Diffie-hellman

Diffie-Hellman
Documentation (http://www.cryptopp.com/docs/ref/class_d_h2.html)
`#include <cryptopp/dh2.h>`

Diffie-Hellman is a key agreement algorithm which allows two parties to establish a secure communications channel. The original Diffie-Hellman is an anonymous protocol meaning it is not authenticated, so it is vulnerable to man-in-the-middle attacks. Crypto++ exposes the unauthenticated protocol through `DH` classes. Extensions to the original Diffie-Hellman includes authentication which hardens the exchange protocol against many man-in-the-middle attacks. The authenticated version of Diffie-Hellman is usually referred to as Unified Diffie-Hellman. Crypto++ offers the Unified Diffie-Hellman through its `DH2` classes.

A number of standard bodies have Diffie-Hellman implementations, including RFC 2631, *Diffie-Hellman Key Agreement Method (http://tools.ietf.org/html/rfc2631)*, ANSI X9.42, *Agreement Of Symmetric Keys Using Diffie-Hellman and MQV Algorithms*, and IEEE P1363, *Standard Specifications for Public Key Cryptography, Annex D*. Each implementation is slightly different and might not interoperate.

When data is in motion, key exchange and transport is usually the starting point. Unfortunately, exchange and transport are often an after thought in a project, especially if a resident cryptography enthusiast or cryptographer is not available. The result can sometimes be a weak system, where an adversary can recover a session key during channel setup due to a weak or flawed exchange and transport (why attack AES directly when its easier to attack a flawed exchange?).

The examples below use a 1024-bit modulus to speed up computation and keep the output manageable. As with asymmetric keys, symmetric ciphers, hashes, and MACs, an appropriate security level should be chosen. This typically means transporting an AES-128 bit key using a prime group with a minimum of 3072-bits, or an elliptic curve over a prime field with a minimum size of 256 bits.

Finally, this page is concerned with two party key agreement. For group key agreement and multicast scenarios, see *Multicast Security: A Taxonomy and Some Efficient Constructions (http://cseweb.ucsd.edu/~daniele/papers/CGIMNP.pdf)* and *Provably Authenticated Group Diffie-Hellman Key Exchange (http://www.di.ens.fr/~bresson/papers/BreChePoi01_full.pdf)*.

Contents

Crypto++ Validation

Crypto++ demonstrates the use of Diffie-Hellman in `validate2.cpp`, functions `SimpleKeyAgreementValidate` and `AuthenticatedKeyAgreementValidate`. The functions have the following signatures.

```
bool SimpleKeyAgreementValidate(SimpleKeyAgreementDomain &d)
```

and

```
bool AuthenticatedKeyAgreementValidate(AuthenticatedKeyAgreementDomain &d)
```

`SimpleKeyAgreementDomain` is non-authenticated Diffie-Hellman and uses class `DH`. `AuthenticatedKeyAgreementDomain` is authenticated Diffie-Hellman (Unified Diffie-Hellman) and uses class `DH2`.

Sample Programs

The first two examples generate Diffie-Hellman parameters and load standardized parameters (that is, initializes a Crypto++ object with standard parameters). The standard parameters are usually provided by bodies such as ANSI, IEEE, IETF, and NIST.

Details Lost in Generation and Initialization discusses security level which are not readily apparent. Its the sort of thing that can subtly make a system insecure, even though the system was built from secure components. Though the topic should be presented first, it is visited after the first two examples so there is a frame of reference.

Generating Parameters (MODP)

Crypto++ generates a safe prime for use in Diffie-Hellman. Generating safe primes can be time consuming. It is not uncommon to experience generations times that appear like a program hang. For those who are interested, rsa_kgen_prof.zip demonstrates how to instrument code to track the generation time using the `ThreadUserTimer` class.

The example below creates the classical parameters *p* and *g*. The program then verifies the prime and generator, calculates and verifies *q*, and prints the order of the subgroup. Most of the interesting work is performed in `nbtheory.cpp` using `PrimeAndGenerator::Generate`. The full program is available in dh-param.zip.

```
DH dh;
AutoSeededRandomPool rnd;

dh.AccessGroupParameters().GenerateRandomWithKeySize(rnd, 1024);

const Integer& p = dh.GetGroupParameters().GetModulus();
cout << "P: " << p << endl;

Integer q = (p-1)/2;
cout << "Q: " << q << endl;

const Integer& g = dh.GetGroupParameters().GetGenerator();
cout << "G: " << g << endl;

Integer r = dh.GetGroupParameters().GetSubgroupOrder();
cout << "Subgroup order: " << r << endl;
```

A typical run of dh-param.exe is shown below.

```
$ ./dh-param.exe
P (1024): ab359aa76a6773ed7a93b214db0c25d0160817b8a893c001c761e198a3694509ebe8
7a5313e0349d95083e5412c9fc815bfd61f95ddece43376550fdc624e92ff38a415783b9726120
4e05d65731bba1ccff0e84c8cd2097b75feca1029261ae19a389a2e15d2939314b184aef707b82
eb94412065181d23e04bf065f4ac413fh
Q (1023): 559acd53b533b9f6bd49d90a6d8612e80b040bdc5449e000e3b0f0cc51b4a284f5f4
3d2989f01a4eca841f2a0964fe40adfeb0fcaeef67219bb2a87ee3127497f9c520abc1dcb93090
2702eb2b98ddd0e67f87426466904bdbaff650814930d70cd1c4d170ae949c98dh-inita58c257
7b83dc175ca2090328c0e91f025f832fa56209fh
G (2): 2.
Subgroup order (1023): 559acd53b533b9f6bd49d90a6d8612e80b040bdc5449e000e3b0f0c
c51b4a284f5f43d2989f01a4eca841f2a0964fe40adfeb0fcaeef67219bb2a87ee3127497f9c52
0abc1dcb930902702eb2b98ddd0e67f87426466904bdbaff650814930d70cd1c4d170ae949c98a
58c2577b83dc175ca2090328c0e91f025f832fa56209fh
```

If we increase the bit size to 3072 to meet security levels for an equivalent AES-128 key we find it takes about 18 minutes to generate the safe prime on a mid-2009 Core 2 Duo.

```
$ date && ./dh-gen.exe 3072 && date
Wed Dec  1 09:25:00 EST 2010
P: bfef6a7e54ba72ba09873e7ad1f3dbac8e4ae5e55534974948d67c38e07baa8036f321b6372
9469c02f70089426b6dcbf3b1f0c3574fc73f2509995e32b5a04e983d4363145f7ba599e760e85
50991d21716f9b298a636b9a929c68a5ccc9af4f3cf1a8815002026df2853efbc28e9e1d7cc75b
a53af6b618bed49b4d447100c4ddfd086d89f90a145ab912e7f335a595715eccb71409f3eaca38
6db495d7c9e054001a514c2cb15c4438ca2e573defed0aaa76a4068bc1b2953094713a48f21168
fbd30020fbe6e977e6d6908a25bd2983ff5e7e1237cf80b3b5f2f99dde22c63977718e073e7f2f
44fea12d82ff53c031874958d6f6d81d156cc78b7a20ef8d31860f0dca106feacc7a2b609dd259
3b1b1c1bc0d116a1ed2472f7507059f8108b0b6a3f69f3661e352c546c523ac85908e32f35155e
9f4f0abbb70c83e122e51a4fee3184c45ecdfd50b5da59fefefc1f0d441074ed2269dc4b274e59
1a8464d2c34f9c2e8cd1db308088503e9bbdf325f67b4830833ca10baae2fd1de7b57h
```

```
G: 2h
Wed Dec  1 09:43:39 EST 2010
```

You can also use the `PrimeAndGenerator` class to generate the domain parameters:

```
AutoSeededRandomPool prng;
Integer p, q, g;
PrimeAndGenerator pg;

pg.Generate(1, prng, 512, 511);
p = pg.Prime();
q = pg.SubPrime();
g = pg.Generator();

DH dh(p, q, g);
SecByteBlock t1(dh.PrivateKeyLength()), t2(dh.PublicKeyLength());
dh.GenerateKeyPair(prng, t1, t2);
Integer k1(t1, t1.size()), k2(t2, t2.size());

cout << "Private key:\n";
cout << hex << k1 << endl;
cout << "Public key:\n";
cout << hex << k2 << endl;
```

Output of the program above will be similar to the following.

```
$ ./cryptest.exe
Private key:
72b45a42371545e9d4880f48589aefh
Public key:
45fdb13f97b1840626f0250cec1dba4a23b894100b51fb5d2dd13693d789948f8bfc88f9200014b2
ba8dd8a6debc471c69ef1e2326c61184a2eca88ec866346bh
```

Parameter Initialization (MODP)

Often times, key exchange will occur using standardized parameters. For example, RFC 3526 (http://tools.ietf.org/html/rfc3526) (*More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*) and RFC 5114 (http://tools.ietf.org/html/rfc5114) (*Additional Diffie-Hellman Groups for Use with IETF Standards*). The standards will provide either {p,g} or {p,q,g}. *p* and *g* are the customary prime and generator. *q* is the order of the subgroup and due to Schnorr.

The example below is available in dh-init.zip, and loads RFC 5114's 1024-bit MODP group specified in section 2.1. Rather than calling `GenerateRandomWithKeySize`, a call is made to `Initialize` since *p*, *q*, and *g* are available.

Note that we do not enforce a safe prime. The parameters were selected by the IETF to provide the stated security level, which reduces the number of modular multiplications to increase efficiency.

```
// http://tools.ietf.org/html/rfc5114#section-2.1
Integer p("0xB10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C6"
        "9A6A9DCA52D23B616073E28675A23D189838EF1E2EE652C0"
        "13ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD70"
        "98488E9C219A73724EFFD6FAE5644738FAA31A4FF55BCCC0"
        "A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708"
        "DF1FB2BC2E4A4371");
```

```
Integer g("0xA4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507F"
        "D6406CFF14266D31266FEA1E5C41564B777E690F5504F213"
        "160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1"
        "909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A28A"
        "D662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C2A24"
        "855E6EEB22B3B2E5");

Integer q("0xF518AA8781A8DF278ABA4E7D64B7CB9D49462353");

DH dh;
dh.AccessGroupParameters().Initialize(p, q, g);

p = dh.GetGroupParameters().GetModulus();
cout << "P : " << p << endl;

g = dh.GetGroupParameters().GetGenerator();
cout << "G : " << std::dec << g << endl;

q = dh.GetGroupParameters().GetSubgroupOrder();
cout << "Subgroup order: " << q << endl;
```

A run of dh-init.exe is shown below. The run is not typical - it always produces the same result.

```
$ ./dh-init.exe
P (1024): b10b8f96a080e01dde92de5eae5d54ec52c99fbcfb06a3c69a6a9dca52d23b616073
e28675a23d189838ef1e2ee652c013ecb4aea906112324975c3cd49b83bfaccbdd7d90c4bd7098
488e9c219a73724effd6fae5644738faa31a4ff55bccc0a151af5f0dc8b4bd45bf37df365c1a65
e68cfda76d4da708df1fb2bc2e4a4371h
G (1024): a4d1cbd5c3fd34126765a442efb99905f8104dd258ac507fd6406cff14266d31266f
ea1e5c41564b777e690f5504f213160217b4b01b886a5e91547f9e2749f4d7fbd7d3b9a92ee190
9d0d2263f80a76a6a24c087a091f531dbf0a0169b6a28ad662a4d18e73afa32d779d5918d08bc8
858f4dcef97c2a24855e6eeb22b3b2e5h
Subgroup order (160): f518aa8781a8df278aba4e7d64b7cb9d49462353h
```

Details Lost in Generation and Initialization

Generating Parameters (MODP) and Parameter Initialization (MODP) are fairly straight forward - they generate or initialize Diffie-Hellman parameters. Unfortunately, a fair amount of detail was not readily apparent (or was lost), and the loss could result in an system that is insecure overall.

In Generating Parameters (MODP), Crypto++ generated a safe prime. A safe prime is of the form `p = 2q + 1`, with *p* and *q* prime. The safe prime *p* offers a subgroup nearly the size of the modulus. That is, a 1024-bit prime usually produces a subgroup of 1023-bits. The order of the subgroup means the size of the data or secret (in bits) can be in the interval [2, 1023]. Since there are no free lunches, this type of Diffie-Hellman group will incur 1023 square and multiply operations during exponentiation.

In contrast, Parameter Initialization (MODP) does not use a safe prime so the size of the data or secret (in bits) is reduced. The primes are of the form `p = qr + 1`, with *p*, *q* prime (this is known a Schnorr Group). From the example above, the IETF provided parameters with a subgroup order of 160 bits. This means the data or secret can be in the range [2, 160] bits. The reduced subgroup order has an upside: the algorithm will only need to perform 160 square and multiply operations, which improves efficiency.

While efficiency is an important design consideration, cryptography must first be secure. There are two known instance problems or attacks on Diffie-Hellman: the first attack is used against classical discrete logarithms (logarithms over a finite field) and is often Index calculus, modified Pollard's rho, or Baby-step giant-step. The

methods run in roughly sub-exponential time (but not polynomial time). The second attack, due to Pohlig and Hellman, attempts to compromise the system by taking advantage of structure of cyclic subgroups. That is, attacks are mounted against the parameter *q*. The Pohlig-Hellman algorithm runs in square root time.

Here is the take-away by example: suppose you need to exchange an AES-128 key for use with bulk encryption. Based on NIST security levels, Table 1: Comparable Algorithm Strengths (reading from the *FF* or finite field column), you need at least a 3072-bit prime modulus and a prime subgroup order of at least 256 bits. The 3072-bit modulus resists the sub-exponential attacks against discrete logarithms, while the 256-bit subgroup provides the necessary security against Pohlig-Hellman.

If you use a safe prime, you waste cycles since there will be approximately 3072 squares and multiplies. You cannot use a RFC 5114 MODP group since the RFC does not offer a '3072-bit MODP Group with 256-bit Prime Order Subgroup'. You would need to (1) move to elliptic curves and use RFC 5114's '256-bit Random ECP Group', (2) accept the wasted cycles, or (3) find a suitable alternative.

Additional reading can be found in many text books, and at least two sci.crypt threads apply: *DH Parameter Generation and Confinement Attacks (http://groups.google.com/group/sci.crypt/browse_thread/thread/7dc7eeb04a09f0ce)* and *RFC 5114 (Additional DH Groups...), 2048-bit MODP Group, and Safe Prime (http://groups.google.com/group/sci.crypt/browse_thread/thread/432c89ab3a917b5d)*.

Validating Parameters

Any time parameters are loaded from an outside source, the parameters **must** be validated. The parameters could have been poorly generated by the other party (for example, a random prime rather than a safe prime), or the parameters could be have been manipulated by an adversary (such as a middle man). If validation fails, immediately abort further processing.

Below, validation has been added to dh-init.exe. The call of interest is `ValidateGroup`, which takes a `RandomNumberGenerator` and `int` specifying a level. Validation can be performed on Crypto++ generated parameters and should be called on parameters received from a peer.

```
// RFC 5114, 1024-bit MODP Group with 160-bit Prime Order Subgroup
Integer p = ...
Integer g = ...
Integer q = ...

DH dh;
dh.AccessGroupParameters().Initialize(p, q, g);

if(!dh.GetGroupParameters().ValidateGroup(rnd, 3))
        throw runtime_error("Failed to validate prime and generator");

p = dh.GetGroupParameters().GetModulus();
cout << "P: " << p << endl;

q = dh.GetGroupParameters().GetSubgroupOrder();
cout << "Subgroup order: " << q << endl;

g = dh.GetGroupParameters().GetGenerator();
cout << "G: " << g << endl;
```

```
Integer v = ModularExponentiation(g, q, p);
if(v != Integer::One())
        throw runtime_error("Failed to verify order of the subgroup");
```

We perform one additional test: $g^q \equiv 1 \bmod p$, which is the call to `ModularExponentiation(g, q, p)`. The test verifies that *g* has order *q* since its not readily apparent that Crypto++ performs the test in `ValidateGroup`. See *DH Parameter Generation and Confinement Attacks (http://groups.google.com/group/sci.crypt/browse_thread/thread/7dc7eeb04a09f0ce)*.

When tracing validation over the integers, we land in `DL_GroupParameters_IntegerBased` from `gfpcrypt.cpp`.

```
bool DL_GroupParameters_IntegerBased::ValidateGroup(RandomNumberGenerator &rng, unsigned int level) const
{
    const Integer &p = GetModulus(), &q = GetSubgroupOrder();

    bool pass = true;
    pass = pass && p > Integer::One() && p.IsOdd();
    pass = pass && q > Integer::One() && q.IsOdd();

    if (level >= 1)
        pass = pass && GetCofactor() > Integer::One() && GetGroupOrder() % q == Integer::Zero();
    if (level >= 2)
        pass = pass && VerifyPrime(rng, q, level-2) && VerifyPrime(rng, p, level-2);

    return pass;
}
```

Finally, `VerifyPrime` from `nbtheory.cpp` is shown below. By default, Crypto++ calls Rabin-Miller 10 times. If the iteration count seems too low, call `RabinMillerTest` directly. Tracing the calls to `IsPrime` and `RabinMillerTest` to uncover the library's terminal tests are left as an exercise.

```
bool VerifyPrime(RandomNumberGenerator &rng, const Integer &p, unsigned int level)
{
    bool pass = IsPrime(p) && RabinMillerTest(rng, p, 1);
    if (level >= 1)
        pass = pass && RabinMillerTest(rng, p, 10);

    return pass;
}
```

Generating Keys (MODP)

With a properly initialized and validated `DH` object, the code to generate a key pair for exchange is trivial. Its so trivial that it does not have an associated download.

```
DH dh;
dh.AccessGroupParameters().Initialize(p, q, g);
...

SecByteBlock privKey(dh.PrivateKeyLength());
SecByteBlock pubKey(dh.PublicKeyLength());
dh.GenerateKeyPair(rnd, privKey, pubKey);
```

Its usually a bad idea to move private key material out from a `SecByteBlock`, but the code below will print the

keys for debugging purposes.

```
Integer a, b;

a.Decode(sharedA.BytePtr(), sharedA.SizeInBytes());
cout << "Shared secret (A): " << std::hex << a << endl;

b.Decode(sharedB.BytePtr(), sharedB.SizeInBytes());
cout << "Shared secret (B): " << std::hex << b << endl;
```

The code will produce an output similar to below.

```
$ ./dh-gen.exe
Private key: 0xE3C81A55665EE9C2243EC738C65ADA19C21282EF
Public key: 0x8AFE36AEFE1CC495843E8B5D255F2724A7B996D1AC3FD5DC9D31BA1ADCAB6AF8
37307322CF965C95F8FDF94A2A87F7E18759933C6B3E5CC6DDFF7576D423C2F8F5FD50FC51A2A6
7A859CF331B2094D243E2775CE3E1C30D3F993D4ED0848685A8785C4B98E0325E4D232A5383CD8
C64EC94115AD70169D25205D37CB45D18398
```

Key Agreement and Transport

After the previous examples, we are finally ready to perform key exchange and arrive at a **shared secret**. Though Diffie-Hellman is most often described as simply g^{ab} , the protocol (and Crypto++) uses a two part, asymmetric key. The private keys are the exponents a and b ; while g^a and g^b are the public keys.

There are two examples of key agreement available. The first uses classical unauthenticated Diffie-Hellman (sometimes referred to as DH1), and the second demonstrates authenticated Diffie-Hellman (referred to as DH2). Authenticated Diffie-Hellman is also known as Unified Diffie-Hellman. Classical Diffie-Hellman uses one key pair from each party during exchange; while unified Diffie-Hellman requires two key pairs from each party during agreement. In the unified model, one key pair is used for classical Diffie-Hellman and the second pair is used for authenticity assurances (that is, signatures).

The shared secret is sometimes called a **key encryption key** or **KEK**. This is because the shared secret is typically **not** used for bulk encryption. The KEK is used to transport the actual symmetric cipher key that will be used for bulk encryption. In discussions of Diffie-Hellman, the symmetric key to be transported by the KEK is sometimes called the **content encryption key** or **CEK**. Taking from RFC 2631, *Diffie-Hellman Key Agreement Method (http://tools.ietf.org/html/rfc2631)*:

> Diffie-Hellman key agreement requires that both the sender and recipient of a message have key pairs. By combining one's private key and the other party's public key, both parties can compute the same shared secret number. This number can then be converted into cryptographic keying material. That keying material is typically used as a key-encryption key (KEK) to encrypt (wrap) a content encryption key (CEK) which is in turn used to encrypt the message data.

Key Agreement (Unauthenticated)

The key exchange below derives a shared secret between the two parties, **A** and **B**, using the `DH` class. If the parties are under attack, three or more parties might be participating (and the original parties would be no wiser).

Each party calls `Agree`, which combines their private key with the other's public key. For those who referenced the library's test code `SimpleKeyAgreementValidate` in `validate2.cpp`, the code below should look familiar since its basically `SimpleKeyAgreementValidate`. Finally, The sample below is available in dh-agree.zip.

```
// RFC 5114, 1024-bit MODP Group with 160-bit Prime Order Subgroup
Integer p = ...
Integer g = ...
Integer q = ...

DH dhA, dhB;
AutoSeededRandomPool rndA, rndB;

dhA.AccessGroupParameters().Initialize(p, q, g);
dhB.AccessGroupParameters().Initialize(p, q, g);

if(!dhA.GetGroupParameters().ValidateGroup(rndA, 3) ||
   !dhB.GetGroupParameters().ValidateGroup(rndB, 3))
        throw runtime_error("Failed to validate prime and generator");
...

////////////////////////////////////////////////////////////

SecByteBlock privA(dhA.PrivateKeyLength());
SecByteBlock pubA(dhA.PublicKeyLength());
dhA.GenerateKeyPair(rndA, privA, pubA);

SecByteBlock privB(dhB.PrivateKeyLength());
SecByteBlock pubB(dhB.PublicKeyLength());
dhB.GenerateKeyPair(rndB, privB, pubB);

////////////////////////////////////////////////////////////

SecByteBlock sharedA(dhA.AgreedValueLength());
SecByteBlock sharedB(dhB.AgreedValueLength());

if(dhA.AgreedValueLength() != dhB.AgreedValueLength())
        throw runtime_error("Shared secret size mismatch");

if(!dhA.Agree(sharedA, privA, pubB))
        throw runtime_error("Failed to reach shared secret (1)");

if(!dhB.Agree(sharedB, privB, pubA))
        throw runtime_error("Failed to reach shared secret (2)");

count = std::min(dhA.AgreedValueLength(), dhB.AgreedValueLength());
if(!count || !VerifyBufsEqual(sharedA.BytePtr(), sharedB.BytePtr(), count))
        throw runtime_error("Failed to reach shared secret (3)");
```

In production, the test of `sharedA` and `sharedB` cannot be performed since the values will be on different hosts. A problem with agreement will not be detected until data starts flowing and fails to authenticate. This can be remedied with a key confirmation (http://security.stackexchange.com/q/45143/29925) protocol.

A typical run of dh-agree.exe is shown below.

```
$ ./dh-agree.exe
Shared secret (A): 2D72B992E9AAB76E95ED591C203438B24888AED245C2DD5ECBFB24356E2
C26D45BEAB9A59763A4819BEA620D251AA321F9EA6E4F6EE0A7312026755E3F5BD2EC4E1D4BED4
177D3215BA446BE31B6F6A28160FE053A441B72B7EEE3CBA23945AC477B51F93E82FDD7AA4BB96
```

```
0A537BEF22F30F11B875FB88446B87B414D679905
Shared secret (B): 2D72B992E9AAB76E95ED591C203438B24888AED245C2DD5ECBFB24356E2
C26D45BEAB9A59763A4819BEA620D251AA321F9EA6E4F6EE0A7312026755E3F5BD2EC4E1D4BED4
177D3215BA446BE31B6F6A28160FE053A441B72B7EEE3CBA23945AC477B51F93E82FDD7AA4BB96
0A537BEF22F30F11B875FB88446B87B414D679905
```

The code above also uses VerifyBufsEqual, which provides a constant time memory comparison to avoid leaking timing information.

Key Agreement (Authenticated)

Unified Diffie-Hellman, sometimes known as Ephemeral Unified Model Scheme (X9.63) or Static Unified Model (NIST), derives a shared secret between the two parties, **A** and **B**, using the `DH2` class. Tampering with the exchange will be detected by the parties via authenticity assurances on the exchanged parameters.

Details of Unified Diffie-Hellman can be found in IEEE's P1363, ANSI X9.63, and NIST's SP 800-56. In the unified model, each party holds two key pairs: one key pair is used to perform classical (unautheticated) Diffie-Hellman, and the second pair is used for signing to ensure authenticity. The classical key pair is called ephemeral in Unified Diffie-Hellman since it is a temporary key pair used only for the current exchange. The signing key pair is the static pair. The public portion of the signing key can be published in a common directory for convenient access since the signing key pair changes infrequently.

Each party calls `Agree`, which combines their private keys with the other's publics keys. For those who referenced the library's test code `AuthenticatedKeyAgreementValidate` in `validate2.cpp`, the code below should look familiar since its basically `AuthenticatedKeyAgreementValidate`. Finally, The sample below is available in dh-unified.zip.

```
// RFC 5114, 1024-bit MODP Group with 160-bit Prime Order Subgroup
Integer p = ...
Integer g = ...
Integer q = ...

DH dh;
AutoSeededRandomPool rnd;

dh.AccessGroupParameters().Initialize(p, q, g);
...

//////////////////////////////////////////////////////////

DH2 dhA(dh), dhB(dh);

SecByteBlock sprivA(dhA.StaticPrivateKeyLength()), spubA(dhA.StaticPublicKeyLength());
SecByteBlock eprivA(dhA.EphemeralPrivateKeyLength()), epubA(dhA.EphemeralPublicKeyLength());

SecByteBlock sprivB(dhB.StaticPrivateKeyLength()), spubB(dhB.StaticPublicKeyLength());
SecByteBlock eprivB(dhB.EphemeralPrivateKeyLength()), epubB(dhB.EphemeralPublicKeyLength());

dhA.GenerateStaticKeyPair(rnd, sprivA, spubA);
dhA.GenerateEphemeralKeyPair(rnd, eprivA, epubA);

dhB.GenerateStaticKeyPair(rnd, sprivB, spubB);
dhB.GenerateEphemeralKeyPair(rnd, eprivB, epubB);

//////////////////////////////////////////////////////////

if(dhA.AgreedValueLength() != dhB.AgreedValueLength())
        throw runtime_error("Shared secret size mismatch");
```

```
SecByteBlock sharedA(dhA.AgreedValueLength()), sharedB(dhB.AgreedValueLength());

if(!dhA.Agree(sharedA, sprivA, eprivA, spubB, epubB))
        throw runtime_error("Failed to reach shared secret (A)");

if(!dhB.Agree(sharedB, sprivB, eprivB, spubA, epubA))
        throw runtime_error("Failed to reach shared secret (B)");

count = std::min(dhA.AgreedValueLength(), dhB.AgreedValueLength());
if(!count || !VerifyBufsEqualp(sharedA.BytePtr(), sharedB.BytePtr(), count))
        throw runtime_error("Failed to reach shared secret");
```

In production, the test in `VerifyBufsEqualp` cannot be performed since the values will be on different hosts. A problem with agreement will not be detected until data starts flowing - the first data packet received with not authenticate.

A typical run of dh-unified.exe is shown below.

```
$ ./dh-unified.exe
Shared secret (A): 816e1f08f64537551a556eae679936319da1be7e1baf13ffe11b5cefd78
de1dc72fcaff539b5d77813ac5231e31dde53806ab99349f82571dcea8ee3e2502f0ee2246b541
3fa5e9bf68040a609506880aab2b76ca97f8a6d9fa23578867acd04793b30b277c69e57e7af20c
60a47d90127351c1f62f6d0ff5c91ea852aedd26a63fed9ce098f99eceecae1083ce1b35778614
8be9acb3476a9f7139e4ac87e6624e76cd4676d59460477033adaa0d349a001de31c9671a3afa9
c84311af1198f6676fd89852fb493c8a6c3b03529705aae389752bcef94ee1b0fe69edddda7d2d
4fdbfe0d3b4a174a93cb8a4011d9c974da7c5cb89f60ee200abc3d18f7bc965h
Shared secret (B): 816e1f08f64537551a556eae679936319da1be7e1baf13ffe11b5cefd78
de1dc72fcaff539b5d77813ac5231e31dde53806ab99349f82571dcea8ee3e2502f0ee2246b541
3fa5e9bf68040a609506880aab2b76ca97f8a6d9fa23578867acd04793b30b277c69e57e7af20c
60a47d90127351c1f62f6d0ff5c91ea852aedd26a63fed9ce098f99eceecae1083ce1b35778614
8be9acb3476a9f7139e4ac87e6624e76cd4676d59460477033adaa0d349a001de31c9671a3afa9
c84311af1198f6676fd89852fb493c8a6c3b03529705aae389752bcef94ee1b0fe69edddda7d2d
4fdbfe0d3b4a174a93cb8a4011d9c974da7c5cb89f60ee200abc3d18f7bc965h
```

Key Transport

With a key encryption key (KEK) in hand, we can now transport a content encryption key (CEK). The CEK will handle bulk encryption over the secure channel for the remainder of the session.

```
// Take the leftmost 'n' bits for the KEK
SecByteBlock kek(sharedA.BytePtr(), AES::DEFAULT_KEYLENGTH);

// CMAC key follows the 'n' bits used for KEK
SecByteBlock mack(&sharedA.BytePtr()[AES::DEFAULT_KEYLENGTH], AES::BLOCKSIZE);
CMAC<AES> cmac(mack.BytePtr(), mack.SizeInBytes());

// Generate a random CEK
SecByteBlock cek(AES::DEFAULT_KEYLENGTH);
rnd.GenerateBlock(cek.BytePtr(), cek.SizeInBytes());

// AES in ECB mode is fine - we're encrypting 1 block, so we don't need padding
ECB_Mode<AES>::Encryption aes;
aes.SetKey(kek.BytePtr(), kek.SizeInBytes());

// Will hold the encrypted key and cmac
SecByteBlock xport(AES::BLOCKSIZE /*ENC(CEK)*/ + AES::BLOCKSIZE /*CMAC*/);
byte* const ptr = xport.BytePtr();

// Write the encrypted key in the first 16 bytes, and the CMAC in the second 16 bytes
// The logical layout of xport:
//   [    Enc(CEK)    ][  CMAC(Enc(CEK))  ]
```

```
aes.ProcessData(&ptr[0], cek.BytePtr(), AES::BLOCKSIZE);
cmac.CalculateTruncatedDigest(&ptr[AES::BLOCKSIZE], AES::BLOCKSIZE, &ptr[0], AES::BLOCKSIZE);
```

A typical run of dh-xport.exe is shown below.

```
$ ./dh-xport.exe
Key encryption key: a0ba6fd018ee91e04dcc2420089d6eda
Content encryption key: 6a38e1f9db305ac3c7acbbf380de5174
Encrypted CEK with CMAC: 334546cbaba10f603ff7f8ac0d99aaaf1734f79170221add0296c643fb440427
```

Previous Wiki Page

Generating a secret key

The following example illustrates how two hosts, Alice and Bob, agree on a shared secret key using Diffie-Hellman key exchange.

Alice generates a prime and base which she shares with Bob. (The prime and base can also simply be hard coded into the application.) Alice then generates a pair of public and private integers. The public part is shared with Bob.

Bob uses the prime and base that he received from Alice to generate a pair of public and private integers. The public part is shared with Alice.

```
////////////////////////////////////////////////////////////////////////
// Alice

// Initialize the Diffie-Hellman class with a random prime and base
AutoSeededRandomPool rngA;
DH dhA;
dh.Initialize(rngA, 128);

// Extract the prime and base. These values could also have been hard coded
// in the application
Integer iPrime = dhA.GetGroupParameters().GetModulus();
Integer iGenerator = dhA.GetGroupParameters().GetSubgroupGenerator();

SecByteBlock privA(dhA.PrivateKeyLength());
SecByteBlock pubA(dhA.PublicKeyLength());
SecByteBlock secretKeyA(dhA.AgreedValueLength());

// Generate a pair of integers for Alice. The public integer is forwarded to Bob.
dhA.GenerateKeyPair(rngA, privA, pubA);

////////////////////////////////////////////////////////////////////////
// Bob

AutoSeededRandomPool rngB;
// Initialize the Diffie-Hellman class with the prime and base that Alice generated.
DH dhB(iPrime, iGenerator);

SecByteBlock privB(dhB.PrivateKeyLength());
SecByteBlock pubB(dhB.PublicKeyLength());
SecByteBlock secretKeyB(dhB.AgreedValueLength());

// Generate a pair of integers for Bob. The public integer is forwarded to Alice.
dhB.GenerateKeyPair(rngB, privB, pubB);

////////////////////////////////////////////////////////////////////////
```

```
// Agreement

// Alice calculates the secret key based on her private integer as well as the
// public integer she received from Bob.
if (!dhA.Agree(secretKeyA, privA, pubB))
        return false;

// Bob calculates the secret key based on his private integer as well as the
// public integer he received from Alice.
if (!dhB.Agree(secretKeyB, privB, pubA))
        return false;

// Just a validation check. Did Alice and Bob agree on the same secret key?
if (VerifyBufsEqualp(secretKeyA.begin(), secretKeyB.begin(), dhA.AgreedValueLength()))
        return false;

return true;
```

Using Diffie-Hellman to generate an AES key

Building on the previous example. Note that the example uses in-place encryption and decryption where the input and output buffers are identical:

```
int aesKeyLength = SHA256::DIGESTSIZE; // 32 bytes = 256 bit key
int defBlockSize = AES::BLOCKSIZE;

// Calculate a SHA-256 hash over the Diffie-Hellman session key
SecByteBlock key(SHA256::DIGESTSIZE);
SHA256().CalculateDigest(key, secretKeyA, secretKeyA.size());

// Generate a random IV
byte iv[AES::BLOCKSIZE];
arngA.GenerateBlock(iv, AES::BLOCKSIZE);

char message[] = "Hello! How are you.";
int messageLen = (int)strlen(plainText) + 1;

////////////////////////////////////////////////////////////////////////
// Encrypt

CFB_Mode<AES>::Encryption cfbEncryption(key, aesKeyLength, iv);
cfbEncryption.ProcessData((byte*)message, (byte*)message, messageLen);

////////////////////////////////////////////////////////////////////////
// Decrypt

CFB_Mode<AES>::Decryption cfbDecryption(key, aesKeyLength, iv);
cfbDecryption.ProcessData((byte*)message, (byte*)message, messageLen);
```

Downloads

dh-param.zip - Diffie-Hellman parameter generation and validation.

dh-init.zip - Diffie-Hellman parameter initialization and validation.

dh-gen.zip - Diffie-Hellman public and private key generation.

dh-agree.zip - Diffie-Hellman key agreement (unauthenticated).

dh-unified.zip - Unified Diffie-Hellman key agreement (authenticated).

dh-xport.zip - Diffie-Hellman key transport (unauthenticated).
