Aicrypto ライブラリについて

14, March, 2001

奥野　琢人

目次

1. はじめに

暗号関係でフリーで有名なライブラリと言えば、OpenSSL(旧 SSLeay)を上げることができる。しかし暗号の研究を行うにあたって、このような外部のライブラリを使用することになれば作成したアプリケーションを一般に公開することが難しくなる。また、ディジタル証明書等の仕組みを理解する上で、外部のライブラリを使用していたのでは深く理解することができない。このような状態を解決するには、ライブラリを自分で作成するのが一番であり、その意図に沿って Aicrypto ライブラリを作成した。

2. ライブラリの性能

2.1 暗号とハッシュ関数

現在、ライブラリが保持している暗号化方式、ハッシュ関数は以下の通りである。

公開鍵暗号	RSA（キー生成可能） ECDSA（楕円署名）
共通鍵暗号	DES（ECB,CBC,CFB） 3DES（ECB,CBC） RC2（ECB,CBC）
ハッシュ関数	MD2, MD5, SHA1, HMAC

初期バージョンでは、OpenSSL と処理速度を比較すると、多少の速度差を感じた。特に RSA での 1024bit 以上の計算や、DES の実行速度はかなりの差があったといえる。しかし、現バージョンでは巨大数演算や DES などで、アセンブラを意識してコーディングし直し、また、多くのアルゴリズム改良を行った結果、それほど速度差を感じないレベルまで高速化をする事が出来た。

2.2 証明書等のファイル形式

ディジタル証明書、CRL、秘密鍵はファイルに格納されるが、このファイル化にあたっては多くの方式がある。現在、ライブラリが扱えるファイル形式は以下の通りである。

証明書	X.509 DER 形式 （*.cer） X.509 PEM 形式 （*.pem）※1 PKCS#7 DER 形式 （*.p7b） PKCS#12 DER 形式 （*.p12, *.pfx）※2
CRL	X.509 DER 形式 （*.crl） X.509 PEM 形式 （*.pem） PKCS#7 DER 形式 （*.p7b） PKCS#12 DER 形式 （*.p12, *.pfx）
秘密鍵	X.509 PEM 形式 PKCS#8 PEM, DER 形式 (*.crtx) PKCS#12 DER 形式 （*.p12, *.pfx）
証明書要求	PKCS#10 DER 形式 （*.req, *.p10） PKCS#10 PEM 形式 （*.pem）

※ 1 拡張子は*.pem と表記してあるが、Microsoft ではこちらも*.cer と表記する。

※ 2 PKCS#12 形式は Netscape と Microsoft では内容が異なっており、読み込みは両方可能で、書き込みは Netscape 方式である。

3. 動作環境とライブラリ構築

3.1 動作環境

本ライブラリは、ソースコードで提供されるため、基本的にコンパイルをすれば UNIX 上のどの OS でも動作する。現在動作確認をおこなった OS は以下の通りである。

Solaris2.5.1（Sparc）, Solaris2.8 (x86),
IRIX5.3, FreeBSD 3.5, LINUX (RED HAT 7.0.1)

どの OS でも、コンパイラには gcc を使用した。

3.2 ライブラリ構築

aicrypto ライブラリのディレクトリ構成は以下の通りである。

Makefile.in	des/	hmac/	md2/	rand/	sha1/	tool/
aiconfig.h.in	doc/	include/	md5/	rc2/	smime/	x509/
configure	ecc/	lib/	pem/	rc4/	ssl/	wincry/
asn1/	ecdsa/	lnm/	pkcs/	rsa/	test/	

このディレクトリがカレントディレクトリの時、

```
hoge% ./configure
hoge% make
```

を実行することで、lib ディレクトリに libaicrypto.a が構築される。また、オブジェクトファイル等を削除したい時は、

```
hoge% make clean
```

を実行すれば良い。なお、rc4 のディレクトリも存在しているが、 rc4 は現状ではライブラリからは外してある。

4. ライブラリの構成

4.1 概説

前節のディレクトリ構成から説明する。

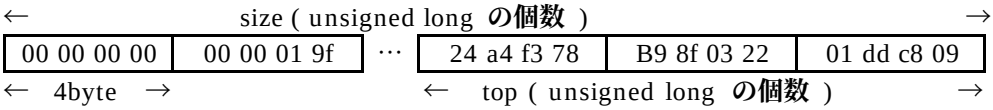
Lnm	巨大数を扱うモジュール（固定値を malloc するため 2048bit まで扱える）
rand	PRNG (擬似乱数生成) Lutz Jaenicke のものをベースとしている
rsa	RSA 公開鍵暗号を扱うためのモジュール
ecc	楕円曲線暗号を扱うためのモジュール P1363 等を参照
ecdsa	楕円 DSA を扱うためのモジュール
des	DES, TripleDES を扱うためのモジュール
rc2	RC2 を扱うためのモジュール
md2	MD2 ハッシュ関数、RFC1319 を参照
md5	MD5 ハッシュ関数、RFC1321 を参照
sha1	SHA1 ハッシュ関数、FIPS PUB 180-1 を参照
hmac	HMAC 鍵付ハッシュ関数、RFC2104 を参照
asn1	ASN.1 のバイナリ形、DER を解析、生成するモジュール
pem	PEM 形式のファイルを読み書きするモジュール
pkcs	RSA 社による規格、PKCS の形式を扱うモジュール
x509	Cert,CRL,Key の構造体を使用し、証明書の認証等を扱うモジュール
tool	パスワード読み込み等、その他の仕事を行うモジュール
smime	S/MIME 用 API 群
ssl	SSLv3 用 API 群
wincry	Windows Crypto API 向け互換 API 群

- rsa は巨大数を使用するため lnm に依存している。
- lnm, ssl, pkcs 等ランダムバイトが必要なモジュールは rand に依存している。
- des～sha1 までは独立したモジュールだが､hmac は他の hash 関数を使用するため､このライブラリでは、md5 と sha1 に依存している。
- asn1 は基本的には独立しているが、rsa-key の DER（PKCS#1）や証明書 DER（X.509）を解析するモジュールも内包している。
- pem, pkcs, x509 は smime, ssl, wincry 以外の他の全てのモジュールを必要とする。
- smime, ssl, wincry は他の全てのモジュールを必要とする。

4.2 lnm（巨大数演算）

巨大数モジュールについて解説する。構造体の構成は以下の通りである。(バージョン 1.2a までは、配列に short 型を使っていたが、バージョン 1.3 以降は long 型を使用。つまり、これらの LN**m**構造体間では互換性が無い。また、バージョン 1.6 からは neg フラグにより負の数の演算も可能になった。)

```
typedef struct large_num{
  unsigned  long *num;
   int    top;
   int    size;
   int    neg;
}LNm;
```



上図の通り、size は unsigned long の配列の個数であり数値は右詰で入っている。この時、数値の先頭を表すために使われうるのが top であり、右詰からの有効な配列の個数を表している。実際には、高速化のため size の値は LN_MAX=128 に固定されている。この値は 128×32=4096bit であり、掛け算を考えるとその半数の 2048bit が実際に有効に使える bit 数になる。このため、RSA 暗号は 2048bit までしか扱えない。

この LNm を使ったプログラムのサンプルを以下に載せる。

```
#include <stdio.h>
#include “large_num.h”

int main(int argc,char **argv){
   unsigned long a1[]={
      0xc3d58d17,0x31e21f38,0xf41e6dcf,0x37d07210};
   LNm  *a,*b,*ret;
   a=LN_alloc(); b=LN_alloc(); ret=LN_alloc();
   LN_set_num(a,4,a1); /* short 型は LN_set_num_s(), char 型は LN_set_num_c() */
   LN_multi(a,a,b); /* b=a*a */
   LN_plus(a,b,ret); /* ret=a+b */
   LN_print(ret);
}
```

この他にも、減算、除算、余りを求める関数や、素数、逆数を求める関数も用意されている。

もし、これらを使用するのであれば、large_num.h や testlnm.c を参考にしてほしい。なお、testlnm.c を動作させる場合は、

```
hoge% make –f Makefile.test
   ： コンパイル実行
hoge% ts
```

で実行することが出来る。

4.3 rsa（RSA 公開鍵暗号）

構造体の構成は以下の通りである。

```
Typedef struct Private_key_RSA{
  int    key_type; /* key identifier */
   int     size;

   int     version;
  LNm     *n;       /* public module */
  LNm     *e;       /* public exponent */
  LNm     *d;       /* private exponent */
  LNm     *p;       /* prime1 */
  LNm     *q;       /* prime2 */
  LNm     *e1;      /* exponent1 -- d mod (p-1) */
  LNm     *e2;      /* exponent2 -- d mod (q-1) */
  LNm     *cof;     /* coefficient -- (q-1) mod p */

  /* DER encode strings */
  unsigned char *der;
}Prvkey_RSA;
```

```
typedef struct Public_key_RSA{
  int    key_type; /* key identifier */
   int     size;

  LNm     *n;       /* public module */
  LNm     *e;       /* public exponent */
}Pubkey_RSA;
```

Prvkey_RSA 構造体は、RSA キーの全ての情報を保持している。Der にはファイルから読み込んだ DER のバイナリ列をそのまま保持しておき、自分でキーを生成した場合は、キー生成時に DER 文を作成し保持する。key_type は、Prvkey_RSA が KEY_RSA_PRV であり、Pubkey_RSA が KEY_RSA_PUB を保持している。size はバイト数（512bit なら 64byte）を保持している。

Pubkey_RSA 構造体は、RSA キーの公開鍵部分であり、後述する Cert 構造体によってポインタが保持され

る。この時、Cert 構造体では Key*が定義されているので、

```
Cert->pubkey=(Pubkey_RSA*)rsapubkey;
```

のようにキャストして保持させる。

関数はそれほど多くなく、簡単に使用できる。まず、それぞれのキーを確保する。

```
Prvkey_RSA *prv;
Pubkey_RSA *pub;
prv =RSAprvkey_new();
pub =RSApubkey_new();
```

つぎに、Prv の中身を作成し、それを Pub にコピーする。なお、以下の関数では生成する素数のバイト数を指定する。この例では 64byte×8=512bit の素数を生成するので、公開鍵 n は 1024bit である。

```
RSAprv_generate(prv,64);
RSAprv_2pub(prv,pub);
```

実際に暗号化を行う場合、入力するバイト列は公開鍵 n よりも小さなバイト数でないといけない。以下では、公開鍵 n が 1024bit の場合の例を挙げる。

```
char in[]="test string.",cry[130],ret[130];
OK_RSA_docrypt_pubkey(strlen(in),in,cry,pub);
OK_RSA_docrypt_prvkey(prv->size,cry,ret,prv);
```

公開鍵で暗号化する時に右詰で行うため、帰り値も右詰である。その辺りは rsa_test.c を、また、関数の入力型は ok_rsa.h 参照するとよい。テスト動作は LNm と同様に make –f Makefile.test にてコンパイルして動作確認する。最後に、キーのメモリ開放を行う。

```
RSAkey_free(pub);
RSAkey_free(prv);
```

この操作によって、それぞれのキーが保持している巨大数のメモリとキー構造体自身のメモリを開放することが出来る。

4.4 ecc（楕円曲線暗号）

4.4.1 楕円曲線上の点の演算

楕円曲線上の点の演算について解説する。演算に関する構造体は以下の通りである。

```
typedef struct EC_Point{
        LNm     *x;
        LNm     *y;
        LNm     *z;
        int     infinity;
}ECp;


#define ECP_BUF   16
```

```
typedef struct Elliptic_Curve_Param{
        int     type;
        int     version;
        LNm *a;
        LNm *b;         /* y^2 = x^3 + a*x + b mod p */
        LNm *p;         /* field modulo p */
        LNm *n;         /* the order of G on E */
        LNm *h;         /* #E = n*h (so,#E is nearly prime)*/
        int     psize;  /* bit length of p */
        int     nsize;  /* bit length of n */
        ECp *G;         /* base Point */
/* temporary buffers for culculation */
        LNm *buf[ECP_BUF];
}ECParam;
```

まず、基本となる点の構造体は 2 次元ではなく Projective 演算に対応出来るよう 3 次元の Jacobian 座標系であり、無限遠点を示す infinity を持っている。また、楕円暗号では負の巨大数を扱うことがあるため、バージョン 1.6 から LNm には負の数の演算を行えるよう拡張が施されている。

また、右側の構造体は楕円パラメータ（楕円曲線と底点）を現している。楕円 DSA や楕円 ElGamal と言った楕円暗号系はこの「システムパラメータ」と秘密鍵・公開鍵は別物であり、通常は公開されているパラメータから秘密鍵を構成する。なお、バージョン 1.6 以前では安全で巡回群を構成するパラメータの生成が行えなかったため、X9.62 で定義されている楕円パラメータを使用していた。

この、パラメータ構造体は以下のように取得する。

```
/* ok_ecc.h を include する */
ECParam *E;
E = ECPm_get_std_parameter(ECP_TYPE_X9_62_192); /* 192bit のパラメータ */
ECPm_free(E);
```

次に、取得したパラメータ上での点の演算について述べる。新しい点の生成と破棄は以下の通りである。

```
ECp *a;
a = Ecp_new();
ECp_free(a);
```

なお、点に座標をセットする場合は、直接 a->x や a->y の構造体の中身を使用する。これらの点とパラメータ

がそろった時、初めて以下に示される曲線上の演算を実行できる。

```
void ECp_add(ECParam *E, ECp *A, ECp *B, ECp *ret);
void ECp_sub(ECParam *E, ECp *A, ECp *B, ECp *ret);
void ECp_multi(ECParam *E, ECp *A, LNm *k, ECp *ret);
```

通常、楕円曲線暗号では点の「加算」「減算」「k 倍算」が使用される。上記の関数は P(x,y)の最も基本的な座標を利用して演算を行う。3 次元の Jacobian 座標系を利用して高速に点の演算を行う場合は、以下の関数を利用できる。

```
void ECp_padd(ECParam *E, ECp *A, ECp *B, ECp *ret);
void ECp_psub(ECParam *E, ECp *A, ECp *B, ECp *ret);
void ECp_pmulti(ECParam *E, ECp *A, LNm *k, ECp *ret);
```

ただし、この場合は最初に LN_long_set(a->z,1);のように点の z 座標に 1 を代入しておき、得られた結果に対して、ECp_proj2af(E, a);を実行して元の 2 次元座標に戻す必要がある。

```
/* 例: E はパラメータ, a,b,ret は点,k は巨大数 */
LN_long_set(a->z,1);
ECp_pmulti(E,a,k,ret);          /* a を k 倍する */
ECp_proj2af(E,ret);             /* ret を元の座標系へ戻す */
```

4.4.2 ecdsa（楕円 DSA 公開鍵署名）

楕円曲線暗号を利用した署名方式である ECDSA の扱いについて解説する。

```
typedef struct Public_key_ECDSA{
  int       key_type; /* key identifier */
  int       size;

  ECp               *W;/* public Point */
  ECParam           *E; /* EC Parameter */
}Pubkey_ECDSA;
```

```
typedef struct Private_key_ECDSA{
  int       key_type; /* key identifier */
  int       size;

  int       version;
  ECp               *W;       /* public Point */
  LNm               *k;       /* private base integer */
  ECParam           *E;       /* EC Parameter */
 /* DER encode strings */
 unsigned char *der;
}Prvkey_ECDSA;
```

ECDSA では、楕円離散対数問題 W=kG をベースに公開鍵署名を実現している。すなわち、G はパラメータの底点であり k は k<r(G の order)のランダムな巨大数(秘密鍵)であり、W が公開鍵(点)である。

以下に用意された関数を使って署名を行う例を挙げる。

```
ECParam *E;
Prvkey_ECDSA *prv;
char *buf, *text="this is sample.";

E=ECPm_get_std_parameter(ECP_TYPE_X9_62_192); /* 192bit パラメータ */
prv = ECDSAprvkey_new();
ECDSAprv_generate(E,prv);           /* ECDSA のプライベートキーを作成 */
buf=ECDSA_get_signature(E, prv,text, strlen(text)); /* 署名を行い buf にセット */
```

この例では、秘密鍵を新たに作成し、その鍵を使用して ECDSA の署名を行っている。バージョン 1.6 では ECDSA の秘密鍵をファイルに保存する関数がないため、実際に証明書に対しての応用は難しい。

次に、ECDSA の署名の検証の例を以下に示す。

```
/* それぞれの変数は上記の例と同様 */
Pubkey_ECDSA *pub;

ECDSAprv_2pub(prv, pub);
if(ECDSA_vfy_signature(E,pub,text,strlen(text),buf))
        printf("error : ECDSA signature test¥n");
```

関数 ECDSA_vfy_Signature()は正常終了した場合は 0 を返し、署名に異常があれば 1 を返すように設計されている。

4.4.3 楕円曲線生成

AiCrypto バージョン 1.7 より、CM 法（虚数乗法）による楕円パラメータ生成が可能になった。参考にした主な文献はソースファイル ecc_gen.c のなかに記述してある。ここでは、新しく楕円パラメータを生成する方法を以下に示す。

```
/* ok_ecc.h を include する */
ECParam *E;
E = ECPm_gen_parameter(192); /* bit 数を指定(8 の倍数である事) */
if(ECPm_verify_parameter(E))
```

```
        printf("verification error!¥n");
    ECPm_free(E);
```

パラメータを生成する場合、指定した bit 数を持つ素数を法としたガロア体上で楕円曲線の生成を行う。ここで行っている事は、ヒルベルト類多項式 $H_D(x)$の解が判っている D=11,19,43 に対して法 p を基に基本となる a,b を計算し、さらにこの a,b のツイスト楕円曲線を計算した上で、底点の総当りを行う。ここで、適当な底点が無ければツイストを作りなおし再び総当りを行う。これを何回か繰り返しても望ましい結果が得られない場合は、素数 p を新たに作り直してやり直す方法を取っている。

この方法では、パラメータ生成時間に大きなばらつきがあり、PIII 700MHz, Solaris2.8, Gcc2.8.1を使用した場合、192bit のパラメータ生成で最小では 1 秒以下、最大では 30 秒程度の時間がかかっている。また、1000 個のパラメータ生成(192bit)を連続して行った結果、7085 秒で終了した。すなわち、１つの曲線生成、底点探索、パラメータ検証の全てを平均 7 秒で終えており、実時間で十分実用的である。

また、P1363 をベースにしてパラメータの検証を行うが、この場合 CM 法を使用しているので、トレース t を大きく取る事ができるため、FR 帰着等の攻撃に強い。また、MOV 帰着への安全性チェックも行っており、安全性のチェックは十分だと考えられる。

なお、このパラメータ生成を実行チェックする場合は、ecc ディレクトリ内で

```
abc% make –f Makefile.test
```

を実行すると ts というコマンドが生成されるため、このコマンドによって ecc のチェックが行える。（パラメータの生成も行うが、速度計測用ではない）

4.5 des, Triple-des, RC2（共通鍵暗号）

共通鍵暗号の関数の扱いについて解説する。まず、キーの構造体はそれぞれ以下の通りである。

```
Typedef unsigned long long ULLONG;

Typedef struct crypt_DES_key{
    int   key_type;/* key identifier */
    int   size;

    ULLONG      list[16];
    ULLONG      iv;
}Key_DES;
```

```
Typedef struct crypt_3DES_key{
    Int    key_type;
    Int    size;

    ULLONG      list1[16];
    ULLONG      list2[16];
    ULLONG      list3[16];
    ULLONG      iv;
}Key_3DES;
```

```
typedef struct crypt_RC2_key{
    int    key_type;
    int    size;

    unsigned short S[64];
    unsigned short iv[4];
}Key_RC2;
```

DES、Triple-DES、RC2 のどれも全て 64bit を 1 ブロックとして暗号化する。DES のプログラムではその 1 ブロックを unsigned long long（64bit）毎に扱う（Windows では_int64)。また、RC2 では unsigned short[4] を 64bit1 ブロックとして扱う。リトルエンディアンとビッグエンディアンの扱いの違いが有るが、aicrypto の関数を使う限り支障なく暗号化と復号化が出来る。

まず、DES のキー生成は、

```
unsigned char key1[] ={
   0x64,0x32,0x32,0x46,0xfa,0x01,0x43,0x25};
Key_DES *dk1;
dk1=DESkey_new(8,key1);
```

のように行う。DESkey_new の第 2 引数が、char 配列で、第１引数が配列の個数である。DES のキー生成には 64bit のブロックが必要なので上記のようにする。もし、配列の個数が 8 個以上だと、それ以降のバイト列は無視され、それ以下だと残りには 0 が入力される。

次に、Triple-DES のキー生成は、

```
/* dk1,dk2,dk3 は確保済みとする  */
Key_3DES *tridk;
tridk=DES3key_new(dk1,dk2,dk3);
```

のように行う。Triple-DES は 3 つの DES キーを使って、暗号化、複合化、暗号化を順に行っている。そのため、このキー生成関数にも、3 つの生成された Key_DES が必要になる。なお、3 番目の引数に NULL を入力すると、第一引数の DES キーを 3 つ目のキーとして選択する。

そして、RC2 のキー生成は、

```
unsigned char key[]={
   0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
   0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
Key_RC2 *rc2k;
rc2k = RC2key_new(16,key);
```

のように行う。RC2 のキー生成には最大 128byte までの char 配列が使用される。構造体で表記されているように、unsigned short S[64]がキーとして使用されるので、かなり大きな文字列をパスワードとして使用する

ことが可能である。なお、引数の型は DES と同様である。

キーの生成が終わったら次に暗号化、複合化を行う。DES では ECB,CBC,CFB のモードに対応しており、Triple-DES、RC2 では ECB,CBC のモードに対応している。まず、一番簡単なブロック毎の暗号化(ECB)を DES で使用するには、

```
unsigned char retc[8],inc[]={
   0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};
unsigned char cry[8];

DES_ecb_encrypt(dk1,8,&inc,&cry);
DES_ecb_decrypt(dk1,8,&cry,&retc);
```

のように暗号化、復号化の関数を使用する。それぞれの関数で、第 1 引数は DES のキーポインタである。第３、第４引数は char 型の入力配列と出力配列であり、入力配列の個数を第２引数で指定する。DES は 64bit ブロックなので、第 2 引数は 8 の倍数が好ましい。それ以外の数値の場合は、特にパディング処理される事なく実行されるため、メモリの大きさには注意が必要である。

これが Triple-DES ならば、

```
DES3_ecb_encrypt(tridk,8,&in,&cry);
DES3_ecb_decrypt(tridk,8,&cry,&ret);
```

と DES の関数と同様の扱いで暗号化、復号化が出来る。

また、RC2 での暗号化と、復号化は以下のように行う。

```
unsigned char in[8],cry[8],ret[8];

RC2_ecb_encrypt(rc2k,8,in,cry);
RC2_ecb_decrypt(rc2k,8,cry,ret);
```

RC2 は 64bit ブロックなので、入力の配列の個数は 8 の倍数であるのが好ましい。もしそれ以外の数値が入力された場合、あまった場所を 0 で補完して暗号化する。ECB モードは以上の通りである。

CBC モードは、暗号化された前のブロックと現在のブロックの排他論理和を暗号化する方式である。この方式は同じ文字列でも異なった文字に暗号化されるため、強度が高い方式である。しかし、逆に途中 1bit でもエラーが存在すると、そのブロック以降は全く復号化出来ない弱さがある。

この方式は、iv（initialize vector:初期化ベクトル）が必要なため、それぞれの暗号化では、キーの生成後に iv をセットしなくてはならない。まず、DES の場合は、

```
/*  キーは生成済み  */
unsigned char ivc[8]={ /*  初期ベクトルは 64bit 必要（配列個数の宣言）  */
   0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77};
DES_set_iv(dk1,ivc);
DES_cbc_encrypt(dk1,8,&in,&cry);
DES_cbc_decrypt(dk1,8,&cry,&ret);
```

のように初期ベクトルをセットし、CBC モードの暗号化、復号化を行う。DES,Triple-DES,RC2 のどの暗号化でも初期ベクトルには 64bit が必要なため、配列の初期値には注意しなくてはならない。

同様に Triple-DES の場合は、

```
DES3_set_iv(tridk,ivc);
DES3_cbc_encrypt(tridk,8,&in,&cry);
DES3_cbc_decrypt(tridk,8,&cry,&ret);
```

のように行う。さらに、RC2 の場合、

```
/* in,cry,ret は unsigned char 配列  */
RC2_set_iv(rc2k,ivc);
RC2_cbc_encrypt(rc2k,8,in,cry);
RC2_cbc_decrypt(rc2k,8,cry,ret);
```

のようにすれば、CBC モードで暗号化、復号化が出来る。DES では CFB モードを実行可能であるが、ここでの説明は省くものとする。

これらのキーを開放するには、それぞれ

```
DESkey_free(dk1);
DES3key_free(tridk);
RC2key_free(rc2k);
```

を実行することで、メモリの開放を行うことが出来る。なお、Triple-DES のキーを生成するために、3 つの DES キーを生成しているので、それらを先に DESkey_free()で開放しておく方が良い。なお、これらの動作を確認するなら、それぞれのディレクトリに test.c のようなファイルが存在するので、Makefile.test で make を行い、作成された実行ファイルを実行すれば良い。

4.6 md2,md5,sha1,hmac（ハッシュ関数）

ハッシュ関数では、直に接する構造体がないため、構造体の解説は行わないものとする。
実際に、動作させるには以下のようにする。

```
unsigned char ret[16];
char in[]=”abc”;
OK_MD5(strlen(in),in,ret);
```

上記の例では MD5 を使用している。OK_MD5 の第 2、第 3 引数が、それぞれ char 配列の入力と出力であり、第 1 引数が入力配列の個数を示している。MD5 は 128bit を返り値として必要とするので、char で 16byte の領域を確保する必要がある。同様に、MD2,SHA1 を使用するプログラムは以下の通り。

```
unsigned char ret[20];
OK_MD2(strlen(in),in,ret);
OK_SHA1(strlen(in),in,ret);
```

また、異なった文字配列を一つの文字列として入力し、出力を得るならば

```
char in2[]=”defghijk”;
MD5_CTX ctx;
MD5Init(&ctx);
MD5Update(&ctx,in,strlen(in));
MD5Update(&ctx,in2,strlen(in2));
MD5Final(ret,&ctx);
```

といった用法も用意されている。上記では MD5 の例を挙げたが、MD2 や SHA1 でも同様なものが用意されている。

次に HMAC の扱いを述べる。HMAC は鍵付きのハッシュ関数であり、実際のハッシュ関数部分は MD5 や SHA1 などに依存している。入力されるキーの長さは 64byte までだが、それ以上の入力があった場合はハッシュ関数に通され縮小化される。

```
char key[]=”sample password”;
HMAC_MD5(strlen(in),in,strlen(key),key,ret);
HMAC_SHA1(strlen(in),in,strlen(key),key,ret);
```

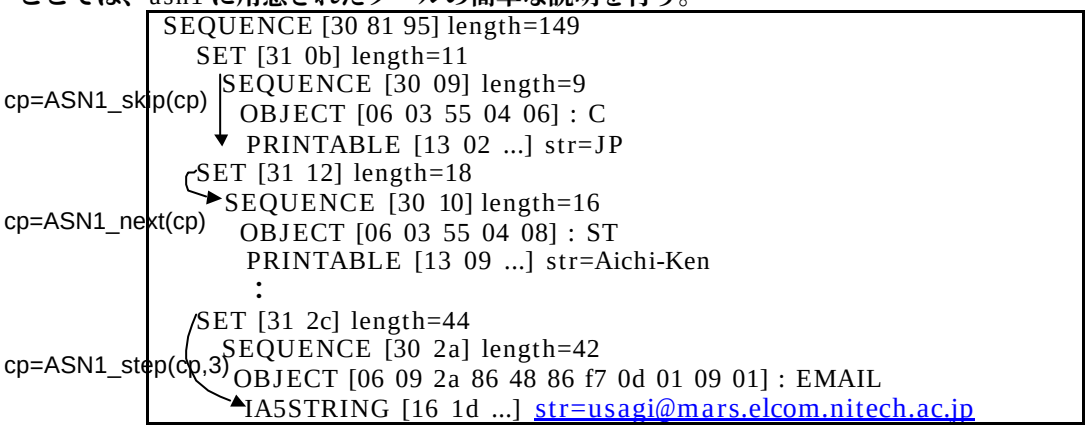
HMAC を使用する場合は、上記のように、キーを設定し使いたいハッシュ関数を使用した HMAC の関数を記述する。HMAC_*()の第 1、第 2 引数で、入力配列数、入力配列、第 3、第 4 引数でキー配列数、キー配列、そして最後の引数が出力配列となる。ここでいう配列は全て char 型である。

4.7 asn1（DER 解析、生成）

asn1 では、DER 文を解析し Cert 構造体、CRL 構造体、Prvkeyk_RSA 構造体、を返す関数群が記述されている。逆にそれらの構造体が与えられた時に DER 文を生成するプログラムは、x509 や rsa などで記述されている。

4.7.1 DER 文解析

ここでは、asn1 に用意されたツールの簡単な説明を行う。



上図の通り、DER 文のバイナリ配列が渡された場合、先頭の正しい TAG を辿っていかないと、正確な解析を行うことが出来ない。DER 文の先頭から、 30 81 95 31 0b 30 … とバイナリ配列が続くのだが、これを unsigned char *der で渡した時、下記の関数で文字列を取り出すことが出来る。

```
int read_der(unsigned char *der){
  char *buf,*cp;
  int  i;

  cp = ASN1_step(der,4);
  buf = ASN1_printable(cp,&i);
  printf("%s¥n",buf);
}
```

この関数では、先頭から SEQUENCE→SET→SEQUENCE→OBJECT→PRINTABLE の順に４つタグをステップして、PRINTABLE の先頭で、ASN1_printable(cp,&i)の関数を呼び出すとその返り値が文字列となる。なお、この関数は"JP"を結果として表示する。また、&i にはそのタグ全体のバイナリ文字列数が代入される。すなわち、このタグは 13 02 .. .. の 4byte で表されるので、i には 4 が代入される。

ASN1_printable に渡すバイナリ列の先頭は、Printable String を示す 0x13 を持っていなくてはならない。もしこれ以外の値を持っていた場合、この関数は NULL を返す。

4.7.2 DER 文生成

次に、DER バイナリ文の作成について解説をする。

```
int write_der(unsigned char *ret, int *ret_len){
  char *cp,str[]="test.";
  int i,j;

  ASN1_set_integer(100,ret,&i);
  cp=ret+i;
  ASN1_set_ia5(str,cp,&j);
  i+=j;
  ASN1_set_sequence(i,ret,ret_len);
}
```

上記の関数では、充分な領域を確保した*ret を関数に与えると、

```
SEQUENCE [30 0a] length=10
  INTEGER[06 01 64] int=100
  IA5STRING [16 05 ...] str=test.
```

のような DER 文が作成され返される。また、数値としてそのバイナリ文の文字数が返される（この場合、12 が返り値である）。

このような、基本的な関数を使用して、証明書や CRL、秘密鍵の DER 文を解析し、構造体を返す関数が、

```
Cert *ASN1_read_cert(unsigned char *in);
```

```
CRL *ASN1_read_crl(unsigned char *in);
Prvkey_RSA *ASN1_read_rsaprv(unsigned char *in);
```

のように用意されている。上記のほか、DER 文には多くのタグのタイプがあるが、これらを使用する場合は ok_asn1.h の中で関数の宣言がされているので、それを参考にして使用すれば良い。ただし、Cert 構造体など、ok_x509.h のインクルードが必要な関数に関してはコメントアウトされている。

4.8 X.509（構造体生成とツール群）

4.8.1 各種構造体

証明書の Verify で使われる期限チェックや署名チェックなどを行う上で、何かしらの構造体で情報を保持し受け渡しする方が、プログラムの構造として良質だといえる。そこで、Cert、CRL、Key の各構造体を用意した。

```
typedef struct x509_certificate{
   long   version;
   long   serialNumber;

   int    signature_algo;
   char   *issuer;
   CertDIR          issuer_dir;

   Validity         time;

   char   *subject;
   CertDIR          subject_dir;

   int    pubkey_algo;
   Key    *pubkey;

   long   issuerUniqueID;
   long   subjectUniqueID;

   CertExt          *ext;

   int    siglen;
  unsigned char *signature;

  /* DER encode strings */
  unsigned char *der;
}Cert, Req;
```

```
typedef struct crypt_key{
   int   key_type; /* key identifier */
   int    size;

  /* type field */
}Key;
```

```
typedef struct validity{
   struct tm   notBefore;
   struct tm   notAfter;
}Validity;

typedef struct certificate_extension CertExt;
struct certificate_extension{
   int    extnID;
   int    critical;
   int    vlen;
  unsigned char *extnValue;
   unsigned char *objid;

   CertExt          *next;
};

typedef struct certificate_dir{
   int    num;
   int    strk[10];
   int    tkind[10];
   char   *tag[10];
}CertDIR;
```

```
typedef struct revoked_list Revoked;
struct revoked_list{
   int    serialNumber;
   struct tm   revocationDate;

   Revoked          *next;
};


typedef struct x509_crl{
   int    signature_algo;
   char   *issuer;
   CertDIR          issuer_dir;

   struct tm   lastUpdate;
   struct tm   nextUpdate;

   Revoked          *next;

   CertExt          *ext;

   int    siglen;
  unsigned char *signature;

  /* DER encode strings */
  unsigned char *der;
}CRL;
```

まず、Cert 構造体から説明する。Cert 構造体は、X.509v3 の形を模倣しており、version、serialNumber 等によって構成されている。CertDIR は Issuer や Subject のディレクトリ情報を保持しており、それをテキスト化したものを*issuer、*subject のように文字列として保持する。これは、証明書同士の比較等に使われる。signature_algo にはシグネチャアルゴリズムを示す数値が入る。この数値は ok_asn1.h に定義されているものを使用し、MD5withRSA ならば OBJ_SIG_MD5RSA を使用する。Validity は証明書の有効期限を保持する構造体で、その要素はそれぞれ struct tm を利用して時間を表わしている。

CertExt 構造体は、X.509v3 で使われる拡張フィールドを保持するためのものでリスト構造をしている。その extnID には ok_asn1.h で定義されている OBJ_NS_CERT_TYPE 等が代入される。der は char 型のポインタで、生成された物やファイルから読み込んだ DER 文を保持している。

この構造体は、証明書と証明書要求の両方で使われる。ファイルを読み込んだ時点で、Issuer が定義されていたら証明書、そうでなければ証明書要求だと判断する。

Key 構造体は、上記の通り形だけを持っていて構造体のキャスト用に使われる。 実際の中身は DES や RSA の方で定義されており、key_type で構造体の中身を判別する。この key_type には key_type.h で定義されている KEY_RSA_PUB、KEY_RSA_PRV といった値が代入される。

CRL 構造体も、X.509 の破棄リストと同様な構造をしている。また、シグネチャアルゴリズムは Cert 構造体と同様の値を使用し、CertDIR も Cert 構造体と同様に使用している。lastUpdate、nextUpdate には UTCTIME の文字列をそのまま保持している。破棄された証明書の情報は Revoked 構造体が持っており、この構造体はリストとして CRL に保持される。Revoked 構造体は、シリアルナンバと UTCTIME の文字列で破棄日を保持している。

4.8.2 ファイル読み込み

PEM 形式や X.509 DER 形式のファイルを上記の構造体に取り込むには以下の関数群を使えば良い。

```
Cert *PEM_read_cert(char *fname);
```

```
CRL *PEM_read_crl(char *fname);
Cert *PEM_read_req(char *fname);
Prvkey_RSA *PEM_read_rsaprv(char *fname);

Cert *ASN1_read_cert(unsigned char *in);
CRL *ASN1_read_crl(unsigned char *in);
Cert *ASN1_read_req(unsigned char *in);
Prvkey_RSA *ASN1_read_rsaprv(unsigned char *in);
```

このうち、上の４つが PEM 形式のファイルから読み込む関数で、引数にはファイル名を与える。返り値はそれぞれ自動的にメモリが確保された構造体のポインタが返される。もしファイルが存在していないか、形式が異なるため読めなかった場合は NULL が返される。下の４つは引数に DER 文字列の先頭を必要とする。そのため、

```
unsigned char *ASN1_read_der(char *fname)
```

をつかって予めファイルから DER を読み込んでメモリを確保したものを、それぞれの関数に渡すような構成になっている。

4.8.3 DER 文生成

証明書構造体の中身を更新したりした場合、新しく DER 文を生成しなくてはならない。これを行う一連の関数群は cert_asn1.c, req_asn1.c, crl_asn1.c にプログラムされている。その中でも、署名を生成して完全な X.509 DER を返す関数は以下の通りである。

```
unsigned char *Cert_toDER(Cert *ct,Key *prv,unsigned char *ret,int *ret_len);
unsigned char *Req_toDER(Req *req,Key *prv,unsigned char *ret,int *ret_len);
unsigned char *CRL_toDER(CRL *crl,Key *prv,unsigned char *ret,int *ret_len);
```

ct や req は DER を生成する元の Cert 構造体である。第 2 引数の Key は、Cert_toDER では CA の秘密鍵を渡す必要があるが、Req_toDER では自身の秘密鍵を与えるという違いがある。なお、第 3 引数には、充分なメモリを確保したバッファを渡すと、そこに DER 文を生成する。この場合の返り値は、受け渡した値 ret と同じであり、エラーがあれば NULL が返される。

また、第三引数に NULL を渡した場合は自動的にメモリを確保してポインタが返される。

```
cert->der = Cert_toDER(cert,pkey,NULL,&i);
```

同様に、CRL_toDER の第 2 引数は CA の秘密鍵を必要とし、第 3 引数には NULL を渡して

```
crl->der = CRL_toDER(crl,pkey,NULL,&i);
```

使用するのが良い。

なお、RSAprv_toDER は、rsa/rsa_asn1.c に記述されており、使い方は上記とほぼ同様である。

4.8.4 ファイル書き込み

ファイルを書き込む場合、次のような関数がある。

```
int PEM_write_cert(Cert *cert,char *fname);
int PEM_write_crl(CRL *crl,char *fname);
int PEM_write_req(Cert *req,char *fname);
int PEM_write_rsaprv(Prvkey_RSA *rsa,char *fname);
```

これらの関数群は、それぞれが保持している der をそのまま PEM の形で書き出す関数である。秘密鍵を書き出す場合は、パスワードを読み込み DES-EDE3-CBC の形で暗号化、保存する。関数の実行にエラーが無い場合は、0 が返され、エラー時には-1 が返される。

もし、X.509 DER の形式で保存したい場合は

```
int ASN1_write_der(unsigned char *der,char *fname);
```

を使用して、

```
ASN1_write_der(cert->der,”a.cer”);
```

のようにすれば DER のまま保存することが出来る。

4.8.5 証明書の Verify

証明書の Verify を行う場合は以下の関数を使用する。

```
int Cert_verify(CertList *list,Cert *cert,int max_depth,int type);
```

この関数は、Verify をおこなう証明書を第 2 引数に取り、その証明書からどの深さまで chain を辿るかを指定するのが第 3 引数である。第 4 引数には、どのようなタイプの Verify を行うか設定する。

```
#define DONT_VERIFY_CRL                     0x0001
#define ALLOW_SELF_SIGN                     0x0002
#define DONT_CHECK_REVOKED                          0x0004
#define IF_NO_CRL_DONT_CHECK_REVOKED        0x0008
```

上から順に説明すると、

1 番目は CRL 自体の Verify を行わない。

2 番目は 自己署名型の証明書に対しエラーを返さない。
3 番目は 証明書の破棄チェックを行わない。
4 番目は もし CRL が無ければエラーを返さず、CRL を使用しないで Verify の継続。
を示している。それぞれの数値は論理和を使って結合することが出来る。
すなわち、

```
err=Cert_verify(list,ct,2, DONT_VERIFY_CRL | DONT_CHECK_REVOKED);
```

のように使用する。また、返り値であるエラーは次の通りで、0 ならばエラー無しである。

```
#define X509_VFY_ERR                    0x0100
#define X509_VFY_ERR_SIGNATURE          0x0200
#define X509_VFY_ERR_SIGNATURE_CRL      0x0300
#define X509_VFY_ERR_NOTBEFORE          0x0400
#define X509_VFY_ERR_NOTAFTER           0x0500
#define X509_VFY_ERR_LASTUPDATE         0x0600
#define X509_VFY_ERR_NEXTUPDATE         0x0700
#define X509_VFY_ERR_REVOKED            0x0a00
#define X509_VFY_ERR_SELF_SIGN          0x0b00
#define X509_VFY_ERR_CA_CHAIN           0x0c00
#define X509_VFY_ERR_NOT_CACERT         0x1000
#define X509_VFY_ERR_ISSUER_CRL         0x1100
#define X509_VFY_ERR_NOT_IN_CERTLIST            0x1200
#define X509_VFY_ERR_UNKOWN_SIG_ALGO            0x1300
```

これらのエラーのうちどれか 1 つを返す。なお、返ってくるエラーのうち、0～15bit までがエラーの起きた深さを表しており、16～31bit までが上記のようなエラーの値である。

4.8.6 Certlist

この Verify 関数の第１引数で使われる Certlist について説明する。この Certlist は CA の証明書や CRL のファイル名のリストが記述されている verify.idx を読み込むことで生成される。

```
typedef struct data_position{
   char   *path;
   char   *fname;
   FILE            *fp;
   fpos_t          pos;
}DataPos;

typedef struct certificate_list CertList;
struct certificate_list{
  CertList *next;
  CertList *prev;

   long   serialNumber;
   char   *subject;
   char   *issuer;

   DataPos         *cert_pos;
  DataPos          *key_pos;
   DataPos         *req_pos;
   DataPos         *crl_pos;

   Cert   *cert;
   Cert   *req;
   Key    *key;
   CRL    *crl;
};
```

上記のような構成をしており、証明書ファイルや CRL のファイルのリスト構造をしている。Verify を行う証明書の Issuer が、このリストの中にあれば、その公開鍵を利用して署名のチェックをおこなう仕組みになっている。

この verify.idx ファイルは、CA 証明書や CRL があるのと同じディレクトリに置いておき、

```
CertList *Certlist_read_list(char *path,char *fname);
```

の関数を使えば、証明書のリストを取得することが出来る。

4.9 pkcs（PKCS ファイルの解析、生成）

PKCS とは、米 RSA 社によって作成された、秘密鍵や証明書の保持について書かれた規格である。Aicrypto では、現在公開鍵暗号として RSA を使用しており、そのため、公開鍵や秘密鍵は PKCS#1 に従って保持されている。また、PKCS に従い秘密鍵の暗号化を行えるが、現在では PKCS#12 準拠の RC2-40bit、RC2-128bit、DES-EDE3-CBC といった暗号化法を扱うことができる。ただし、RC4-40bit、RC4-128bit、で暗号化された秘密鍵ファイルはアルゴリズムを持たないため復号することができない。

4.9.1 PKCS#7

PKCS#7 は S/MIME 等で文書の暗号化や証明書送信を行うための規格である。一般に、証明書は PKCS#7 Signed-DATA で保管される。この形式でファイル化されたものは *.p7s, *.p7b といった拡張子であらわされており、aicrypto で読み込み、書きこみが可能である。

なお、PKCS#7 に関しての細かな仕様については、AiCrypto S/MIME のドキュメントに記載されている。

```
PKCS7 *P7s_read_file(char *fname);
int P7s_write_file(PKCS7 *p7, char *fname);
```

上記の関数を使うことで、*.p7b のファイルを PKCS#12 の Bag-list に展開することが出来る。また、読み込んだファイルを表示するには以下の関数を使用する。

```
void P7_print(PKCS7 *p7);
```

この他、PKCS#7-DATA や PKCS#7-ENCRYPTED の DER 解析や DER 生成を行う関数が用意されている。（aicrypto では主に PKCS#12 によって使用されている）

```
unsigned char *P7_get_data(unsigned char *in,int *ret_len);
unsigned char *P7_get_decrypted(unsigned char *in,int *ret_len);
```

P7_get_data()には、PKCS#7-DATA の DER の先頭を入力する。そのデータの中身をメモリ上に確保しその先頭のアドレスを返り値として返す。P7_get_decrypted()は、PKCS#7-ENCRYPTED を復号化して返す関数である。ここで使用されるキーは、PKCS#12 の MAC 確認で使用されたパスワードを保持しているため、それを基に生成する。もし、新たにパスワードそ設定しておきたいならば、

```
void OK_set_passwd(char *pwd);
```

を使用してパスワード文字列を記憶させ、PKCS#12 に沿ったキー作成法によってキーが生成され、復号化が行える。このパスワードはメモリ上に static に記憶されているため、不要になったら、

```
void OK_clear_passwd();
```

を実行して、文字列のクリアをしなくてはならない。また、DER の生成に関しては

```
unsigned char *P7_data_toDER(int len,unsigned char *in,unsigned char *ret,int *ret_len);
unsigned char *P7_encrypted_toDER(int len,unsigned char *cry,int algo,
                                  unsigned char *ret,int *ret_len);
```

といった関数によって DER 文が生成できる。上記のどちらの関数も ret には生成された DER 文が記述されており、ここには十分に容量が確保されたメモリのポインタを渡さなければならない。また、第２引数の*in や *cry には保持するデータをもつポインタを渡し、その配列の長さを第１引数 len に渡す。

なお、暗号文のアルゴリズム（int algo）には、

```
#define OBJ_P12Pbe_3K3DES       1013
#define OBJ_P12Pbe_2K3DES       1014
#define OBJ_P12Pbe_128RC2       1015
#define OBJ_P12Pbe_40RC2        1016
```

のどれか１つを選択して渡さなければならない。

4.9.2 PKCS#8

PKCS#8 は、秘密鍵をそのまま又は暗号化して保存するためのフォーマット形式である。この形式は PKCS#12 の内部でも使用されているが、PKCS#5 で指定されたアルゴリズムを使って、秘密鍵を独立に保存する事が可能である。

```
Key *P8_read_file(char *fname);
Key *P8enc_read_file(char *fname);
```

ファイルの読み込みは、上記の関数で第 1 引数にファイル名を指定する事で行なうことができる。但し、1 つ目の関数が復号化せずに読みこみ、2 つ目が暗号化をして読み込みをする関数である。

```
int P8_write_file(Key *p8,char *fname);
int P8enc_write_file(Key *p8,char *fname);
```

また、秘密鍵（RSA 他）を PKCS#8 形式で保存するときは、上記の関数を使用する。共に、第 1 引数は秘密鍵へのポインタであり、2 つ目の引数はファイル名である。

4.9.3 PKCS#12

PKCS#12 は、チェイン上の複数の証明書、CRL、秘密鍵をすべて保持できるファイル形式である。このファイルは、証明書、CRL、秘密鍵の全てが暗号化されており、ファイルの読み込みには ImportPassword を必要とする。また、逆にファイルの書き込み時には ExportPassword を設定しなくてはならない。

下記のものが、PKCS#12 に関係する構造体である。

```
typedef struct pkcs12_Bag_list P12_Baggage;
struct pkcs12_Bag_list{
   int                type;
   P12_Baggage     *next;

   char    *friendlyName;
  unsigned char localKeyID[4];
};

typedef struct pkcs12{
   int                version;
   P12_Baggage     *bag;
}PKCS12;

typedef struct pkcs12_BagID_key{
   int                type;
   P12_Baggage     *next;

   char    *friendlyName;
   char    localKeyID[4];

   Key     *key;

}P12_KeyBag;
```

```
Typedef struct pkcs12_BagID_CERT{
   Int                type;
   P12_Baggage     *next;

   Char    *friendlyName;
   Char    localKeyID[4];

   Cert    *cert;

}P12_CertBag;


typedef struct pkcs12_BagID_CRL{
   int                type;
   P12_Baggage     *next;

   Char    *friendlyName;
   Char    localKeyID[4];

   CRL     *crl;
}P12_CRLBag;
```

PKCS#12 は規格の中で SafeBag の中に証明書や秘密鍵を全て保持している。そのため、これらの情報を効率よくアクセスするために単方向リストとして P12_Baggage を定義している。PKCS12 構造体を先頭として P12_Baggage リストがぶら下る構造をとっている。

まず最初に、PKCS#12 ファイルの扱いから述べる。

```
PKCS12 *P12_read_file(char *fname);
int P12_write_file(PKCS12 *p12,char *fname);
```

これらの関数を使用して、PKCS#12 ファイルの読み書きが行える。なお、P12_write_file()では p12 が保持している Baggage をただ単純に書き込むだけなので、事前に秘密鍵や証明書のチェインが正しいかチェックする必要がある。

PKCS#12 ファイルから PKCS12 構造体を構成するのが普通であるが、ここでは CA 証明書、User 証明書、User 秘密鍵からファイルを書き出す簡単なサンプルを著す。

```
{/* Cert *ca, *cert; Prvkey_RSA *key; とする */
   PKCS12 *p12;
   p12=P12_new();
   P12_add_cert(p12,ca,"friendly-name CA",0xff);
   P12_add_cert(p12,cert,"friendly-name USER",0xff);
   P12_add_key(p12,key,"friendly-name USER",0xff);
   P12_write_file(p12,"out.p12");
}
```

上記のようにして、リストの中に Baggage が作成される。作成された Baggage を取り出す関数は、

```
P12_Baggage *P12_find_bag(PKCS12 *p12,int type,unsigned char keyID);
```

である。ここで使われている、type には以下の値を使用する。

```
#define OBJ_P12v1Bag_PKCS8        1102 /*秘密鍵*/
#define OBJ_P12v1Bag_CERT         1103 /*証明書*/
#define OBJ_P12v1Bag_CRL          1104 /* CRL */
```

keyID は localKeyID と呼ばれる、証明書のチェインを判別するときに使用する値で、0 がそのファイルに含まれる最上位の証明書を表す。なお、P12_add_cert()等を使用した場合、ここには 0xff が代入されているので、正しい PKCS#12 ファイルを生成するにはチェインを調べておく必要がある。

PKCS#12 のファイルを書き込むときに DER 文を生成している。このときに使用する関数が以下のものであ

る。

```
Unsigned char *P12_toDER(PKCS12 *p12,unsigned char *der,int *ret_len);
```

この関数の中で、MAC の生成や PKCS#7、PKCS#8 の DER 文の生成を行っている。

この他、Dec_info 構造体などが暗号化の時に使用されているが、これは、salt や password、iv といったバイナリ列を関数の値として渡すのが大変なので、それらの情報を一まとめにしたものである。暗号化や復号化の際に必ず使われているが、ここではその説明を省略するものとする。