

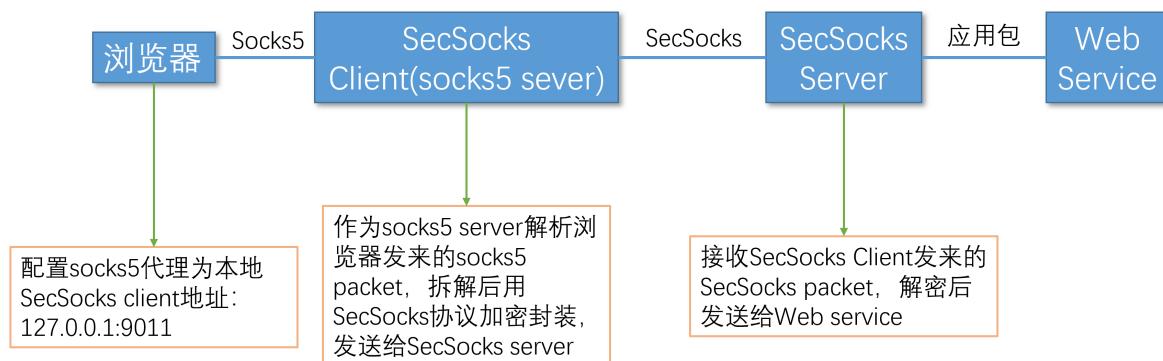
# Lab5 Report

- 计72 周炫柏 2017011460

## 1 代理协议设计思路

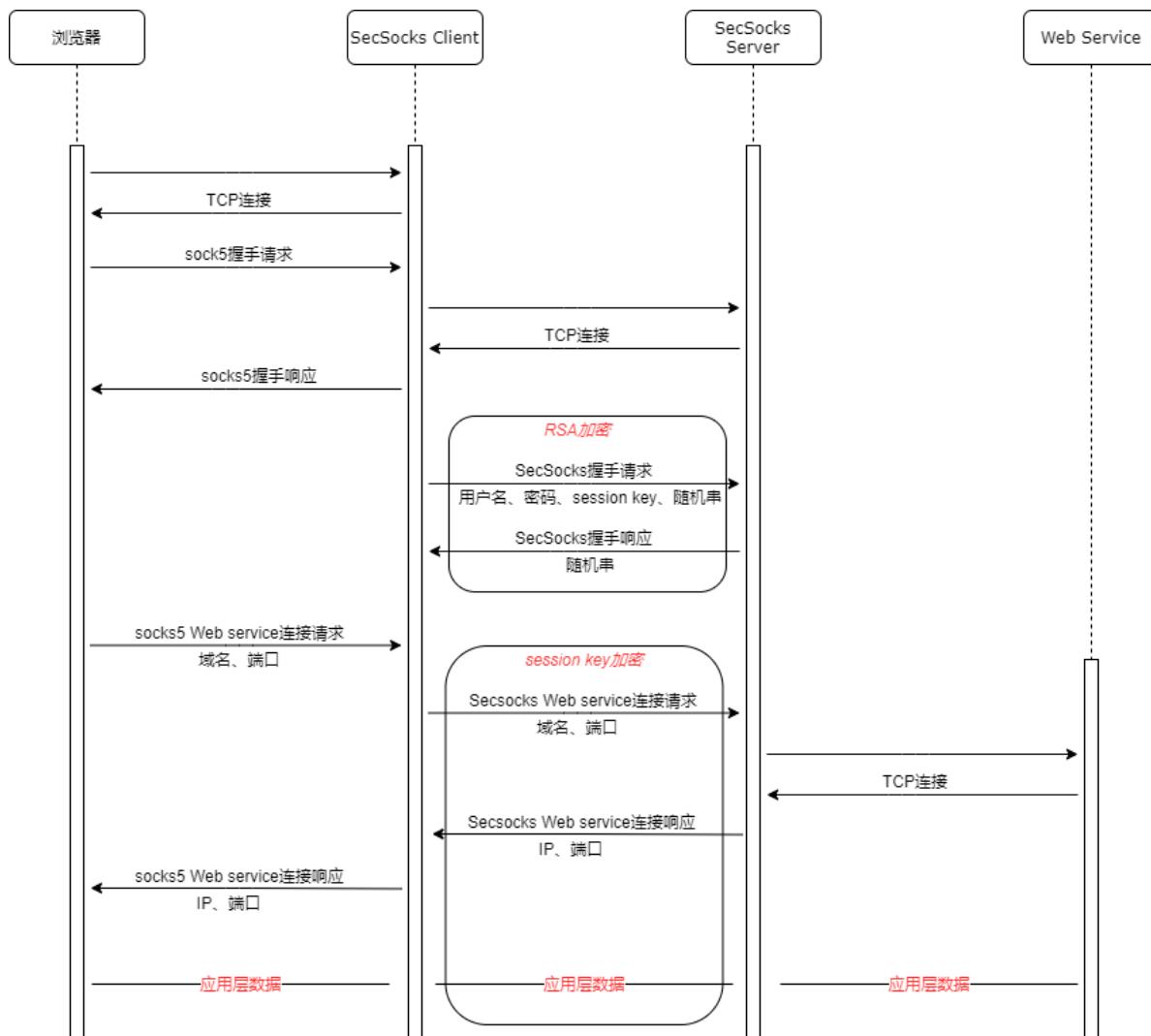
我们对这个代理的功能预期是它工作于socks层，和socks5类似，也就是说在TCP层之上，在TLS和应用层之下，这样TCP连接我们可以用socket管理。比较麻烦的事情是socksify的过程，也就是把应用层的packet中请求的域名和端口解析出来，然后方便proxy server和用户想要访问的web service之间建立TCP连接。这个过程是依赖于应用层协议的，我们认为这部分不是我们代理协议的重点（同时也为了保证我们代理协议的通用性）。因此我们考虑的思路是，在浏览器和本地proxy client之间使用socks5协议进行通信，也就是说我们本地的proxy client同时也是一个socks5 server，我们在proxy client对浏览器发来的socks5协议包进行拆包，提取出我们需要的web service域名和端口等信息，加密后传送给proxy server，proxy server和web service之间建立TCP连接，之后我们的代理协议就可以正常加密和转发TLS和应用层的数据。

我们把我们设计的代理协议命名为**SecSocks**，我们的代理工作示意图如下：



## 2 SecSocks协议工作原理

### 协议泳道图



## 协议消息格式

- client发起连接，身份验证
  - key由client随机生成
  - 该消息用secsocks server的公钥加密

CMD(1 byte)	ulen(1 byte)	plen(1 byte)	klen(1 byte)	username(1 - 255byte)	password(1 - 255byte)	key(1 - 255byte)	padding zero(0 - 1 byte)	random str(10 byte)
0	用户名长度	密码长度	session key长度	用户名	密码	session key长度		随机串

- server验证身份，返回结果
  - 随机串是client之前发送的随机串
  - 该消息用secsocks server的私钥加密

CMD(1 byte)	RESULT(1 byte)	random str(10 byte)
1	0失败、1成功	随机串

- client请求连接Web服务
  - 使用session key加密

CMD(1 byte)	atype(1 byte)	addr(4 byte)	port(2 byte)
2	0(IPv4地址)	IPv4地址	端口

CMD(1 byte)	atype(1 byte)	alen(1 byte)	padding zero(1 byte)	addr(1 - 255 byte)	padding zero(0 - 1 byte)	port(2 byte)
2	1(DOMAINNAME)	域名长度		域名		端口

- server返回连接成功或失败
  - 使用session key加密

CMD(1 byte)	RESULT(1 byte)
3	0(失败)

CMD(1 byte)	RESULT(byte)	addr(4 byte)	port(2 byte)
3	1(成功)	域名	端口

- 开始应用层协议数据传输
  - 使用session key加密

## 3 登录认证功能

我们实验使用的浏览器是Chrome，我们本来打算是使用Chrome的插件进行socks5的用户名和密码设置，然后在SecSocks Client拆解socks5协议包的时候直接提取出其中的用户名和密码，作为我们的SecSocks协议的登录用户名和密码。但是当我们魔改了一个Chrome的代理插件（proxy helper）以后却发现，Chrome内部的socks5 client实现根本就不支持authentication（那这个socks5形同虚设），所以我们使用了比较简陋的方式，在运行socks5 client的时候指定用户名和密码，就像下面这样

```
> python .\secsocks_client.py username password
```

## 4 协议加解密

### 握手过程加密

SecSocks Client中配置有SecSocks Server的公钥，SecSocks握手请求使用公钥加密，SecSocks Server使用私钥解密；握手请求的响应使用私钥加密，使用公钥解密。

- 考虑到我们写的RSA算法比较简单，性能有限，所以我们仅每两个bytes进行加解密，所以映射空间仅仅是 $65536 \times 65536$ ，其实比较容易破解（算法本身也只是一个雏形，这里给破解同学提供了一点思路）
- 握手过程中还传送了一个随机串，握手响应中必须携带这个随机串，以确保SecSocks Server收到了这个握手请求

- 握手请求中携带了SecSocks Client随机出来的session key，因此一步握手里同时也完成了密钥协商的工作

## 后续加密

握手过程之后的流量加密使用的都是session key，所用的算法是playfair

- 每一个TCP连接使用的都是一个单独的session key

## 5 加解密算法设计与实现

非对称加密部分使用的是RSA算法，实现逻辑主要参考了附录中链接的内容。

- RSA算法要求找到两个不同的大整数P、Q，令 $N=P*Q$ ，再从0开始找到第一个与N互质的数e，则得到公钥  $(N,e)$ 。然后通过欧拉公式

$$\Phi(N) = \Phi(P) * \Phi(Q) = (P - 1)(Q - 1)$$

得到 $\Phi(N)$ ，找到满足

$$ed \equiv 1 \pmod{\varphi(N)}$$

的整数d，d称为e关于 $\Phi(N)$ 的模反元素，得到密钥  $(N,d)$ 。由上述de关系可知存在

$$\forall m, m^{ed} \equiv m \pmod{N}$$

具体数学证明省略。因此对于明文转换成的任意整数，通过公钥加密

$$m^e \equiv c \pmod{N}$$

之后传到server端，server端再通过私钥解密

$$c^d \equiv m \pmod{N}$$

这样就能得到原文对应的整数m了，再经过相应处理即可得到原来的明文。

- 具体代码如下：

```
from Crypto.Util.number import *

# 辗转相除法得到最大公约数
def gcd(a,b):
    if a%b == 0:
        return b
    else :
        return gcd(b,a%b)

# 通过最大公约数判断两者是否互质
def isPrime(a,b):
    if gcd(a,b) == 1:
        return True
    else :
        return False

# 求私钥
def rsa_get_key(e, euler):
    k = 1
    while True:
```

```

        if (((euler * k) + 1) % e) == 0:
            return (euler * k + 1) // e
        k += 1

# 根据n,e计算d
def getd(e):
    euler = (p-1)*(q-1)
    k = 1
    while True:
        if (((euler * k) + 1) % e) == 0:
            return (euler * k + 1) // e
        k += 1

# 得到两大素数P、Q以及对应的N
p = getPrime(32)
q = getPrime(32)
n = p*q

# 遍历找到第一个与N互质的整数e
e = 0
while True:
    e = getPrime(16)
    if isPrime(n,e) == True :
        break

# 通过N, e求出d
d = getd(e)

# 将计算得到的N、e、d写入rsa存储文件
filename = 'rsa_ned.txt'
with open(filename, 'a') as file_object:
    file_object.seek(0)
    file_object.truncate()
    file_object.write(str(n))
    file_object.write('\n')
    file_object.write(str(e))
    file_object.write('\n')
    file_object.write(str(d))

```

- server端的代码如下：

```

filename = 'rsa_ned.txt'
# 从rsa文件中读取n、e、d的值
with open(filename, 'r') as f:
    n = int(f.readline()[:-1])
    e = int(f.readline()[:-1])
    d = int(f.readline())

...
print(n)
print(e)
print(d)
"""

# 用公钥加密
def encrypt(a):
    b = []

```

```

for i in range(len(a)):
    b.append(pow(a[i], e, n))
# print(b)
return b

# 用私钥解密
def decrypt(a):
    b = []
    for i in range(len(a)):
        b.append(pow(a[i], d, n))
    # print(b)
    return b

```

对称加密部分使用了playfair算法。

- 我们使用的playfair算法以字符的ASCII码值为加密元，对相邻两个字符进行加密。

考虑到输入字符集为ASCII码0~255的256个字符，正好可以平铺 $16 \times 16$ 的矩阵，我们以字符的ASCII码整除16为行值row，模16为列值col，可以得到每个字符对应的唯一坐标 (row, col)。对任意输入根据以下规则进行加密：

1. 一对一对取字母，如果最后只剩下一个字母，则不变换，直接放入加密串中；
2. 如果一对字母中的两个字母相同，则不变换，直接放入加密串中；
3. 如果字母对出现在方阵中的同一行或同一列，则对调这两个字母；
4. 如果在正方形中能够找到以字母对 (a, b) 为顶点的矩形，则变换成矩形的另一对顶点，且与a同行的字母应在前面。

出于对安全性的进一步考虑，在顺序的ASCII码矩阵中设定关键字session key，session key是一个由client和server协商的字符串，这个字符串中的字符将被排在ASCII码矩阵的最前面，这样可以使对应关系变得更为复杂。此外，我们认为可以让session key的长度也作为一个随机数，这样随机性更加高，安全性也随之上升。

- 具体代码如下：

```

from random import *
import struct

#生成一个n位的随机数
def ran_num():
    n = choice([110, 120, 130, 140, 150])
    str = "".join([choice("0123456789ABCDEF") for i in range(n)])
    return str

#用生成的随机数来组成session key -- 同样只能用一次，保证对称密钥不会改变
def skey():
    a = ""
    str = ran_num()
    for i in range(len(str) // 2):
        chr1 = int(str[i], 16)
        chr2 = int(str[i+1], 16)
        i = i + 2
        str += chr(16*chr1+chr2)
    return len(''.join(set(str))), ''.join(set(str))

#构筑新的16*16映射表
def get_s_arr(s_key):

```

```

b = list()
s = set()
for i in range(len(s_key)):
    if not (s_key[i] in s):
        b.append(s_key[i])
        s.add(s_key[i])

for i in range(256):
    if not chr(i) in s:
        b.append(chr(i))
        s.add(chr(i))
return b

#加解密
def pf_crypt(a,s_arr):
    a = "".join([chr(x) for x in a])
    b = []
    i = 0
    while i in range(len(a)):
        #如果最后只剩下一个字母，则不变换，直接放入加密串中
        if i+1 >= len(a):
            b.append(a[i])
            break
        #如果一对字母中的两个字母相同，则不变换，直接放入加密串中
        if a[i] == a[i+1]:
            b.append(a[i])
            b.append(a[i+1])
            i += 2
            continue
        #如果字母对出现在方阵中的同一行或同一列，则只需简单对调这两个字母
        # print("a[i] ", int(a[i]))
        ind1 = s_arr.index(a[i])
        ind2 = s_arr.index(a[i+1])
        if (ind1 // 16 == ind2 // 16) or (ind1 % 16 == ind2 % 16):
            b.append(a[i+1])
            b.append(a[i])
        #如果有以字母对为顶点的矩形，则该矩形的另一对顶点字母中，与a同行的字母应在前面
        else:
            b.append(s_arr[(ind1 // 16)*16 + (ind2 % 16)])
            b.append(s_arr[(ind2 // 16)*16 + (ind1 % 16)])
        i += 2
    b = "".join(b)
    b = struct.pack("!" + "B"*len(b), *[ord(x) for x in b])
    return b

# k_len, s_key = skey()
# s_arr = get_s_arr(s_key)
# # import struct
# # print([x.encode() for x in s_arr])
# # print(a)
# # a = b'\xfa\x1e'
# a = b'\x02\x01'
# a = b'\x02\x01\x14\x00beacons.gcp.gvt2.com\x01\xbb'
# # # print(s_arr)
# b = pf_crypt(a,s_arr)
# c = pf_crypt(b,s_arr)
# # print("".join(c).encode())
# print(b)

```

```
# print(c)
```

至此完成了对称加密部分的功能。

- 具体通信过程为：client首先使用server提供的公钥将重要信息进行加密（此时传递的信息为用户名、密码、对称加密使用的密钥），server收到后用自己的私钥解密，进行身份的验证，并确定使用对称加密的密钥，回复一个用该密钥加密的连接成功的回复，然后就可以进行正常的数据传输了。

## 6 本地功能测试截图

- Chrome浏览器插件配置SecSocks代理为本地代理地址（我们的代码里附赠了一个我们稍微修改过的插件proxy helper，当然也可以其他任何可以设置sock5代理地址的Chrome插件）



- 运行SecSocks Server代码，监听本地9022端口

```
PS D:\projects\NSE\SecSocks> python .\secsocks_server.py
```

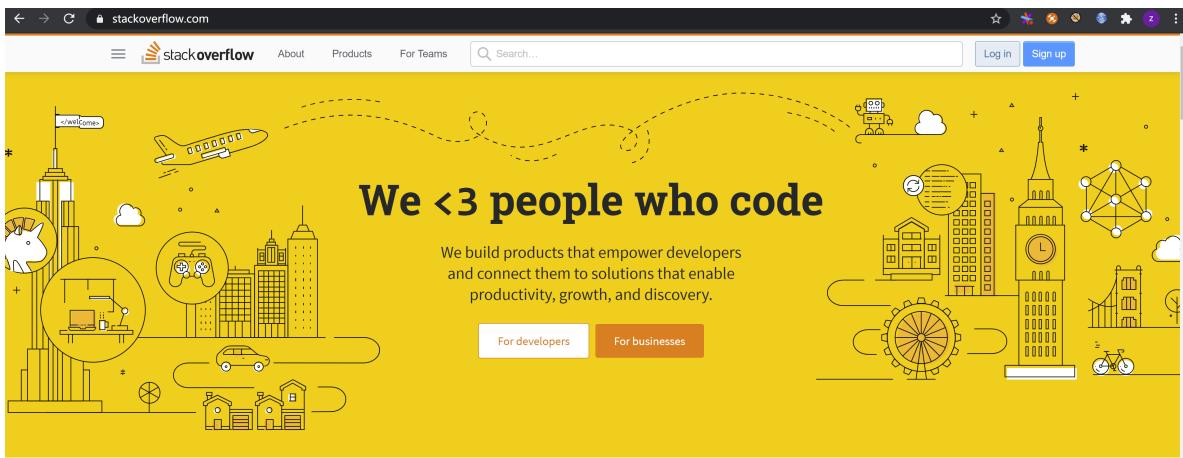
```
17     sec_server_address = '0.0.0.0'  
18     sec_server_port = 9022  
19
```

- 运行SecSocks Client，指定用户名和密码，指定socks5监听本地9011端口，SecSocks试图连接9022端口

```
PS D:\projects\NSE\SecSocks> python .\secsocks_client.py username password
```

```
22     sec_server_address = '127.0.0.1'  
23     sec_server_port = 9022  
24  
25     socks5_server_address = '127.0.0.1'  
26     socks5_server_port = 9011  
27
```

- 使用代理访问stack overflow



## For developers, by developers

Stack Overflow is an [open community](#) for anyone that codes. We help you

- client和server端可以查看到的代理log (左边是server, 右边是client)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
INFO:root:Connected to 203.208.41.70 443
connect success!
连接成功
login success!
INFO:root:Accepting connection from 127.0.0.1:57902
addr resolving
login success!
addr resolving
INFO:root:Connected to 184.25.246.121 443
connect success!
INFO:root:Connected to 103.229.10.173 443
连接成功
connect success!
连接成功
-----
Exception occurred during processing of request from ('127.0.0.1', 57857)
Traceback (most recent call last):
  File "C:\Users\BackyxM\AppData\Local\Programs\Python\Python39\lib\socketserver.py", line 650, in process_request_thread
    self.finish_request(request, client_address)
  File "C:\Users\BackyxM\AppData\Local\Programs\Python\Python39\lib\socketserver.py", line 360, in finish_request
    self.RequestHandlerClass(request, client_address, self)
  File "C:\Users\BackyxM\AppData\Local\Programs\Python\Python39\lib\socketserver.py", line 720, in __init__
    self.handle()
  File "D:\projects\NSE\SecSocks\secsocks_server.py", line 92, in handle
    self.exchange_loop(self.connection, remote)
  File "D:\projects\NSE\SecSocks\secsocks_server.py", line 158, in exchange_loop
    data = remote.recv(4096)
ConnectionResetError: [WinError 10054] 远程主机强迫关闭了一个现有的连接。
-----
INFO:root:Accepting connection from 127.0.0.1:57906
login success!
addr resolving
INFO:root:Connected to 140.82.113.26 443
connect success!
连接成功

```

```

1: python python
ConnectionAbortedError: [WinError 10053] 你的主机中的软件中止了一个已建立的连接。
  File "D:\projects\NSE\SecSocks\secsocks_client.py", line 197, in exchange_loop
    data = client.recv(4096)
-----
ConnectionAbortedError: [WinError 10053] 你的主机中的软件中止了一个已建立的连接。
ConnectionAbortedError: [WinError 10053] 你的主机中的软件中止了一个已建立的连接。
-----
INFO:root:Accepting connection from 127.0.0.1:57893
listening port: 57894
Login success!
INFO:root:Accepting connection from 127.0.0.1:57896
listening port: 57897
Login success!
INFO:root:Accepting connection from 127.0.0.1:57899
listening port: 57900
INFO:root:Accepting connection from 127.0.0.1:57901
Login success!
listening port: 57902
Login success!
INFO:root:Accepting connection from 127.0.0.1:57905
listening port: 57906
Login success!

```

## 一点未解决的问题

在舍友谢兴宇的帮助下，我把这个SecSocks Server部署到了他的服务器上，但是代理效果并不理想，传输过程中很容易出现丢包的情况，导致最终SecSocks Server传送给Web Service的数据不完整，回传的数据同样不完整，TCP连接被Web Service给reset了。目前猜想的原因可能是exchange loop中socket send和recv设置的size太大了，超出了传输路径中的闲置（把4096改更64后情况改善了很多）

```

def exchange_loop(self, client, remote):

    while True:

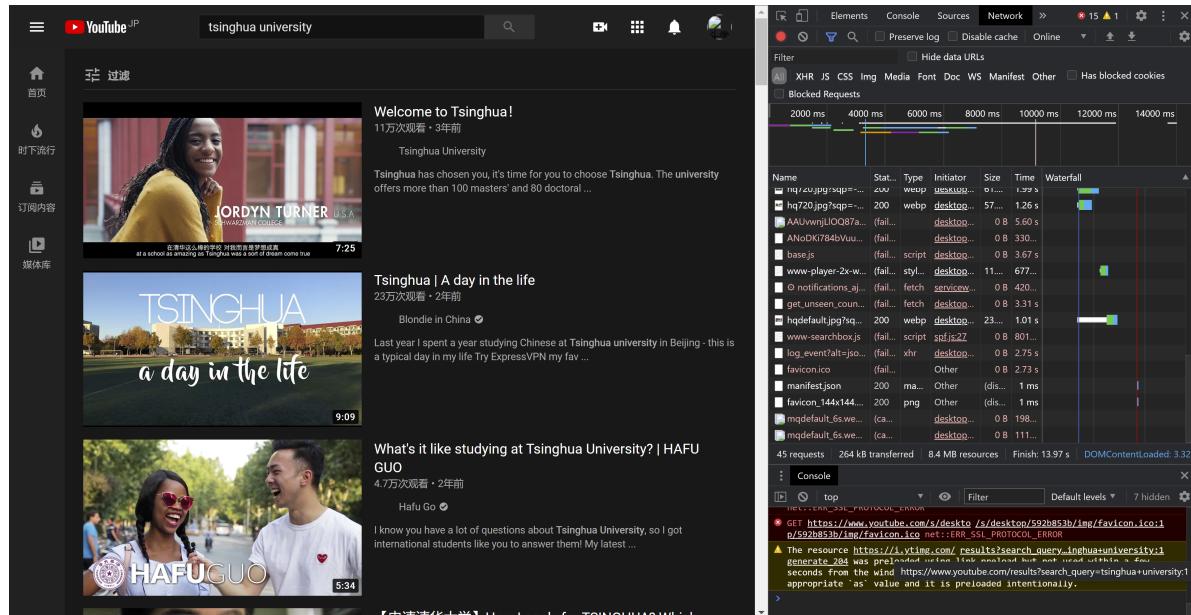
        # wait until client or remote is available for read
        r, w, e = select.select([client, remote], [], [])

        if client in r:
            data = client.recv(4096)
            # print("data", len(data))
            data = self.do_pf_encrypt(data)
            if remote.send(data) <= 0:
                break

        if remote in r:
            data = remote.recv(4096)
            # print("data", len(data))
            data = self.do_pf_decrypt(data)
            if client.send(data) <= 0:
                break

```

下面是size设置成64时的测试截图



## 7 实验分工

小组成员两人合作完成本次实验，邹振华同学主要负责SecSocks协议的实现，周炫柏同学主要负责加解密算法的设计与实现，两人共同完成了SecSocks协议的功能测试与验证

## 8 附录

[[https://ctf-wiki.github.io/ctf-wiki/crypto/asymmetric/rsa/rsa\\_theory-zh/](https://ctf-wiki.github.io/ctf-wiki/crypto/asymmetric/rsa/rsa_theory-zh/)]: