Black-Box Live Protocol Fuzzing

Timo Ramsauer (me@timo-ramsauer.de)

Abstract—Fuzzing is an automated testing technique. Great quantities of test cases are randomly generated and send to a target. Fuzzing should be part of the reliability testing process. Especially for IoT-Devices, there is often no access to the source code or the binaries. They can only be accessed through their network interface. This paper introduces a black-box fuzzer, which alters network traffic. It builds on the assumption, that the network protocol is known, to improve accuracy. It alters traffic in a system consisting of several devices. This allows it to detect faults, which reside in non-starting states, with minimal setup effort.

Index Terms—Black-Box, Fuzzing, Network, MQTT, Protocol, Security.

Contents

1 Introduction

FUZZING is a great tool to quickly test implementation of network protocols and should be part of the assessment of software security. In this work a fuzzer is presented, which is a man-in-the-middle non-deterministic network fuzzer written in Python. The fuzzer randomly changes contents on the network traffic. It supports TCP and can be configured to fuzz each field of the higher level protocols. It builds onto Scapy [1], so it supports dissection of many protocols out of the box. The fuzzer can be configured to use fitting fuzzing methods for each protocol field. Furthermore it can easily be extended to support further protocols and reduces the time the user must invest to implement this technique.

This fuzzer is the first one to dissect packets on the wire and allow altering of multiple packets in one session and for multiple devices. It makes use of the known structure of a huge range protocols. This makes it possible to efficiently test whole systems, such as IoT networks.

1.1 Fuzzing

Fuzzing is a popular highly automated software testing technique, which can be used to increase trust in the absence of vulnerabilities and bugs, by generating random input [2], [3], [4]. It works by feeding the target with plenty of test cases, which may trigger software faults [2], [4]. The challenge is to produce test-cases, which are prone to trigger bugs.

Fuzzing is a negative testing technique. It does not check if a feature works, but tries to find flaws, which lead to security flaws or other undesired behavior. "Unexpected or semi-valid inputs or sequences of inputs are sent to the tested interfaces, instead of the proper data expected by the processing code" [4]. Dumb fuzzing, which just sends random input to its target is measured to be 50% less effective than intelligent fuzzing [4].

1.1.1 Black-box fuzzing

This paper focuses on black-box fuzzing. It works without any knowledge of the targets code. This is often the case, if IoT products are tested. However, knowledge about the communication protocol is assumed. This helps with the generation of semi-valid input. Without the knowledge of the protocol most test cases will immediately be dismissed and the fuzzer is very inefficient. To fuzz stateful protocols the fuzzer is inserted as a Man-in-the-Middle (MitM) between communication partners. The communication between the components is randomly altered based on templates. Using this approach, it can be ensured, that the potentially vulnerable system gets tested in many different states and not only in its initial case.

If there is access to the code or even only access to the binary, coverage can be measured. Symbolic Execution and Evolutionary Fuzzing can be applied. They can more efficiently focus on untested code and perform best [4]. However, often access to code or debug interfaces is not given, therefore black-box fuzzing is chosen in this paper.

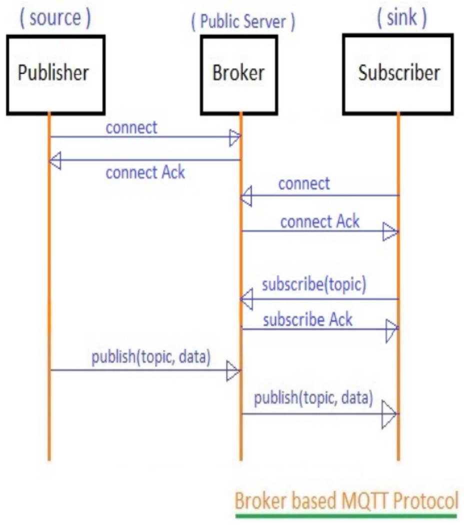


Fig. 1. MQTT Sequence Example [7]

1.1.2 *Target*

Fuzzers can focus on hardening against different attack vectors. They can focus on different layers of the network communication, on the file system format or the file contents [4]. Furthermore, fuzzers may focus on servers or clients [4]. The fuzzer of this paper intends to test an application implemented on several different devices, such as an IoT infrastructure running the MQTT protocol. It focuses on the communication between the devices and on the application layer. Lower layers are expected to use libraries, which should be tested separately using better tailored techniques. The network communication is interesting, because it is exposed to a bigger attack vector.

1.2 MQTT

In this paper the MQTT protocol is used as an example use-case of this fuzzing technique.

> MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. For example, it has been used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios. It is also ideal for mobile applications because of its small size, low power usage, minimised data packets, and efficient distribution of information to one or many receivers. [5]

Shodan currently (11.2018) finds about 65.500 MQTT brokers on the Internet [6]. A vulnerability in a popular broker software would expose a lot of devices to attacks. The protocol is open and fairly simple, but very popular and gets increasing attention, because of the IoT movement. It is therefor a good target for a case study.

MQTT uses a publish and subscribe pattern. It doesn't use a common client-server architecture to communicate, but uses a broker to distribute messages. MQTT clients are devices, which either publish or subscribe to an MQTT broker. They can be very small resource constraint devices. Client libraries are available for a wide range of programming languages and can easily be implemented [8], [9]. The broker is the central system, all messages pass through. It is responsible to receive all messages and forward them to the subscribers. It may have further tasks such as authentication and authorization of clients, filtering of messages and management of persistent client sessions [8].

MQTT connection are based on TCP/IP. First a TCP session needs to be established between the client and the broker [9]. The clients initiate a MQTT session by sending a CONNECT message, which is answered by the broker with a CONNACK [9], [10]. An example setup of a connection is shown in figure 1.

After the connection is successfully initiated, clients can subscribe to so called topics. They do this by sending a SUBSCRIBE request, which is answered by a SUBACK packet [9]. If a client now publishes a message to the same topic it is forwarded to this subscriber [9].

The PUBLISH packet (shown in figure 2), which is used to publish a message can have different QoS-levels. It ranges from a simple PUBLISH message, which is only acknowledged on the TCP layer to a four-part handshake between the sender and the receiver [8], [9]. The sender needs to keep track of the state and may behave differently to incoming messages and therefore expose different vulnerabilities. Because of that, it needs to be tested in all states.

Each packet consists of several different fields having different data types. The packet identifier of an publish packet is for example a two byte integer [9] (compare figure 2). The topic name is an UTF-8 encoded string. It starts with the length of the string (2 byte integer) and then with the actual UTF-8 encoded string. The QoS-flag is two bits long and can have the values 0x00, 0x01 and 0x10. Further more there are binary flags such as the retain flag and the DUP flag of the publish packet. The MQTT header has further more a fairly complicated way to encode the length of the rest of the packet.

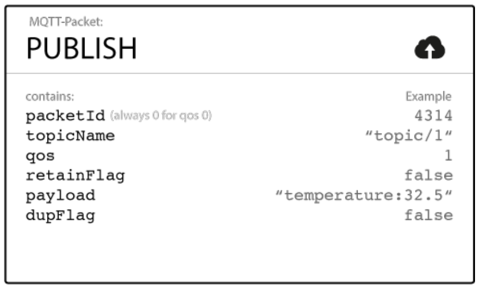
"The Remaining Length is encoded using a variable length encoding scheme which uses a single byte for values up to 127. Larger values are handled as follows. The least significant seven bits of each byte encode the data, and the most significant bit is used to indicate that there are following bytes in the representation. Thus each byte encodes 128 values and a "continuation bit". The maximum number of bytes in the Remaining Length field is four." [9] [1]

The different data types should be fuzzed differently to more efficiently test the program. If the length fields are not tested, they should be recomputed to create a semi-valid packet, which passes first validation checks.

2 Previous Work

There are other tools such as ProxyFuzz [12], which intercept network traffic and fuzz it on its way. However, it is protocol agnostic [4]. This approach is inefficient for many protocols, because a lot of packets are dropped, because of validations such as checksums [4]. Nonetheless, the fuzzer is easy to setup

1. This was not implemented correctly in Scapy, which is used to dissect the packet. It was fixed in pull request #1371 [11].

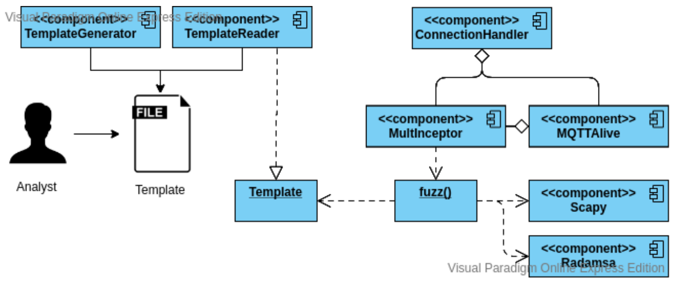


MQTT-Packet:

PUBLISH

contains:	Example
packetId (always 0 for qos 0)	4314
topicName	"topic/1"
qos	1
retainFlag	false
payload	"temperature:32.5"
dupFlag	false

Fig. 2. Fields inside a publish packet [8]



<<component>> TemplateGenerator
<<component>> TemplateReader
<<component>> ConnectionHandler
FILE
Analyst
Template
<<component>> MultInceptor
<<component>> MQTTAlive
Template
fuzz()
<<component>> Scapy
<<component>> Radamsa
Visual Paradigm Online Express Edition

Fig. 3. Component Diagram

[4], which is an advantage. The fuzzer introduced in this paper tries to combine this approach, with protocol knowledge. Because many protocols are already known by Scapy, the ease of use remains, but efficiency is increased.

Other network fuzzers such as the Mutiny Fuzzer [13], allow traffic generation on past recorded traffic using the Radamsa fuzzer. However, it does not modify life traffic.

The Automated Network Protocol Fuzzing Framework (AutoFuzz) "learns a protocol implementation by constructing a Finite State Automaton (FSA) which captures the observed communications between a client and a server" [14] and uses a proxy server comparable to ProxyFuzz.

Polymorph allows fuzzing of live traffic and is able to dissect packets [15]. However TCP breaks, if more than one packet is modified. It does not keep track of the connections. There is a workaround for one system, but it breaks for multiple clients accessing one server. It's not easy to fuzz multiple fields and to use separate methods to fuzz. This is important, because variable length fields can include very different values than a field, which is always a fixed length. It makes the fuzzer inefficient to put in values bigger than the fixed length, because it breaks the packet.

3 The New Fuzzer

The fuzzer is written in Python and allows modification of packets "on the air". The fuzzer is introduces as a Man-in-the-Middle between the different network components, allowing it to fully control the communication. The fuzzer aims to be able to provide its user a simple way to perform fuzzing on a wide range of protocols.

The basic components are shown in figure 3. The `TemplateGenerator` generates templates from a package capture file. It altered by an analyst and is read back in by a `TemplateReader`.

The `ConnectionHandler` creates a `MultInceptor` for each TCP connection, which devices setup. The `MultInceptor` fuzzes one connection using the `fuzz()` method, which alters the fields of each packet depending on the template. It depends on the libraries Scapy and Radamsa to dissect and fuzz the packet.

To check if all devices are still alive they are added to the `MQTTAlive` components, which periodically pings them. If a fault is detected it shuts all threads down.

In the following sections each step of the process will be covered in greater detail. It will be explained, how a user can use the fuzzer, how it was implemented and why certain implementation and process choices were made.

4 Installation of Fuzzer

The implementation of the fuzzer is on https://github.com/ramsaut/mqtt_fuzzing. It can be downloaded using the `git clone` command.

The fuzzer requires Radamsa and several Python packages to function. The installation process of Radamsa is explained at https://gitlab.com/akihe/radamsa. It is recommended to setup a virtual environment with pip as a Python environment. The requirements can be installed using the following command inside the project folder:

```
pip install -r requirements.txt
```

The fuzzer can be configured in the `config.ini` file. If the fuzzer shall be configured by external programs the `dict` in `mqtt_fuzzing.config.config` needs to be overwritten e.g. using the `read_json` method.

The fuzzing process can be started by running the `mqtt_fuzzing.py` script.

5 The Fuzzing Process

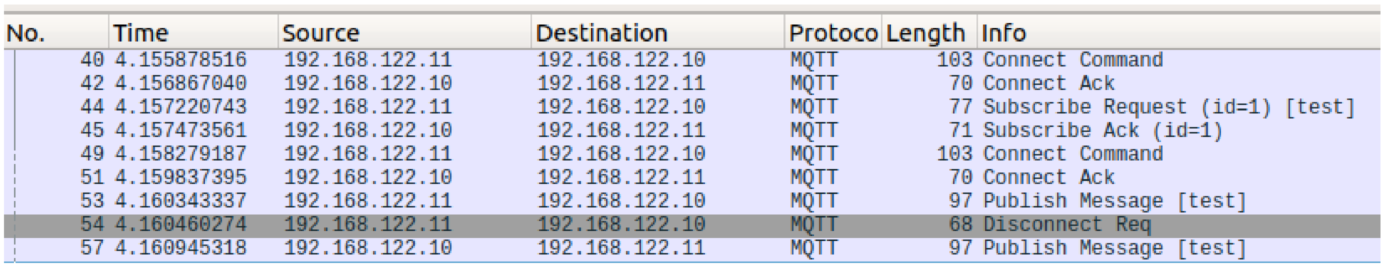
First a test system is setup, which generates network traffic. This traffic is recorded to create a baseline. The capture is recorded and the used protocols are extracted. The security researcher decides, which protocols and which fields inside the protocol shall be fuzzed. The fuzzer is then integrated into the system and the traffic is altered on its way through the fuzzer. The fuzzer needs to check, if the target had any errors. To allow the researcher to analyze how the system failed, the test cases are saved and the error is logged. Following the process is described in greater detail.

5.1 Generating Traffic

The traffic, which needs is altered by the fuzzer first needs to be generated. To efficiently fuzz, there needs to be a high bandwidth of traffic, otherwise the fuzzing process is too slow. The protocol needs to be known to Scapy, which is the Python library used to dissect the packages.

A sample system is implemented to generate traffic. Multiple virtual machines are setup in KVM. One of them exposes an MQTT broker. Further virtual machines can be used as clients. The virtual machines are setup using Ubuntu Core.

> Ubuntu Core is a tiny, transactional version of Ubuntu for IoT devices and large container deployments. It runs a new breed of super-secure, remotely upgradeable Linux app packages known as snaps - and it's trusted by leading IoT players, from chipset



No.	Time	Source	Destination	Protoco	Length	Info
40	4.155878516	192.168.122.11	192.168.122.10	MQTT	103	Connect Command
42	4.156867040	192.168.122.10	192.168.122.11	MQTT	70	Connect Ack
44	4.157220743	192.168.122.11	192.168.122.10	MQTT	77	Subscribe Request (id=1) [test]
45	4.157473561	192.168.122.10	192.168.122.11	MQTT	71	Subscribe Ack (id=1)
49	4.158279187	192.168.122.11	192.168.122.10	MQTT	103	Connect Command
51	4.159837395	192.168.122.10	192.168.122.11	MQTT	70	Connect Ack
53	4.160343337	192.168.122.11	192.168.122.10	MQTT	97	Publish Message [test]
54	4.160460274	192.168.122.11	192.168.122.10	MQTT	68	Disconnect Req
57	4.160945318	192.168.122.10	192.168.122.11	MQTT	97	Publish Message [test]

Fig. 4. Extract of the sample traffic data

vendors to device makers and system integrators. [16]

This makes it very suitable to setup the test environment. Each machine gets a static IP in an separated network. All machines get configured with the Mosquitto MQTT implementation.

One client (192.168.122.11) subscribes to the topic test on the broker (192.168.122.10). Furthermore it publishes a message to the same topic, which is forwarded to the subscriber. The publish is repeated regularly. The traffic is generated using the following bash script:

```
mosquitto.sub -h 192.168.122.10 -p 1883 -t test &

while true; do
mosquitto.pub -h 192.168.122.10 -t test -m "This␣
    ↪ is␣a␣test␣message."
sleep 1
done
```

Furthermore an Arduino MKR1000 is setup with a simple MQTT example. It uses the `Wifi101` and the `PubSubClient` libraries to connect to a wireless network and to implement the MQTT protocol. It connects to an MQTT server then publishes ”hello world” to the topic ”outTopic” every two seconds. Furthermore it subscribes to the topic ”inTopic”, printing out any messages it receives. It assumes the received payloads are strings not binary. If the first character of the topic ”inTopic” is an 1, the Led of the micro-controller is switched ON, else it is switched off. It will reconnect to the server if the connection is lost using a blocking reconnect function. The Arduino is flashed using the Arduino IDE.

5.2 Sample Data

The traffic can be captured on the virtual machine host using Wireshark. An extract of the traffic is shown in figure 4. In the Wireshark screen-shot one can notice, that for a simple subscribe request four packets are send. The connect command and the corresponding acknowledgement start the session. Then a subscribe request and its acknowledgement are send. There are further `ACK` packets on the TCP layer. However, these are transparent to the fuzzer and are therefore not included. The gathered traffic is exported to a pcap file, which can be used in the precomputation of the fuzzer.

5.3 Precomputing

The sample data is accepted as a network packet capture (pcap-file). JSON-templates are generated for each protocol layer (above TCP) and direction (to broker / from broker). The user can later specify, which fields inside packets shall be fuzzed, by altering the JSON-templates.

The name of the layer and the direction of the packet (towards the broker or from the broker) is saved. Furthermore for each field in the layer the values which occurred and the type of the field are saved. An empty fuzzing `dict` in included. Each template looks similar to the following example:

```
{
        "to_broker": true,
        "name": "MQTT",
        "attributes": {
            "fields": {
                "type": {
                    "values": [
                        0,
                        ...
                        14
                    ],
                    "type": "BitEnumField",
                    "fuzzing": {
                        "fuzzer": null,
                        "cases": null
                    }
                },
                ...
                "len": {
                    "values": [
                        0,
                        ...
                        118
                    ],
                    "type": "VariableFieldLenField",
                    "fuzzing": {
                        "fuzzer": null,
                        "cases": null
                    }
                }
            }
        }
    },
```

The fuzzer can be set to:

```
"fuzzing": {
        "fuzzer": "scapy",
        "cases": null
}
```

In this case random values are chosen for the field. It depends on the Scapy `randval()` method. This method sets the field to a random valid value.

The fuzzer can also be set to:

```
"fuzzing": {
        "fuzzer": "radamsa",
        "cases": "packet"
}
```

In this case the general purpose fuzzer Radamsa is used.

> It works by reading sample files of valid data and generating interestingly different outputs from them. The main selling points of radamsa are that it has already found a slew of bugs in programs that actually matter, it is easily scriptable and easy to get up and running. [...] Radamsa is an extremely "black-box" fuzzer, because it needs no information about the program nor the format of the data. [...] Radamsa is intended to be a good general purpose fuzzer for all kinds of data. [17]

This fuzzer is recommended for all fields with variable length. For MQTT these might for example be the topic or the message of an MQTTPublish. The fuzzer bases its random value on the value of the original package. It is block-based and generally keeps the format valid(-ish). In future versions an option to base the new value on a pool of all past values may be added.

The templates are generated using the following code:

```
class TestGenerator():
    generator = TemplateGenerator()

    def test_create_templates(self):
        list = self.generator.read_capture()
        templates = self.generator.create_templates
            ↪ (list)
        self.generator.save_to_disk(templates)
```

A generator is setup using `TemplateGenerator`. A config file is read in and the configuration is saved in the `config` dictionary. A list of packages is read in from the file specified in `config['Fuzzer']['MQTT_Sample']` using the method `self.generator.read_capture()`. Templates are generated using `self.generator.create_templates(list)` and written to disk using `self.generator.save_to_disk(templates)`.

5.4 Reading Templates

After the user adjusted the templates, by choosing the fields, which shall be fuzzed. The templates are read in using the method `readTeamplates()` of `mqtt_fuzzing.gen_template.TemplateReader`. It reads the templates from `config['Fuzzer']['Templates']`.

```
reader = TemplateReader()
templates = reader.readTeamplates()
```

5.5 Altering Packets

To test a system the fuzzer is inserted between the clients and the broker as a Man-in-the-Middle. In a real system the fuzzer might be introduced to the network by ARP spoofing or other means of redirecting the traffic flow. Sometimes systems might allow to set the address of the broker. This is the case for the test system. If SSL is used, the right certificates need to be set up. A tool such as BetterCAP may be used to perform the MitM attack and SSL stripping [18].

The fuzzer is able to handle several connections at the same time. Each connection is handled in its own thread using its own socket. All devices involved in the connections need to be checked, if they are still alive or if there was a fault.

The fuzzer is altering packets on their way. For this purpose the packet is matched to a template, matching the direction and the protocol type. The packet is dissected and fields specified in the template are fuzzed. The packet is then reassembled to a valid packet. Length fields and checksums are recomputed, if they are not fuzzed, to create semi-valid packets, which pass initial validation testing a find problems residing deeper in the process.

To understand the usage of the fuzzer, we take a look at the following snippet:

```
import unittest
from mqtt_fuzzing.intercept import *
from mqtt_fuzzing.gen_template import *

class TestIntercept(unittest.TestCase):
    def setUp(self):
        reader = TemplateReader()
        self.templates = reader.readTeamplates()

    def test_intercept(self):
        c = ConnectionHandler()
        c.set_templates(self.templates)
        c.start()
```

The templates are read in as explained in the previous chapter. To intercept traffic a `ConnectionHandler()` is created and the templates are set. Furthermore the handler is started.

The `ConnectionHandler` automatically opens a socket on the port specified in `config['Broker']['Port']`. Furthermore it creates an `MQTTAlive`, which checks if the broker and all registered clients are still alive or if a bug has been found. This component is further explained in the chapter 5.6.

For each new connection the `ConnectionHandler` creates a `MultInterceptor` and adds it to the list of clients in the `MQTTAlive`.

The `MultInterceptor` opens a `remote_socket` corresponding to the `local_socket`. The `remote_socket` handles the traffic to and from the broker, while the `local_socket` handles the traffic with the client.

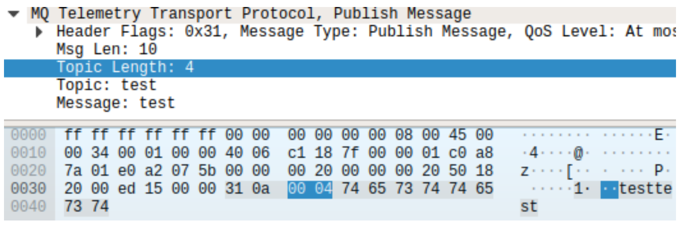
The `MultInterceptor` checks if an SSL connection is handled using the `config['Broker']['SSL']` attribute. If that is the case an SSL connection is opened in both directions using the certificates specified in the config. The analyst needs to configure the client to use the certificate of the fuzzer and the fuzzer needs to use the certificate of the broker.

The interceptor listens for traffic on both sockets. The TCP layer and all below are stripped away, the data is modified using the `modify()` method and is forwarded to the other device.

The modifying is mainly done in the `mqtt_fuzzing.fuzz.fuzz()` method, which is called by `modify()`. It matches each layer of the packet to its template:

```
layer = templates[(to_broker, type(payload).
    ↪ __name__)]
```

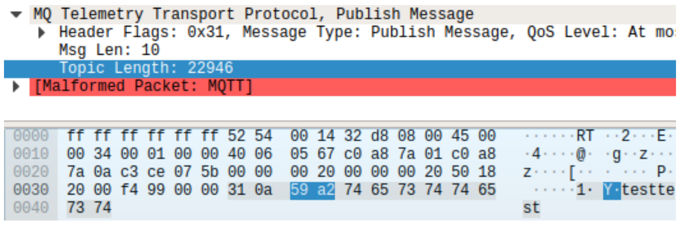
For each direction (to broker, from broker) there is another template. For each field inside the layer the fuzzer looks in the templates and determines the fuzzing method. Currently are two methods implemented. These were explained in chapter 5.3.



```
▼ MQ Telemetry Transport Protocol, Publish Message
  ▶ Header Flags: 0x31, Message Type: Publish Message, QoS Level: At mos
    Msg Len: 10
    Topic Length: 4
    Topic: test
    Message: test

0000  ff ff ff ff ff ff 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 34 00 01 00 00 40 06  c1 18 7f 00 00 01 c0 a8   ·4····@· ········
0020  7a 01 e0 a2 07 5b 00 00  00 20 00 00 00 20 50 18   z····[·· ·  ···P·
0030  20 00 ed 15 00 00 31 0a  00 04 74 65 73 74 74 65    ·····1· ··testte
0040  73 74                                              st
```

(a) Original



```
▼ MQ Telemetry Transport Protocol, Publish Message
  ▶ Header Flags: 0x31, Message Type: Publish Message, QoS Level: At mo
    Msg Len: 10
    Topic Length: 22946
▶ [Malformed Packet: MQTT]

0000  ff ff ff ff ff ff 52 54  00 14 32 d8 08 00 45 00   ······RT ··2···E·
0010  00 34 00 01 00 00 40 06  05 67 c0 a8 7a 01 c0 a8   ·4····@· ·g··z···
0020  7a 0a c3 ce 07 5b 00 00  00 20 00 00 00 20 50 18   z····[·· ·  ···P·
0030  20 00 f4 99 00 00 31 0a  59 a2 74 65 73 74 74 65    ·····1· Y·testte
0040  73 74                                              st
```

(b) Fuzzed

Fig. 5. Comparison of the original package and the fuzzed package

In the example shown in figure 5, an MQTT packet and its fuzzed version can be seen. The topic length is fuzzed. All lower layers are recalculated and the packet is forwarded to the broker.

If certain fields were not fuzzed they are recalculated to create valid packets:

```
# Recompute if values have not been fuzzed
if ('MQTT', 'len') not in been_fuzzed:
del packet['MQTT'].len
if 'MQTTPublish' in packet:
if ('MQTTPublish', 'length') not in been_fuzzed:
del packet['MQTTPublish'].length
if 'MQTTConnect' in packet:
if ('MQTTConnect', 'length') not in been_fuzzed:
del packet['MQTTConnect'].length
if ('MQTTConnect', 'clientIdlen') not in
    ↪ been_fuzzed:
del packet['MQTTConnect'].clientIdlen
```

For MQTT the length fields need to be recalculated. The fuzzer deletes all fields, which get automatically calculated to create a valid packet. The recalculation is handled by Scapy.

5.6 Detecting Faults

Faults need to be detected on a case by case basis. However the architecture is designed to allow easy addition of fault detection.

The fault detection is implemented in the class `MQTTAlive`. `MQTTAlive` can be found in `mqtt_fuzzing.mqtt_ping`. It is used to check if the broker is still alive and to shutdown the fuzzer, if an error is detected. Two MQTT clients are created for this purpose:

```
import paho.mqtt.client as paho
client1 = paho.Client("heartbeatsub")
client2 = paho.Client("heartbeatpub")
```

The first client subscribes to the topic heartbeat. The second clients publishes packages to this topic in the interval specified in `config['Heartbeat']['Frequency']`. If the second client does not receive a packet for a time of `config['Heartbeat']['Timeout']`. The fuzzer is stopped. If a client does not send a message for the time specified in `config['Heartbeat']['Timeout']`. The fuzzing is stopped as well. This is implemented in the `MultInterceptor`.

5.7 Logging

The unaltered traffic is saved to a pcap file. The altered traffic is saved to a pcap file. Found faults are logged.

6 Case Study

The fuzzer was introduced, between a client and a broker. The client was the Arduino MKR1000 mentioned in section 5.1. The broker is using the popular Mosquitto implementation. The outlined fuzzing process was applied an the fuzzer was instructed to fuzz all fields. All fields of variable length were instructed to be fuzzed with Radamsa. All other fields were fuzzed with the Scapy module.

A bug could be found. When processing fields of variable length the broker Mosquitto and the Arduino MQTT library pubsubclient wait for further data.

A common MQTT packet such as a publish packet is fuzzed. A length field e.g. the remaining length field in the header is increased. The package is acknowledged on the TCP layer. Nevertheless, their is no answer on the MQTT layer, a MQTTConnect is for example not answered with an MQTTConnAck. The TCP session is not closed, as it is commonly done, when a faulty packet is received.

Further research needs to be done to validate the following possible attack. The memory to write the payload into (up to 256 MB) is reserved to store each MQTT packet. Multiple connections are opened to the target by the attacker. Because of that, all of the memory is used up resulting in a DoS of the target.

7 Future Work

The program takes a black box approach to fuzz network systems. It can be adopted for other protocols. The following parts, would need to be rewritten. The recomputation needs to be adopted to recompute all length and checksum fields of the specific protocol. This part may be automated in future versions by including an attribute in the template which specifies which fields need recomputation. The MQTTAlive is protocol specific. For other protocols, a specific methods to check if the system is alive needs to be implemented. Furthermore the fuzzer could be extended to parse logs of the MQTT-devices to more reliably detect errors.

Further methods to fuzz the protocol fields should be implemented and their effectiveness needs to be checked.

8 Conclusion

A black-box live fuzzer was implemented and assessed. It is able to fuzz a MQTT system and was able to find faults. It can be used to fuzz IoT devices, without access to their source code. The functionality was demonstrated by fuzzing the MQTT system. However, the methodology can also be used for other protocols.

The fuzzer can intercept multiple connections at the same time. By dissecting the packet and fuzzing single protocol fields using fitting fuzzing techniques, it finds faults more reliable.

References

[1] Philippe Biondi, “Scapy.” [Online]. Available: https://secdev.github.io/

[2] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, “Fuzzing: State of the Art,” *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8371326/

[3] P. Oehlert, “Violating assumptions with fuzzing,” *IEEE Security Privacy*, vol. 3, no. 2, pp. 58–62, Mar. 2005.

[4] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for software security testing and quality assurance*, ser. Artech House information security and privacy series. Norwood, MA: Artech House, 2008, oCLC: ocn213308372.

[5] “MQTT.” [Online]. Available: http://mqtt.org/

[6] “mqtt - Shodan Search,” Nov. 2018. [Online]. Available: https://www.shodan.io/search?query=mqtt

[7] “Intro to REST and MQTT,” Jul. 2017. [Online]. Available: http://wireless.ictp.it/school_2017/Slides/MQTT.pdf

[8] “MQTT Essentials Part 4: MQTT Publish, Subscribe & Unsubscribe,” Feb. 2015. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe

[9] “MQTT Version 3.1.1,” *OASIS Open*, p. 81, Oct. 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[10] “MQTT Essentials Part 6: Quality of Service 0, 1 & 2,” Feb. 2015. [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels

[11] Timo Ramsauer, “Fixed MQTT module for payload length > 127 bytes #1371,” Nov. 2018, original-date: 2015-10-01T17:06:46Z. [Online]. Available: https://github.com/secdev/scapy

[12] “ProxyFuzz - MITM Network Fuzzer in Python,” Jun. 2007. [Online]. Available: https://www.darknet.org.uk/2007/06/proxyfuzz-mitm-network-fuzzer-in-python/

[13] “Mutiny Fuzzing Framework,” Nov. 2018, original-date: 2017-10-27T19:23:53Z. [Online]. Available: https://github.com/Cisco-Talos/mutiny-fuzzer

[14] S. Gorbunov and A. Rosenbloom, “AutoFuzz: Automated Network Protocol Fuzzing Framework,” p. 7, 2010.

[15] Santiago HernÃąndez Ramos, “Polymorph: A Real-Time Network Packet Manipulation Framework,” Apr. 2018.

[16] Canonical, “Ubuntu Core,” 2018. [Online]. Available: https://www.ubuntu.com/core

[17] Aki Helin, “radamsa.” [Online]. Available: https://gitlab.com/akihe/radamsa

[18] “BetterCAP.” [Online]. Available: https://www.bettercap.org/