Polymorph 2.0: Advanced manipulation of network traffic in real time

@santiagohramos

https://github.com/shramos/polymorph

Content

Introduction

Polymorph is a tool that facilitates the modification of network traffic on the fly by allowing the execution of Python code on network packets that are intercepted in real time.

This framework can be used to modify network packets that implement any publicly specified network protocol. Additionally, it can be used to modify privately specified network protocols by creating custom abstractions and fields.

This new version of Polymorph greatly improves the capabilities of the previous version and facilitates the use of the tool. The following sections present different case studies with which you can learn how to use Polymorph 2.0.

Setting up the environment

Before starting with the first section, you must install the tool. To do so, install the following requirements on a Linux operating system:

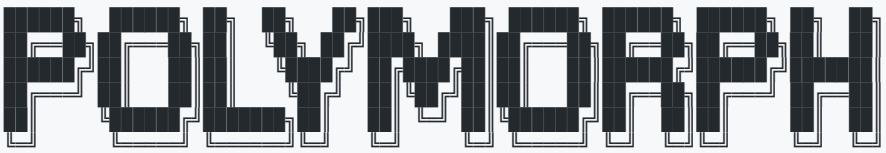
```
apt-get install build-essential python-dev libnetfilter-queue-dev tshark tcpdump python3-pip
wireshark
```

After installing the above requirements, install Polymorph by running the following commands in the terminal:

```
pip3 install git+https://github.com/kti/python-netfilterqueue
pip3 install polymorph
```

If everything went well, you can access Polymorph by using the polymorph command on a terminal of your operating system.

```
kali@kali:~$ polymorph
```




```
                                        < Santiago Hernandez Ramos >

PH >
```

Case Study: Modifying ICMP network packets in real time (I)

Introduction: How does Polymorph work?

The way Polymorph works is quite simple. The first step is to capture a network packet equivalent to the type of network packets you wish to modify on the fly afterwards. To do this, we execute the following command in Polymorph's main interface:

```
PH > capture -f icmp
[+] Waiting for packets...

(Press Ctr-C to exit)
```

The ***capture*** command performs network traffic sniffing in the same way that other utilities such as wireshark or tcpdump do. With the ***-f*** option you can set a filter for the network packets you want to capture. For more information on this command, you can use the ***-h*** option.

After running the ***capture*** command, the next thing we need to do is generate network traffic containing network packets equivalent to the ones we want to modify on the fly. In this case, we generate ICMP traffic.

```
root@kali:/home/kali# ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.020 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.044 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.075 ms
^C
--- localhost ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2039ms
rtt min/avg/max/mdev = 0.020/0.046/0.075/0.022 ms
```

Once the ICMP traffic is generated, we can stop Polymorph's sniffing process with the combination ***ctrl + C***, Polymorph will parse the captured network packets and generate a data structure called **template**.

```
PH > capture -f icmp
[+] Waiting for packets...

(Press Ctr-C to exit)

[+] Parsing packet: 12
[+] Parsing complete!
PH:cap >
```

Template

The **template** has a dissection of all the layers and fields of the different network protocols that implement the captured network packets. Among the attributes it includes for each of the fields are: the field type, the fixed position it occupies within the packet bytes, its representation as a function of type, its value in bytes...

To view all the templates that have been generated from the capture made, we can use the ***show*** command:

```
PH:cap > show
1 Template: ETH / IP / ICMP
2 Template: ETH / IP / ICMP
3 Template: ETH / IP / ICMP
4 Template: ETH / IP / ICMP
5 Template: ETH / IP / ICMP
6 Template: ETH / IP / ICMP
7 Template: ETH / IP / ICMP
8 Template: ETH / IP / ICMP
9 Template: ETH / IP / ICMP
10 Template: ETH / IP / ICMP
11 Template: ETH / IP / ICMP
12 Template: ETH / IP / ICMP

PH:cap >
```

At this point, what we must do is select the template that corresponds to the type of network packet that we would like to modify on the fly. To do this we use the ***template*** command followed by the template number.

If we want to see more detail about which network packet each of the generated templates corresponds to, we can use the ***wireshark*** command that will open this application with the captured network packets.

```
PH:cap > show
1 Template: ETH / IP / ICMP
2 Template: ETH / IP / ICMP
3 Template: ETH / IP / ICMP
4 Template: ETH / IP / ICMP
5 Template: ETH / IP / ICMP
6 Template: ETH / IP / ICMP
7 Template: ETH / IP / ICMP
8 Template: ETH / IP / ICMP
9 Template: ETH / IP / ICMP
10 Template: ETH / IP / ICMP
11 Template: ETH / IP / ICMP
12 Template: ETH / IP / ICMP

PH:cap > template 1
PH:cap/t1 >
```

After doing the above, we are in the interface that will allow us to perform different actions on the selected template and modify network packets on the fly using this template as a reference. Let's start by visualizing the template's content with the ***show*** command:

```
PH:cap/t1 > show

---[ ETH ]---
FT_ETHER dst          = 00:00:00:00:00:00
FT_ETHER src          = 00:00:00:00:00:00
FT_HEX type           = 0x800

---[ IP ]---
FT_BIN_BE version     = 4
FT_BIN_BE hdr_len     = 5
FT_HEX dsfield        = 0x0
FT_INT_BE len         = 84
FT_HEX id             = 0xa3fd
FT_HEX flags          = 0x4000
FT_INT_BE ttl         = 64
FT_INT_BE proto       = 1
FT_HEX checksum       = 0x98a9
FT_IPv4 src           = 127.0.0.1
FT_IPv4 addr          = 127.0.0.1

---[ ICMP ]---
FT_INT_BE type        = 8
FT_INT_BE code        = 0
FT_HEX checksum       = 0x41a8
FT_INT_BE ident       = 13880
FT_INT_BE seq         = 1
FT_ABSOLUTE_TIME data_time= Sep 30, 2020 13:35:20.000000
FT_BYTES data         =
b'j*\n\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./01234567'
```

As can be seen, Polymorph has dissected the ICMP network packet by identifying a set of layers (Ethernet, IP, ICMP) and a set of fields for each of the layers.

Polymorph now "knows" what an ICMP network packet looks like, what position each of the fields occupies within the network packet bytes, and stores as a reference the values that the captured network packet had for each of the fields.

The great advantage of this, is that now Polymorph will allow us to modify network packets in real time by accessing each of the fields of the captured network packet using the names shown in the template for each of the fields.

Functions

To understand how this works, we are going to add one of the fundamental components of the framework, a **function**. As the name implies, functions are code snippets in Python 3 that will be executed on network packets in transit. We can add as many functions as we want, each of which will be executed sequentially on the intercepted network packets.

To add a function, we execute the ***functions*** command as shown below. The ***-a*** option indicates that we are adding a new function, the ***-e*** option indicates the text editor with which we want to edit this function.

```
PH:cap/t1 > functions -a filter_icmp_packets -e emacs
```

When this command is executed, the selected text editor will be opened with a code skeleton in Python 3 similar to the following one:

```
def filter_icmp_packets(packet):

    # your code here

    # If the condition is meet
    return packet
```

The three most important things we should know about functions are

- The packet parameter represents the packet that is being intercepted in real time at that moment
- We can access the contents of the packet that is being intercepted through the name of the layers and fields that have been generated in the template
- If we want to continue executing other functions that we have added, the function must return packet, otherwise, it must return None.

With these main points in mind, we are going to create our first function that will take care of filtering ICMP network packets request and reply.

```
def filter_icmp_packets(packet):
    try:
        if packet['IP']['proto'] == 1:
            if packet['ICMP']['type'] == 8:
                print("ICMP Request. Executing next function...")
                return packet
            elif packet['ICMP']['type'] == 0:
                print("ICMP Reply. Executing next function...")
                return packet
    except:
        return None
```

Save, close the text editor, and the function will be automatically added to Polymorph.

```
PH:cap/t1 > functions -a filter_icmp_packets -e emacs
[+] Function filter_icmp_packets added
```

We can visualize the functions that we have active and the order with the command ***functions -s***:

```
PH:cap/t1 > functions -s
+-------+-------------------------------------------------------------------+
| Order |                            Functions                              |
+=======+===================================================================+
| 0     | def filter_icmp_packets(packet):                                  |
|       |     try:                                                          |
|       |         if packet['IP']['proto'] == 1:                            |
|       |             if packet['ICMP']['type'] == 8:                       |
|       |                 print("ICMP Request. Executing next function...") |
|       |                 return packet                                     |
|       |             elif packet['ICMP']['type'] == 0:                     |
|       |                 print("ICMP Reply. Executing next function...")   |
|       |                 return packet                                     |
|       |     except:                                                       |
|       |         return None                                               |
|       |                                                                   |
+-------+-------------------------------------------------------------------+
```

Intercepting and modifying network packets in real time

Once the function is added, it is time to test it by intercepting network traffic in real time. To do this, we execute the ***intercept -localhost*** command, which will put Polymorph in intercept mode in the loopback interface.

```
PH:cap/t1 > intercept -localhost
[*] Waiting for packets...

(Press Ctrl-C to exit)
```

Now all we have to do is generate network traffic on the system and check the execution of the function on the network packets. If all goes well, we should see a result similar to the one shown below when we generate ICMP traffic by executing the ***ping localhost*** command on a terminal.

```
PH:cap/t1 > intercept -localhost
[*] Waiting for packets...

(Press Ctrl-C to exit)

ICMP Request. Executing next function...
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
ICMP Reply. Executing next function...
```

The possibilities with functions are as many as we can think of, for example, we are going to create a new function to modify the data field of the ICMP Request network packets.

```
def modify_icmp(packet):
    if packet['ICMP']['type'] == 8:
        packet['ICMP']['data'] = packet['ICMP']['data'][:-8] + b'newvalue'
        print("New value inserted")
        return packet
```

The most important thing to keep in mind, is that both to compare a value, as to add a new value to a field, we must respect the type shown in the template. In this case the type is ***FT_BYTES*** that indicates us that this field must receive a value in bytes.

We execute again the command ***intercept -localhost*** and generate ICMP traffic.

```
PH:cap/t1 > intercept -localhost
[*] Waiting for packets...

(Press Ctrl-C to exit)

ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
```

We can observe how after executing the command ***ping localhost*** the machine does not receive a response, this is because we are modifying the content of the ping request packets on the fly, which causes that when the ping reply arrives the value of the data field does not coincide with the one originally sent. We can check this if we use a utility such as wireshark at the same time as modifying the network packets with Polymorph.

```
0000   00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010   00 54 d3 6a 40 00 40 01 69 3c 7f 00 00 01 7f 00   .T.j@.@.i<......
0020   00 01 08 00 70 fe 37 3b 00 01 5e ce 74 5f 00 00   ....p.7;..^.t_..
0030   00 00 ca e7 03 00 00 00 00 00 10 11 12 13 14 15   ................
0040   16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25   .......... !"#$%
0050   26 27 28 29 2a 2b 2c 2d 2e 2f 6e 65 77 76 61 6c   &'()*+,-./newval
0060   75 65                                            ue
```

With the command ***Ctr + C*** we leave the intercept mode and with the command ***save -p path*** we save the template on disk to be able to use it later without the need to add the functions again.

```
PH:cap/t1 > save -p icmp_template
[+] Template saved to disk
```

Finally, we can exit Polymorph with the ***exit*** command.

```
PH:cap/t1 > exit
Are you sure you want to exit? (y/n) y

Bye Bye! See you soon!
```

Case Study: Modifying ICMP network packets in real time (II)

Introduction

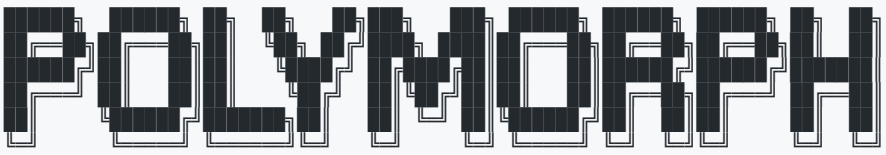
Let's continue with the exercise explained in the previous section and try to make additional on-the-fly modifications to the ICMP network packets so that the ***ping -localhost*** command works properly.

The problem we have with the previous exercise is that we are modifying on the fly the value of the data field of the ICMP Request network packets, and therefore, the ICMP Reply contains a value in the data field that differs from the original. (This can be checked by using tools such as wireshark).

To remedy this, we will make an on-the-fly modification to the data field value of the ICMP Reply network packets by inserting the original value that the ICMP Request network packets had before making the modification.

Importing a template saved on disk

We start by importing the template we have saved in the previous case study, this can be done with the ***import*** command from the main Polymorph interface:




```
PH >
PH >
PH > import -t icmp_template
PH:cap/t0 >
```

Once the template has been imported, we can check that the previously generated functions have already been automatically added with the command ***functions -s***.

```
PH:cap/t0 > functions -s
+-------+-------------------------------------------------------------------------+
| Order |                              Functions                                  |
+=======+=========================================================================+
| 0     | def filter_icmp_packets(packet):                                        |
|       |     try:                                                                |
|       |         if packet['IP']['proto'] == 1:                                  |
|       |             if packet['ICMP']['type'] == 8:                             |
|       |                 print("ICMP Request. Executing next function...")       |
|       |                 return packet                                           |
|       |             elif packet['ICMP']['type'] == 0:                           |
|       |                 print("ICMP Reply. Executing next function...")         |
|       |                 return packet                                           |
|       |     except:                                                             |
|       |         return None                                                     |
|       |                                                                         |
+-------+-------------------------------------------------------------------------+
| 1     | def modify_icmp(packet):                                                |
|       |     if packet['ICMP']['type'] == 8:                                     |
|       |         packet['ICMP']['data'] = packet['ICMP']['data'][:-8] +          |
|       | b'newvalue'                                                             |
|       |         print("New value inserted")                                     |
|       |         return packet                                                   |
|       |                                                                         |
+-------+-------------------------------------------------------------------------+

PH:cap/t0 >
```

In order to replace the value of the ICMP Reply network packets with the original value of the ICMP Request network packets, we must somehow "remember" the value of the ICMP Request network packet before making the modification on the fly. To do this we will use a **global variable**.

Global Variables

Global variables allow us to store values that will persist between executed functions and intercepted network packets.

To modify the function ***modify_icmp*** we execute the same command that we used to create it.

```
PH:cap/t0 > functions -a modify_icmp -e emacs
```

This sentence will open the function that we indicate with the selected text editor. The new function should be similar to the one shown below.

```
def modify_icmp(packet):
    # ICMP Request network packet
    if packet['ICMP']['type'] == 8:
        packet.global_var('orig_data', packet['ICMP']['data'])
        packet['ICMP']['data'] = packet['ICMP']['data'][:-8] + b'newvalue'
        print("New value inserted")
    # ICMP Reply network packet
    elif packet['ICMP']['type'] == 0:
        packet['ICMP']['data'] = packet.orig_data
        print("Original value inserted")
    return packet
```

As we can see, before inserting the new value in the ICMP Request network packet, we save the original value in the global variable ***orig_data*** which we later use to replace the value in the data field of the ICMP Reply packets.

The only thing we have to do at this point is to execute the ***intercept -localhost*** command in Polymorph and generate ICMP traffic executing the ***ping localhost*** command in a terminal of our operating system.

```
PH:cap/t0 > intercept -localhost
[*] Waiting for packets...

(Press Ctrl-C to exit)

ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
Original value inserted
ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
Original value inserted
ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
Original value inserted
ICMP Request. Executing next function...
New value inserted
ICMP Reply. Executing next function...
Original value inserted
```

At this point we can see how we are making a double modification on the fly of ICMP network packets. On the one hand, we insert on the fly a new value in the data field of the ICMP Request network packets, and, on the other hand, when the system generates the ICMP Reply packet with the modified value, we modify on the fly this network packet to insert the original value that the data field had before the first modification.

Case Study: Modifying MQTT network packets in real time

Introduction

In the previous case study we have made an on-the-fly modification of ICMP network packets directed to the same machine (localhost). In this section, we are going to see another use case in which we will have two machines that exchange information through the MQTT network protocol and another machine with Polymorph installed. The aim of the exercise is to modify on the fly the information that is exchanged between the first two machines.

Setting up the environment

The environment used for the development of this case study is quite simple. On the one hand, we have two linux machines (***192.168.71.130***, ***192.168.71.138***) that will communicate through the MQTT protocol. In both machines we must install the utilities that allow us to make this communication:

sudo apt install mosquitto mosquitto-clients

On the other hand, we have the machine ***192.168.71.131*** that will have Polymorph installed and will be in charge of intercepting the communication between the other two and modifying on the fly the network traffic exchanged between them.

To test the communication by MQTT between the two machines (***192.168.71.130, 192.168.71.138***), from now on **machine A** and **machine B**, we open two terminals in **machine A** and we execute, on one hand, the command ***mosquitto*** that will activate the broker in charge of establishing the communications, and, on the other hand, the command ***mosquitto_sub -t test*** that will cause the machine to wait for messages directed to the topic test.

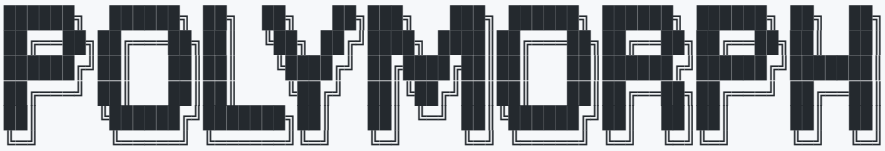
In the **machine B** we open a terminal and we execute the command ***mosquitto_pub -t test -m hello -h 192.168.71.130*** that will cause the sending of the message hello to the machines subscribed under the topic test.

If all went well, machine A should have received the *hello* message. Our objective in this practical case will be to modify this message on the fly.

Intercepting communication between machine A and machine B

This practical case differs from the one presented in the previous section in that communication is not on the same machine but between two different machines. Therefore, in order for Polymorph to "see" the traffic exchanged between the two and modify it on the fly, we must intercept the communication.

There are several ways of intercepting communication between two machines, one of the most common, and the one we are going to use in this practical case, is ARP poisoning. We can use this technique from Polymorph with the ***spoof*** command:




```
PH > spoof -t 192.168.71.130 -g 192.168.71.138
[+] ARP spoofing started between 192.168.71.138 and 192.168.71.130
```

At this point, the network traffic exchanged between machines A and B passes through the machine on which Polymorph is installed (***192.168.71.131***), from now on ***machine C***.

Modifying MQTT network packets in real time

Once we are in the middle of the communication between machine A and machine B, the rest of the steps that we must follow are the same as those indicated in the previous case study.

First, we must capture a MQTT Publish network packet, which is the network packet that contains the message:

```
PH > capture -f mqtt
[+] Waiting for packets...

(Press Ctr-C to exit)

[+] Parsing packet: 4
[+] Parsing complete!
PH:cap > s
1 Template: ETH / IP / TCP / MQTT
2 Template: ETH / IP / TCP / MQTT
3 Template: ETH / IP / TCP / MQTT
4 Template: ETH / IP / TCP / MQTT

PH:cap > template 3
PH:cap/t3 > s

---[ ETH ]---
FT_ETHER dst          = 00:0c:29:72:3c:22
FT_ETHER src          = 00:0c:29:54:0d:00
FT_HEX type           = 0x800

---[ IP ]---
FT_BIN_BE version     = 4
FT_BIN_BE hdr_len     = 5
FT_HEX dsfield        = 0x2
FT_INT_BE len         = 65
FT_HEX id             = 0x8a44
FT_HEX flags          = 0x4000
FT_INT_BE ttl         = 64
FT_INT_BE proto       = 6
FT_HEX checksum       = 0xa013
FT_IPv4 src           = 192.168.71.138
FT_IPv4 addr          = 192.168.71.130

---[ TCP ]---
FT_INT_BE srcport     = 50286
FT_INT_BE dstport     = 1883
FT_HEX len            = 0x80
FT_HEX seq            = 0x7e4efe7b
FT_HEX ack            = 0x532c2b04
FT_BIN_BE flags       = 24
FT_INT_BE window_size_value= 229
```

```
FT_HEX checksum       = 0x98e2
FT_INT_BE urgent_pointer= 0
FT_BYTES options      = b'\x01\x01\x08\nP,\xc1\xa1\xe3\xf9\xb4;'
FT_BYTES payload      = b'0\x0b\x00\x04testhello'

---[ MQTT ]---
FT_HEX hdrflags       = 0x30
FT_INT_BE len         = 11
FT_INT_BE topic_len   = 4
FT_STRING topic       = test
FT_BYTES msg          = b'hello'

PH:cap/t3 >
```

And then we must add the functions that are in charge of filtering and modifying on the fly this type of network packets:

```
def filter_mqtt_pub(packet):
    try:
        if packet["TCP"]["dstport"] == 1883:
            if packet["MQTT"]["hdrflags"] == "0x30":
                print("Topic:", packet["MQTT"]["topic"])
                print("Msg:", packet["MQTT"]["msg"])
                return packet
    except:
        return None
```

```
def mod_mqtt_pub(packet):
    packet['MQTT']['msg'] = b'hhhhh'
    print("New value inserted\n")
    return packet
```

After adding the functions, we execute the ***intercept*** command in Polymorph to wait for a network packet with these characteristics to be sent and make the modification on the fly.

If everything goes well, when we send the ***hello*** message again via MQTT between machine A and B, the modification will be made and machine B will receive the ***hhhhh*** message.

```
PH:cap/t3 > intercept
[*] Waiting for packets...

(Press Ctrl-C to exit)

Topic: test
Msg: b'hello'
New value inserted
```

Structs: Recalculating the length of a field dynamically

So far, there is not much variation from the case study seen in previous sections. However, in the case of ICMP network packets, the data value was a fixed value that was always the same size, in this case, the msg field can be modified by the user and therefore its value is variable. This means that **the value of the *msg* field received in the intercepted network packet in real time may differ from the one found in the generated template**.

If we make a test with the exercise as we have it right now, and we modify the message sent by machine A to machine B so that it is ***hello how are you*** instead of ***hello***:

```
mosquitto_pub -t test -m "hello how are you" -h 192.168.71.130
```

We will notice that Polymorph makes a modification, but does not modify the entire message:

```
santi@lubuntu:~$ mosquitto_sub -t test
hhhhh how are you
```

This is caused because the templates have static values that represent the position of the field in the network packet as a function of the captured network packet. In the captured network packet, the ***msg*** field received the value ***hello*** and therefore had a size of 5 bytes in that network packet.

To solve this type of problem, Polymorph implements the concept of **struct**. These structures allow us to declare expressions that reevaluate the size of fields in real time. The structs are associated with a field within a given layer within the template and require the name of the field to be recalculated, the beginning byte, and the expression that recalculates its size.

For our particular case of study, we can define a **struct** for the ***msg*** field in the following way:

```
PH:cap/t3 > layer mqtt
PH:cap/t3/MQTT > struct -f msg -sb "70 + this.topic_len" -e "this.len - 2 - this.topic_len"
[+] Struct added to field msg

PH:cap/t3/MQTT >
```

We can test if the struct has been created correctly with the ***struct -t msg*** command, which should return the value of the msg field in the template.

```
PH:cap/t3/MQTT > struct -t msg
b'hello'
```

Once this struct is added, the msg field of the network packets intercepted in real time will be recalculated on the fly in the manner indicated before any action is taken on it.

It must be taken into account that if the original value is greater than the value inserted on the fly, we must recalculate other control fields of the packet, such as the *len* field of the IP layer or the *len* field of the MQTT layer.

The new function ***mod_mqtt_pub*** would be the following:

```
def mod_mqtt_pub(packet):
    # Calculating size difference
    orig_size = len(packet['MQTT']['msg'])
    new_value = b'hhhhh'
    diff = len(new_value) - orig_size
    # Inserting new values
    packet['MQTT']['msg'] = new_value
    packet['IP']['len'] += diff
    packet['MQTT']['len'] += diff
    print("New value inserted\n")
    return packet
```

Once this function has been modified, we execute the ***intercept*** command in Polymorph and generate the message ***hello how are you*** from machine A to machine B.

```
PH:cap/t3 > intercept
[*] Waiting for packets...

(Press Ctrl-C to exit)

Topic: test
Msg: b'hello how are you'
New value inserted
```

After the execution, we can see how Polymorph has been able to interpret the new value of the msg field of the network packet intercepted on the fly and modify it to enter a smaller ***hhhhh*** value that corresponds to the one received by machine B.

```
santi@lubuntu:~$ mosquitto_sub -t test
hhhhh
```

To save the template we can use the command ***save -p mqtt_template***.

Creating custom layers and fields

Introduction

Finally, I would like to present the use case where, for some reason (e.g. if it is a network protocol without a public specification) Polymorph is not able to dissect all the layers and fields of the network packet when the template is generated.

For these cases, Polymorph provides us with the ability to create new layers and fields within the template or to modify the type of fields already present.

Modifying the type of a field

Modifying the type of a field that is presented in the template is something trivial but can be very useful when building the functions. **Remember that in the functions we must always consider that the fields we access have the type indicated in the template**.

In the example shown below we change the type of the field ***msg*** to be a string instead of a set of bytes.

```
PH:cap/t3 > layer mqtt
PH:cap/t3/MQTT > show

---[ MQTT ]---
FT_HEX hdrflags       = 0x30
FT_INT_BE len         = 11
FT_INT_BE topic_len   = 4
FT_STRING topic       = test
FT_BYTES msg          = b'hello'

PH:cap/t3/MQTT > field msg
PH:cap/t3/MQTT/msg > type -a

1: FT_INT_BE
2: FT_INT_LE
3: FT_STRING
4: FT_BYTES
5: FT_BIN_BE
6: FT_BIN_LE
7: FT_HEX
8: FT_ETHER
9: FT_IPv4
10: FT_IPv6
11: FT_ABSOLUTE_TIME
12: FT_RELATIVE_TIME
13: FT_EUI64

Select the type of the field: 3
[+] New type Ftype.FT_STRING added to the field.

PH:cap/t3/MQTT/msg > back
PH:cap/t3/MQTT > show

---[ MQTT ]---
FT_HEX hdrflags       = 0x30
FT_INT_BE len         = 11
FT_INT_BE topic_len   = 4
FT_STRING topic       = test
FT_STRING msg         = hello

PH:cap/t3/MQTT >
```

Adding new layers and fields

Adding new layers and fields is something very useful when Polymorph is not able to dissect the entire network packet, usually this happens in protocols without public specification.

To add a new layer, all we have to do is execute the command ***layer -a new_layer***, Polymorph will ask us for some values, such as the layer's position within the bytes of the network packet.

```
PH:cap/t3 > layer -a new_layer
00000000: 00 0C 29 72 3C 22 00 0C  29 54 0D 00 08 00 45 02  ..)r<"..)T....E.
00000010: 00 41 8A 44 40 00 40 06  A0 13 C0 A8 47 8A C0 A8  .A.D@.@.....G...
00000020: 47 82 C4 6E 07 5B 7E 4E  FE 7B 53 2C 2B 04 80 18  G..n.[~N.{S,+...
00000030: 00 E5 98 E2 00 00 01 01  08 0A 50 2C C1 A1 E3 F9  ..........P,....
00000040: B4 3B 30 0B 00 04 74 65  73 74 68 65 6C 6C 6F     .;0...testhello

Start byte of the custom layer: 50
End byte of the custom layer: 79
[+] New layer new_layer added to the Template

PH:cap/t3 > show

---[ ETH ]---
FT_ETHER dst          = 00:0c:29:72:3c:22
FT_ETHER src          = 00:0c:29:54:0d:00
FT_HEX type           = 0x800

---[ IP ]---
FT_BIN_BE version     = 4
FT_BIN_BE hdr_len     = 5
FT_HEX dsfield        = 0x2
FT_INT_BE len         = 65
FT_HEX id             = 0x8a44
FT_HEX flags          = 0x4000
FT_INT_BE ttl         = 64
FT_INT_BE proto       = 6
FT_HEX checksum       = 0xa013
FT_IPv4 src           = 192.168.71.138
FT_IPv4 addr          = 192.168.71.130

---[ TCP ]---
FT_INT_BE srcport     = 50286
FT_INT_BE dstport     = 1883
FT_HEX len            = 0x80
FT_HEX seq            = 0x7e4efe7b
FT_HEX ack            = 0x532c2b04
FT_BIN_BE flags       = 24
FT_INT_BE window_size_value= 229
FT_HEX checksum       = 0x98e2
FT_INT_BE urgent_pointer= 0
FT_BYTES options      = b'\x01\x01\x08\nP,\xc1\xa1\xe3\xf9\xb4;'
FT_BYTES payload      = b'0\x0b\x00\x04testhello'

---[ MQTT ]---
FT_HEX hdrflags       = 0x30
FT_INT_BE len         = 11
FT_INT_BE topic_len   = 4
FT_STRING topic       = test
FT_STRING msg         = hello

---[ NEW_LAYER ]---

PH:cap/t3 >
```

Creating a new field is also a simple task, all we have to do is access the layer where we want to create the new field and execute the command ***field -a new_field***.

```
PH:cap/t0 > layer new_layer
PH:cap/t0/NEW_LAYER > field -a new_field
00000000: 00 0C 29 72 3C 22 00 0C  29 54 0D 00 08 00 45 02  ..)r<"..)T....E.
00000010: 00 41 8A 44 40 00 40 06  A0 13 C0 A8 47 8A C0 A8  .A.D@.@.....G...
00000020: 47 82 C4 6E 07 5B 7E 4E  FE 7B 53 2C 2B 04 80 18  G..n.[~N.{S,+...
00000030: 00 E5 98 E2 00 00 01 01  08 0A 50 2C C1 A1 E3 F9  ..........P,....
00000040: B4 3B 30 0B 00 04 74 65  73 74 68 65 6C 6C 6F     .;0...testhello

Start byte of the custom field: 74
End byte of the custom field: 79

1: FT_INT_BE
2: FT_INT_LE
3: FT_STRING
4: FT_BYTES
5: FT_BIN_BE
6: FT_BIN_LE
7: FT_HEX
8: FT_ETHER
9: FT_IPv4
10: FT_IPv6
11: FT_ABSOLUTE_TIME
12: FT_RELATIVE_TIME
13: FT_EUI64

Select the type of the field: 3
[+] Field new_field added to the layer

PH:cap/t0/NEW_LAYER > show

---[ NEW_LAYER ]---
FT_STRING new_field   = hello

PH:cap/t0/NEW_LAYER >
```

All new layers and fields created support the same manipulations and accesses as the original layers and fields dissected by Polymorph.