<https://www.cnblogs.com/softidea/p/4550552.html>

SSh协议：

　　全称为Secure Shell，即很安全的shell，主要目的是用来取代传统的telnet和r系列命令（rlogin，rsh，rexec等）远程登录和远程执行命令的工具，实现远程登录和远程执行命令加密，防止由于网络监听而出现的密码泄露，从而对系统构成威胁。（telnet协议采用明文传送密码，数据传送过程中也不加密）

　　ssh协议目前有ssh1 和ssh2，其实现在我们主要使用的也是openssh。ssh不仅在登录过程中对密码进行加密传送，而且在登录后执行的命令的数据也进行加密，这样即使别人在网络上监听并截获了你的数据包，他也看不到其中的内容。

  之前为了自己做一套SSH，先自己实现了一套telnet。但经过这么多天的苦逼，发现以前的工作都是徒劳。ssh的协议很繁杂，核心的内容在于密码算法，而且自己很难在网上找到周全的细节讲解与详细的实现，只有靠自己刷RFC和问大神还有就是靠强X我的服务器艰难地完成。

    现计算了下时间，自己做SSH耗费了进两个月的时间，虽然期间也夹着其他的繁杂事物，但自己在这方面确是是耗费了非常大的精力。因为这方面详细资料的匮乏，自己以前也几乎没有接触过密码学方面的东西，很多只有靠自己摸索，所以我得经常拿我自己的服务器来做黑盒测试，我现在服务器上的ss服务器日志全是一些非法连接的记录（—\_—|||）。早知当初就不那么作死非要自己实现他的加密算法和过程，用openssl就很快搞定了。但我还是觉得这次做SSH的精力是我受益匪浅，不仅熟悉了各种加密，并且能靠自己实现并熟练应用了。可能这些对自己帮助不大，但至少和信安的小伙伴也有点吹牛的谈资了~

    这篇文章希望能帮助到想了解ssh2.0协议或是亲手实现ssh协议的小伙伴。

    首先对数据包的格式进行说明：

    数据包由包长度（Packet Length）、填充长读（Padding Length）、信息代码（Msg code）、信息内容与填充值（Padding String） 这5部分组成。信息内容中的一些字符串以4字节长度+该长度数量的字符组成，数值按照网络序排列，例如：abc: 00 00 00 03 (char)a (char)b (char)c 。另外有一种大整数的情况，负数和字符串的表示方式一样，正数需要前导0，例如 4b64： 00 00 00 03 00 4b 64 。

    ssh头的结构体:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | struct sshhead  {      unsigned int tlen;      unsigned char plen;      unsigned char msgcode;      sshhead(){tlen=6;}  }; |

    就拿通过ssh远程控制的一个完整个过程来讲，ssh的过程可分为以下3部分：

    一、版本协商

    二、算法协商与密钥交换

    三、加密通信（可能含有2、3部分）

    这其中第二部分是ssh最为核心的过程，该过程决定了以后通信所要使用的密钥，下面按顺序对每个部分对比着数据包进行详细的讲解并给出实现的过程。

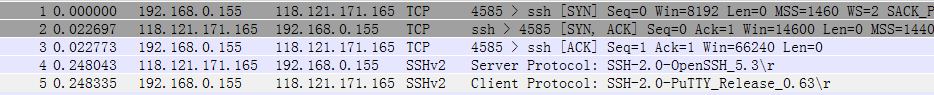
    一、版本协商：

    在建立连接后，客户端与服务器分别向对方发送自己ssh的版本信息（这里的数据格式不同于其他包，只有一行版本号），以\r\n结束。版本的格式如下：

                         SSH-ssh协议版本-详细版本\r\n （几乎只有ssh协议版本之前的信息有效）

    比如我linux上的就是：SSH-2.0-OpenSSH\_5.3\r\n

    Putty的是：          SSH-2.0-PuTTY\_Release\_0.63\r\n

[](http://s3.51cto.com/wyfs02/M00/6C/AC/wKiom1VPZjmTdIXCAAG0K4Nz_IE978.jpg)

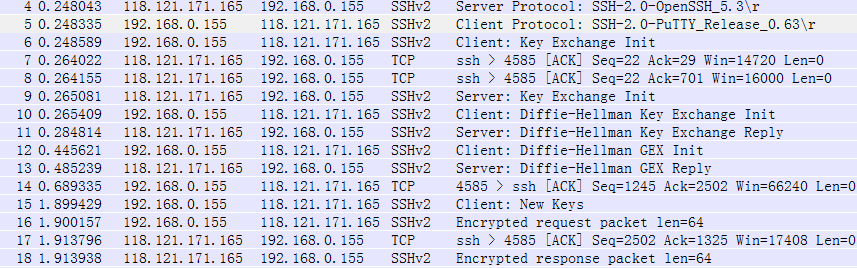
    一般来说，在建立连接后，是先由服务器发版本号过来，单线程处理版本协商的朋友需要注意下。

    在双方收到对方发来的版本号后，会根据两者之中最小的版本来进行接下来的通讯。

    二、算法协商与秘钥交换：

    这部分的内容将会占该文章总篇幅的一半以上。

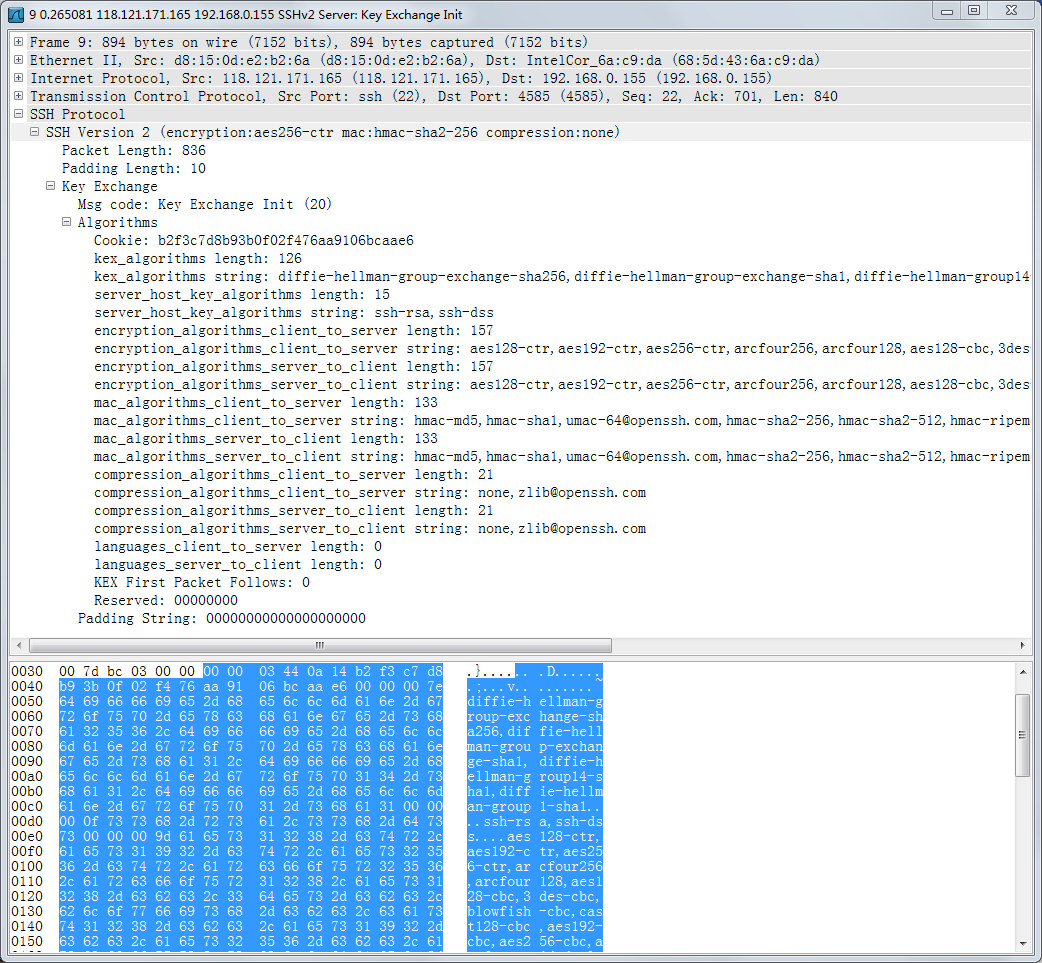
    首先给大家看下整个过程的数据包大概：

[](http://s3.51cto.com/wyfs02/M02/6C/A7/wKioL1VPaBniGxCPAAShyr4gqdU270.jpg)    整个部分是从第6条开始到第15条结束，除去中间的非协议部分，总共有7条数据包。看起来只有这么几条数据包，但其中包含了非常多的过程与隐秘的信息。

    1、算法协商：

        位第6、9数据，分别为双发向对法发送的自己在不同密码需求上支持的算法。

     该数据包的格式:

[](http://s3.51cto.com/wyfs02/M02/6C/AA/wKioL1VQF13DTV-nAAxjoz_gC3M614.jpg)

    按顺序分别是：

        cookie(随机的值，16byte)

        kex\_algorithms（秘钥租交换算法）

        server\_host\_key\_algorithms（服务器主机秘钥，正常情况用处不大，甚至可以不用）

        encryption\_algorithms\_client\_to\_server（两端通信使用的加密算法）

        encryption\_algorithms\_server\_to\_client

        mac\_algorithms\_client\_to\_server（数据校验用的hash算法）

        mac\_algorithms\_server\_to\_client

        compression\_algorithms\_client\_to\_server（压缩算法）

        compression\_algorithms\_server\_to\_client

        languages\_client\_to\_server

        languages\_server\_to\_client

        first\_kex\_packet\_follows

        0（4byte整数，扩展用的）

    每个算法类型可能会有多个不同的算法，这些算法之间使用逗号隔开。

    现在双方知道对方支持的算法，但是应该怎样决定每个类型实际所使用的算法呢？

    每个算法类型列表的第一个算法必须是首选的算法，服务器应以客户端的算法优先级作为考虑，就拿交换算法举例：

        现在服务器有三个算法dh1,dh2,dh3

            客户端有两个算法dh3,dh2

        那么服务器的首选算法是dh1，而客户端是dh3，客户端此时知道服务器有dh3算法，因此客户端就确认使用dh3算法。服务器发现自己的首选算法与客户端不同，而自己拥有客户端的首选算法，因此服务器也确认使用dh3算法。

        再看另一个情况

            服务器：dh1,dh2,dh3

            客户端：dh4,dh3,dh1

        这时服务器没有客户端的首选算法，客户端会使用第二个算法dh3，此时服务器也支持第二个算法，双方将确定使用dh3算法。

    如果服务器和客户端双方没有共同的算法，这次会话将会终止。

    下面是代码实现和服务器之间的版本协商

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #define KEI     (char)20  #define NK      (char)21  //算法名  #define VER "SSH-2.0-WCHRT\_1.0\r\n"  #define COOKIE "0123456789ABCDEF"  #define VKEX "diffie-hellman-group-exchange-sha256"  #define VSHK "ssh-rsa"  #define VECS "aes128-cbc"  #define VESC "aes128-cbc"  #define VMCS "hmac-sha1"  #define VMSC "hmac-sha1"  #define VCCS "none"  #define VCSC "none"  #define VLCS ""  #define VLSC ""  #define KFPF "" |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | bool key\_exchange()  {      sshhead sshh;      sshh.msgcode=KEI;      sshh.tlen-=4;      //将算法列表与信息分别写入缓冲区      mstrin(COOKIE,sshh.tlen);      mstrin(VKEX,sshh.tlen);      mstrin(VSHK,sshh.tlen);      mstrin(VECS,sshh.tlen);      mstrin(VESC,sshh.tlen);      mstrin(VMCS,sshh.tlen);      mstrin(VMSC,sshh.tlen);      mstrin(VCCS,sshh.tlen);      mstrin(VCSC,sshh.tlen);      mstrin(VLCS,sshh.tlen);      mstrin(VLSC,sshh.tlen);      //ed      for(int i=0;i<5;sshh.tlen++,i++)      {          data[sshh.tlen]=(char)0;      }  //载荷的计算与总长度的写入都放在最后      //count padding length      count\_padding(sshh);      sshheadin(sshh);        //没有封装socket      len=send(sock,data,sshh.tlen+4,0);      mrecv(10);        //printf("(%d)",len);      if(data[5]==KEI)      {          return true;      }        return false;  } |

用到的一些功能函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | //向缓冲区填充字符串，长度使用网络字节序  void mstrin(string s,unsigned int &tlen)  {      data[tlen]=(char)(s.length()/(256\*256\*256));      data[tlen+1]=(char)(s.length()/(256\*256));      data[tlen+2]=(char)(s.length()/256);      data[tlen+3]=(char)(s.length());      tlen+=4;      for(int i=0;i<s.length();tlen++,i++)      {          data[tlen]=s[i];      }      data[tlen]='\0';  }    void sshheadin(sshhead &sshh)  {      sshh.tlen-=4;      data[0]=(char)(sshh.tlen/(256\*256\*256));      data[1]=(char)(sshh.tlen/(256\*256));      data[2]=(char)(sshh.tlen/256);      data[3]=(char)(sshh.tlen);      data[4]=(char)sshh.plen;      data[5]=sshh.msgcode;  }    void count\_padding(sshhead &sshh)  {      int k=2;      if(sshh.tlen%8<4)      {          k=1;      }      sshh.plen=(sshh.tlen/8+k)\*8-sshh.tlen;      sshh.tlen=(sshh.tlen/8+k)\*8;  } |

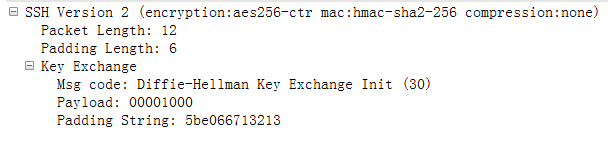
    2、秘钥交换

        在算法协商成功过后，双方便立马进行秘钥组的交换。ssh2.0版本所使用的秘钥组交换协议算法主要使用diffie-hellman-group-exchange-sha算法。

         鉴于该部分内容特别多，我特意在另一篇单独的文章中予以详细介绍，再阅读下文前请先参考该文章：[dh-gex-sha算法详解](http://wchrt.blog.51cto.com/8472636/1649839" \o "" \t "_blank)

        我们数据包的第10到15条都是该部分的内容

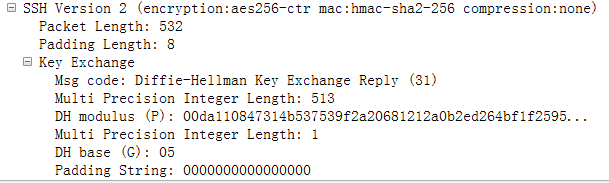
    1、dh key exchange init （C）

[](http://s3.51cto.com/wyfs02/M01/6C/B9/wKiom1VQlETzbGlYAADzwiKKb68920.jpg)

        密钥交换初始化，由客户端先向服务器发送秘钥交换请求的数据包，告知开始秘钥交换。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | //<<<<<<<<<<DH KEX INIT<<<<<<<<<      sshh.tlen=6;      sshh.msgcode=DHKEI;      //payload      mintin(0x1000,sshh.tlen);      count\_padding(sshh);      sshheadin(sshh);        len=send(sock,data,sshh.tlen+4,0);        //dh: set I\_C      dhdata.set\_i\_c(string(data+4,sshh.tlen));      //dhdata.set\_i\_c(string(data,len));      //>>>>>>>>>>>>>>>>>>>>>>>> |

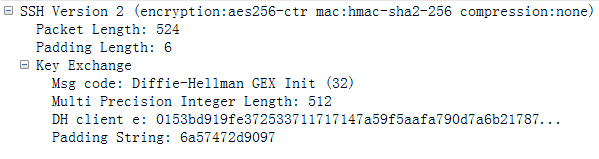
     2、dh key exchange reply  （S)

[](http://s3.51cto.com/wyfs02/M02/6C/B4/wKioL1VQlb2hzXk5AAFqficzp28999.jpg)

        服务器收到客户端发起交换的请求后，将自己用于dh算法的P、G发送给客户端，用于客户端生成dh公私钥。这里的P是一个大素数，而G是大于1的数，G不必过大，10位以内最后，因为按幂运算G能轻易生成特别大的数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | //<<<<<<<<<<DH KEX REPLAY<<<<<<<<<      mrecv(10);        if(data[5]!=DHKER)      {          puts("DH KEX REPLAY error");          return false;      }        //dh: set I\_S      dhdata.set\_i\_s(string(data+4,len-4));      //dhdata.set\_i\_s(string(data,len));        //dh: read P      pos=6;      intlen=readstrint(data+pos);      pos+=4;      Integer p=readstrbigint(data+pos,intlen);      pos+=intlen;      //dh: read G      intlen=readstrint(data+pos);      pos+=4;      Integer g=readstrbigint(data+pos,intlen);      pos+=intlen;      //dh: set G and P      dhdata.set\_g\_and\_p(g,p);      //cout<<dhdata.dh\_p<<" "<<dhdata.dh\_g<<endl;      //>>>>>>>>>>>>>>>>>>>>>>>> |

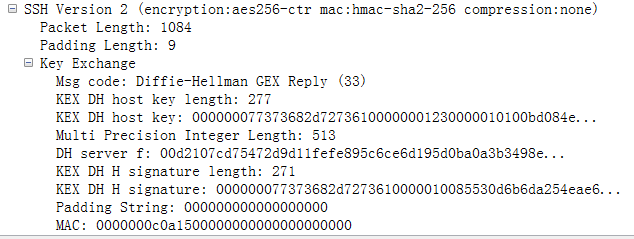
       3、dh gex init   (C)

[](http://s3.51cto.com/wyfs02/M02/6C/B9/wKiom1VQlESyBtOiAAEmKjwKdCI261.jpg)

        客户端收到服务器发过来的P、G后，自己变成根据P、G生成并计算出自己的公钥e。这一步也只需要客户端将生成的e发送给服务器即可。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | //<<<<<<<<<<DH GEX INIT<<<<<<<<<      sshh.tlen=6;      sshh.msgcode=DHGI;        dhdata.comp\_e();      string e=inttostr(dhdata.get\_e(),256);      mstrin(e,sshh.tlen);      count\_padding(sshh);      sshheadin(sshh);        len=send(sock,data,sshh.tlen+4,0);    //debugstr(data,len);      //>>>>>>>>>>>>>>>>>>>>>>>> |

        4、dhgex reply   （S）

[](http://s3.51cto.com/wyfs02/M01/6C/B4/wKioL1VQlb3Q7XbuAAHn6JiQYeA671.jpg)

        重要的来了，服务器收到了客户端发来的e后，便能计算出共享秘钥K，并根据现有信息计算出生成所需秘钥的H。

        这个数据包里面含有如下信息：

        KEX DH host key（K\_S）：

            主机公钥，一般为rsa公钥。完整的格式为：总长度+算法名长度+算法名+证书（n）长度+证书（n）+公钥长度+公钥。

        DH server f ：

            服务器的dh公钥值，客户端收到后便能用f计算出同样的共享秘钥K。

        KEX DH H signature (签名后的H):

            服务器用主机私钥对计算出的hash值H进行签名的结果。格式为：总长度+算法名长度+算法名+签名数据长度+签名值。

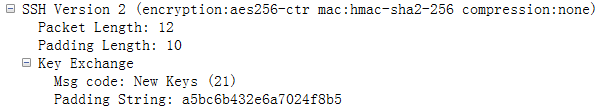
            H的计算方法： H=hash(V\_C||V\_S||I\_C||I\_S||K\_S||e||f||K);

             按顺序用到的值（注意类型）：

|  |  |  |
| --- | --- | --- |
| 类型 | 值 | 说明 |
| string | V\_C | 客户端的初始报文（版本信息：SSH-2.0-xxx，不含结尾的CR和LF） |
| string | V\_S | 服务器的初始报文 |
| string | I\_C | 客户端 SSH\_MSG\_KEX\_INIT的有效载荷（不含开头的数据长度值） |
| string | I\_S | 服务器的同上 |
| string | K\_S | 主机秘钥（dh gex reply（33)过程服务器发送host key (RSA公钥)） |
| mpint | e | 客户端DH公钥 |
| mpint | f | 服务器DH公钥 |
| mpint | K | 共同DH计算结果 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | //<<<<<<<<<<DH GEX REPLAY<<<<<<<<<      mrecv(10);      if(data[5]!=DHGR)      {          puts("DH GEX REPLAY error");          system("pause");          return false;      }      int padlen=data[4];      //dh: set server host key      pos=6;      intlen=readstrint(data+pos);//host key all length      dhdata.set\_k\_s(string(data+pos+4,intlen));      pos+=4;        intlen=readstrint(data+pos);//host key name      pos+=4;      pos+=intlen;      intlen=readstrint(data+pos);//get rsa e and n      pos+=4;      Integer ee=readstrbigint(data+pos,intlen);      pos+=intlen;      intlen=readstrint(data+pos);      pos+=4;      Integer nn=readstrbigint(data+pos,intlen);//set rsa e and n      pos+=intlen;      dhdata.set\_e\_and\_n(ee,nn);        //dh: set dh server f      intlen=readstrint(data+pos);      pos+=4;      dhdata.set\_f(readstrbigint(data+pos,intlen));      pos+=intlen;      //dh: set shka\_name      pos+=4;//h's total length      intlen=readstrint(data+pos);      pos+=4;      dhdata.set\_shka\_name(string(data+pos,intlen));      pos+=intlen;      //dh: set server h      intlen=readstrint(data+pos);      pos+=4;      dhdata.set\_s\_h(string(data+pos,intlen));      pos+=intlen;      pos+=padlen;      //and other MAC//        dhdata.comp\_k();      dhdata.comp\_h(); |

         5、new keys   （C)

[](http://s3.51cto.com/wyfs02/M00/6C/B4/wKioL1VQlb2yH3rJAADMhK9tZOw215.jpg)

          客户端收到服务器的信息后计算出K，并用同样的方式计算出H（服务器和客户端的H都是同一个值）。并使用服务器发过来的K\_S验证服务器发过来的签名后的H，如果验证一致，则说明此次秘钥交换成功。客户端向服务器发送new key，标志秘钥交换过程的结束。如果此次秘钥交换是整个会话的第一次交换，则计算出的H也是整个会话的会话ID（session\_id）。

      秘钥基本信息在网络上的传输与交换，接下来就分别是服务器和客户端各自使用现有信息计算出以后加解密所要使用的秘钥。秘钥计算：

            这里的加密秘钥指的是以后数据通信所用的秘钥，一般用aes算法。

            计算方式：hash(K,H,单个字符,session\_id);

            单个字符指的是单个大写的ASCII字母，根据不同的加密秘钥选择不同的字符来计算。

|  |  |
| --- | --- |
| 字母 | 秘钥 |
| 'A' | 客户端到服务器的初始IV（CBC） |
| 'B' | 服务器到客户端的初始IV |
| 'C' | 客户端到服务器的加密秘钥（对称秘钥） |
| 'D' | 服务器到客户端的加密秘钥 |
| 'E' | 客户端到服务器的完整性秘钥(HMAC) |
| 'F' | 服务器到客户端的完整性秘钥 |

        就以aes-cbc为例子，aes对称加解密所需要用到的值有初始IV与对称秘钥。这里的初始IV指的是cbc模式中加解密的初始向量，第二次加解密需要IV的值，以后的每次的加解密都要依赖于上一次加解密的数据。

    三、加密通信

       此时双方都拥有协商好的算法以及用于加解密的秘钥,现在开始所有传输的全部数据都要进行加密（包含总长度），并使用同样的。

        在加密通信的过程中，双方允许重新发送KEX秘钥交换请求。这时整个秘钥交换过程的数据将会使用现有密钥加解密。在该次秘钥交换的过程中也会生成一个H值，但该H值不会影响到此次会话的session\_id，session\_id只是会话第一次秘钥交换生成的H值。在秘钥交换最后客户端发出new keys请求时。双方会放弃当前使用的秘钥，使用新协商的秘钥继续通信。

        在远程数据的通信过程中，双方使用SSH\_MSG\_CHANNEL\_DATA标志消息类型进行数据传输。

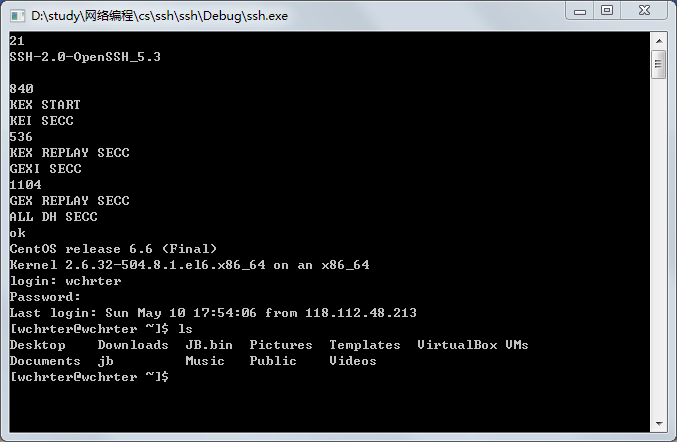
        在秘钥交换完成后第一次对发送数据加密时，首先需要对AES向量进行初始化，即设置对应的IV。aes部分我使用的是CRYPTOPP的aes-cbc算法（在后文的有对该算法的封装）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | en\_c\_to\_s.set\_iv(dhdata.comp\_encry\_key(IVCSF,32));      en\_c\_to\_s.set\_k(dhdata.comp\_encry\_key(ECSF,32));      en\_c\_to\_s.init();        de\_s\_to\_c.set\_iv(dhdata.comp\_encry\_key(IVSCF,32));      de\_s\_to\_c.set\_k(dhdata.comp\_encry\_key(ESCF,32));      de\_s\_to\_c.init(); |

    整个协议用到的主要加密算法的实现与封装：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154 | //mycrypt.h    #ifndef \_MYCRYPT\_H\_\_  #define \_MYCRYPT\_H\_\_    #include<cstring>  #include<string>  #include <iostream>  #include<cmath>  #include "integer.h"  #include "files.h"  #include "hex.h"  #include "sha.h"  #include "modes.h"  #include "osrng.h"    using namespace std;  using namespace CryptoPP;          Integer mkrandomnum(int len);  string inttostr(Integer num,unsigned int radix);  string inttostrnum(Integer num,unsigned int radix);  //大整数转mpint  string inttompint(Integer num,unsigned int radix);  string strtostrnum(string s);  //大整数快速幂运算  Integer fastpower\_comp(Integer a,Integer b,Integer c);    //sha算法封装  class m\_sha  {  public:      string encode\_sha1(string data);      string encode\_sha256(string data);  };  //dh算法实现  class m\_dh  {  public:      Integer dh\_g,dh\_p,dh\_x,dh\_e;      Integer dh\_y,dh\_f;      Integer dh\_k;        void set\_g\_and\_p(const Integer g,const Integer p)      {          dh\_g=g;          dh\_p=p;      }      void set\_y(Integer y)      {          dh\_y=y;      }      void set\_f(Integer f)      {          dh\_f=f;      }      void comp\_e();      Integer get\_e()      {          return dh\_e;      }      void comp\_k();      Integer get\_k()      {          return dh\_k;      }  };  //rsa算法实现  class m\_rsa  {  public:      Integer rsa\_e;      Integer rsa\_n;      void set\_e\_and\_n(Integer e,Integer n)      {          rsa\_e=e;          rsa\_n=n;      }      Integer comp\_rsa\_result(Integer num);  };  //dh gex协议算法实现  class m\_dh\_gex\_sha:public m\_dh,public m\_sha,public m\_rsa  {  public:      string v\_c,v\_s;      string i\_c,i\_s;      string k\_s;      string dh\_h,s\_h;        string shka\_name;        void set\_v\_c(string x)      {          v\_c=x;      }      void set\_v\_s(string x)      {          v\_s=x;      }      void set\_i\_c(string x)      {          i\_c=x;      }      void set\_i\_s(string x)      {          i\_s=x;      }      void set\_k\_s(string x)      {          k\_s=x;      }      void set\_s\_h(string x)      {          s\_h=x;      }      void set\_shka\_name(string x)      {          shka\_name=x;      }      void comp\_h();      string get\_h()      {          return dh\_h;      }      string comp\_encry\_key(char c,const int len);  };  //aes-cbc算法封装  class m\_aes\_cbc  {  public:      string aes\_k;      string aes\_iv;      void set\_k(string x)      {          aes\_k=x;      }      void set\_iv(string x)      {          aes\_iv=x;      }        CBC\_Mode<AES>::Encryption \*aes\_Encryptor;      CBC\_Mode<AES>::Decryption aes\_Decryptor;      void init();      string encode(string data);      string decode(string data);    };      #endif //\_MYCRYPT\_H\_\_ |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302 | //mycrypt.cpp    #include "mycrypt.h"    Integer mkrandomnum(int len)  {      Integer re=0;      for(int i=0;i<len;i++)      {          re\*=10;          re+=abs(rand())%10;      }      return re;  }    string inttostr(Integer num,unsigned int radix)  {      string s1="";      unsigned int k;      while(num>0)      {          k=num%radix;          num/=radix;          //cout<<num<<" ";          //printf("%d\n",k);          s1+=(char)k;      }      string s2="";      for(int i=s1.length()-1;i>=0;i--)      {          s2+=s1[i];      }      return s2;  }    string inttostrnum(Integer num,unsigned int radix)  {      string s1="";      unsigned int k;      while(num>0)      {          k=num%radix;          num/=radix;          if(k<10)          {              s1+='0'+k;          }          else          {              s1+='a'+k-10;          }      }      string s2="";      for(int i=s1.length()-1;i>=0;i--)      {          s2+=s1[i];      }      return s2;  }    string inttompint(Integer num,unsigned int radix)  {      string s="";      s+=(char)0;      s+=inttostr(num,radix);      int len=s.length();      string k="";      while(len>0)      {          k+=(char)(len%256);          len/=256;      }      while(k.length()<4)      {          k+=(char)0;      }      string re;      re+=k[3];      re+=k[2];      re+=k[1];      re+=k[0];      re+=s;        return re;  }    string strtostrnum(string s)  {      string re="";      int k,p;      for(int i=0;i<s.length();i++)      {          p=s[i];          if(p<0)          {              p=(256+s[i]);          }          k=p/16;          for(int j=0;j<2;j++)          {              //              if(k<10)              {                  re+=('0'+k);              }              else              {                  re+=('a'+k-10);              }              k=p%16;          }      }      return re;  }    int intlength(Integer num)  {      int re=0;      while(num>0)      {          num/=10;          re++;      }      return re;  }      Integer fastpower\_comp(Integer a,Integer b,Integer c)  {        /\*unused fast power      Integer re=1;      for(int i=0;i<b;i++)      {          re\*=a;          re%=c;      }      return re;      \*/        //fast power      Integer n=c;      c=1;      while(b!=0)      {          if(b%2!=0)          {              b=b-1;              c=(c\*a)%n;          }          else          {              b=b/2;              a=(a\*a)%n;          }      }      return c;  }    void m\_dh::comp\_e()  {        dh\_x=mkrandomnum(50)+1;      dh\_e=fastpower\_comp(dh\_g,dh\_x,dh\_p);        /\*      cout<<"//"<<endl;      cout<<dh\_g<<endl;      cout<<dh\_x<<endl;      cout<<dh\_p<<endl;      cout<<dh\_e<<endl;      cout<<"//"<<endl;      \*/  }  void m\_dh::comp\_k()  {      dh\_k=fastpower\_comp(dh\_f,dh\_x,dh\_p);  }    Integer m\_rsa::comp\_rsa\_result(Integer num)  {      return fastpower\_comp(num,rsa\_e,rsa\_n);  }      void m\_dh\_gex\_sha::comp\_h()  {      string data="";      /\*      data+=strtostrnum(v\_c);      data+=strtostrnum(v\_s);      data+=strtostrnum(i\_c);      data+=strtostrnum(i\_s);      data+=strtostrnum(k\_s);      data+=inttostrnum(dh\_e,16);      data+=inttostrnum(dh\_f,16);      data+=inttostrnum(dh\_k,16);      \*/      data+=v\_c;      data+=v\_s;      data+=i\_c;      data+=i\_s;      data+=k\_s;      data+=inttompint(dh\_e,16);      data+=inttompint(dh\_f,16);      data+=inttompint(dh\_k,16);          //cout<<endl<<"|"<<data<<"||"<<endl;        dh\_h=encode\_sha256(data);  }    string m\_dh\_gex\_sha::comp\_encry\_key(char c,const int len)  {      string re;        string data="";      data+=inttompint(dh\_k,16);      data+=dh\_h;      data+=c;      data+=dh\_h;      re=encode\_sha256(data);        while(re.length()<len)      {          data=inttompint(dh\_k,16);          data+=dh\_h;          data+=re;          re+=encode\_sha256(data);      }      while(re.length()>len)      {          re.pop\_back();      }      return re;  }      string m\_sha::encode\_sha1(string data)  {      string hash;      SHA1 sha1;      HashFilter hash\_filter (sha1);        hash\_filter.Attach(new HexEncoder(new StringSink(hash), false));      hash\_filter.Put((byte \*)data.c\_str(),data.length());      hash\_filter.MessageEnd();        return hash;  }    string m\_sha::encode\_sha256(string data)  {      string hash;      SHA256 sha256;      HashFilter hash\_filter (sha256);        hash\_filter.Attach(new HexEncoder(new StringSink(hash), false));      hash\_filter.Put((byte \*)data.c\_str(),data.length());      hash\_filter.MessageEnd();        return hash;  }    void m\_aes\_cbc::init()  {        aes\_Encryptor=new CBC\_Mode<AES>::Encryption((unsigned char \*)aes\_k.c\_str(),          aes\_k.length(),          (unsigned char \*)aes\_iv.c\_str());        aes\_Decryptor=new CBC\_Mode<AES>::Decryption ((unsigned char \*)aes\_k.c\_str(),          aes\_k.length(),          (unsigned char \*)aes\_iv.c\_str());  }    string m\_aes\_cbc::encode(string data)  {      string re;      StringSource(data,          true,          new StreamTransformationFilter(\*aes\_Encryptor,          new StringSink(re),          BlockPaddingSchemeDef::BlockPaddingScheme::ONE\_AND\_ZEROS\_PADDING,          true)          );      return re;  }    string m\_aes\_cbc::decode(string data)  {      string re;      StringSource(data,          true,          new StreamTransformationFilter(\*aes\_Decryptor,          new StringSink(re),          BlockPaddingSchemeDef::BlockPaddingScheme::ONE\_AND\_ZEROS\_PADDING,          true)          );      return re;  } |

    ssh的实现就到此终于结束了，截图留念。

[](http://s3.51cto.com/wyfs02/M00/6C/C1/wKiom1VRdHDT8SmJAAGJwMhapbw338.jpg)

    笔者在之初就想使用crypto++来帮助实现ssh过程的密码算法。而刚接触这东西完全搞不懂怎么用，什么编码器、生成器、过滤器、sink...这些概念根本就不懂，网上的使用文档直接就拿这一堆概念加上一堆组合出来的代码来实现一个加密算，没有什么密码学知识，想要快速掌握crypto++几乎是不可能的，当时研究了很久就只是会使用它的hash加密。而后自己硬着头皮实现了整个dh-gex，到后面aes后，发现自己能很自然得理解crypto++的用法了，便自己封装了crypto++的aes算法供使用。

    总之都是好事，以后遇到其他的基于ssl的协议与应用就应能很轻松地理解与实现了。

http://www.cnblogs.com/wchrt/p/4550208.html