GPGME, une bibliothèque pour faire de la cryptographie assez simplement

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 17 mars 2008. Dernière mise à jour le 16 avril 2008

https://www.bortzmeyer.org/gpgme.html

Dans un réseau ouvert tel que l'Internet, la sécurité ne peut guère dépendre d'une confiance aveugle dans le réseau et dans son opérateur. À tout moment, les données qui circulent peuvent être modifiées ou écoutées et il est donc nécessaire d'utiliser la cryptographie pour protéger ces données. Mais cette technique est difficile et pleine de pièges et il n'est pas raisonnable de demander à chaque programmeur de lire le fameux "*Applied Cryptography*" de Schneier puis de le mettre en œuvre. Il vaut mieux utiliser une bibliothèque comme GPGME <http://www.gnupg.org/related_software/gpgme/index.en.html>, qui permet au programmeur non spécialiste de cryptographie d'ajouter des fonctions de cryptographie à ses programmes.

GPGME <http://www.gnupg.org/related_software/gpgme/index.en.html> ("*GnuPG Made Easy*") fait partie de la famille de GnuPG, une mise en œuvre libre et très utilisée, de plusieurs mécanismes de cryptographie et notamment de la norme OpenPGP (RFC 4880 [1]). L'utilisateur final appelle typiquement la commande `gpg` depuis son shell, ou bien il se sert d'un programme qui appelle gpg pour lui. Et comment fait ce programme ? Il existe plusieurs méthodes mais les auteurs de GnuPG recommandent GPGME. Programmer soi-même un algorithme de cryptographie est difficile et le risque de se tromper est très important, annulant ainsi toute sécurité. GPGME, s'il ne peut pas être qualifié d'« amical » pour ses utilisateurs, est néanmoins tout à fait accessible au programmeur non-cryptographe.

GPGME fonctionne en lançant l'exécutable `gpg`. Ce choix de dépendre d'un programme extérieur peut sembler surprenant, mais il est motivé entre autres par le désir de d'assurer que toute les initialisations nécessaires, comme celle du générateur aléatoire, sont bien effectuées, ce que ne pourrait pas garantir une bibliothèque fonctionnant entièrement à l'intérieur du programme hôte.

GPGME fournit par défaut une API en C mais il existe aussi des versions pour d'autres langages, par exemple pyme <http://pyme.sourceforge.net/> pour Python.

Que doit faire le programmeur qui veut, par exemple, signer un texte pour en prouver l'authenticité et l'intégrité ? Simplement appeler `gpgme_op_sign` puis `gpgme_data_read` :

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc4880.txt`

```
gpgme_op_sign(context, clear_text, signed_text,
                      GPGME_SIG_MODE_NORMAL);
nbytes = gpgme_data_read(signed_text, buffer, MAXLEN);
buffer[nbytes] = '\0';
printf("Signed text (%i bytes):\n%s\n", (int)nbytes, buffer);
```

En fait, comme souvent en C, c'est bien plus compliqué. Il y a des fonctions cryptographiques supplémentaires (ici, il faut récupérer la clé) et des fonctions de gestion des données, notamment des tampons qu'utilise GPGME (comme `signed_text` plus haut). Voyons donc les différentes étapes (la documentation de GPGME, livrée avec le logiciel, est excellente, quoique pas forcément adaptée au débutant, qui préférera peut-être regarder les exemples dans le répertoire `tests/gpg` de la distribution).

Il faut d'abord initialiser GPGME et créer un **contexte**. La plupart des opérations seront par rapport à ce contexte.

```
gpgme_ctx_t context;
...
/* Initializes gpgme */
gpgme_check_version (NULL);
error = gpgme_new(&context);
fail_if_err(error);
/* Initializes the context */
error = gpgme_ctx_set_engine_info (context, GPGME_PROTOCOL_OpenPGP, NULL,
                                   KEYRING_DIR);
fail_if_err(error);
```

La macro `fail_if_err`, empruntée aux exemples livrées avec GPGME, doit être appelée systématiquement, C n'ayant pas d'exceptions. L'initialisation du contexte me permet de dire que je veux utiliser la norme OpenPGP et que mon **trousseau** de clé se trouvera en `KEYRING_DIR` (par défaut, GPGME utilisera celui du répertoire de "*login*").

On peut maintenant chercher la clé avec laquelle on signera :

```
gpgme_key_t signing_key;
...
/* Retrieve the key for "Jean Dupont" */
error = gpgme_op_keylist_start(context, "Jean Dupont", 1);
fail_if_err(error);
error = gpgme_op_keylist_next(context, &signing_key);
fail_if_err(error);
error = gpgme_op_keylist_end(context);
fail_if_err(error);
error = gpgme_signers_add(context, signing_key);
fail_if_err(error);
```

Il faut maintenant préparer les tampons de données. La plupart des opérations de GPGME portent sur ces tampons, de type `gpgme_data_t` :

```
gpgme_data_t clear_text, signed_text;
...
/* Prepare the data buffers */
error = gpgme_data_new_from_mem(&clear_text, SENTENCE, strlen(SENTENCE), 1);
fail_if_err(error);
error = gpgme_data_new(&signed_text);
fail_if_err(error);
```

`SENTENCE` est le texte en clair qu'on souhaite chiffrer.

Que reste t-il à faire pour signer ? Ah oui, trouver la phrase de passe qui protège la clé privé. GPGME permet de définir un "*callback*" qui va appeler une fonction de lecture de la phrase de passe. Pour cette démonstration, on se contente d'indiquer cette phrase dans le code source :

```
gpgme_error_t passphrase_cb (void *opaque, const char *uid_hint,
                             const char *passphrase_info,
               int last_was_bad, int fd)
{
  write (fd, PASSPHRASE, strlen(PASSPHRASE));
  return 0;
}
...
gpgme_set_passphrase_cb (context, passphrase_cb, NULL);
```

Maintenant, on peut appeler `gpgme_op_sign` pour signer. Il peut être prudent d'appeler ensuite `gpgme_op_sign_result` qui indiquera le résultat détaillé de l'opération :

```
gpgme_sign_result_t result;
...
error = gpgme_op_sign(context, clear_text, signed_text, GPGME_SIG_MODE_NORMAL);
fail_if_err(error);
result = gpgme_op_sign_result(context);
if (result->invalid_signers) {
      fprintf (stderr, "Invalid signer found: %s\n",
       result->invalid_signers->fpr);
      exit (1);
}
```

Une fois la signature effectuée, pour lire le tampon de données, il faut ramener le curseur à l'origine (ces tampons s'utilisent comme des fichiers plutôt que comme des tableaux), avec `gpgme_data_seek` (si cette fonction renvoie EINVAL, "*Invalid argument*", c'est que GPGME s'est emmêlé les pinceaux entre les "*offsets*" de 32 bits et ceux de 64 bits, voir plus loin les options de compilation). On pourra ensuite lire le résultat :

```
error = gpgme_data_seek (signed_text, 0, SEEK_SET);
fail_if_err(error);
buffer = malloc(MAXLEN);
nbytes = gpgme_data_read(signed_text, buffer, MAXLEN);
if (nbytes == -1) {
    fprintf (stderr, "%s:%d: %s\n",
     __FILE__, __LINE__, "Error in data read");
    exit (1);
}
buffer[nbytes] = '\0';
printf("Signed text (%i bytes):\n%s\n", (int)nbytes, buffer);
```

Un source complet de toute cette opération de recherche de clé et de signature est disponible en (en ligne sur `https://www.bortzmeyer.org/files/gpgme-sign.c`). Il peut se compiler avec, par exemple :

```
% gcc -Wall -o gpgme-sign $(gpgme-config --cflags) \
               $(gpgme-config --libs) \
      gpgme-sign.c
```

Dans certains cas, GPGME mélange des "*offsets*" (comme l'argument de `gpgme_data_seek`) de 32 et de 64 bits. Le problème est documenté dans la section "*Largefile Support (LFS)*" de la documentation de GPGME. Un contournement est de rajouter `-D_FILE_OFFSET_BITS=64` aux arguments de compilation. Merci à Kim-Minh Kaplan pour son aide pour le débogage de ce problème.

Et si on veut chiffrer et pas juste signer ? On utilise `gpgme_op_encrypt` par exemple ainsi :

```
gpgme_data_t clear_text, encrypted_text;
gpgme_key_t recipients[2] = {NULL, NULL};
           /* The array must be NULL-terminated */
...
error = gpgme_op_keylist_start(context, "John Smith", 1);
error = gpgme_op_keylist_next(context, &recipients[0]);
...
error = gpgme_op_encrypt(context, recipients,
   GPGME_ENCRYPT_ALWAYS_TRUST,
                            clear_text, encrypted_text);
```

Un exemple complet est disponible en (en ligne sur `https://www.bortzmeyer.org/files/gpgme-encrypt.c`).

GPGME utilise le trousseau de clés habituel de GPG. Si on veut générer ses propres clés, on peut mais GPGME ne sait pas encore les générer en mémoire ou dans un fichier, il faut passer par un trousseau. `gpgme_op_genkey` permet de générer une clé, les paramètres s'indiquant par une chaîne de caractères mélangeant le XML et les doublets attributs :valeur :

```
#define KEYPARAMS "<GnupgKeyParms format=\"internal\">\n   \
          Key-Type: DSA\n   \
          Key-Length: 512\n   \
          Subkey-Type: ELG-E\n   \
          Subkey-Length: 512\n   \
          Name-Real: John Smith\n   \
          Name-Comment: with stupid passphrase\n   \
          Name-Email: joe@foobar.example\n   \
          Expire-Date: 0\n   \
          Passphrase: abc\n   \
          </GnupgKeyParms>\n"
...
error = gpgme_op_genkey(context, KEYPARAMS, NULL, NULL);
```

Pour Python, nous allons utiliser la bibliothèque pyme <`http://pyme.sourceforge.net`>. C'est une interface **mince** à GPGME, ce qui veut dire qu'elle colle de très près à l'API originale, et qu'elle fait peu d'efforts pour être plus « Pythonique ». Cela permet de prendre la documentation de l'API C pratiquement sans modification.

Voici un exemple de code pyme pour signer un message :

```
#!/usr/bin/python

# Modules necessaires
from pyme import core, callbacks
from pyme.constants.sig import mode
```

```
sentence = "I swear it is true"
passphrase = "abc"

# Renvoie la phrase de passe, qui est dans le code source. Pour des
# demos uniquement !
def mypassphrase(hint, desc, prev_bad):
    print "Passphrase for %s" % hint
    return passphrase

# Texte en clair
plain = core.Data(sentence)
# Le futur texte signe
signed = core.Data()
context = core.Context()
context.set_passphrase_cb(mypassphrase)
# Signer en clair (lisible même si on n'a pas gpg)
context.op_sign(plain, signed, mode.CLEAR)
# Rembobiner (revenir au debut)
sig.seek(0,0)
# Afficher le resultat
print sig.read()
```

Pour vérifier une signature, ce code peut convenir :

```
# Le texte signé est sur l'entrée standard
signed = core.Data(sys.stdin.read())
plain = core.Data()
context = core.Context()

context.op_verify(signed, None, plain)
result = context.op_verify_result()

sign = result.signatures
while sign:
    if sign.status != 0:
        print "BAD signature from:"
    else:
        print "Good signature from:"
    print "  uid:        ", context.get_key(sign.fpr, 0).uids.uid
    print "  timestamp:  ", sign.timestamp
    print "  fingerprint:", sign.fpr
    sign = sign.next
```

On voit qu'il est très proche du code C, jusqu'aux parcours des signatures qui se fait avec `sign.next` alors qu'un code Python natif aurait certainement utilisé des itérateurs.

Dernier exemple Python, un code de signature, encore, mais qui permet de sélectionner une des clés disponibles dans le trousseau, en indiquant son identificateur, ici `0x7F99B55C`. Ce code est l'équivalent de : `gpg --sign --armor --clear --local-user 7F99B55C`en ligne de commande :

```
sentence = "I swear it is true"
passphrase = "abc"
keyID = "7F99B55C"
plain = core.Data(sentence)
sig = core.Data()
context = core.Context()
# The parameter is named "fpr" but can be used for key IDs as well
mykey = context.get_key(fpr=keyID, secret=True)
context.signers_add(mykey)
# passphrase_stdin is predefined in pyme and interactively queries the
# user.
context.set_passphrase_cb(callbacks.passphrase_stdin)
context.op_sign(plain, sig, mode.CLEAR)
sig.seek(0,0)
print sig.read()
```