task_t 指针重大风险预报

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

分析篇

task_t 指针存在漏洞

由 Ian Beer,Project Zero 发布

本文讨论了一个存在于驱动 iOS 和 MacOS 的 XNU 内核核心部分的设计问题。苹果已经发布了两轮缓解策略，紧随其后地，昨天又发布了 MacOS 10.12.1/iOS 10.1 中的重大重构(Apple have shipped two iterations of mitigations followed yesterday by a large refactor in MacOS 10.12.1/iOS 10.1)。我们将关注以下内容：这些 bug 怎样被利用来进行沙箱逃逸并提升权限；我们如何绕过每一个缓解策略。每一步都配有一个可用的 exploit。

一些关于 mach 端口的背景知识

mach 端口是由内核维护的多发送者-单接收者的消息队列。某些特别的 mach 端口。

提供与用户空间相同的消息传送 API，但是发送给它们的消息会被内核消息处理程序同步处理。从这个意义上来说，发给这些端口的消息与系统调用非常像。

任务端口就是这样的一个例子。它们处理那些允许发送者操作一个任务的虚拟内存，并能够访问它的线程的消息。每一个任务有它自己的任务端口。内核占用的消息端口使用 MIG 这个工具来生成序列化代码。

从底层看 IOKit

当在用户空间创建一个新的 IOKit 用户客户端时，你通常会调用 IOKitLib 库的以下方法：

```
kern_return_t
IOServiceOpen(
io_service_t service,
task_port_t owningTask,
uint32_t type,
io_connect_t *connect );
```

IOServiceOpen 调用 MIG 为 io_service_open_extended 进程间通信方法生成序列化代码，并把序列化消息发送给已定的 IOService 端口。mach 陷阱 mach_msg 注意到这个端口被内核占用，并为该消息调用正确的内核 MIG 处理程序，而不是把它排在端口消息队列的队尾。

这里传来的任务端口是 owningTask；这个名字在用户空间和内核代码中相同。它是引起我注意的第一处地方。OwningTask 暗示着一个所属关系，这可能导致内核扩展开发者相信 IOKit 实际在这背后维护了一个所属关系，而这个关系确保 userclient 的生命周期总由 owningTask 的生命周期决定。这是一个危险的假设，本篇博客文章是质疑这个假设的结果。让我们来跟随代码流进入内核。下面是一段来自内核里面的 MIG 为 io_service_open_extended 生成的反序列化代码片段：

```
mig_internal novalue _Xio_service_open_extended(
mach_msg_header_t *InHeadP,
mach_msg_header_t *OutHeadP)
{
...
owningTask = convert_port_to_task(InOP->owningTask.n1lame);

RetCode = is_io_service_open_extended(
        service,
                        owningTask,
                        InOP->connect_type,
                        InOP->ndr,
                        (io_buf_ptr_t)(InOP->properties.address),
                        InOP->propertiesCnt, &OutP->result, &connection);

  task_deallocate(owningTask);
...
}
```

内核已经把所有包含在消息中的权限复制进来，所以 InOP->owningTask.name 实际上是指向一个 struct ipc_port 的指针，而不是用户态看到的 mach 端口名。

下面是 convert_port_to_task:

```
task_t
convert_port_to_task(
  ipc_port_t port)
{
  task_t task = TASK_NULL;

  if (IP_VALID(port)) {
      ip_lock(port);
      if (ip_active(port) &&
              ip_kotype(port) == IKOT_TASK)
      {
          task = (task_t)port->ip_kobject;
          assert(task != TASK_NULL);
          task_reference_internal(task);
      }
      ip_unlock(port);
  }

  return (task);
}
```

它检查了 port 参数，确保是一个任务端口对象，然后通过调用 task_reference 来对任务提供引用，返回 task_t 指针。task_t 是对 struct task 指针的别名，从代码中可以看出，它是一个引用计数对象。

这里 is_is_service_open_extended 仅仅是将 owningTask 传给::newUserClient：

```
  res = service->newUserClient(
```

```
    owningTask,
    (void *) owningTask,
    connect_type,
    propertiesDict,
    &client );
```

newUserClient 是一个 IOService 方法，如果它们想提供多 userclient 类型，它可以被一个 IOService 覆盖。否则默认执行在 IOKit 记录中查询 IOService 的 IOUserClient 子类类名，通过 IOKit 的反射 API（链接：https://bugs.chromium.org/p/project-zero/issues/detail?id=221）分配它，并调用它的::initWithTask 方法。::initWithTask 的默认执行流也不对 owningTask 做任何处理。

查看代码到此，似乎默认情况下 owningTask 并不会保有对 userclient 的引用（这会避免 userclient 对任务进行引用，形成循环引用，事实却完全相反；如果 userclient 想要保持对 owningTask 的引用，它必须对 owningTask 进行引用——不存在隐式的所属关系。

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