AiCrypto S/MIME について

22, June,2000

奥野　琢人

1. はじめに

現在、暗号を利用した規格は数多くあり、それらを利用したアプリケーションも一般的に使われるようになった。こうした規格の中でも高度な認証・暗号通信を行う SSL や S/MIME といったものは広く普及しており、これらを使えるライブラリを作成することは研究を進める上でも大変重要なことだといえる。既に岩田研究室で開発された AiCrypto に対し、S/MIME を扱える API（アプリケーションインターフェース）を付加することで、これから行われる電子公証などの研究開発をより進めやすくすることができる。

2. ライブラリの変更点

今回の S/MIME 実装の結果 AiCrypto の内部で多くの変更が行われた。その中でも次の 2 点が大きく変更された。

1 点目は、PKCS7 の構造体を新しく作り、従来あった PKCS7-Data、PKCS7-Encrypted を扱う関数の他に、PKCS7-Signed、PKCS7-Enveloped を扱うための関数を大幅に追加した。これらの他にも、PKCS7-SignedAndEnveloped、PKCS7-Digested の構造体も既に宣言されている。また、従来では PKCS12 構造体を利用して PKCS7 を扱っていたが、それとは完全に独立し、かつ PKCS12 構造体と互換性を持たせることで、PKCS7 構造体を PKCS12 構造体にキャストするだけで、既にある PKCS12 関連の関数も使えるようになっている。

2 点目は、S/MIME 関連の API が追加されたことである。これらの関数を使うことで、メッセージへの署名、暗号化、署名＋暗号化、復号化、署名の検証が簡単に行えるようになっている。ただし、S/MIME に対する API を公開しただけであり、MIME に関する API は追加されていない。ok_mime.h の中では Mail 構造体など、MIME に関する定義を行っているが、これは AiCrypto とは完全に独立しているため、今回の変更点には加えないものとする。すなわち、S/MIME の関数によって復号化されるデータは、何らかのバイナリデータではなく MIME 形式のままの BASE64 化されたバイナリデータが返ってくる。

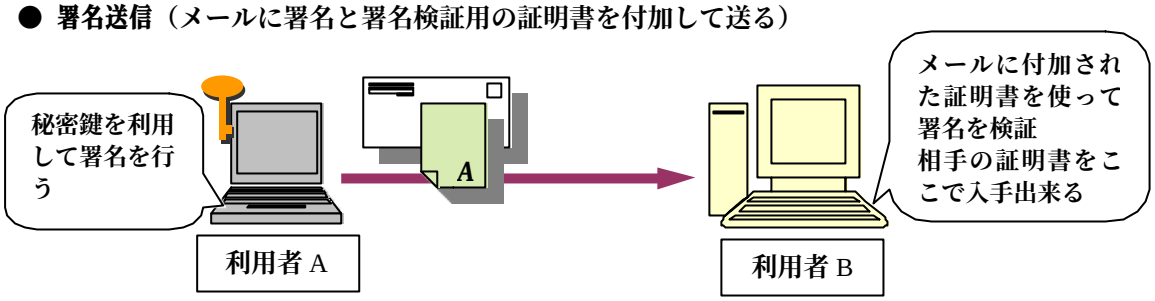
3. S/MIME について

S/MIME とは、電子メールシステム上で高度な認証や暗号通信を行うための規格であり、米 RSA 社によって提案されたものである。現在、S/MIME は多くのメールクライアントに実装されており、Outlook Express や Netscape Messenger といったアプリケーションで簡単に使用することができる。この S/MIME により行えることは、以下の通りである。

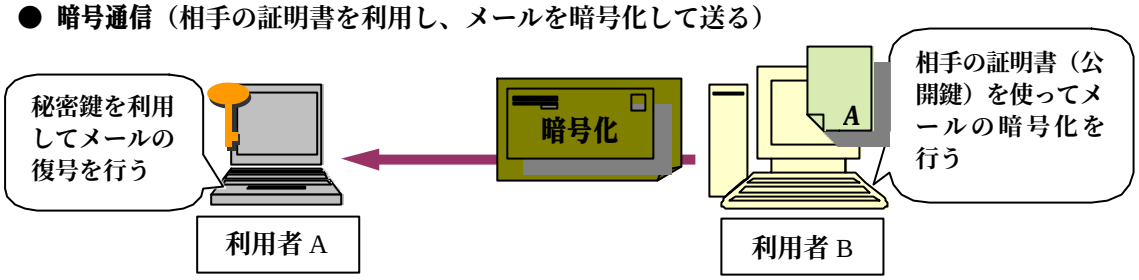
1. 送信者が自己の秘密鍵を利用して、データに署名を行える（本人証明）
2. 相手の証明書（公開鍵）を利用して、データを暗号化できる（暗号通信）
3. データのダイジェストを相手に送信することで、データの改竄チェックができる

また、実際の通信手順は以下の図の通りである。

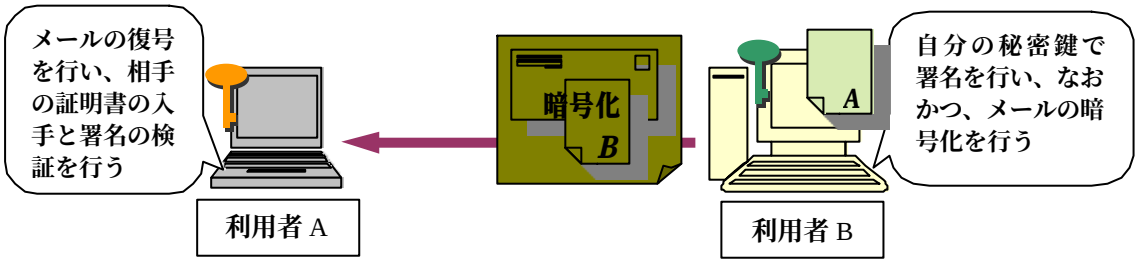
● **署名送信**（メールに署名と署名検証用の証明書を付加して送る）



● **暗号通信**（相手の証明書を利用し、メールを暗号化して送る）



● **署名＋暗号通信**（メールに署名と証明書を付加し暗号化して送る）



上記のような 3 通りの通信手段により、認証、暗号通信は行われる。ただし、メッセージ本体と証明書の暗号化に使われる暗号は共通鍵暗号、すなわち RC2 や DES といった暗号であり、この共通鍵暗号の鍵を共有するために RSA のような公開鍵暗号が使われている。（当然、署名にも公開鍵暗号が使われている）

4. データ形式（PKCS#7）

S/MIME の中で実際に使われるデータ形式は、PKCS7-Signed、PKCS7-Enveloped という名称で定義されている。この PKCS というのは米 RSA 社によって提唱されている、公開鍵暗号、暗号通信に関する通信プロトコルやデータフォーマットの規格であり、現在では広く普及している。この PKCS の 7 番目に提唱された各種データフォーマットを使って、メール本文を ASN.1 形式（バイナリ構文）でカプセル化し、MIME に従って BASE64 化されたものが S/MIME メッセージとして相手に送られる。

```
MIME-Version: 1.0
Content-Type: multipart/signed;
        protocol="application/x-pkcs7-signature";
        micalg=SHA1;
        boundary="----=_NextPart_000_0021_01BFCBD1.EA059120"

…マルチパートで分けられたメール本文…

------=_NextPart_000_0021_01BFCBD1.EA059120
Content-Type: application/x-pkcs7-signature;
        name="smime.p7s"
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
        filename="smime.p7s"
```

図 4.1 multipart による署名

```
MIME-Version: 1.0
Content-Type: application/x-pkcs7-mime;
        smime-type=signed-data;
        name="smime.p7m"
```

図 4.2 signe-data（MIME 化）による署名

上記に示されるのが、「署名」を行った場合の MIME ヘッダ部分である。ここに示されている通り、2 通りの方法がありえる。1 つは、multipart/signed としてマルチパートで MIME メッセージを構成し、メッセージ部分と署名部分を別々に構成する方法である。もうひとつは、application/pkcs7-mime として宣言をし、smime-type で(PKCS7) signed-data を使用することを宣言する方法である。この場合、”smime.p7m”としてファイル名が定義されているが、実際の中身は smime.p7s であり、かつその中にメッセージをカプセル化して保持している。なお、AiCrypto S/MIME では後者のほうを標準として使用する。また、このライブラリでは MIME に関する処理を標準ではサポートしないため、multipart で分けられた署名が送られてきた場合は、外部で multipart を分解したあとで、smime.p7s 本体を関数に渡す必要がある。

上記に示されるのが「暗号通信」を行った場合の MIME ヘッダ部分である。Content-Type は application/pkcs7-mime で宣言され、なおかつ smime-type は(PKCS7) enveloped-data で宣言されている。

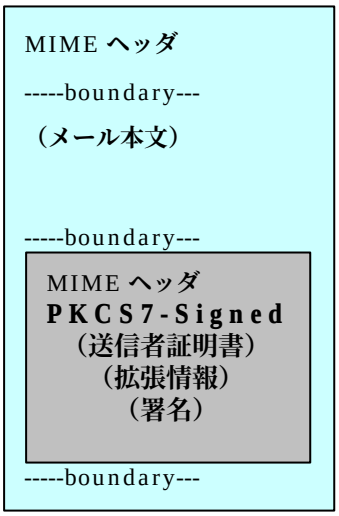
```
MIME-Version: 1.0
Content-Type: application/x-pkcs7-mime;
        smime-type=enveloped-data;
        name="smime.p7m"
```

図 4.3 enveloped-data（MIME 化）による暗号化通信

すなわち、「署名」で使用されるのは PKCS7-Signed 形式のデータフォーマットであり、「暗号通信」に使われるのが、PKCS7-Enveloped 形式のデータフォーマットである。また、「署名＋暗号通信」を構成する方法は、最初に、メッセージ本文を PKCS7-Signed でカプセル化、署名を付加する。このバイナリデータを図 4.2 に示される形で MIME 化した物を暗号化し、PKCS7-Envelope でカプセル化する。その上で図 4.3 の形式で MIME データとして、相手に送

信をおこなう。

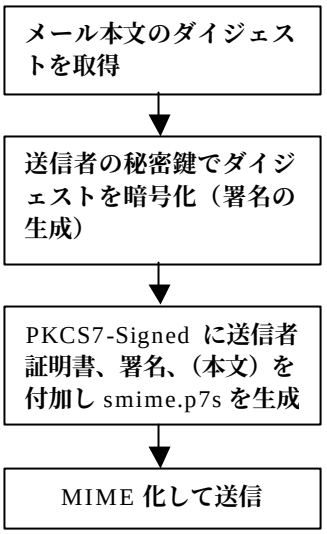
● **署名を行った場合のデータ形式**



署名（形式 1）

署名（形式２）

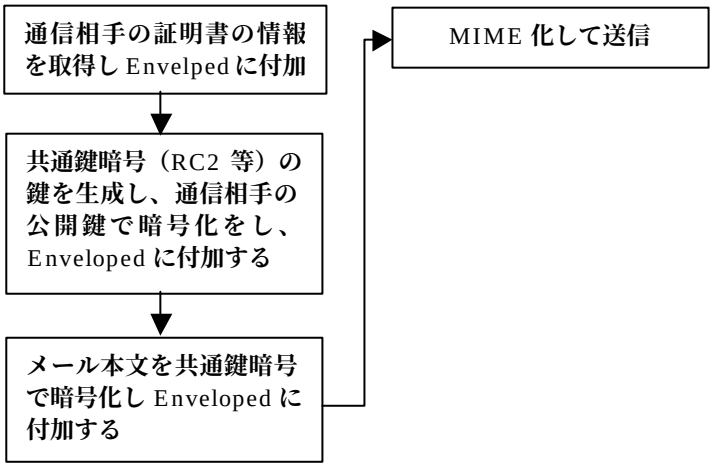
作成手順



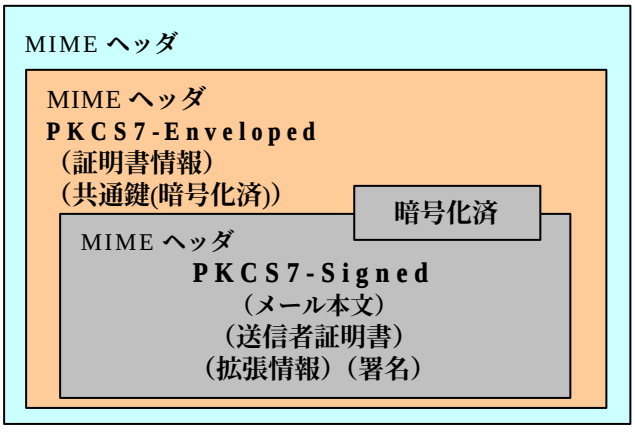
● **暗号通信を行った場合のデータ形式**

暗号通信

作成手順

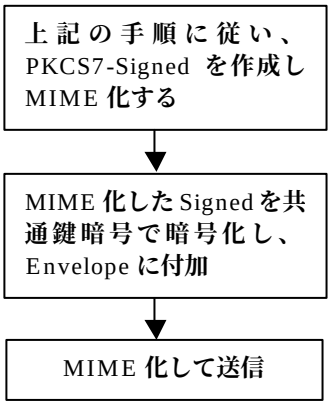


● **署名＋暗号通信を行った場合のデータ形式**



署名＋暗号通信

作成手順



5. PKCS7 構造体

前節に記述した通り S/MIME では PKCS#7 によって定義された Signed-Data-Type と Enveloped-Data-Type をデータフォーマットとして使っている。AiCrypto では、従来 PKCS12 構造体により PKCS7-Signed を扱っていたが、これは複数証明書をファイルで配布するために使用する*.p7b（PKCS7-Signed 形式）を扱えるだけの機能しか保有していなかった。そこで、今回新たに PKCS7 構造体を定義しなおし、その構造体を基に関数群を作成した。

```
/*---- PKCS7 Structures ----*/
typedef struct PKCS7ContentInfo P7_Content;
struct PKCS7ContentInfo{
   int                p7type;
   int                size;
   unsigned char   *data;
};

typedef struct pkcs7{
   int                version;
   P12_Baggage    *bag;
   P7_Content      *cont;
   unsigned char   *der;
}PKCS7;
```

上図に示されるのが、PKCS7 構造体である。PKCS7 構造体は、PKCS12 構造体とコンパチブルに作られている。すなわち、PKCS7 構造体が複数証明書を保持する場合、P12_Baggage として保持するため、void P12_add_cert(PKCS12 *p12,Cert *ct)といった関数に対し、

```
void sample_func(Cert *cert){
        PKCS7 *p7s;
        P7s=P7_new(OBJ_P7_SIGNED);
        P12_add_cert((PKCS12*)p7s,cert);
}
```

といった操作が可能である。また、P7_Content も実際には P7_Signed や P7_Envelope といった構造体がポインタで渡されており、カレントで保持している構造体のタイプは p7type によって判別することが出来る。

```
typedef struct authenticatedAttribute AuthAtt;
struct authenticatedAttribute{
   AuthAtt          *next;
   int                der_size;
   unsigned char   *der;
};

typedef struct P7_SignedDataType{
   int      p7type;

   int      version;
   int      digest_algo;

   int      cnt_size;
   unsigned char *content; /* (option?) */

   SignerInfo       *signer;
}P7_Signed;
```

```
typedef struct setOfSignerInfo SignerInfo;
struct setOfSignerInfo{
   SignerInfo       *next;

   int                version;
   int                serialNum;
   CertDIR          iss_dir;
   char               *iss_str;
   int                digest_algo;

   AuthAtt          *auth;

   int                enc_algo;
   int                sig_size;
   unsigned char   *signature;

   AuthAtt          *unauth;
};
```

上図に示されるのがP7_Signed構造体である。通常、この構造体はPKCS7内部のP7_Contentにポインタで受け渡される。その内部の変数に対するアクセスは、次のようにキャストを行えば問題なく行うことができる。

```
void sample_func(PKCS7 *p7s){
        P7_Signed *sig;
        int i,j;
        sig=(P7_Signed*)p7s->cont;
        i=sig->version;
        j=sig->digest_algo;
}
```

また、p7typeをチェックしOBJ_P7_SIGNEDであることを確かめてからアクセスを行うことで、メモリエラーを防ぐことが出来る。

```
typedef struct EncryptedContentInfo{
  int               type;
  int               enc_algo;
  int               iter;
  int               iv_size;
  unsigned char     *iv;

  int               size;
  unsigned char     *data;
}EncCntInfo;

typedef struct P7_EnvelopedDataType{
  int               p7type;
  int               version;
  RecipInfo         *recipi;
  EncCntInfo        *encCnt;
}P7_Envelope;
```

```
typedef struct RecipientInfo RecipInfo;
struct RecipientInfo{
  RecipInfo         *next;

  int               version;
  int               serialNum;
  CertDIR           iss_dir;
  char              *iss_str;

  int               enc_algo;
  int               size;
  unsigned char     *key;
};
```

上図に示されるのがP7_Envelope構造体である。この構造体も、P7_Signedと同様に、与えられたPKCS7内部のP7_Contentのポインタをキャストして使用する。なお、P7_SignedやP7_EnvelopedをPKCS7構造体に確保する方法は、

```
PKCS7 *p7s = P7_new(OBJ_P7_SIGNED);
PKCS7 *p7m = P7_new(OBJ_P7_ENVELP);
```

のようにすれば、動的にPKCS7内部のメモリを確保することが出来る。このP7_newに与える引数は、

```
#define OBJ_P7_DATA             911
#define OBJ_P7_SIGNED           912
#define OBJ_P7_ENVELP           913
#define OBJ_P7_SIGandENV        914
#define OBJ_P7_DIGESTED         915
#define OBJ_P7_ENCRYPTED        916
```

のように宣言されている。

5.1 p7s、p7m ファイルの扱い

P7_signedやP7_envelopedでは、それぞれをファイル化することができ、p7sまたはp7b、p7mというファイル拡張子を持っている。これらのファイルをやり取りするための関数を以下

に示す。P7s または P7b に関しては、P7s_read_file や P7s_write_file にてファイルの読み込み、書き込みが行える。このとき、必要なデータ型は PKCS7 の P7_Signed である。

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

また、p7m に関しても同様な関数が用意されている。この時必要なデータ型は、PKCS7 の P7_Envelope である。

```
PKCS7 *P7m_read_file(char *fname);
int P7m_write_file(PKCS7 *p7, char *fname);
```

なお、旧来よりあった P7b_read_file や P7b_write_file はマクロにより、P7s 関係の関数に再定義されている。

```
#define P7b_read_file(fn)          P7s_read_file(fn)
#define P7b_write_file(pk,fn)      P7s_write_file(pk,fn)
```

5.2 P7_Signed や P7_Envelope の生成

AiCrypto S/MIME では、SMIME 関連の API を支える下請けの関数として P7_Signed や P7_Envelope の生成や DER 文の解析、生成をおこなう関数郡が用意されている。それらの主なものをここに記す。

```
PKCS7 *P7_new(int type);
void P7_free(PKCS7 *p7);
```

上記の関数は、空の PKCS7 構造体の作成を行うものと、構造体が使用しているメモリの開放を行うものである。type に与える数値は第 5 節に示してある。

```
PKCS7 *P7s_get_signed(PKCS12 *p12, unsigned char *data, int len, int digest_algo);
```

上記の関数により、全ての情報が含まれた PKCS7（P7_Signed）構造体を生成する。与える引数は、自己の証明書、秘密鍵を含む PKCS12 構造体のポインタ、メールの本文、メール本文の長さ、署名で使用するダイジェストアルゴリズムである。

```
PKCS7 *P7m_encrypt_enveloped(PKCS7 *p7b,unsigned char *data,int data_len);
```

上記の関数により、全ての情報が含まれた PKCS7（P7_Envelope）構造体を生成する。与える引数は、自己の証明書を含む PKCS7（証明書のみ保持する不完全な PKCS7 でも良い）又は、PKCS12 構造体のポインタ、メール本文、メール本文の長さである。ここでは指定しないが、通常使う共通鍵暗号は RC2（40bit）である。

```
void P7_signed_toDER(PKCS7 *p7,unsigned char *sig,int *ret_len);
void P7_envelope_toDER(PKCS7 *p7,unsigned char *env,int *ret_len);
```

上記の関数は、それぞれ PKCS7-Signed や PKCS7-Enveloped の DER を生成する関数である。これらの関数を利用するためには、構造体の中身が適当なデータによって占められている必要がある。また、第 2 引数で与えた unsigned char のバッファに DER が生成されるため、この関数を呼び出す前に十分なメモリ領域を malloc 等で確保しておく必要がある。

```
PKCS7 *ASN1_read_p7s(unsigned char *der);
PKCS7 *ASN1_read_p7env(unsigned char *der);
```

上記の関数により、与えられた DER 文より PKCS7-Signed や PKCS7-Enveloped を再構成する。なお、与えられた DER 文が正しい構造を持たない場合は、NULL を返す。

6. S/MIME に関する API について

この節では S/MIME を扱うための API について説明する。第 2 節でも記述した通り、今回の実装では S/MIME に関する API を公式に公開しただけであり、MIME についてはライブラリ使用者が独自に処理をしなくてはならない。また、証明書の Verify 等については S/MIME ではなく AiCrypto の X.509 の機能として実現しているため、AiCrypto に関するドキュメントを参照する必要がある。なお、ここで示される関数を使用するためには、ok_mime.h を include する必要がある。

6.1 署名の生成、証明書の取得

AiCrypto では、第 4 節で示された署名（形式２）に対応している。形式１を使用するためには、MIME の multipart に対応する必要があり、今回のバージョンでは multipart の分解等は行っていない。ただし、分解後に S/MIME メッセージを適当な関数に与えると、証明書の取得や署名の検証を行うことが出来る。

```
char *SMIME_p7s_set_signature(char *msg, int dig_algo, PKCS12 *p12);
```

上記の関数により、署名済み、メッセージ、証明書を含む p7s ファイルの MIME メッセージを関数の返り値として得ることが出来る。msg がメール本文、dig_algo が署名に使用するダイジェストのアルゴリズム（通常 OBJ_HASH_SHA1）、p12 が署名に使用する秘密鍵と、相手に渡す証明書である。

```
PKCS7 *SMIME_p7s_get_certs(char *msg);
PKCS7 *SMIME_p7s_get_msg(char *msg, char **ret);
```

上記の関数は、与えられた S･MIME メッセージ（msg）から証明書（PKCS7）を取り出したり、PKCS7_Signed にカプセル化されたメール本文を取り出したり（**ret）する関数である。

```
int SMIME_p7s_verify(PKCS7 *p7, unsigned char *data, int len);
```

また、上記の関数により、署名の検証が行える。SMIME_p7s_get_certs などにより得られた PKCS7 データは、証明書、署名、メール本文といった情報を全て保持しているため（もとの署名が形式２の場合）、p7 を与えただけで署名の検証を行える。すなわち、multipart で data が別である場合を除けば、SMIME_p7s_verify(p7s,NULL,0)を与えれば問題なく検証が行われる。なお、返り値は”０”が検証成功、”１”が検証失敗である。

6.2 暗号通信の実現

現在の S/MIME ライブラリでは、共通鍵暗号として RC2（40bit）を標準として使用している。これは、S/MIME を定義する RFC にも記述されている通り必要最低限の機能でしかない。ただし、Outlook Express や Netscape Messenger といったアプリケーションでも、基本的にこのアルゴリズムのみサポートしている場合が多いため、問題ないといえる。

```
unsigned char *SMIME_p7m_decrypt(char *msg, PKCS12 *p12);
char *SMIME_p7m_encrypt(char *msg, PKCS7 *p7b);
```

暗号通信を行うために必要なものは、基本的に公開鍵と秘密鍵のみである。そのため、本ライブラリでも引数は極力簡単になっている。まず、復号化に必要なものは暗号化された S/MIME メッセージと秘密鍵を含んだ PKCS12 構造体である。この 2 つを引数で与えると、復号化された

メール本文が返り値として返される。

次に暗号化に必要なものは、メール本文と相手の証明書である。証明書は P7b 形式なため、ルート CA まで含む全ての証明書を引数で渡すことが可能である（が、使用するのは相手本人の証明書のみである）。返り値は、暗号化された S/MIME メッセージである。

6.3 署名＋暗号通信の実現

署名＋暗号通信は 2 段階で行う。これを行う関数が用意されているわけではなく、今まで紹介した関数を利用して、このタイプのメッセージの生成をおこなう。すなわち、

```
/* p12 は自分の秘密鍵、p7b は相手の証明書、msg はメール本文 */
char *tmp, *ret;
tmp=SMIME_p7s_set_msg_sign(msg,OBJ_HASH_SHA1,p12);
ret=SMIME_p7m_encrypt(tmp, p7b);
free(tmp); /* 余分なメモリは開放 */
```

のような手順により、メール本文(msg)から署名＋暗号通信の S/MIME メッセージ(ret)を生成することが可能である。このメッセージからメール本文や証明書を取り出したり、署名の検証を行う方法は以下の通りである。

```
/* p12 は自分の秘密鍵、msg は S/MIME メッセージ */
PKCS7 *p7s;
int check;
char *tmp,*ret;
tmp = (char*)SMIME_p7m_decrypt(msg, p12);
p7s = SMIME_p7s_get_msg(tmp,&ret);
check = SMIME_p7s_verify(p7s,ret,strlen(ret)); /* 署名形式 2 なら(p7s,NULL,0)でも良い */
free(tmp); /* 余分なメモリは開放 */
```

このような手順により、S/MIME メッセージの復号化をおこない、そこからメール本文や証明書を取り出し、さらに署名の検証を行うことが出来る。

□