task_t 指针重大风险预报-Exploit 篇

引言：大家都知道知名意大利天才少年 Luca 放出来的针对<=10.2 版本的 yalu 越狱使用的是对 kernel port 的 buffer overflow 拿到了 kernel_task_port，本文对类似的 task_t 指针做出了针对性的分析，从 mach 端口背景知识，到 IOkit 的相关处理，最终如何利用在堆栈上写出 Exploit，最终甚至给苹果团队给出了修复漏洞的建议，由浅入深，偏辟入里，值得推荐。

本文分三篇推出，分别是分析篇，Exploit 篇，和修复建议篇。

译者注：

- 一些诸如 bug，exploit 之类的行话选择性的翻译，这通常取决于句子的流畅性。
- 不确定的地方在括号中附注了原句
- 超链接附在括号内，方便查看

by ruanbonan

Exploit 篇

查看文档

关于 OS X 内核扩展的资料不是很多。苹果在他们的开发者网站上发布了名为 AppleSamplePCI 的 kext 样本，这为多样化的 IOKit 设计样式提供了示例。下面是 AppleSamplePCI.kext 对于 initWithTask 的实现：

```
bool SamplePCIUserClientClassName::initWithTask(
 task_t owningTask,
 void* securityID,
 UInt32 type,
 OSDictionary* properties)
{
 bool success = super::initWithTask(owningTask,
                                    securityID,
                                    type,
                                    properties);
 fTask = owningTask;
 fDriver = NULL;
 return success;
}
```

这个样本的 userclient 把 owningTask 参数存储在 fTask 成员变量中，没有引用。没有引用就无法保证在这个方法返回后 fTask 指向的任务结构体后不被释放。观察剩下的 kext 样本，我们可以发现一些外部方法使用 fTask 指针来创建内存描述符

如果我们可以使 fTask 指针指向的任务结构体释放，那么它们就是在使用一个悬挂指针了。

在 IDA 中打开一些其他 OS X kext 文件可以很清楚地看到它们中的许多都遵循了一个反面模式：保存一个 task_t 指针，却没有引用。

创建一个 task_t 类型的悬挂指针

mach 消息提供了非常灵活强大的 IPC 构件。你可以做一些灵活的事，比如向你拥有发送或接受权限的 mach 端口发送其他进程的“发送权限”。

因为任务端口给了你对于其他任务的完全控制权，向一个任务端口请求其他任务的 api(task_for_pid)是具有特权的，但是因为所有任务都它们自己的任务端口的发送权，所以，如果我们可以在两个任务中执行代码，我们可以将第二个任务的任务端口的发送权发给第一个任务端口。

在这种情况下，我们使用 Robert Sesek 描述的技术（链接：https://robert.sesek.com/2014/1/changes_to_xnu_mach_ipc.html）通过把发送权隐藏在特定的 bootstrap_port 端口上，然后在父进程和它 fork 出的子进程之间创建一个共享 mach 端口。在 fork 之后，子进程可以恢复这个隐藏端口，返回 bootstrap 端口并设置一个双向的 IPC 通道，这样一来，它就可以通过这个通道来把它的任务端口发回给父进程。

在这个存在漏洞的反面模式中触发 UAF 的方法如下：

- 父进程 fork 出子进程
- 子进程把它的任务端口发回给父进程
- 子进程自旋
- 父进程收到子进程的任务端口，创建一个有漏洞的 IOKit userclient，它将子进程任务端口作为 owningTask 传递
- 父进程销毁它对于子进程任务端口的发送权
- 父进程杀死子进程，释放子进程的任务结构体
- 父进程有了一个带有任务结构体类型悬挂指针的 userclient

第一个 exploit

查看 IOKit 包含有这个 bug 的驱动，其中一个特别有趣——IOSurfaceRootUserClient。下面是苹果开发者文档对于 IOSurface 的叙述：

IOSurface 框架提供了适用于跨进程共享的框架缓冲对象。它被广泛用于允许应用程序移动一个复杂的镜像解压文件，以及将逻辑流放入一个单独进程来增强安全性。

实际上 IOSurfaces 仅仅用来包裹共享内存缓冲区。另外，在 OS X 上我们可以与来自 Safari 内部渲染沙盒以及 ChromeGPU 沙盒的 IOSurface 内核扩展通信。

IOSurfaceRootUserClient 类也有反面模式，与我们在 AppSamplePCI 客户端看到的相同，userclient 把 owningTask 指针的拷贝当做一个成员变量存储，却没有引用。通过一些逆向，我们知道 IOSurfaceRootUserClient 的外部方法是 create_surface，它接受一个键值对作为参数来创建共享内存对象，其他进程可以把这个对象映射到它们自己的地址空间中。通过传递下面的键和值，我们可以获得 IOSurfaceRootUserClient 来把 IOSurface 中一个已存在的用户空间页包裹起来，而不是重新分配一个缓冲区。

```
IOSurfaceAddress:   base_address
IOSurfaceAllocSize: size
IOSurfaceIsGlobal:  true
```

IOSurface 对象实际上仅仅通过调用下面的代码来把 IOSurface::allocate 分配的 IOMemoryDescriptor 包裹起来：

```
IOMemoryDescriptor *
IOMemoryDescriptor::withAddressRange(
 mach_vm_address_t address,
 mach_vm_size_t length,
 IOOptionBits   options,
 task_t         task);
```

IOMemoryDescriptor::withAddressRange 的最后一个 task_t task 参数定义了文件描述符应该为哪一个任务的虚拟内存创建。IOSurface 在这里传递储存了 owningTask 的拷贝的成员变量，却没有对它加以引用 ！如果我们可以让这个任务结构体在初始任务退出时内存先被释放，另一个任务开始时再次分配，并被用作一个具有更高权限的任务的任务结构体，那么 IOMemoryDescriptor 会相信它正在包裹当前进程地址空间的一部分，然而，事实上它正在包裹 IOMemoryDescriptor 内支持这个 IOSurface 的另一个更高权限任务的虚拟内存。

设置 IOSurfaceIsGlobal=true 使得这一个其他进程可以接触这一个 surface，因此通过对另一个拥有我们自己的合法任务端口作为 owningTask 的 IOSurfaceRootUserClient 调用外部方法 lookup_surface，我们可以建立一个 primitive，它允许我们把其他进程地址空间的任意位置映射到我们自己的上面

由于 IOMemoryDescriptor 实际上创建了那些我们可以写入的页的共享内存映射，这些写入也会反映到其他进程中。IOSurfaceRootUserClient 不允许我们把受害的可执行页映射过来，但是我们仍然可以映射一些东西，比如库的__DATA 段。而这是很容易实现的，因为共享库缓存对于所有进程都是相同的虚拟地址。

把 Exploit 合在一起

我们需要设法让任务指针被更高权限的进程重用，然后我们需要一些东西来覆盖目标，已达到代码执行的目的。

任务指针从它们的内核堆空间分配出来，这一点大大简化了事情。我们可以仅仅杀死一个子进程，然后 fork 并 exec 一些带有 suid 标志的二进制程序，它们很可能重用悬挂指针 task_t 指向的空间。

为了覆盖目标，我选择 libc 中的__cleanup 指针作为目标。在进程退出时，它将被调用。我们使用一些技巧来在程序退出前阻塞它，这可以通过把它的 stderr 文件描述符设置成一个满的 pipe 并强制它写入错误信息实现，从而给我们大量的时间来利用父进程中的 bug，并在清空父进程的 pipe 之前覆盖__cleanup 指针。我选择把这个函数指针指向一个 gadget，这个 gadget 的功能是给 RSP 寄存器加上一个大的常数，然后返回。这样做会把栈指针向上移动到 argv，当我们执行这个程序时，我在那里放了一个 ROP 栈，来调用 setuid(0)并执行/bin/bash。ROP 载荷附加有很长的 ret-slide 前缀，因此它在大多数版本的 OS X 上都应该是稳定有效的。

你可以下载这个 Exploit(链接：https://bugs.chromium.org/p/project-zero/issues/attachment?aid=237183)

并查看初始的 bug 报告（链接：https://bugs.chromium.org/p/project-zero/issues/detail?id=831）

由于除了 root 以外，这个 bug 也允许我们获得其他任何权限，所以很容易利用它来绕过 OS X 上的内核代码签名，并加载一个未签名的内核扩展。方法之一，参考 CVE-2016-1757 的 Exploit（链接：https://googleprojectzero.blogspot.ch/2016/03/race-you-to-kernel.html）

虽然这个 Exp 使用了 fork 和 execve，事实上它们是不必要的——触发 bug 的先决条件是你要在两个能够互相发送大量消息的协同操作的进程中执行代码，对于这个 bug 来说，还需要能够和 IOSurface 通信。Damien DeVille 写了一篇博客（链接：http://ddeville.me/2015/02/interprocess-communication-on-ios-with-mach-messages）讨论使用应用程序组来从 iOS app 沙箱内部达到此目的的方法。执行带有 suid 标志的程序也是不必要的：我们可以通过 launchd 或者故意崩溃导致 launchd 运行 CrashRepoter 来找一个 mach 服务，来造成已释放的任务结构体被另一个更高权限的任务重用。

这个 bug 的许多特例已经在 OS X 10.11.6/iOS 9.3.3 中被修复，苹果已经做了缓解措施，避免传递其他任务的任务端口到特定的 IOKit 方法。

后退一些

这个 UAF bug 很有意思，但是它掩盖了更深层次更值得关注的问题。如果 IOSurfaceRootUserClient 现在对 owningTask 调用 task_reference()，且 owningTask 不得不成为一个 userclient 的初始创建者呢？这是否又是一个 bug？

今年早些时候 osxreverser@和我均独立地发布了关于 execve 系统调用的问题报告。在那个案例中，由于在加在一个 suid 程序时 execve 执行特定操作顺序不同导致了竞态条件，这使得新内存映射和旧的无效任务端口之间存在小的竞态窗口。

这是一个更为基础的问题：execve 系统调用实际上不会创建一个新的任务结构体，即使它执行一个更高级的 suid 程序。它仅仅修改已存在的任务结构体来做替代，所有之前就有一个 task_t 指针的对象现在仍然有一个指向更高级任务。

这不是暂时的内存安全——没有 UAF 被牵扯进来。让我们仔细看一看为什么这对于 XNU 是一个大问题。

译者：ruanbonan

原文链接：https://googleprojectzero.blogspot.kr/2016/10/taskt-considered-harmful.html

原文作者：Ian Beer,Project Zero


微信公众号：看雪 iOS 安全小组 我们的微博：weibo.com/pediyiosteam

我们的知乎：zhihu.com/people/pediyiosteam

加入我们：看雪 iOS 安全小组成员募集中：http://bbs.pediy.com/showthread.php?t=212949

[看雪 iOS 安全小组]置顶向导集合贴： http://bbs.pediy.com/showthread.php?t=212685