无线与物联网安全基础

课程项目-重安装键攻击

项目报告

Name: JiaHao Zou
Student ID: 3160102313
Time: 2018/11/7

a. 程序运行说明

项目构建与运行环境：Window 10 下的 Ubuntu 虚拟机，版本号为 16.04.
编译 AP.c 的命令行为：

gcc –o ap AP.c –lcrypto

编译 client.c 的命令行为：

gcc –o client client.c –lcrypto –lpthread

编译 Adversary.c 的命令行为：

gcc –o adveresary Adversary.c –lpthread

当然，我也附上了 **makefile** 文件。
运行时由于是在本地机子上做的模拟与测试，故 IP 地址都是本地环路，为 **127.0.0.1**，且需要**启动三个命令行分别运行**。
三个可执行文件的执行代码可参考如下：

./ap aaa 1234

./adversary 127.0.0.1 1234 5678

./client 127.0.0.1 5678 aaa Packet.txt

另外，要**先执行 ap，再执行 adversary，最后执行 client**。其中 Packet.txt 为我使用的传输数据（由特定单词构成），需要与执行文件放在**同一目录**下。

PS：如果遇见了编译未能通过或者无法运行等情况，可以联系我。
姓名：邹家豪
手机号： 18405895806

b. WPA2 协议正常运行的截图

1. client 连接到 AP 之后发送验证请求“Authentication_Request”；

```
jorho@jorho-virtual-machine:~/IoT_Pro$ ./client 127.0.0.1 1234 aaa Packet.txt
connected to server
--------------------------------------------------
     send msg: Authentication_Request
--------------------------------------------------
```

2. AP 收到来自 client 的验证请求后，随机生成随机数 ANonce 并初始化计数器 r = 0，将 Anonce 与 r 发送给 client；

```
jorho@jorho-virtual-machine:~/IoT_Pro$ ./ap aaa 1234
Accept client 127.0.0.1
--------------------------------------------------
     recv msg: Authentication_Request
--------------------------------------------------
       ANonce: s2iJtKYr6MJYwT2q
            r: 0
     send msg: s2iJtKYr6MJYwT2q&0
--------------------------------------------------
```

3. client 收到 ANonce 后随机生成随机数 CNonce，并使用 ANonce，CNonce 与 MasterKey 计算出传输密钥 TK；

```
     recv msg: s2iJtKYr6MJYwT2q&0
       ANonce: s2iJtKYr6MJYwT2q
            r: 0
       CNonce: SgI9T86Rk450W1cQ
     MsterKey: aaa
--------------------------------------------------
           TK: 00fe5177c655b5897f0d9135b4e51d6b6
--------------------------------------------------
```

4. client 将 CNonce 与计数器 r 一并发送给 AP；

```
     send msg: SgI9T86Rk450W1cQ&0
--------------------------------------------------
```

5. 同样地，AP 根据拿到的 CNonce 与自己的 ANonce 和 MasterKey 计算出与 client 一样的传输密钥 TK；

```
     recv msg: SgI9T86Rk450W1cQ&0
       CNonce: SgI9T86Rk450W1cQ
           TK: 00fe5177c655b5897f0d9135b4e51d6b6
--------------------------------------------------
```

6. AP 发送“Get_CNonce”信号与计数器加一（r = 1）合并的一条消息给 client，告知其已经收到了随机数 CNonce；

```
     send msg: Get_CNonce&1
--------------------------------------------------
```

7. client 收到“Get_CNonce”这个应答信号后，再发送一条 ACK 信号“Finish_Handshake”与计数器(r = 1)给 AP（这就是所谓的 msg4），然后开始初始化之后数据传输加密所要使用

的参数;

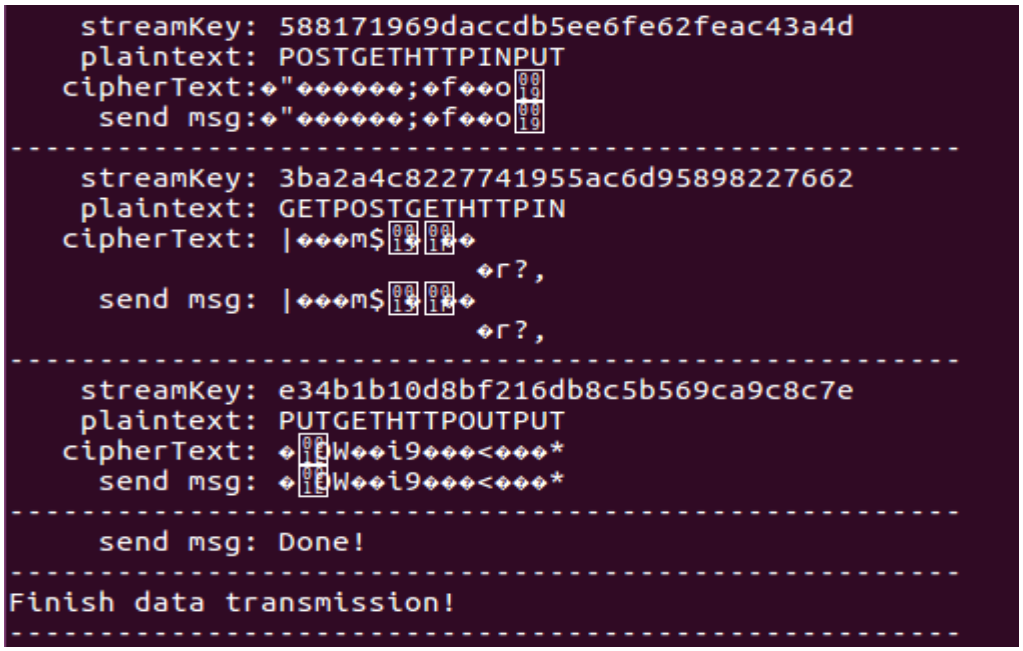
```
     recv msg: Get_CNonce&1
          r+1: 1
     ACK:Get_CNonce
     send msg: Finish_Handshake&1
-------------------------------------------------
Handshake success!
-------------------------------------------------
EncryptionKey: 00fe5177c655b5897f0d9135b4e51d6b6
        Nonce: 0
  MAC address: 00-0C-29-15-02-A2
-------------------------------------------------
```

8. AP 收到 ACK 信号“Finish_Handshake”后同样也开始初始化数据接收解密要用的参数;

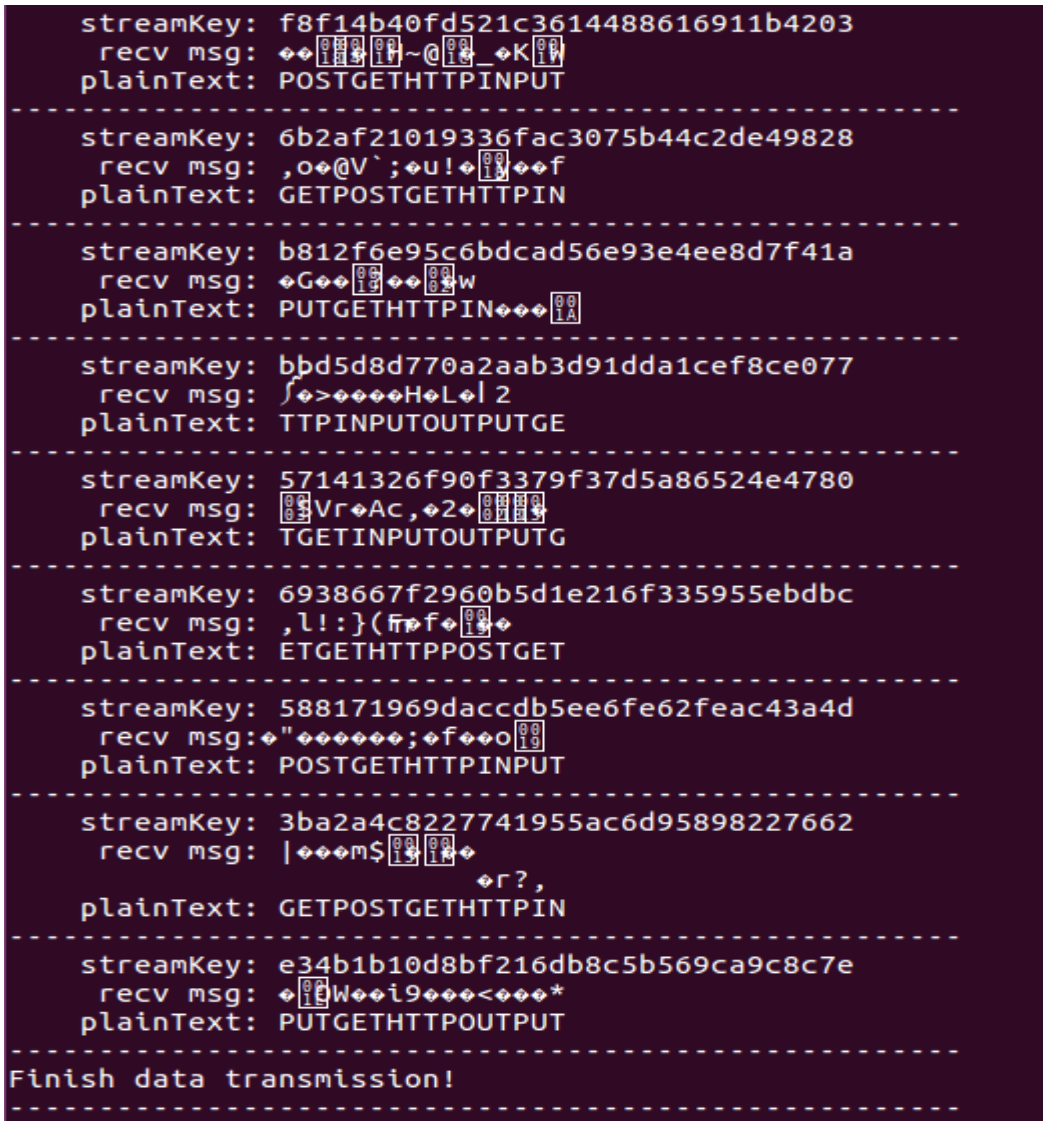
```
     recv msg: Finish_Handshake&1
-------------------------------------------------
Handshake success!
-------------------------------------------------
EncryptionKey: 00fe5177c655b5897f0d9135b4e51d6b6
        Nonce: 0
  MAC address: 00-0C-29-15-02-A2
-------------------------------------------------
```

9. client 与 AP 完成握手开始使用相同的 streamKey 进行数据加密传输与数据解密接收。
client 加密发送的数据:

```
    streamKey: f8f14b40fd521c36144886169l1b4203
    plaintext: POSTGETHTTPINPUT
   cipherText: ◆◆[illegible]
     send msg: ◆◆[illegible]
-------------------------------------------------
    streamKey: 6b2af21019336fac3075b44c2de49828
    plaintext: GETPOSTGETHTTPIN
   cipherText: ,o◆@V`;◆u!◆[illegible]
     send msg: ,o◆@V`;◆u!◆[illegible]
-------------------------------------------------
    streamKey: b812f6e95c6bdcad56e93e4ee8d7f41a
    plaintext: PUTGETHTTPINPUTH
   cipherText: ◆G◆◆[illegible]
     send msg: ◆G◆◆[illegible]
-------------------------------------------------
    streamKey: bbd5d8d770a2aab3d91dda1cef8ce077
    plaintext: TTPINPUTOUTPUTGE
   cipherText: ∫◆>◆◆◆◆H◆L◆l 2
     send msg: ∫◆>◆◆◆◆H◆L◆l 2
-------------------------------------------------
    streamKey: 57141326f90f3379f37d5a86524e4780
    plaintext: TGETINPUTOUTPUTG
   cipherText: [illegible]Vr◆Ac,◆2◆[illegible]
     send msg: [illegible]Vr◆Ac,◆2◆[illegible]
-------------------------------------------------
    streamKey: 6938667f2960b5d1e216f335955ebdbc
    plaintext: ETGETHTTPPOSTGET
   cipherText: ,l!:}(fr◆f◆[illegible]
     send msg: ,l!:}(fr◆f◆[illegible]
-------------------------------------------------
```


streamKey: 588171969daccdb5ee6fe62feac43a4d
plaintext: POSTGETHTTPINPUT
cipherText:
send msg:
streamKey: 3ba2a4c8227741955ac6d95898227662
plaintext: GETPOSTGETHTTPIN
cipherText:
send msg:
streamKey: e34b1b10d8bf216db8c5b569ca9c8c7e
plaintext: PUTGETHTTPOUTPUT
cipherText:
send msg:
send msg: Done!
Finish data transmission!

AP 解密接受的数据:


streamKey: f8f14b40fd521c3614488616911b4203
recv msg:
plainText: POSTGETHTTPINPUT
streamKey: 6b2af21019336fac3075b44c2de49828
recv msg:
plainText: GETPOSTGETHTTPIN
streamKey: b812f6e95c6bdcad56e93e4ee8d7f41a
recv msg:
plainText: PUTGETHTTPIN
streamKey: bbd5d8d770a2aab3d91dda1cef8ce077
recv msg:
plainText: TTPINPUTOUTPUTGE
streamKey: 57141326f90f3379f37d5a86524e4780
recv msg:
plainText: TGETINPUTOUTPUTG
streamKey: 6938667f2960b5d1e216f335955ebdbc
recv msg:
plainText: ETGETHTTPPOSTGET
streamKey: 588171969daccdb5ee6fe62feac43a4d
recv msg:
plainText: POSTGETHTTPINPUT
streamKey: 3ba2a4c8227741955ac6d95898227662
recv msg:
plainText: GETPOSTGETHTTPIN
streamKey: e34b1b10d8bf216db8c5b569ca9c8c7e
recv msg:
plainText: PUTGETHTTPOUTPUT
Finish data transmission!

c. AP.c 与 client.c 代码关键部分分析

在附上 WPA2 协议正常运行截图后，对 AP 与 client 代码中的部分进行解释与分析。

AP.c:

1. **Get_mac()** 函数为获取本地 MAC 地址的子函数，用于获取 MAC 地址生成 keystream；
2. **int2string()** 函数用于将 int 型变量转化为相应的字符串，由于握手过程中使用了计数器 r，计数时为整型量，但参与数据传输时，需要将其转换为相应的字符串便于传输；
3. **sendMsg()** 与 **recvMsg()** 函数为封装好的接受与发送消息的函数，并将每次发送/接收的数据打印至命令行窗口。其中 **sendMsg()** 函数需要传入参数计数器(r)，并返回 r+1，用于表示 AP 又发送了一条消息；
4. **handshake()** 函数为四步握手进行的函数，其中对于 msg4 的接收设置了一个计数器 count，在需要接收 msg4 时接收到 3 次非 msg4 的包(由于 client 发送数据包时用了 sleep(1)控制发送速度，故此时相当于在 sleep(3)的期间内未收到 msg4)，则重新发送 msg3。此外，此函数返回传输密钥 TK 用于接下来接收解密数据；
5. **encryptTransmission()** 函数为接受并解密数据的函数，每次接收 16 字节，接收完每节后执行 Nonce++更新 streamkey；
6. 主函数部分主要是绑定**本地**的 IP，等待连接等一些 socket 编程基本操作。

client.c:

1. **Get_mac(), int2string(), sendMsg(), recvMsg(), handshake()** 函数与 AP.c 中的实现的功能类似，故不再赘述；
2. **encryptTransmission()** 函数与 AP.c 中的 **encryptTransmission()** 函数有一点极大的不同，就是我在 client.c 的数据加密传输函数中使用了**双线程编程**，因为需要在发送加密数据的同时监听 AP 端是否因为 msg4 丢失而再次发送回 msg3；
3. **thread1()** 与 **thread2()** 即为我开的两个线程，thread1 负责监听来自 AP 的 msg4 信号，一旦接收到则设置全局变量 flag，告知 thread2 在接下来的数据传输中需要将 Nonce 重新置 0，再进行加密传输；thread2 则执行数据的加密传输，每次传输前都会判断一下 flag，决定是否需要初始化 Nonce，且没传输 16bytes 的数据就会 sleep(1)以限制数据传输速度。

d. 重安装键攻击具体设计

从 AP.c 与 client.c 的代码设计中，我已经实现了双方对于 msg4 丢失时的处理情况，接下来我写了 Adverrsary.c 作为中间人去实现重安装键攻击，具体设计思路如下：

1. 在主函数创建与处理好相关套接字 socket 后，执行 **msgForward()** 函数实现对 client 与 AP 之间信息的转发，同样是开了两个线程 - **listenClient()**线程用于监听 client 并转发给 AP，**listenAP()**线程用于监听 AP 并转发给 client。只有实现能转发 AP 与 client 之间的消息才能拆开包，查看其中内容，然后实现丢失 msg4，从而逼 client 重新初始化 Nonce，用同样的 keystream 加密数据并发送，从而监听窃取多组由相同 Nonce 加密的数据。

2. 在 **listenClient()**线程中对每个接收到的包做是否是 msg4 的判断，一旦发现是就不发送给 AP，继续接收接下来的来自 client 的消息。下一条消息为由 Nonce = 0 生成的 streamkey 加密的数据，下下条为 Nonce = 1 生成的 streamkey 加密数据……以此类推，直

至重新接收到来自 AP 的 msg3 发送给 client 后重新接收到一个新的 msg4。然后再丢失它重复上述过程，以获取多组由相同 Nonce 加密的数据。

判断 msg4 与窃取由 Nonce = 0 加密数据的关键部分代码如下：

```
recvMsgfromclient();
memcpy(tmp,recvBuf,strlen(msg4));
if((strcmp(tmp,msg4)==0)&&(group0<GROUPNUM)) // capture msg4 and lose it
{
    recvMsgfromclient(); //continue listening the next packet(16-bytes ciphertext encrypted with Nonce 0)
    memcpy(M_Nonce0[group0],recvBuf,DATASIZE); //steal
    //printf("M%d: %s\n",group,M_Nonce0[group]);
    sendMsgtoAP();
    group0++;
    f1=1;
}
```

3. 获取到多组由相同 Nonce 加密的数据存放在字符数组 **M_Nonce0, M_Nonce1, M_Nonce2** 中（由于之前设置的 AP 重发 msg3 的 count 为 3，故只能接收到由 Nonce=0，1，2 的加密数据）

```
//groups of ciphertext encrypted with the same Nonce
char M_Nonce0 [GROUPNUM][DATASIZE+1]={0};
char M_Nonce1 [GROUPNUM][DATASIZE+1]={0};
char M_Nonce2 [GROUPNUM][DATASIZE+1]={0};
```

4. 接下来执行 **crack()** 函数进行破解。先将连续的三组 16bytes 分别由 Nonce=0，1，2 加密的数据合并成一节（共 3*16=48bytes），这样每节之间使用的 keystream 都是一样的。

```
for(j=0;j<DATASIZE;j++){c1[j]=M_Nonce0[0][j];c2[j]=M_Nonce0[1][j];}
for(j=DATASIZE;j<(2*DATASIZE);j++){c1[j]=M_Nonce1[0][j-DATASIZE];c2[j]=M_Nonce1[1][j-DATASIZE];}
for(j=(2*DATASIZE);j<(3*DATASIZE);j++){c1[j]=M_Nonce2[0][j-(2*DATASIZE)];c2[j]=M_Nonce2[1][j-(2*DATASIZE)];}
```

然后对两节使用相同 keystream 的密文（一节 48bytes）进行字典攻击，即执行 **dictionary()** 函数。

5. **dictionary()**函数的设计思路是：传入该两节密文的开始位置（start），进行字典递归判断。

若 start >= SectionSize(48)，即来到了密文节末尾，可知此为一个递归出口，即找到了一种两密文节可能明文情况；

```
if (start == SECTIONSIZE) //recursive export
{
    return 1;
}
```

若 0<=start<= SectionSize(48) – strlen(“output”)，可知此时一节密文必有一个特定的单词”HTTP”或者”POST” 或者”GET” 或者”INPUT” 或者”OUTPUT”，因此假设为其中一个，然后依次异或出另一节密文的明文，判断该明文各字符是否是上述五个单词中包含的 10 个字母字符（H T P O S G E I N U），若都是则可以继续往下递归寻找可能的递归出口，否则走到了死路需要返回。以“HTTP”的字典检测为例：

```
//maybe "HTTP"
if (islegal('H'^stream[start]) && islegal('T'^stream[start + 1]) && islegal('T'^stream[start + 2]) && islegal('P'^stream[start + 3]))
{
    p1[start] = 'H';
    p1[start + 1] = 'T';
    p1[start + 2] = 'T';
    p1[start + 3] = 'P';
    p2[start] = 'H'^stream[start];
    p2[start + 1] = 'T'^stream[start + 1];
    p2[start + 2] = 'T'^stream[start + 2];
    p2[start + 3] = 'P'^stream[start + 3];
    countnum = dictionary(start + 4);
    if (countnum) count += countnum;
}
```

若 SectionSize(48) – strlen("output")<start< SectionSize(48)，此时第一节密文可能就会构不成完整的单词了，如"OUTPU"，"HTT"之类。故更改单词字典检测为以可能字母的检测（可能字母指 5 个单词中可能出现的 10 个字母）。以"H"检测为例：

```
if (islegal('H'^stream[start]))
{
    p1[start] = 'H';
    p2[start] = 'H'^stream[start];
    countnum = dictionary(start + 1);
    if (countnum) count += countnum;
}
```

其中 **islegal()** 函数为检测字符是否为五个单词中的 10 个字母的子函数。

6. 通过上述字典攻击我们仍然会得到几十种可能的两节明文组合，故需要进行过滤。接下来执行了 **filter()**函数。此函数的设计思路是：由于**有些字母在五个单词中只出现了一次**，比如"I"，意味着其后四个字母必然得是'N' 'P' 'U' 'T'以构成"INPUT"；还有'G'，意味着其后两个字母必然得是'E' 'T'"以构成"GET"；还有'H'，其后字母必然为'T' 'T' 'P'以构成"HTTP"；最后还有'S'，两边为'P' 'O'及'T'以构成"POST"。上述的前提是这些字节仍存在于 48bytes 长度的第二个密文对应的明文中。因此可以通过这个特性对每种可能的第二节明文进行判断，一旦不满足上述特性，就可以判断第二节明文**一定不单单由指定的五个单词构成**，这些可能情况就被排除过滤掉了。以'G'为例：

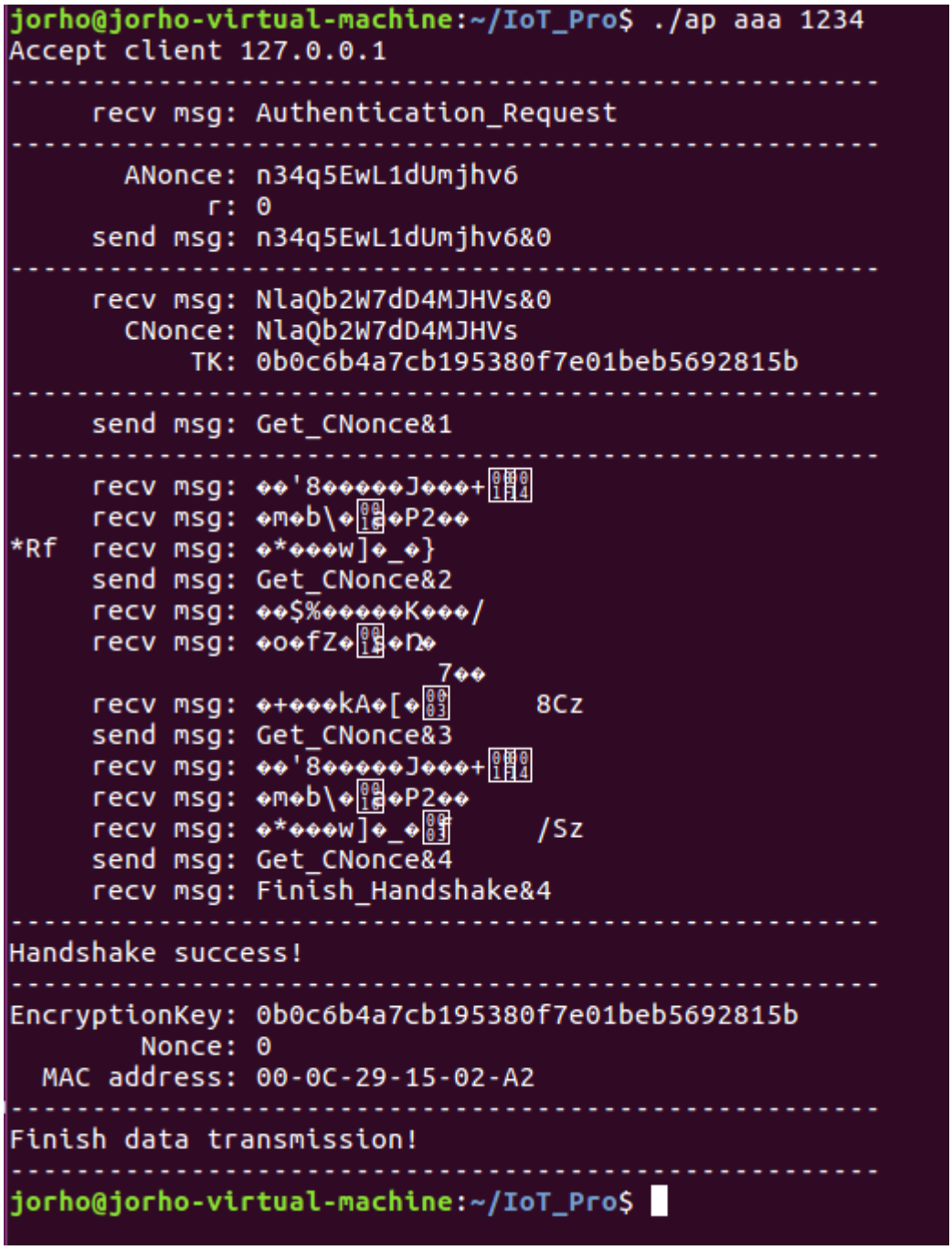
```
if(p2[i]=='G'&&(i<(SECTIONSIZE-strlen("GET"))))
{
    if(p2[i+1]!='E'||p2[i+2]!='T') return 0;
    else continue;
}
```

7. 此时可能的情况就只有几种了，由于在第一节明文的结尾处与第二节明文的开始处，即衔接的地方没有作严格的过滤，但是由于此时情况只有几种，可以将其以及每组对应的 keystream 都输出出来，观察衔接处衔接后是否可以构成五种单词以决定最后真正获取的明文（共 48bytes*2=96bytes），以及正确的 keystream。

此处具体如何判断得到明文与 keystream 可见后面结合运行截图的说明。

e. 重安装攻击运行截图及效果分析

AP 端在重安装攻击中的运行截图如下：

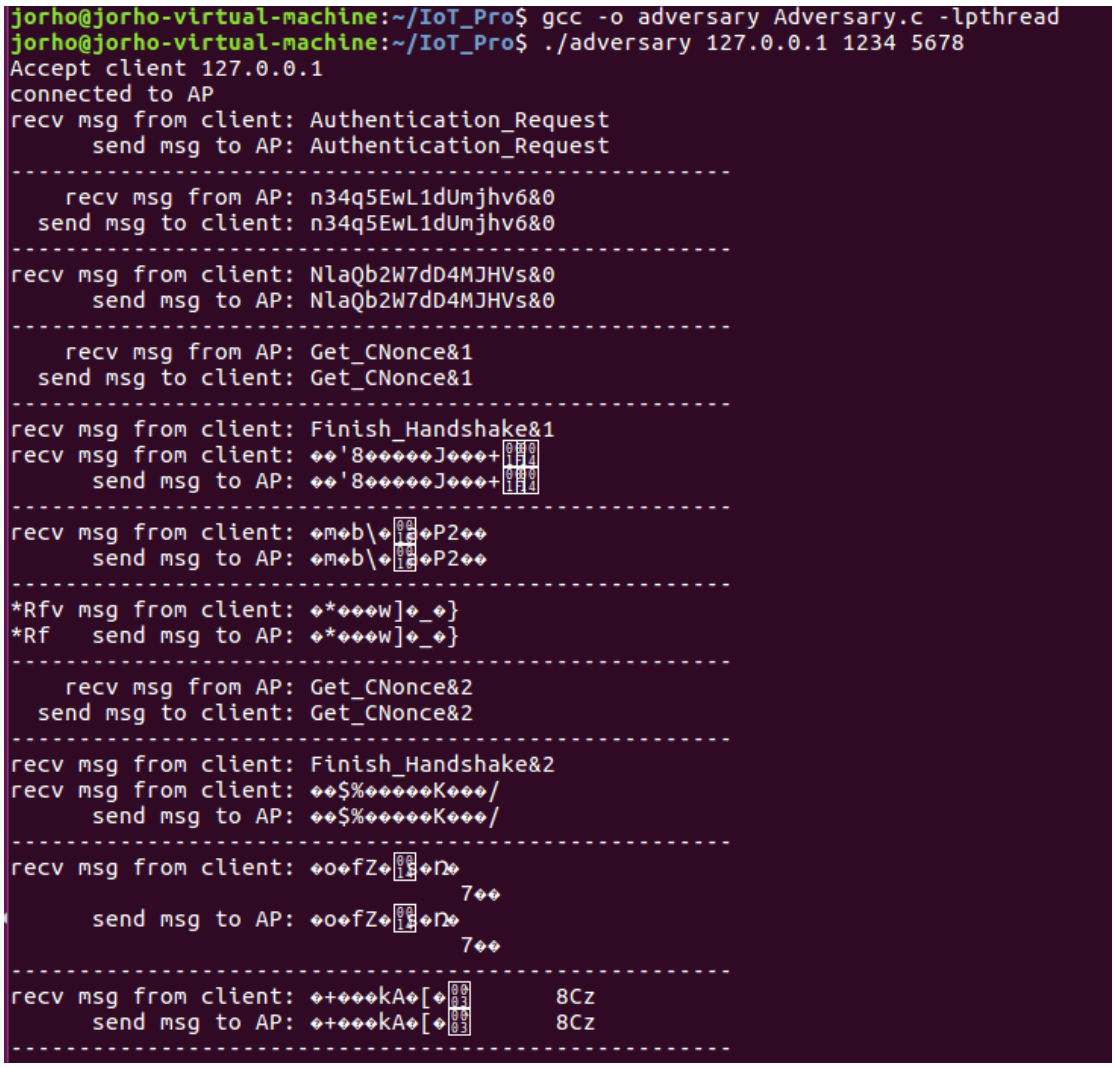


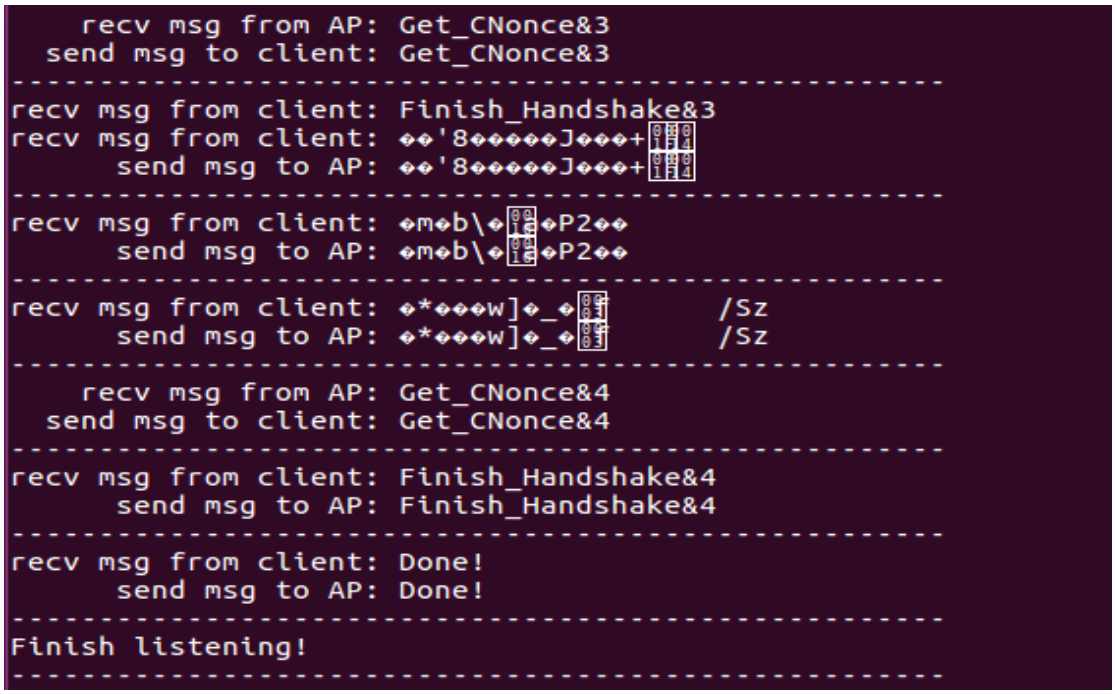
Client 端在重安装攻击中运行截图如下：

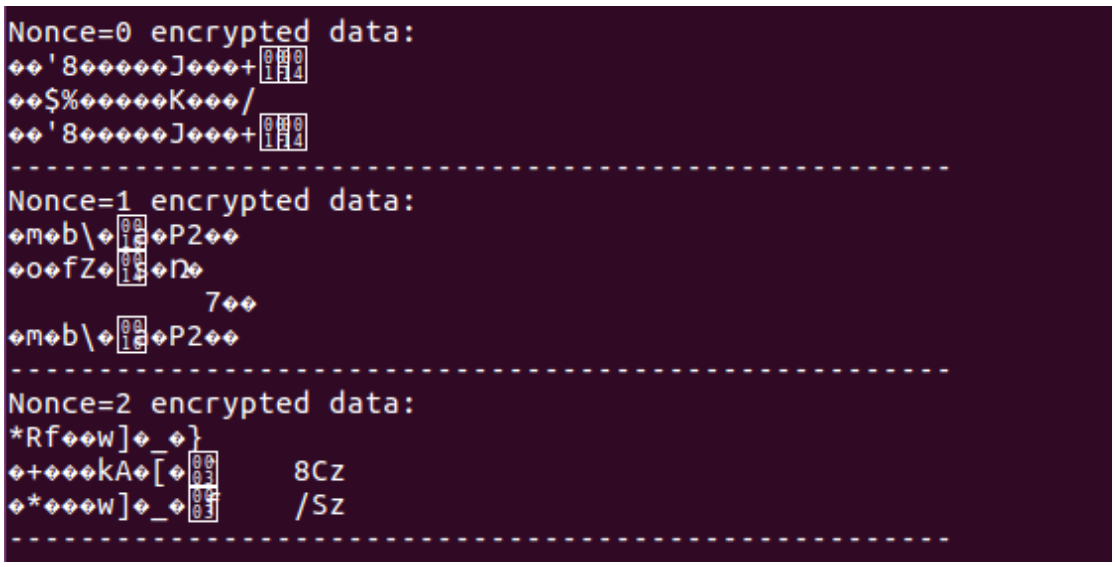
```
jorho@jorho-virtual-machine:~/IoT_Pro$ ./client 127.0.0.1 5678 aaa Packet.txt
connected to server
----------------------------------------------------
     send msg: Authentication_Request
----------------------------------------------------
     recv msg: n34q5EwL1dUmjhv6&0
       ANonce: n34q5EwL1dUmjhv6
            r: 0
       CNonce: NlaQb2W7dD4MJHVs
     MsterKey: aaa
----------------------------------------------------
           TK: 0b0c6b4a7cb195380f7e01beb5692815b
----------------------------------------------------
     send msg: NlaQb2W7dD4MJHVs&0
----------------------------------------------------
     recv msg: Get_CNonce&1
         r+1: 1
     ACK:Get_CNonce
     send msg: Finish_Handshake&1
----------------------------------------------------
Handshake success!
----------------------------------------------------
EncryptionKey: 0b0c6b4a7cb195380f7e01beb5692815b
        Nonce: 0
  MAC address: 00-0C-29-15-02-A2
----------------------------------------------------
    streamKey: e2be746cdee8a7a1ea1eccfb8c7b4a40
    plaintext: POSTGETHTTPINPUT
   cipherText: ◆◆'8◆◆◆◆◆J◆◆◆+[illegible]
     send msg: ◆◆'8◆◆◆◆◆J◆◆◆+[illegible]
----------------------------------------------------
    streamKey: e228813213e44426829ae9fa5c62c48b
    plaintext: GETPOSTGETHTTPIN
   cipherText: ◆m◆b\◆[illegible]◆P2◆◆
     send msg: ◆m◆b\◆[illegible]◆P2◆◆
----------------------------------------------------
    streamKey: 9b7ff3cca52315dc0b854c335d7f062e
    plaintext: PUTGETHTTPINPUTH
*RfcipherText: ◆*◆◆◆w]◆_◆}
*Rf  send msg: ◆*◆◆◆w]◆_◆}
     recv msg: Get_CNonce&2
     send msg: Finish_Handshake&2
```

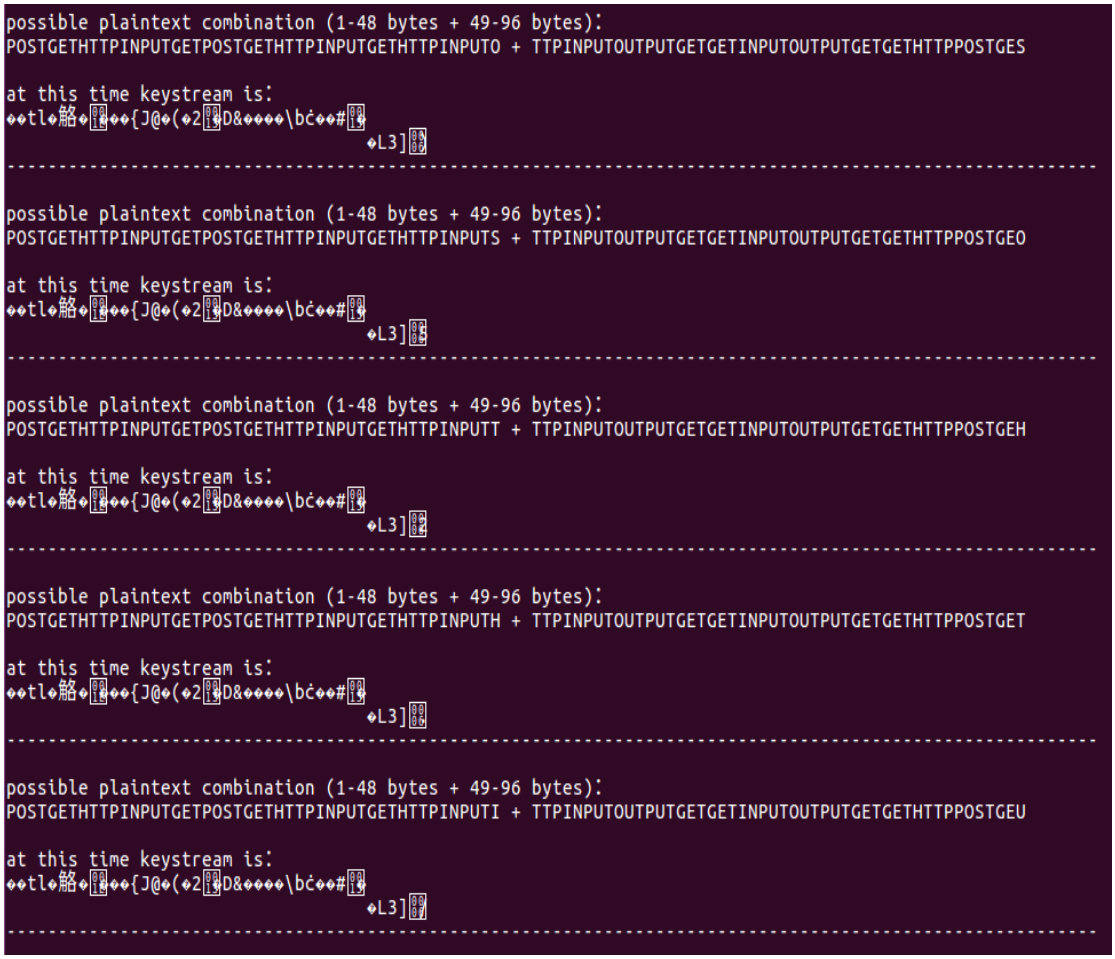
```
     send msg: Finish_Handshake&2
----------------------------------------------------
    streamKey: e2be746cdee8a7a1ea1eccfb8c7b4a40
    plaintext: TTPINPUTOUTPUTGE
   cipherText: ◆◆$%◆◆◆◆◆K◆◆◆/
     send msg: ◆◆$%◆◆◆◆◆K◆◆◆/
----------------------------------------------------
    streamKey: e228813213e44426829ae9fa5c62c48b
    plaintext: TGETINPUTOUTPUTG
   cipherText: ◆o◆fZ◆[illegible]◆n◆
                              7◆◆
     send msg: ◆o◆fZ◆[illegible]◆n◆
                              7◆◆
----------------------------------------------------
    streamKey: 9b7ff3cca52315dc0b854c335d7f062e
    plaintext: ETGETHTTPPOSTGET
   cipherText: ◆+◆◆◆kA◆[◆[illegible]      8Cz
     send msg: ◆+◆◆◆kA◆[◆[illegible]      8Cz
     recv msg: Get_CNonce&3
     send msg: Finish_Handshake&3
----------------------------------------------------
    streamKey: e2be746cdee8a7a1ea1eccfb8c7b4a40
    plaintext: POSTGETHTTPINPUT
   cipherText: ◆◆'8◆◆◆◆◆J◆◆◆+[illegible]
     send msg: ◆◆'8◆◆◆◆◆J◆◆◆+[illegible]
----------------------------------------------------
    streamKey: e228813213e44426829ae9fa5c62c48b
    plaintext: GETPOSTGETHTTPIN
   cipherText: ◆m◆b\◆[illegible]◆P2◆◆
     send msg: ◆m◆b\◆[illegible]◆P2◆◆
----------------------------------------------------
    streamKey: 9b7ff3cca52315dc0b854c335d7f062e
    plaintext: PUTGETHTTPOUTPUT
   cipherText: ◆*◆◆◆w]◆_◆[illegible]      /Sz
     send msg: ◆*◆◆◆w]◆_◆[illegible]      /Sz
     recv msg: Get_CNonce&4
     send msg: Finish_Handshake&4
----------------------------------------------------
     send msg: Done!
----------------------------------------------------
Finish data transmission!
----------------------------------------------------
jorho@jorho-virtual-machine:~/IoT_Pro$
```

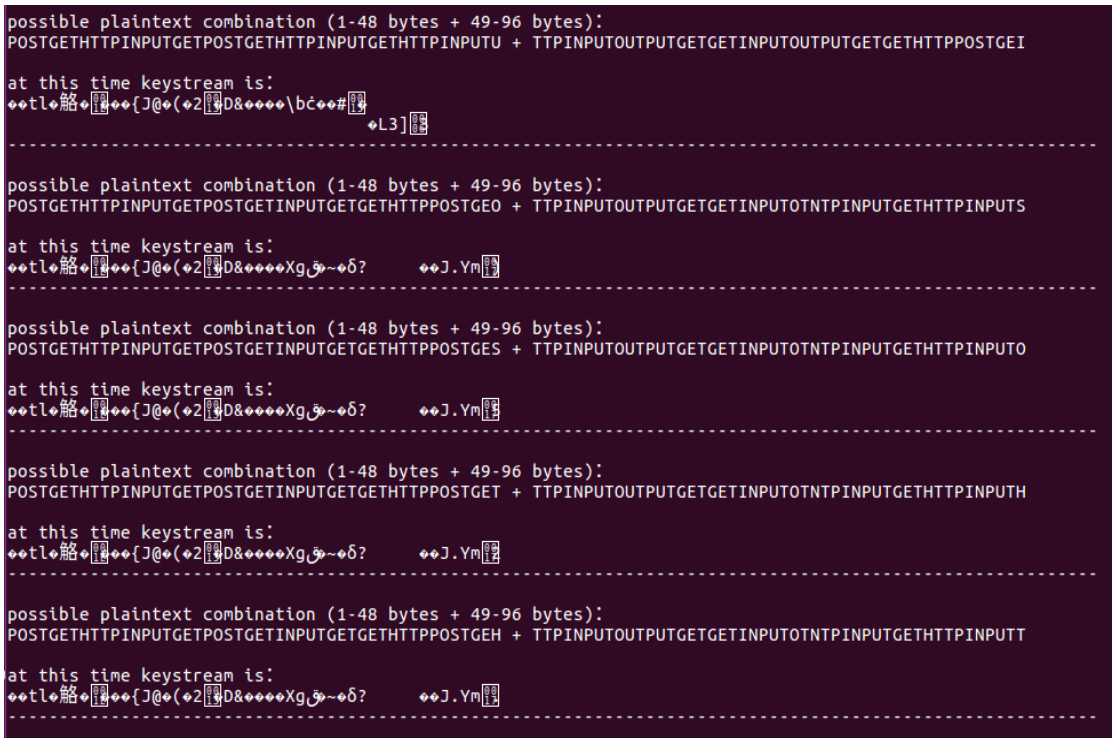
攻击者 adversary 端在重安装攻击中运行截图如下:

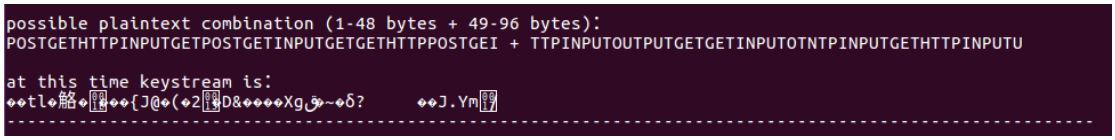





Nonce=0 encrypted data:
Nonce=1 encrypted data:
Nonce=2 encrypted data:


possible plaintext combination (1-48 bytes + 49-96 bytes):
POSTGETHTTPINPUTGETPOSTGETHTTPINPUTGETHTTPINPUTO + TTPINPUTOUTPUTGETGETINPUTOUTPUTGETGETHTTPPOSTGES
at this time keystream is:
possible plaintext combination (1-48 bytes + 49-96 bytes):
POSTGETHTTPINPUTGETPOSTGETHTTPINPUTGETHTTPINPUTS + TTPINPUTOUTPUTGETGETINPUTOUTPUTGETGETHTTPPOSTGEO
at this time keystream is:
possible plaintext combination (1-48 bytes + 49-96 bytes):
POSTGETHTTPINPUTGETPOSTGETHTTPINPUTGETHTTPINPUTT + TTPINPUTOUTPUTGETGETINPUTOUTPUTGETGETHTTPPOSTGEH
at this time keystream is:
possible plaintext combination (1-48 bytes + 49-96 bytes):
POSTGETHTTPINPUTGETPOSTGETHTTPINPUTGETHTTPINPUTH + TTPINPUTOUTPUTGETGETINPUTOUTPUTGETGETHTTPPOSTGET
at this time keystream is:
possible plaintext combination (1-48 bytes + 49-96 bytes):
POSTGETHTTPINPUTGETPOSTGETHTTPINPUTGETHTTPINPUTI + TTPINPUTOUTPUTGETGETINPUTOUTPUTGETGETHTTPPOSTGEU
at this time keystream is:

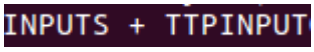




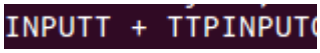
效果分析:

由攻击者的破解运行结果，可以看到两节明文衔接处依次出现了下述可能的情况:

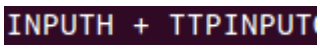
(1)



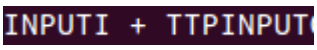
(2)



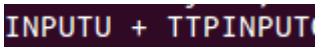
(3)



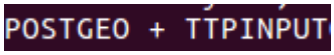
(4)



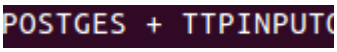
(5)



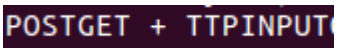
(6)



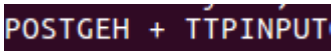
(7)



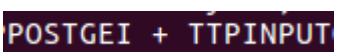
(8)



(9)



(10)



(11)

通过观察可知，只有 **(4)** 是在衔接处也符合条件的，即“INPUT” + “HTTP” + “INPUT“。故输出的经过滤后的 11 种可能情况中，(4) 的明文为正确的明文，伴随其一同输出的 keystream 即为正确的 keystream。因此，我们就根据获取到的有相同 keystream 加密的密文节得到了相应正确的明文节与 keystream。

显然，效果上是可以实现破解，但是最后还是需要通过观察衔接处的单词构成情况从几种情况中得到正确的值，若密文节一长就会增大破解难度，因此**在破解算法上仍然存在着不足和改进空间**，比如可以继续优化，在衔接处也进行字典检测，判断衔接处是否也是合法的

单词，从而做到破解得到的明文可以是唯一的。