vt实现ssdt hook

测试环境：win7 64位 6.1.7600

VT概述：

VT技术的大致含义如上。一个CPU上有vm monitor和guest。在guest中感受不到自己跑在虚拟环境中。执行普通的操作时在guest中的操作和未开启vt时的操作并没有什么不同，但是在执行一些特殊的指令或者进行一些特殊的操作，如执行cpuid指令、切换cr3指令、读写msr寄存器指令、触发异常时，会将控制权交还给vm monitor，通过vm monitor确定应该如何让该指令或者操作得到执行。

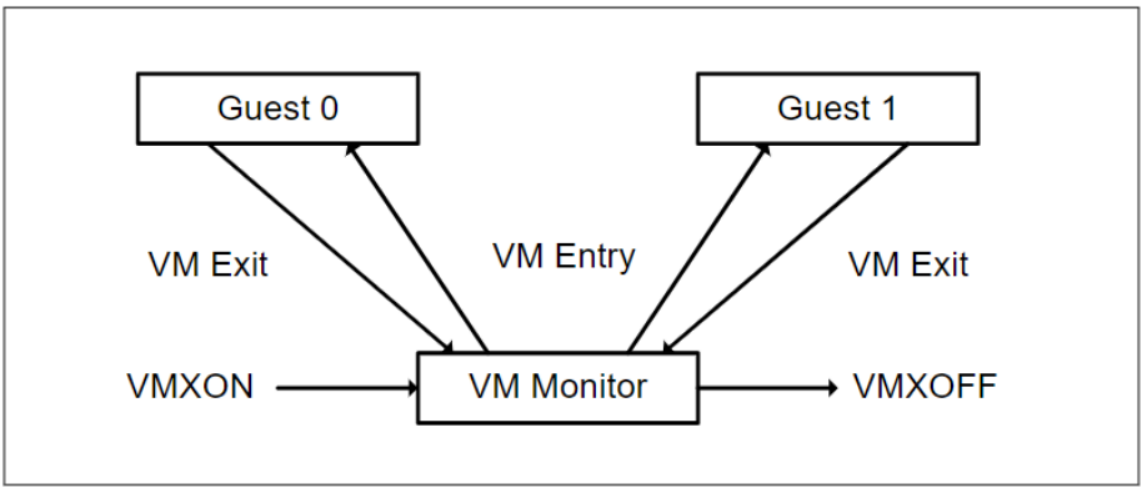
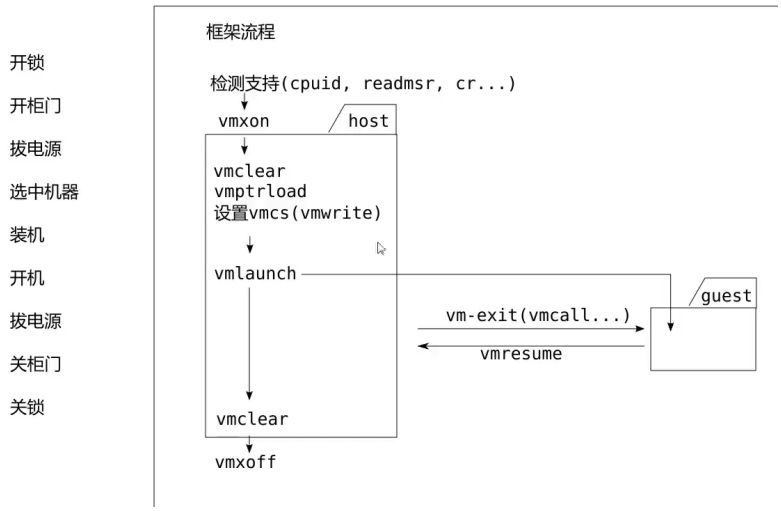


Figure 23-1. Interaction of a Virtual-Machine Monitor and Guests

VT执行的大体流程

- VT的开启与关闭大致分为开锁、开柜门、拔电源、选中机器、装机、开机、拔电源、关柜门、关锁9个步骤。这9个步骤的具体描述如下。此处为大体的描述，在后面会对每一个步骤的具体细节进行具体的说明。



1. 开锁：检测是否支持VT
2. 开柜门：vmxon。需要申请一块内存然后把这块内存的lowpart和highpart作为参数调用vmxon。注意这块内存的开头要填入一个特定的msr寄存器中的值。注意这里vmxon修改cr4寄存器之后，cr0寄存器的页保护和段保护位都不能再置位，如果试图修改cr0的页保护或者段保护位，会触发异常蓝屏。
3. 拔电源：vmclear
4. 选中机器：vmptrload，相当于一个指针。需要通过vmptrload来指向某个机器。如果当前有多个机器的话，在各个机器之间的切换操作也需要用到vmptrload。
5. 装机：设置vmcs。也是要申请一块内存，然后在这块内存里面填入开启虚拟机时虚拟机使用的一些寄存器。要注意这里不能直接操作这块内存区域，要用vmwrite来操作。因为随着版本的更新，原来在这块内存区域的东西可能后来就不在了。vmwrite保证兼容性。
6. 开机vmlaunch
7. 拔电源vmclear
8. 关柜门vmoff
9. 关锁

开锁

1. 通过cpuid指令查询CPU是否支持VT
2. 通过一些特定的MSR寄存器（在31.5.1中有介绍）来确认是否支持开启VT。
3. 申请一块非分页内存，用来存放vmxon区域。其大小在IA32_VMX_BASIC这个寄存器中指定。该内存要4kb对齐（后12位全为0）。实际申请时申请一个页的大小的非分页内存即可。
4. 初始化vmxon区域的版本编号（前4个字节）。版本编号通过IA32_VMX_BASIC这个寄存器的低4字节获取。实际实验时发现这个数为1，理论上把上一步申请到的内存前四字节置1即可，但为了兼容最好还是从寄存器中取值出来。
5. 给cr0和cr4寄存器的特定位置置位，使得其满足如下要求
   - IA32_VMX_CR0_FIXED0 寄存器中为1的位，在cr0中必须为1
   - `IA32_VMX_CR0_FIXED1` 寄存器中为0的位，在cr0中必须为0

    - `IA32_VMX_CR4_FIXED0` 寄存器中为1的位，在cr4中必须为1
    - IA32_VMX_CR4_FIXED1 寄存器中为0的位，在cr4中必须为0
6. 确认IA32_FEATURE_CONTROL寄存器被正确置位。该寄存器的具体细节可以通过《处理器虚拟化技术》2.3.2.1找到。也就是说，该寄存器的bit0是lock为，bit2是outside位。必须保证这两位均为1才可以开启VT。可以大致通过读取该寄存器后将其&5，判断是否两个位均为1。
7. 执行vmxon指令，传进去的参数是一个指向存放vmxon区域的物理地址的指针。如果该指令成功，那么rflags.cf=0，否则rflags.cf=1。

vmxon（开柜门）

- 申请一块4KB对齐的非分页内存，作为vmxon的参数。
- 设置vmxon区域。这一块的说明在intel白皮书24.11.5中。这一块内存是操作系统使用的。在申请区域之后要rtlzeromemory一下，防止没有挂页。其他位都置零即可，之后操作系统进行虚拟机的管理的时候会使用到这块内存。我们需要设置的只有头四个字节。要把msr中 `IA32_VMX_BASIC` 这个寄存器的低4字节填入vmxon的头4字节。否则vmxon的执行会出错。基本上这个寄存器的低4字节会是1。
- 在设置完vmxon区域之后，还需最后一步即可vmxon。需要根据msr寄存器中的指示进行cr0和cr4寄存器的设置。

- 简单解释就是在进行vmxon之前，
- IA32_VMX_CR0_FIXED0 寄存器中为1的值，在cr0中必须为1、IA32_VMX_CR0_FIXED1 寄存器中为0的值，在cr0中必须为0。cr4同理。那么可以通过 cr0 = cr0 | IA32_VMX_CR0_FIXED0 & IA32_VMX_CR0_FIXED1 操作将cr0和cr4进行置位。

申请vmcs内存并vmclear（拔电源）

- 首先申请vmcs内存。和vmxon内存类似，这一块内存的头4个字节也要填入 `IA32_VMX_BASIC` 的低4个字节。vmcs中保存的是虚拟机的各种寄存器和控制区域、vmexit控制区域等。在装机过程中要通过vmwrite进行vmcs的设置。
- 申请vmcs内存并设置头4字节为 `IA32_VMX_BASIC` 的低4字节之后，通过vmclear指令进行初始化。传入的参数与vmxon的参数类似，是指向申请的内存的物理地址的一个指针。

vmptrload（选中机器）

- `vmptrload` 指令和vmxon、vmclear类似，传入的是指向vmcs结构的物理地址的一个指针。只有执行了这行指令之后才能通过 `vmwrite` 进行vmcs结构的填写。

关锁

关锁就是把之前开锁设置的cr0和cr4等等还原，vmoff关闭vt

设置vmcs（装机）

大体包括如下几部分

1. Guest state fields
2. Host state fields
3. vm-control fields
    a. vm execution control
    b. vm exit control
    c. vm entry control

除这5个区域之外，vmcs还有一个区域是vm exit信息区域。这个区域是只读的，存储的是vmx指令失败后失败代码的编号。

设置vmcs控制区的guest和host区域

对于guest区域和host区域，需要填充的字段如下：

长度为16位的字段:

guest-state 字段	guest ES selector	00000800H
	guest CS selector	00000802H
	guest SS selector	00000804H
	guest DS selector	00000806H
	guest FS selector	00000808H
	guest GS selector	0000080AH
	guest LDTR selector	0000080CH
	guest TR selector	0000080EH
	guest interrupt status	00000810H
host-state 字段	host ES selector	00000C00H
	host CS selector	00000C02H
	host SS selector	00000C04H
	host DS selector	00000C06H
	host FS selector	00000C08H
	host GS selector	00000C0AH
	host TR selector	00000C0CH

长度为64位的字段

guest-state 字段	VMCS link pointer（full）	00002800H
	VMCS link pointer（high）	00002801H
	guest IA32_DEBUGCTL（full）	00002802H
	guest IA32_DEBUGCTL（high）	00002803H
	guest IA32_PAT（full）	00002804H
	guest IA32_PAT（high）	00002805H
	guest IA32_EFER（full）	00002806H
	guest IA32_EFER（high）	00002807H
	guest IA32_PERF_GLOBAL_CTRL（full）	00002808H
	guest IA32_PERF_GLOBAL_CTRL（high）	00002809H
	guest PDPTE0（full）	0000280AH
	guest PDPTE0（high）	0000280BH
	guest PDPTE1（full）	0000280CH
	guest PDPTE1（high）	0000280DH
	guest PDPTE2（full）	0000280EH
	guest PDPTE2（high）	0000280FH
	guest PDPTE3（full）	00002810H
	guest PDPTE3（high）	00002811H
host-state 字段	host IA32_PAT（full）	00002C00H
	host IA32_PAT（high）	00002C01H
	host IA32_EFER（full）	00002C02H
	host IA32_EFER（high）	00002C03H
	host IA32_PERF_GLOBAL_CTRL（full）	00002C04H
	host IA32_PERF_GLOBAL_CTRL（high）	00002C05H

长度为32位的字段

guest-state 字段	guest ES limit	00004800H
	guest CS limit	00004802H
	guest SS limit	00004804H
	guest DS limit	00004806H
	guest FS limit	00004808H
	guest GS limit	0000480AH

类　型	字 段 名	ID 值
guest-state 字段	guest LDTR limit	0000480CH
	guest TR limit	0000480EH
	guest GDTR limit	00004810H
	guest IDTR limit	00004812H
	guest ES access right	00004814H
	guest CS access right	00004816H
	guest SS access right	00004818H
	guest DS access right	0000481AH
	guest FS access right	0000481CH
	guest GS access right	0000481EH
	guest LDTR access right	00004820H
	guest TR access right	00004822H
	guest interruptibility state	00004824H
	guest activity state	00004826H
	guest SMBASE	00004828H
	guest IA32_SYSENTER_CS	0000482AH
	VMX-preemption timer value	0000482EH
host-state 字段	host IA32_SYSENTER_CS	00004C00H

可以看到基本上都是进入guest区域之后的寄存器。其中对段寄存器的填充尤其麻烦，需要分别填充base、attribute、limit、selector。因此需要手动获取段寄存器并将其进行拆分然后进行相应的填充。

- 填充guest和host区域时还有四个重要的字段需要获取。分别是进入guest区域之后的rip和rsp以及从guest返回到host区域之后的rip rsp。这里我们要使得进入guest区域之后系统仍然正常进行，还是从原来的地方往下跑。因此

要通过函数获取需要返回的上一层函数的返回地址以及rsp。而对于返回host区域之后的host eip，由于从虚拟机中返回一定是发生了vmexit事件，需要对该事件进行处理，因此从虚拟机中返回之后的rip一定要设置为vmexit事件的处理函数。而rsp则需要重新开辟一块内存区域供vmexit事件处理函数使用。如果还是使用guest返回之前的堆栈，则会破坏堆栈中的内容，导致无法预知的结果。

vm-entry control字段

- 在《处理器虚拟化技术》3.6章节中详细讲解了vm-entry控制类字段的填写与对应的属性。

（1）VM-entry control 字段。
（2）VM-entry MSR-load count 字段。
（3）VM-entry MSR-load address 字段。
（4）VM-entry interruption-information 字段。
（5）VM-entry exception error code 字段。
（6）VM-entry instruction length 字段。

- 需要填写的字段如下在vm-entry时，如果CPU检查到这些字段没有被正确填写，将会抛错并退出。

VM_ENTRY_CONTROLS字段

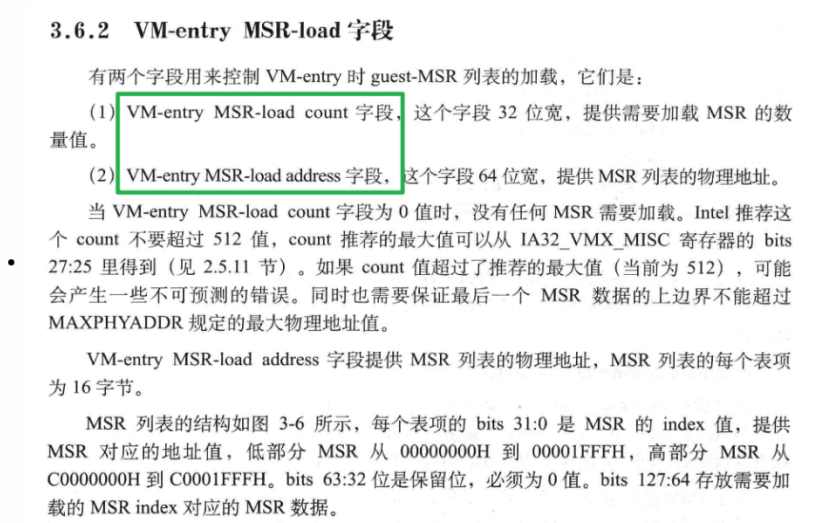
- 这个字段的长度为32位，每个位对应一个控制功能。其控制的是进入虚拟机时处理器所进行的一些操作。如是否在进入虚拟机时加载dr0~dr7寄存器、是否加载时进入IA-32e模式、是否加载IA32_PERF_GLOBAL_CTRL、IA32_PAT、IA32_EFER寄存器等。具体位的作用如书中表3-9所示。

表 3-9

位域	控 制 名	配置	描　述
1:0	保留位	1	固定为 1 值
2	load debug controls	0 或 1	为 1 时，加载 debug 寄存器
8:3	保留位	1	固定为 1 值
9	IA-32e mode guest	0 或 1	为 1 时，进入 IA-32e 模式
10	entry to SMM	0 或 1	为 1 时，进入 SMM 模式
11	deactivate dual-monitor treatment	0 或 1	为 1 时，返回 executive monitor，关闭 SMM 双重监控处理
12	保留位	1	固定为 1 值
13	load IA32_PERF_GLOBAL_CTRL	0 或 1	为 1 时，加载 IA32_PERF_GLOBAL_CTRL
14	load IA32_PAT	0 或 1	为 1 时，加载 IA32_PAT
15	load IA32_EFER	0 或 1	为 1 时，加载 IA32_EFER
31:16	保留位	0	固定为 0 值

- 在查看这个表后可以注意到，有一些位置被固定为了1，有一些位置被固定为了0。这些位置有些是还未被使用的，用于将来拓展其他功能的时候添加的。这些位可能在将来将不再固定为1或0，而是用于控制某个新推出的功能。因此不能把固定为0或者固定为1的位写死填进去。这里需要根据一个算法来算出固定为0和固定为1的位，并填入 `VM_ENTRY_CONTROLS` 寄存器中。

MSR-load字段



3.6.2 VM-entry MSR-load 字段

有两个字段用来控制 VM-entry 时 guest-MSR 列表的加载，它们是：

（1）VM-entry MSR-load count 字段，这个字段 32 位宽，提供需要加载 MSR 的数量值。

（2）VM-entry MSR-load address 字段，这个字段 64 位宽，提供 MSR 列表的物理地址。

当 VM-entry MSR-load count 字段为 0 值时，没有任何 MSR 需要加载。Intel 推荐这个 count 不要超过 512 值，count 推荐的最大值可以从 IA32_VMX_MISC 寄存器的 bits 27:25 里得到（见 2.5.11 节）。如果 count 值超过了推荐的最大值（当前为 512），可能会产生一些不可预测的错误。同时也需要保证最后一个 MSR 数据的上边界不能超过 MAXPHYADDR 规定的最大物理地址值。

VM-entry MSR-load address 字段提供 MSR 列表的物理地址，MSR 列表的每个表项为 16 字节。

MSR 列表的结构如图 3-6 所示，每个表项的 bits 31:0 是 MSR 的 index 值，提供 MSR 对应的地址值，低部分 MSR 从 00000000H 到 00001FFFH，高部分 MSR 从 C0000000H 到 C0001FFFH。bits 63:32 位是保留位，必须为 0 值。bits 127:64 存放需要加载的 MSR index 对应的 MSR 数据。

- 这两个字段用来控制进入虚拟机的时候是否需要加载msr寄存器。这里我们并不需要其在进入虚拟机的时候加载msr寄存器。因为vm的退出和加载是很频繁的。如果每次都加载一遍msr寄存器会降低性能。而且如果我们想要对msr寄存器进行拦截或者hook，还有另外的方法可以进行。因此这两个字段均填为0即可。

vm-entry control字段填写总结

- vm-entry control用来控制进入虚拟机时候的操作。

vm-exit control字段

- vm-exit control字段和vm-entry字段很类似。用来规定退出虚拟机时需要进行的操作。vm-entry时候进行的操作和vm-exit时候进行的操作可以对应起来。vm-exit的时候对msr寄存器进行保存，那么vm-entry的时候就可以加载msr寄存器。在填写字段以及进行功能的设置的时候应注意要将进入虚拟机时的操作和退出虚拟机时的操作对应起来。

vm-execution control字段

- 这是处理操作中最重要的字段，用来设置拦截哪些事件不拦截哪些事件。我们想要对虚拟机中的一些事件进行监控就要对这个字段进行相应的设置。对于vm-execution控制类字段，在《处理器虚拟化技术》3.5章节中有详细的解析。

无条件退出的虚拟机事件

大体框架

- 在《处理器虚拟化技术》3.10.1.2中，描述了vm-exit产生的原因。其中指出了会无条件导致vmexit事件产生的指令，如下图所示可见，

无条件导致 VM-exit 的指令

在 VMX non-root operation 模式中，执行所有 VMX 指令（除 VMFUNC 指令外），CPUID，GETSEC，INVD 及 XSETBV 指令将无条件地导致 VM-exit 发生。

但是，在启用 unrestricted guest 功能的前提下，如果 **guest 运行在实模式**（CR0.PE = 0），执行所有 VMX 指令（除 VMFUNC 指令外）将产生**#UD** 异常，而不是 VM-exit。

- 在虚拟机中，执行所有除vmfunc之外的指令时，都会无条件导致vmexit事件的发生。除此之外，`cpuid`、`getsec`、`invd`、`xsetbv`指令也会无条件导致vmexit事件的发生。那么我们需要进行处理的无条件退出事件如下

18	尝试执行 VMCALL 指令导致 VM-exit
19	尝试执行 VMCLEAR 指令导致 VM-exit
20	尝试执行 VMLAUNCH 指令导致 VM-exit
21	尝试执行 VMPTRLD 指令导致 VM-exit
22	尝试执行 VMPTRST 指令导致 VM-exit
23	尝试执行 VMREAD 指令导致 VM-exit
24	尝试执行 VMRESUME 指令导致 VM-exit
25	尝试执行 VMWRITE 指令导致 VM-exit
26	尝试执行 VMXOFF 指令导致 VM-exit
27	尝试执行 VMXON 指令导致 VM-exit
10	尝试执行 CPUID 指令导致 VM-exit
11	尝试执行 GETSEC 指令导致 VM-exit
13	尝试执行 INVD 指令导致 VM-exit
55	尝试执行 XSETBV 指令而导致 VM-exit

- 在3.4.3 32位字段ID中，可以找到控制区中对应的vmexit信息字段。

只读字段	VM-instruction error
	Exit reason
	VM-exit interrupt information
	VM-exit interrupt error code
	IDT-vectoring information field
	IDT-vectoring error code
	VM-exit instruction length
	VM-exit instruction information

- 可以找到退出原因、导致退出的指令的长度、信息。
  - 只读字段里面的vm-instuction error字段在vm指令失败的时候会被设置，可以从中读取出fail事件发生的原

因。在《处理器虚拟化技术》2.6.3章节中有vmfailvalid事件发生的原因的编号表。

- exit-reason 表示导致vm-exit事件发生的原因
- vm-exit instruction length 表示导致vm-exit事件发生的指令的长度。通过这个字段可以在进行进一步处理的时候更加方便。在我们对产生vmexit的事件进行处理之后，我们不可能重新跳回发生vmexit事件的地方继续执行。比如vmexit事件是由于cpuid指令导致的，那么在对这个事件进行处理之后，就要跳过原来的cpuid，继续执行下一条指令，而不是回去之后仍然执行cpuid对应的那条指令。当然，也不是所有产生vmexit事件的时候都需要跳过原来执行的指令。比如缺页之类的原因造成的vmexit事件，就不能跳过产生vmexit事件的指令。不过大多事件都需要跳过执行的指令。
- vm-exit instruction information 这个字段在以后会用到，其存储的是指令详细信息，以后用到的时候再进行说明。

- 在《处理器虚拟化技术》3.10.1.1中，列出了exit reason字段的组成。如下图

exit reason 字段由几个位域组成，如表 3-23 所示。

表 3-23

位　域	描　述
15:0	保存 VM 退出原因值
27:16	保留位（为 0）
28	为 1 时，指示 SMM VM-exit 时，存在 pending MTF VM-exit 事件
29	为 1 时，指示 SMM VM-exit 从 VMX root-operation 中产生
30	保留位（为 0）
31	为 1 时，表明是在 VM-entry 过程中引发了 VM-exit

- 可见，0~15位表明了vm退出的原因，其他位还有其他的指示作用。因此这里应该将其他位取出，只通过0~15位进行vm退出原因的判断。

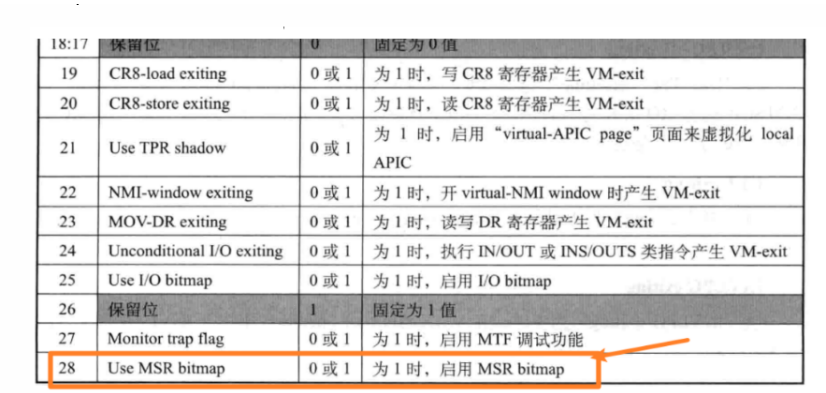
处理退出：

1. 获取指令长度、指令信息、EIP ESP
2. 获取事件码
3. 对事件进行相应的处理
4. rip+=指令长度
5. 将rip和rsp写回并返回，回到虚拟机中产生vm-exit事件的指令的下一条指令处继续执行。

有条件退出的虚拟机事件

对msr寄存器的有条件拦截

- 在上一篇文章中，我们提到了processor-based vm-execution control字段。对这个字段的设置可以让执行一些特定指令或者读写某些特定寄存器的时候产生vm-exit事件。上一篇文章中我们给这个字段填为全0，并未对其进行相应设置。现在我们对其进行一定的设置。如图，可以看到该寄存器的第28位为1的话，则表示将会启动MSR bitmap。



18:17	保留位	0	固定为 0 值
19	CR8-load exiting	0 或 1	为 1 时，写 CR8 寄存器产生 VM-exit
20	CR8-store exiting	0 或 1	为 1 时，读 CR8 寄存器产生 VM-exit
21	Use TPR shadow	0 或 1	为 1 时，启用“virtual-APIC page”页面来虚拟化 local APIC
22	NMI-window exiting	0 或 1	为 1 时，开 virtual-NMI window 时产生 VM-exit
23	MOV-DR exiting	0 或 1	为 1 时，读写 DR 寄存器产生 VM-exit
24	Unconditional I/O exiting	0 或 1	为 1 时，执行 IN/OUT 或 INS/OUTS 类指令产生 VM-exit
25	Use I/O bitmap	0 或 1	为 1 时，启用 I/O bitmap
26	保留位	1	固定为 1 值
27	Monitor trap flag	0 或 1	为 1 时，启用 MTF 调试功能
28	Use MSR bitmap	0 或 1	为 1 时，启用 MSR bitmap

当Use MSR bitmap位为1时：

- MSR bitmap区域为4KB大小（一个页大小）
- 0~1KB：控制编号范围为 00000000~00001fff 的MSR寄存器的读访问是否会产生vm-exit事件。如果对应位为1，则读取该MSR寄存器会产生vm-exit事件
- 1~2KB：控制编号范围为 C0000000~C0001FFF 的MSR寄存器的读访问是否会产生vm-exit事件。如果对应位为1，则读取该MSR寄存器会产生vm-exit事件
- 2~3KB：控制编号范围为 00000000~00001fff 的MSR寄存器的写访问是否会产生vm-exit事件。如果对应位为1，则写入该MSR寄存器会产生vm-exit事件
- 3~4KB：控制编号范围为 C0000000~C0001FFF 的MSR寄存器的写访问是否会产生vm-exit事件。如果对应位为1，则写入该MSR寄存器会产生vm-exit事件

Msr HOOK欺骗PG

- 所谓MSR HOOK下的SSDT HOOK, 就是接管当前的系统调用流程，然后替换其中的SSDT表或函数地址。大体操作就是syscall指令执行时会获取 MSR_LSTAR(KiSystemCall64 地址) 中的值作为RIP，这里只以Win7为例), 我们就可以修改其值指向我们的系统调用流程，代码如下：

```
// Hook Msr Lstar 寄存器
bool VtSsdtHook::VtHookMsrLstar()
{
        //...
        // HOOK LSTAR
        __writemsr(MSR_LSTAR, (ULONG_PTR)Win7SysCallEntryPointer); // 修改 SysCall Rip 流程

        //...
        return true;
}
```

- 然后当有人读取MSR_LSTAR值时候，我们在VT中返回其KiSystemCall64原有地址即可(达到欺骗效果)，代码如下:

```
// 处理读取 MSR VM-EXIT
EXTERN_C
VOID MsrReadVtExitHandler(ULONG_PTR * Registers)
{
        ULONGLONG MsrValue = __readmsr((ULONG)Registers[R_RCX]);

        switch (Registers[R_RCX])
        {
        case MSR_LSTAR: // 读取 MSR RIP
        {
                //KdBreakPoint();
                if (KiSystemCall64Pointer) {
                        MsrValue = (ULONG_PTR)KiSystemCall64Pointer; // SSDT HOOK
                }
        }
        default:
        {
                // 默认正常流程
                Registers[R_RAX] = LODWORD(MsrValue);
                Registers[R_RDX] = HIDWORD(MsrValue);
        }
        break;
        }

        // 走默认流程
        DefaultVmExitHandler(Registers);

        return VOID();
}
```

- 然后在我们自写的系统调用流程中替换SSDT函数地址即可:

```
; ************************复写 Windows7 syscall************************
KiSystemServiceStart PROC
        mov     [rbx+1D8h], rsp
; CurrentThread.TrapFrame = Rsp(当前_KTRAP_FRAME 结构)
```

```
        mov     edi, eax                                                ; Rdi = 系统服务号
        shr     edi, 7                                                  ; Rdi = 系统服务号 >> 7
        and     edi, 20h                                                ; Rdi = (系统服务号 >> 7) & 0x20 = 服务表的索引
        and     eax, 0FFFh                                              ; EAX = 系统服务号
KiSystemServiceStart ENDP
; ***************************************************

; ********************复写 Windows7 syscall**********************
KiSystemServiceRepeat PROC
        ; RAX = [IN ] syscall index
    ; RAX = [OUT] number of parameters
    ; R10 = [OUT] func address
    ; R11 = [I/O] trashed

        lea             r11, offset g_ssdt_table                        ; 切换 ssdt 表
        mov             r10, qword ptr [r11 + rax * 8h]         ; 获取Hook函数地址

        lea             r11, offset g_param_table                       ;
        movzx   rax, byte ptr [r11 + rax]
; RAX = paramter count

        jmp             [KiSystemServiceCopyEndPointer]         ; 跳回原函数流程 (不能 Hook 五个参数以上的函数)
KiSystemServiceRepeat ENDP
; ***************************************************
```