# 一 Guest OS页表管理概述

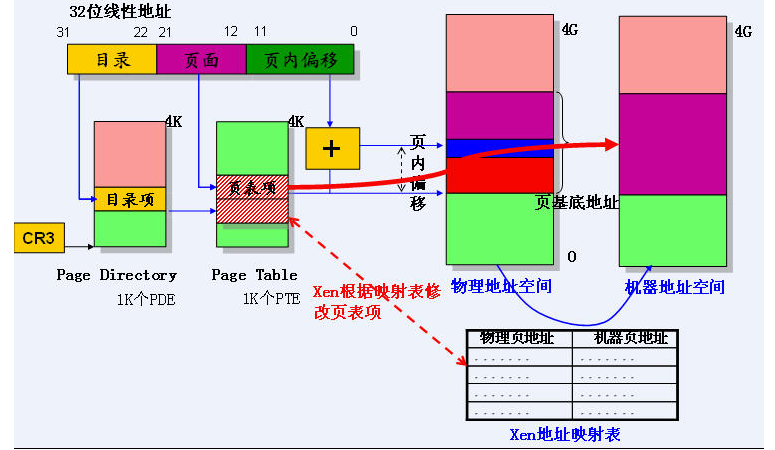
VMM对于客户机物理页信息的维护主要包括引用计数、所有者(虚拟机id)和页类型。对每个客户机物理负的所有者和用途进行跟踪，使得VMM能够对不同的虚拟机进行安全隔离。

客户机物理页按用途可以分为可写页、页表页、页目录页和描述符表页。这些页的类型是互斥的，也就是说，一张客户机物理页不可能同时为两种类型。并且对于每种类型，VMM都维护了一个独立的引用计数，在对页类型进行修改时，该页类型的引用计数必须为零。在Guest OS的页表映射中，页表页、页目录页和描述表页都必须是只读的。这保证了Guest OS不能随意修改自己的页表，也就是Guest OS创建、修改、删除页表的行为都会被Hypervisor所截获，从而保证了一个虚拟机不能随意访问其他虚拟机的内存。

在不能直接修改页表的情况下，Guest OS需要通过相关的超级调用(Hypercall)来更新页表，VMM会对相应页表项进行严格的检查和验证。在进行实际更新之前，VMM会对更新的请求进行验证，任何试图映射到所分配的内存之外物理页的更新都会被会禁止，任何创建到页表页的可写映射的更新也会被禁止。进行验证时，VMM所维护的页类型和引用计数就起关键作用。例如，Guest OS不能创建到页表页的可写映射，因为这要求该页既为页表页，又同时为可读写页，违反了类型的互斥原则。因为超级调用是开销比较大的操作，可以对页表的更新进行批处理，即在一次超调用中进行多个页表的更新。此外，Guest OS更新M2P映射表（位于xen）需要VMM的验证。

**虚拟地址转换：**PV模式下的Xen对于虚拟地址转换采用的是直接模式。Guest OS使用自己的页表访问内存，而Xen需要改写OS内核对于页表的访问方式以及TLB的操作方式，利用hypercall完成MMU的更新。

当新分配一个物理Page时，会首先为该Page分配页表项（PTE entry），然后PTE指向该Page。在Guest OS中，Page对应的PTE项指向物理地址。Hypervisor控制着所有Guest OS的物理地址和及其机器地址关系，通过P2M表（位于guest OS），Hypervisor获得该Page的机器地址，从而修改PTE项将物理地址修改为机器地址。这样Guest OS访问线性地址就能够直接通过CPU的MMU直接找到相应的机器地址。



# 二 漏洞相关的IOTLB问题概述

HYPERVISOR\_mmuext\_op()需要把整个页表的基址设置到cr3，也就是把该进程的每一级页表的页类型从可写页验证（validate）并设置成相应的页表类型。总的来说，Hypervisor主要做了两件事： 1. 通过page table walk做了一系列检查后（第四章会具体讲到），将每一级页表的物理页类型合法设置为页表类型，合法是指页类型被验证有效（validated）；2. 将每一级的页表页设置成页表类型。

在这个过程中，我们会发现，IOTLB发生刷新，这在意料之中。因为Hypervisor通过I/O页表建立起客户物理地址和机器地址间的映射，从而使外设只能访问客户机指定的可写内存区域，其页类型只能是可写页，以避免DMA攻击。

那么，当一个物理页其页类型由可写页被设置成页表时，Hypervisor必须把该物理页从I/O页表中清除，解除相应的映射关系，这是在iommu\_unmap\_page()实现的。当解除了相应的页表项后，其IOTLB必须被刷新，以保证I/O页表的同步和安全。

综上，CVE所爆出的漏洞就在于其IOTLB并没有如我们所分析的那样做刷新操作，从而导致漏洞产生。

下一章我们详细分析客户机中可能触发IOTLB刷新的几种情况。第四章讲到Hypervisor如何处理客户机的相应情况，以及所要执行的相应检查和函数流。

# 三 客户页表管理代码分析

IOTLB刷新的本质在于页类型的变化，对应到客户机内核中，结合强哥的分析，可能有以下几种情况会影响到页类型的改变：

1. 进程创建和销毁；2. 进程中出现缺页异常；3. 程序换入换出；4. 程序申请内存和释放内存；5. 程序动态加载和释放函数库。

注：从完备性上考虑，上述这五种情况可能触发页类型的变化。而从实际打桩所得到的数据来看，触发相关超级调用的调用路径只涉及到上述所说的三种情况：1. 进程创建和销毁；2. 进程中出现缺页异常；3. 程序释放内存。

## 3.1 进程创建

新进程的创建伴随着新页表的建立，这个过程有几个典型的系统调用，例如do\_execve()，do\_fork()等。现分析常用的fork()调用。当创建进程的页表时，关键的函数树control flow如下：

copy\_process()

->copy\_mm()

->dup\_mm()

->dup\_mmap()

->arch\_dup\_mmap()

->xen\_dup\_mmap()

->xen\_pgd\_pin()

->\_\_xen\_pgd\_pin()

->\_\_xen\_pgd\_walk()

->xen\_pin\_page()

上述control flow中，很关键的函数则是\_\_xen\_pgd\_walk()。它通过page walk设置PUD，PMD，PTE三级页表所在页为只读（利用MULTI\_update\_va\_mapping置为RO）。

值得注意的是，linux会对PGD和PTE做挂载（pin），二级，三级页表并不挂载。在xen中，如果我们想validate一级页表，便必须要validate它指向的所有普通内存页；如果我们想validate顶级页表，那么必须要validate底下的所有三级，二级，一级页表。

这样的话，当xen 对顶级页表做了pin操作后，便隐含的实现对于底下所有层页表的validate操作。因此，在进程切换的时候就能减少性能花销，这是引入pin位的目的。至于具体的分析，我们先来看下进程切换：

**进程切换**

页表创建完成后，当新进程被调度上VCPU时，需要加载cr3。cr3寄存器的加载是在进程调度的时候更新的，具体如下：

schedule()

->context\_switch()

->switch\_mm()

->load\_cr3(next->pgd)

-> write\_cr3()

->xen\_write\_cr3()

load\_cr3加载的是mm\_struct->pgd，即线性地址，而实际上加裁到cr3寄存器的是实际的物理地址，write\_cr3(\_\_pa(pgdir))。在装载cr3寄存器时将线性地址通过\_\_pa转换成了物理地址了，所以cr3寄存器是装的是伪物理地址。通过HYPERVISOR\_mmuext\_op()，参数是MMUEXT\_NEW\_BASEPTR, guest physical frame number of the PGD (L3)， 设置好cr3。

在这里，如果PGD在进程创建时已经被挂载（pin），那么会对PGD有一次额外的\_\_get\_page\_type()操作，使它的type\_ref\_count加1，这样当该进程被换下时，其type\_ref\_count并不为零。从而当该进程被换上后，由于PGD的type\_ref\_count不为0，就可以较快地完成\_\_get\_page\_type()里一系列的检查了，从而提高效率。

## 3.2 进程销毁

进程的销毁，伴随着页表的回收，例如利用系统调用do\_exit()中止一个进程。在这个过程中，需要对所有的页表解除挂载。

do\_exit()

->exit\_mm()

->mmput()

->xen\_exit\_mmap()

->xen\_pgd\_unpin()

->\_\_xen\_pgd\_unpin()

->\_\_xen\_pgd\_walk()

->xen\_unpin\_page()

在进程销毁过程中，我们需要销毁进程的mm\_struct，同时释放所有占用的资源。所以在实际操作时，内核会先unpin GPD，然后循环消除各个页表/页目录的只读属性，让他们变为可写页。这里与之前进程创建时挂载页表的操作几乎一致，只不过都是反向操作。

## 3.3 缺页中断的处理

当进程运行时出现缺页中断时，会调用中断处理函数do\_page\_fault()，在这个过程中，如果有需要会分配新的PTE，并调用do\_mmuext\_op超级调用挂载（pin）这个页表项。具体流程如下：

do\_page\_fault()

->\_\_do\_page\_fault()

->handle\_mm\_fault()

->\_\_handle\_mm\_fault()

->\_\_pte\_alloc()

->pmd\_populate()

->paravirt\_alloc\_pte()

->xen\_alloc\_pte()

->xen\_alloc\_ptpage()

->\_\_pin\_pagetable\_pfn()

->handle\_pte\_fault()

->\_\_handle\_page\_fault()

我们可以看到，在缺页中断的处理流程中，在经过一系列的检验之后，如果缺页中断发生在有效的内存区域中，会调用handle\_mm\_fault()。在这个函数中，如果该页PTE并没有分配，那么调用\_\_pte\_alloc()函数，分配出PTE，然后再用handle\_pte\_fault()处理这个中断。而在分配PTE的过程中，xen要求必须要对这个新增的PTE页表进行pin操作，也会触发IOTLB刷新。

## 3.4 程序申请内存

进程通过sys\_brk()申请内存时的函数树并没有涉及到do\_mmuext\_op()。在我看来，Linux一般处理内存申请的方法应该只是在内存地址空间建立数据结构，而具体分配内存，设置每级的页表项应该是在真正访问这部分内存的时候在handle\_page\_fault()中调用，从而更新页表，所以这里没有do\_mmuext\_op()是正常的。

## 3.5 程序释放内存

当一个用户空间应用程序通过malloc()申请内存时，它会调用sys\_brk()系统调用扩大自己进程的数据段边界。同时，在这部分内存被释放的时候，系统的内存管理机制也会记录释放信息，在之后会调用sys\_brk()去缩小该进程的数据段边界。在我们的调研中，sys\_brk()缩小数据段边界的过程中，可能会调用到do\_mmuext\_op()中的unpin操作，具体的函数调用树如下：

sys\_brk()

->do\_munmap()

->unmap\_region()

->free\_pgtables()

->free\_pgd\_range()

->free\_pud\_range()

->free\_pmd\_range()

->free\_pte\_range()

->pte\_free\_tlb()

->\_\_pte\_free\_tlb()

->paravirt\_release\_pte()

->pvops.release\_pte

->xen\_release\_pte()

->xen\_release\_ptpage()

->\_\_pin\_pagetable\_pfn(MMUEXT\_UNPIN\_TABLE)

这个调用树大致就是如果内核在处理sys\_brk()的时候发现是brk\_shrinking的操作，那么调用do\_munmap()去清除这部分数据段的mapping，在这个过程中逐级遍历并释放需要释放的页表。如果发现需要删除整张PTE的时候，那么便调用pvops中的release\_pte()。而xen在处理这个操作的时候需要首先把这个页表卸载（unpin），之后把这个页的类型置为可写页。

# 四 do\_mmuext\_op分析

我们知道NEWBASEPTR在放到cr3之前，要对每级页表进行验证。接下来我们以PTE页表的绑定（op.cmd=PIN\_L1\_TABLE）为例，说明Xen-4.2.3在调用iommu\_unmap\_page()刷新IOMMU的IOTLB之前对客户页表所做的检查。

🡪do\_mmuext\_op()

1. 清空old\_guest\_table，如果该清空操作被抢占，则打断超级调用执行，并且调用hypercall\_create\_continuation在下次调用时继续操作；
2. 检查mmuext\_op的参数count（总共需要做的mmuext\_op操作数）是不是等于MMU\_UPDATE\_PREEMPTED。如果是的话，那么这次调用只是在某次mmuext\_op调用后，为了保险起见再调用一次put\_old\_guest\_table()，所以只需直接返回之前的返回值；
3. 检查参数count的最高有效位，分析这是不是一次被打断之后又继续的超级调用。如果是的话，清除掉这个标记；
4. 检查指针和系统设置，uops指针是否可访问，该domain是否存在，是否开启分页等等；
5. for(i=0;i<count;i++) //循环遍历，分别处理没有mmuext\_op请求
   1. 检查old\_guest\_table是否为空，如果不为空直接打断这次操作(调用hypercall\_create\_continuation)；
   2. 访问uops指针，提取具体操作的struct mmuext\_op，如果失败返回EFAULT；
   3. switch判断mmuext\_op的操作类型如果是MMUEXT\_PIN\_L1\_TABLE操作，那么作如下操作：
      1. 通过xen传过来的GMFN获取MFN信息（物理页的page\_info）；

5.3.2 对这个页做一次额外的get\_page\_type\_preemptible()；

🡪get\_page\_type\_preemptible(page, PGT\_L1\_page\_table)

🡪\_\_get\_page\_type()

该函数是页类型更改的重要函数。具体检查流程参考do\_mmuext\_op.jpg。

在该函数中存在这样一个安全检查，当前页的页类型计数为0，且当前页为可写页，而新类型不是可写页（在我们的case里为一级页表），那么此时需调用iommu\_unmap\_page()去删除I/O页表中对于该内存页的相应表项，从而防止DMA攻击。

🡪iommu\_unmap\_page()

1. 检查iommu是否开启，以及iommu\_ops是否为空；
2. 调用iommu\_ops.umap\_page；

🡪intel\_iommu\_unmap\_page()

1. 检查如果是dom0并且iommu\_passthrough开启的话，那么不用做任何操作，直接return 0；
2. 调用dma\_pte\_clear\_one()，清理掉I/O页表中这个页的页表项；

🡪dma\_pte\_clear\_one()

1. 找到这个内存地址对应的I/O页表中pte的页表地址。如果找不到那么返回。
2. 如果找到了pte，接着Xen继续执行iommu\_dont\_flush\_iotlb()，如果为false，那么调用\_\_intel\_iommu\_iotlb\_flush()。

🡪\_\_intel\_iommu\_iotlb\_flush(domain, addr>>PAGE\_SHIFT\_4K,0,1)

1. 遍历DRHD，对于每个iommu的I/O设备分别操作
   1. 首先检查现在正在操作的这个domain是不是存在于这个设备的domid\_map里，即该domain有没有这个设备的权限；
   2. 调用iommu\_flush\_iotlb\_psi(iommu, iommu\_domid, (paddr\_t)gfn<<PAGE\_SHIFT\_4K, 0, 1, flush\_dev\_iotlb)；

🡪iommu\_flush\_iotlb\_psi()

1. 检查输入的地址与页框对齐；
2. 检查该设备的iommu是否支持psi（page selective invalidation），如果不支持，那么做domain selective flush，调用iommu\_flush\_tlb\_dsi() ，最终调用flush\_iotlb\_qi()（注：在实际的路径中，2，3这两种情况都有，根据设备的不同）;
3. 调用flush\_iotlb\_qi()；

🡪flush\_iotlb\_qi()

okay，就是这里，由于之前在调用\_\_intel\_iommu\_iotlb\_flush时参数设置错误，最终iotlb根本没刷新就返回了。