

CVE-2018-9411：影响多个高权限Android服务的漏洞

[Stefano](#) / 2018-11-07 06:44:00 / 浏览数 3105 [技术文章](#) [技术文章 顶\(0\) 踩\(0\)](#)

原文地址：<https://blog.zimperium.com/cve-2018-9411-new-critical-vulnerability-multiple-high-privileged-android-services/>

## 前言

Zimperium

zLabs的研究员最近披露了一个影响多个高权限Android服务的关键漏洞。Google将其指定为CVE-2018-9411并在[7月安全更新](#)（2018-07-01补丁级别）中进行了修补，包

我为此漏洞编写了一个PoC，来演示如何使用它来从常规非特权应用程序中提升权限。

这篇博文将介绍漏洞和漏洞利用的技术细节。以及漏洞相关背景、漏洞相关的服务和漏洞详情。至于为什么选择某个特定服务作为其他服务的攻击目标，文中会详细阐述。

## Project Treble

Project

Treble对Android内部运作方式进行了大量更改。其中有一个较大的变更，使得许多系统服务都被碎片化了。以前，服务包含AOSP（Android开源项目）和供应商代码。在Treble之后，这些服务全部分为一个AOSP服务以及一个或多个供应商服务，称为HAL服务。有关更多背景信息，参阅[我的BSidesLV演讲](#)和[我之前的博文](#)。

## HIDL

Project Treble引入的分离机制导致了IPC（进程间通信）总数的增加；

先前在AOSP和供应商代码之间通过相同过程传递的数据现在必须通过AOSP和HAL服务之间的IPC。因为Android中的大多数IPC通过[Binder](#)通信，所以谷歌决定新的IPC也

但仅仅使用现有的Binder代码是不够的，Google也决定进行一些修改。首先，他们引入了[多Binder域](#)，以便将这种新型IPC与其他域分开。更重要的是，他们引入了[HIDL](#) - 一种通过Binder IPC传递的数据的新格式。这种新格式有一组新的库集，专门用于AOSP和HAL服务之间的IPC新Binder域。其他Binder域仍使用旧格式。

与旧HIDL格式相比，新HIDL格式的操作有点像层。两者的底层都是Binder内核驱动程序，但顶层是不同的。对于HAL和AOSP服务之间的通信，使用新的库集；

对于其他类型的通信，使用旧的库集。两组库都包含的代码非常相似，以至于某些原始代码甚至直接

被复制到新的HIDL库中（虽然我个人觉得这样复制粘贴并不好）。尽管每个库的用法并不完全相同（你不能简单地用一个替换另一个），但它仍然非常相似。

两组库集都表现为在Binder事务中作为C++对象传输的数据。这意味着HIDL为许多类型的对象引入了自己的新接口，包括了从相对简单的对象（如表示字符串的对象）到更

## 共享内存

Binder

IPC一个重要的方面是使用了共享内存。为了保持简单性和良好性能，Binder将每个事务限制为最大1MB。如果进程希望通过Binder在彼此之间共享大量数据，进程会使用并

为了通过Binder共享内存，进程利用了Binder用来共享文件描述符的功能。文件描述符可以使用mmap映射到内存，并允许多个进程通过共享文件描述符来共享相同的内存。

通过Binder共享内存是HIDL和旧库集之间不同实现的一个例子。在这两种情况下，最终操作都是相同的：一个进程将ashmem文件描述符映射到其内存空间，通过Binder将

在HIDL的情况下，共享内存的一个重要对象是hidl\_memory。如[源代码中所述](#)：“hidl\_memory是一种可用于在进程之间传输共享内存片段的结构”。

## 漏洞

让我们来仔细看看hidl\_memory的成员：

```
private:
    hidl_handle mHandle __attribute__((aligned(8)));
    uint64_t mSize __attribute__((aligned(8)));
    hidl_string mName __attribute__((aligned(8)));
};
```

来自system / libhidl / base / include / hidl / HidlSupport.h的片段（[源代码](#)）

- mHandle - 一个[句柄](#)，是一个包含文件描述符的HIDL对象（在这种情况下只有一个文件描述符）。
- mSize - 要共享的内存大小。
- mName - 应该代表内存的类型，但只有ashmem类型会起作用。


当通过HIDL中的Binder传输这样的结构时，复杂对象（如hidl\_handle或hidl\_string）有自己的自定义代码用于读写数据，而简单类型（如整数）则“原样”传输。这意味着它

这看起来就很奇怪了，为什么内存的大小是64位？为什么不像和旧的库集不一样？32位进程又如何处理这个问题呢？让我们看一下映射hidl\_memory对象的代码（针对ashmem）：

```
Return<sp<IMemory>> AshmemMapper::mapMemory(const hidl_memory& mem) {
    if (mem.handle()->numFds == 0) {
        return nullptr;
    }

    int fd = mem.handle()->data[0];
    void* data = mmap(0, mem.size(), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (data == MAP_FAILED) {
        // mmap never maps at address zero without MAP_FIXED, so we can avoid
        // exposing clients to MAP_FAILED.
        return nullptr;
    }

    return new AshmemMemory(mem, data);
}
```



来自system / libhidl / transport / memory / 1.0 / default / AshmemMapper.cpp的片段（[源代码](#)）

有趣！没有任何关于32位进程的处理，甚至没有提到内存大小是64位。

那这中间发生了什么呢？mmap签名中长度字段的类型是size\_t，这意味着它的位数与进程的位数相匹配。在64位进程中没有问题，一切都是64位。而在32位进程中，它的大小就是32位。也就是说，如果32位进程接收到大小大于UINT32\_MAX（0xFFFFFFFF）的hidl\_memory，则实际的映射内存区域将小得多。例如，对于大小为0x100001000的hidl\_memory，实际的映射内存区域将只有0x100000000。


## 寻找目标

既然有了漏洞，那现在就试着找到一个利用目标。

寻找符合以下标准的HAL服务：

1. 编译为32位。
2. 接收共享内存作为输入。
3. 在共享内存上执行边界检查时，也不会截断大小。例如，以下代码就没有风险，因为它对截断的size\_t执行边界检查：

```
sp<IMemory> mappedMemory = mapMemory(memory);
if (mappedMemory != NULL) {
    size_t memorySize = mappedMemory->getSize();
    doBoundsCheck(memorySize);
}
```



这些是触发漏洞的基本要求，但我认为还有一些可选的更可靠的目标：

1. 在AOSP中有默认接口。虽然供应商最终负责所有HAL服务，但AOSP确实包含某些供应商可以使用的默认接口。在许多情况下，当存在这样的接口时，供应商并不愿意修改它。应该注意的是，尽管HAL服务被设计为只能由其他系统服务访问，但事实并非如此。有一些特定的HAL服务实际上可以由常规的非特权应用程序访问，每个服务都有其自身的接口。
1. 可以从无特权的应用程序直接访问。否则一切都只能存在于假设中，因为我们将讨论这样一个目标，只有在您已经破坏了另一个服务的情况下才能访问它。

幸运的是，有一个满足所有这些要求的HAL服务：android.hardware.cas，AKA MediaCasService。

## CAS

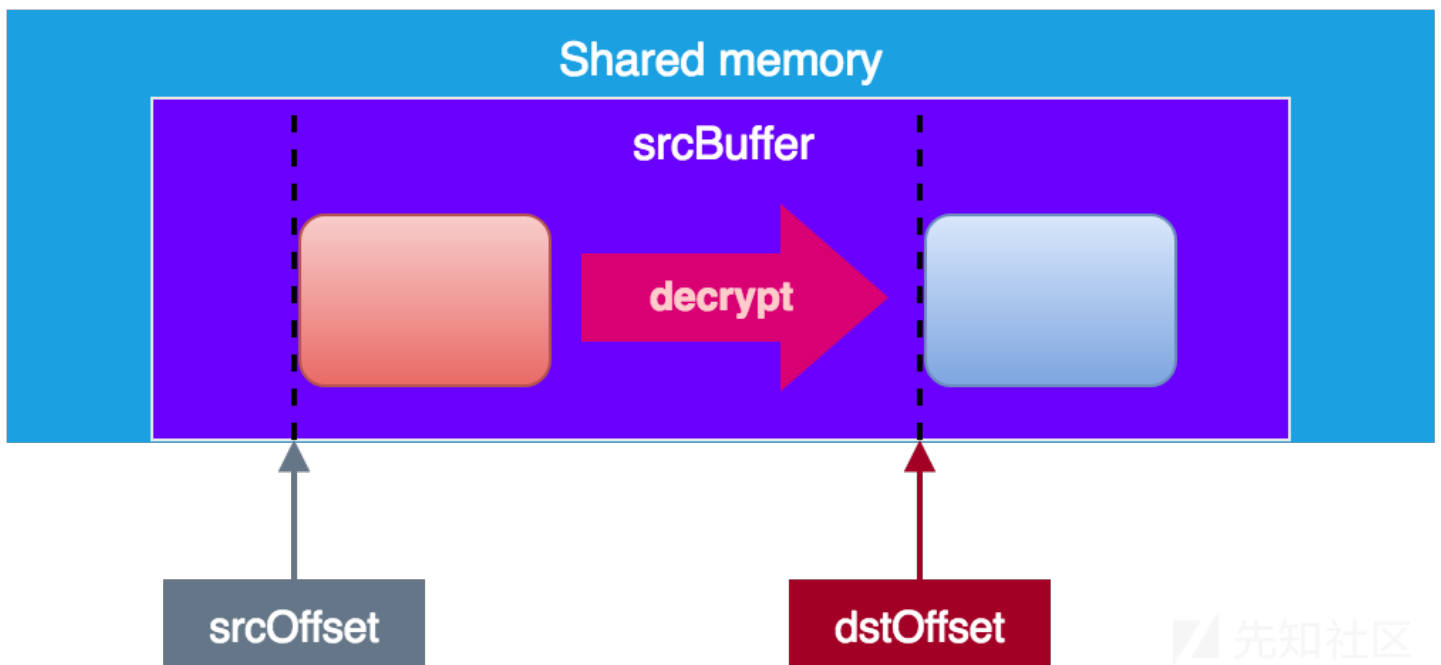
CAS（Conditional Access

System），表示条件访问系统。CAS本身大部分超出了本博文的范围，但总的来说，它与DRM类似（因此[差异并不是很明显](#)）。简单地讲，它的功能与DRM相同 - 存在需要解密的加密数据。

### MediaCasService

首先，MediaCasService确实允许应用程序解密加密数据。参阅[我以前的博客文章](#)，该博文处理了名为MediaDrmServer的服务中的漏洞，您可能会注意到与DRM进行比较。与MediaDrmServer略有不同的是其术语：API不是解密，而是称为解扰（尽管它们最终也会在内部对其进行解密）。

让我们看看[解扰方法](#)是如何运作的（这里省略了一小部分以简化操作）：



不出所料，数据通过共享内存共享。有一个缓冲区指示共享内存的相关部分（称为srcBuffer，但与源数据和目标数据相关）。在此缓冲区上，服务从其中读取源数据以及将这个漏洞看起来很赞！至少服务仅使用hidl\_memory的size来验证它是否适合共享内存，而不是其他的参数。在这种情况下，通过让服务认为我们的小内存区域跨越整个内存查看descramble的代码，验证确实存在这种情况。快速说明：函数validateRangeForSize只检查“first\_param + second\_param <= third\_param”，同时注意可能存在的溢出。

```
sp<IMemory> srcMem = mapMemory(srcBuffer.heapBase);

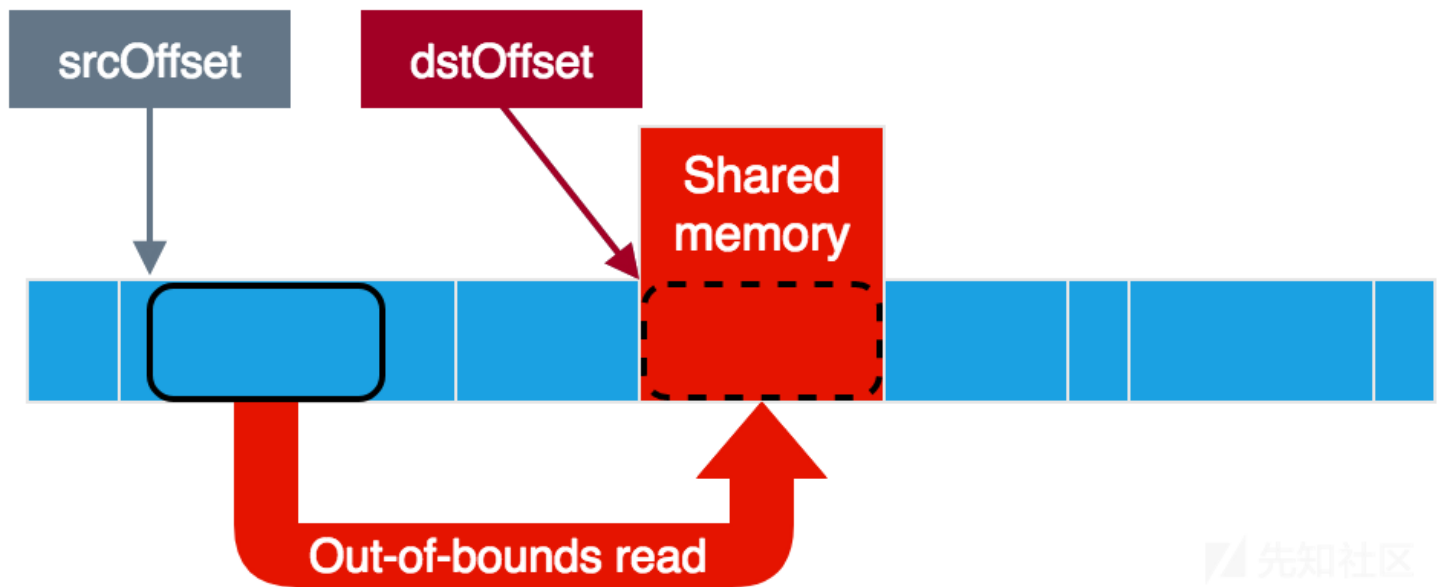
// Validate if the offset and size in the SharedBuffer is consistent with the
// mapped ashmem, since the offset and size is controlled by client.
if (srcMem == NULL) {
    ALOGE("Failed to map src buffer.");
    _hidl_cb(toStatus(BAD_VALUE), 0, NULL);
    return Void();
}

if (!validateRangeForSize(
    srcBuffer.offset, srcBuffer.size, (uint64_t)srcMem->getSize())) {
    ALOGE("Invalid src buffer range: offset %llu, size %llu, srcMem size %llu",
        srcBuffer.offset, srcBuffer.size, (uint64_t)srcMem->getSize());
    android_errorWriteLog(0x534e4554, "67962232");
    _hidl_cb(toStatus(BAD_VALUE), 0, NULL);
    return Void();
}
```

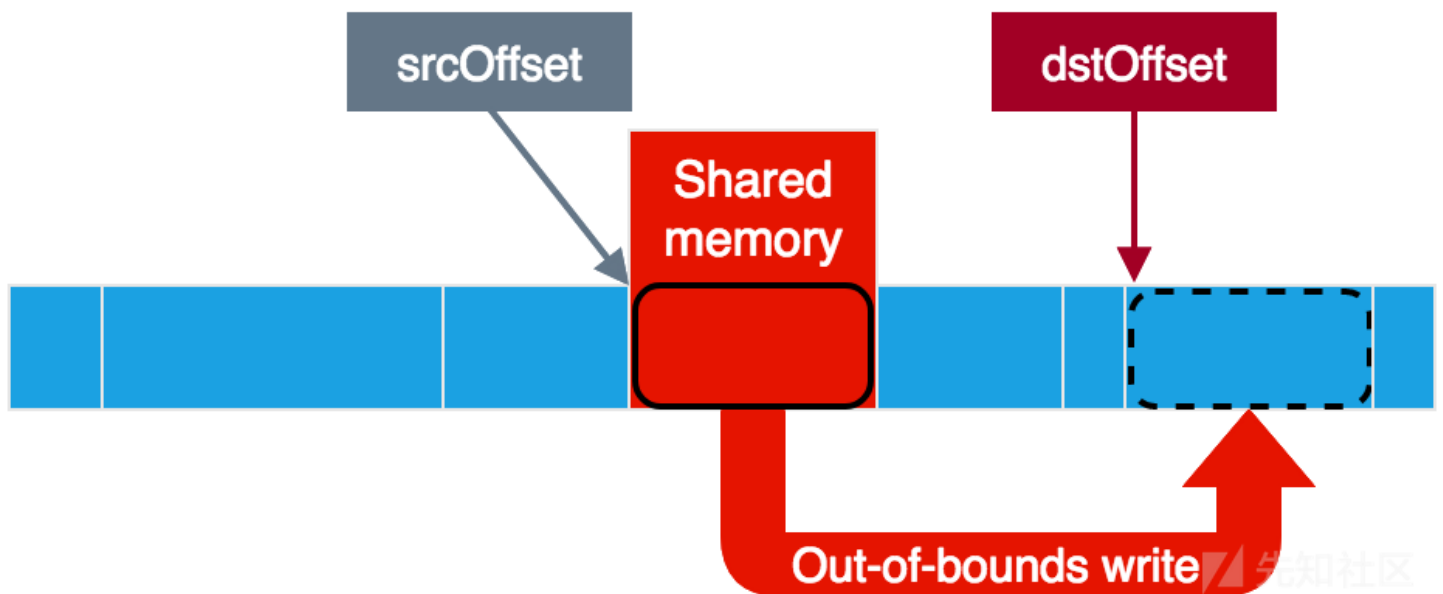
先知社区

来自hardware / interfaces / cas / 1.0 / default / DescramblerImpl.cpp的片段（[源代码](#)）

如上所示，代码根据hidl\_memory大小检查srcBuffer是否位于共享内存中。在此之后，不再使用hidl\_memory，并且针对srcBuffer本身执行其余检查。完美！我们所需要的



使用漏洞来越界读取



使用漏洞进行越界写入

## TEE设备

在使用这个原语编写漏洞之前，让我们考虑一下我们希望这个漏洞实现的目标。[此服务的SELinux规则](#)表明它实际上受到严格限制，并没有很多权限。不过，它还是有一个普

此权限非常有趣，因为它允许攻击者访问各种各样的东西：不同供应商的不同设备驱动程序，不同的TrustZone操作系统和大量的trustlet。在[我之前的博文](#)中，我已经讨论

虽然访问TEE设备确实可以做很多事情，但此时我只需证明我可以获得此访问权限。因此，我的目标是执行一个需要访问TEE设备的简单操作。在Qualcomm TEE设备驱动程序中，有一个相当简单的[ioctl](#)，用于查询设备上运行的QSEOS版本。因此，构建MediaCasService漏洞的目标是运行此ioctl并获取其结果。

## 利用

注意：我的exp针对特定设备和版本 - Pixel 2与2018年5月的安全更新（build fingerprint: "google/walleye/walleye:8.1.0/OPM2.171019.029.B1/4720900:user/release-keys"）。博客文章末尾提供了完整漏洞利用代码的链接。

到目前为止，我们拥有了对目标进程内存的完全读写权限。虽然这是一个很好的开头，但仍有两个问题需要解决：

- ASLR - 虽然我们有完全的读访问权限，但它只是相对于我们共享内存的映射位置；我们不知道它与内存中的其他数据相比在哪里。理想情况下，我们希望找到共享内存的地址以及其他我们感兴趣的数据的地址。
- 对于漏洞的每次执行，共享内存都会被映射，然后在操作后取消映射。无法保证共享内存每次都会映射到同一位置；另一个内存区域完全有可能在其执行后取代它。

让我们看一下这个特定版本的服务内存空间中链接器的一些内存映射：

```
f15e0000-f15e2000 rw-p 00000000 00:00 0 [anon:linker_alloc_vector]
f15e3000-f15e4000 rw-p 00000000 00:00 0 [anon:linker_alloc_small_objects]
f15e6000-f15e7000 rw-p 00000000 00:00 0 [anon:linker_alloc]
f15e7000-f15e8000 r--p 00000000 00:00 0 [anon:linker_alloc]
f15e8000-f15e9000 rw-p 00000000 00:00 0 [anon:linker_alloc_vector]
```

先知社区

链接器恰好在linker\_alloc\_small\_objects和linker\_alloc之间创建了2个内存页（0x2000）的小间隙。这些存储器映射的地址相对较高；此进程加载的所有库都映射到较低的地址。这意味着这个差距是内存中最高的差距。由于mmap的行为是尝试在低地址之前映射到高地址，因此任何映射2页或更少内存区域

让我们在间隙之后直接查看linker\_alloc中的数据：

```
0xf15e6000: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e6010: 0xf155aa60 0x00000001 0x00000000 0x00000000
0xf15e6020: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e6030: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e6040: 0xf15e6010 0x00000001 0x00000000 0x00000000
0xf15e6050: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e6060: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e6070: 0xf15e6040 0x00000001 0x00000000 0x00000000
0xf15e6080: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e6090: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e60a0: 0xf15e6070 0x00000001 0x00000000 0x00000000
0xf15e60b0: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e60c0: 0x00000000 0x00000000 0x00000000 0x00000000
0xf15e60d0: 0xf15e60a0 0x00000001 0x00000000 0x00000000
```

先知社区

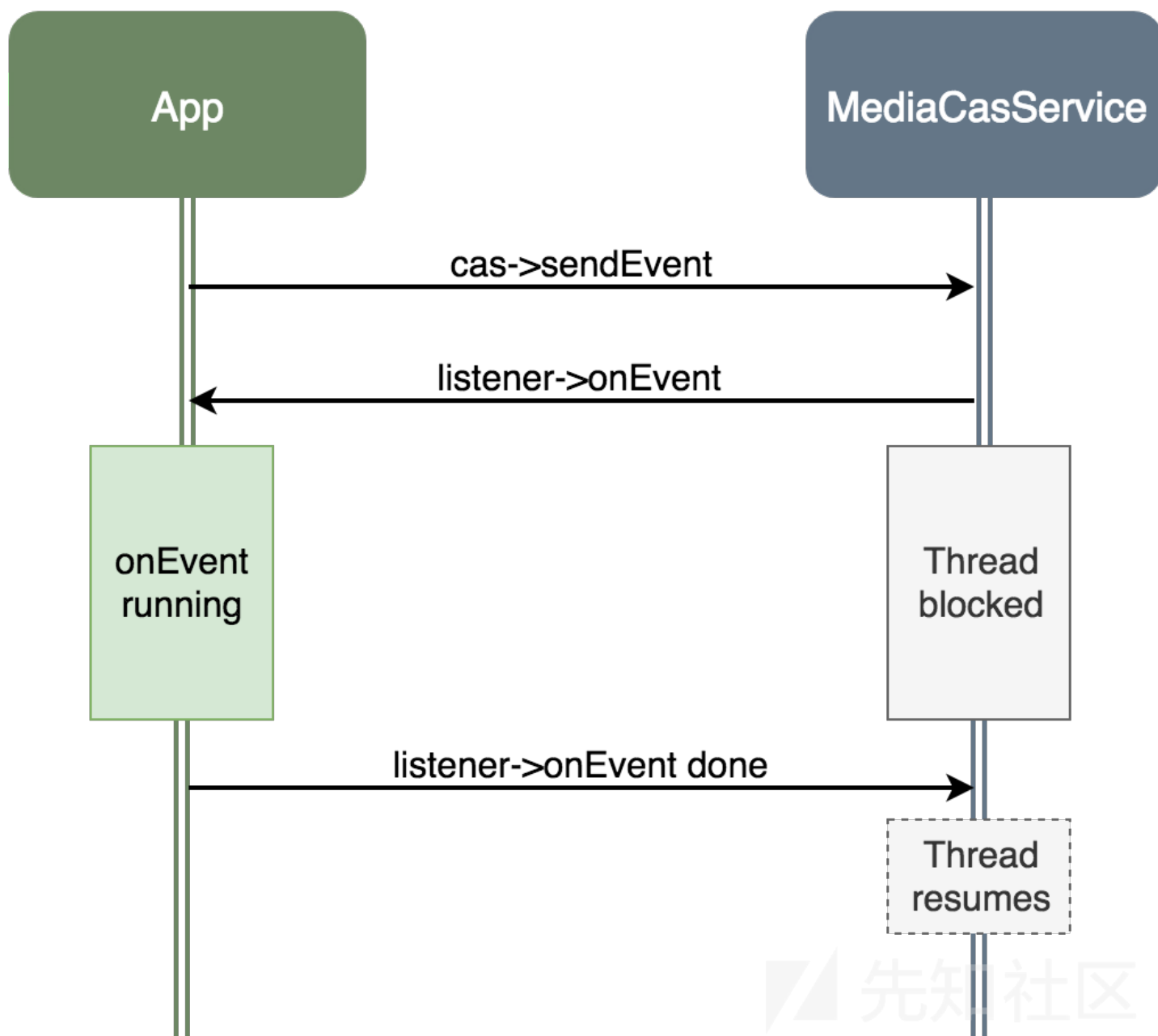
这里的链接器数据对我们非常有帮助；它包含可以轻松指示linker\_alloc内存区域地址的地址。由于漏洞可以让我们相对读取，并且我们已经得出结论，我们的共享内存将在此linker\_alloc之前直接映射，我们可以

到目前为止，我们解决了第二个问题，但只是解决了第一个问题的部分。我们确实有共享内存的地址，但没有其他感兴趣的数据。但是我们感兴趣的其他数据是什么？

### 劫持一个线程

#### MediaCasService

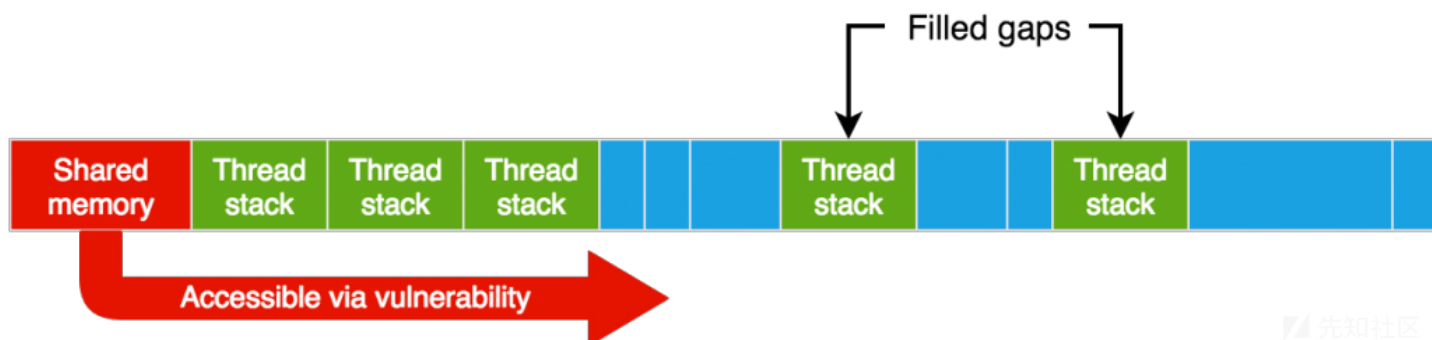
API的一部分是可以为客户端提供事件侦听器。如果客户端提供侦听器，则会在发生不同CAS事件时通知它。客户端也可以自己触发事件，然后将其发送回侦听器。当通过BI一个线程将被阻塞，等待监听器。



#### 触发事件的流程

我们可以在已知的预定线程中，阻止服务中的线程，让其等待我们的下一步操作。一旦我们有一个处于这种状态的线程，我们就可以修改它的堆栈来劫持它；然后，只有在我们完成后，我们才能通过完成处理事件来恢复线程。但是我们如何在内存中找到线程堆栈？

由于我们的确定性共享内存地址非常高，因此该地址与被阻塞的线程堆栈的可能位置之间的距离很大。因为ASLR的存在，使得通过相对于确定性地址找到线程堆栈的可能性相比于只将一个线程带到阻塞状态，我们更倾向使用多个（本例中5个）。这会导致创建更多线程，并分配更多线程堆栈。通过执行此操作，如果内存中存在少量线程堆栈大



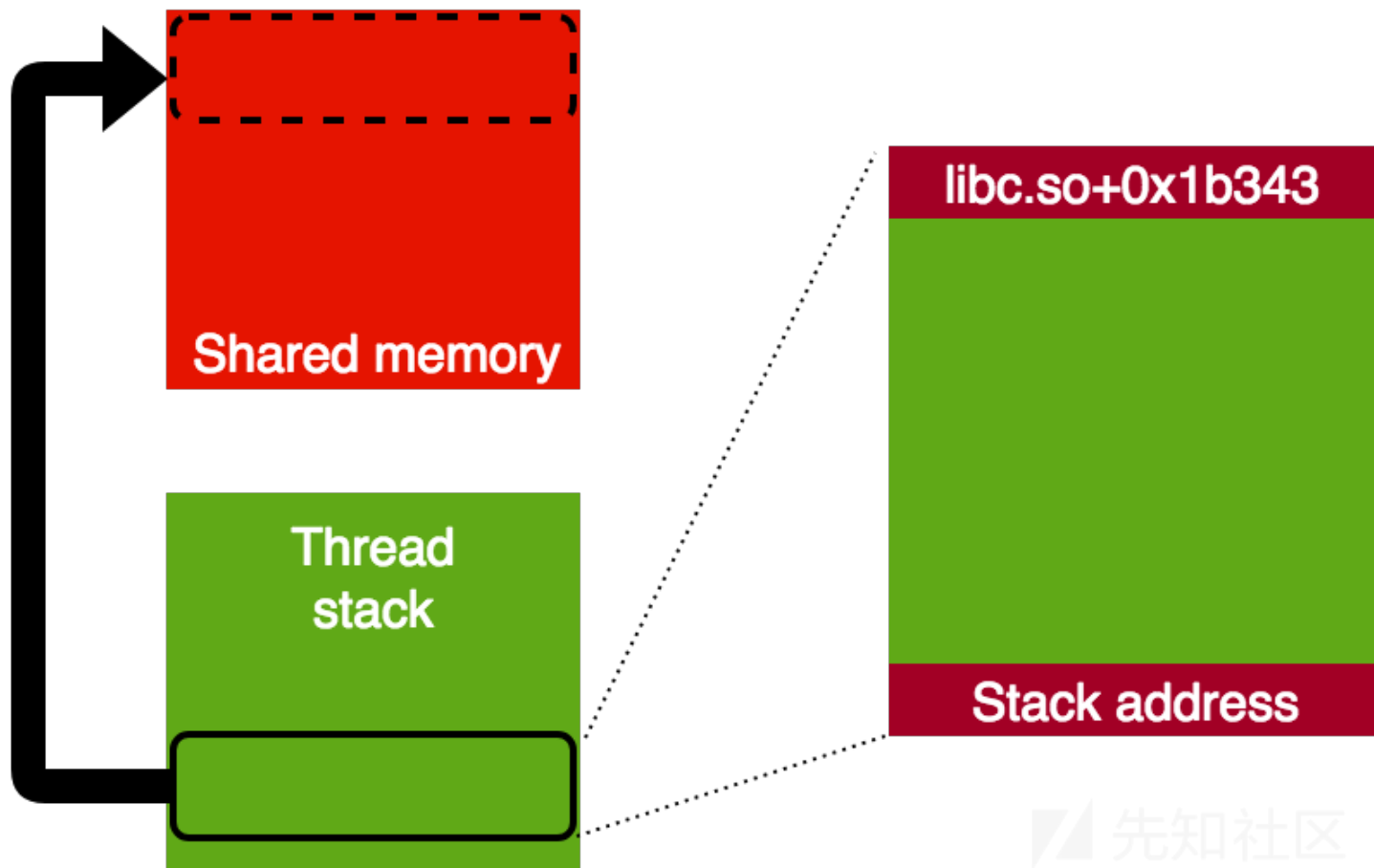
#### 填充间隙并映射共享内存后的MediaCasService内存映射

一个缺点是，有可能其他非预期的东西（如jemalloc堆）会被映射到中间，因此被阻塞的线程堆栈可能不是我们所需要的。有多种方法可以解决这个问题。我决定直接使用服务层。一旦我们的共享内存被阻塞的线程堆栈之前被映射，我们就使用该漏洞从线程堆栈中读取两样东西：



线程堆栈地址，使用pthread元数据，它位于堆栈本身之后的同一内存区域中。

libc映射到的地址，以便稍后使用libc中的小工具和符号构建ROP链（libc具有足够的小工具）。我们通过读取libc中特定点的返回地址（位于线程堆栈中）来实现这一点。



从线程堆栈读取的数据

从现在开始，我们可以使用漏洞读取和写入线程堆栈。我们有确切的共享内存位置地址和线程堆栈地址，因此通过使用地址之间的差异，我们可以从共享内存（具有确定性性

## ROP链

我们拥有可被恢复的被阻塞的线程堆栈的全部权限，因此下一步是执行ROP链。我们确切地知道要用我们的ROP链覆盖堆栈的哪个部分，因为我们知道线程被阻塞的确切状态。

遗憾的是，SELinux对此过程的限制使得我们无法将此ROP链转换为完全任意的代码执行。没有execmem权限，因此无法将匿名内存映射为可执行文件，并且我们无法控制

如前所述，目标是获得QSEOS版本。这里的代码本质上是由ROP链完成的：

```
int fd = open("/dev/qseecom", 0);
ioctl(fd, QSEECOM_IOCTL_GET_QSEOS_VERSION_REQ, stack_addr);
sleep(0xffffffff);
```

stack\_addr是堆栈内存区域的地址，它只是一个我们知道可写的地址，不会被覆盖（堆栈是从底部开始往上构建的），因此我们可以将结果写入该地址，然后使用漏洞读取

构建ROP链本身非常简单。libc中有足够的小工具来执行它，所有符号也都在libc中，我们已经拥有了libc的地址。

完成后，由于我们劫持了一个线程来执行我们的ROP链，因此进程处于一个不稳定的状态。为了使所有内容都处于干净状态，我们只使用漏洞（通过写入未映射的地址）使

## 写在后面

正如我之前在我的BSidesLV演讲和我之前的博客文章中所讨论的那样，[谷歌宣称Project Treble有利于Android的安全性](#)。虽然在许多情况下都是如此，但这个漏洞却是Project Treble与其初衷背道而驰的一个例子。此漏洞位于一个特定的库中，这个库是作为Project Treble的一部分专门引入的，在之前的库中不存在（虽然这些库几乎完全相同）。这次的漏洞存在于常用的库中，因此它会影响许多高权限服务。

GitHub上提供了完整的[exp代码](#)。注意：该漏洞仅用于教育或防御目的；不适用于任何恶意或攻击性用途。

[上一篇：记SECCON 2018的一道智能...](#) [下一篇：伊朗用户的Instagram电报安...](#)

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](#)