

《第十五章 建立安全信道》示例代码

作者：韩露露、杨波

日期：2019年3月1日

说明

本电子文档来源于书籍《深入浅出CryptoPP密码学库》，它最初被存放于GitHub上。任何人都可以复制、传播、使用本示例代码。



简介

《深入浅出CryptoPP密码学库》内容简介：

本书向读者介绍密码学库CryptoPP（或Crypto++）的使用方法和设计原理。CryptoPP是一个用C++语言编写的、开源的、免费的密码程序库，它最初由Wei Dai开发，现由开源社区维护。CryptoPP库广泛应用于学术界、开源项目、非商业项目以及商业项目，它几乎包括了目前已经公开的所有密码算法，支持当前主流的多种系统平台，并且具有良好的设计结构和较高的执行效率。

全书共15章，主要内容包括随机数发生器、Hash函数、流密码、分组密码、消息认证码、密钥派生和基于口令的密码、公钥加密系统、数字签名、密钥协商等，本书涵盖C++程序设计、设计模式、数论和密码学等知识。

本书最大的特点就是以应用为导向、以解决实际工程问题为目标，理论结合实践，将抽象的密码学变成保障信息安全的实际工具。

本书可以作为密码学、网络安全等专业在校学生的上机实验教材，也可以作为信息安全产品开发、科研人员、密码算法实现者的参考手册。



资源

本书更多示例代码：<https://github.com/locomotive-crypto>

Crypto++网站：<https://www.cryptopp.com/>

Crypto++库GitHub地址：<https://github.com/weidai11/cryptopp>

Crypto++库SourceForge地址：<https://sourceforge.net/projects/cryptopp/>

Crypto++库Google论坛：

⇒公告通知地址：<https://groups.google.com/forum/#!forum/cryptopp-announce>

⇒用户群组地址：<https://groups.google.com/forum/#!forum/cryptopp-users>

目录

1	生成数字签名公私钥	1
2	服务端示例代码	2
3	客户端示例代码	8
4	程序执行结果	14
4.1	服务端程序运行结果	14
4.2	客户端程序运行结果	15
5	声明	16

1 生成数字签名公私钥

通信双方（客户端和服务端）分别使用下面的程序产生己方的数字签名公私钥对，并将它们存储于文件中。在示例程序中，公钥存储于文件sin_pubkey.der，私钥存储于文件sin_prikey.der。为了进一步区分与使用客户端和服务端的数字签名公私钥文件，将它们分别命名如下：

→ 客户端：公钥（sin_pubkey_client.der）、私钥（sin_prikey_client.der）

→ 服务端：公钥（sin_pubkey_server.der）、私钥（sin_prikey_server.der）

之后，它们均保留己方私钥，并将公钥公开，使得对方可以获得己方的公钥。在双方进行安全通信之前，它们需要获得的密钥信息如下：

→ 客户端：公钥（sin_pubkey_server.der）、私钥（sin_prikey_client.der）

→ 服务端：公钥（sin_pubkey_client.der）、私钥（sin_prikey_server.der）

```
1 #include<iostream> //使用cout、cin
2 #include<osrng.h> //使用AutoSeededRandomPool
3 #include<files.h> //使用FileSink
4 #include<string> //使用string
5 #include<tiger.h> //使用Tiger
6 #include<eccrypto.h> //使用ECDSA
7 #include<oids.h> //使用secp160r1()
8 using namespace std; //std是C++的命名空间
9 using namespace CryptoPP; //CryptoPP是库的命名空间
10 int main()
11 {
12     AutoSeededRandomPool rng; //定义一个随机数发生器对象
13     ECDSA<ECP, Tiger>::PrivateKey prikey; //定义ECDSA算法私钥对象
14     // 使用标准的椭圆曲线参数初始化私钥对象
15     prikey.Initialize(rng, ASN1::secp160r1());
16     bool bResult = prikey.Validate(rng, 3); //验证私钥的安全级别
17     if (bResult)
18         cout << "私钥符合要求的安全强度" << endl;
19     else
20         cout << "私钥不满足要求的安全强度" << endl;
21     ECDSA<ECP, Tiger>::PublicKey pubkey; //定义公钥对象
22     prikey.MakePublicKey(pubkey); //根据签名私钥产生公钥
23     //保存私钥至文件sin_prikey.der
24     prikey.Save(FileSink("sin_prikey.der").Ref());
25     //保存公钥至文件sin_pubkey.der
26     pubkey.Save(FileSink("sin_pubkey.der").Ref());
27     cout << "产生公私钥对成功..." << endl;
28     return 0;
29 }
```

本示例主要在于演示密码学原语在具体场景中的作用和使用方法。在下面的示例中，我们认为客户端和服务端均能够获得对方产生的数字签名公钥文件。关于密钥的分配机制，读者可以参考密钥管理和PKI等相关技术。

2 服务端示例代码

下面是本章节安全文件传输示例程序的服务端完整代码。

```
1 #include<iostream> //使用cout、cin
2 #include<boost/asio.hpp> //使用asio库网络相关的类
3 #include<fstream> //使用ifstream和ofstream
4 #include<secblock.h> //使用SecByteBlock
5 #include<algorithm> //使用min()函数
6 #include<files.h> //使用FileSink
7 #include<string> //使用string
8 #include<osrng.h> //使用AutoSeededRandomPool
9 #include<files.h> //使用FileSink
10 #include<tiger.h> //使用Tiger
11 #include<sm3.h> //使用SM3
12 #include<eccrypto.h> //使用ECDSA
13 #include<integer.h> //使用Integer
14 #include<dh.h> //使用DH
15 #include<hmac.h> //使用HMAC
16 #include<modes.h> //使用CBC_Mode
17 #include<sm4.h> //使用SM4
18 #include<hex.h> //使用HexEncoder
19 #include<pwdbased.h> //使用PKCS5_PBKDF2_HMAC
20 using namespace std; //C++标准命名空间
21 using namespace CryptoPP; //CryptoPP库命名空间
22 using namespace boost; //boost库命名空间
23 using namespace boost::asio; //boost库的asio库命名空间
24 //功能：将文件filename中size字节长度的内容发送至客户端socket对象
25 //参数filename：要发送的文件名字
26 //参数size：要发送的内容长度
27 //参数sock：客户端socket对象的引用
28 size_t send_file_by_fixed_length(const char* filename, size_t size,
    asio::ip::tcp::socket& sock);
29 //功能：将文件filename中的内容发送至客户端socket
30 //参数filename：要发送的文件名字
31 //参数sock：客户端socket对象的引用
32 //返回值：bool类型。若发送成功，则返回true；否则，返回false。
33 bool send_all_file(const char* filename, asio::ip::tcp::socket& sock
    );
34 //功能：以十六进制的形式将content地址开始的size字节长度数据打印至标准输出设备
35 //参数str：十六进制形式输出数据的前缀信息
36 //参数content：待输出缓冲区的首地址
37 //参数size：缓冲区content的长度
38 void printinfo(const string& str, const byte* content, size_t size);
39 int main()
40 {
```

```

41     try
42     {
43         io_service io;
44         string filename; //存储发送至客户端的文件名字
45         unsigned short port; //存储绑定的端口号
46         cout << "输入要发送的文件名字:";
47         getline(cin, filename); //输入文件名字
48         cout << "输入服务端绑定的端口号:";
49         cin >> port; //输入绑定的端口号
50         //定义端口号对象
51         ip::tcp::acceptor acceptor(io, ip::tcp::endpoint(ip::tcp::v4
           (), port));
52         ip::tcp::socket sock(io); //定义socket对象
53         acceptor.accept(sock); //接受客户端的连接
54         //初始化ECDSA算法-数字签名算法
55         AutoSeededRandomPool rng; //定义一个随机数发生器对象
56         //定义ECDSA算法私钥对象
57         ECDSA<ECP, Tiger>::PrivateKey Sig_server;
58         //加载服务端签名私钥
59         Sig_server.Load(FileSource("sin_prikey_server.der", true).
           Ref());
60         //定义ECDSA算法公钥对象
61         ECDSA<ECP, Tiger>::PublicKey Ver_client;
62         //加载客户端签名公钥
63         Ver_client.Load(FileSource("sin_pubkey_client.der", true).
           Ref());
64         //构造签名器对象, 服务端对DH算法产生的公钥进行签名
65         ECDSA<ECP, Tiger>::Signer sig_server(Sig_server);
66         //构造验证器对象, 服务端对客户端发来的公钥的合法性进行验证
67         ECDSA<ECP, Tiger>::Verifier ver_client(Ver_client);
68         cout << "服务端成功加载己方数字签名私钥和对方数字签名公钥..."<< endl;
69         //初始化DH算法-密钥协商算法
70         Integer p("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C6"
71                 "9A6A9DCA52D23B616073E28675A23D189838EF1E2EE652C0"
72                 "13ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD70"
73                 "98488E9C219A73724EFFD6FAE5644738FAA31A4FF55BCCC0"
74                 "A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708"
75                 "DF1FB2BC2E4A4371h");
76         Integer g("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507F"
77                 "D6406CFF14266D31266FEA1E5C41564B777E690F5504F213"
78                 "160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1"
79                 "909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A28A"
80                 "D662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C2A24"
81                 "855E6EEB22B3B2E5h");
82         Integer q("F518AA8781A8DF278ABA4E7D64B7CB9D49462353h");

```

```

83     DH dh(p, q, g); //定义一个DH算法对象并用标准的参数初始化它
84     cout << "服务端DH算法参数初始化完毕..." << endl;
85     //服务端密钥协商算法产生公私钥对
86     //存储服务端私钥
87     SecByteBlock dh_pri_ser(dh.PrivateKeyLength());
88     SecByteBlock dh_pub_ser(dh.PublicKeyLength()); //存储服务端公钥
89     dh.GenerateKeyPair(rng, dh_pri_ser, dh_pub_ser); //产生公私钥对
90     //对服务端密钥协商产生的公钥签名, 并将签名和公钥发送至服务端
91     //公钥和签名总长
92     size_t msg_sig_len = dh_pub_ser.size() + sig_server.
        SignatureLength();
93     //存储服务端的DH公钥和签名
94     SecByteBlock msg_sig_str_send(msg_sig_len);
95     //对己方DH公钥签名
96     ArraySource Sig_dh_pubkey_Src(dh_pub_ser, dh_pub_ser.size(),
        true,
97         new SignerFilter(rng, sig_server,
98             new ArraySink(msg_sig_str_send, msg_sig_len), true));
99     //发送己方DH公钥和签
100    write(sock, asio::buffer(msg_sig_str_send, msg_sig_len));
101    //接收客户端发来的DH公钥及其签名, 并验证其合法性
102    //存储客户端产生的DH算法公钥
103    SecByteBlock dh_pub_cli(dh.PublicKeyLength());
104    //存储客户端发来的DH公钥和签名
105    SecByteBlock tag_dhpub_cli(msg_sig_len);
106    //接收公钥和签名
107    read(sock, asio::buffer(tag_dhpub_cli, msg_sig_len));
108    //验证客户端的DH公钥和签名
109    ArraySource ArrSrc(tag_dhpub_cli, msg_sig_len, true,
110        new SignatureVerificationFilter(ver_client,
111            new ArraySink(dh_pub_cli, dh_pub_cli.size()),
112            SignatureVerificationFilter::SIGNATURE_AT_END |
113            SignatureVerificationFilter::PUT_MESSAGE |
114            SignatureVerificationFilter::THROW_EXCEPTION)
115        );
116    //打印客户端公钥
117    printinfo("dh_pub_cli", dh_pub_cli, dh_pub_cli.size());
118    //打印服务端公钥
119    printinfo("dh_pub_ser", dh_pub_ser, dh_pub_ser.size());
120    cout << "验证客户端DH算法公钥成功..." << endl;
121    //存储双方协商的共享值
122    SecByteBlock agreedkey(dh.AgreedValueLength());
123    //根据己方DH的私钥和对方的DH公钥产生一个共享的值
124    bool bResult = dh.Agree(agreedkey, dh_pri_ser, dh_pub_cli);
125    if (bResult)

```

```

126         cout << "客户端和服务端协商密钥成功..." << endl;
127     else
128     {
129         throw Exception(Exception::INVALID_ARGUMENT,
130             "客户端和服务端协商密钥失败"); //抛出异常
131     }
132     //打印协商的共享值
133     printf("agreedkey", agreedkey, agreedkey.size());
134     PKCS5_PBKDF2_HMAC<SM3> pbkdf; //定义密钥派生函数
135     //以CBC模式运行分组密码SM4
136     //定义SM4算法加密器对象
137     CBC_Mode<SM4>::Encryption cbc_sm4_enc;
138     //利用密钥派生函数产生分组密码的key、iv、HMAC密钥
139     //KDF算法需要派生的共享信息长度
140     size_t Derive_len = cbc_sm4_enc.DefaultKeyLength() +
141         cbc_sm4_enc.DefaultIVLength()
142         + HMAC_Base::DEFAULT_KEYLENGTH;
143     //存储KDF派生的共享值
144     SecByteBlock DerivedSecretInfo(Derive_len);
145     //派生共享信息
146     pbkdf.DeriveKey(DerivedSecretInfo, Derive_len, agreedkey,
147         agreedkey.size());
148     //打印派生信息
149     printf("DerivedSecretInfo", DerivedSecretInfo,
150         DerivedSecretInfo.size());
151     //存储SM4算法的密钥
152     SecByteBlock sm4_key(cbc_sm4_enc.DefaultKeyLength());
153     //存储SM4算法的初始向量
154     SecByteBlock sm4_iv(cbc_sm4_enc.DefaultIVLength());
155     //依次从派生的信息中截取SM4算法的密钥和初始向量
156     memcpy_s(sm4_key, sm4_key.size(), DerivedSecretInfo, sm4_key
157         .size());
158     memcpy_s(sm4_iv, sm4_iv.size(), DerivedSecretInfo + sm4_key
159         .size(), sm4_iv.size());
160     cbc_sm4_enc.SetKeyWithIV(sm4_key, sm4_key.size(), sm4_iv,
161         sm4_iv.size());
162     //打印SM4算法的密钥
163     printf("sm4_key", sm4_key, sm4_key.size());
164     //打印SM4算法的初始向量
165     printf("sm4_iv", sm4_iv, sm4_iv.size());
166     //加密文件filename
167     FileSource cbc_sm4_encSrc(filename.c_str(), true,
168         new StreamTransformationFilter(cbc_sm4_enc,
169         new FileSink("cbc_sm4_cipher_tmp")));
170     cout << "已完成文件的加密..." << endl;

```



```

167 //存储HMAC'密钥
168 SecByteBlock hmac_key(HMAC_Base::DEFAULT_KEYLENGTH);
169 //从派生的信息中截取HMAC'密钥
170 memcpy_s(hmac_key, hmac_key.size(), DerivedSecretInfo +
171         sm4_key.size() + sm4_iv.size(), HMAC_Base::
            DEFAULT_KEYLENGTH);
172 //打印HMAC算法的密钥
173 printinfo("hmac_key", hmac_key, hmac_key.size());
174 //定义消息认证码对象
175 HMAC<SM3> hmac(hmac_key, hmac_key.size());
176 //认证加密后的文件
177 FileSource cbc_sm4_cipherSrc("cbc_sm4_cipher_tmp", true,
178         new HashFilter(hmac,
179         new FileSink("cbc_sm4_cipher_hamc_tmp"), true));
180 cout << "已完成对加密文件的认证..." << endl;
181 //将加密和认证后的文件发送至客户端
182 send_all_file("cbc_sm4_cipher_hamc_tmp", sock);
183 cout << "已经将加密和认证后的文件发送至客户端..." << endl;
184 while (true); //让程序继续运行, 以便在控制台观察输出结果
185 }
186 catch (const boost::system::system_error& e)
187 { //出现异常
188     cout << e.what() << endl; //异常原因
189 }
190 catch (const Exception& e)
191 { //出现异常
192     cout << e.what() << endl; //异常原因
193 }
194 return 0;
195 }
196 bool send_all_file(const char* filename, asio::ip::tcp::socket& sock
197 )
198 {
199     ifstream in(filename); //打开文件
200     in.seekg(0, ios_base::end); //移动文件指针至文件末尾
201     streamoff length = in.tellg(); //获取文件的长度
202     in.close(); //关闭文件
203     //通知客户端将要发送的文件长度
204     write(sock, asio::buffer(&length, sizeof(streamoff)));
205     //发送整个文件
206     if (length == send_file_by_fixed_length(filename, length, sock))
207         return true; //读取文件中的所有数据成功
208     else
209         return false; //读取文件中的所有数据失败
210 }

```



```

210 size_t send_file_by_fixed_length(const char* filename, size_t size,
    asio::ip::tcp::socket& sock)
211 {
212     static const unsigned int BUFFER_SIZE = 1024; //每次读取的最大字节数
213     unsigned int total = 0; //总共读取的字节数
214     SecBlock<char> buff(min(size, BUFFER_SIZE)); //分配存储空间
215     ifstream in(filename, ios_base::in | ios_base::binary); //打开文件
216     while (size && in.good()) //判断数据是否读取完毕以及文件流当前的状态
217     {
218         in.read(buff, min(size, BUFFER_SIZE)); //从文件读取数据
219         streamsize l = in.gcount(); //本次读取的数据长度
220         asio::write(sock, asio::buffer(buff, l)); //将读取的数据发送至客户端
221         size -= l; //计算剩余需要读取的字节数
222         total += l; //统计读取的字节数
223     }
224     if (!in.good() && !in.eof())
225         throw Exception(Exception::IO_ERROR, "读取文件发生意外");
226     return total;
227 }
228 void printinfo(const string& str, const byte* content, size_t size)
229 {
230     cout << str << ": "; //输出前缀
231     ArraySource hmac_keySrc(content, size, true,
232         new HexEncoder(
233             new FileSink(cout))); //以十六进制形式输出content的内容
234     cout << endl; //换行
235 }

```

执行程序，在服务端输入如下信息，等待客户端的连接。

```

输入要发送的文件名字:: VS2012.5.iso
输入服务端绑定的端口号: 6666
...

```

3 客户端示例代码

下面是本章节安全文件传输示例程序的客户端完整代码。

```
1 #include<iostream> //使用cout、cin
2 #include<boost/asio.hpp> //使用asio库网络相关的类
3 #include<fstream> //使用ifstream和ofstream
4 #include<secblock.h> //使用SecByteBlock
5 #include<algorithm> //使用min()函数
6 #include<files.h> //使用FileSink
7 #include<string> //使用string
8 #include<osrng.h> //使用AutoSeededRandomPool
9 #include<files.h> //使用FileSink
10 #include<tiger.h> //使用Tiger
11 #include<sm3.h> //使用SM3
12 #include<eccrypto.h> //使用ECDSA
13 #include<integer.h> //使用Integer
14 #include<dh.h> //使用DH
15 #include<hmac.h> //使用HMAC
16 #include<modes.h> //使用CBC_Mode
17 #include<sm4.h> //使用SM4
18 #include<hex.h> //使用HexEncoder
19 #include<pwdbased.h> //使用PKCS5_PBKDF2_HMAC
20 using namespace std; //C++标准命名空间
21 using namespace CryptoPP; //CryptoPP库命名空间
22 using namespace boost; //boost库命名空间
23 using namespace boost::asio; //boost库的asio库命名空间
24 //功能：从服务端socket上读取size长度字节的内容并存储于文件filename
25 //参数filename：从服务端接收的数据存储到的文件的名字
26 //参数size：从服务端接收的数据长度（字节）
27 //参数sock：服务端socket对象的引用
28 //返回值：bool类型。若成功接收size长度的数据，则返回true；否则，返回false。
29 bool receive_file(const char* filename, size_t size, asio::ip::tcp::
    socket& sock);
30 //功能：以十六进制的形式将content地址开始的size字节长度数据打印至标准输出设备
31 //参数str：十六进制形式输出数据的前缀信息
32 //参数content：待输出缓冲区的首地址
33 //参数size：缓冲区content的长度
34 void printinfo(const string& str, const byte* content, size_t size);
35 int main()
36 {
37     try
38     {
39         io_service io;
40         ip::tcp::socket sock(io); //定义socket对象
41         string raw_ip; //存储IP地址
```

```

42     unsigned short port; //存储端口号
43     cout << "输入要连接的服务端IP地址: ";
44     getline(cin, raw_ip); //输入连接的服务端IP地址
45     cout << "输入要连接的服务端绑定的端口号: ";
46     cin >> port; //输出要连接的服务端端口号
47     //定义端口号对象
48     ip::tcp::endpoint ep(ip::address::from_string(raw_ip.c_str())
49         ), port);
49     sock.connect(ep); //连接指定IP地址和端口号的服务端
50     //初始化ECDSA算法——数字签名算法
51     AutoSeededRandomPool rng; //定义一个随机数发生器对象
52     //定义ECDSA算法私钥对象
53     ECDSA<ECP, Tiger>::PrivateKey Sig_client;
54     //加载客户端签名私钥
55     Sig_client.Load(FileSource("sin_prikey_client.der", true).
56         Ref());
57     //定义ECDSA算法公钥对象
58     ECDSA<ECP, Tiger>::PublicKey Ver_server;
59     //加载服务端签名公钥
60     Ver_server.Load(FileSource("sin_pubkey_server.der", true).
61         Ref());
62     //构造签名器对象，客户端对DH算法产生的公钥进行签名
63     ECDSA<ECP, Tiger>::Signer sig_client(Sig_client);
64     //构造验证器对象，客户端对服务端发来的公钥的合法性进行验证
65     ECDSA<ECP, Tiger>::Verifier ver_server(Ver_server);
66     cout << "客户端成功加载己方数字签名私钥和对方数字签名公钥..." << endl;
67     //DH密钥协商算法
68     Integer p("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C6"
69         "9A6A9DCA52D23B616073E28675A23D189838EF1E2EE652C0"
70         "13ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD70"
71         "98488E9C219A73724EFFD6FAE5644738FAA31A4FF55BCCC0"
72         "A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708"
73         "DF1FB2BC2E4A4371h");
74     Integer g("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507F"
75         "D6406CFF14266D31266FEA1E5C41564B777E690F5504F213"
76         "160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1"
77         "909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A28A"
78         "D662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C2A24"
79         "855E6EEB22B3B2E5h");
80     Integer q("F518AA8781A8DF278ABA4E7D64B7CB9D49462353h");
81     DH dh(p, q, g); //定义一个DH算法对象并用标准的参数初始化它
82     cout << "客户端DH算法参数初始化完毕..." << endl;
83     //客户端密钥协商算法产生公私钥对
84     //存储客户端产生私钥
85     SecByteBlock dh_pri_cli(dh.PrivateKeyLength());

```

```

84 //存储客户端公钥
85 SecByteBlock dh_pub_cli(dh.PublicKeyLength());
86 dh.GenerateKeyPair(rng, dh_pri_cli, dh_pub_cli); //产生公私钥对
87 //对己方密钥协商产生的公钥签名, 并将签名和公钥发送至服务端
88 //公钥和签名总长
89 size_t msg_sig_len = dh_pub_cli.size() + sig_client.
    SignatureLength();
90 //存储客户端的DH公钥和签名
91 SecByteBlock msg_sig_str_send(msg_sig_len);
92 //对己方DH公钥签名
93 ArraySource Sig_dh_pubkey_Src(dh_pub_cli, dh_pub_cli.size(),
    true,
94     new SignerFilter(rng, sig_client,
95         new ArraySink(msg_sig_str_send, msg_sig_len), true));
96 //发送己方DH公钥和签名
97 write(sock, asio::buffer(msg_sig_str_send, msg_sig_len));
98 //接收服务端发来的DH公钥及其签名, 并验证其合法性
99 //存储服务端发来的DH算法公钥
100 SecByteBlock dh_pub_ser(dh.PublicKeyLength());
101 //存储服务端发来的DH公钥和签名
102 SecByteBlock tag_dhpub_ser(msg_sig_len);
103 //从网络上接收服务端DH公钥和签名
104 read(sock, asio::buffer(tag_dhpub_ser, msg_sig_len));
105 //验证服务端的DH公钥和签名
106 ArraySource ArrSrc(tag_dhpub_ser, msg_sig_len, true,
107     new SignatureVerificationFilter(ver_server,
108     new ArraySink(dh_pub_ser, dh_pub_ser.size()),
109     SignatureVerificationFilter::SIGNATURE_AT_END |
110     SignatureVerificationFilter::PUT_MESSAGE |
111     SignatureVerificationFilter::THROW_EXCEPTION));
112 //打印客户端公钥
113 printf("dh_pub_cli", dh_pub_cli, dh_pub_cli.size());
114 //打印服务端公钥
115 printf("dh_pub_ser", dh_pub_ser, dh_pub_ser.size());
116 cout << "验证服务端DH算法公钥成功..." << endl;
117 //存储双方协商的共享值
118 SecByteBlock agreedkey(dh.AgreedValueLength());
119 //根据己方的DH私钥和对方的DH公钥产生一个共享的值
120 bool bResult = dh.Agree(agreedkey, dh_pri_cli, dh_pub_ser);
121 if (bResult)
122     cout << "客户端和服务端协商密钥成功..." << endl;
123 else
124 {
125     throw Exception(Exception::INVALID_ARGUMENT,
126         "客户端和服务端协商密钥失败"); //抛出异常

```

```

127     }
128     //打印协商的共享值
129     printinfo("agreedkey", agreedkey, agreedkey.size());
130     streamoff length; //存储从服务端将要读取的数据长度
131     //从服务端读取数据
132     read(sock, asio::buffer(&length, sizeof(streamoff)));
133     if (receive_file("cbc_sm4_cipher_hmac_tmp", length, sock))
134         cout << "从服务端成功读取指定长度的文件..." << endl;
135     else
136         throw Exception(Exception::IO_ERROR,
137             "从服务端读取指定长度文件失败...");
138     cout << "读取的文件（已经被加密和认证）长度为" << length
139         << "（字节）" << endl;
140     PKCS5_PBKDF2_HMAC<SM3> pbkdf; //定义密钥派生函数对象
141     //以CBC模式运行分组密码SM4
142     CBC_Mode<SM4>::Decryption cbc_sm4_dec; //定义SM4算法解密器对象
143     //利用密钥派生函数产生分组密码的key、iv、HMAC密钥
144     //KDF算法需要派生的共享信息长度
145     size_t Derive_len = cbc_sm4_dec.DefaultKeyLength() +
146         cbc_sm4_dec.DefaultIVLength()
147         + HMAC_Base::DEFAULT_KEYLENGTH;
148     //存储KDF派生的共享值
149     SecByteBlock DerivedSecretInfo(Derive_len);
150     //派生共享信息
151     pbkdf.DeriveKey(DerivedSecretInfo, Derive_len, agreedkey,
152         agreedkey.size());
153     //打印派生信息
154     printinfo("DerivedSecretInfo", DerivedSecretInfo,
155         DerivedSecretInfo.size());
156     //存储SM4算法的密钥
157     SecByteBlock sm4_key(cbc_sm4_dec.DefaultKeyLength());
158     //存储SM4算法的初始向量
159     SecByteBlock sm4_iv(cbc_sm4_dec.DefaultIVLength());
160     //依次从派生的信息中截取SM4算法的密钥和初始向量
161     memcpy_s(sm4_key, sm4_key.size(), DerivedSecretInfo, sm4_key
162         .size());
163     memcpy_s(sm4_iv, sm4_iv.size(), DerivedSecretInfo + sm4_key
164         .size(), sm4_iv.size());
165     cbc_sm4_dec.SetKeyWithIV(sm4_key, sm4_key.size(), sm4_iv,
166         sm4_iv.size());
167     //打印SM4算法的密钥
168     printinfo("sm4_key", sm4_key, sm4_key.size());
169     //打印SM4算法的初始向量
170     printinfo("sm4_iv", sm4_iv, sm4_iv.size());
171     //存储HMAC密钥

```

```

166     SecByteBlock hmac_key(HMAC_Base::DEFAULT_KEYLENGTH);
167     //从派生的信息中截取HMAC密钥
168     memcpy_s(hmac_key, hmac_key.size(), DerivedSecretInfo +
169             sm4_key.size() + sm4_iv.size(),
170             HMAC_Base::DEFAULT_KEYLENGTH);
171     //打印HMAC算法的密钥
172     printinfo("hmac_key", hmac_key, hmac_key.size());
173     //定义HMAC验证算法对象
174     HMAC<SM3> hmac(hmac_key, hmac_key.size());
175     //验证文件
176     FileSource cipher_unhmac_Src("cbc_sm4_cipher_hmac_tmp", true,
177     new HashVerificationFilter(hmac,
178     new FileSink("cbc_sm4_cipher_tmp"),
179     HashVerificationFilter::PUT_MESSAGE |
180     HashVerificationFilter::HASH_AT_END |
181     HashVerificationFilter::THROW_EXCEPTION));
182     cout << "文件完整性校验完毕..." << endl;
183     //解密文件
184     FileSource cbc_sm4_decSrc("cbc_sm4_cipher_tmp", true,
185     new StreamTransformationFilter(cbc_sm4_dec,
186     new FileSink("receive.txt")));
187     cout << "文件解密完毕..." << endl;
188     while (true); //让程序继续运行，以便在控制台观察输出结果
189 }
190 catch (const boost::system::system_error& e)
191 { //出现异常
192     cout << e.what() << endl; //异常原因
193 }
194 catch (const Exception& e)
195 { //出现异常
196     cout << e.what() << endl; //异常原因
197 }
198 return 0;
199 }
200 void printinfo(const string& str, const byte* content, size_t size)
201 {
202     cout << str << ": "; //输出前缀
203     ArraySource hmac_keySrc(content, size, true,
204     new HexEncoder(
205     new FileSink(cout))); //以十六进制形式输出content的内容
206     cout << endl; //换行
207 }
208 bool receive_file(const char* filename, size_t size, asio::ip::tcp::
socket& sock)
{

```



```
209 static const unsigned int BUFFER_SIZE = 1024; //每次读取的最大字节数
210 SecBlock<char> buff(min(size, BUFFER_SIZE)); //分配存储空间
211 //打开文件
212 ofstream out(filename, ios_base::out | ios_base::binary);
213 while (size && out.good()) //判断数据是否读取完毕以及文件流当前的状态
214 {
215     size_t len = min(size, BUFFER_SIZE); //计算本次读取的数据长度
216     read(sock, asio::buffer(buff, len)); //从服务端读取数据
217     out.write(buff, len); //存储至文件
218     size -= len; //计算剩余需要读取的字节数
219 }
220 if (0 == size)
221     return true; //成功读取指定长度的数据
222 else
223     return false; //无法读取指定长度的数据
224 }
```

执行程序，在客户端输入如下信息。

```
输入要连接的服务端IP: 127.0.0.1
输入要连接的服务端绑定的端口号: 6666
...
```


4 程序执行结果

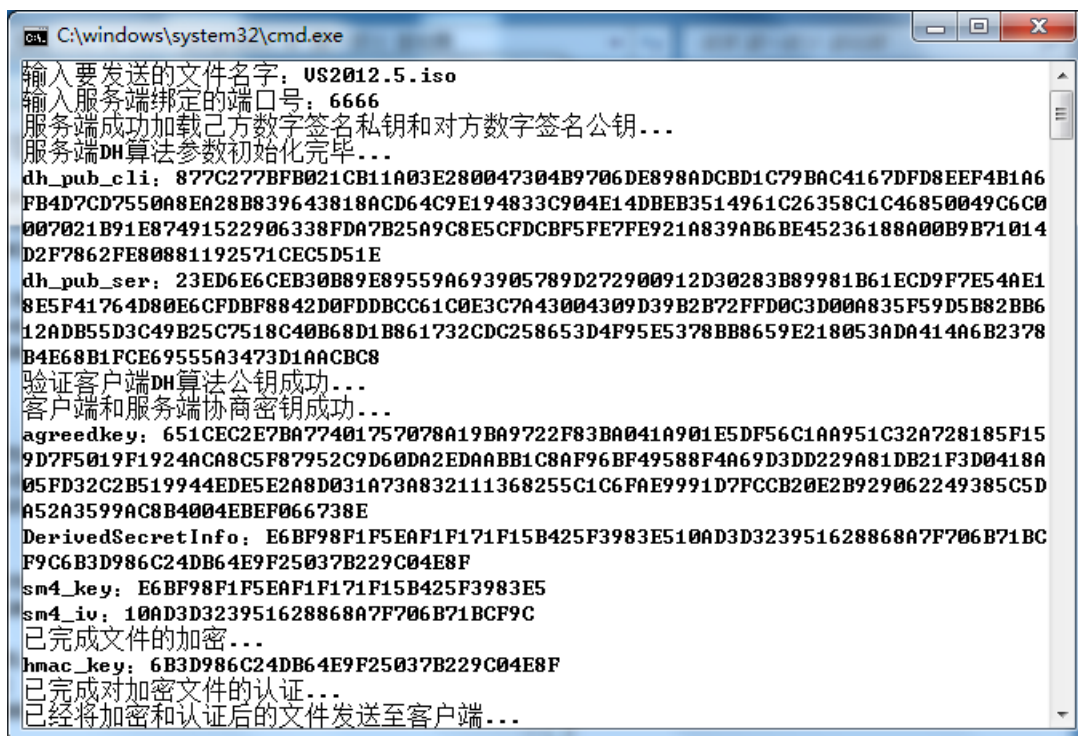
服务端和客户端先后启动，并分别输入上述信息（或其他正确的IP地址和端口号），双方就可以执行文件的安全传输。

4.1 服务端程序运行结果

服务端选取的待发送文件是VS2012.5.iso，它是一个大约为2.36GB的应用程序安装包文件。当服务端程序执行完毕后，可以在它所在的目录下发现另外两个文件，如图1所示。其中，文件cbc_sm4_cipher_tmp是文件VS2012.5.iso被SM4算法加密后的密文，而文件cbc_sm4_cipher_hamc_tmp是文件cbc_sm4_cipher_tmp被HMAC算法认证后的文件，即它由SM4算法加密后的密文和HMAC算法对这个密文计算的消息认证码组成。

名称	修改日期	类型	大小
Release	2018/12/3 22:10	文件夹	
cbc_sm4_cipher_hamc_tmp	2018/12/4 8:33	文件	2,479,267 KB
cbc_sm4_cipher_tmp	2018/12/4 8:33	文件	2,479,267 KB
secure_channel_server.cpp	2018/12/3 22:08	C++ Source	10 KB
secure_channel_server.vcxproj	2018/12/1 10:20	VC++ Project	8 KB
secure_channel_server.vcxpr...	2018/12/1 10:20	VC++ Project Fi...	1 KB
sin_prikey_server.der	2018/11/22 23:23	DER 文件	1 KB
sin_pubkey_client.der	2018/11/22 23:22	DER 文件	1 KB
VS2012.5.iso	2018/5/8 10:42	光盘映像文件	2,479,266 KB

图1 双方完成文件传输后服务端所在目录下含有的文件



```
C:\windows\system32\cmd.exe
输入要发送的文件名字: VS2012.5.iso
输入服务端绑定的端口号: 6666
服务端成功加载己方数字签名私钥和对方数字签名公钥...
服务端DH算法参数初始化完毕...
dh_pub_cli: 877C277BFB021CB11A03E280047304B9706DE898ADCBD1C79BAC4167DFD8EEF4B1A6
FB4D7CD7550A8EA28B839643818ACD64C9E194833C904E14DBEB3514961C26358C1C46850049C6C0
007021B91E87491522906338FDA7B25A9C8E5CFDCBF5FE7FE921A839AB6BE45236188A00B9B71014
D2F7862FE80881192571CEC5D51E
dh_pub_ser: 23ED6E6CEB30B89E89559A693905789D272900912D30283B89981B61ECD9F7E54AE1
8E5F41764D80E6CFDBF8842D0FDDBC61C0E3C7A43004309D39B2B72FFD0C3D00A835F59D5B82BB6
12ADB55D3C49B25C7518C40B68D1B861732CDC258653D4F95E5378BB8659E218053ADA414A6B2378
B4E68B1FCE69555A3473D1AACBC8
验证客户端DH算法公钥成功...
客户端和服务端协商密钥成功...
agreedkey: 651CEC2E7BA77401757078A19BA9722F83BA041A901E5DF56C1AA951C32A728185F15
9D7F5019F1924ACA8C5F87952C9D60DA2EDAABB1C8AF96BF49588F4A69D3DD229A81DB21F3D0418A
05FD32C2B519944EDE5E2A8D031A73A832111368255C1C6FAE9991D7FCCB20E2B929062249385C5D
A52A3599AC8B4004EBEF066738E
DerivedSecretInfo: E6BF98F1F5EAF1F171F15B425F3983E510AD3D323951628868A7F706B71BC
F9C6B3D986C24DB64E9F25037B229C04E8F
sm4_key: E6BF98F1F5EAF1F171F15B425F3983E5
sm4_iv: 10AD3D323951628868A7F706B71BCF9C
已完成文件的加密...
hmac_key: 6B3D986C24DB64E9F25037B229C04E8F
已完成对加密文件的认证...
已经将加密和认证后的文件发送至客户端...
```

图2 与客户端对应的服务端程序某一次的运行结果

服务端和客户端程序某一次的输出结果分别如图2和4所示。其中，dh_pub_cli和dh_pub_ser分别是客户端和服务端在完执行密钥协商算法后各自产生的DH算法公钥。agreedkey是服务端和客户端通过DH算法产生的共享值。DerivedSecretInfo是KDF算法派生的秘密共享信息。sm4_key和sm4_iv分别表示SM4算法所使用的密钥和初始向量。hmac_key是HMAC算法所使用的密钥。

4.2 客户端程序运行结果

当客户端程序执行完毕后，可以在它所在的目录下发现另外三个文件，如图3所示。其中，文件cbc_sm4_cipher_hamc.tmp是它从服务端接收到的文件，这个文件由原始文件被加密后的密文和密文被认证后的消息认证码组成。文件cbc_sm4_cipher_tmp是文件cbc_sm4_cipher_hamc.tmp被验证通过而产生的仅含有加密密文的文件。文件receiver.txt是cbc_sm4_cipher_tmp文件被解密后所对应的原始数据文件（即内容与VS2012.5.iso相同）。

名称	修改日期	类型	大小
Release	2018/12/3 22:09	文件夹	
cbc_sm4_cipher_hamc_tmp	2018/12/3 22:50	文件	2,479,267 KB
cbc_sm4_cipher_tmp	2018/12/3 22:51	文件	2,479,267 KB
receive.txt	2018/12/3 22:55	文本文档	2,479,266 KB
secure_channel_client.cpp	2018/12/3 22:08	C++ Source	9 KB
secure_channel_client.vcxproj	2018/12/1 10:20	VC++ Project	8 KB
secure_channel_client.vcxproj....	2018/12/1 10:20	VC++ Project Fi...	1 KB
sin_prikey_client.der	2018/11/22 23:22	DER 文件	1 KB
sin_pubkey_server.der	2018/11/22 23:23	DER 文件	1 KB

图3 双方完成文件传输后客户端所在目录下含有的文件

```

C:\windows\system32\cmd.exe
输入要连接的服务端IP地址: 127.0.0.1
输入要连接的服务端绑定的端口号: 6666
客户端成功加载己方数字签名私钥和对方数字签名公钥...
客户端DH算法参数初始化完毕...
dh_pub_cli: 877C277BFB021CB11A03E280047304B9706DE898ADCBD1C79BAC4167DFD8EEF4B1A6
FB4D7CD7550A8EA28B839643818ACD64C9E194833C904E14DBEB3514961C26358C1C46850049C6C0
007021B91E87491522906338FDA7B25A9C8E5CFDCBF5FE7FE921A839AB6BE45236188A00B9B71014
D2F7862FE80881192571CEC5D51E
dh_pub_ser: 23ED6E6CEB30B89E89559A693905789D272900912D30283B89981B61ECD9F7E54AE1
8E5F41764D80E6CFDBF8842D0FDDBC61C0E3C7A43004309D39B2B72FFD0C3D00A835F59D5B82BB6
12ADB55D3C49B25C7518C40B68D1B861732CDC258653D4F95E5378BB8659E218053ADA414A6B2378
B4E68B1FCE69555A3473D1AACBC8
验证服务端DH算法公钥成功...
客户端和服务端协商密钥成功...
agreedkey: 651CEC2E7BA77401757078A19BA9722F83BA041A901E5DF56C1AA951C32A728185F15
9D7F5019F1924ACA8C5F87952C9D60DA2EDAABB1C8AF96BF49588F4A69D3DD229A81DB21F3D0418A
05FD32C2B519944EDE5E2A8D031A73A832111368255C1C6FAE9991D7FCCB20E2B929062249385C5D
A52A3599AC8B4004EBEF066738E
从服务端成功读取指定长度的文件...
读取的文件（已经被加密和认证）长度为2538768432（字节）
DerivedSecretInfo: E6BF98F1F5EAF1F171F15B425F3983E510AD3D323951628868A7F706B71BC
F9C6B3D986C24DB64E9F25037B229C04E8F
sm4_key: E6BF98F1F5EAF1F171F15B425F3983E5
sm4_iv: 10AD3D323951628868A7F706B71BCF9C
hmac_key: 6B3D986C24DB64E9F25037B229C04E8F
文件完整性校验完毕...
文件解密完毕...
  
```

图4 与服务端对应的客户端程序某一次的运行结果

5 声明

Cryptography

⇓

⇓

⇓

此为《深入浅出CryptoPP密码学库》随书电子文档，它仅包含书籍中示例程序的源代码。关于示例代码的解释说明，详见书籍相应章节内容。

由于作者水平有限，错误之处在所难免。欢迎通过如下方式反馈相关问题：

⇒ QQ: 1220195669

⇒ 微信: cc1220195669

⇓

⇓

⇓

《深入浅出CryptoPP密码学库》