情 報 保 護 學 會 誌
第 11 券 第 6 號, 2001. 12

Heap과 Stack 영역에서의 경계 체크를 통한 Buffer Overflow 공격 방지 기법에 대한 연구

손 태 식*, 서 정 택**, 은 유 진*, 장 준 교*, 이 철 원**, 김 동 규***

요 약

인터넷 기술의 발전은 정보화 사회로의 촉진이라는 측면 외에 사용 인구의 증가에 따라 해킹, 바이러스·악성 코드의 유포 등 그 역기능적인 측면이 점점 심각한 사회 문제로서 대두되고 있다. 현재 공개 운영체제로 각광 받고 있는 리눅스(Linux) 운영체제의 경우 운영체제 커널은 물론이거니와 관련 프로그램들에 대한 소스가 공개되어 단순한 기술 습득의 목적이 아닌 악의의 목적을 가진 사용자들에 의한 시스템 침해 사례가 빈번한 추세이다. 특히 이러한 시스템 침해 사례 중 프로그램 작성 과정의 오류 및 설계상 실수로 인한 버퍼 오버플로우(Buffer Overflow) 취약성을 이용한 공격은 해킹에 있어 큰 범주를 차지하고 있다. 따라서 본 논문에서는 버퍼 오버플로우 공격에 있어 그 기반이 되는 스택을 이용한 버퍼 오버플로우 및 스택 외에 힙과 같은 메모리 영역을 이용하는 공격 유형에 대하여 분석한다. 그 후 이러한 버퍼 오버플로우 공격 방지를 위한 메모리에서의 경계 검사 기법을 제안하고자 한다. 추후에는 본 논문에서 제안된 기법에 대한 실제 구현과 검증이 필요하다.

Ⅰ. 서 론

최근 인터넷 기술을 기반으로 한 통신망의 발전으로 전 세계의 컴퓨터들이 거대한 하나의 네트워크로 연결되면서 해킹이나 바이러스·악성 코드의 유포와 같은 침해 사고들이 빈번하게 발생하여 많은 피해를 입고 있다[14]. 최근의 침해사례를 살펴보면 컴퓨터 시스템에 대한 해킹수법이 지능화되고 다양해지면서 사용이 늘어가는 리눅스 및 유닉스 기반 운영체제의 시스템 취약점을 이용하는 수법이 해킹의 많은 부분을 차지하고 있다. 특히 리눅스(Linux) 운영체제를 기반으로 하는 시스템에서는 리눅스 운영체제의 소스가 무상으로 배포되는 점을 이용하여 운영체제 커널 및 관련 리눅스 운영체제 기반 프로그램의 소스 분석을 통한 취약성을 이용하여 해킹사고가 많이 발생되고 있다[11]. 리눅스 운영체제는 소스코드의 공개 및 관련 문서의 풍부한 제공으로 쉽게 수정이 가능하며, PC 기반에서 운영이 가능하고 설치비용도 거의 들지 않아 최근 급속도로 보급이 확산되고 있다. 따라서 리눅스를 기반으로 하는 시스템에서는 더더욱 보안에 만전을 기해야 할 것이다.

빈번히 발생하는 리눅스 운영체제 기반의 시스템 침해 사례 중 많은 비중을 차지하고 있는 것이 스택(Stack:이하 스택)과 힙(Heap:이하 힙)과 같은 메모리 영역의 버퍼 오버플로우(Buffer Overflow)를 이용한 공격 기법이다[1]. 버퍼 오버플로우 공격이란 스택에 할당되어 있는 임의의 버퍼에 특정 주소에 저장되어 있는 공격 코드를 실행시키기 위해 공격 코드가 위치한 주소 값을 버퍼를 넘치게(오버플로우)하는 기법을 사용하여 삽입함으로써 추후 함수가 리턴 주소를 참고하는 시점에서 정상적인 리턴 주소 대신 공격 코드가 위치한 주소를 가리키게 만들어 공격을 시도하는 것을 말한다. 버퍼 오버플로우 공격에는 단순히 스택에 할당된 임의의 버퍼를 넘치게 하는 방법 외에도 함수 포인터를 변경하는 방법, "null" 값으로 종료되지 않는 비정상적 스트링을 이용하여 시스템의 오류를 발생시키는 방법 그리고 최근에 나타나기 시작한 포맷 스트링[10]을 이

* 아주대학교 정보통신공학과 정보통신 및 시큐리티 연구실 (tsshon@madang.ajou.ac.kr)
** ETRI 부설 국가보안기술연구소 ({seojt, cheolee}@etri.re.kr)
*** 아주대학교 정보통신공학과 교수 (dkkim@madang.ajou.ac.kr)

용하는 등의 많은 방법이 존재한다. 하지만 본 논문에서는 버퍼 오버플로우 공격 가운데 가장 대표적이며 널리 사용되는 스택 영역에서의 버퍼 오버플로우를 탐지 및 방지할 수 있는 기법과 또한 스택 외의 힙 영역에서의 버퍼 오버플로우 대응방안에 대해서 연구한다[7][8].

본 논문은 다음과 같이 구성되어 있다. 제2장에서는 스택 및 힙 영역에서의 취약성을 이용한 공격 기법을 분석한다. 제3장에서는 분석된 스택 및 힙 영역에서의 공격 기법을 이용하여 실제 공격을 탐지 및 방지 할 수 있는 스택과 힙 영역에서의 메모리 경계 검사 기법을 제안한다. 제4장에서는 본 논문의 결론 및 향후 연구 방향을 서술한다.

II. 스택과 힙 영역에서의 취약성 및 공격 기법 분석

본 장에서는 버퍼 오버플로우 공격에서 주로 사용하는 스택 영역과 힙 영역에서의 버퍼 오버플로우 취약성 및 공격 기법에 대해서 분석한다. 이때 각각의 메모리 영역에 할당되는 버퍼의 형태는 다음과 같다. 스택 영역에 할당되는 버퍼는 "char buffer1[BufSize];" 이와 같은 형태를 가지며 힙 영역에서 할당되는 버퍼는"char *buf=(char*)malloc(BufSize)" 과 같이 "alloc()"계열 함수들에 의해 할당되어 사용된다. 또한 이러한 메모리 영역에 대한 구분이 아닌 프로그램 언어 상의 취약성이나 공격 기법에 따라서 다음과 같은 버퍼 오버플로우 관련 공격 기법들이 존재한다. 배열의 경계를 침범하는 기법, Non-Terminate 스트링(Null로 끝나지 않는 스트링)을 사용하는 기법, "%n" 지시자(directive)를 이용하는 포맷 스트링 기법 그리고 포인터 및 함수 포인터를 변경하는 하는 등의 포인터 주소 변경 기법. 이러한 취약성 및 공격 기법은 사용자의 목적에 따라 단순히 세그먼트 오류 등의 프로그램 혹은 시스템에서의 오류를 발생하는 수준에서 멈출 수도 있지만 또 한편으로는 시스템마다의 고유 특성을 면밀히 분석하여 실제로 슈퍼 유저 권한을 취득하는 것과 같은 실제 시스템 침해로 이어 질 수 있다[13][26].

따라서 본 논문의 2.1절과 2.2절에서는 스택 영역과 힙 영역 각각에서의 메모리 영역 특징에 의한 버퍼 오버플로우 취약성 및 공격 기법을 분석하여 근본적으로 스택 영역 및 힙 영역에서의 버퍼 오버플로우 공격을 방지할 수 있는 방법을 연구한다.

2.1 스택 영역의 버퍼 오버플로우

일반적으로 널리 알려진 버퍼 오버플로우 공격 기법은 보통 스택 영역에 할당된 버퍼를 오버플로우 시켜 함수의 리턴 주소를 변경(overwrite)하는 방법을 지칭한다. 다음의 [예제-1]에서는 할당 된 버퍼들이 어떻게 스택에 저장되는지 알 수 있다.

[예제-1]

```
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
}

int main(void) {
  function(1,2,3);
  return 0;
}
```

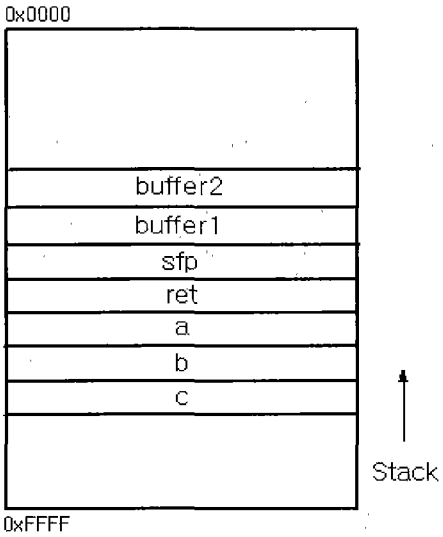
[예제-1]의 프로그램이 "function()"을 호출하는 과정에서 스택에 리턴 주소와 지역 변수들이 저장되는 과정을 파악하기 위해 "gcc" 컴파일러의 "-S" 옵션을 사용하여 어셈블리 코드를 생성한다. 생성된 어셈블리 코드는 다음과 같다:

```
pushl $3
pushl $2
pushl $1
call function
```

위의 어셈블리 코드에서 함수에 대한 3개의 인자들은 스택에 반대순서(3,2,1)로 저장되고 그 후 "call function()"에 의하여 "function()"이 호출된다. 이때 "call" 명령은 현재의 명령 포인터(Instruction Pointer)를 스택에 저장한다.(push) 이때 저장된 명령 포인터를 함수의 리턴 주소(RET)라고 부른다. 호출된 함수 안에서 가장 먼저 처리되는 것은 프로시저 프롤로그과정인 프레임 포인터 값의 설정 및 지역 변수들을 위한 공간 할당으로 실제 위의 [예제-1]에 대한 어셈블리 코드는 다음과 같다:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

위의 어셈블리 코드는 우선 현재 프레임 포인터인 "EBP"(스택 프레임 포인터)를 스택에 밀어 넣고(push %ebp) 그런 다음 "EBP"를 새로운 프레임 포인터로 만들기 위해 현재의 "ESP"(스택 포인터)를 "EBP"에 복사한다.(movl %esp, %ebp) 그런 다음 지역 변수들의 크기만큼 스택 포인터를 감소시킴으로써(subl $20, %esp) 지역 변수들을 위한 공간이 할당되는 것이다. 여기서 메모리는 워드 크기의 배수만큼만 지정되므로 [예제-1]의 경우에 5바이트의 버퍼는 실제로는 8바이트(2워드)만큼의 메모리를 차지할 것이고 10바이트의 버퍼는 12바이트(3워드)만큼의 메모리를 차지한다. 그게 바로 ESP가 20만큼 감소되는 이유이다. 실제 메인 함수에서 "function()"이 호출될 때 스택의 구조는 다음의 [그림 1]과 같다.



[그림 1] 예제-1의 스택 구조

실제로 스택 영역에서의 버퍼 오버플로우는 버퍼가 다룰 수 있는 것보다 더 많은 데이터를 버퍼에 할당함으로써 일어난다. 이때 할당된 버퍼의 크기 이상으로 버퍼에 데이터를 넣어 스택의 리턴 주소가 저장되어 있는 공간에 특정 코드(공격 코드)를 가리키는 주소를 overwrite 함으로써 실제적인 공격이 가능하게 된다. 이러한 스택 영역의 버퍼 오버플로우가 시스템에서 어떻게 발생할 수 있는지 다음의 [예제-2]를 통하여 알 수 있다.

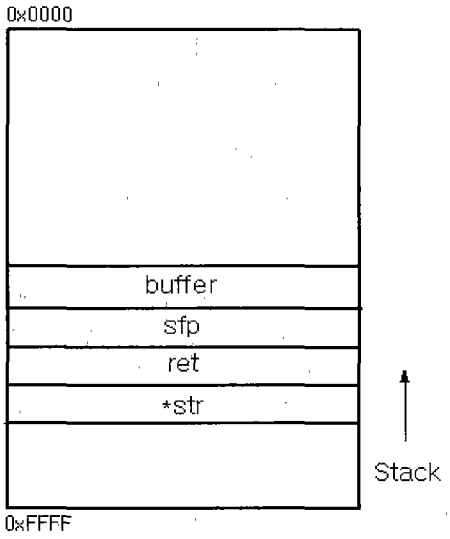
[예제-2]는 전형적인 버퍼 오버플로우 코딩 에러를 포함한 프로그램의 예이다.

[예제-2]

```
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

int main(void) {
    char string[256];
    int i;
    for( i = 0; i < 255; i++)
    string[i] = 'A';
    function(string);
    return 0;
}
```

[예제-2]에서 사용된 함수는 문자열 길이를 확인하지 않고 "strncpy()" 함수 대신에 "strcpy()" 함수를 사용하여 주어진 문자열을 복사한다. 이러한 경우에 있어 프로세스 메모리 구조는 [그림-2]와 같으며 실제로 [예제-2]를 실행한다면 세그먼트 오류가 발생할 것이다. [예제-2]에서 "function()" 함수 내의 "strcpy()"함수는 문자열에서 널문자가 발견 될 때까지, "*str"(string[256])의 내용을 "buffer[16]"로 복사한다. [예제-2]에서 "buffer[16]"의 크기는 16bytes로서 256bytes의 "*str"보다 훨씬 작다.



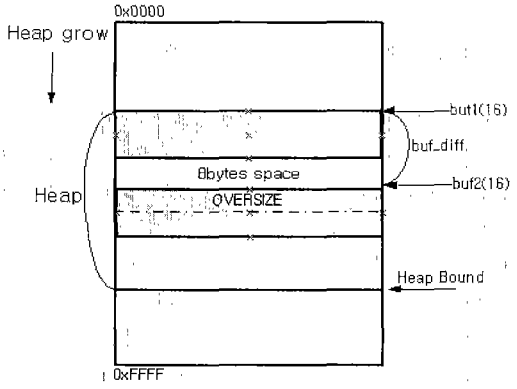
[그림 2] 예제-2의 스택 구조

이것은 스택에 있는 "buffer"뒤의 240바이트가 모두 덮여 쓰게되고 즉, "SFP", "RET" 그리고 심

지어 "*str"까지도 포함한다. [예제-2]의 main() 함수에서는 "string"을 문자 'A'로 채웠었다. 그것의 16진수 문자 값은 "0x41"이고, 이것은 리턴 주소(RET)가 이제 "0x41414141"로 변경되었다는 것을 의미한다. 이 주소는 프로세스 주소공간 바깥에 있다. 따라서 함수가 복귀된 후, 그 주소로부터 다음 명령을 읽으려고 할 때 세그먼트 오류가 발생하게 되는 것이다. 그러므로, 버퍼 오버플로우 공격을 통해 공격자의 공격 코드가 위치한 임의의 주소로 함수의 리턴 주소(RET)를 바꿀 수 있게 된다. 이러한 스택에서의 버퍼 오버플로우 취약성 및 공격 기법을 방지하기 위해서는 프로그램에 사용되는 스트링 라이브러리의 경계 체크가 필수적이다. 하지만 스트링 라이브러리에서는 경계 체크를 지원하는 라이브러리를 가지고 있지만 이러한 라이브러리 역시 Non-Terminate 스트링 취약성과 같은 문제점을 내포하고 있다[16].

2.2 힙 영역의 버퍼 오버플로우

앞서 서론에서도 언급했듯이 데이터 영역은 컴파일 타임에 초기화되며, 그 중에서도 BSS(Block Segment Section) 영역의 경우에는 초기화되지 않은 변수들을 포함한다. 또한 힙 영역은 응용 프로그램에 의해 동적으로 할당되는 메모리 영역이다. 힙 영역의 버퍼 오버플로우는 기본적으로 스택 영역에서의 리턴 주소 overwrite와 유사하다. 하지만 힙 영역은 스택의 리턴 주소와 같이 호출된 함수가 다시 복귀하는 시점에서 실행되는 정해진 값을 overwrite 하는 것이 아니라 버퍼에 할당된 포인터 값을 overwrite 하는 공격 유형이 일반적으로 사용된다. 다음의 [예제-3]은 힙 영역에 할당된 버퍼에 대한 버퍼 오버플로우 공격에 대한 취약성을 보여준다. 만약 데이터 영역에 대한 취약성을 테스트하기 위해서는 프로그램 내부의 버퍼 할당 선언부분에서, "char *buf1 = (char *)malloc(BufSize)"와 같이 선언된 데이터 형을 "static char buffer1[BufSize]"과 같이 수정하면 된다.



[그림 3] 예제-3의 힙 영역 구조

[예제-3]을 실행한 후의 힙 영역의 구조는 [그림-3]과 같다. 즉, [그림-3]에 나타나듯이 "memset (buf1, 'B', (unsigned int)(buf_diff + OVERSIZE));"이 실행되어 "buf1"의 제한 크기를 "OVERSIZE"만큼 넘어서 "Buf2"의 경계를 침범하게된다. 이러한 방법을 사용하여 힙 영역에 인접하게 할당된 버퍼의 내

[예제-3]

```
#define OVERSIZE 8
int main(void)
{
unsigned long buf_diff;
char *buf1 = (char *)malloc(16);
char *buf2 = (char *)malloc(16);

buf_diff = (unsigned long)buf2 - (unsigned long)buf1;
printf("buf1 = %p, buf2 = %p, diff = %p bytes\n", buf1, buf2, buf_diff);
memset(buf2, 'A', 15);
buf2[16] = '\0';
printf("before overflow: buf2 = %s\n", buf2);
memset(buf1, 'B', (unsigned int)(buf_diff + OVERSIZE));
printf("after overflow: buf2 = %s\n", buf2);
return 0;
}
```

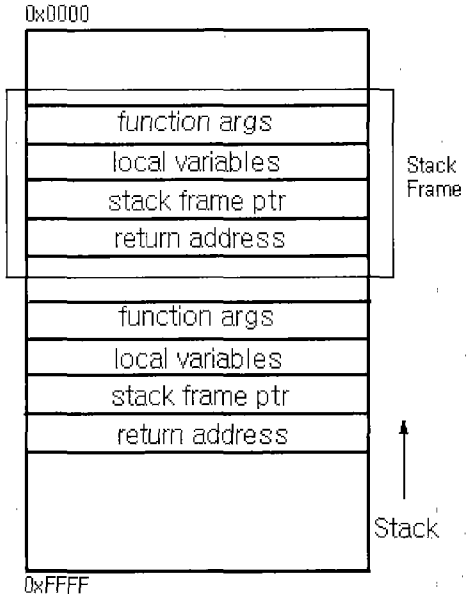
용을 변경하는 것이 가능하다. [예제-3]은 힙 영역의 기본적인 버퍼 오버플로우 기법을 설명한 것으로서 특히 [예제-3]에서 "char *buf1 = (char *) malloc (16), *buf2 = (char *)malloc(16)"처럼 힙 영역의 오버플로우는 스택 영역의 버퍼 오버플로우와 달리 인접한 버퍼를 공략하여 힙 영역에 함수 포인터 등의 값을 변경한다[16].

Ⅲ. 스택과 힙 영역에서의 제안하는 경계 검사 기법의 설계

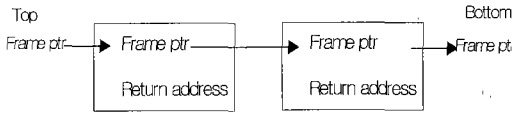
3.1 스택 영역에서의 공격 탐지 기법

앞서의 설명과 같이 스택 영역에서의 버퍼 오버플로우 공격은 프로그램 내에서 실행되는 버퍼에 실제 버퍼의 크기보다 큰 값을 할당하여 오버플로우 된 부분이 스택내 리턴 주소를 overwrite하도록 조작하여 공격을 발생시킨다. 따라서 본 논문에서는 스택 영역에서 실행되는 버퍼의 핸들링 과정에서의 경계 검사를 통하여 발생할 수 있는 공격을 탐지해내고 방지할 수 있는 기반을 마련한다. 하지만 이러한 경계 체크 취약성을 가진 스트링 계열의 함수들(strcpy(), strcat(),...) 이 호출된 경우에 목적장소(실제 값이 복사되는 장소 : dst)의 크기를 알아내는 다음과 같은 어려움이 있다. 첫째로 스택 영역은 [그림 4]와 같이 여러 개의 스택 프레임으로 구성되어 실제로 어떤 스택 프레임에 우리가 원하는 버퍼가 할당되어 있는지 정확히 알아내기가 어렵다. 또한 "strcpy()" 함수의 정의는 다음과 같으며 "strcpy (char *dst, char *src)"에서 "src"의 길이는 이미 "dst"에 할당하려는 스트링이 정해져 있기 때문에 쉽게 "strlen()"등의 스트링 함수를 사용하여 알 수 있지만 "dst"의 경우는 단지 비어 있는 공간에 대한 포인터 주소만 "strcpy()" 함수 내부에서 함수 인자로 전달받기 때문에 "strlen()" 함수나 "sizeof()" 키워드를 사용해서는 그 크기(dst)를 알 수 없다.

따라서 본 논문에서는 현재 "dst"의 크기를 알아내기 위하여 다음과 같은 방법을 사용한다. 스택 영역은 [그림-4] [그림-5]와 같은 구조를 가지는 여러 스택 프레임들로 구성되어 있기 때문에 먼저 "dst"가 포함되어 있는 스택 프레임을 탐색하고 그 후 실제로 "dst"가 포함하고 있는 스택 프레임 포인터 주소와 "dst" 포인터 주소의 차이를 계산하면, 이 값이 "dst"가 가질 수 있는 최대 버퍼 크기가 되는 것이다. 이때 추측된 "dst"의 최대 크기를 사용하여 "src"에서 "dst"로 복사되는 값의 크기가 "dst"의 최대 크기를 넘는지를 검사하며 "dst"의 최대 크기를 넘는 경우에는 버퍼 오버플로우 공격이라고 판단하는 것이 가능하게 된다. [표-1]의 스택 영역에서의 버퍼 오버플로우 공격을 탐지하기 위한 메모리 영역에서의 경계 검사 기법 참조.



[그림-4] 스택 영역내의 스택 프레임들의 구조



[그림-5] 스택 프레임 포인터들의 리스트 구조

[그림-4]와 같이 스택 영역 내부에는 각각의 함수가 호출될 때마다 리턴 주소(Return Address), 지역 변수(Local Variables), 함수 인자(Function Arguments)들을 포함하는 스택 프레임이 생성된다. 또한 이러한 스택 프레임들은 [그림-5]와 같이 스택 영역의 맨 위에 위치하는 스택 프레임 포인터가 하나씩 순차적으로 다음에 위치하는 스택 프레임의 포인터를 가리키는 구조로 구성되어 있어 스택 상위에 위치하고 있는 첫 번째 스택 프레임의 프레임 포인터를 "__bulitin_frame_address()", "__builtin_stack_address()"와 같은 함수를 사용하여 알아내

면 다음 스택 프레임 포인터의 주소는 포인터만 증가시킴으로써 쉽게 알 아 낼 수 있게 된다.

위와 같이 스택 내부의 스택 프레임 포인터들의 특성을 이용하여 "dst"의 길이를 추측할 수 있으며 스택 영역의 버퍼 오버플로우 공격 탐지 과정은 다음의 [표-1]과 같다.

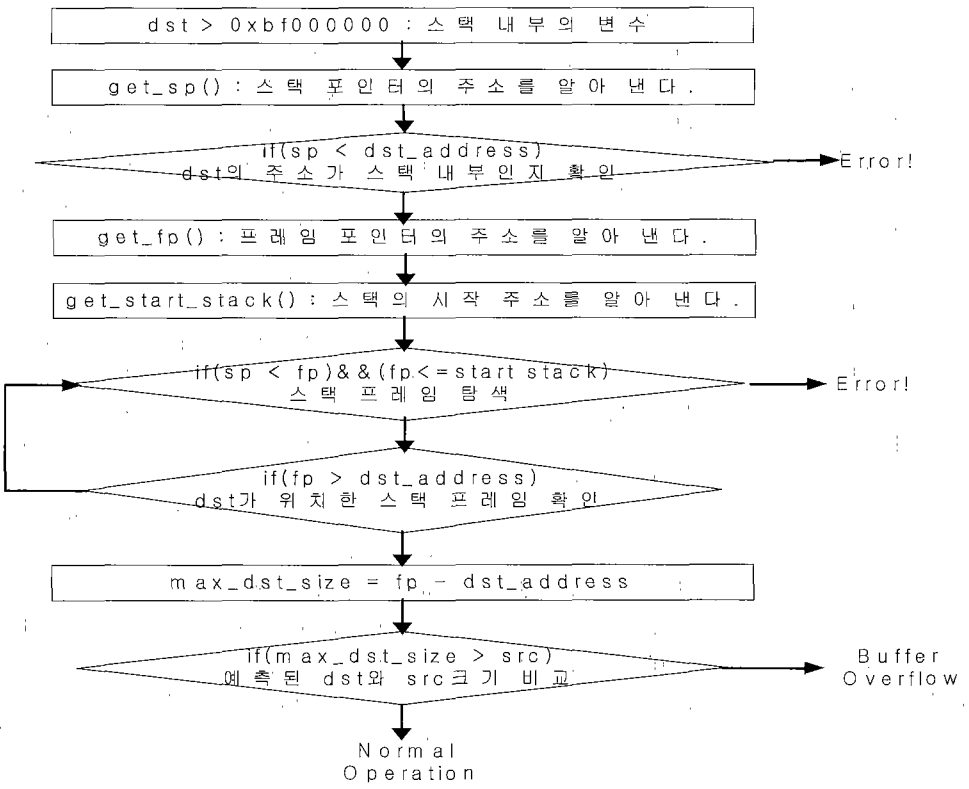
처음으로 "strcpy()"와 같은 함수의 호출이 발생한 경우에 호출된 함수의 버퍼 주소를 사용하여 메모리 영역 중 스택영역인지를 확인한다. 일반적으로 스택 영역은 "0xbf000000"과 같은 형태의 주소를 가지고 있다. 그 후 스택 영역에서의 경계 검사를 위하여 스택 포인터 주소, 프레임 포인터 주소, 스택 시작 주소 값 등을 알아낸다. 그 후 현재 경계 조사를 하고 있는 "dst"가 포함된 스택 프레임 포인터 값을 알아낸 후에 "dst"를 포함하는 스택 프레임 포인터 주소 값에서 "dst"가 위치한 주소 값과의 차가 최대 "dst"에 할당될 수 있는 버퍼의 크기가 된다. 이렇게 알아낸 "dst"의 최대 크기와 "src"의 크기 비교를 통하여 버퍼 오버플로우 공격이 시도되고 있는 지의 여부를 판별 할 수 있다.[28]

3.2 힙 영역에서의 공격 탐지 기법

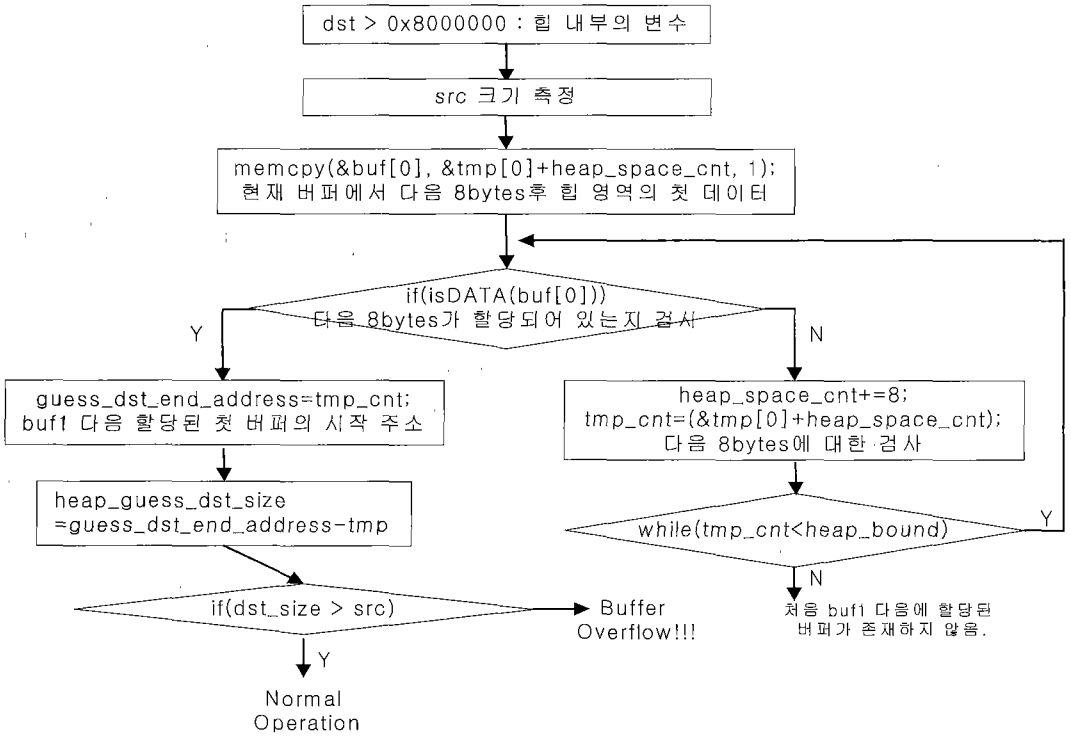
스택 영역에서의 버퍼 오버플로우 공격은 스택 영역 내부에 있는 리턴 주소를 overwrite 함으로써 특정 공격 코드가 있는 주소를 가리키게 하는 것이 가능하였지만, 힙 영역에서는 스택 영역에서의 리턴 주소와 같이 사용할 수 있는 주소 공간이 특별히 존재하지 않는다. 결국 힙 영역에서는 인접한 버퍼사이에서 버퍼 오버플로우를 발생시켜 버퍼에 저장된 데이터를(함수 포인터 등) 바꾸어 다른 프로그램(공격 코드)이 실행 되도록 하는 방법 등을 사용한다.

따라서 본 논문에서는 힙 영역에 존재하는 인접한 버퍼사이의 두 가지 경계 검사 기법을 통하여 힙 영역에서 발생 가능한 버퍼 오버플로우 공격을 탐지하고 방지할 수 방법을 제안한다. 힙 영역은 우선 스택 영역과 다른 몇 가지 특징을 가지고 있다. 우선 스택 영역은 상위 메모리에서 하위 메모리로 영역이

[표-1] 스택 영역에서의 버퍼 오버플로우 탐지기법



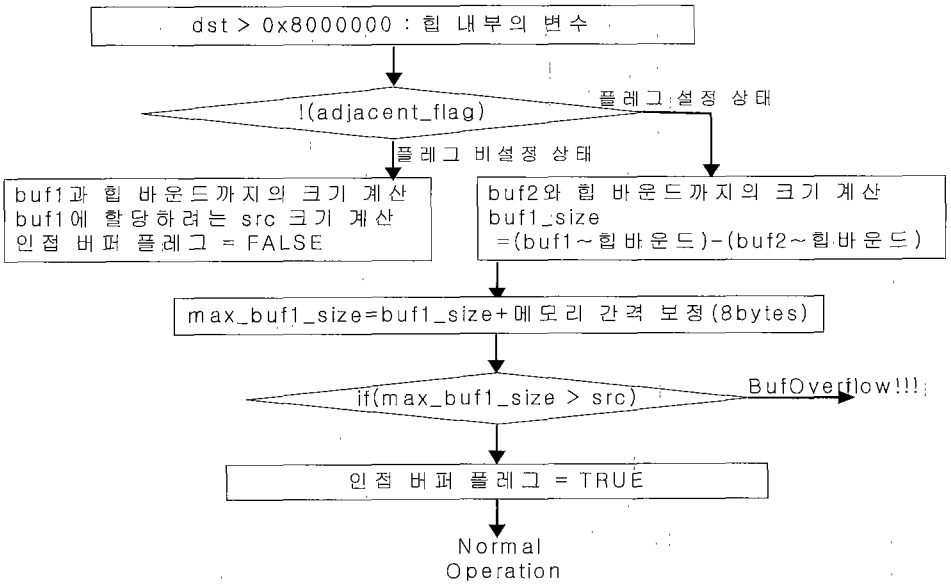
〔표-2〕 힙 영역에서의 버퍼 오버플로우 탐지 기법-1

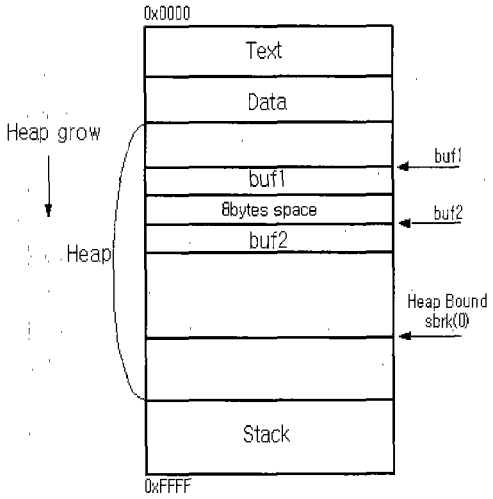


증가하는데 비하여 힙 영역은 하위 메모리에서 상위 메모리로 즉, 스택과 반대방향에서 영역이 증가하는 특징이 있다. 또한 힙 영역에 할당되는 데이터들은 프로그램 실행 시 동적으로 할당되기 때문에 매번 할당되는 버퍼 크기에 대한 정보를 힙 영역에 할당되는 버퍼와 버퍼 사이 8bytes 공간에 기록한다. 〔그림-6〕을 참고하면 힙 영역에 할당된 buf1과 buf2 사이에는 8bytes의 빈 공간이 남아 있으며, 또한 힙에 할당되는 버퍼의 최소 크기는 항상 8bytes단위로 할당되는 것을 알 수 있다. 솔라리스 시스템의 경우에서도 마찬가지로 힙 영역의 최소 크기는 8bytes의 크기를 기준으로 할당된다. 힙 영역의 경계 검사를 위하여 필요한 것은 힙 영역이 가질 수 있는 최대 크기인데 이것은 메모리 영역 중 다음 데이터 세그먼트의 시작을 반환해주는 "sbrk(0)" 함수를 사용하여 알 수 있다. 이와 같은 힙의 특성에 기반 하여 본 논문에서는 힙 영역의 경계 체크를 위한 두 가지 기법을 제안한다. 우선 buf1이 할당되고 그 후 buf2가 할당되는 순간에 인접한 buf1이 오버플로우 되어 buf2의 값이 변경되었는지를 검사하는 방안과 두 번째는 buf2가 할당되고 buf1이 할당되는 순간에 buf1에 할당되는 값이 buf2를 overwrite 하는지를 검사하는 방안이다.

〔표-2〕, 〔표-3〕과 같이 힙 영역에서 인접하게 할당되는 버퍼의 경계 검사를 위한 첫 단계는 함수의 호출이 메모리의 어느 영역에서 이루어졌는지를 판단하는 것이다. 이러한 판단을 위해서 스택 영역에서의 버퍼 오버플로우 공격 탐지와 마찬가지로 버퍼의 주소 값의 범위를 사용한다. 기본적으로 힙 영역은 "0x8000000"이상의 주소를 사용한다. 결국 현재 버퍼에 어떤 값을 할당하기 원하는 "dst"의 포인터 값이 위의 주소 조건을 만족한다면 현재 사용하려는 버퍼는 힙 영역에 할당된 버퍼인 것을 알 수 있고 그 후 힙 영역에서의 버퍼 오버플로우 공격을 방지하기 위한 첫 번째 방안에서는 buf2까지의 힙 영역 사이를 8bytes로 검색하여 할당된 버퍼가 이미 buf1과 buf2 사이에 존재하는지의 여부와 그렇지 않다면 buf1과 buf2사이의 크기를 계산하여 buf1이 할당될 수 있는 최대 영역을 구한다. 이렇게 구한 buf1과 buf2의 최대 크기를 사용하여 buf1에 할당되는 "src"가 "dst"를 넘지 않도록 조절한다.〔28〕

〔표-3〕 힙 영역에서의 버퍼 오버플로우 탐지 기법-2





〔그림-6〕 힙 영역에서의 인접 버퍼 할당

또한 〔표-3〕의 힙 영역에서의 버퍼 오버플로우를 방지하기 위한 두 번째 경계 검사 방안에서는 첫 번째 방안과 마찬가지로 우선 현재 할당을 요청하는 buf가 힙 영역 내부에 존재하는지의 여부를 우선 검사한 후에 인접하여 버퍼가 할당되었을 경우를 검사하기 위한 인접 버퍼 플래그를 체크한다. 이러한 인접 유무 검사 플래그가 설정되었는지를 조사한다. 만일 연속하여 "strcpy()", "memcpy()"등의 버퍼에 값을 할당하는 함수가 호출되는 경우에는 인접 유무 검사 플래그가 설정되어 힙 영역에서 인접한 공간에 두 번째 "buf"에 대해 할당이 요구되는 것을 알 수 있다. 그 후 "sbrk()" 함수를 사용하여 힙 영역의 최대 한계 크기를 알아낸 값을 사용하여 힙 영역의 최대 한계 크기와 목적 주소까지의 차를 계산하여 인접한 "buf1"에 할당 될 수 있는 최대 크기를 계산할 수 있다.

따라서 버퍼에 계산된 최대 크기의 할당 값 보다 큰 값이 요청되면 버퍼 오버플로우 공격이라고 탐지하는 것이 가능하게 된다.

IV. 결 론

본 논문에서는 리눅스 운영체제를 기반으로 하는 시스템에서의 버퍼 오버플로우 공격 기법에 대하여 메모리 내에서의 경계 검사 기법을 통한 대응 방안을 제시하였다. 버퍼 오버플로우는 현재 가장 많이 사용되는 시스템 침해 유형 중의 하나이며 특히 리눅스 운영체제를 기반으로 하는 시스템은 운영체제 및 관련 프로그램들의 소스가 공개되어 시스템의 보안에 있어서 큰 문제점으로 지적되고 있다.

따라서 본 논문에서는 버퍼 오버플로우 공격 기법 중에서도 가장 많은 사용 빈도를 보이는 스택에서의 경계 침범을 이용한 공격과 스택에 대한 대응 도구들의 연구로 인해 스택 외에 힙을 이용하는 경우가

시도되는 점에 착안하여 힙 영역에서의 경계 검사 기법을 제안한다.

제안한 스택에서의 경계 검사 기법은 현재 할당된 버퍼가 존재하는 스택 프레임에서 버퍼에 할당 될 수 있는 최대의 버퍼 크기를 계산함으로써 버퍼 오버플로우를 통한 리턴 주소의 변경을 방지하며 힙 영역에서의 경계 검사 기법은 힙 영역은 스택과 달리 리턴 주소와 같은 중요 변수를 값으로 가지고 있지 않기 때문에 인접 버퍼 사이의 오버플로우를 방지하기 위한 인접 버퍼가 할당될 수 있는 최대 버퍼 크기를 계산하는 기법을 사용하였다.

추후에는 제안 기법에 대한 리눅스 시스템에서의 구현을 통해 시스템 성능 측면과 실제 공격 탐지 측면의 검증이 필요하며 본 논문에서 제안된 스택 및 힙 영역에서의 경계 검사 기법뿐만이 아니라 기존에 존재하는 유사 버퍼 오버플로우 공격 및 새롭게 사용되고 있는 포맷 스트링 공격에 대한 방지 기법들에 대한 연구가 필요하다.

참 고 문 헌

[1] Aleph One, *Smashing The Stack For Fun And Profit*. Phrack volume 7, issue 49. 1996.(http://www.phrack.com/search.phtml?view&article=p49-14.[23.3.01])

[2] *Apache HTTP Server version 2.0 Documentation*. A pache HTTP server Documentation project. (http://httpd.apache.org/docs-2.0 [26.4.2001])

[3] Baratloo,A., T sai, t.,Singh, N. "Libsafe: *Protecting Critical Elements of Stacks*", Bell Labs, Lucent Technologies. 1999. (http://www.avayalabs.comproject/libsafe/index.html[26.3.01])

[4] Bullba and Kil3r, *"Bypassing StackGuard and StackShield"*, Pharck volume 10, issue 56. 2000. (http://www.pharck.com/search.phtml?view&article=p56-5. [17.4.01])

[5] *CERT Advisory CA-1997-05, "MIME Conversion Buffer Overflow in Sendmail Versions 8.8.3 and 8.8.4"*, Carnegie Mellon University. 1997. (http://www.cert.org/advisories/CA_1997-05.html [23.3.01])

[6] *CERT Advisory CA-2001-02 "Multiple Vulnerabilities in BIND"*, Carnegie Mellon University. 2001. (http://www.cert.org/advisories/CA_2001-02.html [23.3.01])

[7] Cowan, C., Pu, C., D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. *"Automatic Detection and Prevention of Buffer_Overflow Attacks"*, 7th "USENIX Security Symposium", San Antonio, TX, January 1998.

[8] Cowan, C., Wagle, F., Pu, C ., Beattlie, S., Walpole, J., *"Buffer overflows: attacks and defenses for th vulnerability of the decade."*, DARPA information Survivability Conference and Exposition, 2000. DISCEX'00. proceedings, Volume: 2, 1999.

[9] *"klog". The Frame Pointer Overwrite*. Phrack Volume 9, issue 55. 1999 (http://www.phrack.com/search.phtml?view&article=p55-8. [23.3.0.11])

[10] Newsham, Tim. *Format String Attacks*. Guardent Inc. 2000. (http://www.guardent.com/rd_whtpt.html [20.3.2001])

[11] Red Hat Linux Security Advisory RHSA-2001: 047-05. Red Hat, Inc. 2001. (http://www.redhat.com/support/errata/RHSA_2001-047.html [8.5.2001])

[12] twitch, *Taking advantage of non-terminated adjacent memory spaces*. Phrack volume 10, issue 56. (http://www.phrack.com/search.phtml?view&article=p5-14. [23.3.01])

[13] Wheeler, D. *Secure Programming for Linux and Unix HOWTO. 2000*. (http://www.dwhelet.com/secure-programs/ [24.05.2001])

[14] CERT coordination center. http://www.cert.org.

[15] Crispin Cowan, Calton Pu, Dave Maier, Heather Hilton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang., *"StackGuard: automatic adaptive detection and prevention*

of buffer-overflow attacks.", In "Proceedings of the 7th USENIX Security Conference". 1998.
[16] Ville Alkkiomaki, "Stack Overwriting attacks and defences in unix environment", http://www.ecurityfocus.com
[17] Openwall Project. *Linux kernel patch from the openwall project.* http://www.openwall.com/linux.
[18] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. *"A first step towards automated detection of buffer overrun vulnerabilities"*, In "Proceedings 7th Network and Distributed System Security Symposium" (to appear), February 2000.
[19] Rafel Wojtczuk. *"Defeating solar designer non-executable stack"*, http://geek-girl.com/bugtraq, January 1998.
[20] Mudge. *How to Write Buffer Overflows.* http://www.10pht.com/advisories/buferro.html
[21] Alfred V. Aho, R. Hopcroft, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass., 1985.
[22] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. *Protecting Systems from Stack Smashing Attacks with StackGuard.* In "Linux Expo", Raleigh, NC, May 1999.
[23] Crispin Cowan, Tim Chen, Calton Pu, and Perry Wagle. *StackGuard 1.1: Stack Smashing Protection for Shared Libraries.* In "IEEE Symposium on Security and Privacy", Oakland, CA, May 1998. Brief presentation and poster session.
[24] Solar Designer. *Non-Executable User Stack.* http://www.openwall.co/linux/.
[25] Chris Evans. *Nasty security hole in lprm.* Bugtraq mailing list, http://geek-girl.com/bugtraq/, April 19 1998.
[26] Nothan P. Smith. *Stack Smashing vulnerabilities in the UNIX Operating System* http://millcomm.com/nate/machines/security/stack-smashing/nate-buffer.ps, 1997.
[27] Rafel Wojtczuk. *Defeating Solar Designer Non-Executable Stack Patch.* Bugtraq mailing list, http://geek-girl.com/bugtraq/, January 30 1998.
[28] TIS Committee, May, 1995, Tool Interface Standard(TIS) Executable and Linking Format(ELF) Specification V.1.2

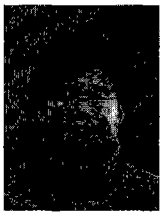
〈者 著 紹 介〉

손 태 식(Tae-Shik Sohn)
학생회원
2000년 2월 : 아주대학교 정보 및 컴퓨터 공학부 졸업(학사)
2000년 3월~현재 : 아주대학교 정보통신전문대학원 정보통신공학과 석사과정
관심분야 : 네트워크·시스템보안, 인터넷프로토콜 보안

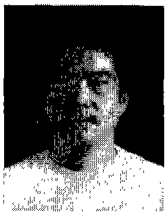
서 정 택(Jung-Taek Seo)
정회원
1999년 2월 : 충주대학교 컴퓨터공학과 졸업 (학사)
2001년 2월 : 아주대학교 대학원 컴퓨터공학과 졸업 (석사)
2000년 11월 ~ 현재 : 국가보안기술연구소 연구원
관심분야 : 정보전, 시스템 및 네트워크 보안, 시스템 평가

은 유 진(You-Jin Eun)
1995년 : 아주대학교 컴퓨터 공학과 졸업(학사)
1997년 2월 : 아주대학교 대학원 컴퓨터 공학과 졸업(석사)
1996년 12월~2000년 2월 : 한국정보보호센터 기술 본부 주임 연구원
1997년 10월~2000년 2월 : 한국정보통신기술협회 정보보호기술 연구위원
2000년 3월~현재 : 아주대학교 컴퓨터 공학과 박사과정

2000년 3월~현재 : (주)시큐브 부사장
관심분야 : 보안 커널, IDS, PKI



장 준 교(Joon-Kyo Jang)
1997년 2월 : 아주대학교 컴퓨터공학과 졸업
1999년 2월 : 아주대학교 컴퓨터공학과 석사 졸업
1999년 3월 ~ 2000년 6월 : 아주대학교 컴퓨터공학과 박사과정 수료
2000년 7월 ~ 현재 : (주)케이사인 무선PKI연구개발팀/팀장
관심분야 : 유무선PKI, 네트워크보안, 무선보안, 암호기술 응용



이 철 원(Cheol-Won Lee)
1987년 2월 : 충남대학교 수학과 졸업 (학사)
1989년 2월 : 중앙대학교 대학원 전자계산학과 (석사)
1989년~1996년 : 한국전자통신연구원 선임연구원
1996년~2000년 : 한국정보보호센터 선임연구원/통신모델링 과제책임자
2000년~현재 : ETRI 부설 국가보안기술연구소 팀장
관심분야 : 컴퓨터 및 네트워크 보안, 정보통신 기반보호, 정보보호시스템 평가기준

김 동 규(Dong-Kyoo Kim)
종신회원
1973년 : 서울대학교 공과대학 응용수학과 졸업(학사)
1979년 : 서울대학교 자연과학대학원 전자계산학 졸업(석사)
1984년 : Kansas State University 전자계산학(박사)
1979년~현재 : 아주대학교 정보 및 컴퓨터공학부 교수
IEEE 802.4,802.6,802.10 Working Group Member, Asiacrypt '96 조직위원회 위원장, 건설교통부 항공교통관제소 신항공 교통관제 시스템 평가위원회 위원, 한국과학기술연구소 연구원, 한국통신학회 상임이사, 한국통신정보보호학회 부회장 역임
관심분야 : 컴퓨터 통신, 정보보호, 프로토콜 엔지니어링