LLM-Assisted Dynamic Rule-Based Vulnerability Scanner for MQTT Brokers

Zakaria Ahmed

[1] Seneca Polytechnic, Toronto ON, CA
`zahmed70@mysenea.ca`

August 2025 Toronto, ON, CA

Abstract

In this thesis a dynamic and rule based MQTT vulnerability scanner is proposed. The methodology makes two important contributions to the field of vulnerability scanning for MQTT brokers. Firstly, it introduces a methodology to continuously update the scanner by introducing modules and rules to allow others to continuously update the scanner to account for new vulnerabilities which are inevitable. Secondly it proposes a fingerprinting module to detect the MQTT broker version when it has been explicitly hidden. These two features have not yet been covered or implemented to account for the increasing number of vulnerabilities within the MQTT brokers. Regarding the fingerprint module, it has been implemented incorrectly on other tools such as MQTTSA or NMAP as these tools subscribe to a topic on the broker to detect the version of the broker without considering that it can easily be blocked by the will of the administrator. The suggested methodology was implemented and tested, and results has shown DreamScan4MQTT is 2.5 x more compatible with broker versions than the leading scanner with 100% compatibility across Mosquitto broker versions. We also integrated an accurate fingerprinting prototype using a rule detection engine which is 100% more accurate than the leading MQTT scanner. These values were calculated after running 5 trials and comparing the results between the leading scanner MQTTSA and DreamScan4MQTT. Our results and evaluation also found that in our testing environment with 5 random version numbers on the same configured broker it consistently can only find 5 misconfigurations or vulnerabilities in the broker. Therefore, we prove that MQTTSA is not a reliable scanner to test the complete security of an MQTT broker as the number of vulnerabilities should increase the more outdated the broker is rather than detecting a constant number of vulnerabilities across version numbers. The proposed methodology was trained against the LLM model GPT-4 to demonstrate its ability to be learned by creating new modules after being trained. Our results showed that an LLM was able to produce a module for DreamScan4MQTT with 100% accuracy demonstrating its ability to be learned and continuously updated in the ever-growing field of cyber security.

Contents

1 Introduction

IoT devices have become increasingly popular within recent decades, and the number of devices is set to double from 15.9 billion in 2023 to an estimated 32.1 billion IoT devices in 2030 [1]. A protocol by the name of MQTT (Message Queuing Telemetry Transport) has taken the IoT world by storm. This is because of the protocol's lightweight nature, ease of scalability, low power consumption, and other benefits. Therefore, it will often be used by IoT developers. The protocol itself is not inherently vulnerable; however, security researchers have found that if not configured correctly, a variety of vulnerabilities can arise. Despite its advantages, a variety of bugs and vulnerabilities have been identified by security researchers, leading to many updates and some security tools being developed over time. The problem is that the vulnerabilities found in MQTT are often very protocol-specific or even application-specific; therefore, traditional black box scanners may take too long to scan the system or may miss these vulnerabilities. An insecure MQTT broker can pose a major risk to an organization, such as information disclosure, as many of these brokers are deployed to the cloud over unencrypted channels. Furthermore, a 2018 study by Avast showed that 32,000 MQTT brokers weren't password-protected, potentially allowing unauthorized access to sensitive data over the public internet [2]. In addition to this, a public search of the Shodan search engine for Mosquitto brokers discovered that the most popular version of Mosquitto was version 1.4.8, with over 11,000 publicly available brokers [3]. This is concerning because version 1.4.8 is very outdated with many vulnerabilities associated with it. Therefore, the need arises for an MQTT-focused vulnerability scanner that can automatically detect vulnerabilities and misconfigurations within MQTT brokers before they are put into production to protect the confidentiality, integrity, and availability of the brokers.

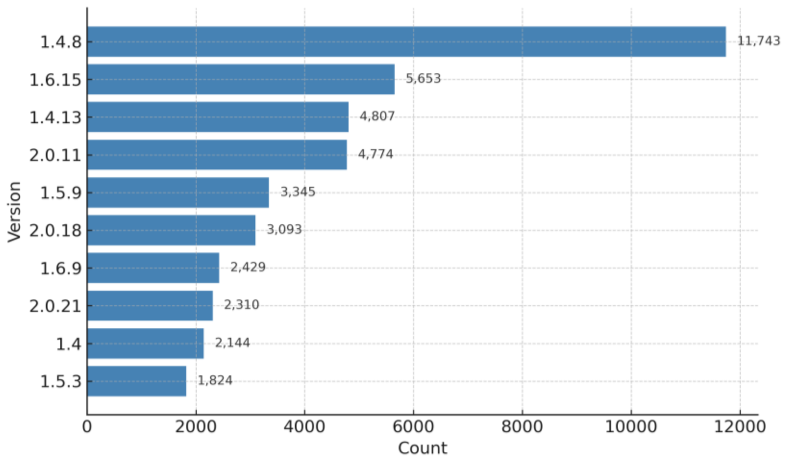


Figure 1 - Shodan Search for Mosquitto brokers 2025 [3]

We present Dreamscan4MQTT, a dynamic rule-based vulnerability scanner that uses a lightweight rule-based engine for matching data derived from network vulnerability tests developed in Python. For the first time, within an MQTT security tool, we introduce concepts from successful vulnerability scanners such as OpenVAS and Nessus to use test scripts rather than hardcoding test logic in a single script. This ensures the program is modular to allow it to be updated easily. There are two major contributions made within this paper. Firstly, we allow our vulnerability scanner to be modular, allowing for easy updates, which are needed for a successful vulnerability scanner as new vulnerabilities are inevitable. We show by creating a modular program that LLMs can even create modules after being trained. For example, our research used the GPT-4o model to generate a 100% accurate authentication check model for MQTT brokers using our methodology. Secondly, we propose an accurate fingerprinting prototype that can gather information regarding the version of Mosquitto brokers even when the '$SYS/broker/version' topic is disabled. We evaluated DreamScan4MQTT based on compatibility across versions, accuracy of modules, and number of vulnerabilities detected. Our evaluations showed that Dreamscan4MQTT detects 1.8x more vulnerabilities than the leading scanner, 2.5x more compatible across broker versions than the leading scanner, 100% compatibility across Mosquitto broker versions and a fingerprinting module that is 100% more accurate than the leading MQTT scanner.

2 Background

The Internet of Things (IOT) connects simple everyday devices like fridges, alarms, and even vending machines to the Internet, allowing us to control or collect data over the Internet without having to physically be there. However, these devices are very lightweight and often run on simple 9V batteries; therefore, they cannot handle the bandwidth of something like a computer, which the internet was designed for. Therefore, these devices created the need for a lightweight protocol designed to be lighter than the HTTP 1.1 and HTTP/2 protocols. [4] Giving birth to the MQTT protocol designed for devices and scenarios where many devices must communicate with each other or exchange data with each other in close to real time without delays. It was designed in 1999 when Andy Stanford-Clark and Arlen Nipper developed the first version of the protocol to monitor the oil pipelines of SCADA systems in a manner that is lightweight and low bandwidth usage to reduce costs associated with bandwidth [6]. It has then evolved into being used within railway systems, point-of-sale systems, slot machines, fire alarms, traffic monitoring, and other fields. It has been adopted by many because of its ability to be lightweight, near-time responsiveness, and low bandwidth, making it very favorable to developers.

2.1 MQTT Overview

The MQTT protocol runs on the application layer over the TCP protocol to facilitate its connections similarly to the HTTP protocol used to power websites. It uses a 'publish-subscribe' model to facilitate communication between machines. This model is advantageous in scenarios where many devices are interconnected because it allows a client to publish messages to only specific topics, and only those subscribed to that topic can receive published messages. Next the server known as a broker routes incoming messages to clients that are interested in receiving that specific type of message. The broker is a device between the client controlling another IoT device, and it is responsible for routing requests to their correct target. While the client that sends a message through the broker is known as a publisher and the client that awaits incoming messages from specific topics is known as a subscriber. This can allow a publisher to do things such as turn on a light bulb from his phone by subscribing to that topic and sending messages through the broker. The broker will then route the request to the IoT device. At times a broker can be on the cloud, or it can be on-premises depending on the scenario and business needs.

2.2 MQTT Protocol

Before sending commands to the MQTT broker, we must first establish a connection with the broker. The MQTT Broker is responsible for accepting connections and allowing authorized individuals to connect. MQTT is a binary-based protocol where the control elements are binary bytes and not text strings [9]. To establish a connection, the MQTT client will have to send a 'CONNECT' control packet to the MQTT server, and the server will reply with a 'Connection Acknowledge' packet to establish a connection between server and client. The Connect control packet can include the following fields to establish a connection between the server and client; we can summarize them into a single table for clarity.

Field / Flag	**Use Case**
ClientID	The ‘ClientID’ is a string used to identify the MQTT client that connects to the server. The ClientID should be unique and if it is empty the server will generate a random client depending on the packet.[4]
CleanSession	The ‘Clean session’ is used to manage and specify behavior after the MQTT client disconnects. Values are 1 and 0, indicating true and false. If the value is set to 1 then the session will last only as long as the network connection. If the value is set to 0 then the connection will be persistent, allowing the client to disconnect and still have all the same messages incoming [4].
Username	The username field is optional. Clients can specify a username for authorization to the MQTT server by setting username flag to 1.
Password	The password field is optional. Clients can specify a password for authorization to the MQTT server by setting the password flag to 1.
Protocol Level	The protocol level field indicates the protocol to use for the MQTT Protocol as there are older and newer values such as MQTT 5 and MQTT 3.1.
Keepalive	The keep alive field is a mechanism used within many protocols such as TCP. It works by defining a time interval in seconds that a client shall notify the broker that connection is indeed alive. Next it then waits for the PINGREQ message from the client and if the broker does not respond with a PINGRESP then the client ends the

	connection saving bandwidth and detecting half open connections. [12]
Will,WillQoS,WillRe-tain, WillTopic and Will Message	The 'Will' flags are special and unique to the MQTT protocol and allow it to take advantage of the last will and testament feature of MQTT. The Will value can be set to true to tell the broker to store a last will related to the session. [5]. If set to true, then it is mandatory to specify what topic will be associated with the flag. The WillQos simply will indicate what the desired quality of service (QOS) for the last will message is. The 'WillRetain' flag shows whether this message will be retained when it's published.[5] If the connection is dropped then the MQTT broker will publish the message within the Will Message field using the QOS indicated.[4]

After the MQTT broker receives the 'Connect' packet with the flags set, it will reply with a 'CONNACK' to indicate that it acknowledged the connection. If the connection was successful, then the broker will give a return code of 0 otherwise the return code can be a variety of different values each with its own meaning.

Return code	**Meaning**
0	Connection was accepted
1	Connection was refused due to requested protocol not being supported by broker.
2	Connection was refused due to invalid client id
3	Connection was refused due to MQTT service not being available
4	Connection was refused due to username or password being invalid/malformed
5	Connection was refused due to failed authorization

Using programming languages such as Python, we can spoof packets to the broker by creating custom MQTT packets with proper flags set and successfully get a successful response from the broker. For us to do that, we must delve into the MQTT packet and how it works.

The MQTT packet is sent over TCP, and it has the following three components: fixed header, variable header, and payload. The fixed header is used for all packets, such as connect and subscribe, and is always present and is mandatory. It contains the message type and size. The second component is the variable header; this part of the packet is optional and contains additional routing and handling information for special types of packets [17]. This part of the packet can contain the protocol name, protocol version, connect flags, keep alive, and properties. Certain parts of the variable header are encoded in UTF-8 strings. Lastly, the payload contains the actual message of what is being sent [17]. For example, an MQTT packet publishing the temperature would be inside the payload, such as payload:"temp"21.5". It also contains other details, such as client ID, username and password. Therefore, these three components make up the foundation for an MQTT packet and can be utilized to spoof packets or for its intended use.

2.3 How MQTT works

After a connection is successfully established between the client and the broker, there are a lot of practical use cases that can be established; however, we must understand some terminology beforehand. Upon connecting to the broker, there are two options: clients can either subscribe to a topic or publish to a topic. The topic is an address that defines where the message shall be delivered and is organized in a 'UTF-8' string format. It is often used in a directory-like structure such as "sensors/temperature/livingroom". When a client publishes data on a topic, then it goes through the broker. Furthermore, the broker then forwards that data to all the clients subscribed to that topic. The broker acts as the man in the middle, facilitating the traffic between the subscriber and the publisher. This allows for a device such as a phone to control the state of a light bulb by sending data to the broker, which can be on a local network or a cloud network, which then forwards that data, such as "on" or "off," to the subscriber of that topic.

2.4 Mosquitto Broker

Mosquitto is an open-source server that acts as a broker for the MQTT protocol for 5.0, 3.1.1, and 3.1, and it was written in C by Roger Light. It is a very popular choice for those who wish to use the MQTT protocol; it has 600 million pulls on Docker and 500,000 pulls this week [14]. A Mosquitto Broker for the MQTT protocol is simple to set up and can be installed on both Windows and Linux operating systems. For our research we will be using Kali Linux to install and use the Mosquitto Broker. The Mosquitto Broker can be installed with the command 'sudo apt-get install mosquitto' and is also widely available on Docker. It requires a configuration file to specify details such as the port number and access details. An example is listed below to show the kind of parameters that can be specified within the configuration file.

```
user root
listener 1883
allow_anonymous true
```

2.5 Vulnerabilities and misconfigurations within MQTT Brokers

Security researchers have been critical to point out vulnerabilities within the MQTT domain that have severe impact on the confidentiality, integrity, and availability of the server. Therefore, we can see since the inception of MQTT, numerous security-related updates have been released, ranging from denial-of-service attacks to including support for TLS encryption, as it appears the protocol was not developed with security in mind. We will highlight security concerns within MQTT brokers in a table format. Please note that the developers of MQTT broker applications such as Eclipse Mosquitto have done a great job at patching many vulnerabilities and providing rapid support, although these issues remain a concern as not all brokers are updated to the latest version or configured to be secure.

Vulnerabilities	Explanation
Lack of Encryption	The MQTT protocol by default runs over port 1883 without encryption. In the context of the MQTT protocol, data privacy and security are crucial in maintaining the confidentiality of our systems. Although not a vulnerability

	but rather a misconfiguration (depending on the risk), as later versions offer support for encryption. Therefore, configuring MQTT brokers to use secure protocols such as SSL/TLS is crucial in preventing attacks such as MITM or information disclosure. As encryption is becoming a standard as we move forward in security.
Lack of Authorization	Many MQTT brokers by default allow anonymous access to the server or can be misconfigured to do so. In cases where the broker is exposed to the cloud, it can pose a significant threat to the system. Therefore, unauthorized access allows anybody to the system to publish or subscribe to topics, which can lead to other attacks. [10].
Weak Credentials	Although not a vulnerability, a misconfiguration could happen through the use of weak credentials such as 'admin:admin' or not setting a password. This misconfiguration can pose a significant threat to the server leading to unauthorized access to the MQTT broker.
Wildcard topic abuse	Wildcard topic abuse can occur if the broker is misconfigured. It can allow subscribers to subscribe to all topics using the wildcard '#' and can potentially lead to information disclosure.
Denial of Service	A common issue with MQTT brokers is that they have been at risk of denial-of-service attacks, especially with older versions even having publicly available proof-of-concept code available for such attacks. This can successfully take the server offline, impacting the availability of the server.
Non MQTT Traffic	Although not vulnerability but a misconfiguration would be improper firewall rules allowing other traffic such as UDP. Considering MQTT runs over TCP, the best practice would be to drop all the UDP traffic and only expose port 1883 or 8883[10].

In addition to these vulnerabilities and misconfigurations, an entire database was created by 'Shodan,' a security company known for its search engine of publicly connected devices. This database contains all common vulnerabilities and exposures (CVEs) related to Mosquitto brokers. The URL 'https://cvedb.shodan.io/cves' can be invoked using an GET request to find a full JSON database of all the associated CVEs within the Mosquitto brokers when provided the CPE string along with the version number. Therefore, if the broker version is publicly available, one can query all the vulnerabilities associated with the version number, which can further be used to craft attacks.

2.6 Vulnerability Scanner

A vulnerability scanner is an automated program used to discover weaknesses in a system that can be exploited by a threat actor for malicious purposes. It is often used by security professionals to help understand the vulnerabilities within a system to patch them before threat actors exploit them. Vulnerability scanners come in different types and are divided into two categories: active and passive scanners. Active scans work by actively simulating different probes to trigger a behavior associated with a vulnerability or a behavior it is looking for. While passive scanners do not actively send out probes but observe traffic to gather information. Moreover, vulnerability scanners can be general and work on all hosts or be designed for specific applications like web applications. For example, the enterprise-grade vulnerability scanner OpenVAS is often used globally in enterprises to detect vulnerabilities within a variety of hosts in the network. Similarly, Nessus is used within large enterprises and is designed to work with a variety of hosts, although Nessus requires a paid license to function. Both Nessus and OpenVAS use network vulnerability tests, which use the NASL coding language inspired by C to check if a host is vulnerable. These NVTs are stored in a database folder, and a server may run tens of thousands of NASL files to check if a target is vulnerable [26]. For example, OpenVAS stores its NVT in the folder '/usr/local/var/lib/openvas/plugins/' and it is often updated to meet the ever-changing dynamics of cyber-security. Therefore, this type of scanner is dynamic, as the developers can add more tests for new vulnerabilities without having to change the source code in depth or at all. Furthermore, a vulnerability scanner after running all the tests would produce a report with all the vulnerabilities sorted based on risk to provide a meaningful result to the end user. These qualities are what make OpenVAS and Nessus the top choice for many security teams worldwide.

3 Literature Review

Within academia and non-academia there has been concern about the security issues that are within the MQTT protocol domain due to a variety of vulnerabilities that have been discovered. In 2014 we can see security research highlight concerns about the security in IoT, leading them to develop a framework called 'Model-based Security Toolkit' that is designed to protect the data and privacy of IoT systems (Ricardo Neisse, Gary Steri, Gianmarco Baldini). Recently we can see researchers still creating papers highlighting security concerns in IoT devices and more specifically in the MQTT protocol. For example, in 2024 researchers Shams Ul Arfeen Laghari, Wenhao Li, Selvakumar Manickam, Priyadarsi Nanda, Ayman Khallel Al-Ani, and Shankar Karuppayah teamed up to create a paper about security attacks within the MQTT ecosystem and provided a framework to mitigate them. This research was crucial in highlighting ongoing attacks within the MQTTMQTT domain. There has also been some work and research on automated systems to detect vulnerabilities within the MQTT domain [15]. All research has concluded that there are a variety of security concerns within the MQTT domain that must be addressed, such as DDOS attacks, lack of authorization, lack of encryption, and information disclosure, although through the consistent stream of updates by open-source brokers these changes have been addressed.

In terms of automated tools for MQTT brokers we can see first an NMAP script developed by security researcher 'Mak Koly Babi' which can detect the version for MQTT brokers relying on the '$SYS/broker/version' for information. This was crucial at allowing threat actors to enumerate the version of brokers highlighting the issue of information disclosure. Furthermore, another critical automated tool is MQTT-PWN developed by Daniel Abeles and Moshe Zioni. This tool provides a framework for penetration testing of MQTT brokers and can be used for malicious purposes. The tool offers several notable features such as:

1. Credential brute-forcer
2. Topic enumerator:
3. Useful information grabber
4. GPS tracker:
5. Sonoff exploiter
6. Shodan integration:
7. Extensibility

This was the first tool we found related to MQTT that was extensible meaning new plugins can be added with ease, which is important as the field of security is ever changing. Researchers have pointed out the dangers of this tool as it made it possible for inexperienced individuals to start hacking MQTT brokers or other IOT devices without much knowledge in a manner like other frameworks like Metasploit.

Regarding research surrounding vulnerability scanners for detecting vulnerabilities within MQTT brokers, security researchers A. Palmieri, P. Prem, S. Ranise, U. Morelli, T. Ahmad created in 2019 a tool called MQTTSA (MQTT Security Analyzer). This tool markets itself as a misconfiguration detector that can detect the top misconfigurations for MQTT brokers and then generate a report. They created this tool with a standalone 712-line Python script that takes in arguments which is then used to actively runs attacks relying on python logic to detect if it is vulnerable. It conducts attacks such as man in the middle (MITM) to intercept credentials on a given network interface, brute-force attacks against brokers using a supplied word list, and information disclosure by checking intercepted messages for sensitive data such as IPs, MACs, emails, passwords and GPS coordinates. Additionally, MQTTSA supports malformed data injection, where it crafts protocol-invalid MQTT packets to test the robustness of the broker's parsing engine. Furthermore, it also runs a test denial-of-service (DoS) capabilities include connection flooding (fast and slow DoS), oversized payload delivery to detect if the broker is vulnerable to a DOS attack. After it runs all the tests it will then produce a PDF report with all the vulnerabilities it found and the risk of the broker. Figure 5 shows a report against MQTT broker on version. Although this vulnerability scanner does not work on older versions of 'MQTT 3.1.1' because it attempts to connect using 'MQTT 5' protocol in the packet header. This is problematic because the older versions of MQTT do not accept the MQTT 5 protocol. While they are the most vulnerable and are in most need of being scanned. Using Shodan a search engine for devices connected to the internet our tests concluding that the most top version is 1.4.8 with currently 11,757 devices operating with that version over the cloud[17]. Therefore, the need to support older versions that use 'MQTT 3.1.1' is crucial as there is currently demand for it. Furthermore, the fingerprinting feature does not work if the $SYS/broker/version topic is disabled, therefore it seems the term fingerprinting was misinterpreted as fingerprinting can be defined as the process of scanning network traffic, launching specifically crafted packets to than narrow down that information to a version number. Therefore, subscripting to a topic that is intended to provide the version number cannot be defined as fingerprinting. It also combines all the data to provide an overall risk such as High or low to give the developers an idea of the security posture.

Another notable vulnerability scanner designed for MQTT brokers is the MQTT Security Scanner by the company EMQX. It is developed in the programming language ‘Go’ and is more organized than the leading scanner **MQTTSA** (MQTT Security Analyzer) because it uses a main script that connects to other folders such as app/mqtt_scanner which contains scripts such as the scanning engine where specific MQTT-related security tests are conducted via a hardcoded script. Nevertheless, the script requires a configuration file and test for the following security issues:

1. Unauthenticated client access
2. Excessive username length
3. Excessive password length
4. Excessive client ID length
5. Repeated rapid connect/disconnect
6. Maximum simultaneous client connections
7. Broker connecting to blacklisted topics
8. Broker not restricting topic hierarchy depth
9. Broker not enforcing maximum topic name length
10. Broker not enforcing message payload size limits
11. Broker accepting non-MQTT data on MQTT port
12. Broker accepting WebSocket connections with incorrect framing
13. Broker supporting deprecated TLS versions (below TLS 1.2)
14. Unexpected broker behavior under MQTT fuzzing for topic generation

A review of the application found major concerns. Firstly, the EMQX scanner was not updated since Jul 19, 2023, which may explain why I was unable to successfully run the script according to the official documentation on both Kali Linux and Windows 11 with the following error ‘panic: runtime error: invalid memory address or nil pointer dereferences’. Therefore, our analysis will be limited to a source code review on the GitHub project page. Furthermore, the EMQX scanner does not test for lack of encryption, wild topic abuse, malicious topic subscription, information disclosure and Unauthorized-retained Message which are critical security issues security research have pointed out.

The research for evaluating vulnerability scanners for websites has been plentiful and fruitful with research dating back to 2007. This research ensured to check for aspects such as the number of vulnerabilities detected, the number of vulnerabilities missed per scanner and the accuracy of the vulnerabilities. Unfortunately, the research in evaluating MQTT vulnerability scanners is almost non-existent with majority of the research in the security issues in the MQTT protocol and limited research in developing a vulnerability scanner because of the lack of vulnerability scanners for MQTT brokers. The research was very good in detecting misconfigurations and vulnerabilities within the MQTT protocol. In our research we attempt to improve and add onto the work by researchers such as 'A. Palmieri, P. Prem, S. Ranise, U. Morelli, T. Ahmad', who have attempted to fill this gap through the development of MQTTSA in 2019. We begin by creating a dynamic tool that can be easily updated over time using a rule detection approach in comparison to hard coding the entire script as MQTTSA did. Their approach by using a 712-line python script makes it discouraging for others to add onto the existing script and creates an unorganized software environment. Furthermore, we attempt to create a fingerprinting module that allows us to detect the version even when it is deliberately blocked which other tools have failed to acknowledge. Lastly, we attempt to include CVE mapping to vulnerabilities by connecting our script to a CVE API to ensure we can have a full list of vulnerabilities associated with the version of the MQTT broker.

4 Methodology of Dreamscan4MQTT Vulnerability Scanner

4.1 Overview

Research has proven time and time again that there are critical security issues in many MQTT brokers exposed to the internet. Furthermore, our research has shown that there are currently thousands of outdated brokers on the public internet highlighting the need for an effective vulnerability scanner. This chapter will discuss the methodology used to design and implement a dynamic, rule-based vulnerability scanner for MQTT brokers. The goal was to develop a scanner that addresses the shortcomings of previous tools by ensuring that the tool is dynamic and extensible, allows for effective finger-printing, detects vulnerabilities using Common Vulnerabilities and Exposure (CVE) mapping. The scanner is inspired by effective tools such as OpenVAS by having a database of tests to run in an organized manner. It is also inspired by Suricata by using rules to match objects against and using CVEs rather than testing only for generic vulnerabilities.

4.2 Design of Program

The proposed scanner is designed to be extensible and includes the following core components and features. The end user will execute the main script, that triggers each module and can dynamically work when new modules are added. Each module is structured as a folder containing two important subfolders:

1. Firstly, each module contains a folder called 'tests' including a python network test script or multiple python network test scripts. These scripts actively check for specific behavior patterns by actively probing the broker by connecting or spoofing packets and recording a log of what happened.
2. Secondly it contains a rules folder with a single python script that contains one or multiple rules that match against the data produced by the tests. After running our tests, the script will automatically store the data in a dictionary format in a JSON file. Lastly, we have rules folder that contains a script with pre-defined rules against the log from the test cases.

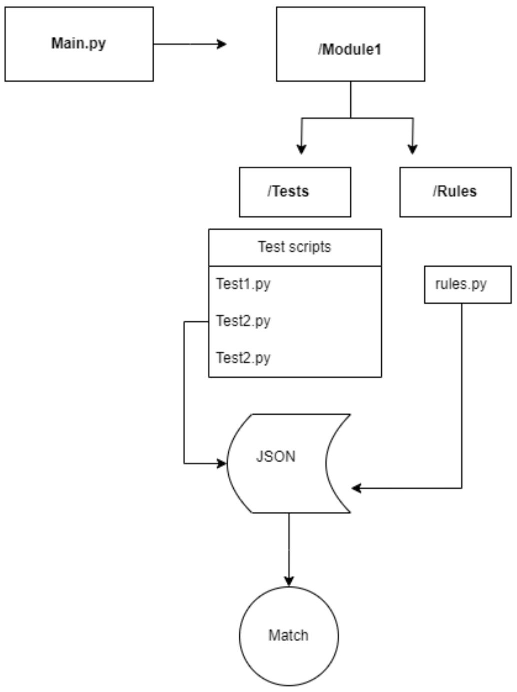


Figure 2 - workflow of algorithm

Figure 2 shows the design and algorithm of our program in more detail. In addition to this we have functions that are run at the end and imported into our main script, these functions are:

CVE Mapper: Our main scripts have a CVE mapper function imported to identify known vulnerabilities against MQTT brokers using the version number. Upon gathering the version number, it then queries data from a public database via an API. This is what makes our methodology different from others allowing us to check for more vulnerabilities and ensuring we are always updated with latest vulnerabilities. These CVE's can be verified and validated using the algorithm in figure 2 , by creating a module for each CVE.

Report Engine: Lastly our main script shall have a report engine function imported, it is used to compile all of the data generated from our program into a solid meaningful report that shows the vulnerabilities, their severity, references and other information.

4.3 Teaching LLMs to Create Modules

By decomposing our scanner into easy-to-understand components it not only makes it easy for humans to understand but also for artificial intelligence to understand our code. This then creates a unique scenario where we can teach large language models (LLMs) such as ChatGPT or Gemini to create new modules after explaining the structure, effectively saving the time and costs associated with developing software. Considering each component follows a consistent structure, such as each module having a separate folder for tests, rules, and result outputs. It makes it ideal for LLMs to be taught how to generate an entirely new module that can fit into our existing project. For example, a user could describe a new module such as checking if passwords are enabled, and an LLM could produce the required Python test scripts, matching detection rules and the user would simply add those into the project folders like a puzzle. This works because our modular design uses decomposition to break up complexity into smaller, understandable parts. Therefore, after explaining to the LLM the structure of the script it becomes easier for LLMs to create a module with accurate rules because the format is already defined. This approach not only helps automate routine development but can be used to quickly adapt the scanner to have thousands of modules that can cover newly discovered CVEs or older ones. In summary, decomposing logic into modular components is not only a part of clean code, but it also enables LLMs and to extend the scanner by simply prompting the LLM of the kind of module we would like to create.

4.4 Test Cases – Spoofing MQTT packets

Within OpenVAS we can see a database folder that contains network vulnerability tests which use the NASL coding language to check if a host is vulnerable. These tests are stored in /usr/local/var/lib/OpenVAS/plugins/’. The main script will run each of these NVTs. Therefore, we will follow suite using python instead of NASL to send active tests to the MQTT broker however we will store results into a JSON file. Moreover, many of these tests require us to spoof MQTT packets for specific behavior and save the result of that behavior in a folder. Other tools such as MQTTSA, use the python library ‘paho.mqtt.client’ , which strips away many of the functionalities need to test for specific behaviors. Our tool uses raw socket programming sent over TCP by creating TCP sockets, adding MQTT headers and sending it directly to the MQTT broker. Giving us more flexibility and freedom to conduct complex tasks like fingerprinting.

4.5 Fingerprinting Methodology

As previously mentioned, many tools failed to properly implement fingerprinting, this is because when the system administrator deliberately hides the version of the broker, the tool does not work. Our tool will attempt to subscribe to the $SYS/broker/version topic and if it is disabled, we will send network tests checking for specific behavior. We will then narrow our results down as much as possible to try to get to a single Mosquitto broker version. This is unique because it can gather information about the broker even when it is deliberately hidden which other tools successfully have done. This technique is known as fingerprinting and is used in other tools in cyber security. Our methodology differs because it attempts to use network tests to gather information about the MQTT broker rather than subscribing to topics.

5 Implementation of DreamScan4MQTT

5.1 Overview

This chapter presents the technical implementation of the proposed dynamic, rule-based vulnerability scanner for MQTT brokers. For our test case we will be testing against the Mosquitto™ broker using a variety of different versions. The scanner is implemented in Python3. The tool , DreanScan4MQTT will be ran on Kali Linux 2024.4 and the broker will be ran using Docker containers.

5.2 Test Cases - Spoofing MQTT packets

Test cases are used to actively probe the data to check for specific responses, we will be going over the test cases used in the fingerprint module to show our implementation of the methodology. As previously mentioned, the fingerprinting module was implemented incorrectly for the tool MQTTSA as fingerprinting should work even when the version is deliberately hidden through the disabling of SYS topics in MQTT brokers.

To conduct fingerprinting we will use open-source intelligence of the Mosquitto broker change log. This change log is publicly available and shows changes per version [19]. Using the changelog, we can create many test cases to have a solid fingerprint of the exact version even when the topic is disabled. This works by spoofing MQTT packets for CVE's and specific updates mentioned in the changelog. For the scope of our research, we will be only attempting to detect one version due to the complexity of such a task although the concept remains the same for all versions. These test cases are implemented using Python scripts and mostly send spoofed packets to the broker. Depending on the response we receive we will create a dictionary log into a JSON file with the test and result. This dictionary format is used to ensure we can process it with our rule detection engine. For example, in one of our scripts, we send a spoofed packet to test if our broker will accept invalid subscriptions which is a bug listed in the Mosquitto 'changelog.txt' for the patch 1.5.7 – that occurred in 2019-02-13. Furthermore, if the length is greater than 0 it shows we got a response back and the broker did not reject the subscription. Using the two imports struct and socket we can spoof a packet to check for this exact behaviour. To conduct this, first, we create a function called encode length. This function is needed because in the OASIS MQTT Version 3.1.1 Specification, Section 2.2.3 Remaining Length, it explains "The Remaining Length is encoded using a variable length encoding scheme which uses a single byte for values up to 127." [21]. Therefore, we must follow that exact format to meet the specifications. We can implement this in Python by following the same algorithm so we may also encode our remaining length when needed according to the official documentation.

```python
def encode_length(length):
    encoded = b""
    while True:
        digit = length % 128
        length //= 128
        if length > 0:
            digit |= 0x80
        encoded += struct.pack("!B", digit)
        if length == 0:
            break
    return encoded
```

code snippet: remaining length algorithm

Next, we need to connect to the broker using an IP address and port to create the connection socket. After establishing a socket connection, we can now create the MQTT packet to connect with the packet in three parts:
• The variable header: This includes the protocol name, for example "MQIsdp". It also includes attributes such as its length (2 bytes), followed by the protocol level byte 3, the connect flags byte 0x02 to set Clean Session, and the keep-alive interval set to 60 seconds (2 bytes). The struct pack is used to convert convert and text lengths into the exact binary format required by the MQTT protocol using format codes like "!H" for a 2-byte unsigned short and "!B" for a single unsigned byte. The! means network (big-endian) byte order [22].

• The **payload**: contains the client ID, followed by its length. The struct pack is used to encode the string into big endian two bytes short integer followed by its length which is 9.

• The **fixed header**: starts with a single byte 0x10, which shows it is a CONNECT packet, followed by the Remaining Length field.
The Remaining Length is computed as the total length of the variable header and payload, then encoded using the MQTT variable-length encoding implemented in the encode length function.

After assembling the CONNECT packet from these fundamental headers, the script sends it to the broker using 'sendall' and then waits to receive the 4-byte CONNACK response to show a connection has been acknowledged.

After a connection is established, the script it will then build and send a separate SUBSCRIBE packet containing an intentionally invalid topic to test whether the broker correctly rejects malformed subscriptions.

```
  topic = b"foo/+/bar"
  tf = struct.pack("!H", len(topic)) + topic + b'\x00'
  vh = struct.pack("!H", 1)
  subscribe    =    struct.pack("!B",    0x82)    +    en-
code_length(len(vh) + len(tf)) + vh + tf

  s.sendall(subscribe)
  suback = s.recv(5)
  result = suback.hex()
  print(result)

  if len(result) > 0:
      log = {
          "test": "Reject invalid un/subscriptions",
          "result": "Not Rejected"
      }
```

The script is an example of something we can test for. In this implementation we test to see if the broker allows invalid subscription topics, which is behaviour in outdated brokers. It first defines the invalid topic b"foo/+bar", which is not allowed in later versions. It then builds the payload (tf) by packing the topic length, the topic itself, and a QoS byte which is ‘b'\x00'. The variable header (vh) is the packet identifier set to 1 to show it’s a subscribe packet. Then, it constructs the fixed header by adding the byte 0x82 which is the SUBSCRIBE packet type with required flags on top of the Remaining Length which was calculated by the encode function. Finally, the SUBSCRIBE packet is then sent to the broker using s.sendall(subscribe). We then wait for a SUBACK response using s.recv(5) and it converts the response to hexadecimal to check for its length depending on the length their will be a different log as if its rejected there will be no response as its 0 if the response is greater than 0 than it was not rejected. Therefore, we can add the log and save it in the results folder which is adjacent to the main script to make decisions. This is one example of a network test, but certain modules can have more than one test or have only one test.

Upon sending the packet we captured the traffic using Wireshark. The Wireshark traffic showed the established TCP connection and the established MQTT connection between the MQTT broker and subscriber. We can see that there was a request to subscribe to foo/+bar which is invalid and then we can see a FIN request to close the TCP connection we can also validate this on the broker side.

```
  1752488873: mosquitto version 1.3.1 (build date 2025-06-08 
21:16:59+0000)
  1752488873: Config loaded from mosquitto.conf.
  1752488873: Opening ipv4 listen socket on port 1883.
  1752488873: Opening ipv6 listen socket on port 1883.
  1752488949: New connection from 192.168.15.195 on port 1883.
  1752488949: New client connected from 192.168.15.195 as raw-
client (c1, ...).
  1752488949: Invalid subscription string from 192.168.15.195, 
disconnecting.
  1752488949: Socket error on client rawclient, disconnecting.
```

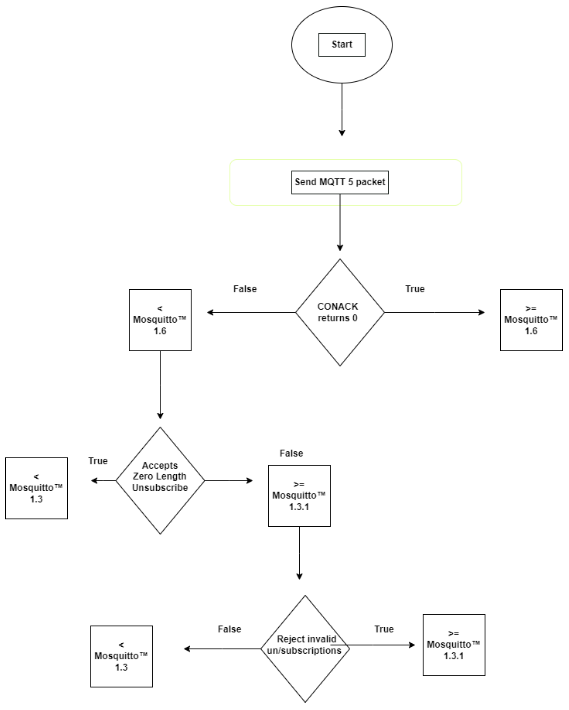
Therefore, the broker had indeed validated that it did disconnect the session due to invalid subscription string, which is unique to versions later than 1.3.1, showing accuracy within our log as it logged that the broker rejected the connection.

5.3 Fingerprinting Prototype

As shown within the literature review there were some inconsistencies with the fingerprint shown in MQTTSA, as a successful fingerprint shall not rely on the MQTT SYS version topic. Rather, it should send packets to probe the broker to get a version. Therefore, we will present a fingerprinting prototype that can detect an approximate version by sending packets to the broker.

Using open-source intelligence, we can use the mosquito-change log text file to check whether certain bugs still exist within our broker. We can first start with a simple test to narrow out as much versions as possible which is to check whether there is support for MQTT v5 which was a major change introduced Mosquitto version 1.6. If the broker does not accept the MQTT v5 packet than we know its under 1.6. Secondly another major change would be to check whether the broker will accept zero length subscription strings which is a bug fixed in Mosquitto version 1.3.1. We can also check if the broker does reject zero length subscription strings. If it does than we can narrow our results down to between 1.4-1.6. To further narrow it down we check if there is - added support for "session present" within CONNACK messages for MQTT v3.1.1.

Which is derived from an update added for version 1.4 of Mosquitto brokers. If there is no support, we now that our broker is under version 1.4. Lastly, we can check if our broker is rejecting invalid subscriptions such as. foo/+bar #/bar which is not allowed in versions above 1.3.1. If it rejects these subscriptions than we can successfully conclude that our broker is between version 1.3.1-1.3.5. We could further narrow it down however there are no major changes that can be externally tested as well as no CVEs. Therefore, this is more than enough information for the end-user. Considering there are no cve between 1.31-1.3.5 we can confidently use the highest version 1.3.5 and generate a report with all the CVEs. Considering there are no CVEs between 1.31-1.3.5 we can confidently use the highest version 1.3.5 and generate a report.



5.4 Implementation of Rule detection Engine

As previously mentioned in the methodology chapter, the test cases produce a log depending on the behavior. This was purposely done so it can be processed with a rule detection engine. We will be integrating the rule detection engine developed by security researcher Spencer McIntyre and his colleagues. The documentation mentions it is a designed to be used as a tool creating general purpose “rule” objects from a logical expression which can then be applied to arbitrary objects to evaluate whether they match’ [23]. This is exactly what we need to be run against our logs therefore we will be using the dictionary format as our standardized format for our logs generated from the test cases for a smooth integration with rule detection engine. Each rule will be placed in the rules folder inside a python script to check for logic

To begin with we will create a rules folder within each module folder in the same directory as the tests folder, this folder will store all the rules script. The rules and logs will be in the following format:

Log: {"test": "x", "result": "y"}
Rule: rule_engine.Rule('test == "x" and y'),

If we get a match for both x and y then the rule will be sent back either true or false. An example of this implementation would be the fingerprint module. What makes this example unique is the fact that there are multiple logs and tests we will need to process. Therefore, after running all the test cases it will produce a log file in a separate directory known results/results.json inside the module folder, below is an example of the results.json file.

```
{"test": "Reject invalid un/subscriptions", "result":
"Rejected"}
{"test": "mqtt_v5_connect", "connack_return_code": 1}
{"test": "session present check", "result": "session
present not supported"}
{"test": "unsub_test", "result": "Connection closed
by broker with no response."}
```

Furthermore, we have the script ‘rules.py’, containing the rules with a rule detection engine imported. The rule detection engine looks for an exact match within the logs, such as the test case and the related log. Below is an example of four rules used in our fingerprinting module. It works by looking for an exact match within the results.json file.

```
  # Define rules
  rules = [
      rule_engine.Rule('test = "mqtt_v5_connect" and
connack_return_code = 1'),
      rule_engine.Rule('test = "unsub_test" and re-
sult = "Connection closed by broker with no re-
sponse."'),
      rule_engine.Rule('test = "Reject invalid
un/subscriptions" and result = "Rejected"'),
      rule_engine.Rule('test = "session present
check" and result = "session present not supported"')
  ]

  rules_matched = []
```

At times the script contains multiple rules to match the test cases and their results; it will open our 'results. json' file then loop through each rule and save the matches in an array with 1 indicating true i.e.: rule matched and 0 indicating false i.e.: no match. Therefore, in the above example we are looking for 4 matches, if the length is four, we can successfully make a conclusion about our fingerprint version and add it in our report.

5.5 Using LLMS to create additional models

Considering we have a working implementation of our dynamic modular scanner, since we have decomposed our program into a variety of different components, it can be trained to large language models to create additional models including rules since we have used decomposition techniques to split up the logic. We will be using ChatGPT model 4.0 to create new modules.

We first send parts of our code to the model one by one and explain the logic to the code just as we would to our peers, we also used the tree command in Linux to explain the organization of the code and file paths to the ChatGPT model 4.0.

After sending the code and the tree output sent the following prompt to ChatGPT using the GPT4 model to create an entire model by asking the prompt 'I wanna (would like to) create a module that checks for authorization meaning if they have a password enabled or not send code (rules , tests as my other code did)' .We have sent the following human friendly prompt to test if the LLM model will actually send back an accurate module. Within seconds we can are able to get an accurate response from the LLM which has been included below.

LLM Response:
Perfect — here's a clean, ready-to-drop module named **Auth-Check** in your style, with:

- **tests/testing.py** to actually test if broker allows connections without authentication
- **rules/rules.py** to define a simple rule that flags this as vulnerable if it succeeds

We can see that the LLM generated a module by sending accurate rules and test-cases with logs by sending a modules/Auth-Check/tests/testing.py and a modules/Auth-Check/rules/rules.py.

5.5.1 LLM generated module

Below is the following rule generated by the LLM to be used on the logs generated from the test scripts.

```
   rule_engine.Rule('test == "Unauthorized access check" and result == "Broker allows unauthenticated connection."'),
      rule_engine.Rule('test == "Unauthorized access check" and result == "Broker rejects unauthenticated connection."'),
  ]
```

Below is the following logs generated by the LLM to be generated after running the test cases.

```
  { "test": "Unauthorized access check", "result": "Broker rejects unauthenticated connection." }
  { "test": "Unauthorized access check", "result": "Broker allows unauthenticated connection." }
```

Therefore, by using decomposing our vulnerability scanner we can train LLMs to generate entire modules saving us the manual time of programming or hiring security engineers which can be costly. As opposed to pasting MQTTSA's 712-line python script directly into an LLM model , it would likely crash or potentially break the structure of the script causing more errors.

5.6 Implementation of functions

Within our vulnerability scanner it is important that we discover and utilize the common vulnerabilities and exposures so we can establish a reference point for our vulnerabilities. Therefore, we will implement a function inside of a separate script that can accurately gather CVE's using the version of the broker. We will also have a separate function to detect the version of the broker. Both functions will be used as an import in the main script so they can be utilized within our program, making it clean for the end-user.

For the version detection script, we will use the mqtt client library to subscribe to the $SYS/broker/version to determine the version if it does not work, we will use the highest value in the fingerprinting module which will store the value of the highest version in a variable. For our CVE discovery script, it will then convert the python variable containing the version number to a CPE value. This is done so we may use it with the Shodan API designed to dump all the vulnerabilities related to the version number in a JSON-like structure with details about the vulnerabilities. We will then have a full list of all the associated vulnerabilities with that version by invoking the API as shown below using a GET request and the data saved into a JSON file.

```
GET  https://cvedb.shdan.io/cves?cpe23=cpe:2.3:a:eclipse:mos-
quitto:1.6.10:*:*:*:*:*:*:*
```

Output:

```
"cves": [
  {
    "cve_id": "CVE-2024-10525",
    "summary": "In Eclipse Mosquitto, from version 1.3.2
through 2.0.18, if a malicious broker sends a crafted
SUBACK packet with no reason codes, a client using
libmosquitto may make out of bounds memory access when
acting in its on_subscribe callback. This affects the
mosquitto_sub and mosquitto_rr clients.",
    "cvss": 9.8,
    "cvss_v3": 3.0,
    "cvss_v3_2": 9.8,
    "cvss_v3_3": 8.0,
    "ranking_eps": 0.1,
    "kev": false,
    "propose_action": null,
    "references": [
      "https://github.com/eclipse/mosquitto/com-
mit/8ab20b4b4204fdcdec78cbdf03c944a6e0e1c",
      "https://gitlab.eclipse.org/security/vulnerabil-
ity-reports/-/issues/190",
      "https://mosquitto.org/blog/2024/10/version-2-0-
19-released/"
    ],
    "published_time": "2024-10-30T12:15:02"
  },
  {
```

Figure 31- example of CVE's being detected using an API and GET request

This can now be processed and added to our report to add meaningful value to the report. This also can be used in conjunction with specifically designed modules to verify each CVE for better accuracy.

5.7 Report Generation

At this stage we have gathered a lot of data from the MQTT broker. Our last step is to gather all this information into a meaningful report. We have created the final script called generate_report.py which is also imported as function by the main script. After our main script is called, we should have a variety of data points as seen in the output of the ls command. This data is gathered from the modules and the CVE API from Shodan. Therefore, our next task is to gather all this data into a single PDF report.

```
  $/home/kali/Desktop/DreamMQTT/results
   ls
  auth-check.json                 fingerprint-matched.json
mqtt_cve_data.json  results.json
```

The script generate_report.py will open the mqtt_cve_data.json and store all the data in variable raw. It will also organize all the CVE's based on score. "Critical (9.0–10.0),High (7.0–8.9)": "Medium (4.0–6.9)","Low (0.1–3.9)" and place them into arrays, so we can refer to them later to allow us to calculate the severity that each vulnerability presents similar to NESSUS and OpenVAS. The script will then create a TEX file to make the report as seen in the snippet below it makes the title and other key fundamentals of a report.

```
  with open("mqtt_cve_report.tex", "w") as f:
          f.write(r"""\documentclass{article}
  \usepackage[utf8]{inputenc}
  \usepackage[margin=1in]{geometry}
  \usepackage{hyperref}
  \title{MQTT CVE Security Report}
  \date{""" + now + r"""}
  \begin{document}
  \maketitle
```

There will be a second section that contains information about the broker, its IP , Port. It uses the variable raw that has all the loaded JSON data to allow us to substitute data from the raw Json into our TEX file.

```
  \section*{Broker Information}
  \begin{itemize}
    \item \textbf{IP Address:} """ + raw["broker_ip"] +
r"""
    \item \textbf{Port:} """ + str(raw["broker_port"]) +
r"""
    \item \textbf{Version:} """ + raw["version_raw"].re-
place('_', r'\_') + r"""
    \item \textbf{CPE:} """ + raw["cpe"].replace('_',
r'\_') + r"""
  \end{itemize}
```

Moreover, the script will then add a vulnerability summary section which will loop through each CVE and add the ID, Score, References and other metadata using python variables and latex. Next, we shall also read the other data for our additional tests such as fingerprints and authentication checking which has data stores in the results folder. We can now create simple latex structures that read the data and store them in a variable and input them into the latex script as seen below in the snippet.

```
  f.write(r"\textbf{Test:} " + test_name + r"\\" + "\n")
  f.write(r"\textbf{Result:} " + result_text + r"\\" +
"\n")
```

Lastly, we can finally end the document in latex and use a subprocess that will convert from latex to pdf providing us with a neat PDF file.

5.8 Main script

```
/home/kali/Desktop/Dreamscan4MQTT
├── main.py
├── modules
│   ├── Auth-check
│   │   ├── rules
│   │   │   └── rules.py
│   │   └── tests
│   │       └── testing.py
│   ├── fingerprint
│   │   ├── rules
│   │   │   └── rules.py
│   │   └── tests
│   │       ├── send_invalid_subscription.py
│   │       ├── send_mqtt_v5_connect.py
│   │       ├── send_sessioncheck.py
│   │       └── send_unsubtest.py
├── reports
├── results
│
│
└── scripts
    ├── generate_report.py
    └── version_detection.py
```

project folder structure

Our main script is called main.py under the folder Dreamscan4MQTT, which is the name of our project and is the folder the end-user would receive when downloading the scanner. The objective is to create a main.py script which can be renamed to dream-scan4mqtt which would act as the front end of the script that the user executes. As previously mentioned it also imports three functions from other scripts

1. function to detect version
2. function to detect vulnerabilities using API
3. function to generate the final report

5.8.1 Dynamic running of scripts.

The main script has 5 main steps:

1. Detect all modules in the program and initialize all associated path names in a variable
2. Automatically run each test script in the tests folder per module
3. Automatically run the rule script per module
4. Repeat until all modules are processed
5. Invoke additional functions needed to complete the programs algorithm.

To ensure our script is dynamic, it should automatically detect the modules, run their test scripts and repeat until all modules have been processed. We will create a function that utilizes the subprocess module to run our tests scripts automatically. The function first finds and analyzes all the module folders. Secondly it uses a loop to run all test scripts in the test folder within each module. and the function is invoked to run the script. Therefore, we will invoke this function later within our script. Moreover, to ensure our script is module we have to avoid adding hard coded file paths such as /home/kali/ as a successful script shall work in different environments and be portable. Therefore, we initialize the main function to begin by getting the absolute path of the current file and than join with the modules path to ensure we can automatically create our paths.

The script then initializes a variable within the module folder to store the path of the tests folder containing the network tests and checks if there are files within the folder. If there are files it will then loop through the tests folder and invoke the function to execute each python script in the test folder and if it cannot find it will print a message to the user. These test cases will produce data in folder called results and will be utilized for the rules section of the program.

After all modules have had their test cases executed the program will identify each rule script within the modules folder. This rules script is a single script containing rules to be ran on our data from the test cases. If our rules had match on the data produced from the test it will record that there was a match or no match.

Lastly the script we will invoke additional functions to scan for the version with all the CVE's and generate a report using all the data created from previous scripts. This order allows us to utilize all the data generated from the rules and test scripts to provide it in a concise matter.

Therefore, the main script serves as the front-end of our script to orchestrate scanning, rule matching, and report generation. The script conducts these tasks in a dynamic way that requires little to no tuning for the end-user making it beginner friendly for IOT developers who are interested in securing their product.

6 Experimental Evaluation and Results

In this chapter, we will present the results obtained from running our experiments on the MQTTSA and DreamScan4MQTT vulnerability scanners. We will also make a comparison of the results to highlight weaknesses and strengths.

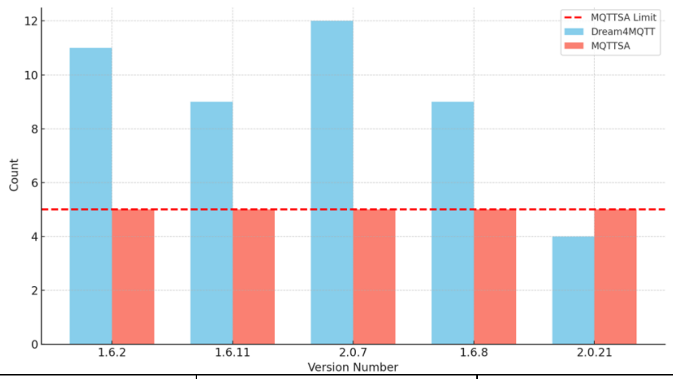
6.1 Vulnerability Detection

In this sub-section we will discuss how many vulnerabilities each scanner was able to detect for a vulnerable MQTT broker. For MQTTSA the command used for the remainder of tests will be:

```
  'python mqttsa.py 192.168.15.21 -i "Wi-Fi" -p 1883 -v 5 -t 60
-m "test message" -fc 10 -sc 5 -mq 50 -u testuser -w wordlists\us-
ers.txt --md --ni -mup 10'
```

In addition, the broker will be misconfigured to allow for unauthenticated connections to the broker for detecting the number of vulnerabilities. For certain tests the broker will block unauthenticated connections to test the accuracy of the scanner.

Number of Vulnerabilities detected per scanner



Version Number	DreamScan4MQTT	MQTTSA
1.6.2	11	5
1.6.11	9	5
2.0.7	12	5
1.6.8	9	5
2.0.21	4	5

Our results showed that our scanner was able on average to detect 1.8x more vulnerabilities than the leading scanner, MQTTSA. There is also an issue with the MQTTSA is it does not seem to detect any vulnerabilities that contain CVE's it continuously detects 5 misconfigurations/vulnerabilities even though older versions are being used. This is problematic because older versions of the broker are much more vulnerable. Therefore, there is a critical error in the methodology of MQTTSA as it does not give any importance to CVEs within its security assessment tool giving misleading results across versions. There is a relationship between vulnerabilities and version, and that is the vulnerabilities shall increase as the version number decreases which was not shown within the MQTTSA tool. That is because MQTTSA only tests for the following security weaknesses, which are Outdated Broker, Use of TLS, Information Disclosure, Accessible Service / Weak Access Control, Unlimited Message Queues while our tool uses a database of CVEs to gain a better insight into all reported bugs and vulnerabilities.

False Positives for MQTTSA

Within the MQTTSA they have a section which lists a variety of vulnerabilities our research discovered a high rate of false positives for the vulnerability ‘outdated broker’ on MQTTSA. We noticed that it does not work when the broker version topic is denied. Therefore, we will run the scanner on 5 brokers that have the SYS topic denied with brokers that are indeed vulnerable to outdated brokers.

False positive for Outdated Broker

Version Number	False Negative for Outdated Broker
1.6	✓
1.6.9	✓
1.6.8	✓
1.6.1	✓
1.6.2	✓
1.6.3	✓

Therefore, our evaluations showed that MQTTSA has a 100% false negative when the SYS version topic is disabled revealing that the broker is not outdated, when all the brokers were indeed outdated in our trials

6.2 Compatibility across brokers

A vulnerability scanner shall be compatible with a variety of versions, as the oldest versions are in most need of being scanned. Therefore, we will test each scanner on a variety of versions to see if it works successfully.

Version Number	MQTTSA	DreamScan4MQTT
1.3.1	Errors	✓
1.6	✓	✓
1.5.3	Errors	✓
1.2.2	Invalid protocol "MQTT" in CONNECT	✓
2.0.21	✓	✓

Our results concluded that MQTTSA was incompatible with 60% of the Mosquitto Brokers in our experiment while DreamScan4MQTT was compatible with 100% of the Mosquitto Brokers in our experiment.

6.3 Accuracy of Fingerprint Module

In this section we will present the findings on the fingerprint module with $SYS/broker/version topic disabled for DreamScan4MQTT and MQTTSA to determine its accuracy.

True Negatives for Fingerprinting

Version Number	DreamScan4MQTT	MQTTSA
2.0.22	✓	✗
1.6	✓	✗
1.5.3	✓	✗
1.2.2	✓	✗
1.4.1	✓	✗

True Positives for Fingerprinting

Trial and Version Number	DreamScan4MQTT	MQTTSA
Trial 1 - 1.3.1	✓	✗
Trial 2 - 1.3.2	✓	✗
Trial 3 - 1.3.3	✓	✗
Trial 4 – 1.3.4	✓	✗
Trial 5 – 1.3.5	✓	✗

Our experiments showed that the fingerprint module was working as expected with a 100% true negative and 100% true positive rate for DreamScan4MQTT. However, for MQTTSA our experiments showed that the fingerprint module was not working at all when the $SYS/broker/version topic was disabled.

6.4 Accuracy of LLM aided Modules - Authentication check for DreamScan4MQTT.

True positives

Version Number	Authentication check module
1.3.1	✓
1.6	✓
1.5.3	✓
1.2.2	✓
2.0.21	✓

True Negatives

Version Number	Authentication check module
2.0.22	✓
1.6	✓
2.0.4	✓
2.0.3	✓
2.0.21	✓

Our experiment concluded that the Authentication check module was working with a 100% True Negative and 100% True positive result when cross referencing with the logs.

7 Discussion and Future Work

7.1 Vulnerability Validation & Future Directions

While DreamScan4MQTT was able to detect more vulnerabilities than MQTTSA it does not yet verify each vulnerability. This is because the project is currently a prototype and needs more work and development. The good news is project was designed to purposefully be scalable and modular to allow others to add more modules to verify each vulnerability. For example, OpenVAS has around 50,000 NVT's, therefore a lot of effort is required to update this script to be at an enterprise level like Nessus and Openvas. However, it is possible as we demonstrated even LLM's can generate modules showing how easy it is for others to learn the methodology and create more modules. Furthermore, in the future it may be worth noting to introduce credentialed scans. This can be implemented where SSH access is provided and a directory to the configuration file to validate vulnerabilities that require access into the broker and configuration file.

7.2 Fingerprint Detection & Failures

Dreamscan4MQTT was able to create a 100% accurate fingerprinting module prototype narrowing our results down to 1.3.1-1.3.5. However, we were unable to meet our goal of detecting the exact version 1.3.1. This is because some versions have such small changes that can only be tested on the client side such as configuration file changes which can be accomplished with credentialed scan. We were advised by the supervisor of this project to measure the time of responses per version to narrow our results down further. We sent spoofed packets containing malformed data to the broker and observed a random time range with no viable differences in the time per version. Therefore, showing that time was of no help when calculating a fingerprint for Mosquitto brokers.

This is potentially due to the way it is designed with changes usually occurring at the source code with the developer ‘Roger Light’ usually making fixes on the GitHub source code after a bug is discovered. At times these changes are minuscule as the source code is changed slightly to fix the bug although we have not attempted at targeting the exact change yet through time analysis. However, we were still able to come very close to detecting a range that is quite small and still provides tremendous value to the end-user. Therefore, we are hoping this can be solved by more advanced researchers as fingerprinting has been achieved by other tools such as p0f and Nmap which observe traffic or send traffic using signatures.

7.3 Integration with other MQTT broker types

Currently the DreamScan4MQTT has not been tested on other brokers such as EMQX and rather the scope has been limited to Mosquitto brokers. However, using the dynamic and modular approach via a rule detection engine, the concepts are the same and new modules designed for other brokers can be integrated into DreanScan4MQTT. In the future, a possibility can exist where fingerprinting is used to detect the exact broker vendor before conducting scanning to get a more personalized result.

8 Conclusion

In conclusion, a dynamic and modular approach to detecting vulnerabilities is more effective than a standalone approach of using single python script. This is because standalone scripts make it difficult for others to add upon which is crucial as vulnerabilities are always being discovered. Therefore, our tool proposed a dynamic and modular rule-based approach to detecting vulnerabilities and misconfigurations within MQTT brokers. Results showed that Dreamscan4MQTT is 2.5 x more compatible across broker versions than the leading scanner with 100% compatibility across Mosquitto broker versions. We also corrected some misconceptions with fingerprinting and integrated an accurate fingerprinting prototype using a rule detection engine which is 100% more accurate than the leading MQTT scanner.

References

[1] Axidio Corporation, "Transforming SCM: The synergy of AI, IoT, and big data analytics," 2024. [Online]. Available: https://axidio.com/blog/iot-big-data-analytics-and-ai-in-scm-2024. [Accessed: Aug. 13, 2025].

[2] Gen Digital, "Avast research finds at least 32,000 smart homes and businesses at risk of leaking data," [Online]. Available: https://press.avast.com/avast-research-finds-at-least-32000-smart-homes-and-businesses-at-risk-of-leaking-data. [Accessed: Aug. 13, 2025].

[3] Shodan, "Shodan search: Mosquitto," [Online]. Available: https://www.shodan.io/search?query=product%3A%22Mosquitto%22. [Accessed: Aug. 13, 2025].

[4] G. C. Hillar, Hands-On MQTT Programming with Python. Packt Publishing, 2018. ISBN: 978-1789138542.

[5] A. Piper, Á. Diaz, and A. Nipper, "IBM Podcast: MQ Telemetry Transport (MQTT) — interview transcript," Nov. 18, 2011. [Online]. Available: https://www.ibm.com/podcasts/software/websphere/connectivity/piper_diaz_nipper_mq_tt_11182011.pdf. [Accessed: Aug. 13, 2025].

[6] HiveMQ, "How to get started with MQTT," [Online]. Available: https://www.hivemq.com/blog/how-to-get-started-with-mqtt/. [Accessed: Aug. 13, 2025].

[7] S. Cope, "MQTT protocol — messages overview," [Online]. Available: http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/. [Accessed: Aug. 13, 2025].

[8] N. Author, "Title unavailable," IEEE Xplore, 2024. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=105522552. [Accessed: Aug. 13, 2025].

[9] A. Author, "MQTT security challenges and solutions," ResearchGate, 2022. [Online]. Available: https://www.researchgate.net/publication/360919708_MQTT_Security_Challenges_and_Solutions. [Accessed: Aug. 13, 2025].

[10] Cedalo AG, "MQTT keep alive explained," [Online]. Available: https://cedalo.com/blog/mqtt-keep-alive-explained. [Accessed: Aug. 13, 2025].

[11] N. Author, "Title unavailable," IEEE Xplore, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/105522551. [Accessed: Aug. 13, 2025].

[12] Docker Inc., "Eclipse Mosquitto (official Docker image)," [Online]. Available: https://hub.docker.com/_/eclipse-mosquitto. [Accessed: Aug. 13, 2025].

[13] Akamai Threat Research, "mqtt-pwn," GitHub repository, [Online]. Available: https://github.com/akamai-threat-research/mqtt-pwn. [Accessed: Aug. 13, 2025].

[14] Shodan, "Mosquitto — version facet search," [Online]. Available: https://www.shodan.io/search/facet?query=product%3A%22Mosquitto%22&facet=version. [Accessed: Aug. 13, 2025].

[15] N. Author, "Analyzing the effectiveness and coverage of web application security scanners," ResearchGate, 2011. [Online]. Available: https://www.researchgate.net/publication/228716324_Analyzing_the_Effectiveness_and_Coverage_of_Web_Application_Security_Scanners. [Accessed: Aug. 13, 2025].

[16] Eclipse Foundation, "Mosquitto changelog," [Online]. Available: https://mosquitto.org/ChangeLog.txt. [Accessed: Aug. 13, 2025].

[17] HiveMQ, "MQTT packets — a comprehensive guide," [Online]. Available: https://www.hivemq.com/blog/mqtt-packets-comprehensive-guide/. [Accessed: Aug. 13, 2025].

[18] OASIS, "MQTT version 3.1.1," [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718023. .

[19] Python Software Foundation, "struct — Interpret bytes as packed binary data," Python 3 Documentation, [Online]. Available: https://docs.python.org/3/library/struct.html#format-characters. .

[20] J. Zerosteiner, "Rule engine," [Online]. Available: https://zerosteiner.github.io/rule-engine/.

[21] Wikipedia, "p0f," [Online]. Available: https://en.wikipedia.org/wiki/P0f.

[22] Greenbone Networks, "Active vs. passive scans," [Online]. Available: https://www.greenbone.net/en/blog/active-passive-scans/.

[23] Inductive Automation, "What is MQTT?," [Online]. Available: https://inductiveautomation.com/resources/article/what-is-mqtt.

[24] P. Brunty, "Understanding MQTT," [Online]. Available: https://brunty.me/post/understanding-mqtt/.

[25] HackerTarget, "OpenVAS tutorial & tips," [Online]. Available: https://hackertarget.com/openvas-tutorial-tips/.