大脏牛（CVE-2017-1000405）

不完整的脏牛修复补丁

脏牛漏洞(CVE-2016－5195)可能是公开后影响范围最广和最深的漏洞之一，这十年来的每一个 Linux 版本，包括 Android、桌面版和服务器版都受到其影响。通过该漏洞可以轻易地绕过常用的漏洞防御方法，攻击到几百万的用户。目前已经有许多关于该漏洞的分析文章，但很少有对其补丁的深入研究。

我们（Bindecy）对该补丁和内容十分感兴趣，更重要的是，尽管漏洞的后果已经十分严重，但我们发现它的修复补丁仍存在缺陷。

回顾脏牛漏洞

首先我们需要完整地来理解一下原始的脏牛漏洞利用方式。考虑到已经有详细的解释（关于脏牛分析的链接）所以我们假设你已经有一定的 Linux 内存管理基础，不再具体讲述之前漏洞的分析。

之前的漏洞是在 get_user_pages 函数中。这个函数能够获取用户进程调用的虚拟地址之后的物理地址，调用者需要声明它想要执行的具体操作（例如写、锁等操作）所以内存管理可以准备相对应的内存页。具体来说，也就是当进行写入私有映射的内存页时，会经过一个 COW（即写即拷）的过程，即只读文件会复制生成一个可以写入的新文件，原始文件可能是私有保护的，但它可以被其他进程映射使用，也可以在修改后重新写入到磁盘中。

现在我们来具体看下 get_user_pages 函数的相关代码。

```
static long __get_user_pages(struct task_struct *tsk, struct
mm_struct *mm,
        unsigned long start, unsigned long nr_pages,
        unsigned int gup_flags, struct page **pages,
        struct vm_area_struct **vmas, int *nonblocking)
{
    // ...
    do {
        struct page *page;
```

```
        unsigned int foll_flags = gup_flags;
        // ...
        vma = find_extend_vma(mm, start);
        // ...

retry:
        // ...
        cond_resched();
        page = follow_page_mask(vma, start, foll_flags, &page_mask);
        if (!page) {
            int ret;
            ret = faultin_page(tsk, vma, start, &foll_flags,
                    nonblocking);
            switch (ret) {
            case 0:
                goto retry;
            case -EFAULT:
            case -ENOMEM:
            case -EHWPOISON:
                return i ? i : ret;
            case -EBUSY:
                return i;
            case -ENOENT:
                goto next_page;
```

```
            }

            BUG();

        }

        // ...


next_page:

        // ...

        nr_pages -= page_increm;

    } while (nr_pages);

    return i;

}
```

整个 while 循环的目的是获取请求页队列中的每个页，反复操作直到满足构建所有内存映射的需求，这也是 retry 标签的作用。

follow_page_mask 读取页表来获取指定地址的物理页（同时通过 PTE 允许）或获取不满足需求的请求内容。在 follow_page_mask 操作中会获取 PTE 的 spinlock-用来保护我们试图获取内容的物理页不被泄露。

faultin_page 函数申请内存管理的权限（同样有 PTE 的 spinlock 保护）来处理目标地址中的错误信息。注意在成功调用 faultin_page 后，锁会自动释放-从而保证 follow_page_mask 能够成功进行下一次尝试，下面是我们使用时可能涉及到的代码。

原始的漏洞代码在 faultin_page 底部：

```
if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))

*flags &= ~FOLL_WRITE;
```

移除 FOLL_WRITE 标志的原因是考虑到只读 VMA（当 VMA 中有 VM_MAYWRITE 标志）使用 FOLL_FORCE 标志的情况，在我们的例子中，pte_maybe_mkwrite 函数不会修改写入字节，然而 faulted-in 页是可以进行写入的。

当页在进行 faultin_page 时经过 COW 循环(有 VM_FAULT_WRITE 标志)，同时 VMA 是不可写入的，那么 FOLL_WRITE 标志将会在下次尝试访问页时移除-只能进行只读权限的请求。

在最初的 follow_page_mask 因为页只读或不存在而失败后，我们会尝试进一步的研究。想象下直到下一次试图获取页的这段时间里，我们跳过 COW 版本（例如使用 madvise(MADV_DONTNEED）。

下一次调用的 faultin_page 将不会有 FOLL_WRITE 标志，所以我们从缓存页中获取能够获取只读版本的页文件。现在因为下一次调用 follow_page_mask 的请求没有 FOLL_WRITE 标志，那么它就会返回只读权限的页-违背了调用者最初写入权限页的请求

基本来看，上述的过程流也就是脏牛漏洞-允许我们对只读权限的内存页进行写入操作。在 faultin_page 中有对应的修复补丁：

```
if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))

*flags |= FOLL_COW; // Instead of *flags &= ~FOLL_WRITE;
```

同时也加入了另一个新的 follow_page_mask 函数：

```
/*

 * FOLL_FORCE can write to even unwritable pte's, but only

 * after we've gone through a COW cycle and they are dirty.

 */

static inline bool can_follow_write_pte(pte_t pte, unsigned int
flags)

{

    return pte_write(pte) ||

        ((flags & FOLL_FORCE) && (flags & FOLL_COW) &&
pte_dirty(pte));

}
```

与减少权限请求数不同，get_user_pages 现在记住了我们经过 COW 循环的过程。之后我们只需要有 FOLL_FORCE 和 FOLL_COW 标志声明过且 PTE 标记为 dirty，就可以获取只读页的写入操作。

这个补丁假设只读权限的复制页永远不会有 PTE 指针调用 dirty bit，这是不是一个靠谱的假设呢？

大页内存管理(THP)

一般来说，Linux 通常使用 4096 字节的页。我们可以增加页表项，或使用更大的页来使系统管理更大的内存。我们使用 Linux 的大内存页，即后者方法来满足我们的需求。

一个大页一般是指 2MB 的长页，一种利用方式是通过大页管理机制，尽管还有其他的利用方式来管理大内存页，但本文不做更多的讨论。

内核会通过分配大页来满足相关的内存需求，THP 是可以交换且不可破坏的（例如分割为普通的 4096 字节页），能够用于匿名,shmem 和 tmpfs 映射（后两者只在新内核版本中使用）。

一般来说（根据编译标志头和机器设置）默认的 THP 设置只允许匿名映射，shmem 和 tmpfs 支持都需要手动打开，而通常 THP 设置都会根据系统运行的内核具体文件来决定是打开还是关闭。

一个重要的优化方式是将普通页聚合为大内存页，一个叫做 khugepaged 的镜像会不断扫描可用于聚合为大页的普通页。显然一个 VMA（虚拟内存空间）必须包含整个连续的 2MB 内存页面才能被用于聚合。

THP 通过 PMD（Pages Medium 目录，PTE 文件下一级）的_PAGE_PSE 设置来打开，PMD 指向一个 2MB 的内存页而非 PTEs 目录。PMDs 在每一次扫描到页表时都会通过 pmd_trans_huge 函数进行检查，所以我们可以通过观察 PMD 指向 pfn 还是 PTEs 目录来判断是否可以聚合。在一些结构中，大 PUDs(上一级目录)同样存在，这会导致产生 1GB 的页。

THP 从 2.6.38 内核版本后都提供支持，在大多数 Android 设备中 THP 子系统都没有被启动。

漏洞

仔细查看脏牛补丁中关于 THP 的部分，我们可以发现大 PMDs 中用了和 can_follow_write_pte 同样的逻辑，其添加的对应函数 can_follow_write_pmd：

```
static inline bool can_follow_write_pmd(pmd_t pmd, unsigned int
flags)

{
```

```
    return pmd_write(pmd) ||
        ((flags & FOLL_FORCE) && (flags & FOLL_COW) &&
pmd_dirty(pmd));
}
```

然而在大 PMD 中，一个页可以通过 touch_pmd 函数，无需 COW 循环就标记为 dirty：

```
static void touch_pmd(struct vm_area_struct *vma, unsigned long addr,
        pmd_t *pmd)
{
    pmd_t _pmd;

    /*
     * We should set the dirty bit only for FOLL_WRITE but for now
     * the dirty bit in the pmd is meaningless.  And if the dirty
     * bit will become meaningful and we'll only set it with
     * FOLL_WRITE, an atomic set_bit will be required on the pmd to
     * set the young bit, instead of the current set_pmd_at.
     */
    _pmd = pmd_mkyoung(pmd_mkdirty(*pmd));
    if (pmdp_set_access_flags(vma, addr & HPAGE_PMD_MASK,
                pmd, _pmd,  1))
        update_mmu_cache_pmd(vma, addr, pmd);
}
```

这个函数在 follow_page_mask 每次获取 get_user_pages 试图访问大页面时被调用。很明显这个注释有问题而现在 dirty bit 并非无意义的。尤其是在使用

get_user_pages 来读取大页时，这个页会无需经过 COW 循环而标记为 dirty，使得 can_follow_write_pmd 的逻辑发生错误。

在此时，如何利用该漏洞就很明显了-我们可以使用类似脏牛的方法。这次在我们获取复制的内存页后，可以污染两次原始页-第一次创建它，第二次写入 dirty bit。

那么一个不可避免的问题来了，究竟有多严重的影响？

漏洞说明

我们通过对一个只读大内存页进行写入操作来展示漏洞的利用，其中唯一约束是 madvise(MADV_DONTNEED)限制我们的内存访问。在 fork 后继承自主进程的匿名大页是一个容易攻击的目标，但他们在销毁后就关闭了-也就是说我们无法再次访问它。

我们发现了两类不应写入内容的可能攻击目标：

- 大零内存页
- 封闭（只读）大内存页

零页

当匿名映射在写入前发生读取错误时，我们会获取叫做零页的物理地址。这个优化系统可以避免系统分配多个未被写入的新建零页。所以同一个零页可能映射在多个不同有不同安全等级的进程中。

同样的原理也应用在大页中-如果没有发生错误，那么就无需创建另一个大页-映射产生一个叫做大零页的特殊页，注意这个特性是可以手动关闭的。

THP，shmem 和封闭文件

在使用 THP 时都会包含 shmem 和 tmpfs 文件，shmem 文件可以通过 memfd_create_syscall 或通过共享映射来创建，tmpfs 文件可以通过 tmpfs(通常为/dev/shm)的指针来创建，两个都可以根据系统设置来映射大内存页。

shmem 文件可以被封闭-封闭文件能够限制用户可以对文件进行的操作类型。这一机制允许相互不信任的进程通过无需额外操作的共享内存来交流信息从而处理共享内存区域的意外操作 (搜索 man memfd_create()函数可以了解更多信息)，一共有三类封闭类型：

- F_SEAL_SHRINK:文件大小不可被缩小
- F_SEAL_GROW:文件大小不可被增加

- F_SEAL_WRITE:文件内容不可被修改.

这些封闭方式都可以通过 funtl syscall 来添加到 shmem 文件中。

POC

我们地 poc 展示了对零页的重新写入，重新写入 shmem 的利用方式可能产生其他类似的漏洞利用方式。

注意在最初地写入零页操作后，它会用一个新的（也是原始的）大页内存管理来替换掉。通过这一方法，我们成功的令多个进程崩溃。对零页区域的重新写入可能导致程序中 BSS 段的错误初始化，常见的漏洞利用方式包括利用它来声明未被初始化的全局变量。

下列崩溃案例展示了这一具体内容，在该案例中，火狐的 JS Helper 线程可能因为%rdx 错误地使用了布尔指针，从而创建了一个使用 NULL-deref 的对象。

```
Thread 10 "JS Helper" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7fffe2aee700 (LWP 14775)]
0x00007ffff13233d3 in ?? () from /opt/firefox/libxul.so
(gdb) i r
rax            0x7fffba7ef080 140736322269312
rbx            0x0 0
rcx            0x22 34
rdx            0x7fffba7ef080 140736322269312
rsi            0x400000000 17179869184
rdi            0x7fffe2aede10 140736996498960
rbp            0x0 0x0
rsp            0x7fffe2aede10 0x7fffe2aede10
r8             0x20000 131072
r9             0x7fffba900000 140736323387392
r10            0x7fffba700000 140736321290240
r11            0x7fffe2aede50 140736996499024
r12            0x1 1
r13            0x7fffba7ef090 140736322269328
r14            0x2 2
r15            0x7fffe2aee700 140736996501248
rip            0x7ffff13233d3 0x7ffff13233d3
eflags         0x10246 [ PF ZF IF RF ]
cs             0x33 51
ss             0x2b 43
ds             0x0 0
es             0x0 0
fs             0x0 0
```

```
gs             0x0 0
(gdb) x/10i $pc-0x10
   0x7ffff13233c3: mov    %rax,0x10(%rsp)
   0x7ffff13233c8: mov    0x8(%rdx),%rbx
   0x7ffff13233cc: mov    %rbx,%rbp
   0x7ffff13233cf: and    $0xfffffffffffffffe,%rbp
=> 0x7ffff13233d3: mov    0x0(%rbp),%eax
   0x7ffff13233d6: and    $0x28,%eax
   0x7ffff13233d9: cmp    $0x28,%eax
   0x7ffff13233dc: je     0x7ffff1323440
   0x7ffff13233de: mov    %rbx,%r13
   0x7ffff13233e1: and    $0xfffffffffff00000,%r13
(gdb) x/10w $rdx
0x7fffba7ef080: 0x41414141 0x00000000 0x00000000 0x00000000
0x7fffba7ef090: 0xeef93bba 0x00000000 0xda95dd80 0x00007fff
0x7fffba7ef0a0: 0x778513f1 0x00000000
```

这是另一个崩溃案例-GDB 在读取火狐调试进程的符号时崩溃报错。

```
(gdb) r
Starting program: /opt/firefox/firefox
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
gnu/libthread_db.so.1".
Program received signal SIGSEGV, Segmentation fault.
0x0000555555825487 in eq_demangled_name_entry (a=0x4141414141414141,
b=<optimized out>) at symtab.c:697
697   return strcmp (da->mangled, db->mangled) == 0;
(gdb) i s
#0  0x0000555555825487 in eq_demangled_name_entry
(a=0x4141414141414141, b=<optimized out>) at symtab.c:697
#1  0x0000555555955203 in htab_find_slot_with_hash
(htab=0x555557008e60, element=element@entry=0x7fffffffdb00,
hash=4181413748, insert=insert@entry=INSERT) at ./hashtab.c:659
#2  0x0000555555955386 in htab_find_slot (htab=<optimized out>,
element=element@entry=0x7fffffffdb00, insert=insert@entry=INSERT)
at ./hashtab.c:703
#3  0x00005555558273e5 in symbol_set_names
(gsymbol=gsymbol@entry=0x5555595b3778,
linkage_name=linkage_name@entry=0x7ffff2ac5254
"_ZN7mozilla3dom16HTMLTableElement11CreateTHeadEv", len=len@entry=48,
    copy_name=copy_name@entry=0, objfile=<optimized out>) at
symtab.c:818
#4  0x00005555557d186f in minimal_symbol_reader::record_full
```

(this=0x7fffffffdce0, this@entry=0x1768bd6, name=<optimized out>,
    name@entry=0x7ffff2ac5254
"_ZN7mozilla3dom16HTMLTableElement11CreateTHeadEv",
name_len=<optimized out>, copy_name=copy_name@entry=48,
address=24546262, ms_type=ms_type@entry=mst_file_text,
    section=13) at minsyms.c:1010
#5 0x00005555556959ec in record_minimal_symbol (reader=...,
name=name@entry=0x7ffff2ac5254
"_ZN7mozilla3dom16HTMLTableElement11CreateTHeadEv",
name_len=<optimized out>, copy_name=copy_name@entry=false,
    address=<optimized out>, address@entry=24546262,
ms_type=ms_type@entry=mst_file_text, bfd_section=<optimized out>,
objfile=0x555557077860) at elfread.c:209
#6 0x0000555555696ac6 in elf_symtab_read (reader=...,
objfile=objfile@entry=0x555557077860, type=type@entry=0,
number_of_symbols=number_of_symbols@entry=365691,
    symbol_table=symbol_table@entry=0x7ffff6a6d020,
copy_names=copy_names@entry=false) at elfread.c:462
#7 0x00005555556970c4 in elf_read_minimal_symbols
(symfile_flags=<optimized out>, ei=0x7fffffffdcd0,
objfile=0x555557077860) at elfread.c:1084
#8 elf_symfile_read (objfile=0x555557077860, symfile_flags=...) at
elfread.c:1194
#9 0x000055555581f559 in read_symbols
(objfile=objfile@entry=0x555557077860, add_flags=...) at
symfile.c:861
#10 0x000055555581f00b in syms_from_objfile_1 (add_flags=...,
addrs=0x555557101b00, objfile=0x555557077860) at symfile.c:1062
#11 syms_from_objfile (add_flags=..., addrs=0x555557101b00,
objfile=0x555557077860) at symfile.c:1078
#12 symbol_file_add_with_addrs (abfd=<optimized out>,
name=name@entry=0x55555738c1d0 "/opt/firefox/libxul.so",
add_flags=..., addrs=addrs@entry=0x555557101b00, flags=...,
parent=parent@entry=0x0)
    at symfile.c:1177
#13 0x000055555581f63d in symbol_file_add_from_bfd (abfd=<optimized
out>, name=name@entry=0x55555738c1d0 "/opt/firefox/libxul.so",
add_flags=..., addrs=addrs@entry=0x555557101b00, flags=...,
    parent=parent@entry=0x0) at symfile.c:1268
#14 0x000055555580b256 in solib_read_symbols
(so=so@entry=0x55555738bfc0, flags=...) at solib.c:712
#15 0x000055555580be9b in solib_add (pattern=pattern@entry=0x0,
from_tty=from_tty@entry=0, readsyms=1) at solib.c:1016
#16 0x000055555580c678 in handle_solib_event () at solib.c:1301

```
#17 0x00005555556f9db4 in bpstat_stop_status (aspace=0x555555ff5670,
bp_addr=bp_addr@entry=140737351961185, ptid=...,
ws=ws@entry=0x7fffffffe1d0) at breakpoint.c:5712
#18 0x00005555557ad1ef in handle_signal_stop (ecs=0x7fffffffe1b0) at
infrun.c:5963
#19 0x00005555557aec8a in handle_inferior_event_1
(ecs=0x7fffffffe1b0) at infrun.c:5392
#20 handle_inferior_event (ecs=ecs@entry=0x7fffffffe1b0) at
infrun.c:5427
#21 0x00005555557afd57 in fetch_inferior_event
(client_data=<optimized out>) at infrun.c:3932
#22 0x000055555576ade5 in gdb_wait_for_event (block=block@entry=0) at
event-loop.c:859
#23 0x000055555576aef7 in gdb_do_one_event () at event-loop.c:322
#24 0x000055555576b095 in gdb_do_one_event () at ./common/common-
exceptions.h:221
#25 start_event_loop () at event-loop.c:371
#26 0x00005555557c3938 in captured_command_loop (data=data@entry=0x0)
at main.c:325
#27 0x000055555576d243 in catch_errors
(func=func@entry=0x5555557c3910 <captured_command_loop(void*)>,
func_args=func_args@entry=0x0,
errstring=errstring@entry=0x555555a035da "",
    mask=mask@entry=RETURN_MASK_ALL) at exceptions.c:236
#28 0x00005555557c49ae in captured_main (data=<optimized out>) at
main.c:1150
#29 gdb_main (args=<optimized out>) at main.c:1160
#30 0x00005555555ed628 in main (argc=<optimized out>, argv=<optimized
out>) at gdb.c:32
(gdb) list
692   const struct demangled_name_entry *da
693     = (const struct demangled_name_entry *) a;
694   const struct demangled_name_entry *db
695     = (const struct demangled_name_entry *) b;
696
697   return strcmp (da->mangled, db->mangled) == 0;
698 }
699
700 /* Create the hash table used for demangled names.  Each hash
entry is
701    a pair of strings; one for the mangled name and one for the
demangled
(gdb)
```

我们的 POC 地址。

总结

这个漏洞展示了补丁跟踪在安全开发周期中的重要性，正如同脏牛和其他漏洞的例子一样，即便是修复过的漏洞也可能产生有风险的修复补丁，这不仅是闭源软件面临的问题，开源软件也有这样的风险。

欢迎在下面提出和交流更多的想法和问题。

公开时间

最初的报道是 17 年 11 月 22 号，几天后发布了相应的修复补丁，补丁修复了当使用者请求写入操作时 touch_pmd 函数 PMD 入口的 dirty bit。

感谢安全团队对该高安全漏洞的长期关注和修复所做出的贡献。

17 年 11 月 22 日-漏洞内容提交到 security@kernel.org 和 linux-distros@vs.openwall.org。

17 年 11 月 22 日-获得 CVE-2017-1000405 编号。

17 年 11 月 27 日-发布修复补丁。

17 年 11 月 29 日-公开漏洞内容。