안전한 암호화를 위한 AES 알고리즘에 대한 이해와 구현코드(Java, C#)

개요

오랜만에 쓰는 글입니다. 요 근래 정신이 너무 없어서 블로그 글도 못쓰고 있던 찰나에 인턴으로 근무하고 있는 회사에서 맡게 된 업무인 C# 암호화, Java 복호화에 대해서 설명하고자 합니다.
먼저, 저희회사에는 C#과 Java로 만들어지는 프로그램이 있었으며, 저는 그 사이에서 중간 단계 역할을 하는 인터페이스를 만드는 업무를 부여받았습니다.

일단, 인터페이스 이전에 제가 보안에는 문외한이였습니다.
그래서 우선적으로 C# 암호화 후 Java에서 복호화를 처리하는 샘플 소스코드를 작성하게 됐습니다.

AES Algorithms combined with Block Chipher mode in C# and Java

STEP 1. 안전한 암호화 방식

저는 말씀드린대로 보안에는 문외한이였습니다. 그래서 수 많은 문서를 보게되었고, 그 중에서 안전한 패스워드 저장 - Naver D2 (https://d2.naver.com/helloworld/318732) 문서를 참고하게 됐습니다.

해당 문서는 매우 좋은 문서라고 볼 수 있다고 저는 자부합니다.
이 문서는 패스워드를 저장하는 방식을 중점으로 다루긴하지만, 제가 사용했던 **PBKDF2, Salt, SHA512**에 대해서 많은 설명을 하고 있기에 많은 부분을 참고하게 됐습니다.

1. 암호의 종류

암호이용활성화 - KISA (https://seed.kisa.or.kr/iwt/ko/intro/EgovCryptographic.do)를 참고하면, 암호의 종류에는 **대칭키 암호, 비 대칭키 암호, 해쉬함수** 등이 있음을 알 수가 있다.

1.1.1 해쉬함수

먼저, **해쉬함수**입니다.
다양한 해쉬함수들이 존재하는데 흔히 많이 아는 MD5, SHA-512등이 있다.

특히, 개발자들은 **단방향 해시 함수(one-way hash function)** 으로 패스워드 생성을 많이 하는 것으로 알고 있다.
단방향 해시 함수의 특징을 설명을 하자면

1. 수학적인 연산을 통해 원본 메세지를 변환하여 암호화된 메세지인 다이제스트 생성
2. 암호화된 메세지로는 원본 메세지를 구할 수 없어야 하며, 이를 '단방향성'이라 한다.

정도로 볼 수 있을 것 같다. **즉, 암호화된 다이제스트로는 원본을 구할 수 없어야한다.** 가 단방향 해시 함수의 핵심이라고 볼 수 있다.
단순하게 설명하자면, 해시는 암호화를 하는데 많이 이용될 뿐이지. 그 자체가 암, 복호화를 위한 목적으로 만들어진 함수가 아니다. **즉, 복호화가 불가능하다! (자세한 내용은 비둘기집의 원리를 참고하자)**

SHA-256의 예시를 들어보면 'hunter2'의 다이제스트는

> *f52fbd32b2b3b86ff88ef6c490628285f482af15ddcb29541f94bcf526a3f6c7*

'hunter3'의 다이제스트는

> *fb8c2e2b85ca81eb4350199faddd983cb26af3064614e737ea9f479621cfa57a*

로 'hunter2'와 'hunter3'의 다이제스트는 완전히 다름을 알 수가 있다. 그러나, 상기 설명한 Naver D2의 문서에서는 **'이것만으로는 패스워드 보안이 충분히 안전하다고 보장할 수 없다'** 라고 얘기한다.

그렇다면, 해쉬함수는 어떠한 문제를 가지고 있을까?

1. **인식가능성(recognizability)**
   동일한 메시지가 언제나 동일한 다이제스트를 가진다면, 공격자가 다이제스트를 많이 확보, 탈취하여 다이제스트와 원본메세지를 찾거나 동일한 효과의 메세지를 찾을 수 있음.

   공격자가 확보한 다이제스트 = rainbow table [1]
   공격자가 raindow table을 활용하여 공격 = rainbow attack [2]

2. **속도(speed)**
   해시 함수는 '자료구조' 강의를 들었던 학부생이면 알겠지만, 해시테이블이라는 자료구조에서 사용하는 것으로 암호학에서 많이 사용되긴 하지만 본래의 목적은 **"짧은 시간에 데이터를 검색하기 위한 것"** 이다.
   따라서, 이러한 해시함수의 빠른 처리 속도를 활용하여 공격자는 매우 빠른 속도로 임의의 문자열의 다이제스트와 해킹할 대상의 다이제스트를 비교할 수 있다.

그렇다면? 이러한 단점을 보완하는 방법이 있을까?

---

1. Salting
솔트(Salt)[3]는 단방향 해시 함수에서 다이제스트를 생성할 때 추가되는 바이트 단위의 임의의 문자열이다.
이 솔트를 활용하여 원본 메세지에 문자열을 추가하여 다이제스트를 생성하는 것을 **솔팅(Salting)** 이라고 한다.

```
예를 들어, "helloworld"라는 원본 문자에 "Dp5BnBuJzdKr3DpE" 솔팅을 사용하여 다이제스트를 생성할 수가 있다.
```

> *Dp5BnBuJzdKr3DpE(salt) + helloworld (plainText) -> (Hash Function) -> DIGEST*

이 방법을 사용하면, 다이제스트를 알아낸다 한들, 공격자가 솔트도 알아야 된다.
더 나아가, 솔트가 각 패스워드마다 다를 경우에는 더욱 더 알아내기 힘들다.

Naver D2 문서에서 제안하는 방식은 **"모든 패스워드가 고유의 솔트를 갖고, 솔트의 길이가 32바이트 이상되는 것을 사용"** 을 권고한다.

2. Key Stretching

키 스트레칭은 위의 단점 중에서 '속도'에 대한 단점을 해결하기 위해서 고안 된 것이다.
핵심만 설명하자면, **빠른 해시함수의 속도로 취약점이 발생한다면, 초기 다이제스트 생성 시 해시함수를 반복(Iteration)하여 시간 소요가 걸리게 하자!** 라는 모티브에서 출발하였고, 일반적인 장비에서 0.2초 이상의 시간이 소요되게 설정한다.

이는 Rainbow Attack뿐만 아니라 Brute-force Attack(무차별 대입 공격)[4]으로 패스워드를 추측하는데 많은 시간이 소요되도록 하기 위한 것이다.

Naver D2 문서에 따르면, 최긍에는 일반적인 장비로 1초에 50억 개 이상의 다이제스트를 비교할 수 있지만, 키 스트레칭을 적용하여 동일한 장비에서 1초에 5번 정도만 비교할 수 있게끔 하는 것이라 적혀있다.

Salting과 Key Stretching을 이용하는 방식을 C#과 Java 코드로 표현하면 아래와 같다.

```
//Java Code
//해당 키를 가지고 Salt 생성 후 SHA512 적용하여, 다이제스트 생성
MessageDigest digest = MessageDigest.getInstance("SHA-512");
byte[] keyBytes = password.getBytes("UTF-8");
byte[] saltBytes = digest.digest(keyBytes);

// in Java (65536번 해싱)
PBEKeySpec pbeKeySpec = new PBEKeySpec(password.toCharArray(), saltBytes, 65536,
256);


//C# Code
//해당 키를 가지고 Salt 생성 후 SHA512 적용하여, 다이제스트 생성
byte[] keyBytes = System.Text.Encoding.UTF8.GetBytes(password);
byte[] saltBytes = SHA512.Create().ComputeHash(keyBytes);

// in C# (65536번 해싱)
Rfc2898DeriveBytes result = new Rfc2898DeriveBytes(keyBytes, saltBytes, 65536);
```

3. Adaptive Key Derivation Functions

먼저 KDF(Key Derivation Function)[5]란?

키를 파생시키는 함수라는 것이다.
여기서 Adaptive Key Derivation Function은 다이제스트를 생성할 때 솔팅과 키 스트레칭을 반복하여, 솔트와 패스워드 외에도 입력 값을 추가하여 공격자가 쉽게 다이제스트를 유추할 수 없도록 하고, 보안의 강도를 선택하는 Adaptive한 매커니즘을 KDF에 추가한 함수라고 볼 수가 있다.

내가 사용했던 함수 하나를 소개해주고, 나머지 함수들은 궁금하면 Naver D2 문서를 참고하기를 바라겠다.

- PBKDF2

  내가 뒤에서 보여줄 소스도 PBKDF2를 사용하여, 만든 키와 IV(초기화벡터)를 사용한다.
  먼저, PBKDF2는 가장 유명한 AKDF로, 솔트를 적용한 후 해시 함수의 반복 횟수를 임의로 선택할 수 있다.
  아주 가볍고, 구현하기 쉬우며(보안의 문외한인 나같은 사람도 구현함.), SHA와 같이 검증된 해시 함수만을 사용한다.

  PBKDF2의 기본 파라미터는 5가지 이다.

  > *DIGEST = PBKDF2(PRF, Password, Salt, i, DLen);*

  - PRF : 난수 (ex : HMAC)
  - Password : 패스워드
  - Salt : 위에서 설명한 솔트
  - i : Iteration(반복) 횟수
  - DLen : 원하는 다이제스트 길이

PBKDF2는 NIST(미국표준기술연구소)에 의해서 승인된 알고리즘이고, 미국 정부 시스템에서도 사용자 패스워드의 암호화된 다이제스트를 생성할 때 사용한다.

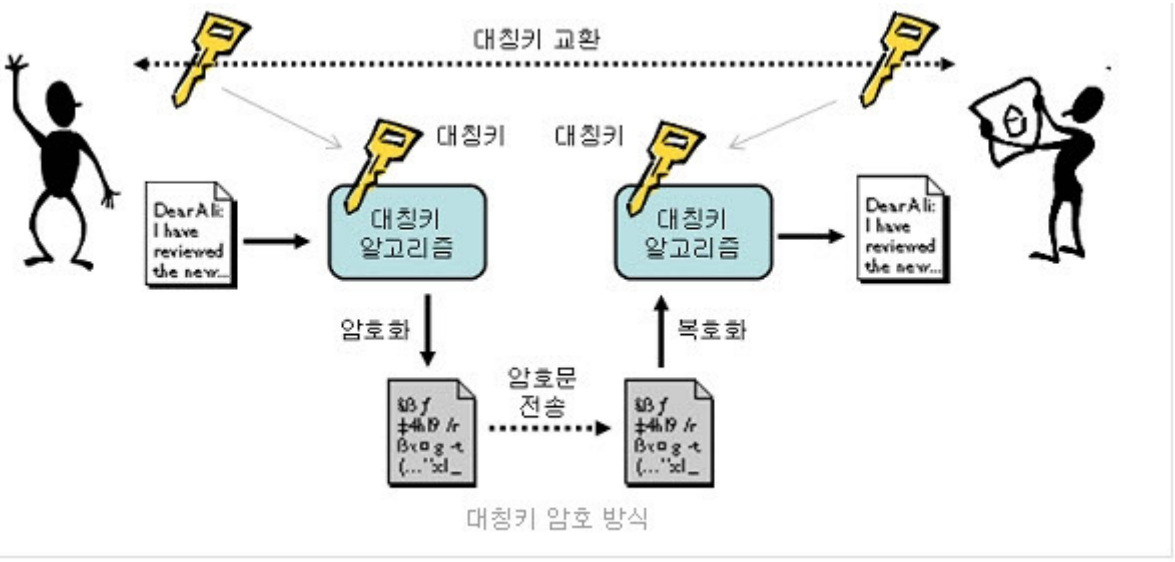
위의 소스를 보면 눈치 챘을 수도 있다.

Java에서 PBEKeySpec과 C#에서 Rfc2898DeriveBytes 클래스는
HMACSHA1기반 의사 난수 생성기를 사용하여 암호 기반 키 파생 기능인 PBKDF2를 구현하는 클래스이다!!

1.1.2 대칭키 암호 방식

대칭 키 암호 방식의 핵심은 **"같은 키를 가지고 암호화, 복호화를 진행한다"** 이다.
즉, 내가 철수에게 암호화된 파일을 건내 받았는데 철수가 열쇠를 주지 않는다면 나는 영영 못열게 되는 암호방식이다.

이는 아래 사진과 같다.



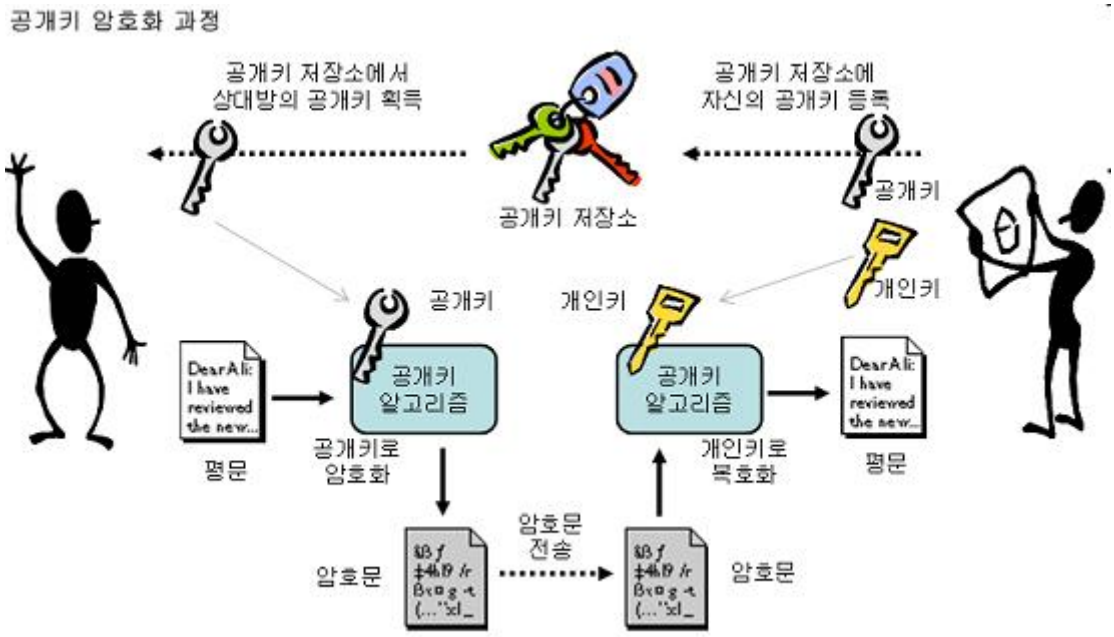
대칭키 암호 방식

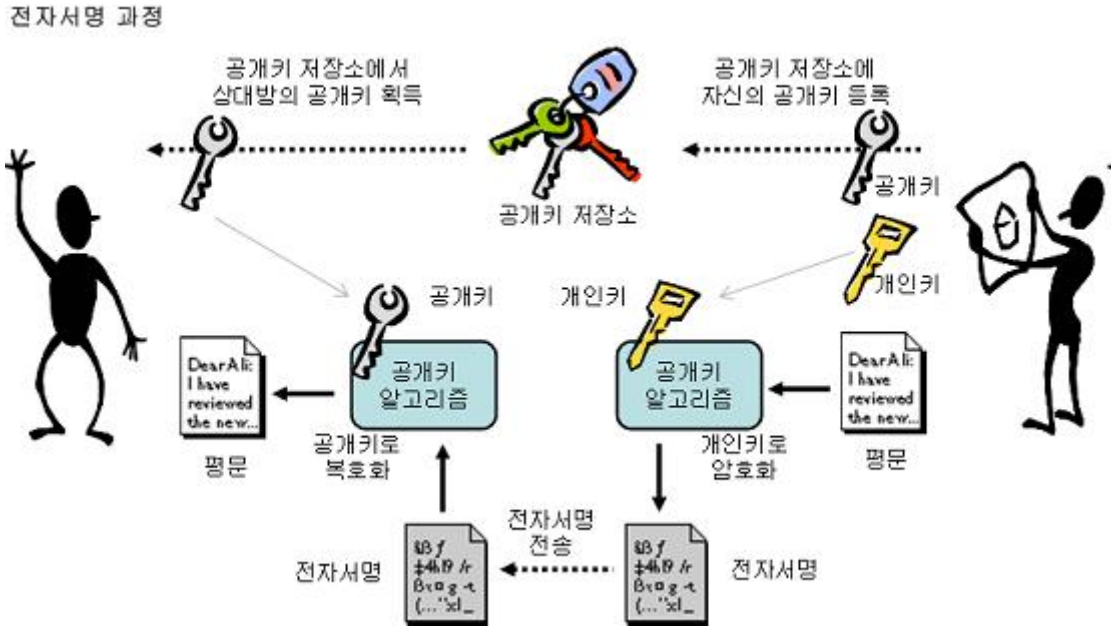
그리고 사진에서 말하는 대칭 키 알고리즘에 바로 **AES 알고리즘**이 포함되어있다.

1.1.3 비 대칭키 암호 방식

그렇다면? **비대칭 키 암호 방식(= 공개 키 암호 방식)**[6]은 무엇일까?
이는 나중에 포스팅을 하겠지만, 리눅스 SSH를 접근할 때 많이 사용되는 방식이다. 일단, Password 접근 방식의 취약점인 무차별 대입 공격(Brute-force Attack)을 막을 수가 있으며, Password를 입력하지도 않아도 접근할 수 있는 편리함이 존재한다. (단 개인 키 생성시 암호를 입력했다면 개인 키의 암호를 입력해야함.)

이는 아래 사진과 같습니다.





1.1.4 정리

단방향 해시 함수는 **'OTP(One-Time Password)'** 라고 생각하면 된다.
한번 생성했으면 다시는 복호화를 할 수가 없다.
그렇다면? 어떻게 우리가 패스워드를 해시로 저장하면 로그인이 되는 것일까?
단순하다. 저장된 DB의 해시 값과 로그인 버튼을 클릭했을 때 전송된 Password로 부터 생성된 해시값이 일치하면 로그인이 되는 것이다.
그래서 대표적인 공격방식으로 사용자의 비밀번호 해시값과 동일한 해시값을 나타내는 공격이나 해시가 탈취당했을 때 많이 비밀번호로 사용하는 문자열을 해싱하여 비교하는 공격이 두가지가 존재한다.
그래서 솔팅과 키 스트레칭이 매우 중요한 것이다!

대칭 키 방식은 흔히 우리가 알고 있는 **'자물쇠'** 를 생각하면 된다.
즉, 키를 잃어버리게 되면 잠긴 자물쇠는 풀 수가 없는 것이다.

공개 키 방식은 **서로 다른 두 개의 열쇠를 사용하는 새로운 자물쇠** 라고 볼 수가 있다.
그러나 이 자물쇠는 하나의 열쇠로 잠궜으면 열 때는 **반드시 잠근 것과는 다른 열쇠를 사용해야한다.**
공개 키 방식은 추후에 업로드하게 될 SSH 로그인 없이 쓰기에서 좀 더 자세히 다뤄보도록하겠다.
(이번 포스트에서의 메인은 대칭 키 암호 방식!!)
그렇다면, **대칭 키 암호 방식인 AES 알고리즘**은 무엇일까?

STEP 2. AES256 알고리즘이란?

글을 읽기 싫은 사람이면 여기를 보자 AES Algorithm (http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html), 단, **영어 만화**이다!
혹은, AES Rijndael Cipher explained as a Flash animation (https://www.youtube.com/watch?v=gP4PqVGudtg)를 참고하자.

누누히 얘기하지만, 나 또한 암호학 전공자가 아닐뿐더러, **"까라면 까야지"** 하는 마음에서 급하게 구현을 했기 때문에 정말로 정확하게 알고리즘을 이해하고 쓰는 글은 아니다. 물론, 어느정도 이해를 하긴 했지만 나는 '구현'이 초점이였기 때문에 핵심적인 알고리즘보다는 어떻게 '동작'을 하는지에 초점을 두어서 설명하고자 한다.

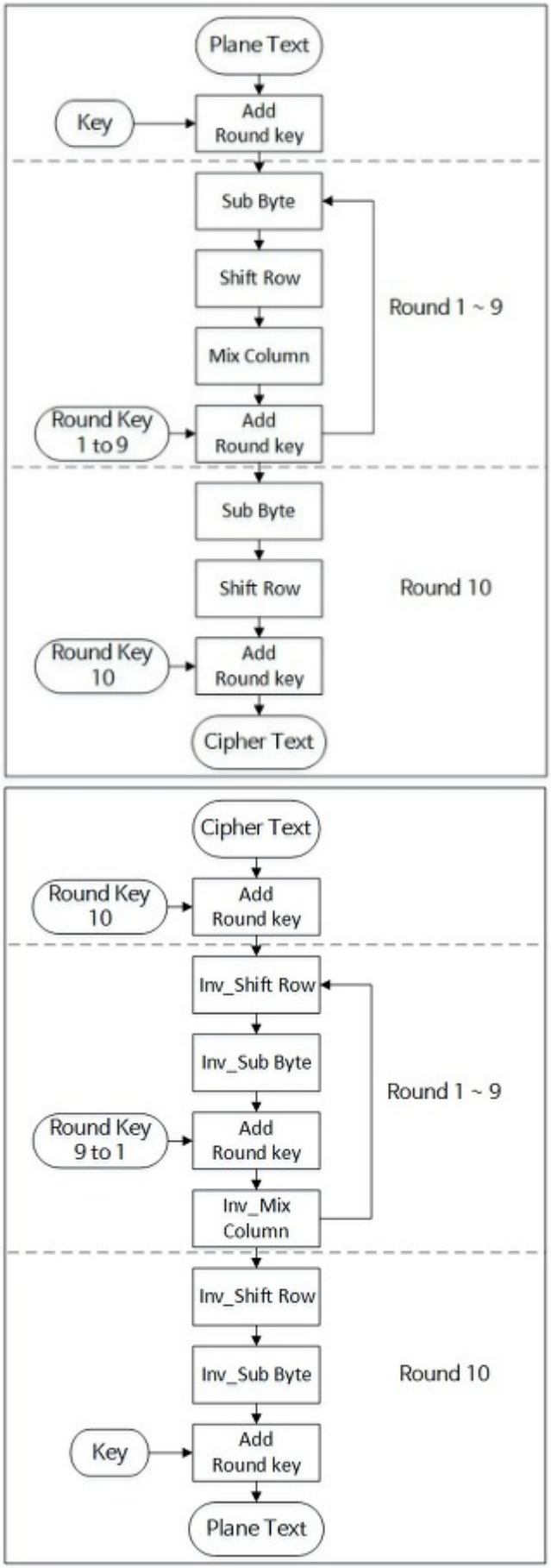
STEP 2.1 Dive into AES Algorithm

- 역사
  C#에서 보면 RijndaelManaged 클래스를 활용하여, AES 알고리즘을 적용하는데 나는 이게 어디서 파생된 것인지 궁금했다.
  일단, AES는 Rijmen과 Daemen이 제안한 알고리즘이 AES 공모전에서 선정되면서 AES로 암호화 표준이 되었다. 즉, Rijndae는 만들었던 사람들의 앞의 3글자를 따서 만들어진 것이다.

- **세부 동작 원리**
  AES 알고리즘의 내부에는 Add Round Key, Sub Byte, Mix Column의 반복으로 이루어진다.
  동영상으로 보기 쉽게 보고 싶다면, AES Rijndael Cipher explained as a Flash animation (https://www.youtube.com/watch?v=gP4PqVGudtg)를 추천한다.

일단, **AES-128, AES-192, AES-256**이라는 단어를 많이 들어봤을 텐데 그것은 키의 길이로 결정된다.
3종류의 키를 사용할 수 있는데 라운드 함수 또한, 128bit 키 사용시에는 10라운드, 192bit에서는 12라운드, 256bit에서는 14라운드를 실행한다.

전체적인 알고리즘은 아래의 사진과 같다. (왼쪽 : 암호화, 오른쪽 : 복호화)



그리고 AES 알고리즘 라운드 내부에서는 4가지 연산이 존재한다. 1개의 자리바꿈 연산과 3개의 치환 연산인데 다음과 같다.

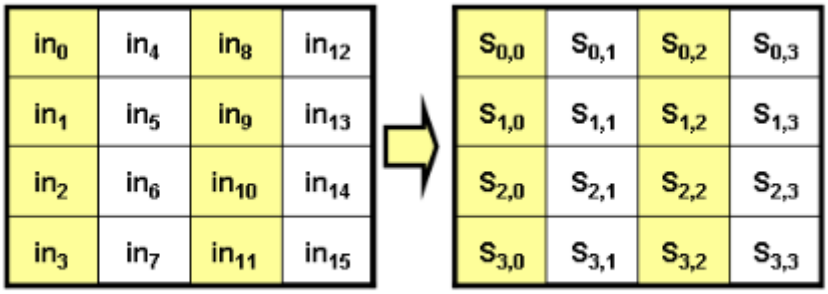
1. S-Box : $GF(2^8)$을 이용한 치환연산
2. Shift Row : 단순 자리바꿈
3. Mix Column : $GF(2^8)$을 이용한 치환연산
4. Add RoundKey : XOR 연산을 이용

여기서 $GF(2^8)$을 이용한 계산이란 기약 다항식[7] $m(x) = x^8 + x^4 + x^3 + x + 1$을 사용하는 다항식 체를 말한다.
위 네 가지 세부 연산은 모두 역이 가능하므로, 따라서 라운드 키를 적용하는 부분을 제외하고는 **그 자체로 어떤 안전성도 제공하지 않는다.**

---

1. **상태(State)**

   AES의 모든 연산들은 상태라고 하는 2차원 바이트 배열에 수행. 이 상태는 항상 4행으로 구성되며, 각 행은 N_B 바이트로 구성된다. AES에서 입력을 상태로 변환하는 방법은 아래 사진과 같다.



<그림 6.3> AES에서 입력을 상태로 변환하는 방법

예를 들어 128 비트 입력이 아래와 같다면, 결과 상태는 사진처럼 나온다.

> *EA835CF00445332D655D98AD8596B0C5*

EA	04	65	85
83	45	5D	96
5C	33	98	B0
F0	2D	AD	C5

<그림 6.4> AES의 상태의 예

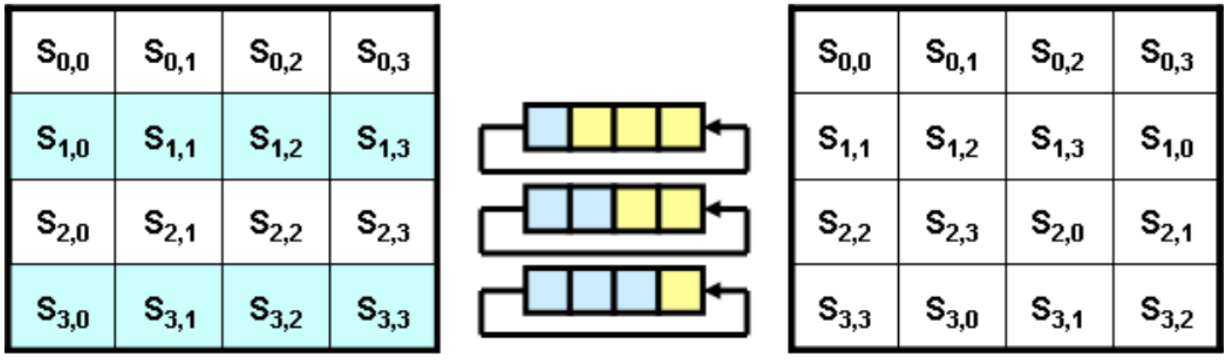
1. **S-Box 치환**

   암호화 과정의 각 라운드에서 가장 먼저 수행되는 연산이다. 2개의 S-Box가 존재 (전방향, 역방향) 두 S-Box는 서로 역 관계에 있으므로, 특정 바 이트 값을 전방향 S-Box로 친환한 후에 다시 역방향 S-Box로 돌리면 원래 값을 얻게 된다.

   전방향 S-Box는 암호화할 때 사용. 역방향 S-Box는 복호화할 때 사용한다.

2. **Shift Row(행이동 자리바꿈)**

   유일한 자리바꿈 연산인데 아래 그림과 같이 이루어진다.



<그림 6.8> AES의 행이동 자리바꿈 연산

   복호화 할 때에는 정반대로 이루어진다. 이 연산은 자리바꿈을 통해 암호화 과정이 평문에 모든 비트에 고루 영향을 주도록 위함이다.

3. **Mix Column(열섞음 치환)**

   이 연산은 상태 행렬을 다음 행렬에 곱하여 값을 치환하게 되는데, 이 때 곱셈 연산은 $GF(2^8)$에서 계산된다.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} * [S] = [S']$$

위의 행렬을 두고, 계산 형태를 보여줄텐데 이 계산은 두 가지 형태로 계산 된다.

- **실제 행렬 곱셈을 아래와 같이 진행.**

$$s'_{(0,0)} = (02 * s_{(0,0)}) \oplus (03 * s_{(1,0)}) \oplus (03 * s_{(2,0)}) \oplus (03 * s_{(3,0)})$$

이 경우 이와 같은 계산을 16번 해야 한다.

- **입력 상태의 열을 아래와 3차 다항식으로 생각하여 진행.**

$$s_0(x) = s_{(3,0)}x^3 + s_{(2,0)}x^2 + s_{(1,0)}x + s_{(0,0)}$$

$x^4 + 1$에서 a(x)=\left{ 03 \right}x^3+\left{ 01 \right}x^2+\left{ 01 \right}x+\left{ 02 \right}에 곱하는 것이다.

이렇게 하여 얻어진 3차 다항식의 각 계수는 출력 상태의 열이 된다. 아래는 주어진 입력 상태에 열섞음 연산에 적용하는 결과를 보여준다.

$$\begin{bmatrix} 87 & F2 & 4D & 97 \\ 6E & 4C & 90 & EC \\ 46 & E7 & 4A & C3 \\ A6 & 8C & D8 & 95 \end{bmatrix} \rightarrow \begin{bmatrix} 47 & 40 & A3 & 4C \\ 37 & D4 & 70 & 9F \\ 94 & E4 & 3A & 42 \\ ED & A5 & A6 & BC \end{bmatrix}$$

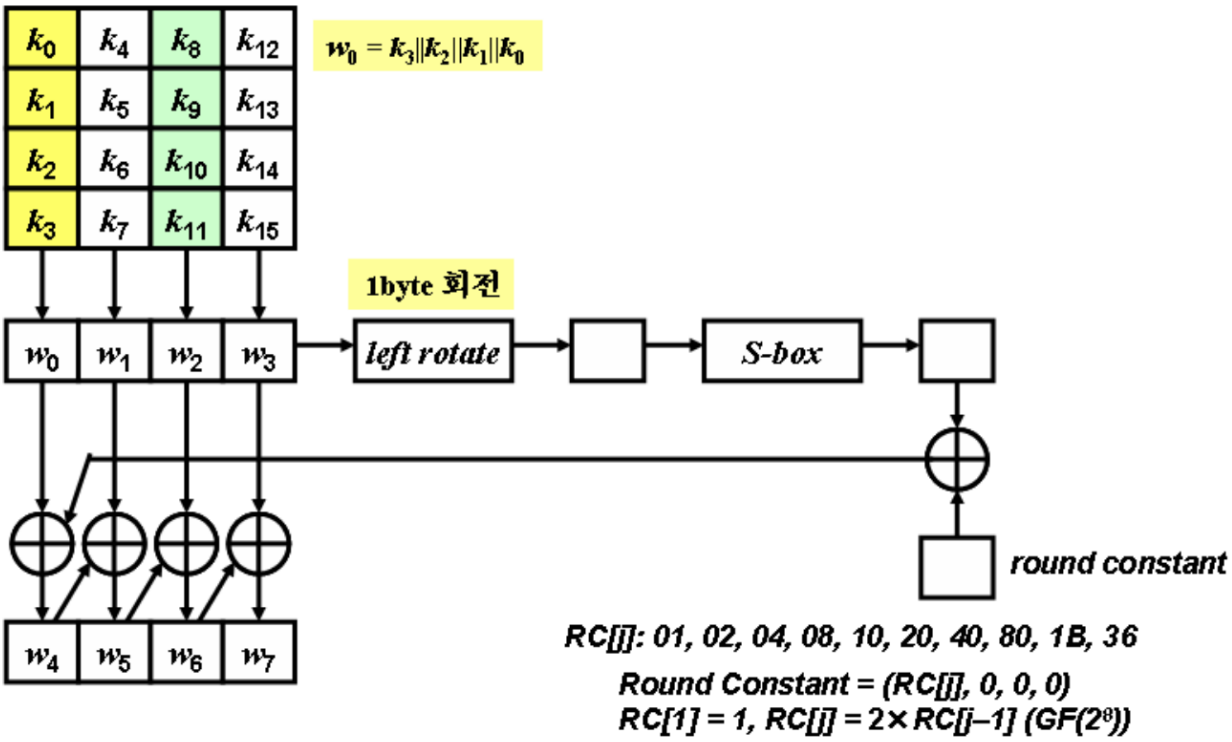
이 예시에서 $s'_{(0,0)}$이 47이 되는 과정을 살펴보면 다음과 같다. (1번 풀이 사용)

$(\{02\} \times \{87\}) \oplus (\{03\} \times \{6E\}) \oplus \{46\} \oplus \{A6\} = \{47\}$

해당 내용의 증명과 내용은 06-AES.pdf - index-of.co.uk (http://index-of.co.uk/Cryptology/06-AES.pdf)를 통해 자세히 볼 수가 있다.

1. **Add RoundKey**

AES는 사용되는 암호키의 길이가 128비트면, 총 44개의 32비트 워드로 확장되어 각 라운드 마다 4개의 32비트 워드를 라운드 키로 사용한다. 전체적인 알고리즘은 위의 그림과 같으며, 아래와 같이 진행된다.



<그림 6.10> AED 키 스케줄링 알고리즘

- STEP1 : 128비트의 키를 4개의 32비트 워드로 바꾼다.
- STEP2 : 첫 4개의 워드 중 마지막 워드는 1바이트 왼쪽 순환 이동된 뒤 S-Box를 이용하여 치환된다. 그 다음에 라운드 상수와 XOR 된다.
- STEP3 : 첫 워드와 기존 4개의 워드 중 두 번째 워드가 XOR 되어 두번째 워드가 생성, 이 결과를 반복한다.
- STEP4 : STEP2~3을 9번 수행하여 각 라운드 키를 생성.

이렇게해서 AES 알고리즘이 내부적으로 어떻게 동작하는지 알게 됐다. **허나, 여기서 끝난 것이 아니다!**

STEP 2.2 Block Chipher mode of operation

AES는 블록 암호 작동 모드(block cipher mode of operation)와 결합이되서 사용하는데 이는 비밀성이나 신뢰성과 같은 정보 서비스를 제공하기 위해 블록 암호를 사용하는 알고리즘이다.[7]
블록 암호 자체는 블록이라고 하는 하나의 고정 길이 비트 그룹의 보안 암호화 변환에만 적합하

다.[8]
작동 방식은 블록보다 큰 데이터 양을 안전하게 변환하기 위해 암호의 단일 블록 작동을 반복적으로 적용하는 방법을 설명한다.

대부분의 모드는 각 엄호화 작업에 흔히 IV(Initializtion Vector)[9]라고 하는 고유한 바이너리 시퀀스를 필요로 한다.
IV는 반복되지 않아야 하며, 일부 모드의 경우 랜덤해야 한다. **초기화 벡터는 동일한 일반 텍스트가 동일한 키로 여러 번 독립적으로 암호화되어도 별개의 암호 텍스트가 생성되도록 하기 위해 사용된다.**[10]
블록 암호 모드는 전체 블록에서 작동하며, 데이터의 마지막 부분이 현재 블록 크기보다 작으면 전체 블록에 패딩되어야 한다. (패딩을 필요로 하지 않는 모드도 존재)

- **IV(Initialization Vector)**
  우리가 알아야 할 핵심 **동일한 키에서 초기화 벡터를 사용하지 말자!**이다.
  CBC의 경우, IV를 재사용하면 첫 번째 블록의 평문에 대한 정보와 두 메세지가 공유하는 공통 접두사가 누설될 수 있다.
  OFB 혹은 CTR 모드의 경우 IV를 다시 사용하면 보안이 완전히 파괴된다.[10]
  내가 코드로 구현한 모드는 CBC인데, CBC 모드에서는 IV는 암호화시 예측할 수 없어야 한다.
  메시지의 마지막 암호문 블록을 다음 메시지의 IV로 다시 사용하는 방법을 적용한다 하면 안전하지가 않다!
  왜냐하면, 공격자가 다음 일반 텍스트를 지정하기 전에 IV를 알고 있으며 이전에 동일한 키로 암호화된 일부 블록의 일반 텍스트에 대한 추측할 수 있으며 공격으로 이어지기 때문이다.(TLS CBC IV Attack의 방식)[11]

- **Padding**
  블록 암호는 고정된 블록 크기에서 작동하지만, 메세지는 다양한 길이로 나타난다.
  좀 더 쉽게 얘기를 하자면, 데이터(메세지)를 블럭으로 암호화 할 때 평문이 항상 블럭 크기(일반적으로 64비트 / 128비트)의 배수가 되지 않을 경우가 존재한다.
  패딩은 **어떻게 평문의 마지막 블록이 암호화 되기 전에 데이터로 채워지는가를 확실히 지정하는 방법** 이다. 복호화 과정에서는 패딩을 제거하고, 평문의 실제 길이를 지정하게 된다.

  간단하게 설명하자면 **암호 블록 사이즈와 데이터 사이즈가 맞지 않을 경우에 배수에 맞춰 빈 공간을 채워주는 방식이라고 볼 수가 있다.**

  아래 PKCS#5와 PKCS#7의 예시를 보자.

  - **PKCS#5**
    암호 블록 사이즈가 8바이트에 맞춰져있다.
    8바이트의 배수로 인풋을 맞춰줘야하는데 패딩에 들어가는 문자는 패딩할 갯수가 들어간다.

    - 예시
      AA 07 07 07 07 07 07 07 : 1바이트 데이터 + 7바이트 패딩
      AA BB 06 06 06 06 06 06 : 2바이트 데이터 + 6바이트 패딩
      AA BB CC 05 05 05 05 05 : 3바이트 데이터 + 5바이트 패딩
      주의 : 데이터가 8바이트인 경우에는 패딩이 필요 없어 보여도, 08을 8번 패딩한다.

      AA A9 A8 A7 A6 A5 A4 A3 08 08 08 08 08 08 08 08 : 8바이트 데이터 + 8바이트 패딩
  - **PKCS#7**
    패딩할 숫자만큼 패딩 값을 채워넣어서 붙여주는 것은 PKCS#5와 동일.
    따라서, 8바이트의 암호화 블럭 크기인 경우 **PKCS#5 = PKCS#7**이다.
    하지만 현대의 암호화에서는 당연히 더 긴 암호화 블럭을 사용하기 때문에 달라진다.
    AES256의 예를 들면, 256bit의 키를 사용하게 되고, 블럭 사이즈는 128bit이므로 32바이트의 키와 16바이트의 암호화 블럭을 사용하게 된다.

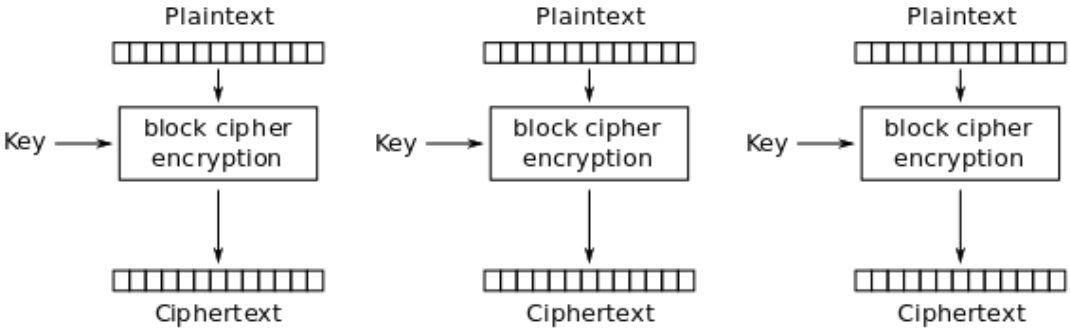
    이 상황에서 PKCS#7은 패딩이 최대 16개까지 가능하다.
    (8바이트인 경우 08을 패딩 8번 하듯이 10으로 패딩 16번을 수행한다.)
    PKCS#7은 최대 가능한 패딩은 FF이므로, 255개가 된다.

    두 개의 차이라 하면, PKCS#5는 8바이트 고정길이, PKCS#7은 1~255바이트의 가변길이이다.

- **Common modes**
  위에서 설명한 내용을 토대로 그러면 어떠한 방식의 블록 암호 운용 방식이 존재하는지 설명하고자 한다.
  여기서의 핵심은 **"블록 암호는 특정한 길이의 블록 단위로 동작하기 때문에, 가변 길이 데이터를 암호화하기 위해서는 먼저 이들을 단위 블록들로 나누어야 하며, 그리고 그 블록들을 어떻게 암호화할지를 정해야 한다."** 이다.
  간단히 얘기하자면, **모드는 블럭 암호화 순서 및 규칙에 대한 표준이라고 보면 될 거 같다.**

- **ECB(Electronic CodeBook)**
  가장 간단한 운용 방식을 가지며, 암호화하려는 평문 메세지를 여러 블록으로 나누어 각각 암호화하는 방식이다.
  ECB는 단점때문에 많이 사용을 안하는데 그 단점은 "같은 암호화 키"를 사용하는 것이다.
  (확산의 부족, Lack of Diffusion)
  즉, 평문 메세지를 여러 블록을 나누어서 암호화 하기때문에 만약, 암호화 메세지를 여러 부분을 나누었을 때 두 블록이 같은 값을 가진다면, 암호화한 결과 역시 같다라는 특징이 있으므로 공격자가 비슷한 메세지를 반복적으로 암호화하는 반복공격에 취약한 성질은 가진다. (확산의 부족으로 설명했는데 동일한 암호 텍스트로 암호화를 하므로, 데이터 패턴이 잘 숨겨지지가 않는다는 뜻)

  그 예시로 다음 그림을 참고하자.

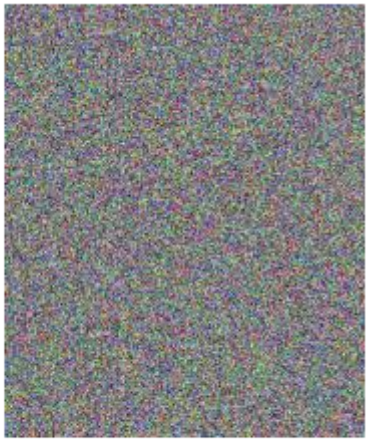


Electronic Codebook (ECB) mode encryption

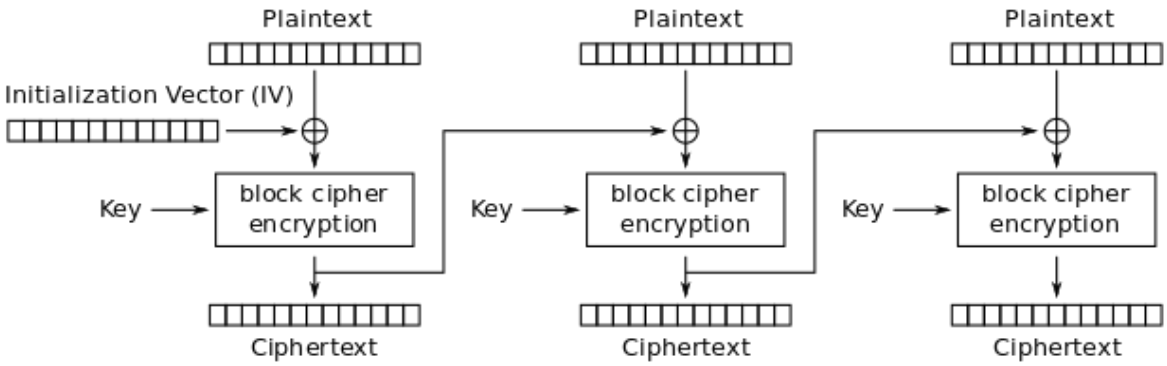
ECB모드가 비트맵 이미지를 암호화하는데 사용될 때를 보면, 각 개별 픽셀의 색상은 암호화되지만, 원본에서 동일한 색상의 픽셀 패턴이 암호화된 버전에 남아 있기 때문에 전체 이미지를 공격자가 인식할 수 있음을 보여준다.


Original image


Encrypted using ECB mode


Modes other than ECB result in pseudo-randomness

- **CBC(Cipher-Block Chaining)**
  CBC는 간단하게 알고리즘만 보자면, 각 블록은 암호화되기 전에 이전 블록의 암호화 결과와 XOR되는데,최초 시행 시에는 최초 평문 블럭과 IV를 XOR연산한다.
  이를 끝까지 반복하는데 평문의 마지막 블럭은 패딩된 블럭이다.
  이 때, IV가 같은 경우 출력 결과가 항상 같기 때문에, 매 암호화마다 다른 IV를 사용하는 것이 매우 중요하다.



Cipher Block Chaining (CBC) mode encryption

STEP 3. 구현

쓰다보니 매우 장문의 글이 되버려서 참 난감하다. 하지만, 이 글을 천천히 여기까지 읽었던 독자분들이라면, 아래의 코드 또한 매우 쉽게 이해 할 수 있다고 다짐한다.

STEP 3.1 C# (AES256/CBC/PKCS#7)

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace Encryption
{
    class AES
    {
        private readonly static string keyStr = "This is Key";
        private readonly static string vector = "This is Vector";

        static void Main(string[] args)
        {
            Encrypt("C:\\test\\100-Sales-Records.zip", "C:\\test\\100-Sales-
RecordsEncrypted.enc");

        }
        /**
         * 32자리의 키값을 이용하여 Rfc2898DeriveBytes 생성
         * @param  password                    절대 유출되서는 안되는 키 값이며, 이
것으로 암호키를 생성
         */
        public static Rfc2898DeriveBytes MakeKey(string password)
        {

            byte[] keyBytes = System.Text.Encoding.UTF8.GetBytes(password);
            byte[] saltBytes = SHA512.Create().ComputeHash(keyBytes);
            Rfc2898DeriveBytes result = new Rfc2898DeriveBytes(keyBytes,
saltBytes, 65536);

            return result;
        }
        /**
         * 16자리의 초기화 벡터값을 이용하여 Rfc2898DeriveBytes 생성
         * @param  iv                    절대 유출되서는 안되는 초기화 벡터 값이며,
이것으로 초기화벡터를 생성
         */
```

```
        public static Rfc2898DeriveBytes MakeVector(string vector)
        {

            byte[] vectorBytes = System.Text.Encoding.UTF8.GetBytes(vector);
            byte[] saltBytes = SHA512.Create().ComputeHash(vectorBytes);
            Rfc2898DeriveBytes result = new Rfc2898DeriveBytes(vectorBytes,
saltBytes, 65536);

            return result;
        }
        /**
         * 복호화 처리 레지달 알고리즘을 사용하여 AES256-CBC 구현.
         * @param inputFile             암호화할 파일
         * @param outputFile            복호화한 후의 파일명
         * @Step
         *  1. File.ReadAllBytes를 통하여 파일의 모든 Byte를 읽어들임.
         *  2. csEncrypt를 활용하여 AES256-CBC로 Encrypte (PlainFile byte[] ->
AES256 Encrypted byte[])
         *  3. 해당 값을 메모리 스트림에 적재
         *  4. msEncrypt.ToArray() : 메모리에 적재된 값을 배열로 읽어 byte[]
encrypted에 적재. (AES256 Encrypted byte[])
         *  5. AES256 Encrypted byte[] -> Base64 Encoded String
         *  6. Base64 Encoded String Write on the dest File
         */
        public static void Encrypt(String source, String dest)
        {
            using (RijndaelManaged aes = new RijndaelManaged())
            {
                //Create Key and Vector
                Rfc2898DeriveBytes key = AES.MakeKey(AES.keyStr);
                Rfc2898DeriveBytes vector = AES.MakeVector(AES.vector);

                //AES256
                aes.BlockSize = 128;
                aes.KeySize = 256;

                // It is equal in java
                // Cipher _Cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
                aes.Mode = CipherMode.CBC;
                aes.Padding = PaddingMode.PKCS7;
                aes.Key = key.GetBytes(32); //256bit key
```

```
                aes.IV = vector.GetBytes(16); //128bit block size


                //processing Encrypt
                ICryptoTransform encryptor = aes.CreateEncryptor(aes.Key, aes.IV);
                byte[] encrypted;

                using (MemoryStream msEncrypt = new MemoryStream())
                {
                    using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
encryptor, CryptoStreamMode.Write))
                    {
                        byte[] inputBytes = File.ReadAllBytes(source);
                        csEncrypt.Write(inputBytes, 0, inputBytes.Length);
                    }
                    encrypted = msEncrypt.ToArray();
                }
                string encodedString = Convert.ToBase64String(encrypted);
                File.WriteAllText(dest, encodedString);
            }
        }
    }
}
```

STEP 3.1 JAVA (AES256/CBC/PKCS#7)

```
package example;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.SecretKeySpec;
import java.io.*;
import java.security.Key;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;
import java.util.Arrays;
import java.util.Base64;

public class Decryption {
    private static final String algorithm = "AES";
    //Java에서는 PKCS#5 = PKCS#7이랑 동일
    //자세한 내용은 http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html (http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html) 참고.
    private static final String blockNPadding = algorithm + "/CBC/PKCS5Padding";
    private static final String password = "This is Key";
    private static final String IV = "This is Vector";

    private static IvParameterSpec ivSpec;
    private static Key keySpec;

    public static void setIvSpec(IvParameterSpec ivSpec) {
        Decryption.ivSpec = ivSpec;
    }
    public static void setKeySpec(Key keySpec) {
        Decryption.keySpec = keySpec;
    }
    public static void main(String[] args) throws Exception {
        MakeKey(password);
        MakeVector(IV);
        // Test-file "100 Sales Records" (5KB zip-file) downloaded at http://eforexcel.com/wp/downloads-18-sample-csv-files-data-sets-for-testing-sales/ (http://eforexcel.com/wp/downloads-18-sample-csv-files-data-sets-for-testing-sales/)
```

```
        // and encrypted (100-Sales-RecordsEncrypted.enc) using the unchanged C#
code
        new Decryption().decrypt(new File("C:/test/100-Sales-
RecordsEncrypted.enc"), new File("C:/test/100-Sales-RecordsDecrypted.zip"));

    }
    /**
     * 32자리의 키값을 이용하여 SecretKeySpec 생성
     * @param  password                    절대 유출되서는 안되는 키 값이며, 이것으로
키스펙을 생성
     * @throws UnsupportedEncodingException 지원되지 않는 인코딩 사용시 발생
     * @throws NoSuchAlgorithmException     잘못된 알고리즘을 입력하여 키를 생성할 경
우 발생
     * @throws InvalidKeySpecException      잘못된 키 스펙이 생성될 경우 발생
     */
    public static void MakeKey(String password)
            throws NoSuchAlgorithmException, UnsupportedEncodingException,
InvalidKeySpecException {
        //암호키를 생성하는 팩토리 객체 생성
        SecretKeyFactory factory =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
        //다이제스트를 이용하여, SHA-512로 단방향 해시 생성 (salt 생성용)
        MessageDigest digest = MessageDigest.getInstance("SHA-512");

        // C# : byte[] keyBytes = System.Text.Encoding.UTF8.GetBytes(password);
        byte[] keyBytes = password.getBytes("UTF-8");
        // C# : byte[] saltBytes = SHA512.Create().ComputeHash(keyBytes);
        byte[] saltBytes = digest.digest(keyBytes);

        // 256bit (AES256은 256bit의 키, 128bit의 블록사이즈를 가짐.)
        PBEKeySpec pbeKeySpec = new PBEKeySpec(password.toCharArray(), saltBytes,
65536, 256);
        Key secretKey = factory.generateSecret(pbeKeySpec);

        // 256bit = 32byte
        byte[] key = new byte[32];
        System.arraycopy(secretKey.getEncoded(), 0, key, 0, 32);
        //AES 알고리즘을 적용하여 암호화키 생성
        SecretKeySpec secret = new SecretKeySpec(key, "AES");
        setKeySpec(secret);
    }
```

```
    /**
     * 16자리 초기화벡터 입력하여 ivSpec을 생성한다.
     * @param  IV                       절대 유출되서는 안되는 키 값이며, 이것으로 키스
펙을 생성
     * @throws UnsupportedEncodingException 지원되지 않는 인코딩 사용시 발생
     * @throws NoSuchAlgorithmException     잘못된 알고리즘을 입력하여 키를 생성할 경
우 발생
     * @throws InvalidKeySpecException      잘못된 키 스펙이 생성될 경우 발생
     * @
     */
    public static void MakeVector(String IV)
            throws NoSuchAlgorithmException, UnsupportedEncodingException,
InvalidKeySpecException {
        SecretKeyFactory factory =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
        MessageDigest digest = MessageDigest.getInstance("SHA-512");
        byte[] vectorBytes = IV.getBytes("UTF-8");
        byte[] saltBytes = digest.digest(vectorBytes);

        // 128bit
        PBEKeySpec pbeKeySpec = new PBEKeySpec(IV.toCharArray(), saltBytes, 65536,
128);
        Key secretIV = factory.generateSecret(pbeKeySpec);

        // 128bit = 16byte
        byte[] iv = new byte[16];
        System.arraycopy(secretIV.getEncoded(), 0, iv, 0, 16);

        IvParameterSpec ivSpec = new IvParameterSpec(iv);
        setIvSpec(ivSpec);
    }
    /**
     * 원본 파일을 복호화해서 대상 파일을 만든다.
     * @param source 원본 파일
     * @param dest 대상 파일
     * @throws Exception
     */
    public void decrypt(File source, File dest) throws Exception {
        Cipher c = Cipher.getInstance(blockNPadding);
        c.init(Cipher.DECRYPT_MODE, keySpec, ivSpec);
        fileProcessing(source, dest, c);
```

```
    }
    /**
     * 파일 복호화 처리
     * @param source 원본 파일
     * @param dest   대상 파일
     * @param c      생성된 Cipher 객체 전달
     * @throws Exception
     * @Step
     *  1. 생성한 파일의 버퍼를 읽어들임.
     *  2. Base64 인코딩된 문자열 -> Base64 디코딩 Byte[]로 변환
     *  3. Base64 디코딩 Byte[] -> Cipher.update를 사용하여 AES256 Decryption 실행
     *  4. Cipher.doFinal()로 마지막 Padding을 추가.
     */
    public void fileProcessing(File source, File dest, Cipher c) throws Exception
{
        InputStream input = null;
        OutputStream output = null;

        try {
            input = new BufferedInputStream(new FileInputStream(source));
            output = new BufferedOutputStream(new FileOutputStream(dest));
            byte[] buffer = new byte[4 * (input.available() / 4)];
            int read = -1;
            while ((read = input.read(buffer)) != -1) {
                byte[] bufferEncoded = buffer;
                if (read != buffer.length) {
                    bufferEncoded = Arrays.copyOf(buffer, read); //버퍼에 읽힌 값을
bufferEncoded에 Array Copy
                }
                byte[] bufferDecoded = Base64.getDecoder().decode(bufferEncoded);
//Base64 Decode
                output.write(c.update(bufferDecoded)); //AES256 Decryption
            }
            output.write(c.doFinal()); // Last Padding add
        } catch (BadPaddingException e){
            e.printStackTrace();
        } finally {
            if (output != null) {
                try {
                    output.close();
                } catch (IOException e) {
```

```
                }
            }
            if (input != null) {
                try {
                    input.close();
                } catch (IOException e) {
                }
            }
        }
    }
}
```

코드를 보면, 돌아가는 코드이나 정말 수정될 부분들이 많은 코드이다. 어떻게 하면 좀 더 보안성을 높을지는 충분히 이번 포스팅에 작성했다고 생각을 한다.

만약, 이 코드를 실제 운영서버에 적용한다하면 말리고 싶다. 허나, 더 좋은 코드로 탈바꿈하여 운영서버에 적용하기에는 좋다고 말하고 싶다. (저도 그러고 있습니다.)

이상으로 포스팅을 마친다.

REFERENCE

1. 안전한 패스워드 저장 - Naver D2 (https://d2.naver.com/helloworld/318732)
2. 암호이용활성화 - KISA (https://seed.kisa.or.kr/iwt/ko/intro/EgovCryptographic.do)
3. Block cipher mode of operation - Wikiwand (http://www.wikiwand.com/en/Block_cipher_mode_of_operation)
4. 공개키알고리즘 Part.1 - 카피캣의 쉬운 암호학 (http://cryptocat.tistory.com/3)
5. AES(ADVANCED ENCRYPTION STANDARD) 암/복호화 알고리즘 - 감파고 (http://newstein03.tistory.com/1)
6. AES 암호 알고리즘(Adavanced Encryption Standard) - Crocus (https://www.crocus.co.kr/1230)
7. 06-AES.pdf - index-of.co.uk (http://index-of.co.uk/Cryptology/)
8. Advanced Encryption Standard(AES), Federal Information Processing Standards Publication 197, Nov. 2001 (http://jima.me/wp-content/uploads/2016/05/Advanced-Encryption-Standard-Wikipedia-the-free-encyclopedia.pdf)
9. [Java] PKCS7padding NoSuchAlgorithmException - Tistroy (http://devbible.tistory.com/451)

1. Rainbow Table - Wikipedia (https://en.wikipedia.org/wiki/Rainbow_table) ↩
2. Dictionary Attack - Wikipedia (https://en.wikipedia.org/wiki/Dictionary_attack) ↩
3. Salt(cryptography) - Wikipedia (https://en.wikipedia.org/wiki/Salt_(cryptography)) ↩
4. Brute-force Attack - Wikipedia (https://en.wikipedia.org/wiki/Brute-force_attack) ↩
5. Key derivation function - Wikipedia (https://en.wikipedia.org/wiki/Key_derivation_function) ↩
6. 공개 키 암호 방식 - 위키백과 (https://ko.wikipedia.org/wiki/%EA%B3%B5%EA%B0%9C_%ED%82%A4_%EC%95%94%ED%98%B8_%EB%B0%A9%EC%8B%9D) ↩
7. NIST Computer Security Division's (CSD) Security Technology Group (STG) (2013). "Block cipher modes". Cryptographic Toolkit. NIST. Archived from the original on November 19, 2012. Retrieved April 12, 2013. ↩ ↩2
8. Cryptography Engineering: Design Principles and Practical Applications. Ferguson, N., Schneier, B. and Kohno, T. Indianapolis: Wiley Publishing, Inc. 2010. pp. 63, 64. ISBN 978-0-470-47424-2. ↩
9. 초기화 벡터 - 위키백과 (https://ko.wikipedia.org/wiki/%EC%B4%88%EA%B8%B0%ED%99%94_%EB%B2%A1%ED%84%B0) ↩
10. Kuo-Tsang Huang, Jung-Hui Chiu, and Sung-Shiou Shen (January 2013). "A Novel Structure with Dynamic Operation Mode for Symmetric-Key Block Ciphers" (PDF). International Journal of Network Security & Its Applications (IJNSA). 5 (1): 19. Archived (PDF) from the original on 2015-11-22. ↩ ↩2
11. B. Moeller (May 20, 2004), Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures, archived from the original on June 30, 2012 ↩

태그: (https://dailyworker.github.io/tags/) Chiphermode (https://dailyworker.github.io/tags/#chiphermode) Cryptography (https://dailyworker.github.io/tags/#cryptography) Java (https://dailyworker.github.io/tags/#java) Security (https://dailyworker.github.io/tags/#security)

업데이트: January 16, 2019

댓글남기기