. Proposed Solution

The action of directly sending and receiving data between the robot's components is inherently very vulnerable to cybersecurity attacks. This is the reason why a centralised server is used to eliminate some of the listed issues. By using the private network "rule28.i4t.swin.edu.au" provided by the university, as well as a message broker protocol MQTT, as the medium in which the messages are sent and received, the system can have adequate level of controlling which parties can access the data. The protocol also allows client to customize their security elements, such as username and password, client id, or topic/sub-topic system, all of which prevent any access from whom don't have the required authentication. This aided tremendously in improving the cybersecurity aspects of the system, as having a centralised server and establishing private rules prevent malicious parties from accessing and tampering with the data, while also simplifying the data transmission to a single point of access, hence increase availability.

However, solely relying on this method is not sufficient to address all the vulnerability in the system. This is because while the centralized server and the customize rule are private, hackers can still find ways to break into the system through port scanning (to gain access to the private server), or by brute force attack (to pass the customise rule). Hence, other methods of reinforcing cybersecurity are necessary to protect the data sent. One such method is implementing symmetric cryptography to encrypt the data published to the topic with a private key. This key is a unique hash key that is share among all the stakeholders. It is randomly generated by the python library cryptography.fernet, and thus cannot be obtained by any ways by outside parties. The published messages used this key to encrypt, and the subscriber can use the same key to decrypt the cipher text received. The used of this method greatly improved the confidentiality of the data sent while also ensuring data integrity by sharing the key from both encryption and decryption.

It is important to note that security is a process of constant trade-of, meaning that improving security also comes with increasing cost and the scale of the system. Finding the balance between these two aspects is crucial in implementing a system, thus a throughout and frequent risk assessment is necessary.

IV. CODE ANALYSIS

The system's components, or simulation as mentioned in the previous section, represented by different files of python code. Each of this code contains a MQTT client, which is coded using the Paho client library. A client can publish and subscribe to a topic through a particular server, in which other client connecting to the same server can also see the published data as long as they subscriber to the same topic. The analysis of these codes is significant in understanding the inner work of the system, while fully appreciating the attempts of reinforcing the system's cybersecurity aspects.

In general, the structure of these codes follows a very similar pattern, with each block of code serves different purposes. As shown in in Fig.1, the first block of code always started with importing relevant libraries, with the main ones are paho.mqtt.client for initiating the MQTT Paho client, time for program delay, and random for random numbers or choices generated. This is the common process for all subsequent codes, as the functions in these libraries are essential for the code's execution.

```
import paho.mqtt.client as mqtt
import time
from random import choice
from cryptography.fernet import Fernet
```

Fig. 1 Libraries import

Since the system implements cryptography as a method to reinforced cybersecurity, the block of code from Fig.2 and 3 are required. This is because the line in Fig.2 initialised a private key, which is a string sequence that is encoded as a byte-like object to use for both encryption and decryption; hence it is shared among all other code files.

```
encryption_key = b'HBDQbtROPt8bRBw6-mwWgMT5bE40SkXoH88QYqQKrHM='
```

Fig. 2 Private key initialized

On the other hand, the code in Fig.3 includes a function to use that key to encrypt the message for publishing; as well as another function to decrypt the received message when the client subscribed to a particular topic. These codes collaborate with one another to establish a secured, two-way communication between clients without sacrificing data integrity.

```
Comment Code
def encrypt_message(message, key):
    f = Fernet(key)
    encrypted_message = f.encrypt(message.encode('utf-8'))
    return encrypted_message

Comment Code
def decrypt_message(encrypted_message):
    f = Fernet(encryption_key)
    decrypted_message = f.decrypt(encrypted_message).decode('utf-8')
    return decrypted_message
```

Fig. 3 Encrypt and decrypt functions

Usually, a client will have one in two distinct purposes: either published a message to a topic or subscribe to a topic and read the message sent in that topic. Depends on which purpose a client has, the programmer can choose to include the functions list in Fig.3 accordingly. A client that only publish a message don't need to decrypt any data, while it is unnecessary for a client whose only purpose is to receive message to encrypt output data. In our codes, however, the clients are usually shoulder both responsibilities, making it necessary to include both functions.

The code depicted in Fig.4 is used to declare the ip address of the centralised server, set up the client along with its name, set up username and password while telling them to connect to the server on a particular port. In this project, the "rule28.i4t.swin.edu.au" is used, which is a server lies on the private side of the Swinburne network. The username and password set are the student ID, whereas the name for the client can be whatever device that the code tries to emulate.

```
mqttBroker = "rule28.i4t.swin.edu.au"
client = mqtt.Client("Smartphone")
client.username_pw_set("103844366", "103844366")
client.connect(mqttBroker, 1883)
```

Fig. 4 MQTT broker server, username and password

As the client is set up in Fig.4, there should be a block of code that handles receiving messages from the subscribed topic, which is demonstrated in Fig.5. The function have three parameters: client, userdata, and message, all of which have their value pass in by the initiated client in Fig.6. The data is decoded, decrypted and print to the terminal for ease of debugging.

```
Comment Code
def on_message(client, userdata, message):
    payload = message.payload.decode('utf-8')
    decrypted_payload = decrypt_message(payload)
    print("Received message:", decrypted_payload)
```

Figure 5 On_message call back function

Subsequently, the project required all MQTT client to subscribe to a public topic and print all its message out. It is important now to explain the hierarchical structure of the topics used. On the top level, where all the message published to the subsequent topics are posted is the topic "103844366", which is the student ID. The followed sub-topic can be either "robot" or "power_bank", each representing an equipment the user is trying to monitor. The state of these equipment can be accessed via the lowest-level sub-topics, which include "battery" or "action", reflecting the actual status of the equipment. A client can subscribe to the highest-level topic by using a wildcard syntax "#" as shown in Fig.6, where "103844366/#" means the client can see messages from all the sub-topic under the "103844366" topic umbrella. Since "103844366" in this case is the topic at the top od the hierarchy, it is equivalent to subscribing to a public topic where any message sent on that server can be acquired.

```
client.loop_start()
client.subscribe("103844366/#")
client.on_message = on_message
```

Fig. 6 Topic subscription and starting client loop

The loop_start() function indicate that the client can loop-reading the incoming message while the on_message function pass the execution of the function in Fig.5 to the client's received message. These functions and code logics are implemented identically across all clients, creating a foundation for other unique logics to be added. Depends on the purpose each client shoulders in the system, the added codes will be discussed in detail in the following sub section.

A. Robot Action

The first code file that should be discussed is the Robot Action file, whose purpose was to randomly publish the robot's next course of action. The inner work behind this logic is shown in Fig. 7, where the actions are declared in a string array called "actions". The choice function is used to randomly choose the elements in the array, which is then being encrypted and published to the topic "103844366/robot/action".

```
actions = ["Run", "Walk", "Raise Arm"]

while True:
    action = choice(actions)
    encrypted_action = encrypt_message(action, encryption_key)
    topic = "103844366/robot/action"
    client.publish(topic, encrypted_action)
    time.sleep(1)
```

Fig. 7 Robot actions are randomly generated and published to the robot/action sub-topic

These logics are put in a never-ending loop, indicating by the "While True:" line. This means the client will keeps on sending the message with one second's interval until the user manually stop the process via the initiation of the shortcut keys "Ctrl+c" to invoke key interrupt exception. It is worth noticing that this method of implementation is quite scalable and modular, as the specific type of actions the robot can take, and the number of these actions can be changed and updated without the needs to significantly alter the code structure.

B. Robot Battery

The code file that emulates the robot's battery as well as the power back up mechanism is our next field of interest. The manipulation of the robot's battery depends on the messages from the topic this client is subscribed to, which is the previously discussed Robot Action topic. As seen in Fig.

8, the message received from the topic needed to be decrypted and stored in a global variable, which can be called upon whenever needed.

```
Comment Code
def on_message_public(client, userdata, message):
    global data_public
    encrypted_data = message.payload
    data_public = decrypt_message(encrypted_data, encryption_key)
    print("Received message:", data_public)

Comment Code
def on_message_action(client, userdata, message):
    global data
    encrypted_data = message.payload
    data = decrypt_message(encrypted_data, encryption_key)
```

Fig. 8 On_message_action function in Robot Battery client

Since this client needs to subscribe to two topics: the public topic as a general requirement and the robot action topic for battery manipulation; the usual on_message function is duplicated to handle each topic separately. These functions (shown in Fig.8) are coupled to the client's distinctive topics via the message_call_back_add function in Fig.9 with pass in topics and corresponding on_message functions.

```
mqttBroker = "rule28.i4t.swin.edu.au"
client = mqtt.Client("battery_percentage")
client.username_pw_set("103844366", "103844366")
client.message_callback_add("103844366/#", on_message_public)
client.message_callback_add("103844366/robot/action", on_message_action)
client.connect(mqttBroker, 1883)
```

Fig. 9 Robot Battery client handles two unique topics

It is appropriate to now discussed the mechanics in which the client handle the logic behind the robot's battery manipulation. Initially, the battery state of the two-equipment available is initialized as in Fig. 10. This will be changed based on the actions received from the subscribed topic.

```
battery_power_bank = 100
battery_decrease_power_bank = 0
battery_robot = 100
battery_decrease_robot = 0
```

Fig. 10 Initial state of the available equipment

The message from the subscribed Robot Action topic only contains a single string that represent the robot actions; thus, it can be directly compared with the expected string in an if-else statement shown in Fig.11. The chosen number of battery percentage decreased are arbitrary and can be changed accordingly. The max() functions with the lower bound value of zero is used to determine the battery level of the robot after it has been deducted with the changed battery percentage decreased. This was to ensure the resulted battery level will always be positive, hence increase the reliability of the simulation.

The next block of code in Fig. 11 indicates the kind of message the client wants to publish to the "103844366/robot/battery" topic. During the discharging phase, this message just tells the user that is the percentage of battery remaining in the robot. As soon as the battery level reach 100 or 0, however, the message published changed to "Robot is fully charged!" and "No battery left!" respectively, alerting the user in the case of emergency.

```
while True:
    if (data == "Run"):
        battery_decrease_robot = 10
    elif (data == "Walk"):
        battery_decrease_robot = 6
    elif (data == "Raise Arm"):
        battery_decrease_robot = 2
    else:
        battery_decrease_robot = 0

    battery_robot = max(0, battery_robot - battery_decrease_robot + battery_decrease_power_bank)

    if (battery_robot >= 100):
        battery_decrease_power_bank = 0
        message_robot = "Robot is fully charged!"
    elif (battery_robot <= 0):
        battery_decrease_robot = 0
        message_robot = "No battery left!"
    else:
        message_robot = f"{battery_robot} % remaining for robot"

    encrypted_message_robot = encrypt_message(message_robot, encryption_key)
    client.publish(topic_robot, encrypted_message_robot)
    time.sleep(1)
```

Fig. 11 Robot's battery manipulation based on its next course of action

In addition to the robot's battery, the system also needs to handle the power bank's battery level based on that of the robot, as it will act as a back-up power source for the robot in the case of emergency. This logic is explained in Fig. 12, where if the battery level of the power bank is larger than zero and the battery level of the robot is less than 30, the battery percentage decrease changes to 10. This is stored in the "message_power_bank" variable and published to the "103844366/power_bank/battery". Similar to robot's battery, the message published will changed according to the power bank's energy level, ensuring that the user knows when the back up power source runs out.

```
if (battery_power_bank > 0):
    battery_power_bank -= battery_decrease_power_bank
    if (battery_robot < 30):
        battery_decrease_power_bank = 10
    message_power_bank = f"{battery_power_bank} % remaining for power bank"
else:
    battery_decrease_power_bank = 0
    message_power_bank = "No battery left!"

encrypted_message_power_bank = encrypt_message(message_power_bank, encryption_key)
client.publish(topic_power_bank, encrypted_message_power_bank)
time.sleep(1)
```

Fig. 12 Power bank's battery manipulation based on robot's battery level

C. Controller

The controller client represents the system that takes in command and initiate an action based on that command. In this case, the command is to turn on the generator whenever the power bank's battery level is below 30 percent, which is coded using the logic in Fig. 13.

```
while True:
    try:
        if (int(data.split(" ")[0]) < 30 or data == "No battery left!"):
            print("Initiating Power Generator")
        time.sleep(1)
    except ValueError:
            print("Initiating Power Generator")
            time.sleep(1)
```

Fig. 13 Controller sending commands based on the message received from the power bank's battery level

The client is firstly subscribed to the power bank battery topic, from which it received and decrypted data. This data is in the form of a sentence updating the power bank's battery state to the user, which is hard to analyse, given that the

structure of this sentence changes once the battery level reach 0. Hence, the data is split using the split function from the inherent python library, with the delineator of a blank space between the words in the sentence. Doing this, the information on the battery level can be extracted and compared to our made-up threshold of 30; while the original, unsplit data can be used to directly compared to the hard coded message of "No battery left!". When either of these conditions satisfy, the controller will print the message "Initiating Power Generator" on the terminal, indicating that the controller is taking action in respond to the alert sent by the power bank. It is also worth noting about the try-except block, which eliminate the error where the machine tries to split the "No battery left!" message and turn it into an integer to compare.

D. User

In an attempt to create a simple user interface, where one can subscribe to any topic on the same centralised server, the "user" python code is created. This is done by first creating and configuring a normal client code, except the topic this client use to subscribe to is replace with an input() function that print out the argument and store the user input as a string. This is demonstrated clearly in Fig. 14, where the input argument urges the user to type in an input go their choosing, which is pass as a topic for the client to subscribed to before printing out the received message indefinitely.

```
client.loop_start()
topic = input("Please enter the wanted topic\n")
client.subscribe(topic)
client.on_message = on_message

while True:
    time.sleep(1)
```

Fig. 14 Taking input from the user as the subscribed topic

E. Instruction on how to run the code

Usually, python code can be run directly in the Visual Studio Code IDE due to the software allows users to open terminals on there. However, the code demonstrated in this project needs to have specific libraries installed to be able to run, which can not be done in Visual Studio Code's built terminal. Hence, it is best to run the code on the command prompt terminal on a Window machine, which required some knowledge on file directory and command.

Firstly, the required libraries needed to be downloaded, which can be done via the "pip install" command. On the machine's terminal, typing "pip install paho-mqtt" and "pip install cryptography" after the download is finished to install paho-mqtt and cryptography client respectively. Subsequently, on the location at which the python code is saved, the user can right clicked on the page's blank space and choose "Open in Terminal" option from the drop down menu. This removes the needs to change the directory when opening the terminal from the start menu.

It is now just a matter of initiating the python file by the command "py" or "python", followed by the file's name with the extension. For instance, if the file name is client, then the command will be "py client.py" at the directory the file is saved in. In this project, the order in which the file is executed is very important, as some files use the data published by the others. The recommended sequence of execution is:

1. RobotAction.py
2. RobotBattery.py
3. Controller.py
4. User.py

The User.py file can be run whenever the user want, as it will always started with the argument "Please enter the wanted topic:" before subscribe to any topics. However, keep in minds that the RobotBattery.py file relies on RobotAction.py files and Controller.py files relies on RobotBattery.py files, hence subscribing to the topic on a lower hierarchy will result in errors.

It is also important to mention that while running the client file that constantly have messages print out on the terminal, the user can use the key combination "Ctrl+c" to stop the file from running. This is useful if the user wants to change something in the code and wants to compare the output before and after the change.

V. CONCLUSION

In conclusion, the implementation of a centralized monitoring system, utilizing a private server and MQTT protocol, offers a secure means to automatically monitor the status of machines. This approach capitalizes on MQTT's flexibility, allowing for customized security elements and a publish/subscribe model that restricts data visibility to clients with matching security elements and subscriptions. The application of this system, primarily focused on monitoring a robot's battery percentage and ensuring fail-safe mechanisms during crucial tasks, addresses the critical need for machine status surveillance.

While MQTT boasts a tightly regulated nature, cybersecurity concerns have been thoughtfully examined, prompting the deployment of countermeasures. Notably, cryptography, employing a shared private key for both published and subscribed messages, adds an additional layer of security to fortify the system against potential threats. This approach represents a vital attempt in addressing the evolving demands for secure and efficient machine monitoring in today's interconnected and data-sensitive environments.

REFERENCES

[1] M. Mouloua, R. Parasuraman, R. Molloy, and B. Hilburn, *Automation and Human Performance*, 1st edition. Routledge, 2018, pp. 91–115.

[2] M. Kande, A. Isaksson, R. Thottappillil, and N. Taylor, "Rotating Electrical Machine Condition Monitoring Automation—A Review," *Machines*, vol. 5, no. 4, p. 24, Oct. 2017, doi: https://doi.org/10.3390/machines5040024.

[3] G. Pistoia, *Battery Operated Devices and Systems: From Portable Electronics to Industrial Products*. Elsevier, 2008. Accessed: Oct. 25, 2023. [Online]. Available: https://books.google.com.au/books?hl=en&lr=&id=117GF5bJH3UC&oi=fnd&pg=PP1&dq=Corded+vs.+Battery-Powered+Devices&ots=k0ataaN4pE&sig=fehYbgvHhmEzFzImGbHoRSVcFqE#v=onepage&q=Corded%20vs.%20Battery-Powered%20Devices&f=fals