

연 + 재 + 순 + 서	연 + 재 + 가 + 이 + 드
1회 2003.12 OpenSSL 암호화 프로그래밍 첫걸음	운영체제 윈도우 98/NT/2000/XP, 리눅스, 유닉스
2회 2004.1 OpenSSL API를 이용한 비밀키 암호화, MD 프로그래밍	개발도구 에디터, C 컴파일러
3회 OpenSSL API를 이용한 공개키 암호화 프로그래밍	기초지식 기본 암호화 이론, C 프로그래밍
4회 OpenSSL API를 이용한 인증서, SSL 프로그래밍	응용분야 SSL 기반 통신, 데이터 암호화/복호화, 인증서 관리 등 암호화가 필요한 모든 프로그램



OpenSSL 암호화 프로그래밍 2

OpenSSL API를 이용한 비밀키 암호화, MD 프로그래밍

이번 호는 OpenSSL API를 사용하여 비밀키 암호화와 MD를 구현하는 것에 대해 다룰 것이다. 비밀키 암호화와 메시지 다이제스트는 암호화의 가장 기본적인 분야로 거의 대부분의 암호화 기반 구조에서 사용되며, 패스워드 저장 같이 쉽게 응용될 수 있는 분야가 많으므로 암호화 프로그래밍에서 중요하다. 이번 호를 통해 OpenSSL API가 어떠한 구조를 가지고 암호화를 수행하는지 대충 감을 잡을 수 있을 것이다.

이번 호는 OpenSSL API를 사용하여 비밀키 암호화와 메시지 다이제스트(Message Digest, 이하 MD)를 구현하는 것에 대해 다룰 것이다. 비밀키 암호화와 MD는 암호화의 가장 기본 분야로 거의 대부분의 암호화 기반 구조에서 사용되기도 하지만, 패스워드 저장 같이 쉽게 응용될 수 있는 분야가 많으므로 암호화 프로그래밍에서 중요하다. 주로 구현 위주로 진행되지만 암호화를 처음 접하는 사람을 위해 이론적인 부분도 간단히 짚고 넘어갈 것이다.

비밀키 암호화

비밀키 암호화란 비밀키를 사용한 암호화 방법을 말한다. 비밀키는 단어가 의미하듯 암호화를 하는 주체가 아닌 다른 누군가에게 알려지면 안 되는 비밀

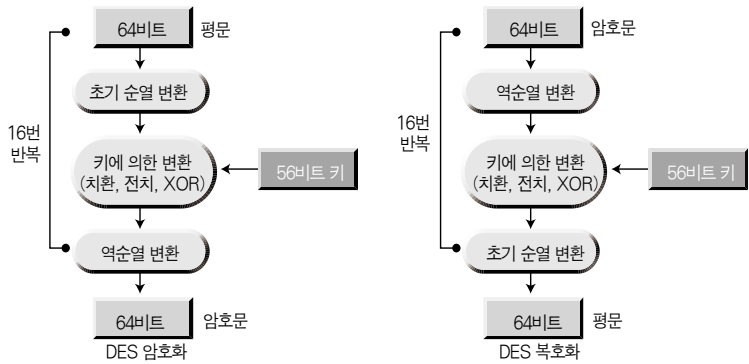
남태혁 | ntt@searchcast.net

현재 검색 엔진 전문 회사 서치캐스트의 개발 팀장으로 근무하고 있다. 멀티미디어 검색 엔진에 관련된 일을 하고 있으며, 암호화 프로그래밍에 관한 책을 집필 중이다.

스러운 키이다. 여기서 키는 물론 암호화/복호화에 사용되는 열쇠이며 키를 가진 사람만이 암호화/복호화를 할 수 있다. 비밀키 암호화는 같은 키를 사용하여 암호화/복호화를 한다는 중요한 특징을 가진다. 그래서 비밀키 암호화를 대칭 키 암호화라고도 부른다.

키는 비밀키 암호화에서 가장 중요한 역할을 한다고 할 수 있다. 이미 지난 호에서 비밀키를 생성하는 방법에 대해서 알아봤는데, 여기서는 비밀키 암호화에서 암호화/복호화를

<그림 1> DES 암호화, 복호화 과정



할 때 키가 어떻게 사용되는지 알아보도록 하자. <그림 1>은 대표적인 비밀키 암호화 알고리즘인 DES의 암호화, 복호화 과정을 나타낸 것이다.

DES 알고리즘은 56비트의 키를 이용한 치환, 전치, XOR의 방법으로 64비트의 데이터를 암호화, 복호화한다는 것을 알 수 있으며, 키의 중요성을 알 수 있을 것이다. 당연한 얘기지만 비밀키의 길이는 길면 길수록 더 안전하다. <표 1>에 현재 많이 사용되는 비밀키 암호화 알고리즘이 사용하는 키의 길이를 정리했다. 이번엔 블록에 대해 알아보자. 비밀키 암호화 알고리즘은 크게 두 가지 타입이 있다.

- ◆ 스트림 암호(stream ciphers)
- ◆ 블록 암호(block ciphers)

스트림 암호에서는 평문을 암호화할 때 한번에 하나의 비트 또는 하나의 바이트 단위로 암호화한다. RC4가 스트림 암호의 대표적 알고리즘이다. 블록 암호에서는 평문을 암호화할 때 여러 개의 비트를 묶어서 암호화한다. 이 비트의 묶음을 블록이라고 한다. DES나 AES가 블록 암호의 대표적 알고리즘이다. 사실 대다수의 비밀키 암호화 알고리즘은 블록 암호방식이라고 할 수 있다. 그리고 알고리즘에 따라 사용하는 블록 길이가 다른데, <표 1>에 각 비밀키 암호화 알고리즘이 사용하는 블록 길이를 정리했다. 블록 암호는 암호화할 때 블록을 어떻게 사용하느냐에 따라 4가지의 모드로 나뉜다.

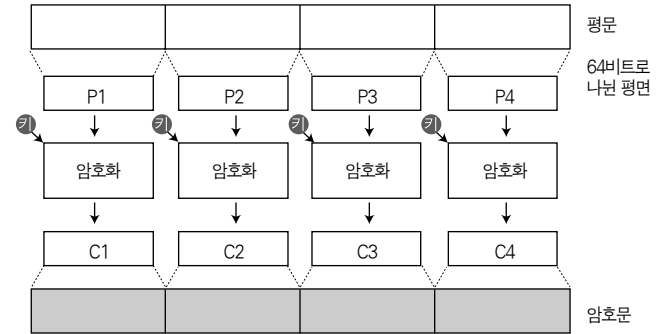
- ◆ ECB(Electric CodeBook)
- ◆ CBC(Cipher Block Chaining)
- ◆ CFB(Cipher FeedBack)
- ◆ OFB(Output FeedBack)

이중에서 가장 간단한 ECB 모드에 대해 알아보자. <그림 2>를 보면 대충 이해될 것이다.

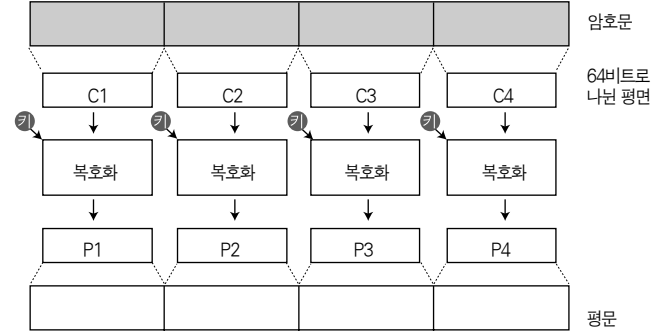
중요한 것은 암호화할 평문을 일정 길이의 블록으로 나눈다는 것이다. 만약 평문이 일정한 블록 길이로 나뉘지 않으면 '패딩(padding)'이라고 하는 빈 데이터를 마지막에 덧붙인다. 그리고 나뉜 각각의 블록을 하나씩 키로 암호화한 후 생성된 암호 블록들을 하나로 합친다. 끝으로 패딩은 제거한다. CBC 모드는 ECB 모드에 단지 XOR 연산 과정이 추가된 것이다. <그림 4>는 CBC 모드의 암호화 과정이다.

CBC 모드에서는 블록의 XOR 연산을 하기 때문에 초기 연산 블록으로 사용되는 IV(Initiation Vector)가 필요하다. 이 IV는 블록 중의 하나를 선택하거나 블록 길이와 같은 길이의 임의의 데이터를 사용한다.

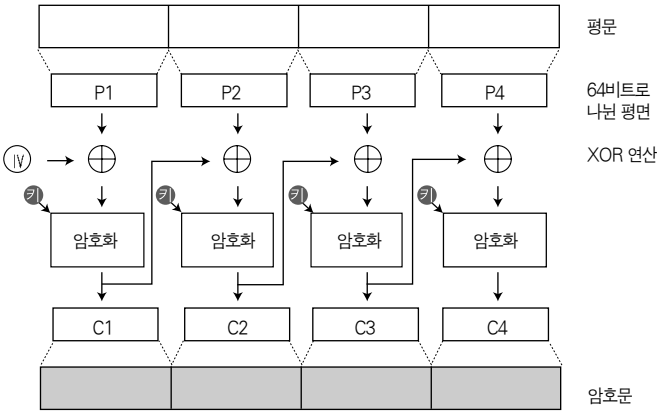
<그림 2> ECB 모드 암호화



<그림 3> ECB 모드 복호화



<그림 4> CBC 모드 암호화



비밀키 암호화 프로그래밍

OpenSSL 라이브러리로 비밀키 암호화 알고리즘을 구현하기 위해 세 가지 방법을 사용할 수 있다.

<표 1> 비밀키 암호화 알고리즘과 사용키 길이, 블록 길이

비밀키 암호화 알고리즘	키 길이	블록 길이
DES	56비트	64비트
3DES	168비트	64비트
AES	128, 192, 256비트	128, 192, 256비트
IDEA	128비트	64비트
Blowfish	32~488비트(주로 128비트 사용)	64비트
Cast	40~128비트(주로 128비트 사용)	64비트
RC2	가변	64비트
RC4	0~2048비트	스트림
RC5	0~2048비트	64, 128비트
SEED	128비트	128비트

❶ 각각의 암호화 알고리즘에 해당하는 암호화 패키지를 사용하는 방법 : OpenSSL API에는 각 암호화 알고리즘에 해당하는 패키지가 준비되어 있다. 예로, DES 암호화 알고리즘을 제공하는 ‘DES 패키지’가 있고, AES 암호화 알고리즘을 제공하는 ‘AES’ 패키지가 있다. 이 패키지들은 가장 저수준의 API라고 할 수 있다. 따라서 사용자가 제어할 수 있는 부분이 좀더 많다는 장점이 있지만, 알고리즘에 따라 다른 API 패키지를 사용해야 하므로 암호화 알고리즘을 변경하려고 할 때 많은 코드를 수정해야 한다는 단점이 있다.

❷ EVP API 패키지를 사용하는 방법 : EVP 패키지는 여러 가지 암호화 알고리즘에 대해 공통된 프로그래밍 인터페이스를 제공한다. 비밀키 암호화뿐만 아니라 MD, 공개키 암호화까지 EVP 패키지를 사용해서 구현할 수 있다. EVP 패키지는 암호화의 모든 기능을 쉽게 사용할 수 있게 해주면서 알고리즘들에 대한 공통된 인터페이스를 제공하므로 많은 이점이 있다. EVP 패키지는 **❶**에서 설명한 암호화 알고리즘에 대한 패키지를 랩핑하는 역할을 한다.

❸ BIO를 사용하는 방법 : 입출력을 담당하는 BIO 패키지를 사용해서 비밀키 암호화를 할 수 있다. 지난 호에 설명했던 것처럼 입출력 BIO 중간에 비밀키 암호화를 담당하는 Filter BIO 넣어서 BIO 체이닝을 만든다. 그리고 입출력을 하면 데이터는 중간에 암호화된다. BIO 패키지도 EVP 패키지와 마찬가지로 각각의 암호화 알고리즘 패키지를 랩핑한다. BIO 패키지를 통한 암호화는 입출력을 항상 BIO에 의존해야 하므로 프로그램 내에서 다른 입출력 방법을 사용한다면 **❷** 또는 **❸** 방법을 이용해야 한다.

앞의 세 가지 방법 모두 동일한 암호화 기능을 제공한다. 하지만 각각 다른 패키지를 사용하므로 세 방법은 모두 프로그래밍 인터페이스가 다르다. 본 연재에서는 **❷**과 **❸**에 대해서만 설명하겠다.

암호화 구조체

EVP API를 사용한 비밀키 암호화 작업을 수행하기 위해 가장 먼저 해야 할 일은 암호화 구조체를 만드는 것이다. 이 암호화 구조체는 암호화를 수행하는 동안의 모든 정보들이 저장되는 곳이라고 할 수 있

다. 비밀키 암호화 알고리즘에 따라 키의 길이라든지 사용하는 블록 길이가 다르다는 것은 이미 앞에서 배웠는데, 암호화 구조체에는 이러한 각 알고리즘에 대한 정보들이 저장된다. 그리고 IV, 키와 같은 비밀키 암호화에 사용되는 중요한 값들이 저장되기도 하고, 암호화 결과로 생성되는 중간 데이터들이 저장되기도 한다.

EVP_CIPHER

EVP_CIPHER는 ‘암호화 알고리즘 구조체’라고 부르는 OpenSSL API의 구조체이다. 이 구조체는 비밀키 암호화 알고리즘에 대한 정보를 저장한다. 이 구조체를 통해 EVP API의 여러 함수에 ‘이 비밀키 암호화 알고리즘을 사용해야’라는 것을 알려 줄 수 있다. EVP_CIPHER는 두 가지 방법으로 얻을 수 있다. 첫 번째는 EVP_CIPHER를 반환하는 함수를 사용하는 것인데, 이 함수 이름은 ‘알고리즘 명+키 길이+블록 모드’ 형식으로 되어 있다. 두 번째는 EVP_get_cipherbyname 함수를 사용하는 것인데, 인자로 ‘알고리즘 명+키 길이+블록 모드’ 형식의 스트링을 넣으면 이에 해당하는 EVP_CIPHER를 반환한다. 다음은 128비트의 키를 사용하고 ECB 모드의 AES 알고리즘에 해당하는 EVP_CIPHER를 얻는 예이다.

```
// 두 가지 방법의 EVP_CIPHER 생성
EVP_CIPHER *c = EVP_aes_128_ecb();
EVP_CIPHER *c = EVP_get_cipherbyname("aes-128-ecb");
```

EVP_CIPHER_CTX

EVP_CIPHER_CTX는 ‘암호화 컨텍스트(Cipher Context)’라고 부른다. 이 구조체는 비밀키 알고리즘을 통한 암호화가 진행되면서 생성되는 데이터들을 저장하는 역할을 한다. 예를 들어, 우리가 앞에서 배웠듯 DES 알고리즘은 각 블록에 대해 16번의 암호화 과정을 반복하는데 이런 과정 중간에 생성되는 데이터들을 저장할 공간이 필요하게 되며 IV나 키를 저장할 공간이 필요할 것이다. EVP_CIPHER_CTX는 이러한 암호화, 복호화에 필요한 중간 데이터들을 저장한다. EVP_CIPHER_CTX를 생성하는 방법은 간단하다. 포인터 변수를 만들고 EVP_CIPHER_CTX_init 함수로 초기화하면 된다.

```
// EVP_CIPHER_CTX 생성하기
EVP_CIPHER_CTX *ctx;
EVP_CIPHER_CTX_init(ctx);
```

암호화 컨텍스트 EVP_CIPHER_CTX는 암호화시 생성되거나 필요한 데이터를 저장하는 공간이므로 다른 암호화 과정이라면 다른 암

호화 컨텍스트를 만들어서 사용해야 한다. 만약 같은 암호화 컨텍스트를 다른 암호화 과정마다 공유해서 사용한다면 심각한 상황이 벌어질 것이다.

EVP API를 이용한 비밀키 암호화

EVP API는 암호화 방법이나 암호화 알고리즘에 상관없이 공통된 프로그래밍 인터페이스를 제공한다고 했다. EVP API는 <그림 5>처럼 세 단계의 과정을 거쳐 암호화, 복호화를 수행한다.

간단하게도 세 단계는 모두 각각 하나의 함수가 담당한다. 각 함수 이름은 순서대로 EVP_EncryptInit_ex, EVP_EncryptUpdate, EVP_EncryptFinal_ex이다. 여기서 OpenSSL API의 EVP API를 이용한 비밀키 암호화 프로그래밍 과정을 정리해 보자.**❸**과 **❹**번의 과정은 지난 호에서 설명했다).

- ❶ 암호화 알고리즘과 블록 모드, 키 길이를 나타내는 EVP_CIPHER구조체를 생성한다.
- ❷ 암호화 과정 수행 중에 생성되는 정보를 저장할 EVP_CIPHER_CTX 컨텍스트를 생성한다.
- ❸ 암호화 과정에서 사용할 키를 생성한다.
- ❹ 암호화 과정에서 초기화 벡터로 사용될 IV 값을 생성한다.
- ❺ EVP_EncryptInit_ex 함수를 사용해 초기화 과정을 수행한다.
- ❻ EVP_EncryptUpdate 함수를 사용해 업데이트 과정을 수행한다.
- ❼ EVP_EncryptFinal_ex 함수를 사용해 종료 과정을 수행한다.

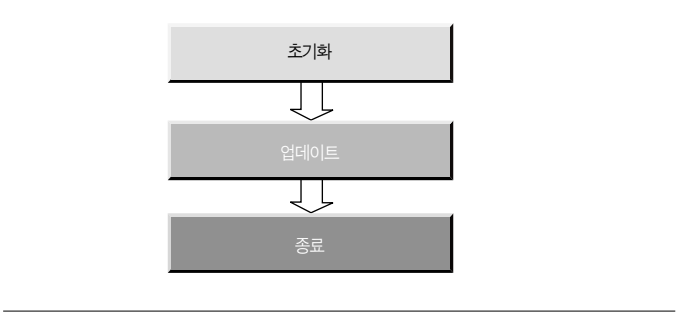
초기화 과정

초기화 과정은 이전 과정에서 만든 EVP_CIPHER, EVP_CIPHER_CTX, 키, IV를 OpenSSL API에 알려주는 과정이다. 즉 초기화 과정은 암호화에 필요한 기본적인 객체들을 설정하는 과정이라고 할 수 있다. 다음 EVP_EncryptInit_ex 함수를 보자.

```
int EVP_EncryptInit_ex ( IN EVP_CIPHER_CTX *ctx,const
                        IN EVP_CIPHER *type,
                        IN ENGINE *impl,
                        IN unsigned char *key,
                        IN unsigned char *iv);
```

모두 앞에 ‘IN’이 붙었으므로 입력 파라미터라는 것을 알 수 있다. ENGINE은 하드웨어 암호화 엔진을 나타내는 구조체인데 하드웨어 엔진을 사용하지 않으면 NULL 값을 넣는다. 함수 성공시 1, 실패시 0이 리턴된다. 사실 이 함수는 EVP_CIPHER에 저장된 비밀키 암호화 알고리즘의 정보와 키, IV 값을 EVP_CIPHER_CTX에 저장하는 역할을 한다.

<그림 5>EVP API의 세 단계 과정



업데이트 과정

업데이트 과정은 실제로 암호화를 수행하는 과정이다. 초기화 과정에서 얻은 알고리즘 정보, 키, IV를 가지고 평문을 암호화한다. 평문은 지금의 업데이트 과정에서 입력된다. 비록 업데이트 과정이 평문 데이터를 암호화해 그 결과를 반환하는 역할을 하지만, 이 업데이트 과정만으로 모든 평문 데이터가 암호화되는 것은 아니다. 다음에 설명할 종료 과정을 거쳐야 모든 입력 데이터의 암호화된 데이터를 얻을 수 있다. 업데이트 과정에서 모든 암호화된 데이터를 얻을 수 없는 이유는 블록 암호화에는 패딩이 들어가기 때문인데, 이런 블록 끝의 패딩된 데이터를 때문에 업데이트 다음의 종료 과정이 한번 더 있는 것이다. 하지만 패딩을 제외한 대부분의 데이터는 업데이트 과정을 통해 암호화돼 반환된다. 즉 마지막 블록을 제외하고는 여기서 암호화된다. 만약, 패딩이 들어가지 않는 경우, 즉 암호화할 데이터가 암호화 알고리즘의 블록 길이로 나뉘질 경우는 모든 데이터가 이 단계에서 암호화될 것이다.

```
int EVP_EncryptUpdate( IN EVP_CIPHER_CTX *ctx, // 암호화 컨텍스트
                      OUT unsigned char *out, // 출력 암호문
                      OUT int *outl, // 출력 암호문의 길이
                      IN unsigned char *in, // 입력 평문 데이터
                      IN int inl); // 입력 평문 데이터의 길이
```

여기서 주의할 점은 출력 변수는 입력 변수보다 한 블록 길이 정도 크기가 커야 한다는 점이다. 그 이유는 출력 데이터에는 입력 데이터에 패딩 데이터가 덧붙기 때문이다. 따라서 안전하게 출력 변수는 입력 변수보다 한 블록 길이 정도 크게 잡아야 한다.

종료 과정

종료 과정은 아주 단순하다. 업데이트 과정에서 마치지 못한 남은 데이터를 암호화한다. 즉 마지막 블록을 암호화한다. 안전을 위해 종료 과정을 꼭 수행해야 한다. EVP_EncryptFinal_ex 함수를 보자.

```
int EVP_EncryptFinal_ex( IN EVP_CIPHER_CTX *ctx,    // 암호화 컨텍스트
                        OUT unsigned char *out,    // 출력 암호문
                        OUT int *outl);            // 출력 암호문의 길이
```

데이터 암호화해 보기

지금까지 배운 것을 이용해 프로그램을 만들어 보자. 지금부터 만들 프로그램은 ‘plain.txt’ 란 파일에서 평문 데이터를 읽어 암호문을 만든 후 ‘encrypt.bin’ 파일에 저장한다. 그리고 간단하게 하기 위해 키와 IV 값은 직접 넣어 주었다. 입출력에 관련된 함수는 생략했다. 전체 소스는 ‘이달의 디스켓’을 참고하길 바란다.

```
#define IN_FILE  "plain.txt"           // 평문 파일
#define OUT_FILE  "encrypt.bin"       // 암호문이 저장될 파일
```

```
int _tmain(int argc, _TCHAR* argv[])
{
```

```
    // 키와 IV 값은 편의를 위해 직접 만든다.
    unsigned char key[] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
    unsigned char iv[] = {1,2,3,4,5,6,7,8};
```

```
    BIO *errBIO = NULL;
    BIO *outBIO = NULL;
```

```
    // 에러 발생의 경우 해당 에러 스트링 출력을 위해 미리 에러 스트링들을 로딩
    ERR_load_crypto_strings();
```

```
    // 표준 화면 출력 BIO 생성
    if ((errBIO=BIO_new(BIO_s_file())) != NULL)
        BIO_set_fp(errBIO,stderr,BIO_NOCLOSE|BIO_FP_TEXT);
    // 파일 출력 BIO 생성
    outBIO = BIO_new_file(OUT_FILE,"wb");
    if (!outBIO)
    { // 에러가 발생한 경우
        BIO_printf(errBIO,"파일 [%s]을 생성하는데 에러가 발생했습니다.", OUT_FILE);
        ERR_print_errors(errBIO);
        exit(1);
    }
```

```
    // 파일에서 읽는다(readFile 함수의 정의는 전체 소스를 참고).
    int len;
    unsigned char * readBuffer = readFile(IN_FILE,&len);
```

```
    // 암호화 컨텍스트 EVP_CIPHER_CTX 생성. 초기화
    EVP_CIPHER_CTX ctx;
    EVP_CIPHER_CTX_init(&ctx);
```

```
    // 초기화
```

```
    EVP_EncryptInit_ex(&ctx, EVP_bf_cbc(), NULL, key, iv);
```

```
    // 초기화가 끝난 후에 해야 한다. 암호문 저장할 버퍼 생성
    unsigned char * outbuf = (unsigned char *)malloc(sizeof(unsigned char) *
        (len + EVP_CIPHER_CTX_block_size(&ctx)));
```

```
    int outlen, tmplen;
    // 업데이트, 마지막 블록을 제외하고 모두 암호화
    if(!EVP_EncryptUpdate(&ctx, outbuf, &outlen, readBuffer,
        strlen((char *)readBuffer)))
        return 0;
```

```
    // 종료, 마지막 블록을 암호화
    if(!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmplen))
        return 0;
```

```
    // 암호문 길이는 업데이트, 종료 과정에서 나온 결과의 합
    outlen += tmplen;
    EVP_CIPHER_CTX_cleanup(&ctx);
```

```
    BIO_printf(errBIO, "암호문 생성이 완료되었습니다.\n[%s] 파일에 암호문이 저장되었습니다.", OUT_FILE);
```

```
    // 파일에 암호문 내용을 출력한다.
    BIO_write(outBIO, outbuf, outlen);
```

```
    // 객체 제거
    BIO_free(outBIO);
    return 0;
}
```

EVP API를 이용한 비밀키 복호화

복호화 과정은 앞서 설명한 암호화 과정과 아주 비슷하다. 다만 초기화, 업데이트, 종료 과정을 수행하는 함수 이름만이 다를 뿐이다. 다음에 EVP API를 사용한 비밀키 복호화 프로그래밍 과정을 정리했다. 암호화 과정의 경우와 거의 같다는 것을 알 수 있다.

❶ 복호화 알고리즘과 블록 모드, 키 길이를 나타내는 EVP_CIPHER 구조체를 생성한다.

❷ 복호화 과정 수행 중에 생성되는 정보를 저장할 EVP_CIPHER_CTX 컨텍스트를 생성한다.

❸ 복호화 과정에서 사용할 키를 생성한다.

❹ 복호화 과정에서 초기화 벡터로 사용될 IV 값을 생성한다.

❺ EVP_DecryptInit_ex 함수를 사용해 초기화 과정을 수행한다.

❻ EVP_DecryptUpdate 함수를 사용해 업데이트 과정을 수행한다.

❼ EVP_DecryptFinal_ex 함수를 사용해 종료 과정을 수행한다.

초기화 과정은 암호화의 경우와 똑같다. EVP_DecryptInit_ex 함수를 보자.

```
int EVP_DecryptInit_ex ( IN EVP_CIPHER_CTX *ctx,
                        IN const EVP_CIPHER *type,
                        IN ENGINE *impl,
                        IN unsigned char *key,
                        IN unsigned char *iv);
```

업데이트 과정도 암호화 시와 유사하다. 복호화시 업데이트 과정은 패딩을 제외한 나머지 부분을 복호화한다. 패딩을 제거하는 역할은 다음의 종료 과정에서 하게 된다. EVP_DecryptUpdate 함수를 보자.

```
int EVP_DecryptUpdate( IN EVP_CIPHER_CTX *ctx,    // 암호화 컨텍스트
                      OUT unsigned char *out,    // 출력 복호문
                      OUT int *outl,            // 출력 복호문의 길이
                      OUT unsigned char *in,     // 입력 암호문
                      OUT int inl);             // 입력 암호문의 길이
```

역시 안전하게 종료 과정을 수행해야 한다.

```
int EVP_DecryptFinal_ex( IN EVP_CIPHER_CTX *ctx,    // 암호화 컨텍스트
                        OUT unsigned char *out,    // 출력 복호문
                        OUT int *outl);           // 출력 복호문의 길이
```

이제 전의 비밀키 암호화 예제 프로그램이 만들었던 암호문을 복호화하는 프로그램을 만들 차례이다. 하지만 암호화 프로그램과 유사하기 때문에 지면에 실는 것은 생략하니 ‘이달의 디스켓’을 참고하길 바란다.

BIO를 이용한 비밀키 암호화

지난 호에 입출력 패키지인 BIO API에 관해 설명했다.中间的의 Filter BIO에 비밀키 암호화 BIO를 넣음으로써 파일 입출력이라든지 소켓 입출력의 중간에 데이터를 암호화할 수 있다고 했다. 이번 호에서는 BIO API를 사용해 비밀키 암호화를 수행하는 예제 프로그램을 만들어 본다. 이번에 만드는 예제 프로그램은 앞서의 비밀키 암호화 예제 프로그램과 같이 파일에서 평문을 읽어서 암호화한 후 암호문을 파일로 저장한다. 그러나 앞서와 달리 키와 IV 값을 직접 넣어주지 않고 패스워드 값을 통해 만들어 보겠다.

눈여겨봐야 할 함수는 BIO_set_cipher이다. 이 함수는 BIO에 EVP_CIPHER를 연결시켜 비밀키 암호화 Filter BIO를 만들어 준다. 이 함수는 ‘evp.h’에 선언되어 있다.

```
void BIO_set_cipher( IN BIO *b,                // BIO
                   IN const EVP_CIPHER *c,    // EVP_CIPHER
                   IN unsigned char *k,       // 키
                   IN unsigned char *i,       // IV
                   IN int enc);               // 1이면 암호화, 0이면 복호화
```

마지막 파라미터인 ‘enc’의 값에 1을 주면 암호화를 수행하는 BIO가 만들어지고, 0을 주면 복호화를 수행하는 BIO가 만들어진다.

그럼 예제를 보자.

```
#define IN_FILE  "plain.txt"
#define OUT_FILE  "encrypt.bin"
#define CIPHER  "des-ecb"
```

```
int _tmain(int argc, _TCHAR* argv[])
{
```

```
    BIO *errBIO = NULL;
    BIO *outBIO = NULL;
    BIO *encBIO = NULL;
```

```
    EVP_CIPHER *cipher = NULL;
    // 패스워드 값
    char * password = "mypass";
    // Salt를 저장, 길이는 8
    unsigned char salt[8] = {1,2,3,4,5,6,7,8};
    // 키와 IV가 저장될 변수를 정의, 길이는 OpenSSL에서 알아서 정함.
    unsigned char key[EVP_MAX_KEY_LENGTH],iv[EVP_MAX_IV_LENGTH];
```

```
    // 에러 발생의 경우 해당 에러 스트링 출력을 위해 미리 에러 스트링들을 로딩
    ERR_load_crypto_strings();
    // 동적인 EVP_CIPHER 생성을 위해 비밀키 알고리즘을 내부에 로드
    OpenSSL_add_all_ciphers();
    // 동적인 EVP_CIPHER 생성
    cipher = EVP_get_cipherbyname(CIPHER);
```

```
    // 표준 화면 출력 BIO 생성
    if ((errBIO=BIO_new(BIO_s_file())) != NULL)
        BIO_set_fp(errBIO,stderr,BIO_NOCLOSE|BIO_FP_TEXT);
```

```
    // 파일 출력 BIO 생성
    outBIO = BIO_new_file(OUT_FILE, "wb");
    if (!outBIO)
    { // 에러가 발생한 경우
        BIO_printf(errBIO, "파일 [%s]을 생성하는데 에러가 발생했습니다.", OUT_FILE);
        ERR_print_errors(errBIO);
        exit(1);
    }
    // 비밀키 암호화 BIO 생성
    encBIO = BIO_new(BIO_f_cipher());
    if (encBIO == NULL)
    { // 에러가 발생한 경우
        BIO_printf(errBIO, "비밀키 암호화 BIO 생성 에러");
        ERR_print_errors(errBIO);
        exit(1);
    }
```

```
    // 패스워드를 사용해서 키와 IV 생성
    EVP_BytesToKey(cipher,EVP_md5(), salt,
        (unsigned char *)password,strlen(password), 1, key,iv);
```

```
    // 비밀키 암호화 BIO에 EVP_CIPHER와 키, IV 연결
    BIO_set_cipher(encBIO,cipher,key,iv,1); // 1이면 암호화, 0이면 복호화
```

```
    // 파일에서 읽는다.
```

```
int len;
unsigned char * readBuffer = readFile(IN_FILE,&len);

// 파일 출력 BIO 위에 암호화 BIO를 연결한다.
encBIO = BIO_push(encBIO,outBIO);

// 체인 BIO에 암호문을 출력한다.
BIO_write(encBIO, (char *)readBuffer, len);

// 모든 내용을 출력 후 BIO를 비운다.
BIO_flush(encBIO);

BIO_printf(errBIO, "파일 [%s]에 암호문이 저장되었습니다.", OUT_FILE);
// 객체 제거 - 모든 체인의 BIO가 제거된다.
BIO_free(encBIO);

return 0;
}
```

이미 설명한 내용들이 대부분이지만 간단히 프로그램의 흐름을 짚어 보기로 하겠다. EVP_get_cipherbyname 함수를 통해 EVP_CIPHER를 만든다. 그리고 파일 입력 BIO, 출력 BIO를 만든다. 그런 다음에 EVP_BytesToKey 함수를 이용해 ‘mypass’로 주어진 패스워드에서 키와 IV 값을 생성해 낸다. 이렇게 생성한 키와 IV, EVP_CIPHER를 BIO_set_cipher 함수에 전달해 암호화 BIO를 만든다. 그리고 BIO_push 함수로 BIO를 연결한 다음 BIO_write 함수에 평문을 넣으면 평문은 암호화돼 출력될 것이다. 복호화를 수행하고 싶다면 앞 예제에서 BIO_set_cipher의 ‘enc’ 파라미터를 1 대신 0으로만 바꾸면 된다.

MD 암호화 프로그래밍

OpenSSL API를 이용해 MD 프로그램을 만들 때 비밀키 암호화와 같이 세 가지 방법을 사용할 수 있다. 즉 각각의 암호화 알고리즘에 해당되는 암호화 패키지를 사용하는 방법, EVP API 패키지를 사용하는 방법, BIO를 사용 하는 방법이 있다. 이 세 가지 방법의 차이점은 이미 설명한 바와 같다. 여기서는 두 번째 방법인 EVP API를 사용하는 방법만 설명한다. BIO를 이용한 방법은 앞서 비밀키 암호화의 경우와 거의 같다. 다만 Filter BIO를 MD BIO로만 바꾸면 된다.

〈표 2〉MD 알고리즘과 해시 값의 길이	
MD 알고리즘	해시 값의 길이
MD2	128비트
MD4	128비트
MD5	128비트
MCD2	128비트
SHA1	160비트
RIPEMD-160	160비트

MD 알고리즘 정리

MD는 임의의 길이의 데이터를 압축하여 일정한 길이의 데이터로 만든다. 이 결과 값을 ‘해시 값’이라고 한다. 입력 데이터가 1바이트만 달라져도 출력 해시 값은 50% 이상이 변화된다. 따라서 MD를 사용하면 데이터의 변경이 있었는지 쉽게 알 수 있다. 〈표 2〉는 많이 사용되는 MD 알고리즘과 해시 값의 길이를 정리한 것이다.

암호화 구조체

비밀키 암호화 프로그래밍에서 EVP_CIPHER에 해당하는 MD의 구조체는 EVP_MD이다. EVP_CIPHER와 같이 EVP_MD는 MD 알고리즘의 정보를 나타낸다. 자신이 원하는 MD 알고리즘에 맞는 EVP_MD를 생성한 다음 이를 사용하면 된다. EVP_MD를 생성하는 방법도 두 가지가 있는데, ‘EVP_알고리즘 명’ 형식으로 되어 있는 함수를 사용하는 것과 ‘알고리즘 명’을 인자로 넣은 EVP_get_digestbyname 함수를 사용하는 방법이 있다. 다음은 MD5 알고리즘에 해당하는 EVP_MD를 얻는 예이다.

```
EVP_MD *md1 = EVP_md5();
EVP_MD *md2 = EVP_get_digestbyname("md5");
```

그리고 MD 과정을 진행하면서 데이터들이 저장되는 구조체는 EVP_MD_CTX이다. EVP_CIPHER_CTX와 비슷한 의미를 가지고 있으므로 따로 설명하지는 않겠다.

EVP API를 이용한 MD 프로그래밍

EVP API의 MD 프로그래밍 과정은 비밀키 암호화의 경우와 마찬가지로 초기화, 업데이트, 종료 순으로 진행된다. 초기화 과정은 EVP_MD_CTX를 설정하는 과정이다. EVP_MD의 정보를 EVP_MD_CTX에 저장한다. 초기화 과정을 담당하는 함수는 EVP_DigestInit_ex이다.

```
int EVP_DigestInit_ex ( IN EVP_MD_CTX *ctx,    // MD 컨텍스트
                       IN const EVP_MD *type,  // MD 구조체
                       IN ENGINE *imp1,);      // 하드웨어 엔진, 없으면 NULL
```

업데이트 과정은 실제로 MD를 수행하는 과정이다. 이 과정을 담당하는 함수는 EVP_DigestUpdate이다.

```
int EVP_DigestUpdate(  IN EVP_MD_CTX *ctx,    // MD 컨텍스트
                      IN const void *in,      // 입력 메시지 데이터

                      IN unsigned int inl);    // 입력 메시지 데이터의 길이
```

종료 과정은 생성된 해시 값을 출력하는 과정이다. 이 과정을 담당하는 함수는 EVP_DigestFinal이다.

```
int EVP_DigestFinal(  IN EVP_MD_CTX *ctx,      // MD 컨텍스트
                     OUT unsigned char *out,    // 출력 해시 값
                     OUT unsigned int *outl);    // 출력 해시 값의 길이
```

앞의 세 함수 모두 성공하면 1, 실패하면 0이 리턴된다.

MD 프로그램을 만들어 보자

그럼 실제로 MD 과정을 수행하는 프로그램을 만들어 보자. 간단하게 만들기 위해 평문을 직접 프로그램 안에 넣고, 만들어지는 해시 값은 화면으로 출력한다.

```
int _tmain(int argc, _TCHAR* argv[])
{

// MD 구조체를 생성
EVP_MD_CTX mdctx;
// 암호화 구조체를 저장할 변수 정의
const EVP_MD *md;

// 다이제스트할 평문을 직접 변수에 넣음
char message1[] = "테스트 메시지입니다.\n";
char message2[] = "안녕하세요\n";

// 생성된 압축 해시 값을 저장할 변수 정의. 길이는 OpenSSL에서 정함
unsigned char hashValue[EVP_MAX_MD_SIZE];
// 생성된 압축 해시 값의 길이를 저장할 변수 정의
unsigned int hashLen, i;
```

```
// 동적인 암호화 구조체 생성을 위해 모든 다이제스트 알고리즘 로딩
OpenSSL_add_all_digests();
// SHA 암호화 구조체 생성
md = EVP_get_digestbyname("sha");
// 리턴 값이 0이면 에러. 프로그램 종료
if(!md) {
    printf("암호화 구조체를 생성할 수 없습니다.");
    exit(1);
}
```

```
// 컨텍스트 초기화
EVP_MD_CTX_init(&mdctx);
```

```
// MD 첫 과정. 초기화
EVP_DigestInit_ex(&mdctx, md, NULL);
```

```
// 첫 메시지 압축
EVP_DigestUpdate(&mdctx, message1, (unsigned int)strlen(message1));
```

```
// 두 번째 메시지 압축
```


```
EVP_DigestUpdate(&mdctx, message2, (unsigned int)strlen(message2));
// 해시 값 생성
EVP_DigestFinal_ex(&mdctx, hashValue, &hashLen);
```

```
// 컨텍스트 해제
EVP_MD_CTX_cleanup(&mdctx);
```

```
// 키 길이 출력
printf("해시 길이: %i 바이트\n",hashLen);
```

```
// 생성 해시 값 화면 출력.
printf("해시 값 : ");
for(i = 0; i < hashLen; i++) printf("%02x", hashValue[i]);
{
    printf("\n");
}
return 0;
}
```

실전 암호화 프로그래밍

이번 호를 통해 OpenSSL API가 어떠한 구조를 가지고 암호화를 수행하는지 대충 감을 잡을 수 있을 것이다. 다른 암호화 라이브러리보다 특별하게 간단한 구조로 되어 있는 것은 아니지만 암호화의 기능을 충실히 구현해 놓았다는 것을 알 수 있을 것이다. 그리고 비밀키 암호화, MD 프로그래밍에 대한 OpenSSL API 프로그래밍 방법을 기본 이론과 몇 가지 예제 프로그램을 덧붙여 설명했는데 이어지는 연재에도 비슷한 방식으로 진행할 예정이다. 다음 호에서는 공개키 암호화 프로그래밍에 관한 내용을 다룬다. 아마도 이번 호보다는 어려운 부분이 많겠지만 그만큼 여러분의 암호화 능력을 향상시켜 줄 수 있을 것이다. 

장리 | 박은영 | whoami@korea.net.com



이 + 달 + 의 + 디 + 스 + 켓
OpenSSL2.zip <http://www.imaso.co.kr>

참 + 고 + 자 + 료

- 1 OpenSSL : <http://www.openssl.org>
- 2 Java Security API : <http://java.sun.com/security/>
- 3 Microsoft MSDN Security : <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/securityanchor.asp>
- 4 RSA Security : <http://www.rsasecurity.com/>
- 5 한국정보보호진흥원 : <http://www.kisa.or.kr/>
- 6 퓨처시스템 암호 센터 : <http://www.securitytechnet.com/crypto/algorithm/intro/intro.html>