**codebook & chiper**

code : 어떤 메시지를 암호화하는데 사용하는 방법(단어나 구가 다른 어떤것으로 변환하게 하는 방법)

codebook : 단어나 구절을 암/복호화 하는데 사용하는 문서

cipher : 개별적인 문자, bit 수준에서 메시지 암호화

메세지가 코드에 의해 변환되고 그 뒤 chiper에 의해 변환된다.

chiper : 암호, 암/복호화 시스템/알고리즘

평문 : plaintext, clear text

암호문 : chipertext, cypertogram

**Cryptography : 암호기법 (암호화 방법)**

Cryptanalysis : 암호해독 (키 없이 복원) => Cryptanalyst (암호문 깸)

Cryptology : 암호학(키로 암/복호화) => Cryptographers (암호문 생성)

**Cryptography : 암호기법 (암호화 방법)**

메시지를 쉽게 이해할 수 없는 형태로 바꾸고, 다시 원래의 형태로 복원하는 것.

기밀성, 데이터 무결성, 부인방지, 인증을 포함하는 정보보안을 제공하는 수단이나 방법

**Symmetric key, Asymmetric key**

key => 기밀성, 무결성, 인증 등을 하기위해 사용하는 암호화 알고리즘의 input

키를 안전하게 보관하는 것이 중요 => TPM(trusted platform module), secure co-processor, smartcards...

**keyspace**

모든 가능한 키들의 집합이다.

keyspace가 크면 안전하지만 bit길이가 길어지고, 그만큼 계산량도 많아진다.

**crypto(암호화)**

Only the key is secret => 케르크호프스의 원칙

공격자는 시스템에대해 알고 오직 키만 모른다.

암호화 알고리즘은 비밀이 아니며, 오직 키만 비밀이다.

비밀 알고리즘은 노출 되었을때 위험해지고, 언젠가는 노출된다.

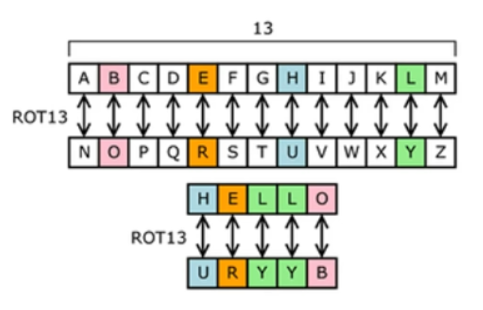
따라서 open하여 취약점들을 없앤다.

Substitution(대치,치환)/Transposition(전치(위치를 바꿈))

Substitution(대치, 치환)

1:1 혹은 1:多 대응

하나의 symbol을 또 다른것으로 대체함



예시)

1. caeasar 암호(= shift chiper, Additive cipher)



shift chiper, Additive cipher

테이블이(가) 표시된 사진

자동 생성된 설명

k = key만큼 shift함

ex) key = 7

테이블이(가) 표시된 사진

자동 생성된 설명

C = (P + k) mod 26

P = (C - k) mod 26

Brute-force attack

위의 방법으로 암호화된 암호문을 Key가 1일때, 2,3,4,5,...26 일때의 경우를 모두 조사하면 평문을 찾을 수 있다.

따라서 위의 암호화 방법은 안전하지 않으므로 keyword를 추가하여 복잡하게 만들 수도 있다.

2. keyword 추가(전수조사 막기위해)

예를 들어 keyword가 ZEBRAS이면

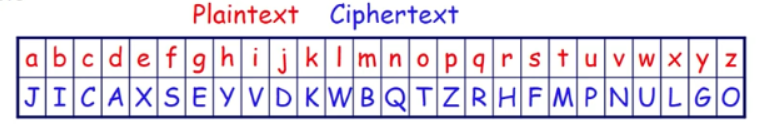
텍스트이(가) 표시된 사진

자동 생성된 설명

ZEBRAS를 먼저 놓고, Z, E, B, R, A, S를 제외한 알파벳을 놓아 평문을 치환한다.

=> 가능한 keyCaesar 암호화보단 깨기 어려워진다.

또는



알파벳을 순차적으로 대응하지 않고 섞어서 대응하여도 복잡해진다.

하지만, 위의 방식들은 모두 1대1 대응이므로 문자의 빈도수를 세면 유추가능해진다.

항상 고정적으로 하나의 평문 문자가 하나의 암호문 문자로 대응되어 유추하기 쉽다. => 단일문자 치환 암호

(자주나오는 모음이나 diagram(2개문자), tigram(3개문자))

3. 비즈네로 암호 (빈도수 유추 막기위해)

예시) keyword - pascal

먼저 평문의 알파벳을 순서에 맞는 숫자에 대응하고

키워드 문자를 숫자에 대응하여

C = (P + K) mod 26 하여 암호문을 구한다.

테이블이(가) 표시된 사진

자동 생성된 설명

s -> H, h- >H

s -> H, s -> S

와 같이 1대1 대응이 아니라 1대多 대응이 된다.

따라서 빈도수 공격을 완화시킬 수 있다.

비즈네로 암호는 이러한 방법을 사용한다.

실제론, 아래와 같이 비즈네르 table을 이용하여

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

키워드(Darkly)를 대응시켜 (K, P)와 같이 대응되는 문자를 테이블에서 찾아 암호문을 만든다.

하지만 이 방법도 키워드가 반복되므로 주기(키 길이)를 예측 할 수 있어 유추 가능해진다(암호 원리 파악 가능).

(평문 내용과 키워드 위치가 반복된다, 언어구조와 빈도 정보로 유추)

텍스트이(가) 표시된 사진

자동 생성된 설명

따라서 keyword의 길이를 평문길이만큼으로 하는 one-time-password을 사용하면 해결할 수 있지만, keyword를 관리하는 것이 어렵다.

4. one time pad

깨질 수 없는 안전한 암호 기법이다. 그 이유는 ...

1. 키(s)의 길이가 평문과 같거나 길다. => 암호문은 평문에 대한 어떠한 정보도 제공하지 않는 것처럼 보인다.
2. 키(s) 정보는 두 당사자만 알고 있다.
3. otp는 랜덤 생성해야하며 한번만 사용하므로 이전에 사용한 키는 사용되지 않는다.

하지만 키의 분배가 어려운 문제가 있다. 따라서 이론적으로만 안전하다.

c= m xor s, 평문을 키와 xor연산하여 암호문을 얻는다.

테이블이(가) 표시된 사진

자동 생성된 설명

(평문이 8개의 문자를 사용하여 8bit 키를 사용함)

테이블이(가) 표시된 사진

자동 생성된 설명

복호화 역시 같은키로 한다.

하지만 여기서 Alice와 Bob이 키를 주고 받을때, 중간에 이중첩자 Charlie가 다른키를 Bob에게 알려준다면, Bob은 복호화문을 이해하기 어렵거나 확인을 위해 Alice에게 직접 찾아가야한다.

혹은 Alice가 포로로 잡혓을때, 적들에게 잘못된 키를 알려준다면 적들은 해당 키로 복호화한 평문이 진짜인지 아닌지 판단할 수 없다.

다른 방법으론,

5. keyword 뒤 평문을 배치하여 키 길이를 맞춘다. (키 = keyword + 평문) (keyword 반복 문제 해결)

텍스트이(가) 표시된 사진

자동 생성된 설명

현대 암호에서는 이러한 방식을 응용한다.

2bit의 평문을 S-Box(substitution box)를 통하여 4bit로 확장시킨다.

테이블이(가) 표시된 사진

자동 생성된 설명테이블이(가) 표시된 사진

자동 생성된 설명

정리하면,

monoalphabetic cipher(단일문자 치환 암호)는 전체 메시지에 대해 고정된 치환(1대1 대응)을 하여 빈도 수 공격으로 유추 가능해진다.

polyalphabeti cipher(복합 문자 치환 암호)는 평문에서 문자의 위치에 따라 대체되는 문자가 다르다 (1:多 대응),

하지만 기본적으로 문자 단위로 처리되므로, 키워드 반복시 주기를 알 수 있고, (모음), (모음은 자음에 의해 분리된다), (diagrams(연속되는 2개의 문자)), (Q뒤에는 U가 따라옴)과 같은 정보로 유추할 수 있다.

6. polygraphic substituion ciper (다중문자 대치) => Hill Cipher => Block 사용

하나의 평문 문자를 또 다른 암호문자 하나로 대응하면 즉, 문자 하나씩 암호문으로 변환할 경우에 빈도수 공격에 취약해진다.

다중문자 대치는 이웃한 문자를 그룹으로 만들어 그룹단위로 취급한다. => block 단위block의 크기를 n이라 하면

\_n = 2일 경우) \_

*평문 : MISSISSIPPI*

*Block : MI SS IS SI PP IK*

*(Block 크기를 위해 더미 k를 추가함)*

*MI -> EQ, SS->GC ....와 같이 치환한다.*

n, 즉 block이 커질 수록 빈도수 공격에 강해진다.

Playfiar cihper (n = 2) => 2x2 hill ciper이다.Y = Ax mod26(Y = 암호문, A = 키, x = 평문)다음과 같이 표현할 수 있다.

텍스트, 시계, 게이지이(가) 표시된 사진

자동 생성된 설명

평문 : AT, Key : CDDG 인 경우)C -> 3, D->4, D->4, G->6

따라서

텍스트, 시계, 손목시계, 게이지이(가) 표시된 사진

자동 생성된 설명

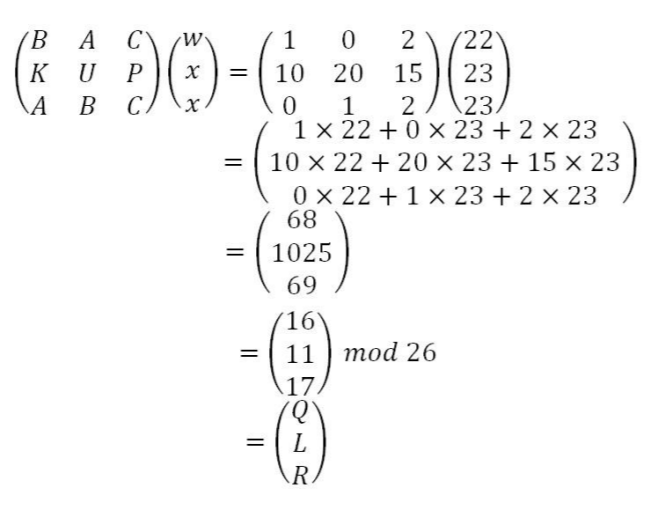
k를 표현할 수 있고, 평문 AT를 k로 암호화하면(k는 정방행렬로 만들어야한다. 암호화는 행렬의 곱이고 복호화는 역행렬을 만들어 곱을한다. 따라서 계산식을 쉽게 만들 수 있어야한다.)

테이블이(가) 표시된 사진

자동 생성된 설명

5, 10 => FK가 나온다.(mod 26인 이유는 알파벳을 숫자로 표현했기 때문)

n을 3으로한 경우)



3x4 즉, 블럭의 크기를 12로 하여도 된다. => block 크기 가변적

복호화는 k의 역행렬을 곱하고 mod26을 하여 평문을 얻을 수 있다.

긴 평문 예시)

예시 - WANT\_HELP.

WAN/ T\_H/ ELP/ ... (더미 .. 추가)

22 0 13/ 19 28 7/ 4 11 15/ 26 26 26 (알파벳을 숫자로 바꿈)

키 = A

텍스트, 시계이(가) 표시된 사진

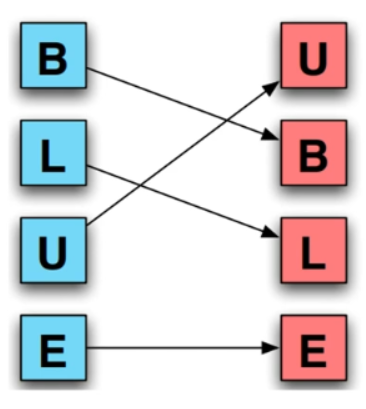
자동 생성된 설명테이블이(가) 표시된 사진

자동 생성된 설명

25 23 21/ 23 14 1/ 17 4 14/ 19 11 12

암호문 : ZWV XOB REO TLM

**Transposition(전치(위치를 바꿈))**



문자들의 순서를 바꾼다.

Columnar Transposition

1. simple

컬럼의 크기를 정하고 그에 맞게 평문을 자른다.

ENEMY TANKS APPROACHING HILL EIGHT SIX THREE

텍스트, 낱말맞추기게임, 전자기기, 키보드이(가) 표시된 사진

자동 생성된 설명

만약 컬럼의 개수의 배수가 아니면 더미를 추가한다.

그후, 컬럼별로 가져온다.

ENOHHR NKAITE ESCLSE MAHLIS YPIEXT TPNITO ARGGHP

만약 block의 크기가 5라면 5씩 잘라 표현한다.

ENOHH RNKAI TEESC LSEMA HLISY PIEXT TPNIT OARGG HPxxx (더미 추가)

2. 컬럼 순서 지정

컬럼에 순서(key)를 지정하여 순서에 따라 컬럼을 읽는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

혹은 알파벳(key)으로 순서를 줄 수도 있다.,

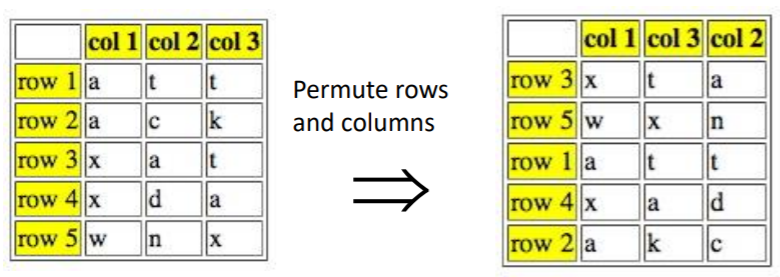
텍스트이(가) 표시된 사진

자동 생성된 설명

=> EATI THIH MEXN ETMG MEDT

3. Double Transposition (행과 열 둘다 사용)

평문 : attackxatxdawnx (공백 대신 x를 사용하였다.)



먼저 행을 (3,5,1,4,2)의 순서로 바꾸고 열을 (1,3,2)순으로 바꾼다.

열로 읽었을 경우 : xwaxa txtak antdc

행으로 읽을 경우 : xta wxn att xad akc

4. Route Transposition Cipher

4.1. Rail-Fence Cipher

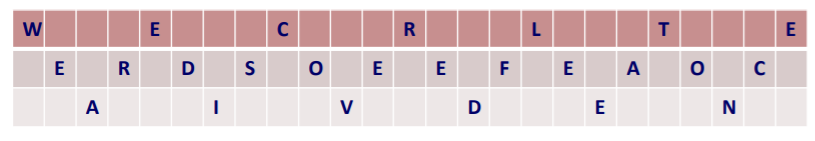
예시 rail 2개)

평문 = Meet me after the toga party



예시 rail 3개)

평문 = WE ARE DISCOVERED. FLEE AT ONCE



WECRL TEERD SOEEF EAOCA IVDEN

4.2. 위 방법외에 입력위치를 다르게 할 수 도 있다.

ex)

REINFORCEMENTS ARRIVING NOW

텍스트, 쇼지, 낱말맞추기게임, 창문이(가) 표시된 사진

자동 생성된 설명

NMRGI FEEAR NNEOC NSIIO RRTVW

혹은 삼각형 모양으로도 할 수 있다.

4.3. Triangular Pattern

텍스트, 쇼지, 낱말맞추기게임이(가) 표시된 사진

자동 생성된 설명

RMIFE VEONI RIRTN NCSGE ANROW

P- box

현대의 전치암호는 P(permutaiotion) box를 사용한다.

그냥 순서를 바꿔주는 p-box,

텍스트, 시계, 표지판이(가) 표시된 사진

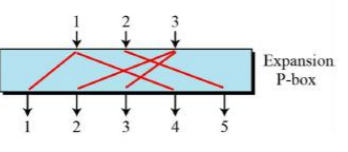
자동 생성된 설명

압축을 해주는 p-box

텍스트, 장치이(가) 표시된 사진

자동 생성된 설명

확장을 해주는 p-box



현대 암호인 AES는 Pbox를 사용한다.

튜링

튜링 머신

what can be computed. => 실제로 어디까지 계산가능한가?

어떤것, 어디까지 계산을 할 수 있는지 조사하는데 도움을 줄수 잇는 단순한 추상화된 컴퓨팅 기기

컴퓨터가 어디까지 가능한지(computability)를 나타내는 기본적인 모델임.

튜링 테스트

Can Machine think?

기계가 생각을 할 수 있는 가?

A : 기계 (질문시 사람인척 할 것임)

B : 사람

C(사람) : A, B가 사람인지 기계인지 판단(text기반 질문으로)

만약 C가 A, B를 구분하지 못한다면, A(기계)는 인공지능을 가진것으로 판단한다.

혼돈 : 암호문의 통계적 특성과 암호키 값과의 관계를 가능한 복잡하게 하는 기법, 암호문을 보고 원문을 유추하기 힘들다.

- 평문의 통계적 특성들을 잘 숨겨야한다.

- 복잡한 치환 알고리즘을 사용한다.

- 예시 : hill cipher

- 암호문의 각 문자는 키의 일부분에 의존함.

확산 : 암호문의 통계적 특성이 평문의 통계적 특성과 무관하게 하는 기법, 알고리즘의 패턴을 추론하기 힘들다.

- 평문의 통계적인 구조가 암호문에 골고루 흩어지게함

- 암호문에서 평문의 통계적 특성이 안나타남

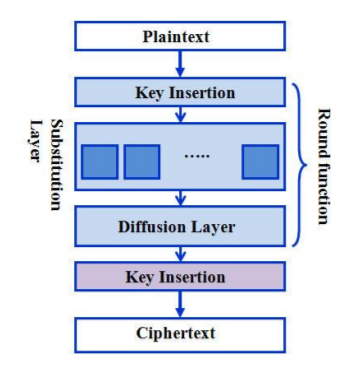
- 평문 블럭 또는 암호화키의 bit하나가 달라지면 암호문에서 큰 변화가 됨

시저암호화 비즈네로 암호화는 혼돈과 확산의 속성을 갖고 있지 않다. 따라서 빈도수 공격에 취약하다.

많은 현대 암호화들은 혼돈과 확산의 속성을 가지고 설계되었다.

DES : 혼돈과 확산을 가지는 블록 암호화이다.

AES : 혼돈과 확산을 가지며, substituion-permuatation network이다.



Block 암호화 : 64/128 bit block

- DES(64)

- AES(128)

- SEED

Block 단위로 처리되어 빈도수와 중복패턴을 줄여주기 때문에, block의 크기는 클수록 안전하다.

=> 시저암호는 블럭단위로 처리하지 않아 빈도수 공격에 취약하고 hill 암호화는 블럭단위로 처리하여 빈도수 공격에 안전하다.

Stream 암호화 : 1 bit 혹은 byte 단위로 처리한다.

- RC4 stream cipher

- otp => 빠르게 하기 위해 xor 연산을 한다. 키가 반복될시 위험하다.

Block 암호화 Stream 암호는 처리단위가 다르다.

DES와 AES 둘다 대칭키 암호화알고리즘이다.

DES는 64bits block이지만 AES는 128bits block이다.

키의 길이 또한 AES가 더 길다.

DES는 Substituion, permutation을 사용하지만, AES는 substitution, shift, bit mixing을 사용

DES 알고리즘은 비공개이고 AES는 공개되었다.

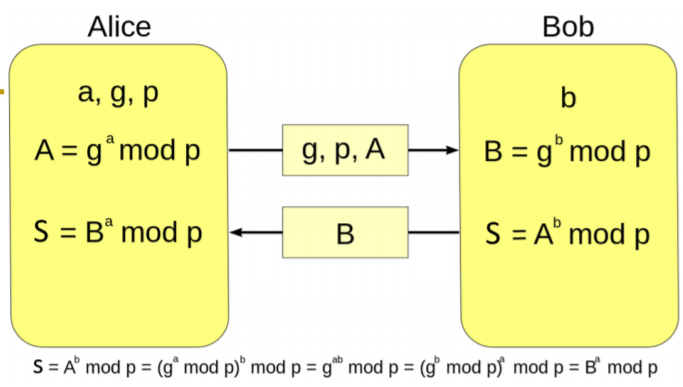
=> kerckhoff's principle: 키 제외 나머지는 모두 공개,  키의 길이가 길수록 안전성이 높아진다. (전사적 공격이 어려워짐)

대칭키

- 키 분배의 문제가 발생된다.

따라서

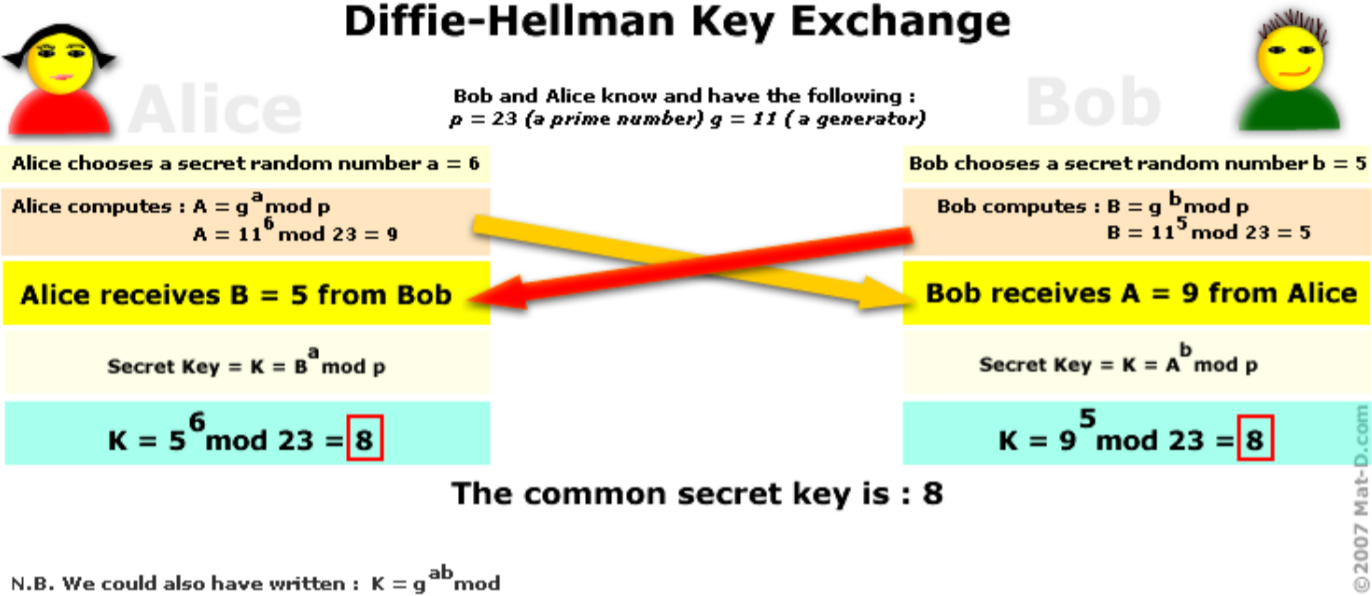
Diffie-Hellman 키 교환 : 안전하지 않은 통신 채널로도 키를 공유할 수 있게 된다.



Alice는 a, g, p를 정하고 Bob에게 g, p, A를 전달한다.

Bob은 b를 정하고 B를 Alice에게 전달한다.

Bob과 Alice는 각각 A, B로 같은 S를 얻는다.



위와 같은 방식으로 대칭키를 분배할 수 있다.

- 대칭키, 비대칭키 키의 개수

먼저, 대칭키는 n명이 있을때 n(n-1)/2만큼의 키가 필요하다 => O(n^2)

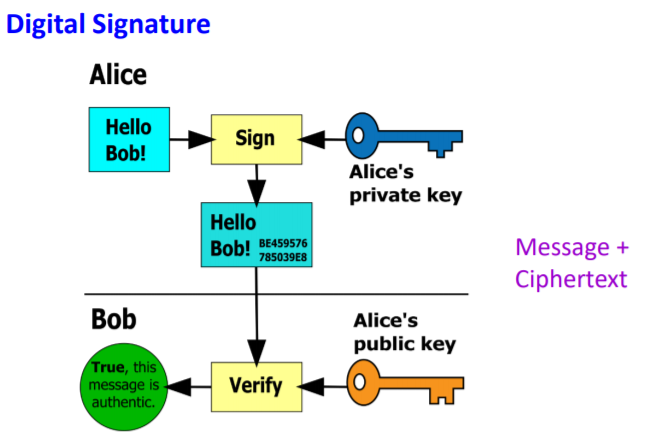
비대칭키는 n명이 있을때 2n만큼의 키가 필요하다 => O(n)

따라서 비대칭키가 필요한 키의 개수가 적다.

비대칭키를 사용하지 않고 대칭키를 사용하는 이유는 대칭키는 계산량이 비교적 적은 반면 비대칭키(공개키)는 계산량이 많기 때문이다.

비대칭키는 서명에도 사용된다.

A의 개인키로 암호화한다면 A의 공개키로 복호화 가능하므로 A라고 확인 할 수 있다.



대칭키 : 서로 비밀키를 공유, 키의 개수 n(n-1)/2, 메시지 암호화에 사용, 덜 복잡하여 빠르다. 기밀성, DES, AES, SEED, IDEA

공개키 : 인증, 서명, 짧은 메시지 암호화에 사용, 공개키로 암호화시 보낸사람을 확인하는 서명이 필요하다. 알고리즘이 복잡하여 느리다. 기밀성, 인증, 부인방지(전자서명), 공개키 관리(Public Key Insfrastructure), RSA, ECC, ElGamal

**\* 64 bit 구조에서 32 bit 프로그램으로 컴파일 하는 법**

gcc -m32 -o test test.c

-m32 옵션 컴파일시 bits/libc-header-start.h: No such file or directory 오류

=> sudo apt-get install gcc-multilib g++-multilib

**\* 형식 지정자**

%d => 부호가 있는 정수, decimal signed int

%u => 부호가 없는 정수, unsigned int

%i => any integer (decimal, octal, hexadecimal)

%hd => half decimal, decimal의 반틈이므로 2byte, short 이다.

%hu => unsigned short(unsigned int의 half)

%hhd => char (half의 half이므로 8bit=1byte), 8비트 단위의 부호가 있는 정수 출력

%hhu => unsigned char, 8비트 크기의 부호가 없는 정수 출력

%ld => decimal signed long

%lu => unsigned long

%o => octal integer, 8진수

%x => hexadecimal integer, 16진수

**\* printf (“%d, %u, %hd, %hu, %c\n”, -1, -1, -1, -1, -1);**

changmin@ubuntu:~/Documents$ ./test  
-1, 4294967295, -1, 65535, �

%d => 부호가 있는 정수이므로 -1이 출력

%u => 부호가 없는 정수이므로 언더플로우 발생하여 큰수가 됨

%hd => 2byte, 부호가 있는 short이므로 -1 출력

%hu => 2byte, 부호가 없는 short이므로 언더플로우 발생하여 unsigned short가 표현할 수 있는 최대 수가 됨

**\* 오버플로우 or wraparound**

어떤 정수 값이 지정된 16bit 크기에 저장될 수 없게 커지게 되면 overflow가 발생한다.

(매우작은수가 되거나 음수가 되버린다. 0이 되버리기도 한다 (표현할 수 없는 상위 비트는 날려버리기 때문))

오버 프로우가 발생하면 자원관리와 실행흐름에 문제가 생긴다.

결과적으로 약점이 생기고 C.I.A(기밀성, 무결성, 가용성)이 깨진다.

가용성 : 오버플로우로 인해서 예측하지 못하는 행동으로 연결될 수 있다(crash 유발가능). loop에서 인덱스로 사용될 경우 무한루프가 될 수도 있다. => DoS유발가능

무결성 : 데이터에서 중요한 값이 손상될 수 있다. 또한 buffer overflow를 일으킬 수 있다.(배열의 인덱스가 넘친다거나), 메모리 손상 유발 가능

기밀성, 접근제어 : 약점이 buffer overflow를 유발하고 그로인해 임의의 코드를 실행할 수 있게 된다. 따라서 보안 정책의 범위를 벗어나게 된다.

integer 오버플로우는 직접적으로 실행흐름을 바꾸거나 악용될 가능성은 적다.

하지만, 어떤 경우는 heap overflow 즉, 버퍼 오버플로우를 유발 할 수 있다.

따라서 임의의 코드를 실행할 수 있게 되는 더 나쁜 상황이 된다.

**오버플로우 예시)**

#include<stdio.h>

#include<stdio.h>

void main(void){

unsigend short us = 65535; //unsigned short max값

short ss = 32767; // short max값

unsigned char uc = 255; //max

signed char sc = 127; //max

printf("unsigned\_short = %d (%u), signed\_short = %d (%u)\n", us, us, ss, ss);

//unsigned short= 65535 (65535), signed short = 32767 (32767)

//%d, %u 는 signed int, unsigned int범위임으로 short 정상출력됨

printf("unsigned\_char = %d (%u), signed\_char = %d (%u)\n\n", uc, uc, sc, sc);

//unsigned char = 255 (255), signed char = 127 (127)

//%d, %u 는 signed int, unsigned int범위임으로 char 정상출력됨

printf("unsigned\_short = %hd, %hu, %hhd, %hhu\n", us, us, us, us);

//unsigned short= -1, 65535, -1, 255

//%hd는 signed int의 절반이므로 2byte, short범위이다. ->

//->65535가 1111 1111(2)이므로 2의 보수표현으로 생각하여 -1로 출력됨

//%hu는 unsigned int의 절반이므로 정상 출력된다.

//%hhd는 signed int의 절반의 절반이므로 1byte범위이다. ->

//->16bit를 8bit로 출력하므로 앞으 8bit는 잘리고 1111 1111 1111 1111(2)->

//->1111 1111(2), 위와 동일하게 2의 보수 표현으로 생각하여 -1이 출력됨

//%hhu는 unsigned int의 절반의 절반이므로 정상 출력된다.

printf("unsigned\_short+1 = %hd, %hu, %hhd, %hhu\n", us+1, us+1, us+1, us+1);

//unsigned short+1 = 0, 0, 0, 0

//unsigned short의 최대값에 1을 더하면 short의 범위를 벗어나게 된다.

//16bit를 넘어서서 할당된 16bit로 표현이 불가하여 0으로 출력된다.

printf("signed\_short = %hd, %hu, %hhd, %hhu\n\n", ss, ss, ss, ss);

//signed short = 32767, 32767, -1, 255

//%hhd는 signed int의 절반의 절반이므로 1byte(8bit), ->

//->short(16bit)인데 8bit로 출력하므로 위와 동일하게 앞 8비트가 잘리고 2의 보수표현으로 -1

printf("unsigned\_char = %hd, %hu, %hhd, %hhu\n", uc, uc, uc, uc);

//unsigned char = 255, 255, -1, 255

//%hhd는 부호가 있는 8bit이고, 부호가 없는 8bit를 출력할때 2의 보수 표현으로 -1이 출력

printf("unsigned\_char+1 = %hd, %hu, %hhd, %hhu\n", uc+1, uc+1, uc+1, uc+1);

//unsigned char+1 = 256, 256, 0, 0

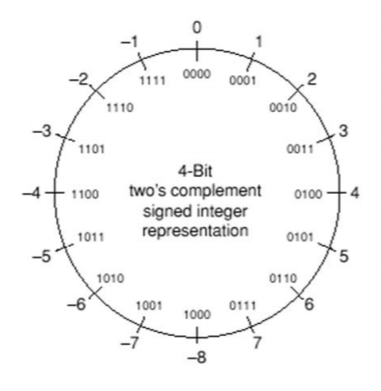
//%hhd, %hhu 둘다 8bit인데 unsigned\_char+1이 되면 1 0000 0000(2)가 되므로

// 0000 0000(2)로 0이된다.

}

**\* usigned와 signed 차이**

텍스트, 장치, 측정기이(가) 표시된 사진

자동 생성된 설명텍스트, 영수증이(가) 표시된 사진

자동 생성된 설명

2의 보수를 취하여 최상위 비트가 1이면 음수로 취급한다.

ex) 2+ (-3)

3 => 0011

1의 보수 => 1100

2의 보수 => 1101

따라서 0010 + 1101 = 1111

1111은 -1이된다.

8bit의 경우

0111 1111(2) ~ 1000 0000(2)가 범위가 된다 (1111 1111(2) 는 -1 => 위 원형 그림 참고)

127 ~ (-128)

- 2의 보수 덧셈

(-4) + (-1)

1100 + 1111 => 1 1011 => 1011 = -5

(-4) + (+4)

1100 + 0100 => 1 0000 = 0

(+5) + (+4)

0101 + 0100 => 1001 (-7) =오버플로우발생

(-7) + (-6)

1001 + 1010 => 1 0011 => 0011 (3)= 오버플로우 발생

**\* Overflow / Underflow**

overflow : INT\_MAX = 2147483647 (0X 7FFF FFFF), 값이 INT\_MAX보다 크면 segmentation fault가 유발된다.

underflow : INT\_MIN = 0x 8000 000 (0x8 = 1000(2)), 값이 INT\_MIN보다 작으면 segmentation fault

INT\_MAX = 2174483647(0x7FFF FFFF)

INT\_MIN = -2147483648(0x8000 0000)

UINT\_MAX = 4294967295(0xFFFF FFFF)

**오버플로우 예시)**

#include <stdio.h>

int main(void){

unsigned int num = 0xffffffff;

printf("num = %u (0x%x)\n", num, num);

printf("num + 1 = 0x%x\n", num + 1);

return 0;

}

/\* EOF \*/

The output of this program looks

like this:

num = 4294967295 (0xffffffff)

num + 1 = 0x0

num은 INT\_MAX이다.

%u = unsigned int이므로 num = 4294967295 (0xffffffff)

num+1은 INT\_MAX 범위를 벗어나므로 0x 1 0000 0000 => 0x 0000 0000 => 0x0이 된다.

#include <stdio.h>

int main(void) {

int n;

n = 0x7fffffff;

printf(“n = %d (0x%x)\n", n, n);

printf(“n + 1 = %d (0x%x)\n", n + 1 , n + 1);

return 0;

}

/\* EOF \*/

The output of which is:

n = 2147483647 (0x7fffffff)

n + 1 = -2147483648 (0x80000000)

n은  signed int에서 MAX 양수이다.

%d => 2147483647 (0x7fffffff)

n + 1 => 0x8000 0000가 된다. 이진수로 변환하면 1000 0000 0000 0000 0000 0000 0000 0000(2)이므로

2의 보수 표현으로 -2147483648이 된다.

**- 더하기/빼기/곱에서도 발생함**

#include <limits.h>

unsigned int ui1, ui2 , usum ;

/∗ Initialize ui1 and ui2 ∗/

usum = ui1 + ui2 ;

/\* ui1 = 0x7FFFFFF2

ui2 = 0x6FFFAAAA \*/

0x7FFF FFF2 + 0x6FFF AAAA를 덧셈하면 INT범위를 벗어나 오버플로우 발생

signed int si1, si2, result;

/\* initialize si1 and si2 \*/

result = si1 – si2;

/\* si1 = 0xFFFFBEEF

si2 = 0x6ABCCAFE \*/

0xFFFF BEEF - 0x6ABC CAFE를 하면 음수 - (양수)로 언더플로우가 발생한다.

(0xF = 1111(2) = 15)

signed int si1 , si2 , result;

/∗ Initialize si1 and si2 ∗/

result = si1 \* si2 ;

곱의 경우도 오버플로우가 발생한다

#include <limits.h>

int i;

unsigned int j;

i = INT\_MAX; // 2,147,483,647

i++;

printf (“i = %d \n”, i);

j = UINT\_MAX; // 4,294,967,295

j++;

printf (“j = %u \n”, j);

출력

i = -2,147,483,648

j = 0

i=INT\_MAX의 경우 +1을 하여 오버플로우

j=UINT\_MAX의 경우 +1을 하여 오버플로우로 0이됨

#include <stdio.h>

int main(void){

int l, x;

l = 0x40000000;

printf("l = %d (0x%x)\n", l, l);

x = l + 0xc0000000;

printf("l + 0xc0000000 = %d (0x%x)\n", x, x);

x = l \* 0x4;

printf("l \* 0x4 = %d (0x%x)\n", x, x);

x = l - 0xffffffff;

printf("l - 0xffffffff = %d (0x%x)\n", x, x);

return 0;

}

l = 1073741824 (0x4000 0000)

l + 0xc000 0000 = 0 (0x0)

(0xc = 12 = 1100(2) 따라서 최상위 비트가 1이므로 음수임)

l \* 0x4 = 0 (0x0)

0x4를 곱하면 큰수가 됨

l - 0xffffffff = 1073741825 (0x4000 0001)

(0xFFFF FFFF = -1)이므로 +1이 됨

**언더 플로우 예시)**

i = signed int, j = unsigned int

i = INT\_MIN;

i--;

printf("i = %d\n", i);

j = 0;

j--;

printf("j = %u\n", j);

INT\_MIN에서 1을 빼면 언더플로우로 INT\_MAX가 된다.

0x80000000 + 0x7fffffff (-1) = 0x7ffff ffff

( 0x0000 0001 = 1 -> 0xFFFF FFFE (1의 보수)-> 0xFFFF FFFF(2의보수) =-1)

**CWE-190의 오버플로우 예시)**

텍스트이(가) 표시된 사진

자동 생성된 설명

table\_ptr = (img\_t\*)malloc(sizeof(img\_t)\*num\_imgs);

malloc은 오버플로우 발생 가능성이 있다.

sizeof(img\_t)\*num\_imgs => img\_t 구조체의 크기는 10kb이고 num\_imgs는 부호가 있는 정수이다.

num\_imgs가 큰 수일 경우, 이 둘을 곱하면 오버플로우가 발생할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

xmalloc => malloc을 개선함(요청된 메모리의 크기를 할당할 수 없으면 에러를 표시하고 프로그램을 종료함)

xmalloc(nresp\*sizeof(char\*))

만약 nresp 가 1,073,741,284이면

1,073,741,284 \* 4 = 4,294,967,296로

UINT\_MAX = 4,294,967,295, 최대범위를 벗어나게 된다.

따라서 오버플로우로 response는 0이 되고,

밑의 for문은 4,294,967,296번 루트를 돌고

response[i] = packet\_get\_string(NULL)은 실제 할당받은 공간은 작은데 계속 할당을 받게 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

while loop에서 bytesRec을 체크하는데

byteRec은 byteRec에 getFromInput(buf + bytesRec)을 더한 값이다.

만약 byteRec이 오버플로우가 일어나 작은 값이 된다면 문제가 발생한다.

int myfunction(int \*array, int len) {

int \*myarray, i;

myarray = malloc(len \* sizeof(int)); /\* [1] \*/

if(myarray == NULL){

return -1;

}

for(i = 0; i < len; i++) { /\* [2] \*/

myarray[i] = array[i];

}

return myarray;

}

[1]경우 len이 큰 경우 오버플로우가 일어나 작은값이 malloc 될 수 있다.

따라서 [2]에서 len은 크지만 myarray가 작으므로 문제가 발생한다.

int catvars(char \*buf1, char \*buf2, unsigned int len1, unsigned int len2) {

char mybuf[256];

if ( (len1 + len2) > 256 ) { /\* [3] \*/

return -1;

}

memcpy(mybuf, buf1, len1); /\* [4] \*/

memcpy(mybuf + len1, buf2, len2);

do\_some\_stuff(mybuf);

return 0;

}

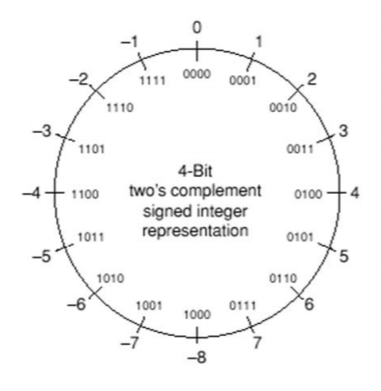
[3]경우 len1 + len2가 값이 커서 오버플로우가 발생하면 if문을 통과하고 밑의 memcpy를 수행한다.

buf1에서 len1크기만큼 mybuf로 복사하므로 버퍼 오버플로우가 발생할 수 있다.

첫번째 memcpy를 통과하더라도 두번째 memcpy에서도 위에서 설명한 것처럼 문제가 발생할 수 있다.

만약 len1이 0x0000 0002이고 len2가 0xffff ffff라면

첫번째 memcpy는 통과하지만 두번째 memcpy에선 len2가 unsigned int로 바뀌면서 큰값이 되어 버퍼 오버플로우가 발생할 수 있다.

텍스트, 장치, 측정기이(가) 표시된 사진

자동 생성된 설명

**\* Integer Overflow를 막는 방법 예시**

(자바 코드)

int sum(int a, int b) {

int c = a + b;

if (a > 0 && b > 0 && c < 0)

throw new MyOverfowException(a, b);

return c;

}

int prod(int a, int b) {

int c = a \* b;

if (a > 0 && b > 0 && c < 0)

throw new MyOverfowException(a, b);

return c;

}

c = a+b 연산을 한 후

a>0, b>0이고 c<0이면 오버플로우가 발생한 경우므로 예외 처리한다.

밑의 곱의 경우도 마찬가지이다.

static void update\_value(char op) {

if (op == '+') {

if (value + 1 > value) value ++;

else printf (“too big! \n”);

} else {

if (value - 1 < value) value --;

else printf(“too small! \n”);

}

}

int addOvf(int\* result, int a, int b) {

if( a > INT\_MAX - b) return -1;

else {

\*result = a + b;

return 0;

}

}

-덧셈 연산을 한 후 오버플로우가 발생하는지 체크한다.

if (value + 1 > value)

     value ++;

-덧셈 연산전 오버플로우가 발생하는지 확인한다.

a > INT\_MAX - b 이면 오버플로우이다.

(max값에서 b를 뺀값보다 a가 크다면 a와 b를 더하면 max값 보다 클것이다.

즉, 현실세계에서 보면  a+b > INT\_MAX 이므로, 합이 MAX값을 넘어서 컴퓨터에선 오버플로우가 발생)

(INT\_MAX는 limit.h에 선언되어 있다.)

if( a > INT\_MAX - b)

     return -1;

다른 방법으로 SW개발 전 단계에서 고려하는 것이다.

-요구사항 분석 : 안전한 언어나 안전한 컴파일러를 사용한다.

-설계 : 안전한 라이브러리나 프레임워크를 사용한다(SafeInt or IntegerLib)

-구현 : 사용하는 범수가 범위내에 있는지 확인한다. 최대값과 최소값 범위를 체크한다.

#include <stdio.h> /\* ex2.c ʹ loss of precision \*/

int main(void) {

int m;

short s;

char c;

m = 0xcafe76ef;

s = m;

c = m;

printf("l = 0x%x (%d bits)\n", m, sizeof(m) \* 8);

printf("s = 0x%x (%d bits)\n", s, sizeof(s) \* 8);

printf("c = 0x%x (%d bits)\n", c, sizeof(c) \* 8);

return 0;

}

changmin@ubuntu:~/Desktop/c$ ./aaa   
l = 0xcafe76ef (32 bits)   
s = 0x76ef (16 bits)   
c = 0xffffffef (8 bits)

s는 16bit=2byte만을 표현할 수 있고, 0xcafe 76ef가 4byte=32bit이므로  상단 16bit가 날라가고 0x76ef가 남는다.

0x7 => 0111(2)이므로 최상단 비트가 1이 아니여서 양수이다. 따라서 앞자리는 모두 0이된다. (%x시 32비트로 출력이된다. => 이부분은 아직 잘 이해하지 못하엿다....)

따라서 0x000076ef

c는 8bit=1byte를 표현할 수 있으므로 0xef만 남게된다.

0xe = 14 = 1110(2)로 최상단 비트가 1이므로 음수이다.

0xe는 음수임을 나타내므로 결과 값의 나머지 앞 부분도 음수 표시위해 1, 즉 모두 f가 된다.

\* 부호버그

부호가 없는 정수를 부호가 있는 정수로 바꾸거나

부호가 있는 정수를 부호가 없는 정수로 바꾸려할 때 발생한다.

예시)

#include <stdio.h>

#include <string.h>

int main(int argc, char \*argv[]) {

char buf[20];

int i = atoi(argv[1]);

//atoi는 아스키를 int로 바꾸어준다.

memcpy (buf, argv[2], i\*sizeof(int));

printf("the number is:%d=%d\n",i, i\*sizeof(int));

printf("the buffer is: %s\n",buf);

}

-----------------------------------------------------------------------------------------

changmin@ubuntu:~/Documents$ ./test3 1 AAAA  
the number is:1=4  
the buffer is: AAAA�vbV

=> 1 AAAA를 입력하면,

i는 1이되고 i\*sizeof(int)는 4가된다.

"�vbV"는 memcpy가 null을 추가하지 않고 복사하기때문에 그런것같다.

-----------------------------------------------------------------------------------------  
changmin@ubuntu:~/Documents$ ./test3 111 AAAA  
the number is:111=444  
the buffer is: AAAA  
\*\*\* stack smashing detected \*\*\*: <unknown> terminated

Aborted (core dumped)

111 AAAA를 입력하면

i는 111이되고 i\*sizeof(int)는 444가된다.

따라서 argv[2]에서 444만큼 가져와 buf 에 copy하려한다.

---------------------------------------------------------------------------------------  
changmin@ubuntu:~/Documents$ ./test3 -1 AAAA  
Segmentation fault (core dumped)

-1 AAAA를 입력하면

i는 -1이되고 i\*sizeof(int)는 -4가된다.

memcpy 원형을 보면

#include <string.h>

void \*memcpy(void \*dest, const void \*src, size\_t num);

인데 size\_t는 unsigned int 타입이다.

따라서 -4가 아니라 언더플로우가 일어나 엄청나게 큰 수로 바뀌게 된다.

따라서 버퍼 오버플로우가 발생한다.

gdb를 사용하여 테스트해보면

(위와 같은 코드이다.)

#include <stdio.h>

#include <string.h>

int main(int argc, char \*argv[]) {

char buf[20];

int i = atoi(argv[1]);

//atoi는 아스키를 int로 바꾸어준다.

memcpy (buf, argv[2], i\*sizeof(int));

printf("the number is:%d=%d\n",i, i\*sizeof(int));

printf("the buffer is: %s\n",buf);

}

(파이썬을 설치하지 않아 A 44번 B 4번을 일일이 입력했다...)

(gdb) r 12 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB  
Starting program: /home/changmin/Documents/test3gdb 12 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB  
the number is:12=48  
the buffer is: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB  
\*\*\* stack smashing detected \*\*\*: <unknown> terminated  
  
Program received signal SIGABRT, Aborted.  
0xf7fd5059 in \_\_kernel\_vsyscall ()

=> argv[0]에 12, argv[1]에 A\*44+B\*4를 입력하였다.

=>buf의 크기는 20인데 memcpy에서 12\*4(i\*sizeof(int))를 하여 48크기만큼 memcpy하려하여 버퍼가 넘치는 버퍼 오버플로우가 발생한다.

이번엔 argv[0]에 -1073741810을 입력하였다.

(gdb) r -1073741810 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB  
The program being debugged has been started already.  
Start it from the beginning? (y or n) y  
Starting program: /home/changmin/Documents/test3gdb -1073741810 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB  
the number is:-1073741810=56  
the buffer is: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB  
\*\*\* stack smashing detected \*\*\*: <unknown> terminated  
  
Program received signal SIGABRT, Aborted.  
0xf7fd5059 in \_\_kernel\_vsyscall ()

=> -1073741810 \* 4 (i\*sizeof(int))값이 memcpy의 인자로 들어가는데 usigned int로 바뀌면서 값이 56으로 변환이 된다. 따라서 버퍼의 크기 20을 넘게된다.

\* CWE-195의 부호변환 에러

텍스트이(가) 표시된 사진

자동 생성된 설명

=> readdata함수는 unsigend int형을 반환한다.

=> 함수 안에서 if문을 보면, 조건 만족시 amount에 -1을 넣고 그 값을 반환한다.

=> usigned int 형으로 바뀌어 반환되므로 언더플로우가 발생한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

=> 이 경우 또한 usigned  int형을 반환하는데 amount가 음수값이 된다면 언더플로우가 발생할 것이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

=> if문에서 numHeaders가 100보다 큰지 체크한다.

=> 하지만 int형이 numHeaders가 INT\_MAX보다 큰값이 된다면 오버플로우로 음수가 될 것이다.

=> 따라서 ExitError를 실행하지 않고 malloc을 한다.

=> 하지만 malloc에서 unsigned int형으로 바뀌면서 언더 플로우가 발생된다.

- Software Security가 중요한 이유?

Software는 종류가 다양하고,

----------------------------------------------------------------------------------

λ Systems software : OS, compiler, loader

λ Business software : Payroll, accounting

λ Scientific and engineering software

ν Computer-aided design, simulation, weather prediction, …

λ Internet software: ν B2C: business-to-customer (e.g., amazon.com)

ν Facebook, Google Chrome, …

λ PC software : Spreadsheets, word processing, games, …

λ Embedded software

ν Cars, microwave ovens, cable boxes, light switches,

ν “smart dust”, …

λ Mobile applications

----------------------------------------------------------------------------------

컴퓨터 보안의 많은 것들은  SW로 구현이 된다.

알고리즘, access control 등 대부분이 SW로 구현되기 때문에 Software Security가 중요하다.

따라서 구현할 때 쓰는 sw가 취약하면 의미가 없으므로, SW가 취약하면 보안은 무조건 깨진다.

만약, 강력한 암호 알고리즘을 도입한다해도 구현하는 SW가 문제 있다면 그 보안은 의미가 없을 것이다.

따라서 SW는 빈약한 보안의 기초가 될 수 있다.

(SW가 빈약하면 보안 자체의 기초가 흔들린다.)

\* Software 위기의 원인

- 예산 초과

- 시간 초과

- 비효율적인 SW

- 저품질의 SW

- 때때로 요구사항을 만족 못함

- 관리불가(유지보수가 힘듬 (코드가 복잡하므로))

- 원하는 결과를 내놓지 않음 or 소스코드를 넘겨주지 않음(문제 확인이 어려움)

이러한 문제를 해결하기 위해 Software Engineering이 나오게 되었다.

\* Software

Software는 실행가능한 프로그램이고, 소스코드, 라이브러리, 문서(유저 요구사항, 실제 명세서, 가이드/메뉴얼 등)이다.

그 중 핵심 기능은

데이터를 처리, 전송, 저장하는것이고

정보를 생산, 관리, 드러내는 것이다.

\* Bugs, Defects, Weaknesses, and Vulnerabilities

----------------------------------------------------------------------------------

Improper initialization

Side effects

Scoping

Operator precedence

Divide-by Zero

Infinite loop

Type confusion (illegal downcasts)

Deadlock

Integer Overflow / Underflow

Memory leak

Use-after-free

Buffer overflow = Buffer overrun

Time-of-check-to-time-of-use flaw

Format string bug

----------------------------------------------------------------------------------

위와 같이 다양한 종류가 있다.

취약점 보단 덜 위험하지만 문제를 일으킬 수 있다.

- bug와 취약점의 차이

bug : 의도치 않은 방향으로 잘못 행동하는 프로그램 내의 결함이다.

취약점 : bug 중에서도 공격자에 의해서 악용될 수 있는 것, 공격자가 접근할 수 있어야하고 공격자가 공격할 능력이 있어야하고 시스템 내에서 결함이 있어야한다.

**SW Bugs 예시)**

**-------------------------------------------------------------**

**(초기값)**

typedef unsigned int uint;

int getmin(int \*arr, uint len){

int min;

for(int i=0;i<len;i++)

min = (min < arr[i]) ? min : arr[i];

return min;

}

min에 대한 초기값이 설정되어 있지않아 정상적으로 실행되지 않을 수 있다.

inn min = 0과 같이 초기 값을 설정해주면 된다.

**(side effects)**

if(foo == 12 || (bar = 13))

baz == 12;

bar = 13은 항상 참이다.

foo와 baz의 값이 선언되어있지 않고 비교되고 있다.

**(범위)**

int a;

void calc(int b){

int a = b\*12;

if(b+24 == 96)

a = b;

}

printf("a=%d\n", a);

=> 전역변수, 지역변수 둘다  a로 선언하여 모호하다.

=> calc함수내의 a는 함수가 종료되면 없어지므로 전역변수 a의 값이 출력된다.

**(control flow)**

int x,y;

for(x=0;x<xlen;x++)

for(y=0;y<ylen;y++);

pix[y\*xlen + x] = x\*y;

두번째 for문에 ;처리하여 원하는 결과가 출력되지 않는다.

if (isbad(cert))

goto fail;

if (invalid(cert))

goto fail;

goto fail;

L10 : printf("Hello, world\n");

goto L10

goto fail;

goto fail;

을 하면 if문을 통과하고 두번째 goto fail이 항상 실행된다.

goto를 잘 못 사용하면 무한루프(L10으로 계속 이동)에 걸릴 수 있다.

**(control flow - loop)**

float x = 0.1;

while(x!=1.1){

x=x+0.1;

printf("X=%f\n", x);

}

부동 소수점은 ==으로 비교하면 안된다.

실수는 무한히 많은데 이 실수를 유한 개의 비트로 표현하기 위해서는 근삿값으로 표현해야 하기 때문이다. =>부동소수점 반올림 오차

#include <stdio.h>

#include <float.h> // float의 머신 엡실론 값 FLT\_EPSILON이 정의된 헤더 파일

#include <math.h> // float의 절댓값을 구하는 fabsf 함수를 위한 헤더 파일

int main()

{

float num1 = 0.0f;

float num2 = 0.1f;

// 0.1을 10번 더함

for (int i = 0; i < 10; i++)

{

num1 = num1 + num2;

}

// num1: 1.000000119209290

if (fabsf(num1 - 1.0f) <= FLT\_EPSILON) // 연산한 값과 비교할 값의 차이를 구하고 절댓값으로

// 만든 뒤 FLT\_EPSILON보다 작거나 같은지 판단

// 오차가 머신 엡실론 이하라면 같은 값으로 봄

printf("true\n"); // 값의 차이가 머신 엡실론보다 작거나 같으므로 true

else

printf("false\n");

return 0;

}

FLT\_EPSILON(머신 엡실론)을 사용해서 오차를 감안하여 실수를 비교해야한다.

어떤 실수를 가장 가까운 부동소수점 실수로 반올림하였을 때 상대 오차는 항상 머신 엡실론 이하이다.

머신 엡실론은 반올림 오차의 상한값이며 연산한 값과 비교할 값의 차이가 머신 엡실론보다 작거나 같다면 두 실수는 같은 값이라 할 수 있다.

double, long double을 사용한다면 머신 엡실론은 DBL\_EPSILON, LDBL\_EPSILON을 사용한다.

1. 연산한 값과 비교할 값의 차이를 구한다.

(값의 차이는 math.h 헤더 파일의 fabsf 함수를 사용하여 절댓값으로 만드므로 차이가 음수여도 상관없다.)

2. FLT\_EPSILON 보다 작거나 같은지 판단

3.연산한 값과 비교할 값의 차이가 머신 엡실론보다 작거나 같다면 두 실수는 같은 값이라 판단

**(control flow - loop)**

int k = 1

int val = 0;

while (k = 10) {

val++;

k++;

}

printf (“k = %d, val = %d \n”, k, val);

while(k=10)

k=10은 항상 참이므로 무한 루프가 된다.

**(Null pointer 역참조)**

int \*ptr = NULL;

printf(“Value of ptr: %d\n”, ptr);

int \*p = 0; //NULL 대입

\*p = 1; //NULL 포안터 역참조, 주소 0에 접근하는 포인터생성후 값 할당 시도함

/\* -------------------------------------\*/

int length;

char \*buff;

scanf (“%d”, &length);

buff = (char \*) malloc(length+1); // always Not NULL?

strcpy(buff, “Hello World! Welcome!”);

첫번째 코드는 포인터 변수 P에 NULL을 대입하고 주소가 0인 포인터에 값을 할당하려하였다.

따라서 세그멘테이션 결함이 발생한다.

if(p==NULL){

     exit(1);

}

과 같이 NULL 포인터를 체크해주어야한다.

두 번째 코드는

malloc시 시스템에 메모리가 부족하거나 메모리 할당 조건이 맞지않아 메모리 할당을 하지 못하면 null을 반환하게 된다. 따라서 buff에 null 포인터가 들어가게된다. 따라서 malloc이 null을 반환하였는지 체크하여야한다.

void Pointer(int \*ptr) {

\*ptr = \*ptr + 5;

}

main(void) {

int num = 10;

Pointer(&num);

Pointer(NULL);

}

Pointer(NULL)

null을 인자로 넘겨주면 Pointer함수에서 null에 값을 대입하려하여 null 포인터 역참조 에러가 발생한다.

**(연산자 우선순위)**

node \*find(node \*\*curr, val){

while(\*curr != NULL)

if(\*curr->val == val) return \*curr;

else

\*curr = \*curr->next;

}

(\*curr)->val로 하여야한다.

#include<stdio.h>

void main(){

int x,a,b,c,d,e,f;

a=7;b=6;c=5;d=4;e=3;f=2;

x = a&b+c\*d&&e^f==7;

printf("X = %d\n", x);

}

x = 1

연산자 우선순위는

(\*) -> (+) -> (==) -> (&) -> (^) -> (&&)

이다.

따라서

(a&(b+(c\*d)))&&(e^(f==7))가 된다.

(a&(b+(c\*d))) => 7&(6+(5\*4)) =  20&7 (10100 & 00111) => 100(2) = 4

(e^(f==7)) => 3^(2==7) => 11 xor 00 = 11(2) => 3

3&&4는 참이므로 1이된다 (3과 4는 참이므로)

따라서 x에는 1이 들어가게된다.

**(Integer Security)**

char cresult, c1, c2, c3;

c1 = 100; c2 = 90; c3= -120

cresult = c1 + c2 + c3;

printf(“%c, %d, %c, %d\n”, cresult, cresult, c3, c3);

changmin@ubuntu:~/Desktop/c$ ./d  
F, 70, �, -120

char형은 8비트이다.

c1+c2=190에서 char범위를 벗어난다.

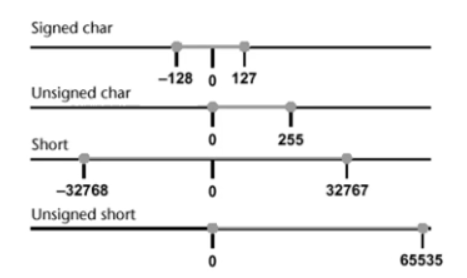
임시적인 정수 승격으로 c1, c2, c3는 int형으로 변환되고

(임시적인 정수승격 : 일반적으로 CPU가 처리하기에 가장 적합한 크기의 정수 자료형은 int형이다. 즉, int형 연산의 속도가 다른 자료형의 연산속도에 비해서 동일하거나 더 빠르다. 따라서, int보다 작은 크기의 정수형 데이터는 int형 데이터로 형 변환이 되어서 연산된다.)

계산이 끝난뒤에 값이 잘리게 된다.

따라서 c1+c2+c3는 70이된다.

char 범위는 -128~127이므로 70이 출력된다.



short s1 = 32000;

short s2 =1500;

s1 = s1 + s2;

printf("%h, %d\n", s1, s1);

changmin@ubuntu:~/Desktop/c$ ./e  
%, -32036

(%h =>short 형으로 출력,  invalid conversion specifier -Wformat-invalid-specifier 라고 되는데 %h 형식을 지원하지 않는듯 하다..... short형은 %hd를 사용해야하는 것 같다.)

32000+1500은 short범위를 벗어나므로 오버플로우가 발생한다.

**(out-of-bounds write)**

텍스트이(가) 표시된 사진

자동 생성된 설명

배열의 크기를 벗어나거나

텍스트이(가) 표시된 사진

자동 생성된 설명

memcpy시 마지막 인자가 unsigned int형으로 변환되면서 언더플로우가 발생될 수 있다.

(returnChunckSize(desBuf)의 반환값이 -1일때 -1-1은 -2가 되고 unsigned int로 변환시 언더플로우발생으로 매우 큰수가 된다.)

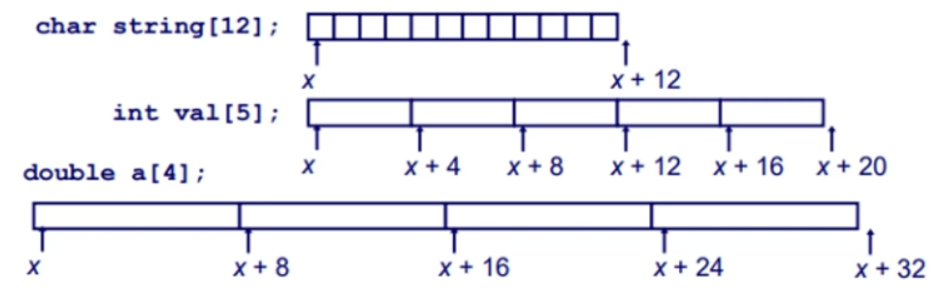
- Integral

|  |  |  |  |
| --- | --- | --- | --- |
| intel | GNU assembler | bytes | c |
| byte | b | 1 | [unsigned] char |
| word | w | 2 | [unsigned] short |
| double word | l | 4 | [unsigned] int |
| quad word | q | 8 (컴퓨터마다 다름 4 or 8) | [unsigned] long int (x86-64) |

- Floating point

|  |  |  |  |
| --- | --- | --- | --- |
| intel | GNU assembler | bytes | c |
| single | s | 4 | float |
| double | l | 8 | double |
| extended | t | 10/12/16(system, os, complier 마다 다름) | long double |

배열의 할당

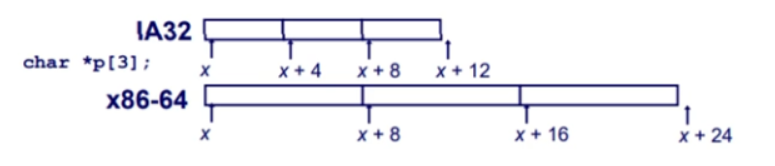


char 배열의 주소 크기는 1 byte

int 배열의 주소 크기는 4byte (한 원소당 4byte를 차지하므로)

double 배열의 주소 크기는 8byte

-포인터 배열



32bit 구조는 address bus가 4byte이므로 주소도 4byte

64bit 구조는 address bus가 8byte가 가능하다. 하지만 8byte는 큰 공간이므로 메모리 낭비라고 생각하여 6byte씩 표현하는 경우도 있다.

## 

## 

## Array operations

예시

int val[5];

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 5 | 2 | 1 | 3 |

주소 : x -> x+4 ->x+8 ->x+12->x+16

val[4] = 3

val = x

val+1 =x+4

&val[2] = x+8

val[5] = ?

\*(val+1) = 5

val +i = x+4i

예시1)

#include <stdio.h>

typedef int zip\_dig[5];

int main() {

zip\_dig cmu = {1, 5, 2, 1, 3};

zip\_dig mit = {0, 2, 1, 3, 9};

zip\_dig ucb = {9, 4, 7, 2, 0};

printf("%p \n", cmu);

printf("%p \n", mit);

printf("%p \n", ucb);

/\*-----------------------------------\*/

printf(“%d, ”, cmu[8]);

printf(“%d, ”, cmu[11]);

printf(“%d, ”, ucb[-5]);

printf(“%d\n”, ucb[-15]);

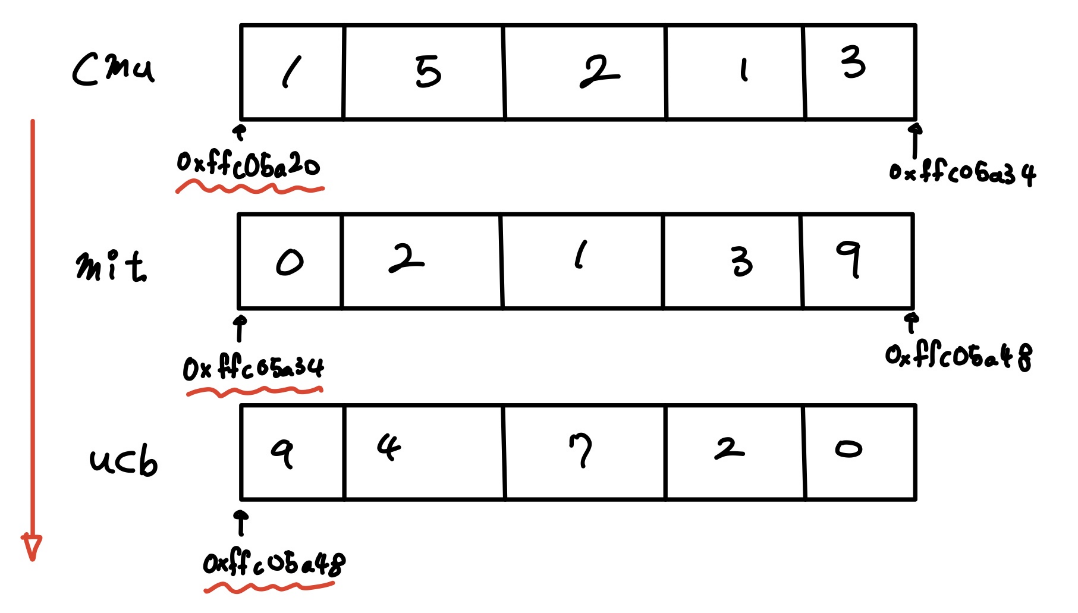
}

**(32bit로 컴파일)**

텍스트이(가) 표시된 사진

자동 생성된 설명

stack protector가 설정되어있으므로 낮은 주소부터 할당이 된다.



연속으로 배열이 할당이되고 낮은 주소부터 할당이 되는 것을 알 수 있다.

0x14 =>20byte 만큼 주소가 차이난다.

텍스트이(가) 표시된 사진

자동 생성된 설명

-fstack-protector을 하면 64->78->8c로 스택이 자라고

-fno-stack-protector을 하면 bc -> a8 -> 94로 낮은주소로 스택이 자란다.

**(64bit로 컴파일)**

텍스트이(가) 표시된 사진

자동 생성된 설명

주소체계도 다르고, 0x20만큼 주소가 차이나는 것을 알 수 있다. 32byte만큼 주소차이가 난다.

테이블이(가) 표시된 사진

자동 생성된 설명

(오타, 0x7ffe1eadaa40이 아니라50이다....)

예시 2)

#include <stdio.h>

typedef int zip\_dig[5];

int main() {

zip\_dig cmu = {1, 5, 2, 1, 3};

zip\_dig mit = {0, 2, 1, 3, 9};

zip\_dig ucb = {9, 4, 7, 2, 0};

printf("%p \n", cmu);

printf("%p \n", mit);

printf("%p \n", ucb);

printf("%d, %d, %d, %d\n",mit[-5], mit[-4], mit[8], mit[9]);

}

**(stack proctector)**

텍스트이(가) 표시된 사진

자동 생성된 설명

=>

0x20만큼 주소가 차이남 (5개의 원소)

**(no stack protector)**

텍스트이(가) 표시된 사진

자동 생성된 설명

=>

-fno-stack-protector(스택 보호기법 해제) -> 높은위치에서 낮은 위치로 성장

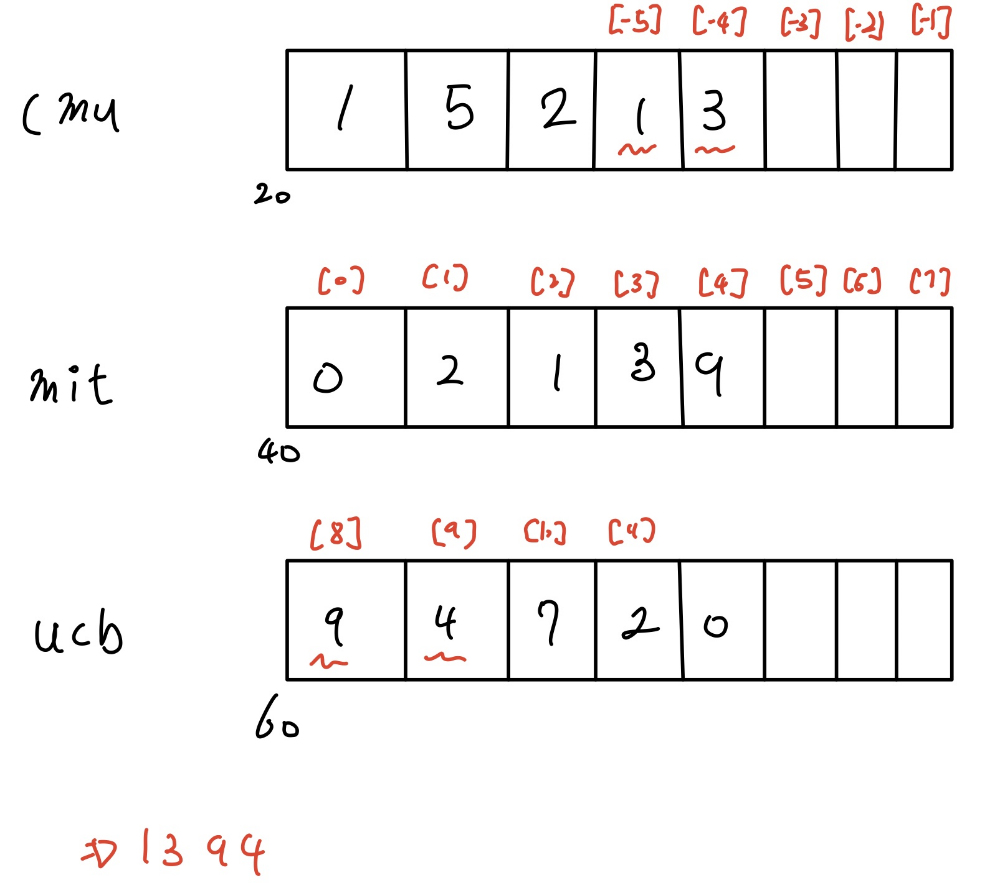
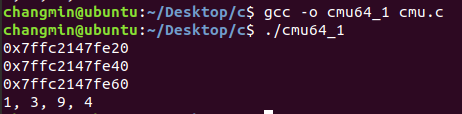
5c -> 48 ->34

테이블이(가) 표시된 사진

자동 생성된 설명

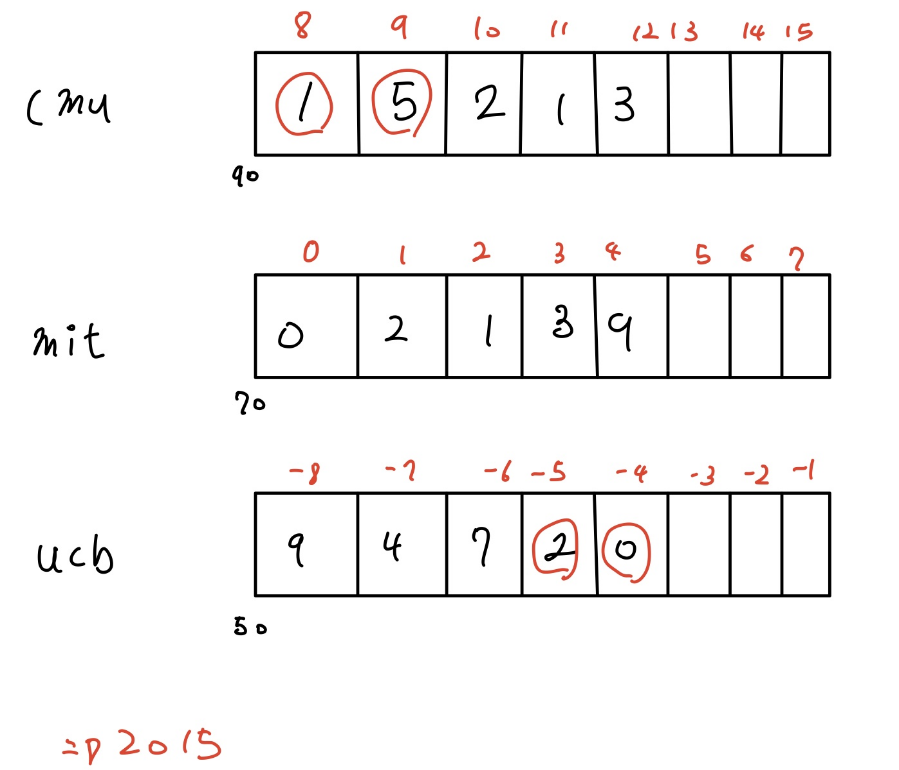
**주소체계와 접근하는 방식이 달라짐!**

**(64bit의 경우)**



**(64bit no stack protector)**

텍스트이(가) 표시된 사진

자동 생성된 설명

예시 3)

#include <stdio.h>

typedef int zip\_dig[5];

int main() {

zip\_dig cmu = {1, 5, 2, 1, 3};

zip\_dig mit = {0, 2, 1, 3, 9};

zip\_dig ucb = {9, 4, 7, 2, 0};

printf("%p \n", cmu);

printf("%p \n", mit);

printf("%p \n", ucb);

printf("cmu[8] : %d \n", cmu[8]);

printf("cmu[12] : %d \n", cmu[12]);

printf("mit[8] : %d \n", mit[8]);

printf("mit[-4] : %d \n", mit[-4]);

printf("ucb[-4] : %d \n", ucb[-4]);

printf("ucb[-12] : %d \n", ucb[-12]);

printf("\n");

printf(" \*(cmu+9) : %d \n", \*(cmu+9));

printf(" \*(cmu+11) : %d \n", \*(cmu+11));

printf(" \*(mit-4) : %d \n", \*(mit-4));

printf(" \*(ucb-5) : %d \n", \*(ucb-5));

}

텍스트이(가) 표시된 사진

자동 생성된 설명

예시 4)

#include <stdio.h>

void main() {

int k = 100;

int a[4] = {0, 1, 2, 3};

int b[4] = {4, 5, 6, 7};

int c[4] = {8, 9, 10, 11};

a[-1] = 0xAA;

a[7] = 0xBB, a[10] = 0xCC;

c[-2] = 0xDD, c[-6] = 0xEE;

printf("k = %d (0x%x)\n", k, k);

for(int i=0; i<12; i++) {

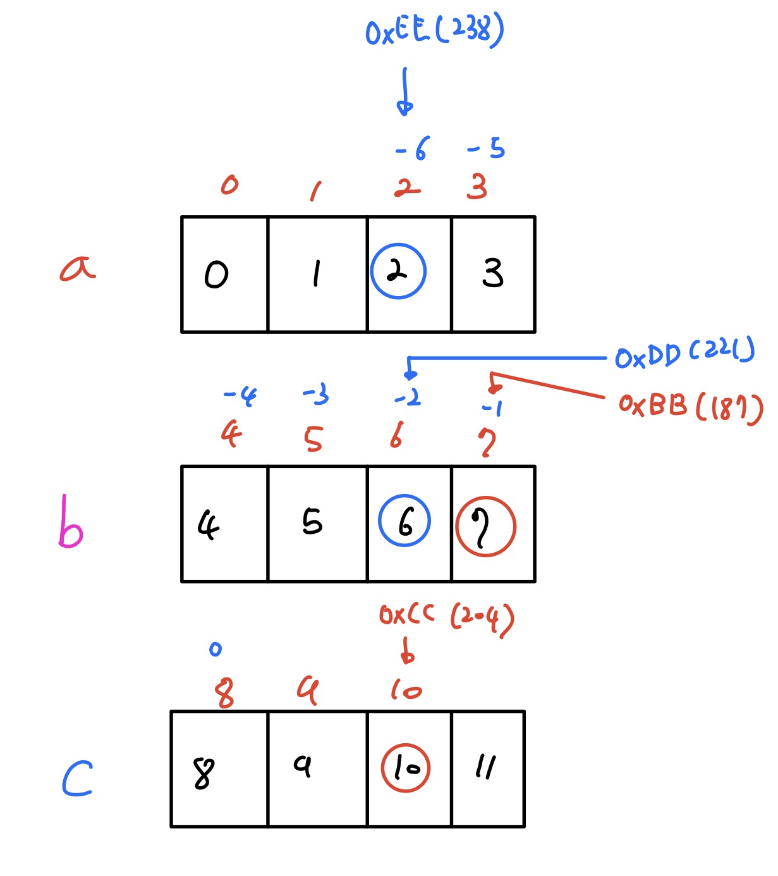
printf ("%3d (0x%x) ", a[i], a[i]);

if ((i+1) % 4 == 0) printf ("\n");

}

}

텍스트, 표지판이(가) 표시된 사진

자동 생성된 설명

=> 인덱스에 음수가 가능

=> 인덱스가 본래 의미한 것보다 더 큰 값도 가능(다른 위치도 배열의 인덱스로 접근가능)

따라서 c는 메모리 관련 취약점이 많음

**(64bit)**

텍스트, 표지판이(가) 표시된 사진

자동 생성된 설명

**(64bit stack보호 기법해제)**

텍스트이(가) 표시된 사진

자동 생성된 설명

## 

## Byte ordering

=> 취약점을 분석할때 컴퓨터 구조에 맞는 데이터를 투입해야하므로 알아야함



little endian은 최소 유효 byte가 낮은 주소를 차지한다.

big endian은 최소 유효 byte가 높은 주소를 차지한다.

type별 크기

테이블이(가) 표시된 사진

자동 생성된 설명

=> big/little endian에 따라 c[0]이 낮은주소/높은주소 일 수 있다.

테이블이(가) 표시된 사진

자동 생성된 설명

Revere Engineering(Backwards, back engineering)

Forward Engineering

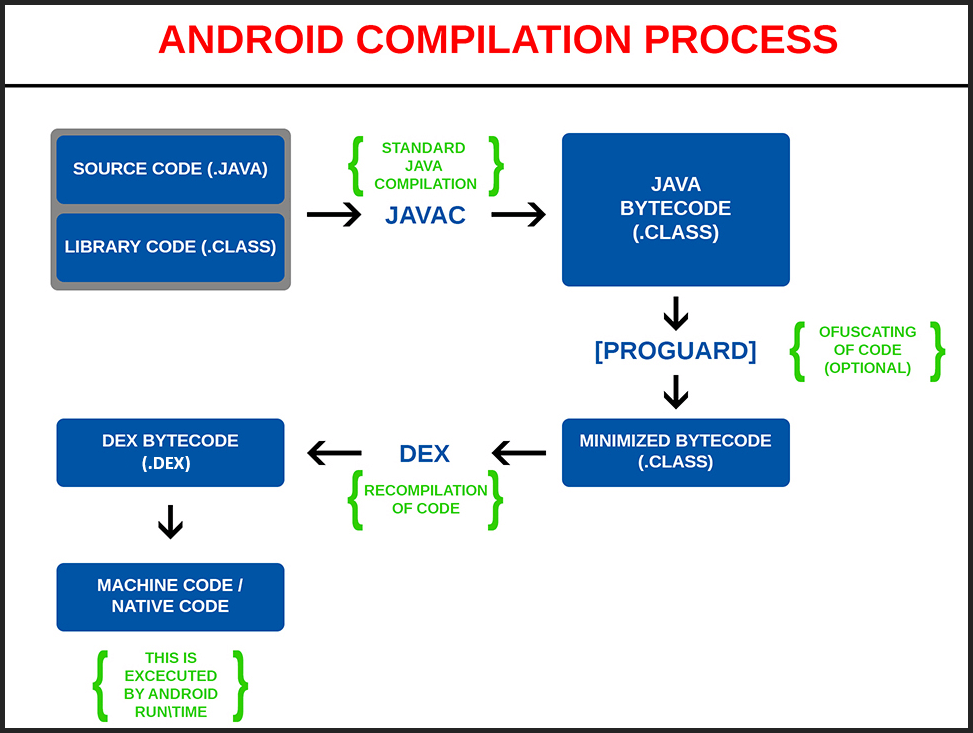
요구사항분석 -> 설계 -> 구현

요구사항분석 <(요구사항 복구)- 설계 <(설계 복구)- 구현

역공학은 이를 역으로 복구하는 것을 reverse engineering이라 함

* 오래된 프로그램을 유지보수해야할 때, 분석하여 설계원칙과 요구사항을 알아내야하므로 역공학을 사용하기도 함

예시) 안드로이드 컴파일 과정은 Forward Engineering의 일부임



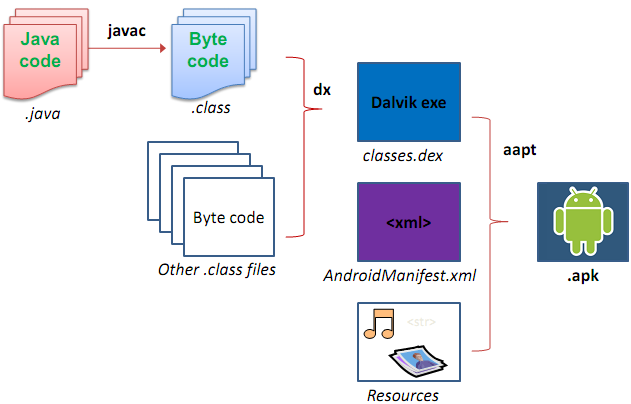
소스코드를 자바로 작성(.java) ->컴파일 -> java Bytecode ->Proguard(최적화, 난독화) -> dex(davik executable)로 변환 -> android runtime에 의해서 기계어로 번역되어 자동

.java ->java Bytecode (.class) -> minimized Bytecode -> Dex Bytecode (.dex) -> Machine code

텍스트, 벽, 모니터, 탑재이(가) 표시된 사진

자동 생성된 설명

최신 안드로이드에선 Android Runtime 사용(native code 로 동작, 성능상의 이유), kotlin도 사용



안드로이드 역공학

apk = android application package (dex, resource 등 파일이 들어가 있음)

de-complier를 이용하여 dex는 smali (중간 어셈블리 언어)로 바뀌어짐,

asset = resource관련

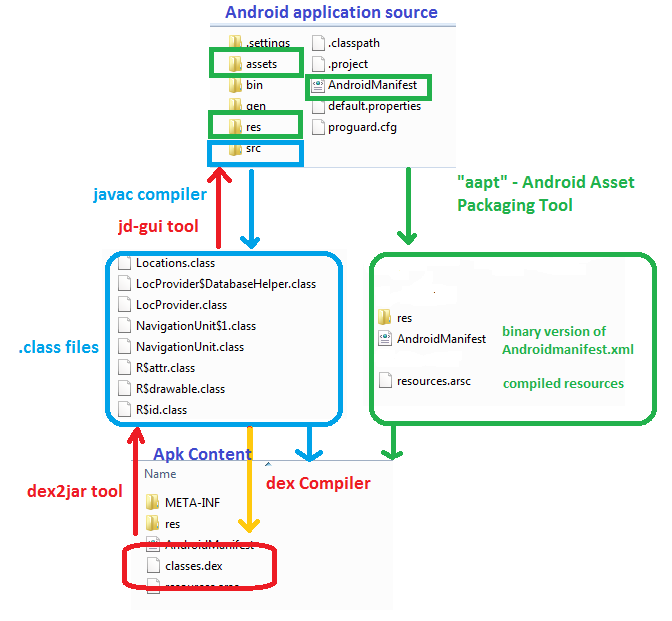
META-INFO = sign관련

텍스트, 벽, 스크린샷, 화면이(가) 표시된 사진

자동 생성된 설명

smali to java를 이용하여 smali코드를 java코드로 바꾸어줌

따라서 java 소스코드를 볼 수 있다.



역공학은 주로 어떤 정보가 이용가능 하지 않을 때(오래전에 만들었거나 소스코드가 공개되지 않을때), 실행코드로부터 놓친 지식이나, 아이디어, 설계 원리를 파악하기 위해 수행한다.

시스템의 일부로 어떤 기능, 어떤 설계원리가 있는지 분석하는 과정이라 할 수 있다.

악성코드 분석가들은 역공학이 필요 => 악성코드는 소스가 공개되어 있지 않기 때문

Software Reverse Engineering

sw뿐만아니라 hw에도 적용될 수 있음

* 프로그램을 분해해서 그 내부 구조를 파악 하는 것.
* 기술과 컴퓨터, sw개발에 대한 이해가 필요함
* code breaking, puzzle solving, programming, logival analysis 필요

Binary reverse engineering

* 소스코드를 사용할 수 없을 때 사용하는 기법
* reverse code engineering

역공학이 필요한 경우

* 연구목적
* 취약점을 분석하기 위해
* 이미 배포되어 사용되는 프로그램들에 패치를 해야 할 때(binary level에서 패치할때 필요함, 보안과 관련잇음)
* 프로그램 내의 비밀 정보를 획득하기 위해(공격자 관점)
* 프로토콜을 분석하기 위해 (공격을 위한 정보 탐색)
* 보호기법을 우회하기 위해 (copy 목적)

역공학 응용

1. 보안에 관련해서 사용

* 악의적인 프로그램의 binary auditing
* 암호 알고리즘 역공학 => 연구자들이 암호화 제품을 평가해야함(얼만큼 안전한가).
* Digital Right Management (DRM) - 디지털 컨텐츠 권리 관리, 저작권 관리

(비용을 지불한 만큼만 사용할 수 있게)

(cracker들을 copy protection 기법을 우회하려 함)

1.1 악성 프로그램

* 멀웨어 작성자, 분석가 둘다 역공학함
* 멀웨어 작성자 => os나 다른 sw의 취약점을 파악하여, 악성코드를 감염(전파)시키려함.

(바이러스는 양성 프로그램에 기성하여 돌아감, 매크로 프로그램을 조작시켜 자신을 전파 시켜야함)

(악성코드는 자기자신을 전파시키기 위해 os나 다른 sw의 취약점을 이용함 ex)Morris Internet Worm)

* anti-virus 개발자

악성코드를 분해하여 분석하기 위해 => 치료가능

1.2 Auditing Program Binaries

소스코드를 사용할 수 없는 경우 & 중요한 프로그램일 경우

중요한 프로그램의 취약점을 찾을 때

ex) bug bounty, 모의 해킹

1.3 DRM

불법 복제 방지위해 copy protection 기술을 사용함.

DRM도 그와 유사 (DRM은 적용되는 컨텐츠가 주로 디지털 미디어 컨텐츠임)

공격자들은 DRM기술을 우회하기 위해 역공학을 사용함.

2. sw 개발에서

* sw 품질 향상, 평가하기 위해
* 유지보수를 위해

Tool or Methods

* disassembler, decompiler => 정적 분석 (실행할 필요 X)

(IDA Pro, JADAX(Dex to Java Decomplier), dex2jar, apktool)

(ILDasm (disassembler for MS Intermediate language, MSIL)

* Debugger ( 동적 코드 분석 )

(JEB)

* Hex Editor

binary(실행 파일 분석)

* PE Analyzer

MS windows는 PE포맷을 사용 (안드로이드는 Dex 사용)

int copy\_something(char \*buf, int len){

char kbuf[800];

if(len > sizeof(kbuf)){ /\* [1] \*/

return -1;

}

return memcpy (kbuf, buf, len); /\* [2] \*/

}

=> if문에서 len의 값이 음수라면 return의 memcpy에서 언더플로우가 발생된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

=> short형으로 선언된 len이 음수라면 if문을 만족하여 memcpy시 언더플로우를 일으키게된다.

=>(len이 SHORT\_MAX보다 크면 오버플로우로 음수가 될 것이다.)

int table[800];

int insert\_in\_table (int val, int pos) {

if ( pos > sizeof(table) / sizeof(int) ) {

return -1;

}

table[pos] = val;

return 0;

}

 일이 벌어

만약 pos값이 0x8000 0000(0x8 = 1000(2))이거나 0x9000 0000(1001(2))이면

최상위 비트가 1이므로 음수가 된다.

따라서 table[음수값] =val; 이 되어버린다.

=> 위와 같은 일이 벌어지면 무결성이 깨지게 된다.

cwe 119

메모리 버퍼의 경계가 있는데 경계와 관련된 연산에 대한 제한, 제약을 적절히 처리하지 못할때 생길 수 있는 약점

어떤 소프트웨어는 메모리 버퍼와 관련된 연산을 할 수 있다.

하지만 그 연산은 버퍼의 의도된 경계 바깥쪽의 영역을 read/wrtie할 수 있음

연산의 제약조건은 버퍼의 의도된 경계내에서 연산해야

버퍼의 경계내의 메모리를 읽고 쓰기 해야됨

어떤 언어는 직접적으로 메모리 위치를 참조하는데 그 위치가 실제로 유효한지 아닌지 확인 하지 않음

프로그램이 참조할 수 있는 유효한 주소인지 체크하지 않아 다른 변수나 자료구조 내부 프로그램 메모리 영역을 읽거나 쓸 수 있음 = >연산 제약을 제대로 처리 하지 않았기 때문에

공격으로 연결되면 임의의 코드를 실행할 수 있고 임의의 제어흐름으로 변조, 민감한 정보 읽기, 시스템이 망가짐

cwe 119 = > 버퍼 오버플로우라고도 함

버퍼 오버플로우는 사람마다 다른 의미로 사용가능

어떤 도구는 버퍼의끝을 넘어서서 write하는것

다른 개발자는 버퍼의 경계밖(버퍼의 시작점, 끝 밖의 공간)에서 읽거나 쓰는것.

또 어떤 사람들은 버퍼의 끝 이후에 어떠한 행동을 하는 것(읽기, 쓰기 가능)

사람마다 다르기때문에 혼란스러운 용어임

따라서 메모리 경계에서 부적절한 연산 제한이 올바른 표현이다.'

이러한 약점의 결과는 CIA가 깨짐

인가되지 않은 코드,명령실행, 메모리 변조 가능

한바이트만 조작할 수 있음

가용성, 기밀성

어떤 메모리를 읽을수 잇고 시스템이 비정상적으로 종료될수 잇음 (dos) cpu같은 자원을 소모(메모리도 가능)

메모리 경계를 넘어서면 메모리의 붕괴를 가져와 시스템이 붕괴됨

해당 프로그램이 공격을 받아서 무한루프상태로 갈수도 있다.

기밀성

read memory

주어진 자료구조 범위를 넘어서서 읽으면 민감한 정보를 읽을 수도 잇다.

메모리의 현재 버퍼의 위치와 같은 정보

어떻게 완화??

소프트웨어 보안은 개발생명 전 주기에서 고려

각 단계에서 취약점을 완화할 숭 ㅣㅆ다

요구사항분석 => 언어선택을 잘 해야. cwe119에 강한 언어를 사용하며 ㄴ좋다, c보단 자바와 같은 언어(자체적으로 메모리 관리를 함) ada나 c#은 오버플로우 보호기법을 적용하고 잇음

그렇지만 이러한 언어들은 binary가 아님 실제 실행시 native코드로 변환 되어야함. 따라서 native code와 상호작용하는 인터페이스는 오버플로우 약점에 취약(자바 가상머신은 c로 만들어져잇음)

언어를 윗단에서 좋은 언어를 사용하더라도 완전하게 해결하는 것은 아님

아키텍쳐및 설계

라이브러리나 프레임워크를 잘 사용

검증이된 라이브러리나 프레임 워크 사용 => safe c String 라이브러리, strsafe.h

string 관련 오버플로우 문제를 해결 => 완벽X

많은 오버플로우는 string고 ㅏ연관 없음, 하지만 string관련 문제를 해결가능

cwe119취약점을 방어할 숭 ㅣㅆ다.

빌드 컴파일 단계

컴파일 할때 최신 컴파일러사용, 컴파일 옵션을 버퍼 오버플로우를 탐지할 수 잇게 사용

MS visual studio 에선 GS flag, Fedora/red hat에선 fortify source gcc flag사용

공격이나 취약점을 조기에 탐지할 숭 ㅣㅆ음

하지만 완변하지 ㄴ않음

구현단계

자신이 작성중인 버퍼가 원하는 대로 작동하는지 두번이상 확인

strcpy -> strncpy

gets->fgets

memcpy보다 안전한 것 사용

안전한 라이브러리 함수 사용

운영단계

운영 환경을 좀더 안전하게 만들어라

관리자는 aslr , position-independent executable (pie)와 같은 기능을 사용해서 특정한 주소를 공격자가 예측하지 못하게 해라

공격자는 버퍼의 주소를 예측하기 어려움

NX비트 활용 (hw기법)

임의의 코드가 스택이나 데이터 세그먼트에 들어갈 확률이 높은데,  data execution protection(nx) 를 하면 코드 세그먼트가 아닌 스택이나 데이터 세그먼트 코드는 실행하기 못하게함.기본적으로 프로세스의 메모리 레이아웃에서는 코드, 텍스트 세그먼트의 코드만 실행해야함. 공격자들은 자기가 실행하고 싶은 코드를 스택이나 ㄴ=데이터 세그먼트에 넣음,

기본적으로 코드(텍스트) 세그먼트는 read only 여서 코드 주입 불가