
看雪·第六届安全开发者峰会
基于硬件虚拟化技术的新一代二进制分析利器
程聪 阿里云安全-系统安全专家
2022 SDC

自我介绍

现就职于阿里云安全-系统安全团队

主要研究方向:

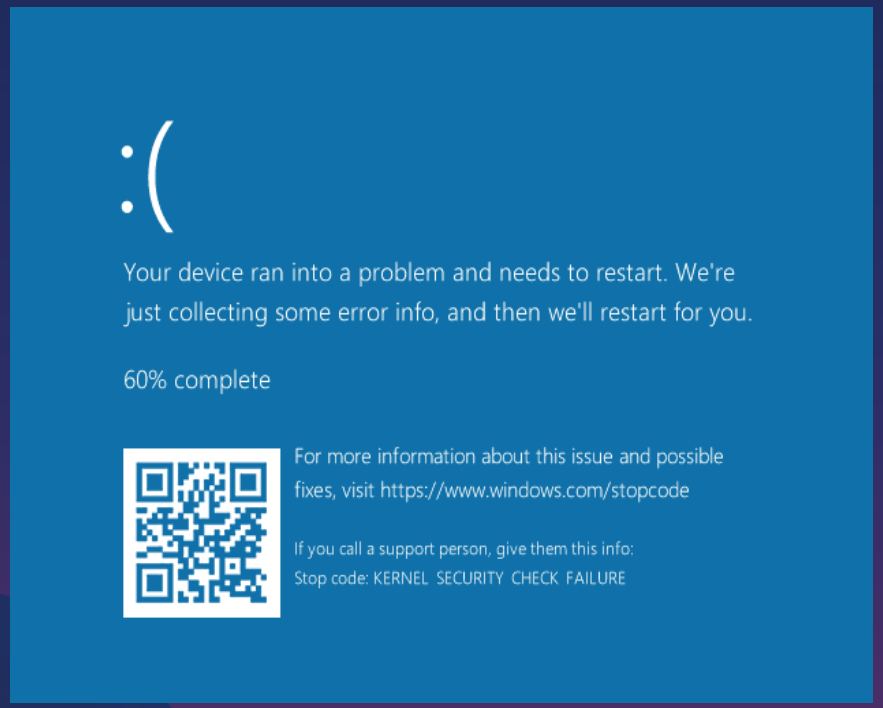
- 病毒检测
- 主机安全
- 内核安全
- 虚拟化安全
- 二进制攻防

演讲内容

- 背景介绍
- QEMU/KVM简介
- 无影子页ept hook
- 虚拟化调试器
- 内核级trace
- 总结

背景介绍

从PatchGuard谈起



大家先来看下左边这张图片，搞内核安全的应该不陌生。windows x64内核引入patchguard后，对内核敏感部分，进行patch、hook，修改msr、IDT表等操作，都会触发蓝屏

但是有很多场景还是需要对内核进行hook，安全研究人员发现可以借助硬件虚拟化特性，实现ept hook，来兼容patchguard

传统的ept hook一般使用影子页来实现，我们发现这种方法存在一些问题。本次分享会介绍一种新方法，巧妙解决这些问题

虚拟化平台

要实现ept hook，首先需要一个类似左图的虚拟化平台，提供整体的框架

从图中能看出，开发一个完整的vmm工作量太大，所以我们选择基于现有的虚拟化平台进行二次开发，但一些专注安全领域的开源平台如hyperplatform，或多或少都存在各种各样的问题，最终我们选择了基于qemu\kvm做二次开发

本次分享会介绍如何基于qemu\kvm，快速打造无影子页ept hook，虚拟化调试器、内核级trace工具

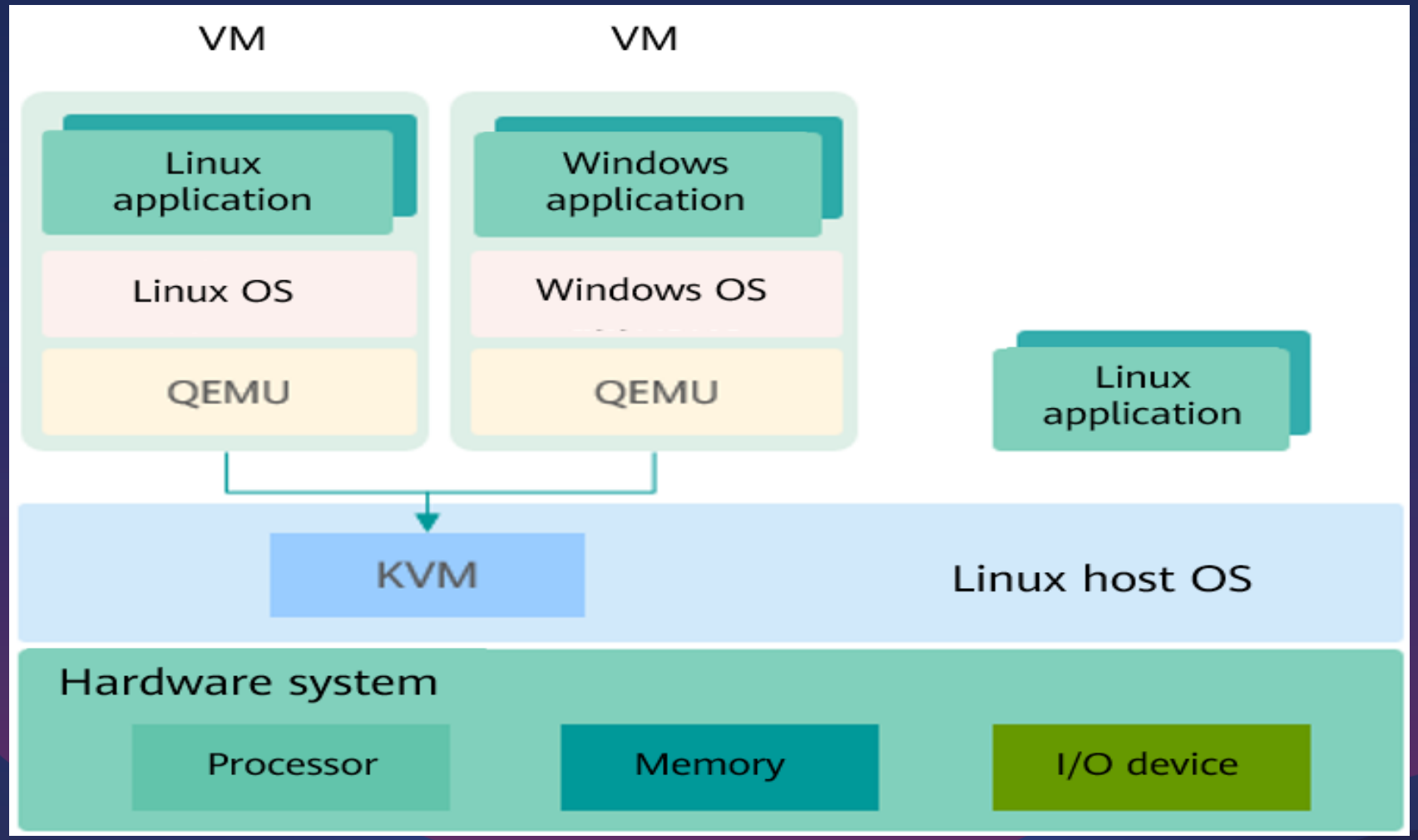
为什么选择QEMU/KVM

相比于hyperplatform等虚拟化平台，qemu\kvm有以下优势

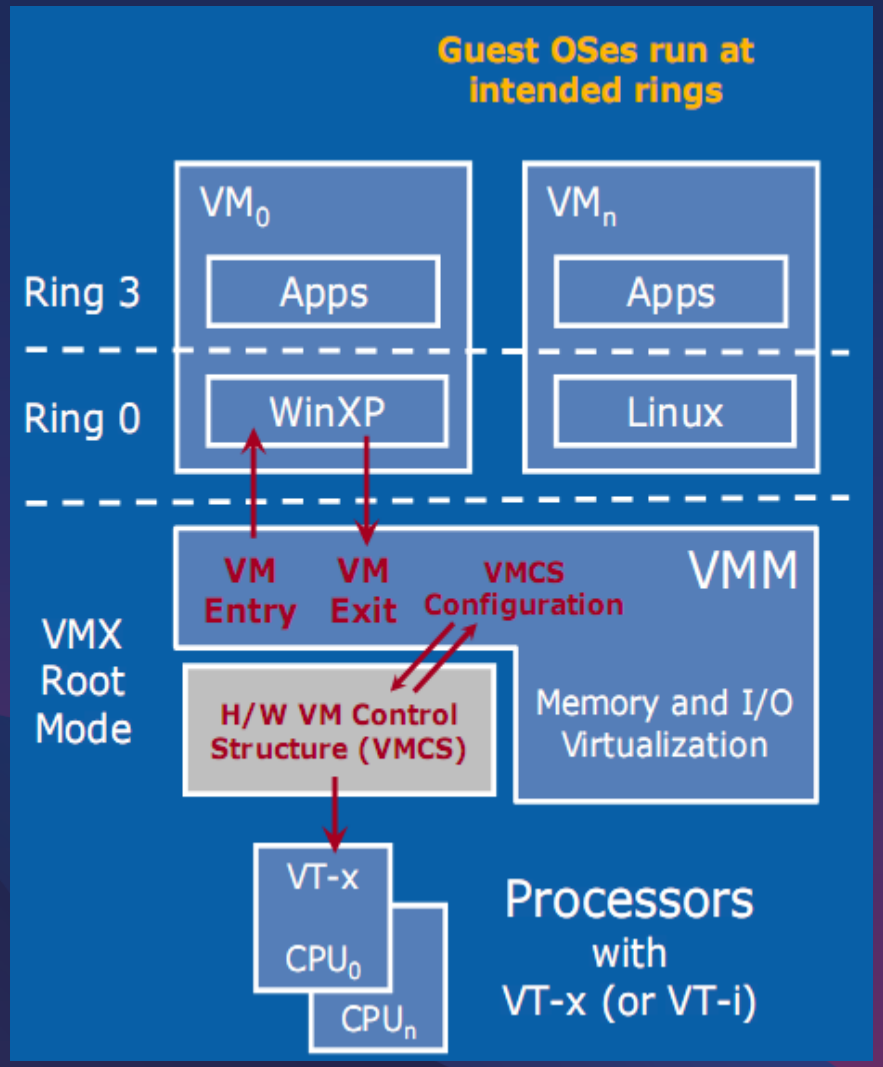
- 代码完善度高，鲁棒性好，稳定性高，性能开销小
- 支持windows、linux等多种guest os
- 背靠linux内核，各种基础设施齐全，方便二次开发
- 在云上广泛使用，环境不会被特殊针对
- 支持gpu透传和虚拟化，可以运行图形化程序
- 支持嵌套虚拟化，可以运行vmm程序

QEMU/KVM简介

QEMU/KVM整体架构

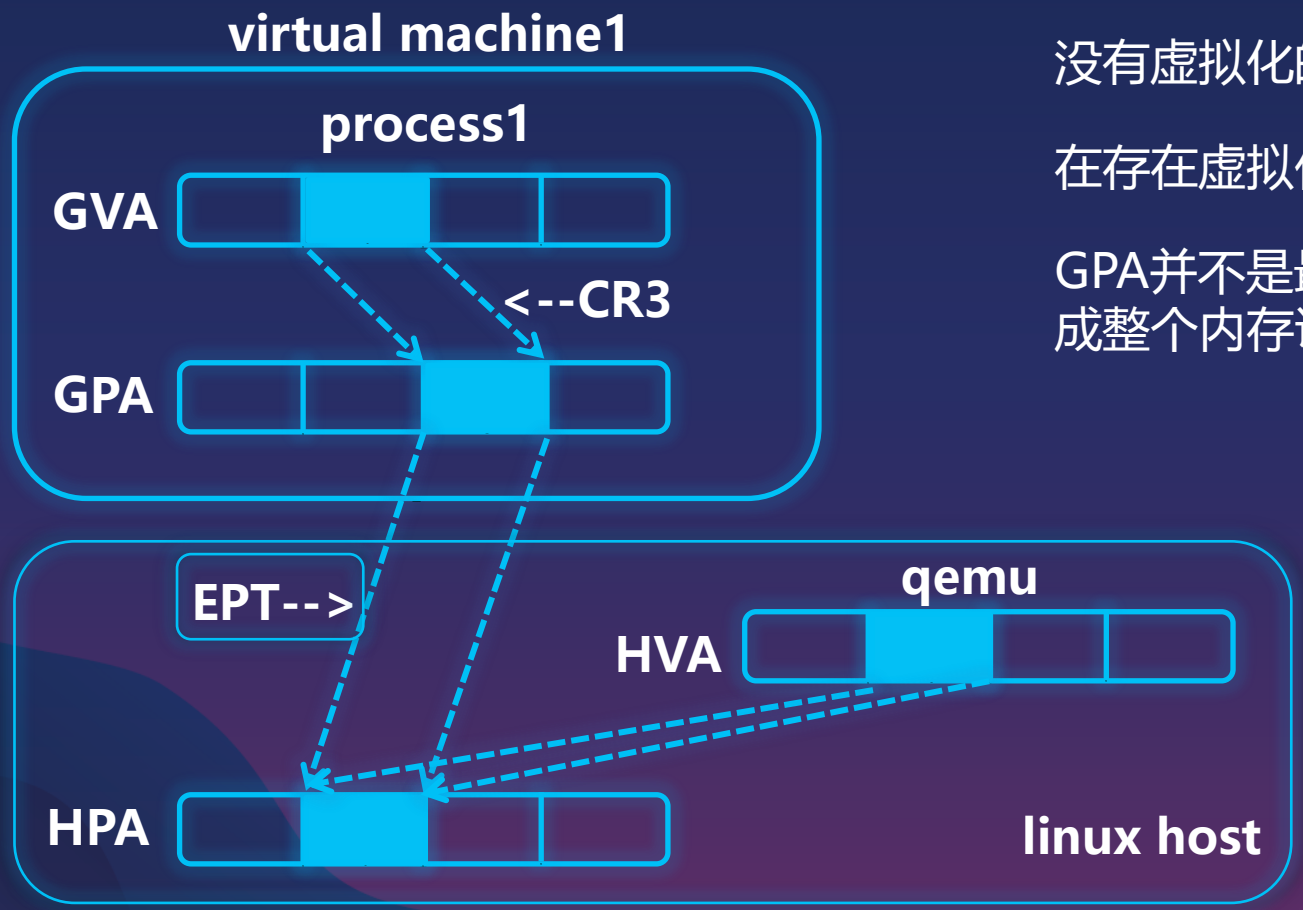


cpu虚拟化



如左图所示，guest操作系统在(ring 0)上运行，同时vmm运行在具有更高特权级别(ring -1) 上。执行系统调用等不涉及关键指令的指令，vmm不会介入。这样guest操作系统就可以为其应用程序提供高性能的内核服务。当guest使用特权指令(比如cpuid)，或者发生异常时会产生vmexit，从guest退出到host中，host处理完成后，再通过vmentry回到guest

内存虚拟化



没有虚拟化的情况下，内存地址翻译如左图所示，只有VA->PA

在存在虚拟化的情况下，GVA->GPA的翻译发生在虚拟机内部

GPA并不是最终的物理内存，还需要通过EPT翻译成HPA，才完成整个内存访问

qemu的HVA也映射为HPA，所以一般来说GPA对应着qemu的HVA

EPT翻译过程跟普通页表类似，也是通过多级页表实现，手动去掉页表项某些权限，就可以达到监控和欺骗的目的

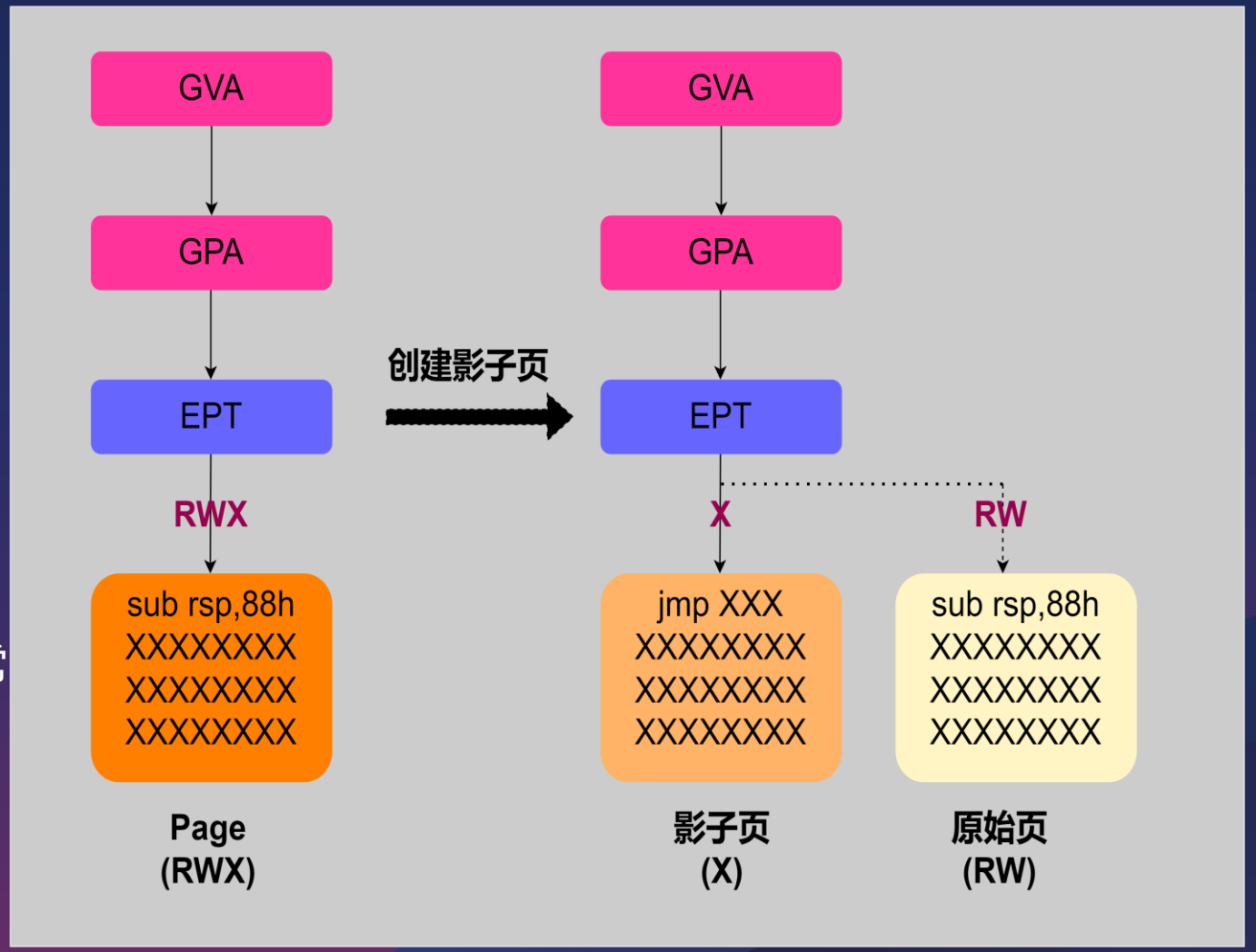
EPT相关操作都在host上进行，对guest内核和应用程序不可见，从而可以进行降维打击

无影子页ept hook

影子页ept hook

- 如右图，我们在原始页页面的基础上，创建一个只有X权限影子页，原始页保留RW权限，并在影子页上进行hook，修改头部指令为jmp
- 当有cpu执行此页时，就会触发hook逻辑
- 当有cpu读取此页时，由于影子页只有X权限，会产生ept violation，从而vmexit到host，host将页面切换成原始页(RW)进行读写，cpu会读取到原始的内容，我们达到了欺骗(无痕)的效果
- 下次cpu再执行原始页(RW)时，同样会触发异常，我们再将页面切换回影子页(X)执行

这种方法存在什么问题?



存在的问题

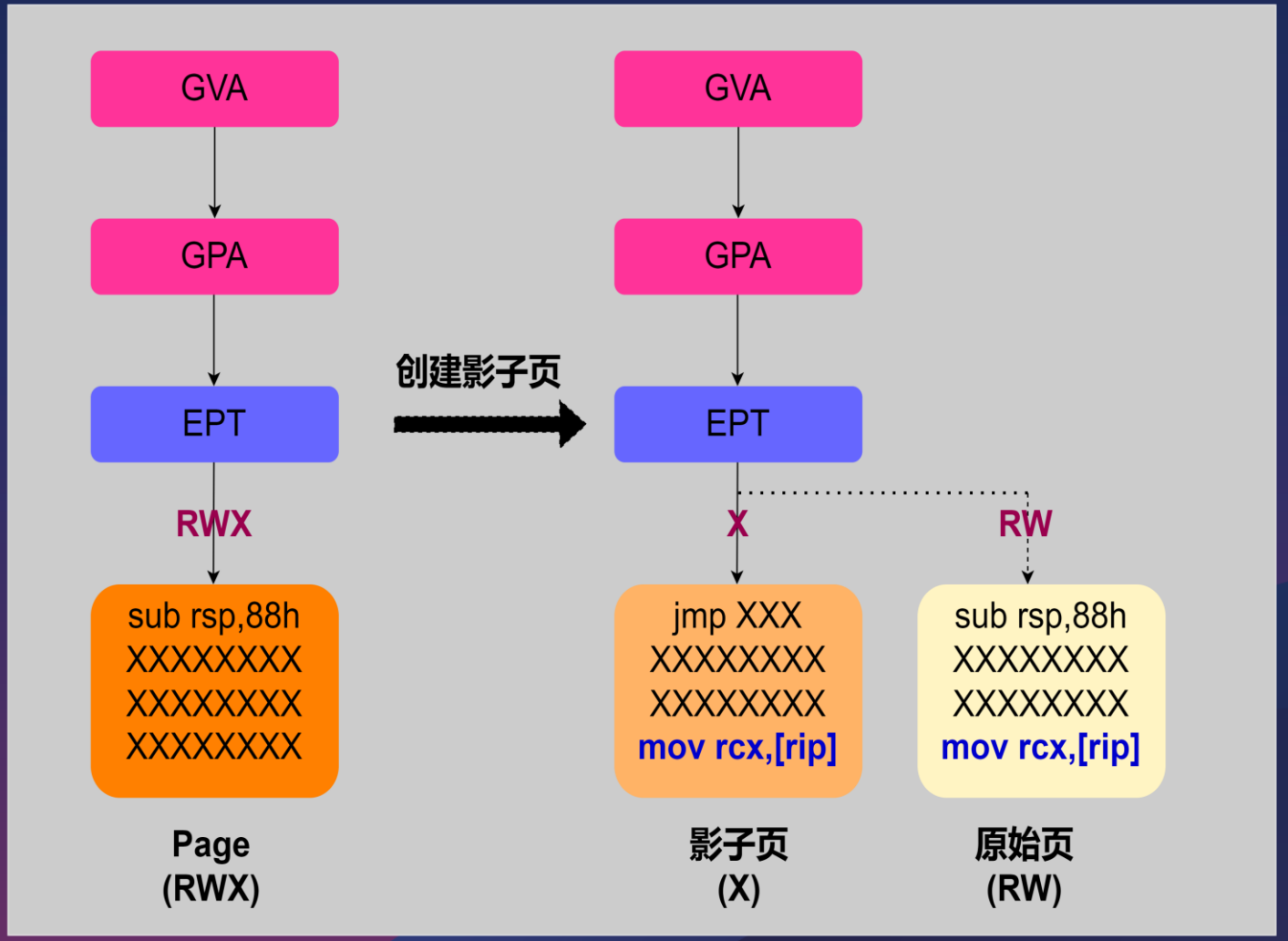
遇到右图自己读写自身页面的指令会怎么样?

- 在影子页(X)上执行mov rcx,[rip]时，会读取当前页面，由于当前页面只有X权限，会产生异常并切换到原始页(RW)去执行读取，由于RW页没有X权限，又会产生异常并切换回影子页(X)，来回切换进入死锁

这种读写自身页面的场景很常见，比如

- switch case语句，在某些情况下会在当前代码页面，编译生成jump table，这种情况在windows内核中很常见
- 代码自修改，这种在用户态中比较常见，比如一些壳、对抗代码中

我们如何改进?



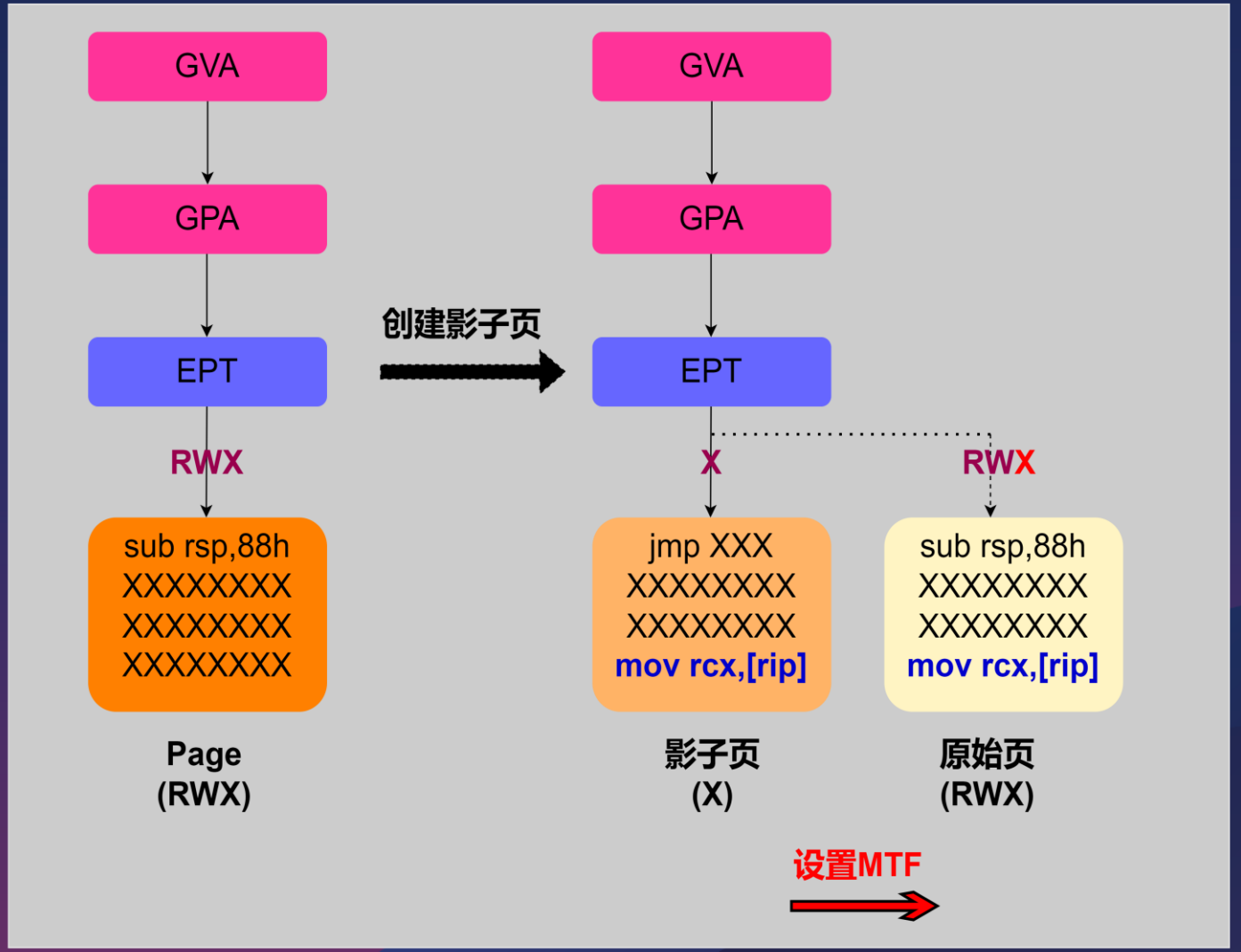
改进后的解决方案

如右图所示，改进后的方案不去除原始页的X权限，遇到页面自读写的指令，从X切换到RWX，原始页可顺利执行此指令

在执行完此指令后，我们需要切换回影子页(X)，否则后续cpu执行时hook逻辑不再生效，这个切换回去的方法一般通过设置MTF来实现

MTF:全称是Monitor Trap flag，可以理解为单步异常，当host设置该标志位时，回到guest执行完一条指令后会触发vmexit，exit理由为MTF

改进后还有什么问题?



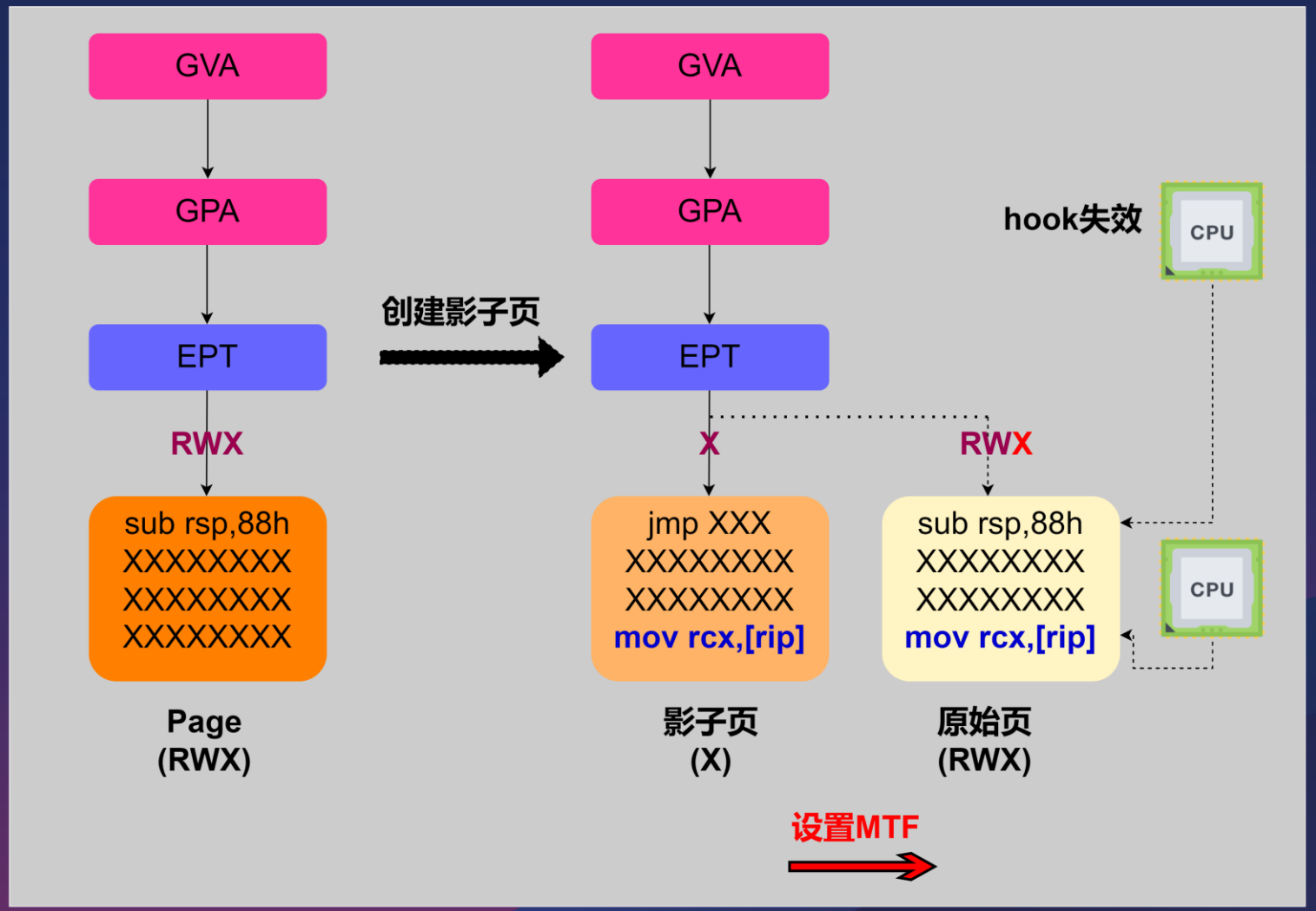
依旧存在的问题

如右图，由影子页(X)切换到原始页(RWX)单步执行时，如果其他核正在执行此页面，会出现hook失效

如何规避这种情况?

出现这个问题的主要是因为一个核的页面权限影响了另外一个核，可以给每个核设置一套自己的EPT页表来规避，但是这种方法内存占用多，性能损耗很大

类似kvm这种商业化落地的vmm，由于性能等因素一般都是共用一套EPT页表，这种情况下，我们如何解决这个问题?



思考

从前面可以看到，进行ept hook最核心的点是要页面切换，为什么要切换，是为了欺骗cpu读写，让其读写到修改前的页面内容

有没有不切换页面，同时可以欺骗cpu读写的方法?

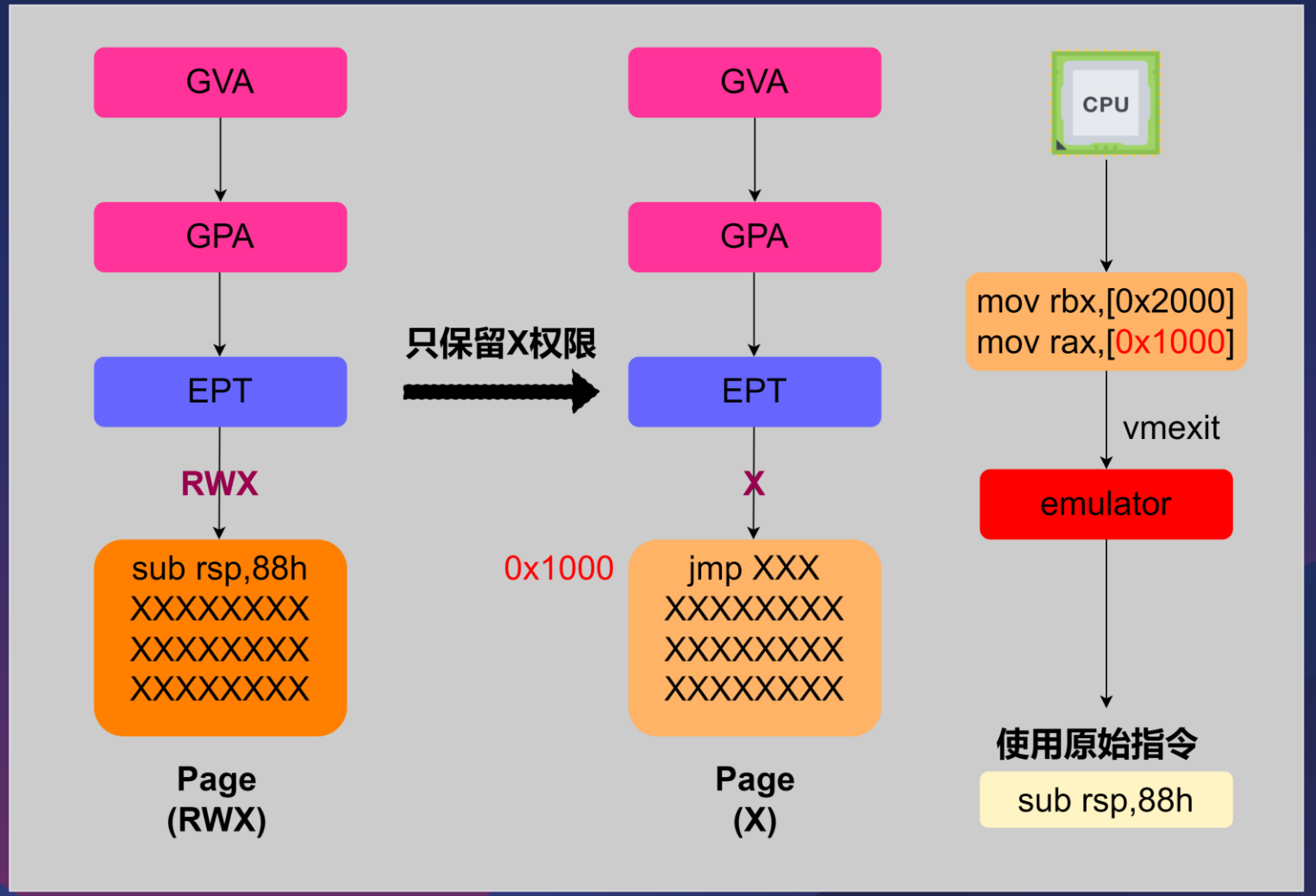
答案是模拟执行

模拟执行

引入了模拟执行，整个过程就变得非常容易，不再需要影子页

大家看右图，我们将原始页面的RW权限去掉只保留X权限，假设我们的页面地址是0x1000，我们修改头部指令为jmp就可以实现hook。当有cpu执行mov rax,[0x1000]读取页面时，产生异常并vmexit，host接管并模拟执行此指令，模拟涉及到被修改的指令都用原始指令替换，这样cpu读到的就是修改前的sub rsp,88h指令

总结来说就是，涉及非ept异常的页面如mov rbx,[0x2000]，使用真实执行，涉及到ept异常页面，如mov rax,[0x1000]，会产生读取异常并触发模拟执行



模拟器选择

有了前面模拟执行的方案，我们选择什么模拟器呢？unicorn?bochs?

其实KVM本身就自带了一个x86模拟器，我们可以直接使用

具体的代码在arch\x86\kvm\emulate.c

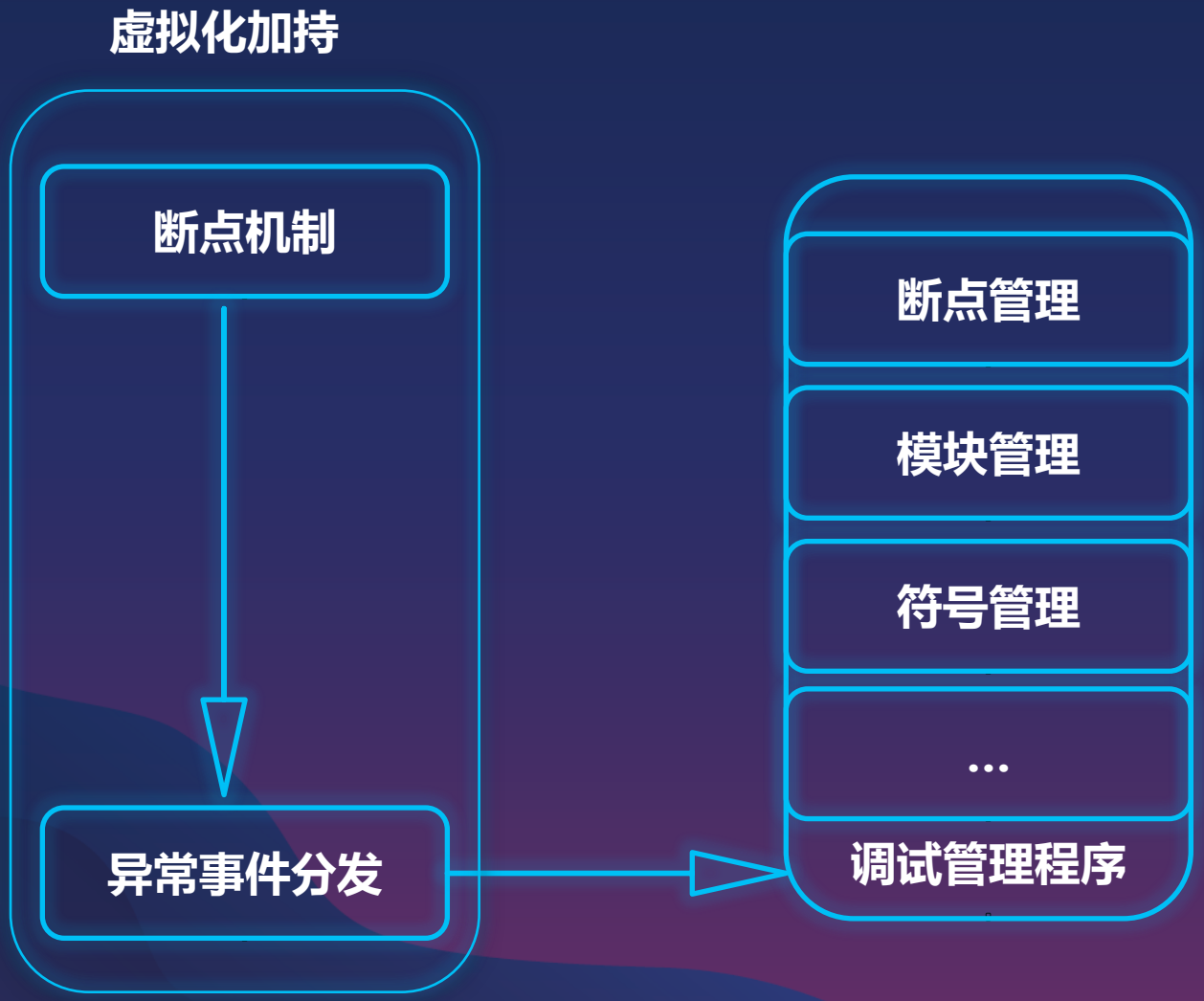
只需要对其中的部分模拟函数进行修改，在读取某些指令时，用原始指令替换即可

这样我们就通过ept + 模拟器实现了无影子页ept hook，解决了影子页存在的问题

基于这种软硬结合的思路，我们还很容易就能实现虚拟化调试器、内核级指令trace等其他工具

虚拟化调试器

什么是虚拟化调试器



如左图，一般的调试器需要有断点机制、异常事件分发、调试管理程序三个部分

这里所说的虚拟化调试器指的就是将调试框架中，易被对抗的部分使用虚拟化来实现，比如断点机制、异常事件分发，使用虚拟化加持后，传统的反调试方法对我们完全无效

异常事件分发涉及的点比较多，我们今天重点来介绍下虚拟化实现断点机制，断点分为软件和硬件断点，我们先来介绍下软件断点

软件断点原理

Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

软件断点在x86中就是指令int3，它的二进制代码opcode是0xCC，当程序执行到int3指令时，会引发异常。随后操作系统会将异常抛给调试器，从而调试器就有了处理断点的机会

软件断点需要写入指令，因此很容易被对抗，常见的对抗方法有crc，函数头部检测等

软件断点检测实例

```
while (1)
{
    auto hFile = CreateFileW(L"a.txt", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    /*
        进行一些写入操作...
    */
    CloseHandle(hFile);
    Sleep(100);

    if (*(uint8_t*)CreateFileW == 0xcc) //检测是否下断点
    {
        MessageBoxA(0, "detected!!!", "detected!!!", 0);
        exit(0);
    }
}
```

如何隐藏软件断点?

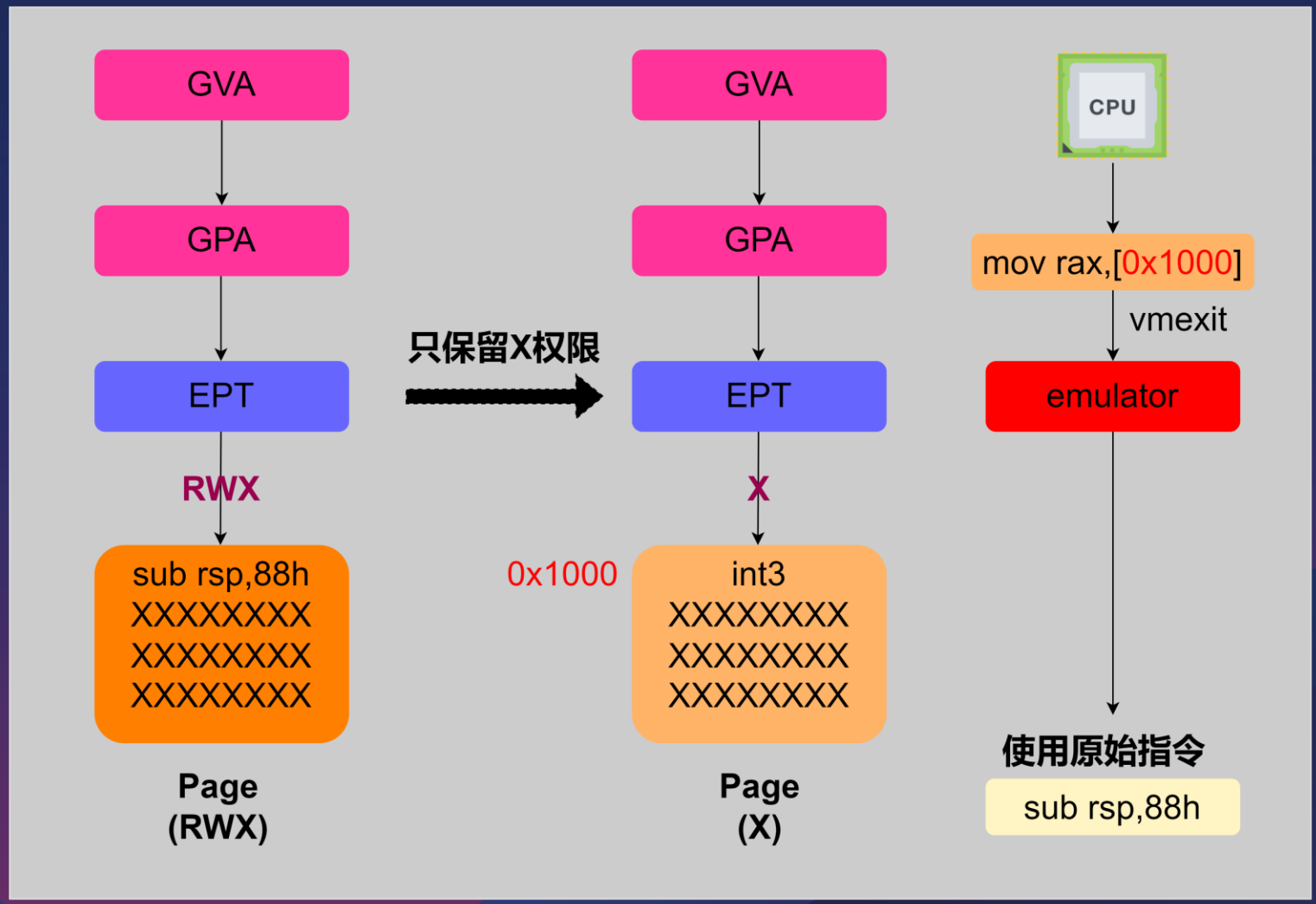
此例子循环的进行一些文件操作，同时会不停的检测CreateFileW头部指令，如果我们在调试时给CreateFileW下软件断点，调试器就会写入0xcc，程序会检测到并弹窗退出

隐藏软件断点

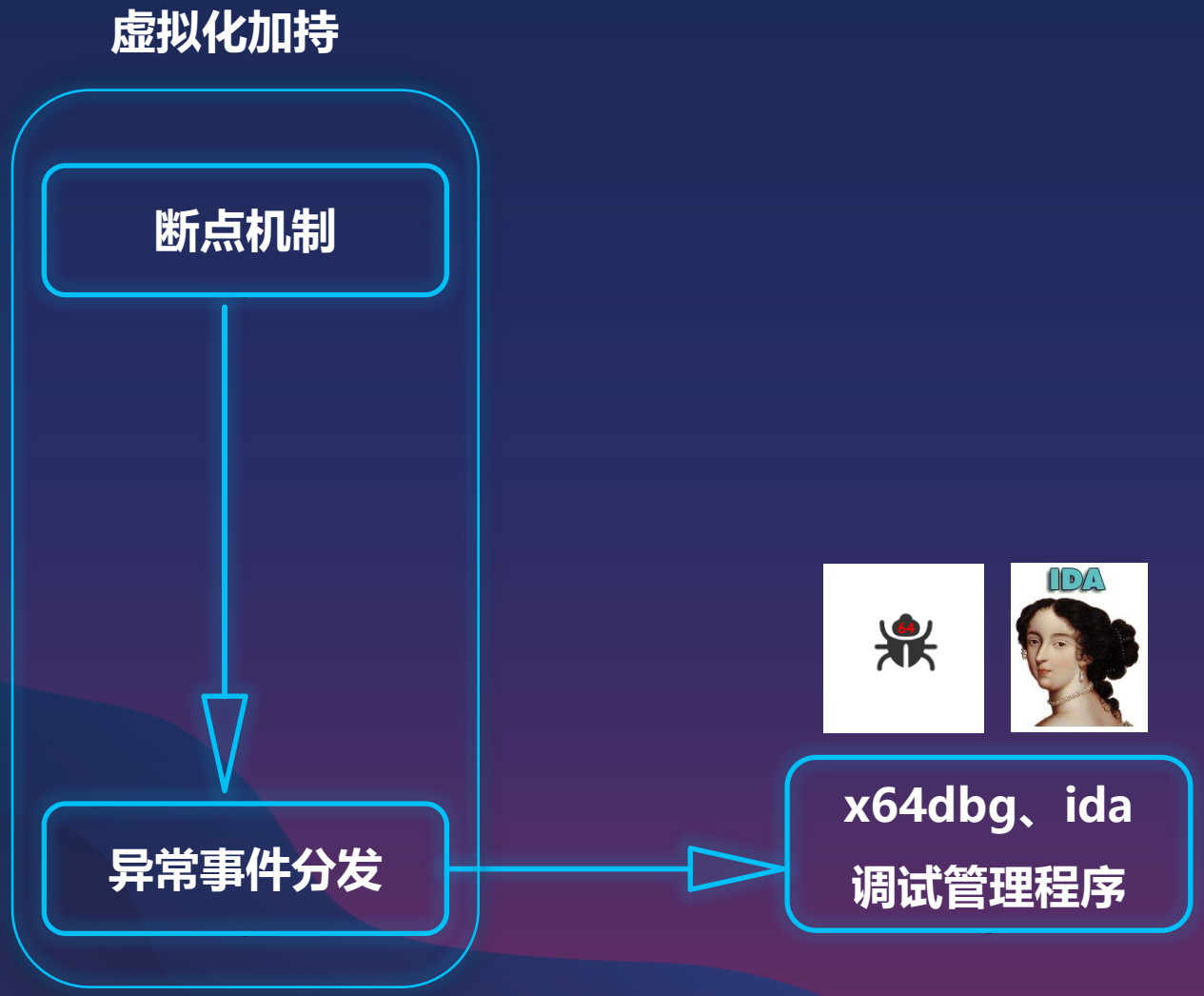
有了前面ept hook模拟执行的方法，隐藏软件断点就变得非常容易，只需要把原来hook插入的jmp修改成int3即可

当cpu执行X页面时，遇到int3会产生异常，调试器可以接收到异常并处理

如右图mov rax,[0x1000]指令去读取0x1000时，由于页面只有X权限，会触发ept violation，被kvm接管后进入模拟执行逻辑，模拟执行会欺骗cpu，返回0x1000中的原始内容sub rsp,88，达到隐藏断点的目的



编写虚拟化调试器



介绍完了隐藏软件断点的原理，我们该如何编写具有隐藏软件断点的调试器呢？

类似HyperDbg这样从头开发？

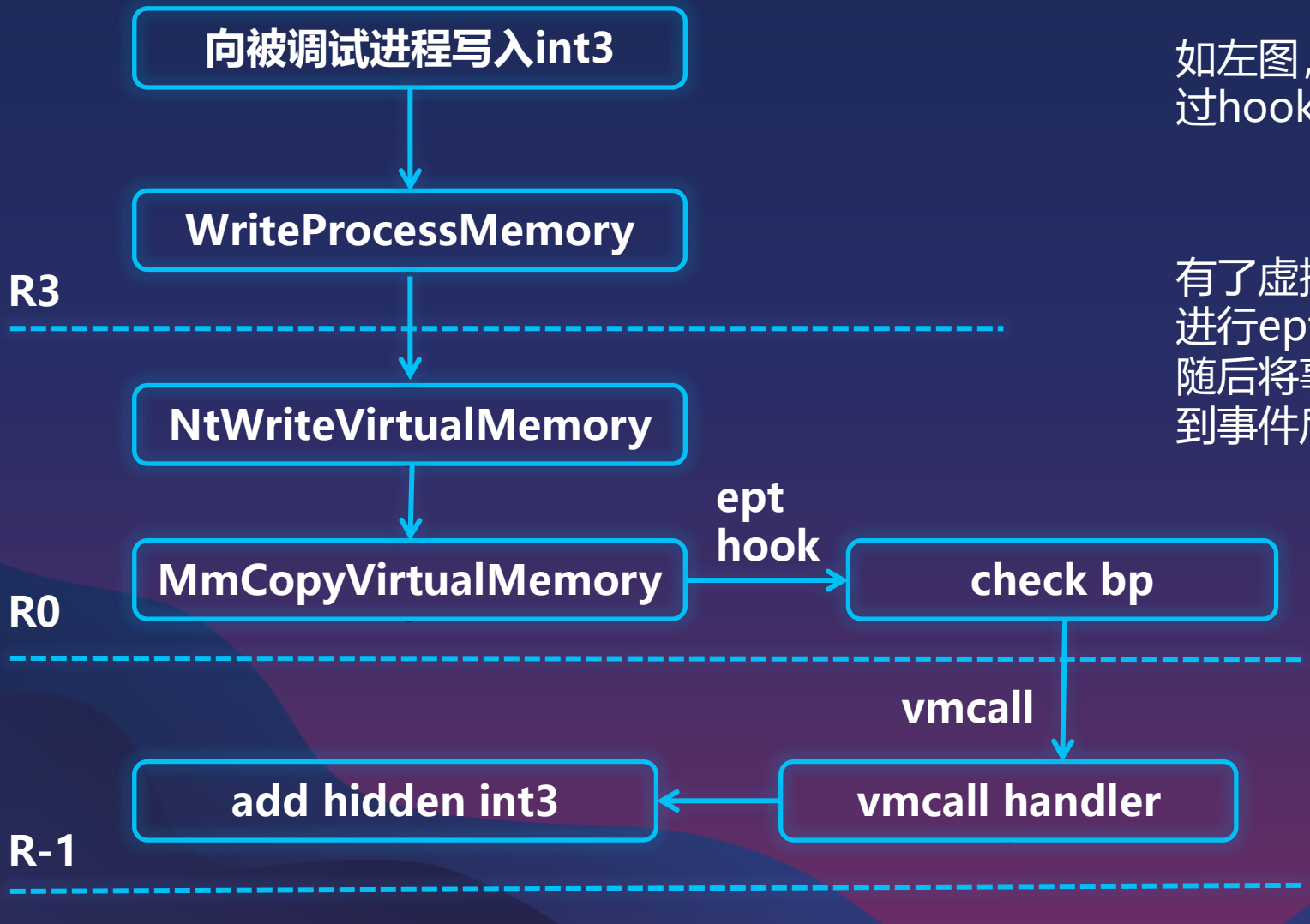
从头开发存在以下问题

- 工作量巨大
- UI交互不如商业化产品友好
- 用户切换成本高

因此我们选择去适配加持市面上已经成熟的调试器如x64dbg,ida，这样一方面切换成本低，一方面更加稳定

x64dbg这类开源调试器很好适配，但是ida等一些调试器不开源，如何在不进行任何patch的情况下进行适配加持？

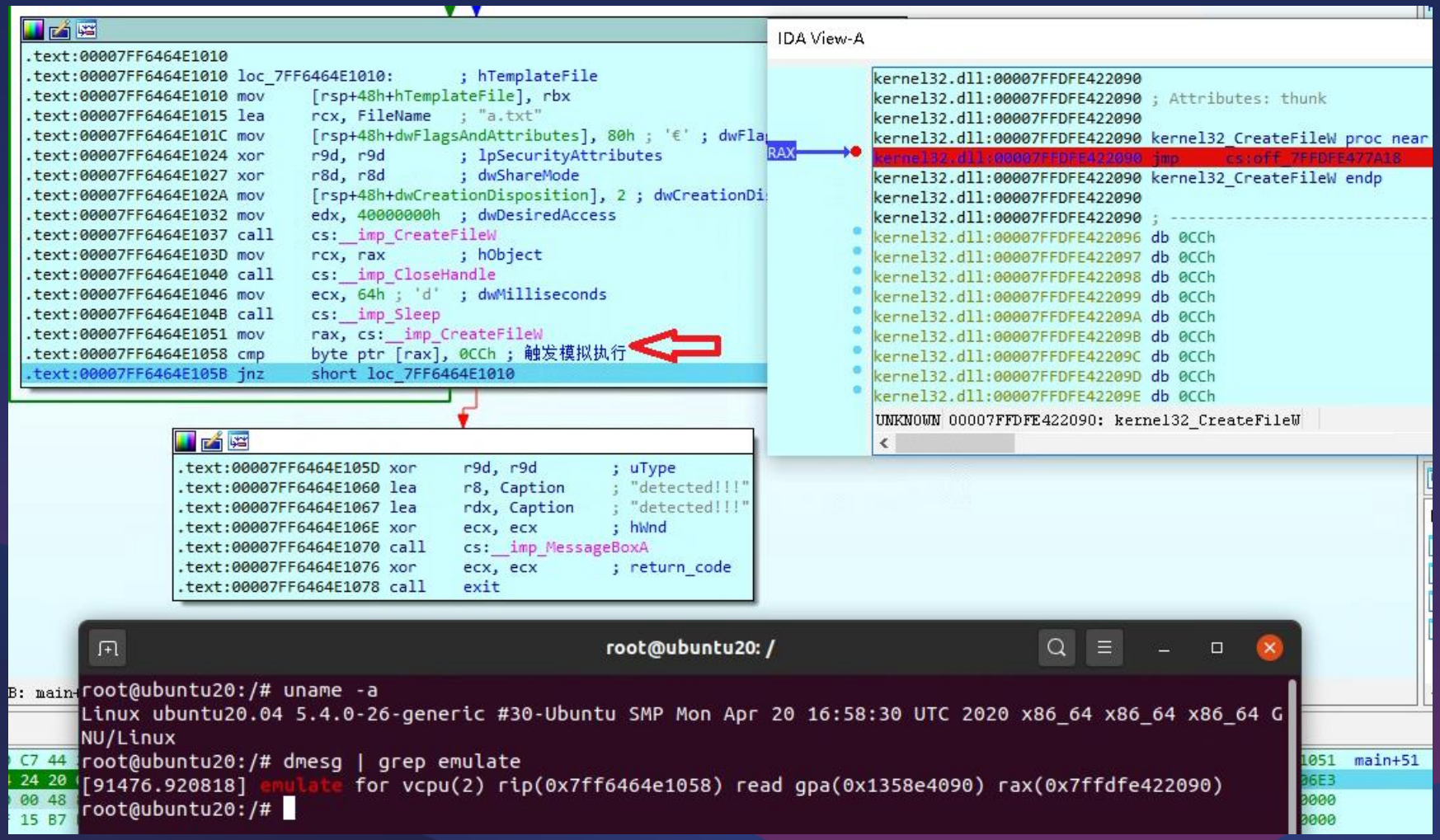
加持现有调试器



如左图，是ida下软件断点的过程，我们可以通过hook捕获到这种行为

有了虚拟化后,我们对MmCopyVirtualMemory进行ept hook，检测是否是调试器在下断点，随后将事件通过vmcall转发给kvm，kvm接收到事件后，将int3断点转化成隐藏断点

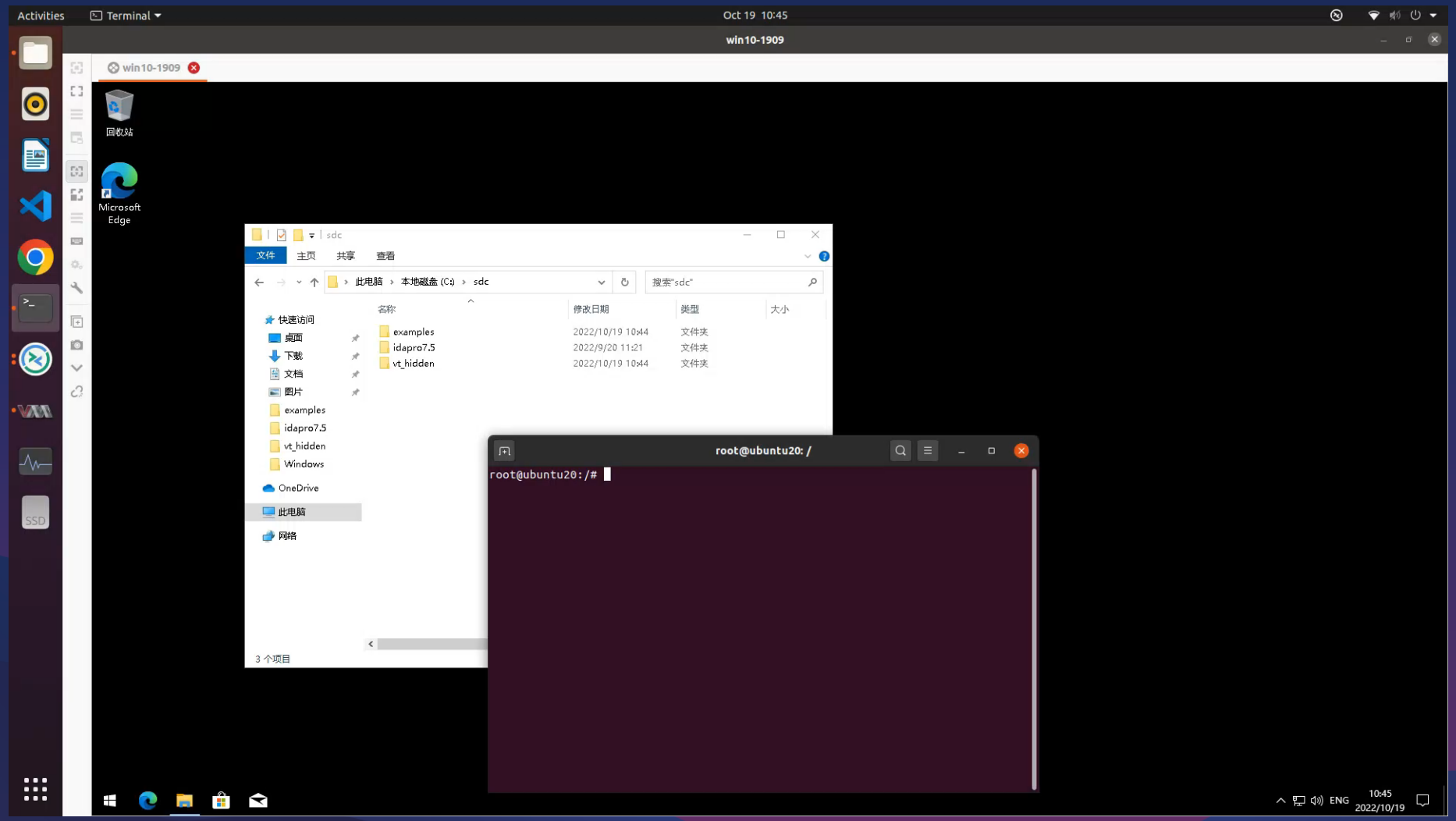
隐藏软件断点演示



左图就是前面软件断点检测的例子，当我们在createfilew下断点后，kvm会感知到并将createfilew所在地址0x7FFDFE422090对应的物理页设置成只保留X权限，并写入0xcc,当0x7FF6464E1058处指令cmp byte ptr[rax],0xcc读取0x7FFDFE422090时，会触发模拟执行，cpu会读取到原始指令

从下面kvm日志中能看到此次模拟执行的rip，vcpu等信息

隐藏软件断点视频演示



只需要将ida等调试器放入vt_hidden目录下，无需任何修改patch，就能获得虚拟化加持，绕过检测

容易踩的坑

• guest 内存被交换到磁盘	mdl锁住内存，防止换页
• copy on write问题	写入一份相同内存的内容，保证当前GVA->GPA唯一性
• qemu 换页导致ept失效	qemu启动命令行添加mlock，防止qemu进程换页

硬件断点原理

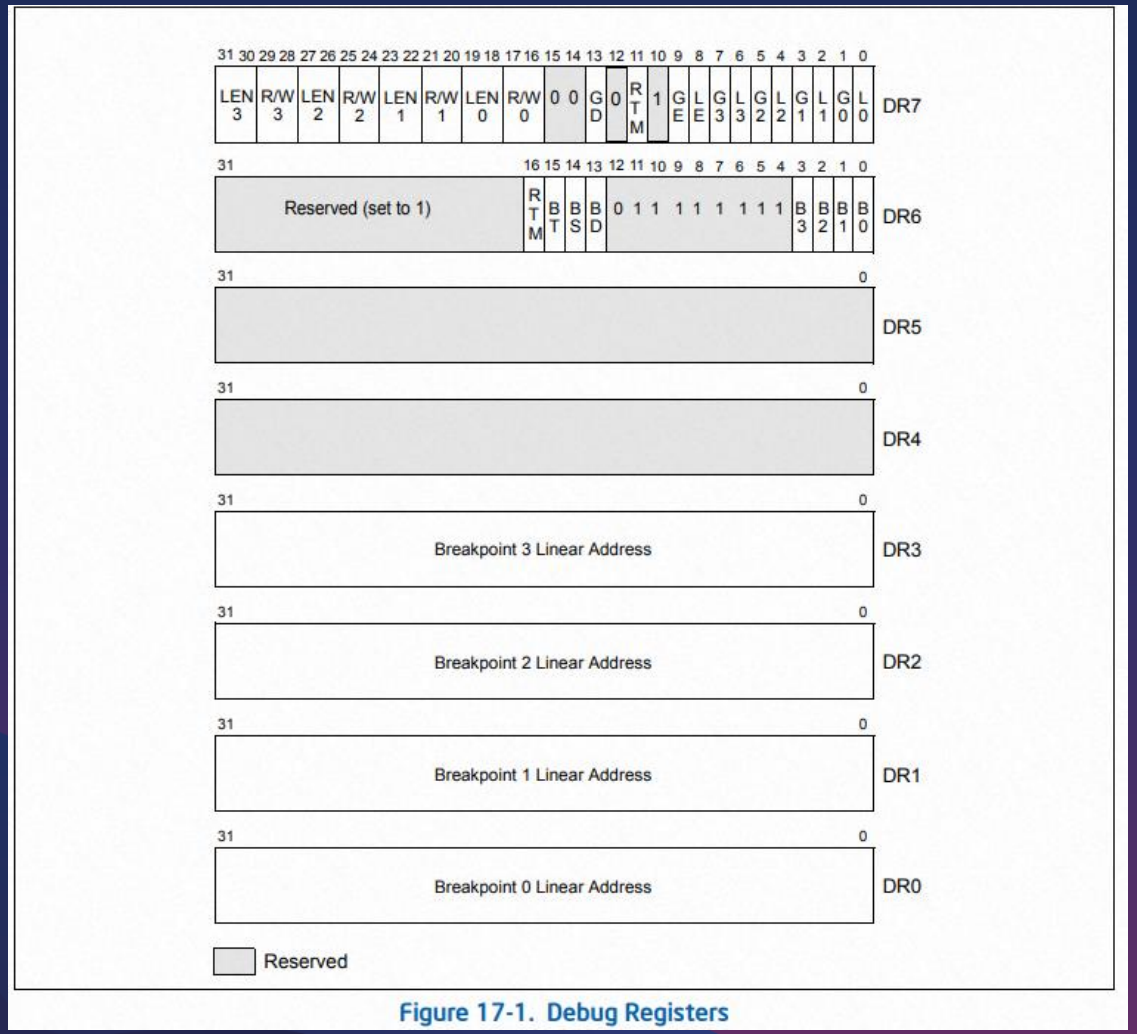


Figure 17-1. Debug Registers

x86提供8个调试寄存器（DR0~DR7）用于硬件调试。其中前四个DR0~DR3是硬件断点寄存器，可以放入内存地址或者IO地址，还可以设置为执行、修改等条件，CPU在执行到满足条件的指令就会自动停下来，一般用于监控数据读写，因此也叫做数据断点。硬件断点十分强大，但缺点是只有四个，同时也比较容易被检测

硬件检测实例

```
static int global_var = 1;
while (1)
{
    auto ref_var = global_var;  //读取变量
    printf("global_var = %d", ref_var);
    Sleep(100);


    CONTEXT ctx = { 0 };
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    GetThreadContext(GetCurrentThread(), &ctx);

    //检测硬件断点
    if (ctx.Dr0 != 0 || ctx.Dr1 != 0 || ctx.Dr2 != 0 || ctx.Dr3 != 0)
    {
        MessageBoxA(0, "detected!!!", "detected!!!", 0);
        exit(0);
    }
}
```

如何隐藏硬件断点?

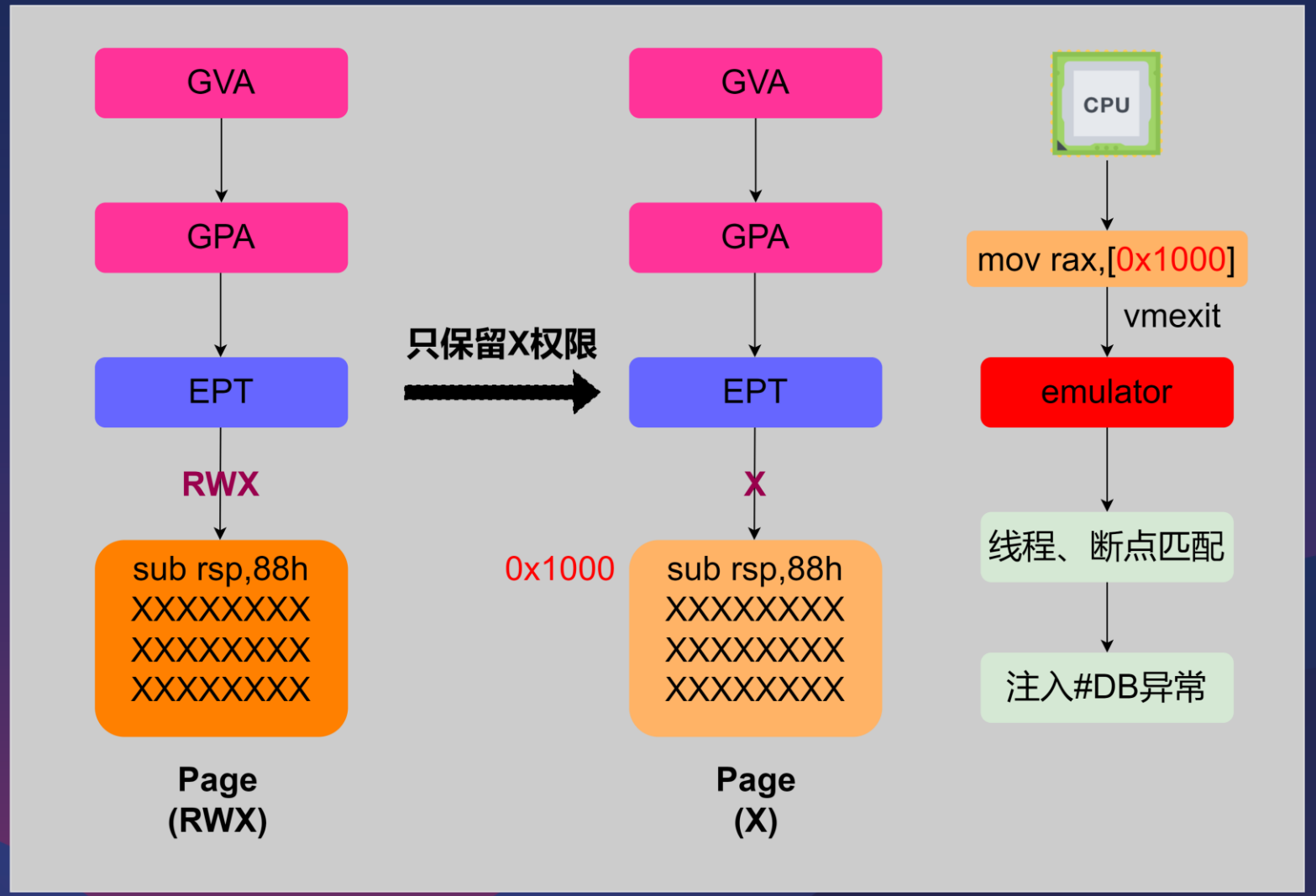
此例子循环读取变量并打印，如果我们在调试时给global_var下硬件读写断点，调试器会将global_var的地址写入Dr0，程序通过GetThreadContext检测到Dr0中存在有效地址，命中硬件断点检测逻辑，进入异常流程，弹窗并退出

隐藏硬件断点

从右图可以看到隐藏硬件断点跟隐藏软件断点原理基本类似，差别在于不需要patch内存，以读写断点为例，只需要去掉页面RW权限。在命中时根据当前线程、断点进行匹配，匹配成功后kvm会向guest注入#DB，x86硬件断点只有4个，用了我们这种模拟实现，可支持“无限硬断”

注入#DB的实现比较简单，直接调用kvm现有的中断注入函数即可

如何获取guest当前线程?



kvm中获取guest当前线程

Guest里面如何获取当前线程?

```
; PETHREAD KeGetCurrentThread(void)
                public KeGetCurrentThread
KeGetCurrentThread proc near            ;
                                        ;
                mov     rax, gs:188h    ;
                retn
KeGetCurrentThread endp
```

以上是windows x64内核获取当前线程的代码，可以看到实现非常简单，直接获取的gs:188h

直观能想到的获取方法

1、kvm中直接执行这段代码?

- 显然不行，gs不是guest的，并且cr3已经切换到host，guest内存空间不可直接访问

2、gsbase + kvm_read_guest_virt?

- gsbase虽然是vmexit时guest的gs，但测试发现它不正确，需要修正
- 修正后我们再用kvm_read_guest_virt去读取，发现依旧读取失败

为什么需要修正？修正后为什么还是读取失败?

失败原因分析

```
KiSystemCall64Shadow proc near          ; DATA X
                                        ; sub_14

var_110         = byte ptr -110h

                swapgs
                mov     gs:9010h, rsp
                mov     rsp, gs:9000h
                bt      dword ptr gs:9018h, 1
                jb      short loc_140A141A4
                mov     cr3, rsp
```

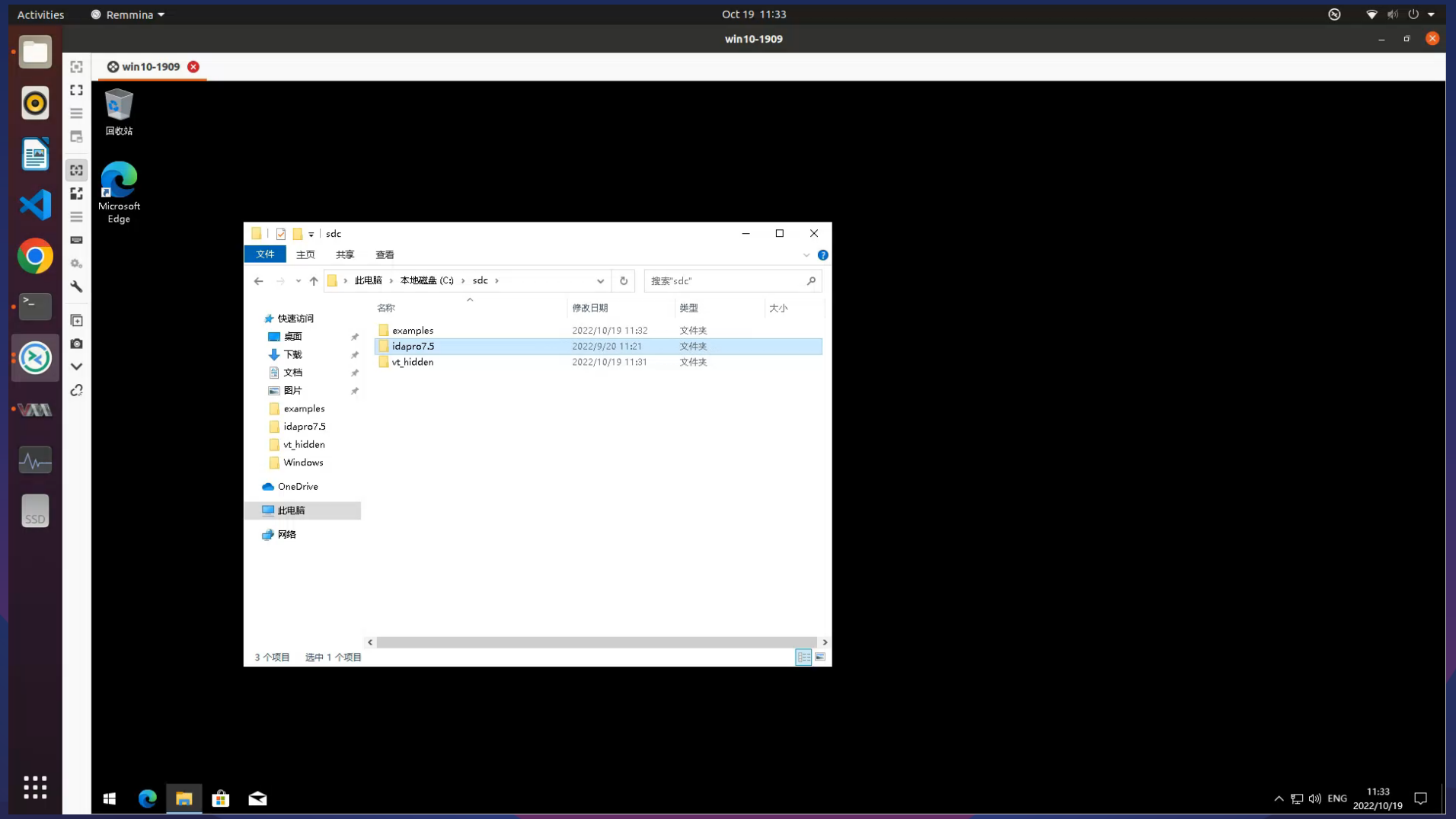
经过研究发现，我们模拟的硬件断点触发vmexit的时机是R3，此时的gs指向teb，只有在R0时，gs才会指向kpcr，KeGetCurrentThread才能正确索引

如左图，KiSystemCall64Shadow是系统调用入口函数，可以看到第一条执行的就是swapgs，这个指令研究过cpu投机执行的应该不陌生，根据intel手册它会将GS.base=MSR.C0000102H(IA32_KERNEL_GS_BASE)

因此我们只需要在kvm中获取MSR.C0000102就能得到正确的gsbase，获取msr有现成的函数vmx_get_msr

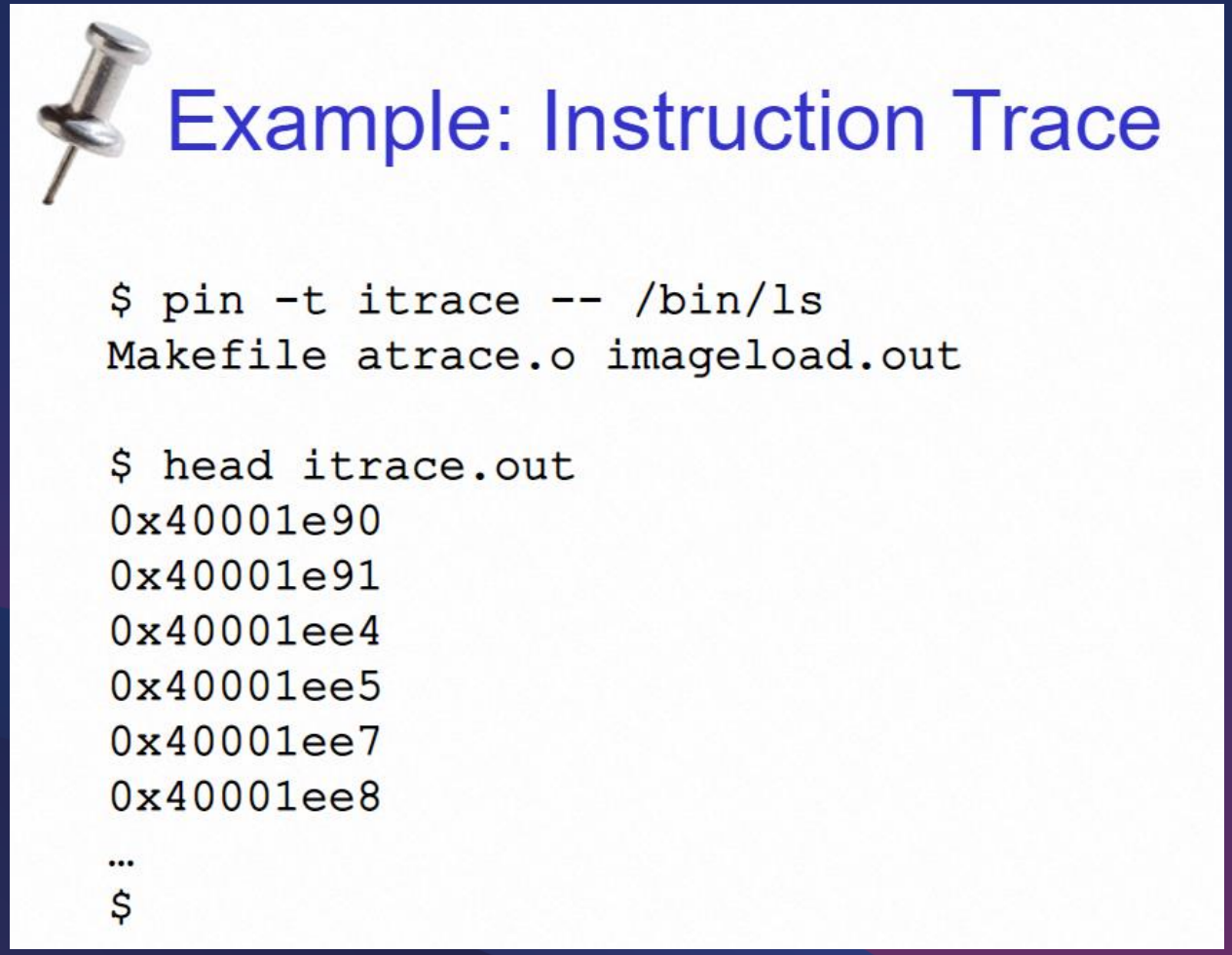
kvm_read_guest_virt失败的原因类似，当vcpu.cpl=3时，会导致这个函数鉴权失败，我们需要用更底层的函数来绕过

隐藏硬件断点视频演示


Activities
Remmina
Oct 19 11:33
win10-1909
win10-1909
回收站
Microsoft Edge
sdc
文件
主页
共享
查看
此电脑 › 本地磁盘 (C:) › sdc ›
搜索"sdc"
名称
修改日期
类型
大小
快速访问
桌面
下载
文档
图片
examples
idapro7.5
vt_hidden
Windows
OneDrive
此电脑
网络
examples 2022/10/19 11:32 文件夹
idapro7.5 2022/9/20 11:21 文件夹
vt_hidden 2022/10/19 11:31 文件夹
3 个项目 选中 1 个项目
ENG
11:33
2022/10/19

内核级trace

插桩实现指令trace

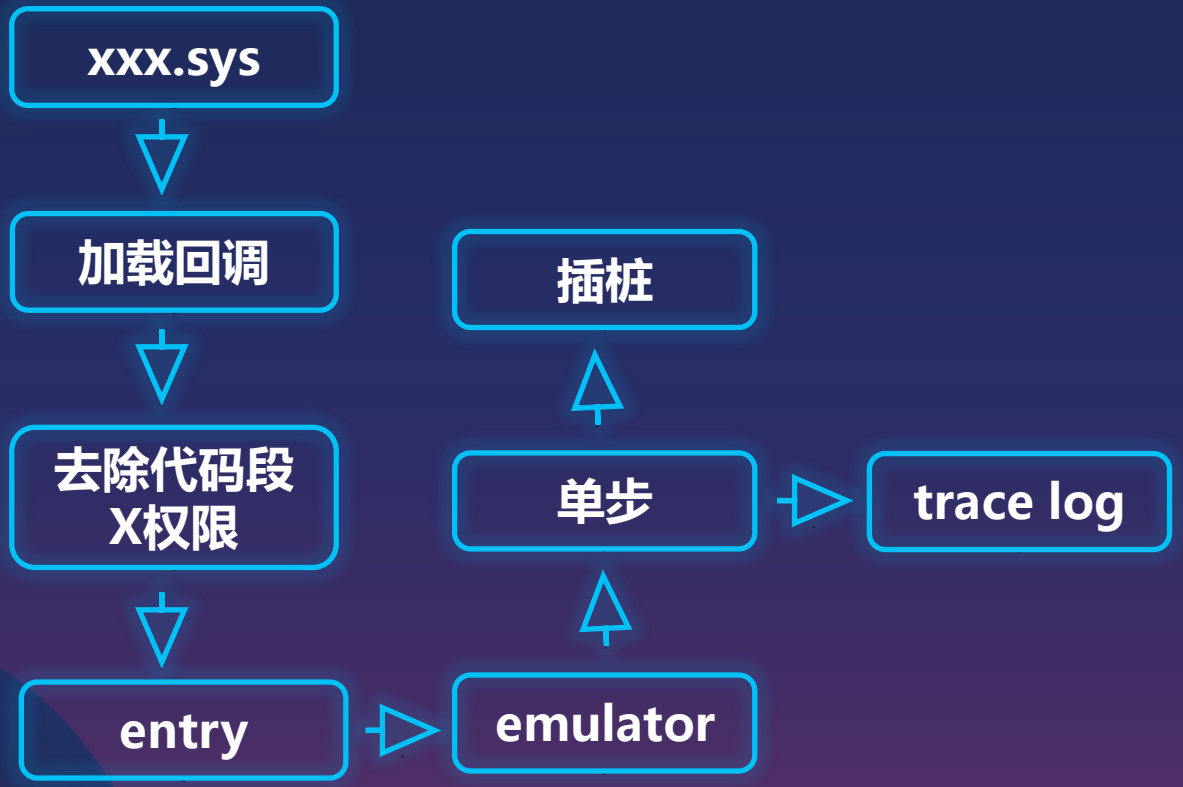


左图是基于intel pin二进制插桩，实现的trace工具，它可以trace出ls程序执行的指令序列

实际的指令trace信息会比图中更全面，在脱壳、去vmp虚拟化、二进制分析等场景都能用到

但是intel pin等常见的二进制插桩工具目前都不支持内核态，我们如何借助硬件虚拟化来实现内核态trace?

内核级trace



整体思路还是真实执行+模拟执行，左边是我们将一个驱动程序xxx.sys生成trace的全过程

也有一些其他生成内核级trace的方法，比如在调试器中生成，纯QEMU模拟生成，对比它们，我们使用EPT+模拟器软硬结合的方法，性能消耗更小，噪音更小，更不容易被对抗

总结

总结

我们介绍了如何基于硬件虚拟化特性，配合模拟器，实现无影子页ept hook，解决了传统方法存在的问题。同时介绍了如何基于qemu\kvm快速打造虚拟化调试器、内核级trace工具

Q/A

欢迎添加我的微信进一步交流!

程聪

ID: kingofmycc
昵称: Ae0LuS