Botan 1.4.x Tutorial

Jack Lloyd
lloyd@randombit.net

Contents

1 Introduction

This document essentially sets up various simple scenarios and then shows how to solve the problems using Botan 1.4.x. It's fairly simple, and doesn't cover many of the available APIs and algorithms, especially the more obscure or unusual ones. It is a supplement to the API documentation and the example applications, which are included in the distribution.

To quote the Perl man page: '"There's more than one way to do it." Divining how many more is left as an exercise to the reader.'

This is *not* a general introduction to cryptography, and most simple terms and ideas are not explained in any great detail.

Finally, most of the code shown in this tutorial has not been tested, it was just written down from memory. If you find errors, please let me know.

2 Initializing the Library

The first step to using Botan is the create a `LibraryInitializer` object, which handles creating various internal structures, and also destroying them at shutdown. Essentially:

```
#include <botan/botan.h>
/* include other headers here */

int main()
   {
   LibraryInitializer init;
   /* now do stuff */
   return 0;
   }
```

3 Symmetric Cryptography

3.1 Encryption with a passphrase

Probably the most common crypto problem is encrypting a file (or some data that is in-memory) using a passphrase. There are a million ways to do this, most of them bad. In particular, you've have to protect against weak passphrases, people reusing a passphrase many times, accidental and deliberate modification, and a dozen other potential problems.

We'll start with a simple method that is commonly used, and show the problems that can arise. Each subsequent solution will modify the previous one to prevent one or more common problems, until we arrive at a good version.

In these examples, we'll always use Blowfish in CBC mode. Blowfish has been around almost 10 years at this point, and is well known and trusted. The main reason for choosing Blowfish over, say, TripleDES, is because Blowfish supports nearly arbitrary key lengths, allowing us to easily try many different ways of generating the keys. For production code, another algorithm (such as TripleDES or AES) may be more appropriate. Whenever we need a hash function, we'll use SHA-1, since that is a common and well-known hash that is thought to be secure.

In all examples, we choose to derive the IV from the passphrase. Another (probably more common) alternative is to generate the IV randomly and include it at the beginning of the message. Either way is

acceptable, and can be secure. The method used here was chosen to make for more interesting examples (because it's harder to get right), and may not be an appropriate choice for some environments.

First, some notation. The passphrase is stored as a `std::string` named *passphrase*. The input and output files (*infile* and *outfile*) are of types `std::ifstream` and `std::ofstream` (respectively).

3.1.1 First try

We hash the passphrase with SHA-1, and use the resulting hash to key Blowfish. To generate the IV, we prepend a single '0' character to the passphrase, hash it, and truncate it to 8 bytes (which is Blowfish's block size).

```
HashFunction* hash = get_hash("SHA-1");

SymmetricKey key = hash->process(passphrase);
SecureVector<byte> raw_iv = hash->process('0' + passphrase);
InitializationVector iv(raw_iv, 8);

Pipe pipe(get_cipher("Blowfish/CBC/PKCS7", key, iv, ENCRYPTION));

pipe.start_msg();
infile >> pipe;
pipe.end_msg();
outfile << pipe;
```

3.1.2 Problem 1: Buffering

There is a problem with the above code, if the input file is fairly large as compared to available memory. Specifically, all of the encrypted data is stored in memory, and then flushed to *outfile* in a single go at the very end. If the input file is big (say, a gigabyte), this will be most problematic.

The solution is to use a `DataSink` to handle the output for us (writing to *outfile* will be implicit with writing to the `Pipe`). We can do this by replacing the last few lines with:

```
Pipe pipe(get_cipher("Blowfish/CBC/PKCS7", key, iv, ENCRYPTION),
          new DataSink_Stream(outfile));

pipe.start_msg();
infile >> pipe;
pipe.end_msg();
```

3.1.3 Problem 2: Deriving the key and IV

Hash functions like SHA-1 are deterministic; if the same passphrase is supplied twice, then the key (and in our case, the IV) will be the same. This is very dangerous, and could easily open the whole system up to attack. What we need to do is introduce a salt (or nonce) into the generation of the key from the passphrase. This will mean that the key will not be the same each time the same passphrase is typed in by a user.

There is another problem with using a bare hash function to derive keys. While it's inconceivable that an attacker could brute-force a 160-bit key, it would be fairly simple for them to compute the SHA-1 hashes of various common passwords ('password', the name of the dog, SO's middle name, etc) and try those as keys. So we want to slow the attacker down if we can, and an easy way to do that is to iterate the hash function a bunch of times (say, 1024 to 4096 times). This will involve only a small amount of effort for a

legitimate user (since they only have to compute the hashes once, when they type in their passphrase), but an attacker, trying out a large list of potential passphrases, will be seriously annoyed by this.

In this iteration of the example, we'll kill these two birds with one stone, and derive the key from the passphrase using a S2K (string to key) algorithm (these are also often called PBKDF algorithms, for Password Based Key Derivation Function). In this example, we use PBKDF2 with HMAC(SHA-1), which is specified in PKCS #5. We replace the first four lines of code from the first example with:

```
S2K* s2k = get_s2k("PBKDF2(SHA-1)");
// hard-coded iteration count for simplicity; should be sufficient
s2k->set_iterations(4096);
// 8 octets == 64 bit salt; again, good enough
s2k->new_random_salt(8);
SecureVector<byte> the_salt = s2k->current_salt();

// 28 octets == 20 for key + 8 for IV
SecureVector<byte> key_and_IV = s2k->derive_key(28, passphrase);

SymmetricKey key(key_and_IV, 20);
InitializationVector iv(key_and_IV + 20, 8);
```

To complete the example, we have to remember to write out the salt (stored in *the_salt*) at the beginning of the file. The receiving side needs to know this value in order to restore it (by calling the *s2k* object's **change_salt** function) so it can derive the same key and IV from the passphrase.

3.1.4 Problem 3: Protecting against modification

As it is, an attacker can undetectably alter the message while it is in transit. It is vital to remember that encryption does not imply authentication (except when using special modes which are specifically designed to provide authentication along with encryption, like OCB and EAX). For this purpose, we will append a message authentication code to the encrypted message. Specifically, we will generate an extra 160 bits of key data, and use it to key the "HMAC(SHA-1)" MAC function. We don't want to have the MAC and the cipher to share the same key; that is very much a no-no.

```
// 48 octets == 20 for blowfish key + 8 for IV + 20 for hmac key
SecureVector<byte> keys_and_IV = s2k->derive_key(48, passphrase);

SymmetricKey key(keys_and_IV, 20);
InitializationVector iv(keys_and_IV + 20, 8);
SymmetricKey mac_key(keys_and_IV + 28, 20);

Pipe pipe(new Fork(
             new Chain(
                get_cipher("Blowfish/CBC/PKCS7", key, iv, ENCRYPTION),
                new DataSink_Stream(outfile)
                ),
             new MAC_Filter("HMAC(SHA-1)", mac_key)
             )
   );

pipe.start_msg();
infile >> pipe;
pipe.end_msg();
```

```
// now read the MAC from message #2. Message numbers start from 0
SecureVector<byte> hmac = pipe.read_all(1);
outfile.write((const char*)hmac.ptr(), hmac.size());
```

The receiver can check the size of the file (in bytes), and since it knows how long the MAC is, can figure out how many bytes of ciphertext there are. Then it reads in that many bytes, sending them to a Blowfish/CBC decryption object (which could be obtained by calling `get_cipher` with an argument of `DECRYPTION` instead of `ENCRYPTION`), and storing the final bytes to authenticate the message with.

3.1.5 Problem 4: Cleaning up the key generation

The method used to derive the keys and IV is rather inelegant, and it would be nice to clean that up a bit, algorithmically speaking. A nice solution for this is to generate a master key from the passphrase and salt, and then generate the two keys and the IV (the cryptovariables) from that.

Starting from the master key, we derive the cryptovariables using a KDF algorithm, which is designed, among other things, to "separate" keys so that we can derive several different keys from the single master key. For this purpose, we will use KDF2, which is a generally useful KDF function (defined in IEEE 1363a, among other standards). The use of different labels ("cipher key", etc) makes sure that each of the three derived variables will have different values.

```
S2K* s2k = get_s2k("PBKDF2(SHA-1)");
// hard-coded iteration count for simplicity; should be sufficient
s2k->set_iterations(4096);
// 8 octet == 64 bit salt; again, good enough
s2k->new_random_salt(8);
// store the salt so we can write it to a file later
SecureVector<byte> the_salt = s2k->current_salt();

SymmetricKey master_key = s2k->derive_key(48, passphrase);

KDF* kdf = get_kdf("KDF2(SHA-1)");

SymmetricKey key = kdf->derive_key(20, master_key, "cipher key");
SymmetricKey mac_key = kdf->derive_key(20, master_key, "hmac key");
InitializationVector iv = kdf->derive_key(8, master_key, "cipher iv");
```

3.1.6 Final version

Here is the final version of the encryption code, with all of the changes we've made:

```
S2K* s2k = get_s2k("PBKDF2(SHA-1)");
s2k->set_iterations(4096);
s2k->new_random_salt(8);
SecureVector<byte> the_salt = s2k->current_salt();

SymmetricKey master_key = s2k->derive_key(48, passphrase);

KDF* kdf = get_kdf("KDF2(SHA-1)");

SymmetricKey key = kdf->derive_key(20, master_key, "cipher key");
SymmetricKey mac_key = kdf->derive_key(20, masterkey, "hmac key");
InitializationVector iv = kdf->derive_key(8, masterkey, "cipher iv");
```

```
Pipe pipe(new Fork(
             new Chain(
                get_cipher("Blowfish/CBC/PKCS7", key, iv, ENCRYPTION),
                new DataSink_Stream(outfile)
                ),
             new MAC_Filter("HMAC(SHA-1)", mac_key)
             )
   );

outfile.write((const char*)the_salt.ptr(), the_salt.size());

pipe.start_msg();
infile >> pipe;
pipe.end_msg();

SecureVector<byte> hmac = pipe.read_all(1);
outfile.write((const char*)hmac.ptr(), hmac.size());
```

3.1.7 Another buffering technique

Sometimes the use of `DataSink_Stream` is not practical for whatever reason. In this case, an alternate buffering mechanism might be useful. Here is some code which will write all the processed data as quickly as possible, so that memory pressure is reduced in the case of large inputs.

```
pipe.start_msg();
SecureBuffer<byte, 1024> buffer;
while(infile.good())
   {
   infile.read((char*)buffer.ptr(), buffer.size());
   u32bit got_from_infile = infile.gcount();
   pipe.write(buffer, got_from_infile);

   if(infile.eof())
      pipe.end_msg();

   while(pipe.remaining() > 0)
      {
      u32bit buffered = pipe.read(buffer, buffer.size());
      outfile.write((const char*)buffer.ptr(), buffered);
      }
   }
if(infile.bad() || (infile.fail() && !infile.eof()))
   throw Some_Exception();
```

3.2 Authentication

After doing the encryption routines, doing message authentication keyed off a passphrase is not very difficult. In fact it's much easier than the encryption case, for the following reasons: a) we only need one key, and b) we don't have to store anything, so all the input can be done in a single step without worrying about it taking up a lot of memory if the input file is large.

In this case, we'll hex-encode the salt and the MAC, and output them both to standard output (the salt followed by the MAC).

```
S2K* s2k = get_s2k("PBKDF2(SHA-1)");
s2k->set_iterations(4096);
s2k->new_random_salt(8);
OctetString the_salt = s2k->current_salt();

SymmetricKey hmac_key = s2k->derive_key(20, passphrase);

Pipe pipe(new MAC_Filter("HMAC(SHA-1)", mac_key),
          new Hex_Encoder
);

std::cout << the_salt.to_string(); // hex encoded

pipe.start_msg();
infile >> pipe;
pipe.end_msg();
std::cout << pipe.read_all_as_string() << std::endl;
```

3.3 User Authentication

Doing user authentication off a shared passphrase is fairly easy. Essentially, a challenge-response protocol is used - the server sends a random challenge, and the client responds with an appropriate response to the challenge. The idea is that only someone who knows the passphrase can generate or check to see if a response is valid.

Let's say we use 160 bit (20 byte) challenges, which seems fairly reasonable. We can create this challenge using the global RNG:

```
byte challenge[20];
Global_RNG::randomize(challenge, sizeof(challenge), Nonce);
// send challenge to client
```

After reading the challenge, the client generates a response based on the challenge and the passphrase. In this case, we will do it by repeatedly hashing the challenge, the passphrase, and (if applicable) the previous digest. We iterate this construction 4096 times, to make brute force attacks on the passphrase hard to do. Since we are already using 160 bit challenges, a 160 bit response seems warranted, so we'll use SHA-1.

```
HashFunction* hash = get_hash("SHA-1");
SecureVector<byte> digest;
for(u32bit j = 0; j != 4096; j++)
   {
   hash->update(digest, digest.size());
   hash->update(passphrase);
   hash->update(challenge, challenge.size());
```

```
   digest = hash->final();
   }
delete hash;
// send value of digest to the server
```

Upon receiving the response from the client, the server computes what the response should have been based on the challenge it sent out, and the passphrase. If the two responses match, the the client is authenticated. Otherwise, it is not.

An alternate method is to use PBKDF2 again, using the challenge as the salt. In this case, the response could (for example) be the hash of the key produced by PBKDF2. There is no reason to have an explicit iteration loop, as PBKDF2 is designed to prevent dictionary attacks (assuming PBKDF2 is set up for a large iteration count internally).

4 Public Key Cryptography

4.1 Basic Operations

In this section, we'll assume we have a `X509_PublicKey*` named *pubkey*, and, if necessary, a private key type (a `PKCS8_PrivateKey*`) named *privkey*. A description of these types, how to create them, and related details appears later in this tutorial. In this section, we will use various functions which are defined in `look_pk.h` – you will have to include this header explicitly.

4.1.1 Encryption

Basically, pick an encoding method, create a `PK_Encryptor` (with **get_pk_encryptor**()), and use it. But first we have to make sure the public key can actually be used for public key encryption. For encryption (and decryption), the key could be RSA, ElGamal, or (in future versions) some other public key encryption scheme, like Rabin or an elliptic curve scheme.

```
PK_Encrypting_Key* key = dynamic_cast<PK_Encrypting_Key*>(pubkey);
if(!key)
   error();
PK_Encryptor* enc = get_pk_encryptor(*key, "EME1(SHA-1)");

byte msg[] = { /* ... */ };

// will also accept a SecureVector<byte> as input
SecureVector<byte> ciphertext = enc->encrypt(msg, sizeof(msg));
```

4.1.2 Decryption

This is essentially the same as the encryption operation, but using a private key instead. One major difference is that the decryption operation can fail due to the fact that the ciphertext was invalid (most common padding schemes, such as "EME1(SHA-1)", include various pieces of redundancy, which are checked after decryption).

```
PK_Decrypting_Key* key = dynamic_cast<PK_Decrypting_Key*>(privkey);
if(!key)
   error();
PK_Decryptor* dec = get_pk_decryptor(*key, "EME1(SHA-1)");

byte msg[] = { /* ... */ };

SecureVector<byte> plaintext;

try {
  // will also accept a SecureVector<byte> as input
  plaintext = dec->decrypt(msg, sizeof(msg));
}
catch(Decoding_Error)
{
  /* the ciphertext was invalid */
}
```

4.1.3 Signature Generation

There is one difficulty with signature generation that does not occur with encryption or decryption. Specifically, there are various padding methods which can be useful for different signature algorithms, and not all are appropriate for all signature schemes. The following table breaks down what algorithms support which encodings:

Signature Algorithm	Usable Encoding Methods	Preferred Encoding(s)
DSA / NR	EMSA1	EMSA1
RSA	EMSA1, EMSA2, EMSA3, EMSA4	EMSA3, EMSA4
Rabin-Williams	EMSA2, EMSA4	EMSA2, EMSA4

For new applications, use EMSA4 with both RSA and Rabin-Williams, as it is significantly more secure than the alternatives. However, most current applications/libraries only support EMSA2 with Rabin-Williams and EMSA3 with RSA. Given this, you may be forced to use less secure encoding methods for the near future. In these examples, we punt on the problem, and hard-code using EMSA1 with SHA-1.

```
PK_Signing_Key* key = dynamic_cast<PK_Signing_Key*>(privkey);
if(!key)
   error();
PK_Signer* signer = get_pk_signer(*key, "EMSA1(SHA-1)");

byte msg[] = { /* ... */ };

/*
   You can also repeatedly call update(const byte[], u32bit), followed by a
   call to signature(), which will return the final sig of all the date that
   was passed through update(). sign_message() is just a stub that calls
   update() once, and returns the value of signature().
*/
SecureVector<byte> signature = signer->sign_message(msg, sizeof(msg));
```

4.1.4 Signature Verification

In addition to all of the problems with choosing the correct padding method, there is yet another complication with verifying a signature. Namely, there are two varieties of signature algorithms - those providing message recovery (that is, the value that was signed can be directly recovered by someone verifying the signature), and those without message recovery (the verify operation simply returns if the signature was valid, without telling you exactly what was signed). This leads to two slightly different implementations of the verification operation, which user code has to work with. As you can see however, the implementation is still not at all difficult.

```
PK_Verifier* verifier = 0;

PK_Verifying_with_MR_Key* key1 =
   dynamic_cast<PK_Verifying_with_MR_Key*>(pubkey);
PK_Verifying_wo_MR_Key* key2 =
   dynamic_cast<PK_Verifying_wo_MR_Key*>(pubkey);

if(key1)
   verifier = get_pk_verifier(*key1, "EMSA1(SHA-1)");
else if(key2)
   verifier = get_pk_verifier(*key2, "EMSA1(SHA-1)");
else
   error();

byte msg[] = { /* ... */ };
byte sig[] = { /* ... */ };

/*
   Like PK_Signer, you can also do repeated calls to
      void update(const byte some_data[], u32bit length)
   followed by a call to
      bool check_signature(const byte the_sig[], u32bit length)
   which will return true (valid signature) or false (bad signature).
   The function verify_message() is a simple wrapper around update() and
   check_signature().

*/
bool is_valid = verifier->verify_message(msg, sizeof(msg), sig, sizeof(sig));
```

4.1.5 Key Agreement

WRITEME

4.2 Working with Keys

4.2.1 Reading Public Keys (X.509 format)

There are two separate ways to read X.509 public keys. Remember that the X.509 public keys are simply that: public keys. There is no associated information (such as the owner of that key) included with the public key itself. If you need that kind of information, you'll need to use X.509 certificates.

However, there are surely times when a simple public key is sufficient. The most obvious is when the key is implicitly trusted, for example if access and/or modification of it is controlled by something else (like filesystem ACLs). In other cases, it is a perfectly reasonable proposition to use them over the network as an anonymous key exchange mechanism. This is, admittedly, vulnerable to man-in-the-middle attacks, but it's simple enough that it's hard to mess up (see, for example, Peter Guttman's paper "Lessons Learned in Implementing and Deploying Crypto Software" in Usenix '02).

The way to load a key is basically to set up a `DataSource` and then call **X509::load_key**, which will return a `X509_PublicKey*`. For example:

```
DataSource_Stream somefile("somefile.pem"); // has 3 public keys
X509_PublicKey* key1 = X509::load_key(somefile);
X509_PublicKey* key2 = X509::load_key(somefile);
X509_PublicKey* key3 = X509::load_key(somefile);
// Now we have them all loaded. Huzah!
```

At this point you can use **dynamic_cast** to find the operations the key supports (by seeing if a cast to `PK_Encrypting_Key`, `PK_Verifying_with_MR_Key`, or `PK_Verifying_wo_MR_Key` succeeds).

There is a variant of **X509::load_key** (and of **PKCS8::load_key**, described in the next section) which take a filename (as a `std::string`). These are just convenience functions which create the appropriate `DataSource` for you and call the main **X509::load_key**.

4.2.2 Reading Private Keys (PKCS #8 format)

This is very similar to reading raw public keys, with the difference that the key may be encrypted with a user passphrase:

```
DataSource_Stream somefile("somefile");
std::string a_passphrase = /* get from the user */
PKCS8_PrivateKey* key = PKCS8::load_key(somefile, a_passphrase);
```

You can, by the way, convert a `PKCS8_PrivateKey` to a `X509_PublicKey` simply by casting it (with **dynamic_cast**), as the private key type is derived from `X509_PublicKey`. As with `X509_PublicKey`, you can use **dynamic_cast** to figure out what operations the private key is capable of; in particular, you can attempt to cast it to `PK_Decrypting_Key`, `PK_Signing_Key`, or `PK_Key_Agreement_Key`.

Sometimes you can get away with having a static passphrase passed to **load_key**. Typically, however, you'll have to do some user interaction to get the appropriate passphrase. In that case you'll want to use the the `UI` related interface, which is fully described in the API documentation.

4.2.3 Generating New Private Keys

Generate a new private key is the one operation which requires you to explicitly name the type of key you are working with. There are (currently) two kinds of public key algorithms in Botan: ones based on the integer factorization (IF) problem (RSA and Rabin-Williams), and ones based on the discrete logarithm (DL) problem (DSA, Diffie-Hellman, Nyberg-Rueppel, and ElGamal). Since discrete logarithm parameters

(primes and generators) can be shared among many keys, there is the notion of these being a combined type (called `DL_Group`).

To create a new DL-based private key, simply pass a desired `DL_Group` to the constructor of the private key - a new public/private key pair will be generated. Since in IF-based algorithms, the modulus used isn't shared by other keys, we don't use this notion. You can create a new key by passing in a `u32bit` telling how long (in bits) the key should be.

There are quite a few ways to get a `DL_Group` object. The best is to use the function **get_dl_group**, which takes a string naming a group; it will either return that group, if it knows about it, or throw an exception. Names it knows about include "IETF-n" where n is 768, 1024, 1536, 2048, 3072, or 4096, and "DSA-n", where n is 512, 768, or 1024. The IETF groups are the ones specified for use with IPSec, and the DSA ones are the default DSA parameters specified by Java's JCE. For DSA and Nyberg-Rueppel, use the "DSA-n" groups, and for Diffie-Hellman and ElGamal, use the "IETF-n" groups.

You can also generate a new random group. This is not recommend, because it is very slow, particularly for "safe" primes, which are needed for Diffie-Hellman and ElGamal.

Some examples:

```
RSA_PrivateKey rsa1(512); // 512-bit RSA key
RSA_PrivateKey rsa2(2048); // 2048-bit RSA key

RW_PrivateKey rw1(1024); // 1024-bit Rabin-Williams key
RW_PrivateKey rw2(1536); // 1536-bit Rabin-Williams key

DSA_PrivateKey dsa(get_dl_group("DSA-512")); // 512-bit DSA key
DH_PrivateKey dh(get_dl_group("IETF-4096")); // 4096-bit DH key
NR_PrivateKey nr(get_dl_group("DSA-1024")); // 1024-bit NR key
ElGamal_PrivateKey elg(get_dl_group("IETF-1536")); // 1536-bit ElGamal key
```

To export your newly created private key, use the PKCS #8 routines in `pkcs8.h`:

```
std::string a_passphrase = /* get from the user */
std::string the_key = PKCS8::PEM_encode(rsa2, a_passphrase);
```

You can read the key back in using **PKCS8::load_key**, described in the section "Reading Private Keys (PKCS #8 format)", above. Unfortunately, this only works with keys that have an assigned algorithm identifier and standardized format. Currently this is only the RSA, DSA, DH, and ElGamal algorithms, though RW and NR keys can also be imported and exported by assigning them an OID (this can be done either through a configuration file, or by calling the function **OIDS::add_oid** in `oids.h`). Be aware that the OID and format for ElGamal keys is not exactly standard, but there does exist at least one other crypto library which will accept the format.

The raw public can be exported using:

```
std::string the_public_key = X509::PEM_encode(rsa2);
```

5 X.509v3 Certificates

Using certificates is rather complicated, so only the very basic mechanisms are going to be covered here. The section "Setting up a CA" goes into reasonable detail about CRLs and certificate requests, but there is a lot that isn't covered (else this section would get quite long and complicated).

5.1 Importing and Exporting Certificates

Importing and exporting X.509 certificates is easy. Simply call the constructor with either a `DataSource&`, or the name of a file:

```
X509_Certificate cert1("cert1.pem");

/* This file contains two certificates, concatenated */
DataSource_Stream in("certs2_and_3.pem");

X509_Certificate cert2(in); // read the first cert
X509_Certificate cert3(in); // read the second cert
```

Exporting the certificate is a simple matter of calling the member function **PEM_encode**(), which returns a `std::string` containing the certificate in PEM encoding.

```
std::cout << cert3.PEM_encode();
some_ostream_object << cert1.PEM_encode();
std::string cert2_str = cert2.PEM_encode();
```

5.2 Verifying Certificates

Verifying a certificate requires that we build up a chain of trust, starting from the root (usually a commercial CA), down through some number of intermediate CAs, and finally reaching the actual certificate in question. Thus, to verify, we actually have to have all of those certificates on hand (or at the very least, know where we can get the ones we need).

The class which handles both storing certificates, and verifying them, is called `X509_Store`. We'll start by assuming that we have all of the certificates we need, and just want to verify a cert. This is done by calling the member function **validate_cert**, which takes the `X509_Certificate` in question, and an optional argument of type `Cert_Usage` (which is ignored here; read the section in the API doc titled "Verifying Certificates for information). It returns an enum; `X509_Code`, which, for most purposes, is either `VERIFIED`, or something else (which specifies what circumstance caused the certificate to be considered invalid). Really, that's it.

Now, how to let `X509_Store` know about all those certificates and CRLs we have lying around? The simplest method is to add them directly, using the functions **add_cert**, **add_certs**, **add_trusted_certs**, and **add_crl**; for details, consult the API doc or read the `x509stor.h` header. There is also a much more elegant and powerful method: `Certificate_Store`s. A certificate store refers to an object which knows how to retrieve certificates from some external source (a file, an LDAP directory, a HTTP server, a SQL database, or anything else). By calling the function **add_new_certstore**, you can register a new certificate store, which `X509_Store` will use to find certificates it needs. Thus, you can get away with only adding whichever root CA cert(s) you want to use, letting some online source handle the storage of all intermediate X.509 certificates. The API documentation has a more complete discussion of `Certificate_Store`.

5.3 Setting up a CA

WRITEME

6 Special Topics

This chapter is for subject which don't really fit into the API documentation or into other chapters of the tutorial.

6.1 GUIs

There is nothing particularly special about using Botan in a GUI-based application. However there are a few tricky spots, as well as a few ways to take advantage of an event-based application.

6.1.1 Initialization

Generally you will create the `LibraryInitializer` somewhere in `main`, before entering the event loop. One problem is that some GUI libraries take over `main` and drop you right into the event loop; the question then is how to initialize the library? The simplest way is probably to have a static flag that marks if you have already initialized the library or not. When you enter the event loop, check to see if this flag has not been set, and if so, initialize the library using the function-based initializers. Using `LibraryInitializer` obviously won't work in this case, since it would be destroyed as soon as the current event handler finished. You then deinitialize the library whenever your application is signaled to quit.

6.1.2 Interacting With the Library

In the simple case, the user will do stuff asynchronously, and then in response your code will do things like encrypt a file or whatever, which can all be done synchronously, since the data is right there for you. An application doing something like this is basically going to look exactly like a command line application that uses Botan, the only major difference being that the calls to the library are inside event handlers.

Much harder is something like an SSH client, where you're acting as a layer between two asynchronous things (the user and the network). This actually isn't restricted to GUIs at all (text-mode SSH clients have to deal with most of the same problems), but it is probably more common with a GUI app. The following discussion is fairly vague, but hopefully somewhat useful.

There are a few facilities in Botan that are primarily designed to be used by applications based on an event loop. See the section "User Interfaces" in the main API doc for details.

6.1.3 Entropy

One nice advantage of using a GUI is opening a new method of gathering entropy for the library. This is especially handy on Windows, where the available sources of entropy are pretty questionable. In many versions, `CryptGenRandom` is really rather poor, and the Toolhelp functions may not provide much data on a small system (such as a handheld). For example, in GTK+, you can use the following callback to get information about mouse movements:

```
static gint add_entropy(GtkWidget* widget, GdkEventMotion* event)
   {
   if(event)
      Global_RNG::add_entropy(event, sizeof(GdkEventMotion));
   return FALSE;
   }
```

And then register it with your main GTK window (presumably named *window*) as follows:

```
gtk_signal_connect(GTK_OBJECT(window), "motion_notify_event",
                   GTK_SIGNAL_FUNC(add_entropy), NULL);

gtk_widget_set_events(window, GDK_POINTER_MOTION_MASK);
```

Even though we're catching all mouse movements, and hashing the results into the entropy pool, this doesn't use up more than a few percent of even a relatively slow desktop CPU. Note that in the case of using X over a network, catching all mouse events would cause large amounts of X traffic over the network, which might make your application slow, or even unusable (I haven't tried it, though).

This could be made nicer if the collection function did something like calculating deltas between each run, storing them into a buffer, and then when enough of them have been added, hashing them and send them all to the PRNG in one shot. This would not only reduce load, but also prevent the PRNG from overestimating the amount of entropy it's getting, since its estimates don't (can't) take history into account. For example, you could move the mouse back and forth one pixel, and the PRNG would think it was getting a full load of entropy each time, when actually it was getting (at best) a bit or two.