정보보안론 5주차 강의

과제

1. **BOF(Buffer Overflow) 코드의 문제점을 파악하고 보안을 적용하시오.**
2. **Check_password() 함수의 스택구조를 확인하시오.**
3. **메모리핵 툴을 이용하여 메모리값을 변조하도록 하시오.**
4. **프로그램 진행방향을 조작하시오.**

! 시험 ! password123을 넣었는데 correct~! 라고 뜨는 이유

메모리 세그먼트

Code Segment	: 프로그램 실행 코드가 저장되는 공간
Stack Segment	: 함수 호출 시 생성되는 스택 프레임이 저장되는 공간
Data Segment	: 전역 변수, static 변수 저장
Extra Segment	: 추가적으로 사용하는 세그먼트

명령어는 모두 Code Segment 로 보낸다.

변수는 모두 Stack Segment 로 보낸다.

- ➔ 지역변수
- ➔ Stack 정보
- ➔ Return Adrees
- ➔ Error 처리

문자열 상수는 Data Segment 로 보낸다.

(영역이 3가지로 나뉘어짐)

- ➔ rdata ; read Only
- ➔ data : read Write
- ➔ rat : 함수 주소

프로그래머를 위해 비워둔 Extra Segment

Check_password() 함수의 스택구조를 확인

스택 프레임 구조

몇개더 …				<- esp
		0	0	<- pw (지역 변수 – char 배열, 10B + padding)
0	0	0	0	
0	0	0	0	
flag = 0				<- 지역변수 할당
8byte EBP				
mem_ebp				<- 이전 함수의 ebp
return address				<- ebp
				<- ESP
main()				<- EBP
OS stack				

Return Address (복귀 주소) → 함수가 끝나면 돌아갈 위치

이전 EBP (saved base pointer) → 호출한 함수의 스택 프레임 기준점

flags = 0 (레지스터 상태) → 상태 플래그 저장

EBP (base pointer) → 스택 프레임 기준점

ESP (stack pointer) → 현재 스택의 맨 위를 가리킴

Main() 함수가 호출되면 스택 프레임이 생기고, 모든 스택에 밑 바닥에는 자동으로 Return Address가 만들어진다.

함수가 끝날 때 Return Address로 복귀한다.

BOF(Buffer Overflow) 코드의 문제점을 파악하고 보안을 적용

보안 적용 방법 1 : 보안 옵션이 있는 컴파일로 가는 방법 (visual c++ 에서 gets_s()를 사용 한다.)

보안 적용 방법 2 : 변수를 명확하게 써야한다.

보안 적용 방법 3 : 돌아 오는 값을 확실하게 비교 해야한다.

[*] hecktest.c

```c
#include <stdio.h>

int check_password()
{
    int flag=0;         // 버퍼오버플로우 발생
    char pw[10] = "";

    printf("input password : ");
    gets(pw); // 보안 적용 방법 1 (보안 옵션이있는 컴파일로 가는 것) : visual c++ 에서 gets_s()를 사용 하는 것을 고려

    if( strcmp (pw,"password") == 0)
        flag = 1;
    else            // 보안 적용 방법 2 (변수를 명확하게 써야한다.) : 이 코드가 있어야 한다.
        flag = 0; // ""

    return flag;
}

int main()
{
    if( check_password() == 1) // 보안 적용 방법 3 : 돌아 오는 값을 확실하게 비교 해야한다.
    {
        printf("Correct~!\n");
    }
    else
    {
        printf("Wrong~!\n");
    }
    getch();
    return 0;
}

```

! 시험 - 보안 적용 방법 (위의 사진 참고)!

메모리핵 툴을 이용하여 메모리값을 변조

프로그램 진행방향을 조작

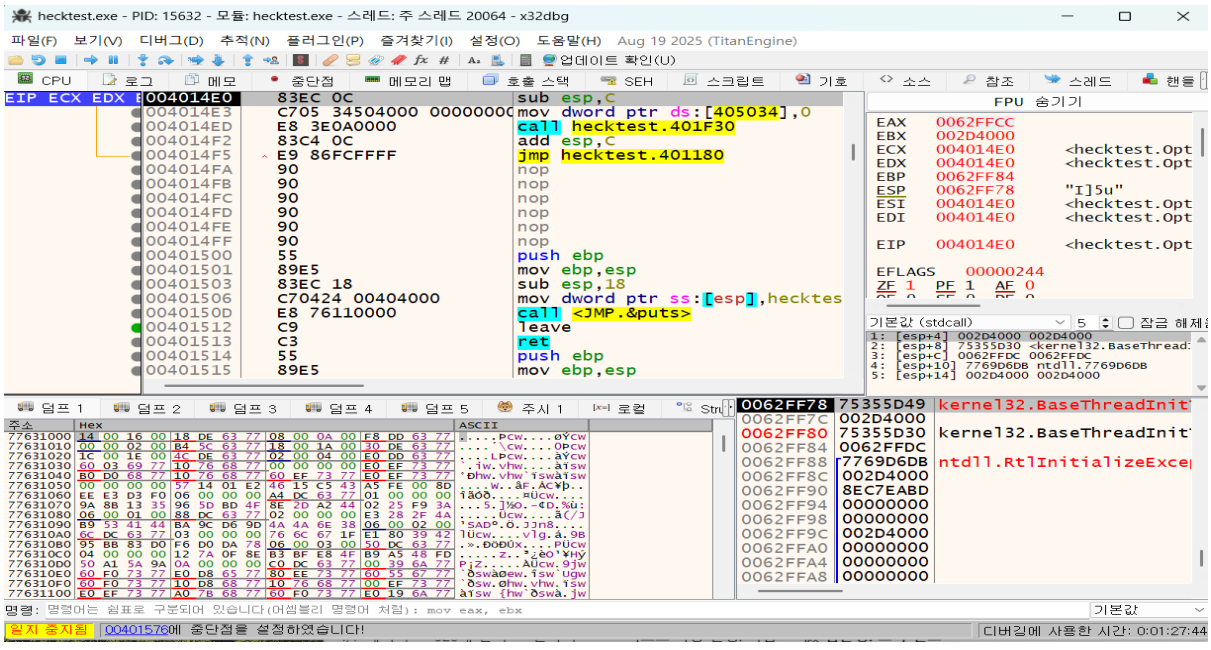
코드에 hack_function() 추가

hecktest.c

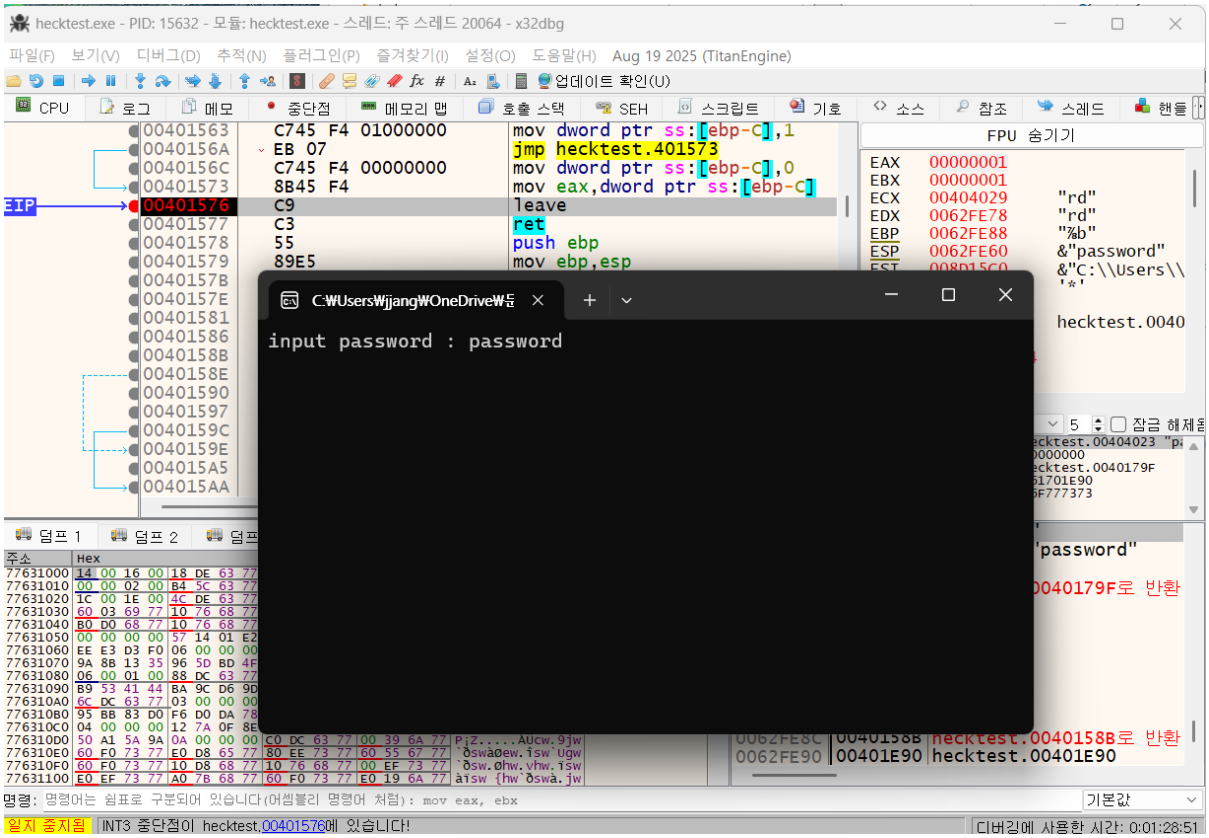
```c
#include <stdio.h>

int hack_function()
{
    printf("Hack Function~!!\n");
}


int check_password()
{
    int flag=0;         // 버퍼오버플로우 발생
    char pw[10] = "";

    printf("input password : ");
    gets(pw); // 보안 적용 방법 1 (보안 옵션이있는 컴파일로 가는 것) : visual c++ 에서 gets_s()를 사용 하는 것을 고려

    if( strcmp (pw,"password") == 0)
        flag = 1;
    else          // 보안 적용 방법 2 (변수를 명확하게 써야한다.) : 이 코드가 있어야 한다.
        flag = 0; // ""

    return flag;
}

int main()
{
    if( check_password() == 1) // 보안 적용 방법 3 : 돌아 오는 값을 확실하게 비교 해야한다.
    {
        printf("Correct~!\n");
    }
    else
    {
        printf("Wrong~!\n");
    }
    getch();
    return 0;
}
```

스냅샷 이용

push ebp 부터 ret 까지가 한 모듈 , 리턴 주소를 변경하면 hack_function()으로 가도록 할 수 있다.

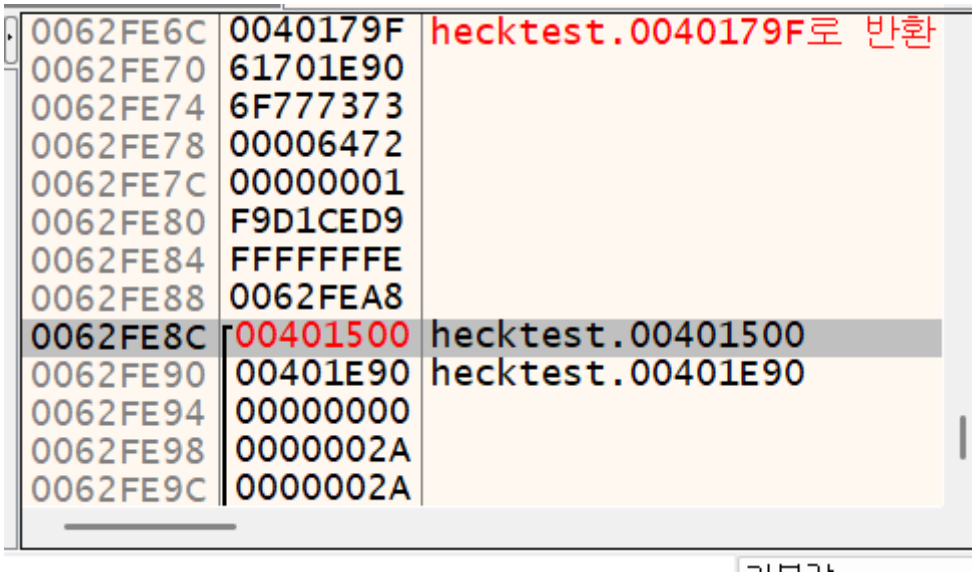


hack_function() 끝나는 곳에 break 걸기



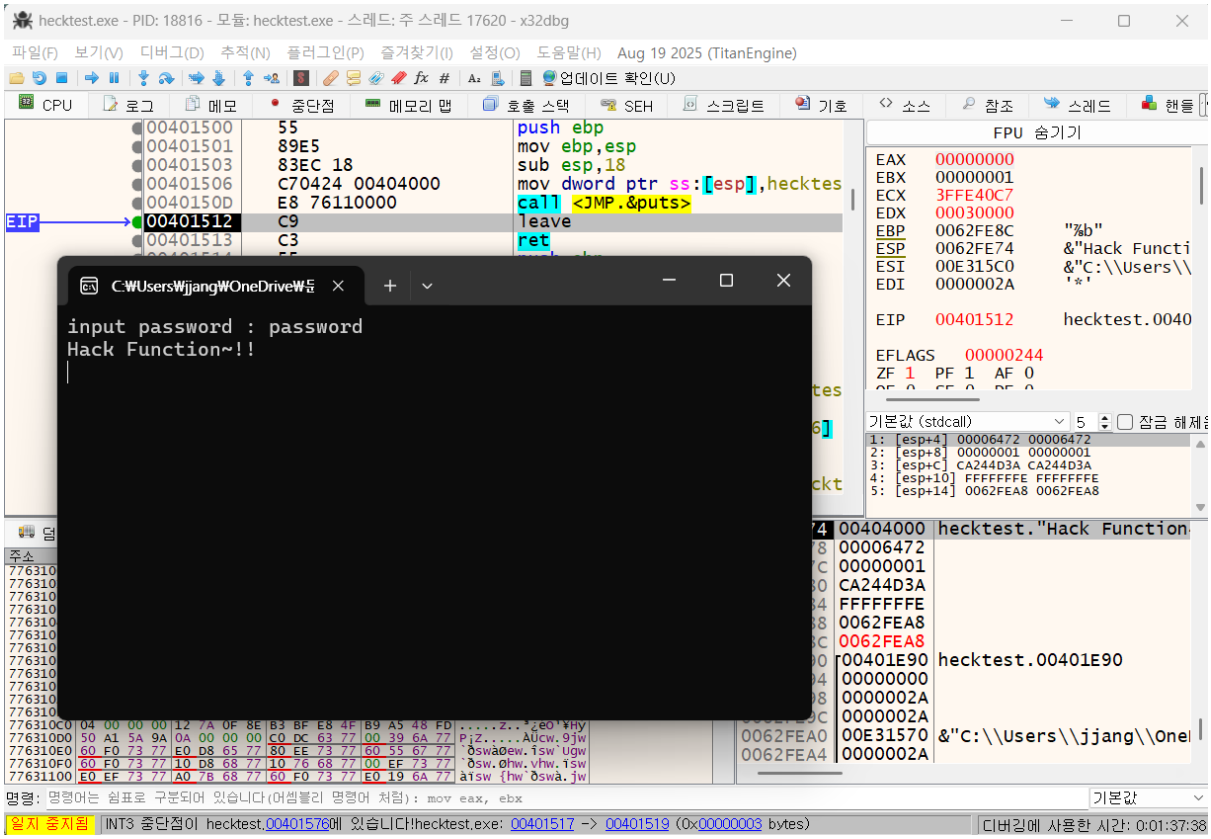
EBP(마지막 스택) 주소 확인 -> 00401500

메모리값 변조 -> 0040158B를 00401500으로 뒤에 두개를 변경



프로그램 진행반향 조작

leave : 파괴하는 명령 -> F8 입력



! 시험 – 스택 영역과 힙 영역 !

스택 영역 -> 지역변수, Stack 정보, Return Adrees, Error 처리

프로그램 로직이 동작하기 위한 인자와 프로세스 상태를 저장하는 데 사용.

레지스터의 임시 저장, 서브루틴 사용 시 복귀 주소저장, 서브루틴에 인자 전달 등에 사용.

힙 영역

프로그램이 동작할 때 필요한 데이터 정보를 임시로 저장하는 데 사용

프로그램이 실행될 때까지 알 수 없는 가변적인 양의 데이터를 저장하기 위해 프로그램 프로세스가 사용할 수 있도록 미리 예약된 메인 메모리 영역.

힙 영역은 프로그램에 의해 할당되었다가 회수되는 작용을 되풀이함

힙의 기억 장소는 포인터 변수를 통해 동적으로 할당받고 돌려주며, 연결 목록이나 나무, 그래프 등의 동적인 데이터 구조를 만드는 데 반드시 필요함

프로그램 실행 중에 해당 힙 영역이 없어지면 메모리 부족으로 이상 종료

! 시험 - 각각의 레지스터의 용도 !

레지스터 : CPU의 임시 메모리로 CPU 연산과 어셈블리어의 동작에 필요

표 5-1 80×86 CPU의 레지스터

범주	80386 레지스터	이름	비트	용도
범용 레지스터	EAX	누산기(accumulator)	32	산술 연산에 사용(함수의 결과 값 저장)
	EBX	베이스 레지스터(base register)	32	특정 주소 저장(주소 지정을 확대하기 위한 인덱스로 사용)
	ECX	카운트 레지스터(count register)	32	반복적으로 실행되는 특정 명령에 사용(루프의 반복 횟수나 좌우 방향 시프트 비트 수 기억)
	EDX	데이터 레지스터(data register)	32	일반 데이터 저장(입출력 동작에 사용)
세그먼트 레지스터	CS	코드 세그먼트 레지스터 (code segment register)	16	실행 기계 명령어가 저장된 메모리 주소 지정
	DS	데이터 세그먼트 레지스터 (data segment register)	16	프로그램에서 정의된 데이터, 상수, 작업 영역의 메모리 주소 지정
	SS	스택 세그먼트 레지스터 (stack segment register)	16	프로그램이 임시로 저장할 필요가 있거나 사용자의 피호출 서브루틴이 사용할 데이터와 주소 포함
	ES, FS, GS	엑스트라 세그먼트 레지스터 (extra segment register)	16	문자 연산과 추가 메모리 지정에 사용되는 여분의 레지스터
포인터 레지스터	EBP	베이스 포인터(base pointer)	32	SS 레지스터와 함께 스택 내의 변숫값을 읽는 데 사용
	ESP	스택 포인터(stack pointer)	32	SS 레지스터와 함께 스택의 가장 끝 주소를 가리킴
	EIP	명령 포인터(instruction pointer)	32	다음 명령어의 오프셋(상대 위치 주소)을 저장하며, CS 레지스터와 합쳐져 다음에 수행될 명령의 주소 형성

범주	80386 레지스터	이름	비트	용도
인덱스 레지스터	EDI	목적지 인덱스(destination index)	32	목적지 주소의 값 저장
	ESI	출발지 인덱스(source index)	32	출발지 주소의 값 저장
플래그 레지스터	EFLAGS	플래그 레지스터(flag register)	32	연산 결과 및 시스템 상태와 관련된 여러 가지 플래그 값 저장

프로그램 실행 구조

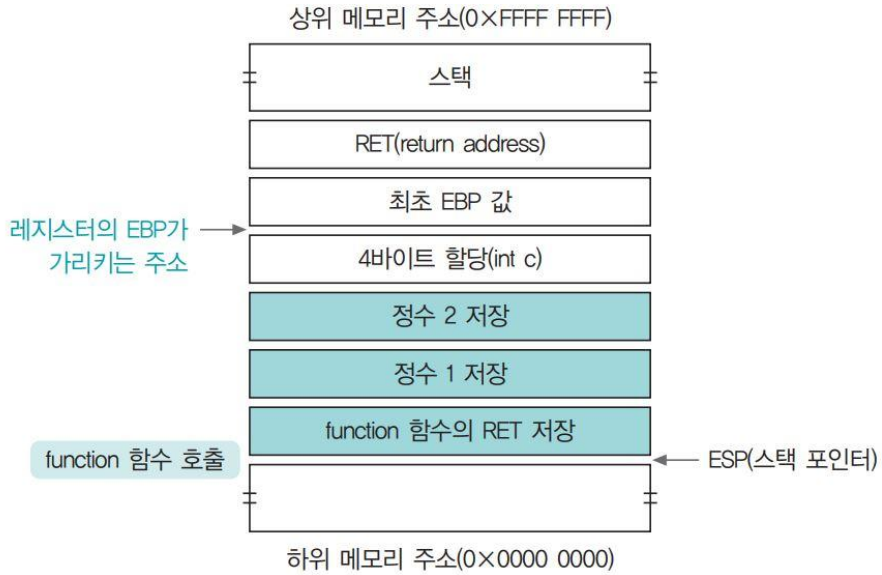


그림 5-6 pushl $2, pushl $1, call function 실행 시 스택의 구조

! 시험 - 버퍼 오버플로 공격의 대응책 !

버퍼 오버플로에 취약한 함수 불사용

strcpy(char *dest, const char *src);

strcat(char *dest, const char *src);

getwd(char *buf);

gets(char *s);

fscanf(FILE *stream, const char *format, ...);

scanf(const char *format, ...);

realpath(char *path, char resolved_path[ ]);

sprintf(char *str, const char *format);