Introduction à PolarSSL

Sp0tty

6 mai 2011

Table des matières

1 Présentation

1.1 Il était une fois ...

PolarSSL est une librairie qui implémente le protocole SSL/TLS ainsi que de nombreux algorithmes cryptographiques en C. Son implémentation très légère la rend idéale pour les systèmes embarqués puisqu'elle est compatible avec les architectures ARM, PowerPC, MIPS et Motorola 68000.

Pour la petite histoire, cette librairie est la continuité de la librairie *XySSL* dont le développeur était *Christophe Devine* (Aircrack ca vous parle ?).

Ce qui fait la force de cette librairie est sa simplicité d'utilisation : le code source est auto-documenté !

1.2 Ce qu'on a à disposition

Voici la liste de tous les composants cryptographiques que l'on a :

Chiffrement symétrique : AES, Triple-DES, DES, ARC4, Camellia, XTEA ;

Fonctions de hachage : MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 ;

Générateur de nombres pseudo-aléatoires : HAVEGE ;

Chiffrement asymétrique : RSA (PKCS#1 v1.5) ;

Négociation de clés : Diffie-Hellman ;

SSL/TLS : SSLv3, TLSv1.0, TLSv1.1 ;

X.509 : gestion de certificats.

La liste est tout de même moins grande que celle de la librairie *OpenSSL* mais c'est ce qu'il se fait de plus courant.

Attention : ceci n'est pas un cours crypto, juste un apperçu d'une librairie crypto donc je suppose que vous êtes familiers avec tous les termes.

1.3 Installation

```
wget http://polarssl.org/code/releases/polarssl-0.14.3-gpl.tgz
tar -xvzf polarssl-0.14.3-gpl.tgz
cd polarssl-0.14.3
make && make install
```

2 Crypto symétrique : exemples d'utilisation

Certainement la partie la plus facile à appréhender, la crypto symétrique regroupe les fonctions de hachage et les algorithmes symétriques de chiffrement.

Au programme : fonctions de hachage (MD5), algorithmes de chiffrement (DES, AES, Camellia, ARC4) et génération d'aléa.

2.1 Fonctions de hachage

Prenons par exemple la fonction de hachage MD5. Nous avons besoin d'analyser le fichier include/polarssl/md5.h afin de connaitre l'API :

- void md5_starts(md5_context *ctx);
- void md5_update(md5_context *ctx, const unsigned char *input, int ilen);
- void md5_finish(md5_context *ctx, unsigned char output[16]);
- void md5(const unsigned char *input, int ilen, unsigned char output[16]);
- int md5_file(const char *path, unsigned char output[16]);
- void md5_hmac_starts(md5_context *ctx, const unsigned char *key, int keylen);
- void md5_hmac_update(md5_context *ctx, const unsigned char *input, int ilen);
- void md5_hmac_finish(md5_context *ctx, unsigned char output[16]);
- void md5_hmac_reset(md5_context *ctx);
- void md5_hmac(const unsigned char *key, int keylen, const unsigned char *input, int ilen, unsigned char output[16]);
- int md5_self_test(int verbose);

On a donc la possibilité de faire un MD5 sur une suite d'octets, sur un fichier, un HMAC-MD5 sur un suite d'octets, de tester la fonction MD5 avec des vecteurs de tests.

Ce qui est valable ici pour le MD5 est aussi valable avec les autres fonctions de hachage disponible dans le librairie. Il faudra juste adapter la taille du résultat en fonction de la fonction utilisée.

2.1.1 MD5 - Suite d'octets

La fonction qui nous concerne est void md5(const unsigned char *input, int ilen, unsigned char output[16]). Prenons une chaine d'octets aléatoire et calculons-en son MD5 :

```
#include "polarssl/md5.h"
...
unsigned char my_rand_string[32]; //buffer
unsigned char hash[16];           //hash
...
/* Fill the buffer with 32 bytes from /dev/urandom */
fread(my_rand_string, 1, 32, f_urandom);
/* MD5 it */
md5(my_rand_string, 32, hash);
...
```

Bien évidemment ce code ce qu'il nous faut si la taille du buffer dont on veut obtenir le MD5 est connue à l'avance et si cette taille est raisonnable (problème de mémoire saturée, etc).

Maintenant, voici le code équivalent écrit pour avoir une lecture bufferisée en 512 octets de 2^{20} octets tirés de /dev/urandom :

```
#include "polarssl/md5.h"
...
md5_context md5ctx;
int len_to_be_hashed;
unsigned char buffer[512]; //buffer
unsigned char hash[16];    //hash
...
/* init */
md5_starts(&md5ctx);
len_to_be_hashed = 1<<20;
...
while(len_to_be_hashed > 0)
{
    fread(buffer 1, 512, f_urandom);
```

```
    md5_update(&md5ctx, buffer, 512);
    len_to_be_hashed -= 512
}
/* Get the result */
md5_finish(&md5ctx, hash);
/* Clean MD5_context */
memset(&md5ctx, 0, sizeof(md5_context));
...
```

2.1.2 MD5 - HMAC

Le HMAC est un procédé qui utilise une fonction de hachage et une clé symétrique.
Comme pour un simple calcul MD5, nous pouvons procéder des 2 façons.
Sans lecture bufferisée :

```
#include "polarssl/md5.h"
...
unsigned char my_rand_string[32]; //buffer
unsigned char my_key[16]; //key
unsigned char hmac[16]; //hmac
...
md5_hmac(my_key, 16, my_rand_string, 32, hmac);
...
```

et maintenant avec lecture bufferisée :

```
#include "polarssl/md5.h"
...
md5_context md5ctx;
int len_to_be_hashed;
unsigned char buffer[512]; //buffer
unsigned char my_key[16]; //key
unsigned char hmac[16]; //hash
...
/* init */
md5_hmac_starts(&md5ctx, my_key, 16);
len_to_be_hashed = 1<<20;
...
while(len_to_be_hashed > 0)
{
    fread(buffer 1, 512, f_urandom);
    md5_hmac_update(&md5ctx, buffer, 512);
    len_to_be_hashed -= 512
}
/* Get the result */
md5_hmac_finish(&md5ctx, hmac);
/* Clean MD5_context */
memset(&md5ctx, 0, sizeof(md5_context));
...
```

2.2 Chiffrement symétrique

Dans les fonctions de chiffrement symétrique qui sont disponibles dans la librairie PolarSSL vous avez toujours 3 étapes à réaliser :

1. initialisation du contexte de chiffrement (cadencement des sous-clés) ;
2. (dé)chiffrement ;
3. mise à zero du contexte (sinon les clés restent en mémoire).

2.2.1 DES

Regardons l'API dans le fichier include/polarssl/des.h :
- void des_setkey_enc(des_context *ctx, const unsigned char key[8]) ;
- void des_setkey_dec(des_context *ctx, const unsigned char key[8]) ;
- void des3_set2key_enc(des3_context *ctx, const unsigned char key[16]) ;
- void des3_set2key_dec(des3_context *ctx, const unsigned char key[16]) ;
- void des3_set3key_enc(des3_context *ctx, const unsigned char key[24]) ;
- void des3_set3key_dec(des3_context *ctx, const unsigned char key[24]) ;

– int des_crypt_ecb(des_context *ctx, const unsigned char input[8], unsigned char output[8]) ;
– int des_crypt_cbc(des_context *ctx, int mode, int length, unsigned char iv[8], const unsigned char *input, unsigned char *output) ;
– int des3_crypt_ecb(des3_context *ctx, const unsigned char input[8], unsigned char output[8]) ;
– int des3_crypt_cbc(des3_context *ctx, int mode, int length, unsigned char iv[8], const unsigned char *input, unsigned char *output) ;
– int des_self_test(int verbose) ;

Donc, à la lecture de l'API, nous avons à notre disposition les algorithmes DES, 3DES avec les modes opératoires ECB et CBC. On notera que le padding pour le mode CBC est à notre charge.

Le code ci-dessous illutre le chiffrement avec l'algorithme symétrique DES-CBC

```
#include "polarssl/des.h"
[...]
int ret;
unsigned char des_key[8];        // cle DES de 64 bits
unsigned char iv[8];             // IV de 64 bits
unsigned char plain[512];        // texte clair de 512 octets
unsigned char cipher[512];       // texte chiffre de 512 octets
unsigned char decipher[512];     // texte dechiffre de 512 octets
des_context cipher_ctx;          // contexte de chiffrement
des_context decipher_ctx;        // contexte de dechiffrement
[...]
fread(des_key, 1, 8, f_urandom); // cle aleatoire venant de /dev/urandom
fread(plain, 1, 512, f_urandom); // texte aleatoire venant de /dev/urandom
[...]
/* Chiffrement */
memset(iv, 0, 8);                // IV à 0
des_setkey_enc(&cipher_ctx, des_key);
ret = des_crypt_cbc(&cipher_ctx, ENCRYPT, 512, iv, plain, cipher);
memset(&cipher_ctx, 0, sizeof(des_context));
if(ret != 0)
{
    printf("[-] cipher KO\n");
    return 1;
}
printf("[+] cipher OK\n");
[...]
/* Dechiffrement */
memset(iv, 0, 8);                // IV à 0
des_setkey_dec(&decipher_ctx, des_key);
ret = des_crypt_cbc(&decipher_ctx, DECRYPT, 512, iv, cipher, decipher);
memset(&decipher_ctx, 0, sizeof(des_context));
if(ret != 0)
{
    printf("[-] decipher KO\n");
    return 1;
}
printf("[+] decipher OK\n");
[...]
if(memcmp(plain, decipher, 512) == 0)
printf("[+] cipher/decipher OK\n");
else
printf("[-] cipher/decipher KO\n");
[...]
```

2.2.2 AES

L'utilisation de l'AES est semblable à celle du DES. On notera la présence du mode opératoire CFB en supplément de l'ECB et le CBC. Attention tout de même à la taille de clé de l'AES et à la taille de bloc (donc taille de l'IV). Vous trouverez l'API dans le fichier include/polarssl/aes.h.

2.2.3 Camellia

L'utilisation de Camellia est identite à celle de l'AES. Vous trouverez l'API dans le fichier include/polarssl/camellia.h

2.2.4 ARC4

A la différence du DES de l'AES et de Camellia, ARC4 est algorithme de chiffrement symétrique par flots, donc pas besoin de mode opératoire.

l'API est disponible dans le fichier include/polarssl/arc4.h :

- void arc4_setup(arc4_context *ctx, const unsigned char *key, int keylen) ;
- int arc4_crypt(arc4_context *ctx, int length, const unsigned char *input, unsigned char *output) ;
- int arc4_self_test(int verbose) ;

Maintenant, le traditionnel bout de code pour vous montrer l'utilisation de l'ARC4 :

```
#include "polarssl/arc4.h"
[...]
int ret;
unsigned char key[16];           // cle ARC4 de 128 bits
unsigned char plain[512];        // texte clair de 512 octets
unsigned char cipher[512];       // texte chiffre de 512 octets
unsigned char decipher[512];     // texte dechiffre de 512 octets
arc4_context cipher_ctx;         // contexte de chiffrement
arc4_context decipher_ctx;       // contexte de dechiffrement
[...]
fread(key, 1, 16, f_urandom);    // cle aleatoire venant de /dev/urandom
fread(plain, 1, 512, f_urandom); // texte aleatoire venant de /dev/urandom
[...]
/* Encrypt */
arc4_setup(&cipher_ctx, key, 16);
ret = arc4_crypt(&cipher_ctx, 512, plain, cipher);
memset(&cipher_ctx, 0, sizeof(arc4_context));
if(ret != 0)
{
    printf("[-] cipher KO\n");
    return 1;
}
printf("[+] cipher OK\n");
[...]
/* Decrypt */
memset(iv, 0, 8);                // IV à 0
arc4_setup(&decipher_ctx, key, 16);
ret = arc4_crypt(&decipher_ctx, 512, plain, cipher);
memset(&decipher_ctx, 0, sizeof(arc4_context));
if(ret != 0)
{
    printf("[-] decipher KO\n");
    return 1;
}
printf("[+] decipher OK\n");
[...]
/* plain ?= decipher */
if(memcmp(plain, decipher, 512) == 0)
printf("[+] cipher/decipher OK\n");
else
printf("[-] cipher/decipher KO\n");
[...]
```

2.3 Génération d'alea

Le module Havege s'occupe de collecter de l'entropie et de le retraiter afin de fournir de l'aléa de bonne qualité.

Un petit tour par l'API, disponible dans le fichier include/polarssl/havege.h pour voir ce qui nous attend :

- void havege_init(havege_state *hs) ;
- int havege_rand(void *p_rng) ;

C'est donc très simple à utiliser, la preuve en lignes de code :

```
#include "polarssl/havege.h"
[...]
havege_state ctx;
unsigned char random[512];
[...]
havege_init(&ctx);
for(i = 0; i < 512; i++)
```

```
    random[512] = havege_rand(&ctx);
[...]
```

2.4 Que faire maintenant ?

Nous voici maintenant en possession d'un générateur d'aléa, d'algorithmes de chiffrement symétrique et de fonctions de hachage. Il ne vous reste plus qu'à faire un utilitaire qui va chiffrer un fichier et garantir son integrité.

3 Crypto asymétrique : exemples d'utilisation

Maintenant que l'utilisation de la crypto symétrique est maîtrisée, passons à l'étape d'après : la crypto asymétrique. On pourrait la vulgariser en donnant la définition suivante : *clé de chiffrement != clé de déchiffrement.*

Au programme : échange de clés Diffie-Hellman, chiffrement RSA et signature RSA.

3.1 Echange de cles Diffie-Hellman

Le protocole Diffie-Hellman permet à deux personnes de se mettre d'accord sur un secret commun qu'ils auront tous les 2 influencés de façon équitable. Pour faire simple, voici les étapes du protocole de base :

1. p et g sont des valeurs publiques ;
2. Alice génère un secret a, calcule $A = g^a[p]$, envoie A à Bob ;
3. Bob génère un secret b, calcul $B = g^b[p]$, envoie B à Alice ;
4. Alice et Bob calculent chacun de leur côté le secret commun K de la façon suivante :
   - Alice calcule $K = B^a[p]$;
   - Bob calcule $K = A^b[p]$;

Plutôt que de faire un copier-coller, je vous laisse lire les sources des fichiers programs/pkey/dh_client.c et programs/pkey/dh_server.c afin de voir comment bin utiliser l'API (disponible dans le fichier include/polarssl/dhm.h.

3.2 RSA

PolarSSL implémente pour le moment que la norme de padding PKCS#1v1.5, libre à vous de proposer un patch pour supporter le padding OAEP et PSS.

Un petit tour dans le fichier include/polarssl/rsa.h et voici l'API :

- void rsa_init(rsa_context *ctx, int padding, int hash_id) ;
- int rsa_gen_key(rsa_context *ctx, int (*f_rng)(void *), void *p_rng, int nbits, int exponent) ;
- int rsa_pkcs1_encrypt(rsa_context *ctx, int (*f_rng)(void *), void *p_rng, int mode, int ilen, const unsigned char *input, unsigned char *output) ;
- int rsa_pkcs1_decrypt(rsa_context *ctx, int mode, int *olen, const unsigned char *input, unsigned char *output, int output_max_len) ;
- int rsa_pkcs1_sign(rsa_context *ctx, int mode, int hash_id, int hashlen, const unsigned char *hash, unsigned char *sig) ;
- int rsa_pkcs1_verify(rsa_context *ctx, int mode, int hash_id, int hashlen, const unsigned char *hash, unsigned char *sig) ;

3.2.1 Génération de clés RSA

```
#include "polarssl/rsa.h"
[...]
int ret;
rsa_context ctx;
havege_state hs;
[...]
havege_init(&hs);
rsa_init(&ctx, RSA_PKCS_V15, SIG_RSA_MD5);
ret = rsa_gen_key(&ctx, havege_rand, &hs, 1024, 3);  //RSA-1024, e=3
if(ret == 0)
    printf("[+] RSA generation OK\n");
else
    printf("[-] RSA generation KO\n");
[...]
rsa_free(&ctx);
[...]
```

3.2.2 Chiffrement RSA

On va supposer que les cles sont déjà générées. Pour petit rappel, on chiffre avec la clé publique du destinataire.

```
#include "polarssl/rsa.h"
[...]
int ret;
int olen;
rsa_context ctx;
havege_state hs;
unsigned char plain[64];
unsigned char cipher[64];
unsigned char decipher[64];
[...]
ret = rsa_pkcs1_encrypt(&ctx, havege_rand, &hs, RSA_PUBLIC, 128, plain, cipher);
if(ret == 0)
    printf("[+] RSA encrypt OK\n");
else
    printf("[-] RSA encrypt KO\n");
[...]
ret = rsa_pkcs1_decrypt(&ctx, RSA_PRIVATE, cipher, decipher, 64);
if(ret == 0)
    printf("[+] RSA decrypt OK\n");
else
    printf("[-] RSA decrypt KO\n");
[...]
/* plain ?= decipher */
if(memcmp(plain, decipher, 64) == 0)
printf("[+] cipher/decipher OK\n");
else
printf("[-] cipher/decipher KO\n");
[...]
rsa_free(&ctx);
[...]
```

3.2.3 Chiffrement RSA

On va supposer que les cles sont déjà générées. Pour petit rappel, on signe avec sa clé privée.

```
#include "polarssl/rsa.h"
#include "polarssl/md5.h"
[...]
int ret;
int olen;
rsa_context ctx;
unsigned char hash[16];
unsigned char sign[128];
[...]
ret = rsa_pkcs1_sign(&ctx, RSA_PRIVATE, SIG_RSA_MD5, 128, hash, sign);
if(ret == 0)
    printf("[+] RSA sign OK\n");
else
    printf("[-] RSA sign KO\n");
[...]
ret = rsa_pkcs1_verify(&ctx, RSA_PUBLIC, SIG_RSA_MD5, 128, hash, sign);
if(ret == 0)
    printf("[+] RSA verif OK\n");
else
    printf("[-] RSA verif KO\n");
[...]
/* plain ?= decipher */
if(memcmp(plain, decipher, 64) == 0)
printf("[+] cipher/decipher OK\n");
else
printf("[-] cipher/decipher KO\n");
[...]
rsa_free(&ctx);
[...]
```