Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 8261746, 11 pages
https://doi.org/10.1155/2018/8261746

Research Article

MQTT Security: A Novel Fuzzing Approach

Santiago Hernández Ramos,[1] **M. Teresa Villalba**,[2] **and Raquel Lacuesta**[3]

[1]*Telefónica Digital, Madrid, Spain*
[2]*Universidad Europea de Madrid, Madrid, Spain*
[3]*Universidad de Zaragoza, Teruel, Spain*

Correspondence should be addressed to Santiago Hernández Ramos; santiago.hernandezramos@telefonica.com

Received 30 September 2017; Accepted 10 January 2018; Published 26 February 2018

Academic Editor: Syed H. Ahmed

The Internet of Things is a concept that is increasingly present in our lives. The emergence of intelligent devices has led to a paradigm shift in the way technology interacts with the environment, leading society to a smarter planet. Consequently, new advanced telemetry approaches appear to connect all kinds of devices with each other, with companies, or with other networks, such as the Internet. On the road to an increasingly interconnected world, where critical devices rely on communication networks to provide an essential service, there arises the need to ensure the security and reliability of these protocols and applications. In this paper, we discuss a security-based approach for MQTT (Message Queue Telemetry Transport), which stands out as a very lightweight and widely used messaging and information exchange protocol for IoT (Internet of Things) devices throughout the world. To that end, we propose the creation of a framework that allows for performing a novel, template-based fuzzing technique on the MQTT protocol. The first experimental results showed that performance of the fuzzing technique presented here makes it a good candidate for use in network architectures with low processing power sensors, such as Smart Cities. In addition, the use of this fuzzer in widely used applications that implement MQTT has led to the discovery of several new security flaws not hitherto reported, demonstrating its usefulness as a tool for finding security vulnerabilities.

1. Introduction

Today, cities face complex challenges, including sustainable urban development, reduction of pollution and energy consumption, and safety [1]. IoT (Internet of Things) is considered the core technology for building Smart Cities, as it is based on the concept "everything can be connected to the Internet." The development of cheaper sensors and other devices, as well as the adoption of cloud services, is providing new opportunities to develop new services for improving quality of life in cities. As cities grow, interest in exploring new IoT technologies increases. Some examples of how IoT technologies can support the building of Smart Cities are as follows:

(i) Smart street lights with sensors for detecting cars' movement and sending data to control when to switch them on or off to save energy

(ii) Reducing water consumption in parks

(iii) Health personnel attending to citizens in emergency situations with access to their medical records in real time [2–4]

(iv) The consumption and regulation of electricity controlled by smart meters and sensors that send data in real time [3, 5, 6]

Due to IoT, technology-enabled devices located in different places communicating with each other and generating large volumes of data, information becomes more difficult to protect [7]. Compromised availability, integrity, or confidentiality of these data can have an adverse and direct effect on people's lives [8]. Consequently, there is a need to implement mechanisms that verify the security of IoT devices, the network protocol they use to exchange information, and the applications developed for them.

In this paper, we propose a framework to improve the security of applications implementing the protocol MQTT

(Message Queue Telemetry Transport), a widely used protocol for sharing data exchanged between IoT devices. MQTT is an extremely simple and lightweight messaging protocol, with a publish/subscribe architecture, designed to be straightforward to deploy, and capable of supporting thousands of clients with a single server. In addition, MQTT provides reliability and efficiency in adverse conditions. All these features make this protocol one of the most used protocols for the communication between smart devices, with a high number of applications based on it, increasing rapidly over time [9, 10].

To test the security of the applications that implement MQTT, we have created a framework based on a verification technique called fuzzing. Fuzzing is a testing technique for finding vulnerabilities in software applications [11] by sending unexpected input data to target systems and then monitoring the results. Typically, it consists of an automatic or semiautomatic process, which comprises the sending and repeated manipulation of data to the system under study. All fuzzers can be classified into two broad categories [12]: mutation-based and generation-based fuzzers. Mutation-based fuzzers apply mutations on existing data samples to create the test space, while generation-based fuzzers create test cases from scratch by modelling the target protocol or file format. As generation-based approaches are more complex and time-consuming, we focus on mutation-based fuzzer approaches along with a novel fuzzing technique based on templates. The aim of this technique is to reduce the effort and increase the productivity of users when performing security verification of the applications that implement the MQTT protocol. This new technique allows completely automated generation of a template with the fields we want to test for each network packet. It also enables definition of other specifications, such as the fields for which we want to filter the traffic or the values that we want to insert or send by default.

The rest of the paper is organized as follows. In Section 2, we briefly explain the basic concepts needed to understand the presented work. Section 3 discusses the related work regarding MQTT protocol security and the modern fuzzing approaches. Section 4 deals with the basic elements of the framework and the implementation of the concepts and methods discussed above. Section 5 then introduces the architecture of the fuzzer, and Section 6 shows the results of the experimentation phase. Finally, the conclusions are presented.

2. Background and Motivation

2.1. Message Queue Telemetry Transport. MQTT uses a publish/subscribe messaging pattern that enables a loose coupling between the information provider, called the *publisher*, and consumers of the information, called *subscribers*. This is achieved by introducing a message broker between the publishers and the subscribers.

Compared with the traditional point-to-point pattern, the advantage of this model is that the publishing device or the application does not need to know anything about the subscribing one, and vice versa. We can distinguish three MQTT essential concepts that will remain present throughout the development of the paper.

Table 1: MQTT fixed header.

Bit->	7	6	5	4	3	2	1	0
Byte 1	Msg type				DUP	QoS		Retain
Byte 2	Remaining length							

Table 2: Some solutions that use MQTT.

Brokers	Clients	Smart home
Mosquitto	CocoaMQTT	Homegear
ActiveMQ	emqttc	Domoticz
hbmqtt	mqtt-client	Lelylan
HiveMQ	M2Mqtt	cul2mqtt
Moquette	mqtt_cpp	aqara-mqtt
Mosca	mqttex	Home.Pi
VerneMQ	Paho	Home Assistant
hrotti	rumqtt	pimatic
SurgeMQ	hbmqtt	FHEM

(i) Topics. The *publishers* are responsible for cataloguing the messages they send in topics. A topic defines the content of a message or a category in which the message can be classified. Topics are important because while in the point-to-point protocols messages are sent to a specific address, in a publish/subscribe pattern, messages are distributed based on the selected topics by the subscriber. By subscribing to a particular topic, the subscriber will receive all messages sent with that topic by any publisher.

(ii) Client. MQTT clients connect to a broker to exchange messages. They must subscribe to topics and can publish information to other entities connected to the same broker by providing a topic.

(iii) Broker. MQTT brokers are servers acting as intermediaries for the messages. MQTT protocol messages' format consists of three parts: a fixed header, shown in Table 1; a variable header; and a payload. Fuzzers consider the fields and positions of the header for inserting data to perform the fuzzing process.

MQTT is one of the most used protocols worldwide as shown in Table 2.

2.2. Fuzzing Processes. The phases of a fuzzing process are highly variable and depend on many factors, such as the application being tested or the programmer's experience [13]. However, there is a set of basic steps that are always followed, regardless of the approach or application being analysed. In the development of the tool that will be presented below, the following phases have been considered:

(i) Identifying an objective: the first step in every fuzzing process consists of identifying the target which could be an application, a protocol, or even the function of a specific library. The target here is both the MQTT protocol and the applications implementing the protocol.

(ii) Identifying the entry points: almost all exploitable vulnerabilities are caused by applications that accept user values processed without being properly checked in advance. Enumerating the input vectors is one of the crucial aspects for the fuzzing process to succeed. In the end, anything that can be sent from the client to the target system should be considered as an input vector. This includes headers, filenames, environment variables, and registry keys.

(iii) Generating the fuzzing data: once the input vector has been identified, we must generate appropriate data to perform the fuzzing process. A high degree of automation generating the test cases is important, as numerous cases must be generated.

(iv) Executing the test cases: this step is closely linked to the previous one and consists of the process of sending the data packets to the target system. As in the previous stage, process automation is essential.

(v) Exception monitoring: a vital part of the fuzzing process is monitoring exceptions. Causing the crash of the target system after sending numerous data packets has no benefit if the particular packet that caused the error cannot be determined. Monitoring can take many forms and is closely linked to the target system and the type of fuzzing being performed.

3. Related Work

3.1. Security in MQTT Protocol. Although research on MQTT security is still scant, some incipient work has been presented about its security issues. Almost all security problems that arise are related to the state in which the protocol works by default. Because MQTT is a simple protocol designed for devices with low processing power, by default, the protocol tries to minimize the processing needed to exchange messages, which means that serious security problems arise. Most of these shortcomings can be solved with an adequate protocol configuration. The following are some of the most common security issues that can be solved through proper protocol configuration [14, 15]:

(i) Lack of authentication: the MQTT protocol does not provide a secure authentication mechanism by default, which can lead to spoofing the identity of some of the participants in the communication or the sending of unauthorized data. This problem can be easily solved by configuring the protocol features adequately. When it comes to authentication, the protocol itself provides username and password fields in the CONNECT message enabling clients to send a username and password when connecting to an MQTT broker.

(ii) Lack of authorization: MQTT clients, after connecting to a broker, can publish messages or subscribe to topics. Each authenticated client can publish and subscribe to all kinds of topics even without proper authorization. This may be a significant problem, because the protocol itself does not provide any mechanism to carry it out and therefore the responsibility lies with the broker. In spite of this, it can be easily solved through the implementation of topic permissions on the broker side [16].

(iii) Lack of confidentiality: MQTT relies on TCP as transport protocol, which means that by default the connection does not use an encrypted communication. This means that packets can be spied on by an attacker listening on the same network. To avoid this, almost all MQTT brokers allow encryption of the whole MQTT communication, using TLS instead of plain TCP.

(iv) Lack of integrity: when MQTT systems have untrusted clients or unidentified MQTT clients have access to the MQTT broker and topics, data integrity of sent messages should be checked, especially when TLS is not used. MQTT supports three mechanisms to provide integrity to exchanged packets: Checksum, MAC, and Digital Signatures.

Other approaches have been also presented by some authors. Moreover, some research has tried to deal with the general problems of IP-based protocols used by IoT devices, one of which is MQTT. In these cases, the authors focus on the security of this type of device as part of a broader spectrum, treating the layers of protection that can wrap around the TCP/IP protocol and the security architectures and models that best fit IoT networks [17]. In addition to confidentiality, other security features have been addressed; [18] deals with the problem that smart devices have when they do not have enough processing capacity to use asymmetric encryption algorithms to perform authentication tasks, and it proposes a new authentication approach based on operations that consume few resources, such as hash functions or OR operations. Some other interesting approaches focus on how to force compliance with the optional security features that the MQTT protocol can implement. *SecKit* is a model-based security toolkit that tries to force the use of a series of security policies, so that the protocol implements some protection measures that are not found in its default implementation [19]. There is also research that continues to focus on the security limitations that this protocol poses by design and proposes frameworks to improve their security in the transporting of information between the parties involved in the connection by adding extra layers such as SSL/TLS [20, 21].

The considerations that appear at the beginning of this section show that the protocol security flaws are related to its operation and, in particular, to the way in which it exchanges information. This paper aims to contribute to the improvement of the security of the MQTT protocol regarding the verification of the devices that implement it. When applications that implement the MQTT protocol process a package incorrectly, serious security failures such as denial of service or remote execution of arbitrary code can occur [22, 23], even when the security measures mentioned above are met. Evaluating applications that implement such a protocol in the different parts of the connection (client and broker) to verify their behaviour when receiving incorrect or unexpected data can help to avoid certain serious security breaches.

3.2. Modern Fuzzing. As has been said in previous sections, there are many types of fuzzing and many ways to perform this technique, including IoT fuzzing as the one presented for the Modbus protocol [24] by the *IoT Systems book* [25]. However, in general, as Aitel stated in his paper [26], modern fuzzing tries to solve three major problems with respect to traditional fuzzing.

(i) If the network protocol is defined by an API with which the client and server are implemented, it is very likely that these predefined functions will make certain checks on the data that is sent and will consequently have an indirect influence on the fuzzing process.

(ii) Even given complete knowledge of the protocol, creating a client for a protocol can be a considerable undertaking, and that client is rarely portable to other protocols, even those of a similar nature.

(iii) Often, testers have only limited knowledge of the protocols under attack or of the ways the protocol may break.

To solve this, modern fuzzing tools, like Boofuzz [27], SNOOZE [28], and KiF [29] or [30], among others, propose a block-based approach, which consists of decomposing the protocols into length fields and data fields and providing the user with a framework for creating such tools without having to worry about the control fields (such as lengths or checksums) of the lower layers. This is the approach that most current frameworks use, and it has a very simple foundation. If, when a network packet consisting of several layers is available, with the upper layers being the application ones and the lower layers being the physical layers, we would like to perform fuzzing testing on one of the application layers, it would not be enough to enter the testing value and send the message, since the underlying layers may contain control fields, which if not updated correctly would lead to rejection of the packet upon reaching the server, before the value inserted was processed. To solve this problem, some structures called blocks were proposed. In them, a series of variables is grouped previously defined by the framework, which occupy a specified size. A set of these variables form a block, and the blocks can be opened and closed as follows [31]:

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata");
s_binary("01020304");
s_block_end("somepacketdata");
```

What is achieved with this is that the control fields of the lower layers are recalculated automatically, once all the blocks have been closed and, consequently, the user does not have to worry about processing them.

If we focus on the particular topic of fuzzing the MQTT protocol, very few references or tools can be found about it. The only public tool of which the authors have evidence is mqtt-fuzz [32], whose main utility is to verify the protocol in a fast and traditional way, without providing too much complexity. In addition, other methods of fuzzing or formal testing have been presented for the MQTT protocol, albeit with a different aim. This is the case with [33], in which the authors discuss formal methods of network protocol verification through finite-state machines and labelled transition systems. The focus is on demonstrating how most of the implementations of MQTT do not meet the standard. CG-Fuzzing is a fuzzy algorithm for ZigBee, with a focus on generating an efficient number of test cases.

3.3. Proxy Fuzzing. Proxy fuzzing is a widespread and a barely studied technique, resulting from some of its current limitations. Some work has been carried out in relation to this technique, such as ZAP Proxy [34], Burp Proxy [35], ProxyFuzz [36]. What all this work has in common is that the fuzzer must be placed in the middle of the connection, between the client and the server, to serve as a relay agent. To effectively accomplish this task, both the client and the server must be configured manually or automatically by some IP trickery, for example, ARP spoofing. This allows the client and the server to look for one another at the address of the proxy, so the client sees the proxy as the server, and vice versa. This fuzzing method provides several improvements over previous processes, such as simplicity of use. However, it is a difficult technique to implement, which is why the tools for implementing this technique are barely known and introduce extremely basic fuzzing techniques. This technique has led to a patent [37].

4. Methods

In this section, we will show how we have implemented the methods mentioned in the previous sections, along with other new approaches to creating a fuzzing tool for the MQTT protocol.

4.1. Fuzzing MQTT Messages. The process of fuzzing a protocol or the applications that implement it entail knowing in some way the specification of said protocol, either through its public documentation or by reverse engineering techniques. Once we know its specification and we can interpret the bytes of a package, we must select the packages and fields that are of interest for inserting information with the intention of verifying that the application that processes them does so correctly.

In the case of MQTT, no reverse engineering process is required, since its specification is public [38]. Therefore, we need only look in the specification documents for the type of

Table 3: Types of MQTT messages.

Packet	Description
CONNECT	Connect to the server
CONNACK	Ack of connect msg
PUBLISH	Publish a topic
PUBACK	Ack of publish msg
PUBREC	Publication received
PUBREL	Publication sent
PUBCOMP	Publication completed
SUBSCRIBE	Client subscription
SUBACK	Ack of subscribe msg
UNSUBSCRIBE	Unsubscribe petition
UNSUBACK	Ack of unsubscribe msg

Table 4: Publish packet variable header nonnormative example.

Byte position	Description
Topic name	
Byte 1	Length MSB (0)
Byte 2	Length LSB (3)
Byte 3	"a" (0x61)
Byte 4	"/" (0x2F)
Byte 5	"b" (0x62)
Packet identifier	
Byte 6	Packet identifier MSB (0)
Byte 7	Packet identifier LSB (10)

packages which are exchanged and the fields that are in their variable header and payload (Table 3).

If we look a little more in depth at the type of packets exchanged by the protocol, we quickly realize that the message *PUBLISH* is likely to be the one in which most information is transmitted and therefore the one in which the most processing is performed by the applications that implement the protocol. Once we have identified this type of packages (PUBLISH, CONNECT, SUBSCRIBE, etc.), we study their variable header (Table 4) to select the type of fields and the fields' positions in bytes, into which the test cases will be inserted to carry out the fuzzing process. Once the fields where the test cases are inserted have been selected, we look for the control fields, which will be recalculated once the test case is inserted. Finally, we look for a field that unequivocally identifies the package in order to be able to filter it "on the fly."

4.2. Advance Proxy Fuzzing. To apply the fuzzing process, we use the fuzzing proxy technique explained in previous sections.

As we have already stated, this technique is not very widespread, and the tools that perform it are outmoded and present a great degree of deficiency compared to modern techniques. However, if we study in depth the advantages of applying this technique, we can verify that it allows us to solve several of the deficiencies presented by modern fuzzing. These deficiencies are presented below.

4.2.1. Fuzzing Different Components of the Connection. In general, the current fuzzing tools are only designed to verify some points of the connection. This means that if we use a tool to test a particular server, it cannot normally be used to repeat the process on a client, or at least not without investing a great deal of effort in modifying the structure of the framework. The solution to this problem marks one of the main characteristics of the tool that is being presented, since the objective to be achieved is to reduce the effort on the part of the user for verification of the security of applications that implement the MQTT protocol. With the proxy technique, because the fuzzer is in the middle of the communication, the main objective is the packages that circulate between the different components. Thus, the fuzzing tool is not built for a particular server or client, but for a given package set. Because the specification of the packages is standard for all applications that implement the protocol, the fuzzing process decouples completely from the point of the connection (client, broker, etc.) that is performing testing, focusing solely on the packages that are being exchanged.

4.2.2. Fuzzing Messages Based on Previous Responses. In some situations, it is not possible to apply fuzzing to certain packets in a protocol to determine whether the values are correctly processed by the target machine. This is because some of its fields are based on a previous message. If you want to test a particular value of a package that has a random handle field that has been previously sent by the server, it is not enough to establish a connection and continuously send this type of package, since they will be rejected for having an incorrect handle field, and the destination application would never get to process the value, and therefore the fuzzing process would not be carried out in any of the cases.

This is another problem solved with the proxy approach. Since messages that are filtered and processed by the fuzzer come from a legitimate client and broker that establish a legitimate connection, fields that have been previously sent from one end to the other will remain intact and with the proper value.

4.3. Template-Based Fuzzing. In this paper, we present a novel, template-based fuzzing technique that aims to solve the problems presented in Sections 2.1 and 2.2.

As explained earlier, the current fuzzing tools use an approach that tries to simplify work for users by recalculating the control fields automatically using the block-based technique. Even so, this method continues to prove highly complex for users who wish to perform security checks on a specific protocol. The code that is shown below represents all the sentences that are required to implement a small program that allows application of fuzzing to four messages in a very simple protocol (FTP) through a framework called *Boofuzz* [27], which is widely used nowadays, and the successor to *Sulley* [39], which in turn is heavily influenced by *SPIKE*. As can be seen, the definition of complex protocols in this type of framework is still a tedious task, in addition to requiring a thorough knowledge of the tool itself and the entire specification of the protocol. It is at this point that the template-based approach would be useful.

```
def main():

    session = Session(

        target=Target(
        connection=SocketConnection("127.0.0.1",
        8021,proto=\tcp")))

    s_initialize("user")
    s_string("USER")
    s_delim(" ")
    s_string("anonymous")
    s_static("\r\n")
    s_initialize("pass")
    s_string("PASS")
    s_delim(" ")
    s_string("james")
    s_static("\r\n")
    s_initialize("stor")
    s_string("STOR")
    s_delim(" ")
    s_string("AAAA")
    s_static("\r\n")
    s_initialize("retr")
    s_string("RETR")
    s_delim(" ")
    s_string("AAAA")
    s_static("\r\n")
    session.connect(s_get("user"))
    session.connect(s_get("user"),s_get("pass"))
    session.connect(s_get("pass"),s_get("stor"))
    session.connect(s_get("pass"),s_get("retr"))
    session.fuzz()
```

The template-based approach works as follows:

(i) The tool listens in the middle of the communication as if it were a sniffer, using the proxy technique. The user has previously had to provide a series of parameters whereby the packets that pass through it will be filtered. These are the fields that were discussed in previous sections.

(ii) The user generates traffic between the client and the legitimate server of the protocol that he or she wants to fuzz. When the packets that were specified in the previous point are intercepted by the tool they are filtered and processed.

(iii) After processing the package, a *.json* template is automatically generated with the following format.

This portion of the template shows the MQTT *Publish* layer of an MQTT package. As you can see, each of the fields in the package appears and two extra attributes are added to each: *fuzzable* and *recalculate*. All the user has to do to apply fuzzing to a particular field of a package is to modify the *fuzzable* attribute by assigning the value true. The user will also have to assign the value true to the recalculate attribute of the fields that are considered to be recalculated automatically in order to maintain packet consistency. The tool will automatically enter the verification values in the fields that have been marked as *fuzzables* and will also recalculate all the fields in the package that have the *recalculate* flag set to *true*.

As we can see in Figure 1, the generation time of the templates is reasonably fast, and the generation algorithm is $O(n)$, which means that the generation time remains constant, regardless of the number of templates generated.

Thus, the third problem is solved, since the user does not have to know any details of the structures used by the tool or the protocol itself, besides the fields to which he or she wants to apply fuzzing, and in any case, the fields that he or she wants to recalculate. Note that, in order to make modifications to the template, the user does not require any special tool; this can be edited with a common text editor, as long as the *.json* structure is maintained.

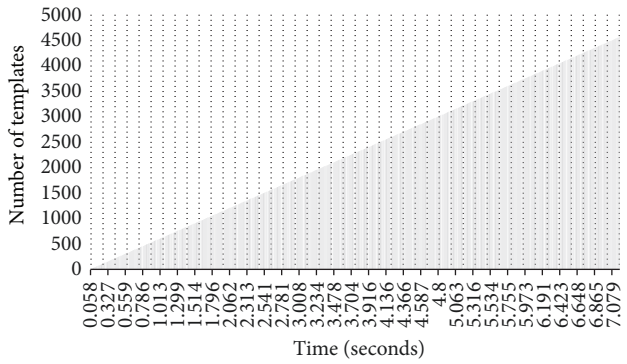

Figure 1: Time used by the application to generate templates.

4.4. Test Cases Generation. Generating the values with which an application is going to be tested is an important part of the fuzzing process [40, 41]. Often, the user is interested in running the test with a custom dictionary that has been generated with an external application; at other times, meanwhile, the user is interested in automatic generation of test cases with a certain degree of intelligence. The proposed tool implements both approaches and tries to maintain simplicity of use in both cases. The details of how this is implemented are discussed in the following section.

5. System Design

This section discusses the design considerations motivating our design and then describes the MQTT fuzzer system architecture.

5.1. Architecture of the Fuzzer. In this section, we will discuss the structure of the tool from the point of view of its design and implementation. We will take a tour of the main modules of which it is composed and its functionality. In addition, some secondary actions that the tool must carry out in order to make the fuzzing process satisfactory will be discussed.

Figure 2 shows the general architecture. The tool is composed of the following modules.

Mitmfuzzer. The *mitmfuzzer* module is the driver from which the rest of the application functions are called. Within it, the arguments that the user enters are parsed; this is done using the python module *argparse* [42]. In addition, it provides a small interface that shows the state of activity of the tool in a given situation.

Sniffer. The *sniffer* module is one of the main features of the tool. It is responsible for listening in the middle of a connection in order to filter and process the packages. It thus filters those specified by the user to subsequently generate a template. The core of the implementation of this module is based on *Scapy* [43], a framework for low-level treatment of network packets, which supports a large number of protocols. Once this module has detected a package selected by the user, it processes and serializes it in a certain format, which enables its processing using the python programming language. This package will be provided to the *template* module for template generation.

Template. This module receives a package in a certain format from the *sniffer* module and is in charge of processing it in order to generate the template in *.json* format. The generated templates are stored in a templates directory and will be used later by the fuzzer to identify the packages and fields to be fuzzed and recalculated.

Fuzzer. The fuzzer module is the most important module of the tool, as it performs the process of listening, packet filtering, generation, and insertion of test cases. The input of this module is a template file that must have been generated previously by the *template* module. Through the use of *iptables* and *nfqueue* [44, 45], the module continues listening to the communication as if it were a sniffer, redirecting the packages that are not identified with the template that has been introduced, and filtering and processing the packages that match the template. When a matching package is processed, all of its fields are compared to those in the template, looking to see whether one of them has been selected by the user to be fuzzed. In the case of one or more fields, the module checks whether the user has entered a custom dictionary to perform the test process: if so, the module will retrieve one of the test cases provided by the user and enter it in the field that was indicated for verification. Where the user has not provided a custom test case, the module will call *Radamsa* [46], passing as a parameter the file with the valid example case that is in *validcases/fieldnamedirectory*. Radamsa is a stock generator specially designed for software verification. It works by reading sample files that contain correct data, and through a series of algorithms it mutates this data, thus providing some intelligence, so that the generated results are more likely to lead to an error. *Radamsa* will automatically generate 50 different test cases and the module will take one of them to enter it in the field to be fuzzed. It should be noted that the generation of test cases performed by the module is continuous and infinite; when the module exhausts the 50 test cases generated, it automatically calls Radamsa to generate another 50 new cases.

Scapy. Scapy is a library for packet manipulation that supports a large number of network protocols. It has been considered necessary to name it in the application architecture, because it forms an important part of the core of the application. The advantage of Scapy, in addition to its extensive protocol support, is that it uses a block-based approach. This means that if you modify one of the fields in a package, you can recalculate the lengths and other control fields very simply and automatically. When a package arrives at the *Fuzzer* module, it sends it to Scapy for processing; Scapy then returns a structure that represents the package and which is easy to manipulate. After you have finished manipulating the MQTT package, Scapy takes this manipulated package, which is probably incorrect due to inconsistencies in control fields such as length fields (if text has been inserted or deleted) or checksum fields (if any byte of the package has been modified), recalculates all control fields using a block-based approach, and encapsulates the data as it was in the original

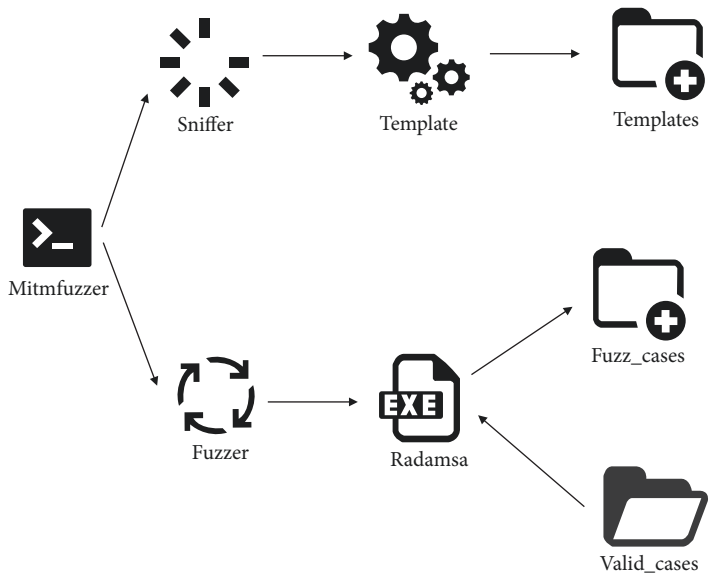

Figure 2: Architecture of the MQTT fuzzer.

package. It forwards this packet to the *Fuzzer* module, and the *Fuzzer* forwards it to the legitimate application. It is worth highlighting that Scapy did not have support for the MQTT protocol, and since it is the protocol object of study in this paper, we extended the library provided by adding support to MQTT. Currently, the module developed is part of the official repository of Scapy [47].

5.2. Test Cases Generation Implementation. In this section, we describe how the automatic generation of test cases has been applied in the implementation phase of the framework.

5.2.1. Automatic Generation of Test Cases. As explained in the description of the architecture of the tool, the automatic generation of test cases is carried out by an external application called Radamsa. This application is known to have been used to discover vulnerabilities like CVE-20073641 and CVE-2007-3644 (archive_read_support_format_tar.c library vulnerabilities), CVE-2008-6536 (7-zip program vulnerability), and CVE-2010-2482 (LibTIFF 3.9.4 vulnerability) among many others. The way in which the proposed tool uses this module is as follows: in the directory of the tool there are two important folders: a directory called *valid-cases*, composed of a set of subdirectories, one for each field of the package to be investigated. Inside these subdirectories, there are one or more sample files with correct data for that particular field. These will be provided to Radamsa to mutate them and generate the test cases. On the other hand, there is another directory called *fuzz-cases*, inside which a directory has been created for each of the fields to be fuzzed in a certain package. Radamsa automatically generates all field-specific test cases inside it, so that the tool subsequently retrieves them and inserts them into the packages.

5.2.2. Using Custom Test Cases. If, instead of using the automatic generation of test cases, it is desired to use a set of cases, generated either with another tool or manually, the user can do so in a straightforward manner by performing the following steps on the directory structure explained above:

(i) Inside the *fuzz-cases* directory, create a subdirectory with the exact name of the field you want to fuzz.

(ii) Inside the created subdirectory, enter all the test cases, one per file. The order or the name that is given to the file is not relevant.

6. Experimentation and Results

In this section, we present the results of applying the tool to a series of applications that are widely used today. All the test scenarios that are presented have been carried out in a controlled environment. The tools that are tested are open source and their use is free.

6.1. Performance Considerations. In this section, we have taken into account the performance implications of the tool. The section is divided into several subsections that evaluate the different functionalities of the presented tool and the impact of each of them on its performance.

6.1.1. Packet Processing. As has been presented in previous sections, the tool is located in the middle of the communication between a client and a broker, and from there it begins to modify all the network packets that flow between both, applying the proxy fuzzing technique. Because of this, much of the processing load of the tool corresponds to the modification and processing of packets on the fly, understanding processing such as insertion of test cases in the packets data fields and the recalculation of all the control fields of the previous layers.

Bearing this in mind, one of the aspects we have measured is the processing time per package. As can be seen in Figure 3,

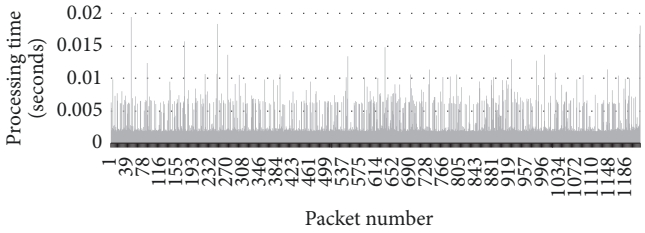

Figure 3: Processing time per package.

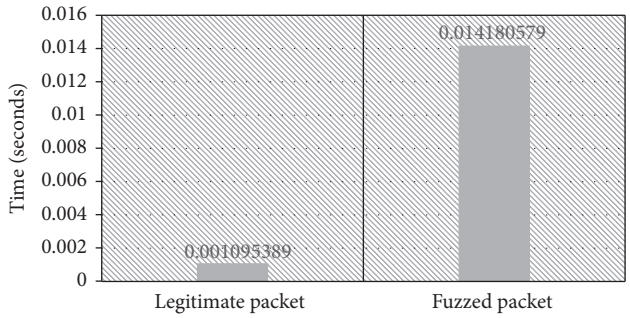

Figure 4: Difference between the transit time of a legitimate and a fuzzed package.

the processing time of each of the packages to which a test case is inserted remains relatively constant, with some variations due to the test case being inserted. If the test case has a longer length, the processing time will be longer because it will require recalculation of more fields. For the construction of the graph, a subset of 1300 network packets has been considered, which have reported an average processing time of 0.003699 seconds. This can be considered an acceptable time to keep the connection stable.

Once the processing time has been calculated for each packet, the arrival delay of a set of 100 packets after being processed has been calculated. This has been done because not only the overall processing time of a packet consists of inserting a test case and recalculating all the control fields of the lower layers, but also it is necessary to consider the delays caused by sending the packet from kernel space to user space so that it can be modified, sending of the package from user space to kernel space so that it can be sent, the additional time that it takes to be transported through the network, and so on.

As can be seen in Figure 4, the transit time of a fuzzed packet increases by approximately 90% with respect to the time of a legitimate packet, a total of 0.013085 seconds, which remains an acceptable time to maintain the connection stable without excessive delays.

6.1.2. Fuzzer Load and CPU Consumption. In previous sections, we have shown the number of templates that the tool is capable of generating and the time it takes to generate them. Another important performance measure is the number of test cases that the fuzzer is able to insert per time unit and the CPU consumption of the host machine.

The fuzzer has several customization features that allow you to select the time between test cases inserted, so that

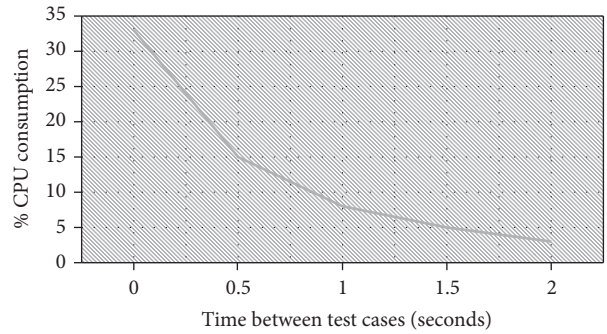

Figure 5: CPU consumption in relation to delay time between test cases.

you can insert everything as fast as possible or from time to time. In order to evaluate the CPU consumption of the host machine, different periods of time between inserted test cases have been taken into account.

As we can see in Figure 5, the CPU consumption of the machine that houses the fuzzer varies considerably depending on the delay time that is left between each inserted test case. This allows customizing the tool to be used in environments with fewer resources.

6.2. Application Scenarios. The term "application scenarios" refers to the possibilities offered by the tool within a connection to apply fuzzing to its elements. As explained in the previous sections, with the adopted approach it is possible to fuzz the different points of the connection of a protocol. The following are a series of use cases based on the MQTT protocol:

(i) Pub-fuzzer-broker-Sub: in this case, the tool would be placed between the client that is posting a message and the broker, in such a way that the tool could fuzz the messages that flow from the client to the server and the messages sent from the server to the client that is publishing.

(ii) Pub-broker-fuzzer-Sub: in this case, the scenario would change a little: the fuzzer would be between the broker and the client that is subscribed, waiting for the reception of messages. The tool could fuzz the messages from the broker to the client that is listening and the messages from the client to the broker, which will normally be acknowledgments.

6.3. Results. After applying the tool to some of the current brokers and clients, the fuzzer has been able to detect several failures that have led to denial of service and that may be potentially exploitable to perform other types of attack techniques. Some of these failures are as follows:

(i) Denial of service to the *MOQUETTE* broker v0.10 after the incorrect processing of a fuzzing package and throwing a Java exception that breaks the application

(ii) Error in handling the incoming connections by the broker *MOQUETTE* v0.10 after parsing a fuzzing package, which originates a connection reset

(iii) Denial of service of a *MOSQUITTO* client v1.4.11 that is subscribed to a certain topic when it receives a fuzzing message from the broker

This demonstrates that the fuzzing approach used provides real results in applications widely used by IoT devices around the world, and it can therefore be used as a security measure to ensure that devices in a given network meet minimum security standards.

7. Conclusion

The aim of this work was to contribute to improving the security of IoT devices and more specifically of the applications that implement a protocol widely used by Internet of Things (MQTT) as communication protocol for exchanging information. For this purpose, we developed a framework to perform security tests on MQTT. The tool implements a novel fuzzing technique based on templates, which according to our knowledge has not been used previously. The fuzzing technique presented here contributes to the field by improving some of the deficiencies of current fuzzers. The significant contribution of this framework is that it provides flexibility to fuzz the different points of a connection without making any adaptation effort. Among the other contributions explained during this paper, it is worth highlighting that it allows fuzzing packages that are based on a previously provided packet, and it facilitates portability and error reporting by exchanging templates. Finally, this technique simplifies the security analysis of the MQTT protocol to both users and applications by using the template-based approach, providing a way to fuzz the protocol without knowing or defining its specification.

Experimentation results gave acceptable processing time per fuzzed package. Moreover, it was observed that the tool behaves differently depending on the time that passes between each test case inserted. We were able to reduce the CPU consumption in the host machine to a minimum value of 2%. This flexibility to control the CPU consumption allows the use of this tool in environments with low processing power devices, such as Smart Cities. The tool was used to test vulnerabilities in widely used clients such as MOQUETTE or MOSQUITTO, with problems reported such as denial of service and communication resets of brokers. These discovered vulnerabilities probe the effectiveness of the tool.

The framework also has some limitations. The most significant one is related to the reporting and detection of errors. Error detection is carried out through the execution of the application to be tested under a debugger. Obviously, this needs to be automated to improve efficiency and usability. Another significant limitation is that the framework is currently only available for verifying MQTT protocol security; therefore the tool is not efficient for IoT architectures implementing several different protocols.

To improve the aforementioned limitations, as part of a future project, we will extend the framework to allow the verification of a wider range of network protocols used by IoT devices. Additionally, we are analysing the possibility of using the tool as a service that performs a security analysis of all the elements that are incorporated into a network for the first time. This would make it possible to ensure a minimum level of security and reliability for all the components of the infrastructure.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015.

[2] B. Chowdhury and M. U. Chowdhury, "RFID-based real-time smart waste management system," in *Proceedings of the In 2007 Australasian Telecommunication Networks and Applications Conference*, pp. 175–180, 2007, https://doi.org/10.1109/ATNAC.2007.IEEE.

[3] B. Padmavathi, "Implementation of IOT Based Health Care Solution Based on Cloud Computing," *International Journal of Engineering and Computer Science*, 2016.

[4] D. Gachet Páez, M. de Buenaga Rodríguez, E. Puertas Sánz, M. T. Villalba, and R. Muñoz Gil, "Healthy and wellbeing activities' promotion using a Big Data approach," *Health Informatics Journal*, p. 146045821666075, 2017.

[5] D. G. Páez, M. de Buenaga Rodríguez, E. P. Sánz, M. T. Villalba, and R. M. Gil, "Big Data Processing Using Wearable Devices for Wellbeing and Healthy Activities Promotion," in *Ambient Assisted Living. ICT-based Solutions in Real Life Situations*, vol. 9455 of *Lecture Notes in Computer Science*, pp. 196–205, Springer International Publishing, Cham, 2015.

[6] S. S. S. R. Depuru, L. Wang, V. Devabhaktuni, and N. Gudi, "Smart meters for power grid - Challenges, issues, advantages and status," in *Proceedings of the 2011 IEEE/PES Power Systems Conference and Exposition, PSCE 2011*, USA, March 2011.

[7] R. Roman, J. Zhou, and J. Lopez, "On the features and challenges of security and privacy in distributed internet of things," *Computer Networks*, vol. 57, no. 10, pp. 2266–2279, 2013.

[8] M. T. Villalba, M. de Buenaga, D. Gachet, and F. Aparicio, "Security analysis of an IoT architecture for healthcare," *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, vol. 169, pp. 454–460, 2016.

[9] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, "A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1125–1142, 2017.

[10] M. B. Yassein, M. Q. Shatnawi, and D. Al-Zoubi, "Application layer protocols for the Internet of Things: A survey," in *Proceedings of the 2016 International Conference on Engineering and MIS, ICEMIS 2016*, mar, September 2016.

[11] H. Yang, Y. Zhang, Y.-P. Hu, and Q.-X. Liu, "IKE vulnerability discovery based on fuzzing," *Security and Communication Networks*, vol. 6, no. 7, pp. 889–901, 2013.

[12] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerabilty discovery*, Addison-Wesley, Boston, Mass, USA, 2007.

[13] H. Yang, Y. Zhang, Y. Hu, and Q. Liu, "IKE vulnerability discovery based on fuzzing," *Security and Communication Networks*, vol. 6, no. 7, pp. 889–901, 2013.

[14] HiveMQ, "Enterprise MQTT Broker 2016," https://www.hivemq.com/wp-content/uploads/hivemq-product-sheet-v2-1.pdf.

[15] HiveMQ, https://www.hivemq.com/blog/mqtt-security-fundamentals-authenticationusername-password.

[16] I. Hedi, I. Špeh, and A. Šarabok, "IoT network protocols compa rison for the purpose of IoT constrained networks," in *Proceedings of the 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2017*, pp. 501–505, Croatia, May 2017.

[17] T. Heer, O. Garcia-Morchon, R. Hummen, S. L. Keoh, S. S. Kumar, and K. Wehrle, "Security challenges in the IP-based Internet of Things," *Wireless Personal Communications*, vol. 61, no. 3, pp. 527–542, 2011.

[18] A. Esfahani, G. Mantas, R. Matischek et al., "A Lightweight Authentication Mechanism for M2M Communications in Industrial IoT Environment," *IEEE Internet of Things Journal*, pp. 1-1.

[19] R. Neisse, G. Steri, and G. Baldini, "Enforcement of security policy rules for the internet of things," in *Proceedings of the 2014 10th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, WiMob 2014*, pp. 165–172, Cyprus, October 2014.

[20] S. Shin, K. Kobara, C.-C. Chuang, and W. Huang, "A security framework for MQTT," in *Proceedings of the 2016 IEEE Conference on Communications and Network Security, CNS 2016*, pp. 432–436, USA, October 2016.

[21] A. Manzoor, "Securing Device Connectivity in the Industrial Internet of Things (IoT)," in *Connectivity Frameworks for Smart Devices*, Computer Communications and Networks, pp. 3–22, Springer International Publishing, Cham, 2016.

[22] J. Foster, V. Osipov, N. Bhalla, N. Heinen, and D. Aitel, "Buffer Overflow Attacks," *Buffer Overflow Attacks*, 2005.

[23] K. Kaspersky and A. Chang, "Remote code execution through Intel CPU bugs," in *Proceedings of the In Hack In The Box (HITB)*, Malaysia, 2008.

[24] D. Reynders, S. Mackay, and E. Wright, *Modbus overview. Practical Industrial Data Communications*, 10.1016/b978–3/50012-7, 2004.

[25] D. Serpanos and M. Wolf, "Security Testing IoT Systems," in *In Internet-of-Things (IoT) Systems*, pp. 77–89, Springer, Cham, Switzerland, 2017.

[26] D. Aitel, *The advantages of block-based protocol analysis for security testing*, Immunity Inc, February 2002.

[27] J. Pereyda, boofuzz: Network Protocol Fuzzing for Humans, http://boofuzz.readthedocs.io/en/latest/.

[28] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 4176, pp. 343–358, 2006.

[29] H. J. Abdelnur, R. State, and O. Festor, "KiF: A stateful SIP fuzzer," in *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications, IPTComm '07*, pp. 47–56, USA, July 2007.

[30] S. Gorbunov and R. Rosenbloom, *AutoFuzz, Automated network protocol fuzzing framework*, Department of Mathematical and Computation Sciences, University of Toronto, Mississauga, Canada, 2010.

[31] D. Aitel, An Introduction to SPIKE, the Fuzzer Creation Kit, https://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt.

[32] "Github.org mqtt_fuzz," https://github.com/F-Secure/mqtt_fuzz.

[33] K. Mladenov, S. van Winsen, C. Mavrakis, and K. P. M. G. Cyber, Formal verification of the implementation of the MQTT protocol in IoT devices,.

[34] "OWASP.org, ZAP Proxy," http://www.zaproxy.org/.

[35] "PortsWigger.net, BurpSuite," https://portswigger.net/bur.

[36] "Github.com, ProxyFuzz," https://github.com/SECFORCE/proxyfuz.

[37] L. Landauer, "Fuzzing Requests And Responses Using A Proxy," U.S. Patent Application No. 11/276,454.

[38] OASIS.org, "MQTTVersion3.1.1:OASISStandard," http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[39] A. Takanen, J. Demott, and M. Charles, "Fuzzing for Software Security Testing and Quality Assurance , Artech House Information Security and Privacy," *Fuzzing for Software Security Testing and Quality Assurance , Artech House Information Security and Privacy*, 2008.

[40] A. A. Sofokleous and A. S. Andreou, "Batch-optimistic test-cases generation using genetic algorithms," in *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2007*, pp. 157–164, Greece, October 2007.

[41] R. Abbassi, S. Guemara, and F. El, "Towards a test cases generation method for security policies," in *Proceedings of the 16th International Conference on Telecommunications, ICT 2009*, pp. 41–46, Morocco, May 2009.

[42] "Python.org, Argparse," https://docs.python.org/3.4/library/argparse.html.

[43] "Scapy A Python Tool For Security Testing," *Journal of Computer Science & Systems Biology*, vol. 8, no. 3, 2015.

[44] J. Alan, "Netfilter and IPTables - A Structural Examination," *SANS Institute*, 2004.

[45] "Netfilter.org, Netfilter," https://www.netfilter.org/.

[46] "University of Oulu, Radamsa," https://www.ee.oulu.fi/roles/ouspg/Radams.

[47] "Scapy.org, MQTT layer for Scapy," https://goo.gl/oo45XC.