EC Cevelop 발표 소논문

BufferOverflow 공격을 이용한 C언어 함수의 취약점 탐구 및 대안 제시

studying vulnerabilities in C language functions using BufferOverflow attacks and presenting alternatives

서울과학기술대

18101290

황인서

2018년 6월

목 차

표 차례

그림 차례

초 록

BufferOverflow 공격을 이용한 C언어 함수의 취약점 탐구 및 대안 제시

서울과학기술대
18101290
황인서

본 연구는 BufferOverflow 공격의 개념을 메모리 구조, 레지스터, 어셈블리어의 개념을 통해 프로세스를 실행 과정을 분석함으로써 이해한다. 또한 리눅스 환경에서 BufferOverflow 공격을 실습하기 위해 디버깅 방법과 프로세스 분석 방법을 배운다. BufferOverflow 공격의 대처 방안으로 운영체제 차원, 프로그래밍 차원에서 알아보았다. 하지만 실질적으로 BufferOverflow 공격의 대상이 되는 임베디드 시스템은 운영체제 차원의 기법을 적용하기 힘들기 때문에 BufferOverflow를 막는 코드 작성법을 익히는 것을 강조하였다.

Ⅰ. 서론

1. 연구 배경

Visual Studio을 이용해 C언어 프로그래밍을 하는 도중 scanf 함수에 취약점이 있다며 쓰지 못하게 되었다. 나는 이 함수에 어떤 취약점이 있고 어떻게 악용될 수 있는 지 궁금했고, 조사 결과 BufferOverflow 공격이란 것을 처음 알게 되었다.

최근 IOT의 확산에 힘입어 임베디드 시스템이 폭발적으로 증가하는 추세이다. 임베디드 시스템은 그것이 실행되는 장비의 성능 등의 제약 때문에 C언어로 이루어져 있다. 현재 IOT가 적용된 사물에는 사용자의 민감한 정보가 포함돼 있거나, 사용자의 생활에 직접적으로 연관된 것들이 많다. 따라서 우리는 C언어 함수에서 발생할 수 있는 취약점인 BufferOverflow을 정확히 이해할 필요가 있고 BufferOverflow 공격의 대처 방안을 익힐 필요가 있으며, 따라서 본 연구를 진행하게 되었다.

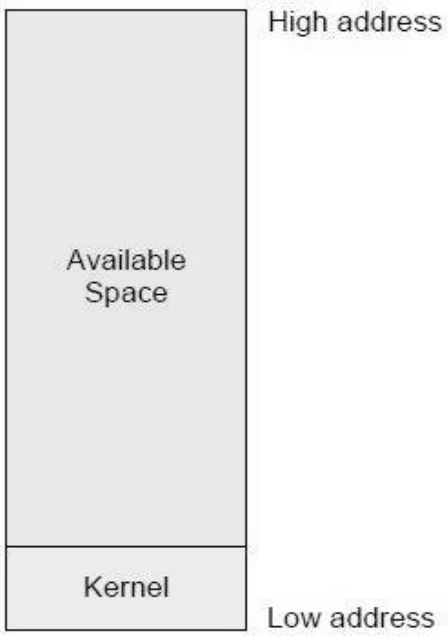
2. 연구 목적

BufferOverflow 공격의 원리를 메모리 구조와 레지스터의 개념을 통해 이해하고 어셈블리 언어를 이용한 코드 분석을 통해 그 과정을 자세히 살펴본다. 또한 리눅스 환경에서 여러 유용한 기능을 통해 취약한 코드를 만들어 보고 분석하고 실습해보며 BufferOverflow 공격의 개념을 정확히 이해한다. 나아가 어떤 함수가 BufferOverflow 공격에 취약하고 그의 대처 방안은 무엇인지 살펴봄으로써 BufferOverflow 공격의 대처 능력을 기르는 것이 목표이다.

Ⅱ. 이론적 배경

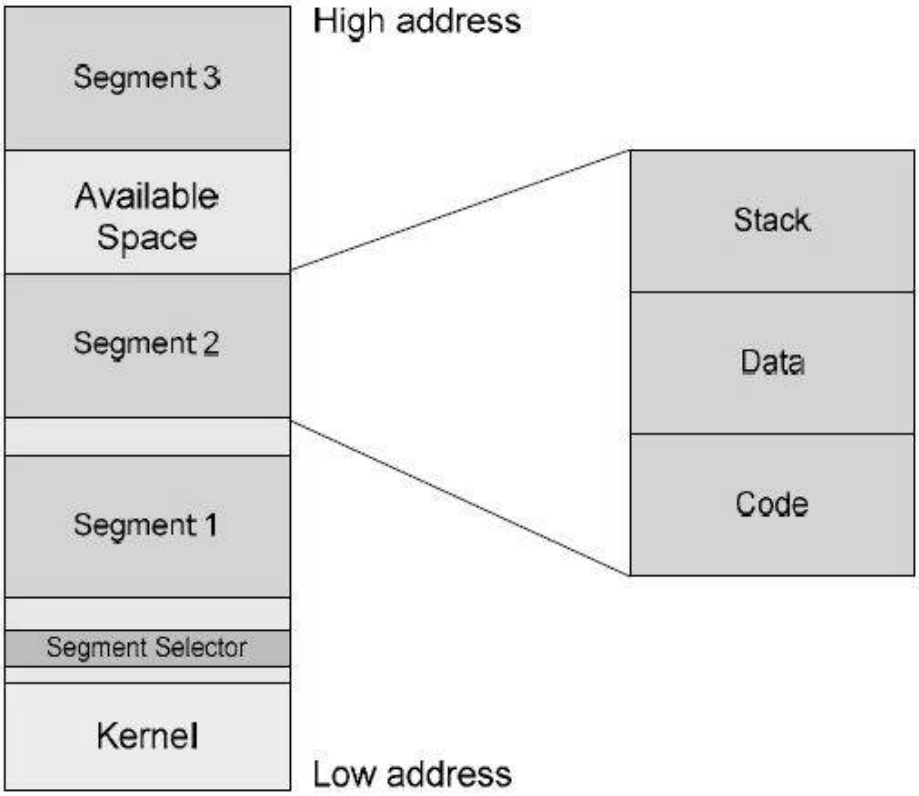
1. 8086 메모리 구조와 프로세스의 구조

(1) 메모리 구조



〈그림 2-1. 8086 memory structure〉

8086 시스템의 기본적인 메모리 구조는 〈그림 2-1〉과 같다. 시스템이 초기화되기 시작하면 시스템은 커널을 메모리에 적재시키고 가용 메모리 영역을 할당하게 된다. 커널에는 시스템의 운영에 필요한 기본적인 명령어가 들어 있다. 운영체제는 프로세스를 segment 라는 단위로 묶어서 가용 메모리 영역에 저장한다. 그 구조는 〈그림 2-2〉와 같다.

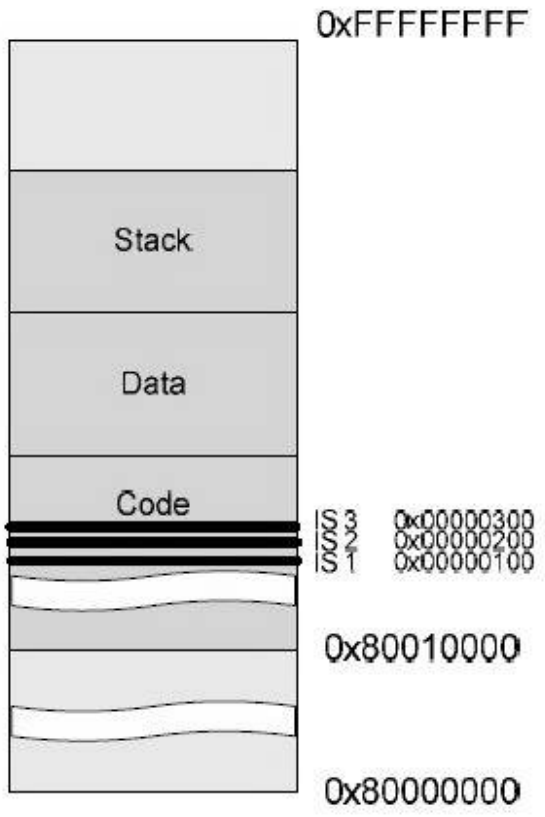


〈그림 2-2. segmented memory model〉

〈그림2-2〉의 메모리에는 여러 개의 프로세스가 적재되어 있으며 이는 멀티태스킹을 나타낸다. 하나의 segment는 〈그림 2-2〉의 오른편에 나와 있는 구조를 가지고 있다. 각각을 code segment, data segment, stack segment라고 한다. 이 외에도 heap segment, bss segment, text segment 등이 있지만 이번 연구에선 다루지 않는다.

(2) segment의 구조

code segment에는 시스템이 알아들을 수 있는 명령어 즉 instruction이 들어 있다. 이것은 컴파일러가 만든 기계어 코드이다. 프로세스가 실행되면서 수많은 instruction 사의의 분기, 점프, 시스템 호출 등이 일어나는데, 그것을 수행하기 위해 메모리상의 특정 위치에 있는 명령을 지정해 주어야 한다. 이를 위해 segment는 logical address와 physical address를 사용한다. logical address는 명령어의 segment의 시작 주소로부터의 논리적인 주소라고 할 수 있고, physical address는 그 명령어의 실제 물리적인 주소이다. 따라서 physical address는 segment의 시작주소에 logical address를 더한 것과 같고 이때 segment의 시작주소를 offset이라 하며 이는 segment selector부터 얻는다.



〈그림 2-3. address〉

〈그림 2-3〉에서 보는 바와 같이 segment가 실제로 위치하고 있는 메모리상의 주소를 0x80010000이라고 가정하자. code segment 내에 들어 있는 하나의 instruction IS 1을 가리키는 주소는 0x00000100이라. 이것은 logical address이고 이 instruction의 실제 메모리상의 주소는 segment offset인 0x80010000과 segment 내의 수조 0x00000100을 더한 0x80010100이 된다. 따라서 이 segment가 메모리상의 어느 위치에 있더라도 segment selector가 segment의 offset을 알아내어 해당 instruction의 정확한 위치를 찾아낼 수 있게 된다.

data segment에는 프로그램이 실행 시에 사용되는 데이터가 들어간다. 여기서 말하는 데이터는 전역변수와 정적 변수를 말한다.

stack segment에는 현재 수행되고 있는 handler, task, program이 사용하는 데이터 영역으로 우리가 사용하는 버퍼가 이 stack segment에 자리 잡게 된다. 또한 지역변수와 매개변수가 들어가는 곳이다.

스택은 처음 생성될 때 필요한 크기만큼 만들어지고 프로세스의 명령에 의해 데이터를 저장 해 나가는 과정을 거치게 되는데 이것은 stack pointer(SP)라고 하는 레지스터를 통해 이루어진다. SP는 스택의 맨 꼭대기를 가리키고 있으며 PUSH과 POP등의 instruction에 의해 데이터를 저장하거나 읽어 들인다.

2. 8086 레지스터 구조

지금까지 segment의 구조를 알아보았다. CPU가 프로세스를 실행하기 위해서는 프로세스의 instruction을 CPU에 적재시켜야 할 것이다. 흩어져 있는 여러 명령어 집합과 데이터를 적절하게 집어내고 처리하기 위해서는 여러 가지의 저장 공간이 필요하다. 또한 CPU가 빨리 읽고 쓰기를 해야 하는 데이터들이므로 CPU 내부에 존재하는 메모리를 사용하게 되는 데, 이런 저장 공간을 레지스터라고 한다.

레지스터는 그 목적에 따라 **범용 레지스터**(General-Purpose register), **세그먼트 레지스터**(Segment register), **플래그 레지스터**(Program status and control register), **인스트럭션 포인터**(Instruction pointer)로 구성된다. 이번 연구에선 범용 레지스터와 인스트럭션 포인터 개념만 사용할 예정이다.

범용 레지스터는 논리 연산, 수리 연산에 사용되는 피연산자, 주소를 계산하는데 사용되는 피연산자, 그리고 메모리 포인터가 저장되는 레지스터다.
세그먼트 레지스터는 code segment, data segment, stack segment를 가리키는 주소가 들어가 있는 레지스터다.
플래그 레지스터는 프로그램의 현재 상태나 조건 등을 검사하는 데 사용되는 플래그들이 있는 레지스터다.
인스트럭션 포인터는 다음 수행해야 하는 명령(instruction)이 있는 메모리상의 주소가 들어가 있는 레지스터다.

(1)범용 레지스터

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

〈그림 2-4. 범용 레지스터〉

범용 레지스터는 프로그래머가 임의로 조작할 수 있게 허용되어 있는 레지스터다. 일종의 4개의 32bit 변수라고 생각하면 된다. 예전의 16bit 시절에는 각 레지스터를 AX, BX, CX, DX등으로 불렀지만 32bit 시스템으로 전환되면서 E(Extended)가 앞에 붙어 EAX,

EBX, ECX, EDX 등으로 불린다. AX, BX, CX. DX는 또한 상위부분 AH, AL...(생략)으로 이루어져 있다. EAX, EBX, ECX, EDX 레지스터 들은 프로그래머의 필요에 따라 아무렇게나 사용해도 되지만 레지스터의 목적에 따라, 또한 나중에 코드의 이해를 돕기 위해 그 목적에 맞게 사용해 주는 것이 좋다. 컴파일러 또한 이러한 목적에 맞게 사용되고 있다. 각 레지스터의 목적은 〈표 1〉 과 같다.

〈표 2-1. 범용 레지스터의 목적〉

레지스터	목적
EAX (Extended Accumulator Register)	곱셈과 나눗셈 명령에서 사용, 함수의 반환 값을 저장.
EBX (Extended Base Register)	ESI나 EDI와 결합해 인덱스에 사용된다.
ECX (Extended Counter Register)	반복 명령어를 사용할 때 반복 카운터를 저장한다. ECX 레지스터에 반복할 횟수를 지정해 놓고 반복 작업을 수행한다.
EDX (Extended Data Register)	EAX와 같이 사용되며 부호 확장 명령 등에 활용된다.
ESI (Extended Source Index)	데이터를 복사하거나 조작할 때 소스 데이터 주소가 저장된다. ESI 레지스터가 가리키는 주소에 있는 데이터를 EDI 레지스터가 가리키는 주소로 복사하는 용도로 많이 사용된다.
EDI (Extended Destination Index)	복사 작업을 할 때 목적지 주소가 저장된다. 주로 ESI 레지스터가 가리키는 주소의 데이터가 복사된다.
EBP (Extended Base Pointer)	하나의 스택 프레임의 시작 주소가 저장된다. 현재 사용되는 스택 프레임이 살아있는 동안 EBP의 값은 변하지 않는다. 현재 사용한 스택 프레임이 사라지면 이전에 사용되던 스택 프레임을 가리키게 된다.
ESP (Extended Stack Pointer)	하나의 스택 프레임의 끝 지점 주소가 저장된다. PUSH, POP 명령어에 따라서 ESP의 값이 4바이트씩 변한다.

(2)Instruction pointer

Instruction pointer 레지스터는 다음에 실행할 명령어가 저장된 메모리 주소가 저장된다. 현재 명령어를 모두 실행한 다음에 EIP 레지스터에 저장된 주소에 있는 명령어를 실행한다. 실행 전에 EIP 레지스터에는 다음 실행해야 할 명령어가 있는 주소 값이 저장된다.

3. 어셈블리어

code segment 내부의 instruction은 모두 기계어로 이루어져 있다. 기계어는 숫자들의 규칙조합임으로 프로그래밍을 하기에 상당히 난해하다. 그래서 이 기계 명령어를 좀 더 이해하기 쉬운 기호 코드를 나타낸 것 (기계어와 1:1로 대응된 명령을 기술하는 언어)이 어셈블리어이다. 어셈블리 언어는 그 코드가 어떤 일을 할지를 추상적이 아닌, 직접적으로 보여준다. 논리상의 오류나 수행 속도, 수행 과정을 명확히 해준다는 점에서 직관적인 언어이다. 어셈블리 언어를 사용하면 메모리에 대한 이해도도 높아진다. 어셈블리를 익히고, 배우는 데 있어서는 여러 가지 목적이 있을 수 있다. 컴퓨터 시스템 구조를 좀 더 깊게 이해하고, 메모리상의 데이터나 I/O기기를 직접 엑세스하는 등의 고급언어에서는 할 수 없는 조작을 위해서이다. 프로그램의 최적화 및 리버스 엔지니어링을 위해서도 필요하다.

BufferOverflow 공격을 하기 위해서는 code segment 내부의 instruction을 정확히 분석해야 하는데 기계어로 이루어져 있는 instruction을 어셈블리어로 변환하여 분석하게 된다. 따라서 어셈블리어 명령을 충분히 이해 할 필요가 있다.

(1)어셈블리 명령어의 구성

〈표 2-2. 어셈블리 명령어의 구성〉

구성	Label	명령어	제 1오퍼랜드	제 2 오퍼랜드	설명문
예시	L1	mov	%eax	%ebx	comment

어셈블리 명령어의 문법은 Intel 방식과 AT&T 방식이 있다. 필자의 Linux 환경에서는 AT&T 방식이 사용되고 있으며 이번 연구에서는 이 방식만 다루기로 한다.

명령어 다음에 오는 레지스터 이름이나 값들은 operand라고 한다. mov %eax, %ebx에서 %eax를 제 1 오퍼랜드, %ebx를 제 2 오퍼랜드라고 한다. mov %eax, %ebx는 ebx에 eax를 복사하여 저장한다는 의미이다. 이 때 제 1 오퍼랜드에는 특정한 값 또는 데이터의 주소에서 데이터를 읽어오는 방식으로 사용 될 수 있다. 오퍼랜드는 하나만 필요 할 수도 있다. 주로 제 2 오퍼랜드가 명령어의 대상이 된다. 다음 〈표2-4〉은 주요 어셈블리 명령어들이다. mov 명령어는 다양하게 활용될 수 있기 때문에 따로 정리하도록 한다.

(2)어셈블리 명령어의 종류

〈표 2-3. mov 명령어의 활용〉

지정 방식	예시	설명
즉시 지정 방식	mov $0x1, %eax	eax에 (16진수)1을 넣는다.

레지스터 지정 방식	mov %esp, %ebp	ebp에 esp의 값을 넣는다.
직접 주소 지정방식	mov %eax, %0x80482f2	주소 0x80482f2에 있는 값을 eax에 할당한다.
레지스터 간점 주소 지정 방식	mov (%ebx), %eax	ebx의 값을 (간접적으로)주소로 하여 eax에 할당
베이스 상대 주소 지정 방식	mov 0x4(%esi), %eax	esi 레지스터에서 4바이트를 더한 주소의 값을 eax 레지스터에 할당한다.

〈표 2-4. 어셈블리 명령어 종류〉

명령어	예시	설명	분류
push	push %eax	eax의 값을 스택 상위에 저장. 스택 포인터(esp)도 워드 크기만큼 증가한다.	스택 조작
pop	pop %eax	스택 가장 상위에 있는 값(워드만큼)을 꺼내서 eax에 저장 esp는 바로 전의 데이터를 가리킨다.	스택 조작
lea	lea (%esi), %ecx	%esi의 주소 값을 %ecx에 옮긴다.	주소이동
inc	inc %eax	%eax의 값을 1 증가시킨다.	데이터 조작
dec	dec %eax	%eax의 값을 1 감소시킨다.	데이터 조작
add	add %eax, %ebx	레지스터나 메모리의 값을 덧셈할 때 쓰인다.	논리, 연산
sub	sub $0x8, %esp	레지스터나 메모리의 값을 뺄셈할 때 쓰인다.	논리, 연산
call	call proc	프로시저를 호출한다.	프로시저
ret	ret	호출했던 지점의 다음 지점으로 이동한다.	프로시저
leave	leave	함수의 에필로그 작업을 해준다.	프로시저
cmp	cmp %eax, %ebx	레지스터와 레지스터 값을 비교	비교
jmp	jmp proc	특정한 곳으로 분기	분기
int	int $0x80	OS에 할당된 인터럽트 영역을 system call	인터럽트
nop	nop	아무 동작도 하지 않는다.	

4. 프로세스 실행 과정

이번 장에서는 프로세스가 메모리에 적재되어 메모리와 레지스터가 어떻게 변화하는지, 명령어가 실행하는 방식 등을 배울 것이다. 이를 위하여 간단한 프로그램으로 예를 들도록 하겠다. 아래의 프로그램을 보자.

```
void function(int a, int b, int c)
{
        char buffer1[150];
        char buffer2[150];
}

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

〈그림 2-5. simple.c〉

위 프로그램은 별 동작도 하지 않는 아주 간단한 프로그램이다. 컴파일러를 통해 어셈블리 코드를 얻으면 다음과 같다.

```
0x804831f <main+35>:    nop
0x804831e <main+34>:    ret
0x804831d <main+33>:    leave
0x804831a <main+30>:    add     $0x10, %esp
0x8048315 <main+25>:    call    0x80482f4
0x8048313 <main+23>:    push    $0x1
0x8048311 <main+21>:    push    $0x2
0x804830f <main+19>:    push    $0x3
0x804830c <main+16>:    sub     %0x4, %esp
0x804830a <main+14>:    sub     %eax, %esp
0x8048305 <main+9>:     mov     $0x0, %eax
0x8048302 <main+6>:     and     $0xfffffff0, %esp
0x80482ff <main+3>:     sub     $0x8, %esp
0x80482fd <main+1>:     mov     %esp, %ebp
0x80482fc <main>:       push    %ebp
0x80482fd <function+7>: ret
0x80482fa <function+6>: leave
0x80482f7 <function+3>: sub     $0x28, %esp
0x80482f5 <function+1>: mov     %esp, %ebp
0x80482f4 <function>:   push    %ebp
```

〈그림 2-6. simple.asm〉

이제 이 프로그램이 실행되는 과정을 하나하나 분석해 보도록 하자

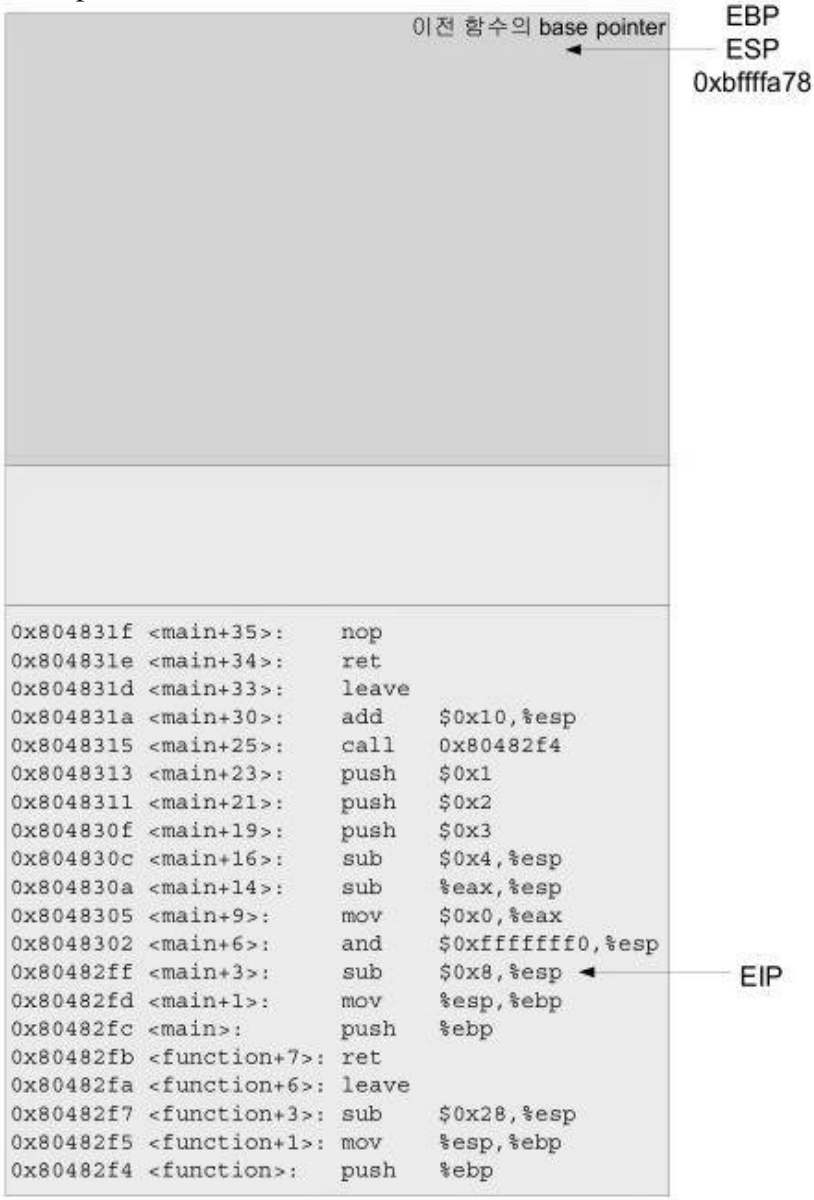
〈그림 2-7. simple.c 프로그램이 실행 될 때의 segment의 모습〉

(1)Step 1

〈그림 2-8. step 1〉

좌측의 그림과 같이 프로그램이 시작될 때 EIP는 main()함수의 시작점을, ESP는 스택의 맨 꼭대기를 가리키고 있다. ESP는 높은 주소로부터 낮은 주소로 향해가고 데이터 또한 그 방향으로 쌓인다. EBP를 저장하고 EBP에 ESP를 저장하는 이유는 이전에 수행하던 함수의 데이터를 보존하기 위해서이다. 그래서 함수가 시작될 때에는 이렇게 stack pointer와 base pointer를 새로 지정하는데 이러한 과정을 **함수 프롤로그 과정**이라고 한다.

(2)step 2



〈그림 2-9. step 2〉

push %ebp
를 수행하여 이전 함수의 base pointer를 저장하면 stack pointer는 4바이트 아래인 0xbffffa78을 가리키게 된다.

mov %esp, %ebp
를 수행하여 ESP 값을 EBP에 복사한다. 이렇게 함으로써 함수의 base pointer와 stack pointer가 같은 지점을 가리키게 된다.

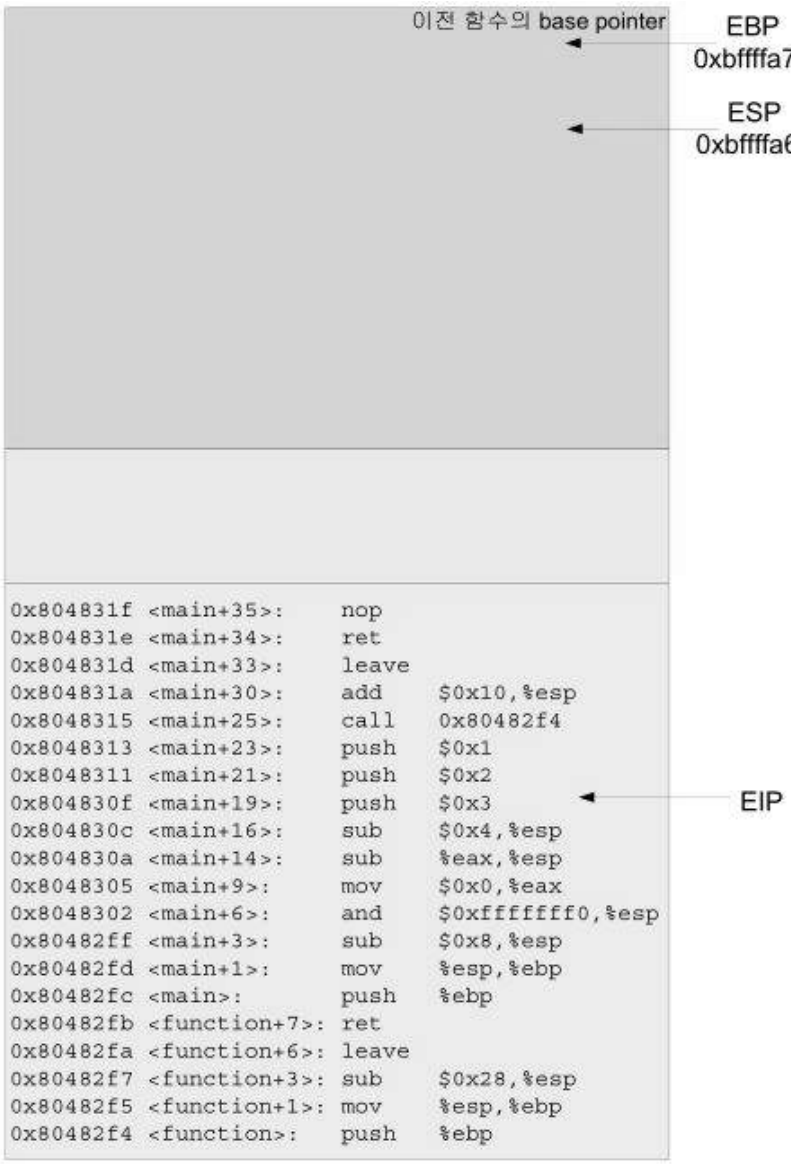
sub $0x8, %esp
는 ESP에서 8을 빼는 명령어이다. 따라서 ESP는 8바이트 아래 지점을 가리키게 되고 스택에 8바이트의 공간이 생긴다. 이것을 **스택이 8바이트 확장되었다**고 말한다. 이 명령이 수행되고 나면 ESP에는 0xbffffa70이 들어간다.

and $0xfffffff0, %esp
은 ESP와 11111111 11111111 11111111 11110000 과 AND 연산을 한다. 이것은 ESP의 주소 값의 맨 뒤 4bit를 0으로 만들기 위함이다. 별 의미 없는 명령어이다.

mov $0x0, %esp sub %eax, %esp
EAX 레지스터에 0을 넣고 ESP에 들어있는 값에서 EAX에 들어있는 값을 뺀다. EAX가 0이므로 별 의미 없는 명령어이다.

sub $0x4, %esp
스택을 4바이트 확장하였다. 따라서 ESP에 들어있는 값은 0xbffffa6c가 된다.

(3)step 3



〈그림 2-10. step3〉

지금까지 명령을 수행한 모습은 좌측 그림과 같다 ESP는 12 바이트 이동하였다.

push $0x03
push $0x02
push $0x01

이것은 function(1,2,3)을 수행하기 위해 인자값 1, ,2, 3을 차례로 넣어주는 것이다. 순서가 3, 2, 1 이유는 스택에 꺼낼 때는 거꾸로 나오기 때문에 그렇다.

call 0x80482f4

은 0x80482f4에 있는 명령을 수행하라는 것이다. 그 주소에는 function함수가 자리 잡은 것을 알 수 있다.

call 명령은 함수를 호출할 때 사용되는 명령으로 함수 실행이 끝난 다음 다시 이 후 명령을 계속 수행할 수 있도록 이 후 명령이 있는 주소를 스택에 넣은 다음 EIP에 함수의 시작 지점의 주소를 넣는다. 즉 add $0x10, %esp 명령이 있는 주소인 0x804831a의 값을 스택에 저장해둔다.

따라서 함수 수행이 끝나고 나면 이제 어디에 있는 명령을 수행해야 하는지를 스택에서 POP하여 알 수 있게 되는 것이다. 이것이 BufferOverflow 공격에서 가장 중요한 return address이다.

이제 EIP에는 function 함수가 있는 0x80482f4가 들어가게 된다.

(4)step 4

〈그림 2-11. step4〉

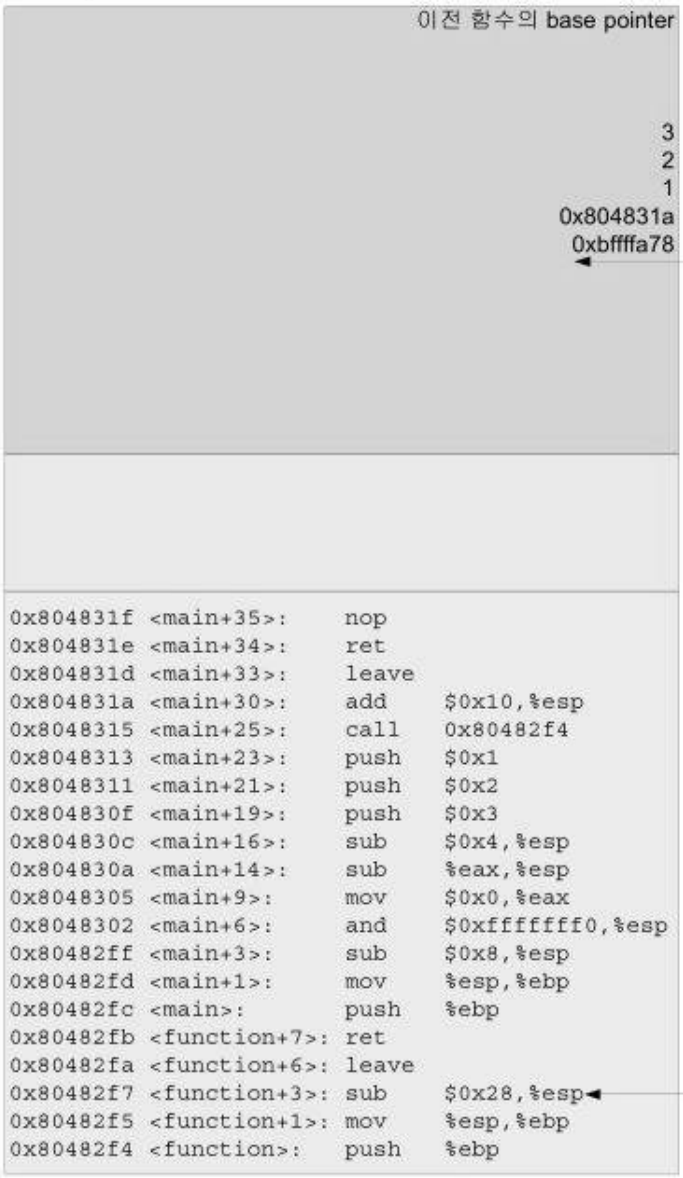
이제 EIP는 function() 함수가 시작되는 지점을 가리키고 있고 스택에는 main()함수에서 넣었던 값들이 쌓여있다.

push %ebp

move %esp, %ebp

function() 함수에서도 마찬가지로 함수 프롤로그가 수행된다. main()함수에서 사용하던 base pointer가 저장되고 stack pointer를 function함수의 base pointer로 삼는다.

(5)step 5



〈그림 2-12. step5〉

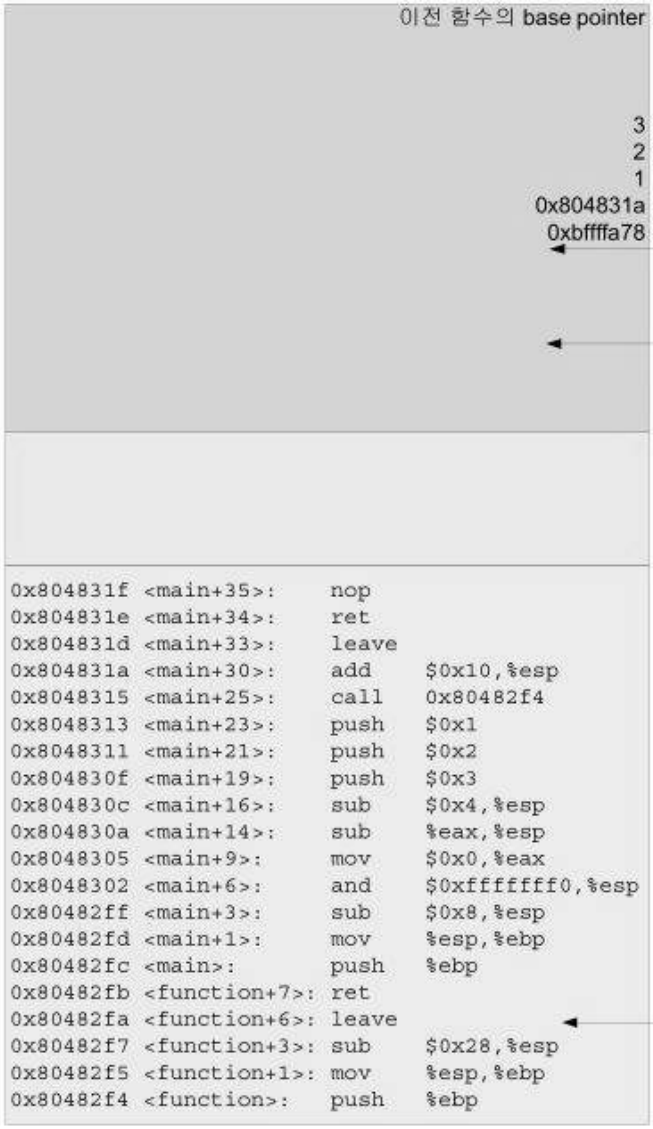
function() 함수의 프롤로그가 끝나고 나서

sub $0x28, %esp

는 스택을 40바이트 확장하는 것이다.

40바이트가 된 이유는 simple.c의 function() 함수에서 지역변수로 buffer1[15]와 buffer[10]을 선언했기 때문인데, buffer1[15]는 총 15바이트가 필요하지만 스택은 word(4byte) 단위로 읽고 쓰이기 때문에 16바이트가 할당되고 buffer2[10]을 위해 12바이트가 할당되어 총 28비트가 할당되어야 하지만 컴파일러의 버전에 따라서 16바이트 배수로 확장될 수도 있으며 dummy값이 들어갈 수도 있기 때문에 총40바이트가 할당된 것이다. 이렇게 임시적으로 확장한 공간이 buffer인 것이다.

(6)step 6



이렇게 만들어진 버퍼에는 이제 우리가 필요한 데이터를 쓸 수 있게 된다.

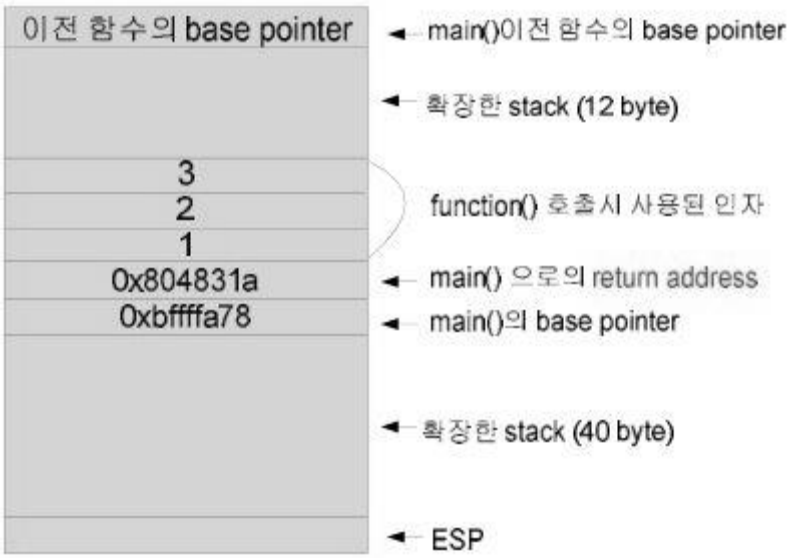
보통

mov $0x41, [%esp-4]

mov $0x42, [%esp-8]

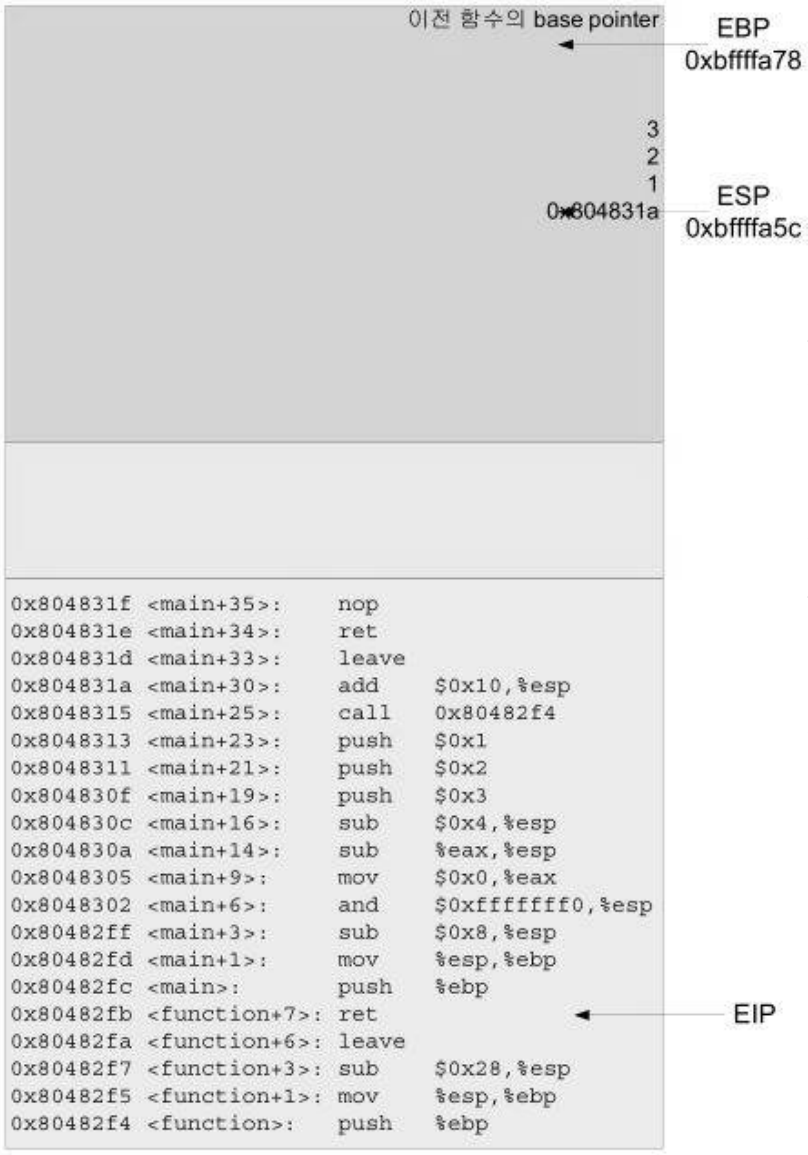
와 같은 형식으로 ESP 기준으로 스택의 특정 지점에 데이터를 복사해 넣는 방식으로 동작한다.

〈그림 2-13. step6〉



〈그림 2-14. function() 수행 중 스택의 모습〉

(7)step 7



이제 leave instruction을 수행했다. leave instruction은 함수 프롤로그 작업을 되돌리는 일을 한다. 위에서 본 대로 함수 프롤로그는

push %ebp
mov %esp, %ebp
이었다. 이것을 되돌리는 작업은

mov %ebp, %esp
pop %ebp
이다. leave instruction 하나가 위의 두 가지 일을 한꺼번에 하는 것이다. stack pointer를 이전의 base pointer로 잡아서 function()함수에서 확장했던 스택 공간을 없애버리고 PUSH해서 저장해 두었던 이전 함수 즉, main()함수의 base pointer를 복원시킨다.

〈그림 2-15. step7〉

POP을 했으므로 stack pointer는 1 word 위로 올라갈 것이다.

그러면 이제 stack pointer는 return address가 있는 지점을 가리키고 있을 것이다.

ret instruction 은 이전 함수로 return 하라는 의미이다. EIP 레지스터에 return address를 POP 하여 집어넣는 역할을 한다. 굳이 표현하자면

pop %eip

라고 할 수 있겠지만 앞에서 설명한 대로 EIP 레지스터는 직접적으로 수정할 수 없기 때문에 위와 같은 명령이 먹히지는 않는다.

(8)step 8

〈그림 2-16. step8〉

ret

을 수행하고 나면 return address는 pop되어 EIP에 저장되고 stack pointer는 1 word 위로 올라간다.

add $0x10, %esp

는 스택을 16바이트 줄인다. 따라서 stack pointer는 0x804830c에 있는 명령어를 수행하기 이전의 위치로 돌아가게 된다.

leave

ret

를 수행하게 되면 각 레지스터들의 값은 main()함수 프롤로그 작업을 되돌리고 main() 함수 이전으로 돌아가게 된다. 그 함수는 init_process()함수로 운영체제가 호출하는 함수이며 자세히 알 필요는 없다.

5. BufferOverflow 공격의 원리

지금까지 프로세스의 실행 과정을 살펴 보았다. BufferOverflow 공격은 크게 Stack BufferOverFLow 공격과 Heap BufferOverflow 공격으로 나뉜다. 이 연구에서는 Stack BufferOverflow 만 다루기로 하며 세부 원리에 대해 자세히 살펴보도록 한다.

(1)Buffer

버퍼(buffer)란 시스템이 연산 작업을 하는 데 있어 필요한 데이터를 일시적으로 메모리 상의 어디엔가 저장하는 데 그 저장 공간을 말한다. 문자열을 처리할 것이라면 문자열 버퍼가 되겠고 수열이라면 숫자형 데이터 배열이 되겠다. 대부분의 프로그램에서는 바로 이러한 버퍼를 스택에 생성한다.

(2)BufferOverflow

BufferOverflow (이하 BOF) 공격은 미리 준비된 버퍼에 버퍼의 크기 보다 큰 데이터를 씀으로써 공격자가 원하는 대로 프로세스의 동작 흐름을 바꾸는 것이다. 〈그림 2-14〉에서 보는 스택의 모습은 40바이트의 스택이 준비되어 있으나 40바이트 보다 큰 데이터를 쓰면 버퍼가 넘치게 되고 프로그램은 에러를 발생시키게 된다. 만약 40 바이트의 데이터를 버퍼에 쓴다면 아무런 지장이 없을 것이다. 하지만 41~44바이트의 데이터를 쓴다면 상위 공간의 base poonter를 수정하게 될 것이다. 더 나아가 45~48 바이트의 데이터를 쓴다면 return address가 저장되어 있는 공간을 침범하게 될 것이고 48바이트 이상을 쓴다면 그 이전에 스택에 저장되어 있던 데이터마저도 바뀌게 될 것이다.

여기서 시스템에게 첫 명령어를 간접적으로 내릴 수 있는 부분은 return address 가 있는 위치이다. return address는 현재 함수의 base pointer 바로 위에 있으므로 그 위치는 변하지 않는다. 공격자가 base pointer를 직접적으로 변경하지 않는다면 정확히 해당 위치에 있는 값이 EIP에 들어가게 되어 있다. 공격자는 실행시키고자 하는 코드의 주소를 return address에 넣어줌으로서 EIP를 간접적으로 수정하여 원하는 코드를 실행시킬 수 있다. 이것이 BOF 공격의 기본 원리이다.

(3)C언어의 취약 함수

C언어에서 BOF 공격에 취약한 함수는 당연하게도 buffer를 사용하는 함수이다. 그 중 몇 가지를 살펴보겠다.

〈표 2-5. C언어의 BOF 취약 함수〉

함수	설명
scanf(const char *format, ...)	엔터가 입력될 때 까지 입력을 받는다. 공백이 없다면 버퍼의 사이즈를 고려하지 않고 입력을 보두 받아들이기 때문에 BOF가 발생할 수 있다.
strcpy(char *dest, const char *src)	dest 에 scr을 복사하는 함수이다. 버퍼의 크기를 고려하지 않아서 스택 내부에 공격코드를 삽입할 수 있고 BOF가 발생할 수 있다.
gets(char *s)	버퍼의 사이즈를 전혀 고려하지 않고 입력을 모두 받아들인다. 따라서 BOF가 발생할 수 있다.
strcat(char *dest, const char *src)	src의 내용을 dest 끝에 붙인다. 버퍼의 사이즈를 고려하지 않으므로 BOF가 발생할 수 있다.
fscanf(FILE *stream, const char *format, ...)	scanf와 일맥상통한다.

Ⅲ. BufferOverflow 공격 실습

2장 까지는 BOF 공격의 및 실습에 필요한 개념을 알아봤다면 이번 장에서는 BOF 공격에 취약한 코드를 직접 짜보고 공격을 해봄으로써 BOF 공격을 정확하게 이해하는 것을 목표로 한다. 다음 코드는 실습을 위해 짠 것이다.

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

void printKey()
{
        printf("key : du^82i!<\n");
}

int main()
{
        char S[10];
        char Password[10] = "!1234his";
        scanf("%s",S);
        if(!strcmp(S,Password))
                printKey();
        else
                printf("Wrong!!\n");

}
```

〈그림 3-17. bof.c〉

프로그램의 구조는 간단하다. 사용자로부터 문자열을 입력받고 그것이 미리 정해둔 Password와 같으면 printKey 함수를 호출하여 Key 값을 얻어내며, 다르면 Wrong 이 출력된다.

이 코드에서 문제가 되는 부분은 scanf() 함수이다. scanf() 함수는 버퍼를 이용하는 대표적인 함수이므로 공격자가 scanf() 함수 내부에서 마련된 buffer보다 큰 값을 넣어주면 메모리 침범이 일어날 수 있다.

BOF 공격을 실습하는 환경은 Linux Ubuntu-16-04-32bit 이다. 우리가 할 공격의 목표는 scanf()를 호출할 때 넣어둔 return address 에 메모리 침범을 활용해 printKey() 함수의 주소를 넣어두는 것이다. 그러면 Password를 몰라도 printKey() 함수가 호출되며 키 값을 얻을 수 있다. 이러한 작업을 수행하기 위해선 code segment 영역의 instruction을 어셈블리어를 통해 분석하여 스택에 할당된 버퍼의 크기, 공격 코드의 주소 등을 정확히 알아야 할 것이다.

1. 디버깅을 위한 리눅스의 사용

리눅스는 디버깅을 위한 편리하고 다양한 도구를 제공한다. 기본적으로 gcc 명령어를 통해 컴파일 하고 gdb 명령어를 통해 디버깅 한다.

```
gcc -o bof bof.c
```

〈그림 3-18. gcc 명령어 예시1〉

〈그림 3-18〉 은 bof.c 파일을 컴파일 하여 bof라는 이름의 실행 파일을 만드는 명령어 이다. 여기서 실습을 위해 몇가지 옵션을 주어야 한다.

-fno-stack-protector

다음 장에서 제대로 소개하겠지만 리눅스는 운영체제 차원에서 BOF 공격을 막기 위해 ASLR과 Stack Canary등등의 다양한 보호 기법을 적용해 놓았다. 따라서 실습을 하기 위해서 이런 스택 보호 기능을 해제해야 한다.

-mpreferred-stack-boundary=2

컴파일러는 여러 의도를 가지고 스택 상의 데이터 사이에 더미(빈 공간)를 생성하게 되는데 그렇게 되면 return address의 위치를 정확히 파악하기 힘들기 때문에 기능을 해제하는 것이다.

```
gcc -fno-stack-protector -mpreferred-stack-boundary=2 -o bof bof.c
```

〈그림 3-19. gcc 명령어 예시2〉

따라서 〈그림 3-19〉와 같이 컴파일을 해주면 실행파일은 준비는 완료되었다.

이제 gdb를 통해 프로그램을 디버깅 해보자.

```
his@his-VirtualBox:~/바탕화면$ gdb bof
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bof...(no debugging symbols found)...done.
(gdb) Quit
(gdb)
```

〈그림 3-20. gdb 디버깅 모드〉

〈그림 3-20〉은 gdb의 디버깅 모드로 돌입한 모습이다. gdb에는 유용한 기능이 많은데 실습에 꼭 필요한 기능정도만 살펴보도록 하겠다.

〈표 3-6. gdb 명령어〉

<table>
<tr><th>명령어</th><th colspan="2">기능</th><th>예시</th></tr>
<tr><td>gdb file</td><td colspan="2">file을 gdb로 로드한다.</td><td>gdb bof</td></tr>
<tr><td>b address</td><td colspan="2">address에 브레이크 포인트를 설정한다.(함수 이름도 가능)</td><td>b main</td></tr>
<tr><td>disas function</td><td colspan="2">function을 disassemble하여 보여준다.</td><td>disas main</td></tr>
<tr><td rowspan="6">x/범위,출력형식,
범위의단위,메모
리주소(함수이름)</td><td colspan="2">주어진 주소 내의 메모리 값이 무엇인지 확인한다.</td><td rowspan="6">x/20wx $esp
esp가 가리키는
메모리로부터 높은 주소
쪽으로 4byte씩 20개를
출력한다.
x/10i $eip
현재 eip의 명령어로부터
10줄의 어셈블리어를
출력한다.
x/s 0x8040000
0x8040000에서 시작하는
문자열을 출력한다.</td></tr>
<tr><td>출력형식</td><td>범위의단위</td></tr>
<tr><td rowspan="2">x : 16진수</td><td>b : 1byte</td></tr>
<tr><td rowspan="2">h : 2byte</td></tr>
<tr><td rowspan="2">s : 문자열</td></tr>
<tr><td rowspan="2">w : 4byte</td></tr>
<tr><td rowspan="2">i : 명령어</td><td>g : 8byte</td></tr>
</table>

2. 공격을 위한 프로세스 분석

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804849e <+0>:     push   %ebp
   0x0804849f <+1>:     mov    %esp,%ebp
   0x080484a1 <+3>:     sub    $0x14,%esp
   0x080484a4 <+6>:     movl   $0x33323121,-0x14(%ebp)
   0x080484ab <+13>:    movl   $0x73696834,-0x10(%ebp)
   0x080484b2 <+20>:    movw   $0x0,-0xc(%ebp)
   0x080484b8 <+26>:    lea    -0xa(%ebp),%eax
   0x080484bb <+29>:    push   %eax
   0x080484bc <+30>:    push   $0x804858f
   0x080484c1 <+35>:    call   0x8048370 <__isoc99_scanf@plt>
   0x080484c6 <+40>:    add    $0x8,%esp
   0x080484c9 <+43>:    lea    -0x14(%ebp),%eax
   0x080484cc <+46>:    push   %eax
   0x080484cd <+47>:    lea    -0xa(%ebp),%eax
   0x080484d0 <+50>:    push   %eax
   0x080484d1 <+51>:    call   0x8048340 <strcmp@plt>
   0x080484d6 <+56>:    add    $0x8,%esp
   0x080484d9 <+59>:    test   %eax,%eax
   0x080484db <+61>:    jne    0x80484e4 <main+70>
   0x080484dd <+63>:    call   0x804848b <printKey>
   0x080484e2 <+68>:    jmp    0x80484f1 <main+83>
   0x080484e4 <+70>:    push   $0x8048592
   0x080484e9 <+75>:    call   0x8048350 <puts@plt>
   0x080484ee <+80>:    add    $0x4,%esp
   0x080484f1 <+83>:    mov    $0x0,%eax
   0x080484f6 <+88>:    leave
   0x080484f7 <+89>:    ret
End of assembler dump.
(gdb)
```

<그림 3-21. disas main>

disas main 명령어를 통해 main 함수의 어셈블리 코드를 얻었다. 이는 프로세스의 code segment 에 들어있는 instruction을 기계어로 번역한 것이다. 첫 줄부터 필요한 부분만 분석하도록 하겠다.

(1)step 1

push %ebp

mov %esp,%ebp

함수의 프롤로그 과정이다. 크게 중요하지 않다.

sub $0x14,%esp

stack pointer를 20(10진수)바이트 확장한다. 즉 현재 스택에는 20만큼의 공간이 있다.

(2)step 2

movl $0x33323121,-0x14(%ebp)

movl $0x73696834,-0x10(%ebp)

movw $0x0,-0xc(%ebp)

스택 영역에 어떠한 주소 값을 넣어주는 듯하지만 특별한 의미를 찾지 못했다. BOF공격에 큰 영향은 주지 않는다.

lea -0xa(%ebp),%eax

push %eax

push $0x804858f

eax 에 현재 ebp의 10만큼의 아래 주소값을 넣어준 다음 스택에 eax를 push해준다. 이는 scanf의 두 번째 인자 값으로 입력받을 공간의 주소 값이다.

```
(gdb) x/s 0x804858f
0x804858f:      "%s"
```

〈그림 3-22. x/s 명령어 사용〉

x/s 명령어를 통해 0x804858f에 들어있는 값을 확인해 보면 %s가 들어있다. 즉 scanf의 첫 번째 인자 값이다.

(3)step 3

call 0x8048370 〈__isoc99_scanf@plt〉

sncaf 함수를 호출하는 명령어이다. 함수의 두 번째 인자로 ebp의 10바이트 만큼의 아래 주소를 주었다. 현재 스택은 24바이트의 공간이 할당되었다. 처음 ebp를 넣었으므로 ebp의 크기인 4바이트, 그다름 esp를 20바이트 확장했기 때문이다. 그리고 scanf의 두 번째 인자 값을 보아 스택의 맨 꼭대기에서 4바이트 아래와 14바이트 아래 사이의 공간이 문자열 변수 S의 공간이라고 할 수 있다.

이제 scanf 함수가 실행되면 return address가 스택에 들어가고 scanf의 프롤로그 과정이 수행될 것이다. 그리고 scanf는 사용자 입력을 받아들이기 위해 버퍼를 할당할 것인데, 스택의 더미값을 해제하는 옵션을 주었으므로 정확히 10바이트가 할당되었을 것이다.

우리의 목적은 return address에 printKey함수의 주소를 넣는 것이므로 마련된 버퍼인 10 바이트와 스택의 ebp값을 아무 값으로 채우고 printKey의 주소를 입력하면 되겠다. main +63 행을 보면 printKey 함수를 호출하는 데 그 옆을 보면 printKey함수의 주소가 0x804848b 임을 알 수 있다.

3. BufferOverflow 공격 코드 작성법

실습할 공격의 목적은 printKey() 함수를 강제로 호출하는 것이다. 따라서 우리는 return address의 자리에 printKey의 주소를 넣어주어야 할 것이다. 그러면 EIP에는 printKey의 주소가 들어갈 것이고 key를 출력하게 될 것이다.

따라서 입력을 받아들이기 위해 할당한 버퍼의 크기 10에 ebp의 크기 4 총 14만큼은 아무 값으로 채우고 return address에 들어갈 printKey주소를 넣어주면 될 것이다.

단순히 AAAAAAAAAAAAAA₩x08₩x04₩x84₩x8b 로 입력하면 될 것 같지만 여기엔 2가지 문제가 있다.

(1)byte order

현존하는 시스템들은 두 가지의 바이트 순서(byte order)를 가지게 되는데 이는 big endian 방식과 little endian 방식이 있다. big endian 방식은 바이트 순서가 낮은 메모리 주소에서 높은 메모리 주소로 되고 little endian 방식은 높은 메모리 주소에서 낮은 메모리 주소로 되어있다. IBM 370컴퓨터와 RISC 기반의 컴퓨터들 그리고 모토로라 마이크로프로세서는 big endian 방식으로 정렬하고 그 외의 일반적인 IBM 호환 시스템, 알파 침의 시스템들은 모두 little endian 방식을 사용한다

예를 들어 74E3FF59라는 16진수 값을 저장한다면 big endian에서는 낮은 메모리 영역부터 값을 채워 나가서 74E3FF59가 순서대로 저장된다. 반면 little endian에서는 59FFE374의 순서로 저장된다. little endian이 이렇게 저장 순서를 뒤집어 놓는 이유는 수를 더하거나 빼는 셈을 할 때 낮은 메모리 주소 영역의 변화는 수의 크기 변화에서 더 적기 때문이다. 예를들어 74E3FF59에 1을 더한다고 하면 74E3FF5A가 될 것이고 메모리상에서의 변화는 높은 메모리 영역에 자리를 잡게 하겠다고 하는 것이 little endian 방식의 논리이다. 높은 메모리에 있는 바이트가 변하면 수의 크기는 크게 변한다는 말이다. 하지만 한 바이트 내에서 bit의 순서는 big endian 방식으로 정렬된다.

요약하면 낮은 자리의 값이 변화가 더욱 잦고 따라서 변경하기 쉬운 자리에 놓겠다는 것이 little endian 방식이다.

이러한 byte order의 문제 때문에 공격 코드의 바이트를 정렬할 때에는 이러한 문제점을 고려해야 한다. 그러므로 공격 코드를 작성할 때에는
AAAAAAAAAAAAAA₩x8b₩x84₩x04₩x08 이 더 적절하다.

하지만 이대로 입력하면 문제가 발생한다.

(2)문자열 입력의 문제
문제는 ₩x8b라는 16진수를 한 바이트로 넣어주어야 하지만 입력을 받을 때 '₩', 'x', '8', 'b'와 같이 각각 개별의 문자로 받아들여 4개의 문자로 구성된 문자열이 돼버린다는 것이다. 따라서 16진수 8b에 해당하는 문자를 넣어주어야 한다. 이는 다양한 방법이 있을 수 있겠다. 필자는 python 코드를 이용해보았다.

```
python -c "print 'A'*14 + '₩x8b₩x84₩x04₩x08'" | ./bof
```

〈그림 3-23. python을 이용한 공격코드 삽입〉

〈그림 3-22〉와 같이 python의 print 함수를 이용해 bof 파일에 문자열을 삽입하여 실행할 수 있다.

결과는 다음과 같다

```
his@his-VirtualBox:~/바탕화면$ python -c "print 'A'*14 + '₩x8b₩x84₩x04₩x08'" | ./bof
Wrong!!
key : du^82i!<
세그멘테이션 오류 (core dumped)
```

〈그림 3-24. bof 공격 코드 삽입 결과〉

〈그림 3-23〉과 같이 강제로 printKey 함수를 호출해 키 값을 얻은 것을 볼 수 있다.

Ⅳ. BufferOverflow 공격 대처 방안

1. linux 환경에서의 메모리 보호기법

BufferOverflow 공격 기법은 간단하면서도 강력한 해킹 기법이다. 따라서 이미 대부분의 운영체제에 그에 대한 방어 기법이 마련되어 있다. 이 장에선 특히 Linux 환경에서 운영체제가 어떤 방식으로 BOF 공격을 막는지 알아볼 것이다.

(1)ASLR(Address Space Layout Randomization)
메모리상의 공격을 어렵게 하기 위해 스택이나 힙, 라이브러리 등의 주소를 랜덤으로 프로세스 주소 공간에 배치함으로써 실행할 때 마다 주소가 바뀌게 하는 기법이다.

(2)Stack Canary
함수 진입시 스택에 return address를 저장할 때 이 정보들이 공격자에 의해 덮어씌워지는 것으로부터 보호하기 위해 스택상의 변수들의 공간과 return address 사이에 특정한 값을 추가하는데 이 값을 Canary 라고 한다.
공격자가 return address를 변조하려고 시도하면 Canary까지 덮어씌워지게 된다. 그리고 프로그램이 종료될 때 canary 값과 원본 값을 비교하여 그 값이 다르면 BOF 공격을 받았다고 판단하여 프로그램을 강제 종료시킨다.

(3)PIE(Position Independent Executable)
PIE는 위치 독립 실행파일로서 실행할 때마다 매핑되는 주소가 어디든 상관없이 실행되는 파일이다. 즉 프로세스의 주소가 무엇 상관없으며 실행될 때 마다 매핑되는 주소가 다르다. 따라서 프로세스의 특정 주소의 값을 수정하는 것과 같은 공격을 방어할 수 있다.

(4)DEP(Data Execution Prevention)
DEP란 데이터 영역에서 코드가 실행되는 것을 막는 기법이다. 만약 공격자가 BOF 공격을 수행해 return address를 스택상의 한 주소(쉘 코드가 위치한 주소)로 변경했다고 하자. DEP가 적용되면 쉘코드에 실행권한이 없어 실행되지 않고 프로그램에 대한 예외처리 후 종료된다.

(5)ASCII-Armor
ASCII-Armor 란 공유 라이브러리 영역의 상위 주소에 0x00을 포함시키는 방법이다. 이 기법은 RTL(Return To Library) 공격에 대응하기 위한 방법으로, 공격자가 라이브러리를 호출하는 BOF 공격을 해도 NULL 바이트가 삽입되어서 접근을 하지 못하도록 한다.

2. 코드 작성 시 방지

지금까지 리눅스의 BOF 공격 대응 기법을 보았다. 실제로 리눅스 환경에서 실행되는 C언어 프로그램에 BOF 공격을 하기란 쉽지 않다. 하지만 모든 운영체제가 이와 같은 기능을 가지고 있는 것은 아니고 그럴 수도 없다. 임베디드 시스템 같은 경우는 동작하는 환경이 리눅스보다 훨씬 규모도 작고 성능도 낮다. 따라서 C언어를 이용해 시스템을 구성한다. 한편 최근 IOT가 쏟아져 나오고 있으며 그 제약 때문에 BOF 공격에 대비하지 않은 시스

템이 많기 때문에 프로그래밍 차원에서 BOF 공격을 막는 방법을 알아 둘 필요가 있다.

만약 visual studio를 사용하여 C언어 코딩을 한다면 간단하다. visual studio에서 제공해 주는 대체 함수가 있기 때문이다. 그 함수는 다음과 같다.

〈표 4-7. vs studio 제공 대체 함수〉

취약 함수	대체 함수
scanf	scanf_s
strcpy	strcpy_s
gets	gets_s
strcat	strcat_s
facsnf	facsnf_s

다음과 같이 접미사로 _s를 붙인 후 입력받을 데이터의 크기를 인자로 넘기면 된다.

물론 컴파일러에 상관없이 프로그래머가 대응할 수 있는 방법도 있다. 예시를 통해 알아보도록 한다.

〈표 4-8. BOF 공격 대응 예시〉

취약 함수	대응 예시
char buf[10]; gets(buf);	char buf[10]; fgets(buf, sizeof(buf), stdin); 입력받을 크기와 스트림을 인자로 넣어준다.
char buf[10]; scanf(“%s”,buf);	char buf[10] scanf(“%9s”,buf) format string에 입력받을 크기를 넣어준다. (fcanf도 동일하다.)
char buf[10]; strcpy(buf, str);	char buf[10]; strncpy(buf, str,sizeof(buf)-1);
char buf[10]; strcat(buf, str);	strncat(buf, str, sizeof(buf)-strlen(buf)-1)

이와 같이 입력받을 크기를 지정하여 메모리 침범이 일어나지 않도록 하는 방식으로 대응을 할 수 있다.

Ⅴ. 결론

지금까지 BOF 공격의 원리, C언어 함수의 취약점에 대해 알아보았고, 실습을 통해 개념을 더욱 확실히 하였다. 또한 BOF를 막기 위한 다양한 방어 기법을 알아보았다. BOF 공격은 공격자 마음대로 프로그램의 실행 흐름을 뒤바꾸는 기법이며 원하는 코드를 실행키기거나 최악의 경우 그 프로그램이 동작하는 시스템의 관리자 권한을 얻을 수도 있다. IOT의 확산에 힘입어 임베디드 시스템이 폭발적으로 증가하는 시점에 현실적으로 모든 시스템에 BOF 공격 방어 기법을 적용하기란 불가능하다. 현재 IOT가 적용된 사물에는 사용자의 민감한 정보를 포함한 기기가 많다. 또는 사용자에게 직접적으로 피해를 줄 수도 있다. 예를 들어 군사 병기 내부의 정보나 전기 자동차 내부 시스템 해킹 등등의 중요한 정보를 잃거나 심하면 인명에 위협이 될 수도 있다. 따라서 우리는 BOF 공격을 이해하고 대처하는 방법을 알 필요가 있었다. 따라서 본 논문에서는 좀 더 능동적인 차원에서 코딩을 할 때에 BOF 공격을 막는 방법에 대해 알아보았고 그것은 대체로 사용자의 입력을 받을 크기를 명시해줌으로써 문제를 해결할 수 있음을 보았다.

참고문헌

[1] 해커 지망자들이 알아야 할 Buffer Overflow Attack의 기초 - 달고나
[2] Parse and Parse Assembly - b0BaNa
[3] Stack based buffer overflow Exploitation Tutorial - Saif El-Sherei
[3] 리버싱 입문 - 조성문
[4] Buffer Overflow & Embedded System - shad0w