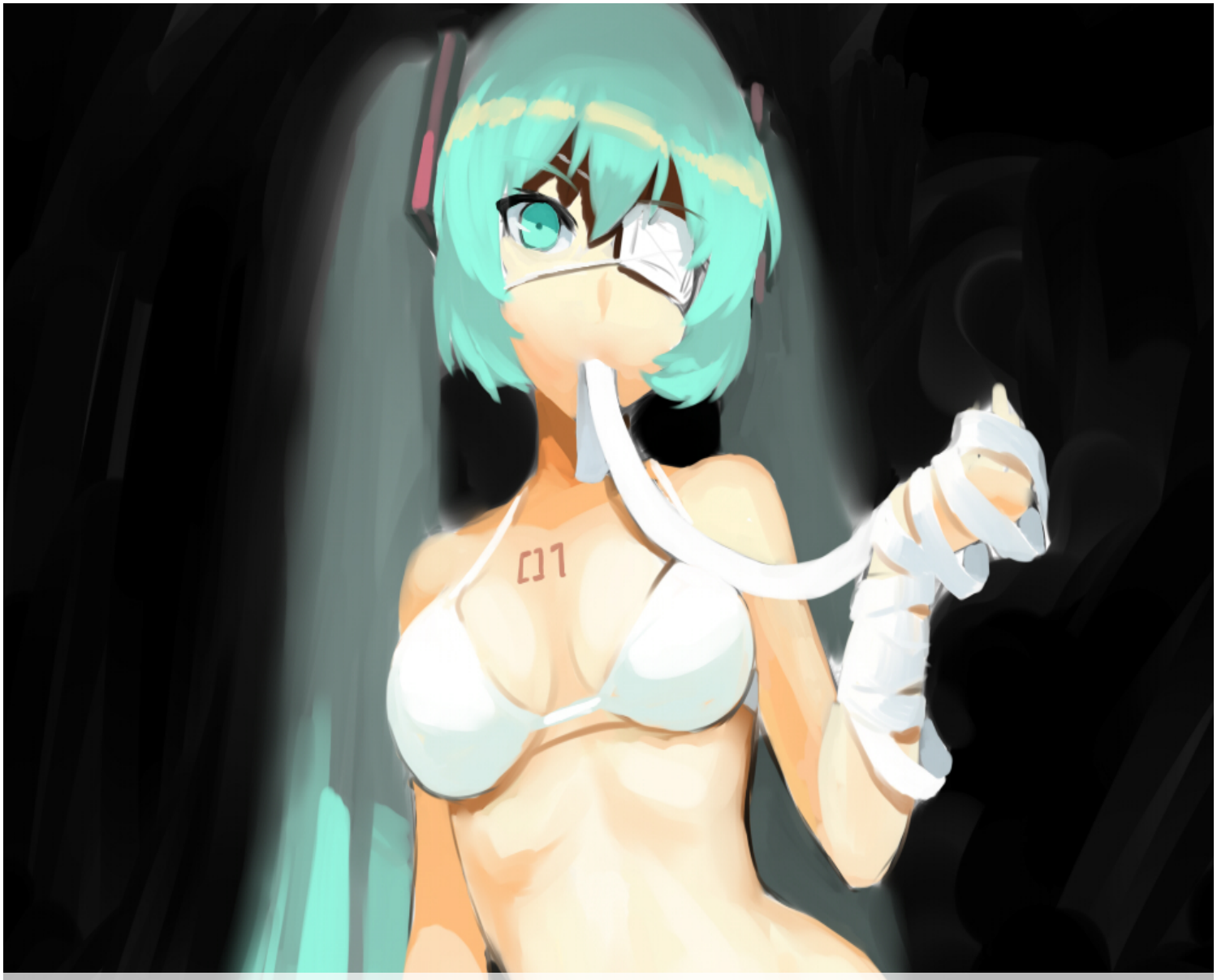

task_t 指针重大风险预报——PoC task_t considered harmful - many XNU EoPs

译者：银雁冰 校对：我有亲友团 原文作者：ianbeer@google.com

原文链接：https://bugs.chromium.org/p/project-zero/issues/detail?id=837

CVE-2016-1757 是由于 exec 运行期间资源条件竞争导致 port 失效而产生的安全漏洞

CVE-2016-1757 是一个涉及到在 exec 操作期间，端口结构顺序失效的条件竞争漏洞。

详情：

当一个 suid 二进制程序被执行，尽管 task struct 与执行程序前状态保持一致，但是它执行前的 task 和 task port 确实已失效。

当执行一个 suid 二进制程序时，虽然这个任务的旧任务以及线程端口已经无效，但是它的任务结构却还保持着相同的状态。

在此期间，执行前 task 没有自我复制和产生一个新的 task。

如果没有 fork 或者创建新的任务，

这就意味着任何指向之前 task struct 的指针如今指向一个进程 euid 为 0 的进程的 task struct。（即拥有 root 权限的执行环境）

许多 IOKit 驱动程序都保存着 task struct 指针作为它们的一部分，可以参考我之前的 bug 报告中的一些例子。

在这些例子中，我提到了另一个 bug，若 IOKit 驱动程序未引用 task struct，则如果杀死相应的 task，然后 fork 和执行一段 suid root 二进制程序，

我们能够通过一个 euid 为 0 的虚拟内存的 task struct 指针获得 IOKit object 交互。

我们就可以得到 IOKit 对象，并通过 task struct 指针，与一个 euid 为 0 的进程的虚拟内存进行交互。

（还有一种攻击方式：你也可以通过强制产生一个恢复 task struct 的服务程序来逃逸沙盒）

（你也可以通过强制 launchd 生成一个将会重新利用已被释放的 task struct 的服务，来实现沙盒逃逸。）

反之若这些 IOKit 驱动程序引用 task struct，没关系！

当然，再进一步，即使这些 IOKit 驱动程序对 task struct 作了引用，也无所谓！

（至少在没有 suid 二进制程序运行时）

（至少在 suid 二进制程序运行时没有问题。）

因为用户端的用户空间客户端在 time A 拥有发送至 task port 的权限，但当从 task port 传递至 IOKit 并不意味着仍然有发送权限，仅仅是因为 IOKIt 驱动程序实际调用的是 task struct 指针。

就 IOSurface 而言，这个允许我们方便的发送任意代码至虚拟内存 euid 为 0 的读写区域。

以 IOSurface 为例，这使得我们可以轻松的 map euid 为 0 的进程的虚拟内存里的任意可读写区域，并且重新写入。

大量 IOKit 驱动程序存储 taks struct 指针，使用它们操作用户空间虚拟内存（如 ioacceleratorFamily2,IOthunderboltFamily,IOSurface）或者依赖于 taks struct 指针去执行权限检测（如 IOHIDFamily）

另外一个有趣的例子是 stack 中的 task struct 指针

MIG 文件中相对应的用户层 / 内核层中的 task port 如下格式

```
type task_t = mach_port_t
#if KERNEL_SERVER
        intran: task_t convert_port_to_task(mach_port_t)
```

convert_port_to_task 如下：

```
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);
}
```

task port 转变为相对应的 task struct 指针，该指针引用于 task struct，但仅仅是确保它不被释放，

而非为了执行二进制程序导致它自己的 euid 不变。

而不是保证它的 euid 不会变成 suid root 程序执行的结果。

尽管 task port 不再有效，但只要 port lock 解除锁定，task 就可以执行标记为 suid 的二进制程序，task strut 指针就依然有效。

这就产生了大量的有趣的条件竞争。

grep 所有.defs 文件的源代码，需要一个 task_t 来找到它们;-)

在这个 exp 中，我将证明最有趣的环节：task_threads

让我们一起来看一下 task_threads 实际是如何工作的，包括由 MIG 产生的核心代码。

在 task_server.c(一个自动产生的文件，若找不到该文件，先 build XNU)

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

   RetCode = task_threads(target_task, (thread_act_array_t *)&(OutP->act_list.address), &OutP->act_listCnt);
   task_deallocate(target_task);
```

This gives us back the task struct from the task port then calls task_threads: (unimportant bits removed)

```
   task_threads(
      task_t                task,
      thread_act_array_t    *threads_out,
      mach_msg_type_number_t    *count)
```

```
  {
      ...
      for (thread = (thread_t)queue_first(&task->threads); i < actual;
                ++i, thread = (thread_t)queue_next(&thread->task_threads))
{
          thread_reference_internal(thread);
          thread_list[j++] = thread;
      }

      ...

          for (i = 0; i < actual; ++i)
              ((ipc_port_t *) thread_list)[i] =
convert_thread_to_port(thread_list[i]);
          }
      ...
  }
```

task_threads 利用 task struct 指针通过 threads 列表迭代 threads（来遍历线程列表），

然后 creates 发送指令给 task_threads，task_threads 发送指令返回给

用户空间，

然后赋予它们发送权限，随后在用户空间得到发送返回

过程中出现锁定和解锁，但是锁定和解锁是不相关的。

如果 task 同时运行 suid 标记为 root 的二进程代码会发生什么？

执行代码相关联的两部分主要是 ipc_task_reset 和 ipc_thread_reset

```
void
ipc_task_reset(
    task_t        task)
{
    ipc_port_t old_kport, new_kport;
    ipc_port_t old_sself;
    ipc_port_t old_exc_actions[EXC_TYPES_COUNT];
    int i;

    new_kport = ipc_port_alloc_kernel();
    if (new_kport == IP_NULL)
        panic("ipc_task_reset");

    itk_lock(task);

    old_kport = task->itk_self;
```

```
    if (old_kport == IP_NULL) {
        itk_unlock(task);
        ipc_port_dealloc_kernel(new_kport);
        return;
    }

    task->itk_self = new_kport;
    old_sself = task->itk_sself;
    task->itk_sself = ipc_port_make_send(new_kport);
    ipc_kobject_set(old_kport, IKO_NULL, IKOT_NONE); <-- point (1)
```

... then calls:

```
ipc_thread_reset(
    thread_t    thread)
{
    ipc_port_t old_kport, new_kport;
    ipc_port_t old_sself;
    ipc_port_t old_exc_actions[EXC_TYPES_COUNT];
    boolean_t    has_old_exc_actions = FALSE;
    int            i;
```

```
new_kport = ipc_port_alloc_kernel();
if (new_kport == IP_NULL)
    panic("ipc_task_reset");

thread_mtx_lock(thread);

old_kport = thread->ith_self;

if (old_kport == IP_NULL) {
    thread_mtx_unlock(thread);
    ipc_port_dealloc_kernel(new_kport);
    return;
}

thread->ith_self = new_kport; <-- point (2)
```

Point (1)从旧的 task port 清除 task struct pointer，然后重新分配一个新的 port 给 task

Point (2)对应的 thread port 同上.

调用执行 exec 的进程 B 和处理 task_threads（）的进程 A 以及 imagine 下面是执行过程：

Process A: target_task = convert_port_to_task(In0P->Head.msgh_request_port); // 得到指向 B 进程的 task struct 指针

Process B: ipc_kobject_set(old_kport, IKO_NULL, IKOT_NONE); // B 进程使旧的 task port 失效以至于不（再）拥有 task struct 的指针

Process B: thread->ith_self = new_kport // B 进程重新分配一个新的 thread ports 和激活（并设置）他们

Process A: ((ipc_port_t *) thread_list)[i] = convert_thread_to_port(thread_list[i]); // A 进程读取和转变为新的 thread port 对象!

这里最基本的问题不是这个特殊的资源（条件）竞争，事实上是当最先指定一个 task struct 指针后，你不能依赖拥有一个相同的 euid 的 task struct 指针。

exploit:

这段利用代码说明一个 euid 为 0 进程的 thread port 竞争资源，

这段 poc 仅仅利用了这种条件竞争来得到一个 euid 为 0 的程序的线程端口。

一旦运行利用代码，我仅仅需要跟随（放置了）一小段 ROP payload 插入 ret－slide。然后使用 thread port 设置 RIP 到 gadget 添加了大量的 rsp、X，随后会弹出 shell，只需要运行一段时间，将会出现竞争情况。

测试系统　MacBookAir5,2 OS X 10.11.5(15F34)

在 mac os10.12 更优化的利用代码，对于内核版本不高于 10.12 的都有效。