Implication Of SSH Server And Client

SSH 服务器与客户端的设计与实现（LaTeX）

申恒恒
网络工程 13-1
13031110141
heng960509@gmail.com

张波
网络工程 13-1
13031110165

摘要

传统的网络服务程序, 如 rsh,FTP,POP 和 Telnet 其本质上都是不安全的; 因为它们在网络上用明文传送数据, 用户帐号和用户口令, 很容易受到中间人 MITM（man-in-the-middle）攻击方式的攻击. 就是存在另一个人或者一台机器冒充真正的服务器接收用户传给服务器的数据, 然后再冒充用户把数据传给真正的服务器. 而 SSH 可以对所有传输的数据进行加密, 也能够防止 DNS 欺骗和 IP 欺骗. 所以最常用的方式就是利用 SSH（安全外壳）来发送流量, 但是对于大部分（约 99.81943%）的 Windows 主机来说, 不存在 SSH 客户端. 在这里会利用网络编程中的套接字编程结合 Paramiko[1]库开发 SSH 应用程序.

Paramiko 是基于 Python 实现的 SSH2 远程安全连接, 支持认证及密钥方式. 可以实现远程命令执行, 文件传输, 中间 SSH 代理等功能, 相对于 Pexpect[2], 封装的层次更高, 该类封装了传输, 通道等等方法, 使其更贴近 SSH 协议的功能. 因此由于封装了 SSH2 协议更高层次的功能, 本篇论文会将阐述基于 Paramiko 的 SSH 客户端和服务器设计和具体实现, 目的是通过抽象的代码封装来认识 SSH 协议通信流程和通信过程中的关键要点.

关键字：SSH; 中间人攻击; python 语言实现

[1]参见 Github 库 paramiko:https://github.com/paramiko/paramiko

[2]Pexpect 是一个用来启动子程序并对其进行自动控制的 Python 模块,Pexpect 可以用来和像 ssh,ftp,passwd,telnet 等命令行程序进行自动交互.

目 录

1 需求分析

由于设计的为基于 SSH 协议的客户端和服务器, 由于 SSH 服务器是远程连接服务器, 所以在这里会主要会描述关于远程连接服务器的种类, 具体功能以及相关的流程.

1.1 远程连接服务器的概念及分类

远程连接服务器是一种通过文字或者图形接口的方式来远程登录系统, 让用户在远程的终端前面登录 Linux 主以取得可操作主机的接口（Shell）, 而登录后的操作感觉上就像坐在系统前面一样.

根据登录的连接界面分类, 主要有图形接口和文字接口两种类型. 而文字接口又可以划分为加密传输和非加密传输两种, 故大致有三类, 具体分类如下：

1. 文字接口明文传输：Telnet,RSH 等为主, 目前非常少用.
2. 文字接口加密：SSH 为主, 已经取代上述的 Telnet,RSH 等明文传输方式
3. 图形接口：XDMCP,VNC,XRDP 等较为常见

1.2 数据的明文传输和密文传输

明文传输是指数据包在网络上传输时, 该数据包的内容为数据的原始格式, 也就是说当使用 Telnet 登录远程主机时, 输入的帐号密码将以明文的歌好似显示在网络上, 如果被网络上那个类似于 tcpdump 的监听工具捕获到, 那么账号和密码将有可能会被窃取 !

由于目前的明文传输的 Telnet,RSH 等连接服务器已经被 SSH 取代, 并且在实际应用上很少能看到 Telnet 和 RSH 了, 基于此, 可以看到目前远程接口连接服务器主要以 SSH 为主, 为此在此会具体介绍有关 SSH 应用, 和阐述 SSH 的安全性.

1.3 SSH 服务器

由于早期的远程接口连接服务器都是以明文传输为主, 而且协议存在很大的安全问题, 所以之后出现了基于 SSH 协议的 SSH 服务器, 对于 SSH 来说, 它可以通过数据包加密技术将等待传输的数据包加密后再传输到网络上, 因此相对明文传输来说,SSH 有很大优势, 因此 SSH 取代了很多以明文传输的应用, 比如 Finger,R Shell 等.

1.4 SSH 协议的加密技术

SSH 的加密技术主要依照非对称加密的技术和思想, 来对要发送到消息进行加密, 非对称加密系统的相关术语描述如下：

1. 公钥：公钥又被称作加密密钥, 提供给远程主机进行加密的行为, 即用公钥来对消息进行加密, 然而这个公钥可以被任何人使用.

2. 私钥：私钥又被称作解密密钥, 本地主机用自己私钥来对远程主机使用自己的公钥加密的数据进行解密, 因为私钥很重, 所以在网络上不能传送, 因此只能保存在本地主机上.

通过上面的非对称加密系统的描述可知, 在真实环境下, 每台主机都应该拥有自己的私钥和公钥, 将公钥发送给于自己进行通信的远程主机, 然后远程主机使用自己的公钥进行加密, 然后将加密后的数据发送给本地主机, 本地主机然后通过自己的私钥对密文进行解密, 注意, 此时的加密密钥和解密密钥是不同的, 具体的公钥和私钥在进行数据传输时的过程图如图 1 所示.

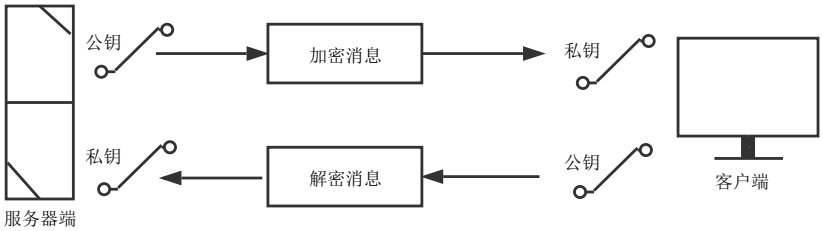

图 1: 公钥和私钥在数据传输时的角色示意图

对于 SSH 服务器和客户端来说, 都基于非对称加密系统, 因此 SSH 服务器和客户端的连接步骤如图 2 所示：

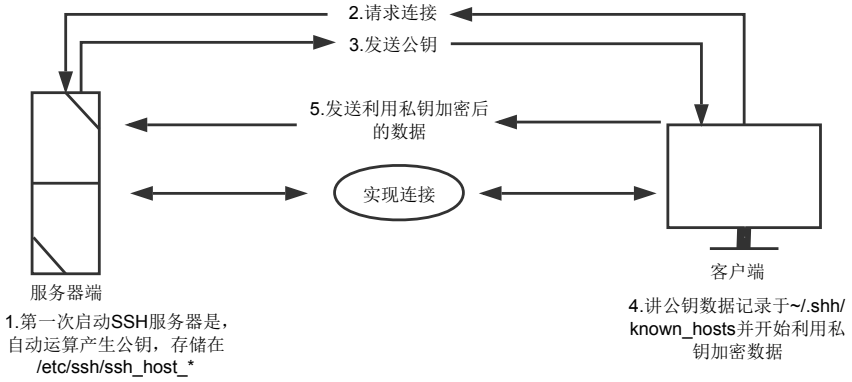

图 2: SSH 服务器与客户端的连接步骤示意图

2 总体概述

采用经典的 C/S 架构, 将服务器端和客户端分而治之, 利用 Paramiko 函数库快速的实现.

2.1 安全策略

2.1.1 客户端安全验证

1.（基于密码的安全验证）知道帐号和密码, 就可以登录到远程主机, 并且所有传输的数据都会被加密. 但是, 可能会有别的服务器在冒充真正的服务器, 无法避免被“中间人”攻击.

2.（基于密钥的安全验证）需要依靠密钥, 也就是你必须为自己创建一对密钥, 并把公有密钥放在需要访问的服务器上. 客户端软件会向服务器发出请求, 请求用你的密钥进行安全验证. 服务器收到请求之后, 先在你在该服务器的用户根目录下寻找你的公有密钥, 然后把它和你发送过来的公有密钥进行比较. 如果两个密钥一致, 服务器就用公有密钥加密“质询”（challenge）并把它发送给客户端软件. 从而避免被“中间人”攻击.

2.1.2 服务器安全验证

1. 在第一种方案中, 主机将自己的公用密钥分发给相关的客户端, 客户端在访问主机时则使用该主机的公开密钥来加密数据, 主机则使用自己的私有密钥来解密数据, 从而实现主机密钥认证, 确保数据的保密性.

2. 在第二种方案中, 存在一个密钥认证中心, 所有提供服务的主机都将自己的公开密钥提交给认证中心, 而任何作为客户端的主机则只要保存一份认证中心的公开密钥就可以了. 在这种模式下, 客户端必须访问认证中心然后才能访问服务器主机.

2.2 程序流程

对于传统的 SSH 服务器和客户端设计, 首先根据 SSH 服务器端和客户端的连接步骤设计了如下的类流程结构图, 如图 3 所示.

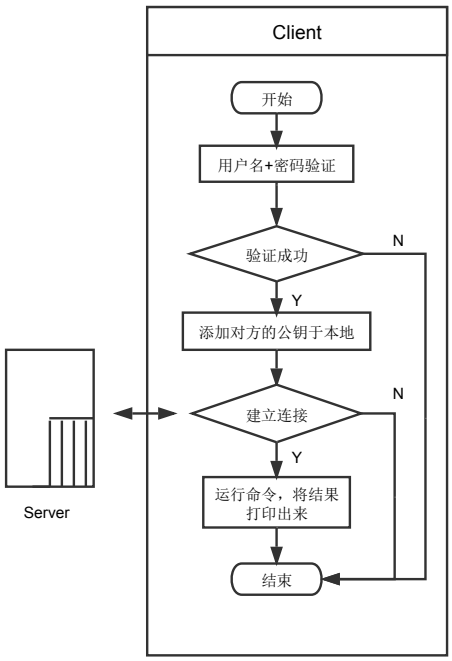

图 3: 传统的 SSH C/S 模型结构图

但是为了更好的设计跨平台的一个 SSH 服务器和客户端工具,Windows 本身并不存在 SSH 服务器, 所以经过修改传统[3]的 SSH 服务器和客户端, 使得在使用 SSH 的时候, 可以反向将命令从 SSH 服务器发送到客户端.

[3]这里的传统指的是在 UNIX 或 Linux 平台运行的程序流程

3 详细设计

3.1 客户端设计

在客户端实现方面，由于客户端在这里只是实现了执行对方发送过来的命令，并且将其执行的结果反向发送给对方，并在对方的显示器上反向输出出来. 下面为具体实现代码.

```
def ssh_command(ip, user, passwd, command):
  client = paramiko.SSHClient()
  # client.load_host_keys('./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)
    while 1:
      command = ssh_session.recv(1024)
      try:
        cmd_output = subprocess.check_output(command, shell=True)
        ssh_session.send(cmd_output)
      except Exception, e:
        ssh_session.send(str(e))
    client.close()
  return
```

在这里, 创建了名为 *ssh_command* 的函数, 该函数的功能为连接到 SSH 服务器并发送给服务器端一条消息，然后将服务端发送过来的消息打印出来，接着进入无限循环，在循环体内，接受服务器端发送过来的命令，然后利用 *subprocess.check_output()* 生成一个子进程，并且传递两个参数，在这里使用了 Python 的函数库:subprocess 库.subprocess 提供了强大的进程创建接口，在本程序中首先 *shell=True* 表示 Python 将先运行一个 shell, 利用这个 shell 来解释 *command* 命令，然后这个方法返回子进程向标准输出的输出结果，也就是说将在本机执行 *command* 命令，并且将执行结果保存在 *cmd_output* 中，然后将返回结果发送给服务器端. 以上为客户端执行过程.

另外 Paramiko 支持密钥验证来代替密码验证, 在现实情况下基本都为密钥认证, 但是为了方便起见, 在这里利用了传统的用户名和密码的方式进行验证. 因为连接两端的主机（server 端和 client 端）都在控制之下, 所以通过设置策略 *paramiko.AutoAddPolicy()* 自动添加和保存目标 SSH 服务器的 SSH 密钥.

```
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
```

set_missing_host_keys 方法表示为设置连接的远程主机没有本地主机密钥或这 HostKeys 对象的策略, 目前支持三种策略，分别为 AutoAddPolicy，RejectPolicy（默认）,WarningPolicy，仅限于使用于 SSHClient 类，其中 AutoPolicy 代表自动添加主机名和主机密钥到本地的 HostKeys 对象，并将其保存，不依赖于 *load_system_host_keys()* 的配置，即使～*/.ssh/known_hosts* 不存在也不产生影响. 而 RejectPolicy 表示自动拒绝未知未知主机名和密钥，依赖于 *load_system_host_keys()* 的配置.WarningPolicy 表示用于记录一个未知的主机密钥的 Python 警告，并接受它，功能上与 AutoAddPolicy 相似，但是未知主机会有警告，但出于本项目的要求，在这里采用了 AutoAddPolicy 策略.

然后开始进行连接, 最后假设连接成功, 通过调用 *ssh_command* 函数, 发出第一条命令 clientConnected

```
ssh_command('192.168.1.3', 'administrator', 'heng130509', 'ClinetConnected')
```

3.2 服务器端设计

由于客户端程序较为简单,那么具体的复杂操作均在服务器端完成,首先在这里使用了 Paramiko 中的实例文件中的密钥，在这里主要使用了 RSAKey 类，接受参数，参数为密钥文件[4]，这里主要的功能就是一个 RSAKey 可以被用于签名和验证 SSH2 数据.

```
host_key = paramiko.RSAKey(filename = 'test_rsa.key')
```

开启了一个套接字监听，下面为具体实现程序:

```
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, e:
  print '[-] listen failed :' +str(e)
  sys.exit(1)
```

首先利用 *socket.socket()* 函数创建一个套接字对象，套接字构造方法的第一个参数为地址组，在这里使用了 socket.AF_INET 表示为 IPv4 地址族，第二个参数表示为套接字类型，在这里为 TCP 类型的套接字，所以在这里创建了一个 TCP 套接字，如果按照传统的方式做，创建完套接字对象后，然后绑定地址和端口，然后监听连接的客户端, 但是在实际情况中，会出现一种情况，无论有意或者无意关闭套接字，但是还想继续始终在同一个端口上运行套接字服务器，这种情况被称作“套用套接字地址”，对于 SSH 服务器客户端连接来说这是非常必要的，在每次断开连接后，无需更换端口，继续保持连接，所以在这里调用了 setsockopt() 方法，修改地址重用状态的值，再按照常规的步骤，把套接字绑定到一个指定的地址和端口上然后启动监听，这里的最大连接数为 10，然后等待连接，将一个客户端成功建立连接的时候，将接受到的客户端的套接字对象保存到 client 变量中，将远程建立的细节保存到 addr 变量中.

接下来会用到 SSH 管道，SSH 管道是两台机器间的安全连接，经常被称为“SSH 隧道”，或者“端口转发”。具体的实现如下:

```
class Server(paramiko.ServerInterface):
  """创建SSH管道"""
  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_PROHIBITED
  def check_auth_password(self, username, password):
    if (username == 'administrator') and(password == 'hadoop'):
      return paramiko.AUTH_SUCCESSFUL
    return paramiko.AUTH_FAILED
```

创建一个 Server 类，在这里继承了 paramiko.ServerInterface 父类的所有属性和方法，由于 paramiko.ServerInterface 类调用了 Paramiko 主进程，所以已经基本封装了所有有关 SSH 服务器所有功能，在这里将不会做太多，这里创建的 Server 类只是在继承父类的基础上，添加一些基本的验证。首先 Server 类里面的 *___init___* 方法时 Python 类的构造函数，并在类里面初始化一些

[4]密钥文件可以在 https://raw.githubusercontent.com/paramiko/paramiko/master/tests/test_rsa.key 下载到

变量，在这里对每个实例化类对象，都创建一个进程，*check_channel_request* 函数时继承父类的方法，然后在继承过来的方法的基础上添加了验证，*check_channel_request* 函数接受两个参数 kind 和 chanid, 功能为确定一个给定类型的信道请求将被授予，并返回 OPEN_SUCCEEDED 或者错误代码。当客户端请求一个通道，验证完成后，此方法被称为服务器模式.*check_auth_password* 方法同样也是父类的方法，用来确定是否由客户提供特定的用户名和密码是用于验证身份　接受。如果密码不被接受返回 AUTH_FAILED ，如果密码被接受则返回 AUTH_SUCCESSFUL.

之后配置认证模式，代码如下：

```
try:
  shSession = paramiko.Transport(client)
  shSession.add_server_key(host_key)
  server = Server()
  try:
      shSession.start_server(server=server)

  except paramiko.SSHEXceptionn, x:
    print '[-] SSH negotition failed'
```

一个 SSH 传输通常连接到一个网络流(通常是一个套接字),比如在这里实现的是 paramiko.Transport(client)，就是连接了上面的接受连接的客户端的套接字对象 client，然后协商加密的会话，进行身份验证，然后在整个会话中创建流隧道，称为通道. shSession.add _server_key(host_key) 表示在服务器模式下添加一个主机密钥。当作为一个服务器，在 SSH2 协商中主机密钥用于签署某些数据包，从而使客户可以信任我们是谁。因为这是用于签名，密钥必须包含私钥信息，而不只是只有公钥。在这里传递了一个 RSAKey 对象的文件，然后创建 Server 对象，然后启动该服务。后面 try...except... 块为异常处理。

一旦服务开启之后，等待连接，当一个客户端认证成功[5] 并返回 ClientConnected 消息, 然后输入到 *sh_sshserver* 的任何命令将发送给 *sh_sshclient* 并在 *sh_sshclient* 上执行, 输出结果将返回给 *sh_sshserver*. 具体实现如下

```
while 1:
  try:
    command = raw_input("Enter command:").strip('\n')
    if command != 'exit':
      chan.send(command)
      print chan.recv(1024) + '\n'
    else:
      chan.send('exit')
      print 'exiting'
      shSession.close()
      raise Exception("Exit")
    except KeyboardInterrupt:
      shSession.close()
  except Exception, e:
    print '[-] Caught exception: ' + str(e)
    try:
      shSession.close()
    except:
      pass
    sys.exit(1)
```

[5]若连接成功, 则会打印 [+] Authenticaed!

4 测试环节

作为示例, 将在 windows 主机上同时运行客户端和服务器端代码步骤如下:

1. 运行服务器端

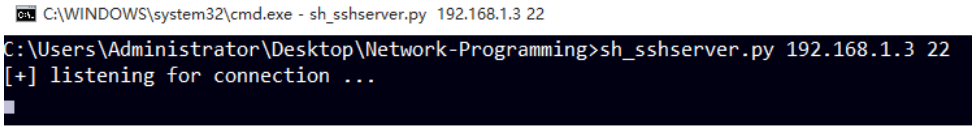

图 4: 运行服务器端

2. 使用客户端连接, 客户端连接成功, 客户端返回给服务器端 ClientConnected 消息

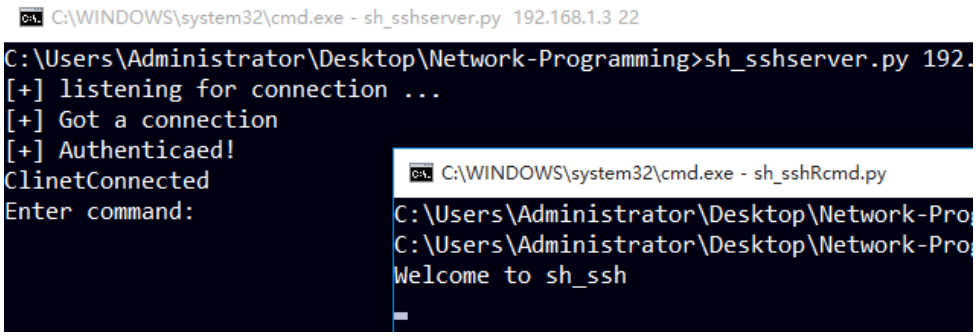

图 5: 客户端连接成功

3. 在服务器端执行一条命令, 在客户端看不到任何情况, 但是命令在客户端已经执行了, 并且客户端将执行的结果返回给 SSH 服务端.

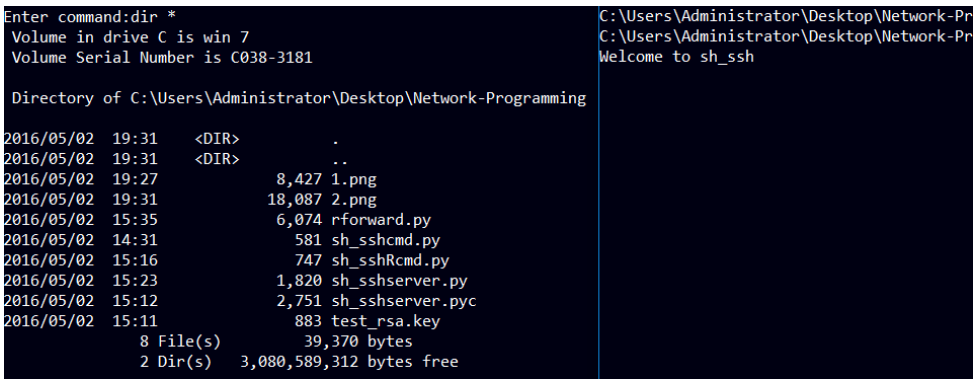

图 6: 连接成功后, 在服务端执行命令, 且回显打印在屏幕上

参考文献

[1] 鸟哥, 鸟哥的 LINUX 私房菜-服务器架设篇（第三版）, 北京, 机械工业出版社,2015,311 ～ 322

[2] Justin Seitz,Python 黑帽子-黑客与渗透测试编程之道, 北京, 电子工业出版社,2015,29 ～ 35

[3] 刘天斯,Python 自动化运维-技术与最佳实践, 北京, 机械工业出版社,2015,79 ～ 103

A 源码

server.py

```
import paramiko
import threading
import subprocess
import sys
import socket


host_key = paramiko.RSAKey(filename = 'test_rsa.key')

class Server(paramiko.ServerInterface):
  """docstring for Server"""
  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_PROHIBITED
  def check_auth_password(self, username, password):
    if (username == 'administrator') and(password == 'hadoop'):
      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, e:
  print '[-] listen failed :' +str(e)
  sys.exit(1)

print '[+] Got a connection'

try:
  shSession = paramiko.Transport(client)
  shSession.add_server_key(host_key)
  server = Server()
  try:
    shSession.start_server(server=server)

  except paramiko.SSHExceptionn, x:
    print '[-] SSH negotition failed'
  chan = shSession.accept(20)
  print '[+] Authenticaed!'
  print chan.recv(1024)
  chan.send('Welcome to sh_ssh')
  while 1:
    try:
      command = raw_input("Enter command:").strip('\n')
```

```
      if command != 'exit':
        chan.send(command)
        print chan.recv(1024) + '\n'
      else:
        chan.send('exit')
        print 'exiting'
        shSession.close()
        raise Exception("Exit")
    except KeyboardInterrupt:
      shSession.close()
except Exception, e:
  print '[-] Caught exception: ' +  str(e)
  try:
    shSession.close()
  except:
    pass
  sys.exit(1)
```

client.py

```
import paramiko
import threading
import subprocess

def ssh_command(ip, user, passwd, command):
  client = paramiko.SSHClient()
  # client.load_host_keys('./known_hosts')
  client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
  client.connect(ip, username=user,port=8000, password=passwd)
  ssh_session = client.get_transport().open_session()

  if ssh_session.active:
    ssh_session.send(command)
    print ssh_session.recv(1024)
  while 1:
    command = ssh_session.recv(1024)
    try:
      cmd_output = subprocess.check_output(command, shell=True)
      ssh_session.send(cmd_output)
    except Exception, e:
      ssh_session.send(str(e))
  client.close()
  return
ssh_command('127.0.0.1', 'administrator', 'heng130509', 'ClinetConnected')
```