

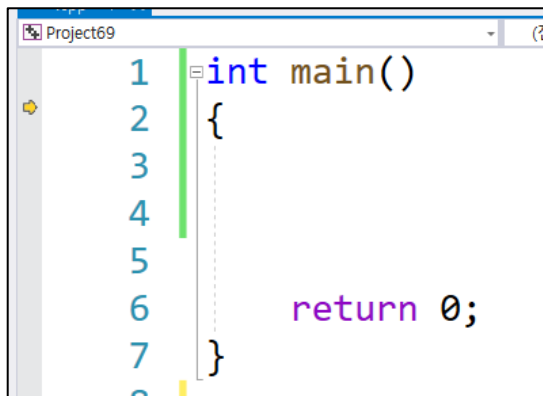
# Assembly (x86) Basic

Bufferoverflow 공격의 이해를 위한 준비과정

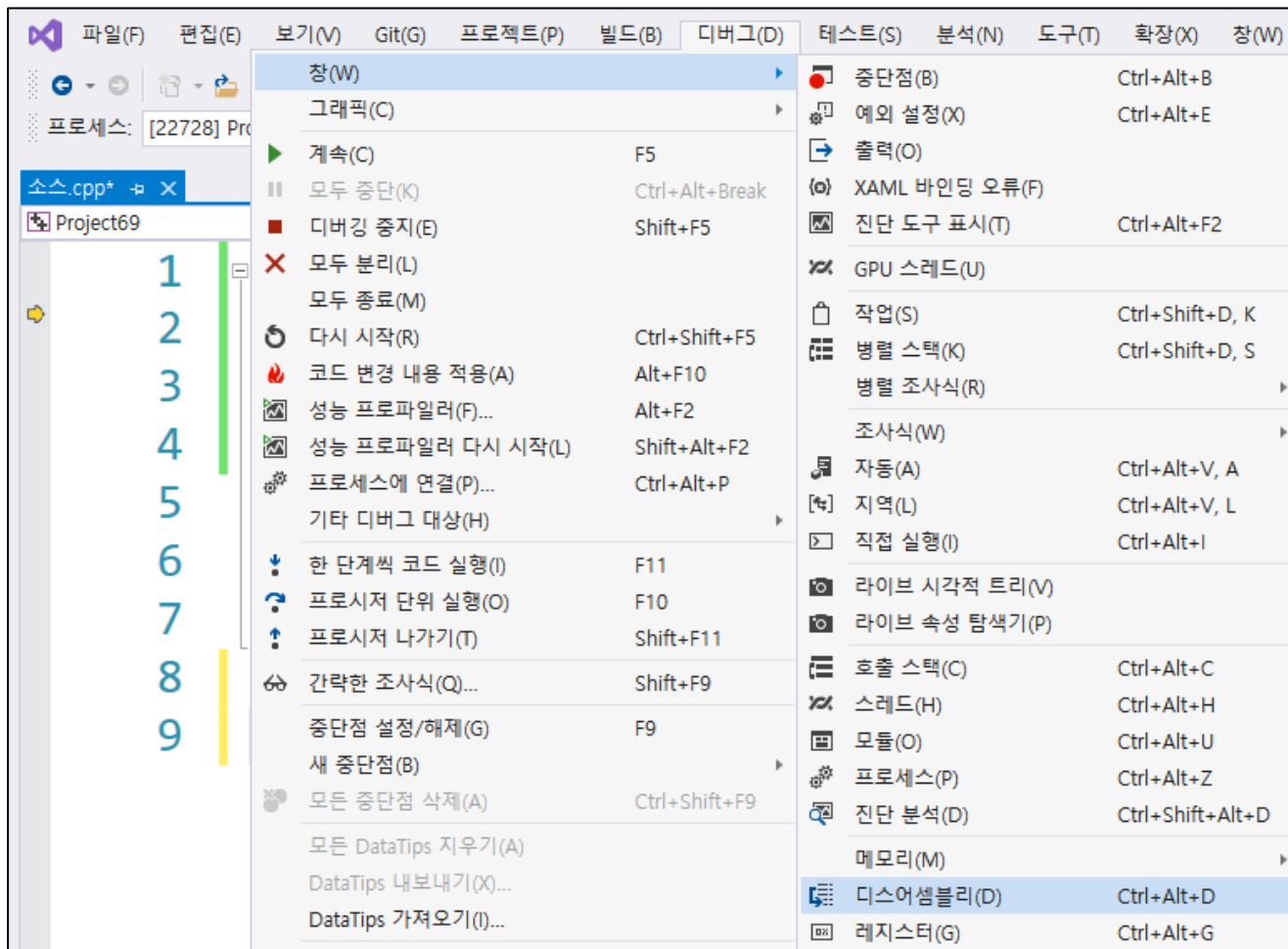
# 개발 환경 창 띄우기

Trace 후,  
메뉴에서 다음 창을 켜다.

1. 디어셈블리
2. 레지스터
3. 메모리

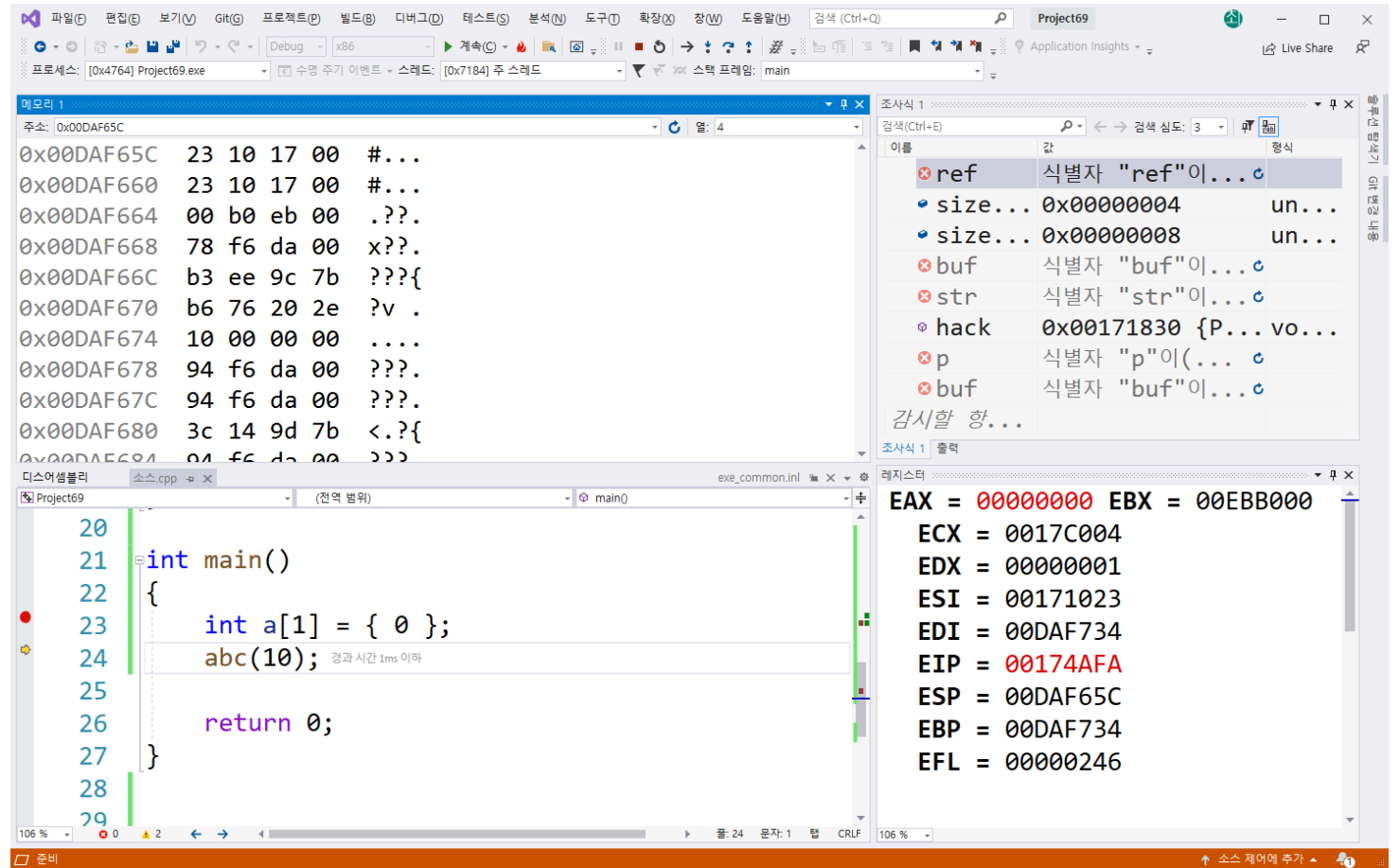


```
1 int main()
2 {
3
4
5
6 return 0;
7 }
```



# 창 배치

세 창을  
다음과 같이 배치한다.



# Trace 연습

## 기본 레지스터

- EAX
- EBX
- ECX
- EDX

```
int main()
{
    _asm {
        mov EAX, 1
        mov EAX, 2
        mov EAX, 3

        mov EBX, 0xA
        mov EBX, 0xB
        mov EBX, 0xC
    }

    return 0;
}
```

# 레지스터에 E를 붙이는 이유

AX : 16비트 용 레지스터

- EAX : 32비트
- RAX : 64비트

# EIP 레지스터의 용도 확인해보기

IP : Instruction Pointer

```
int main()
{
    int a = 0; 경과 시간 1ms 이하
    a = 1;
    a = 2;

    for (int i = 0; i < 5; i++) {
        a = i;
    }

    return 0;
}
```

# 함수 호출시 EIP 동작 확인

abc 함수 호출 부분에서  
F11을 눌러 함수 내부로 진입  
• EIP값을 확인한다.

```
000D1DCE  xor     ecx,ecx
000D1DD0  mov     eax,0CCCCCCCCh
000D1DD5  rep stos dword ptr es:[edi]
000D1DD7  mov     ecx,offset _A26DCBE1_
000D1DDC  call    @__CheckForDebuggerJui
        abc();
000D1DE1  call    abc (0D13A2h)
```

```
2  void abc() {
3      int a = 10;
4  }
5
6  int main()
7  {
8      abc(); 경과 시간 1ms 이하
9
10     return 0;
11 }
```

# Push POP

스택의 동작을 이해한다.

- 아래에서 위로 쌓인다.

0xFFF

Heap

0x000

Stack



메모리 1					
주소: ESP		크기: 4			
0x00BBFD34	01 00 00 00	....			
0x00BBFD38	23 10 06 00	#...			
0x00BBFD3C	23 10 06 00	#...			
0x00BBFD40	00 30 de 00	.0?.			
0x00BBFD44	05 00 00 00	....			
0x00BBFD48	64 fd bb 00	d??.			

```
_asm {  
    push 1  
    push 2  
    push 3  
    push 4  
    push 0xA  
    push 0xB  
  
    pop EAX  
    pop EBX  
    pop ECX  
    pop EDX  
  
    push 0xDEADBEEF  
    pop EAX  
  
    push 1  
}
```



# 변수값을 바꾸어본다

메모리값 변조

```
#include <stdio.h>

int main()
{
    int a = 0xABCD1234;

    printf("%X", a);

    return 0;
}
```

# 코드 영역 메모리 확인

메모리 : &a 로 검색

- 인텔 / AMD : Little Endian

메모리 1				
주소:	0x004FF880			
0x004FF880	34	12	cd	ab

```
#include <stdio.h>

int main()
{
    int a = 0xABCD1234;

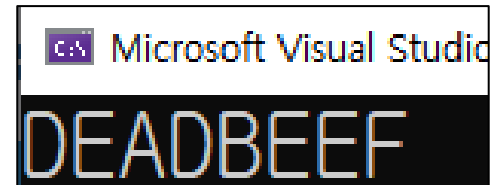
    printf("%X", a);

    return 0;
}
```

# 메모리 값 변경

메모리값을 변경 후 F5를 누른다.

메모리 1					
주소: 0x004FF880					
0x004FF880	ef	be	ad	de	????
0x004FF884	cc	cc	cc	cc	????



# 함수 호출 분석하기

함수 호출 시,  
돌아올 주소를 스택에 저장한다.

```
#include <stdio.h>

void abc()
{
    printf("#");
}

int main()
{
    abc();

    return 0;
}
```

# 코드 수정하기

10을 보낸다.

```
#include <stdio.h>
```

```
void abc(int a)
{
}
```

```
int main()
{
    abc(10);

    return 0;
}
```

# 함수 호출 전

push 10 (0xA) 후 call을 한다.

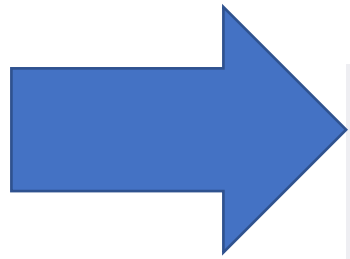
```
00E8250C  call    @__CheckForDebugger
00E82511  push    0Ah
00E82513  call    abc (0E813A2h)
00E82518  add     esp,4
```

레지스터	
EAX	= 00E8C000
EBX	= 00645000
ECX	= 00E8C000
EDX	= 00000001
ESI	= 00E81023
EDI	= 0095FED4
EIP	= 00E82511
ESP	= 0095FE08
EBP	= 0095FED4
EFL	= 00000246

# 돌아올 주소 위치 확인

a = 10 값 바로 위 스택에 0xE82518 돌아갈 주소가 적혀있음

이곳을  
변조할 예정



0x007BFA1C	18	25	e8	00	.%?.
0x007BFA20	0a	00	00	00	....

# Buffer Overflow 공격

메모리를 변조하여, 원하는 코드 삽입 또는 실행하는 공격법



# 다음 코드를 이해해보자. (Trace)

```
소스.cpp  x
Project69  (전역 범:

1  #include <stdlib.h>
2
3  void hack() {
4      system("cmd.exe");
5  }
6
7  void abc(int a)
8  {
9      *(&a - 1) = (int)hack;
10 }
11
12 int main()
13 {
14     abc(10);
15
16     return 0;
17 }
```

```
C:\Users\minco\source\repos\Project69\Debug\Project69.exe
Microsoft Windows [Version 10.0.19042.1415]
(c) Microsoft Corporation. All rights reserved.

C:\Users\minco\source\repos\Project69\Project69>/
```

# buf 추가하기

buf 배열 추가

```
#include <stdlib.h>
#include <stdio.h>

void hack() {
    system("cmd.exe");
}

void abc(int a)
{
    char buf[3] = { 0 };
    buf[0] = 0x77;
}

int main()
{
    abc(10);

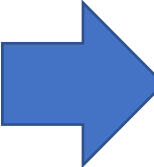
    return 0;
}
```

# 함수 호출시 상황

스택에 리턴될 주소가 들어가있다.

```
00392515 rep stos    dword ptr es:[edi]
00392517 mov      ecx,offset _A26DCBE1_소스@c
0039251C call    @__CheckForDebuggerJustMyC
          abc(10);
00392521 push    0Ah
00392523 call    abc (03913A2h)
00392528 add     esp,4
```

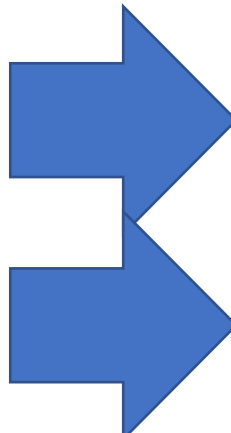
경과 시간 1ms 이하



0x006FF91C	e8 f9 6f 00	??o.
0x006FF920	00 a0 45 00	.?E.
0x006FF924	80 2d a9 7b	.-?{
0x006FF928	75 89 6f 65	u?oe
0x006FF92C	01 c0 39 00	.?9.
0x006FF930	01 c0 39 00	.?9.
0x006FF934	28 25 39 00	(%9.
0x006FF938	0a 00 00 00	....
0x006FF93C	23 10 39 00	#.9.
0x006FF940	?? ?? ?? ??	#.?

# buf 값과 12칸 떨어져있음

12칸 Buffer Overflow 공격 시도



0x006FF91C	e8	†9	6†	00	??o.
0x006FF920	00	a0	45	00	.?E.
0x006FF924	cc	cc	cc	cc	????
0x006FF928	77	00	00	cc	w..?
0x006FF92C	cc	cc	cc	cc	????
0x006FF930	08	fa	6f	00	.?o.
0x006FF934	28	25	39	00	(%9.
0x006FF938	0a	00	00	00	....
0x006FF93C	23	10	39	00	#.9.

# 수동 Injection

12칸 후에 hack 주소 삽입

```
#include <stdlib.h>

void hack() {
    system("cmd.exe");
}

void abc(int a)
{
    char buf[3] = { 0 };
    buf[0] = 0x77;

    int* p = (int*)(buf + 12);
    *p = (int)hack;
}

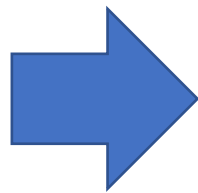
int main()
{
    abc(10);

    return 0;
}
```

# 메모리를 강제로 변조해서 호출해보자.

hack 함수로 강제 변경

0x010FF948	cc	cc	cc	cc	????
0x010FF94C	77	00	00	cc	w..?
0x010FF950	cc	cc	cc	cc	????
0x010FF954	2c	fa	0f	01	,?..
0x010FF958	a8	18	d1	00	?..?
0x010FF95C	0a	00	00	00	....



0x010FF948	cc	cc	cc	cc	????
0x010FF94C	77	00	00	cc	w..?
0x010FF950	cc	cc	cc	cc	????
0x010FF954	2c	fa	0f	01	,?..
0x010FF958	0e	11	d1	00	..?.
0x010FF95C	0a	00	00	00	....
0x010FF960	??	??	??	??	....

# 버퍼 오버플로우 공격

scanf를 넣어주자.

```
#include <stdlib.h>
#include <stdio.h>

void hack() {
    system("cmd.exe");
}

void abc(int a)
{
    char buf[3] = { 0 };
    buf[0] = 0x77;

    scanf("%s", buf);
}

int main()
{
    printf("%X\n", hack);
    abc(10);

    return 0;
}
```

# buf에 원하는 코드 값 넣어보기

키보드로 쓰기 어려운 문자를 넣는 방법

- 파이썬 Script로 셸을 실행한다.

```
scanf("%s", buf);  
printf("%X %X\n", buf[0], buf[1]);
```

```
C:\Users\minco\source\repos\Project69\Debug>python -c "print('\x12\x34')" | Project69.exe  
12 34  
C:\Users\minco\source\repos\Project69\Debug>  
C:\Users\minco\source\repos\Project69\Debug>  
C:\Users\minco\source\repos\Project69\Debug>
```

0x80 (128) 이상 숫자를 scanf로 입력받을 수 없다.



# hack 주소 변경

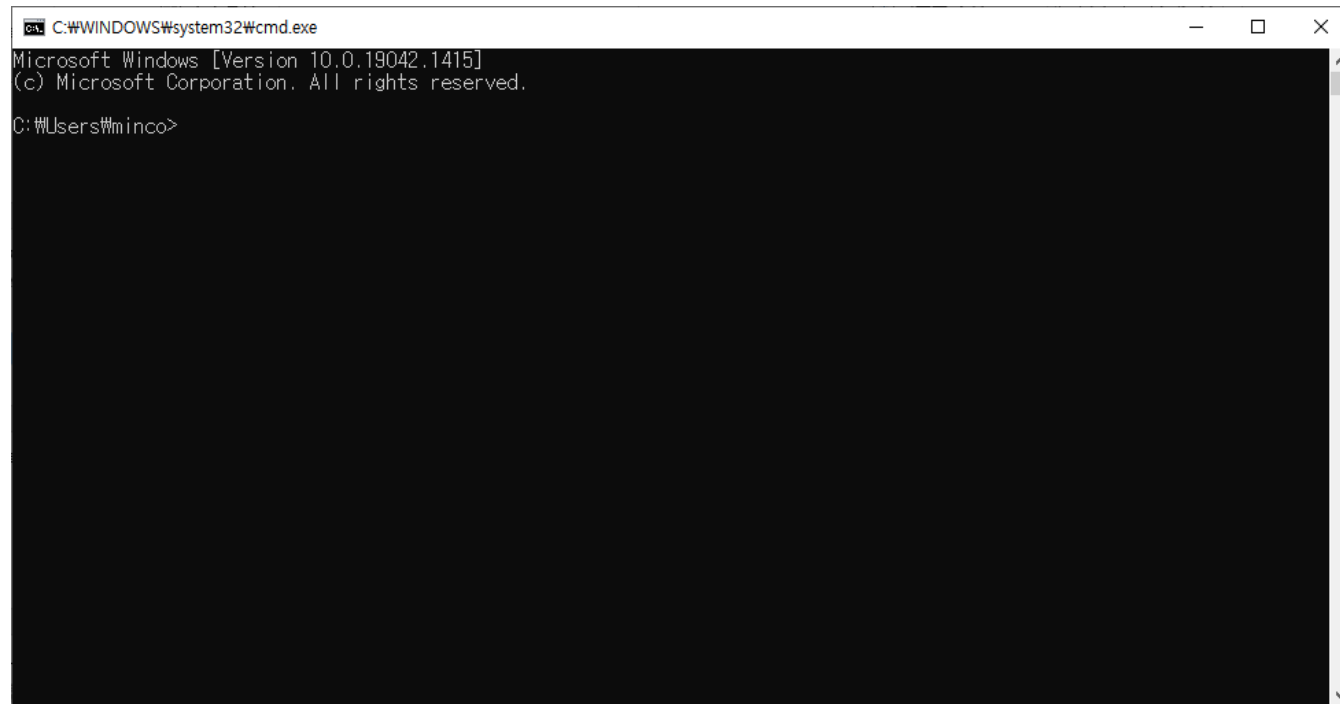
hack 메모리 주소가

0x00 ~ 0x7F 사이로 이루어진 주소가 나올때 까지  
소스코드를 수정해보자.

```
int main()  
{  
    int a[1] = { 0 };  
    abc(10);  
  
    return 0;  
}
```

# [도전] 버퍼 오버플로우 공격해보기

cmd shell이 수행되도록 한다.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19042.1415]
(c) Microsoft Corporation. All rights reserved.

C:\Users\minco>
```

# 정수 Overflow

정수 값에 대한 Overflow 방지

# 정수 13억 x 2의 값 = 음수

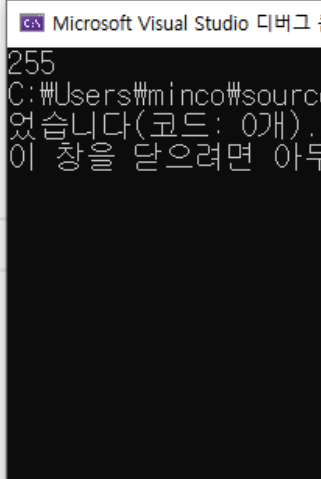
Overflow가 발생해도 에러를 발생시키지 않음

- (stdint.h) uint8\_t 에 음수를 넣는 경우

```
#include <stdio.h>
#include <stdint.h>
int main()
{
    uint8_t x;
    x = -1;

    printf("%d", x);

    return 0;
}
```



The image shows a code editor with a C program that declares a `uint8_t` variable `x` and assigns it the value `-1`. It then prints `x` using `printf("%d", x);`. To the right, a debugger window titled "Microsoft Visual Studio 디버그 콘솔" (Microsoft Visual Studio Debug Console) shows the output of the program. The first line of output is "255", which is the unsigned integer representation of -1 in a 256-byte `uint8_t` container. Subsequent lines show a file path and some Korean text.

# Overflow 값을 예상할 수 있어야 한다.

다음 결과를 예측해보자.

```
uint8_t x = -5;  
  
printf("%d", x);
```

```
uint8_t x = -3;  
x += 5;  
  
printf("%d", x);
```

```
uint8_t x = 10;  
  
x += 100;  
x += 100;  
x += 100;  
  
printf("%d", x);
```

# 정수 Overflow

수 역전현상 발생

```
#include <stdio.h>
#include <stdint.h>

uint32_t DM() {
    return 0xDDAABBFF;
}

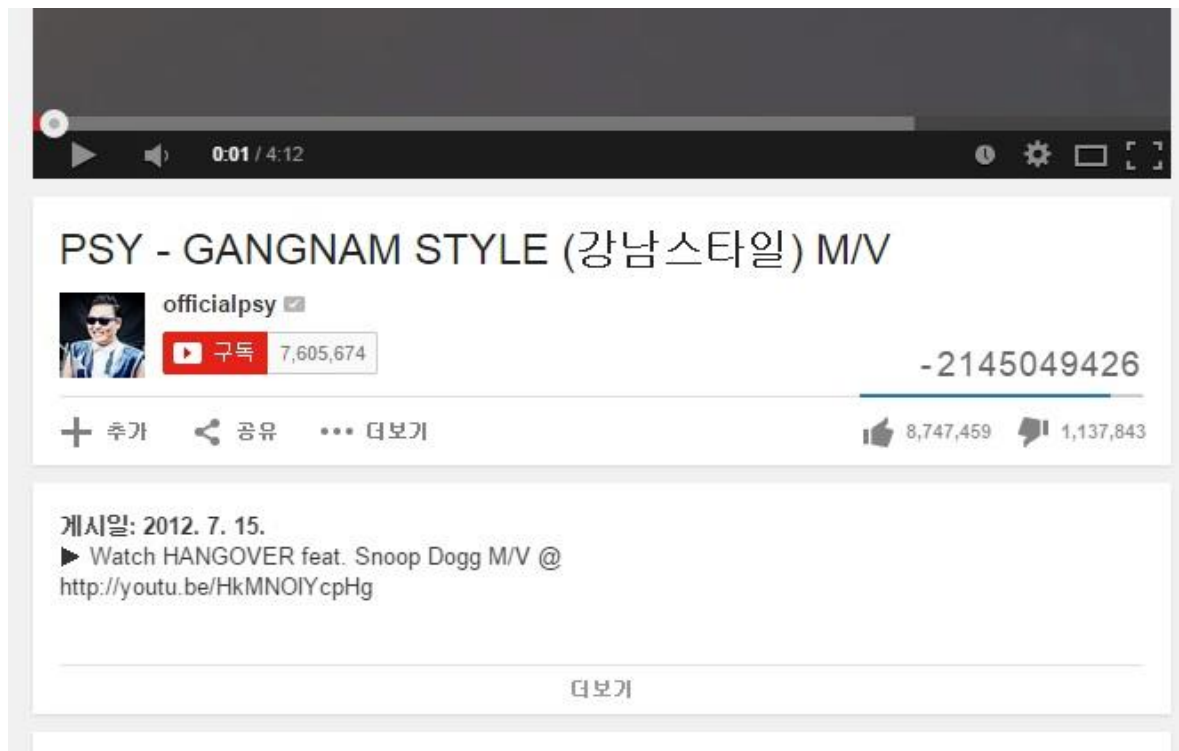
int main()
{
    int data = DM();
    int n = 0x77BBCCDD;

    if (data > n) printf("DATA WIN");
    else printf("N WIN");

    return 0;
}
```

# [도전 1] Overflow 해결하기

다음 취약점을 해결하자.



```
#include <stdio.h>
#include <stdint.h>

int cnt = INT32_MAX - 5;

void click() {
    cnt++;
    printf("View = %d\n", cnt);
}

int main()
{
    click();
    click();
    click();
    click();

    return 0;
}
```

# [도전 2] Latency 문제 해결하기

다음과 같은 버그 발생 가능

- start = 21억 측정 시작
- end = -21억 측정 종료

```
#include <stdio.h>
#include <time.h>

void action() {
    //...
}

int main()
{
    clock_t start = clock();

    action();

    clock_t end = clock();
    printf("%d ms", end - start);

    return 0;
}
```