task_t 指针重大风险预报-修复建议篇

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

修复建议篇

XNU 不是 Unix 也不是 Mach

在纯净的 Mach 微内核中，使旧任务端口无效已经能够很充分地避免任何其他进程通过执行权限提升来保持对任务的控制，但是 XNU 不是微内核。之前我们看了把 mach 任务端口转换成任务结构体指针的内核函数 convert_port_to_task。这个指针可以被使用，并在内核中传递，而不没有发送消息的开销。例如，当 IOKit 想要控制一个进程的虚拟内存时，它直接调用相关的内核函数即可，而不必发送 mach 信息给 mach_vm MIG 子系统（理论上来说是可以这样做的）。

要弄明白这个机制还可以这样想：所有工作在内核中的 MIG 子系统（IOKit，mach_vm，tasks，threads，semaphores 等等）相互之间都直接连接。他们可以简单地调用目标函数，而不必通过 MIG IPC 层。这明显快了很多，但是带来了开销。

所有 task_t 指针都是潜在的安全 bug

权衡来看，没有一个有权使用资源的中心点能够被去除。在内核中，当特权执行发生时，它们不能仅仅让一个任务端口无效，就期望这会起作用，因为内核内部的 MIG 子系统不使用任务端口，仅仅在处于用户/内核边界时，它们会在任务端口和任务结构体指针之间做转化。内核不知道在哪里所有的内核指针会指向一个任务结构体；它无法期望去使它们无效化。

这是一个比最初的引用计数 bug 严重的多的问题。当一个权限提升操作发生时，execve 不能创建一个新进程；任务结构体保持原状，仅仅是特权改变了。这意味着内核中每一个单独的 task_t 指针都是一个潜在的安全 bug——在你获得它的权限后，没有锁机制

让你断言一个任务结构体的权限未曾改变，内核代码在某时获得一个任务结构体的权限也不意味着之后它应该具有这个权限。

在堆上：重写 IOSurface Exploit

实际上，为了让最初的 IOSurface Exploit 在正确的 task_reference(owningTask 调用时也能有效，我们仅仅需要稍微改变一下。我们不再让子进程把它的任务端口传回给父进程，而是在子进程中创建一个 IOSurfaceRootUserClient（正确地使用子进程自己的任务端口），然后把这个 userclient 端口传回给父进程。

然后子进程通过 execve 来执行一个带有 suid 权限的程序，这将会把任务的 EUID 设置为 0，且不释放任务结构体。父进程依然有给 IOSurfaceRootUserClient 发送消息的权限，并且这个 userclient 的 owningTask 现在的 EUID 是 0.父进程可以像之前那样继续执行，阻塞子进程，映射目标的 libc __DATA 段，覆盖一个函数指针并且取消对子进程的阻塞，这样子进程会尝试退出，并执行 ROP 栈代码。这个新的 Exploit 也绕过了 10.11.6 中加入的缓解策略，该缓解策略禁止创建带有其他任务的任务端口的 userclient。

注意，这个 Exploit 没有失败的案例——没有竞态条件需要获得，没有可能出错的 UAF。这个 Exploit 应该在所有版本低于 10.11.6 的 OS X 上生效。

这个原始的 exploit 比 UAF 的威力稍微小了一些，UAF 可以帮助你从非常严格的沙箱中逃逸，而在这个 exploit 中你的确需要调用 execve。这些在堆上储存 task_t 指针的 IOKit 对象仅仅是冰山一角。

在栈上：利用 task_threads

回到用户/内核边界处，当 convert_port_to_task 把一个从用户空间收到的任务端口转换成任务结构体指针时，这个任务可能执行一个 suid 或者有权限的二进制程序来提升它的权限。即使这个任务结构体指针没有存储在堆上，仍然可能存在可以利用的 bug。案例之一是下面的内核 MIG task_threads 方法：

```
kern_return_t
task_threads(
 task_t target_task,
```

```
 thread_act_array_t *act_list,
 mach_msg_type_number_t *act_listCnt );
```

被赋予一个任务端口的发送权限的情况下，这个方法返回这个任务所有线程的线程端口的发送权限。下面是内核中 MIG 自动生成的代码片段：

```
 target_task = convert_port_to_task(
   In0P->Head.msgh_request_port); // (1)
 RetCode = task_threads(
             target_task,
             (thread_act_array_t *)&(OutP->act_list.address),
             &OutP->act_listCnt);
 task_deallocate(target_task);
```

我们可以看到任务端口被转换成了任务结构体指针，它接下来被存储在局部变量 target_task 中，这个局部变量的生存周期是这个函数调用的生存周期。

下面是来自 task_threads 的相关代码：

```
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]);
// (2)
     }
   ...
 }
```

这段代码在线程列表中不断循环迭代地收集 struct thread 指针，然后把那些结构体线程转化为线程端口，并返回。代码中有少量的锁，但是它们是不相关的。

如果任务同时正在执行一个带有 suid 标志的程序，会发生什么？

相关的 exec 代码部分有两点，在 ipc_task_reset 和 ipc_thread_reset 中：

```
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); // (3)
```

紧跟着的是对 ipc_thread_reset 的调用：

```
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; // (4)
```

我们把执行 exec 的进程命名为 B，调用 task_threads()的进程命名为 A，想象下面的交叉执行过程：

```
A:

 target_task = convert_port_to_task(
   In0P->Head.msgh_request_port); // (1)
```

A 从栈上获得了指向进程 B 的任务结构体的指针。

```
B:

 ipc_kobject_set(old_kport, IKO_NULL, IKOT_NONE); // (3)
```

B 执行了带有 suid 权限的程序，并且使旧任务端口无效，因此它不再拥有任务结构体指针。

```
B:

 thread->ith_self = new_kport; // (4)
```

```
B 分配了新的线程端口并启动

A:

((ipc_port_t *) thread_list)[i] = convert_thread_to_port(thread_list[i]); // (2)

A 为 B 的特权线程读入并转换新的线程端口对象，这给了 A 一个特权线程端口。
```

一个线程端口的发送权限会给你完整的寄存器控制权限。这个 exploit 和之前的两个执行模式有些类似，不同的是一旦它获得线程端口，它可以直接把 RIP 指向我们的 gadget 地址，而不必覆盖一个函数指针。竞态窗口非常小，所以需要一个很特别的交叉执行才可以，但是这是可以实现的。查看 exploit（链接：https://bugs.chromium.org/p/project-zero/issues/attachment?aid=237182）和最初的 bug 报告（链接：https://bugs.chromium.org/p/project-zero/issues/detail?id=837）。

第二轮缓解策略

iOS 10/MacOS 10.12 引入了另外的缓解策略，同样可以绕过。

首先，在 IOKit 方面，userclient 的生命周期现在直接与创建的任务绑定。其次，ipc_kobject 服务有一处缓解措施来检测如果 MIG 内核方法因为竞态导致了 execve 调用，就强制使这个方法执行失败：

```
/*
* Check if the port is a task port, if its a task port then
* snapshot the task exec token before the mig routine call.
*/
ipc_port_t port = request->ikm_header->msgh_remote_port;
if (IP_VALID(port) && ip_kotype(port) == IKOT_TASK) {
 task = convert_port_to_task_with_exec_token(port, &exec_token);
}

(*ptr->routine)(request->ikm_header, reply->ikm_header);

/* Check if the exec token changed during the mig routine */
```

```
if (task != TASK_NULL) {
 if (exec_token != task->exec_token) {
   exec_token_changed = TRUE;
 }
 task_deallocate(task);
}
```

缓解策略中有三处缺陷：

1. 它仅仅审查了第一个参数，但是有的内核 MIG 方法会在其他位置接受一个任务端口。

2. 它仅仅检查任务端口，然而 thread_ports 也受到了相似的影响。

3. 它仅仅缓解了那些我们需要获得 MIG 调用返回资源（比如端口）的 bug。但是还有大量的其他方法是直接修改进程状态，而非返回新端口。

绕过第二轮缓解策略

虽然我们不再能够直接通过 task_threads 获得新的线程端口，还是有一些绕弯子的途径来达到目的。我们仅仅需要一个能够修改状态，而不是直接返回一些有用的东西（比如任务端口）的 API。

task_set_exception_port 允许我们为一个任务设置一个异常端口。当异常抛出时（比如非法访问内存）内核会发送一个异常消息给注册过的异常处理例程。对于我们很重要的是，这个异常消息包含了任务以及造成异常的线程的线程端口。

与内核中绝大多数地方带有一个 task_t 在栈上一样，这个 API 也存在有漏洞的竞态条件。在进程 A 中我们持续调用 task_set_exception_ports()来传递进程 B 的任务端口，同时 B execve 执行一个带有 suid 权限的程序：

```
mig_internal novalue _Xtask_set_exception_ports(
 mach_msg_header_t *InHeadP,
 mach_msg_header_t *OutHeadP) {
...
 task = convert_port_to_task(In0P->Head.msgh_request_port); // (1)

 OutP->RetCode =
   task_set_exception_ports(task,
```

```
                            In0P->exception_mask,
                            In0P->new_port.name,
                            In0P->behavior,
                            In0P->new_flavor);
 task_deallocate(task);
...



kern_return_t
task_set_exception_ports(
 task_t                 task,
 exception_mask_t       exception_mask,
 ipc_port_t             new_port,
 exception_behavior_t   new_behavior,
 thread_state_flavor_t new_flavor)
{
...
 itk_lock(task); // (2)

 for (i = FIRST_EXCEPTION; i < EXC_TYPES_COUNT; ++i) {
   if ((exception_mask & (1 << i)) ) {
     old_port[i] = task->exc_actions[i].port;
     task->exc_actions[i].port = ipc_port_copy_send(new_port); // (3)
     task->exc_actions[i].behavior = new_behavior;
     task->exc_actions[i].flavor = new_flavor;
     task->exc_actions[i].privileged = privileged;
   }
...
 itk_unlock(task);
...
```

进程 B 调用 execve 来执行一个特权 suid 程序：

```

ipc_task_reset(
 task_t  task)
{
...
 itk_lock(task); // (4)
...
```

```
 ip_lock(old_kport);
 ipc_kobject_set_atomically(old_kport, IKO_NULL, IKOT_NONE); // (5)
 task->exec_token += 1;
 ip_unlock(old_kport);

 ipc_kobject_set(new_kport, (ipc_kobject_t) task, IKOT_TASK);

 for (i = FIRST_EXCEPTION; i < EXC_TYPES_COUNT; i++) {
...
   if (!task->exc_actions[i].privileged) {
     old_exc_actions[i] = task->exc_actions[i].port;
     task->exc_actions[i].port = IP_NULL; // (6)
   }
 }

 itk_unlock(task); //(7)
```

我们寻找下面这样的交叉执行情景：

```
A:

task = convert_port_to_task(In0P->Head.msgh_request_port); // (1)

B:

itk_lock(task); // (4)

ipc_kobject_set_atomically(old_kport, IKO_NULL, IKOT_NONE); // (5)

task->exc_actions[i].port = IP_NULL; // (6)

itk_unlock(task); //(7)

A:

itk_lock(task); // (2)

task->exc_actions[i].port = ipc_port_copy_send(new_port); // (3)
```

我们很容易在竞态条件取得对 task_threads 的先机，因为锁保证了所有对我们有利的事情。我们要做的仅仅是循环调用 task_set_exception_ports 并且希望(4)处的 B 接过任务锁之前，(1)能够被 A 调用。实践中 Exp 在几微秒内就可以赢得竞态条件。

最后的工作实际上是确保当赢得竞态条件时，我们强制子进程引发一个异常，把它的任务和线程端口发给我们。我们可以通过在执行 suid 目标前带一个非常小的值调用 setrlimit(RLIMIT_STACK)

来实现。这意味着我们将要执行的二进制程序的栈空间很小，很快就会导致段错误。

在父进程中，一旦 task_set_exception_port 调用失败，我们就尝试从异常端口接收消息，设置一个短的 timeout。如果接收到消息，我们在竞态中取得先机，这个消息中包含 euid 为 0 的进程的任务和线程端口。这种情况下，Exp 在任务中分配了一些 RWX 内存，并把一个 shellcode 拷贝到这个地方。shellcode 做的事情如下：

```
struct rlimit lim = {0x1000000, 0x1000000};
setrlimit(RLIMIT_STACK, lim);
setuid(0);
char* argv[2] = {"/bin/bash", 0};
execve("/bin/bash", argv, 0);
```

shellcode 把栈长度设置回一个很大的值，用 setuid(0) 来避免 bash 丢失权限，最后打开一个 shell。

这个 Exp 应该会在 MacOS/OS X 版本<=10.12.0 时稳定工作。

最后的修复

这不是一类容易修复的 bug。XNU 的设计导致了 task_t 指针到处存在，而且我们提到的问题不只影响到 task_t 指针；线程也会受到这个问题的影响。苹果决定重构在装载二进制程序时用来分配新任务和线程结构体的 execve 的代码，应该会解决问题。这是一个工作量相当大的事情，苹果为修复这些 bug 投入的努力值得赞赏，我期待在 MacOS 10.12.1 版本源码中看到新的代码。

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

•