linux内存管理中的缓存失效漏洞

Edvison / 2018-10-10 20:36:34 / 浏览数 2605 技术文章 技术文章 顶(0) 踩(0)

本文是A cache invalidation bug in Linux memory management的翻译文章。

前言

本文介绍了一种利用从内核版本3.16以来就存在的Linux内核漏洞（CVE-2018-17182）的方法。
虽然这个漏洞本身在代码中从相对较强的沙盒上下文也可以访问，但本文仅介绍在使用未配置为高安全性的Linux内核的环境中利用它的方法。
（具体来说是Ubuntu 18.04与内核linux-image-4.15.0-34-generic，版本为4.15.0-34.37）。 这将演示内核配置如何影响利用内核漏洞的难度。

漏洞报告和漏洞利用已提交到我们的issue tracker中（issue 1664）

该漏洞在较新的稳定版本4.18.9,4.14.71,4.9.128,4.4.157和3.16.58中修复。

The bug

每当发生用户空间页错误时，比如当必须根据需要对页面进行分页时，Linux内核必须查找包含故障地址的VMA（虚拟内存区域; struct vm_area_struct）以确定处理故障的措施。而查找VMA的慢路径（在find_vma()中）必须走VMA的红黑树。
为了减少性能损失，Linux还有一个快速路径，如果最近使用过VMA，它可以绕过该树来遍历。

快速路径的实现也一直在变化。从版本3.15开始，Linux使用有四个插槽的per-thread VMA缓存，在mm/vmacache.c和include/linux/vmacache.h中实现。
每当通过慢路径执行查找成功时，vmacache_update()会在数组current-> vmacache.vmas中存储指向VMA的指针，从而允许下一次查找使用快速路径。

请注意，VMA缓存是按线程进行的，但VMA与整个进程相关联（更准确地说，使用结构mm_struct;从现在开始，这种区别将在很大程度上被忽略，因为它与此漏洞无关）。
因此，当释放VMA时，必须使所有线程的VMA高速缓存无效 - 否则，下一个VMA查找将指向空指针。
但是，由于进程可以包含许多线程，因此只需遍历所有线程的VMA缓存就会出现性能问题。

为了解决这个问题，struct mm_struct和per-thread的struct vmacache都标有序列号;
当VMA查找快速路径在vmacache_valid()中发现current->vmacache.seqnum和current->mm->vmacache_seqnum不匹配时，它会擦除当前线程的VMA缓存的内容并更

mm_struct和VMA缓存的序列号长度只有32位，这意味着它们可能会溢出。
为了确保当current->mm->vmacache_seqnum实际增加232次时VMA缓存不能错误地显示为有效，vmacache_invalidate()（递增current->mm->vmacache_seqnum的
当current->mm->vmacache_seqnum换行为零时，它会调用vmacache_flush_all()来擦除与current->mm关联的所有VMA缓存的内容。
执行vmacache_flush_all()非常耗时：它会遍历整个机器上的每个线程，检查它与哪个struct mm_struct相关联，然后在必要时刷新线程的VMA缓存。

在版本3.16中，添加了一个优化：如果结构mm_struct仅与单个线程相关联，则vmacache_flush_all()将不执行任何操作，具体取决于每个VMA高速缓存失效之前是否进行V
因此，在单线程进程中，VMA缓存的序列号始终接近mm_struct的序列号：

```
/*
* ■■■■■■■■■■■■■■■■
* ■■■■■■■■■■■■■mm■seqnum■■■■■■■■■■■■■■seqnum■
* ■■■■■■■■■■■■■■■■
*/
if (atomic_read(&mm->mm_users) == 1)
return;
```

但是，这种优化是不正确的，因为它没有考虑如果先前的单线程进程在mm_struct的序列号已经置零后立即创建新线程会发生什么。
在这种情况下，第一个线程的VMA缓存的序列号仍然是0xffffffff，第二个线程可以再次驱动mm_struct的序列号到0xffffffff。
此时，第一个线程的VMA缓存（可以包含空指针）将再次被视为有效，允许在第一个线程的VMA缓存中使用释放的VMA指针。

这个漏洞的修复通过将序列号更改为64位，从而使溢出不可行，并删除溢出处理逻辑来实现。

可达性和影响力

从根本上说，这个漏洞可以由任何可以运行足够长时间来溢出引用计数器的进程触发（如果MAP_FIXED可用，则大约一个小时）并且能够使用mmap()/munmap()（管理内

为了简单起见，我的漏洞使用了各种其他内核接口，因此不仅仅是在这些沙箱内部工作;
特别是，它使用/dev/kmsg读取dmesg日志，并使用eBPF阵列通过用户控制的，可变的单页分配来垃圾邮件内核的页面分配器。
如果是愿意花更多时间来进行攻击的攻击者应该避免使用此类接口。

有趣的是，看起来Docker在其默认配置中不会阻止容器访问主机的dmesg日志，如果内核允许普通用户访问dmesg -
而容器中不存在/dev/kmsg，由于某种原因，seccomp策略将syslog()系统调用列入白名单。

BUG_ON(), WARN_ON_ONCE(),和dmesg

首次use-after-free访问的功能是vmacache_find()。首次添加此函数时-在引入漏洞之前-它按如下方式访问VMA缓存：

```
for (i = 0; i < VMACACHE_SIZE; i++) {
            struct vm_area_struct *vma = current->vmacache[i];

            if (vma && vma->vm_start <= addr && vma->vm_end > addr) {
                    BUG_ON(vma->vm_mm != mm);
                    return vma;
            }
    }
```

当此代码遇到缓存的VMA，其边界包含提供的地址addr时，它检查VMA的vm_mm指针是否与预期的mm_struct匹配 - 除非发生内存安全问题，否则应始终如此 -
如果没有，则以BUG_ON()断言失败而终止。
BUG_ON()旨在处理内核线程检测到严重问题的情况，这些问题无法通过从当前上下文中消失来干净地处理。在默认的上游内核配置中，BUG_ON()通常会将带有寄存器转储
这有时会阻止系统的其余部分继续正常工作 - 例如，如果崩溃的代码持有一个重要的锁，那么任何试图获取该锁的其他线程将会死锁 -
但它通常能成功地使系统的其余部分保持在合理可用的状态。 只有当内核检测到崩溃处于关键环境（例如中断处理程序）时，才会导致整个系统崩溃。

相同的处理程序代码用于处理内核代码中的意外崩溃，例如页面错误和非白名单地址的一般保护错误：默认情况下，如果可能，内核将尝试仅终止有问题的线程。

内核崩溃的处理是可用性，可靠性和安全性之间的权衡。
系统所有者可能希望系统尽可能长时间地运行，即使让系统的某些部分崩溃。如果发生突然的内核恐慌则会导致重要服务的数据丢失或停机。
同样，系统所有者也许希望在没有外部调试器的情况下调试实时系统上的内核错误; 如果在触发bug后整个系统终止，则可能更难以正确调试问题。
另一方面，企图利用内核漏洞的攻击者可能会获得在不触发系统重启的情况下多次尝试攻击的能力;
并且当攻击者能够读取第一次攻击产生的崩溃日志时，甚至可以利用该信息进行更复杂的二次攻击。

内核提供了两个可用于调整此行为的sysctl，具体取决于所需的权衡：

> 当BUG_ON()断言触发或内核崩溃时，kernel.panic_on_oops会自动导致内核崩溃; 可以使用构建配置变量CONFIG_PANIC_ON_OOPS配置其初始值。
> 默认情况下它在上游内核中是关闭的 - 默认情况下在分发中启用它可能是一个坏主意 - 但它就是（比如由Android启用）
>
> kernel.dmesg_restrict控制非root用户是否可以访问dmesg日志，其中包括内核崩溃的寄存器转储和堆栈跟踪;
> 可以使用构建配置变量CONFIG_SECURITY_DMESG_RESTRICT配置其初始值。 它在上游内核中默认关闭，但是可以由某些分发启用，比如Debian。
> （Android依靠SELinux阻止访问dmesg。）

举个例子，Ubuntu就无法实现这些功能。

我们在提交的同一个月内对之前的代码进行了修改：

```
for (i = 0; i < VMACACHE_SIZE; i++) {
              struct vm_area_struct *vma = current->vmacache[i];
-              if (vma && vma->vm_start <= addr && vma->vm_end > addr) {
-                          BUG_ON(vma->vm_mm != mm);
+              if (!vma)
+                          continue;
+              if (WARN_ON_ONCE(vma->vm_mm != mm))
+                          break;
+              if (vma->vm_start <= addr && vma->vm_end > addr)
                          return vma;
-              }
      }
```

这个修改过的代码正在随Ubuntu这样的发行版发布中。

这里的第一个变化是空指针的健全性检查在地址比较之前。 第二个更改更有趣：BUG_ON()替换为WARN_ON_ONCE()。

WARN_ON_ONCE()将调试信息输出到dmesg，类似于BUG_ON()打印的内容。
与BUG_ON()的区别在于WARN_ON_ONCE()仅在第一次触发时打印调试信息，并且继续执行：现在，当内核在VMA缓存查找快速路径中检测到空指针时 -
换句话说，当它启发式地检测到use-after-free后 - ，它只是从快速路径中脱离出来，然后又回到了红黑树的路径中。 该过程正常运行。

这符合内核的策略，即默认情况下尽可能地保持系统运行;
如果由于某种原因在这里触发了use-after-free漏洞，内核可能会启发式地减轻其影响并保持该过程正常工作。

即使内核发现了内存损坏，也只打印警告的策略对于当内核注意到与安全相关的事件(如内核内存损坏)时应该引起内核恐慌的系统来说是有问题的。简单地使WARN()触发内

对这个函数做了第三个重要的改变;然而，这种变化是相对较新的，将首先出现在4.19内核中，该内核尚未发布，因此它与攻击当前部署的内核无关。

```
for (i = 0; i < VMACACHE_SIZE; i++) {
-               struct vm_area_struct *vma = current->vmacache.vmas[i];
+               struct vm_area_struct *vma = current->vmacache.vmas[idx];

-               if (!vma)
```

```
-                       continue;
-               if (WARN_ON_ONCE(vma->vm_mm != mm))
-                       break;
-               if (vma->vm_start <= addr && vma->vm_end > addr) {
-                       count_vm_vmacache_event(VMACACHE_FIND_HITS);
-                       return vma;
+               if (vma) {
+#ifdef CONFIG_DEBUG_VM_VMACACHE
+                       if (WARN_ON_ONCE(vma->vm_mm != mm))
+                               break;
+#endif
+                       if (vma->vm_start <= addr && vma->vm_end > addr) {
+                               count_vm_vmacache_event(VMACACHE_FIND_HITS);
+                               return vma;
+                       }
             }
+               if (++idx == VMACACHE_SIZE)
+                       idx = 0;
     }
```

在这次修改之后，将跳过完整性检查，除非内核是用CONFIG_DEBUG_VM_VMACACHE选项构建的。

漏洞利用：增加序列号

该利用必须增加大约233次序列号。因此，用于增加序列号的原语的效率对整个利用程序的运行来说非常重要。

可能会导致每个系统调用产生两个序列号增量，如下所示:创建一个跨越三个页面的匿名VMA。然后重复使用带有MAP_FIXED的mmap()，用等效的VMA替换中间页面。这导

漏洞利用：替换VMA

枚举所有能利用use-after-free，而不释放slab的后备页（根据/proc/slabinfo，Ubuntu内核每个vm_area_struct slab使用一个页）回到伙伴分配器/页分配器的方法：

在同一个进程中重用vm_area_struct。然后进程就可以使用这个VMA，但是这不会产生任何有趣的结果，因为进程的VMA缓存无论如何都可以包含指向VMA的指针。

释放vm_area_struct，使其位于slab分配器的空闲列表上，然后尝试访问它。不过，至少Ubuntu使用的SLUB分配器用一个内核地址替换了vm_area_struct的前8个字节(->vm_start <= addr && VMA ->vm_end > addr无法实现，因此没有发生任何有趣的事情。

释放vm_area_struct，使其位于slab分配器的空闲列表上，然后在另一个进程中分配它。这将导致命中WARN_ON_ONCE()(除了一个非常窄的竞争条件，不容易反复触发

释放vm_area_struct，使其位于slab分配器的freelist上，然后从已与vm_area_struct slab合并的slab进行分配。这需要存在aliasing slab; 在Ubuntu 18.04 VM中，似乎不存在这样的slab。

因此，要利用这个漏洞，有必要将备份页释放回页面分配器，然后以某种方式重新分配页面，以允许在其中放置受控数据。可以使用各种内核接口；例如：
管道页：

- 优点:在分配上不擦除
- 优点:如果splice()可用，则允许在页面内任意偏移位置进行写入
- 优点:页面对齐
- 缺点:如果不首先释放页面，然后重新分配它，就不能执行多次写操作

BPF maps:

- 优点：可以从用户空间反复读写内容
- 优点：页面对齐
- 缺点：在分配上擦除

我们的利用代码将使用BPF maps

漏洞利用：从dmesg泄露指针

我们的漏洞利用想要获取下面的信息：

- mm_struct的地址
- use-after-free的VMA地址
- 加载内核代码的地址

至少在Ubuntu
18.04内核中，前两个在WARN_ON_ONCE()触发的寄存器转储中直接可见，因此可以很容易地从dmesg中提取:mm_struct的地址在RDI中，VMA的地址在RAX中。然而，

内核回溯可以包含多组寄存器集:当堆栈回溯逻辑遇到中断帧时，它会生成另一个寄存器转储。由于我们可以通过用户空间地址上的页错误触发WARN_ON_ONCE()，并且用
copy_to_user()/...），我们可以选择一个具有相关信息的调用点。事实证明，当R8仍然包含指向eventfd_fops结构的指针时，写入eventfd会触发一个usercopy。

当利用代码运行时，它将VMA替换为零内存，然后对损坏的VMA缓存触发VMA查找，故意触发WARN_ON_ONCE()。这产生了一个警告，看起来如下:

```
[ 3482.271265] WARNING: CPU: 0 PID: 1871 at /build/linux-SlLHxe/linux-4.15.0/mm/vmacache.c:102 vmacache_find+0x9c/0xb0
[...]
[ 3482.271298] RIP: 0010:vmacache_find+0x9c/0xb0
[ 3482.271299] RSP: 0018:ffff9e0bc2263c60 EFLAGS: 00010203
[ 3482.271300] RAX: ffff8c7caf1d61a0 RBX: 00007fffffffd000 RCX: 0000000000000002
[ 3482.271301] RDX: 0000000000000002 RSI: 00007fffffffd000 RDI: ffff8c7c214c7380
[ 3482.271301] RBP: ffff9e0bc2263c60 R08: 0000000000000000 R09: 0000000000000000
[ 3482.271302] R10: 0000000000000000 R11: 0000000000000000 R12: ffff8c7c214c7380
[ 3482.271303] R13: ffff9e0bc2263d58 R14: ffff8c7c214c7380 R15: 0000000000000014
[ 3482.271304] FS: 00007f58c7bf6a80(0000) GS:ffff8c7cbfc00000(0000) knlGS:0000000000000000
[ 3482.271305] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 3482.271305] CR2: 00007fffffffd000 CR3: 00000000a143c004 CR4: 00000000003606f0
[ 3482.271308] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[ 3482.271309] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
[ 3482.271309] Call Trace:
[ 3482.271314] find_vma+0x1b/0x70
[ 3482.271318] __do_page_fault+0x174/0x4d0
[ 3482.271320] do_page_fault+0x2e/0xe0
[ 3482.271323] do_async_page_fault+0x51/0x80
[ 3482.271326] async_page_fault+0x25/0x50
[ 3482.271329] RIP: 0010:copy_user_generic_unrolled+0x86/0xc0
[ 3482.271330] RSP: 0018:ffff9e0bc2263e08 EFLAGS: 00050202
[ 3482.271330] RAX: 00007fffffffd008 RBX: 0000000000000008 RCX: 0000000000000001
[ 3482.271331] RDX: 0000000000000000 RSI: 00007fffffffd000 RDI: ffff9e0bc2263e30
[ 3482.271332] RBP: ffff9e0bc2263e20 R08: ffffffffa7243680 R09: 0000000000000002
[ 3482.271333] R10: ffff8c7bb4497738 R11: 0000000000000000 R12: ffff9e0bc2263e30
[ 3482.271333] R13: ffff8c7bb4497700 R14: ffff8c7cb7a72d80 R15: ffff8c7bb4497700
[ 3482.271337] ? _copy_from_user+0x3e/0x60
[ 3482.271340] eventfd_write+0x74/0x270
[ 3482.271343] ? common_file_perm+0x58/0x160
[ 3482.271345] ? wake_up_q+0x80/0x80
[ 3482.271347] __vfs_write+0x1b/0x40
[ 3482.271348] vfs_write+0xb1/0x1a0
[ 3482.271349] SyS_write+0x55/0xc0
[ 3482.271353] do_syscall_64+0x73/0x130
[ 3482.271355] entry_SYSCALL_64_after_hwframe+0x3d/0xa2
[ 3482.271356] RIP: 0033:0x55a2e8ed76a6
[ 3482.271357] RSP: 002b:00007ffe71367ec8 EFLAGS: 00000202 ORIG_RAX: 0000000000000001
[ 3482.271358] RAX: ffffffffffffffda RBX: 0000000000000000 RCX: 000055a2e8ed76a6
[ 3482.271358] RDX: 0000000000000008 RSI: 00007fffffffd000 RDI: 0000000000000003
[ 3482.271359] RBP: 0000000000000001 R08: 0000000000000000 R09: 0000000000000000
[ 3482.271359] R10: 0000000000000000 R11: 0000000000000202 R12: 00007ffe71367ec8
[ 3482.271360] R13: 00007fffffffd000 R14: 0000000000000009 R15: 0000000000000000
[ 3482.271361] Code: 00 48 8b 84 c8 10 08 00 00 48 85 c0 74 11 48 39 78 40 75 17 48 39 30 77 06 48 39 70 08 77 8d 83 c2 01 83
[ 3482.271381] ---[ end trace bf256b6e27ee4552 ]---
```

此时，该漏洞可以创建一个包含正确mm_struct指针(从RDI泄漏)的伪VMA。它还通过引用伪数据结构来填充其他字段(通过使用从RAX的泄漏的VMA指针创建指向伪VMA的

漏洞利用：JOP（最无聊的部分）

通过利用在现有的页面上覆盖一个伪可写的VMA的能力，或者类似的东西，很有可能以一种非常优雅的方式利用这个漏洞；然而，这种利用只是使用了经典的面向跳转编程

为了再次触发use-after-free，对没有可分页条目的地址执行写入内存访问。此时，内核的页面错误处理程序通过page_fault -> do_page_fault ->
do_page_fault -> handle_mm_fault -> handle_mm_fault -> handle_pte_fault -> do_fault -> do_shared_fault -> __do_fault 执行间接调用:

```
static int __do_fault(struct vm_fault *vmf)
{
struct vm_area_struct *vma = vmf->vma;
int ret;

ret = vma->vm_ops->fault(vmf);
```

vma是我们控制的VMA结构，所以在这一点上，我们可以获得指令指针控制。R13包含一个指向vma的指针。下面使用的JOP链;它相当粗糙(例如，它会在完成任务后崩溃)，

首先，移动VMA指针到RDI：

```
ffffffff810b5c21: 49 8b 45 70 mov rax,QWORD PTR [r13+0x70]
ffffffff810b5c25: 48 8b 80 88 00 00 00 mov rax,QWORD PTR [rax+0x88]
ffffffff810b5c2c: 48 85 c0 test rax,rax
ffffffff810b5c2f: 74 08 je ffffffff810b5c39
ffffffff810b5c31: 4c 89 ef mov rdi,r13
ffffffff810b5c34: e8 c7 d3 b4 00 call ffffffff81c03000 <__x86_indirect_thunk_rax>
```

然后，完全控制RDI：

```
ffffffff810a4aaa: 48 89 fb mov rbx,rdi
ffffffff810a4aad: 48 8b 43 20 mov rax,QWORD PTR [rbx+0x20]
ffffffff810a4ab1: 48 8b 7f 28 mov rdi,QWORD PTR [rdi+0x28]
ffffffff810a4ab5: e8 46 e5 b5 00 call ffffffff81c03000 <__x86_indirect_thunk_rax>
```

此时，我们可以调用run_cmd()，它使用空格分隔的路径和参数列表作为唯一的参数，生成一个root特权用户模式助手。这使我们能够运行root特权提供的二进制文件。(感
helper...)

在启动usermode helper之后，内核会因为页错误而崩溃，因为JOP链没有干净地终止;但是，由于这只会杀死发生错误的进程，所以并不是很重要。

修复时间线

此错误报告于2018-09-12。两天后，即2018-09-14，在内核树的上游进行了修复。与其他软件供应商的修复时间相比，这是非常快的。在这一点上，下游厂商理论上可以支

然而，上游内核中的修复并不意味着用户的系统实际上已经修复了。对于使用基于上游稳定分支的分发内核的用户，向其发送修复程序的正常流程大致如下:

1. 一个补丁落在了内核的上游。
2. 这个补丁被反向移植到一个向上支持的稳定内核中。
3. 发行版将向上支持的稳定内核的更改合并到其内核中。
4. 用户安装新的发行版内核。

注意，补丁在第1步之后就公开了，这可能允许攻击者利用漏洞，但是用户只有在第4步之后才受到保护。

在本例中，在补丁公开5天后，对上行支持的稳定内核4.18、4.14、4.9和4.4的后端版本发布了2018-09-19，此时发行版可以将补丁引入。

上游稳定内核更新的发布非常频繁。例如，查看4.14 stable内核的最后几个稳定版本，这是最新的上游长期维护版本:
4.14.72 on 2018-09-26
4.14.71 on 2018-09-19
4.14.70 on 2018-09-15
4.14.69 on 2018-09-09
4.14.68 on 2018-09-05
4.14.67 on 2018-08-24
4.14.66 on 2018-08-22

4.9和4.4长期维护内核的更新频率相同;只有3.16长期维护内核在最近一次更新(2018-09-25(3.16.58)和上一次更新(2018-06-16(3.16.57)之间没有收到任何更新。

然而，Linux发行版通常并不经常发布发行版内核更新。例如，Debian
stable发布了一个基于4.9的内核，但是截止到2018-09-26，这个内核最近一次更新是2018-08-21。类似地，Ubuntu
16.04发布了一个内核，最近更新时间是2018-08-27。Android每个月只发布一次安全更新。因此，当上游稳定内核中存在安全关键的修复程序时，用户仍然可能需要数周时

在这种情况下，安全问题在2018-09-18的oss-security邮件列表上公布，并在2018-09-19进行了CVE分配，这使得向用户发送新的分发内核的需求更加清晰。
仍然：截至2018-09-26，Debian和Ubuntu（在16.04和18.04版本中）都将这个漏洞跟踪为unfixed：
https://security-tracker.debian.org/tracker/CVE-2018-17182
https://people.canonical.com/~ubuntu-security/cve/2018/CVE-2018-17182.html

Fedora在2018-09-22向用户推送了更新:https://bugzilla.redhat.com/show_bug.cgi?id=1631206#c8

总结

这个漏洞显示了内核配置对编写内核漏洞利用的难度有多大的影响。尽管简单地打开每一个与安全相关的内核配置选项可能是一个坏主意，但它们中的一些——比如kernel.
sysctl——在启用时似乎能提供合理的权衡。

修复时间表显示了内核处理严重安全漏洞的方法，非常有效地快速登陆git主树中的修复程序，但在发布上游修复程序和修复程序实际可供用户使用的时间之间留下了一个曝
- 此时间窗口足够大，以至于攻击者可以在此期间编写内核漏洞利用程序。

上一篇：攻击者是如何将PHP Phar包伪... 下一篇：SQL 注入总结

1. 0 条回复
   - 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板