Python绝技之端口扫描

0x01 前置知识

optparse库

作用：为脚本传递命令参数功能

```
parser = optparse.OptionParser("[*] Usage : ./portscanner.py -u <target
host> -p <target port>")
//添加使用信息，随便你写不写

parser.add_option('-u',dest='tgtHost',type='string',help='specify target
host')
//是为解析器添加选项，定义命令行参数（重点）
name or flags,nargs,action,desk,const,default,type,choices,help
下面一个一个说明：
name or flags:就是参数的名称或标志 -f --file,-q --quit 等
nargs：命令行参数的个数，一般使用通配符表示，其中，'?'表示只用一个，'*'表示0到多
个，'+'表示至少一个
action:存储方式,指定接收一个参数时如何处理,store(存储在变量dest里)store_true(设置
dest为true),
       store_false(设置dest为false),store_const(需要与const配合),append(将参数追
加到列表里),
       count(计数器+1),callback(调用某个特定函数)
dest:存储的变量
const:与action=const相配合,存储常量值
type:值的类型
default:默认值
help:帮助提示信息
metavar:提醒所期待参数(会变成大写)
如果我们需要多个参数,就需要添加多个add_option()语句

(options, args) = parser.parse_args()  来解析命令行的参数,并将参数值保存到options
中,
```

实例：

```
def main():
    parser = optparse.OptionParser("[*] Usage : ./portscanner.py -u <target host>
 -p <target port>")
    parser.add_option('-u',dest='tgtHost',type='string',help='specify target host
')
    parser.add_option('-p',dest='tgtPort',type='string',help='specify target port
[s]')
    (options,args) = parser.parse_args()
```

```
┌──(root💀kali)-[/home/kali/Desktop/py3]
└─# python3 port_scan.py -h 192.168.179.1 -p 20,21,22,80,443
Usage: [*] Usage : ./portscanner.py -u <target host> -p <target port>

Options:
  -h, --help  show this help message and exit
  -u TGTHOST  specify target host
  -p TGTPORT  specify target port[s]
```

可以看到这里存在一个默认参数，-h，这个不可以设置，否则会报错。

第一个表示参数的名称，第二个dest就是默认储存在tgtHost这个变量中，第三个指明值的类型，help就是后面的帮助信息。

最后一个就是启动参数并且接受。

> 其实在python2中使用的是optparse库，在python3中使用的是argparse库

queue库

简介：

queue是python中的标准库，俗称队列。

在python中，多个线程之间的数据是共享的，多个线程进行数据交换的时候，不能够保证数据的安全性和一致性，所以当多个线程需要进行数据交换的时候，队列就出现了，队列可以完美解决线程间的数据交换，保证线程间数据的安全性和一致性。

queue模块有三种队列以及构造函数：

FIFO:先进先出队列

```
queue.Queue(max)
```

LIFO:先进后出

```
queue.LifoQueue(max)
```

级别越低越先出来

```
queue.PriorityQueue(max)
```

queue常用方法：

queue.qsize() 返回队列的大小

queue.empty() 如果队列为空，返回True,反之False

queue.full() 如果队列满了，返回True,反之False

queue.full 与 maxsize 大小对应

queue.get([block[, timeout]])获取队列，立即取出一个元素， timeout超时时间

queue.put(item[, timeout]]) 写入队列，立即放入一个元素， timeout超时时间

queue.get_nowait() 相当于queue.get(False)

queue.put_nowait(item) 相当于queue.put(item, False)

queue.join() 阻塞调用线程，直到队列中的所有任务被处理掉, 实际上意味着等到队列为空，再执行别的操作

queue.task_done() 在完成一项工作之后，queue.task_done()函数向任务已经完成的队列发送一个信号

threading库

简介：

在python3中，通过该threading模块提供线程的功能。原来的thread模块已经废弃。但是，threading模块中有个Thread类是模块中最主要的线程类。

threading常用的方法和属性

方法与属性	描述
current_thread()	返回当前线程
active_count()	返回当前活跃的线程数，1个主线程+n个子线程
get_ident()	返回当前线程
enumerater()	返回当前活动 Thread 对象列表
main_thread()	返回主 Thread 对象
settrace(func)	为所有线程设置一个 trace 函数
setprofile(func)	为所有线程设置一个 profile 函数
stack_size([size])	返回新创建线程栈大小；或为后续创建的线程设定栈大小为 size
TIMEOUT_MAX	Lock.acquire(), RLock.acquire(), Condition.wait() 允许的最大超时时间

theading模块包含以下的类：

- Thread: 基本线程类
- Lock：互斥锁
- RLock：可重入锁，使单一进程再次获得已持有的锁(递归锁)
- Condition：条件锁，使得一个线程等待另一个线程满足特定条件，比如改变状态或某个值。
- Semaphore：信号锁，为线程间共享的有限资源提供一个"计数器"，如果没有可用资源则会被阻塞。
- Event：事件锁，任意数量的线程等待某个事件的发生，在该事件发生后所有线程被激活。
- Timer：一种计时器
- Barrier：Python3.2新增的"阻碍"类，必须达到指定数量的线程后才可以继续执行。

多线程

有两种方法来创建多线程：一种是继承Thread类，并重写它的run()方法；另一种是实例化threading.Thread对象时，将线程要执行的任务函数作为参数传入线程。（我们多数都是采用第二种）

> threading.Thread(self, group=None, target=None, name=None,agrs=(),kwargs=None, *, daemon=None)

- 参数group是保留的
- target是一个可调用的对象，在线程启动后执行（函数）
- name是现成的名字。默认值是“Thread-N”，N是一个数字
- args和kwargs分别表示调用target时的参数列表和关键字参数（参数）

Thread类常用的方法和属性

方法与属性	说明
start()	启动线程，等待CPU调度
run()	线程被cpu调度后自动执行的方法
getName()、setName()和name	用于获取和设置线程的名称。
setDaemon()	设置为后台线程或前台线程（默认是False，前台线程）。如果是后台线程，主线程执行过程中，后台线程也在进行，主线程执行完毕后，后台线程不论成功与否，均停止。如果是前台线程，主线程执行过程中，前台线程也在进行，主线程执行完毕后，等待前台线程执行完成后，程序才停止。
ident	获取线程的标识符。线程标识符是一个非零整数，只有在调用了start()方法之后该属性才有效，否则它只返回None。
is_alive()	判断线程是否是激活的（alive）。从调用start()方法启动线程，到run()方法执行完毕或遇到未处理异常而中断这段时间内，线程是激活的。
isDaemon()方法和daemon属性	是否为守护线程
join([timeout])	调用该方法将会使主调线程堵塞，直到被调用线程运行结束或超时。参数timeout是一个数值类型，表示超时时间，如果未提供该参数，那么主调线程将一直堵塞到被调线程结束。

自定义线程类

对于yhreading模块的thread类，本质上是执行了他的run()方法。因此可以自定义线程类，让他继承thread类，然后重新run()方法即可。

线程锁

由于线程之间的任务执行是CPU进行随机调度的，并且每个线程可能只执行了n条指令之后就被切换到别的线程了。当多个线程同时操作一个对象，如果没有很好地保护该对象，会造成程序结果的不可预期，这被称为“线程不安全”。**为了保证数据安全，我们设计了线程锁，即同一时刻只允许一个线程操作该数据。**线程锁用于锁定资源，可以同时使用多个锁，当你需要独占某一资源时，任何一个锁都可以锁这个资源，就好比你用不同的锁都可以把相同的一个箱子锁住是一个道理。

可以采用join()等待，可是这样多线程就变成了单线程，正确的做法是使用线程锁。threading中定义了几种线程锁：

- Lock 互斥锁
- RLock 可重入锁
- Semaphore 信号
- Event 事件
- Condition 条件
- Barrier “阻碍”

互斥锁

互斥锁是一种独占锁，同一时刻只有一个线程可以访问共享的数据。使用很简单，初始化锁对象，然后将锁当做参数传递给任务函数，在任务中加锁，使用后释放锁。

可重入锁

RLock的使用方法和Lock一模一样，只不过它支持重入锁。该锁对象内部维护着一个Lock和一个counter对象。counter对象记录了acquire的次数，使得资源可以被多次require。最后，当所有RLock被release后，其他线程才能获取资源。在同一个线程中，RLock.acquire()可以被多次调用，利用该特性，可以解决部分死锁问题。

信号

类名：BoundedSemaphore。这种锁允许一定数量的线程同时更改数据，它不是互斥锁。比如地铁安检，排队人很多，工作人员只允许一定数量的人进入安检区，其它的人继续排队。

事件

类名Event, 事件线程锁的运行机制：全局定义了一个Flag，如果Flag的值为False，那么当程序执行wait()方法时就会阻塞，如果Flag值为True，线程不再阻塞。这种锁，类似交通红绿灯（默认是红灯），它属于在红灯的时候一次性阻挡所有线程，在绿灯的时候，一次性放行所有排队中的线程。事件主要提供了四个方法set()、wait()、clear()和is_set()

- clear()方法会将事件的Flag设置为False
- set()方法会将Flag设置为True
- wait()方法将等待“红绿灯”信号
- is_set():判断当前是否"绿灯放行"状态

条件

类名：Condition。Condition称作条件锁，依然是通过acquire()/release()加锁解锁。

- wait([timeout])方法将使线程进入Condition的等待池等待通知，并释放锁。使用前线程必须已获得锁定，否则将抛出异常。
- notify()方法将从等待池挑选一个线程并通知，收到通知的线程将自动调用acquire()尝试获得锁定（进入锁定池），其他线程仍然在等待池中。调用这个方法不会释放锁定。使用前线程必须已获得锁定，否则将抛出异常。
- notifyAll()方法将通知等待池中所有的线程，这些线程都将进入锁定池尝试获得锁定。调用这个方法不会释放锁定。使用前线程必须已获得锁定，否则将抛出异常。

定时器

定时器Timer类是threading模块中的一个小工具，用于指定n秒后执行某操作。一个简单但很实用的东西。

通过with语句使用线程锁

所有的线程锁都有一个加锁和释放锁的动作，非常类似文件的打开和关闭。在加锁后，如果线程执行过程中出现异常或者错误，没有正常的释放锁，那么其他的线程会造到致命性的影响。通过with上下文管理器，可以确保锁被正常释放。

全局解释器锁（GIL）

- **在大多数环境中，单核CPU情况下，本质上某一时刻只能有一个线程被执行。**多核CPU时，则可以支持多个线程同时执行。**但是在Python中，无论CPU有多少核，同时只能执行一个线程，这是由于GIL的存在导致的。**
- GIL的全称是Global Interpreter Lock(全局解释器锁)，是Python设计之初为了数据安全所做的决定。**Python中的某个线程想要执行，必须先拿到GIL。可以把GIL看作是执行任务的"通行证"，并且在一个Python进程中，GIL只有一个。**拿不到通行证的线程，就不允许进入CPU执行。GIL只在CPython解释器中才有，因为CPython调用的是c语言的原生线程，不能直接操作cpu，只能利用GIL保证同一时间只能有一个线程拿到数据。**在PyPy和JPython中没有GIL。**
- Python多线程的工作流程：
    - a.拿到公共数据
    - b.申请GIL
    - c.Python解释器调用操作系统原生线程
    - d.CPU执行运算
    - e.当该线程执行一段时间消耗完，无论任务是否已经执行完毕，都会释放GIL
    - f.下一个被CPU调度的线程重复上面的过程
- Python针对不同类型的任务，多线程执行效率是不同的：
    - **对于CPU密集型任务(各种循环处理、计算等等)**: 由于计算工作多，ticks计数很快就会达到阈值，然后触发GIL的释放与再竞争（多个线程来回切换是需要消耗资源的），所以Python下的多线程对CPU密集型任务并不友好
    - **IO密集型任务(文件处理、网络通信等涉及数据读写的操作)**：多线程能够有效提升效率(单线程下有IO操作会进行IO等待，造成不必要的时间浪费，而开启多线程能在线程A等待时，自动切换到线程B，可以不浪费CPU的资源，从而能提升程序执行效率)。所以Python的多线程对IO密集型任务比较友好。
    - 实际中使用的建议：**Python中想要充分利用多核CPU，就用多进程。**因为每个进程有各自独立的GIL，互不干扰，这样就可以真正意义上的并行执行。**在Python中，多进程的执行效率优于多线程(仅仅针对多核CPU而言)。同时建议在IO密集型任务中使用多线程，在计算密集型任务中使用多进程。**另外，深入研究Python的协程机制，你会有惊喜的。

多线程

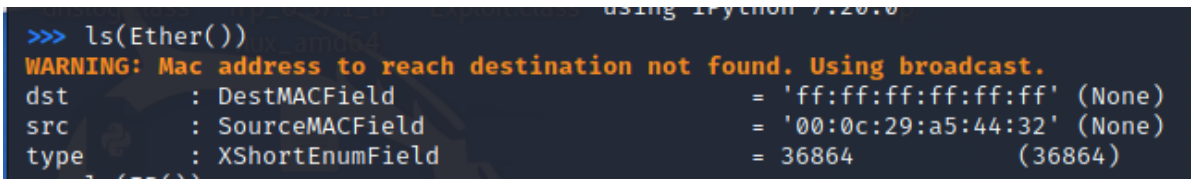
Scapy库

这个库非常强大，可以自定义发包。

在Scapy中每一个协议就是一个类。只需要实例化一个协议类，就可以创建一个该协议的数据包。

Scapy采用分层的形式来构造数据包，通常最下面的一个协议为Ether，然后是IP，在之后是TCP或者UDP，然后是应用层。

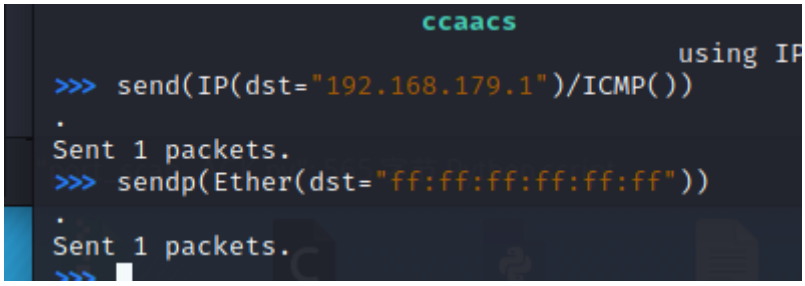
常见协议的属性



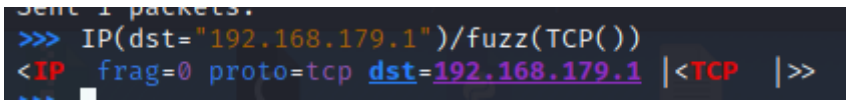
```
type      : XShortEnumField                = 36864          (36864)
>>> ls(IP())
version   : BitField  (4 bits)             = 4              (4)
ihl       : BitField  (4 bits)             = None           (None)
tos       : XByteField                     = 0              (0)
len       : ShortField                     = None           (None)
id        : ShortField                     = 1              (1)
flags     : FlagsField  (3 bits)           = <Flag 0 ()>    (<Flag 0 ()>)
frag      : BitField  (13 bits)            = 0              (0)
ttl       : ByteField                      = 64             (64)
proto     : ByteEnumField                  = 0              (0)
chksum    : XShortField                    = None           (None)
src       : SourceIPField                  = '127.0.0.1'    (None)
dst       : DestIPField                    = '127.0.0.1'    (None)
options   : PacketListField                = []             ([])
>>> ls(TCP())
sport     : ShortEnumField                 = 20             (20)
dport     : ShortEnumField                 = 80             (80)
seq       : IntField                       = 0              (0)
ack       : IntField                       = 0              (0)
dataofs   : BitField  (4 bits)             = None           (None)
reserved  : BitField  (3 bits)             = 0              (0)
flags     : FlagsField  (9 bits)           = <Flag 2 (S)>   (<Flag 2 (S)>)
window    : ShortField                     = 8192           (8192)
chksum    : XShortField                    = None           (None)
urgptr    : ShortField                     = 0              (0)
options   : TCPOptionsField                = []             (b'')
>>> ls(UDP())
sport     : ShortEnumField                 = 53             (53)
dport     : ShortEnumField                 = 53             (53)
len       : ShortField                     = None           (None)
chksum    : XShortField                    = None           (None)
```

常见的函数

Scapy中提供了多个用来完成发送数据包的函数，首先来看一下其中的**send()**和**sendp()**。这两个函数的区别在于send()工作在第三层，而sendp()工作在第二层。简单地说，send()是用来发送IP数据包的，而sendp()是用来发送Ether数据包的。



以上两个函数只发不收，如果希望发送一个内容是随机填充的数据包，而且又要保证数据报的正确性，可以使用**fuzz()**函数。



在scapy中提供了三个用来收发数据包的函数，分别是sr(),sr1()这两个函数工作在第三层，而srp工作在第二层

```
>>> sr1(IP(dst="192.168.179.1")/ICMP())
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0x0 len=28 id=61790 flags= frag=0 ttl=64 proto=icmp chks
um=0xa1a3 src=192.168.179.1 dst=192.168.179.140 |<ICMP  type=echo-reply code=0 ch
ksum=0xffff id=0x0 seq=0x0 |<Padding  load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>>
>>>
```

```
x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>>
>>> sr(IP(dst="192.168.179.1")/ICMP())
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
(<Results: TCP:0 UDP:0 ICMP:1 Other:0>,
 <Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>)
```

sr()函数是scapy的核心，他的返回值是两个列表，第一个列表是收到了应答的包和对用的应答，第二个列表是未收到应答的列表的包

可以使用summary查看包的内容

```
AttributeError: 'list' object has no attribute 'summory'
>>> ans.summary()
IP / ICMP 192.168.179.140 > 192.168.179.1 echo-request 0 ==> IP / ICMP 192.168.17
9.1 > 192.168.179.140 echo-reply 0 / Padding
```

sr1()函数和sr()函数作用基本一样，但是值返回一个应答包。只需要使用一个列表就可以保存这个函数的返回值。

```
>>> p=sr1(IP(dst="192.168.179.1")/ICMP())
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
>>> p
<IP  version=4 ihl=5 tos=0x0 len=28 id=61793 flags= frag=0 ttl=64 proto=icmp chks
um=0xa1a0 src=192.168.179.1 dst=192.168.179.140 |<ICMP  type=echo-reply code=0 ch
ksum=0xffff id=0x0 seq=0x0 |<Padding  load='\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>>
>>>
```

还有一个厉害的函数**sniff**，就是捕获数据包。不仅可以捕获ip地址，协议，还可以用上"and","or"等关系运算符。

```
Scapy_Exception: Failed to compile filter expression 192.168.179.1 (-1)
>>> sniff(filter="host 192.168.179.1")

^C<Sniffed: TCP:97 UDP:36 ICMP:10 Other:80>
>>> sniff(filter="icmp and host 192.168.179.1",count=5,iface="eth0")
^C<Sniffed: TCP:0 UDP:0 ICMP:0 Other:0>
```

另外还有两个挺重要的参数，一是iface，二是count.iface指明要监听的网卡，count知名监听到多少数量就停止。

`_`在scapy中这个符号表示是上一条语句的执行结果

`summary()`函数用来以摘要的形式显示pkt的内容，这个摘要长度为一行

`nsummary()`函数与`summary()`函数作用相同，只是操作对象是单个数据包

[scapy官方手册]

TCP首部格式

源端口（2字节），目的端口（2字节），序号（4字节），确认号（4字节），数据偏移（4位），保留（6位），标志（6位），窗口（2字节），检验和（2字节），紧急指针（2字节），选项

序号：TCP是面向字节流的，在一个TCP链接中传送的字节流的起始序号必须在连接建立时设置。首部中的序号字段值则指的是本报文段所发送的数据的第一个字节的序号。

确认号：期望收到对方下一个报文段的第一个数据字节的序号。

数据偏移：指出该TCP报文段的数据起始处距离TCP报文段的起始处有多远。

保留：以后使用，现在保留是0

标志：

- 紧急URG(URGent):为1时，表明紧急指针有效。他告诉系统此报文又紧急数据，应该立即传送。，而不要按照原来的排队顺序来传送。
- 确认ACK(ACKnowledgment):为1时，确认号字段才有效。
- 推送PSH(PuSH):当两个应用进行交互式通信时，有时在一段的应用进程希望在键入一个命令后立即就能够收到对方的相应。
- 复位RST(ReSeT):为1时，表明TCP连接中出现严重差错，必须释放连接，然后重新建立连接。为1时，用来拒绝一个非法的报文段或者拒绝打开一个连接。
- 同步SYN(SYNchronization):在建立连接是用来同步序号。
- 终止FIN(FINish):用来释放一个连接。

窗口：窗口指的是发送本报文段的一方的接收窗口。

校验和：校验和字段检验的范围包括首部和数据这两部分。

紧急指针：紧急指针在URG=1时才有效。

选项：长度可变

0x02 TCP Connect扫描

与对方主机进行一次完整的三次握手，会在对方主机留下记录，不建议使用。

完整流程是：

你发一个带SYN的包给服务器，如果端口是开的，服务器会接受并且返回一个带有SYN和ACK标识的数据包给客户端，随后客户端回返回带有ACK和RST标识的包，此时服务器与客户端建立了链接。如果端口关闭，就会返回一个RST包。

0x03 TCP SYN扫描

与对方主机仅仅只是进行一次通信，发一个syn过去，只要对面回复了syn+ack就算开放了。我们也不必要回ack了，这样在对方主机就不会留下记录。主要用于躲避防火墙的检测。

完整流程：

客户端发送一个带有SYN表示和端口号的TCP数据包，如果端口开放，服务器会返回一个SYN和ACK的包，但是，这时客户端不会返回RST+ACK，而是返回一个只有RST标识的数据包。如果端口关闭，服务器会返回一个RST数据包。

0x04 TCP FIN扫描

客户端会向服务器发送带有FIN标识的端口号的数据包。如果没有返回则说明端口开放，如果返回一个RST，则说明端口关闭。如果返回一个ICMP数据包，类型为目标不可达（类型为3），以及ICMP代码为1，2，3，9，10，13，则说明端口被过滤了无法确定了端口状态。

0x05 ACK扫描

ACK扫描不是用于发现端口开启或者关闭状态，而是用于发现服务器上是否存在有防火墙，**他的结果只能说明端口是否被过滤。**

客户端会发送一个带有ACK标识的端口号给服务器，如果服务器返回一个带有RST标识的TCP数据包，则说明端口没有被过滤。

如果目标服务器没有任何回应或者ICMP错误类型为3，且代码为1，2，3，9，10，13的数据包，则说明端口有防火墙。

0x06 TCP Xmas 扫描

Xmas扫描又称圣诞树扫描，在这个扫描中，客户端会向服务器带有PSH，FIN，URG标识的端口号的数据包给服务器。

如果端口是开放的，那么不会有任何回应。

如果返回了一个带有RST表示的TCP数据包，则说明端口处于关闭状态。

如果目标服务器没有任何回应或者ICMP错误类型为3，且代码为1，2，3，9，10，13的数据包，则说明端口被过滤了无法确定是否属于开放状态。

0x07 TCP Null扫描

TCP Null扫描又称空扫描，在此次扫描中，客户端发出的TCP数据包仅仅只会包含端口号而不会有任何其他的信息。

如果端口是开放的，那么不会有任何回应。

如果返回了一个带有RST表示的TCP数据包，则说明端口处于关闭状态。

如果目标服务器没有任何回应或者ICMP错误类型为3，且代码为1，2，3，9，10，13的数据包，则说明端口被过滤了无法确定是否属于开放状态。

0x08 TCP Window扫描

客户端向服务器发送一个带有ACK标识的端口号的TCP数据包，在窗口扫描中，收到了TSR数据包后，会检查窗口大小的值。如果非0，表示开放。如果是0，表示不开放。

0x09 UDP扫描

UDP无连接的，无连接的协议不会事先简历客户端和服务器之间的通信信道，只要客户端到服务器之间存在可信信道，就会向对方发送数据。

当客户端发送了一个带有端口号的UDP数据包，如果服务器返回了UDP数据包，则都那口开放。

Tips：

在sr和sr1里面写入verbose=False，可以不显示发包的样子。

设置verbose=true(默认)

```
def fin(ip,port):
    ans=sr1(IP(dst=ip)/TCP(dport=int(port),flags="F"),timeout=10)
```

```
└─# python3 test.py -t 192.168.179.1 -p 20,22,23,80,443,
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
CLosed:20
Begin emission:
Finished sending 1 packets.
*
```

设置为False

```
def three(ip,port):
    ans=sr1(IP(dst=ip)/TCP(dport=int(port),flags="S"),timeout=10,verbose=False)
    if(ans.getlayer(TCP).flags == 0x12):
```

```
┌──(root💀kali)-[/home/kali/Desktop/py3/test]
└─# python3 test.py -t 192.168.179.1 -p 20,22,23,80,443,45,66,88 -m connect_sya
Closed:20
Open:22
Closed:23
Open:80
```

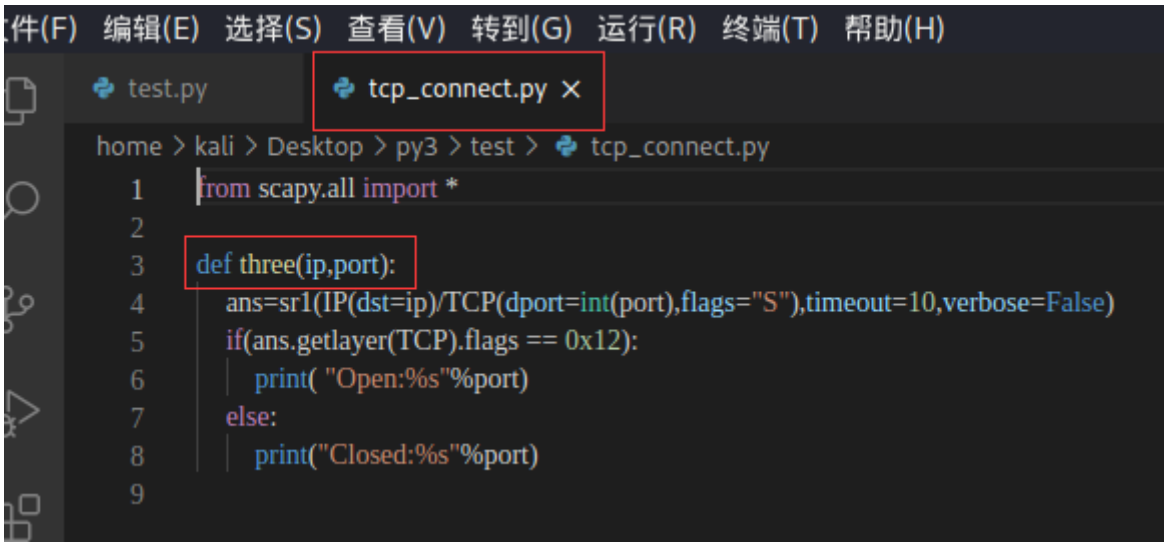
还要注意一点，dport必须是int类型的。

```
def three(ip,port):
    ans=sr1(IP(dst=ip)/TCP(dport=int(port),flags="S"),timeout=10,verbose=False)
```

sr，sr1这些发包的函数都是scapy.sendrecv里的。

IP，TCP这些构造数据包的函数都是scapy.layers.inet里的。

多文件引入，介绍一个简单的例子




```
from tcp_connect import three
from tcp_connect_syn import syn
from tcp_connect_ack import ack
from tcp_connect_FIN import fin
from tcp_connect_null import null
from tcp_connect_Xmas import xmas
import optparse
```

首先在文件中写一个函数，然后把文件和引入他的文件放在同一个文件夹下，接着引入。首先是包，再然后是函数。

还有一个全局变量的声明在函数里需要使用global

python当中没有switch这种结构

[原理实现]

[官方中文文档]