

CryptoPP 库使用

参考书籍：《深入浅出CryptoPP密码学库》

crypto++ 密码学库：<https://github.com/weidai11/cryptopp>

电子书下载：https://gitee.com/locomotive_crypto

string 和 SecByteBlock类型互换

C++

```
1 // SecByteBlock 转 string
2 SecByteBlock iv; ... // C++-style cast
3 std::string token = std::string(reinterpret_cast<const char*>(iv.data()),
  iv.size());
4
5 // string 转 SecByteBlock
6 std::string str; ... // C++-style cast
7 SecByteBlock sbb(reinterpret_cast<const byte*>(str.data()), str.size());
```

第四章 初始CryptoPP库

Hex编解码字符串

C++

```
1  #include <iostream>
2  #include <filters.h>
3  #include <hex.h>
4  using namespace std;
5  using namespace CryptoPP;
6
7  int main() {
8      // 第一种编码方式
9      HexEncoder hex;
10     string str = "I like";
11     string hexstr;
12     hex.Detach(new StringSink(hexstr));
13     hex.Put(reinterpret_cast<byte*>(&str[0]),str.size()); // 注意是会追加写入的
14     cout << "str:" << str << endl;
15     cout << "hexstr:" << hexstr << endl;
16
17     // 第二种编解码写法 Source -> Filter -> Sink 这是一种Pipeline的方式
18     string encode,decode;
19     StringSource enc(str, true, new HexEncoder(new StringSink(encode)));
20     StringSource dec(hexstr, true, new HexDecoder(new StringSink(decode)));
21     cout << "encode:"<<encode << endl;
22     cout << "decode:"<<decode << endl;
23 }
```

第五章 随机数生成器

主要可以关注GenerateBlock方法，生成指定字节长度的随机数

C++

```
1  #include<osrng.h> // 可以使用 AutoSeededRandomPool , 该随机器不用设置种子
2  #include<rng.h> //包含LC_RNG算法的头文件
3  #include<iostream> //使用cout、cin
4  using namespace std; //std是C++的命名空间
5  using namespace CryptoPP; //CryptoPP是CryptoPP库的命名空间
6  #define Array_Size 64
7  int main()
8  {
9      //定义一个LC_RNG随机数发生器对象, 并设置其种子
10     LC_RNG rng(time(0));
11     cout << "产生一个比特的随机数: " << rng.GenerateBit() << endl;
12     cout << "产生一个字节的随机数: " << rng.GenerateByte() << endl;
13     byte output[Array_Size + 1] = {0}; //定义一个缓冲区
14
15     //产生Array_Size字节长度的随机数
16     rng.GenerateBlock(output, Array_Size); // 这里也可直接传入SecByteBlock
17     cout << "产生Array_Size长度的随机数 (十六进制): " << endl;
18     for (int i = 0; i < Array_Size; ++i)
19     { //将获得的随机数转换成十六进制并输出
20         printf("%02X", output[i]);
21     }
22     cout << endl;
23     cout << "产生一个100到1000之间的随机数: " << rng.GenerateWord32(100, 1000) <<
endl;
24
25     //丢弃掉随机数发生器接下来产生的100个字节数据
26     rng.DiscardBytes(100);
27     int array[] = { 1,2,3,4,5,6,7,8,9,10 };
28     rng.Shuffle(array, array + 10); //打乱数组array中元素的顺序
29
30     return 0;
31 }
```

第六章 Hash函数

C++

```
1  #include<sha.h> //使用SHA384
2  #include<filters.h>
3  #include<hex.h>
4  #include<iostream> //使用cout、cin
5  using namespace std; //std是C++的命名空间
6  using namespace CryptoPP; //CryptoPP是CryptoPP库的命名空间
7  int main()
```

即可

```
7 int main()
8 {
9     try
10    {
11        SHA256 sha; //定义一个SHA256的类对象
12        byte msg[] = "I like cryptography very much";
13
14        // 使用pipeline范式
15        SecByteBlock tmp(msg, sizeof(msg)-1);
16        string r;
17        StringSource s1 (tmp, tmp.size(),true,
18                        new HashFilter(sha, // 使用其他hash函数, 更换一下类型
19
20                        new HexEncoder(
21                        new StringSink(r))));
22        cout << r << endl;
23        r.clear(); // 清空一下, 不然后面会追加
24
25        // 连续两次Hash256, 再用Hex编码输出
26        SHA256 sha2;
27        StringSource s2 ("I like cryptography very much",true,
28                        new HashFilter(sha,
29                        new HashFilter(sha2,
30                        new HexEncoder(
31                        new StringSink(r))))) );
32        cout << r << endl;
33
34        SecByteBlock digest(sha.DigestSize()); //申请内存空间以存放消息摘要
35        //CalculateDigest()相当于Update()+Final()
36        // Update用来向sha输入, Final计算hash值, 同时重置hash函数内部状态
37        sha.CalculateDigest(digest, msg, sizeof(msg) - 1);
38        cout << "digest2=";
39        for (size_t i = 0; i < sha.DigestSize(); ++i)
40        { //以十六进制输出Hash值
41            printf("%02X", digest[i]);
42        }
43        cout << endl;
44
45        //计算msg消息的Hash值
46        bool res;
47        res = sha.VerifyDigest(digest, //可能抛出异常
48                             msg, sizeof(msg) - 1); //去掉字符串最后的'\0'
49        cout << "res = " << boolalpha << res << endl; // 这里的boolalpha 是为了
50        输出bool值true或者false
51    }
52    catch (const Exception& e)
```

```

53         cout << e.what() << endl; //异常原因
54     }
55
56     return 0;
57 }

```

第十一章 公钥密码学数学基础

大整数 与 大素数生成

☐ 可以再研究一下大整数的一些用法

C++

```

1  #include<integer.h> //使用Integer
2  #include<iostream> //使用cout、cin
3  #include<osrng.h> //使用AutoSeededRandomPool
4  #include<nbtheory.h> //使用PrimeAndGenerator、VerifyPrime
5  using namespace std; //std是C++的命名空间
6  using namespace CryptoPP; //CryptoPP是CryptoPP库的命名空间
7  int main()
8  {
9      AutoSeededRandomPool rng; // 定义随机数发生器对象
10     // 定义PrimeAndGenerator对象，利用随机数发生器rng产生素数p和q
11     // 要求产生的p是1024比特的素数，q是512比特的素数
12     PrimeAndGenerator pag(1, rng, 1024, 512);
13     Integer p = pag.Prime(); // 获取素数p的值
14     Integer q = pag.SubPrime(); // 获取素数q的值
15     Integer r = (p - 1) / q / 2; // 计算r的值，因为p=2*r*q+1, delta=1
16     cout << "p(" << p.BitCount() << "): " << p << endl; // 打印p的值及比特数
17     cout << "q(" << q.BitCount() << "): " << q << endl; // 打印q的值及比特数
18     cout << "r(" << r.BitCount() << "): " << r << endl; // 打印r的值及比特数
19     if (VerifyPrime(rng, r, 10)) // 验证r是否为素数
20     {
21         cout << "r是素数" << endl; // 如果r为素数，则输出该信息
22     }
23     else
24     {
25         cout << "r不是素数" << endl; // 如果r不为素数，则输出该信息
26     }
27     return 0;
28 }

```

第十三章 数字签名

电子书给出的参考是ECNR数字签名算法

[ECDSA椭圆曲线 - Crypto++](#)

https://www.cryptopp.com/wiki/Elliptic_Curve_Digital_Signature_Algorithm -》关于压缩公钥的实现可以参考 Compressed Point 这一部分

私钥和公钥生成、保存

C++

```
1  #include<integer.h>//使用Integer
2  #include<iostream>//使用cout、cin
3  #include<osrng.h>//使用AutoSeededRandomPool
4  #include<eccrypto.h>
5  #include <oids.h>
6  #include <files.h>
7  #include <filters.h>
8  #include <filesystem>
9  #include <hex.h>
10 #include <base32.h>
11 using namespace std;//std是C++的命名空间
12 using namespace CryptoPP;//CryptoPP是CryptoPP库的命名空间
13 void ECDSA_Generate()
14 {
15     /*
16     // 指定 private exponent 32字节随机数, 生成对应的私钥
17     string exp =
18         "E4A6CFB431471CFCAE491FD566D19C87082CF9FA7722D7FA24B2B3F5669DBEFB";
19
20     HexDecoder decoder;
21     decoder.Put((byte*)&exp[0], exp.size());
22     decoder.MessageEnd();
23
24     Integer x;
25     x.Decode(decoder, decoder.MaxRetrievable());
26
27     privateKey.Initialize(ASN1::secp256r1(), x);
28     */
29     AutoSeededRandomPool prng;
30     ECDSA<ECP, SHA256>::PrivateKey privateKey;
```

```

31
32     privateKey.Initialize( prng, ASN1::secp256k1() );
33
34     /* 使用ByteQueue 方便将公私钥存储在内存中, 如果要持久化到磁盘, 可以使用FileSink
35     ByteQueue queue;
36     privateKey.Save(queue);
37     privateKey.Load(queue);
38     */
39
40     // 验证密钥强度
41     bool result = privateKey.Validate( prng, 3 );
42     cout << boolalpha << result << endl;
43
44     Integer p = privateKey.GetPrivateExponent();
45     cout << "len:" << p.BitCount() << " Private Key:" << p << endl;
46     string priHexKey;
47     HexEncoder encoder(new StringSink(priHexKey));
48     p.Encode(encoder, 32);
49     cout << "priHexKey:" << priHexKey.size() << " " << priHexKey << endl;
50
51     // Save private key in PKCS #8 format
52     FileSink fs1( "../../keys/private.ec.der", true /*binary*/ );
53     privateKey.Save( fs1 );
54
55     // Generate publicKey
56     ECDSA<ECP, CryptoPP::SHA256>::PublicKey publicKey;
57     privateKey.MakePublicKey(publicKey);
58     const ECP::Point& q = publicKey.GetPublicElement();
59     const Integer& qx = q.x;
60     const Integer& qy = q.y;
61     string qxHex, qyHex;
62     HexEncoder encodex(new StringSink(qxHex));
63     HexEncoder encodery(new StringSink(qyHex));
64     qx.Encode(encodex, 32);
65     qy.Encode(encodery, 32);
66     cout << "len:" << qx.BitCount() << " Public Point x:" << qx << endl;
67     cout << "qxHex:" << qxHex << endl;
68     cout << "len:" << qy.BitCount() << " Public Point y:" << qy << endl;
69     cout << "qyHex:" << qyHex << endl;
70
71     // Save public key in X.509 format
72     FileSink fs2( "../../keys/public.ec.der", true /*binary*/ );
73     publicKey.Save( fs2 );
74
75 }

```

私钥和公钥的加载、签名和认证

C++

```
1 // 头文件和上述一样
2
3 void ECDSA_LOAD(){
4     AutoSeededRandomPool prng;
5     FileSource fs1( "../.../keys/private.ec.der", true /*pump all*/ );
6     FileSource fs2( "../.../keys/public.ec.der", true /*pump all*/ );
7     ECDSA<ECP, SHA256>::PrivateKey privateKey;
8     ECDSA<ECP, SHA256>::PublicKey publicKey;
9     // 加载私钥, 私钥格式: PKCS #8
10    privateKey.Load( fs1 );
11    publicKey.Load(fs2);
12
13    // 用私钥进行签名
14    ECDSA<ECP, SHA256>::Signer signer(privateKey);
15    string message = "Yoda said, Do or do not. There is no try.";
16    string signature;
17
18
19    StringSource s( message, true /*pump all*/,
20                   new SignerFilter( prng,
21                                   signer,
22                                   new StringSink( signature )
23                               ) // SignerFilter
24    ); // StringSource
25    cout << "signature len:" << signature.size() << " output:" << signature <<
endl;
26
27    // 将签名 (包含R, S) 转换成 DER 格式
28    std::string derSign;
29    // Make room for the ASN.1/DER encoding
30    derSign.resize(3+3+3+2+signature.size());
31    size_t converted_size = DSAConvertSignatureFormat(
32        (byte*) (&derSign[0]), derSign.size(), DSA_DER,
33        (const byte*) (signature.data()), signature.size(), DSA_P1363);
34    derSign.resize(converted_size);
35    cout << "DER len:" << derSign.size() << " DER:" << derSign << endl;
36    string hexDER;
37    StringSource toDER(derSign, true, new HexEncoder(new StringSink(hexDER)));
38    cout << "DER hex:" << hexDER << endl;
39
40
41    // 进行验证
42    bool result;
43    ECDSA<ECP, SHA256>::Verifier verifier(publicKey);
```



```
43     ECDSA<ECP, SHA256>::verifier verifier(publickey);
44     StringSource ss( signature+message, true /*pump all*/,
45                     new SignatureVerificationFilter(
46                         verifier,
47                         new ArraySink( (byte*)&result, sizeof(result) )
48                     ) // SignatureVerificationFilter
49     );
50
51     // 传统的C 方式 - 函数调用形式
52     //     result = verifier.VerifyMessage( (const byte*)message.data(),
53     //                                     message.size(), (const byte*)signature.data(), signature.size() );
54     if(result)
55         std::cout << "Verified signature on message" << std::endl;
56     else
57         std::cerr << "Failed to verify signature on message" << std::endl;
58 }
```