# 利用CVE-2021-43267

**导言**

几周前，Max van Amerongen（@[maxpl0it](https://twitter.com/maxpl0it)）披露了Linux内核TIPC子系统中的堆溢出漏洞。他在[哨兵实验室网站上](https://www.sentinelone.com/labs/tipc-remote-linux-kernel-heap-overflow-allows-arbitrary-code-execution/)发布了关于这个错误的详细文章。

这是一个非常清晰的剪切堆缓冲区溢出，我们控制溢出的大小和数据。我决定开始一个小的漏洞开发冒险，看看在有常见缓解措施的内核（SMEP/SMAP/KASLR/KPTI）上利用这个错误有多难。

如果你来这里是为了找到新的内核开发策略，你选错了博客，对不起！

**TIPC？**

要引用[TIPC网页](http://tipc.sourceforge.net/intro.html)：

您是否曾希望即使在集群节点之间传输数据时也能方便Unix域套接字？您自己在哪里确定要绑定和使用地址？您不必执行DNS查找并担心IP地址的地方？您无需启动计时器来监控对等套接字的连续性在哪里？然而，如果没有那种套接字类型的缺点，例如挥之不去的风险？

嗯。。我没有。但话又说回来，我只是个机会主义的庸俗者。

欢迎来到**透明进程间通信**服务，简称TIPC，它为您提供所有这些，等等。

谢谢你邀请我。

**如何给udp翻倒？**

要使用Linux内核提供的TIPC支持，您必须编译启用TIPC的内核，或加载许多流行发行版随附的TIPC模块。为了轻松与TIPC子系统接口，您可以使用iproute2中的tipc实用程序。

例如，要列出所有节点链接，您可以发布tipc link list。我们希望通过UDP与TIPC子系统交谈，为此，我们必须启用UDP承载媒体。这可以使用tipc bearer enable media udp name <NAME> localip <SOMELOCALIP>。

在引擎盖下，tipc用户域实用程序使用netlink消息（使用地址系列AF\_TIPC）来做它的事情。有趣的是，这些网络链接消息可以由任何非特权用户发送。因此，即使没有现有的TIPC配置，这个错误仍然可以被利用。

**如何到达脆弱的代码路径？**

因此，现在我们知道如何为TIPC*启用*UDP侦听器，我们如何实际访问易受攻击的代码？我们必须将自己呈现为一个有效的node并建立一个link，然后才能触发MSG\_CRYPTO代码路径。我们可以在TIPC网页上找到一个[协议规范](http://tipc.sourceforge.net/protocol.html)，详细说明了有关传输、寻址方案、碎片化等所有内容。

不过，那是很多非常干燥的东西。我做了一些pCAP，做了一个提示-over-udpsession设置，经过一些挥手和读取，内核源将其缩小到我们需要发出的几个数据包，然后我们才能开始发送我们感兴趣的消息。

简而言之，典型的TIPC数据报以标题开头，标题由至少六个32位单词组成，按大端字节顺序排列。这些通常被称为w0到w5。此标头（可选）后跟有效负载。w0对TIPC版本、标头大小、有效负载大小、消息“协议”进行编码。还有一个标志，指示这是否是顺序消息。

w1对特定于协议的消息类型进行编码（除其他外）。

标题还在w3中指定node\_id，这是每个数据包中包含的唯一标识符。通常，节点将其IPv4地址编码为node\_id。

了解标题格式的各种位字段的一个快速方法是咨询[net/tipc/msg.h](https://elixir.bootlin.com/linux/latest/source/net/tipc/msg.h)。

为了建立有效的节点链接，我们发送三个数据包：

protocol: LINK\_CONFIG -> message type: DSC\_REQ\_MSG

protocol: LINK\_PROTOCOL -> message type: RESET\_MSG

protocol: LINK\_PROTOCOL -> message type: STATE\_MSG

The LINK\_CONFIG packet will advertise ourselves. The link is then reset with the LINK\_PROTOCOL packet that has the RESET\_MSG message type. Finally, the link is brought up by sending a LINK\_PROTOCOL packet with the STATE\_MSGmessage type.

Now we are actually in a state where we can send MSG\_CRYPTO TIPC packets and start playing with the heap overflow bug.

**腐蚀一些记忆**

A MSG\_CRYPTO TIPC packet payload looks like this:

struct tipc\_aead\_key {

char alg\_name[TIPC\_AEAD\_ALG\_NAME];

unsigned int keylen;

char key[];

};

正如哨兵实验室所详述的，kmalloc’d堆缓冲区的长度是通过**从TIPC数据包头**中提取有效负载大小来确定的。然后，密钥将memcpy'd到此缓冲区中，长度为MSG\_CRYPTO结构中指定的长度。

起初，您会认为这意味着溢出使用不受控制的数据..但您可以发送与有效负载的实际长度相干的TIPC数据包（通过使tipc\_hdr.payload\_size小于实际有效负载大小）。这将通过所有检查并到达memcpy，而无需丢弃有效负载的其余部分，从而让我们完全控制溢出的数据。太好了！

keylen中指定的长度将直接传递给kmalloc。

内核使用kmalloc分配的较小（高达8KiB）堆对象最终进入按大小分组的缓存（功率为两个）。您可以偷看/proc/slabinfo，并查看以kmalloc-为前缀的条目，以了解通用对象缓存大小的概述。

由于我们可以控制将被分配和溢出的堆缓冲区的大小，我们可以选择我们的对象最终在哪个缓存中。这是一个很棒的能力，因为它允许我们溢出到类似大小的相邻内核对象中！

**击败KASLR**

如果我们想进行任何形式的控制流劫持，我们需要信息泄露来披露（至少）一些内核文本/数据地址，以便我们推断随机基地址。

如果我们能在其结构的早期找到一些包含指针/偏移/长度字段的内核对象，并且可以以某种方式将数据返回给用户域，那就太好了。

在谷歌搜索时，我无意中发现了宾夕法尼亚州立大学一些学生的这[篇很酷](https://dl.acm.org/doi/10.1145/3372297.3423353)的[论文](https://dl.acm.org/doi/10.1145/3372297.3423353)，他们把具有此类属性的物体称为“弹性物体”。他们对这个问题的研究相当详尽，涵盖了Linux、BSD和XNU。我绝对建议去看看。

由于我们可以任意选择大小的分配，我们可以自由瞄准任何方便的弹性物体。我选择了msg\_msg，这是其他开发人员的[热门选择](https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html)。;-)

The structure of a msg\_msg as defined in [include/linux/msg.h](https://elixir.bootlin.com/linux/latest/source/include/linux/msg.h):

/\* one msg\_msg structure for each message \*/

struct msg\_msg {

struct list\_head m\_list;

long m\_type;

size\_t m\_ts; /\* message text size \*/

struct msg\_msgseg \*next;

void \*security;

/\* the actual message follows immediately \*/

};

You can easily allocate msg\_msg objects using the msgsnd system call. And they can be freed again using the msgrcv system call. These system calls are not to be confused with the sendmsg and recvmsg system calls btw, great naming scheme!

If we corrupt the m\_ts field of a msg\_msg object we can extend its size and get a relative kernel heap out-of-bounds read back to userland when retrieving the message from the queue again using the msgrcv system call.

A small problem with this is that overwriting the m\_ts field also requires trampling over the struct list\_head members (a prev/next pointer). When msgrcv is called and a matching message is found, it wants to unlink it from the linked list.. but since we’re still in the information leak stage of the exploit, we can’t put any legitimate/meaningful pointers in there. Luckily, there’s a flag you can pass to msgrcv called MSG\_COPY, which will make a copy of the message and *not* unlink the original one, avoiding a bad pointer dereference.

因此，基本策略是像这样对齐三个对象：

msg\_msg

msg\_msg

some\_interesting\_object

and proceed to free the first msg\_msg object and allocate the MSG\_CRYPTO key buffer into the hole it left behind. We corrupt the adjacent msg\_msg using the buffer overflow and subsequently leak data from some\_interesting\_objectusing the msgrcv system call with the MSG\_COPY flag set.

我选择泄露tty\_struct的数据，这些数据可以通过open()'ing /dev/ptmx轻松分配。此tty\_struct包含与tty相关的各种状态，并从以下开始：

struct tty\_struct {

int magic;

struct kref kref;

struct device \*dev; /\* class device or NULL (e.g. ptys, serdev) \*/

struct tty\_driver \*driver;

const struct tty\_operations \*ops;

int index;

...

}

这种结构的一个很好的特点是一开始的magic，它使我们能够通过将其与预期值TTY\_MAGIC（0x5401）进行比较来轻松确认泄漏的有效性。几个成员之后，我们找到了struct tty\_operations \*ops，该操作指向与各种操作相关的[函数指针列表](https://elixir.bootlin.com/linux/latest/source/include/linux/tty_driver.h#L247)。这指向内核.data中的某个地方，因此我们可以用它来击败KASLR！

根据泄露的tty\_struct是属于主pty还是从属pty（它们是成对分配的），我们最终会找到地址ofptmptm\_unix98\_ops或pty\_unix98\_ops。

作为额外的奖励，泄露tty\_struct还允许我们计算tty\_struct的地址，因为tty\_struct.ldisc\_sem.read\_wait.next指向自己！

**获得$RIP（或恐慌尝试）**

Naturally, the tty\_operations pointer is a nice target for overwriting to hijack the kernel execution flow. So in the next stage of the exploit we start by spraying a bunch of copies of a fake tty\_operations table. We can accurately guesstimate the address of one of these sprayed copies by utilizing the heap pointer we leaked in the previous step.

现在我们反复：分配msg\_msg，分配tty\_struct（s），免费msg\_msg，触发TIPC错误（希望）溢出到tty\_struct。为了确认我们实际上过度写了tty\_struct（第一部分），我们在打开/dev/ptmx获得的fd上调用ioctl，这将调用tty\_struct.ops.ioctl，如果我们设法劫持了这个对象的ops指针，应该可以让我们控制$RIP。如果不是这样，我们再次close()pty，以免耗尽资源。

**避免ROP（大部分）**

在内核中跳到哪里？我们可以在某个地方设置一个ROP堆栈，并将堆栈旋转到其中......但这很快就会变得混乱，特别是一旦您需要像从未发生过一样进行清理和恢复内核线程。

如果我们从ops列表中查看ioctl回调的原型，我们看到：

int (\*ioctl)(struct tty\_struct \*tty, unsigned int cmd, unsigned long arg);

我们可以轻松地从用户域调用中设置cmd和arg！

因此，我们有效地调用了一个任意函数调用，我们控制第2和第3个参数（分别为RSI和RDX）。嗯，cmd实际上被截断为32位，但这已经足够了。

让我们试着寻找一些允许我们进行任意写入的小工具序列：

$ objdump -D -j .text ./vmlinux \

| grep -C1 'mov %rsi,(%rdx)' \

| grep -B2 ret

..

ffffffff812c51f5: 31 c0 xor %eax,%eax

ffffffff812c51f7: 48 89 32 mov %rsi,(%rdx)

ffffffff812c51fa: c3 ret

..

这个很方便，它也清除了rax，以便漏洞可以判断调用是否成功。我们现在有一些任意的64位写入小工具！（对于任意值的某些定义，该值始终是32位控制+32位零，但无论如何）

**认识我的朋友：modprobe\_path**

好的，我们有这个磨损的任意写入小工具，我们可以反复调用，我们该覆盖什么？经典的内核漏洞将针对当前任务的cred结构，将特权提升到uid0。我们当然可以像构建任意写入一样构建任意读取机制..但是让我们尝试其他方法。

在各种情况下，内核将生成用户域进程（使用内核中的usermode\_helper基础设施）在认为必要时加载其他内核模块。当然，加载这些模块的过程是：modprobe。modprobe二进制文件的路径存储在名为modprobe\_path的全局变量中，特权用户可以通过sysfs节点/proc/sys/kernel/modprobe设置。

这是使用我们的写作小工具进行覆盖的完美候选者。如果我们覆盖此路径并说服内核我们需要一些额外的模块支持，我们可以调用任何可执行文件作为根！

这些modprobe场景之一是，当您尝试运行一个没有已知魔法的二进制文件时，我们在fs/exec.c中看到：

/\*

\* cycle the list of binary formats handler, until one recognizes the image

\*/

static int search\_binary\_handler(struct linux\_binprm \*bprm)

{

..

if (request\_module("binfmt-%04x", \*(ushort \*)(bprm->buf + 2)) < 0)

..

}

request\_module ends up invoking call\_modprobe which spawns a modprobe process based on the path from modprobe\_path.

**结束语**

我希望你喜欢阅读。请随时联系以指出任何不准确之处或反馈。完整的漏洞代码可以[在这里](https://haxx.in/files/blasty-vs-tipc.c)找到

成功的漏洞应该如下所示：

