关于hypercall

调研EPT代码时发现，当利用xc\_domain\_create()创建HVM domain时，会调用do\_domctl(xch, &domctl))，但是xen代码中还有个do\_domctl(XEN\_GUEST\_HANDLE\_PARAM(xen\_domctl\_t) u\_domctl)，这两个函数是什么关系呢？

首先do\_domctl(xch, &domctl))会调用do\_xen\_hypercall（），这应该是一个domain0应用层调用hypercall服务的接口，do\_xen\_hypercall继续执行：

int do\_xen\_hypercall(xc\_interface \*xch, privcmd\_hypercall\_t \*hypercall)

{

return xch->ops->u.privcmd.hypercall(xch, xch->ops\_handle, hypercall);

}

Xc\_linux\_osdep.c (tools\libxc)文件中，（pc\_cpus\_init🡪cpu\_x86\_init🡪 xc\_evtchn\_open

xc\_evtchn\_open🡪 xc\_interface\_open\_common🡪 xc\_osdep\_get\_info）

xc\_osdep\_info\_t xc\_osdep\_info = {

.name = "Linux Native OS interface",

.init = &linux\_osdep\_init,

.fake = 0,

};

linux\_osdep\_init中

static struct xc\_osdep\_ops \*linux\_osdep\_init(xc\_interface \*xch, enum xc\_osdep\_type type)

{

switch ( type )

{

case XC\_OSDEP\_PRIVCMD:

return &linux\_privcmd\_ops;

case XC\_OSDEP\_EVTCHN:

return &linux\_evtchn\_ops;

case XC\_OSDEP\_GNTTAB:

return &linux\_gnttab\_ops;

case XC\_OSDEP\_GNTSHR:

return &linux\_gntshr\_ops;

default:

return NULL;

}

}

static struct xc\_osdep\_ops linux\_privcmd\_ops = {

.open = &linux\_privcmd\_open,

.close = &linux\_privcmd\_close,

.u.privcmd = {

.alloc\_hypercall\_buffer = &linux\_privcmd\_alloc\_hypercall\_buffer,

.free\_hypercall\_buffer = &linux\_privcmd\_free\_hypercall\_buffer,

.hypercall = &linux\_privcmd\_hypercall,

.map\_foreign\_batch = &linux\_privcmd\_map\_foreign\_batch,

.map\_foreign\_bulk = &linux\_privcmd\_map\_foreign\_bulk,

.map\_foreign\_range = &linux\_privcmd\_map\_foreign\_range,

.map\_foreign\_ranges = &linux\_privcmd\_map\_foreign\_ranges,

},

};

.hypercall = &linux\_privcmd\_hypercall，继续查看

static int linux\_privcmd\_hypercall(xc\_interface \*xch, xc\_osdep\_handle h, privcmd\_hypercall\_t \*hypercall)

{

int fd = (int)h;

return ioctl(fd, IOCTL\_PRIVCMD\_HYPERCALL, hypercall);

}

查看ioctl定义发现int ioctl(int fd, int request, ...);，其他就没有什么线索了。

查资料可知，xen/arch/x86/x86\_32/entry.S中定义了超级调用表，通过超级调用号索引就可以方便的找到对应的处理函数。超级调用页：超级调用页是Xen为Guest OS准备的一个页，可以做到不同Guest OS有不同的超级调用页内容。实现过程为**\_HYPERVISOR\_xxxx ==> 在超级调用页上找到相应的代码 ==> \_hypercall2()调用超级调用页内的代码==> 通过hypercall\_page\_initialise实现HVM、Dom0、DomU的不同跳转 ==> 调用hypercall（）来检查有效性、执行相应的服务例程(do\_xxhyper名\_)并返回。**

超级调用只能由内核来调用，而应用程序是无法直接调用的。应用程序申请超级调用的过程为：

打开Xen提供的内核驱动：/proc/xen/privcmd；通过ioctl系统调用来间接调用hypercall：

fd = open("/proc/xen/privcmd", O\_RDWR);   
privcmd\_hypercall\_t hcall = {   
\_\_HYPERVISOR\_print\_string,   
{message, 0, 0, 0, 0}   
};   
ioctl(fd, IOCTL\_PRIVCMD\_HYPERCALL, &hcall);

复杂一点的超级调用申请的过程为：（以\_HYPERVISOR\_domctl超级调用为例）

通过pyxc\_domain\_create()获取要创建的domain的相关信息；通过xc\_domain\_create()创建控制结构体变量domctl；通过do\_domctl()生成超级调用请求；传递请求到OS内核：do\_xen\_hypercall()do\_privcmd通过ioctl来完成由3环到1环的转变，并完成超级调用。

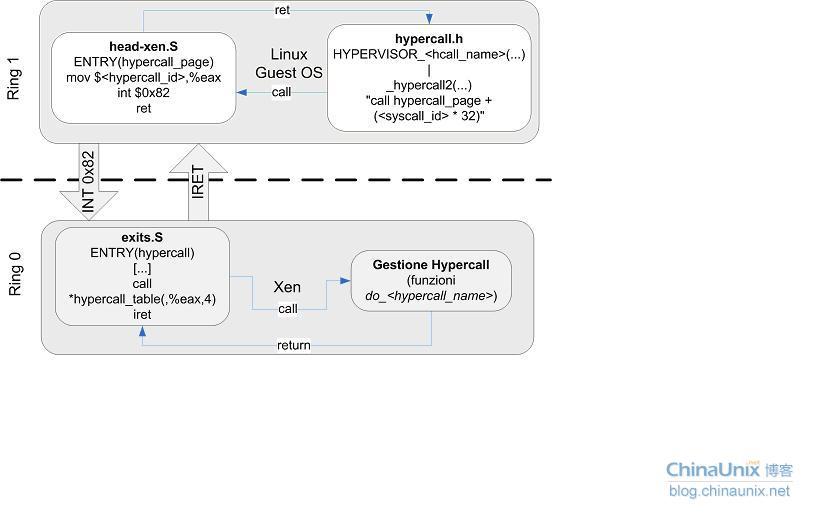
//出自论文Evolution in Kernel Debugging using Hardware Virtualization With Xen

Xen hypervisor Interactions The /proc/xen/privcmd device is implemented as a device driver in the dom0’s paravirtualized kernel. For the device, the IOCTL\_PRIVCMD\_MMAP ioctl is implemented by making a HYPERVISOR\_mmu\_update hyper-call in Xen. And like all the other hyper calls, the IOCTL\_PRIVCMD\_HYPERCALL ioctl is implemented by making a call at right the offset in the hypercall\_page.

The hypervisor\_page has the handlers for the hyper calls with various parameters. And it is initialized by the Xen hypervisor at the do-main creation time. The initialization of the hypercall\_page involves writing the appropriate code in the page for hyper call handlers. For x86\_64 domain-0 the hypercall\_page is initialized with syscall handlers; for i386 it is it is initialized with int 0x82 calls. For supervisor\_mode\_kernel i386 domain-0 kernel it is initialized with the long call instruction. All these methods get the processor execution control into the Xen hypervisor, and call the appropriate function from the hypercall\_table. After the execution of the

hypercall function, the hypervisor returns control to the next instruction after the hyper call in the dom0, by making an iret or long call, or sysret instruction.

//出自某网页

[](http://blog.chinaunix.net/attachment/201203/17/1858380_1331950135E5i2.jpg)

由以上分析可知，do\_domctl(xch, &domctl))会调用相应的服务例层

long do\_domctl(XEN\_GUEST\_HANDLE\_PARAM(xen\_domctl\_t) u\_domctl)，接着执行

case XEN\_DOMCTL\_createdomain:

d = domain\_create(dom, domcr\_flags, op->u.createdomain.ssidref);

ether扩展：

先在domctl.h中定义相关的超级调用页和相关数据结构，如下：

+#define XEN\_DOMCTL\_ether 99

+#define XEN\_DOMCTL\_ETHER\_SINGLE\_STEP 0

+#define XEN\_DOMCTL\_ETHER\_BRANCH\_STEP 1

+#define XEN\_DOMCTL\_ETHER\_SYSCALL\_STEP 2

+#define XEN\_DOMCTL\_ETHER\_DISABLE 3

+#define XEN\_DOMCTL\_ETHER\_INIT 4

+#define XEN\_DOMCTL\_ETHER\_GET\_SYSENTER 5

+#define XEN\_DOMCTL\_ETHER\_SET\_SYSENTER 6

+#define XEN\_DOMCTL\_ETHER\_ADD\_CR3 7

+#define XEN\_DOMCTL\_ETHER\_REMOVE\_CR3 8

+#define XEN\_DOMCTL\_ETHER\_READ\_GUEST 9

+#define XEN\_DOMCTL\_ETHER\_ADD\_NAME 10

+#define XEN\_DOMCTL\_ETHER\_TERMINATE 11

+#define XEN\_DOMCTL\_ETHER\_MEMWRITE 12

+#define XEN\_DOMCTL\_ETHER\_UNPACK 13

+#define XEN\_DOMCTL\_ETHER\_SS\_DETECT 14

+

+struct xen\_domctl\_ether {

+ /\* which domctl function to execute \*/

+ uint32\_t command\_code; /\* IN \*/

+ uint64\_t sysenter\_cs;

+ uint64\_t sysenter\_eip;

+ uint64\_t cr3\_value;

+ struct ether\_communication comm;

+ /\* a VA in the domain you want to read from \*/

+ uint32\_t guest\_va;

+ /\* pointer to a buffer in the caller in which to write data \*/

+ unsigned char\* guest\_buffer;

+ /\* size of buffer pointed to by guest\_buffer \*/

+ int data\_length;

+ int on\_or\_off;

+};

+

+typedef struct xen\_domctl\_ether xen\_domctl\_ether\_t;

+DEFINE\_XEN\_GUEST\_HANDLE(xen\_domctl\_ether\_t);

+

struct xen\_domctl {

uint32\_t cmd;

uint32\_t interface\_version; /\* XEN\_DOMCTL\_INTERFACE\_VERSION \*/

@@ -459,6 +502,7 @@ struct xen\_domctl {

struct xen\_domctl\_hvmcontext hvmcontext;

struct xen\_domctl\_address\_size address\_size;

struct xen\_domctl\_sendtrigger sendtrigger;

+ struct xen\_domctl\_ether ether;

uint8\_t pad[128];

} u;

};然后在应用层调用相关超级调用发，找到相应服务例层domctl.c do\_domctl中case XEN\_DOMCTL\_ether:做ether相关的功能。

类似的，参照ether功能代码结构，可以做与硬件虚拟化相关的系统调用监控，内存写监控，单步调试等功能模块，在dom0发起调用，完成一个系统。