

An abstract graphic featuring three blue circles of varying sizes, each composed of concentric rings. Two thin blue lines intersect at the top left, forming a V-shape that points towards the circles. A large, light gray vertical rectangle is positioned on the right side of the page. The bottom right corner features a large blue circle, partially cut off by the edge of the page.

NewBluePill: 深入理解硬件虚拟机

So many open source projects. Why not Open your Documents?

Superymk
2009-5-1

发布记录

版本	日期	作者	说明
1.00M1	2009-8-25	Superymk	

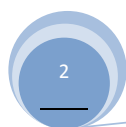
版权说明

本文档版权归原作者所有。

在免费、且无任何附加条件的前提下，可在网络媒体中自由传播。

如需部分或者全文引用，请事先征求作者意见。

如果本文对您有些许帮助，表达谢意的最好方式，是将您发现的问题和文档改进意见及时反馈给作者。当然，倘若有时间和能力，能为技术群体贡献自己的所学为最好的回馈。



目录

前言	6
本书简介	8
一、 概述	10
Hypervisor 概述	10
虚拟化的历史	10
硬件虚拟化技术 (HEV)	10
HEV 技术应用模型	12
已有的 HEV 技术平台介绍	12
SVM	13
AMD IOMMU	16
Intel-VTx	18
Intel-VTd	21
NewBluePill 项目介绍	22
二、 体验 NewBluePill	24
编译 NewBluePill	24
演示 NewBluePill	26
调试 NewBluePill	29
PART1 HEV 技术相关知识	32
三、 深入 HEV 技术细节	32
HEV 下虚拟机启动过程	32
启动过程模型	32
VT 技术下开启虚拟机的过程	35
SVM 技术下开启虚拟机的过程	37
HEV 下虚拟机关闭过程	37
VT 技术下关闭 Hypervisor 和虚拟机的过程	37
SVM 技术下关闭 Hypervisor 和虚拟机的过程	37
HEV 下 #VMEXIT 事件的产生和处理	38
VT 技术下的 #VMEXIT 事件的产生和处理	38
SVM 技术下的 #VMEXIT 事件的陷入和处理	38
HEV 下虚拟机关键数据结构	39
VT 技术下的 VMCS 结构体	39
SVM 技术下的 VMCB 结构体	49
HEV 中的双层地址翻译	52
VT 扩展页表技术	52
SVM 嵌套页表技术	54
总结	55
PART2 深入研究 NewBluePill	59
四、 NewBluePill 的启动和卸载	59
NewBluePill 驱动的启动过程	59
构建私有页表	59
初始化调试系统	60

构建 Hypervisor 并将操作系统放入虚拟机	60
进入 NewBluePill 的世界	61
NewBluePill 驱动的卸载过程	78
拆除 Hypervisor 恢复原宿主机信息	78
关闭调试系统	81
拆除私有页表	81
五、 NewBluePill 内存系统	82
相关文件	82
x64 下的地址翻译	82
NewBluePill 内存隐藏技术	84
内存系统的初始化	84
内存系统的使用	91
内存系统的关闭	92
总结	92
六、 NewBluePill 陷入事件管理系统	93
相关文件	93
Trap 元素的生成、注册机制	93
Trap 元素的触发机制	94
阶段 1 触发	95
阶段 2 触发	95
各处理函数功能和实现	99
VT 技术实现中各处理函数功能和流程	99
SVM 技术实现中各处理函数功能和流程	104
七、 NewBluePill 反探测系统	111
探测 NewBluePill	111
通过指令执行耗时分析	111
通过观察 TLB 变化分析	112
Blue Chicken 策略	112
相关文件	112
功能介绍和详细分析	112
时间欺骗——指令追踪策略	113
相关文件	113
功能介绍和详细分析	113
八、 NewBluePill 调试系统	116
相关文件	116
功能概述	116
实现细节	117
总结	121
PART3 实验部分	123
九、 动手写自己的第一个 HVM 程序	123
实验目的	123
实验概述	123
十、 移植 NewBluePill 到 32 位系统	125
实验目的	125

实验概述.....	125
提示.....	126
十一、 开发基于 HEV 技术的注册码验证器	127
实验目的.....	127
实验概述.....	127
十二、 NewBluePill 完全隐藏了?	129
实验目的.....	129
实验概述.....	129
A. 其它有关 HVM 技术的项目.....	131
Xen	131
KVM.....	131
V3VEE.....	132
PinOS.....	132
BitVisor.....	133
B. 其它安全技术.....	134
Intel TXT 技术	134
SVM 和 TPM 模块的结合.....	134
C. 相关软件和参考文档.....	135
相关软件.....	135
参考文档.....	135

前言

本书是国内外第一本基于源代码详细分析硬件虚拟化实现细节的书籍。

硬件虚拟化技术是一项崭新的技术，说其崭新，不仅是因为其诞生到如今仅有 4 年，更是因为学术界针对其进行的研究才刚刚开始，相关的软件产品和组件更少，只有诸如 VMWare、Xen、Hyper-V 等虚拟机软件利用到了相关的技术，甚至于 Intel 和 AMD 旗下最新的产品也尚未实现硬件虚拟化技术的所有特性。

然而硬件虚拟化技术却有着广泛的前景，不仅在虚拟化领域，在计算机安全、可信计算、调试等领域都有很大发展空间。即便在当下最热门的云计算领域，硬件虚拟化技术也是其基础技术之一。

笔者学习研究硬件虚拟化技术始于 2008 年底，一方面为硬件虚拟化技术的能力所深深吸引，另一方面却发现无论是国内还是国外，没有一本详细深入讲解硬件虚拟化的书籍，材料无非只有 Intel 和 AMD 的手册。对一个初学者而言，过于详细的技术介绍反而成了掌握技术的最大障碍，同时由于硬件虚拟化技术引入了新的指令，在学习过程中不可避免的需要掌握 C 语言，汇编甚至机器码的开发技巧，这一切也让笔者在初涉该领域时颇感头痛。

随着对硬件虚拟化技术的深入了解和一系列相关项目研究，觉得很有必要推广硬件虚拟化技术，降低入门难度，让更多的人了解它并开发利用它，尽管在此过程中，市面上出现了一些介绍硬件虚拟化技术的书籍，但它们只注重于介绍技术本身，或者作为介绍虚拟化技术的补充部分，缺少结合实际代码的分析，因此很难真正的将这一技术推广。鉴于此，笔者决定专以介绍硬件虚拟化技术为主题，选定 NewBluePill 项目作为研究素材写下此书。之所以选定该项目，是因为它体积较小，逻辑清晰，是 CPU Rootkit 的创始之作，以硬件虚拟化技术为基础，同时包含很多新的想法。因此笔者使用该项目全景展现从入门时的探索学习过程，也正是这个过程让笔者自身受益匪浅，从一名仅略懂 x86 和 ARM 汇编入门知识的嵌入式学习者成为了一名 x86、x64 平台硬件虚拟化技术的研究者，这个过程中还学会了 Windows 驱动开发。所以阅读本书不要求读者有多深的相关知识，由浅入深的介绍和探索研究将一步一步带领读者进入硬件虚拟化技术的世界。

为了更好的让读者掌握书中内容，推荐笔者结合 NewBluePill 项目源代码阅读本书。编写本书时取自当时的 0.32 版，当读者看到本书时，可能已不是这个版本，推荐读者尽量下载这个版本的源代码，因为书中所出现代码行号均是依据此版本。同时设计了若干实验穿插于其中，在书的最后还设计了几个大型的有挑战性的实验，读者可以在理解相应章节内容后，结合实验亲身感受硬件虚拟化技术。

本书的主要部分首先宣讲 NewBluePill 的启动、运行和卸载场景，这之后详细讲解 NewBluePill 各个组成模块，通过这样的安排，既便于初学者对硬件虚拟化技术有大体印象，熟悉开发逻辑，又方便了需要深入理解 NewBluePill 的读者对模块细节的需要。同时，本书强调介绍 NewBluePill 的各个重要数据结构，好的数据结构是程序的灵魂，一个定义优秀的数据结构能够使人立刻读懂其中的思路，所以我们更着重于介绍其中所涉及的数据结构，力求用最短的篇幅说明 NewBluePill 的世界。但是读者一定要知道，本书力在引导读者去探索 NewBluePill、探索硬件虚拟化，因此本书并不具体到代码中涉及的每个角落。

对于有志于或正在从事硬件虚拟化技术相关系统设计或实现的读者，笔者建议在阅读本书后一定要关注附录部分，其中涉及到了有联系的技术，截止到初稿定稿时为止的有影响力

的项目和项目分析，以及相关论文和文献。一个项目的成功总是由于他站在了另一个成功项目的肩膀上，本书的另一个目的，就是通过提供对这些已有项目的介绍，从而给开发者提供大量素材，对于需要快速开发的开发者来说，只需通过阅读本书最后项目介绍，找到合适于自己的项目，在许可证权限内剪裁相关文件即可，本书可以为您省去许多到互联网上搜索相关项目的时间，而对于研究者，也可以通过阅读本书获得大量的参考文献来源。

希望读者在每读完一部分后能做个小结，记录下重要的数据结构和 NewBluePill 对该模块的处理流程。同时，建议读者反复阅读此书，对于初学者，建议在第一遍阅读本书时先跳过“Part 1 HEV 技术相关知识”部分，在阅读 NewBluePill 流程分析时再对照该部分的叙述，结合代码明白其原因和实现方法，而在再次阅读本书或利用本书进行硬件虚拟化技术应用开发时，着重关注这一部分，获得最具体的细节分析。我们相信，读者在阅读此书过程中，一定会有一种在导游陪同下探索一个新的岛屿，最后站在一个高峰俯瞰全岛的感觉。

虽然在成稿前后经过了反复修改，发现了很多错误并予以更正，但是我们仍然无法保证本书已经没有错误。我们只能说，所有已发现的错误都已改正，读者在阅读过程中发现任何错误，欢迎与我们联系，我们欢迎您的批评指正。

在本书的成书过程中，特别感谢戚正伟老师和徐昊所给的指导帮助和对本书的审核工作，如果没有他们大量无私的帮助，笔者无法在短时间内掌握 NewBluePill 项目的核心思想并如此深入的探索研究虚拟化技术，他们对新技术敏锐的眼光以及应用新技术为我国可信计算研究和产品作出的贡献和决策，令人钦佩。同时，对上海交大虚拟机项目组在编写本书期间所给的鼓励、支持和审核工作表示感谢，他们在本书的编写过程中提供了大量的硬件设备以供实验，并且通过与他们的讨论使得笔者更快的理解了 VT 下 NewBluePill 很多模块的处理过程。

此外，还要感谢星月王者、金毅、顾玉婷，在成书过程中，他们利用业余时间贡献了很多很好的思路和资料，同时他们也为本书部分实验和过程分析付出了大量的辛勤劳动，由衷的感谢你们所有人。

最后感谢编辑人员

批注 [S1]: “Part 1 HEV 技术相关知识”

批注 [S2]: 最后感谢编辑人员

于淼

2009-8-22

本书简介

《NewBluePill: 深入理解硬件虚拟机》是专门为学习研究硬件虚拟化技术的读者准备的一本书。本书入门简单, 具备 x86 基础和嵌入式设备程序开发经验的读者即可阅读本书, 通过阅读本书, 读者能够熟悉硬件虚拟化技术并能在其基础上开发相关应用程序。对于研究者而言, 本书附录中出现的论文和书籍提供了丰富的引用资源; 而对于开发者来说, 本书中对相关项目的分析和相关技术的介绍提供了丰富的开发素材。同时对所有读者, 本书结合 NewBluePill 项目源码和穿插实验将快速带您进入硬件虚拟化技术的世界。

本书结构

实验部分

本书的实验部分需要您在被调试机器上安装 64 位 Windows 操作系统, 并且在调试机上安装 Windbg 调试工具。在本书中, 各章节内的实验均出现在“实验”标题方框中, 其中包含了完成实验的具体步骤, 示例输出和简要分析。我们建议读者按照书中步骤完成各个实验, 以加深印象。在本书最后还有几个大型实验, 只是提供了完成实验需要注意的地方和部分示例输出, 并未随书提供相关源代码, 鼓励读者独立完成实验。

未涵盖课题

硬件虚拟化技术包含了很多方面, 本书由于是通过讲解 NewBluePill 项目源码, 因此并未涉及到某些硬件虚拟化技术的细节部分, 比如 VT-d, EPT 的具体细节, 以及 Intel® TXT 技术的使用详情, 这些部分在诸如 VMWare Workstation 6.5.1 和 Xen 较高版本中都有所体现。

如果读者确实需要了解这些方面的内容或者需要更深探索硬件虚拟化技术, 请参考 Intel 和 AMD 相关文档描述。

警告

本书所分析的 NewBluePill 项目源码是 0.32 公开版本, 作者 Invisible Things Lab, 所使用 Intel 手册是 2009 年 3 月版, AMD 手册是 2007 年 9 月版 (Rev 3.14)。在读者阅读本书时, 很可能已经可以下载到更新的源代码和手册。推荐读者下载上文所列代码和手册版本, 以求和书中内容和引用的统一。

此外, 由于 NewBluePill 是研究型项目且运行在系统底层, 在运行时难免存在漏洞。读者在做相关实验时请务必提前关闭任何打开的文档, 以避免可能的由于系统崩溃造成的数据丢失, 如果确实造成损害, NewBluePill 项目作者和本书作者对此不承担任何责任。

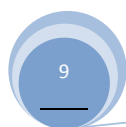
书中表示法

本书涉及的表示方法:

<i>Windows Internals, 4th Edition</i>	书名
Hypervisor	英文名词
http://www.bluepillproject.org/	超级链接
HvmSubvertCpu()	书中代码及函数名

支持

对于在本书中遇到的任何技术性错误或说明不准确的地方,或者更深层次讨论硬件虚拟化和相关技术,我们欢迎您发送信件到 superymkvmm@hotmail.com, 或者访问 <http://nbpdiscovery.spaces.live.com/> 获取最新动态并留下您的评价。



一、概述

在这一章中，我们先介绍一些贯穿全书的概念，比如 Hypervisor，VT-x，VT-d，SVM 等等。然后我们会简略介绍下 NewBluePill 项目背景及其所采用的硬件虚拟化技术。

这一章只是介绍这些技术大致的轮廓，详细内容会在后面各章节中逐一介绍。

Hypervisor 概述

虚拟化的历史

在讨论 Hypervisor 之前首先谈谈虚拟，虚拟（virtualization）指对计算机资源的抽象，一种常用的定义是“虚拟就是这样的一种技术，它隐藏掉了系统，应用和终端用户赖以交互的计算机资源的物理性的一面，最常做的方法就是把单一的物理资源转化为多个逻辑资源，当然也可以把多个物理资源转化为一个逻辑资源（这在存储设备和服务器上很常见）”

实际上，虚拟技术早在 20 世纪 60 年代就已出现，最早由 IBM 提出，并且应用于计算技术的许多领域，模拟的对象也多种多样，从整台主机到一个组件，其实打印机就可以看成是一直在使用虚拟化技术的，总是有一个打印机守护进程运行在系统中，在操作系统看来，它就是一个虚拟的打印机，任何打印任务都是与它交互，而只有这个进程才知道如何与真正的物理打印机正确通信，并进行正确的打印管理，保证每个 job 按序完成。

长久以来，用户常见的都是进程虚拟机，也就是作为已有操作系统的一个进程，完全通过软件的手段去模拟硬件，软件再翻译内存地址的方法实现物理机器的模拟，比如较老版本的 VMWare, VirtualPC 软件都属于这种。

在 2005 年和 2006 年，Intel 和 AMD 都开发出了支持硬件虚拟技术的 CPU，也就是在这时，x86 平台才真正有可能实现完全虚拟化¹。在 2007 年初的时候，Intel 还进一步的发布了 VT-d 技术规范，从而在硬件上支持 I/O 操作的虚拟化。随着硬件虚拟化技术越来越广泛的采用，开发者也开始虚拟技术来做一些其他的事情：当前 HVM 已经在虚拟机，安全，加密等领域上有所应用，例如 VMware Fusion, Parallels Desktop for Mac, Parallels Workstation 和 DNGuard HVM，随着虚拟化办公和应用的兴起，相信虚拟化技术也会在未来得到不断发展。

硬件虚拟化技术（HEV）

有了虚拟技术的基本概念，下面我们谈谈硬件虚拟化技术。硬件虚拟化技术（Hardware Enabled Virtualization，本书中简称 HEV），也就是在硬件层面上，更确切的说是在 CPU 里（VT-d 技术是在主板上北桥芯片支持），对虚拟技术提供直接支持，并通过这种设计提高虚拟效率、降低开发难度。在硬件虚拟化技术诞生前，编写虚拟机过程中，为了实现多个虚拟机上的真实物理地址隔离，需要编程实现把客户机的物理地址翻译为真实机器的物理地址。同时也需要给不同的客户机操作系统编写不同的虚拟设备驱动程序，使之能够共享同一真实硬件资源。硬件虚拟化技术则在硬件上实现了内存地址甚至于 I/O 设备的映射，因此大大简化了编

¹ 完全虚拟化(Full Virtualization)，完整虚拟底层硬件，这就使得能运行在该底层硬件上的所有操作系统和它的应用程序，也都能运行在这个虚拟机上。

写虚拟机的过程。而其硬件直接支持二次寻址和 I/O 映射的特性也提升了虚拟机在运行时的性能。¹

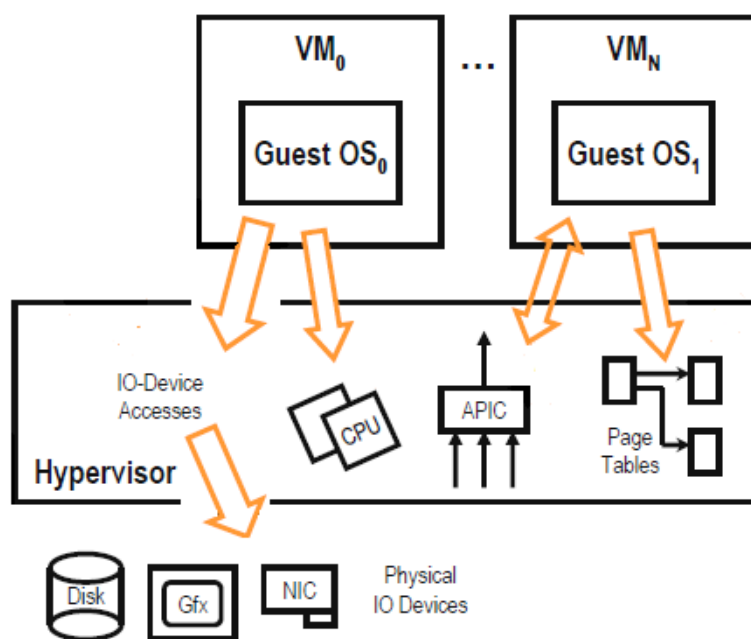


图 1.1 硬件虚拟化技术架构示意图

在硬件虚拟化技术中，一个重要的概念就是 HVM（Hypervisor Virtual Machine²），在使用硬件虚拟化技术时它会创建出特权层，该层提供给虚拟机开发者，用来实现虚拟硬件与真实硬件的通信和一些事件处理操作（如图 1.1），因此 Hypervisor 的权限级别要高于等于操作系统权限。

¹ 一些优化技术也在硬件中被采用，比如专门用于二次寻址的 TLB，详细信息可以参考 Intel 和 AMD 的手册

² 在本书中简称为 Hypervisor

HEV 技术应用模型

批注 [S3]: 还是叫“最佳实践”？

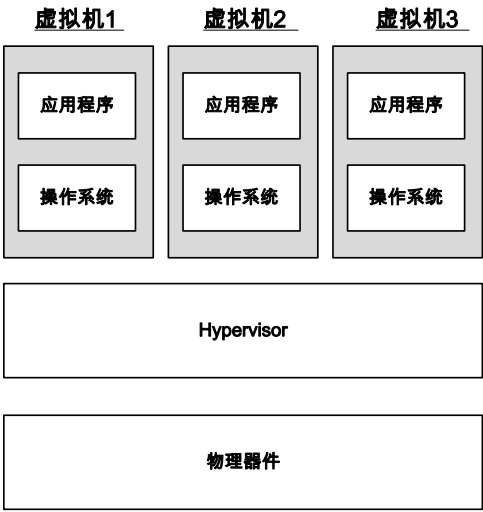


图 1.2 Hypervisor 使用架构图

前文中已经提过,Hypervisor 层的权限要高于等于操作系统的权限。操作系统的内核态已经处在了 Ring0 特权级上，因此 Hypervisor 层实际上要运行在一个新的特权级别上，我们称之为“Ring -1”特权级。同时需要新的指令，寄存器以及标志位去实现这个新增特权级的功能。

作为一种最佳实践方案，一般 Hypervisor 层的实现都是越简单越好。一方面，简单的实现能够尽量降低花在 Hypervisor 上的开销¹，毕竟大多数这些开销在原先的操作系统上是不存在的。另一方面，复杂的程序实现容易引入程序漏洞，Hypervisor 也是如此，且一旦 Hypervisor 中的漏洞被恶意使用，由于其所处特权级高于操作系统，将使隐藏在其中的病毒、恶意程序很难被查出。

已有的 HEV 技术平台介绍

现今两大主要硬件厂商 Intel 和 AMD 均以推出了支持硬件虚拟化技术的产品，两者大体功能和实现方法近似（意料之中，因为两家公司在过去你死我活的市场拼斗中，每次也都是实现功能和方法类似，只不过名字不同罢了）。下面我们简略介绍下这两家公司的支持 HEV 技术的平台，读者可以首先对这两种平台有概念。后面的章节中会有对这两个平台技术细节的更详细描述。

¹ 关于 Hypervisor 的开销问题，后面的章节会有介绍

SVM

概述

AMD 芯片支持硬件虚拟化的技术被称作 AMD-V (在技术文档中也被称为 SVM,其全称是 AMD Secure Virtual Machine, 在本书中我们仍称其为 SVM)。其主要是通过一组能够影响到 Hypervisor 和 Guest Machine (客户机, 下文简称 Guest) 的中断实现的。AMD-V 技术设计目标如下:

- 引入客户机模式 (Guest Mode)¹
- Hypervisor 和 Guest 之间的快速切换
- 中断 Guest 中特定的指令或事件(events)
- DMA 外部访存的保护
- 中断处理上的辅助并对虚拟中断 (virtual interrupt) 提供支持
- 新的嵌套页表用来实现地址翻译
- 一个新的 TLB (其实就是一个 Cache) 来减少虚拟化造成的性能下降。
- 对系统安全的支持

新的客户机模式 通过 VMRUN 指令即可进入这种新的处理器²模式, 当进入客户机模式后, 为了辅助虚拟化过程, 一些 x86 汇编指令的语义会发生变化。

外部访问保护 过去客户机 (Guest) 可以直接访问选定的 I/O 设备。现在硬件上已经实现这样的安全功能, 能够阻止某个虚拟机拥有的某个设备访问其它虚拟机的内存。

中断上的支持 为了辅助中断的虚拟化, 下列各项现在已经得到硬件支持, 并且可以通过配置 VMCB 结构体³的方法使用

- 1) 拦截物理中断分发 (Intercepting physical interrupt delivery) 发生在物理硬件上的中断能够让虚拟机发生一个中断, 陷入 Hypervisor, 从而使得 Hypervisor 可以首先处理这个中断。
- 2) 虚中断 (Virtual Interrupts) Hypervisor 能够将提供给客户机 (Guest) 一套虚拟的中断机制。它是这样实现的, Hypervisor 会给这个客户机复制出来一份 EFLAGS.IF 用做中断屏蔽位 (Interrupt Mask Bit), 同时复制 APIC⁴中的中断优先级寄存器提供给客户机, 从而客户机就会去操纵这套假的中断机制, 而不是直接去操纵物理中断。
- 3) 共享物理 APIC AMD 的 SVM 技术能够允许多个 Guest 共享同一物理 APIC, 同时又能保护这个 APIC 以免某个客户机不慎或恶意的在未经其它客户机许可的情况下, 将可接收中断优先级设置为高优先级, 从而清空了所有其它 Guest 的中断。

¹ x86 上原有处理器模式包括保护模式(Protected Mode), 管理模式(SMM), 实模式(Real Mode)

² 注意: 本书中强调处理器 (Processor) 和 CPU 的差别, 文中若无特别说明, 处理器指逻辑处理器 (Logical Processor)。在多核时代, 一个 CPU 上可能会有多个核, 而在操作系统视角中, 一个核才是一个逻辑处理器, 因此通过操作系统查看的逻辑处理器数量往往大于真实 CPU 的数量, 并且逻辑处理器才是能够运行 Hypervisor 的基础。

³ VMCB 结构体, Virtual Machine Control Block, 也称 VMCB 控制块, Intel 的相应结构体名称为 Virtual Machine Control Sector, VMCS。这个控制块用于通知物理 Processor 要拦截的事件, 以及在进出 Hypervisor 上下文切换时保存 Hypervisor 和 Guest 的各项寄存器, 后面的章节中会有对这个结构体的详细介绍

⁴ APIC, Advanced Programmable Interrupt Controller, 高级可编程中断控制器, 第三章有关于此主题内容。

被标记的 TLB (Tagged TLB) 为了降低 Guest 模式和 VMM 模式切换开销，，TLB 上新加了一个 ASID 标记 (Address Space Identifier)，这个标记可以区分 TLB 上的一块地址是 Hypervisor 范围内的地址还是 Guest 的地址，从而加速了地址翻译。

安全方面的支持 现在提供的安全方面的支持主要是利用和 TPM 模块 (Trusted Platform Module)¹ 的交互，基于与安全 Hash 值的比较。

在 SVM 技术下的 Hypervisor 生命周期如图 1.3 所示：

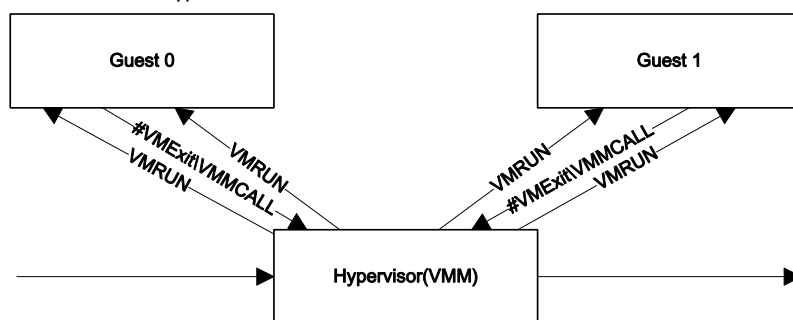


图 1.3 SVM 技术中 Hypervisor 的生命周期

软件开启虚拟机管理器 (VMM) 运行环境后，通过使用 VMRUN 指令使得目标系统正式运行在虚拟机中。当某条指令产生了 #VMEXIT 事件或者虚拟机显式的发出了 VMCALL 指令后会陷入 VMM 中。待其处理完这个事件，可以继续通过 VMRUN 指令将控制权移交回虚拟机。最后在某个时刻，Hypervisor 必须手动拆除 Hypervisor 环境并恢复相关 MSR 寄存器状态，Hypervisor 才会被关闭。

新的地址翻译机制

AMD 引入了新的地址翻译技术——嵌套页表翻译 (Nested Page Table, NPT)，用于支持两级地址翻译，这样就使得虚拟机管理器不用再自己软件维护一套影子页表²

¹ TPM 模块会在“附录 B 其它安全技术”一章中介绍。

² 影子页表 (Shadow Page Tables)，常见于过去传统的虚拟机管理器中，由于存在从 Guest 线性地址到 Guest 物理地址，从 Guest 物理地址到真实物理地址两层地址翻译，所以在过去一般是要虚拟机软件自己维护两套页表去做这样的地址翻译。

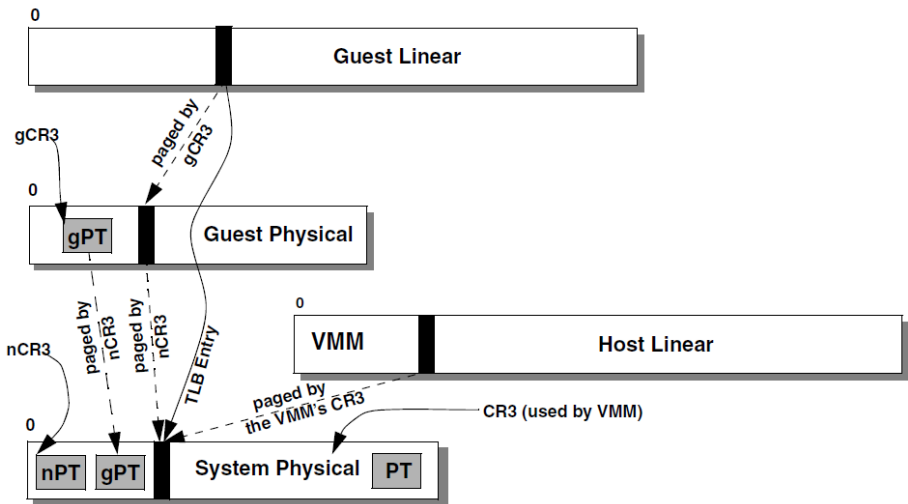


图 1.4 嵌套页表翻译地址过程¹

嵌套页表翻译地址的过程如图 1.4 所示，这套机制的实现允许了从 Guest 线性地址到真实物理地址的翻译，也允许了在 Hypervisor 范围内的 Host 线性地址到真实物理地址的翻译。同时专门附加了一个 TLB 缓冲区，用于缓存从 Guest 线性地址到真实物理地址的映射，从而提升了虚拟机的运行性能。

Note 关于嵌套页表技术的详细解释会在“第二章 深入 HEV 技术细节”一章中介绍，完整的描述请参考 *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*, Chapter 15.24 Nested Paging

批注 [S4]: “第二章 深入 HEV 技术细节”
章号
批注 [S5]: 记得所有书名用斜体

相关结构和汇编指令

在 SVM 中，VM 控制块被称为 VMCB (Virtual Machine Control Blocks)，其信息主要分为两块，第一块是控制信息存储部分，同时也包含是否允许拦截某特定异常的遮罩(interception enable mask)，Guest 中不同的指令和事件都能以修改 VMCB 中相应控制位的方法拦截，SVM 支持的两类主要的拦截是异常拦截和指令拦截，第二部分则是 guest 的状态信息保存，这里会保存段寄存器以及大部分的虚拟内存的入口控制寄存器，不过浮点寄存器信息不会被保存。需要注意的是 VMCB 在不同的处理器间不共享，并且 VMCB 一定要保证是在 4K 页对齐的连续物理内存空间中。

SVM 中主要的指令有以下这些：

- **VMLoad** 从 VMCB 加载 guest 的状态，VMCB 与 guest 是有对应关系的。
- **VMMCall** 通过该方法 guest 可以与 VMM 显式的交流，方法是利用生成 #VMEXIT 从 guest 层退到 VM 层。
- **VMRun** 加载 VMCB，并开始执行 guest 层的指令，VMCB 的物理地址将通过 RAX 获得，这个 VMCB 对应于要执行的 guest

¹ 此图摘自 AMD64 Architecture Programmer's Manual, Volume 2: System Programming

- **VMSAVE** 存储处理器状态的子集到VMCB中，这个VMCB的物理地址由RAX寄存器给出。
- **STGI** 用于设置全局中断标志（Global Interruption Flag）为1，这个指令属于Secure Virtual Machine。
- **CLGI** 用于设置全局中断标志（Global Interruption Flag）为0，同样这个指令属于Secure Virtual Machine。
- **INVLPGA** 使得TLB上一个ASID和一个虚拟页(Virtual Page)之间的映射关系无效，这个指令属于Secure Virtual Machine。
- **SKINIT** 安全的重新初始化CPU，使得CPU可以开始执行一段受信任的程序（trusted software）其方法是将该代码进行安全的哈希比较(secure hash comparison)。这也就是开发者可以开发一个更安全的VMM loader。这种安全手段可以在TPM模块的帮助下发挥更大作用
- 改进的MOV指令 现在的MOV指令可以直接读写CR8寄存器（任务优先级寄存器 Task Priority Register），因此可以用来提高SVM应用的性能。

基于 SVM 的 Hypervisor 开发逻辑

其实由上文的描述可以看出，开发基于SVM的Hypervisor最主要是编写一个循环，这个循环要包含VMRUN命令以便从VM层启动一个Guest虚拟机，也要包含一段程序用于处理当#VMEXIT发生后的异常情况，这其中可能要手动做一些必要的保存现场和恢复现场的工作，具体造成异常的起因等均可通过读取VMCB中的数据获得。不过SVM没有提供一个显示终止Hypervisor的指令，因此若有需要，则要用其它方法关闭Hypervisor。NewBluePill中对SVM的支持就是这样实现的，我们会在深入探究NewBluePill的章节中详细展示怎样使用这些指令。

AMD IOMMU

简介

AMD IOMMU 在 SVM 技术基础上进一步加入了对外围设备访存的管理和保护，尤其是DMA 操作，这也就使得外围设备的虚拟化成为可能。AMD IOMMU 技术主要带来以下三方面的好处：

- 在 64 位平台上对先前 32 位设备的直接支持(过去需要多次移动数据以解决内存限制)
- 保护模式下用户模式应用程序对指定外围设备的安全访问
- 虚拟机中客户操作系统对指定外围设备的安全访问

为了达到这样的目的，IOMMU 需要如下三个数据结构：

I/O 页表（I/O Page Tables），用于在 I/O 设备访存时提供权限检查和地址翻译。

设备映射表（Device Table），用于指定某个设备属于哪个域（Domain），并指定要使用的 I/O 页表。

中断重映射表（Interrupt Remapping Table），用于到来中断的权限检查和中断重映射。

其实 IOMMU 非常类似于 MMU，都是在进行权限检查和地址映射，只是在 AMD 平台上，由于一个 AMD IOMMU 只针对经过它的 DMA 和中断进行处理，因此每条总线上必须有一个

IOMMU。

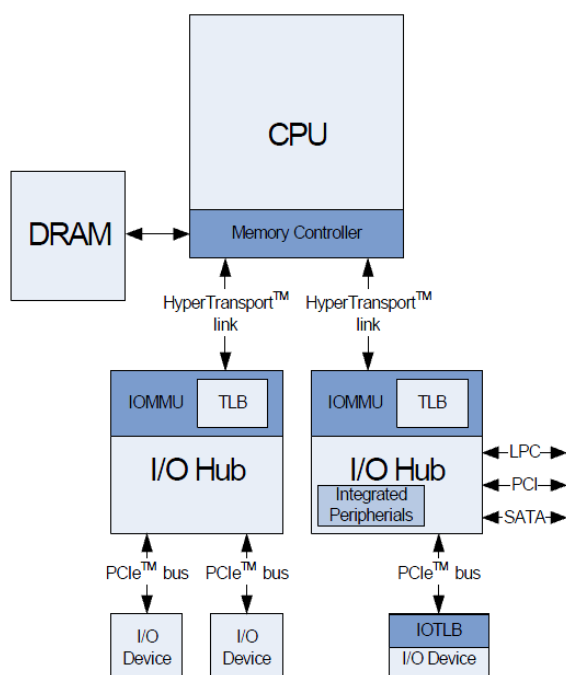


图 1.5 支持 AMD IOMMU 功能的平台架构图¹

此外，对于 AMD IOMMU 来说，它只关心来自外围设备的请求并对其进行权限检查和地址翻译，对于其它请求，比如设备发现和配置，来自内存的传入数据请求等，AMD IOMMU 不会进行任何处理。

使用模型

AMD IOMMU 包含域（Domain）的概念，在使用时，IOMMU 将设备分配到指定的一个域，并指定要对其使用的 I/O 页表。之后，IOMMU 会拦截这个 I/O 设备读写系统内存的操作，并对请求地址进行权限检查和地址翻译。

需要注意的是，AMD IOMMU 现在不支持外围设备的共享，也就是说，一个特定的设备在某个时刻当前只能属于唯一域，由于 I/O 操作大多比较慢，因此不能在同一设备被不同虚拟机使用时会发生问题。

¹ 该图摘自 AMD IO Virtualization Technology (IOMMU) Specification

Intel-VT_x

概述

Intel 芯片支持硬件虚拟化的技术被称为 Intel VT 技术 (Intel® Virtualization Technology)。与 SVM 一样, 其主要也是通过一组能够影响到 VMM 层和 Guest Machine 的事件实现的。

在 VT 技术中, 与 SVM 类似的, 设计架构上同样存在两种角色——虚拟机管理器 (Virtual Machine Monitors, VMM) 和客户机 (Guest), 两者分处在 VMX root 模式和 VMX non-root 两种模式下。VT 技术的设计目标是:

对于 VMM 层: (进入此层则代表进入了 VMX root 模式)

- 为每个虚拟机提供虚拟处理器, 并且可以在恰当的时候把它放在真正的物理处理器上, 从而使得这个虚拟处理器可以处理指令。
- VMM 层可以控制处理器资源, 物理内存, 管理中断和 I/O 操作

对于 Guest Machine: (进入此层则代表进入了 VMX non-root 模式)

- 每个虚拟机使用相同的接口来使用虚拟处理器, 内存, 存储设备等资源
- 每个虚拟机可以独立的不受干扰的运行, 虚拟机间都是相互独立的
- 对于虚拟机来说, VMM 层像是完全透明的。

在 VT 技术下的 Hypervisor 生命周期如图 1.6 所示:

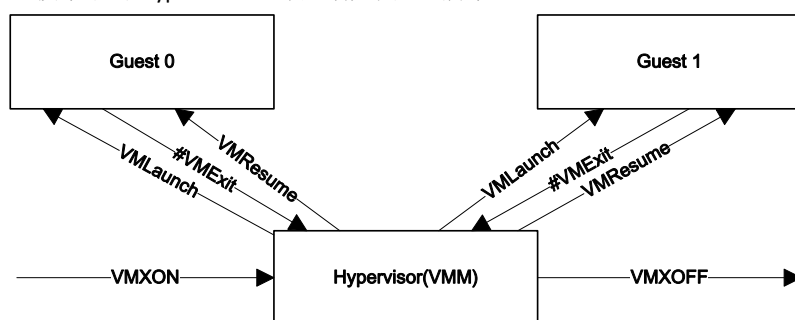


图 1.6 VT 技术中 Hypervisor 的生命周期

图示表明, 软件通过执行 VMXON 指令进入 VMX Root 模式下, 开启了虚拟机管理器的运行环境。然后通过使用 VMLaunch 指令使得目标系统正式运行在虚拟机中。当某条指令产生了 #VMEXIT 事件后, 会陷入虚拟机管理器中, 待其处理完这个事件, 可以通过 VMXResume 指令将控制权移交回发生 #VMEXIT 事件的虚拟机。直到某个时刻, 在 Hypervisor 中显示的调用了 VMXOFF 指令, Hypervisor 才会被关闭。

新的地址翻译机制

Intel 同样引入了新的地址翻译技术——扩展页表翻译 (Extended Page Table, EPT), 用于支持两级地址翻译。

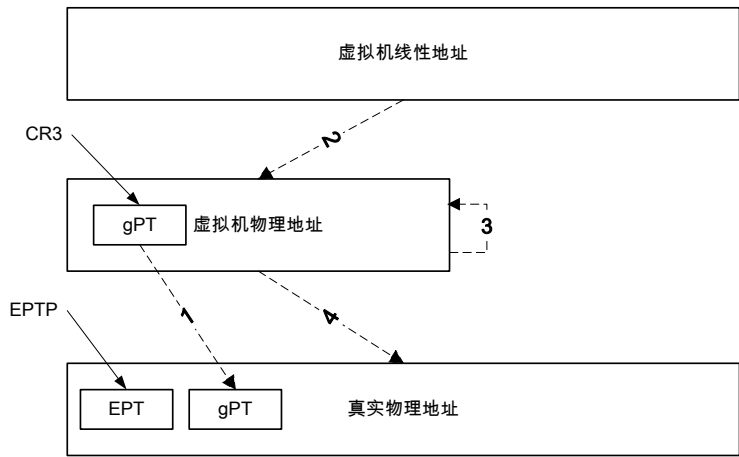


图 1.7 嵌套页表翻译地址过程

扩展页表翻译地址的过程如图 1.7 所示，与 SVM 技术下的嵌套页表翻译机制极其相似，这套机制也允许了从 Guest 线性地址到真实物理地址的翻译，并利用 TLB 提升了虚拟机的运行性能。

Note 关于扩展页表技术的详细解释会在“第四章 深入 HEV 技术细节”一章中介绍，完整的描述请参考 Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3B, Chapter 24. Support for Address Translation

相关结构和汇编指令

在 VT 技术中，VM 控制块被称为 VMCS（Virtual Machine Control Structure）。VMCS 包括三个组成部分：

表 1.1 VMCS 区域的组成部分

Byte 偏移量	内容
0	VMCS 版本标志（Revision Identifier）
4	VMX 退出原因指示器（VMX-abort indicator） ¹
8	VMCS 数据区

如表 1.1 所示，VMCS 区域的前四个字节用于 VMCS 版本标志，不同的 VMCS 格式对应的版本号也不同，而这个物理处理器可以加载的 VMCS 结构体的版本号会存储在 MSR 寄存器中，因此这样的设计也就给未来发展留下了空间。

在 VMCS 数据区中，主要有如下几个组成部分：

表 2.2 VMCS 数据区主要组成部分

名称	作用
----	----

¹ 如果在 VM Exit 的时候遇到问题，就会发生 VMX Abort，一旦发生，那么这个逻辑处理器会进入关闭状态（Shutdown State）

虚拟机状态保存区 (Guest State Area)	当发生了#VMEXIT 事件时虚拟机当前状态保存于此, 在重新进入虚拟机的时候再利用此处的数据恢复虚拟机的状态
宿主机状态保存区 (Host State Area)	当发生了#VMEXIT 事件时宿主机的状态利用此处数据恢复
虚拟机运行控制域 (VM Execution Control Fields)	此处数据定义了虚拟机在什么情况下发生#VMEXIT 事件, 对 VMX non-root 模式有影响
VMEXIT 行为控制域 (VM Exit Control Fields)	此处数据定义了了在#VMEXIT 事件发生时要做的附加工作 (比如保存调试寄存器, 加载全局性能控制寄存器等这些工作)
VMEntry 行为控制域 (VM Entry Control Fields)	此处数据定义了在发生#VMEntry 事件时 (通常是因为调用了 VMResume 汇编指令) 要做的附加工作。
VMEXIT 相关信息域 (VM Exit Information Fields)	此处数据在发生#VMEXIT 事件时自动记录了发生原因和该事件的具体种类。这个域是只读的

VMX Abort 和 VMCS 数据区结构和用法会在后续章节中详细介绍。

VT 技术在设计时注明, 没有任何标志位用于区分一个逻辑处理器 (Logical Processor) 当前正在执行 VMX root 模式下的指令还是执行的 VMX non-root 模式下的指令, 这就确保了 Hypervisor 对虚拟机完全透明——因为虚拟机无从判断它当前是否运行在一个虚拟机下。最后要注意的是 VMCS 同样要求保证是在 4K 页对齐的连续物理内存空间中。

VT 中主要的指令有以下这些:

维护VMCS结构体的指令

- **VMPTRLD** 参数为VMCS块的物理地址。该指令用来激活一块VMCS。修改该处理器的当前VMCS指针 (Current-VMCS Pointer) 指向传入的VMCS物理地址, 并且激活该VMCS, 如果要维护一块VMCS则必须先激活该VMCS。(否则不能用这些指令来维护)
- **VMPTRST** 用来存储当前VMCS指针 (VMCS块物理地址) 到指定位置。
- **VMCLEAR** 该指令用来使一块VMCS变为不活跃状态。该指令将标记为已启动状态 (Launch State) 的VMCS设置为不活跃状态 (Inactive State/Clear State) 并且更新该VMCS块所有区域信息并确保写入VMCS块内存中 (这也就把对应虚拟机和Hypervisor的最新信息同时写入到VMCS块中), 如果带操作的VMCS块就是当前VMCS指针所指向的VMCS块, 那么该指针会被设置为无效地址
- **VMREAD** 通过指定的VMCS Encoding从当前VMCS块中读取一个参数。
- **VMWRITE** 通过指定的VMCS Encoding从当前VMCS块中写入一个参数。

与虚拟机管理器有关的指令

- **VMCALL** 这条指令用于Guest和Hypervisor进行通信。执行该汇编指令会产生一个#VMEXIT事件, 从而使得可以陷入Hypervisor中。
- **VMLAUNCH** 这条指令用于启动当前VMCS指针所指的一个虚拟机, 并且移交控制权给Guest。
- **VMRESUME** 这条指令用于从Hypervisor中恢复虚拟机的执行, 并且移交控制权给Guest。
- **VMXOFF** 这条指令用于关闭Hypervisor。在下次执行VMXON开启Hypervisor前不得执行虚拟机相关汇编指令。
- **VMXON** 这条指令用于处理器进入VMX模式下, 执行该指令后也就可以运行

Hypervisor。传入的参数必须是4K页对齐的物理地址，这段内存用于支持后续VMX相关的操作。

VMLAUNCH 和 VMRESUME 指令的异同

VMLAUNCH 和 VMRESUME 命令都是将控制权移交到虚拟机，那么两者的区别呢？

两者运行的时机不同！

1. VMLAUNCH 指令会检查当前 VMCS 的启动状态是不是不活跃状态（相应标记位清空）。成功运行结果是该 VMCS 被标记为已启动状态。
2. VMRESUME 指令会检查当前 VMCS 的启动状态是不是已启动状态

所以，必须利用 VMLAUNCH 指令启动一个虚拟机。以后的某个时候，因为 VMEXIT 事件而陷入 Hypervisor 中，这个时候要恢复虚拟机的运行则要利用 VMRESUME 指令，正如图 1.6 所示。

管理VT相关的TLB的控制指令

- INVEPT 这条指令用于EPT地址翻译中，使TLB中缓存的地址映射失效
- INVVPID 这条指令用于在TLB中使某个VPID对应的地址映射失效

基于 VT 的 Hypervisor 开发逻辑

利用 VT 技术开发 Hypervisor 的过程不同于利用 SVM 技术的开发过程，最主要的差别是在 VT 技术中，Guest 和 Hypervisor 下面要运行的指令地址（RIP/EIP）是可以在 VMCS 中设置的，同时 Hypervisor 就是用于处理 VMEXIT 事件，因此就像现代操作系统为系统调用设置一个统一入口，并将入口地址存入 MSR 寄存器一样，在 VT 中，通常也将 Hypervisor 的这个入口 IP 设置为事件处理函数入口地址（Event Dispatcher Address）。在事件处理的最后，又通过一个 VMXRESUME 指令统一的返回到 Guest 的指令执行流程中。对于事件发生信息，同样可以通过读取 VMCB 中相应数据获得。NewBluePill 中也有对 VT 技术的支持，我们同样会在深入探究 NewBluePill 的章节中详细展示怎样使用这些指令。

Intel-VTd

简介

类似于 AMD IOMMU，Intel 推出了 VT-d 虚拟化平台¹来对 DMA 和中断进行虚拟化。在 Intel 的实现中，同样是针对来自外部设备的 DMA 和中断请求进行处理，需要利用 I/O 页表和中断重映射表等数据结构，并且不提供设备的共享支持。

然而，支持 Intel-VTd 的平台架构却不同于 AMD IOMMU，前者在北桥中加入了 VT-d 功

¹ 针对 Core i7 的 I/O 虚拟化支持内部称为 vt-d2，它对 vt-d 各项特性进行了强化，更方便于虚拟机使用外部设备。

能（如图 1.8 所示），从而不必针对每条总线加入一个 IOMMU 硬件。

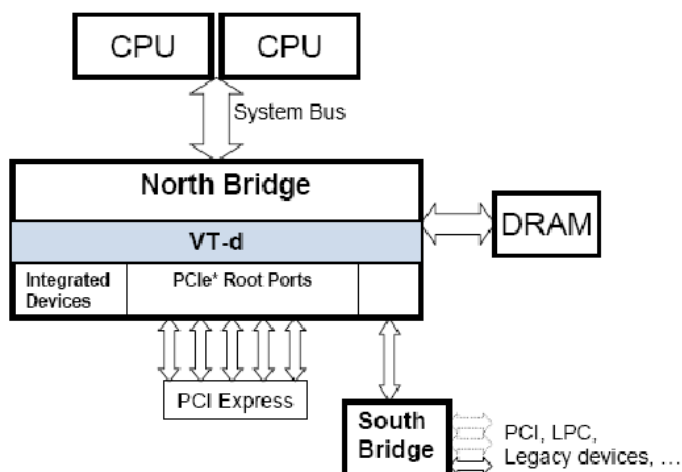


图 1.8 支持 VT-d 功能的平台架构图¹

NewBluePill 项目介绍

NewBluePill 项目 (<http://www.bluepillproject.org/>) 诞生于 2007 年，由 Invisible Things Lab 开发，现在对外公开的版本是 nbp-0.32-public 版本。该项目从 2007 年第三季度以来得到了 Phoenix 公司的大力支持。²

该项目目的是证明并实现这样一种软件隐藏模型：

- 不用已有的方法的隐藏自己
- 即使它的隐藏方法众所周知，其它软件也不能探测到它
- 即使它的实现代码众所周知，其它软件也不能探测到它

可以看出，该目的与非对称密码的设计目的有异曲同工之妙。该项目充分发掘并利用利用 HEV 功能。当前在公开版本上已经实现的功能包括：

- 支持 SVM 和 VT-x 技术构建 Hypervisor
- 在操作系统运行时刻动态加载和卸载，因此在操作系统完全不知情的情况下将操作系统放入了虚拟机中继续运行。
- 在 AMD 平台上支持嵌套 Hypervisor
- 一套自己的页表，用于实现内存隐藏，因此在操作系统中无法访问到 NewBluePill 的内存
- 反 Hypervisor 探测技术（Anti-Hypervisor Detector）：RDTSC 欺骗
- 反 Hypervisor 探测技术（Anti-Hypervisor Detector）：组织可信时间源的检测（Blue-Chicken 技术）

在其未公开的版本上，实现的功能包括：

- 在 Intel VT-x 平台上实现嵌套 Hypervisor

¹ 该图摘自 Intel® Virtualization Technology for Directed IO

² Phoenix 公司的虚拟化技术产品 HyperSpace，具体信息可以参考网上资料。与之类似的还有华硕公司（Asus Inc.）的 Instant On 技术

这些特性最终使得即使 NewBluePill 后于操作系统启动/停止，操作系统却完全无法感知到 NewBluePill 的存在，这也就是 NewBluePill 的主要设计目标（如图 1.9 所示）。

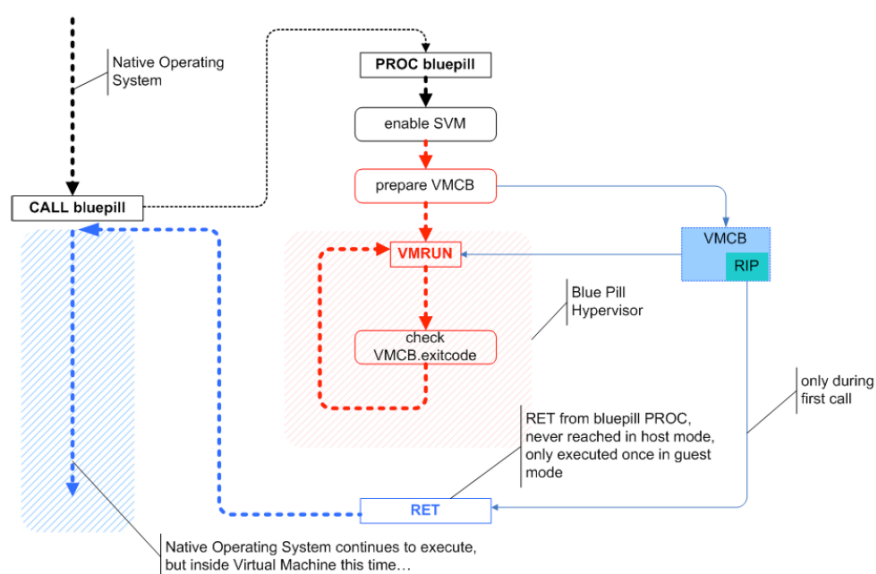


图 1.9 NewBluePill 的实现目标及思路¹

版权信息

本书引用 NewBluePill 代码版权归属于 Invisible Things Lab

/*

* Copyright holder: Invisible Things Lab

*

* This software is protected by domestic and International

* copyright laws. Any unauthorized use (including publishing and

* distribution) of this software requires a valid license * from the copyright holder.

*

* This software has been provided for the educational use

* only during the Black Hat training and conference. This

* software should not be used on production systems.

*

*/

¹ 本图来源 *Subverting Vista™ Kernel For Fun And Profit*, Joanna Rutkowska

二、 体验 NewBluePill

前面我们对硬件虚拟化技术和市场上主流硬件虚拟化技术进行了分析,并简要分析了硬件虚拟化技术模型。为了后面更好的理解 NewBluePill 各个模块的功能和交互关系,同时保证实验的顺利进行,在这一章中我们将介绍编译和运行 NewBluePill 的过程,并借此展开 NewBluePill 实际运行中的神奇体验。

首先介绍下笔者的平台,在整个项目中用了两台计算机

PC1 (调试机): Intel Core 2 6300, 1G RAM, XP SP2 (x86) + windbg + WDK6001.18001

PC2(被调试机): Intel Core 2 6300, 1G RAM, Windows Server 2008 Beta 1(X64), NewBluePill 只能运行于这台机器上。

编译 NewBluePill

通过阅读本书前面部分的介绍,是不是很想亲自动手体验下呢?不过先别急,“工欲善其事,必先利其器”,还是先把工具准备好再说。

工具一共有下面几个:

1. Windbg
2. DebugView¹
3. InstDrv²
4. Windows Driver Kits (WDK 6001.18001)

总体来说编译 NewBluePill 的过程很简单。

步骤 1. 首先确保手上有有了 nbp-0.32-public.zip 这个代码³。然后解压缩到一个根目录,在这里我们假设是 D 盘。目录结构应该是这样的:

¹ DebugView, 可以到 <http://download.sysinternals.com/Files/DebugView.zip> 下载

² InstDrv, 可以到 <http://dl2.csdn.net/fd.php?i=23314208212665&s=0affa2ecb56fc0dcc14cff07345a388e> 下载

³ NewBluePill 项目源代码可以从 <http://www.bluepillproject.org/> 下载

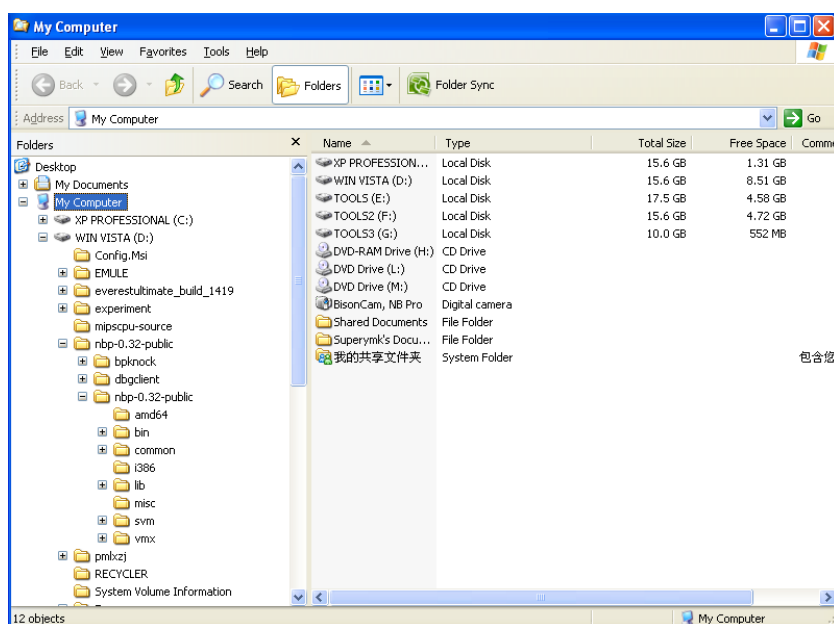


图 3.1 NewBluePill 项目目录结构

步骤 2. 然后打开 Launch Windows Vista and Windows Server 2008 x64 Checked Build Environment 编译环境¹:

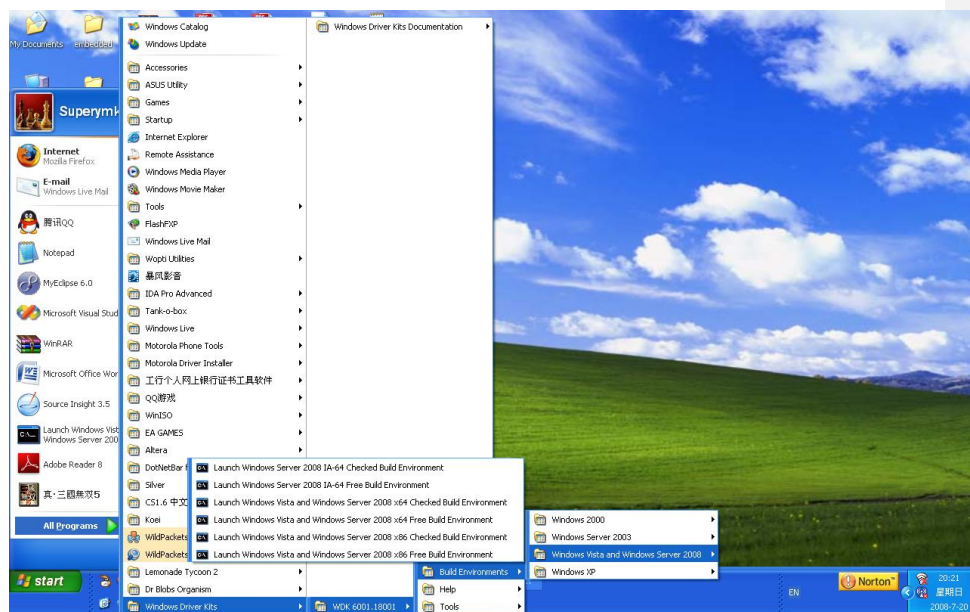


图 3.2 WinDDK 编译环境快捷方式的位置

步骤 3. 在该编译环境中执行 `nbp-0.32-public\NewBluePill-0.32-public\build_code.cmd`, 如果编译成功则会出现以下窗口:

¹ 如果 NewBluePill 要在 Windows 2003 x64 上运行, 那么选择 Launch Windows Server 2003 x64 Checked Build Environment。

```

1>Compiling - vnx\vnxddebug.c
2>Building Library - lib\amd64\svm.lib
1>Building Library - lib\amd64\vmx.lib
1>BUILD: Compiling and Linking d:\nbp-0.32-public\nbp-0.32-public\common directory
1>Assembling - amd64\msr.asm
1>Assembling - amd64\svm-asm.asm
1>Assembling - amd64\vmx-asm.asm
1>Assembling - amd64\common-asm.asm
1>Assembling - amd64\regs.asm
1>Assembling - amd64\cpuid.asm
1>Assembling - amd64\intstubs.asm
1>Compiling - common\newbp.c
1>Compiling - common\hvm.c
1>Compiling - common\portio.c
1>Compiling - common\comprint.c
1>Compiling - common\hypercalls.c
1>Compiling - common\traps.c
1>warnings in directory d:\nbp-0.32-public\nbp-0.32-public\common
1>d:\nbp-0.32-public\nbp-0.32-public\common\traps.c : warning C4819: The file contains a character that cannot be represented in the current code page (936). Save the file in Unicode format to prevent data loss
1>Compiling - common\interrupts.c
1>Compiling - common\common.c
1>Compiling - common\paging.c
1>Compiling - common\sprintf.c
1>Compiling - common\chicken.c
1>Compiling - common\dbgclient.c
1>Linking Executable - bin\amd64\newbp.sys
BUILD: Finish time: Sun Jul 20 20:22:39 2008
BUILD: Done

30 files compiled - 2 Warnings
2 libraries built
1 executable built

D:\nbp-0.32-public\nbp-0.32-public>ctags -R
'ctags' is not recognized as an internal or external command,
operable program or batch file.

D:\nbp-0.32-public\nbp-0.32-public>

```

图 3.3 显示编译成功信息的 WinDDK 控制台

如果看到这个提示，恭喜你，编译成功了！

演示 NewBluePill

运行 NewBluePill 就有一定要求了，首先要求必须运行在支持硬件虚拟化技术（HEV）的 CPU 上，并且推荐在 64 位或者支持虚拟 64 位技术的 CPU 上运行，原因是虽然 NewBluePill 程序中附带了支持 32 位 CPU 的代码，但是有几个函数在编译时（Vista x86 Checked Mode）会出问题¹，而且有几个函数是未实现的，所以还是在 x64 上去跑吧。

下面是详细步骤：

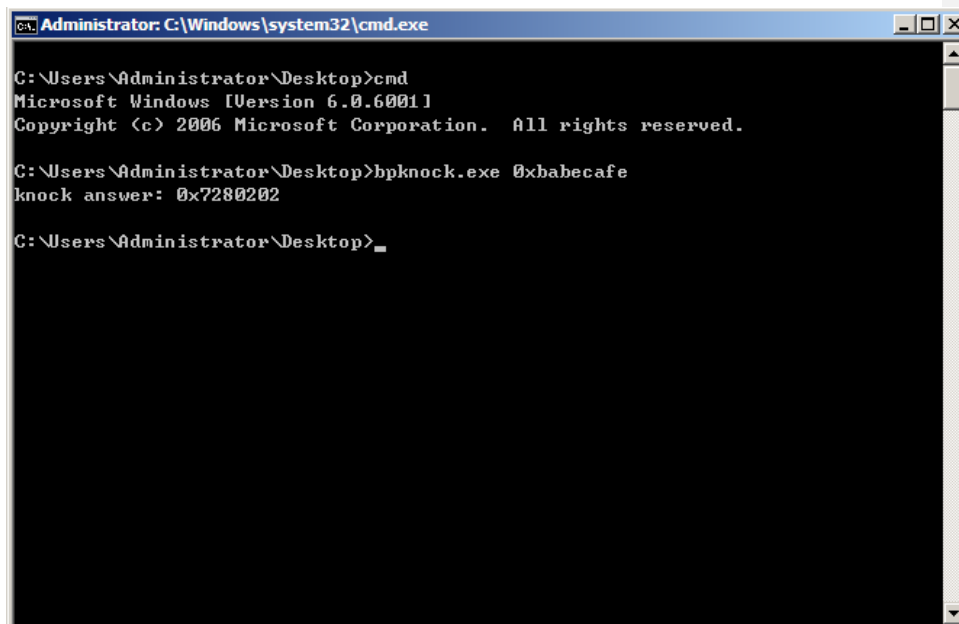
步骤 1：重启被调试机，按 F8，然后选择 Disable Driver Signature Enforcement(切记一定要用这个模式启动，否则不能加载未签名的驱动程序)

步骤 2：去 nbp-0.32-public 主目录及其子目录内找到下面几个编译生成的二进制文件：bpknock.exe, dbgclient.sys, newbp.sys

批注 [S6]: 此处出现“第十二章 移植 NewBluePill 到 32 位系统”章号

¹ 这个问题会作为实验内容“第十二章 移植 NewBluePill 到 32 位系统”留给读者解决

步骤 3: 运行下 bpknock 0xbabecafe 看下没运行 NewBluePill 的输出结果。



```
Administrator: C:\Windows\system32\cmd.exe
C:\Users\Administrator\Desktop>cmd
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Desktop>bpknock.exe 0xbabecafe
knock answer: 0x7280202

C:\Users\Administrator\Desktop>_
```

图 3.4 未加载 newbp 驱动的 bpknock 程序输出结果

步骤 4: 打开 DebugView, 在 DebugView 中的 Capture 菜单中选中下列项:

Capture Global Win32

Capture Kernel

Enable Verbose Kernel Output(这个一定要选中)

Pass-Through

Capture Events

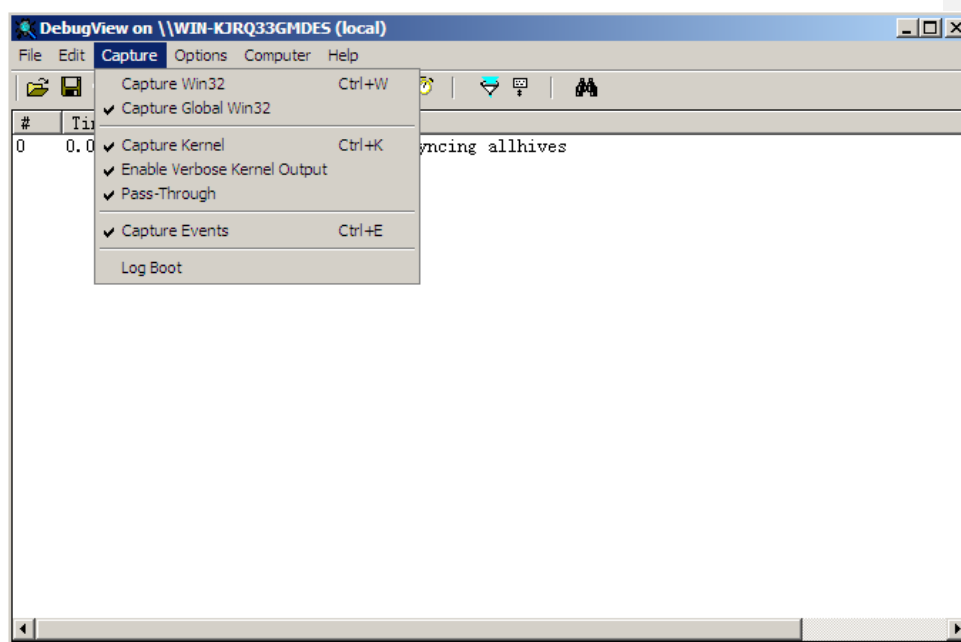
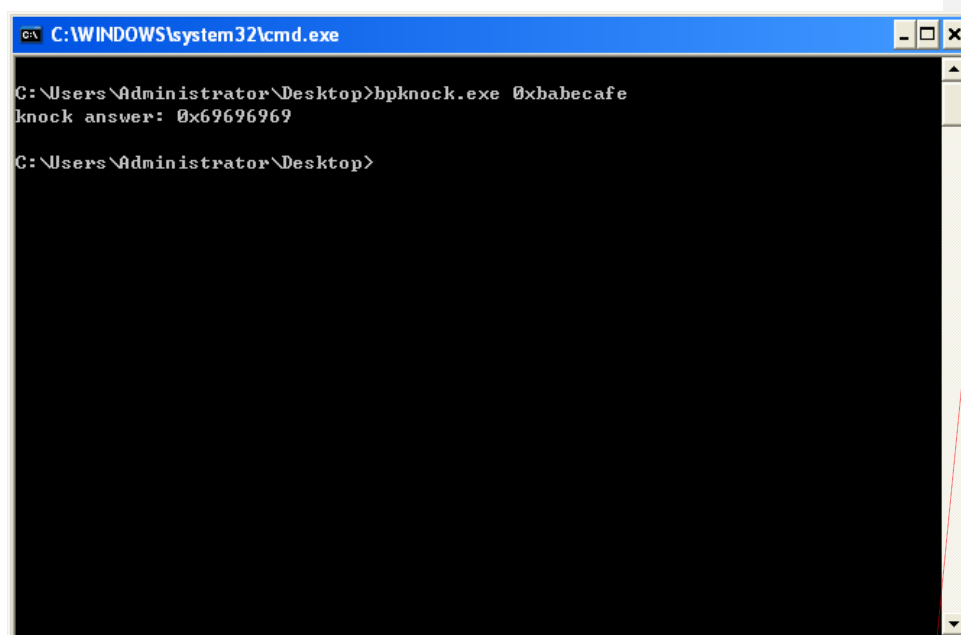


图 3.5 配置 DebugView

然后打开 InstDrv，先后加载并启动 dbgclient.sys 驱动和 newbp.sys 驱动¹

步骤 6：再运行下 bpknock 0xbabecafe 看下运行了 nbp 的输出结果。（如图 3.6 所示）



批注 [S7]: 以后再把这个换成 win2k8 下面的

图 3.6 加载 newbp 驱动的 bpknock 程序输出结果

¹ 这两个驱动分别在各自文件夹的 bin 子目录下

调试 NewBluePill

调试 NewBluePill 需要用到 WinDbg,主要过程如下:

步骤 1. 为了调试过程中可以下断点(切记做这一步只是为了以后能够调试,并且使得 NewBluePill 驱动只能运行在操作系统的 debug 模式下),修改 common 目录下的 newbp.c 文件,在 DriverEntry 方法的一开始添加 CmDebugBreak()方法调用¹(如图 3.7 所示),重新编译。修改后的代码如下:

```
00046: NTSTATUS DriverEntry (  
00047:     PDRIVER_OBJECT DriverObject,  
00048:     PUNICODE_STRING RegistryPath  
00049: )  
00050: {  
00051:     NTSTATUS Status;  
00052:     CmDebugBreak();  
00053:     #ifdef USE_COM_PRINTS  
00054:     PtoInit ((PUCHAR) COM_PORT_ADDRESS);  
00055:     #endif  
00056:     ComInit ();  
00057:     Status = MmInitManager ();  
00058:     ...  
}
```

图 3.7 添加 CmDebugBreak()方法的位置

步骤 2: 参考 Debugging Windows Vista² 修改被调试机启动项和调试项(这一步只需做这一次就可以)

步骤 3: 重启被调试机,可以看到启动项中多了一个 DebugEntry [debugger enabled]项,选中它按 F8,然后选择 Disable Driver Signature Enforcement 项启动

步骤 4: 调试机上设置_NT_SYMBOL_PATH 环境变量,指向 newbp.pdb 所在的目录,用于链接符号表。

步骤 5: 调试机上启动 WinDbg,单击 File 菜单选择 Kernel Debugging,在弹出的对话框输入 Baud Rate 为 115200,Port 用 com1。³这是由于刚才在演示过程的第一步我们用的是默认配置,如果调试端口发生相应改变,这里也要改。

¹ CmDebugBreak()函数实际上是一个 int 3 调用,在非调试模式的 Windows 下,这时这个中断的处理程序未注册,因此执行 int 3 指令会死机。

² 文章来源: http://www.microsoft.com/whdc/driver/tips/debug_vista.msp

³ 除了利用串口线调试外,也可以利用 1394 线进行调试,这样做的好处是数据传输速度更快。具体方法可以参考网上相关资料。

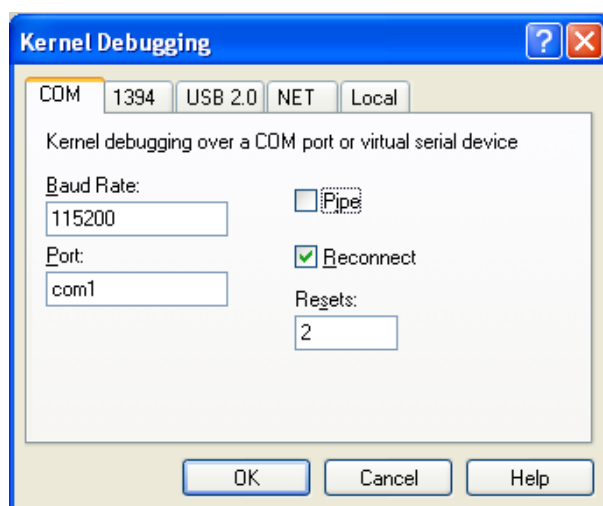


图 3.8 配置 Windbg

步骤 6: 被调试机上先后加载并启动 dbgclient.sys 和 newbp.sys 两个驱动, 运行 bpknock 程序, 开始调试。

如果出现 symbol 不能被加载的情况可以试试 WinDbg 中的.reload 命令, 如果不行可以试试用.sypath 在 WinDbg 运行时设定 symbol 路径, 然后.reload 重新加载符号表。

成功情况下的截图:

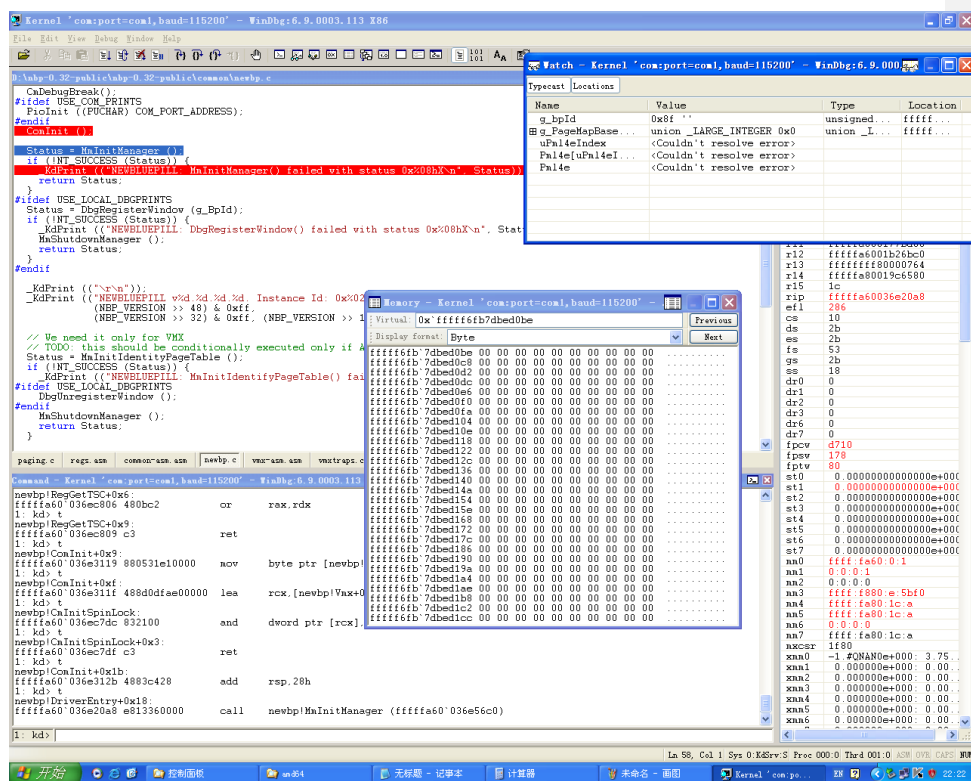


图 3.9 成功搭建调试平台

Ok, 有了调试平台, 我们就可以揭开 NewBluePill 的层层面纱了。

PART1 HEV 技术相关知识

三、 深入 HEV 技术细节

在第一章中，我们简单介绍了 SVM 和 VT 技术，但是他们是如何被具体使用的呢？在本章中我们将详细介绍这些技术的细节：

- HEV 下虚拟机的启动过程
- VMEXIT 事件的产生和处理
- HEV 下虚拟机的关闭过程
- SVM 和 VT 技术的关键数据结构
- HEV 中的双层地址翻译

直接阅读本章，可能会觉得理解其中内容却印象不深，推荐在阅读完全书后再次阅读本章——在理解了 NewBluePill 代码后，对本章内容自然会有更深的认识。

HEV 下虚拟机启动过程

“物有本末,事有终始,知所先后,则近道矣”——《大学》

想要了解 HEV 技术的本质，则要了解 HEV 要解决的问题和怎样解决这些问题。要熟悉这些，就要沿着虚拟机开启——运行——关闭的过程，看 HEV 技术是怎样融入其中的。所以我们首先就来看看在 HEV 技术的帮助下，虚拟机是怎样启动的。

启动过程模型

首先介绍下有了 HEV 技术后，启动虚拟机的方式。使用了硬件虚拟化技术的虚拟机可以有三种引导 Guest 操作系统的方式：

1. 存在特殊 OS/Host OS，后启动 Hypervisor 的虚拟机启动过程
 2. 存在特殊 OS/Host OS，先启动 Hypervisor 的虚拟机启动过程
 3. 不存在特殊 OS/Host OS，先启动 Hypervisor 的虚拟机启动过程
- 存在特殊 OS/Host OS，后启动 Hypervisor 的虚拟机启动过程。采用这种启动过程的虚拟机代表是 KVM，其启动过程如下：
 - a) 先启动宿主 Linux 操作系统
 - b) 在 Linux 中启动 KVM 设备，从而启动了 Hypervisor
 - c) 启动虚拟机，作为 Linux 进程运行

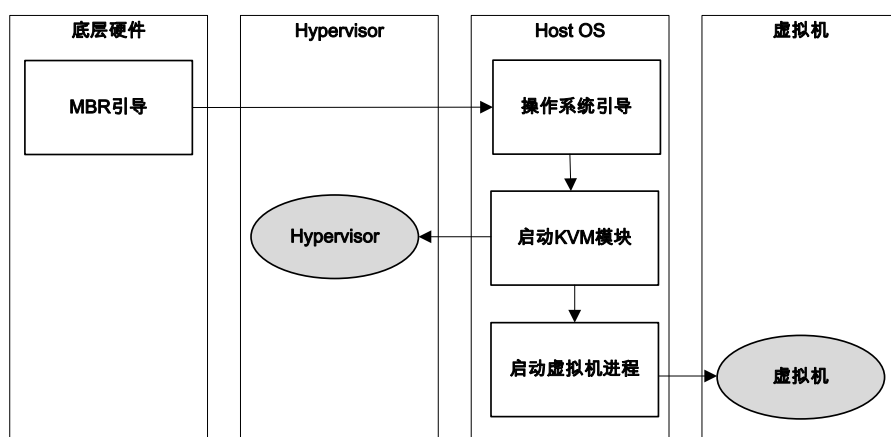


图 2.1 KVM 中虚拟机的启动过程

启动过程如图 2.1，可以看出，KVM 启动虚拟机的模式说明它不想脱离进程级虚拟机的本质，但是它要利用虚拟化技术进行加速。这样做的缺点在于需要一个 Host OS 充当载体。除 KVM 外，VMWare6.5 以上版本也是采用类似的架构，使用支持 HEV 技术的 CPU 进行加速。但是它们都需要再另外安装相应 Guest OS 上的驱动。NewBluePill 也属于这样的一个启动模型，略有不同的是在成功启动 NewBluePill 后，它会把宿主 Windows 操作系统置于虚拟机中运行，详细过程可以参考本书后续章节。

- 存在特殊 OS/Host OS，先启动 Hypervisor 的虚拟机启动过程。采用这种启动过程的虚拟机代表是 Xen，其启动过程如下：
 - a) 先创建并启动 Hypervisor
 - b) 引导 Dom0
 - c) 由 Hypervisor 和 Dom0 一起协作创建虚拟机
 - d) 启动该虚拟机¹

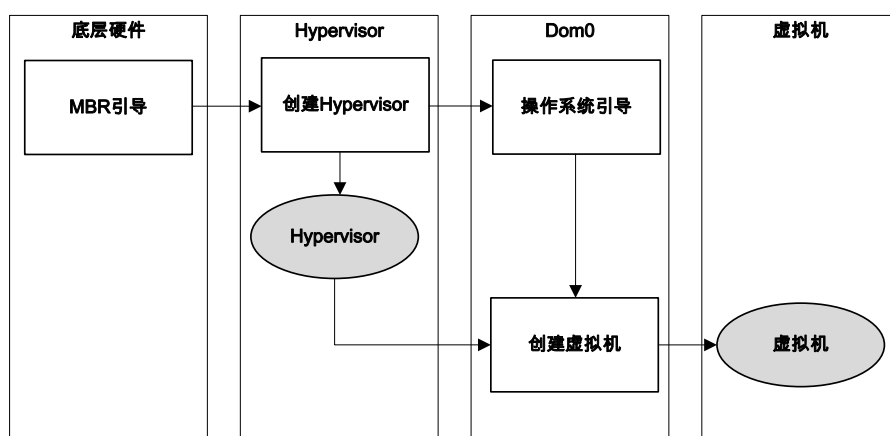


图 2.2 Xen 中虚拟机的启动过程

¹ Xen 中具体创建和启动虚拟机的过程会在“第 14 章 其它有关 HEV 项目”中介绍

启动过程如图 2.2，可以看出，Xen 中仍存在 Dom0 是因为它要适应过去未出现 HEV 技术时的架构，所以无论是 Dom0 还是 Hypervisor 的实现都比较笨重，并且安装和配置也比较麻烦，同样需要另外安装相应 Guest OS 上的驱动。但是不可忽视的是 Xen 的虚拟化效率最高。

- 不存在特殊 OS/Host OS，先启动 Hypervisor 的虚拟机启动过程。当前暂时没有采用这种启动过程的虚拟机软件（暂时称之为 UVM, Unknown Virtual Machine），其启动过程如下：

- a) 先创建并启动 Hypervisor
- b) 从 Hypervisor 中创建并启动虚拟机

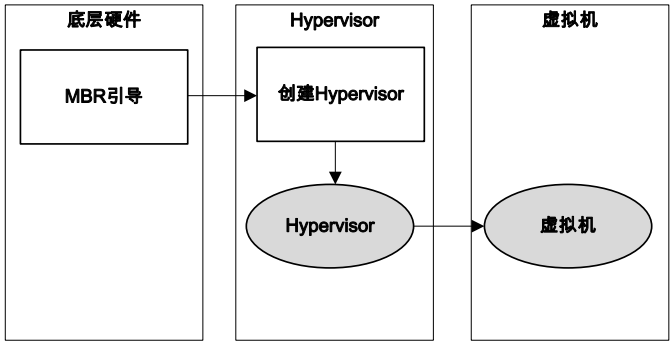


图 2.3 UVM 中虚拟机的启动过程

启动过程如图 2.3，这样的启动过程包括了如下组件：

表 2.1 UVM 模型下启动过程主要组件（传统 BIOS 启动）

组件	运行模式	作用
主引导扇区代码（MBR）	16 位实保护模式	读取并加载活动分区启动扇区（Active Partition's Boot Sector）
启动扇区（Boot Sector）	16 位实保护模式	读取并运行磁盘上的 Hypervisor 创建程序
Hypervisor 创建程序	16 位实保护模式，虚拟机模式	创建并初始化 Hypervisor，并创建至少一个虚拟机
虚拟机引导程序	虚拟机模式	任何已有的 BIOS 初始化程序或操作系统的 MBR 引导程序，目的是初始化虚拟机

批注 [S8]: 这种情况下，并不一定是启动扇区来读取 Hypervisor 运行的。例如在 EFI 情况下，是直接由 EFI 负责加载运行的。

这种虚拟机的设计目标在于：不需要在 Guest OS 中安装任何支持驱动。换句话说，Hypervisor 对于 Guest OS 完全透明，从而实现完全虚拟化（Full Virtualization）。这种方式的缺点是：Hypervisor 可能实现会很笨重，因而虚拟化效率不高，也会影响到系统安全，虚拟机的配置和管理可能也不易呈现给用户。

实验：阅读 Xen 和 KVM 的初始化部分代码

在 Xen 和 KVM 中，Hypervisor 的初始化代码都是用 C 编写的。阅读 Hypervisor 的初始化代码，对照图 2.1 和 2.2 体会其初始化的过程。

VT 技术下开启虚拟机的过程

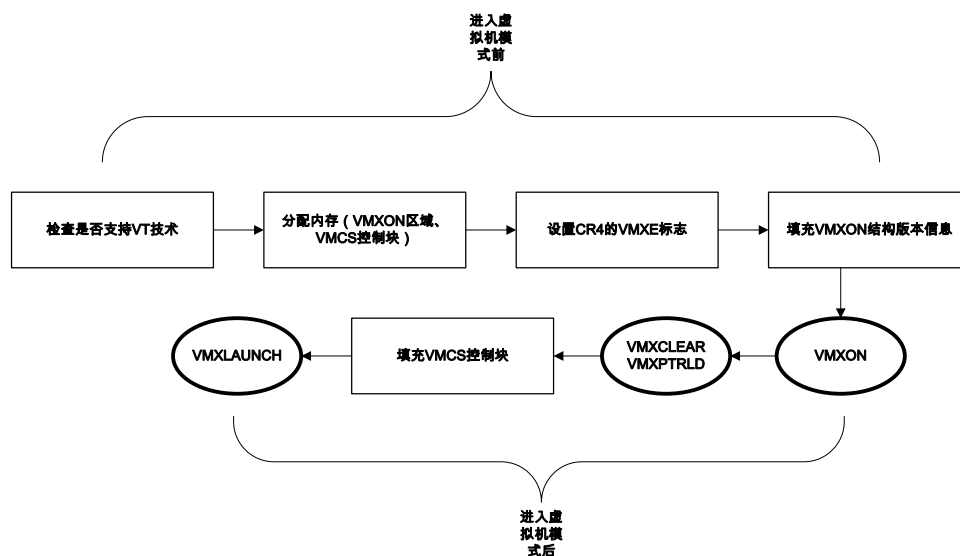


图 2.4 VT 技术下开启虚拟机的过程

在 VT 技术下进入虚拟机模式，开启虚拟机的全过程如图 2.4 所示。

首先检查当前 CPU 是否支持硬件虚拟化技术，这可以通过使用 `CPUID` 指令检查是否 `CPUID.1:ECX.VMX[bit 5]=1`（这句话表示操作数为 1 执行 `CPUID` 指令，检查返回的 `ECX` 寄存器 bit 5 位，也就是 `VMX` 位，查看结果是否为 1，下文均用这种表示法）。

开启虚拟机前，必须设置 `CR4.VMXE[bit 13]=1`，并且在内存中分配出来 `VMXON` 区域（`VMXON Region`）和 `VMCS` 控制块，后者也可以在进入虚拟机模式后再分配。需要注意的是，他们两者都必须分配在 4K 页对齐的内存区域上。

然后还需要初始化 `VMXON` 区域，需要把 `VMCS` 的版本号表示符（`VMCS Revision Identifier`）写入 `VMXON` 区域当中，该版本号可以通过访问 `IA32_VMX_BASIC` MSR 寄存器获得。除此以外不需要任何其它操作。只是要注意的是，在 `VMXON` 和 `VMXOFF` 指令之间的代码区中不要访问或者修改这个 `VMXON` 区域。

这之后通过执行 `VMXON` 指令即可以进入虚拟机模式。要注意的是，如果在执行 `VMXON` 指令时发现 `CR4.VMXE=0`，那么 `VMXON` 指令会发生无效操作数的异常（`#UD`¹）。最后，一旦进入虚拟机模式，`CR4` 和 `CR0` 寄存器中的一些与虚拟机模式相关的位将无法设置。

关于 `VMXON` 指令（`VMXON Instruction`）

`VMXON` 指令能否执行，同样也受 `IA32_FEATURE_CONTROL_MSR` 寄存器控制：

- Bit 0 是置锁位（Lock Bit） 如果该位为 0，那么 `VMXON` 指令不能执行；如果该位

¹ 关于异常的说明，请参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1* 中 Chapter 6.4 Interrupts and Exceptions 一节。

为 1，那么 WRMSR（写 MSR 寄存器指令）不能去写这个寄存器。该位在系统上电后便不能修改。BIOS 通过修改这个寄存器来设置是否支持虚拟化操作。在支持虚拟化操作的情况下，BIOS 还要设置 Bit 1 和 Bit 2

■ Bit 1 指示 VMXON 指令能否在 SMX¹操作环境中执行 如果这一位为 0，那么 VMXON 指令不能在 SMX 操作环境中执行

■ Bit 2 指示 VMXON 指令能否在 SMX 操作环境外执行 如果这一位为 0，那么 VMXON 指令不能在 SMX 操作系统外执行

如图 2.4 所指出的那样，执行 VMXON 指令后，我们需要执行 VMCLEAR 指令来初始化 VMCS 控制块。在第一章中我们介绍了 VMCS 区域的组成部分，在使用中，处理器会使用 VMCS 数据区来维护 VMCS 版本特定信息（implementation-specific information），VMCLEAR 指令会初始化这些信息，VT 技术推荐先执行 VMCLEAR 指令从而设置 VMCS 启动状态为空状态（clear），再执行 VMPTRLD 指令激活该 VMCS 块。

关于 VMCS 的启动状态

在第一章中我们在介绍了 VMLAUNCH 指令和 VMRESUME 指令的区别，里面同样涉及到 VMCS 的启动状态。

实际上，一个 VMCS 块的启动状态包含空状态（Clear）和已启动状态（Launched）两种状态，VMCLEAR 指令会把指定的 VMCS 块置为空状态，而 VMXRESUME 和 VMXLAUNCH 两个指令会根据该状态的进行相应的操作。

要注意的是，通过 VMWRITE 指令是不能修改这个状态的。而且在转移一个 VMCS 块到另一个逻辑处理器上时，需要先使用 VMCLEAR 指令设置启动状态为空状态，再用 VMLAUNCH 启动该 VMCS 块所代表的虚拟机。因此尽量避免在不同逻辑处理器间转移执行 VMCS 块的操作。

利用 VMPTRLD 加载 VMCS 块之后，需要开发者按照需要配置 VMCS 块具体内容，VMCS 相关具体内容，会在本章稍后详加解释。

当这一切都设置好以后，利用 VMXLAUNCH 指令启动该 VMCS 块所代表虚拟机，至此虚拟机和 Hypervisor 均已初始化完毕，并能够成功开启虚拟机。

¹ SMX（安全扩展模式，Safer Mode Extensions）

SVM 技术下开启虚拟机的过程

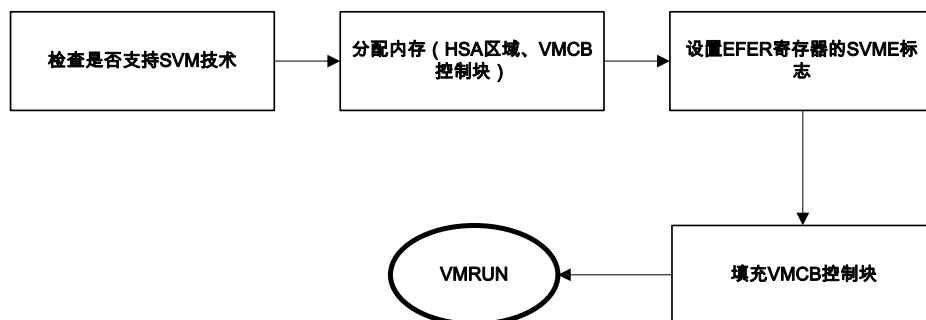


图 2.5 SVM 技术下开启虚拟机的过程

在 SVM 技术下进入虚拟机模式，开启虚拟机的全过程如图 2.5 所示。

与 VT 技术下情况类似，必须首先检查当前 CPU 是否支持硬件虚拟化技术。通过使用 CPUID 指令检查是否 `CPUID.8000_0001:ECX.SVM[bit 2]=1` 来判定当前 CPU 对 SVM 的支持情况。

开启虚拟机前，必须设置 `EFER.SVME[bit 12]=1`，并分配 Hypervisor 状态存储空间（Host State Area，HSA）和 VMCB 控制块。需要注意的是，在 SVM 技术中，Hypervisor 状态存储空间独立于 VMCB 控制块存在，其物理地址必须写入到 `VM_HSAVE_PA` MSR 寄存器中。

之后开发者按照需要配置 VMCB 结构体具体内容，包括对拦截事件的配置和虚拟机初始状态的填充，后者需要首先调用 `VMSAVE` 指令预先初始化部分虚拟机状态。相关内容会在本章稍后和后续章节中详加解释。

HEV 下虚拟机关闭过程

VT 技术下关闭 Hypervisor 和虚拟机的过程

对于 Hypervisor 的关闭，Intel 技术文档要求必须在 Hypervisor 下，通过执行 `VMXOFF` 指令来关闭虚拟机模式，通过检查 `RFLAGS[CF]=0` 和 `RFLAGS[ZF]=0` 来判断是否关闭成功，然后可以清除 `CR4.VMXE` 的值。

要注意的是，如果 Hypervisor 开启了 SMM 监控器（SMM Monitor），那么一定要先关闭这个监控器再拆除 Hypervisor，否则执行 `VMXOFF` 指令不成功。

另一个要注意的问题是，成功卸载 Hypervisor 后，由于此时已经没有了虚拟机模式，所以如果需要继续运行操作系统的话，那么必须人工的还原操作系统运行环境，很多时候这指的是人工将 VMCS 保存的虚拟机环境填充为当前环境，一个显而易见的例子是：如果不人工设置 `RIP/EIP`，那么操作系统此时不可能继续执行指令。

SVM 技术下关闭 Hypervisor 和虚拟机的过程

SVM 下没有特定的命令用于关闭虚拟机模式，因此开发者关闭虚拟机模式的过程就是恢复 `EFER.SVME` 和 `VM_HSAVE_PA` MSR 寄存器到开启前的状态。

与 VT 技术类似，成功卸载 Hypervisor 后，如果需要继续运行操作系统的话，同样必须人工还原操作系统运行环境，从而使得操作系统可以继续执行后续指令。

HEV 下#VMEXIT 事件的产生和处理

在第一章中我们已经分别介绍过 VT 和 SVM 技术下 Hypervisor 的生命周期，从中我们可以发现#VMEXIT 事件是导致从虚拟机陷入 Hypervisor 的关键事件。本节我们将探索 SVM 和 VT 技术下的#VMEXIT 事件的产生和处理过程

VT 技术下的#VMEXIT 事件的产生和处理

VT 技术下的#VMEXIT 事件分为无条件产生和有条件产生两种，前者包括诸如 CPUID、GETSEC、INVD、XSETBV 和所有 VMX 相关指令，在虚拟机中，这些指令的执行会立即产生 #VMEXIT 事件，陷入到 Hypervisor 中。

有条件陷入指令/事件占据了大多数情况，比如 I/O 访问，MSR 寄存器访问，HLT 等。这些指令只有配置了 VMCS 结构体相应部分，甚至需要配置相应位图才可以拦截。

在 VT 技术中，一旦发生了一个#VMEXIT 事件，处理过程如图 2.6 所示

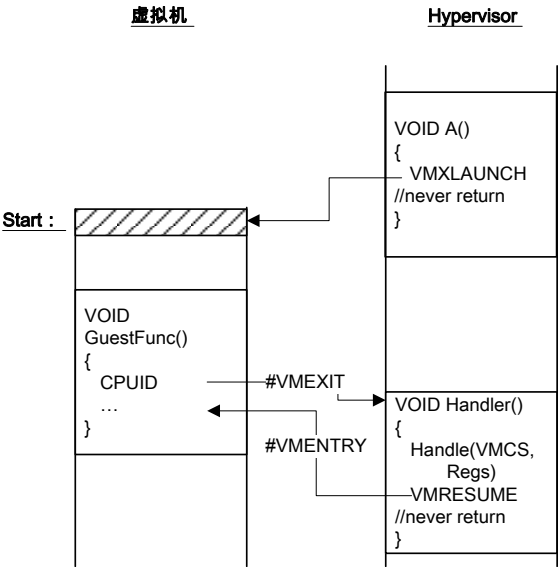


图 2.6 VT 技术下#VMEXIT 事件的完整处理过程

由于在 VMCS 中需要指定陷入到 Hypervisor 后的事件处理入口地址，因此在开发者在基于 VT 技术实现 Hypervisor 时，都会有一个事件分发函数作为#VMEXIT 事件处理入口函数。也正是由此，VT 技术下一个完整的#VMEXIT 事件处理更接近于利用 SYSENTER 指令完成的系统调用处理流程。

SVM 技术下的#VMEXIT 事件的陷入和处理

SVM 技术并未对#VMEXIT 事件进行分类。无论何种情况，一旦产生#VMEXIT 事件，SVM 都会按照图 2.7 所示处理。

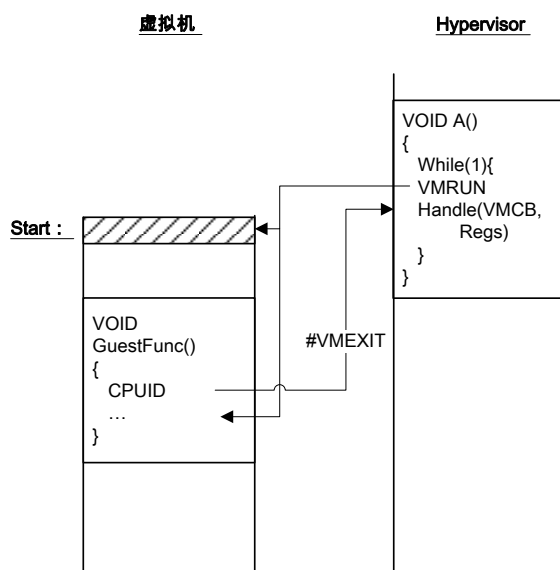


图 2.7 SVM 技术下 #VMEXIT 事件的完整处理过程

在 SVM 技术下, VMRUN 指令既是开启虚拟机的指令, 又是恢复虚拟机继续执行的指令, 同时由于在 VMCB 结构体中不需要定义类似于 VT 技术中的 #VMEXIT 事件处理入口地址, 因此每次回到 Hypervisor 上下文必然是继续之前的 VMRUN 指令继续运行。所以在基于 SVM 技术开发 Hypervisor 时, 一定要注意 VMRUN 指令后必定跟随 #VMEXIT 事件处理过程, 当处理完该事件后, 还要调用 VMRUN 指令回到原先虚拟机上下文, 这也就是此处需要一个循环的原因。

这样的处理方式部分原因是 SVM 指令比较少, 但是在事件处理能力上, SVM 和 VT 是一样的。

HEV 下虚拟机关键数据结构

VT 下的 VMCS 结构体和 SVM 下的 VMCB 结构体在开启虚拟机的过程中起着至关重要的作用, 这一节我们就将深入探索 VMCS、VMCB 结构体的细节。

VT 技术下的 VMCS 结构体

在上一章中我们提到, VMCS 区域由 VMCS 版本标志 (Revision Identifier)、VMX 退出原因指示器 (VMX-abort indicator)、VMCS 数据区三部分构成。

在使用 VMCS 区域前, 应当首先设置 VMCS 版本标志, 因为文档中说明, 这个域是由软件负责设置的, 并且 VMPTRLD 指令在执行时会检查该 VMCS 块设置的版本标志与目标处理器能够接受的 VMCS 版本是否相符, 不相符则会抛出异常。通常软件通过读取 IA32_VMX_BASIC MSR 寄存器来设置 VMCS 版本标志, 因为这个 MSR 寄存器存有该处理器能够接受的 VMCS 版本标志。

VMX 退出原因指示器的产生是因为在 VM Exit 事件的时候如果遇到问题, 系统就会发生 VMX Abort 事件, 这会导致该逻辑处理器进入关闭状态。对于一个激活了的 VMCS, 一个 VMX Abort 事件的发生并不会修改 VMCS 数据区, 因此我们需要一种机制去判断是什么原因导致了 VMX Abort 事件的发生。通常, 造成 VMX Abort 事件的原因包括: 保存客户虚拟机 MSR

寄存器失败、当前 VMCS 区域损坏、加载 Hypervisor 的 MSR 寄存器失败等等。

VMCS 数据区构成了 VMCS 区域的主体，第一章中我们介绍了 VMCS 数据区的六个主要组成部分，下面我们将逐一介绍这六个主要部分又是怎样构成的。

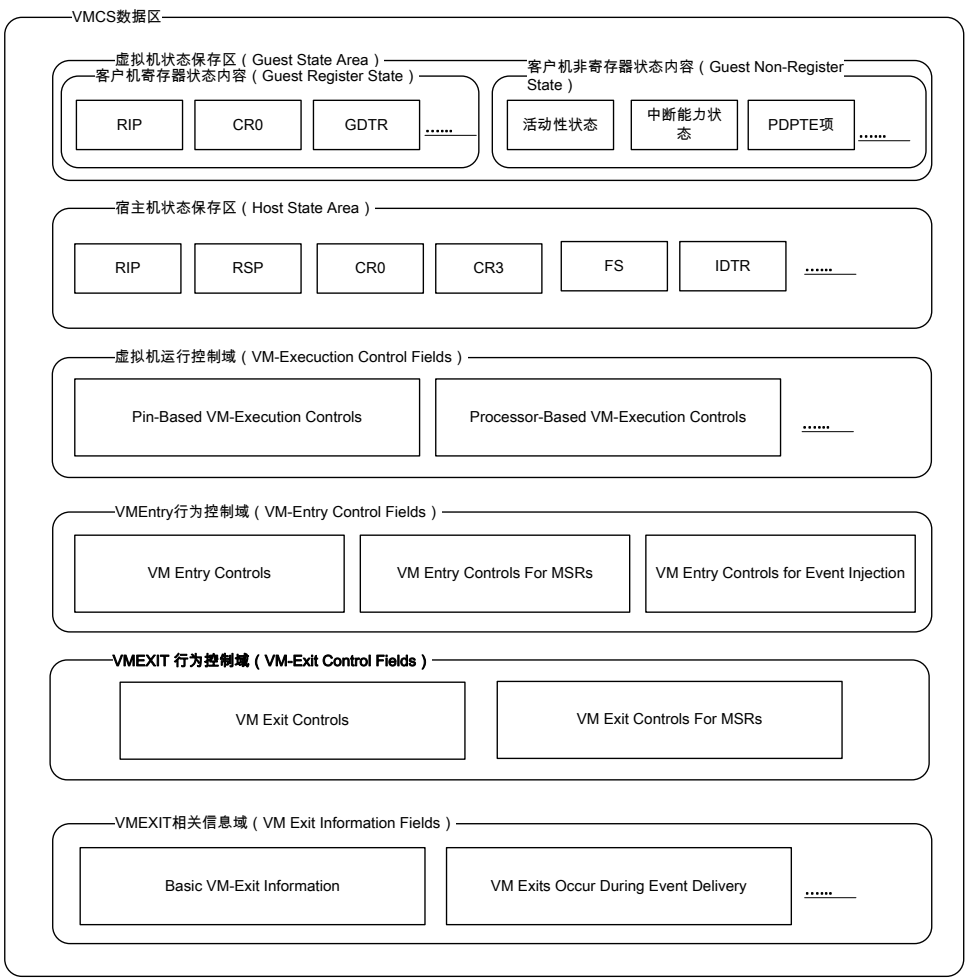


图 2.8 VMCS 数据区结构图

虚拟机状态保存区

虚拟机状态保存区(Guest-State Area)保存了用于描述客户虚拟机状态的寄存器，当发生#VMEXIT 事件的时候，虚拟机的状态会自动保存到这些域当中，而当发生#VMENTRY 事件的时候，这些域中的值又被用来恢复虚拟机的运行状态。

这个区域可以保存的内容分为客户机寄存器状态内容 (Guest Register State) 和客户机非寄存器状态内容 (Guest Non-Register State) 两部分。客户机寄存器状态内容 (x86 上分别对应各自 x86 寄存器)：

- 1) 控制寄存器 CR0,CR3,CR4

- 2) 调试寄存器 DR7
- 3) RSP, RIP 和状态寄存器 RFLAGS
- 4) CS,SS,DS,ES,FS,GS,LDTR 和 TR 寄存器的下面各项
 - 选择子 (Selector)
 - 基址 (Base Address)
 - 段长 (Segment limit)
 - 访问权限 (Access Rights)
- 5) GDTR 和 IDTR 信息
 - 基址 (Base Address)
 - 段长 (Segment limit)
- 6) 一些 MSR 寄存器, 包括 IA32_DEBUGCTL, IA32_SYSENTER_CS, IA32_SYSENTER_ESP, IA32_SYSENTER_EIP, IA32_PERF_GLOBAL_CTRL, IA32_PAT, IA32_EFER

客户非寄存器状态内容包括:

- 1) 活动性状态 (Activity State) 这项表明了当前逻辑处理器的活动性,也就是能否正常处理客户机程序指令, 包含 4 个状态:
 - a) 活跃的 (Active) 说明当前逻辑处理器可以执行客户机指令
 - b) 中断的 (HLT) 当前逻辑处理器不能执行客户机指令, 因为执行了一条 HLT 指令。
 - c) 关闭的 (Shutdown) 当前逻辑处理器不能执行客户机指令, 因为发生了严重的错误。
 - d) 等待 SIPI 中断 (Wait-for-SIPI) 当前逻辑处理器不能执行客户机指令, 因为在等待 Startup-IPI 中断¹
- 2) 中断能力状态 (Interruptibility State) x86 架构支持某些事件能被阻塞一段时间的特性, 包括被 STI 开中断指令阻塞, 被 MOV SS 阻塞, 被 SMI 中断阻塞和被 NMI 中断阻塞。这个域就包含了对应的描述信息。
- 3) 推迟调试的异常 (Pending Debug Exceptions) x86 支持延迟发送一些调试异常, 如某些情况下的单步调试。这个域就包含了对应的描述信息。
- 4) VMCS 连接指针 (VMCS Link Pointer) 该域保留, 用于未来扩展。
- 5) VMX 抢占计时器值 (VMX-Preemption Timer Value) 该域保存了虚拟机的 VMX 抢占计时器计数值。
- 6) PDPTE 项 (Page Directory Pointer Table Entries) 虚拟机状态保存区只有在虚拟机运行控制域中开启 EPT 模式才会用到此域, 其中包括 4 个 64 位数据: PDPTE0~PDPTE3。

关于 VMX 抢占计时器 (VMX-Preemption Timer)

VMX 抢占计时器是 VT 技术中这样的一种特性, 如果在 #VMEntry 事件发生后, 处理器硬件发现在虚拟机运行控制域中“启用 VMX 抢占计时 (Activate VMX-Preemption Timer)”被设置, 那么交由虚拟机执行指令时, 将启用一个 **VMX 抢占计时器**, 该计时器会倒数, 当倒数为 0 时会触发一个 #VMEXIT 事件陷入 Hypervisor 中。

如果“启用 VMX 抢占计时 (Activate VMX-Preemption Timer)”被设置, 同时 VMX 抢占

¹ SIPI (Startup Inter-Processor Interrupt), 用于多核平台初始化其它处理器。具体可参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* 手册

计时器值为 0，那么#VMEXIT 事件会在#VMENTRY 事件发生后执行任何指令前发生（但是如果有推迟调试的异常（Pending Debug Exceptions），它们将先于#VMEXIT 事件得到处理）。

VMX 抢占计时器的计时间隔（Timer Interval）是可以通过 IA32_VMX_MISC 寄存器设置的（0~31），且为 2 的整数次幂于 TSC 寄存器值的变化。比如，如果我们设置该寄存器内容为 5，那么表示 TSC 寄存器中的值每增加 32，VMX 抢占计时器计数减 1。

但是要注意，由于节能模式的关系，所以当逻辑处理器在 C-states 的 C0, C1, C2 之外的其它状态时，VMX 抢占计时器不会产生#VMEXIT 事件¹

对于系统管理中断(SMIs, System Management Interrupts)和系统管理模式(SMM, System Management Mode)，VMX 抢占计时器的行为依赖于 Hypervisor 是否要处理这个模式：

- 默认情况下，VMX 抢占计时器在虚拟机处理系统管理中断时，也会继续倒数计时，只不过在倒数为 0 的情况下，VMX 抢占计时器要继续等到虚拟机脱离系统管理模式后再产生#VMEXIT 事件。

- 如果 Hypervisor 的设定指出同样要监管系统管理中断和系统管理模式时，这个时候 VMX 抢占计时器的行为和在处理一般 VMEXIT 事件、VMEntry 事件时一样。

宿主机状态保存区

宿主机（Hypervisor）状态保存区（Host-State Area）记录了所有有关 Hypervisor 的状态信息，正如第一章中所提到的，这个区域保存的内容会在每次发生#VMEXIT 事件时恢复到相应的寄存器中，以恢复 Hypervisor 的执行环境。

与虚拟机状态保存区不同，宿主机状态保存区只能存储有关寄存器的信息：

- 1) 控制寄存器 CR0, CR3, CR4
- 2) RIP, RSP
- 3) CS, SS, DS, ES, FS, GS 和 TR 寄存器的下面各项
 - 选择子（Selector）
- 4) FS, GS, TR, GDTR 和 IDTR 信息
 - 基址（Base Address）
- 5) 一些 MSR 寄存器，包括 IA32_SYSENTER_CS, IA32_SYSENTER_ESP, IA32_SYSENTER_EIP, IA32_PERF_GLOBAL_CTRL, IA32_PAT, IA32_EFER

虚拟机运行控制域

虚拟机运行控制域(VM-Execution Control Fields)管理着虚拟机的运行，包含下列域：

- 1) 基于针脚的虚拟机执行控制（Pin-Based VM-Execution Controls）

这个域用来管理中断等异步事件（Asynchronous Event）²，如“启用抢占计时”就是在这个域中设置的。对于其中保留位的设置，软件要参考 IA32_VMX_PINBASED_CTLs 和 IA32_VMX_TRUE_PINBASED_CTLs 两个 MSR 寄存器的内容。

- 2) 基于处理器的虚拟机执行控制（Processor-Based VM-Execution Controls）

¹ 有关节能模式和 C-states 的详细信息，可以参考网上相关资料

² 有些中断不受该域控制，触发 VMEXIT 事件

该域包含两个 4 字节值，用于管理执行特定指令而产生的同步事件（Synchronous Event）。这个控制域分为主要基于处理器的虚拟机执行控制和次要基于处理器的虚拟机执行控制两个，前者包括对 HLT、INVLPG、MWAIT、RDPMSR、RDTSC、CR3 读取/存储、使用 I/O Bitmap 等等这些产生 VMEXIT 事件的控制；后者包括对开启 EPT 地址翻译、开启 VPID¹、开启虚拟 APIC（高级可编程中断控制器，关于此部分的介绍，可参考随后“关于可编程中断控制器”部分）等等的控制。对于其中保留位的设置，前者要参考 IA32_VMX_PROCBASED_CTL0 和 IA32_VMX_TRUE_PROCBASED_CTL0 两个 MSR 寄存器的内容，后者要参考 IA32_VMX_PROCBASED_CTL12 MSR 寄存器。

3) 异常位图（Exception Bitmap）

该域仅有 32 位长，每一位代表当某种异常发生时，硬件会自动产生 VMEXIT 事件；如果某一位为 0，则表示这个异常会通过 IDT 表正常处理。这其中要注意的是缺页异常的处理略有不同，需要借助于 VMCS 中另外的两个域（Page Fault Error Code Mask 和 Page Fault Error Code Match）来处理。

4) I/O 位图地址（I/O Bitmap Addresses）

该域包含两个 64 位长的物理地址，指向两块 I/O 位图，只有在 Primary Processor-Based VM-Execution Controls.Use I/O Bitmaps[bit 25]=1 的情况下才会被使用。逻辑处理器在处理 I/O 指令时，会根据这些 I/O 位图而在访问相应地址时产生 VMEXIT 事件。VT 技术要求这些位图必须在 4K 页对齐的位置上。

5) 时间戳寄存器偏移值（Time-Stamp Counter Offset）

虚拟机运行控制域还包括一个时间戳寄存器偏移值域，这个域在 Primary Processor-Based VM-Execution Controls.RDTSC Exiting[bit 12]=0 且 Primary Processor-Based VM-Execution Controls.Use TSC Offsetting[bit 3]=1 时起作用。当客户机利用 RDTSC、RDTSCP 指令或者访问 IA32_TIME_STAMP_COUNTER MSR 寄存器的时候，得到的结果将是真实的值加上这个偏移量的和。

6) 虚拟机/Hypervisor 屏蔽和 CR0/CR4 访问隐藏设置（Guest/Host Masks and Read Shadows for CR0 and CR4）

这个域主要对 CR0 和 CR4 寄存器进行保护。虚拟机/Hypervisor 屏蔽中置位 1 的位说明 CR0 和 CR4 寄存器的相应位只能由 Hypervisor 修改，否则产生 VMEXIT 事件，置位 0 的部分说明 CR0 和 CR4 的相应位可以在虚拟机中修改。

7) CR3 访问目标控制（CR3-Targeting Controls）

包含最多 4 个 CR3 目标值²，当在虚拟机中执行 CR3 的赋值操作时，如果赋予的值是这些目标值中任意一个，那么硬件不会产生 VMEXIT 事件。

8) APIC 访问控制（Controls for APIC Accesses）

访问本地 APIC 的寄存器有三种方法：通过 xAPIC 模式访问、通过 x2APIC 模式访问、

¹ 开启 VPID (Virtual-Processor Identifier)意味着缓存在翻译线性地址时会根据 VPID 进行翻译，其优点已在第一章中介绍。

² 未来可能在此处容纳更多的 CR3 目标值，因此在使用前参考 IA32_VMX_MISC MSR 寄存器来查看当前处理器详细信息。

在 64 位模式下通过 `mov CR8` 指令来访问任务优先级寄存器 (Task Priority Register, TPR)¹。APIC 的访问控制实际上也是 VT 技术对 APIC 的虚拟化, 在这个域中, 通过配置 “使用影子 TPR (Use TPR Shadow)”、“虚拟化 APIC 访问 (Virtualize APIC Accesses)”、“虚拟化 x2APIC 模式 (Virtualize x2APIC Mode)” 对 APIC 进行虚拟访问。

9) MSR 位图地址 (MSR-Bitmap Address)

该域包含一个指向 MSR 位图区域的物理地址, 当 Primary Processor-Based VM-Execution Controls.Use MSR Bitmaps[bit 28]=1 时会被使用。MSR 位图区域占据 4K 大小内存, 分为 4 个连续区域: 读低地址/高地址 MSR、写低地址/高地址 MSR。根据配置, 当相应的 MSR 访问执行时, 会发生 VMEXIT 事件。

10) 执行体 VMCS 指针 (Executive-VMCS Pointer)

该域 64 位长, 用于 VT 技术对系统管理中断 (SMI) 和系统管理模式 (SMM) 进行监管 (也称 Dual-Monitor Treatment)

11) EPT 指针 (Extended Page Table Pointer)

该域包含了 EPT 页表的基地址 (也就是指向 EPML4 级页表的物理地址), 以及一些 EPT 页表的配置信息, 当 Secondary Processor-Based VM-Execution Controls.Enable EPT[bit 1]=1 时启用。

12) 虚拟机标示符 (Virtual Processor Identifier, VPID)

虚拟机表示符定义为 16 位长, 当 Secondary Processor-Based VM-Execution Controls.Enable EPT[bit 5]=1 时启用。

关于可编程中断控制器

■ x86 上的中断控制器

在大多数很古老的 x86 系统上都有一个 i8259A 可编程中断控制器 (Programmable Interrupt Controller, PIC), 或者新一些的, i82489 高级可编程中断控制器 (Advanced Programmable Interrupt Controller, APIC)。APIC 兼容 PIC, 前者拥有 256 条中断线, 而后者仅拥有 15 条中断线; 前者支持多核技术, 而后者并不支持。APIC 架构图如图 2.9 所示

¹ 请参考 Intel 手册相关部分以获得有关三种 APIC 访问方法的详细资料。

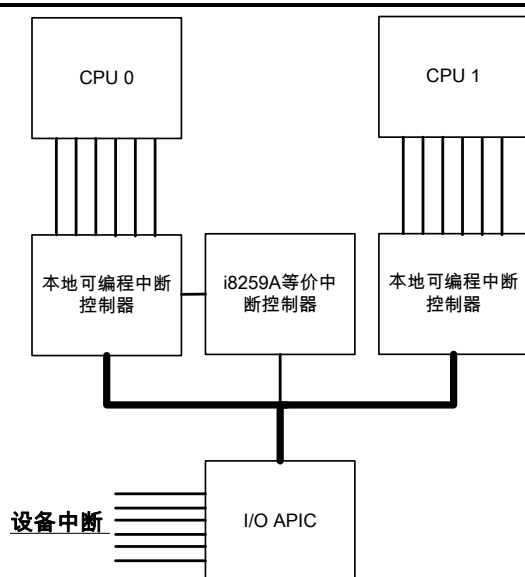


图 2.9 可编程中断控制器（APIC）架构图

APIC 包括下列组成部分：

- ◆ I/O APIC 用于从设备接收中断
- ◆ 本地可编程中断控制器（Local APIC） 本地中断控制器利用一条私有总线，从 I/O APIC 上接收中断，然后向与其关联的 CPU 发送中断。
- ◆ i8259A 等价中断控制器（i8259A Equivalent PIC） 负责把 APIC 的输入信号翻译成 PIC 等价的信号，用于实现在 APIC 上对 PIC 的兼容

APIC 私有总线也要负责决定要把该中断信号发送到哪个 Local APIC 上（Windows 有权决定是否要利用 I/O APIC 的这个算法），这样做可以更好的利用处理器本地性。

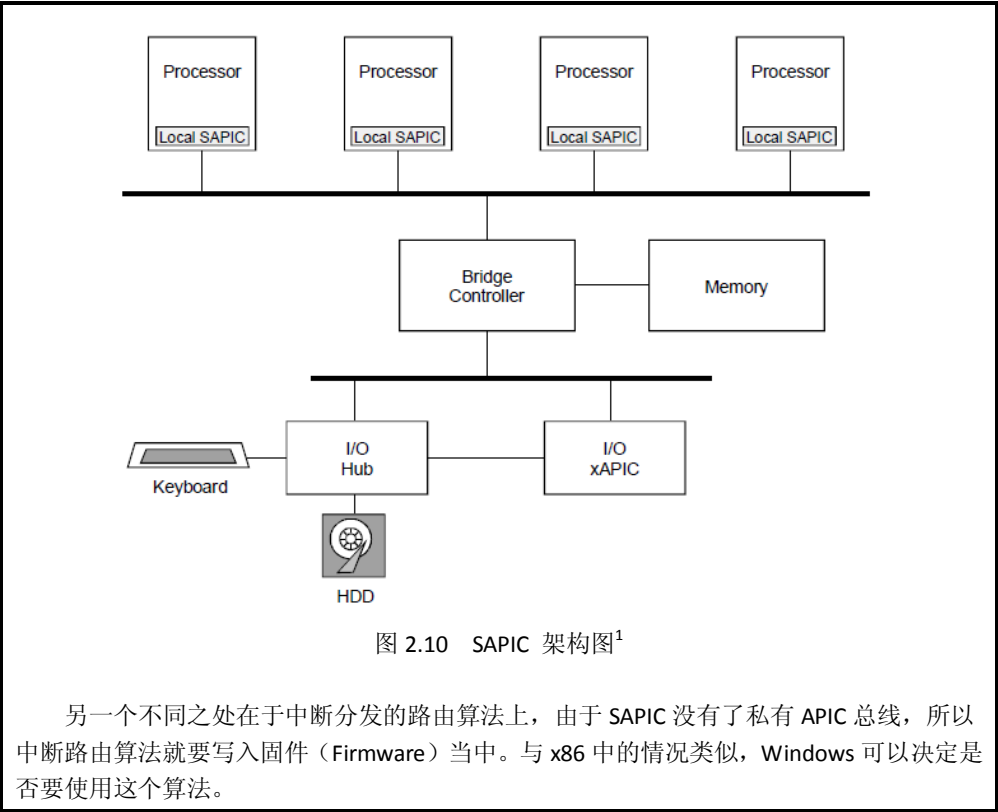
Local APIC 的另外一个功能是发送处理器间中断（Inter-Processor Interrupts, IPI）到其它的 Local APIC 上，考虑下列情况：每当一个时钟中断发生的时候，在多核平台上，显然只应该有一个处理器去负责更新系统时间，否则系统时间将会在一次时钟中断下被多次更新。这个时候就需要 IPI 的帮忙了。其它情况比如更新 TLB Cache、在 DISPATCH_LEVEL 中断级别下调度一个线程、系统崩溃、系统关闭时均需要 IPI 中断的帮忙。

■ x64 上的中断控制器

因为 x64 平台要兼容于 x86 平台，所以 x64 平台上的中断控制器与 x86 平台的相同。但是 x64 平台上的 Windows 要求一定要有 APIC 作为中断控制器。

■ IA64 上的中断控制器

IA64 平台使用 SAPIC（Streamlined Advanced Programmable Interrupt Controller）作为其中断控制器。SAPIC 的 I/O APIC 不再使用一条私有总线传输中断，而是利用系统总线，进而通过北桥控制器来传输中断（见图 2.10）。这样做的好处是能够更快的传输中断。



实验：查看本机上的 APIC 配置

在多核机器上，可以在 windbg 中利用!apic 命令查看本机 APIC 配置,“0: kd”表明当前运行在 CPU 0 上，m 表明当前中断被屏蔽。

```
0: kd> !apic
Apic @ fffe0000 ID:0 (40011) LogDesc:01000000 DestFmt:ffffff TPR FF
TimeCnt: 03f07410clk SpurVec:1f FaultVec:e3 error:0
Ipi Cmd: 02000000`000008e1 Vec:E1 FixedDel Lg:02000000 edg high
Timer.: 00000000`000300fd Vec:FD FixedDel Dest=Self edg high m
Linti0.: 00000000`0001001f Vec:1F FixedDel Dest=Self edg high m
Linti1.: 00000000`000004ff Vec:FF NMI Dest=Self edg high
TMR: 63, 83, B1
IRR: B1, D1
ISR: D1
```

Note 阅读 *Intel Itanium Processor Family Interrupt Architecture Guide* 文档可以获得关于 IA64 SAPIC 的更详细的指导信息

¹此图摘自 *Intel Itanium Processor Family Interrupt Architecture Guide*

VMEntry 行为控制域

VMEntry 行为控制域(VM-Entry Control Fields)定义了 VMEntry 事件发生后硬件要立即做的事情，主要包括三部分：VMEntry 基本操作控制设置 (VM Entry Controls)、VMEntry MSR 寄存器操作控制设置 (VM Entry Controls For MSRs) 和 VMEntry 注入事件控制设置 (VM Entry Controls for Event Injection)。

VMEntry 基本操作控制设置包括：

- 1) 加载调试寄存器内容
 - DR7
 - IA32_DEBUGCTL MSR 寄存器
- 2) 虚拟机是否进入 x64 支持模式 (x86 架构上永远为 0)
- 3) 进入系统管理模式 (SMM)
- 4) 关闭 Dual-Monitor Treatment
- 5) 加载 IA32_PERF_GLOBAL_CTRL MSR 寄存器
- 6) 加载 IA32_PAT MSR 寄存器
- 7) 加载 IA32_EFER MSR 寄存器

对于其它保留位的设置，软件必须根据 IA32_VMX_ENTRY_CTL5 MSR 寄存器和 IA32_VMX_TRUE_ENTRY_CTL5 MSR 寄存器内容设置。

对于 VMEntry MSR 寄存器操作控制设置，细心的读者可能发现，前面我们在描述客户机状态域时提到过，硬件会在发生 VMEntry 事件后自动加载一些有关虚拟机状态 MSR 寄存器。这里 VMEntry MSR 寄存器操作控制设置允许开发人员恢复更多的 MSR 寄存器，主要通过下面两个域配置：

- 1) VMEntry MSR 寄存器加载数量 (VMEntry MSR-Load Count)
- 2) VMEntry MSR 寄存器加载地址 (VMEntry MSR-Load Address)

VMEntry 事件可以在所有虚拟机状态恢复完毕后，通过客户机 IDT 表触发一个中断。VMEntry 注入事件控制设置就是用来配置这个特性的，它主要有如下三个可配置部分：

- 1) VMEntry 中断信息域 (VM Entry Interruption Information Field)
- 2) VMEntry 异常错误码 (VM Entry Exception Error Code)
- 3) VMEntry 指令长度 (VMEntry Instruction Code)¹

VMEXIT 行为控制域

VMEXIT 行为控制域(VM-Exit Control Fields)定义了 VMEXIT 事件发生后硬件要立即做的事情，主要包括两部分：VMEXIT 基本操作控制设置 (VM Exit Controls) 和 VMEXIT MSR 寄存器操作控制设置 (VM Exit Controls For MSRs)。

VMEXIT 基本操作控制设置包括：

- 1) 保存调试寄存器内容
 - DR7

¹ 对于软中断 (Software Interrupt)、软件异常 (Software Exception) 和特权软件异常 (Privileged Software Exception)，这个域用来决定填充到异常堆栈上的 RIP 地址指针值。

■ IA32_DEBUGCTL MSR 寄存器

- 2) Hypervisor 地址空间大小 (x86 架构上永远为 0)
- 3) 加载 IA32_PERF_GLOBAL_CTRL MSR 寄存器
- 4) VMEXIT 保留外部中断原因信息 (Acknowledge Interrupt on Exit)
- 5) 保存 IA32_PAT MSR 寄存器
- 6) 加载 IA32_PAT MSR 寄存器
- 7) 保存 IA32_EFER MSR 寄存器
- 8) 加载 IA32_EFER MSR 寄存器
- 9) 保存 VMX 抢占计时器值

VMEXIT MSR 寄存器操作控制设置类似于上文中的 VMEntry MSR 寄存器操作控制设置, 它允许开发人员在 VMEXIT 事件发生后保存更多有关虚拟机状态的 MSR 寄存器, 并恢复更多有关 Hypervisor 状态的 MSR 寄存器, 主要通过下面四个域实现:

- 1) VMEXIT MSR 寄存器保存数量 (VM-Exit MSR-Store Count)
- 2) VMEXIT MSR 寄存器保存地址 (VM-Exit MSR-Store Address)
- 3) VMEXIT MSR 寄存器加载数量 (VM-Exit MSR-Load Count)
- 4) VMEXIT MSR 寄存器加载地址 (VM-Exit MSR-Load Address)

VMEXIT 相关信息域

VMEXIT 相关信息域 (VM Exit Information Fields) 包含了最近 #VMEXIT 事件相关的信息。这是一个只读的域, 尝试利用 VMWRITE 指令向这个域中写入信息会失败。该域主要包括五个部分: VMEXIT 事件基本信息区 (Basic VM-Exit Information)、向量化事件 VMEXIT 信息区 (VM Exits Due to Vectored Events)、事件分发时 VMEXIT 信息区 (VM Exits Occur During Event Delivery)、指令执行时 VMEXIT 信息区 (VM Exits Due to Instruction Execution)、VM 指令错误信息域 (VM-Instruction Error Field)

VMEXIT 事件基本信息区包括:

- 1) 退出原因 (Exit Reason)
- 2) 退出条件 (Exit Qualification)
- 3) 客户机线性地址 (Guest-linear address)
- 4) 客户机物理地址 (Guest-physical address)

向量化事件 VMEXIT 信息区用于异常, 外部中断, NMI 等造成的 VMEXIT 事件的信息呈现。包括:

- 1) VMEXIT 中断信息 (VMExit Interruption Information)
- 2) VMEXIT 中断异常号 (VMExit Interruption Error Code)

事件分发时 VMEXIT 信息区用于在虚拟机中传播事件时发生的 VMEXIT 事件的信息记录。包括:

- 1) IDT 向量表信息 (IDT-Vectoring Information)
- 2) IDT 向量表异常号 (IDT-Vectoring Error Code)

指令执行时 VMEXIT 信息区, 该域记录在虚拟机中执行某些指令时发生的 VMEXIT 事件的信息。包括:

VMEXIT 指令长度 (VMEXIT Instruction Length)

VMEXIT 指令信息（VMEXIT Instruction Information）
VM 指令错误信息域描述了执行虚拟机操作指令（VMX Operation）发生的最后一个 Non-Faulting 错误的信息。

SVM 技术下的 VMCB 结构体

在上一章中我们提到,VMCB 结构体由控制域(Control Area)和虚拟机状态域(Guest-State Save Area)两部分组成,前者用于记录如下两方面内容:

- 1) 要拦截的虚拟机指令/事件
- 2) 返回虚拟机执行后首先要做的动作（事件注入等）

不同于 VT, 在 SVM 技术下虚拟机启动前只检查当前平台是否支持 SVM 技术并且 BIOS 开启了 SVM 支持, 而不会检查诸如 VMCS 版本号等项。同时, VMCB 结构体的退出信息域（VM-Exit Info）也集成于控制区中, 没有关于 VMCB 的状态描述部分（VMCS 结构体是有状态的, 可以参考前文相应部分描述), 因此在这方面 VMCB 提供的信息略少于 VMCS。

下面我们将详细介绍 VMCB 结构体的控制域和虚拟机状态域部分。

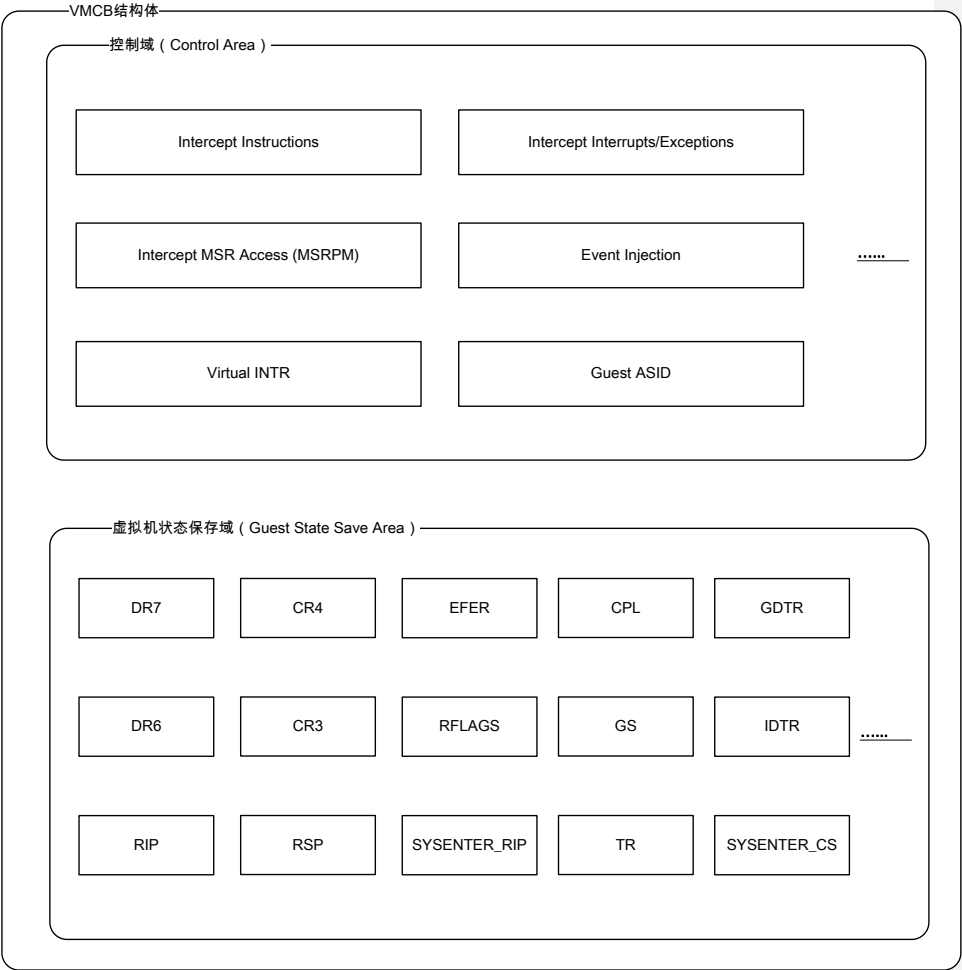


图 2.11 VMCB 控制块结构图

控制域

VMCB 控制域（Control Area）位于 VMCB 结构体起始的 1024 字节上，分为如下的几个部分：

1) 虚拟机指令/事件拦截区（前 141 位）

通过对指定位置位来达到拦截虚拟机相应事件/指令执行的目的，如拦截 CPUID 指令执行，拦截对 CR 处理器的访问等，需要注意的是，对于某些拦截（比如 IO 和 MSR 访问），不仅要在此处置位，还要在后面初始化并挂载相应 Permission Map 到 VMCB 控制域中。

2) IO Permission Map/MSR Permission Map

用于拦截指定的 I/O 端口，MSR 寄存器访问。对此的具体描述可参考“第四章 NewBluePill 启动过程”。需注意的是与 IOMMU 相对，IO Permission Map 针对的是 CPU 发出的 I/O 访问指令。

批注 [S9]: “第四章 NewBluePill 启动过程” 章号

3) 时间戳偏移量（Time-Stamp Counter Offset）

时间戳偏移量域（TSC_OFFSET）位于 VMCB 结构体控制域偏移量 50h，同样也是用于虚拟机在调用 RDTSC 和 RDTSCP 指令时，返回偏移后的时间戳值，与 VT 技术下的 VMCS 结构体中的时间戳寄存器偏移值部分功能类似。

4) 虚拟机地址空间描述符（Guest ASID）和 TLB 缓存策略

虚拟机地址空间描述符和 TLB 缓存策略均记录在 VMCB 结构体控制域偏移量 58h 的地方。前文中我们已经介绍过 ASID 的作用，通过利用 ASID，可以区分 TLB 上的一块地址是 Hypervisor 范围内的地址还是某个虚拟机的地址，从而加速地址翻译。而对于 TLB 缓存策略，在当前 SVM 技术实现中，一旦启用，则在每次 VMRUN 指令执行后强制清空 TLB。

5) 中断控制域（Virtual TPR & Virtual INT Request）

中断控制域位于 VMCB 结构体控制域偏移量 60h，SVM 利用该域的设置来拦截和注入虚拟机中断。其中包括

- V_INTR_MASKING: 该域用来决定虚拟机能否阻塞可屏蔽中断，当设置该位后，Hypervisor 的 EFLAGS.IF 可以在进入虚拟机上下文前被保存，并且在 Hypervisor 中能够处理物理中断¹。当该位清空时，所有的物理中断和虚拟中断都根据虚拟机的 EFLAGS.IF 处理。
- V_TPR: 作为一个虚拟的任务优先级寄存器（Task Priority Register，TPR），它的使用方式同样被 V_INTR_MASKING 决定，当 V_INTR_MASKING 置位后，虚拟机只能看到 V_TPR 内容，并以此控制虚拟中断。
- V_IGN_TPR: 用来指示当前等待中的虚拟中断不受 TPR 影响。此时不会依据 V_TPR 进行中断优先级比较。

¹ 物理中断与虚拟中断的区别是：屏蔽虚拟中断意味着不允许该任务处理这个中断，而不是真的从硬件上屏蔽掉了这个中断。

Note 关于 SVM 技术下注入虚拟中断、虚拟中断拦截等等更详细的信息，请参考 *AMD64 Architecture Programmer's Manual, Volume 2: System Programming, Chapter 15.20 Interrupt and Local APIC Support*

6) VMEXIT 相关信息域 (EXITINFO)

该域作用与 VT 技术下的 VMEXIT 相关信息域功能类似，包括四个部分，EXITCODE、EXITINFO1、EXITINFO2 和 EXITINTINFO，根据产生#VMEXIT 事件的原因不同而填入不同的信息，消息格式也相应变化，因此在使用 SVM 技术时，务必参考手册正确解析这些信息。

7) 事件注入

事件注入域 (Event Injection) 位于 VMCB 结构体控制域偏移量 A8h，被用来指定在返回虚拟机上下文后要伴随发生的中断或事件。该域的格式与 VMEXIT 相关信息域中的 EXITINTINFO 一致，需要注意的是，如果注入的事件不可能在当前虚拟机所处模式下发生，那么虚拟机会关闭，并返回 VMEXIT_INVALID 错误类型。

8) 分支跳转记录虚拟化 (Last Branch Record Virtualization)

与 VT 技术相比，SVM 技术实现了更强大的对分支跳转记录 (Last Branch Record, LBR) 的虚拟化功能，可被用于对当前系统的调试和性能分析。其主要包括如下四项：

- LastBranchFromIP: 跳转源地址
- LastBranchToIP: 跳转目标地址
- LastExceptionFromIP: 记录除#DB 和 ICEBP 外其余异常/中断造成跳转的源地址
- LastExceptionToIP: 记录除#DB 和 ICEBP 外其余异常/中断造成跳转的目标地址

虚拟机状态保存域

VMCB 虚拟机状态保存域 (State Save Area) 位于 VMCB 结构体后 3076 字节上，这部分记录了虚拟机在退出虚拟机上下文时的相关寄存器信息：

1) ES,CS,SS,DS,FS,GS 的下列各项

- 段选择子
- 属性 (访问权限)
- 基址
- 段长

2) GDTR、LDTR、IDTR 和 TR 信息

- 段选择子
- 属性 (访问权限)
- 基址
- 段长

3) 当前所处特权级 CPL

4) 处理器特性寄存器 EFER

5) 控制寄存器 CR0、CR3、CR4 等

- 6) 调试寄存器 DR7、DR6
- 7) 状态寄存器 RFLAGS
- 8) RIP, RSP
- 9) SYSENTER_CS、SYSENTER_RSP、SYSENTER_RIP、STAR、LSTAR、CSTAR MSR 寄存器¹
- 10) 分支跳转记录 MSR 寄存器

关于 SVM 中 MSR 访问的拦截和 MSR Permissions Map

关于 VMSAVE 和 VMLOAD

关于 VT 和 SVM 下的事件注入机制 (Event Injection)

关于 Global Interrupt Flag (GIF)

批注 [S10]: 第四章已出现

批注 [S11]: 第四章已出现

批注 [S12]: 第四章已出现

批注 [S13]: 第四章已出现

不同于 VT 结构实现, 在 SVM 技术中, Hypervisor 的状态不保存在 VMCB 中, 而是保存在 VM_HSAVE_PA MSR 寄存器指定的物理地址上。

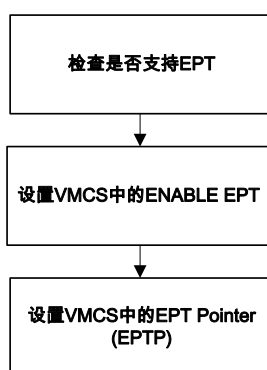
批注 [S14]: 阶段 1 初始化过程也提到这点, 最后需统一怎样写

HEV 中的双层地址翻译

为了提高虚拟机的执行效率, AMD 和 Intel 都在各自的硬件虚拟化技术产品中加入了对双层地址翻译的支持。从而加速了从虚拟机线性地址到真实物理内存地址的翻译。在技术文档中, VT 技术对应的是扩展页表技术 (Extended Page Table, EPT), 而 SVM 技术对应的是嵌套页表技术 (Nested Page Table, NPT), 需要注意的是, 由于该技术只是硬件虚拟化技术的一个辅助技术, 因此无论是 AMD 还是 Intel 都未在旗下所有支持 HEV 技术的处理器上实现²。本节我们将详细分析这两种技术。

VT 扩展页表技术

启用过程



¹ STAR、LSTAR、CSTAR MSR 寄存器保存执行 SYSCALL 指令时跳转地址, SYSENTER_CS、SYSENTER_RSP、SYSENTER_RIP MSR 寄存器保存执行 SYSENTER 指令时跳转地址。

² 目前, Intel 的 Core i7 系列处理器已支持 EPT 技术。开发时, 请参考手册, 根据 CPUID 相应功能号返回结果确定目标平台对 EPT/NPT 支持情况。

图 2.12 EPT 启用过程

启用 EPT 前首先要检查 VMCS 结构体内基于处理器的执行控制域（Processor-based VM-execution Control）bit 31 是否为 0，获得当前处理器对 EPT 的支持情况。随后将该位置 1（ENABLE EPT 置位）。之后需要在 VMCS 结构体的虚拟机执行控制域（VM-execution control）指定 EPTP（Extended-Page-Table Pointer）。

翻译过程

由于在 VMM 下和虚拟机下分别存在地址翻译过程，因此 EPT 在使用时要靠 EPTP 和 CR3 两者所指向的页表，前者指向 Hypervisor 中用于将虚拟机物理地址翻译到真实物理地址的 EPT 页表，后者指向虚拟机中用于将虚拟机线性地址翻译到虚拟机物理地址的虚拟机页表。需要注意的是，CR3 存储的地址是虚拟机物理地址，而不是真实物理地址，但是在 EPT 技术中可以选择是否利用 EPT 页表翻译 CR3 所存储地址。

EPT 下的地址翻译过程如图 2.13 所示，主要分四个过程¹：

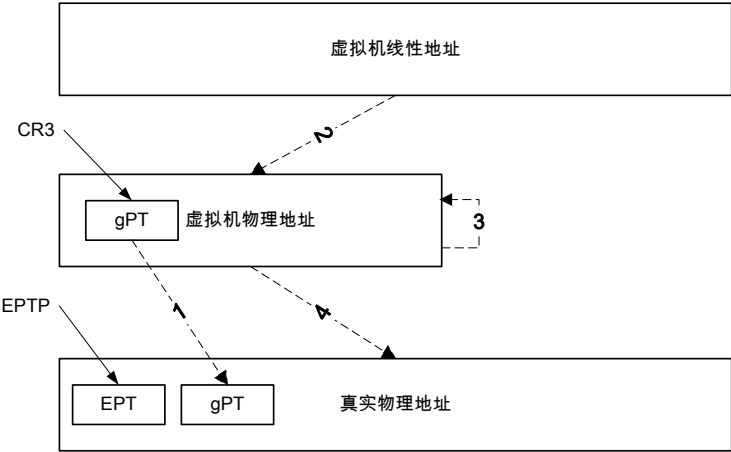


图 2.13 EPT 地址翻译过程

- 1. 利用 EPT 页表将 CR3 所指 gPT（虚拟机页表，下同）顶级地址翻译为真实物理地址，从而可以找到 gPT 的真实内容。
- 2. 根据 gPT 将虚拟机线性地址翻译为虚拟机物理地址。
- 3. 检查该虚拟机物理地址的访问权限。
- 4. 根据 EPT 将虚拟机物理地址翻译到真实物理地址。

关于页面异常

EPT 在翻译过程中，可能发生两种导致#VMEXIT 事件的页面异常：EPT 页表配置错误(EPT

¹ 并不意味着仅有 4 步，因为虚拟机页表每级页表项存储的都是虚拟机物理地址，这些物理地址所指向的次级页表/页面只有在遵照 EPT 地址翻译过程翻译后才可使用。

Misconfigurations) 和 EPT 页表违规 (EPT Violations), 这两者都是在利用 EPT 页表翻译过程中发生异常所致。

EPT 页表配置错误 (EPT Misconfigurations) 常用于说明 EPT 页表中某个页表项内容错误, 比如违背了读写规则或指定规则不被当前处理器支持等。

EPT 页表违规 (EPT Violations) 更接近于普通页面异常, 比如访问了某个未生效 (Invalid) EPT 页面。

对于由此造成的 #VMEXIT 事件, Hypervisor 开发者可以根据 VMX 退出原因和退出信息域获得充足的信息。

SVM 嵌套页表技术

启用过程

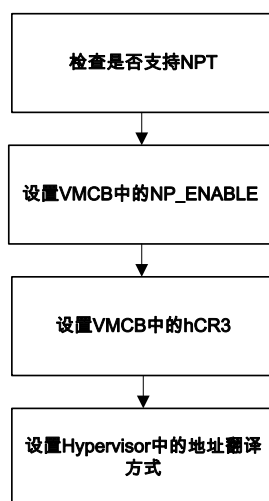


图 2.14 NPT 启用过程

启用 NPT 的过程很简单, 首先通过检查 CPUID.8000_000A:EDX[0] 来确定当前平台是否支持 NPT, 随后设置 VMCB 结构体中的 NP_ENABLE 开启虚拟机的 NPT 翻译模式。之后需要指定在 VMM 中进行额外地址翻译页表的 hCR3 物理地址 (Host CR3, hCR3), 最后要注意的是, 还需要在 Hypervisor 中指定地址翻译方式, 因为这额外的地址翻译步骤遵照 Hypervisor 中指定地址翻译方式进行翻译。

翻译过程

与 EPT 技术不完全相同, NPT 在使用时存在两个 CR3, 分别是 nCR3 和 gCR3, 前者相当于 EPT 技术中的 EPTP, 后者相当于 EPT 技术中的 CR3, 将虚拟机线性地址翻译到虚拟机物理地址。同样的, gCR3 存储的地址是虚拟机物理地址, 而不是真实物理地址。

由此, NPT 下的地址翻译过程如图 2.15 所示, 由于其与 EPT 基本相同, 故不再赘述其地址翻译过程。

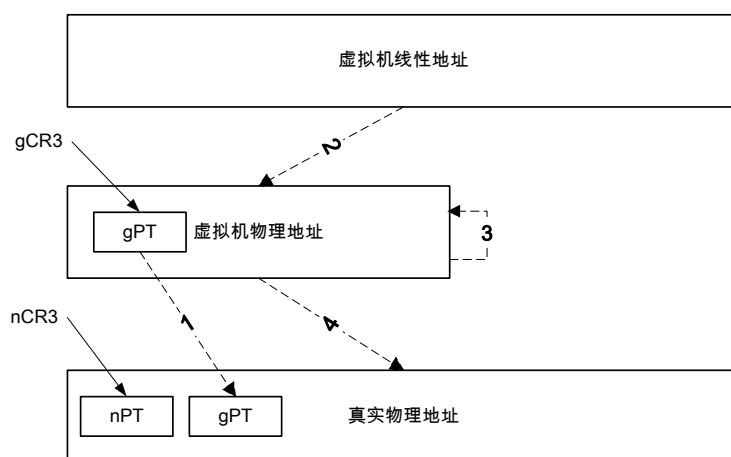


图 2.15 NPT 地址翻译过程

关于页面异常

在 NPT 地址翻译的四步中，每一步均有可能发生页面异常，其中第 1、4 两个过程会导致 #VMEXIT(NPF)，而 2、3 两个过程只是导致一般的 #PF。

对于前者，由于产生了 #VMEXIT 事件，因此会陷入 Hypervisor 中，对此 Hypervisor 开发者可以检查 VMCB 结构体中的 EXITINFO2 域以获得造成陷入的待翻译虚拟机物理地址，在 EXITINFO1 中记录了页面异常原因。

AMD 的安全性（15.25）

总结

SVM 和 VT 不同之处和使用时应注意的地方

通过前文的描述，看上去 SVM 技术和 VT 技术十分相似，但是实际上两者还是有一些不同之处。在开发过程中必须注意到这些不同之处，它们是正确并且高效实现 Hypervisor 的关键。

SVM 的开发逻辑中，VMRUN 和事件处理程序要处于同一循环中，这是因为 Hypervisor 的事件处理程序入口在 VMRUN 的下一条地址上，而在 VT 技术中，由于可以自由指定这个入口地址，因此可以在 VMCS 块中指定一个函数作为事件处理入口函数。

SVM 采用 ASID 作为 TLB 中 Guest 和 Hypervisor 地址的标记，而 VT 采用 VPIDs (Virtual-Processor Identifiers) 作为 TLB 中不同虚拟机地址翻译的缓存标记，因此 VT 技术的缓存策略更精细所以更好些。

虽然 SVM 技术和 VT 技术都可以管理中断，管理资源，访问控制，但两者具体处理行为有一些差别，VT 技术将能造成 #VMEXIT 事件的指令分为两种：无条件陷入的指令和有条件陷入的指令。SVM 技术则分为了异常拦截和指令拦截，其中一些异常虽然会造成陷入，但是同时也会自动标记相应的异常寄存器 (Exception Specific Registers)。应用的时候一定要根据手册上的描述给出相应的实现。

SVM 和 VT 技术在使用时一定要注意到，#VMEXIT 事件的产生来源于异常而不是行为，比如用户可以拦截 RDMSR 指令，但是发生的 sysenter 指令却不能拦截到，是因为在这种情况下，虽然 sysenter 有读取 MSR 寄存器的操作，但是因为没有提前在 VMCS/VMCB 中设定处理器遇到 sysenter 产生异常，所以处理器执行到 sysenter 指令当然也就不会产生 #VMEXIT 事件。

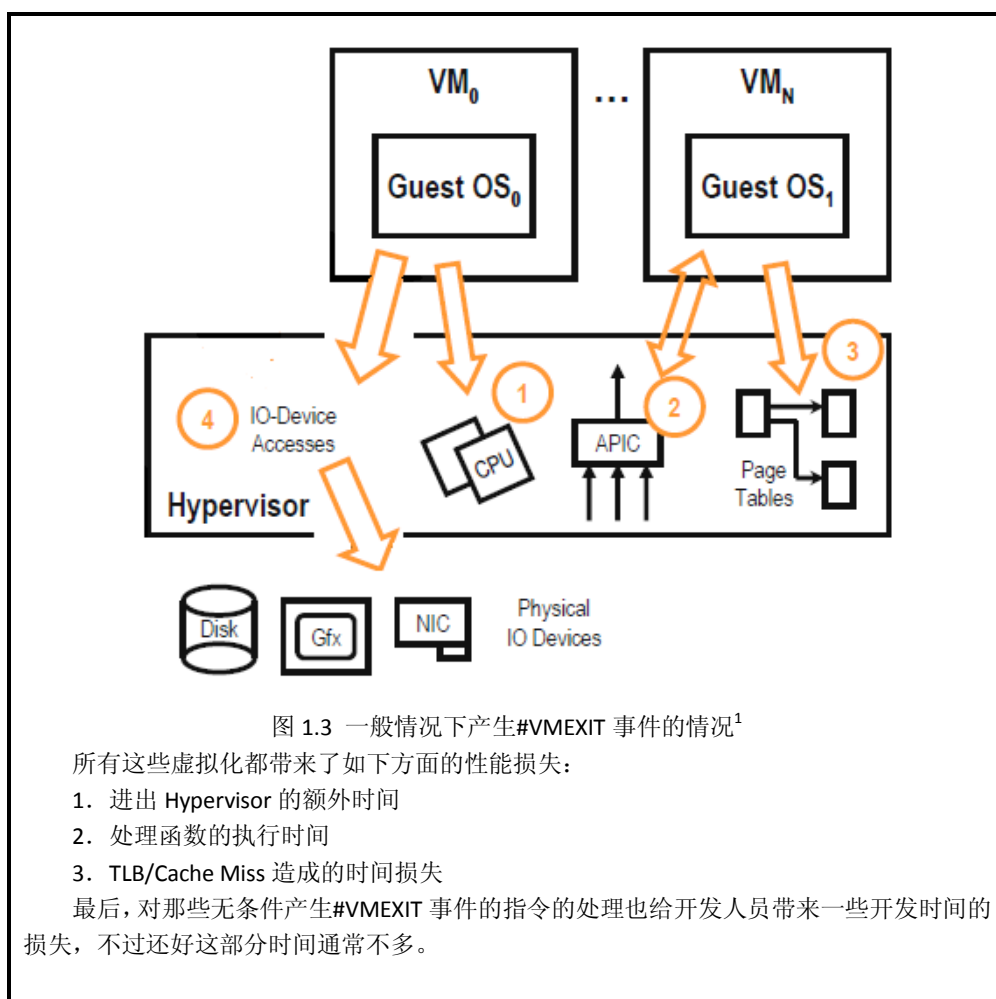
HEV 技术所带来的性能损耗

新技术在使得开发人员的世界变得更加美好的同时，也不可避免的带来性能上的冲击。

HEV 技术中，性能损耗最大的地方在于 Hypervisor 的引入及其所造成的需要进出 Hypervisor。一个最简单的例子，在普通的 x86 保护模式下，运行时刻执行到 CPUID 指令时，处理器会根据 EAX (RAX) 寄存器的值直接读取 MSR 寄存器，并把结果写到 EAX~EDX (RAX~RDX) 寄存器。但是在 Guest 模式下并且设置对 CPUID 指令进行拦截，那么每当 Guest OS 执行到 CPUID 指令时，处理器都会产生 #VMEXIT 事件，从而陷入 Hypervisor 中对该指令进行相应处理，这个过程中涉及到 Guest 模式寄存器的保存，Host 模式寄存器的恢复，填充 VMCS/VMCB 中相应内容 (都是一系列处理器自动完成的内存操作)，然后 Hypervisor 中不可缺少的有 CPUID 指令陷入的处理，最后在 Hypervisor 处理完后，处理器要回到 Guest 模式，这又涉及到 Host 当前寄存器的保存，Guest 模式寄存器的恢复，以及 VMCS/VMCB 中相应内容的填充。显然，花费在这上面的指令周期数将是保护模式下 CPUID 一条汇编指令所消耗的指令周期数的成千上万倍以上。

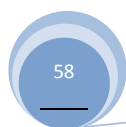
而在更一般的情况下，下列四种产生 #VMEXIT 事件的情况都是需要处理的：

1. 访问特权级别的 CPU 的状态 (Access to Privileged CPU State)
2. 中断虚拟化 (Interrupt Virtualization)
3. 页表虚拟化 (Page-Table Virtualization)
4. IO 设备虚拟化 (IO-device virtualization)



Note 关于此章内容更详细的信息，请参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* 和 *AMD64 Architecture Programmer's Manual, Volume 2: System Programming, Chapter 15 Secure Virtual Machine*

¹ 此图摘自 *Intel® Virtualization Technology Processor Virtualization Extensions and Intel® Trusted execution Technology*, Gideon Gerzon



PART2 深入研究 NewBluePill

批注 [S15]: 在介绍了 2 家的技术后，加一些自己的总结，或者对比。不要光是介绍内容，要有自己的分析。

四、 NewBluePill 的启动和卸载

在这一章中，我们将探究 NewBluePill 驱动的启动和关闭过程，从而将 NewBluePill 各组件串接起来（本章不涉及 dbgclient.sys 的启动过程，“第八章 NewBluePill 调试系统”会对其加以说明）。启动和卸载过程在 NewBluePill 中占据很大的比重，这一点可以从相关代码所占总代码量比重上看出：约 30% 的源文件均与启动和卸载过程有关。所以了解 NewBluePill 的启动和卸载，将对了解其功能实现有极大帮助。

在后续章节中，我们将逐一探索每个组件是如何完成其功能的。

批注 [S16]: 第八章 NewBluePill 其它系统
章号

NewBluePill 驱动的启动过程

NewBluePill 驱动入口在 common\newbp.c 文件中，入口函数为 DriverEntry 函数（Newbp.c 第 46 行）。这个函数流程图如图 4.1 所示：

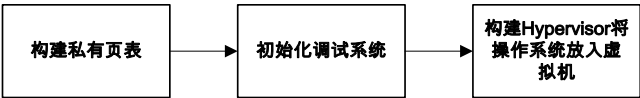


图 4.1 NewBluePill 启动流程图

下面我们将按照图 4.1 逐一介绍每部分运行过程。

构建私有页表

在 DriverEntry() 函数中，从 58 行到 62 行，以及从 79 行到 114 行，都是在完成私有页表的构建。（对于内存相关部分，本章中我们只是列举出被调用的函数，每个函数的详细作用我们会在“第五章 NewBluePill 内存系统”中阐述）

批注 [S17]: “第五章 NewBluePill 内存系统”章号

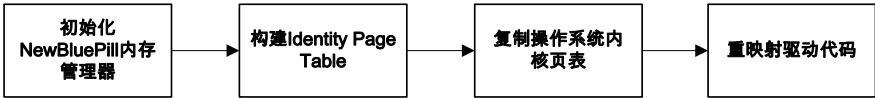


图 4.2 NewBluePill 初始化过程中构建私有页表的流程

- **初始化 NewBluePill 内存管理器** Newbp.c 中第 58 行到第 62 行，该代码调用函数 MmInitManager(), 该函数会在内存中分配新空间作为 NewBluePill 自己的页表，并按照 x64 地址翻译机制构建出页表结构。

- **构建 Identity Page Table¹** Newbp.c 中第 79 行到第 87 行
- **复制操作系统内核页表** Newbp.c 中第 89 行到第 97 行，该代码调用函数 `MmMapGuestKernelPages()`，该函数会根据当前 Windows 操作系统的内核页表内容，填充 NewBluePill 自己的页表。
- **重映射驱动代码** Newbp.c 中第 98 行到第 114 行，调用函数 `MmMapGuestPages()`。NewBluePill 作为驱动，必定要占据内核空间，此处调用该函数，就是要把自己占用的操作系统内核页面空间的页表信息复制到自己的页表中，从而为以后实现页表隐藏打下基础。

初始化调试系统

`DriverEntry()` 函数的 63 行到 75 行在初始化 NewBluePill 的调试系统。（对于调试系统部分，本章中我们只是列举出被调用的函数，每个函数的详细作用我们会在“第八章 NewBluePill 调试系统”中阐述）

NewBluePill 初始化本地调试窗口，是通过调用函数 `DbgRegisterWindow()` 实现的，这个函数主要作用是根据当前 NewBluePill 驱动实例的唯一 ID（驱动运行时读取处理器时间寄存器（Time Stamp Counter, TSC），并将低八位作为这个 ID），分配一段共享内存，并将打印信息全部保存在这段内存上，从而使得本机调试变得方便。（`dbgclient.sys` 会读取这段共享内存的内容并发送到调试机上，其实也可以调整下让它将这些内容保存在磁盘上）

NewBluePill 也直接支持利用串口发送调试信息到调试机上（不需 `dbgclient.sys` 的帮助）

构建 Hypervisor 并将操作系统放入虚拟机

构建 Hypervisor，并将操作系统放入虚拟机的工作，是在 `DriverEntry` 函数中的 116 行到 132 行完成的，主要调用的函数有两个：

- `HvmlInit()` 函数
- `HvmSwallowBluepill()` 函数

`HvmlInit()` 函数的作用是：确定当前系统架构是否支持 HEV 技术，并确定 NewBluePill 支持哪种 HEV 技术（Intel VT/AMD SVM）。最后根据获得的信息，将相应的处理函数组和平台信息捆绑在 `Hvm` 结构体上，该结构体可以通过在 windbg 下输入 `dt Hvm` 命令实现。

实验：查看 Hvm 结构体

在 NewBluePill 运行时，您可以在 windbg 下使用 `dt Hvm` 命令查看当前平台对应的 Hvm 结构体内容：

`Lkd`

批注 [S18]: 给出该实验的实验结果

实际上，检查是否支持 HEV 技术，和确定平台的函数（两者都由 `Hvm->ArchIsHvmImplemented()` 实现）在后面的 `HvmSubvertCpu()` 函数中（被

¹ 关于 Identity Page Table 的详细细节会在“第五章 NewBluePill 内存系统中”介绍

HvmSwallowBluepill()调用,后面会讲到)再次出现,我们认为重复检查是不必要的。

HvmSwallowBluepill()函数的作用是:该函数及其子函数给每个处理器(Processor)安装 NewBluePill 的 Hypervisor,这个函数也是 NewBluePill 主要逻辑的初始化入口。下面,我们就将进入这个入口背后的世界。

进入 NewBluePill 的世界

当我们进入了 HvmSwallowBluePill()函数,也就踏入了 NewBluePill 的世界。为了便于理解,我们人为的把启动过程分为两个阶段,进入虚拟机模式前的初始化部分称为阶段 1 初始化(Phase 1),进入虚拟机模式后的初始化部分称为阶段 2 初始化(Phase 2)。

阶段 1 初始化

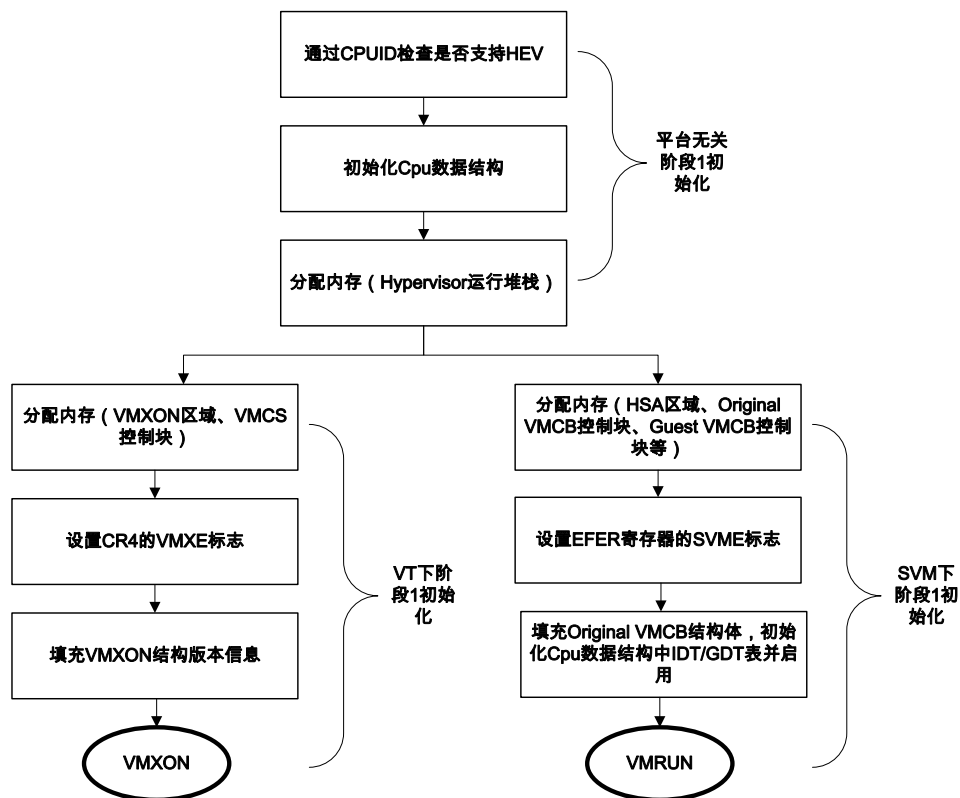


图 4.3 阶段 1 初始化流程图

阶段 1 初始化主要任务是为每个处理器开启虚拟机模式。

HvmSwallowBluePill() 函数首先会在操作系统范围内申请一个互斥锁 g_HvmMutex (common\Hvm.c 第 556 行),当多个 NewBluePill 驱动同时启动/卸载时,该锁能够保证每个时刻只能有一个 NewBluePill 实例运行,以避免嵌套设置失败。随后, HvmSwallowBluePill() 函数遍历每个处理器 (common\Hvm.c 第 558 行~581 行)调用 CmDelverToProcessor() 函数,进而通过回调手段调用 HvmSubvertCpu() 函数,后者

用来指导对 VT/SVM 两种平台执行同样的侵染过程。

HvmSwallowBluePill() 函数中对每个处理器遍历的代码如图 4.4 所示, 注意这个模式所采用的手法, 遍历内部传入了回调函数, 并且既要处理 CmDeliverToProcessor() 的运行结果, 又要处理回调函数的运行结果。该模式在安装和卸载过程中均有出现。¹

```

00558: for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors; cProcessorNumber++) {
00559:
00560:     _KdPrint (("HvmSwallowBluePill(): Subverting processor #d\n", cProcessorNumber));
00561:
00562:     Status = CmDeliverToProcessor (cProcessorNumber, CmSubvert, NULL, &CallbackStatus);
00563:
00564:     if (!NT_SUCCESS (Status)) {
00565:         _KdPrint (("HvmSwallowBluePill(): CmDeliverToProcessor() failed with status 0x+08hX\n", Status));
00566:         KeReleaseMutex (&g_HvmMutex, FALSE);
00567:         HvmSpitOutBluepill ();
00568:         return Status;
00569:     }
00570:
00571:
00572:     if (!NT_SUCCESS (CallbackStatus)) {
00573:         _KdPrint (("HvmSwallowBluePill(): HvmSubvertCpu() failed with status 0x+08hX\n", CallbackStatus));
00574:         KeReleaseMutex (&g_HvmMutex, FALSE);
00575:         HvmSpitOutBluepill ();
00576:         return CallbackStatus;
00577:     }
00578:
00579:     return Status;
00580: }
00581: } end for cProcessorNumber=0;cP... ?
00582:
00583: KeReleaseMutex (&g_HvmMutex, FALSE);
00584:
00585: if (KeNumberProcessors != g_uSubvertedCPUs) {
00586:     HvmSpitOutBluepill ();
00587:     return STATUS_UNSUCCESSFUL;
00588: }

```

图 4.4 HvmSwallowBluePill() 函数中对每个处理器遍历的代码

CmDeliverToProcessor() 函数用于提高中断优先级 (DISPATCH_LEVEL IRQL), 并在指定的处理器上执行一个函数, 从而可以不被打断²的在该处理器上执行这个函数, 同时在执行最后会恢复到原先执行该指令的处理器组设置上 (也称亲核性, CPU Affinity, 参见 Common.c 代码第 359 行)。在安装过程中, 通过它来执行 HvmSubvertCpu() 函数, 从而保证 HvmSubvertCpu() 函数的运行都作用于指定的处理器。

Important 在操作系统中断优先级高于等于 DISPATCH_LEVEL 的情况下, 不能使用可分页的内存, 因为访问一个已换出页的内存地址会发生一个内存页换入 (swap-in) 操作, 而该操作是在较低的中断优先级 (确切的说: APC_LEVEL) 进行的。这种情况下, 操作系统会蓝屏。

正确的做法是: 如果确实要在 DISPATCH_LEVEL 或更高的中断优先级下分配内存, 则要从操作系统的不分页内存池 (Nonpaged pool) 中分配内存, 该段空间在操作系统存活期间内驻留在内存。

我们可以通过研究 MmAllocatePages() 函数 (common\Paging.c 第 275 行) 和其调用者出现的逻辑位置看到 NewBluePill 驱动是怎样注意这个问题的。

HvmSubvertCpu() 函数主要做了两件事情:

- 1) 创建并初始化 CPU 数据结构 (Hvm.c 中第 365 行到 418 行, 453 行到 459 行, 后者用于构建 NewBluePill 自己的 GDT、IDT 表, 并在虚拟机模式中启用)。
- 2) 指导怎样侵染一个核。(Hvm.c 中第 420 行到 465 行, 不包括 453 行到 459 行)

¹ 在这段代码中一个小的 Bug 是 for 循环应该使用 KeQueryActiveProcessors() 函数而不是 KeNumberProcessors 变量来获得当前处理器个数, 这个 bug 会影响到支持热插拔 CPU 的系统, 后果是可能导致在这种系统上某些处理器未被侵染, 或重复侵染而导致内存泄露、系统异常等其他问题。

² 确切地说还是会被打断的, 比如时钟中断就可以打断 NewBluePill 的运行, 但是这些打断不会干扰到 NewBluePill 的运行。

CPU 数据结构定义在 Hvm.h 文件的第 53 行，该数据结构在阶段 1 中初始化，阶段 2 中被使用。查找陷入处理函数、页表隐藏以及实现 Blue Chicken 反虚拟机探测技术都需要这个结构体的帮助，可以说，这个结构体是继 VMCS/VMCB 之后第二重要的结构体。下面先详细解释下这个数据结构

```
typedef struct _CPU
{
    PCPU      SelfPointer;          /*必须放在第一个，与处理事件逻辑有
    关*/

    union
    {
        SVM      Svm;
        VMX      Vmx;
    };                             /*标示所运行在何种平台上 SVM/VT，里
    面含有构建该平台下 Hypervisor 和虚拟机的关键信息*/

    ULONG      ProcessorNumber;     /*该处理器在操作系统中的处理器序号
    */

    ULONG64     TotalTscOffset;     /*实际上未被用到*/

    LARGE_INTEGER LpicBaseMsr;     /*实际上未被用到*/
    PHYSICAL_ADDRESS LpicPhysicalBase; /*实际上未被用到*/
    PCHAR       LpicVirtualBase;   /*实际上未被用到*/

    LIST_ENTRY  GeneralTrapsList; /*该处理函数列表用于处理一般原因陷
    入的 VMEXIT 事件，比如对 cr0-cr4 寄存器的操作*/

    LIST_ENTRY  MsrTrapsList;      /*该处理函数列表用于处理操作 MSR 寄
    存器而产生的 VMEXIT 事件，比如 rdmsr 指令和 wrmsr 指令*/

    LIST_ENTRY  IoTrapsList;       /*该处理函数列表用于处理因 I/O 操作
    而产生的 VMEXIT 事件，实际上没有用到1*/

    PVOID       SparePage;         /*存储一个空页面（虚假页面）的引用，
    用于后面的页表隐藏操作*/

    PHYSICAL_ADDRESS SparePagePA; /*SparePage 原来的物理地址*/
    PULONG64     SparePagePTE;     /*存储 SparePage 页所对应页表结构中
    的 Page Table Entry*/

    PSEGMENT_DESCRIPTOR GdtArea;   /*NewBluePill 自己的 GDT 表空间*/
    PVOID        IdtArea;          /*NewBluePill 自己的 IDT 表空间*/
};
```

¹ 可能算一个 bug，当前确实有 I/O 造成的 VMEXIT 事件的处理函数，但是相关信息却并未放到 IoTrapsList 链表中。不过影响不大。MsrTrapsList 也有相似的问题，只有 NewBluePill 对 SVM 技术支持的实现中用到了这个链表。

```

        PVOID      HostStack;          /*NewBluePill Hypervisor 在该处理器上
的运行时堆栈空间，16 页大小，这段空间顶端放置了 CPU 结构体*/
        BOOLEAN    Nested;             /*是否是嵌套 NewBluePill*/

#ifdef INTERCEPT_RDTSCs

        // variables for RDTSC tracing and cheating
        ULONG64     Tsc;                /*当因为 RDTSC 指令陷入时，要返回的
虚假时间*/
        ULONG64     LastTsc;           /*上次因为 RDTSC 指令陷入返回的虚假
时间*/
        ULONG64     EmulatedCycles;    /*两次 RDTSC 指令陷入间，NewBluePill
积累的时间欺骗量*/
        int         Tracing;           /*两次 RDTSC 指令陷入间，NewBluePill
还能记录欺骗时间的指令数 */
        int         NoOfRecordedInstructions; /*两次 RDTSC 指令陷入间，
NewBluePill 已经记录欺骗时间的指令数 */

#endif

#ifdef BLUE_CHICKEN

        int         ChickenQueueSize;  /*用于记录 ChickenQueueTable 当前大
小*/
        ULONG64     ChickenQueueTable[CHICKEN_QUEUE_SZ]; /*用于记录每次
陷入 Hypervisor 时的时间寄存器值*/
        int         ChickenQueueHead, ChickenQueueTail; /* 指 向
ChickenQueueTable 头尾元素位置*/

        UCHAR       OriginalTrampoline[0x600]; /*NewBluePill 卸载时，用
于构造弹簧床代码的内存区域*/

#endif

        ULONG64     ComPrintLastTsc; /*上次 Com 口输出时间，调试系统会利用这
个值*/

    } CPU,
    *PCPU;

```

实验：查看 Cpu 结构体

在 NewBluePill 调试状态下，您可以在 windbg 下，通过 bp 命令设置断点，使得被调试机系统停在 NewBluePill 包含 Cpu 结构的函数内，然后使用 dt Cpu 命令查看 Cpu 结构体内容。

比如加载 NewBluePill 驱动后，当驱动停在 CmDebugBreak() 后，在 windbg 中输入 bp VmxDispatchCpuId,然后按 F5。当驱动停在 VmxDispatchCpuId() 后，输入 dt Cpu:

Lkd

批注 [S19]: 给出该实验的实验结果

有一点需要注意，那就是在 HvmSubvertCpu() 函数中处理 SparePage 的地方 (hvm.c 第 414 行)，在为 Cpu 结构体的 SparePagePTE 赋值的时候：
Cpu->SparePagePTE = (PULONG64)((((ULONG64)(Cpu->SparePage) >> 9) & 0x7fffffff8)+PT_BASE);
通过该方法获得 SparePage 页面在页表中的 Page Table Entry。

关于页表项状态位

在 Hvm.c 的 417 行：
*Cpu->SparePagePTE |= (1 << 4);
这条语句是在设置其所指向的 PTE 是否禁用 Cache。关于硬件页表项各状态位的含义，可以参考 Windows Internals, 4th Edition 的 Chapter 7. Memory Management 中的 Address Translation 部分中的相关内容，或者 Intel/AMD 手册中的相关部分。

HvmSubvertCpu() 函数还指导了 NewBluePill 如何侵染一个核。过程按先后顺序分为三步：

- a) Hvm->ArchRegisterTraps(Cpu) 注册 VMEXIT 事件处理函数
- b) Hvm->ArchInitialize(Cpu, CmSlipIntoMatrix, GuestRsp)
开启虚拟机模式，成为 Hypervisor 并构建虚拟机以装入原来的操作系统
- c) Hvm->ArchVirtualize(Cpu) 开启虚拟机

可以看到，实际上 Hvm 结构体起到了硬件抽象的作用，对上层屏蔽了 SVM 和 VT 技术两者的差异，提高了代码的复用性。下面，我们将深入 NewBluePill 适应于每个技术的具体实现，查看阶段 1 是如何完成的。

VT 技术下阶段 1 的初始化

在 NewBluePill 对 VT 技术的支持中，HvmSubvertCpu() 函数中 Hvm 结构体的三个函数对应关系如下：

表 4.1 Hvm 结构体中三函数对应关系

Hvm 结构体中函数	映射函数 (vmx\Vmx.c 和 vmx\Vmxtraps.c)
Hvm->ArchRegisterTraps()	VmxRegisterTraps()
Hvm->ArchInitialize()	VmxInitialize()
Hvm->ArchVirtualize()	VmxVirtualize() ¹

¹ 该函数在阶段 2 初始化过程中被用到，因此不在本节中介绍。

VmxRegisterTraps 函数

首先是 VmxRegisterTraps() 函数，该函数通过调用 TrInitializeGeneralTrap() 函数将各事件与处理函数捆绑起来成为一个新的 Trap 元素，并通过调用 TrRegisterTrap() 函数将这个 Trap 存入 Cpu 结构的 GeneralTrapsList 链表中¹。同时，由于 MSR 操作、I/O 操作、和其他通用操作间的不同，所以在 Trap 元素中又使用了另外的类型来存储三者 Trap 处理特定的信息。Trap 元素及其相关 NBP_TRAP_DATA_GENERAL、NBP_TRAP_DATA_MSR、NBP_TRAP_DATA_IO 结构体定义如下：

```
typedef struct _NBP_TRAP
{
    /*_NBP_TRAP 结构体存储了 Trap 类型, Trap 发生原因, Trap 处理函数等一系列的信息,
    用于在发生该类型 Trap 时搜索适当的处理函数*/
    LIST_ENTRY    le;                /*Windows 系统链表元素必备域, 且必须在
    头部*/

    TRAP_TYPE     TrapType;          /*记录该 Trap 的类型, 可以是
    TRAP_DISABLED、TRAP_GENERAL、TRAP_MSR、TRAP_IO 之一*/
    TRAP_TYPE     SavedTrapType;     /*用于在该 Trap 元素被 Disable 时, 备份
    TrapType, 以供在该元素被 Enable 时恢复*/

    union
    {
        NBP_TRAP_DATA_GENERAL    General;
        NBP_TRAP_DATA_MSR        Msr;
        NBP_TRAP_DATA_IO         Io;
    };                               /*用于该 Trap 元素记录附加信息, 详细内容
    参见这三个结构体注释*/

    NBP_TRAP_CALLBACK    TrapCallback; /*记录若该 Trap 元素被选中,
    那么应该调用的事件处理函数*/
    BOOLEAN              bForwardTrapToGuest; /*记录是否应该把这个事件继续
    上传, 在处理 MSR 造成的 VMEXIT 时会用到*/

} NBP_TRAP,
*PNBP_TRAP;
```

Trap 结构体定义 (Traps.h 中第 67 行到 87 行)

```
typedef struct _NBP_TRAP_DATA_GENERAL
{
    ULONG          TrappedVmExit; /*记录该 Trap 元素对于什么原因造成的
    #VMEXIT 事件进行处理, 该域在查找过程中用作键*/
```

¹ NewBluePill 对 VT 技术的支持实现中，所有的事件处理的 Trap 元素都放到了 Cpu->GeneralTrapsList 链表中，虽然与 VT 技术特性有关，但也应该算是一个 bug，不过不影响当前运行。

```
        ULONG64    RipDelta;                /*这个值用于 NewBluePill Hypervisor 处理完
#VMEXIT 事件返回虚拟机后，Guest_Rip 上要加的字节数以跳过触发 VMEXIT 事件的指令，
如果为 0 则说明需要在陷入 Hypervisor 后，从 VMCS 中获得该值*/
```

```
    } NBP_TRAP_DATA_GENERAL,
    *PNBP_TRAP_DATA_GENERAL;
```

NBP_TRAP_DATA_GENERAL 结构体定义 (Traps.h 中第 36 行到 41 行)

```
typedef struct _NBP_TRAP_DATA_MSR
{
    ULONG32    TrappedMsr;                /*记录该 Trap 元素对哪个地址的 MSR 寄存器
访问进行拦截，在 NewBluePill 的 SVM 技术实现中被使用到*/
    UCHAR      TrappedMsrAccess;         /*记录该 Trap 元素对哪种 MSR 寄存器访问操
作进行拦截 (读/写)，在 NewBluePill 的 SVM 技术实现中被使用到*/
    UCHAR      GuestTrappedMsrAccess;    /*记录在安装本 NewBluePill Hypervisor
实例前，当前环境对该 MSR 寄存器访问操作的拦截情况 (读/写)，这也可以在嵌套
NewBluePill Hypervisor 的情况下表示外层 Hypervisor 对该 MSR 寄存器的访问拦截情况。在
NewBluePill 的 SVM 技术实现中被使用到*/
} NBP_TRAP_DATA_MSR,
*PNBP_TRAP_DATA_MSR;
```

NBP_TRAP_DATA_MSR 结构体定义 (Traps.h 中第 43 行到 49 行)

```
typedef struct _NBP_TRAP_DATA_IO
{
    ULONG TrappedPort;                /*记录该 Trap 元素对哪个地址的 I/O 操作进行拦
截，实际在 NewBluePill 中未被使用到*/
} NBP_TRAP_DATA_IO,
*PNBP_TRAP_DATA_IO;
```

NBP_TRAP_DATA_IO 结构体定义 (Traps.h 中第 51 行到 55 行)

结合“第二章 深入 HEV 技术细节”中的内容我们可以看到, 在 VmxRegisterTraps() 函数中, NewBluePill Hypervisor 对所有的无条件产生#VMEXIT 事件的汇编指令都添加了相应的处理函数, 如表 4.2 所示。

批注 [S20]: “第二章 深入 HEV 技术细节” 章号

表 4.2 VT 下 NewBluePill Hypervisor 对无条件陷入绑定的处理函数

造成无条件陷入的指令	对应硬件 VMEXIT 原因标记 ¹	处理函数 (vmx\vmxtraps.c)
VMCALL	EXIT_REASON_VMCALL	VmxDispatchVmxInstrDummy
VMCLEAR	EXIT_REASON_VMCLEAR ²	VmxDispatchVmxInstrDummy
VMLAUNCH	EXIT_REASON_VMLAUNCH	VmxDispatchVmxInstrDummy
VMRESUME	EXIT_REASON_VMRESUME	VmxDispatchVmxInstrDummy

¹ 所有这些标记定义 Vmx.h 中, 具体这些定义为什么这样取值, 请参考 Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B 手册 Appendix I: VMX Basic Exit Reasons

² 在源代码中有一个 bug, 那就是在 Vmxtraps.c 的第 518 行和 519 行两个都是 EXIT_REASON_VMCALL, 其中一个应该是 EXIT_REASON_VMCLEAR。

VMPTRLD	EXIT_REASON_VMPTRLD	VmxDispatchVmxInstrDummy
VMPTRST	EXIT_REASON_VMPTRST	VmxDispatchVmxInstrDummy
VMREAD	EXIT_REASON_VMREAD	VmxDispatchVmxInstrDummy
VMWRITE	EXIT_REASON_VMWRITE	VmxDispatchVmxInstrDummy
VMXON	EXIT_REASON_VMXON	VmxDispatchVmxInstrDummy
VMXOFF	EXIT_REASON_VMXOFF	VmxDispatchVmxInstrDummy
CPUID	EXIT_REASON_CPUID	VmxDispatchCpuid
INVD	EXIT_REASON_INVD	VmxDispatchINVD

此外，也对有条件陷入的指令添加了处理函数，表 4.3 列举了它们：

表 4.3 VT 下 NewBluePill Hypervisor 对有条件陷入绑定的处理函数

造成有条件陷入的指令	对应硬件 VMEXIT 原因标记 ¹	处理函数（vmx\Vmxttraps.c）
RDMSR	EXIT_REASON_MSR_READ	VmxDispatchMsrRead
WRMSR	EXIT_REASON_MSR_WRITE	VmxDispatchMsrWrite
CR Ops	EXIT_REASON_CR_ACCESS	VmxDispatchCrAccess
异常	EXIT_REASON_EXCEPTION_NMI ²	VmxDispatchException
RDTSC	EXIT_REASON_RDTSC	VmxDispatchRdtsc
I/O 操作 ³	EXIT_REASON_IO_INSTRUCTION	VmxDispatchIoAccess

关于这些处理函数的功能，我们会在“第六章 NewBluePill 陷入事件管理系统”进行详细讨论。

批注 [S21]: “第六章 NewBluePill 陷入事件管理系统” 章号

VmxInitialize 函数

在注册完各陷入事件处理函数后，NewBluePill 会调用 VmxInitialize() 函数进行后续的阶段 1 初始化工作。

```
typedef struct _VMX
{
    PHYSICAL_ADDRESS VmcsToContinuePA; /*用于 Nested NewBluePill 情况。由于在 VT 技术实现中不考虑 Nested NewBluePill，因此该值等于 Vmx.OriginalVmcsPA*/
    PVOID _2mbVmcbMap;

    PHYSICAL_ADDRESS OriginalVmcsPA; /*VT 技术中用于控制虚拟机的
```

¹ 所有这些标记定义 Vmx.h 中，具体这些定义为什么这样取值，请参考 Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B 手册 Appendix I: VMX Basic Exit Reasons

² Intel 手册中说明，该值 (0) 既可以表示虚拟机由于异常而产生 VMEXIT 事件，也可以表示虚拟机由于 NMI 异常而产生 VMEXIT 事件。在 NewBluePill 中，根据 VMCS 块的配置，该值代表发生异常的陷入原因。

³ 注意到在 Vmxttraps.c 第 589 行，只有在头文件中预先定义 VMX_ENABLE_PS2_KBD_SNIFFER，NewBluePill Hypervisor 才会处理访问 I/O 造成的陷入。

```

VMCS 结构体物理地址*/
    PVOID                OriginalVmcs;                /*VMCS 结构体指针*/
    PHYSICAL_ADDRESS      OriginalVmxonRPA;            /*为了开启虚拟机环境所需的
VMXON 区域的物理地址*/
    PVOID                OriginaVmxonR;              /*该指针指向 VMXON 区域*/

    PHYSICAL_ADDRESS      IOBitmapAPA;                /*下面四项分别指向两段 I/O
Bitmap, 在 VMCS 的设置中使用*/
    PVOID                IOBitmapA;

    PHYSICAL_ADDRESS      IOBitmapBPA;
    PVOID                IOBitmapB;

    PHYSICAL_ADDRESS      MSRBitmapPA;                /*指向 MSR Bitmap, 在 VMCS 的设置
中被使用*/
    PVOID                MSRBitmap;                  /*用于存储在 VMCS 结构体中使用的
MSR 位图*/

    ULONG64               GuestCR0;                  /*用于缓存虚拟机中 CR0 控制寄存
器的新值, 主要用于 CR0 寄存器造成的陷入的处理函数中。在 VmxInitialize()也被赋值*/
    ULONG64               GuestCR3;                  /*用于缓存虚拟机中 CR3 控制寄存
器的新值, 主要用于 CR3 寄存器造成的陷入的处理函数中。在安装 NewBluePill 驱动后, 虚
拟机任何更新 CR3 寄存器的操作结果都会被 NewBluePill 记录在该域中。当虚拟机重新打开
分页模式时, 该域会重新写入 CR3 寄存器。在 VmxInitialize()也被赋值*/
    ULONG64               GuestCR4;                  /*用于缓存虚拟机中 CR4 控制寄存
器的新值, 主要用于 CR4 寄存器造成的陷入的处理函数中。在 VmxInitialize()也被赋值*/
    ULONG64               GuestEFER;                 /*用于缓存虚拟机中 EFER MSR 寄存
器1的值, 在处理 MSR 和控制寄存器操作造成的陷入时均被用到。在 VmxInitialize()也被赋值
*/
    UCHAR                 GuestStateBeforeInterrupt[0xc00];

} VMX,
*PVMX;

```

VMX 结构体定义 (vmx\Vmx.h 中第 205 行到 231 行)

VmxInitialize() 函数主要职责是初始化 Cpu 结构体中的 Cpu->Vmx 项, 这之中最重要的莫过于分配 Vmxon 空间 (Vmxon Regions) (Vmx.c 文件第 519 行)、VMCS 空间 (Vmx.c 文件第 530 行)、IOBitmap (I/O 位图, Vmx.c 文件第 544 行和 554 行) 和 MsrBitmap (Msr 位图, Vmx.c 文件第 564 行)。具体这四个空间的作用可参考“第二章 深入 HEV 技术细节”中相关内容描述。

当这些结构体都已经被分配出来后, 阶段 1 的初始化工作也将要结束。在

¹ EFER MSR 寄存器 (Extended Feature Enable MSR), 这个寄存器包括了一些该处理器是否支持以及正在使用一些扩展特性的信息, 比如是否支持 x64 的信息就存储在其中。具体信息可参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A* 和网上相关资料。

批注 [S22]: “第二章 深入 HEV 技术细节” 章号

VmxInitialize() 函数的第 573 行，程序调用 VmxEnable() 函数。

VmxEnable() 函数中每个有关设置的步骤都很关键。首先设置 CR4 寄存器第 13 位为 1（该位也称 CR4.VMXE，用于执行 VMXON 开启虚拟机模式的前提）（Vmx.c 文件第 46 行），然后为了保险起见，检查 CR4 和 IA32_FEATURE_CONTROL，确保当前系统支持开启虚拟机模式。接下来在 Vmxon 区域中填充 VMCS 版本号（Vmx.c 文件第 58 行到第 59 行）。最后通过调用 VmxTurnOn() 函数（实际就是 vmxon 汇编指令）开启虚拟机模式。

至此 VT 技术下阶段 1 的初始化工作全部完成。

SVM 技术下阶段 1 的初始化

与 NewBluePill 对 VT 技术支持情况类似，在对 SVM 技术的支持中，HvmSubvertCpu() 函数中 Hvm 结构体的三个函数对应关系如下：

表 4.4 Hvm 结构体中三函数对应关系（AMD 平台）

Hvm 结构体中函数	映射函数（svm\Svm.c 和 svm\Svmtraps.c）
Hvm->ArchRegisterTraps()	SvmRegisterTraps()
Hvm->ArchInitialize()	SvmInitialize()
Hvm->ArchVirtualize()	SvmVirtualize()

SvmRegisterTraps 函数

类似于 VmxRegisterTraps() 函数，在 NewBluePill 驱动运行于 AMD 平台上时，SvmRegisterTraps() 函数用于注册各个#VMEXIT 事件的处理函数。该函数同样通过调用 TrInitializeGeneralTrap() 函数将各事件与处理函数捆绑起来成为新的 Trap 元素，也同样调用 TrRegisterTrap() 函数将各个 Trap 元素存入 Cpu 结构体相应链表中。不过不同于 VT 技术下实现的是，在对 SVM 技术的实现中，NewBluePill 还调用 TrInitializeMsrTrap() 函数（Svmtraps.c 第 834 行），使用 Cpu 结构体中的 MsrTrapsList 来另外存储 MSR 访问 Trap 元素。此外，在构建 Trap 元素时，NBP_TRAP_DATA_GENERAL 的 RipDelta 域也常常提前指定（传参为非 0 值）。

结合“第二章 深入 HEV 技术细节”中的内容，在 SvmRegisterTraps() 函数中，NewBluePill Hypervisor 对下列#VMEXIT 事件添加了相应的处理函数，如表 4.5 所示。表 4.6 列出了 AMD 平台上 NewBluePill Hypervisor 监控的 MSR 寄存器和操作。

批注 [S23]: “第二章 深入 HEV 技术细节” 章号

表 4.5 SVM 下 NewBluePill Hypervisor 对各种#VMEXIT 事件的处理函数

造成无条件陷入的指令	对应硬件 VMEXIT 原因标记 ¹	处理函数（svm\Svmtraps.c）
VMCALL	VMEXIT_VMRUN	SvmDispatchVmrunk
VMCLEAR	VMEXIT_VMLOAD	SvmDispatchVmload
VMLAUNCH	VMEXIT_VMSAVE	SvmDispatchVmsave
VMRESUME	VMEXIT_CLGI	SvmDispatchClgi

¹ 所有这些标记定义 Vmc.h 中，具体这些定义为什么这样取值，请参考 AMD64 Architecture Programmer's Manual Volume 2: System Programming 手册 Appendix C:SVM Intercept Exit Codes

VMPTRLD	VMEXIT_STGI	SvmDispatchStgi
VMPTRST	VMEXIT_SMI	SvmDispatchSMI
VMREAD	VMEXIT_EXCEPTION_DB	SvmDispatchDB
VMWRITE	VMEXIT_CPUID	SvmDispatchCpuid
VMXON	VMEXIT_RDTSC	SvmDispatchRdtsc
VMXOFF	VMEXIT_RDTSCP	SvmDispatchRdtscp

表 4.6 SVM 下 NewBluePill Hypervisor 对因 MSR 寄存器访问造成陷入的处理函数

被监控的 MSR 寄存器号	监控操作	处理函数 (svm\Svmtraps.c)
MSR_EFER	读写	SvmDispatchEFERAccess
MSR_VM_HSAVE_PA	读写	SvmDispatchVM_HSAVE_PAAccess
MSR_TSC	读	SvmDispatchMsrTscRead

在“第六章 NewBluePill 陷入事件管理系统”中，我们会详细讨论这些处理函数的功能。

批注 [S24]: “第六章 NewBluePill 陷入事件管理系统” 章号

SvmInitialize 函数

NewBluePill 调用 SvmInitialize() 函数来完成 SVM 技术实现下的阶段 1 初始化工作。

```
typedef struct _SVM
{
    PHYSICAL_ADDRESS    VmcbToContinuePA;    /*用于 Nested NewBluePill 情况，记录接下来要启动的虚拟机对应的 VMCB 结构体*/

    BOOLEAN              bGuestSVME;          /* 存储有虚拟机中的虚拟 EFER MSR 寄存器是否开启了 EFER.SVME 位（可能与 Vmcb->efer 不统一，但以该域为准，详细原因会在后续章节中描述）。*/
    PHYSICAL_ADDRESS     GuestHsaPA;          /* 存储了为虚拟机提供的虚拟 VM_HSAVE_PA MSR 寄存器内容*/

    PVMCB                GuestVmcb;          /* 指向要求当前层 NewBluePill Hypervisor 挂载的 VMCB 结构体，在开启和处理嵌套 NewBluePill 时被使用。*/
    PHYSICAL_ADDRESS     GuestVmcbPA;        /* 记录当前层 NewBluePill Hypervisor 内处理的最后一个 GuestVmcb 对应的物理地址。*/

    PVMCB                OriginalVmcb;       /* 记录用于构建当前层 NewBluePill Hypervisor 对应 VMCB 结构体的线性地址。*/
    PHYSICAL_ADDRESS     OriginalVmcbPA;     /* OriginalVmcb 对应物理地址 */

    PCHAR                OriginalMsrPm;      /* 指向记录有当前层 NewBluePill Hypervisor 要拦截的 MSR 寄存器访问的 MSR Permission Map 线性地址。*/
    PHYSICAL_ADDRESS     OriginalMsrPmPA;    /* OriginalMsrPm 所对应的物理地址*/
}
```



```

    PVMCB                NestedVmcb;           /* 指向外层 NewBluePill
Hypervisor 真正挂载的修改过的 VMCB 结构体（内层 NewBluePill Hypervisor），这也是被真正
使用的嵌套 VMCB 结构体。这些修改包括启用对 VMRUN 指令以及访问 EFER, VM_HSAVE_PA
MSR 寄存器的拦截*/

    PHYSICAL_ADDRESS     NestedVmcbPA;         /* 记录 NestedVmcb 所对应的
物理地址*/

    PUCHAR                NestedMsrPm;         /* 指向记录有 NestedVmcb 所
对应内层 Hypervisor 要拦截的 MSR 寄存器访问的 MSR Permission Map 线性地址。*/
    PHYSICAL_ADDRESS     NestedMsrPmPA;       /*记录 NestedMsrPm 所对应的
物理地址*/

    PVOID                Hsa;                 /* NewBluePill Hypervisor 状态存储
空间*/
    PHYSICAL_ADDRESS     HsaPA;               /* NewBluePill Hypervisor 状态存储
空间物理地址*/

    BOOLEAN               GuestGif;           /* 表示为虚拟机虚拟的 Global
Interrupt Flag*/

    PNBP_TRAP             TrapSMI, TrapNMI, TrapINIT, TrapDB; /* 对 SMI、
NMI、INIT、DB 和 EFER MSR 寄存器访问这几个 Trap 元素的外部引用，这些 Trap 元素会在几
个不同的函数内使用，因此在外保留对它们的引用。*/
    PNBP_TRAP             TrapMsrEfer;

    ULONG32               AsidMaxNo;          /* 当前 CPU 支持的地址空间描述
符（ASID）最大索引值*/
    BOOLEAN               Erratum170;         /*记录当前 CPU 是否存在#170 号缺
陷（Erratum #170）1*/

} SVM,
*PSVM;

```

SVM 结构体定义（svm\Svm.h 中第 42 行到 79 行）

SvmInitialize() 函数主要职责是初始化 Cpu 结构体中的 Cpu->Svm 项，这其中包括分配 NewBluePill Hypervisor 状态存储空间（Host Save Area, HSA²）（Svm.c 文件第 959 行）、Original VMCB 空间（Svm.c 文件第 967 行到 968 行）、Guest VMCB 空间（Svm.c 文件第 978 行）和 Nested VMCB 空间（Svm.c 文件第 985 行到 986 行），后两个空间的构造和变化构成了嵌套 NewBluePill 的基础。VMCB 结构体的构造和作用可参考“第二章 深入 HEV 技术细节”中相关内容描述。

批注 [S25]: “第二章 深入 HEV 技术细节” 章号

¹ 在 AMD 平台上，有一组 MSR 寄存器用于记录当前 CPU 存在的操作系统可见的 CPU 缺陷（Operating System-Visible Workarounds, OSVW），有关其详细信息，可参考 AMD64 Architecture Programmer's Manual Volume 2: System Programming 手册 Chapter 17. OS-Visible Workaround Information. 其中 Erratum #170 指不能在多核上同时运行一个以上虚拟机。

² 不同于 VT 结构实现，在 SVM 技术中，Hypervisor 的状态不保存在 VMCB 中，而是保存在 VM_HSAVE_PA MSR 寄存器指定的物理地址上。

`SvmInitialize()` 函数首先调用 `SvmCheckErratums()` 函数来检查当前 AMD CPU 是否有 Erratum #170 缺陷 (`Svm.c` 第 952 行), 后者通过检查当前 CPU 的型号信息来判定是否有该缺陷 (其实一个更好的方法是通过检查 `OSVW` 中相应寄存器) 并将检测结果记录到 `Cpu` 结构体的 `Svm.Erratum170` 项中。然后, `SvmInitialize()` 函数通过 `CPUID` 指令, 传入功能号 `8000_000A` 来获得当前 CPU 支持的地址描述符数量 (`Svm.c` 第 953 行), 并将所支持最大序号存入 `Cpu` 结构体的 `Svm.AsidMaxNo` 项中。之后, `SvmInitialize()` 函数为 NewBluePill Hypervisor 状态存储空间、Original VMCB 空间、Guest VMCB 空间和 Nested VMCB 空间分配内存 (`Svm.c` 第 959 行到 992 行), 由于对当前 NewBluePill Hypervisor 来说, 此时尚未开启嵌套 NewBluePill, 因此设置 `Cpu` 结构体中的 `Svm.VmcbToContinuePA` 为 `OriginalVmcbPA` 值 (`Svm.c` 第 995 行)。当所有内存空间均已成功分配后, `SvmInitialize()` 函数调用 `SvmSetupControlArea()` 函数来设置 Original VMCB 结构体的控制部分 (VMCB Control Area, 可参考“第二章 深入 HEV 技术细节”中相关内容描述) (`Svm.c` 第 997 行到 1000 行)。

批注 [S26]: “第二章 深入 HEV 技术细节” 章号

`SvmSetupControlArea()` 函数主要功能是在 `Cpu->OriginalVmcb` 指向的 VMCB 结构体中设置要监控的行为。该函数首先设置对 MSR 寄存器的访问监控, 在 `Svm.c` 第 493 行到 503 行, 函数先为 `MsrPm` 和 `NestedMsrPm` 这两个 MSR Permissions Map 分配内存, 由于此时只是在设置当前层 NewBluePill Hypervisor, 因此在 `Svm.c` 第 505 行, `SvmSetupControlArea()` 函数只是调用 `SvmSetupMsrInterceptions()` 函数配置 `MsrPm`。

`SvmSetupMsrInterceptions()` 函数根据每个业已注册的监控 MSR 访问的 Trap 元素来配置 `MsrPm`。可以看到, 该函数通过遍历 `Cpu` 结构体中的 `MsrTrapsList` 链表, 对其中每个 `TRAP_MSR` 类型元素调用 `SvmInterceptMsr()` 函数, 后者将根据 SVM 技术规范中对 MSR Permissions Map 的描述来配置对相应 MSR 寄存器访问的拦截。

关于 SVM 中 MSR 访问的拦截和 MSR Permissions Map

与 VT 技术一样, SVM 同样支持对 MSR 寄存器的访问拦截, 并且都使用 Bitmap 的方法来存储对细化拦截其中每个 MSR 寄存器的读写操作。在 SVM 下, 这个 Bitmap 被称为 MSR Permissions Map, 不同于 VT 技术实现中 MSR Bitmap 只占 4K 物理内存, MSR Permissions Map 要占到 8K, 这是由于 SVM 支持更大的 MSR 地址范围。

MSR Permissions Map 要求是页对齐的物理地址, 同时缓存策略为写回 (Writeback)。为了使用一块指定的 MSR Permissions Map, 其物理地址要写入到 VMCB 的 `MSRPM_BASE_PA` 域中。对于要拦截访问的每个 MSR 寄存器, 在 MSR Permissions Map 中均有两位控制, 较低位用于读控制, 较高位用于写控制——设值为 1 是即可拦截相应操作。

当 MSR 访问造成 #VMEXIT 事件后, VMCB 的 `EXITINFO1` 域记录原因。关于 MSR Permissions Map 的更详细信息, 可参考 *AMD64 Architecture Programmer's Manual Volume 2: System Programming* 手册 Chapter 15.10. MSR Interrupts。

`SvmSetupMsrInterceptions()` 函数成功返回后, `SvmSetupControlArea()` 函数会进一步调用 `SvmSetupGeneralInterceptions()` 函数配置 `OriginalVmcb` 结构体来拦截 NewBluePill 前面注册的各个 `TRAP_GENERAL` 类型的 Trap 元素所对应事件。

`SvmSetupGeneralInterceptions()` 函数首先通过设置 `VMCB[00CH(Control Area)].MSR_PROT` 位激活对 MSR 访问的拦截 (`Svm.c` 第 438 行)。随后遍历 `Cpu` 结构体中的 `GeneralTrapsList` 链表, 对其中每个 `TRAP_GENERAL` 类型的 Trap 元素调用 `SvmInterceptEvent()` 函数 (`Svm.c` 第 462 行)。由“第二章 深入 HEV 技术细节”中相

批注 [S27]: “第二章 深入 HEV 技术细节” 章号

关内容我们可以看到, 由于 SVM 技术实现中 VMCB Control Area 前 141 位与 141 个 Exit Code 是一一对应的, 因此 `SvmInterceptEvent()` 函数实际就是在根据每个 Trap 元素要监控的 Exit Code 而设置 VMCB 结构体相应位。

`SvmSetupGeneralInterceptions()` 函数成功返回后, `SvmSetupControlArea()` 函数为 `Cpu` 结构体中的 `Svm.OriginalMsRpm`、`Svm.NestedMsRpm` 等域赋值 (`Svm.c` 第 515 行到 519 行) 并填充 `OriginalVmcb` 结构体中的 `msrpm_base_pa`、`guest_asid`¹ 和 `tlb_control` 域 (针对 Erratum #170) (`Svm.c` 第 527 行到 542 行)。

当 `SvmSetupControlArea()` 函数顺利执行完后, `SvmInitialize()` 函数继续调用 `SvmEnable()` 函数来为虚拟机的开启做准备。 `SvmEnable()` 函数的设置步骤同 `VmxEnable()` 函数一样重要, 均是在设置虚拟机开启关键寄存器。首先读取 `EFER` 寄存器的值 (`Svm.c` 第 140 行), 然后考虑嵌套 NewBluePill 情况, 判断是否已经激活 `EFER[bit 12, SVM]` (`Svm.c` 第 143 行到 147 行)², 之后设置 `EFER[bit 12, SVM]` 为 1 并为了保险起见, 检查该位是否设值成功 (`Svm.c` 第 158 行), 确保当前系统支持开启虚拟机模式。

随后, `SvmInitialize()` 函数进行另外一项重要工作——设置虚拟机的初始运行状态 (`Guest State`), 这项工作通过执行 `SvmInitGuestState()` 函数完成。与 VT 技术实现下的 `VmxSetupVMCS()` 函数功能类似, 该函数配置了 VMCB 结构体的段选择寄存器、控制寄存器 (CR)、调试寄存器 (DR) 等项。下面我们就结合“第二章 深入 HEV 技术细节”中相关内容, 探索该函数配置 VMCB 结构体的细节。

`SvmInitGuestState()` 函数首先通过调用 `SvmVmsave()` 函数 (`Svm.c` 第 204 行), 执行 `VMSAVE` 汇编指令来保存当前 Processor 的一系列状态信息。

批注 [S28]: “第二章 深入 HEV 技术细节”章号

关于 VMSAVE 和 VMLOAD 与客户机状态子集

在“第一章 概述”中介绍 SVM 的相关结构和汇编指令时, 我们提到 `VMSAVE` 和 `VMLOAD` 指令保存/恢复的是客户机状态子集, 包括下列状态和寄存器内容:

- FS, GS, TR, LDTR (包括隐藏部分)
- KernelGsBase
- STAR, LSTAR, CSTAR, SFMASK
- SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP

而在 VT 技术实现中, 这些域中的一部分是可以在 VMCS 中配置的, 因此在 Hypervisor 和虚拟机中要求上述状态和寄存器内容不同时, 必须采用其它妥协处理方法。关于 `VMSAVE` 和 `VMLOAD` 更详细信息, 可参考 *AMD64 Architecture Programmer's Manual Volume 2: System Programming* 手册 Chapter 15.14. `VMSAVE` and `VMLOAD` Instructions。

批注 [S29]: “第一章 概述”章号

紧接着, `SvmInitGuestState()` 函数填充关于 IDT、GDT 表的信息 (`Svm.c` 第 214 行到 219 行)。由于 NewBluePill Hypervisor 并未使用到虚拟中断, 因此在 `Svm.c` 第 221 行, 函数将 VMCB 结构体的 `vintr` 域 (也就是 `0x60h`) 置为 0, 同样, VMCB 结构体的 `eventinj` 域 (也就是 `0xA8h`) 也置为 0, 说明 NewBluePill Hypervisor 此时不会注入任何事件到虚拟机中。

¹ 这里 `guest_asid` 赋值为 `Cpu->Svm.AsidMaxNo`, 其实完全可以采用每次构建 VMCB 时赋值一个不同的 `guest_asid` 的做法, 以利用 TLB 针对 ASID (地址描述符) 的翻译加速策略。同时要注意的是, VMCB 中 ASID 域不得为 0, 因为 ASID0 保留给 Hypervisor 使用。

² 实际可以不需要这个判断而直接设置。

关于 VT 和 SVM 下的事件注入机制（Event Injection）

在 VT 和 SVM 技术中，都提供了事件注入机制用于返回到虚拟机执行后，可以在继续执行后续指令前，首先处理一个指定的异常或外部中断。

注入的异常和外部中断与其在虚拟机中发生的情况基本上完全一样，只有少数几种情况例外。而在 Intel 和 AMD 的各自实现中，具体事件注入机制方法有一些不同，详细信息可参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B, Chapter 20.8.3 VM-Entry Controls for Event Injection* 和 *AMD64 Architecture Programmer's Manual Volume 2: System Programming* 手册 Chapter 15.19. Event Injection 一章。

随后，`SvmInitGuestState()` 函数调用 `MmCreateMapping()` 函数将 GDT 和 IDT 表所在内存区域映射到私有页表上（`Svm.c` 第 224 行到 225 行，关于私有页表的详细信息，我们会在“第五章 NewBluePill 内存系统”中介绍）并初始化 CS、DS、ES 和 SS 等段寄存器（`Svm.c` 第 234 行到 237 行），在前文中我们提到，由于执行了 `VMSAVE` 指令，因此 FS 和 GS 这两个段寄存器内容已经自动保存到 VMCB 结构体的虚拟机状态区中，对比 VT 技术下的 `VmxSetupVMCS()` 函数，后者则需要手动填充虚拟机中所有这些段寄存器。

最后，`SvmInitGuestState()` 函数设置虚拟机的特权级（CPL）、控制寄存器（CR）、调试寄存器等信息（`Svm.c` 第 244 行到 253 行）¹并设置好虚拟机在成功开启后要继续执行的指令地址（`GuestRip`）以及要利用的数据堆栈（`GuestRsp`）。至此，Original VMCB 结构体初始化完毕²。

VMCB 结构体设置完毕后，NewBluePill 在 SVM 下的初始化工作也将结束。最后，`SvmInitialize()` 函数会调用 `SvmSetHsa()` 函数（`Svm.c` 第 1017 行），通过写入 `VM_HSAVE_PA` MSR 寄存器指定 Hypervisor 运行状态保存地址。然后初始化 NewBluePill 反探测系统（`Svm.c` 第 1018 行到 1024 行，详细信息可参考“第七章 NewBluePill 反探测系统”），之后为 `Cpu` 结构体中的 `Svm.GuestGif` 赋值为 1，以表明为虚拟机提供虚拟的 Global Interrupt Flag，同时调用 `CmClgi()` 函数清空 Global Interrupt Flag，为模拟 Global Interrupt Flag 做准备。

批注 [S30]: “第五章 NewBluePill 内存系统”

关于 Global Interrupt Flag（GIF）

在 SVM 技术中，Global Interrupt Flag 是一个可以控制处理器能否处理中断和事件的位。这个位可以利用 `STGI` 和 `CLGI` 指令来设置和清除。当该位清除时，诸如 `INIT`、`NMI`、外部 `NMI` 等中断源被阻塞，只有当 `GIF==1` 时才能处理，而对于诸如由于 `DR` 寄存器匹配造成的中断命中，在 `GIF==0` 时完全被忽略。然而，同样是中断命中，如果起因是 `EFLAGS.TF` 被置位，那么无论 `GIF==0` 还是 `GIF==1`，处理器都会按照正常流程处理。

关于 GIF 的详细信息，可参考 *AMD64 Architecture Programmer's Manual Volume 2: System Programming* 手册 Chapter 15.16. Global Interrupt Flag, `STGI` and `CLGI` Instructions 一章。

`SvmInitialize()` 函数成功返回后，NewBluePill 在 SVM 平台上运行时，

¹ 在 `Svm.c` 第 253 行有一个对 VMCB 结构体中虚拟机 `rax` 初始值的设置，笔者认为在 VMCB 结构体中仅保存 `rax` 初始值是不必要的，完全可以通过利用控制进出虚拟机模式的入口来保存/恢复所有通用寄存器的值，而不是保存在 VMCB 中。在 NewBluePill 中此处赋值为 0 也可以说明并未使用该域。

² 有一点需要注意，在开发中对于 VMCB 中未使用到的域尽量设置为 0，否则在开启虚拟机时很容易不能通过 VMCB 结构体正确性检测。

HvmSubvert() 函数还会调用 SvmVirtualize() 函数, 这个函数的唯一作用就是调用 SvmVmrn() 函数 (Svm.c 第 1181 行), 后者通过执行 VMRUN 指令, 根据传入的 Cpu->Svm.VmcbToContinuePA 物理地址开启虚拟机, 要注意由于开启虚拟机后, 机器会继续执行定义在 VMCB 中虚拟机 RIP 地址上的指令, 因此这个函数是不会返回的, 如果返回则一定说明执行失败 (STATUS_UNSUCCESSFUL)。至此, NewBluePill 在 SVM 平台上的初始化工作全部完成。

阶段 2 初始化

进入虚拟机模式后, NewBluePill 开始阶段 2 初始化工作, 即在虚拟机模式下进一步初始化 Hypervisor 和构造虚拟机执行环境。由于 SVM 技术实现中进入虚拟机模式前一定要初始化完毕 VMCB 等关键结构体, 因此 NewBluePill 对 SVM 技术实现中在阶段 1 就已完成所有初始化工作, 不存在阶段 2 初始化过程。

VT 技术下阶段 2 的初始化

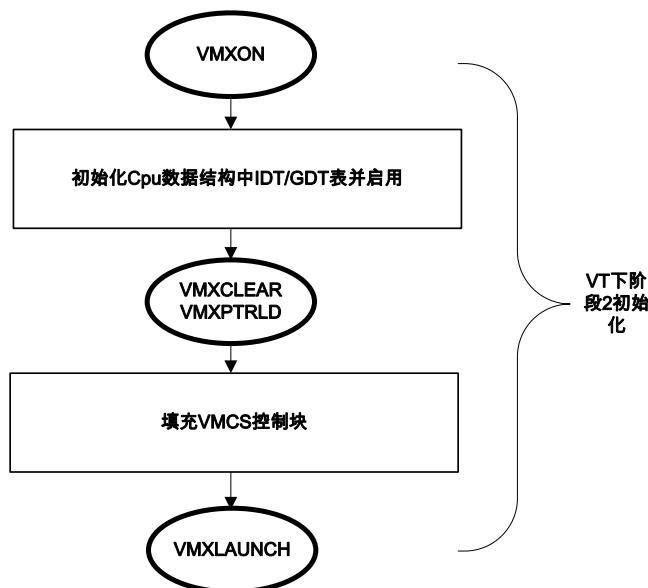


图 4.5 VT 技术下 NewBluePill 阶段 2 初始化流程图

NewBluePill 在执行完 VMXON 指令后返回到 VmxInitialize() 函数中继续运行, 该函数接下来的工作就是:

1. 初始化 VMCS 块 (Vmx.c 文件第 578 行到 584 行)
2. 填充虚拟机初始状态 (Vmx.c 文件第 588 行到 600 行)

通过阅读第二章 VMCS 结构体详细介绍, 我们可以发现 NewBluePill 遵照了 VMCS 的初始化方法, 首先在第 578 行为 VMCS 结构体的头部赋予版本号信息, 并且赋值内容参考了 MSR_IA32_VMX_BASIC MSR 寄存器, 紧接着, NewBluePill 调用了 VmxSetupVMCS() 函数进一步设置 VMCS 结构体。

VmxSetupVMCS() 函数主要功能是通过设置 VMCS 块来配置 NewBluePill Hypervisor。

函数首先设置各个段选择子（Vmx.c 文件第 282 行到 295 行），随后配置 I/O 位图和 MSR 位图，可以看到，这两部分位图在此处只是在 VMCS 中挂载了地址，而没有进行进一步的配置（在定义了监听键盘事件的情况下，I/O 位图相应项被配置）。紧接着在第 316 行，NewBluePill 依次初始化了 VMCS 块中的时间戳、VMCS Link Pointer（置为空，后面不会用到）、虚拟机调试控制寄存器等项。

随后，根据需求初始化两个重要的域：基于针脚的虚拟机执行控制（Pin-Based VM-Execution Controls）和基于处理器的虚拟机执行控制（Processor-Based VM-Execution Controls）。前者赋值为 0，说明屏蔽了由外部中断、NMI 中断、虚 NMI 和 VMX 抢占计时器造成的 #VMEXIT 事件，后者根据是否要使用 MSR 位图、监听键盘和监控 RDTSC 指令进行相应配置。

第 347 行到 349 行对是否监控页面异常进行配置，这里的配置是所有的页面异常都不会产生 VMEXIT 事件。

关于页面异常（Page Fault）产生 VMEXIT 事件

在 Intel 的手册中提到，页面异常在一定条件下可以产生 VMEXIT 事件。

当一个页面异常发生后，逻辑处理器首先用页面错误号（Page Fault Error Code, PFEC）和 VMCS 结构体中的页面错误号掩码（PFEC Mask）进行与操作，如果结果与页面错误号匹配值（PFEC Match）相等，则检查异常位图（Exception Bitmap）中 Bit 14 是否为 1，如果是则发生 VMEXIT 事件；如果结果与页面错误号匹配值不相等，则如果异常位图中 Bit 14 为 0 的话，同样也会发生 VMEXIT 事件。

因此，如果想要监控所有页面异常造成的 VMEXIT 事件，可以把异常位图 Bit 14 置 1，然后将页面错误号掩码和页面错误号匹配值置 0，反之如果不想监控页面异常，那么可以在异常位图 Bit 14 置 1 的情况下，将页面错误号掩码置 0，页面错误号匹配值置为 FFFFFFFFH。

随后在第 351 行到第 358 行，NewBluePill 配置为如果是外部中断造成的 VMEXIT 事件，那么这个中断原因会被保存在 VMEXIT 退出信息区中，同时根据当前平台决定所运行的虚拟机是虚拟 x86 平台还是虚拟 x64 平台。

从 374 行到 380 行，NewBluePill 继续初始化虚拟机，填充包括 IDT 表长，GDT 表长等信息。从 384 行到 397 行，NewBluePill 则继续初始化 Hypervisor，对 CR0、CR4 和 CR3 做处理。后面从 403 行到 418 行，NewBluePill 填充虚拟机的 CR0、CR3、CR4 和段选择子，设置好 Debug 寄存器后，在第 452 行到第 457 行，填充虚拟机将要运行的指令地址（GuestRip）和堆栈地址（GuestRsp），以及为 sysenter 指令入口填充地址。最后，NewBluePill 从第 486 行到第 494 行设置 Hypervisor 处理 sysenter 指令入口地址（其实就是虚拟机的 sysenter 处理地址，没有用到这个域），并设置好 VMEXIT 事件发生后进入 Hypervisor 的指令地址，也可以看作是 VMEXIT 事件的统一处理入口，这个入口指向 VmxVmexitHandler() ¹ 函数，并且设置处理函数用堆栈，注意其地址是 Cpu 结构体地址，这是故意这样设置的，因为在处理过程中会利用这样的栈设置操作 Cpu 结构体 ²。至此 VMCS 初始化完毕。

VMCS 结构体设置完毕后，NewBluePill 的初始化工作也将要结束。在 VT 平台上运行时，HvmSubvert() 函数还会调用 VmxVirtualize() 函数，这个函数的唯一作用就是执行

¹ 该函数及其功能会在“NewBluePill 陷入事件管理系统”一章中进行描述。

² 具体原因可参考网上关于函数调用规范的资料。

VMXLaunch 指令¹，正式启动虚拟机（Vmx.c 第 768 行），要注意的是这个函数不应该返回，因此如果返回则一定表明 STATUS_UNSUCCESSFUL。

至此，NewBluePill 在 VT 平台上的初始化过程全部完成。

NewBluePill 驱动的卸载过程

研究完启动过程后，接下来我们看下 NewBluePill 的卸载过程。卸载代码的入口是 DriverUnload() 函数（Newbp.c 第 20 行），与启动过程对应的，卸载过程流程如下：

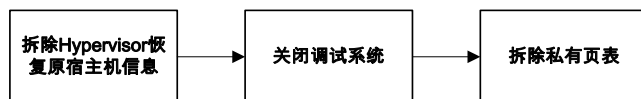


图 4.6 NewBluePill 卸载过程流程图

下面我们将按照图 4.6 逐一介绍每部分运行过程。

拆除 Hypervisor 恢复原宿主机信息

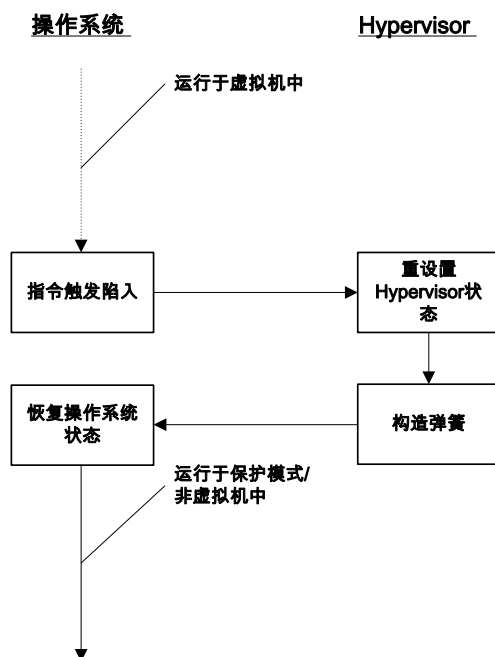


图 4.7 NewBluePill 卸载 Hypervisor 及恢复宿主机信息流程图

NewBluePill 卸载 Hypervisor 及恢复操作系统原先运行环境流程如图 4.7 所示。首先通过调用 HvmSpitOutBluepill() 函数（Newbp.c 第 31 行）来执行拆除 Hypervisor，并把操

¹ 在 x86 平台上还会将 Cpu 结构体放置在 VMCS 中 Host_Stack 指定的位置。

作系统放回原保护模式的工作，这项工作也是整个卸载过程中要做的主要工作。与启动过程不同的是，无论是 VT 技术还是 SVM 技术都没有提供直接的指令支持自动完成这一过程，甚至在 SVM 的实现中都没有用于显式关闭虚拟机模式的指令。同时又要求必须陷入到 Hypervisor 内部卸载，因此我们要注意 NewBluePill 为了实现卸载 Hypervisor 所采用的手法。

¹

HvmSpitOutBluepill() 函数的作用是：该函数及其子函数通过 Hypercall 的手段为每个处理器卸载 NewBluePill Hypervisor 并进行后续恢复工作。可以看到，这里所采用的手法与 HvmSwallowBluepill() 函数类似，因此不再具体解释该函数具体实现。

在 Hvm.c 第 529 行，调用 HvmLiberateCpu() 函数，该函数又通过调用 HcMakeHypercall() 函数（Hvm.c 第 491 行）向 Hypervisor 送出申请卸载的消息（NBP_HYPERCALL_UNLOAD）。HcMakeHypercall() 函数会根据下层具体实现 HEV 技术的硬件（Intel/AMD）来执行相应的 Hypercall 实现在 Hypervisor 下的卸载。

下面我们分别介绍 NewBluePill 对 VT 技术和 SVM 技术实现的卸载过程：

VT 技术下 NewBluePill 的卸载实现

在 VT 技术下，NewBluePill 的卸载是通过 VMCALL 指令陷入的，并且在陷入过程中同样通过参数传入消息 NBP_HYPERCALL_UNLOAD，不过遗憾的是，默认情况下，NewBluePill 并没有在 VmxHandleInterception() 函数（Vmx.c 第 128 行）中对其进行处理，换句话说，默认情况下 NewBluePill Hypervisor 在 Intel 平台上是不能正常关闭的²。

不过我们可以注意到，在 VmxHandleInterception() 函数中有这样一行（Vmx.c 第 164 行）：

```
Hvm->ArchShutdown (Cpu, GuestRegs, TRUE);
```

虽然默认情况下没有开启 BLUE_CHICKEN 开关，但是 Blue Chicken 作为一种反虚拟机探测技术，在感知到有软件在尝试探测虚拟机时，会暂时拆除 Hypervisor 和虚拟机，所以实际上，利用这个函数是可以实现卸载 Hypervisor 功能的。

在 NewBluePill 对 VT 技术支持的实现中，这个函数指向 VmxShutdown() 函数，它负责具体卸载 Hypervisor 的步骤，同时为恢复操作系统在非虚拟机中执行设置弹簧床。该函数首先通过调用 VmxGenerateTrampolineToGuest() 函数（Vmx.c 第 740 行）设置弹簧床。

进入 VmxGenerateTrampolineToGuest() 函数可以发现，该函数通过传入的一块新内存作为弹簧床代码区，然后利用 CmGenerateMovReg() 函数将保存的虚拟机状态生成一系列机器码填入到该内存中，这些机器码对应的汇编指令功能是：将存储的虚拟机寄存器内容填充到 Hypervisor 对应寄存器中，从而 Hypervisor 和虚拟机环境完全一样，这就为虚拟机模式的拆除后代码继续执行做好准备，要注意的一点是要继续执行的虚拟机指令，通过探索 VmxGenerateTrampolineToGuest() 函数可以发现，该指令地址被放在执行 VmxGenerateTrampolineToGuest() 函数堆栈的栈顶位置（体现为 VmxGenerateTrampolineToGuest() 函数最后一次对 CmGeneratePushReg() 函数的调用就是为了将 Guest_RIP+VM_EXIT_INSTRUCTION_LEN 的值压入堆栈，Vmx.c 第 695 行，如图 4.8 所示），由于弹簧床作为函数形式调用，因此根据函数调用规范，该地址作为函数返回地址，因此后续操作系统/应用程序代码可以继续执行。

¹ 实际上 NewBluePill 在卸载过程中是有 bug 的，在运行时也会发现卸载它会死机，不过这并不影响我们理解 NewBluePill 的卸载手法。

² 在后面的移植 NewBluePill 的实验中，我们要解决这个 bug。

批注 [S31]: 做实验看看 NewBluePill 是否能正确卸载。

```

00686:     if (bSetupTimeBomb) {
00687: #ifdef BLUE_CHICKEN
00688:     CmGenerateMovReg (&Trampoline[uTrampolineSize], &uTrampolineSize, REG_RAX, (ULONG64) HvmSetupTimeBomb);
00689: #endif
00690:     } else {
00691:     CmGenerateMovReg (&Trampoline[uTrampolineSize], &uTrampolineSize, REG_RAX,
00692:                     VmxRead (GUEST_RIP) + VmxRead (VH_EXIT_INSTRUCTION_LEN));
00693:     }
00694:     CmGeneratePushReg (&Trampoline[uTrampolineSize], &uTrampolineSize, REG_RAX);
00695:     CmGenerateMovReg (&Trampoline[uTrampolineSize], &uTrampolineSize, REG_RAX, GuestRegs->rax);
00696:     CmGenerateMovReg (&Trampoline[uTrampolineSize], &uTrampolineSize, REG_RAX, GuestRegs->rax);
00697:     CmGenerateMovReg (&Trampoline[uTrampolineSize], &uTrampolineSize, REG_RAX, GuestRegs->rax);
00698: #ifdef _X86_
00699:     CmGenerateIretq (&Trampoline[uTrampolineSize], &uTrampolineSize);
00700: #else
00701:     CmGenerateIretq (&Trampoline[uTrampolineSize], &uTrampolineSize);
00702: #endif
00703: }

```

之后未出现
Push操作

图 4.8 VmxGenerateTrampolineToGuest () 函数继续执行后续虚拟机指令的方法

VmxGenerateTrampolineToGuest () 函数执行完后, VmxShutdown () 函数将会执行 VmxDisable () 函数做最后拆除 Hypervisor 的工作, 该函数执行 VMXOFF 指令并清空 CR4[VMXE]位, 最后, 函数以代码形式执行弹簧床代码 (Vmx.c 第 744 行), 至此, VT 平台下虚拟机模式正式关闭。

SVM 技术下 NewBluePill 的卸载实现

在 SVM 技术下, NewBluePill 的卸载过程被完全实现。与 VT 平台上过程类似, NewBluePill 在 SVM 平台上的卸载是通过调用 CPUID 指令陷入的, 并且在陷入过程中同样传入消息 NBP_HYPERCALL_UNLOAD。一旦陷入 Hypervisor 后, NewBluePill 首先在 SvmDispatchCpuid () 函数中判断是否接收到卸载请求 (Svmtraps.c 第 609 行), 如果是则调用 HcDispatchHypercall () 函数正式开始 NewBluePill 在 SVM 平台下的卸载工作, 这里要注意的是, 由于卸载 NewBluePill Hypervisor 后原虚拟机操作系统会被移回做为宿主机操作系统继续执行, 因此在 HcDispatchHypercall () 函数中有必要先手动将虚拟机下次运行指令地址前移跳过正在执行的 CPUID 指令 (Hypercalls.c 第 45 行到 46 行)。随后调用 SvmShutdown () 函数完成卸载过程主要任务, 同时由于最后卸载完成后要继续执行原虚拟机中下一条指令而没有机会再从 HcDispatchHypercall () 函数返回。

与 VmxShutdown () 函数功能类似, SvmShutdown () 函数负责 NewBluePill 在 SVM 平台上的具体卸载 Hypervisor 的步骤, 并为恢复操作系统在非虚拟机中执行设置弹簧床, 后者通过调用 SvmGenerateTrampolineToLongModeCPL0 () 函数 (Svm.c 第 1159 行) 完成。

在 SvmGenerateTrampolineToLongModeCPL0 () 函数中, 该函数同样是将传入的新内存作为弹簧床代码区, 并利用 CmGenerateMovReg () 函数将存储在 VMCB 结构体中的虚拟机寄存器内容填充到 Hypervisor 对应寄存器中去, 为拆除虚拟机模式做准备, 具体过程可参考前文中对 VT 技术下 NewBluePill 卸载过程的描述。

随后, SvmShutdown () 函数设置 GIF, 开中断 (Svm.c 第 1161 行到 1162 行), 恢复该 NewBluePill Hypervisor 实例安装之前的 GIF 状态 (在嵌套 NewBluePill 存在情况下会造成陷入), 然后利用 Svm.c 第 1164 行的判断来确保在嵌套 NewBluePill 存在情况下不会误关闭所有虚拟机, 而 SvmDisable () 函数可以清空 EFER.SVME 位并清空 VM_HSAVE_PA MSR 寄存器, 从而恢复两者在装载 NewBluePill Hypervisor 前的状态。最后, 函数以代码形式执行弹簧床代码 (Svm.c 第 1167 行), 至此, SVM 平台下虚拟机模式正式关闭。

关闭调试系统

DriverUnload() 函数的 38 行到 40 行在关闭 NewBluePill 的调试系统。(调试系统函数的详细作用我们会在“第八章 NewBluePill 调试系统”中阐述)

NewBluePill 关闭本地调试窗口，是通过调用函数 DbgUnregisterWindow()实现的，这个函数主要作用是向窗口对应设备发送命令，通知其输出所有缓存信息然后撤销 NewBluePill 端为此设置的共享内存。

拆除私有页表

DriverUnload() 函数最后会关闭内存管理器，由于此时操作系统已经在使用 NewBluePill 自己维护的页表，所以 NewBluePill 分配的内存映射可从中找到。在释放页面时 MmShutdownManager()会遍历一个内存信息链表，该链表原来用于存储 NewBluePill 私有内存的内存信息，通过该链表逐一释放自身所占每个页面。

至此，NewBluePill 的卸载过程全部完成。

五、NewBluePill 内存系统

一个普通的 Hypervisor 完全可以借助于操作系统的内存管理来满足自身需求，但是我们在第一章中提到，NewBluePill 意在证明可以利用硬件虚拟化技术来实现一个高隐蔽性的 Hypervisor，为了提高其透明性，NewBluePill 实现了内存隐藏技术，通过使用自己的内存系统来完成运行时的内存请求，如内存分配、内存回收等。

本章中，我们首先简单介绍 x64 下的地址翻译过程，然后我们着重分析 NewBluePill 的内存隐藏技术的实现和相关函数的功能。

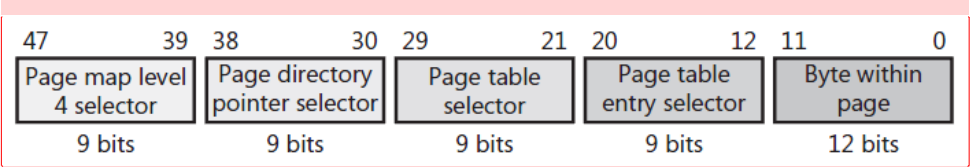
相关文件

```
NewBluePill-0.32-public\common\Paging.c
NewBluePill-0.32-public\common\Paging.h
NewBluePill-0.32-public\common\Common.c
NewBluePill-0.32-public\common\Common.h
```

x64 下的地址翻译

在保护模式下，CPU 发出线性地址，内存管理单元（Memory Management Unit, MMU）根据当前 CR3 寄存器所指向的页表物理地址将该线性地址翻译为物理地址进行内存访问，该过程称为地址翻译。

在 x64 体系结构中，线性地址的结构如图 5.1 所示。不同于 x86 体系结构，每级页表寻址部分长度变为 9 位，这是由于在 x64 体系结构中，普通页大小仍为 4K，然而数据表示却为 64 位长，因此一个 4K 页在 x64 体系结构下只能包含 512 项内容。所以为了保证页对齐和以页为单位的页表内容换入换出（Swap in/out），在 x64 下每级页表寻址部分长度定为 9 位。当前，x64 体系结构中在处理地址时只使用到了低 48 位，在未来更高位将被使用于寻址更多的内存。



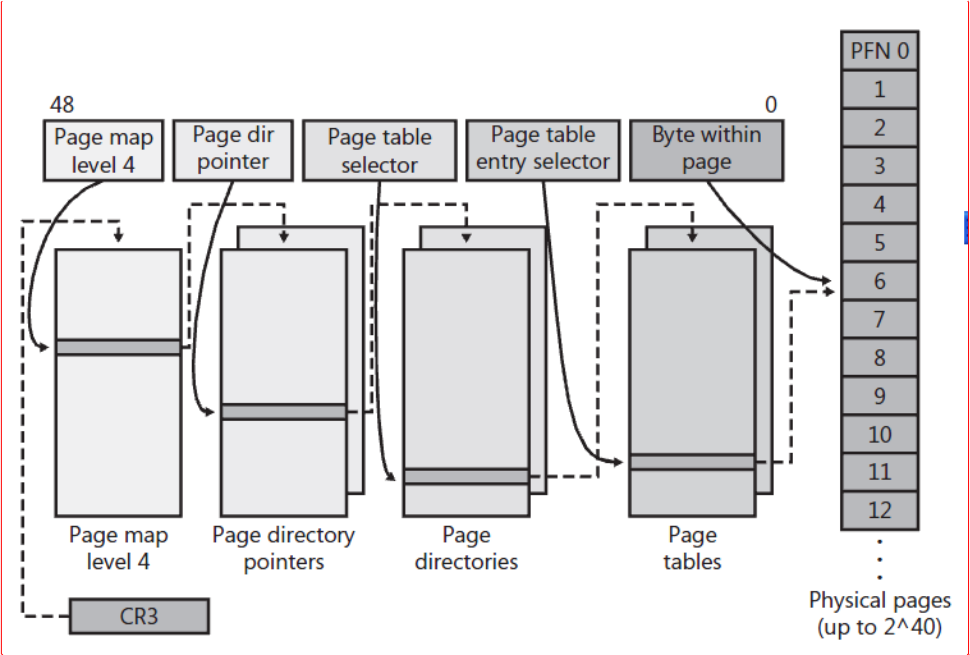
批注 [S32]: 该图摘自 Windows Internals，将来会换图

图 5.1 x64 虚拟地址结构

为了正确翻译 x64 线性地址，其页表也从 x86 下的两级变为四级，这就使得 x64 的地址翻译过程如图 5.2 所示¹。在 x64 体系结构中，每级页表包含 512 项（2⁹ 个）下级页表的指

¹ 需要注意的是，图中标明物理页号上限为 2⁴⁰，而当前 x64 体系结构中仅有 Bit[12-39]这 28 位用于存储页表项，这说明当前 x64 平台支持最多 1TB 的物理内存（2²⁸⁺¹²），Windows 之所以当前支持最多 2⁴⁰ 物理页号，完全是出于对未来扩展的支持，因为在 x64 平台页表项结构中，Bit[40-51]是保留位，正好与 Bit[12-39]构成 40 位，即将这 40 位全部视作物理页号。

针，该指针称为页表项，描述了存储下级页表的物理页面（Physical Page）的物理页号（Physical Page Number, PFN）和页面状态信息，如此按级寻址直到最后一级页表项指向目标物理页面，并利用 12 位的页内偏移量在该物理页面内寻址。



批注 [S33]: 该图摘自 Windows Internals, 将来会换图

图 5.2 x64 地址翻译过程¹

实验：观察 x64 平台上 Windows 的地址翻译过程

在 Windows 系统运行时，您可以在 windbg 下使用!process 和!pte 命令查看一个线性地址翻译到物理地址的过程。其中，翻译所使用页表的 CR3 是通过!process 命令输出中的 DirBase 确定的。

Lkd

批注 [S34]: 给出该实验的实验结果

x64 也支持 2M 和 1G 的大页，通过扩展页内偏移量的长度为后 21 位和 30 位，在次低级和次高级页表上即可直接指向大页。使用大页的好处是缩短了寻址时间，但是在以页为粒度的内存管理中，大页会造成更多的页内空间浪费。

Important 除了传统翻译模式外，在 x64 平台上也有 PAE（Physical Address Extension，物理地址扩展）模式，但是 NewBluePill 驱动当前没有针对此实现另外的页表。在移植 NewBluePill 到 x86 平台时，为了提升兼容性，建议既要针对普通地址翻译过程实现内存隐藏，又要针对 PAE 模式实现内存隐藏。

关于 PAE 模式的详细信息，可参考 *Windows Internals, 4th Edition* 的 Chapter 7. Memory Management 中的 Address Translation 部分中的相关内容，或者 Intel/AMD 手册中的相关部分。

¹ 在 NewBluePill 中，这四级页表的缩写分别为 PML4、PDP、PD 和 PT，以下我们均采用 NewBluePill 中的缩写来描述每级页表。

页表项的结构如图 5.3 所示，其中后 12 位用作标志位，表示当前页面状态。在前面观察 Windows 地址翻译过程实验中，windbg 输出中类似于--DA--UW 的部分也表示页表项的标志位，但是要注意，Windbg 中页表项标志位的表示输出与硬件页表项结构与含义有一些差异。后文会介绍，NewBluePill 内存系统在构建私有页表时，同样也填充了页表项标志位。

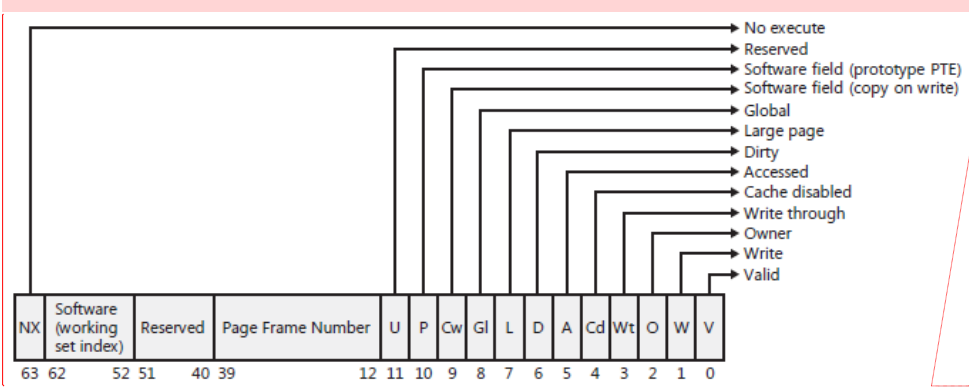


图 5.3 x64 体系结构中页表项的结构

批注 [S35]: 该图摘自 Windows Internals, 将来会换图

NewBluePill 内存隐藏技术

由于支持后于操作系统启动的动态安装/卸载特性，所以操作系统出于管理驱动的目的，在 NewBluePill Hypervisor 启动时必定会修改相应页表项，这也就影响到了 NewBluePill Hypervisor 的隐蔽性。为了弥补这个不足，NewBluePill 实现了内存隐藏技术，其做法是通过使用自己的内存管理系统维护私有页表，同时修改操作系统中有关自身的相应页表项，将其指向一个虚假页面。从而一方面从操作系统中隐藏自己，另一方面保证自己在激活后可以引用到自身内存空间，进一步提升了自身的隐蔽性，图 5.4 展示了其实现模型。

在熟悉了 x64 体系结构地址翻译过程后，下面我们就按照 NewBluePill 内存系统各函数的调用顺序开始对其各组成部分的探索。

内存系统的初始化

前私有页表初始化

前私有页表初始化（Pre-Private Page Table Initialization）主要工作是初始化辅助数据结构，并为顶级页表分配空间。在 DriverEntry() 函数的第 58 行到 62 行，NewBluePill 调用 MmInitManager() 函数，该函数也是整个 NewBluePill 内存系统的初始化函数，其主要流程如图 5.5 所示：

MmInitManager() 函数首先初始化 g_PageTableList 链表和 g_PageTableListLock 链表锁（Paging.c 第 613 行到 614 行），注意该链表在整个 NewBluePill 内存系统中是一个非常重要的数据结构，被用于存储分配内存空间描述信息，该信息将在 NewBluePill Hypervisor 安装过程中被用于指导内存隐藏工作的进行。随后函数利用 Windows API 为私有 PML4 级页表分配内存（Paging.c 第 616 行）并获得其物理地址。在第 626 行，MmInitManager() 函数调用 MmSavePage() 函数保存对刚刚分配出来的私有 PML4 级页表的描述信息，从传参中我们可

以看到，在 NewBluePill 中，HostAddress 指 NewBluePill Hypervisor 寻址用的虚拟地址，

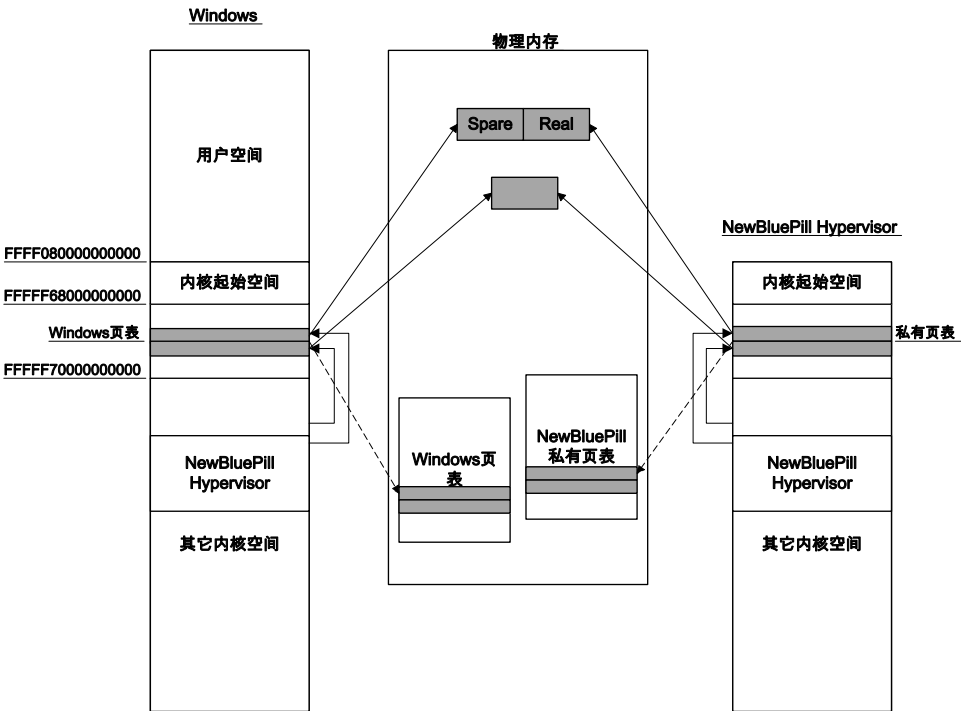


图 5.4 NewBluePill 内存隐藏实现模型

PhysicalAddress 指真实物理内存地址，而 GuestAddress 指 Guest OS 寻址用虚拟地址。在后面我们可以看到，NewBluePill 为了实现方便，把页面（非用于页表）的 GuestAddress 作为 HostAddress 直接挂在了 NewBluePill 的私有页表上。

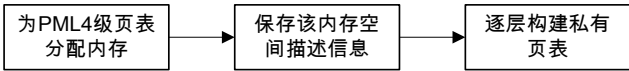


图 5.5 MmInitManager() 函数主要流程

MmSavePage() 函数主要作用是根据传入的内存空间描述信息构造并填充 ALLOCATED_PAGE 结构体实例 AllocatedPage(Paging.c 第 46 行到 59 行),最后将 AllocatedPage 插入到 PageTableList 末尾。

```
typedef struct _ALLOCATED_PAGE
{
    LIST_ENTRY    le;

    ULONG        Flags; /*用于存储 NewBluePill 内存系统自定义标志位，如 AP_PAGETABLE 等*/

    PAGE_ALLOCATION_TYPE    AllocationType; /* 该内存类型，根据 NewBluePill 内存系统定义，有 PAT_DONT_FREE、PAT_POOL 和 PAT_CONTIGUOUS 三种类型*/
    ULONG            uNumberOfPages;        /*存储内存分配页数*/
};
```

```

    PHYSICAL_ADDRESS    PhysicalAddress; /*该内存的物理页号*/
    PVOID               HostAddress; /*该内存存在 NewBluePill Hypervisor 私有页
表中的虚拟页号*/
    PVOID               GuestAddress; /*该内存存在 Windows 中的虚拟地址（这里不是虚
拟页号）*/

} ALLOCATED_PAGE,
*PALLOCATED_PAGE;

```

ALLOCATED_PAGE 结构体定义（common\Paging.h 中第 33 行到 48 行）

需要注意的是，在 MmSavePage() 函数的第 51 行到第 52 行，PhysicalAddress 仅取中间的 40 位，HostAddress 仅取高 52 位，由前面对 x64 体系结构页表结构的介绍可以得知，实际上，PhysicalAddress 由于仅是供 NewBluePill 私有页表地址翻译使用，因此更接近于物理页号的概念（或许在该变量应该使用 PhysicalFrameNumber 更清晰易懂）；而 HostAddress 之所以保留了高 52 位，是因为根据 Intel 和 AMD 手册规范，当前虚拟地址高 16 位是符号扩展虚拟地址 bit 47 后的结果，这种对 PhysicalAddress 和 HostAddress¹的处理方法在 NewBluePill 内存系统中很常见。

执行成功后，NewBluePill 回到 MmInitManager() 函数继续执行，在第 630 行处调用 MmCreateMapping() 函数开始构建 NewBluePill 私有页表。由于 NewBluePill 要保证其 Hypervisor 可以在运行时调用 Windows 函数并访问 Windows 相关变量，因此它使用 Windows x64 所定义的页表存放虚拟地址空间来存储 NewBluePill 私有页表。此处就是将 PML4_BASE 所定义的线性地址映射到自己的私有页表上，接下来，NewBluePill 开始私有页表构建的初始化工作。

关于页表存放虚拟地址空间

细心的读者可能会发现在 MmInitManager() 函数的第 625 行和 630 行都使用到了 PML4_BASE 这个常量，并且与其一组的还有 PDP_BASE、PD_BASE 和 PT_BASE 这几个常量，它们的定义如下：

```

#define PML4_BASE      0xFFFFF6FB7DBED000
#define PDP_BASE       0xFFFFF6FB7DA00000
#define PD_BASE        0xFFFFF6FB40000000
#define PT_BASE        0xFFFFF68000000000

```

这些看似很怪的取值其实是 Windows x64 版本的四级页表在虚拟地址中的起始位置，也称为页表存放虚拟地址空间，实际上，这些地址可通过以下方法算得：

```

PD_BASE    = ((PT_BASE & 0x0000FFFFFFFFF000) >> 12) * 8 + PT_BASE
            = 0xFFFFF6FB40000000

PDP_BASE    = ((PD_BASE & 0x0000FFFFFFFFF000) >> 12) * 8 + PT_BASE
            = 0xFFFFF6FB7DA00000

PML4_BASE   = ((PDP_BASE & 0x0000FFFFFFFFF000) >> 12) * 8 + PT_BASE
            = 0xFFFFF6FB7DBED000

```

显然，这种定义也造成 x64 下 Windows 512GB 的页表存储空间呈线性排列如下：

¹ 有时候也对 VirtualAddress 进行类似于 HostAddress 的处理，不过其中的道理是相同的，故不再赘述。

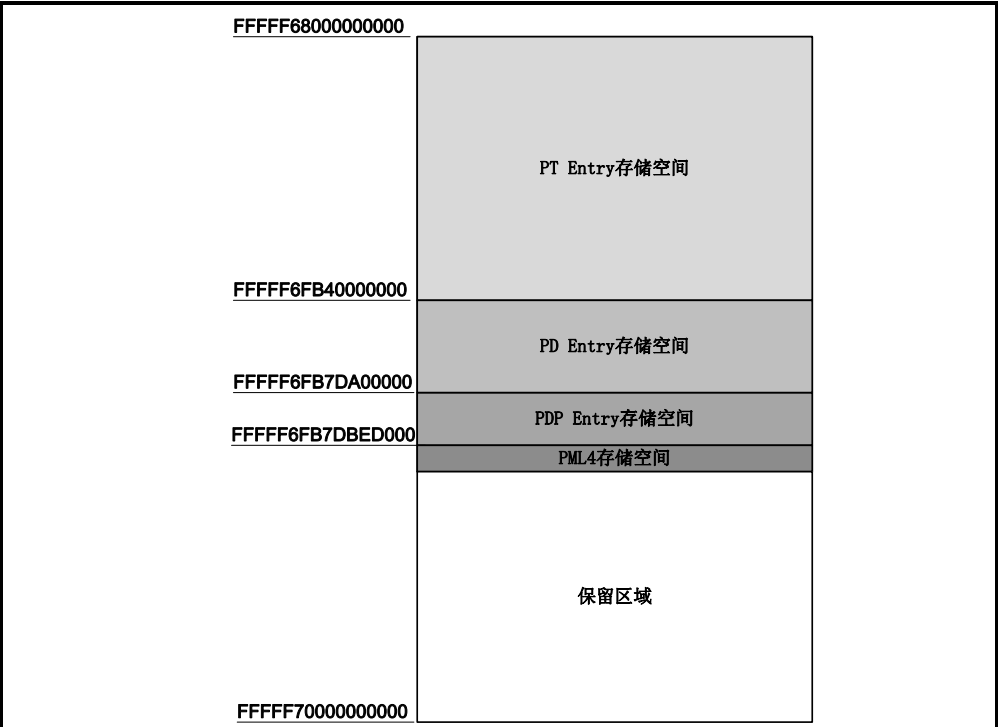


图 5.6 Windows x64 页表虚拟地址空间的线性结构

由于在这些地址上存储的都是每级的第一个页表，所以可以在 windbg 中使用!pte 命令查看 windows 中这些地址的定义。

给出输出

关于该页表存放虚拟地址空间进一步的信息，可以参考 *Windows Internals, 4th Edition* 的 Chapter 7. Memory Management 中的 Virtual Address Space Layouts 部分中的相关内容。

批注 [S36]: Windbg windows x64 下!pte 输出。

私有页表构建初始化

NewBluePill 私有页表构建的初始化工作（Private Page Table Initialization）主要靠 MmCreateMapping() 和 MmUpdatePageTable() 两个函数完成。在图 5.4 中我们已经展现了 NewBluePill 私有页表与 Windows 页表实际上必须处于同一线性地址，只是由于其中某些页表项包含的物理地址不同而实现了内存隐藏效果。

当前私有页表构建初始化工作完成后，MmInitManager() 函数最后会调用 MmCreateMapping() 函数开始逐级构建私有页表，而第一个通过构建页表映射的地址，就是 PML4_BASE 线性地址和 g_PageMapBasePhysicalAddress 物理地址（Paging.c 第 630 行）。

在 NewBluePill 中，一个新的线性地址和物理地址的映射创建到 NewBluePill 私有页表过程如图 5.7 所示，该过程在 NewBluePill 创建私有页表和利用私有页表进行内存回收分配时均会被使用到。

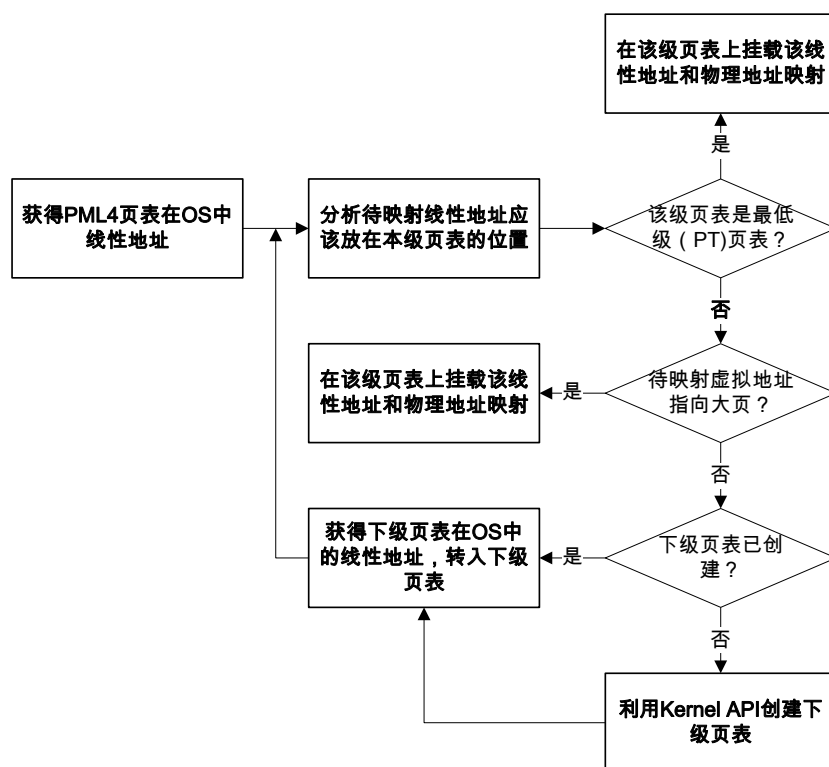


图 5.7 在 NewBluePill 私有页表中创建地址映射过程

该过程主要靠 `MmCreateMapping()` 函数和 `MmUpdatePageTable()` 函数配合完成。前者作为 NewBluePill 内存系统在私有页表上记录物理地址和虚拟地址的映射的入口函数，引导该过程完成。其主要逻辑是首先获得 PML4 页表在操作系统中的线性地址（Paging.c 第 433 行），然后调用 `MmUpdatePageTable()` 函数开始页表中地址映射的逐级创建（Paging.c 第 441 行）。需要注意的是，在对 `MmUpdatePageTable()` 函数的调用中，第一个传参是 `Pml4Page->GuestAddress`，结合前文我们对 `GuestAddress`、`HostAddress` 和 `PhysicalAddress` 的解释可以知道，此处传入操作系统下的虚拟地址可以使得 NewBluePill 很容易在 Windows 下使用这块内存，避免了直接操作物理内存的繁琐。

`MmUpdatePageTable()` 函数是一个自递归的函数，而这个递归就是用来实现图 5.7 中的对每一级页表进行操作的循环过程的。该函数首先根据传入的 `PageTableLevel` 参数（取值范围 1-4，对应 PT、PD、PDP、PML4）分析出待映射线性地址在本级页表中应处的位置（Paging.c 第 157 行到 158 行）。随后判断当前是否已经针对 PT 级页表或在 PD 级页表上进行大页映射（Paging.c 第 160 行），如果是则修改相应页表项内容完成私有页表对地址映射的记录（Paging.c 第 194 行到 201 行）然后返回；否则函数根据待映射线性地址的值计算出所对应的本级页表项所在的虚拟地址（Paging.c 第 205 行到 208 行）¹。

之后，`MmUpdatePageTable()` 函数判断 `LowerPageTablePA`（也就是本级页表项所存储的物理页号）是否为空。如果为空，则说明本级页表上待映射虚拟地址对应的子级页表尚未创建，因此 `MmUpdatePageTable()` 函数需要利用 Windows 内核函数为子级页表申请内存（Paging.c 第 215 行到 217 行），获得其物理地址（Paging.c 第 220 行）并调用 `MmSavePage()` 函数保存新申请页面信息（Paging.c 第 222 行到 230 行）。在 `MmUpdatePageTable()` 函数

¹ 计算方法可参考图 5.6 Windows x64 页表虚拟地址空间的线性结构

第 236 行，此时必定存在一个有效的 LowerPageTablePA 物理页号，因此可以完成从本级页表挂载新创建子级页表的任务（Paging.c 第 236 行到 241 行），并调用 MmCreateMapping() 函数在私有页表中记录新创建子级页表物理地址和 NewBluePill Hypervisor 下线性地址的映射关系（Paging.c 第 243 行）（记录原因与 NewBluePill 内存系统在私有页表中记录 PML4_BASE 线性地址和 g_PageMapBasePhysicalAddress 物理地址映射关系原因相同——NewBluePill 要保证其 Hypervisor 可以利用私有页表寻址到页内容，当然也包括页表内容）。如果一开始判断 LowerPageTablePA 不为空，NewBluePill 会进行一些有效性检查（Paging.c 第 253 行到 267 行）并定位开始处理下一级页表（Paging.c 第 269 行）。

Identity Page Table 的构建

NewBluePill 中还存在一个 Identity Page Table 用于对将来某时刻虚拟机环境禁用分页模式的支持。在 Newbp.c 第 79 行，NewBluePill 调用函数 MmInitIdentityPageTable() 来完成构建 Identity Page Table 的工作。

关于 Identity Page Table 和在虚拟机中构建实模式执行环境

最初支持 Intel VT 技术的处理器需要在执行硬件虚拟化扩展指令时 CR0.PE 和 CR0.PG 必须为 1，这就要求在继续执行一个虚拟机时，它的分页模式必须开启，因此虚拟机只能运行在保护模式下或者虚拟 8086 模式下（Virtual-8086 Mode）。但是在某些时候，运行在保护模式下的程序会要求暂时禁止分页机制，无疑这种情况会导致硬件虚拟化扩展指令的失效。一种变通的方法就是建立这样的 Identity Page Table 用于在客户机中模拟未分页保护模式的效果，在这种情况下，CPU 发出的地址可直接视作物理地址。

另外一个受其影响的例子是关于在虚拟机中构建实模式执行环境，由于实模式下系统不支持分页，所以为了在虚拟机中运行在实模式下，可以采用两种变通的方法：

1. 在虚拟机管理器（VMM）中构建一个指令模拟器
2. 构建一个虚拟 8086 模式的执行环境来支持实模式。

更多信息请参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B*, Chapter 26.2 Supporting Processor Operating Modes In Guest Environments。

NewBluePill 下的 Identity Page Table 是一个可寻址前 64G 物理内存空间的页表（也可以是 4G，详细描述见后文），该页表包含了 64 个 PDPTE（PDP 级页表 Entry），每个 PDPTE 又包含了 512 个 PDE（PD 级页表 Entry），每个 PDE 又指向一个 2M 的大页，因此可寻址共 $64 \times 512 \times 2M = 64G$ 大小的内存空间。从 Paging.c 第 690 行到第 723 行，NewBluePill 从低级到高级依次构建 Identity Page Table 的每一级，这其中，从构建每级页表物理页号的部分我们可以看出 Identity（同一性）的含义。

之后在 Paging.c 第 725 行到第 755 行，NewBluePill 针对是否要兼用于 4G 内存平台实现两种不同策略决定最后 Identity Page Table 的寻址大小¹，一种称之为 Long Mode 处理，一种称之为 Legacy Mode 处理。前者处理结果是 Identity Page Table 用于寻址前 64G 物理内存空间，并使用 g_IdentityPageTableBasePhysicalAddress 作为页表物理地址；而后者处理结果是

¹ 这里 NewBluePill 源代码存在一个 Bug，在 Paging.c 第 742 行处应该是 if(!FirstPdeVa_Legacy)，而且整个函数在处理 Legacy Mode 时显然会浪费一些空间。

Identity Page Table 仅能用于寻址前 4G 物理内存空间¹，其页表物理地址存储于 g_IdentityPageTableBasePhysicalAddress_Legacy 之中。

映射操作系统空间到私有页表

构建好 Identity Page Table 之后，NewBluePill 开始映射操作系统整个内核空间到私有页表上，从而达到在 NewBluePill Hypervisor 中仍能正常调用内核 API 和访问 Windows 变量的目的。

在 Newbp.c 第 89 行，NewBluePill 通过调用 MmMapGuestKernelPages() 函数完成这一功能。MmMapGuestKernelPages() 函数遍历 PML4 页表的高 256 项 (Paging.c 第 540 行)，这段空间对应操作系统内核空间，如果发现某个 PML4 项被使用，则调用 MmWalkGuestPageTable() 函数对其所指向的 PDP 页表进行进一步遍历 (Paging.c 第 541 行到 545 行)。

MmWalkGuestPageTable() 函数是挂载 Windows 内核空间各页到 NewBluePill 私有页表的主要函数，它同样使用自我递归的手段来完成遍历 Windows 内核空间页表的任务。首先进行页表项有效性检查 (Paging.c 第 483 行到 484 行)，对于任何有效的页表项，MmWalkGuestPageTable() 函数获得其指向的次级页表并逐一分析其中各项。然后，针对当前位于物理内存中的页面²，判断如果该页面挂载了一个大页或已经是 PT 级页表 (Paging.c 第 488 行到 490 行)，则分析该页并利用 MmCreateMapping() 函数将其映射到私有页表上。

在这里，分析操作系统页面的过程主要是为了获得虚拟页号所对应的 VirtualAddress 值，于是在 Paging.c 第 492 行到 496 行中，MmWalkGuestPageTable() 函数使用了类似于 MmUpdatePageTable() 函数获得 GlobalOffset 的方法，根据虚拟页号利用 Windows 中页表项全局偏移量来推算出虚拟地址。要注意此处左移 9 位和左移 18 位的操作，其原因是在 Windows x64 中，每个页表项与虚拟地址一样需要 64 位表示，也就是 8 个字节，因此相邻页表项的全局偏移量差值为 8，在此之上，分别左移 9 位和 18 位便可以得到该虚拟页号所对应 VirtualAddress 值。随后在 Paging.c 第 497 行到 498 行，函数根据前文所提到 AMD 和 Intel 的 x64 虚拟地址规范，符号扩展该虚拟地址 bit 47 位填充到高 16 位上，至此，VirtualAddress 构造完毕。之后，函数获得该页表项包含的物理页号 (Paging.c 第 500 行) 并跳过针对 Windows 页表存放区域的 VirtualAddress 地址 (Paging.c 第 502 行到 504 行)，因为在私有页表上映射该部分 VirtualAddress 地址会导致私有页表被抹去，导致自身失效。最后在私有页表上创建映射，在对大页的处理中，NewBluePill 内存系统当前做法是将其视作 512 个小页映射 (Paging.c 第 514 行到 519 行) 而对小页，则直接调用 MmCreateMapping() 函数完成映射工作 (Paging.c 第 520 行)。

该函数执行完成后，程序回到 Newbp.c 继续执行。在 Newbp.c 文件第 98 行到 104 行，NewBluePill 驱动会调用 MmMapGuestPages() 函数将自身挂载到私有页表上并将自身空间的描述信息添加到 g_PageTableList 链表尾部。至此 NewBluePill 内存系统初始化工作完成。

批注 [S37]: 如果考虑 MmMapGuestTSS64 要修改这句话。

¹ 这样做估计出于对 Windows 操作系统已有驱动兼容性的考虑，有些驱动不能良好支持 4G 以上空间的寻址，不过这样的驱动不能通过微软的 Hardware Compatibility List Testing。

² 由于操作系统内核页面同样是可以换入 (Swap in) / 换出 (Swap out) 的，一个页面当前位于物理内存中说明它最近曾经被使用或正在被使用。因此，为了得到一个操作系统可运行的鲜活镜像 (Fresh Image)，构建私有页表时仅考虑这些位于物理内存中的系统空间即可，不需考虑换出到磁盘上的内存页。

内存系统的使用

探索过整个 NewBluePill 内存系统的初始化过程，我们了解了 NewBluePill 中私有页表是如何构建的，那么这个系统又是如何被使用的呢，在使用过程中私有页表又是怎样发挥作用的呢？本节我们将着重探索其中奥秘。

私有页表上的分配内存

在 NewBluePill 内存系统中，内存的申请主要是通过 `MmAllocatePages()`、`MmAllocateContiguousPages()`、`MmAllocateContiguousPagesSpecifyCache()` 三个函数完成的，它们的作用如下表所示：

表 5.1 内存分配函数的作用

函数名	作用
<code>MmAllocatePages()</code>	分配池类型内存
<code>MmAllocateContiguousPages()</code>	分配物理地址连续的内存区域，要求处理器缓存该内存区域（ <code>MmCached</code> ）
<code>MmAllocateContiguousPagesSpecifyCache()</code>	分配物理地址连续的内存区域，要求处理器使用指定的缓存策略

这三个内存分配函数既从客户机 Windows 操作系统上申请内存，又在私有页表上进行相应的更新，这样做是因为通过在客户机上申请内存，Windows 会认为相应的物理页号已分配，因此未来这些内存存在被显式释放前是不会再分配给其它程序的，这也就保证了 NewBluePill Hypervisor 不会因为内存冲突而损坏。

由于三者逻辑基本相同，我们仅探索其中的 `MmAllocatePages()` 函数。

`MmAllocatePages()` 函数首先进行参数检查（`Paging.c` 第 285 行到 286 行），然后利用内核 API 的 `ExAllocatePoolWithTag()` 函数分配内存（`Paging.c` 第 288 行到 290 行）。获得物理地址后，对该内存的每页通过调用 `MmSavePage()` 函数保存页描述信息，注意此处区别对待首页（`PAT_POOL` 类型）和该内存区域中的后继页面（`PAT_DONT_FREE` 类型）。之后调用 `MmCreateMapping()` 函数在私有页表上映射该内存区域中的每个页面（`Paging.c` 第 306 行到 311 行）。成功执行后，新申请内存区域已经挂载到私有页表上。

实验：查看 NewBluePill 的内存使用情况

我们可以通过一些工具来查看 NewBluePill Hypervisor 的内存使用情况，以证明确实隐藏了这些空间。比如 PoolMon（Windows Internals,404 页）

批注 [S38]: 给出实验结果

私有页表的生效

结合“第四章 NewBluePill 的启动和卸载”我们可以知道，NewBluePill 私有页表的生效

批注 [S39]: “第四章 NewBluePill 的启动和卸载” 章号

是由于其物理地址 `g_PageMapBasePhysicalAddress` 在虚拟机启动时被换入 CR3 寄存器所致。由图 5.4 也可以发现, NewBluePill 私有页表可以看作是在复制了原有 Windows 的页表区域以后替代了其中对 NewBluePill 驱动的页表项和页表自身空间的页表项。由于无论是切换 CR3 内容还是复制页表, 对 Windows 而言都是开销很小很微不足道的操作, 因此 Windows 几乎无法探测到这些操作带来的后果。

内存隐藏的实现

NewBluePill 中内存隐藏的实现, 与 NewBluePill 内存系统初始化过程中的 `g_PageTableList` 链表息息相关。在 NewBluePill 驱动即将启动其 Hypervisor 前, 可以遍历 `g_PageTableList` 链表中每一项, 根据其中存储的内存块描述信息调用 `Common\Common.h` 中的 `CmPatchPTEPhysicalAddress()` 函数即可修改 Windows 操作系统页表中相应虚拟地址映射的物理地址, 该函数的核心思想就是仅修改 PTE 的物理页号部分。然而遗憾的是, 在 NewBluePill 公开代码中, 我们并没有看到相应逻辑, 这可能是由于作者出于安全考虑而在公开版本中移除了相应代码。其结果就是使用公开版本的 NewBluePill 是不能获得内存隐藏效果的。

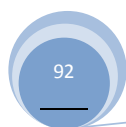
内存系统的关闭

在“第四章 NewBluePill 的启动和卸载”中我们提到, 在 NewBluePill 驱动卸载时会调用 `MmShutdownManager()` 函数完成关闭 NewBluePill 内存系统的任务。实际上, 私有页表此时已替换了 Windows 页表, 因此 `MmShutdownManager()` 函数的主要过程就是依据 `g_PageTableList` 链表存储的分配内存描述, 逐一释放每个内存区域。要注意的是, 对于数页大小的内存区域, 由于其首页内存在 NewBluePill 中视作 `PAT_POOL` 类型而通过 `ExFreePool` 函数释放了全部该内存区, 后继页面因为被视作 `PAT_DONT_FREE` 类型而被忽略, 否则会造成重复释放某一内存页的问题。

总结

NewBluePill 通过修改操作系统页表维护私有页表的方法, 进一步提升了自身的隐藏性, 使得很难通过一般方法来探测 NewBluePill 的存在。

但是 NewBluePill 并未完全公开内存隐藏部分的代码, 当前运行结果是, 它虽然构建了私有页表, 但是并未修改操作系统的页表, 因此所有的内存使用情况仍能被操作系统觉察。而且, NewBluePill 并未针对硬件 Cache 和操作系统 Cache 做太多处理, 这也潜在成为了探测 NewBluePill 的方法。



六、NewBluePill 陷入事件管理系统

在前面我们介绍 NewBluePill 的启动过程时，我们提到了在 NewBluePill 中存在一套陷入事件管理机制，同时列举出了一些事件/指令对应的处理函数。实际上，在 NewBluePill 中，陷入事件和处理函数的捆绑生成 Trap 元素、注册、激活是按照这样的流程进行的：

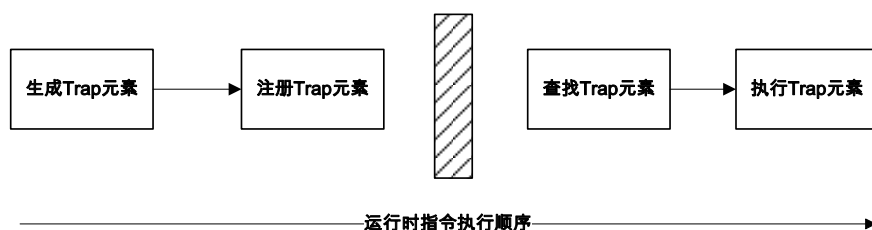


图 6.1 Trap 元素的生成、注册和执行

本章中，我们将深入研究 NewBluePill 陷入事件管理系统，并对这些处理函数的具体作用展开解释。

相关文件

NewBluePill-0.32-public\common\Traps.c
NewBluePill-0.32-public\common\Traps.h
NewBluePill-0.32-public\vmx\Vmxttraps.c
NewBluePill-0.32-public\vmx\Vmxttraps.h
NewBluePill-0.32-public\svm\Svmtraps.c
NewBluePill-0.32-public\svm\Svmtraps.h

Trap 元素的生成、注册机制

在描述 NewBluePill 的启动过程中我们已经提到，Trap 元素是 NewBluePill 用于捆绑 VMEXIT 事件和相应处理函数的基本单元，生成信息全部保存在 Trap 结构体中。

Trap 元素的生成是通过 `TrInitializeGeneralTrap()` 函数、`TrInitializeMsrTrap()` 函数和 `TrInitializeIoTrap()` 函数¹实现的。具体创建过程如下：

- 1) 为 Trap 元素分配内存
- 2) 填充 Trap 的类型和处理函数（`TRAP_DISABLED`²、`TRAP_GENERAL`、`TRAP_MSR`、

¹ `TrInitializeIoTrap()` 函数在 NewBluePill 中并未具体实现。

² 通过 `TrTrapDisable()` 函数或者 `TrTrapEnable()` 函数控制这个类型。当一个 Trap 元素被 Disable 后，它所捆绑

TRAP_IO)

3) 根据 Trap 类型初始化其它内部域

具体数据结构及各项含义在 NewBluePill 的启动过程中已经提到，故不在此阐述。

NewBluePill 陷入事件处理函数的注册是通过 `TrRegisterTrap()` 函数 (`Traps.c` 第 19 行) 实现的。这个函数的职责是根据传入的 Trap 元素的类型，将 Trap 元素插入到相应链表中 (`Cpu` 结构体的 `GeneralTrapsList`、`MsrTrapsList` 或 `IoTrapsList`)。

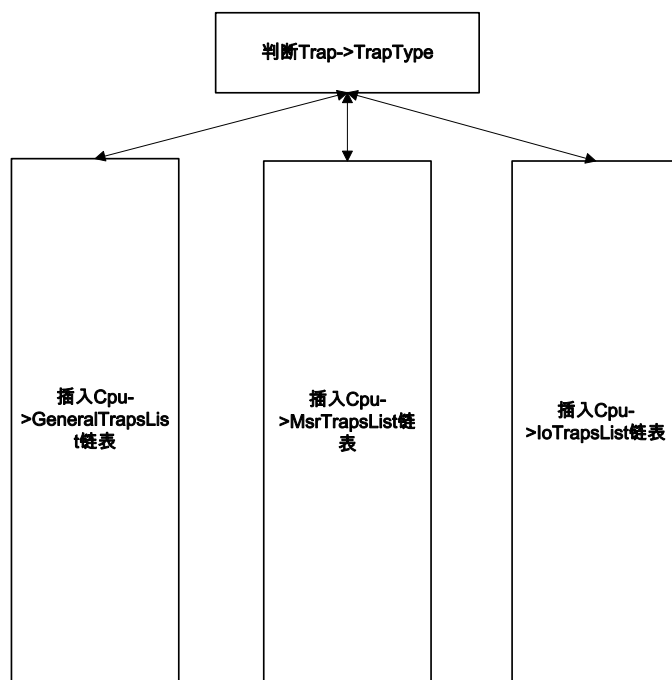


图 6.2 `TrRegisterTrap()` 函数的职责

由于使用的是 Windows 提供的链表机制，所以在反注册一个 Trap 元素的时候只需提供 Trap 元素的挂钩，即可利用 Windows API 删除整个元素，`TrDeregisterTrap()` 函数就是利用这样的机制反注册一个 Trap 元素的。

Trap 元素的触发机制

由于 VT 技术和 SVM 技术的实现差异，因此一个处理函数的触发过程在两个平台上也并不相同，我们可以人为的把它分为两个阶段，如图 6.3 所示。

的处理函数不会被执行，但是该 Trap 元素仍然存在于列表内。

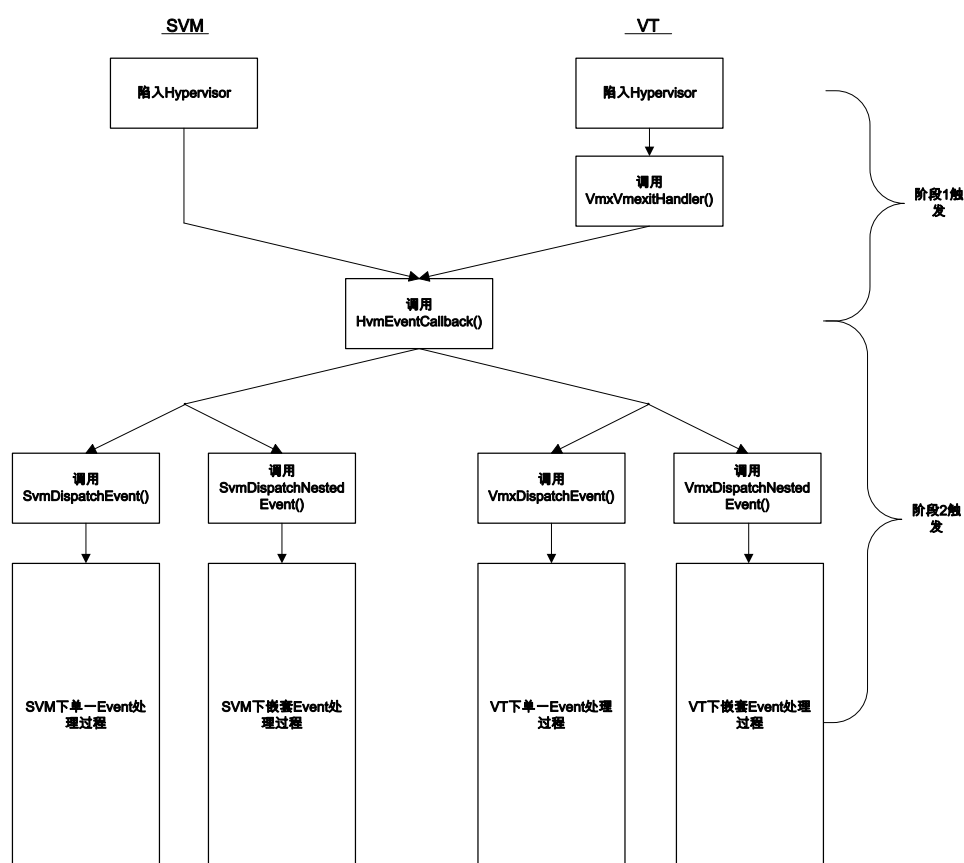


图 6.3 Trap 元素的触发机制图

阶段 1 触发

阶段 1 触发的过程比较简单，主要目标是进入 `HvmEventCallback()` 这个硬件抽象层的事件分发函数。NewBluePill 对 SVM 的实现中，在 `amd64\Svm-asm.asm` 文件第 109 行，`SvmVmrun()` 函数中，可以发现一旦发生 `#VMEXIT` 事件陷入 Hypervisor 后，会先保存各个通用寄存器以供事件处理函数在执行时可以获取虚拟机中断时的状态，然后立即执行 `HvmEventCallback()` 函数。而在 NewBluePill 对 VT 技术的实现中，VMCS 块中定义了一旦发生 `#VMEXIT` 事件陷入 Hypervisor，那么会执行 `VmxVmexitHandler()` 函数（`amd64\Vmx-asm.asm` 第 244 行），这个函数同样会保存各个通用寄存器后立即执行 `HvmEventCallback()` 函数。至此阶段 1 触发完成。

阶段 2 触发

阶段 2 触发过程比较复杂，因此我们按 VT 技术和 SVM 技术分开介绍。

VT 技术下的阶段 2 触发

首先，我们介绍下在 VT 技术中阶段 2 触发操作细节。对于 VT 技术，尽管在 `HvmEventCallback()` 函数中依旧可以调用 `Hvm->ArchIsNestedEvent()` 函数，但是由于 NewBluePill 没有针对 VT 技术实现嵌套事件分发功能，所以其对应的 `VmxIsNestedEvent()` 函数将永远返回 `FALSE`。且 `VmxDispatchNestedEvent()` 函数也不做任何事情，因此 NewBluePill 对 VT 技术阶段 2 触发的支持仅限于对单一事件的处理，`HvmEventCallback()` 函数通过调用 `Hvm->ArchDispatchEvent()` 函数（`Hvm.c` 文件第 244 行）实现这一点。

`Hvm->ArchDispatchEvent()` 函数对应函数是 `VmxDispatchEvent()` 函数，该函数会调用 `VmxHandleInterception()` 函数，并会通知它不要让虚拟机再继续处理这个事件。

`VmxHandleInterception()` 函数包含处理事件的主要逻辑，正如图 6.1 后半部分所示，它的工作包括获取陷入原因（Exit Code），指导查找 Trap 元素（`Vmx.c` 第 148 行到 153 行）、执行找到的 Trap 元素（`Vmx.c` 第 156 行到 158 行），并在打开 BLUE_CHICKEN 反 HVM 探测的情况下判断当前是否要暂时隐藏自己（`Vmx.c` 第 159 行到 167 行）。

查找 Trap 元素是通过 `TrFindRegisteredTrap()` 函数实现的（`Traps.c` 文件第 259 行）。这个函数会根据陷入原因而选择正确的 Trap 元素存储链表（`Traps.c` 文件第 275 行到 290 行），对于 Intel 平台，则一律指向 `GeneralTrapsList`。然后以 `Trap->General.TrappedVmExit` 的值为关键字逐一遍历该链表每个元素，并返回找到的第一个 Trap 元素。

执行 Trap 元素是通过 `TrExecuteGeneralTrapHandler()` 函数实现的（`Traps.c` 文件第 240 行）。该函数执行一个 Trap 元素内部捆绑的处理函数（`Traps.c` 文件第 251 行），同时最关键的是，在执行成功这个处理函数后，程序会调用 `Hvm->ArchAdjustRip()` 函数来调节虚拟机的 RIP（指令指针），使其在获得控制权后可以执行下一条指令¹，但是要注意的是，在执行 `Hvm->ArchAdjustRip()` 函数前，起到 RIP 偏移量作用的变量 `Trap->General.RipDelta` 必须已经设置好，它的设置工作通常是由各个处理函数内部实现。

最后执行的 Blue Chicken 策略，简单的说是在判断是否在短时间内发生了大量的 VMEXIT 陷入，如果发生，那么 NewBluePill 怀疑有人正在尝试探测自己，当前的实现中，NewBluePill 会通过 `Hvm->ArchShutdown()` 函数卸载自己。

当所有这些工作执行完后，`HvmEventCallback()` 函数返回，`VmxVmexitHandler()` 函数执行 `vmx_resume`（`Vmx-asm.asm` 文件第 258 行）返回到虚拟机中，虚拟机继续运行。整个 Trap 元素处理事件全部执行完成。

SVM 技术下的阶段 2 触发

接下来，我们介绍下在 SVM 技术中阶段 2 触发操作细节。由于 NewBluePill 对 SVM 技术实现中支持嵌套 NewBluePill 和嵌套事件处理，因此一个典型的嵌套 NewBluePill 环境如图 6.4 所示：

¹ 在其它场合中，如果是因为 Page Fault 陷入，那么也可以用相同的手法在换页后使虚拟机重新执行发生页面异常的指令。

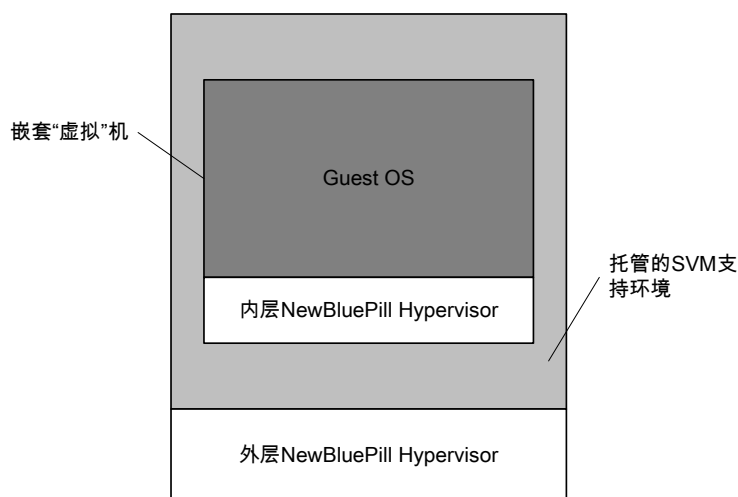


图 6.4 典型的嵌套 NewBluePill 执行形式

此时对于外层 NewBluePill Hypervisor 来说，必定有 $\text{Cpu} \rightarrow \text{Svm.VmcbToContinuePA.QuadPart}$ 不等于 $\text{Cpu} \rightarrow \text{Svm.OriginalVmcbPA.QuadPart}$ ，而对内层 NewBluePill Hypervisor 则相反。这是因为此时外层 NewBluePill Hypervisor 的 $\text{Cpu} \rightarrow \text{Svm.VmcbToContinuePA.QuadPart}$ 项存储的是内层 Hypervisor 的 $\text{Cpu} \rightarrow \text{Svm.GuestVmcbPA.QuadPart}$ 地址，具体过程在后文对 SVM 技术实现中个处理函数功能和流程中会介绍。

不同于 VT 下仅存在一个 NewBluePill Hypervisor 的简单情况，嵌套 NewBluePill 的创建和事件处理过程更加复杂，如图 6.5 所示。在后文探索 SVM 技术下对 VMLOAD、VMSAVE 和 VMRUN 指令的处理时会着重讲述这一过程每阶段的实现。

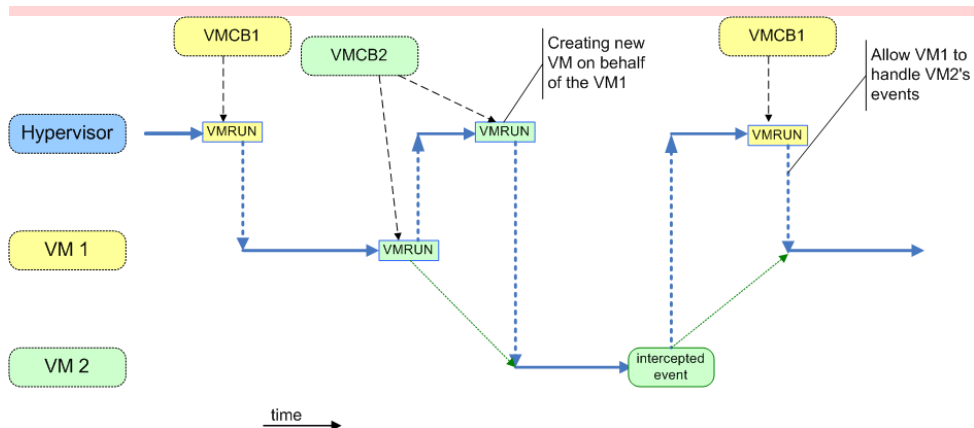


图 6.5 嵌套 NewBluePill 的创建和事件处理过程

在 NewBluePill 对 SVM 技术实现下， $\text{HvmEventCallback}()$ 函数每当调用 $\text{Hvm} \rightarrow \text{ArchIsNestedEvent}()$ 函数时，实际调用的 $\text{SvmIsNestedEvent}()$ 函数会判断应该在内层还是在外层 NewBluePill Hypervisor 中处理，所依据方法就是上文中提到的内外层 NewBluePill Hypervisor 的区别。对于前者， $\text{HvmEventCallback}()$ 函数会进一步调用 $\text{SvmDispatchEvent}()$ 函数，而对于后者， $\text{HvmEventCallback}()$ 函数会调用

批注 [540]: 该图摘自 IsGameOver(), 将来会换图

SvmDispatchNestedEvent() 函数处理。

下面我们依次分析 SvmDispatchEvent() 函数和 SvmDispatchNestedEvent() 函数。首先探索 SvmDispatchEvent() 函数，该函数调用 SvmHandleInterception() 函数 (Svm.c 第 862 行到 864 行)，并会通知它不要让虚拟机再继续处理这个事件，之后，由于 NewBluePill 在 SVM 的实现中会拦截 SMI 等中断，因此函数在 Svm.c 第 868 行到 874 行特别加入处理，在虚拟机虚拟 GIF 开启的情况下，如果拦截到了中断则利用事件注入机制将被拦截中断重新注入回虚拟机中。最后，SvmDispatchEvent() 函数设置虚拟机的虚拟 EFER.SVME (Svm.c 第 878 行到 890 行)。

SvmHandleInterception() 函数作用与 VmxHandleInterception() 函数类似，其主要工作也是获取陷入原因 (Exit Code)、指导查找 Trap 元素 (Svm.c 第 570 行到 602 行，第 617 行到 637 行)、执行找到的 Trap 元素 (Svm.c 第 603 行到 606 行，第 639 行到 642 行)，宾在打开 BLUE_CHICKEN 反 HVM 探测的情况下判断当前是否要暂时隐藏自己 (Svm.c 第 607 行到 615 行)¹。但是与 VmxHandleInterception() 函数进行比较，明显 SvmHandleInterception() 函数要具体很多，这是因为 NewBluePill 针对 SVM 的实现中，细化了对 MSR 寄存器访问的拦截处理，从 Svm.c 第 573 行到 617 行这段代码可以看出，它通过 VMCB 结构体的 ExitInfo 域分析出虚拟机具体对 MSR 寄存器的访问操作类型 (读/写) 并与注册在 Trap 元素中的信息进行比较 (Svm.c 第 593 行到 601 行)，同时，在执行该 Trap 元素时，使用的也是 TrExecuteMsrTrapHandler() 函数而不是通用的 TrExecuteGeneralTrapHandler() 函数——虽然在 TrExecuteMsrTrapHandler() 函数中只是简单的打印一些调试信息同时调整虚拟机指令指针向后偏移 2 个字节。

```
00571:
00572: switch (Vmc->exitcode) {
00573: case VMEXIT_MSR:
00574:
00575:     TrappedMsr = (ULONG32) (GuestRegs->rcx & 0xffffffff);
00576:
00577:     if (Vmc->exitinfo1 == (MSR_INTERCEPT_READ >> 1))
00578:         bCurrentMsrAccess = MSR_INTERCEPT_READ;
00579:     else
00580:         bCurrentMsrAccess = MSR_INTERCEPT_WRITE;
00581:
00582:     // note that HvmFindRegisteredTrap() will return a Trap pointer without checks for intercepted MSR access;
00583:     // i.e. it will return a Trap for R access when current event is WRMSR.
00584:
00585:     if (!NT_SUCCESS (Status)) {
00586:         _KdPrint (("SvmHandleInterception(): Failed to find a trap handler for MSR 0x%08hX, status 0x%08hX!\n",
00587:             TrappedMsr, Status));
00588:         SvmShutdown (Cpu, GuestRegs, FALSE); // This is a fatal error -- uninstall...
00589:         break;
00590:     }
00591:     // we found a trap handler, check the trapped access
00592:
00593:     if (! (Trap->Msr.TrappedMsrAccess & bCurrentMsrAccess)) {
00594:         // we don't intercept this MSR access, fix the msrpm and retry the intercepted instruction
00595:
00596:         _KdPrint (("SvmHandleInterception(): Current MSR (0x%08hX) is access is not trapped!\n", TrappedMsr,
00597:             bCurrentMsrAccess == MSR_INTERCEPT_READ ? "read" : "write"));
00598:
00599:         SvmShutdown (Cpu, GuestRegs, FALSE); // This is a fatal error -- uninstall...
00600:         break;
00601:     }
00602:     // we have a valid trap handler for this interception, call it
00603:     if (!NT_SUCCESS (Status = TrExecuteMsrTrapHandler (Cpu, GuestRegs, Trap, WillBeAlsoHandledByGuestHv))) {
00604:         _KdPrint (("SvmHandleInterception(): HvmExecuteMsrTrapHandler() failed with status 0x%08hX\n", Status));
00605:         Vmc->rip += 2;
00606:     }
00607: #ifdef BLUE_CHICKEN
00608:     ChickenAddInterceptTsc (Cpu);
00609:     if (ChickenShouldUninstall (Cpu)) {
00610:         _KdPrint (("SvmHandleInterception(): CPU%d: Chicken Says to uninstall\n", Cpu->ProcessorNumber));
00611:
00612:         // call HvmSetupTimeBomb()
00613:         Hvm->ArchShutdown (Cpu, GuestRegs, TRUE);
00614:     }
00615: #endif
00616:     break;
00617:
00618: case VMEXIT_IOIO:
00619:
```

图 6.6 SvmHandleInterception() 函数对 MSR 寄存器访问的处理

而 SvmDispatchNestedEvent() 函数主要作用是按照表 6.1 所示分发当前 #VMEXIT

¹ 由 NewBluePill 针对 VT 技术实现，此处的反 HVM 探测应该放在 switch 结构之外，这里的写法应该是一个 Bug

事件到当前层或内层 NewBluePill Hypervisor 处理。

表 6.1 SvmDispatchNestedEvent() 函数事件分发依据

相应 Trap 元素可在当前层 NewBluePill Hypervisor 找到	Trap 元素类型	bForwardTrapToGuest 标记	处理方法
否	N/A	True	分 发 到 内 层 NewBluePill Hypervisor (当前层虚拟机)
是	TRAP_GENERAL	True	分 发 到 内 层 NewBluePill Hypervisor (当前层虚拟机)
是	TRAP_MSR	True	分 发 到 内 层 NewBluePill Hypervisor (当前层虚拟机)
是	其它情况		分 发 到 当 前 层 NewBluePill Hypervisor

由于此时是要在当前层 NewBluePill Hypervisor 中处理嵌套虚拟机中#VMEXIT 事件, 因此函数首先对此进行确定 (Svm.c 第 703 行到 708 行), 之后调用 TrFindRegisteredTrap()函数在当前 Hypervisor 中查找是否注册处理相应事件, 通过填充 bInterceptedByGuest 和 bInterceptedByUs 变量来构造表 6.1 所示分发依据 (Svm.c 第 711 行到 736 行)。随后 SvmDispatchNestedEvent() 函数判断 bInterceptedByGuest 变量取值以决定是否让内层 NewBluePill Hypervisor 处理该事件, 如果确定此情况, 那么内层 NewBluePill Hypervisor 会通过更新 Cpu->Svm.VmcbToContinuePA 域 (Svm.c 第 754 行) 来开启当前层虚拟机处理该事件。SvmDispatchNestedEvent() 函数最后, 在不将事件分发到内层 NewBluePill Hypervisor 处理的情况下, 通过调用 SvmHandleInterception() 函数 (Svm.c 第 835 行) 实现事件在当前层 Hypervisor 的处理。

各处理函数功能和实现

熟悉了 NewBluePill 事件处理机制后, 我们看看在 NewBluePill 中定义了哪些 VMEXIT 原因的处理函数及其具体执行流程, 同样的, 我们将会分开描述 VT 技术和 SVM 技术各处理函数的实现。

VT 技术实现中各处理函数功能和流程

前面我们曾经在表 4.2 和表 4.3 中列举过 VT 技术下陷入原因和处理函数的对应关系, 接下来我们将详细阐述每个处理函数的流程。

VmxDispatchVmxInstrDummy() 函数

VmxDispatchVmxInstrDummy() 函数用于处理因 VMCALL、VMCLEAR、VMLAUNCH、VMRESUME、VMPTRLD、VMPTRST、VMREAD、VMWRITE、VMXON、VMXOFF 指令造成的陷入。这个函数不做任何事情, 只是负责更新 Trap->General.RipDelta 域以便虚拟机执行后续指

令 (Vmxtrap.c 第 37 行)。

要注意的是, 它还同时更新虚拟机的状态寄存器 (RFLAGS) (Vmxtrap.c 第 39 行), 该行作用是提供给虚拟机这些虚拟化指令执行失败 (VMFAIL Invalid) 的假象。

伪造虚拟化指令 (VMX Operations) 执行结果

可以通过伪装函数 (pseudo-functions) 来欺骗虚拟机虚拟化指令执行的结果。这些伪装函数包括 VMsucceed, VMfail, VMfailInvalid 和 VMfailValid。它们不是真正的函数, 而是通过修改该状态寄存器 (RFLAGS) 的值来伪装虚拟化指令的执行结果。比如:

VMsucceed 的功能: CF<-0, PF<-0, AF<-0, ZF<-0, SF<-0, OF<-0

VMfail 的功能: 如果 VMCS 指针有效, 则是 VMfailValid, 否则是 VMfailInvalid

VMfailValid 的功能: CF<-0, PF<-0, AF<-0, ZF<-1, SF<-0, OF<-0, VM-instruction error field 域设置为 ErrorNumber。

VMfailInvalid 的功能: CF<-1, PF<-0, AF<-0, ZF<-0, SF<-0, OF<-0

更多信息请参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B, Chapter 5.2 Conventions*。

VmxDispatchCpuid() 函数

VmxDispatchCpuid() 函数用于处理因 CPUID 指令造成的陷入。函数流程如图 6.7 所示:

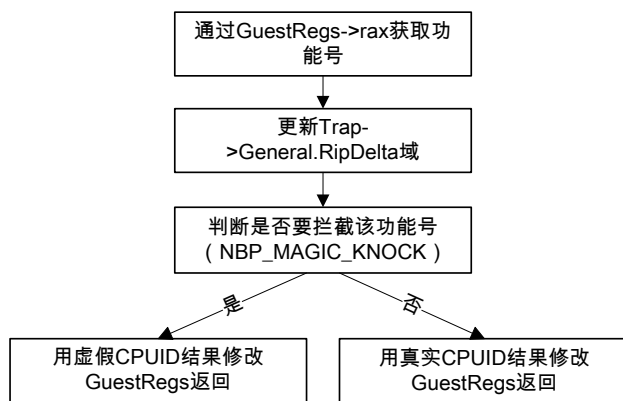


图 6.7 VmxDispatchCpuid() 函数流程图

首先获取虚拟机陷入时其 RAX 寄存器的值 (GuestRegs->rax), 该值将视作 CPUID 指令的功能号 (Vmxtrap.c 第 55 行), 然后更新 Trap->General.RipDelta 域以便虚拟机执行后续指令 (Vmxtrap.c 第 61 行), 随后判断如果该功能号是 NewBluePill 想要篡改的功能号, 则篡改 GuestRegs 相应部分, 从而在恢复虚拟机运行后返回给中断程序伪造的数据。

VmxDispatchINVD() 函数

VmxDispatchINVD() 函数用于处理因 INVD 指令造成的陷入。由于 INVD 指令只是负责让处理器缓存失效, 因此这个函数与 VmxDispatchVmxInstrDummy() 函数一样不做什么事情, 只是负责更新 Trap->General.RipDelta 域以便虚拟机执行后续指令 (Vmxtrap.c 第 499 行)。需要注意的是, 由于 INVD 指令并不是虚拟化操作指令 (VMX Instructions), 因此在 VmxDispatchINVD() 函数的最后直接返回, 而没有像 VmxDispatchVmxInstrDummy() 函数那样还有一步伪造虚拟化指令执行结果的步骤。

VmxDispatchMsrRead() 函数

VmxDispatchMsrRead() 函数用于处理因 RDMSR 指令造成的陷入。虚拟机中的 RDMSR 指令之所以能够陷入是因为在前面对 VMCS 结构体的设置中, 默认情况下 NewBluePill 将 Primary Processor-Based VM Execution Controls.Use MSR bitmaps[bit 28] 设置为 0, 根据 Intel 文档的规定, 这样设值就会造成虚拟机在执行 RDMSR 指令时发生 #VMEXIT 事件陷入到虚拟机中。VmxDispatchMsrRead() 函数的功能是: 对于 Hypervisor 和虚拟机两者同一 MSR 寄存器号, 其内容却不相同的, 返回虚拟机 VMCS 结构体中对应 MSR 寄存器内容; 否则返回 Hypervisor 的 MSR 寄存器内容 (其实理论上也可以返回虚拟机 VMCS 结构体中对应 MSR 寄存器内容, 但是考虑到两者被改变的 MSR 寄存器极少, 而且读取虚拟机 VMCS 结构体中 MSR 寄存器内容的参数和普通读取 MSR 寄存器内容的参数是要做映射的, 工作量会很大)。

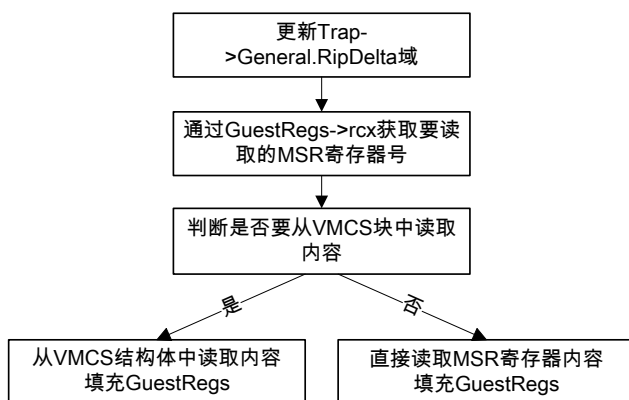


图 6.8 VmxDispatchMsrRead() 函数流程图

VmxDispatchMsrRead() 函数流程如图 6.8 所示。首先更新 Trap->General.RipDelta 域, 然后获得虚拟机要读取的 MSR 寄存器号 (Vmxtraps.c 第 106 行), 随后根据该寄存器号选择是读取虚拟机 VMCS 结构体中对应 MSR 寄存器内容 (Vmxtraps.c 第 109 行到 124 行) 还是读取 Hypervisor 的 MSR 寄存器内容 (Vmxtraps.c 第 129 行到 130 行)。最后通过修改 GuestRegs 的 RAX 和 RDX 寄存器, 将返回结果传回虚拟机。

这里存在一个问题, 对于 GUEST_SYSENTER_CS、GUEST_SYSENTER_ESP、GUEST_SYSENTER_EIP 这三个 MSR 寄存器, 实际上通过阅读前面章节中对启动过程的描述我们可以发现, 他们的内容在 Hypervisor 和虚拟机中是一样的, 所以此处 NewBluePill 作者选

择仍从虚拟机 VMCS 结构体中读取，可能是为了代码运行时更保险一些。

VmxDispatchMsrWrite() 函数

VmxDispatchMsrWrite() 函数用于处理因 WRMSR 指令造成的陷入。与 RDMSR 指令一样，虚拟机中的 WRMSR 指令之所以能够陷入同样是因为前面对 VMCS 结构体的设置中 Primary Processor-Based VM Execution Controls.Use MSR bitmaps[bit 28]被设置为 0。

类似的，VmxDispatchMsrWrite() 函数首先更新 Trap->General.RipDelta 域，然后获得虚拟机要写入的 MSR 寄存器号 (Vmxtraps.c 第 157 行) 和要写入的值 (Vmxtraps.c 第 159 行到 160 行)，随后同样的，根据该寄存器号选择是写入虚拟机 VMCS 结构体中对应 MSR 寄存器域 (Vmxtraps.c 第 163 行到 177 行) 还是直接写入 MSR 寄存器 (Vmxtraps.c 第 183 行到 184 行)。

VmxDispatchCrAccess() 函数

VmxDispatchCrAccess() 函数用于处理因操作 CR0,CR3,CR4 控制寄存器造成的陷入，其主要流程如图 6.9 所示。函数首先更新 Trap->General.RipDelta 域 (Vmxtraps.c 第 221 行)，随后函数读取 VMCS 控制块中 VMEXIT 相关信息域 (VM Exit Information Fields) 中的退出条件域 (Exit Qualification)，并从中分离出被操作的控制寄存器号 cr (Vmxtraps.c 第 226 行) 和通用寄存器号 gp (Vmxtraps.c 第 225 行)。

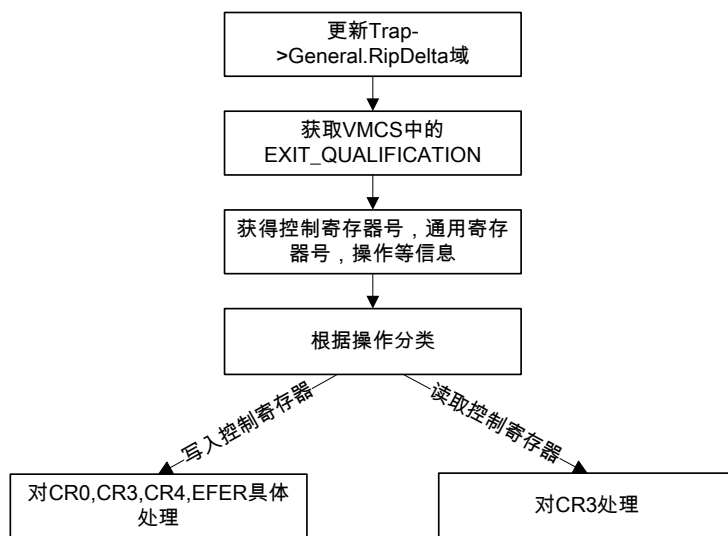


图 6.9 VmxDispatchCrAccess() 函数流程图

接下来 VmxDispatchCrAccess() 函数根据虚拟机对控制寄存器操作类型进行不同的处理过程。首先看看在写入控制寄存器操作中，VmxDispatchCrAccess() 函数是如何处理的。

在虚拟机准备写入 CR0 控制寄存器的情况下，函数首先复制要写入的内容到 Cpu->Vmx.GuestCR0 中 (Vmxtraps.c 第 235 行)，有一点要注意的是，该行代码使用 GuestRegs

结构体的方式决定了其中的元素位置是不能变化的。接下来函数判断 CR0 新值是否启用了分页机制 (Vmxtraps.c 第 236 行), 如果启动了分页机制, 函数会把 Cpu->Vmx.GuestCR3 域写入 VMCS 结构体中 Guest_CR3¹, 同时根据 EFER 寄存器的 EFER_LME (是否支持 x64) 来决定修改 EFER_LMA (是否开启 x64); 如果虚拟机禁用了分页机制, 那么函数会备份 VMCS 结构体中 Guest_CR3 内容, 并且修改 EFER_LMA, 不开启 x64 模式。最后修改 VMCS 结构体中的 CR0_READ_SHADOW 域, 同时调用 VmxUpdateGuestEfer() 决定是否要为虚拟机开启 x64 模式。²

在虚拟机准备写入 CR0 控制寄存器的情况下, 函数首先复制要写入的内容到 Cpu->Vmx.GuestCR3 中 (Vmxtraps.c 第 267 行), 然后将新值写入到 VMCS 结构体的 Guest_CR3 返回。

对于 CR4 控制寄存器过程基本相同, 同样包括复制新值和更新 VMCS 结构体的 Guest_CR4 的操作。

而对于虚拟机将控制寄存器写入通用寄存器的操作, VmxDispatchCrAccess() 函数仅针对 CR3 做了处理。过程非常简单, 只是将 Cpu->Vmx.GuestCR3 域的内容复制到 GuestRegs 结构体相应变量 (Vmxtraps.c 第 306 行到 314 行)。

VmxDispatchException() 函数

VmxDispatchException() 函数用于处理因虚拟机发生异常而造成的陷入。这个函数是一个未完成的函数。首先读取出虚拟机的中断请求号 (Vmxtraps.c 第 454 行), 如果是调试异常³则直接返回, 然后为了保证计时的准确性对 VMCS 结构体中的 GUEST_INTERRUPTIBILITY_INFO 域赋值, 使得不允许虚拟机中任何事件能被阻塞一段时间。随后该函数记录累计的执行周期数 (Vmxtraps.c 第 473 行), 并在追踪一定数量的虚拟机指令后清除虚拟机的 RFLAGS 的 TF 位, 停止单步执行。最后的记录过程可与 SVM 中相应实现相同, 但是 VT 和 SVM 一样这部分的实现并不完整。

VmxDispatchRdtsc() 函数

VmxDispatchRdtsc() 函数用于处理因 RDTSC 指令而造成的陷入。这个函数也是 NewBluePill 用于实现时间欺骗的关键函数, 关于时间欺骗的具体原理我们会在“第七章 NewBluePill 反探测系统”叙述。该函数在更新 Trap->General.RipDelta 域 (Vmxtraps.c 第 412 行) 之后, 根据时间欺骗策略填充作为结果的 Cpu->Tsc 域 (Vmxtraps.c 第 418 行到 422 行), 然后恢复各个中间变量 (Vmxtraps.c 第 429 行到 432 行) 并通过修改虚拟机的 RFLAGS 的 TF 位开启单步中断 (Vmxtraps.c 第 436 行), 从而可以在虚拟机恢复运行后, 在停止追踪指令前, 执行每条指令都会陷入到 Hypervisor 中并由 VmxDispatchException() 函数处理, 最后修改 GuestRegs 结构体返回结果。

批注 [S41]: “第七章 NewBluePill 反探测系统”

¹ 这是因为在安装 NewBluePill 后, 虚拟机所有对 CR3 的操作结果都会被 NewBluePill 记录在 Cpu->Vmx.GuestCR3 中, 所以该域保存着 CR3 的最新值。

² CR0 和 CR4 的处理过程最后都返回 False, 说明 NewBluePill 并不能正确处理写入这两个寄存器造成的陷入, 这一点要特别注意。

³ 有关 Windows 下 x86/x64 异常和中断号对应关系, 可以参考 *Windows Internals, 4th Edition Chapter 3*

VmxDispatchIoAccess() 函数

VmxDispatchIoAccess() 函数用于处理因 I/O 操作而造成的陷入。该函数主要功能是获取键盘和鼠标 I/O 操作的信息，如数据传送方向 In/Out、数据大小、数据值等。函数主要流程如图 6.10 所示：

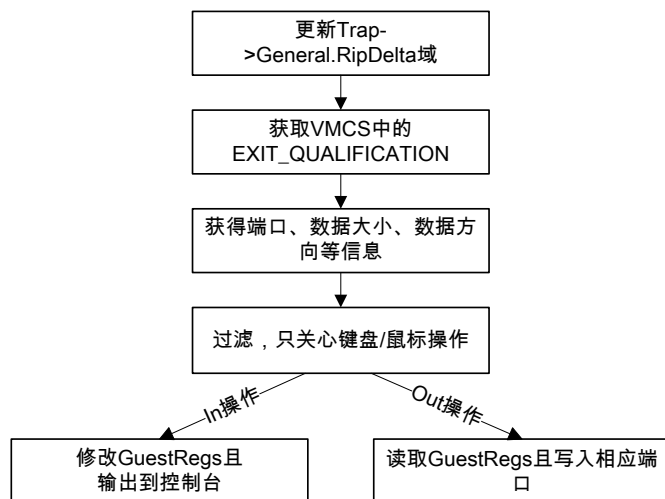


图 6.10 VmxDispatchIoAccess() 函数流程图

首先同样是更新 Trap->General.RipDelta 域 (Vmxtraps.c 第 356 行)，随后函数读取 VMCS 控制块中 VMEXIT 相关信息域 (VM Exit Information Fields) 中的退出条件域 (Exit Qualification)，并由此获得端口、数据大小、数据方向等信息 (Vmxtraps.c 第 362 行到 368 行)，同时通过调用 init_scancode() 函数来初始化键盘扫描码。之后，对于 In I/O 指令操作，该函数会修改 GuestRegs->rax 域来传入虚拟机中，同时，如果操作对象是键盘，那么该函数还会打印出所按的键。而对于 Out I/O 指令，该函数负责读取 GuestRegs->rax 域，并将内容输出到指定设备。

Note 关于退出条件域 (Exit Qualification) 对 I/O 操作记录信息的说明，请阅读 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B, Chapter 23.5* 相关部分

SVM 技术实现中各处理函数功能和流程

前面我们曾经在表 4.5 中列举过 SVM 技术下陷入原因和处理函数的对应关系，不同于 NewBluePill 在 VT 技术下的实现，在 SVM 中，大多数事件处理函数内部不需要显式的对 RipDelta 域更新，这是因为在注册处理函数时 RipDelta 域已经提前在 Trap 元素中指定（可参考“第四章 NewBluePill 的启动和卸载”中 SVM 相关部分）。接下来我们将详细阐述每个处理函数的流程。

批注 [S42]: “第四章 NewBluePill 的启动和卸载”

SvmDispatchVmrn() 函数

SvmDispatchVmrn() 函数用于处理在虚拟机中（尤其是在执行嵌套 NewBluePill Hypervisor 时）中执行 VMRUN 指令造成的陷入，其主要流程如图 6.11 所示。

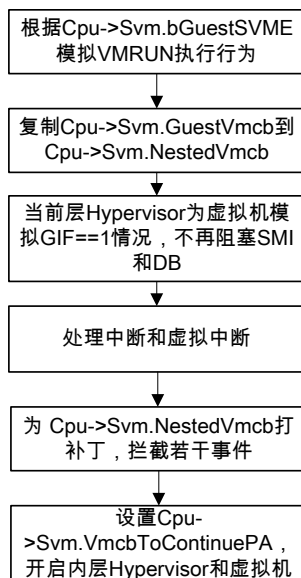


图 6.11 SvmDispatchVmrn() 函数流程图

函数首先根据 Cpu->Svm.bGuestSVME 判断当前虚拟机下 EFER.SVME 位是否已设置 (Svmtraps.c 第 239 行到 243 行)，由“第二章 深入 HEV 技术细节”可以得知，设置 EFER.SVME 位是开启虚拟机模式的前提，当前层 NewBluePill Hypervisor 意在模拟 EFER MSR 寄存器，对该特性的模拟自然也是其中重要的一部分，如果此时该位为 0，则说明之前虚拟机尚未设置该位，因此向虚拟机中插入一个 Exception 并返回 (Svmtraps.c 第 241 行)。随后，在启用 SVM_USE_NESTEDVMCB_REWRITING 的情况下（覆写 Svm.NestedVmcb），函数将要求挂载的 VMCB 结构体物理地址赋值到 Cpu->Svm.GuestVmcbPA 域 (Svmtraps.c 第 246 行)，并调用 HvmCopyPhysicalToVirtual() 函数复制该 VMCB 结构体到 Svm.NestedVmcb 中 (Svmtraps.c 第 250 行到 258 行)。

HvmCopyPhysicalToVirtual() 函数用于在启用内存隐藏机制下将一个物理地址所指向的内存区域复制到指定的虚拟地址上，主要是通过调用 CmPatchPTEPhysicalAddress() 函数恢复/重新修改原有操作系统页表上虚拟地址到物理地址的映射实现，关于内存隐藏机制，可参考“第五章 NewBluePill 内存系统”的相关部分。

回到 SvmDispatchVmrn() 函数，在 Svmtraps.c 第 269 行调用 SvmEmulateGif1ForNestedGuest() 函数，模拟 GIF==1 让虚拟机自己处理 SMI 和 DB 中断，之后，根据启用虚拟中断与否和此时虚拟机中 RFLAGS 的 Interrupt Flag 位进行相应处理 (Svmtraps.c 第 274 行到 282 行)。

然后在 Svmtraps.c 第 286 行，SvmDispatchVmrn() 函数调用 SvmSetupGeneralInterceptions() 来配置 Svm.NestedVmcb 所指向的 VMCB 结构体，从而使得待挂载 VMCB 结构体与当前层 NewBluePill Hypervisor 具有一样的拦截行为，为进一

批注 [543]: “第二章 深入 HEV 技术细节” 章号

步加载嵌套 NewBluePill 做准备。但是细心的读者可能会发现，NewBluePill Hypervisor 的一部分拦截行为是在 MSRPM 中被定义的，那么这部分又是怎样被注入待挂载 Hypervisor 中的呢？

答案就是图 6.12 中的代码，这部分代码负责将当前层拦截 MSR 访问的策略写入待加载 Hypervisor 的 MSRPM 中，尽管在 NewBluePill 公开代码中没有启用这部分并且有些地方没有实现，但是从大致逻辑上我们可以看到，在尽量使用待加载 MSRPM 基础上通过调用 SvmSetupMsrInterceptions() 函数，将策略写入 Hypervisor 中。

```
00291:
00292: #if 0 //FIXME: right now we don't need this, but a full VMM most likely would need that (the code below should work fine)
00293: // copy & patch the guest MSRPM
00294: # if DEBUG_LEVEL>2
00295: _KdPrint (("SvmDispatchVmrund(): Guest VMCB: TLB CONTROL = 0x%x\n", Cpu->Svm.NestedVmcb->tlb_control));
00296: _KdPrint (("SvmDispatchVmrund(): Guest VMCB: V_INTR = 0x%x\n", Cpu->Svm.NestedVmcb->vintr.UCHARs));
00297: _KdPrint (("SvmDispatchVmrund(): Guest VMCB: V_INTR_MASKING = 0x%x\n",
00298: Cpu->Svm.NestedVmcb->vintr.fields.intr_masking));
00299: _KdPrint (("SvmDispatchVmrund(): Guest VMCB: V_TPR = 0x%x\n", Cpu->Svm.NestedVmcb->vintr.fields.tpr));
00300: _KdPrint (("SvmDispatchVmrund(): Guest VMCB: rflags = 0x%x\n", Cpu->Svm.NestedVmcb->rflags));
00301: If (Cpu->Svm.NestedVmcb->eventinj.fields.v)
00302: _KdPrint (("SvmDispatchVmrund(): Guest VMCB: EVENTINJ: vec = 0x%x, type = 0x%x, ev = %x, v = %x, errorcode = 0x%x\n",
00303: Cpu->Svm.NestedVmcb->eventinj.fields.vector, Cpu->Svm.NestedVmcb->eventinj.fields.type,
00304: Cpu->Svm.NestedVmcb->eventinj.fields.ev, Cpu->Svm.NestedVmcb->eventinj.fields.v,
00305: Cpu->Svm.NestedVmcb->eventinj.fields.errorcode));
00306: # endif
00307:
00308: GuestMsrPmPa.QuadPart = Cpu->Svm.NestedVmcb->msrpm_base_pa;
00309:
00310: # if DEBUG_LEVEL>2
00311: _KdPrint (("SvmDispatchVmrund(): Guest MSRPM PA: 0x%x\n", GuestMsrPmPa.QuadPart));
00312: # endif
00313:
00314: if (GuestMsrPmPa.QuadPart) {
00315: // guest has specified a MsrPm, patch it to intercept all what we need
00316:
00317: if (!NT_SUCCESS
00318: (Status = HvmCopyPhysicalToVirtual (Cpu, Cpu->Svm.NestedMsrPm, GuestMsrPmPa, SVM_MSRPM_SIZE_IN_PAGES))) {
00319: _KdPrint (("SvmDispatchVmrund(): Failed to read guest MSRPM, status 0x%08h\n", Status));
00320:
00321: // continue the nested VM
00322: Cpu->Svm.VmcbToContinuePA.QuadPart = Vmcb->rax;
00323: return TRUE;
00324: }
00325:
00326: } else {
00327: // guest hypervisor doesn't want to intercept any MSR rv, but we have to.
00328: // Indicate we want to trap nothing else but EFER and VM_HSAVE_PA rw.
00329: _KdPrint (("SvmDispatchVmrund(): Uppss! Guest h/v doesn't intercept EFER access!\n"));
00330: RtlZeroMemory (Cpu->Svm.NestedMsrPm, SVM_MSRPM_SIZE_IN_PAGES * PAGE_SIZE);
00331: TODO: set EFER bits!!!
00332: }
00333:
00334: // apply all our traps to the guest msrpm
00335: if (!NT_SUCCESS (Status = SvmSetupMsrInterceptions (Cpu, Cpu->Svm.NestedMsrPm))) {
00336: _KdPrint (("SvmDispatchVmrund(): *** VmcbSetupMsrInterceptions() failed with status 0x%08hX, continuing ***\n",
00337: Status));
00338: }
00339:
00340: Cpu->Svm.NestedVmcb->msrpm_base_pa = Cpu->Svm.NestedMsrPmPa.QuadPart;
00341:
00342: #endif
00343:
```

图 6.12 修改待挂载 Hypervisor 的 MSRPM

然后，SvmDispatchVmrund() 函数决定要采用的缓存策略，通过与 Svm.c 文件中的 SvmSetupControlArea() 函数最后相似的方法进行判断并将决策写入 Cpu->Svm.NestedVmcb->tlb_control 域中 (Svmtraps.c 第 350 行到 365 行)。最后，函数改写 Cpu->Svm.VmcbToContinuePA (Svmtraps.c 第 368 行或 371 行)，无论开启 SVM_USE_NESTEDVMCB_REWRITING 与否，该域指向下一个要启用的 VMCB 结构体物理地址。

SvmDispatchVmload() 函数

SvmDispatchVmload() 函数用于处理在虚拟机中（尤其是在构建嵌套 NewBluePill Hypervisor 时）中执行 VMLOAD 指令造成的陷入。该函数根据 VMLOAD 指令语义，获取虚拟机中陷入前的 RAX 值 (Svmtraps.c 第 37 行到 38 行) 并将其视作要加载的 VMCB 结构体物理地址，在 Hypervisor 内部执行 VMLOAD 指令以托管的方式恢复此结构体部分信息到寄存器 (Svmtraps.c 第 46 行)，最后成功返回。

SvmDispatchVmsave() 函数

SvmDispatchVmsave() 函数用于处理在虚拟机中执行 VMSAVE 指令造成的陷入。这个函数的具体流程与 SvmDispatchVmload() 函数相同，也是获取待加载的 VMCB 结构体物理地址，并在 Hypervisor 内部执行 VMSAVE 指令从而以托管的方式保存寄存器部分信息到该结构体 (Svmtraps.c 第 79 行)。

SvmDispatchClgi() 函数

由于 NewBluePill 在 SVM 下的实现需要为虚拟机虚拟 GIF (其目的在于将虚拟机中在 GIF==0 情况下本应该被忽略的中断和事件¹拦截下并放到某级 NewBluePill Hypervisor 中处理)，因此拦截了 CLGI 和 STGI 指令的执行，前者就是通过 SvmDispatchClgi() 函数处理的。其中，最关键的地方在于调用了 SvmEmulateGif0ForGuest() 函数 (Svmtraps.c 第 108 行) 在 NewBluePill Hypervisor 中启用了对 SMI 和 DB 的拦截 (Svm.c 第 79 行到 82 行)。之后，函数成功返回。

SvmDispatchStgi() 函数

如上文所述，该函数用于处理虚拟机中因执行 STGI 指令而造成的陷入。与 SvmDispatchClgi() 函数类似，该函数最重要的地方是对 SvmEmulateGif1ForGuest() 函数的调用，后者通过在 NewBluePill Hypervisor 中屏蔽对 SMI 和 DB 的拦截而起到了对虚拟机 GIF==1 环境的模拟。

SvmDispatchSMI() 函数

SvmDispatchSMI() 函数用于收集虚拟机进入 GIF==0 情况后发生的 SMI 中断，但是该函数在公开的 NewBluePill 代码中并未实现完全。

SvmDispatchDB() 函数

SvmDispatchDB() 函数用于收集虚拟机进入 GIF==0 情况后发生的 DB 异常，但是该函数在公开的 NewBluePill 代码中并未实现完全。

SvmDispatchCpuid() 函数

与 VT 技术下 VmxDispatchCpuid() 函数功能相同，SvmDispatchCpuid() 函数用于 NewBluePill 在 SVM 平台上处理因 CPUID 指令造成的陷入。由于不需要处理 RipDelta 域以

¹ 包括 SMI 和 DB (Debug Exception)，更详细的信息可参考 AMD64 Architecture Programmer's Manual Volume 2: System Programming 手册 Chapter 15.16. Global Interrupt Flag, STGI and CLGI Instructions 一章

及加入了对卸载请求的相应逻辑, 因此 `SvmDispatchCpuid()` 函数流程如图 6.13 所示:

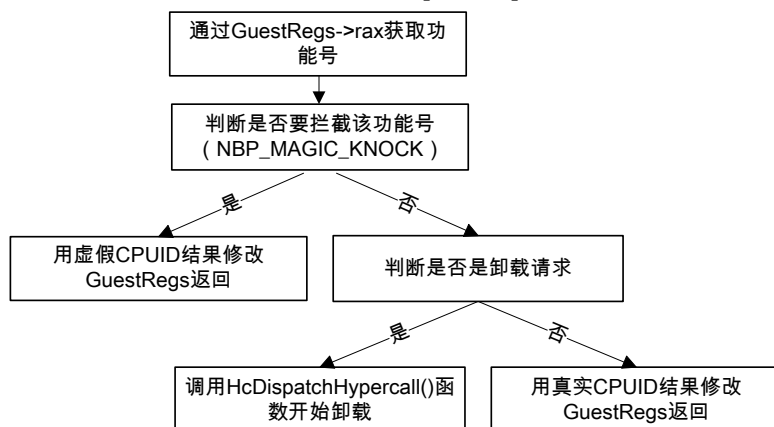


图 6.13 `SvmDispatchCpuid()` 函数流程图

由于该函数实现细节与 `VmxDispatchCpuid()` 函数大体相同, 因此不再赘述。

`SvmDispatchRdtsc()` 函数

`SvmDispatchRdtsc()` 函数十分类似于 VT 技术下 `VmxDispatchRdtsc()` 函数, 因此关于该函数流程可参考对 `VmxDispatchRdtsc()` 函数的描述, 有关时间欺骗具体原理可参考“第七章 NewBluePill 反探测系统”的相关叙述。

批注 [S44]: “第七章 NewBluePill 反探测系统”

`SvmDispatchRdtscp()` 函数

`SvmDispatchRdtscp()` 函数用于处理在虚拟机中执行 `RDTSCP`¹ 指令造成的陷入, 在 NewBluePill 的公开版本中, 该函数实现代码与 `SvmDispatchRdtsc()` 函数相同, 只是缺少了最后对 `ECX` 寄存器结果的处理。

`SvmDispatchEFERAccess()` 函数

`SvmDispatchEFERAccess()` 函数用于处理虚拟机中因访问 `EFER MSR` 寄存器而造成的陷入, 由于 `EFER.SVME` 位是虚拟机模式运行和存在的关键, 而出于反探测和嵌套 NewBluePill Hypervisor 的目的, 虚拟机中操作系统“看见”的 `EFER.SVME` 位可能与实际 `EFER.SVME` 相同, 也有可能不同, 这就是 NewBluePill 为何虚拟 `EFER MSR` 寄存器的原因, 其函数流程如图 6.14 所示:

函数首先判断虚拟机对 `EFER` 寄存器的访问形式 (读/写) (`Svmtraps.c` 第 414 行) 并据此在后面采取不同的操作。如果是读取 `EFER` 寄存器, `SvmDispatchEFERAccess()` 函数会根据 `Cpu->Svm.bGuestSVME` 判断虚拟机中当前是否启用了 `EFER.SVME` 位 (`Svmtraps.c` 第

¹ `RDTSCP` 指令 (Read Time Stamp Counter and Processor ID), 该函数在 `RDTSC` 基础上还可以返回储存在 `TSC_AUX MSR` 寄存器的内容, 详细描述可参考 *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions* 手册 System Instruction Reference 一章

424 行), 如果没有, 则屏蔽该位, 返回存储在 VMCB 结构体中的虚拟 EFER 寄存器其余内容。

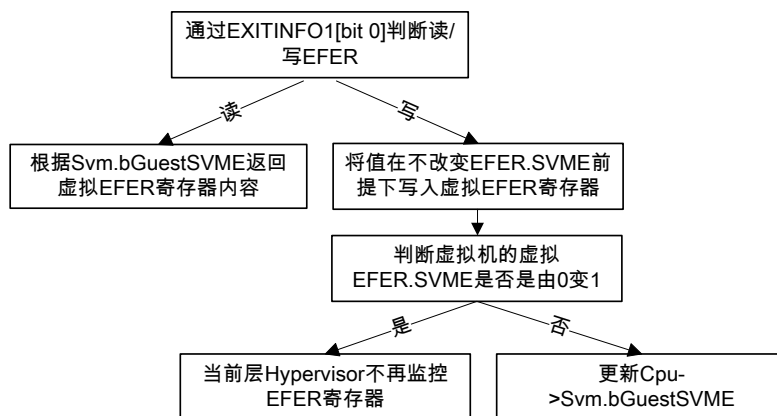


图 6.14 SvmDispatchEFERAccess() 函数流程图

如果是写入 EFER 寄存器, SvmDispatchEFERAccess() 函数首先在 blsSVMEOn 中存储虚拟机中待写入的值是否启用了 EFER.SVME 位, 由于在返回到虚拟机中继续执行时, Vmcb->efer 中的内容会被换入到 EFER MSR 寄存器中, 因此必须确保 Vmcb->efer.SVME=1 (Svmtraps.c 第 446 行), 除此以外的其余各位被写入内容代替。在开启 SVM_AUTOREMOVE_EFER_TRAP 开关的情况下, SvmDispatchEFERAccess() 函数会在虚拟机的虚拟 EFER 寄存器 SVME 由 0 变 1 时, 从当前层 Hypervisor 中移除对 EFER 的访问监控, 最后成功返回。

SvmDispatchVM_HSAVE_PAAccess() 函数

SvmDispatchVM_HSAVE_PAAccess() 函数用于处理虚拟机中因访问 VM_HSAVE_PA MSR 寄存器而造成的陷入, 虽然破坏该 MSR 寄存器的内容一般不会 (但是可以) 导致虚拟机模式关闭, 但该 MSR 寄存器由于存储有 NewBluePill 要监控的 MSR 寄存器访问, 因此也是 NewBluePill Hypervisor 运行的关键。图 6.15 展示了该函数流程:

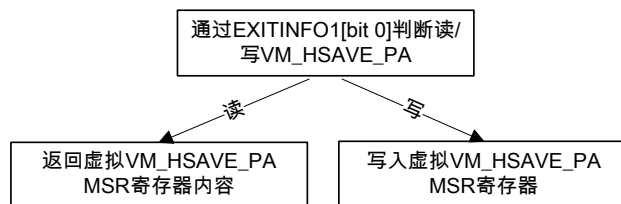


图 6.15 SvmDispatchVM_HSAVE_PAAccess() 函数流程图

无论读写, 均是面对作为虚拟 VM_HSAVE_PA MSR 寄存器的 Cpu->Svm.GuestHsaPA 域。由于在 Svm 结构体初始化后 GuestHsaPA 域一直为 0 且从未改变, 因此确保在使用时完全符合对实际 VM_HSAVE_PA MSR 寄存器的访问结果, 达到了模拟的目的。

SvmDispatchMsrTscRead() 函数

SvmDispatchMsrTscRead() 函数用于处理虚拟机中因访问时间戳寄存器（TSC）而造成的陷入，该函数实现代码与 SvmDispatchRdtsc() 函数相同，故不再赘述。

七、 NewBluePill 反探测系统

有了内存隐藏技术和针对多数陷入情况而实现的陷入处理，NewBluePill Hypervisor 看上去已经实现的很透明了，然而，NewBluePill 运行过程中的蛛丝马迹却还是能暴露 Hypervisor 自身，而面对这一切，NewBluePill Hypervisor 又是怎样应对的呢？本节我们就会介绍 NewBluePill 的探测原理和它对应实现的反探测技术。

探测 NewBluePill

根据 NewBluePill 的运行特点，探测 NewBluePill 的方法主要是如下几种：

- 设法创造陷入，通过测量时间判断是否 Hypervisor 在处理事件
- 执行一条可能陷入的指令，查看 TLB 上有多少项发生了变化从而确定是否存在 Hypervisor。
- 探测当前是否开启虚拟机模式

下面我们就将分别介绍每种探测方法和 NewBluePill 的对策：

通过指令执行耗时分析

通过在虚拟机中测量指令执行所消耗时间来判断当前是否存在 NewBluePill Hypervisor 是一个不错的方法，这种方法利用了引入 HEV 技术后带来的开销，如图 7.1 所示：

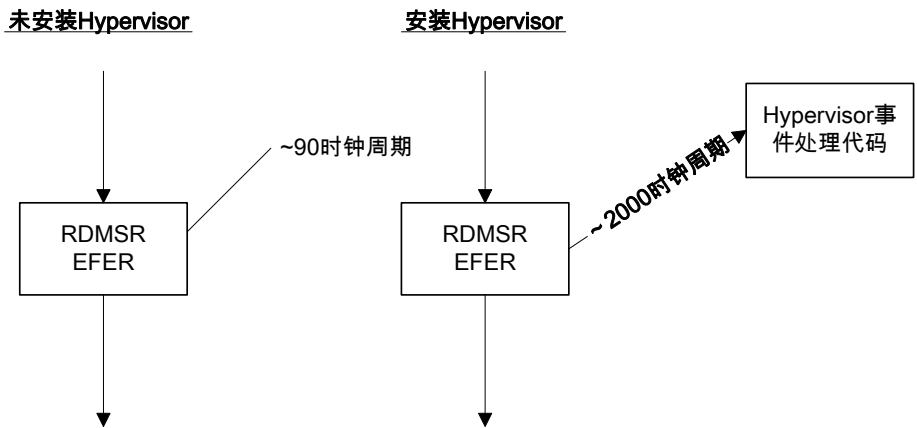


图 7.1 引入 NewBluePill Hypervisor 带来的时间开销

获得当前时间的方法有四种：

- 1) 读取 CPU 的时间戳寄存器（Timer Stamp Counter）
- 2) 利用其它本地时钟（如 HPET¹）

¹ HPET （High Precision Event Timer，高精度计时器），未来将会替代真实计时器，每个 HPET 最多可以有 32 个计时器。有关 HPET 的具体资料请参考网上相关内容。

3) 真实计时器 (Real Time Clock)

4) 外部计时器 (External Clock)

第一种方法利用 RDTSC 指令读取 CPU 的时间戳寄存器, 这种方法获得的时间精度最高, 可以精确到时钟周期, 但是也是最容易欺骗的一种方法。

第二种方法同样有很高的精确度, 但是 Hypervisor 通过对中断的监控也可以欺骗它的返回值。

真实计时器是一种 I/O 设备, 计时精度为毫秒级, Hypervisor 仍能通过对 I/O 操作的监控欺骗返回值。

最后一种方法利用网络传输时间信息, 且一般采用加密手段 (如 NTP 协议), 这就给 Hypervisor 的欺骗带来很大难度, 只能针对某种特定实现欺骗。但是这种方法得到的时间精度很低, 通常为 10ms 级, 在一个典型的 1GHz 主频的处理器上, 在这个时间间隔内已经执行了成百上千万条指令了。

NewBluePill 对于本地时间戳计时器的欺骗手段是利用指令追踪策略, 如果这种探测升级为短时间内大量的探测的话, NewBluePill 的对策是 Blue Chicken 策略, 本节稍后就会介绍这两个策略。

通过观察 TLB 变化分析

而通过观察 TLB 的变化去判断当前是否存在 NewBluePill Hypervisor 也是一种不错的手段。

众所周知, TLB 用于缓存最近访问的内存, 无论是代码段或是数据段。在 NewBluePill Hypervisor 存在的情况下, 一旦因为虚拟机执行某条指令而发生陷入, 那么返回虚拟机中后 TLB 中内容一定是被污染的, 只要有办法探测到这种污染就可以了, 同样的, 如果是短时间内大量的探测, NewBluePill 同样可以使用 Blue Chicken 策略自保。

然而, 这种方法和第一种分析指令耗时的方法本质上确实在探测是否有 Hypervisor 存在而不是在探测是否存在 NewBluePill 等恶意 Hypervisor。考虑到当前和未来不断涌现的利用硬件虚拟化技术的软件, 探测是否开启虚拟机模式及 Hypervisor 存在而确定是否存在类似于 NewBluePill 这样的恶意软件是行不通的。

Blue Chicken 策略

相关文件

NewBluePill-0.32-public\common\Chicken.c

NewBluePill-0.32-public\common\Chicken.h

功能介绍和详细分析

Blue Chicken 策略是这样的一种策略: 当 NewBluePill 发现在短时间内出现了大量因为 VMEXIT 事件造成的陷入, NewBluePill 就怀疑有人在试图探测自己的存在, 这时就会暂时卸载自己实现隐藏。

为了实现这一功能，Blue Chicken 采用了如图 7.2 的循环队列数据结构用于存储每次陷入发生时时间戳寄存器的值（TSC Value）。

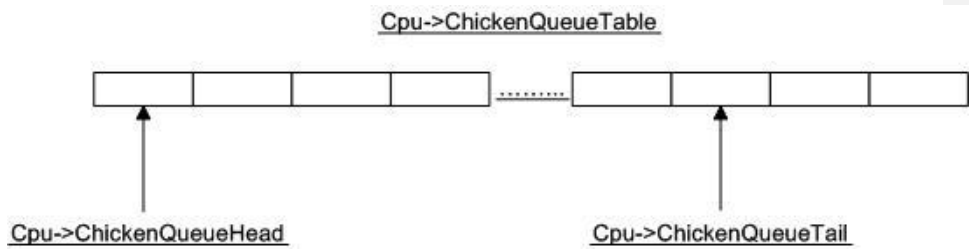


图 7.2 Blue Chicken 策略数据结构

进队操作通过 `ChQueueEnqueue()` 函数完成，进队元素永远插入 `Cpu->ChickenQueueTail` 所指的数组位置，当 `Cpu->ChickenQueueTail` 域已经指向了数组最后位置时，在队列未满的情况下指向数组头部，最后更新 `Cpu->ChickenQueueTail` 域。

出队操作通过 `ChQueueDequeue()` 函数完成，`Cpu->ChickenQueueHead` 指向的元素最先出队，然后更新 `Cpu->ChickenQueueHead` 所指的位置。

`ChickenShouldUninstall()` 函数是 Blue Chicken 策略的核心函数。该函数首先获得队列头尾两元素内部所存储的值（`Chicken.c` 函数第 100 行），然后判断如果在一定时间限度内发生了 `CHICKEN_QUEUE_SZ` 多次陷入的话（`Chicken.c` 函数第 101 行），那么返回 `TRUE` 通知 NewBluePill Hypervisor 卸载自身（`Vmx.c` 文件第 161 行，`Svm.c` 文件第 609 行）。

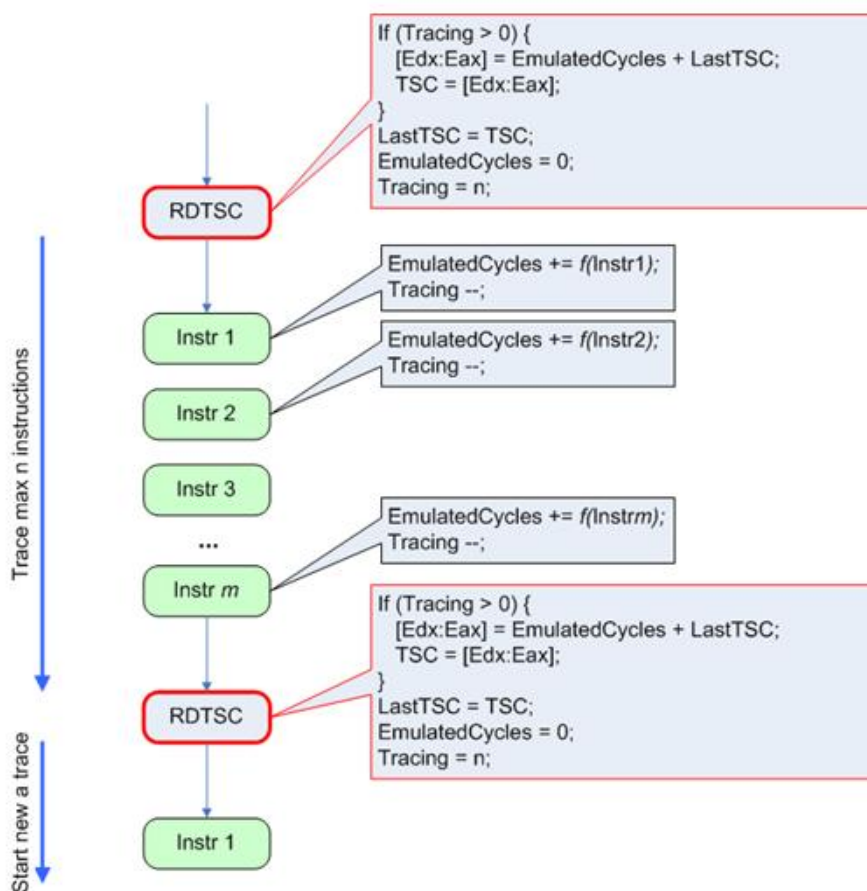
时间欺骗——指令追踪策略

相关文件

NewBluePill-0.32-public\vmx\Vmxtraps.c
NewBluePill-0.32-public\svm\Svmtraps.c

功能介绍和详细分析

NewBluePill 的时间欺骗策略是通过指令追踪（Instruction Tracing）方法完成的，示意图如下所示：

图 7.3 NewBluePill 指令追踪策略示意图¹

NewBluePill 假设，探测器在检测时必然会反复执行一段指令并测量所消耗的时间。所以为了能够更精确的欺骗时间，NewBluePill 选择测量这段程序中的每条指令的真实执行时间。为了达到这一目的，一个关键要素就是在虚拟机访问时间戳寄存器后²，NewBluePill Hypervisor 必须开启单步中断，这样才能监控虚拟机后面执行的若干条指令，获得这些指令的执行信息。

接触过其它嵌入式平台的读者可能会问，为什么在这里我们不能使用通过手册中的信息生成的一张填充有每种指令执行周期的表格呢？

NewBluePill 之所以不这么做是因为，当前绝大多数的处理器都有流水线和缓存，这些因素造成了在运行时我们无法精确的知道每条指令在执行时具体消耗多少指令周期。此外，在比较特殊的情况下，如果某条指令正好处于页面的边缘，那么换页引入的开销我们也是无法通过预先生成的表格考虑到的，因此，运行时刻指令周期数的测量精度要高于预先静态生成表格这种方法的精度。

这种策略具体于每种平台的实现在代码中体现为 INTERCEPT_RDTSCs 开关之间的部分，所涉及的具体处理函数为：

¹ 该图摘自 IsGameOver, <http://www.bluepillproject.org/stuff/IsGameOver.ppt>

² 从第六章相关处理函数中我们可以看到，这包括了 RDTSC, RDTSCP 指令和对 TSC MSR 寄存器的直接操作。

SVM 平台:

- SvmDispatchDB() 函数
- SvmDispatchRdtsc() 函数
- SvmDispatchRdtscp() 函数
- SvmDispatchMsrTscRead() 函数

VT 平台:

- VmxDispatchRdtsc() 函数
- VmxDispatchException() 函数

这些函数具体过程在本书第六章中已有具体描述, 故不在此赘述。

Note 更多关于 NewBluePill 反探测技术的内容请参考 <http://www.bluepillproject.org/> 相关内容以及 *Subverting Vista™ Kernel for Fun and Profit*, Joanna Rutkowska

八、NewBluePill 调试系统

Ok, 经过前面这些的讲解, 想必各位读者已经对 NewBluePill 各部分机理已经有了很多了解, 那么作为一个运行在如此底层并且具备一定代码量的系统, NewBluePill 的打印信息是怎样送出的呢(每次分析蓝屏信息和 dump 文件肯定不是最好的方法, 通过分析打印信息能够节省很多调试时间)? 为什么不利用 WinDDK 的 API 而非要自己实现输出呢? 本章将对这些问题作出解释。

批注 [S45]: 这个部分还需要更深入的探索

相关文件

```
Dbgclient\Dbgclient.c
NewBluePill-0.32-public\dbgclient\Dbgclient.c
Dbgclient\Dbgclient.h
NewBluePill-0.32-public\dbgclient\Dbgclient.h
Dbgclient\Dbgclient_ioctl.h
NewBluePill-0.32-public\dbgclient\Dbgclient_ioctl.h
NewBluePill-0.32-public\common\Portio.c
NewBluePill-0.32-public\common\Portio.h
NewBluePill-0.32-public\common\Comprint.c
NewBluePill-0.32-public\common\Comprint.h
```

功能概述

在 NewBluePill 中, 打印信息的输出主要有两种方式, 通过端口输出和通过共享内存窗口本地输出。后一种方式的示意图如图 8.1 所示:

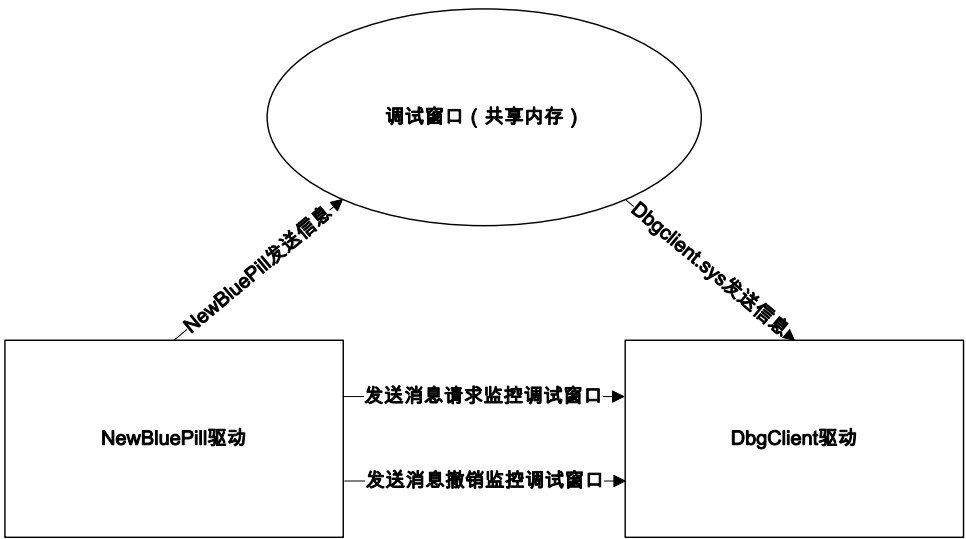


图 8.1 NewBluePill 本地输出信息方式示意图

这样做的原因在于，如果在 NewBluePill 内部完全通过 WinDDK API 中的 DbgPrint() 函数输出信息的话，那么在.....情况下会因为.....而死机。因此，为了获得这部分的调试信息，NewBluePill 创建一段内存，并把所有调试信息都写到这段内存中，随后当系统的 IRQ 级别降低时，Dbgclient.sys 会读取这些内容并在虚拟机中安全的调用 DbgPrint() 函数，将信息打印出来。

批注 [S46]: 调查之

实现细节

下面我们看下 NewBluePill 调试系统具体是怎样实现的，NewBluePill 的调试系统与 HEV 技术本身无关，因此本章所叙述内容完全适用于 VT 平台和 SVM 平台。
我们将按照信息生产者 NewBluePill 和信息消费者 DbgClient 分别予以介绍。

NewBluePill 端调试系统部分

初始化过程

在“第四章 NewBluePill 的启动和卸载”中的启动过程时，我们提到过 NewBluePill 调试系统的初始化，本节我们将以此为入口探索 NewBluePill 端调试系统是如何初始化以及被使用的。
在 Newbp.c 文件第 53 行到 56 行,DriverEntry() 函数中,这几行代码在对 NewBluePill 调试系统做一部分初始化工作：如果准备使用串口输出调试信息，则利用 PioInit() 函数指定输出信息用串口，然后无论是串口输出还是本地输出，都会通过调用 ComInit() 函数（CommonInit 的缩写）为该调试系统绑定 NewBluePill 全局唯一 ID 号 g_BpId，并初始化控制写入调试信息的自旋锁 g_ComSpinLock。
随后，程序运行到 Newbp.c 文件第 64 行时，该行在 NewBluePill 启用本地输出时启用，

通过调用 `DbgRegisterWindow()` 函数，NewBluePill 调试系统申请了 5 页大小的内存作为调试窗口内存区，同时填充 `DebugWindow` 结构体¹，用于保存对该调试窗口的描述信息。

```
typedef struct _DEBUG_WINDOW
{
    UCHAR  bBpId;           /*保存 NewBluePill 全局唯一号（实例间唯一）*/
    PVOID  pWindowVA;       /*保存 NewBluePill 调试窗口虚拟地址*/
    ULONG  uWindowSize;     /*保存 NewBluePill 调试窗口大小 */
} DEBUG_WINDOW,
*PDEBUG_WINDOW;
```

最后，`DbgRegisterWindow()` 函数会调用 `DbgSendCommand()` 函数将 `DebugWindow` 结构体实例发送到 `DbgClient` 驱动（`common\Dbgclient.c` 文件第 72 行，该驱动将自身注册为“\\Device\\ntldbgclient”对象），`DbgClient` 驱动在接收到该消息后就会把该实例添加到监视列表上。`DbgSendCommand()` 函数的作用就是利用 Windows 设备消息发送机制向 `DbgClient` 注册设备发送信息。至此，NewBluePill 端调试系统初始化完成。

卸载过程

NewBluePill 端调试系统的卸载过程相对简单一些，它并不需要显式地做例如释放自旋锁资源等等的过程，因为在 NewBluePill 生命周期结束后，Windows 会管理这些。在卸载过程中唯一要做的事情就是在启用本地调试信息输出的情况下调用 `DbgUnregisterWindow()` 函数（`Newbp.c` 文件第 39 行）释放调试窗口所占内存，并通知 `DbgClient` 将对该 NewBluePill 调试窗口的监视从监视列表中去除掉。`DbgUnregisterWindow()` 函数仍会调用 `DbgSendCommand()` 函数（`common\Dbgclient.c` 文件第 83 行），不过这次发送的消息换成了请求移除一个监视调试窗口的消息。

使用过程

在 NewBluePill 中，打印消息全部使用 `_KdPrint` 宏，该宏定义在 `Common\Common.h` 的第 92 行到第 96 行。在启用调试信息输出的情况（`ENABLE_DEBUG_PRINTS`，默认开启）下，NewBluePill 会通过调用 `ComPrint()` 函数（定义在 `common\Comprint.c` 文件第 106 行）进行统一的打印信息输出管理工作。

`ComPrint()` 函数同时管理对本地调试窗口的信息输出和通过串口的信息输出，函数首先获得旋锁，保证了某一时刻只能对一个打印信息请求进行处理。在打开 `COMPRINT_OVERFLOW_PROTECTION` 开关的情况下，该函数可以对溢出情况进行处理，即禁止在 `COMPRINT_QUEUE_TH` 时钟周期内输出超过 `COMPRINT_QUEUE_SZ` 行字符串，为了达到这个目的，这里同样维护了一个类似于 Blue Chicken 策略的循环队列，用于填充最近打印若干行字符串发生的时刻（同样是保存时间戳寄存器内容），组装字符串所用到的 `snprintf()` 函数（`common\Comprint.c` 文件第 164 行）是一个第三方开源库提供的，在此不进行分析。此

¹ 该结构体实例既用于 NewBluePill 端又会通过设备通信方式传输到 `DbgClient` 端，因此 `DebugWindow` 结构体的定义会在两端出现且定义一致。

后函数里通过调用 `_ComPrint()` 函数 (`common\Comprint.c` 文件第 175 行) 将要输出的字符串送到指定设备, 这个函数会处理输出到端口或者本地调试窗口的具体细节。最后 `ComPrint()` 函数释放旋锁, 整个处理过程结束。

DbgClient 端调试系统部分

DbgClient 用于接收当前系统中所有 NewBluePill 实例发出的调试信息, 并利用 Windows API 打印出来。从 NewBluePill 端调试系统部分的初始化过程我们可以看出, DbgClient 必须先于 NewBluePill 启动, 这样才可以在 Windows 中注册相应设备从而能够接收到 NewBluePill 发送的消息。下面我们就介绍下 DbgClient 的启动、卸载和使用过程。

初始化过程

DbgClient 作为一个独立的驱动而存在, 因此它有自己的 `DriverEntry()` 入口函数 (定义在 `dbgclient\Dbgclient.c` 文件第 283 行)。该函数的工作流程图如图 8.2 所示:

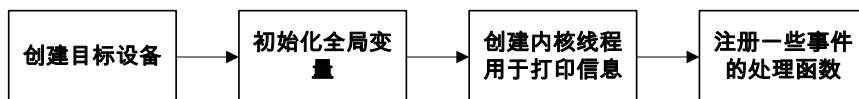


图 8.2 DbgClient 入口函数流程图

进入 `DriverEntry()` 函数, 首先为目标设备声明了 “`\\Device\\itldbgclient`” 和 “`\\DosDevices\\itldbgclient`”¹ 两个设备名 (`dbgclient\Dbgclient.c` 文件第 293 行到 294 行), 并分配了 `DEBUG_WINDOW_IN_PAGES` 页数的内存 (`dbgclient\Dbgclient.c` 文件第 297 行), 在将来使用 `DbgPrint()` 函数打印信息时, 会首先复制要打印的内容到该段内存中, 起到了缓存和快速处理²的作用。随后 `DriverEntry()` 函数调用系统 API, 将 DbgClient 驱动自身创建为设备, 并以这两个设备名注册这同一个设备 (`dbgclient\Dbgclient.c` 文件第 303 行到 314 行)。

随后 `DriverEntry()` 函数初始化 `g_DebugWindowsList`, `g_DebugWindowsListMutex`, `g_ShutdownEvent` 这几个全局变量 (`dbgclient\Dbgclient.c` 文件第 316 行到 318 行), 这三个全局变量用途如下:

- `g_DebugWindowsList` DbgClient 被设计用来同时监控多个调试窗口的输出信息, 因此该列表用于存储每个调试窗口对应 `DebugWindow` 结构体实例。
- `g_DebugWindowsListMutex` 该互斥锁用于互斥执行 `PrintData()` 函数, 该函数负责调用 `DbgPrint()` 函数打印信息。
- `g_ShutdownEvent` 这个对象被用于保证打印线程在 DbgClient 驱动被卸载前不会停止。使用了 Windows 内部对象的消息通知技术 (Event Notification)。

初始化全局变量工作完成后, `DriverEntry()` 函数会创建内核线程, 用于打印各个调试窗口的输出信息 (`dbgclient\Dbgclient.c` 文件第 320 行到 349 行), 其中会测试下所创建的线程拥有的访问权限 (`dbgclient\Dbgclient.c` 文件第 334 行到 347 行)。

¹ 该设备名会把设备注册到系统全局对象空间上 (`\\GLOBAL??\\`)。

² 一般情况下, I/O 操作慢于内存操作, 用这种机制可以提升 NewBluePill 生产消息的速度。

此时打印信息线程已成功开启并运行，`DriverEntry()` 函数最后做的工作就是注册事件处理函数 `DriverDispatcher()` (`dbgclient\Dbgclient.c` 文件第 352 行到 354 行)，该函数在 `NewBluePill` 开启、关闭、向 `DbgClient` 设备发送命令时会被执行。

此时 `DbgClient` 设备顺利启动。

卸载过程

`DbgClient` 通过调用 `DriverUnload()` 函数执行卸载工作，其卸载过程如图 8.3 所示：

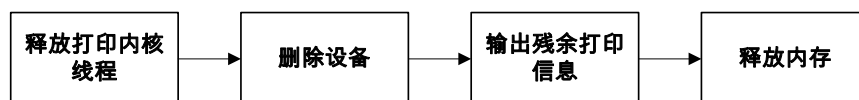


图 8.3 `DbgClient` 卸载过程流程图

该函数首先向打印线程监听的 `g_ShutdownEvent` 事件发送消息，这会使得线程退出循环结束 (`dbgclient\Dbgclient.c` 文件第 61 行到 69 行)，然后判断 `g_pScanWindowsThread` 是否指向在初始化过程中所创建的打印内核线程，如果是则释放对它的引用 (`dbgclient\Dbgclient.c` 文件第 250 行到 252 行)，此时该线程引用为 0，`Windows` 也就可以自动终止该线程并回收其占用的空间。随后，`DriverUnload()` 函数删除自身所代表的设备 (`dbgclient\Dbgclient.c` 文件第 256 行到 259 行) 并调用 `PrintData()` 函数输出可能留在调试窗口中的残余的打印信息，随后遍历 `g_DebugWindowsList` 链表释放其中每个元素所占用的空间，最后函数释放 `g_pDebugString` 所占用的内存空间 (`dbgclient\Dbgclient.c` 文件第 275 行到 276 行)。至此，`DbgClient` 的卸载过程全部完成。

使用过程

通过对 `NewBluePill` 端调试系统启动、关闭和使用过程的介绍，我们可以知道它主要利用了 `Windows` 的对象通信机制。作为调试信息的接收者，`DbgClient` 在初始化过程的最后定义了 `NewBluePill` 端调试系统使用它的时候，均要移交给 `DriverDispatcher()` 函数做具体处理工作，下面我们就看看 `DbgClient` 是如何被使用的。

`DriverDispatcher()` 函数仅起到过滤事件的作用，它会调用 `DeviceControl()` 函数 (`dbgclient\Dbgclient.c` 文件第 224 行) 仅针对 `IRP_MJ_DEVICE_CONTROL` 造成的触发进行处理¹。`DeviceControl()` 函数根据 `NewBluePill` 端调试系统发送的消息命令类型进行不同的处理：如果是 `IOCTL_REGISTER_WINDOW` 类型，也就是 `NewBluePill` 端调试系统请求 `DbgClient` 监控一个调试窗口，那么 `DeviceControl()` 函数会首先遍历监控链表 `g_DebugWindowsList`，查看是否已存在对这个调试窗口的监视 (`dbgclient\Dbgclient.c` 文件第 112 行到 123 行)，如果不存在则将其封装到一个 `DEBUG_WINDOW_ENTRY` 类型结构体中并将该实例添加到监控链表最后位置上 (`dbgclient\Dbgclient.c` 文件第 125 行到 145 行)。

如果是 `IOCTL_UNREGISTER_WINDOW` 类型，也就是 `NewBluePill` 端调试系统请求 `DbgClient` 撤销监控一个调试窗口，那么 `DeviceControl()` 函数会首先调用 `PrintData()` 函数输出所有调试窗口的残余信息 (`dbgclient\Dbgclient.c` 文件第 162 行)，随后遍历监控链表

¹ 有关 `Windows IRP` (Input/Output Request Packets) 的具体信息，请参考网上相关内容。

g_DebugWindowsList, 移除目标调试窗口信息(dbgclient\Dbgclient.c 文件第 166 行到 185 行)。

而 DbgClient 所创建的线程则在 DbgClient 生命周期内一直运行, 在 DbgClient 被卸载前, 该线程执行的 ScanWindowsThread() 函数由于一直等待不到 g_ShutdownEvent 事件而超时, 因此陷入死循环中 (dbgclient\Dbgclient.c 文件第 61 行到 69 行), 从而得以不断执行 PrintData() 函数。

PrintData() 函数是负责利用 WinDDK 中 DbgPrint() 函数输出信息的核心函数, 该函数首先利用互斥锁 g_DebugWindowsListMutex 确保自己的执行不会被打断, 然后遍历监控链表 g_DebugWindowsList 中每一个调试窗口, 并复制其中内容到本地临时内存中 (dbgclient\Dbgclient.c 文件第 31 行), 随后遍历要打印的内容, 将回车符替换为 NULL, 再利用 DbgPrint() 函数逐行打印这些内容 (dbgclient\Dbgclient.c 文件第 35 行到 41 行)。当打印任务完成后释放互斥锁并成功返回。

总结

正如本章开始中所提到的那样, 如果在 NewBluePill 内部完全通过 WinDDK API 中的 DbgPrint() 函数输出信息的话, 那么在..... 情况下会因为..... 而死机, 读者可以通过修改_KdPrint 的宏定义来测试这一点, 并用 windbg 观察此时的输出和函数调用堆栈, 相信一定有意意外收获。

这也就提醒了我们, 在底层上实现自己的系统, 调试系统的支持必不可少, 它就像脚手架一样帮助我们实现这个系统, 当已有的支持不能很好的帮助我们实现这一目标时, 我们必须自己实现——好的调试系统能够减少大量的开发时间和测试时间。

批注 [S47]: 调查之

关于 Bpknock 触发器

由于 bpknock 很简单, 所以本书中不再另开章节对其进行专门讲述。

Bpknock 程序是 NewBluePill 的演示程序 (bpknock\Bpknock.c 文件), 其作用是通过调用 cpuid 这条汇编指令触发 #VMExit 事件, 使得 NewBluePill 陷入 Hypervisor 处理异常, 最后将自己定义的返回结果赋值给相应寄存器, 再回到 OS 中读出该修改过的返回结果, 从而证明 NewBluePill 确实生效。核心函数分析如下:

```
ULONG32 __declspec(naked) NBPCall (ULONG32 knock) { // 使用
__declspec(naked)来自己管理函数调用堆栈
    __asm {
        push    ebp
        mov     ebp, esp
        push    ebx           ; 保护 ebx,ecx,edx 寄存器
        push    ecx
        push    edx
        cpuid           ; 要用 cpuid 触发异常陷入 VMM
        pop     edx
        pop     ecx
        pop     ebx
    }
```

```
    mov     esp, ebp
    pop     ebp      ;恢复局部栈
    ret
}
```

PART3 实验部分

这部分出现的几个实验难度较高，同时由于所需步骤和知识点穿插于本书各个章节，因此没有具体实验步骤。建议在通读并理解 NewBluePill 全部流程后，再独立完成后续几个实验，相信一定会加深您的理解，发现新的宝藏。

九、动手写自己的第一个 HVM 程序

实验目的

1. 实践 VT/SVM 技术相关指令
2. 实践 VT/SVM 技术中虚拟机的启动过程
3. 掌握 Hypervisor 的 On the fly 安装方式的原理

实验概述

按照几乎所有新技术新语言的学习习惯，我们同样从 Helloworld 开始做自己的第一个基于 HEV 技术的 Hypervisor，当然为了便于学习和调试，我们采用了跟 NewBluePill 一样的 On the fly 方式安装。

我们的 Helloworld HVM 程序通过如下方法实现：拦截 CPUID 指令调用，根据从 Console 过来的传入值生成相应传出值，但是如果传入值为 100，那么我们修改 eax,ebx,edx 三个寄存器的内容为“Helloworld!”，为了证明我们的结果确实是通过拦截 CPUID 传出的，可以做如下对比实验：卸载掉我们的 HelloWorld 驱动，然后通过 console 传入 100，查看输出，如果不再是“Helloworld!”而是其它字符串，那么我们的实验成功。

由于仅是为了学习 HEV 技术基础，所以我们在删除了 NewBluePill 内存系统、NewBluePill 反探测系统和 NewBluePill 调试系统。如果仅在 VT/SVM 其中一种平台上实验，我们甚至可以删除了 NewBluePill 的平台无关性部分。这样一来，我们可以得到一个最简单的 Hypervisor，这将大大简化代码尺寸并便于我们理解和以后更进一步的开发。

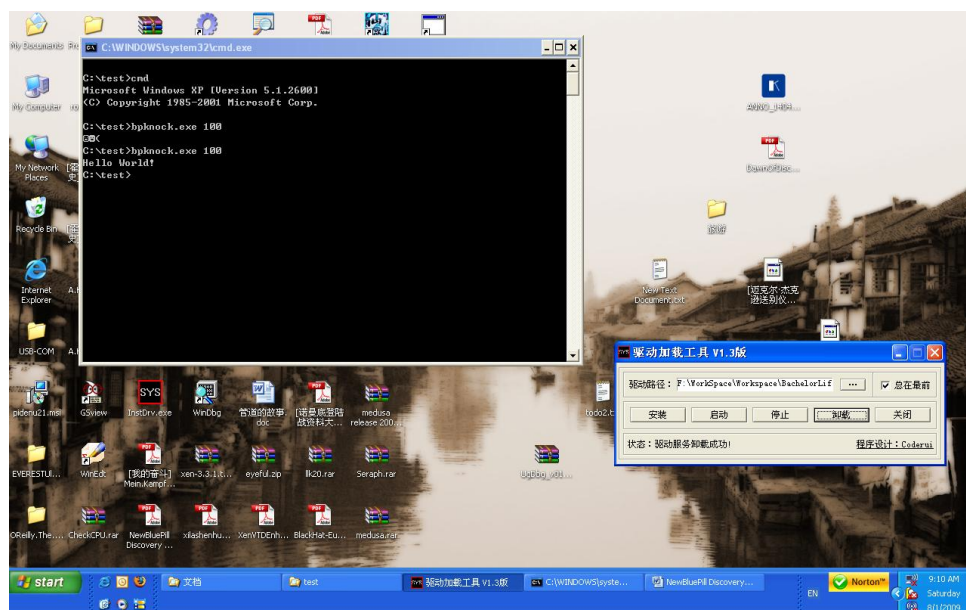


图 9.1 Helloworld Hypervisor 运行效果图

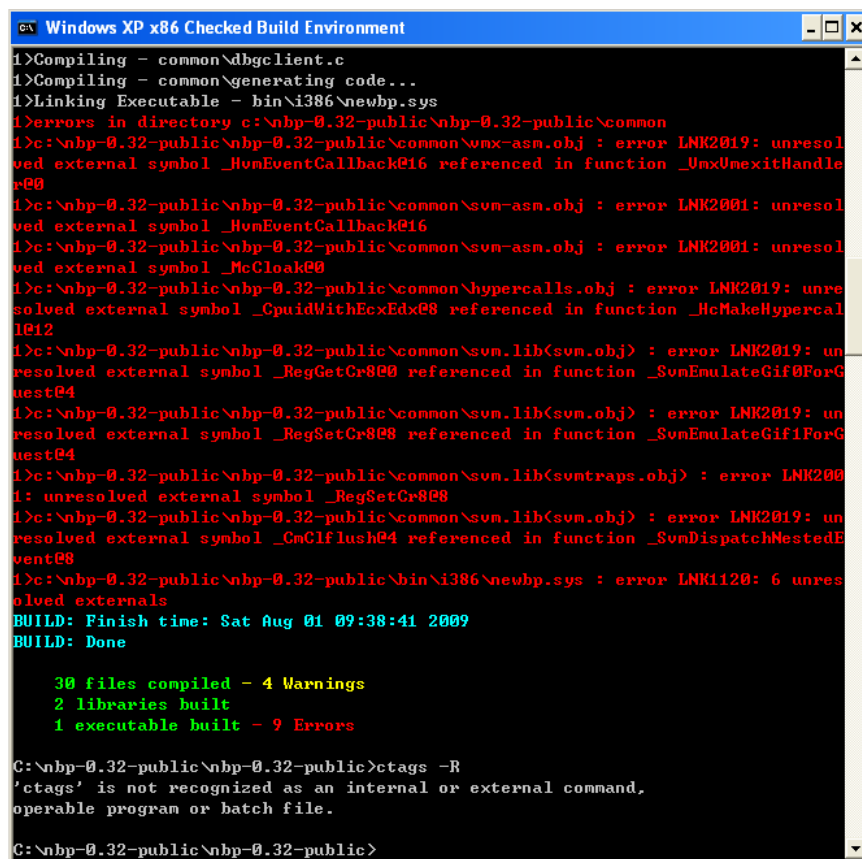
十、移植 NewBluePill 到 32 位系统

实验目的

1. 熟悉 NewBluePill 整体作用和各个模块作用
2. 熟练运用 VT/SVM 技术，尤其是 VMCS/VMCB 配置
3. 熟悉 x64 和 x86 下地址翻译（包括 PAE 模式）和 Windows 页表结构，并能熟练构造与维护页表。

实验概述

本移植实验目标是将 NewBluePill 从 x64 平台上移植到 x86 平台上，并能保证其继续运行。直接在 x86 环境下编译 NewBluePill 项目会出错，这是因为在 NewBluePill 公开代码中，x86 部分并未完全实现。



```
Windows XP x86 Checked Build Environment
I>Compiling - common\dbgclient.c
I>Compiling - common\generating code...
I>Linking Executable - bin\i386\newbp.sys
I>errors in directory c:\nbp-0.32-public\nbp-0.32-public\common
I>c:\nbp-0.32-public\nbp-0.32-public\common\vmx-asm.obj : error LNK2019: unresolved external symbol _HvmEventCallback@16 referenced in function _UmxFmexitHandle
r@0
I>c:\nbp-0.32-public\nbp-0.32-public\common\svm-asm.obj : error LNK2001: unresolved external symbol _HvmEventCallback@16
I>c:\nbp-0.32-public\nbp-0.32-public\common\svm-asm.obj : error LNK2001: unresolved external symbol _HcCloak@0
I>c:\nbp-0.32-public\nbp-0.32-public\common\hypercalls.obj : error LNK2019: unresolved external symbol _CpuidWithEcxEcx@8 referenced in function _HcMakeHypercal
l@12
I>c:\nbp-0.32-public\nbp-0.32-public\common\svm.lib(svm.obj) : error LNK2019: unresolved external symbol _RegGetCr8@0 referenced in function _SvmEmulateGif0ForG
uest@4
I>c:\nbp-0.32-public\nbp-0.32-public\common\svm.lib(svm.obj) : error LNK2019: unresolved external symbol _RegSetCr8@8 referenced in function _SvmEmulateGif1ForG
uest@4
I>c:\nbp-0.32-public\nbp-0.32-public\common\svm.lib(svmtraps.obj) : error LNK2001: unresolved external symbol _RegSetCr8@8
I>c:\nbp-0.32-public\nbp-0.32-public\common\svm.lib(svm.obj) : error LNK2019: unresolved external symbol _CmClflush@4 referenced in function _SvmDispatchNestedE
vent@8
I>c:\nbp-0.32-public\nbp-0.32-public\bin\i386\newbp.sys : error LNK1120: 6 unresolved externals
BUILD: Finish time: Sat Aug 01 09:38:41 2009
BUILD: Done

30 files compiled - 4 Warnings
2 libraries built
1 executable built - 9 Errors

C:\nbp-0.32-public\nbp-0.32-public>ctags -R
'ctags' is not recognized as an internal or external command,
operable program or batch file.

C:\nbp-0.32-public\nbp-0.32-public>
```

图 10.1 直接编译 x86 版本的 NewBluePill 的出错信息

本实验的任务，就是完全实现 NewBluePill x86 部分，并能正确运行。由于这其中除硬件虚拟化技术外，还牵涉到 Windows 内存管理和页表管理以及段寄存器、GDTR 和 IDTR 的处理，因此建议在实验过程中要参考 Windows Internals 和 Intel/AMD 手册相关部分。

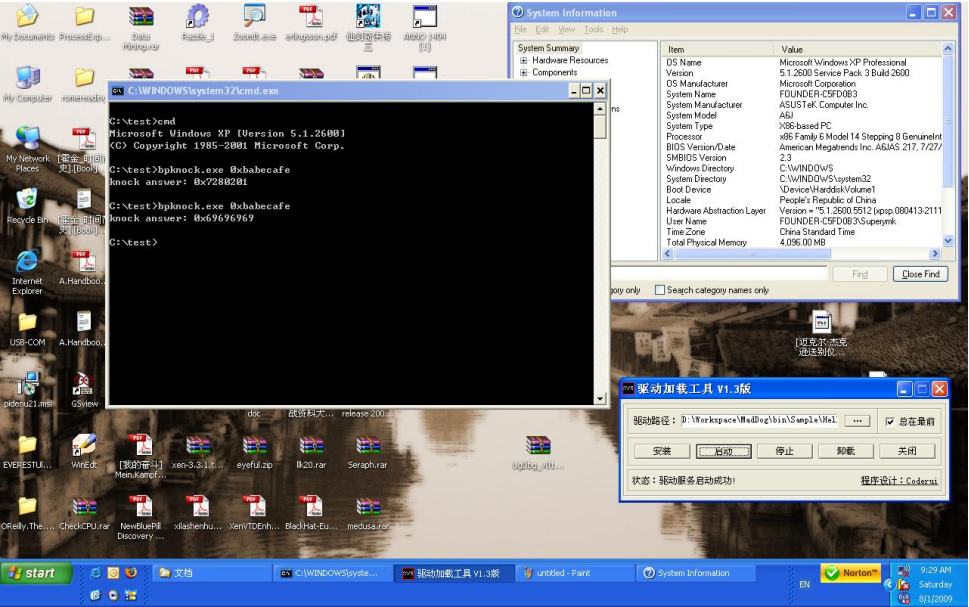


图 10.2 x86 平台下 NewBluePill 运行效果图

提示

除了寄存器组，寄存器长度，地址翻译模式外，要特别注意 VMCS/VMCB 结构体的配置，其中有一些地方的配置是 x64 特有的，如果不对其进行修改是无法在 x86 下正常运行的。

此外，在原来的 NewBluePill 中存在一些 Bug，如不能正确停止 NewBluePill 驱动，返回错误的值等等，可在移植过程中修复这些 Bug。

十一、 开发基于 HEV 技术的注册码验证器

实验目的

1. 实践 VT 技术相关指令
2. 实践 VT 技术中 VMX 抢占计时器技术
3. 实践 NewBluePill 的内存隐藏技术

实验概述

在前面的实验中，我们已经对虚拟化技术有了初步的了解。这个实验将展示虚拟化技术在实际生活中的运用。

如今，共享软件和大多数的商业软件都在使用注册码技术来保护自己的版权，然而，由于验证程序处于 Ring-3 特权态（少数处于 Ring-0 特权态），因此很容易通过动态分析的手段改变跳转/改变语义或者推算出算法，从而破解掉软件。

这个问题的根源在于两个：

- 1) 注册码验证系统的运行时空间操作系统可见
- 2) 注册码的验证一般只有一次¹

引入虚拟化技术后，由于我们拥有了比操作系统更高的权限，再加上我们在 NewBluePill 中所看到的内存隐藏技术，所以我们可以尝试去解决这个问题。考虑下面的模型：

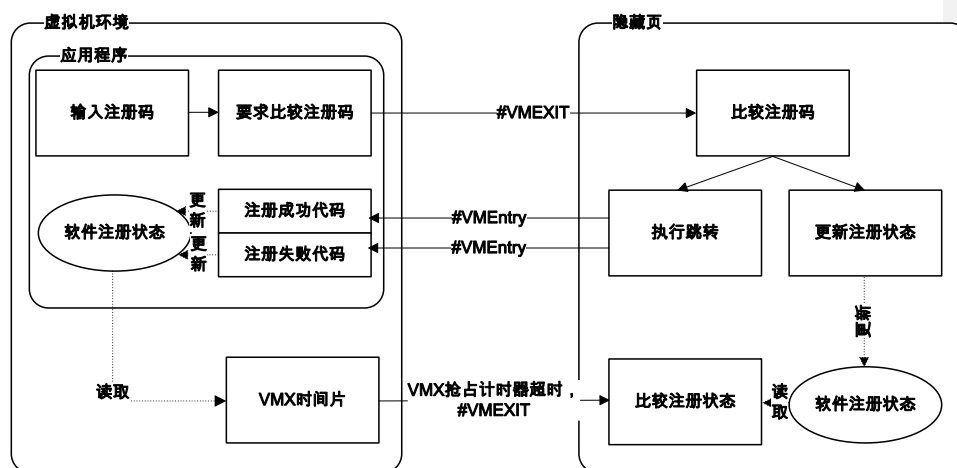


图 11.1 注册码验证器流程图

¹ 其实这个问题也与操作系统可见所有用户程序空间有关，在普通情况下，软件多次验证注册码基本不能带来任何好处。

由图 11.1 所示，将比较注册码的关键部分内存隐藏，从而无法通过动态分析看到这部分内存。此外，通过 VMX 抢占计时器（VMX Preemption Timer）每过一定时间判断一次软件真实注册状态¹，防止破解者通过修改应用程序的注册失败代码语义而修改保存在应用程序中的软件注册状态²。当在 Hypervisor 中发现应用程序的软件注册状态不等于自己保持的注册状态时，终止应用程序的运行。

¹ VMX 抢占计时器（VMX Preemption Timer）可能只在很少的 CPU 上被支持，作为一种替代解决方案，我们可以采用 CR3 计数的方法——定量 CR3 切换发生后，视作超时。这种方法的缺点是会造成稍多一些的 #VMEXIT 事件。

² 在应用程序部分保存软件注册状态可以起到优化的作用，当其状态为未注册时，可以不必再去通过 VMX 抢占计时器去陷入检查。

十二、 NewBluePill 完全隐藏了？

实验目的

1. 探索 x86、x64 体系结构高级主题——缓存（Cache）
2. 更深层次探索 VT/SVM 技术限制

实验概述

本实验在于讨论 NewBluePill 是否真的不可被探测到。NewBluePill 已经实现了时间欺骗和 BlueChicken 策略，而按照定义，如果 VT 和 SVM 技术提供 Full Virtualization，那么 NewBluePill 确实不能被发现，换句话说，确实从虚拟机的视线中完全隐藏。

然而仔细阅读手册我们可以发现，当前 VT 技术的做法是在每次 #VMEXIT 事件发生时强制刷新缓存，SVM 技术则在某些芯片上不会这么做，也就是说，我们可以在虚拟机中通过探测缓存变化来确定是否存在 Hypervisor。

本实验思想如图 12.1 所示，首先用已知地址映射填充 TLB 所有项，然后设法陷入 Hypervisor，由于 CPU 需要执行 Hypervisor 中代码，不可避免的需要改变 TLB 中的一些地址映射，从而可以通过回到虚拟机中检测 TLB 内容的方法检测到 Hypervisor 的存在。

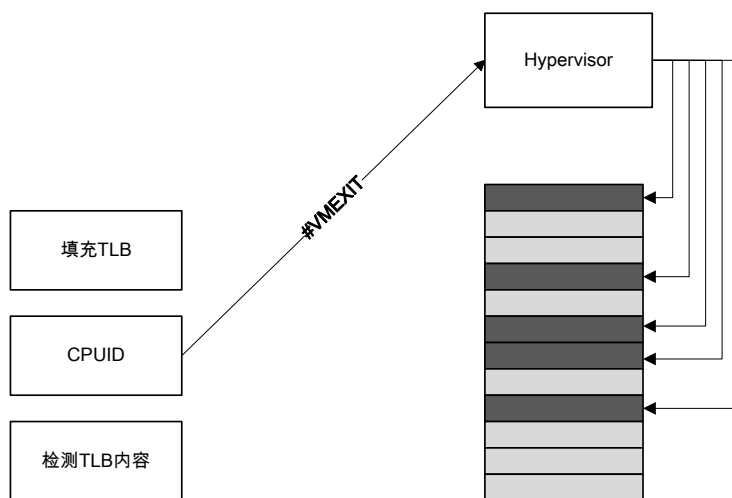


图 12.1 探测 Hypervisor 实验思想

然而，在 x86、x64 体系结构中，TLB 是无法通过编程手段直接访问的，因此必须通过地址访问的方法才能达到填充 TLB 的目的。我们可以通过 CPUID 指令获得一个处理器的具体 Cache 信息，例如，在 AMD (Family 15) 芯片上，L1 数据缓存信息如下：

- 数据缓存大小：64KB（L1）
- 相连缓存器（Cache Associativity）：2-way

- 每缓存单元大小：64 字节

由此可以推算

- 单元数： $64\text{KB}/64\text{B} = 1024$ 个
- 组数量 (Sets)： $1024/2 = 512$
- Index 宽度： $\log_2(512) = 9$
- Offset 宽度： $\log_2(64) = 6$

TLB 信息如下：

- 4K 页缓存项：32 (L1)
- 2M 页缓存项：8 (L1)
- 4K 页缓存项：512 (L2)

所以，我们可以分配 32 个 4K 页面，同时一定要注意控制访问地址的 index 值，避免 L1 数据缓存冲突，否则即使在未安装 Hypervisor 情况下，遍历 TLB 各项映射也许访问 L2 Cache。如此我们就可以通过测试访问 TLB 项的时间判定是否有 Hypervisor 的存在。

A. 其它有关 HVM 技术的项目

Xen

Xen 作为一个知名开源虚拟机项目，从 3.0 起便引入了对硬件虚拟化技术的支持，当前，Xen 已经发布了 3.4 版本。Xen 使用的是虚拟化技术中的混合虚拟化模型（Hybrid Virtualization），如图 A.1 所示。

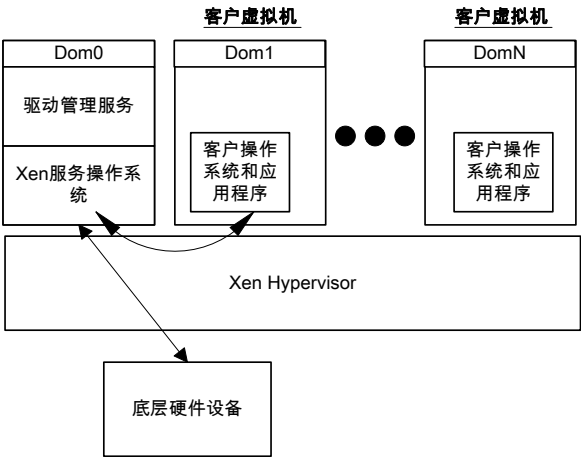


图 A.1 Xen 虚拟化模型

通过使用这种方式的虚拟化，由于 Xen 对客户虚拟机暴露了统一的接口，因此客户虚拟机的透明迁移成为可能，客户虚拟机不必关心也看不到底层硬件，因此也就不必担心迁移目标物理机是否与自身产生兼容问题，只要目标机正在运行 Xen 系统，虚拟机就可以无缝迁移于其上。

此外，另一个优点是，通过采用这种模型，利用适当的方法，即使 Dom0 发生系统崩溃，那么也能够仅对客户虚拟机造成最小影响，一个方法就是将虚拟机当前状态保存到硬盘上。

当然这种模型也有一定的缺点，比如：由于客户虚拟机每次访问底层硬件设备都要与 Dom0 交互，自然引入了一定的开销，Xen 为了尽量降低由此带来的开销，使用了超虚拟化技术（Paravirtualization），尽量降低客户虚拟机和 Dom0 之间的交互，但这样做就造成了需要在客户操作系统中安装一些驱动，因此改变了客户操作系统。

KVM

KVM 是另外一个知名的开源虚拟机项目，用于在 Linux 操作系统上对 x86 平台提供虚拟化支持，从 Linux 内核版本 2.6.20 起，KVM 被包含到了内核当中。KVM 使用的是虚拟化技术中的基于操作系统的虚拟化模型（OS-Hosted Virtualization），如图 A.2 所示。

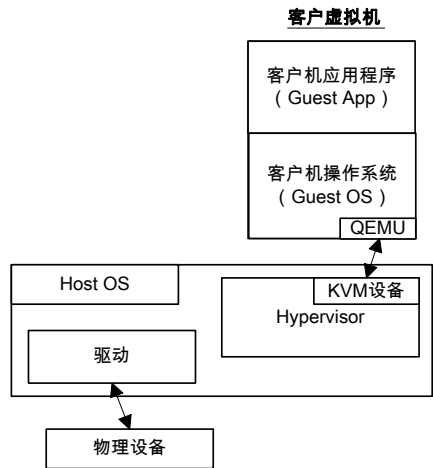


图 A.2 KVM 虚拟化模型

这种虚拟化模型同样支持客户虚拟机的透明迁移，除此以外，由于该模型借助于目标操作系统上的已有驱动程序，因此可以大大降低部署 Hypervisor 到不同硬件系统上的难度。

同时在该模型中，也可以利用操作系统已经提供的一些方法辅助虚拟机的运行，比如 I/O 设备的发现和探测等。

但是该模型也存在一些缺点，由于虚拟机和 Hypervisor 依赖于宿主操作系统，因此它的可靠性，可用性和安全性均受宿主操作系统所限。宿主操作系统一旦重启，那么所有的虚拟机不得不停止。

另外，由于这种模型中，每个客户虚拟机常以一个独立进程的形式出现，因此客户虚拟机的运行收到宿主操作系统进程调度的影响，这在虚拟化一个实时操作系统时是不可接受的。

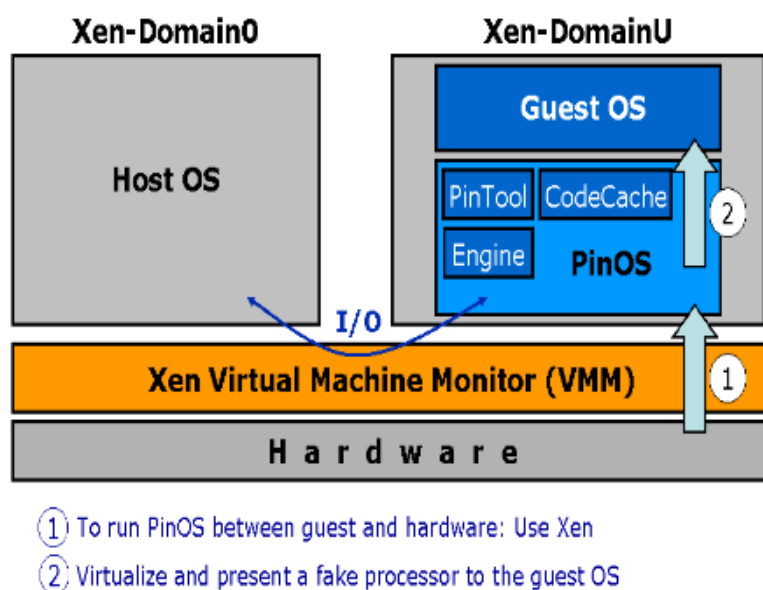
V3VEE

V3VEE 项目 (<http://v3vee.org>) 是一个开源虚拟机监控层框架项目，由西北大学和新墨西哥州大学主持。该项目基于硬件虚拟化技术，意在创建一个独立于任何操作系统的 Hypervisor 框架，开发者可以在编译时刻和运行时刻对基于该框架开发的 Hypervisor 进行配置，从而加速 Hypervisor 的开发工作。

当前，该项目主要用于学术目的，在 WIOV 会议和 GoVirtual 社区上已经出现了一些有关研究。

PinOS

PinOS 项目 (<http://www.ckluk.org/ck/pinos.html>) 是另外一个基于硬件虚拟化技术的项目，它基于 Xen 实现了一个操作系统动态运行测量统计框架，同时不需要另外修改已有操作系统。其构架图如图 A.3 所示：

图 A.3 PinOS 系统架构图¹

该项目也是硬件虚拟化技术可用于其它方面的一个明证。

BitVisor

BitVisor 项目 (<http://www.bitvisor.org/>) 发布于 2009 年 3 月 31 日，它利用硬件虚拟化技术提供了一个安全服务实现平台，支持诸如信息加密，外围设备数据流拦截等安全功能的实现。

由于其开源和微 Hypervisor 的特性，因此该项目也很适合于进行硬件虚拟化技术的研究，尤其是其中将安全和硬件虚拟化技术的结合，与 PinOS 项目一样，大大扩展了硬件虚拟化技术的应用范围。

¹ 该图摘自 PinOS 项目网站

B. 其它安全技术

通过全书对硬件虚拟化技术的介绍我们可以发现,硬件虚拟化技术在不影响已有应用程序、操作系统和硬件平台的情况下引入了新的逻辑层,由于其拥有更高的特权级,因此该层可被用于安全领域。在本书前文中我们也提到 TPM 模块, Intel Trusted Execution Technology (Intel TXT) 利用该模块实现另一种硬件安全技术,其在与硬件虚拟化技术配合使用时会发挥出更强的保护能力,下面我们就将介绍 Intel TXT 技术。

Intel TXT 技术

为了抵御恶意软件攻击, Intel TXT 技术提供了一系列基于硬件的安全解决方案,借此来提供安全的软件执行环境,主要包括四方面的能力:

- 受保护的内存空间用于存储敏感数据和程序段,未授权的进程不可能访问到这些物理内存。
- 利用密钥对敏感进程所使用的数据进行加密,无论这些数据正在被使用还是已经写入磁盘当中。
- 一系列自我证明机制用于保证 Intel TXT 环境的安全,同时保证通过身份验证的程序能运行在保护空间当中。
- 受保护的输入输出:通过提供受保护的外部设备输入输出通道,防止未授权的程序监听设备真实数据流。

Intel TXT 技术定义了两种身份验证模型,本地验证和远程验证。前者通过利用本地 TPM 模块对当前执行环境测度和比较完成任务,后者则主要依靠本地 TPM 通知远程设备/模块验证需求,这两种模型为不同软件引入了充足的保护能力。

在与 Intel VTx 和 Intel VTd 技术结合时, Intel TXT 可以提供对 Hypervisor 身份的检查,从而保证 Hypervisor 的安全,确保 Hypervisor 数据和代码的私密性。

SVM 和 TPM 模块的结合

正如在很多方面 Intel 和 AMD 技术的相似,在对 TPM 模块的支持和使用上,AMD 的 SVM 技术也对此进行了相应设计。

这主要是应用在安全性方面,通过执行 SKINIT 指令,处理器被重新初始化并建立一个安全执行环境,随后安全加载器 (Secure Loader) 被复制到 TPM 设备上进行检查,由于 TPM 设备是一个外部硬件设备,这也就防止了 SKINIT 指令被模拟执行,之后,安全加载器就可以启动 Hypervisor 并保证其可信的执行了。

Note 在本书中我们并不详细探究 Secure SVM 技术,关于 Secure Loader Block 和可信多处理器引导,请参考 *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*, Chapter 15.26 Secure Startup with SKINIT 和 15.27 Security Exception (#SX)

c. 相关软件和参考文档

相关软件

- AMD SimNow™ Simulator: <http://developer.amd.com/cpu/simnow/Pages/default.aspx>
- VMware Fusion
- Parallels Desktop for Mac
- Parallels Workstation
- DNGuard HVM

参考文档

- [1] Virtualization <http://www.answers.com/virtualization>
- [2] Full Virtualization, http://en.wikipedia.org/wiki/Full_virtualization
- [3] An Introduction to Hardware-Assisted Virtual Machine (HVM) Rootkits
<http://www.megasecurity.org/papers/hvmrootkits.pdf>
- [4] Pacifica – Next Generation Architecture for Efficient Virtual Machines
http://developer.amd.com/assets/WinHEC2005_Pacifica_Virtualization.pdf
- [5] AMD I/O Virtualization Technology (IOMMU) Specification
http://support.amd.com/us/Embedded_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf
- [6] Intel® Virtualization Technology for Directed I/O
<http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art02.pdf>
- [7] AMD64 Architecture Programmer's Manual, Volume 2: System Programming, Rev 3.14
- [8] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B
- [9] 64 비트 윈도우 커널 분석 AMD64 (AMD64 架构的 64 位 Windows 内核分析)
<http://greemate.tistory.com/attachment/ck010000000001.pdf>
- [10] 64 비트 윈도우 커널 분석 IA64 (IA64 架构的 64 位 Windows 内核分析)
<http://greemate.tistory.com/attachment/dk010000000001.pdf>
- [11] Windows Internals, Fourth Edition
- [12] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B
- [13] Trusted Execution Technology http://en.wikipedia.org/wiki/Trusted_Execution_Technology
- [14] Intel® Trusted Execution Technology Overview
http://developer.intel.com/technology/security/downloads/TrustedExec_Overview.pdf