Building a simple Proxy Fuzzer for the MQTT protocol using the Polymorph framework

@santiagohramos
https://github.com/shramos/polymorph

This article shows how easy you can build a fuzzer for the MQTT protocol by using the Polymorph framework.

I will start by assuming that the reader knows the MQTT protocol. For those who do not know it, you can find more information here. The first thing we will do is prepare the environment where we will perform the fuzzing, in this case, it will be very simple, a Kali Linux machine in which we will install the following dependencies:

Polymorph framework

```
apt-get install build-essential python-dev libnetfilter-queue-dev tshark tcpdump python3-pip
```

```
pip3 install --process-dependency-links polymorph
```

Mosquitto

```
apt-get install mosquitto mosquitto-clients
```

Radamsa

```
sudo apt-get install gcc make git wget
```

```
git clone https://github.com/aoh/radamsa.git && cd radamsa && make && sudo make install
```

With all this installed, we are ready to start!

Before starting the construction of the fuzzer, we are going to test our mosquitto installation in localhost. To do this, we are going to open two terminals and execute a client that is going to subscribe to a certain topic and another one that publishes to that same topic. In the following image you can see the commands and the result.

Well, now that we have tested the communication between both clients, we are going to open Polymorph and begin with the capture and modification of MQTT packets in real time.
In our particular case, we are going to fuzz the msg field of the \textit{MQTTPublish} packets, notice that the modification of any other field would be done in exactly the same way. Also, for simplicity, we are going to modify MQTT packets that implement the IPv4 protocol. Sometimes you will see that Polymorph captures the MQTT protocol over IPv6, to temporarily disable IPv6 you can use the following command over the \textit{loopback} interface:

\texttt{sudo sh -c \textasciitilde echo 1 > /proc/sys/net/ipv6/conf/loopback/disable_ipv6}'

Having said that, we are going to access the Polymorph main interface, to do that, we only have to introduce the \texttt{polymorph} command from a Linux terminal.

```
root@kali:~# polymorph

POLYMORPH

< Santiago Hernandez Ramos >

PH > 
```

Once here, let’s start with the construction of the \textit{fuzzer}. As in this case we will not need to intercept the communication between two machines because the clients will be in localhost, we do not need to use any spoofing technique. We can simply use the \texttt{capture} command to start the packet sniffing process.

```
root@kali:~# polymorph

POLYMORPH

< Santiago Hernandez Ramos >

PH > capture
[*] Waiting for packets...

(Press Ctr-C to exit)
```

Our goal at this point is to capture one of the packets that we want to modify, so that the framework converts it into a \texttt{template} and we can work on it. Therefore, while the tool is sniffing packets, we place ourselves in our MQTT client and send an \texttt{MQTTPublish} message to the client that is listening.

Once this is done, we use \texttt{Ctr-C} to finish the sniffing process in Polymorph and use the \texttt{show} command to show the captured packets on the screen.
As you can see, most of the packets include a last Raw layer, which means that, at first glance, they have not been interpreted/dissected correctly by the primary dissectors. With the dissect command, we use more advanced dissectors that give us a representation of the part of the packets that have not been represented.
Now that we have a more concrete representation of all the bytes of the packets that we have captured, what we must do is choose the template that corresponds to the packet that we want to modify. We can use the wireshark command to open this application and perform a more detailed filtering. Once the template is selected, we access it using the template command.

Right now, we are in the context of the selected template. With the show command we can see the different layers and fields that it has, as well as the type of them. The template concept is the most important abstraction of the framework, and it is what allows the user to access the captured packets in real time using simple syntax in the code he writes to perform complex processing on them. Furthermore, it is the container in which all the conditional functions and structures of the framework are stored when we save a session.

Once this is done, the conditional functions come into play (preconditions, postconditions and executions). When the user enters the intercept command in the template interface, the machine that hosts Polymorph will stop forwarding the packets at the kernel level and start sending those packages to the tool to be processed before being forwarded. The conditional functions defined by the user will be executed in each of the intercepted packets.

Let's see a simple example of how these functions work, we are going to add the following precondition to our current template using the command precs -a test_condition.

if you are using the default editor, pico, remember not to mix tabs and spaces, better use only spaces to indent the code. (You can specify another editor that is in your PATH using the option -e):

```python
def test_precondition(packet):
    print("The next packet arrive:")
    print(packet.raw)
    return packet
```

Enter "y" to keep the code on disk and enter precs -s to visualize the added precondition.
Now, introduce the `intercept` command:

```bash
$ printf/tle > intercept -i iptables -A INPUT -j NFQUEUE --queue-num 1
[*] Waiting for packets...

(Press Ctrl-C to exit)
```

The next packet arrive:
```
0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000 0
0x0100100000000000 0x0100100000000000 0x0100100000000000 0x0100100000000000 0
0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000 0
```

Look how all the packets that flow through the machine are processed by the tool in real time and the precondition we have added is executed on each of them. We can test it just by sending a MQTT Publish message from one MQTT client to the broker.

The conditional functions are another important abstraction of the framework and work as follows. When a packet is intercepted in real time, the conditional functions defined by the user are executed on it following a certain order, first the `preconditions` are executed in the order in which the user has added them to the `template`, then the `executions` and finally the `postconditions`. If at any point of the execution of any of the three types of conditional functions the value `None` is returned by one of them, the execution chain is broken and the packet is forwarded as it is at that moment. On the other hand, if the packet that is received as an argument is returned, the chain of execution of conditions continues. Remember that the packet that is received as argument is the packet that has been intercepted in real time at that moment.

Once this is understood, we are going to exit the `intercept` mode in Polymorph by entering `Ctrl-C` (in this way, the machine that hosts Polymorph only forwards the packets without passing them through the tool). After that, we are going to add the following `preconditions`, `executions` and `postconditions`, which, I insist, when we start intercepting will be executed on each of the packets that are intercepted. To eliminate the test precondition that we added before, use `precs -d test_condition`.

### Preconditions

Two preconditions have been added using the commands:

```
precs -a global_vars -e editor
precs -a filter_mqttpublish -e editor
```

The first precondition, `global_vars`, is creating a global variable that will remain constant for all intercepted packets. It will be used to store all the test cases that we will use to fuzz the `MQTTPublish` packets.
On the other hand, the second precondition, `filter_mqttpublish`, will filter the incoming packets so that they only continue executing the rest of the conditional functions those whose `msgtype` field is equals to 48. Notice that thanks to the `template` abstraction, Polymorph knows the position that the `msgtype` field occupies within the bytes of the intercepted packet, and therefore, the user can access it much more easily.

```python
def filter_mqttpublish(packet):
    try:
        if packet['RAW.MQTT']['msgtype'] == 48:
            return packet
    except:
        return None
```

**Executions**

The execution is a bit longer than the preconditions, but it remains simple. The piece of code shown below performs the following tasks:
1. Transforms the intercepted packet into a Scapy representation. We do this to be able to interact more easily with the fields of the MQTT layer, especially with the lengths, which are encoded. I wrote the MQTT specification for Scapy a while ago, you can find it [here](#).
2. We check if fuzzing values remain in our list of test cases. The list is stored in the global variable created above. If the list is empty, we invoke Radamsa to generate more test cases and we stored them in the global variable.
3. Finally, we use Scapy to insert the fuzzing value in the `msg` field of the packet and we eliminate that value from our list, so that it is not inserted twice. In addition, we recalculate the control fields, such as `lengths` and `chksums`.

```python
def insert_value(packet):
    import subprocess
    from os import listdir
    from os.path import join
    from scapy.all import IP
    from scapy.contrib.mqtt import MQTT
    # Building a Scapy packet
    pkt = IP(packet.raw[14: ])
    # Retrieving the fuzzing case
    if not packet.fuzz_cases:
        valid_cases = "valid_cases"
        dpath = "fuzz_cases"
        subprocess.check_call(['radamsa',
            '-o',
            join(dpath, "fuzz-%n.%s"),
            '-n',
            '58',
            '-r',
            valid_cases])
        packet.fuzz_cases = [open(join(dpath, f), 'rb').read() for f in listdir(dpath)]
```
24. # Inserting the value and recalculating some fields
del pkt['IP'].len
del pkt['IP'].chksum
del pkt['TCP'].chksum
del pkt['MQTT'].len
del pkt['MQTTPublish'].length
pkt['MQTTPublish'].topic = packet.fuzz_cases.pop()
pkt.show2()
packet.raw = bytes(pkt)
return packet

That’s all we need to build a **Proxy Fuzzer** for the MQTT protocol using Polymorph! To put it into operation, we will create two directories in the PATH from which we have run Polymorph, one called **valid_cases** and another called **fuzz_cases**. These directories will be used by Radamsa to read valid test cases and mutate them in cases that may unravel in a possible vulnerability. We can add some valid cases like the following ones.

```bash
root@kali:~/valid_cases# ls
test1.txt test2.txt test3.txt
root@kali:~/valid_cases# cat test1.txt
test1.txt
root@kali:~/valid_cases# cat test3.txt
mensaje
root@kali:~/valid_cases# cat test3.txt
mensaje
hello world
root@kali:~/valid_cases#
```

Once this is done we simply go to the **template** interface in Polymorph and enter the **intercept** command. After that, we go to our MQTT client and publish a message, preferably with a long value so that there are no problems with the sequence numbers of the TCP/IP session.

We can observe how the packet is modified in real time and the value produced by Radamsa is introduced. What we could do now is making a simple loop in **Bash** and let it test a significant number of test values. Also, the most common thing would be that we run the
application that we are testing under a debugger, so we can capture the exceptions that occur and analyze them.

```bash
#!/bin/bash

while true; do
    mosquitto_pub -t `python3 -c "print('A'*10000)"` -m 'hello'
done
```

Finally, we can use the `save` command from the `template` interface to export the `template` and import it into Polymorph with the command `polymorph -t template.json` in another machine, so you can share it with your colleagues! I leave mine here!

https://gist.github.com/shramos/2b98867d2c344b36bfee6a7c799fbb8f