**Cryptography Next Generation (CNG) for ECDSA**

[[](https://medium.com/@billatnapier?source=post_page-----0cb318d7bada--------------------------------)](https://medium.com/@billatnapier?source=post_page-----0cb318d7bada--------------------------------)

[[ASecuritySite: When Bob Met Alice](https://medium.com/asecuritysite-when-bob-met-alice?source=post_page-----0cb318d7bada--------------------------------)](https://medium.com/asecuritysite-when-bob-met-alice?source=post_page-----0cb318d7bada--------------------------------)

[Prof Bill Buchanan OBE](https://medium.com/@billatnapier?source=post_page-----0cb318d7bada--------------------------------)

·

Follow

Published in

[ASecuritySite: When Bob Met Alice](https://medium.com/asecuritysite-when-bob-met-alice?source=post_page-----0cb318d7bada--------------------------------)

·

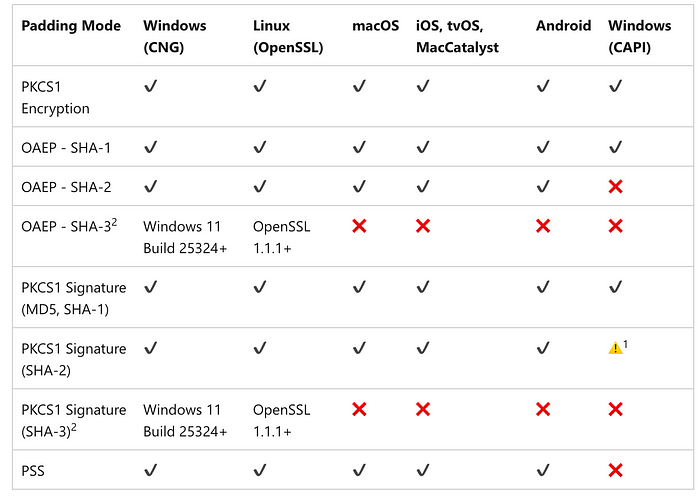
4 min read

·

6 days ago

4

I know the title sounds so interesting, but this article is just **plain old vanilla** ECDSA digital signing. But it is a good example of showing the key elements of a digital signature. So what is CNG? Well, CNG is Microsoft’s attempt at unbundling OpenSSL from the Windows operating system and producing core .NET code that will run only on a Windows machine.



Overall, Microsoft has tended to be rather slow in adopting new cryptography methods. CNG was released in 2021 and is seen as a direct long-term replacement for the rather cumbersome CryptoAPI. The core classes are defined with a CNG in the name, such as:

ECDiffieHellmanCng  
ECDsaCng

When compiled, the CNG class will get a warning message. For example, with ECDsaCng() we get:

This call site is reachable on all platforms.   
'ECDsaCng' is only supported on: 'windows'.

Thus, if you need your code to be compatible with Mac OS and Linux systems, you would probably avoid using CNG.

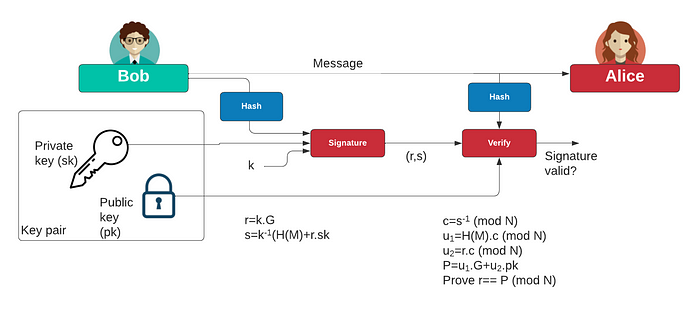
**Method**

With Elliptic Curve Cryptography (ECC) we can use a Weierstrass curve form of the form of *y*²=*x*³+*ax*+*b*(mod *p*) . Bitcoin and Ethereum use secp256k1 and which has the form of *y²*=*x*³+7 (mod *p*). In most cases, though, we use the NIST-defined curves. These are SECP256R1, SECP384R1, and SECP521R1, but we also use SECP224R1 and SECP192R1. SECP256R1(NIST P256) has 256-bit (x,y) points, and where the private key is a 256-bit scalar value (*a*) and which gives a public key point of:

*a*.*G*

**Creating the signature**

An outline of ECDSA is:



**Code**

First we create a folder named “bc\_ecdsa\_cng, and then go into that folder. We can create a Dotnet console project for .NET 8.0 with:

dotnet new console --framework net8.0

Initially Bob has a message to sign, he must first convert it to a byte array with:

var msg=Encoding.ASCII.GetBytes(message);

Next we can create a CNG object for Bob using a given curve (curvename):

var c = ECCurve.CreateFromFriendlyName(curvename);  
  
   
var bob = new ECDsaCng(c);

We can then take a SHA-256 hash of the message and produce the signature with [[here](https://asecuritysite.com/csharp/dotnet_ecdsa)]:

var h1=SHA256.Create();   
ar res= h1.ComputeHash(msg);  
  
var sighash=bob.SignHash(res);

This signature will have the (r, s) values. We can display these with [[here](https://asecuritysite.com/csharp/dotnet_ecdsa)]:

var r=Convert.ToHexString(sighash[0..(sighash.Length/2)]);  
var s=Convert.ToHexString(sighash[(sighash.Length/2)..sighash.Length]);  
   
Console.WriteLine("\nSignature (r,s)):\t{0}, {1}",r,s);

Now, Bob sends the message and signature to Alice, along with his exported public key [[here](https://asecuritysite.com/csharp/dotnet_ecdsa)]:

var bobPublic = bob.ExportSubjectPublicKeyInfoPem();

Alice receives Bob’s public key, and check the message hash with the signature [[here](https://asecuritysite.com/csharp/dotnet_ecdsa)]:

alice.ImportFromPem(bobPublic);  
var rtn=alice.VerifyHash(res,sighash);

If rtn is true, the signature checks out.

**Displaying keys**

If Bob has a private key of D and a point key point of Q, we can display these in several forms, including as hex values, a PEM format, and a DER format [[here](https://asecuritysite.com/csharp/dotnet_ecdsa)]:

var b = bob.ExportParameters(true);  
Console.WriteLine("\n\nBob Private Key D:\t\t{0}",Convert.ToHexString(b.D));  
Console.WriteLine("Bob Private Key (DER):\t\t{0}",Convert.ToHexString(bob.ExportPkcs8PrivateKey()));  
Console.WriteLine("Bob Private Key (PEM):\n{0}",bob.ExportECPrivateKeyPem());  
   
Console.WriteLine("\nBob Public Key Q:\t\t({0}, {1})",Convert.ToHexString(b.Q.X),Convert.ToHexString(b.Q.Y));  
Console.WriteLine("Bob Public Key (DER):\t\t{0}",Convert.ToHexString(bob.ExportSubjectPublicKeyInfo()));  
Console.WriteLine("Bob Public Key (PEM):\n{0}",bob.ExportSubjectPublicKeyInfoPem());

The full code is [[here](https://asecuritysite.com/csharp/dotnet_ecdsa)]:

namespace CngECDSA  
{  
 using System.Security.Cryptography;  
 using System.Text;  
class Program  
 {  
   
 static void Main(string[] args)  
  
 {  
  
 var curvename="nistP256";  
 var h=CngAlgorithm.Sha256;  
 var hashmethod="SHA256";  
 var message="Hello";  
try {  
 if (args.Length >0) message=args[0];  
 if (args.Length >1) curvename=args[1];  
 if (args.Length >2) hashmethod=args[2];  
  
 var c = ECCurve.CreateFromFriendlyName(curvename);  
 var msg=Encoding.ASCII.GetBytes(message);  
   
 var bob = new ECDsaCng(c);  
  
 var h1=SHA256.Create();   
 var res= h1.ComputeHash(msg);  
 if (hashmethod=="SHA256") { var h2=SHA256.Create(); res= h2.ComputeHash(msg); }   
 if (hashmethod=="MD5") { var h3=MD5.Create(); res= h3.ComputeHash(msg); }  
 if (hashmethod=="SHA1") { var h4=SHA1.Create(); res= h4.ComputeHash(msg); }  
 if (hashmethod=="SHA384") { var h5=SHA384.Create(); res= h5.ComputeHash(msg); }  
 if (hashmethod=="SHA512") { var h6=SHA512.Create(); res= h6.ComputeHash(msg); }  
  
   
 var sighash=bob.SignHash(res);  
 var alice = new ECDsaCng();  
 var bobPublic = bob.ExportSubjectPublicKeyInfoPem();  
 alice.ImportFromPem(bobPublic);  
 var rtn=alice.VerifyHash(res,sighash);  
   
   
 Console.WriteLine("Message:\t{0}",message);  
 Console.WriteLine("Hash:\t\t{0}",Convert.ToHexString(res));  
 Console.WriteLine("Curve:\t\t{0}",curvename);  
 Console.WriteLine("Hash:\t\t{0}",hashmethod);  
 var r=Convert.ToHexString(sighash[0..(sighash.Length/2)]);  
 var s=Convert.ToHexString(sighash[(sighash.Length/2)..sighash.Length]);  
   
 Console.WriteLine("\nSignature (r,s)):\t{0}, {1}",r,s);  
 Console.WriteLine("Verified:\t\t{0}",rtn);  
  
   
 var b = bob.ExportParameters(true);  
 Console.WriteLine("\n\nBob Private Key D:\t\t{0}",Convert.ToHexString(b.D));  
 Console.WriteLine("Bob Private Key (DER):\t\t{0}",Convert.ToHexString(bob.ExportPkcs8PrivateKey()));  
 Console.WriteLine("Bob Private Key (PEM):\n{0}",bob.ExportECPrivateKeyPem());  
   
 Console.WriteLine("\nBob Public Key Q:\t\t({0}, {1})",Convert.ToHexString(b.Q.X),Convert.ToHexString(b.Q.Y));  
 Console.WriteLine("Bob Public Key (DER):\t\t{0}",Convert.ToHexString(bob.ExportSubjectPublicKeyInfo()));  
 Console.WriteLine("Bob Public Key (PEM):\n{0}",bob.ExportSubjectPublicKeyInfoPem());  
  
   
   
 } catch (Exception e) {  
 Console.WriteLine("Error: {0}",e.Message);  
 }  
 }  
 }  
}

A sample run for NIST P256 gives [[here](https://asecuritysite.com/csharp/dotnet_ecdsa)]:

Message: Hello World!  
Hash: 7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069  
Curve: nistP256  
Hash: SHA256  
  
Signature (r,s)): BC37DFD5737E3E3F43E3F635639C13B7626F1BC072B3524E1B2A92A7C616F2E3, AFD77CF1C7AF81984DA7F659D3A39E264F5C5074968211DAA1DBED92BE8C37A9  
Verified: True  
  
Bob Private Key D: 85779B1292E4EC6ABBD26B237A9E4BC7A6563A8788909794C3D1BB61374C7EBA  
Bob Private Key (DER): 3081A2020100301306072A8648CE3D020106082A8648CE3D03010704793077020101042085779B1292E4EC6ABBD26B237A9E4BC7A6563A8788909794C3D1BB61374C7EBAA00A06082A8648CE3D030107A14403420004A279736886AEC3353E2192A53BC955FFEDA5DDCD12F153E8ACA3852E0CCC68435D51D260177061E44899DD68E1FC3AC9C9BF58E038083FB77DA2FB0529FF3FAEA00D300B0603551D0F310403020080  
Bob Private Key (PEM):  
-----BEGIN EC PRIVATE KEY-----  
MHcCAQEEIIV3mxKS5Oxqu9JrI3qeS8emVjqHiJCXlMPRu2E3TH66oAoGCCqGSM49  
AwEHoUQDQgAEonlzaIauwzU+IZKlO8lV/+2l3c0S8VPorKOFLgzMaENdUdJgF3Bh  
5EiZ3Wjh/DrJyb9Y4DgIP7d9ovsFKf8/rg==  
-----END EC PRIVATE KEY-----  
Bob Public Key Q: (A279736886AEC3353E2192A53BC955FFEDA5DDCD12F153E8ACA3852E0CCC6843, 5D51D260177061E44899DD68E1FC3AC9C9BF58E038083FB77DA2FB0529FF3FAE)  
Bob Public Key (DER): 3059301306072A8648CE3D020106082A8648CE3D03010703420004A279736886AEC3353E2192A53BC955FFEDA5DDCD12F153E8ACA3852E0CCC68435D51D260177061E44899DD68E1FC3AC9C9BF58E038083FB77DA2FB0529FF3FAE  
Bob Public Key (PEM):  
-----BEGIN PUBLIC KEY-----  
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEonlzaIauwzU+IZKlO8lV/+2l3c0S  
8VPorKOFLgzMaENdUdJgF3Bh5EiZ3Wjh/DrJyb9Y4DgIP7d9ovsFKf8/rg==  
-----END PUBLIC KEY-----

**Conclusions**

The main things to learn from this? There’s a public key, a hashing method, a private key, and a signature (r,s), and they all come together to create a world of digital trust. Try it here:

<https://asecuritysite.com/csharp/dotnet_ecdsa>

and for ECDH key exchange:

<https://asecuritysite.com/csharp/dotnet_ec>