Securing MQTT Communication on IoT Devices

Maximilian J. Dobrei
101103400

Carleton University
COMP4905 Honours Project
Summer 2022

Supervised by David Barrera

Abstract

MQ Telemetry Transport (MQTT) is a communication protocol designed to be used by the Internet Of Things (IoT) [1]. MQTT is well known for its lightweight code base, scalability, and variable quality of service [1]. These are all important factors to consider when trying to provide adequate service to IoT devices despite the variety of environments in which IoT devices are used. Many IoT devices have relatively little computing power and may not have a reliable network connection. Alongside the aforementioned characteristics, MQTT supports security features such as TLS for encryption and OAuth to allow authentication [1]. However, these security features are completely optional, and must be explicitly enabled by the implementer. This project aims to explore the feasibility of providing enhanced security to existing deployments of IoT devices using MQTT that lack any of the optional security features. To do so, custom endpoints to the communication protocol will be implemented that offer opportunistic encryption of messages, as well as authentication. While I was successful in this endeavor, the implementation of these custom endpoints revealed several challenges with this approach, and informed future design choices. As part of the conclusion of this report I will examine different avenues to consider in terms of the encryption schemes, concurrent programming, and robust testing.

Table of Contents

1 - Introduction 4

2 - Background

2.1 - MQTT 5

2.2 - Networking principles and Firewalls 7

3 - Technical Specification

3.1 - Implementation details 8

3.2 - Security Analysis 15

4 - Future Work

4.1 - Cryptographic Functions 18

4.2 - Scalability 20

4.3 - Testing 21

5 - Conclusion 23

Appendix A - Changes to AMQTT 24

List of Figures

1.1 Publisher to Broker half of revised communication architecture including supplementary endpoints 9

1.2 Broker to Subscriber half of revised communication architecture including supplementary endpoints 10

2. Command to invoke the MQTT publisher 12

3. RSA Public Key in PEM format 12

4. Sample MQTT CONNECT packet 12

5.1 Sample Broker output showing MQTT CONNECT packet 13

5.2 Sample Broker output showing MQTT CONNACK packet 13

6. Sample MQTT PUBLISH packet 13

7.1 Sample Broker output showing MQTT PUBLISH packet 14

7.2 Sample Broker output showing MQTT PUBACK packet 14

8. Sample MQTT DISCONNECT packet 14

9. Sample Broker output showing MQTT DISCONNECT packet 14

10. MQTT PUBLISH packet 17

11. Encrypted MQTT PUBLISH packet 17

12.1 Code at line 448 in the AMQTT client.py file before changes 24

12.2 Modified code at line 448 of the AMQTT client.py file 25

1 - Introduction

MQTT has been well adopted and deployed in the growing IoT field. MQTT claims to have been utilized in a diverse set of industries such as transportation, smart homes, manufacturing, and more [2]. The breadth of MQTT is exemplified through the large number of programming languages and computing devices, some popular and some more obscure, that implementations for MQTT clients & brokers can be found for [3]. MQTT encourages implementers to consider using the popular Transport Layer Security (TLS) protocol to encrypt messages and to keep them confidential [1]. They also suggest implementing OAuth, a technology that is widely used for authentication purposes [1]. OAuth lets the MQTT broker generate access tokens that can be given to MQTT clients. Only clients with valid access tokens can then establish a connection to the broker. Despite the creators of MQTT claiming that the protocol is “security enabled”, the MQTT standard forces the implementer to enable these optional security related technologies.

The IoT field comprises many millions if not billions of devices that range from simple sensors to elaborate voice recognition systems, such as popular smart home devices like the Amazon Alexa or Google Home. One of the primary security concerns when it comes to the IoT field is the level of support that devices receive post deployment. Some devices, once deployed, will not receive any further software or hardware updates. In terms of devices using the MQTT protocol, this means that for some devices, the security features that were enabled at deployment will never get changed or updated.

In the worst case scenario where a device is deployed with no security features enabled, the device represents a major attack vector that malicious actors can target. Even those with TLS or OAuth enabled at deployment could be compromised and present a new attack vector, should any vulnerabilities be discovered in those respective technologies. With this in mind, it becomes clear that post deployment software updates are extremely important to the security of IoT devices, including those using MQTT. In a situation where post deployment software updates are not realistic or are being ignored, another solution is required. The approach being explored through this project is a set of custom entities, which I refer to as ‘proxies’. These proxies can act on behalf of existing MQTT clients and brokers, and offer opportunistic encryption as well as authentication.

2 - Background

2.1 - MQTT

MQTT as a communication protocol follows a “publish and subscribe architecture” [1]. When an MQTT client has new data it wants to publish, it connects to the MQTT broker and publishes the data under a specific topic [1]. The MQTT broker maintains a list of connected clients and the topics these clients “subscribe” to, if any [1]. When the MQTT broker receives new data it checks the list of clients and forwards the new data to any clients that subscribe to the matching topic [1]. This architecture allows for clients to only publish data once; the burden is put on the MQTT broker to forward the data to all clients that listen for it [1]. As MQTT clients can act as both publishers and subscribers, this architecture provides great flexibility on the client side [1].

When a client publishes new data it has 3 choices for the “quality of service” of the message. Level 0 provides ‘At most once’ delivery: “where messages are delivered according to the best efforts of the operating environment. Message loss can occur” [1]. Level 1 provides ‘At least once’ delivery: “messages are assured to arrive but duplicates can occur” [1]. Level 2 provides ‘Exactly once’ delivery [1]. MQTT allows the client side to determine what level of quality of service is appropriate for the situation [1].

The MQTT broker has options to help it manage how MQTT clients can connect, as well as how many clients can connect at once [1]. It can also filter topics by including a whitelist or blacklist of topic names [1]. These fundamental settings for the broker can be changed to ensure efficient operation depending on the exact requirements of the job. Importantly, the MQTT broker also dictates the security protocol that will be used throughout all stages of communications. A username/password system can be used for authentication of clients if OAuth is not put in place, and TLS can be used to encrypt communications and provide message confidentiality [1]. MQTT clients must comply with all security settings the broker has enabled or they will be unable to publish or receive any data.

2.2 - Networking principles and Firewalls

The MQTT protocol allows communications to take place using Transmission Control Protocol (TCP) or via websockets. TCP is a good choice as it is a time tested tool that has been around since 1974 [4]. TCP has endured for so long because it supports reliable and ordered delivery of messages between two hosts, and because it enforces extensive error checking and error handling during the communication process. At different stages of a TCP transmission, many decisions have to be made. A firewall can be thought of as a set of rules that have to be checked before the normal decision making process can continue. A firewall can exist at different steps of the communication; for example a firewall can be set up on your local computer, or on your home network's router. Firewalls are a powerful tool that can explicitly refuse certain connections to be made, or the opposite. Furthermore, firewalls can actually manipulate what data is being sent, and where it is being sent. One such feature is known as Destination Network Address Translation (DNAT). Using DNAT, a firewall can change the destination that a message is being sent to.

3 - Technical Specification

3.1 - Implementation details

When an MQTT client or broker attempts to publish new data, the message is inspected by a firewall, afterwhich DNAT is used to change the destination address to the address of the proxy that has been setup. The proxy itself is both a simple TCP client and server in one. It listens for new connections on a particular address and also stores a target address it will attempt to send new information to. Both addresses can be specified by the user as command line arguments. The proxy, upon receiving a new message, attempts to establish a connection to the proxy that may be listening on behalf of the original destination. If a connection is made, and the message it has received contains new data being published, it can be encrypted, given a digital signature, and sent. When a proxy receives an encrypted message, it can verify the digital signature and perform a decryption to retrieve the original message. It can then forward the original message to the intended destination.

The development environment used Ubuntu 20.04.4 LTS as the underlying UNIX based operating system. AMQTT [Version 0.10.1][5] for Python was utilized to set up a single MQTT broker and two clients. For simplicity, one client operated as the publisher and the other client operated as the subscriber. These were hosted locally on the machine. The custom proxies that were implemented were also locally hosted. The linux utility ‘iptables’ [6] was used to set up the firewall and the relevant firewall rules.

The proxies were written in Python [Version 3.8.10] [7] , and utilized the built in socket module to communicate over TCP. I designed a base class using object oriented principles, and further specified 4 kinds of subclasses. Depending on where in the communication scheme the proxy is found, it has different duties.

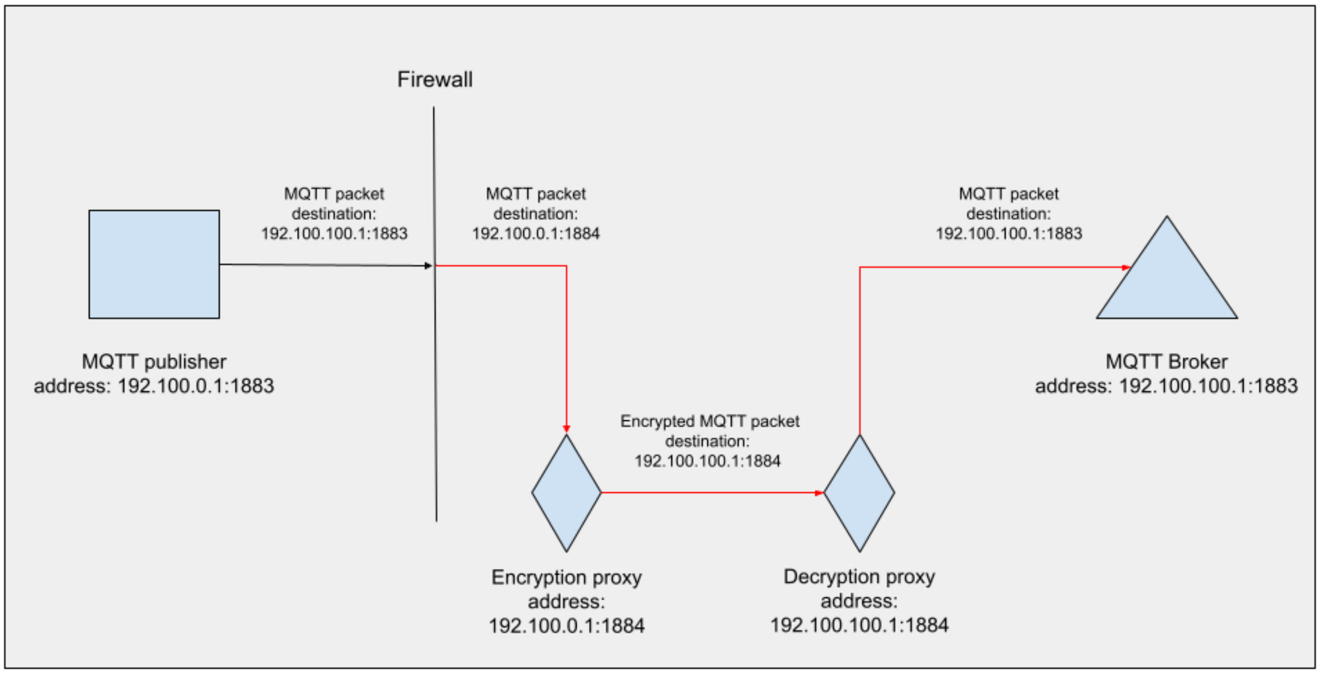

Firewall
MQTT packet destination: 192.100.100.1:1883
MQTT packet destination: 192.100.0.1:1884
MQTT packet destination: 192.100.100.1:1883
MQTT publisher address: 192.100.0.1:1883
MQTT Broker address: 192.100.100.1:1883
Encrypted MQTT packet destination: 192.100.100.1:1884
Encryption proxy address: 192.100.0.1:1884
Decryption proxy address: 192.100.100.1:1884

Figure 1.1: *Publisher to Broker half of the revised MQTT communication protocol including the proxies*

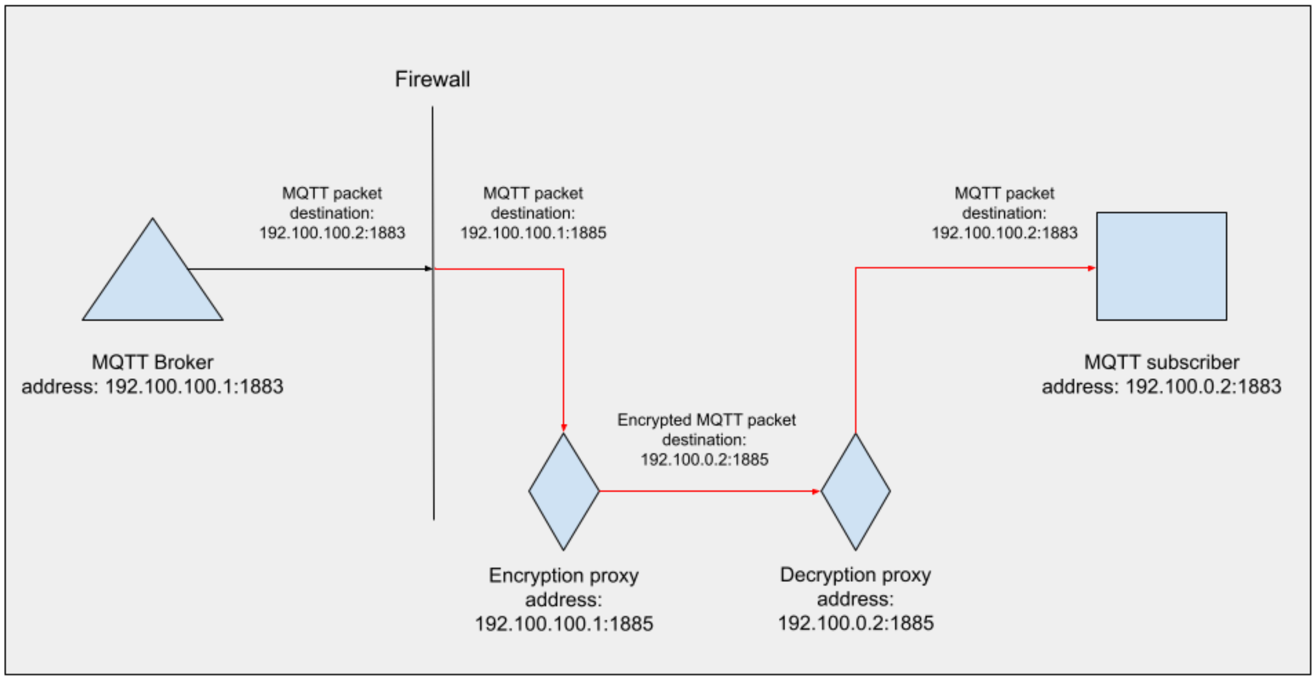

Firewall
MQTT packet destination: 192.100.100.2:1883
MQTT packet destination: 192.100.100.1:1885
MQTT packet destination: 192.100.100.2:1883
MQTT Broker address: 192.100.100.1:1883
MQTT subscriber address: 192.100.0.2:1883
Encrypted MQTT packet destination: 192.100.0.2:1885
Encryption proxy address: 192.100.100.1:1885
Decryption proxy address: 192.100.0.2:1885

Figure 1.2: *Broker to Subscriber half of the revised MQTT communication protocol including the proxies*

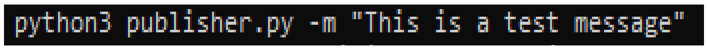
As shown in Figure 1.1, a proxy intercepting packets from a publisher must be able to connect to another proxy, and encrypt and digitally sign messages. A proxy listening for messages being sent to the broker must be able to accept a connection from another proxy, verify digital signatures and decrypt messages, as well as forward the data to the broker. For the broker to subscriber half of the communication shown in Figure 1.2, these cryptographic steps are inverted as the published data is transferred from the broker to the client acting as the subscriber.

Note that Figures 1.1 and 1.2 only show the transmission in one direction. When a broker or subscriber receives an MQTT PUBLISH packet they will respond with an acknowledgement packet of their own. This acknowledgement packet will traverse the exact same path as the original packet in the opposite direction, with only one slight difference. The encryption proxies can store the address of the sender of the original packet, corresponding to the publisher (Figure 1.1) or the broker (Figure 1.2). This means they can send the acknowledgement packet directly to the publisher or broker, and the packet does not need to be inspected or modified by the firewall.

You may also remark that Figures 1.1 and 1.2 show that the proxies are hosted on the same address as the MQTT entity they listen on behalf of, albeit with a different port number. This helps provide a distinction between the proxies and the roles they serve. In the case of the MQTT broker there is both a decryption proxy and encryption proxy listening on behalf of it. The two proxies are separate entities so they cannot share the same address and the same port number. For this reason, the proxies shown in Figure 1.1 use port 1884, and the proxies shown in Figure 1.2 use port 1885.

While Figure 1.1 and Figure 1.2 show the general architecture that communications follow, I have developed and implemented a protocol for completing the key exchange and performing the cryptographic functions. This protocol works in conjunction with the existing MQTT standard. The following example assumes one MQTT client is set up to act as the publisher, one MQTT broker is set up, one encryption proxy is set up, and one decryption proxy is set up. The MQTT client publishes its data using Quality of Service level 1.

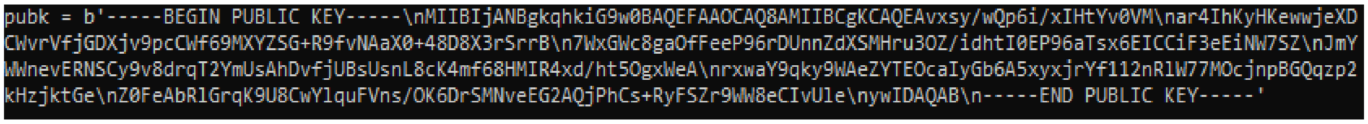
1. An MQTT client decides to publish new data it has collected. It attempts to connect to the MQTT broker by sending an MQTT CONNECT packet.



```
python3 publisher.py -m "This is a test message"
```

Figure 2: *Sample command to invoke the MQTT publisher and send the broker a message containing the data "This is a test message".*

2. The encryption proxy receives the packet sent from the publisher. The encryption proxy attempts to establish a connection to a decryption proxy by sending its public key.

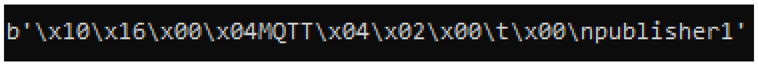


```
pubk = b'-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAvxsy/wQp6i/xIHtYv0VM\nar4IhKyHKewwjeXD
CWvrVfjGDXjv9pcCWf69MXYZSG+R9fvNAaX0+48D8X3rSrrB\n7WxGWc8gaOfFeeP96rDUnnZdXSMHru3OZ/idhtI0EP96aTsx6EICCiF3eEiNW7SZ\nJmY
WWnevERNSCy9v8drqT2YmUsAhDvfjUBsUsnL8cK4mf68HMIR4xd/ht5OgxWeA\nrxwaY9qky9WAeZYTEOcaIyGb6A5xyxjrYf112nR1W77MOcjnpBGQqzp2
kHzjktGe\nZ0FeAbR1GrqK9U8CwY1quFVns/OK6DrSMNveEG2AQjPhCs+RyFSZr9WW8eCIvUle\nywIDAQAB\n-----END PUBLIC KEY-----'
```

Figure 3: *An example of a public key. All public keys are transmitted in the PEM format.*

3. The decryption proxy receives the encryption proxy's public key. It responds with its own public key.

4. The encryption proxy receives the decryption proxies public key. It then forwards the CONNECT packet to the decryption proxy.

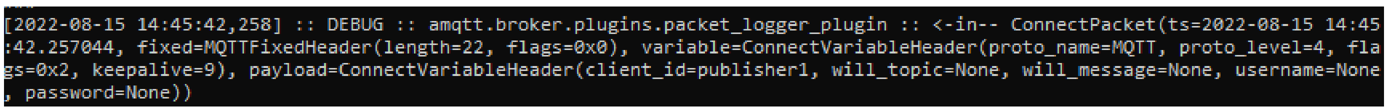


```
b'\x10\x16\x00\x04MQTT\x04\x02\x00\t\x00\npublisher1'
```

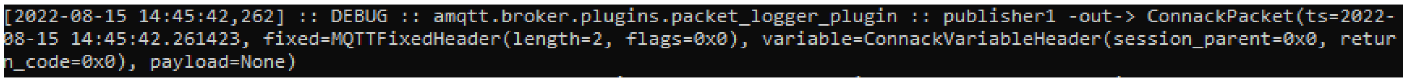
Figure 4: *The MQTT CONNECT packet created by the publisher.*

5. The decryption proxy receives the CONNECT packet. It forwards the CONNECT packet to the broker.

6. The broker receives the CONNECT packet from the decryption proxy. It responds with an MQTT CONNACK packet.



```
[2022-08-15 14:45:42,258] :: DEBUG :: amqtt.broker.plugins.packet_logger_plugin :: <-in-- ConnectPacket(ts=2022-08-15 14:45
:42.257044, fixed=MQTTFixedHeader(length=22, flags=0x0), variable=ConnectVariableHeader(proto_name=MQTT, proto_level=4, fla
gs=0x2, keepalive=9), payload=ConnectVariableHeader(client_id=publisher1, will_topic=None, will_message=None, username=None
, password=None))
```

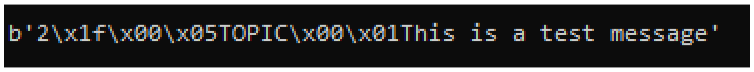


```
[2022-08-15 14:45:42,262] :: DEBUG :: amqtt.broker.plugins.packet_logger_plugin :: publisher1 -out-> ConnackPacket(ts=2022-
08-15 14:45:42.261423, fixed=MQTTFixedHeader(length=2, flags=0x0), variable=ConnackVariableHeader(session_parent=0x0, retur
n_code=0x0), payload=None)
```

Figure 5.1 & 5.2: *Sample output of the broker receiving the CONNECT and sending a CONNACK.*

7. The decryption proxy sends the broker's CONNACK packet back to the encryption proxy. The encryption proxy sends the broker's CONNACK packet back to the publisher.

8. The publisher receives the CONNACK packet and sends an MQTT PUBLISH packet, containing the new data.



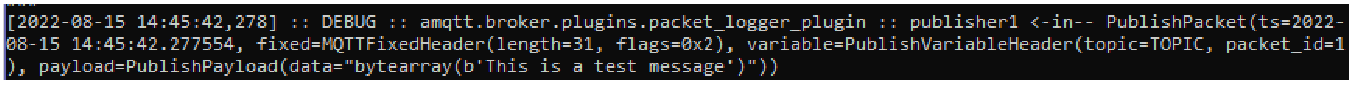
```
b'2\x1f\x00\x05TOPIC\x00\x01This is a test message'
```

Figure 6: *An MQTT PUBLISH packet.*

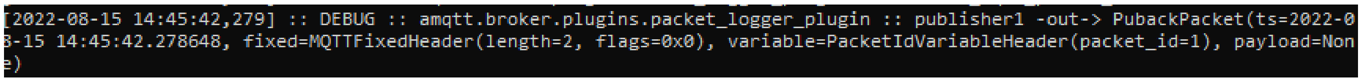
9. The encryption proxy receives the PUBLISH packet, and encrypts the entire packet. It then creates a digital signature of the ciphertext. The encryption proxy sends both the ciphertext and the digital signature as one packet to the decryption proxy.

10. The decryption proxy receives the ciphertext and digital signature. It verifies the digital signature before decrypting the ciphertext. It forwards the original PUBLISH packet to the broker.

11. The broker receives the PUBLISH packet, and responds with an MQTT PUBACK packet.



```
[2022-08-15 14:45:42,278] :: DEBUG :: amqtt.broker.plugins.packet_logger_plugin :: publisher1 <-in-- PublishPacket(ts=2022-
08-15 14:45:42.277554, fixed=MQTTFixedHeader(length=31, flags=0x2), variable=PublishVariableHeader(topic=TOPIC, packet_id=1
), payload=PublishPayload(data="bytearray(b'This is a test message')"))
```

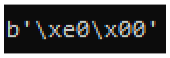


```
[2022-08-15 14:45:42,279] :: DEBUG :: amqtt.broker.plugins.packet_logger_plugin :: publisher1 -out-> PubackPacket(ts=2022-0
8-15 14:45:42.278648, fixed=MQTTFixedHeader(length=2, flags=0x0), variable=PacketIdVariableHeader(packet_id=1), payload=Non
e)
```

Figure 7.1 & 7.2: *Sample output of the broker receiving the PUBLISH and sending a PUBACK.*

12. The decryption proxy sends the PUBACK packet back to the encryption proxy. The encryption proxy sends the PUBACK packet back to the publisher.

13. The publisher receives the PUBACK packet and sends an MQTT DISCONNECT packet.

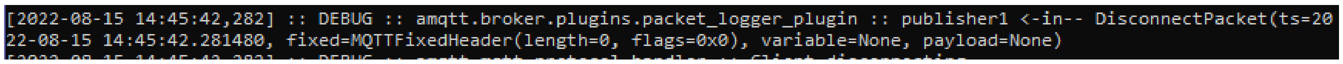


```
b'\xe0\x00'
```

Figure 8: *MQTT DISCONNECT packet.*

14. The encryption proxy receives the DISCONNECT packet and forwards it to the decryption proxy. The decryption proxy receives the DISCONNECT packet and forwards it to the broker.

15. The broker receives the DISCONNECT packet. The data has been published. The publisher, encryption proxy, decryption proxy, and broker all sever their respective connections.



```
[2022-08-15 14:45:42,282] :: DEBUG :: amqtt.broker.plugins.packet_logger_plugin :: publisher1 <-in-- DisconnectPacket(ts=20
22-08-15 14:45:42.281480, fixed=MQTTFixedHeader(length=0, flags=0x0), variable=None, payload=None)
```

Figure 9: *Sample output of the broker receiving the DISCONNECT packet.*

For the broker to subscriber half of the communication, the steps would be nearly identical. Importantly, because these proxies are meant to be implemented to existing

MQTT clients and brokers, there may not always be a proxy at the other endpoint of the communication. This means that proxies that are attempting to encrypt MQTT publish messages may not be able to connect to a proxy that can decrypt the message. In the above use case, the encryption proxy would instead establish a connection to the broker itself, and forward all packets from the publisher to the broker unencrypted. Since it is not guaranteed that the packets will be encrypted, the messages are considered ‘opportunistically’ encrypted.

3.2 - Security Analysis

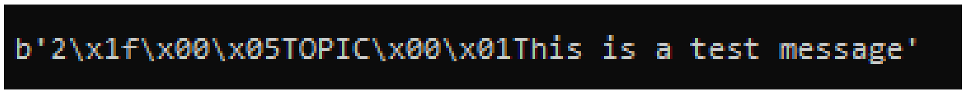
The Pycryptodome module [Version 3.15.0] [8] was used for all cryptographic functionality. I chose to use the Rivest-Shamir-Adleman (RSA) algorithm as the encryption scheme due to its use of public key infrastructure, and the fact that it can achieve the two security goals for this project [9]. RSA can be used for encryption to keep the contents of messages confidential, and also be used to create digital signatures for authentication purposes [9].

Since the proxies have to establish a connection with each other over an insecure medium, the logical conclusion was to use public key cryptography to minimize the attack surface of the proxies themselves. That being said, once the proxies are up and running, they are largely meant to be ignored. The keys are generated when the program starts, meaning that the same key will be used for as long as the proxy is running. This could be a cause for concern in the long term.

One such concern is that it would be difficult for the implementer of the proxy to determine if its private key had in fact been compromised. If the attacker chose to use the private key to publish malicious or falsified data it would be more likely to be noticed. However, if the attacker chose to only decrypt and monitor the data being published, it would be incredibly difficult for the implementer to detect the issue.

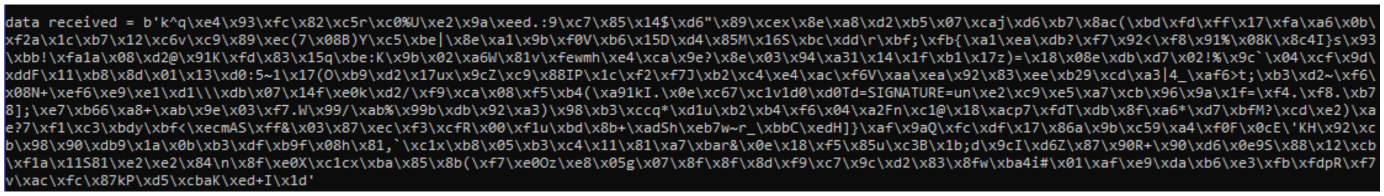
Assuming the attacker has the ability to monitor the entire communications channel, it would be trivial for them to retrieve one or both of the proxies' public keys during the course of its uptime. The keys themselves are 2048 bits long as is recommended by the authors of Pycryptodome and the cryptography community in general [10]. This length should be sufficient to protect against a brute force attack to retrieve a corresponding private key, however as computing power gets greater, that will change. In a few years, or sooner depending on the overall security of the RSA scheme, it may be viable for attackers to retrieve a private key from the corresponding public key with a length of 2048 bits. If that were to happen, it would compromise the entire security of this project and its goals. Section 4.1 covers what changes could be made to ensure this does not happen.

The use of RSA also impacts some of the other functionality of the proxies. For instance, the size of the message being encrypted cannot exceed the key length. This means that for messages that are larger than 2048 bits, they need to be divided into blocks that get sent separately. The proxy receiving such a message also needs to be able to reconstruct the original message from the separate blocks. When the message gets encrypted and digitally signed, the corresponding ciphertext and signature tag are also the length of the keysize. This means that any message smaller than 2048 bits will be transformed and sent as a message that is twice the size of the key length. 4096 bits of overhead for every MQTT PUBLISH packet could be a significant amount of extra work depending on the computing and network conditions.



```
b'2\x1f\x00\x05TOPIC\x00\x01This is a test message'
```

Figure 10: *Sample output from the encryption proxy. Above is the data that gets sent from the publisher to the encryption proxy.*



```
data received = b'k^q\xe4\x93\xfc\x82\xc5r\xc0%U\xe2\x9a\xeed.:9\xc7\x85\x14$\xd6"\x89\xcex\x8e\xa8\xd2\xb5\x07\xcaj\xd6\xb7\x8ac(\xbd\xfd\xff\x17\xfa\xa6\x0b\
xf2a\x1c\xb7\x12\xc6v\xc9\x89\xec(7\x08B)Y\xc5\xbe|\x8e\xa1\x9b\xf0V\xb6\x15D\xd4\x85M\x16S\xbc\xdd\r\xbf;\xfb{\xa1\xea\xdb?\xf7\x92<\xf8\x91%\x08K\x8c4I}s\x93
\xbb!\xfa1a\x08\xd2@\x91K\xfd\x83\x15q\xbe:K\x9b\x02\xa6W\x81v\xfewmh\xe4\xca\x9e?\x8e\x03\x94\xa31\x14\x1f\xb1\x17z)=\x18\x08e\xdb\xd7\x02!%\x9c`\x04\xcf\x9d\
xddF\x11\xb8\x8d\x01\x13\xd0:5~1\x17(O\xb9\xd2\x17ux\x9cZ\xc9\x88IP\x1c\xf2\xf7J\xb2\xc4\xe4\xac\xf6V\xaa\xea\x92\x83\xee\xb29\xcd\xa3|4_\xaf6>t;\xb3\xd2~\xf6\
x08N+\xef6\xe9\xe1\xd1\\\xdb\x07\x14f\xe0k\xd2/\xf9\xca\x08\xf5\xb4(\xa91kI.\x0e\xc67\xc1v1d0\xd0Td=SIGNATURE=un\xe2\xc9\xe5\xa7\xcb\x96\x9a\x1f=\xf4.\xf8.\xb7
8];\xe7\xb66\xa8+\xab\x9e\x03\xf7.W\x99/\xab%\x99b\xdb\x92\xa3)\x98\xb3\xccq*\xd1u\xb2\xb4\xf6\x04\xa2Fn\xc1@\x18\xacp7\xfdT\xdb\x8f\xa6*\xd7\xbfM?\xcd\xe2)\xa
e?7\xf1\xc3\xbdy\xbf<\xecmAS\xff&\x03\x87\xec\xf3\xcfR\x00\xf1u\xbd\x8b+\xadSh\xeb7w~r_\xbbC\xedH]}\xaf\x9aQ\xfc\xdf\x17\x86a\x9b\xc59\xa4\xf0F\x0cE\'KH\x92\xc
b\x98\x90\xdb9\x1a\x0b\xb3\xdf\xb9f\x08h\x81,`\xc1x\xb8\x05\xb3\xc4\x11\x81\xa7\xbar&\x0e\x18\xf5\x85u\xc3B\x1b;d\x9cI\xd6Z\x87\x90R+\x90\xd6\x0e9S\x88\x12\xcb
\xf1a\x11S81\xe2\xe2\x84\n\x8f\xe0X\xc1cx\xba\x85\x8b(\xf7\xe0Oz\xe8\x05g\x07\x8f\x8f\x8d\xf9\xc7\x9c\xd2\x83\x8fw\xba4i#\x01\xaf\xe9\xda\xb6\xe3\xfb\xfdpR\xf7
v\xac\xfc\x87kP\xd5\xcbaK\xed+I\x1d'
```

Figure 11: *Sample output from the decryption proxy. Above is the data that gets sent from the encryption proxy to the decryption proxy.*

In Figure 10, you'll note that in the middle of the random bits there is a plaintext string, "=SIGNATURE=". When the encryption takes place, and the digital signature is created, I chose to put them one after the other in one large message. I used the string as a delimiter so that the decryption proxy is able to easily split the message into its two components, the ciphertext and the signature. While it was indeed convenient on my end, it should be noted that it would be just as convenient for any attackers monitoring the communication channel. Best practices suggest that this string be omitted entirely should this project receive any future work.

4 - Future Work

Due to time constraints, there were several areas in which the work done was simplified to ensure this proof of concept could be completed. If further research and more time were to be spent on this project, I have several suggestions that should be taken into consideration.

4.1 Cryptographic functions

While RSA should be considered a time tested tool, having been first described in 1978 and widely used for nearly as long [9], there have been many advancements in cryptographic technologies in the past 40 years.

The benefits of using public key cryptography and corresponding asymmetric ciphers are still important to consider, namely that the key exchange between proxies happens over an insecure communication channel. That being said, using the Diffie-Hellman key exchange protocol [11], it is possible for the two proxies to securely establish a shared secret key that can be used with a symmetric cipher like the Advanced Encryption Scheme (AES) [12]. This process would create a new shared secret key each time the proxies established a connection with one another, and would solve the problem with using long lived keys. The tradeoff is that more computing power would be required to generate a new key each time a connection is established. That being said, in general symmetric ciphers are able to encrypt messages much faster than asymmetric cipher alternatives. Reducing the time spent encrypting and decrypting data may be desirable in settings where low latency is required.

Another benefit of using a symmetric cipher would be the ability to use block ciphers, a specific kind of symmetric cipher. Block ciphers work by dividing the input into chunks, called blocks, and encrypting blocks of the input one at a time. Currently this approach has to be manually done when trying to encrypt messages longer than the RSA key length. Using a block cipher that is designed to encrypt blocks of input at a time would resolve the need to worry about input size exceeding key length.

One drawback of using a symmetric cipher for encryption and decryption is that symmetric ciphers do not support creating and verifying digital signatures for authentication purposes. To achieve both security goals of providing confidentiality as well as authentication, a hybrid approach would need to be adopted. A symmetric cipher such as AES could still be used for encryption and decryption, however a separate asymmetric cipher system such as the Edwards-curve Digital Signature Algorithm (EdDSA) [13] would be implemented alongside AES. EdDSA is part of the latest research in asymmetric cryptography algorithms having first been published in 2011 and with there being two efforts to standardize the algorithm in recent time, one by the Internet Research Task Force in 2017 [13], and one by the United States National Institute of Standards and Technology in 2019 [14]. Notable technologies that use

EdDSA include OpenSSH [15], OpenSSL [16], and even the Apple Watch and Iphone [17]. Not only does EdDSA create and verify digital signatures faster than RSA and other asymmetric ciphers, it maintains a comparable level of security.

Between the Diffie-Hellman key exchange and the use of public key infrastructure, this hybrid approach would require more transmissions between proxies in order to establish a connection than using either symmetric or asymmetric cryptography alone. That being said, the hybrid approach would greatly bolster the security of the proxies and achieve both security goals. The hybrid approach also provides access to all the functionality and advantages of both symmetric and asymmetric ciphers.

4.2 Scalability

The current implementation of the proxies has a serious limitation when it comes to scalability. In terms of IoT devices using the MQTT protocol, there can be up to millions of devices connecting to each other and publishing or receiving data. The current design of the proxies can only handle one connection at a time. This means that a broker would need as many decryption proxies as there are publishers, and as many encryption proxies as there are subscribers. Deploying thousands of each proxy would not only be costly, it would be a waste of computing power. Furthermore, in the development environment the role of publisher and subscriber were completely separate MQTT clients. However, as previously noted the MQTT protocol supports MQTT clients acting as both a publisher and a subscriber at the same time. This indicates that in actuality a single proxy needs to be able to perform both the encryption and decryption steps.

Both these problems could be solved by developing the proxies using concurrent programming techniques. The Asyncio module for Python acts as a high level event scheduler for functions and coroutines, and it was also designed with networking in mind, as it supports opening and manipulating network sockets. This means that asyncio could be used to facilitate managing multiple TCP connections at once, and it could even manage the separate tasks of encrypting, decrypting, creating digital signatures, and verifying digital signatures. Developing the proxies using asyncio or similar techniques would hugely impact the ability for the proxies to scale with the size of IoT systems.

4.3 Testing

Upon reviewing the use case shown in section 3, it should be clear that there are many steps during which an error could be disastrous and hinder the successful completion of subsequent steps. For example, if the decryption proxy cannot successfully verify the digital signature, it has to assume the communication has been compromised, and it should not decrypt the ciphertext. At that point, it has no message to deliver to the broker, and no response to send back towards the publisher.

While I made a sincere effort to program the proxies to be able to appropriately deal with these kinds of errors, it is an exceptionally difficult task. It is always beneficial to perform exhaustive testing to ensure unexpected errors do not cause the program to halt execution. A practical implementation of these proxies would need to be able to at least pass the following tests and others:

1. Publishing messages longer than the RSA key length.
2. Publishing messages that contain non standard ASCII characters, such as messages with text in another language.
3. Publishing numerical data.
4. Handling bad user input (for example validating the format of network addresses specified by the user).
5. Handling an error when establishing a connection with another proxy, an MQTT client, or an MQTT broker.
6. Handling an error during the key exchange between two proxies.
7. Handling any errors that occur in the cryptographic functions (encrypting / decrypting, creating / verifying digital signatures).
8. Handling any network errors (for example losing internet connectivity in the middle of a communication).
9. Handle the case where an entity that is not a proxy and not an MQTT client / broker attempts to establish a connection to a proxy.

Furthermore, it would be vital to conduct performance tests to ensure the proxies can keep up with the demand of the deployment environment. It is important that this list of test cases not be considered comprehensive, as different problems can arise depending on exact implementation details.

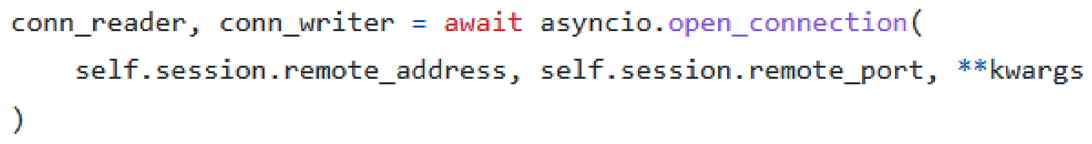
5 - Conclusion

To summarize, this project achieved its goal of enhancing the security of existing MQTT deployments by adding in supplementary endpoints to the communication protocol. The custom proxies were able to act on behalf of MQTT clients and brokers, and utilize the RSA encryption scheme to provide confidentiality of published data as well as authentication. For a practical application of these proxies, suggestions were made for using alternative encryption schemes to mitigate the security concerns of using long lived keys and to better suit the needs of the program. The use of asyncio and similar concurrent programming techniques were discussed in order for the proxies to accommodate the large scale of IoT systems. A broad set of test cases was suggested to ensure that the proxies would be able to continue properly functioning despite the numerous problems that can arise during program execution. This project represents a promising improvement of the security of IoT deployments that use MQTT and that do not receive adequate post deployment support in the form of software or hardware updates.

Appendix A - Changes to AMQTT

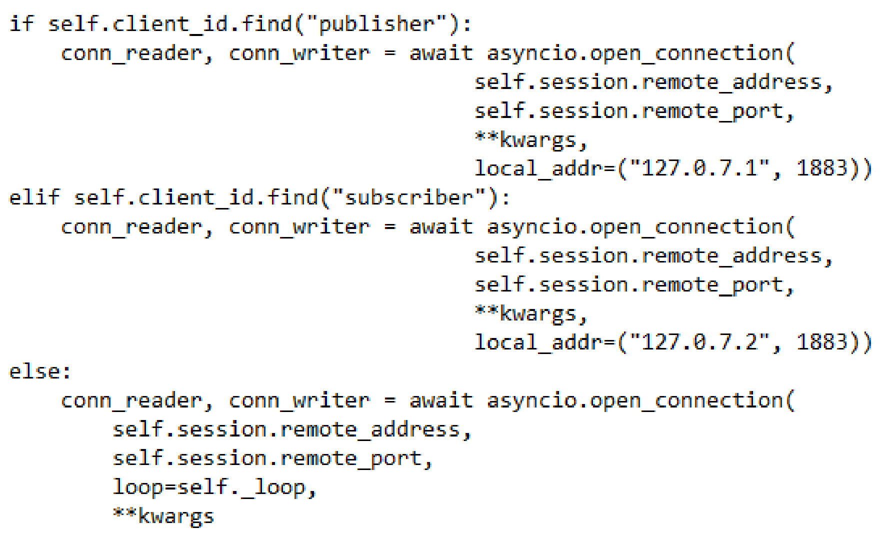
In order to complete the current implementation of the proxies, I had to make some changes to the library I used for the MQTT client and broker implementations. Unfortunately AMQTT [Version 0.10.1] does not allow the user to specify a network address for the MQTT clients to be hosted on. This functionality was required to facilitate creating the firewall rules that intercept the packets of MQTT clients. Unless this problem is rectified by changing the code of AMQTT itself, the current implementation of the proxies will NOT work.

The changes were made at line 448 in the client.py file.



```
conn_reader, conn_writer = await asyncio.open_connection(
    self.session.remote_address, self.session.remote_port, **kwargs
)
```

Figure 12.1: *Code at line 448 in the AMQTT client.py file before changes*



```
if self.client_id.find("publisher"):
    conn_reader, conn_writer = await asyncio.open_connection(
                                self.session.remote_address,
                                self.session.remote_port,
                                **kwargs,
                                local_addr=("127.0.7.1", 1883))
elif self.client_id.find("subscriber"):
    conn_reader, conn_writer = await asyncio.open_connection(
                                self.session.remote_address,
                                self.session.remote_port,
                                **kwargs,
                                local_addr=("127.0.7.2", 1883))
else:
    conn_reader, conn_writer = await asyncio.open_connection(
        self.session.remote_address,
        self.session.remote_port,
        loop=self._loop,
        **kwargs
```

Figure 12.2: *Modified code at line 448 of the AMQTT client.py file*

Note that the addresses specified in the modified code (127.0.7.1 and 127.0.7.2) could be changed as needed.

Due to time constraints I could not follow the proper process of getting this feature implemented in the AMQTT project. Future work would be to fork the AMQTT project, implement the feature in the AMQTT code, and create a pull request.

An alternative work around could be to specify additional firewall rules to accept all packets being sent by the encryption and decryption proxy to the broker, and intercepting all other packets being sent to the broker. That being said, special care has to be taken with the order and specifics of firewall rules. This approach may end up creating other issues down the line.

6 - References

1. MQTT Version 3.1.1. *Edited by Andrew Banks and Rahul Gupta. 29 October 2014*. OASIS Standard.
   http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
   Latest version:
   http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html
   [Online, accessed 2022/08/17)
2. MQTT Use Cases. *MQTT.org 2022.*
   https://mqtt.org/use-cases/
   [Online, accessed 2022/08/17]
3. MQTT Software. *MQTT.org, 2022.*
   https://mqtt.org/software/
   [Online, accessed 2022/08/17]
4. SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM. *Vington Cerf, Yogen Dalal, Carl Sunshine 1974.*
   https://datatracker.ietf.org/doc/html/rfc675
   [Online, accessed 2022/08/17]
5. aMQTT: Community Driven LTS for HBMQTT. *Github user Yakifo & other contributors, 2022.*
   https://github.com/Yakifo/amqtt

[Online, accessed 2022/08/17]

6 - References (cont.)

6. iptables. *Netfilter, 2022*.

   https://git.netfilter.org/iptables/

[Online, accessed 2022/08/17]

7. Python version 3.8.10. *Python Software Foundation, 2021.*

   https://www.python.org/downloads/release/python-3810/

[Online, accessed 2022/08/17]

8. Pycryptodome version 3.15.0. *Pycryptodome.org, 2022.*

   https://www.pycryptodome.org/src/introduction

[Online, accessed 2022/08/17]

9. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.

   R.L Rivest, A. Shamir, L. Adleman, 1978

   https://people.csail.mit.edu/rivest/Rsapaper.pdf

[Online, accessed 2022/08/17]

10. Pycryptodome API Documentation RSA Public Keys. *Pycryptodome.org,*

    2022.

    https://www.pycryptodome.org/src/public_key/rsa

[Online, accessed 2022/08/17]

11. Secure Communications over Insecure Channels. *Ralph C. Merkle, 1978.*

    https://dl.acm.org/doi/pdf/10.1145/359460.359473

[Online, accessed 2022/08/17]

6 - References (cont.)

12. Announcing the Advanced Encryption Standard. *United States National Institute of Standards and Technology, 2001.* https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf

    [Online, accessed 2022/08/17]

13. Edwards-Curve Digital Signature Algorithm (EdDSA). Internet Research Task Force, 2017.

    https://datatracker.ietf.org/doc/html/rfc8032

    [Online, accessed 2022/08/17]

14. Digital Signature Standard. United States National Institute of Standards and Technology, 2019.

    https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5-draft.pdf

    [Online, accessed 2022/08/17]

15. Changes since OpenSSH 6.4. OpenBSD project, 2014.

    http://www.openssh.com/txt/release-6.5

    [Online, accessed 2022/08/17]

16. OpenSSL CHANGES. OpenSSL Project, 2019.

    https://www.openssl.org/news/cl111.txt

    [Online, accessed 2022/08/17]

6 - References (cont.)

17. System security for WatchOS. Apple Inc., 2022.

    https://support.apple.com/en-ca/guide/security/secc7d85209d/web

    [Online, accessed 2022/08/17]