voucher_swap：利用iOS 12中的MIG引用计数

ret2nullptr / 2019-07-11 09:05:00 / 浏览数 5166 安全技术 漏洞分析 顶(0) 踩(0)

本文是翻译文章，原文链接：https://googleprojectzero.blogspot.com/2019/01/voucherswap-exploiting-mig-reference.html

在这篇文章中，我将描述我如何发现和利用CVE-2019-6225，这是XNU的task_swap_mach_voucher()函数中的MIG引用计数漏洞。我们将看到如何利用iOS 12.1.2上的这个错误来构建虚假内核任务端口，使我们能够读写任意内核内存。（这个错误是由@S0rryMybad独立发现的）

在后面的文章中，我们将看看如何使用这个错误作为分析和绕过Apple在A12设备(例如iPhone XS)上ARMv8.3■■■■(PAC)的起点。

一个奇怪的发现

MIG是一个产生Mach消息解析代码的工具，错误的MIG语义造成漏洞并不是什么新鲜的事：例如，伊恩·比尔async_wake利用一个漏洞，即IOSurfaceRootUserClient将
11.1.2 MIG语义管理。

大多数先前与MIG相关的问题是MIG服务例程不遵守对象生命周期和所有权的语义的结果。通常，MIG所有权规则表示如下：

1. 如果MIG服务例程返回成功，则它将获取所有传入资源的所有权。
2. 如果MIG服务例程返回失败，那么它将不会获得传入的资源的所有权。

不幸的是，正如我们将要看到的，此描述并未涵盖MIG管理的内核对象的完整复杂性，这可能导致意外错误。

这是_Xsemaphore_destroy()的相关代码：

```
task = convert_port_to_task(In0P->Head.msgh_request_port);

   OutP->RetCode = semaphore_destroy(task,
           convert_port_to_semaphore(In0P->semaphore.name));
   task_deallocate(task);
#if __MigKernelSpecificCode
   if (OutP->RetCode != KERN_SUCCESS) {
       MIG_RETURN_ERROR(OutP, OutP->RetCode);
   }

   if (IP_VALID((ipc_port_t)In0P->semaphore.name))
       ipc_port_release_send((ipc_port_t)In0P->semaphore.name);
#endif /* __MigKernelSpecificCode */
```

函数convert_port_to_semaphore()接受Mach端口并在底层信号量对象上生成引用，而不消耗端口上的引用。如果我们假设上面代码的正确实现，没有泄漏或消耗额外

1. 成功时，semaphore_destroy()应该使用信号量引用。
2. 如果失败，semaphore_destroy()应该仍然消耗信号的引用计数值。

因此，semaphore_destroy()似乎不遵循MIG语义的传统规则：正确的实现总是取得信号量对象的所有权，无论服务例程是返回成功还是失败。

这当然引出了一个问题：管理MIG语义的完整规则是什么？是否存在违反这些其他MIG规则的代码实例？

糟糕的swap

在我对扩展的MIG语义的研究中，我发现了函数task_swap_mach_voucher()。这是osfmk/mach/task.defs中的MIG定义

```
routine task_swap_mach_voucher(
               task            : task_t;
               new_voucher     : ipc_voucher_t;
       inout   old_voucher     : ipc_voucher_t);
```

这是来自_Xtask_swap_mach_voucher()的相关代码，自动生成的MIG包装器：

```
mig_internal novalue _Xtask_swap_mach_voucher
      (mach_msg_header_t *InHeadP, mach_msg_header_t *OutHeadP)
{
...
  kern_return_t RetCode;
  task_t task;
  ipc_voucher_t new_voucher;
  ipc_voucher_t old_voucher;
```

```
...
  task = convert_port_to_task(In0P->Head.msgh_request_port);

  new_voucher = convert_port_to_voucher(In0P->new_voucher.name);

  old_voucher = convert_port_to_voucher(In0P->old_voucher.name);

  RetCode = task_swap_mach_voucher(task, new_voucher, &old_voucher);

  ipc_voucher_release(new_voucher);

  task_deallocate(task);

  if (RetCode != KERN_SUCCESS) {
      MIG_RETURN_ERROR(OutP, RetCode);
  }
...
  if (IP_VALID((ipc_port_t)In0P->old_voucher.name))
      ipc_port_release_send((ipc_port_t)In0P->old_voucher.name);

  if (IP_VALID((ipc_port_t)In0P->new_voucher.name))
      ipc_port_release_send((ipc_port_t)In0P->new_voucher.name);
...
  OutP->old_voucher.name = (mach_port_t)convert_voucher_to_port(old_voucher);

  OutP->Head.msgh_bits |= MACH_MSGH_BITS_COMPLEX;
  OutP->Head.msgh_size = (mach_msg_size_t)(sizeof(Reply));
  OutP->msgh_body.msgh_descriptor_count = 1;
}
```

再一次，假设正确的实现不会泄漏或消耗额外的引用计数值，我们可以推断出`task_swap_mach_voucher()`的以下预期语义：

1. `task_swap_mach_voucher()`没有对`new_voucher`的引用; 该`new_voucher`引用是借来的，不应该被消耗掉。
2. `task_swap_mach_voucher()`包含对应该使用的`old_voucher`输入值的引用。
3. 失败时，`old_voucher`的输出值不应对指向的凭证对象持有任何引用。
4. 成功时，输出值`old_voucher`持有一个凭证的引用，而它由`task_swap_mach_voucher()`到_`Xtask_swap_mach_voucher()`得到，其经由后者消耗`convert_vou`

考虑到这些语义，我们可以与实际实现进行比较。这是来自`XNU 4903.221.2`的[osfmk/kern/task.c](#)的代码，可能是一个占位符实现：

```
kern_return_t
task_swap_mach_voucher(
      task_t          task,
      ipc_voucher_t   new_voucher,
      ipc_voucher_t   *in_out_old_voucher)
{
  if (TASK_NULL == task)
      return KERN_INVALID_TASK;

  *in_out_old_voucher = new_voucher;
  return KERN_SUCCESS;
}
```

此实现不符合预期的语义：

1. `in_out_old_voucher`的输入值是`task_swap_mach_voucher`拥有的凭证引用。但是无条件地覆盖它，而不是首先调用`ipc_voucher_release()`，`task_swap_m`
2. 值`new_voucher`不归`task_swap_mach_voucher()`所有，但它在`in_out_old_voucher`的输出值中返回。这会消耗不是`task_swap_mach_voucher()`持有的凭证

因此，`task_swap_mach_voucher()`实际上包含两个引用计数问题！

我们可以通过使用凭证作为第三个参数调用`task_swap_mach_voucher()`来泄漏凭证对象的引用计数，并且我们可以通过将凭证作为第二个参数来减少凭证对象的引用计

进一步的调查显示，`thread_swap_mach_voucher()`包含一个类似的漏洞，但iOS 12中的更改使漏洞无法利用。

凭证的其他相关内容

为了掌握这个漏洞的影响，了解更多关于Mach■■的信息是有帮助的，尽管全部细节对于利用并不重要。
Mach凭证由内核中的`ipc_voucher_t`类型表示，具有以下结构定义：

```
/*
* IPC Voucher
*
* Vouchers are a reference counted immutable (once-created) set of
* indexes to particular resource manager attribute values
* (which themselves are reference counted).
*/
struct ipc_voucher {
  iv_index_t      iv_hash;        /* checksum hash */
  iv_index_t      iv_sum;         /* checksum of values */
  os_refcnt_t     iv_refs;        /* reference count */
  iv_index_t      iv_table_size;  /* size of the voucher table */
  iv_index_t      iv_inline_table[IV_ENTRIES_INLINE];
  iv_entry_t      iv_table;       /* table of voucher attr entries */
  ipc_port_t      iv_port;        /* port representing the voucher */
  queue_chain_t   iv_hash_link;   /* link on hash chain */
};
```

正如注释所示，IPC凭证代表一组任意属性，这些属性可以通过Mach消息中的发送权限在进程之间传递。Mach■■的主要相关者似乎是Apple的libdispatch库。

与我们相关的ipc_voucher的唯一字段是iv_refs和iv_port。其他字段与管理凭证对象的全局列表和存储凭证所代表的属性有关，这两个凭证都不会在漏洞利用中使用。

从iOS 12开始，iv_refs的类型为os_refcnt_t，它是32位引用计数，允许值范围为1-0x0fffffff（即7个f，而不是8个f）。试图保留或释放超出此范围的引用计数将引发错误。

iv_port是指向ipc_port对象的指针，该对象表示此用户空间的凭证。只要在iv_port设置为NULL的ipc_voucher上调用convert_voucher_to_port()，它就会被

要创建Mach凭证，可以调用host_create_mach_voucher()陷阱。此功能采用描述凭证属性的"配方"，并返回代表凭证的凭证端口。但是，由于凭证是不可变的，因此有

这是不合适的！

有许多不同的方法可以利用这个bug，但是在这篇文章中我将讨论我的最爱：增加一个外部的Mach端口指针，使其指向管道缓冲区。

现在我们已经了解了漏洞是什么，现在是时候确定我们可以用它做什么了。正如您所期望的那样，一旦ipc_voucher的引用计数降至0，就会被释放内存。因此，我们可以利

但是，释放凭证仅在随后以有趣的方式重新使用已取用的凭证时才有用。这有三个组件：存储指向已释放凭证的指针，使用有用的东西重新分配已释放的凭证，以及重用存储

让我们考虑第一步，存储指向凭证的指针。内核中有一些地方直接或间接存储凭证指针，包括structipc_kmsg的ikm_voucher字段和struct thread的ith_voucher字段。其中，最容易使用的是ith_voucher，因为我们可以通过调用thread_get_mach_voucher()和thread_set_mach_voucher()直接从

存储一个对凭证的引用，接着我们可以使ith_voucher指向一个释放的凭证，然后使用我们的bug删除刚才添加的引用，最后在用户空间中取消分配凭证端口以释放凭证。

接下来考虑如何使用有用的东西重新分配凭证。ipc_voucher对象存在于他们自己的zalloc区域ipc.vouchers中，因此我们可以轻松地将我们释放的凭证与另一个凭证

为了弄清楚我们应该重新分配什么类型的对象，首先检查我们将如何在线程的ith_voucher字段中使用凭证的悬空指针是有帮助的。我们有几个选项，但最简单的方法是调

但是，使thread_get_mach_voucher()如此有用的原因是它将凭证的Mach端口返回给用户空间。我们有两种方法可以利用它。如果释放的ipc_voucher对象的iv_por，则该指针直接被解释为ipc_port指针，thread_get_mach_voucher()将其作为Mach发送权返回给我们。另一方面，如果iv_port为NULL，则convert_voucher_to_port()将返回一个新分配的凭证端口，该端口允许我们继续操纵从用户空间释放的凭证的引用计数。

这让我产生了使用外线端口重新分配凭证的想法。在消息中发送大量Mach端口权限的一种方法是在端口描述符中列出端口。当内核在外部端口描述符中复制时，它会分配一。

由于我们可以控制数组中的哪些元素是有效端口以及哪些是MACH_PORT_NULL，因此我们可以确保使用NULL覆盖凭证的iv_port字段。这样，当我们在用户空间中调用thr
再次使用引用计数bug来修改 释放的凭证的**iv_refs**字段，这将改变与iv_refs重叠的任何数量的外部端口指针的值。

当然，我们还没有解决确保iv_refs字段有效的问题。如前所述，如果我们想要重用freedipc_voucher而不触发内核恐慌，iv_refs必须在1 - 0x0fffffff的范围内。

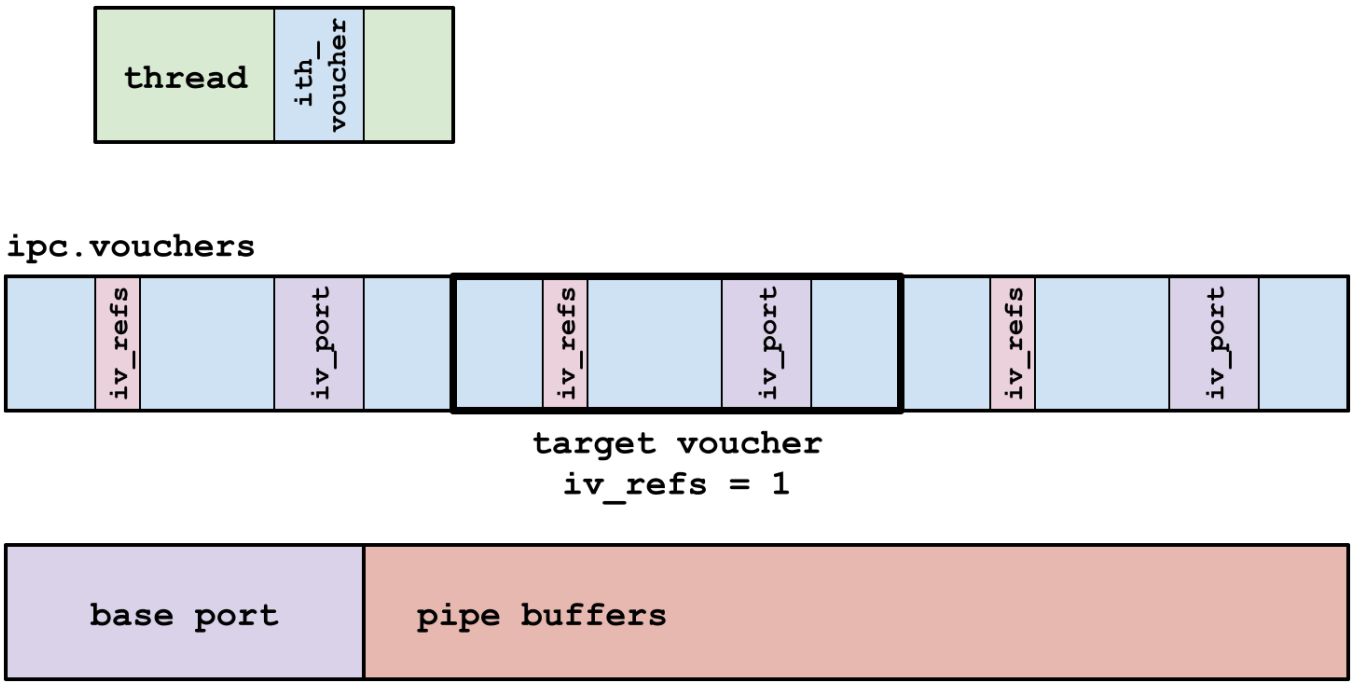
该ipc_voucher结构是为0x50字节，iv_refs字段是在偏移0x8中;
由于iPhone是little-endian(■■■)，这意味着如果我们使用一系列外部端口重新分配释放的凭证，iv_refs将始终与ipc_port指针的低32位重叠。让我们调用与iv_refs基本端口重叠的Mach 端口。使用MACH_PORT_NULL或MACH_PORT_DEAD作为基本端口将导致iv_refs为0■0xffffffff■两者都无效。因此，剩下的唯一选择是使用一个真实的Mach端口作为基本端口，使得iv_refs与真正的低32位覆盖ipc_port指针。
这很危险，因为如果基本端口地址的低32位是0或大于0x0fffffff，则访问已经释放的凭证将会发生混乱。幸运的是，最近的iOS设备上的内核堆分配表现得非常好：zalloc页面将从低地址开始从0xffffffe0xxxxxxxx范围分配，因此只要堆

利用思路

这为我们提供了利用此漏洞的思路：

1. 分配一页Mach凭证。

2. 在线程的ith_voucher字段中存储指向目标凭证的指针，并使用漏洞删除添加的引用。
3. 取消分配凭证端口，释放所有凭证。
4. 强制区域gc并使用一系列外部端口重新分配
   已释放凭证的页面。使用指向基本端口的指针的低32位重叠目标凭证的**iv_refs**字段，并将凭证的**iv_port**字段与**NULL**重叠。
5. 调用thread_get_mach_voucher()以检索与外部端口重叠的凭证的凭证端口。
6. 再次使用此漏洞修改重叠凭证的iv_refs字段，该字段会更改外部基本端口指针，使其指向其他位置。
7. 一旦我们收到包含外部端口的Mach消息，我们就会获得一个解释为ipc_port的任意内存的发送权。



管道相关利用(Pipe dreams)

那么我们应该如何获得发送权？理想情况下，我们能够完全控制我们收到的假ipc_port的内容，而不必通过解除分配然后重新分配支持假端口的内存来玩冒险游戏。

Ian实际上在他的multi_path和empty_list漏洞利用管道缓冲区中提出了一个很好的技术。到目前为止，我们的漏洞利用允许我们修改指向基本端口的外部指针，使其指向其

此时，我们可以在用户空间中收到包含外部端口的消息。此消息将包含对**ipc_port**的发送权限，该ipc_port与我们的一个管道缓冲区重叠，因此我们可以通过读取和写入

TFP0

一旦我们拥有一个完全可控的**ipc_port**对象的发送权限，漏洞利用流程就清晰了。

我们可以使用相同的旧pid_for_task()技巧构建一个基本的内核内存读取原语：将我们的端口转换为伪任务端口，以便伪任务的bsd_info字段（它是指向**proc**结构的指

我们可以通过将我们的假任务结构体放在我们发送到假端口的Mach消息中来解决这个问题，之后我们可以读取与端口重叠的管道缓冲区，并从端口的ip_messages获取包含

这种方法的一个不幸后果是它为每个4字节读取泄漏了一个**ipc_kmsg**结构。因此，我们希望尽可能快地构建一个更好的读取原语，然后释放所有泄露的消息。

为了获得管道缓冲区的地址，我们可以利用它驻留在基本端口地址的已知偏移量。我们可以在虚假端口上调用mach_port_request_notification()来添加一个请求，
name)，就会通知基本端口。这会导致伪端口的ip_requests字段指向一个新分配的数组，该数组包含指向基本端口的指针，这意味着我们可以使用内存读取原语来读出基

此时我们可以在管道缓冲区内构建一个伪内核任务，为我们提供完整的内核读、写。接下来，我们使用mach_vm_allocate()分配内核内存，在该内存中编写一个新的伪内

这就是完全利用！您可以在此处找到适用于iPhone XS，iPhone XR和iPhone
8的漏洞利用代码：voucher_swap。源代码中提供了对漏洞利用技术的更深入，逐步的技术分析。

撞bug

我在2018年12月6日向Apple报告了这个漏洞，截至12月19日Apple已经发布了iOS 12.1.3
beta版本16D5032a，修复了这个问题。由于这对Apple来说是一个令人难以置信的快速转变，我怀疑这个错误是由其他一方首先发现并报告的。

后来我了解到这个bug是由Qihoo 360 Vulcan Team的Qixun
Zhao（@S0rryMybad）独立发现和利用的。有趣的是，我们都是通过semaphore_destroy()引发了这个错误;
因此，我不会惊讶地发现这个错误在被修复之前已广为人知。Mybad将此漏洞用作天府杯远程越狱的一部分， 你可以阅读他获得tfp0的策略。

总结

这篇文章研究了P0问题1731的发现和利用，这是一个关于IPC凭证引用计数问题，其根源在于未能跟踪外部对象的MIG语义。在新引导后几秒钟运行时，此处讨论的漏洞利

在某种程度上，令人惊讶的是，这种"容易"的漏洞仍然存在：毕竟，XNU是开源的，并且对这样的有价值的错误进行了严格的审查。但是，MIG语义非常不直观，并且与编

这个错误也是一个很好的提醒，占位符代码也可能引入安全漏洞，应该像功能代码一样严格审查，无论它看起来多么简单。

最后，值得注意的是，在利用这个漏洞时，我所遇到的最大问题是，允许的引用计数值的范围有限，在12版之前的iOS版本中甚至都不是问题。在早期的平台上，这个bug总

我的下一篇文章将展示如何利用这个漏洞来分析Apple的指针认证实现，最终形成一种技术，允许我们为使用A键签名的指针伪造PAC。这足以通过JOP调用任意内核函数或

上一篇：Redis 4.x RCE分析 下一篇：使用Microsoft域绕过防火墙

1. 0 条回复
   - 动动手指，沙发就是你的了！