攻守兼备

如何绕过碉堡攻击敌后和如何搭一个坚固的碉堡

VictorV 360 Vulcan team

前情提要

- 如何突破虚拟机防线
- 如何安全编程
- 如何减少攻击利用链

大家都懂的道理 -- 没有不透风的墙

- 优秀的开发者+某些方面的短板 = 潜在的破绽
- 庞大的软件系统+参与众多的开发者 = 潜在的破绽
- 击鼓传花+文档不全 = 潜在的破绽

从0开始，讲虚拟化逃逸

突破防线 从一处裂缝到全线崩溃

基本概念

基本概念—CPU寄存器

- 常见的64位intel CPU寄存器: rax, rbx, rdx, rcx, r8, r9, rsp, rbp, rip, rdi, rsi 等, 32位为eax, ebx…
- 函数返回值会在函数返回时放在rax寄存器
- rip寄存器指示了下一个cpu指令的位置, 控制了它就意味着控制了程序
- Windows上64位程序函数参数调用的寄存器顺序: rcx, rdx, r8, r9, [rsp+20]…
  - 分别对应第一个参数,第二个参数…

示例

```
typedef unsigned int u32;
typedef unsigned short u16;

u32 sum(u32 a, u32 b) {
    return a + b;
}

int main(int argv, char **argc) {
    u32 s = sum(argv, 2);
    return s;
}
```

```
mov         edx,2
mov         ecx,dword ptr [argv]
call        sum (07FF6F2361127h)
mov         dword ptr [s],eax

mov         eax,dword ptr [s]

lea         rsp,[rbp+0E8h]
pop         rdi
pop         rbp
ret
```

edx赋值为2

ecx赋值为变量argv的值

调用函数sum

```
mov         eax,dword ptr [b]
mov         ecx,dword ptr [a]
add         ecx,eax
mov         eax,ecx

lea         rsp,[rbp+0C8h]
pop         rdi
pop         rbp
ret
```

取形参b, a的值

求和操作

将返回值放至rax

取rsp指向的值作为返回地址

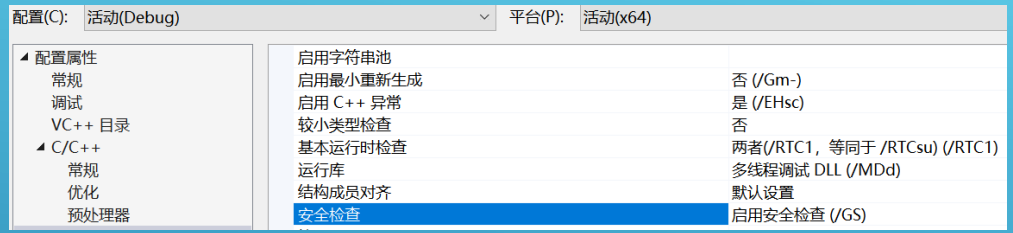
基本概念—堆与栈

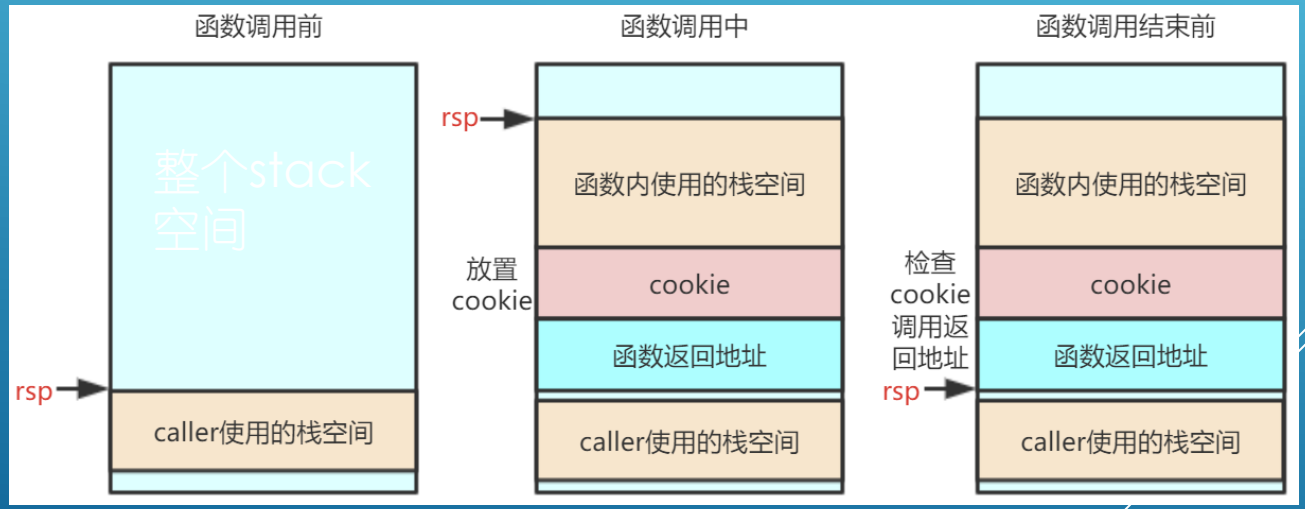
- 堆: 程序的内存空间, 每次申请时由系统映射返回内存; free某个内存时, 会检查相邻堆头.
- LFH堆: 将堆中一个大的segment切割成等大的chunk, 每次申请等大内存时, 系统随机从中选取一个chunk返回. Chunk虽然有堆头, free时不检查相邻堆头.
- 栈: 特殊的堆, 用于存放函数中的临时变量和函数返回地址; rsp寄存器指示了当前栈的位置.

WINDOWS 部分安全机制

- GS 栈保护
- LFH 随机化
- ASLR 地址随机化
- CFG 动态调用保护

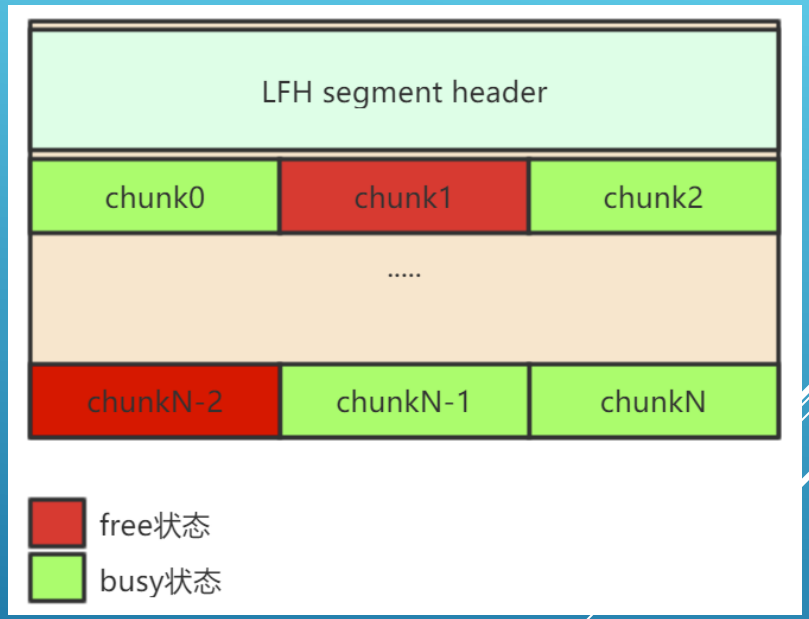
GS安全检查


配置(C): 活动(Debug)
平台(P): 活动(x64)
配置属性
常规
调试
VC++ 目录
C/C++
常规
优化
预处理器
启用字符串池
启用最小重新生成 否 (/Gm-)
启用 C++ 异常 是 (/EHsc)
较小类型检查 否
基本运行时检查 两者(/RTC1，等同于 /RTCsu) (/RTC1)
运行库 多线程调试 DLL (/MDd)
结构成员对齐 默认设置
安全检查 启用安全检查 (/GS)


函数调用前
函数调用中
函数调用结束前
整个stack空间
rsp
caller使用的栈空间
rsp
函数内使用的栈空间
放置cookie
cookie
函数返回地址
caller使用的栈空间
检查cookie
调用返回地址
rsp
函数内使用的栈空间
cookie
函数返回地址
caller使用的栈空间

LFH随机化

- 每次申请属于LFH的堆，返回的chunk序号是随机的，避免在发生堆溢出时，预测出下一个堆的内容
- 与host的操作系统版本有关



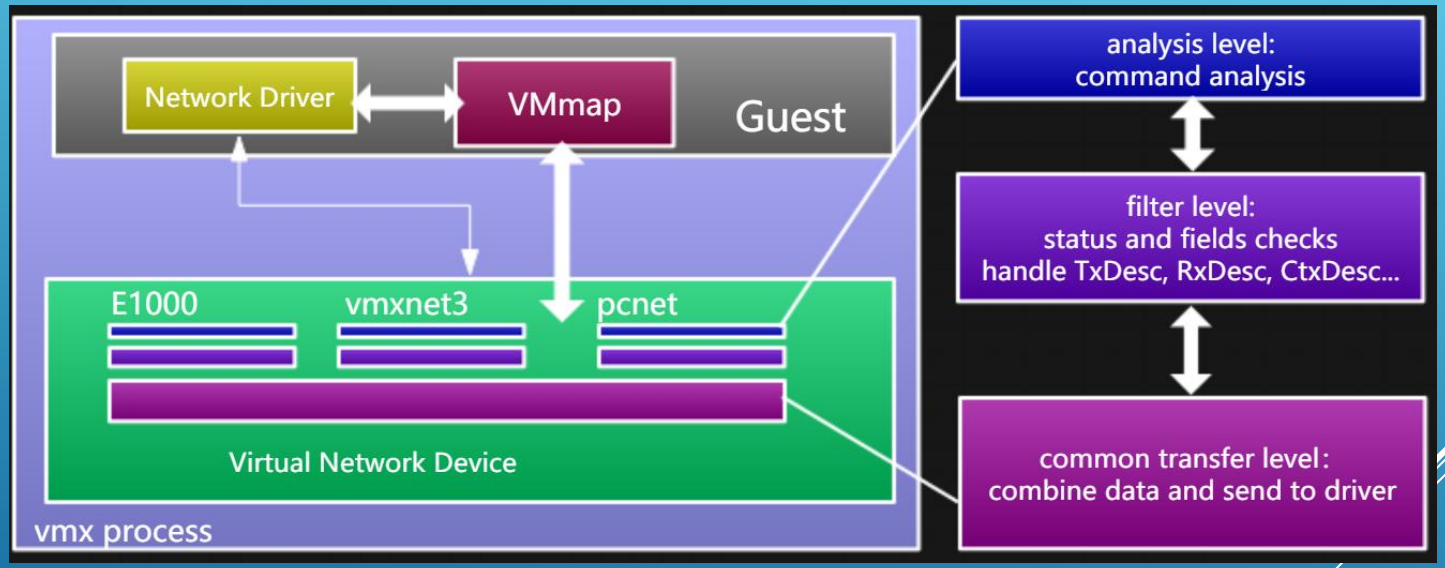
CFG动态调用保护

将函数指针放置于rax

```
mov         edx,2
mov         ecx,dword ptr [rsp+50h]
mov         rax,qword ptr [rsp+30h]
call        qword ptr [__guard_dispatch_icall_fptr (07FF77FF80010h)]
mov         dword ptr [rsp+28h],eax
mov         eax,dword ptr [rsp+28h]
add         rsp,40h
```

```
u32 sum(u32 a, u32 b) {
    return a + b;
}

int main(int argv, char **argc) {
    u32(*sum_func)(u32 a, u32 b);
    sum_func = sum;
    u32 s = sum_func(argv, 2);
    return s;
}
```

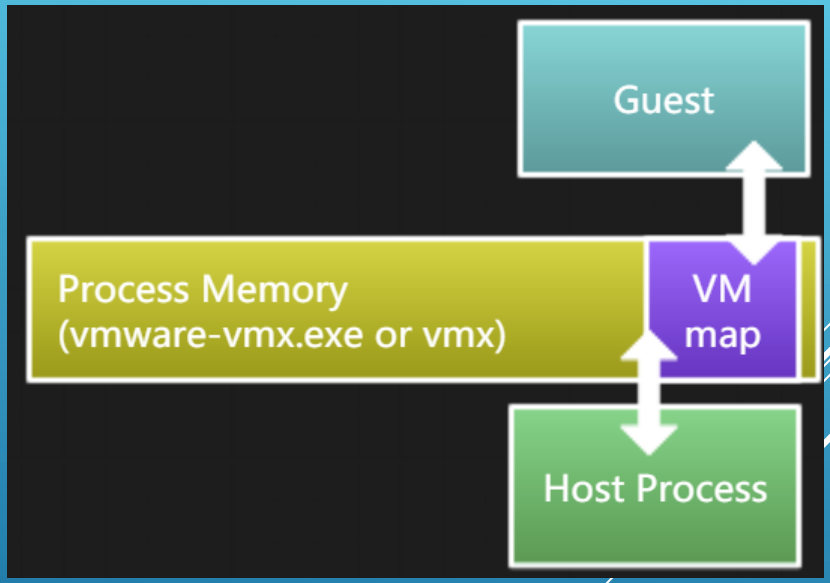
WORKSTATION基本信息


Network Driver
VMmap
Guest
E1000
vmxnet3
pcnet
Virtual Network Device
vmx process
analysis level:
command analysis
filter level:
status and fields checks
handle TxDesc, RxDesc, CtxDesc...
common transfer level:
combine data and send to driver

E1000网卡的架构

虚拟机内存与进程内存的映射关系

- 虚拟机的物理内存实际上在vmware-vmx进程的内存空间中.
- 假如虚拟机的物理地址为0x1000, 则正常转换后, 在进程中的地址为vmmap_start+0x1000.



SVGA虚拟设备在中的MOB结构体

- Struct Mob{

  …

  +58h u64 vmaddr;//　虚拟机物理内存的地址经过转换函数后的地址

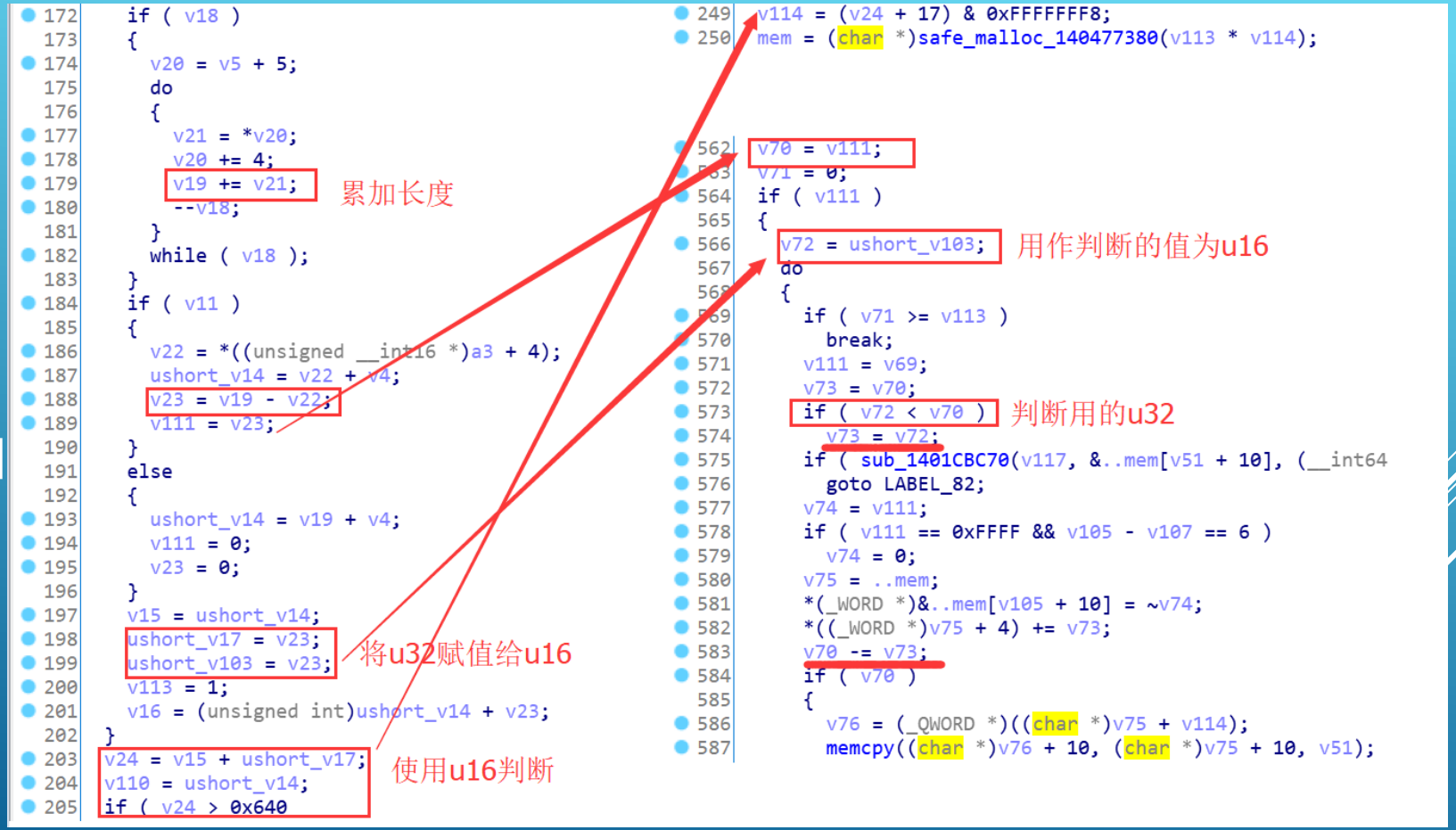
  +60h u32 size;// 指示vmaddr的大小

  +64h u32 reference;

- };

BUG

BUG1



未初始化衍生的BUG 2

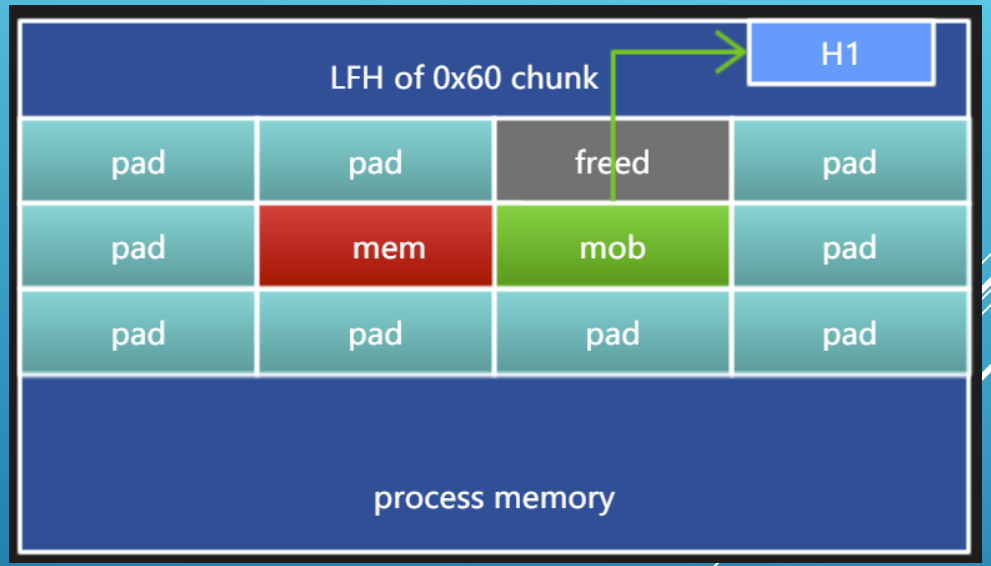
```
147         *(_WORD *)&v33[16] = *(_WORD *)(v10 + 656);
148         *(_WORD *)&v33[18] = *(_WORD *)(v10 + 660);
149         *(_DWORD *)&v33[20] = v20;
150         v33[24] = v18;
151         v33[25] = v19;          在这个分支做了初始化操作
152         v33[26] = 0;
153         if ( !*(_BYTE *)(v10 + 540) || !(*a5 & 0x40) )
154           goto LABEL_45;
155         v21 = (unsigned int)*a5 >> 16;
156       }
157       else
158       {
159         v22 = *a5;          在这个分支却没有初始化
160         v17 = (_WORD *)(v10 + 608);
161         v23 = *a5;
162         *(_WORD *)&v33[16] = 0;
163         v33[0] = BYTE1(v23) & 1;
164         v33[1] = (v22 >> 9) & 1;
165         if ( !_bittest(&v16, 0x19u) )
166           goto LABEL_62;
167         if ( !*(_BYTE *)(v10 + 540) || !_bittest(&v16, 0x1Eu) )
168           goto LABEL_45;
169         v21 = *(unsigned __int16 *)(v5 + 14);
170       }

190       v28 = vul_handle_packet_ring_1401CC780(v27, (_WORD *)v8, v33);
191       v29 = v28;
```

```
588     v77 = a3[24] == 0;
589     ..mem = (char *)v76;
590     if ( !v77 )
591     {
592       v78 = (char *)v76 + v106;
593       v79 = ntohs(*((_WORD *)v78 + 7));
594       v80 = htons(v79 + 1);
595       *((_WORD *)v78 + 7) = v80;
596       if ( v80 )
597       {
598         v81 = (unsigned __int16)~*((_WORD *)v78 + 10);
599         v82 = htons(1u);
600         *((_WORD *)v78 + 10) = ~(v82
601                                + v81
602                                + (((unsigned int)v82 + v81) >> 1
603                                + (((unsigned __int16)(v82 + v81)
604       }
605     }
606     if ( a3[25] )          由于未初始化，导致条件可以成立
607     {
608       v83 = ntohl(*(_DWORD *)((char *)v76 + v107 + 14)) + ushort
609       *(_DWORD *)((char *)v76 + v107 + 14) = htonl(v83);
610       v84 = htons(ushort_v103);
611       v85 = v84 + v108;
612       v108 += v84;
613       if ( v83 < ushort_v103 )
614         v108 = htons(0xFFFEu) + v85;
```

漏洞的利用

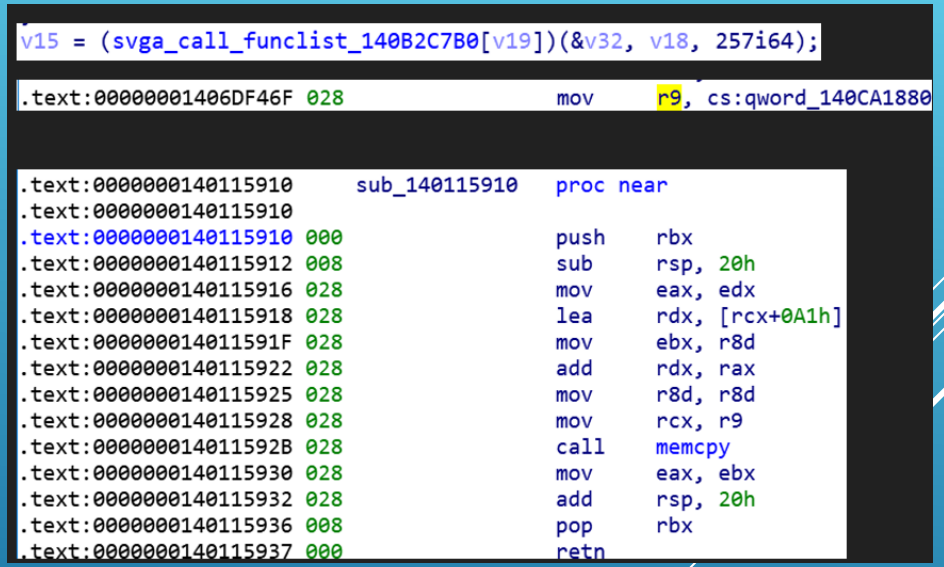
任意读写的实现

1. 申请足够多mob结构体来填充一个segment
2. 释放其中一个mob, 申请一个可溢出的堆, 并跳跃修改mob的size字段
3. 通过mob越界读写, 读取后续内存, 搜索进程地址
4. 通过泄露的地址, 修改mob表, 伪造表地址



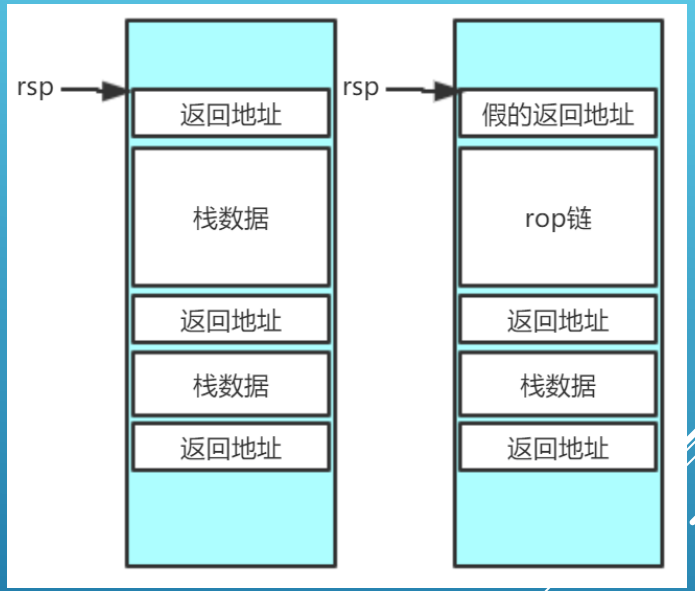
泄露信息

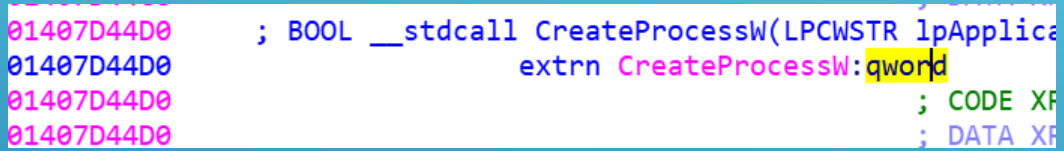
1. 修改svga命令的动态调用表为sub_140115910函数
2. 通过一些svga命令可以设置r9为指定地址
3. 通过调用svga命令触发调用表，调用sub_140115910，将rcx指向的+0xA1的值拷贝倒r9指向的内存，完成栈地址泄露



RIP控制

- 再次调用mob拷贝，将地址指向泄露的栈地址，拷贝构造的虚假返回地址，实现劫持rip.





- 利用任意读写泄露CreateProcessW的地址
- 搜索ROP实现函数调用

执行命令

巩固防线 从每一个开发者开始

编程中常见的安全错误

常见的安全错误

- Use after free & double free
- 内存溢出
- 未初始化引用
- 线程竞争

UAF & DOUBLE FREE

- 把某个内存地址赋值给其它结构体或全局变量后，忘记在释放时清空.
  - 谁赋值谁就负责把释放的操作给写上，避免遗漏.
- 递归调用的检查不严格，在上一次释放前，下一层已经释放. (外部演示 xhci bug)
  - 在递归中的内存，最好加一个是否释放的flag

内存溢出

- 不同变量类型之间赋值
  - 比如 u16 len = u32 length; 检查用的u32 length, 申请内存却用len.
- 检查完后还在做加减操作, 导致未检查到
  - 打补丁容易遗漏, 导致补丁位置出错.
- 完全忘了检查
  - 在每次调用前都做好检查, 不要一劳永逸
  - Malloc申请时, size 为 0 依然会返回正确地址, 如果不检查就用, 就会很危险

示例

- a,b如果为0, 则sum为0, 导致申请了内存大小为0的内存, 造成内存溢出

- 虽然检查了sum的整数溢出, 但是在检查完后做了加减操作, 导致整数溢出. (常见于打补丁的位置错误)

```
u16 a,b,sum;
sum = (a+b+0xf)&0xf0;
if(sum < a || sum < b){
  printf("interger overflow");
  return;
}
char *mem = malloc(sum);
```

```
u16 a,b,sum;
sum = a+b;
if(!sum || sum < a || sum < b){
  printf("interger overflow");
  return;
}
sum = sum+2;
char *mem = malloc(sum);
memcpy(mem, src, a);
```

未初始化引用

- malloc后不清零
  - 用calloc代替，calloc会清零申请的内存，避免忘记重置
- 申请的栈或者堆未重置
- 忽略函数返回值

PWN2OWN 2017

```
4  char v2; // [rsp+20h] [rbp-58h]
5  char v3; // [rsp+88h] [rbp+10h]
6  int v4; // [rsp+8Ch] [rbp+14h]
7
8  if ( a1 == 1 )
9  {
10   get_id_140535530((idtemp *)&v3, (idctx *)&sidCtx_140CA1178);
11   while ( v4 )
12   {
13     v1 = (unsigned int *)ret_id_mem_140535060((idtemp *)&v3, (__int64)&sidCtx_140CA1178);
14     if ( v1 )
15     {
16       if ( v1[2] & 4 )
17       {
18         vul_get_otable_struct_140151C20(1, *v1, 1i64, (unsigned __int64)&v2);
19         dma_safe_free((dma_out *)&v2);
```

忽略返回值，导致v2未初始化就引用

线程竞争

- 把VM的物理内存数据直接使用
  - 如果需要使用VM传递的值，需要把值复制到进程空间后再使用
- 危险内存没有加锁就操作
  - 只要是存在多线程的，就一定要加锁

```
44      v18 = *(_QWORD *)(v17 + *(_QWORD *)(v7 + 104));
45      if ( v18 )
46      {
47        data_of_VM = *(_QWORD *)(v18 + 8);
48        v20 = ((unsigned __int16)(*(_DWORD *)(data_of_VM + 8) >> 21) + 1) & 0x7FF;
49        if ( ((unsigned __int16)(*(_DWORD *)(data_of_VM + 8) >> 21) + 1) & 0x7FF
50          && (unsigned __int8)*(_DWORD *)(data_of_VM + 8) == -31 )
51        {
52          v21 = *(unsigned int *)(data_of_VM + 12);
53          if ( !*(_DWORD *)(data_of_VM + 12)
54            || !check_dmaaddr_1403B3800(v21, ((unsigned __int16)(*(_DWORD *)(data_o
55          {
56            log_print_1403D4B40((__int64)"UHCI: Bad %s pointer %#I64x\n", (__int64)
57            *(_DWORD *)(v25 + 1632) = 160;
58            sub_140528200(v14);
59            return 0;
60          }
61          dma_get_toa2_1403B7120(v21, a7, v20, 0, 5);
62          v16 = a8;
63        }
64        *v13 = v20;
```

加强防线 让攻击者举步维艰

这样编程，让程序更安全

- 结构体中的buf指针放在size之前
- 临时数组放在函数最后声明
- 内存不要放程序空间的地址
- 减少可变大小的内存申请(比如使用0x1000对齐替代任意大小)
- 固定大小的数组使用时一定要检查索引范围(比如全局数组)
- 栈变量的指针尽量不要传递给动态call

这样编程，可以减少攻击者利用链

答疑