Elliptic Curve Digital Signature Algorithm

Elliptic Curve Digital Signature Algorithm, or ECDSA, is one of three digital signature schemes specified in FIPS-186 (http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf). The current revision is Change 4, dated July 2013. If interested in the non-elliptic curve variant, see Digital Signature Algorithm.

Before operations such as key generation, signing, and verification can occur, we must chose a field and suitable domain parameters. For demonstration purposes, this example will use F_p as the field. This means one template argument to ECDSA will include `ECP`. If we were using F_{2^m} , the template argument would be `EC2N`.

Crypto++ supplies a set of standard curves approved by ANSI, Brainpool, and NIST. Crypto++ does not provide curve generation functionality. If you need a custom curve, see Elliptic Curve Builder. Also see the Elliptic Curve Cryptography wiki page.

Contents

Crypto++ Validation

Crypto++'s performs ECDSA validation in `ValidateECDSA` from `valdat2.cpp`. Both F_p and F_{2^m} are demonstrated.

Choice of Fields

ECDSA is defined over both prime fields F_p and binary fields F_{2^m}. To operate over a prime field, use `ECP`:

```
ECDSA<ECP, SHA256>::Signer signer;
ECDSA<ECP, SHA256>::Verifier verifier;
```

For binary fields, specify `EC2N`:

```
ECDSA<EC2N, SHA256>::Signer signer;
ECDSA<EC2N, SHA256>::Verifier verifier;
```

Key Generation

The following discusses public and private key generation.

Private Key

To generate a private key for signing, perform the following. Note that `Initialize` functions which take a `RandomNumberGenerator` will generate a private key.

```
AutoSeededRandomPool prng;
ECDSA<ECP, SHA1>::PrivateKey privateKey;

privateKey.Initialize( prng, ASN1::secp160r1() );
bool result = privateKey.Validate( prng, 3 );
if( !result ) { ... }
```

An alternative to using a random number generator and OID is shown below. In the code below, the OID is used to construct an object for domain parameters.

```
ECDSA<ECP, SHA1>::PrivateKey privateKey;
DL_GroupParameters_EC<ECP> params(ASN1::secp160r1());

privateKey.Initialize(prng, params);
```

Once the private key has been generated, the private exponent can be retrieved as follows.

```
const Integer& x = privateKey.GetPrivateExponent()
```

Calling `GenerateRandom` as shown below will **not** work as expected - it will cause a `NotImplemented` exception with the message, *DL_GroupParameters_EC<EC>: curve generation is not implemented yet*.

```
AutoSeededRandomPool prng;

ECDSA<ECP, SHA1>::PrivateKey privateKey;
privateKey.GenerateRandom(prng, g_nullNameValuePairs);
```

Be careful when working with ECDSA's parameterized arguments. The following will save and load properly, despite the difference between SHA1 and SHA256. When a key is saved or loaded, it is done so in PKCS #8 (private key) or X509 format (public key). The key formats are ignorant to the objects which use them (such as ECDSA).

```
ECDSA<ECP, SHA1>::PrivateKey k1;
k1.Initialize( prng, secp160r1() );

const Integer& x1 = k1.GetPrivateExponent();
cout << "K1: " << std::hex << x1 << endl;

ByteQueue queue;
k1.Save(queue);

ECDSA<ECP, SHA256>::PrivateKey k2;
k2.Load(queue);

const Integer& x2 = k2.GetPrivateExponent();
cout << "K2: " << std::hex << x2 << endl;
```

```
$ ./cryptopp-test.exe
K1: fb468a2d41678c92b5b41b25f4e48c96700278e2h
K2: fb468a2d41678c92b5b41b25f4e48c96700278e2h
```

Signer

To generate a private key for signing using a `Signer`, perform the following.

```
AutoSeededRandomPool prng;
ECDSA<ECP, SHA1>::Signer signer;

signer.AccessKey().Initialize( prng, ASN1::secp160r1() );
bool result = signer.AccessKey().Validate( prng, 3 );
if( !result ) { ... }
```

We can also use a `PrivateKey` to initialize a signer:

```
ECDSA<ECP, SHA1>::PrivateKey privateKey
...
ECDSA<ECP, SHA1>::Signer signer(privateKey);

bool result = signer.AccessKey().Validate( prng, 3 );
if( !result ) { ... }
```

Public Key

To derive a public key from the private key, perform the following.

```
privateKey.MakePublicKey( publicKey );

bool result = publicKey.Validate( prng, 3 );
if( !result ) { ... }
```

Once the public key has been derived, the public element can be retrieved as follows.

```
const ECP::Point& q = publicKey.GetPublicElement()

const Integer& qx = q.x;
const Integer& qy = q.y;
```

Verifier

To use a high level verifier, perform the following.

```
ECDSA<ECP, SHA1>::PublicKey verifier( publicKey );
signer.AccessKey().Verify( prng, 3 );
```

Though unlikely, we could perform the following if the signer object were handy (the same signer and verifier usually do not exist in the same program).

```
ECDSA<ECP, SHA1>::PrivateKey privateKey = ...;
ECDSA<ECP, SHA1>::Signer signer( privateKey );
...

ECDSA<ECP, SHA1>::PublicKey verifier( signer );
bool result = verifier.AccessKey().Verify( prng, 3 );
if(!result) { ... }
```

Key Initialization

The following demonstrates public and private key initialization using previously generated parameters.

Private Key

Given a curve and the private exponent, perform the following to initialize the private key.

```
ECDSA<ECP, SHA1>::PrivateKey privateKey;
const Integer x = ...;       // private exponent

privateKey.Initialize( ASN1::secp160r1(), x );

bool result = privateKey.Validate( prng, 3 );
if( !result ) { ... }
```

In the example below, the private key is a hex encoded integer.

```
string exp = "E4A6CFB431471CFCAE491FD566D19C87082CF9FA7722D7FA24B2B3F5669DBEFB";

HexDecoder decoder;
decoder.Put((byte*)&exp[0], exp.size());
decoder.MessageEnd();

Integer x;
x.Decode(decoder, decoder.MaxRetrievable());

privateKey.Initialize(ASN1::secp256r1(), x);
```

Public Key

Similar to a private key, a public key is initialized as follows using the public element.

```
ECDSA<ECP, SHA1>::PublicKey publicKey;
const ECP::Point q = ...;    // public element

publicKey.Initialize( ASN1::secp160r1(), q );

bool result = publicKey.Validate( prng, 3 );
if( !result ) { ... }
```

In the example below, the public point is a simple concatenation of the `x` and `y` parts.

```
string pt = "2DB45A3F21889438B42C8F464C75292BACF5FDDB5DA0B492501B299CBFE92D8F"
            "DB90FC8FF4026129838B1BCAD1402CAE47FE7D8084E409A41AFCE16D63579C5F";

HexDecoder decoder;
decoder.Put((byte*)&pt[0], pt.size());
decoder.MessageEnd();

ECP::Point q;
size_t len = decoder.MaxRetrievable();

q.identity = false;
q.x.Decode(decoder, len/2);
q.y.Decode(decoder, len/2);

ECDSA<ECP, SHA256>::PublicKey publicKey;
publicKey.Initialize( ASN1::secp256r1(), q );
```

Key Persistence

The following uses `Load` and `Save` to read and write a PKCS #8 private or X509 public keys. For a complete discussion of PKCS #8 private keys and X509 public keys, see Keys and Formats.

Private Key

Given a public key and private key, we perform the following to save a private key to disk.

```
// Save private key in PKCS #8 format
FileSink fs( "private.ec.der", true /*binary*/ );
privateKey.Save( fs );
```

To load the private key from disk, perform the following:

```
// Load private key in PKCS #8 format
FileSource( "private.ec.der", true /*pump all*/ );
privateKey.Load( fs );

bool result = privateKey.Validate( prng, 3 );
if( !result ) { ... }
```

Signer

If working with a higher level `ECDSA<ECP, SHA1>::Signer`, use `AccessKey` to access the private key and call `Load` or `Save` on the returned object.

```
ECDSA<ECP, SHA1>::Signer signer( privateKey );

ByteQueue queue;
signer.AccessKey().Save( queue );
```

A ByteQueue is convenient for transferring a private key in memory. To save to disk, we can use a `FileSink`.

```
FileSink fs( "private.ec.der", true /*binary*/ );
signer.AccessKey().Save( fs );
```

Saving the key to disk using an ASCII encoding is shown below.

```
HexEncoder encoder( new FileSink( "private.ec.der", false /*binary*/ ));
signer.AccessKey().Save( encoder );
```

Public Key

Public keys are persisted similar to private keys.

```
// Save public key in X.509 format
FileSink fs( "public.ec.der", true /*binary*/ );
publiceKey.Save( fs );
```

```
// Load public key in X.509 format
FileSource fs( "public.ec.der", true /*pump all*/ );
publiceKey.Load( fs );

bool result = publicKey.Validate( prng, 3 );
if( !result ) { ... }
```

Verifier

If working with a higher level `ECDSA<ECP, SHA1>::Verfier`, use `AccessKey` to access the public key and call `Load` or `Save` on the returned object.

```
ECDSA<ECP, SHA1>::Verifier verifier( publicKey );

ByteQueue queue;
verifier.AccessKey().Save( queue );
```

A `ByteQueue` is convenient for transferring a public key in memory. To save to disk, we can use a `FileSink`.

```
FileSink fs( "public.ec.der", true /*binary*/ );
verifier.AccessKey().Save( fs );
```

Saving the key to disk using an ASCII encoding is shown below.

```
HexEncoder encoder( new FileSink( "public.ec.der", false /*binary*/ ));
verifier.AccessKey().Save( encoder );
```

Keys, Signers and Verifiers

Key Initialization shows how to generate or load a key. Message Signing and Message Verification shows how to sign and verify a message using Signers and Verifiers. Below is an example of how you can move among `PrivateKey`, `PublicKey`, `Signer` and `Verfier`.

```
ECDSA<ECP, SHA1>::Signer signer;
ECDSA<ECP, SHA1>::Verifier verifier;

ECDSA<ECP, SHA1>::PrivateKey& sKey = signer.AccessKey();
sKey.Initialize(prng, ASN1::secp160r1());
ECDSA<ECP, SHA1>::PublicKey& pKey = verifier.AccessKey();
sKey.MakePublicKey(pKey);
```

Message Signing

Signing a string is as follows. Note that a PRNG is required because the Digital Signature Standard specifies a per-message random value.

```
ECDSA<ECP, SHA1>::PrivateKey privateKey;
privateKey.Load(...);
ECDSA<ECP,SHA1>::Signer signer( privateKey );

AutoSeededRandomPool prng;
string message = "Yoda said, Do or do not. There is no try.";
string signature;

StringSource s( message, true /*pump all*/,
    new SignerFilter( prng,
        signer,
        new StringSink( signature )
    ) // SignerFilter
); // StringSource
```

The previous signing example used filters, while the example below uses traditional C-style function calls. After completion, `signature` will hold a properly sized binary string with the signature of the message.

```
ECDSA<ECP, SHA1>::Signer signer;
signer.AccessKey().Initialize( prng, secp160r1() );

string message = "Do or do not. There is no try.";

// Determine maximum size, allocate a string with the maximum size
size_t siglen = signer.MaxSignatureLength();
string signature(siglen, 0x00);

// Sign, and trim signature to actual size
siglen = signer.SignMessage( prng, (const byte*)&message[0], message.size(), (byte*)&signature[0] );
signature.resize(siglen);
```

Message Verification

ECDSA is a Signature Scheme with Appendix. This means that the original message must be presented to the verifier function. Verification is performed as follows.

By default, the filter uses flags equal to `SIGNATURE_AT_BEGIN | PUT_RESULT`, so a boolean value is placed in `result`. See the SignatureVerificationFilter page for details on the filter and the flags.

```
ECDSA<ECP, SHA1>::PublicKey publicKey;
publicKey.Load(...);
ECDSA<ECP,SHA1>::Verifier verifier(publicKey);
```

```
// Result of the verification process
bool result = false;

// Exactly what was signed in the previous step
string message = ...;
// Output from the signing operation in the previous step
string signature = ...;

StringSource ss( signature+message, true /*pump all*/,
    new SignatureVerificationFilter(
        verifier,
        new ArraySink( (byte*)&result, sizeof(result) )
    ) // SignatureVerificationFilter
);

// Verification failure?
if( !result ) {...}
```

If we wanted the `SignatureVerificationFilter` to throw on verification failure, the code is shown below. Note we should be prepared to catch a `SignatureVerificationFailed` exception.

```
static const int VERIFICATION_FLAGS = SIGNATURE_AT_BEGIN | THROW_EXCEPTION;
StringSource ss( signature+message, true /*pump all*/,
    new SignatureVerificationFilter(
        ECDSA<ECP,SHA1>::Verifier(publicKey),
        NULL, /* No need for attached filter */
        VERIFICATION_FLAGS
    ) // SignatureVerificationFilter
);
```

The previous verification example used filters, while the example below uses traditional C-style function calls. It builds upon the signing example above.

```
// Sign, and trim signature to actual size
siglen = signer.SignMessage( prng, (const byte*)&message[0], message.size(), (byte*)&signature[0] );
signature.resize(siglen);

ECDSA<ECP, SHA1>::PublicKey publicKey;
publicKey.Load(...);
ECDSA<ECP, SHA1>::Verifier verifier(publicKey);

bool result = verifier.VerifyMessage( (const byte*)&message[0], message.size(), (const byte*)&signature[0], si
gnature.size() );
if(result)
  cout << "Verified signature on message" << endl;
else
  cerr << "Failed to verify signature on message" << endl;
```

Domain Parameters

At times, it may be useful to dump domain parameters. We can use a single function to print both the public key and private key by accepting `DL_GroupParameters_EC<ECP>` as a parameter to the function since (not suprisingly) it is common to both objects.

```
ECDSA<ECP, SHA1>::PublicKey publicKey;
publicKey.Load(...);

PrintDomainParameters( publicKey );
```

```
void PrintDomainParameters( const DL_GroupParameters_EC<ECP>& params )
{
    cout << "Modulus:" << endl;
    cout << " " << params.GetCurve().GetField().GetModulus() << endl;

    cout << "Coefficient A:" << endl;
    cout << " " << params.GetCurve().GetA() << endl;

    cout << "Coefficient B:" << endl;
    cout << " " << params.GetCurve().GetB() << endl;

    cout << "Base Point:" << endl;
    cout << " X: " << params.GetSubgroupGenerator().x << endl;
    cout << " Y: " << params.GetSubgroupGenerator().y << endl;

    cout << "Subgroup Order:" << endl;
    cout << " " << params.GetSubgroupOrder() << endl;

    cout << "Cofactor:" << endl;
    cout << " " << params.GetCofactor() << endl;
}
```

A typical output from a 160 bit curve is shown below.

```
Modulus:
 1461501637330902918203684832716283019653785059327
Coefficient A:
 1461501637330902918203684832716283019653785059324
Coefficient B:
 163235791306168110546604919403271579530548345413
Base Point
 X: 425826231723888350446541592701409065913635568770
 Y: 203520114162904107873991457957346892027982641970
Subgroup Order:
 1461501637330902918203687197606826779884643492439
Cofactor:
 1

Private Exponent:
 542053241409584231606641348297804563147909139481

Public Element:
 X: 133066281928656700310125674035982115736779332 8918
 Y: 294709699265533759639226491877167235372762906422
```

Signature Sizes

The following table was compiled to offer a comparison of domain parameters and relative signature sizes. *Crypto++ Curve* specifies the programmatic name for the approved curve. The signature sizes are displayed in bytes.

The signature format is P1363 and the size is the `r || s` concatenation. Note the size of `r` and `s` depend on the field size, and not the size of the hash. Both `r` and `s` will each be padded on the left with 0's to ensure each is exactly the same size as the field. That also means the number of bytes in `r || s` is `2 * FieldSize()`.

Well Known Curve	Signature Size
secp112r1	28
secp128r1	32
secp160r1	40

secp192r1	48
secp224r1	56
secp256r1	64
secp384r1	96
secp521r1	132

Precomputed Hashes

Sometimes you may want to sign a precomputed hash. The code below allows you to sign a precomputed hash by copying the input to the hash function to the output of the hash function. Effectively it is an identity function. Also see Sign precomputed hash with ECDSA or DSA (http://stackoverflow.com/q/45186660/608639) on Stack Overflow.

Generally speaking signing a precomputed hash is a bad idea, especially if you don't compute the hash yourself. You don't want to disgorge creating the message digest from applying the secret key. You should avoid doing it.

```
$ cat test.cxx
#include "cryptlib.h"
#include "secblock.h"
#include "eccrypto.h"
#include "osrng.h"
#include "oids.h"
#include "hex.h"

#include <iostream>
#include <string>

using namespace CryptoPP;

template <unsigned int HASH_SIZE = 32>
class IdentityHash : public HashTransformation
{
public:
    CRYPTOPP_CONSTANT(DIGESTSIZE = HASH_SIZE)
    static const char * StaticAlgorithmName()
    {
        return "IdentityHash";
    }

    IdentityHash() : m_digest(HASH_SIZE), m_idx(0) {}

    virtual unsigned int DigestSize() const
    {
        return DIGESTSIZE;
    }

    virtual void Update(const byte *input, size_t length)
    {
        size_t s = STDMIN(STDMIN<size_t>(DIGESTSIZE, length),
                                        DIGESTSIZE - m_idx);
        if (s)
            ::memcpy(&m_digest[m_idx], input, s);
        m_idx += s;
    }

    virtual void TruncatedFinal(byte *digest, size_t digestSize)
    {
        ThrowIfInvalidTruncatedSize(digestSize);

        if (m_idx != DIGESTSIZE)
```

```
            throw Exception(Exception::OTHER_ERROR, "Input size must be " + IntToString(DIGESTSIZE));

        if (digest)
            ::memcpy(digest, m_digest, digestSize);

        m_idx = 0;
    }

private:
    SecByteBlock m_digest;
    size_t m_idx;
};

int main(int argc, char* argv[])
{
    AutoSeededRandomPool prng;

    ECDSA<ECP, IdentityHash<32> >::PrivateKey privateKey;
    privateKey.Initialize(prng, ASN1::secp256r1());

    std::string message;
    message.resize(IdentityHash<32>::DIGESTSIZE);
    ::memset(&message[0], 0xAA, message.size());

    ECDSA<ECP, IdentityHash<32> >::Signer signer(privateKey);
    std::string signature;

    StringSource ss(message, true,
                        new SignerFilter(prng, signer,
                            new HexEncoder(new StringSink(signature))
                        ) // SignerFilter
                    ); // StringSource

    std::cout << "Signature: " << signature << std::endl;

    return 0;
}
```

OpenSSL and Java Interop

ECDSA sometimes causes confusion when interop'ing with other libraries like OpenSSL and Java. The problem usually reduces to OpenSSL and Java use an ASN.1/DER signature format, and Crypto++ uses a IEEE P1363 format. Also see Cryptographic Interoperability: Digital Signatures (http://www.codeproject.com/Articles/25590/Cryptographic-Interoperability-Digital-Signatures), ECDSA sign with OpenSSL, verify with Crypto++ (http://stackoverflow.com/q/17316178/608639) and ECDSA sign with BouncyCastle and verify with Crypto++ (http://stackoverflow.com/q/48783809/608639).

The easiest way to interop is have Crypto++ convert the ASN.1/DER signature to a P1363 signature. The function of interest is DSAConvertSignatureFormat (http://github.com/weidai11/cryptopp/blob/master/dsa.cpp), which converts between ASN.1/DER, P1363 and OpenPGP formats.

Test Program

Here is the test program. Refer to Test Setup below for the generation of the parameters.

```
#include "cryptlib.h"
#include "eccrypto.h"
#include "dsa.h"
```

```
#include "sha.h"
#include "hex.h"

#include <iostream>

using namespace CryptoPP;

int main(int argc, char* argv[])
{
    // Load DER encoded public key
    FileSource pubKey("secp256k1-pub.der", true /*binary*/);
    ECDSA<ECP, SHA1>::Verifier verifier(pubKey);

    // Java or OpenSSL created signature. It is ASN.1
    //   SEQUENCE ::= { r INTEGER, s INTEGER }.
    const byte derSignature[] = {
        0x30, 0x44, 0x02, 0x20, 0x08, 0x66, 0xc8, 0xf1,
        0x6f, 0x15, 0x00, 0x40, 0x8a, 0xe2, 0x1b, 0x40,
        0x56, 0x28, 0x9c, 0x17, 0x8b, 0xca, 0x64, 0x99,
        0x37, 0xdc, 0x35, 0xad, 0xad, 0x60, 0x18, 0x4d,
        0x63, 0xcf, 0x4a, 0x06, 0x02, 0x20, 0x78, 0x4c,
        0xb7, 0x0b, 0xa3, 0xff, 0x4f, 0xce, 0xd3, 0x01,
        0x27, 0x5c, 0x6c, 0xed, 0x06, 0xf0, 0xd7, 0x63,
        0x6d, 0xc6, 0xbe, 0x06, 0x59, 0xe8, 0xc3, 0xa5,
        0xce, 0x8a, 0xf1, 0xde, 0x01, 0xd5
    };

    // P1363 'r || s' concatenation. The size is 32+32 due to field
    // size for r and s in secp-256. It is not 20+20 due to SHA-1.
    byte signature[0x40];
    DSAConvertSignatureFormat(signature, sizeof(signature), DSA_P1363,
                          derSignature, sizeof(derSignature), DSA_DER);

    // Message "Attack at dawn!"
    const byte message[] = {
        0x41, 0x74, 0x74, 0x61, 0x63, 0x6b, 0x20, 0x61,
        0x74, 0x20, 0x64, 0x61, 0x77, 0x6e, 0x21, 0x0a
    };

    // Standard signature checking in Crypto++
    bool result = verifier.VerifyMessage(message, sizeof(message), signature, sizeof(signature));
    if (result)
        std::cout << "Verified message" << std::endl;
    else
        std::cout << "Failed to verify message" << std::endl;

    return 0;
}
```

And here is the result of running the test program.

```
$ ./test.exe
Verified message
```

Test Setup

Here is the setup used for the sample code shown above.

```
$ cat test.txt | openssl dgst -ecdsa-with-SHA1 -sign secp256k1-key.pem -keyform DER > test.sig
```

The command uses and produces the following parameters.

```
$ cat test.txt
Attack at dawn!

$ hexdump -C test.txt
00000000  41 74 74 61 63 6b 20 61  74 20 64 61 77 6e 21 0a  |Attack at dawn!.|
00000010

# Create private key in PEM format
$ openssl ecparam -name secp256k1 -genkey -noout -out secp256k1-key.pem

$ cat secp256k1-key.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIO0D5Rjmes/91Nb3dHY9dxmbM7gVfxmB2+OVuLmWMbGXoAcGBSuBBAAK
oUQDQgAEgVNEuirUNCEVdf7nLSBUgU1GXLrtIBeglIbK54s91HlWKOKjk4CkJ3/B
wGAfcYKa+DgJ2IUQSD15K1T/ghM9eQ==
-----END EC PRIVATE KEY-----

# Convert private key to ASN.1/DER format
$ openssl ec -in secp256k1-key.pem -inform PEM -out secp256k1-key.der -outform DER

$ dumpasn1 secp256k1-key.der
  0 116: SEQUENCE {
  2   1:   INTEGER 1
  5  32:   OCTET STRING
        :     ED 03 E5 18 E6 7A CF FD D4 D6 F7 74 76 3D 77 19
        :     9B 33 B8 15 7F 19 81 DB E3 95 B8 B9 96 31 B1 97
 39   7:   [0] {
 41   5:     OBJECT IDENTIFIER secp256k1 (1 3 132 0 10)
        :     }
 48  68:   [1] {
 50  66:     BIT STRING
        :       04 81 53 44 BA 2A D4 34 21 15 75 FE E7 2D 20 54
        :       81 4D 46 5C BA ED 20 17 A0 94 86 CA E7 8B 3D D4
        :       79 56 28 E2 A3 93 80 A4 27 7F C1 C0 60 1F 71 82
        :       9A F8 38 09 D8 85 10 48 3D 79 2B 54 FF 82 13 3D
        :       79
        :     }
        :   }

# Create public key from private key
$ openssl ec -in secp256k1-key.der -inform DER -pubout -out secp256k1-pub.der -outform DER

$ dumpasn1 secp256k1-pub.der
  0  86: SEQUENCE {
  2  16:   SEQUENCE {
  4   7:     OBJECT IDENTIFIER ecPublicKey (1 2 840 10045 2 1)
 13   5:     OBJECT IDENTIFIER secp256k1 (1 3 132 0 10)
        :     }
 20  66:   BIT STRING
        :     04 81 53 44 BA 2A D4 34 21 15 75 FE E7 2D 20 54
        :     81 4D 46 5C BA ED 20 17 A0 94 86 CA E7 8B 3D D4
        :     79 56 28 E2 A3 93 80 A4 27 7F C1 C0 60 1F 71 82
        :     9A F8 38 09 D8 85 10 48 3D 79 2B 54 FF 82 13 3D
        :     79
        :   }

# Sign the message using the private key
$ cat test.txt | openssl dgst -ecdsa-with-SHA1 -sign secp256k1-key.der -keyform DER > test.sig

# Dump the signature as hex
$ hexdump -C test.sig
00000000  30 44 02 20 08 66 c8 f1  6f 15 00 40 8a e2 1b 40  |0D. .f..o..@...@|
00000010  56 28 9c 17 8b ca 64 99  37 dc 35 ad ad 60 18 4d  |V(....d.7.5..<tt>.M|
00000020  63 cf 4a 06 02 20 78 4c  b7 0b a3 ff 4f ce d3 01  |c.J.. xL....O...|
00000030  27 5c 6c ed 06 f0 d7 63  6d c6 be 06 59 e8 c3 a5  |'\l....cm...Y...|
00000040  ce 8a f1 de 01 d5                                 |......|
00000046

# Dump the signature as ASN.1/DER
$ dumpasn1 test.sig
  0  68: SEQUENCE {
  2  32:   INTEGER
```

```
      :     08 66 C8 F1 6F 15 00 40 8A E2 1B 40 56 28 9C 17
      :     8B CA 64 99 37 DC 35 AD AD 60 18 4D 63 CF 4A 06
36  32:   INTEGER
      :     78 4C B7 0B A3 FF 4F CE D3 01 27 5C 6C ED 06 F0
      :     D7 63 6D C6 BE 06 59 E8 C3 A5 CE 8A F1 DE 01 D5
      :   }
```

Downloads

ECDSA-Test.zip - Crypto++ ECDSA sample program using filters

ECDSA-Test-C.zip - Crypto++ ECDSA sample program using C Style API