title: CVE-2016-4669分析与调试
date: 2016-11-07 14:57:00

categories: [CVE, OS X]
tags: [XNU,CVE,POC]

0x00 摘要

本文记录了对CVE-2016-4669的POC的调试中遇到的问题，以及相关知识的整理，该漏洞的原始报告在这里。原始报告中的内容，本文不在复述。

0x01 基础知识

1.1 MIG

`MIG`是`Mach`系统中使用的一种自动生成代码的脚本语言，以`.def`结尾。通过工具生成的代码分为`xxx.h`,`xxxClient.c`和`xxxServer.c`三个部分，在编译应用层程序时，和`xxxClient.c`文件一起编译，使用自动生成的代码。这里就是`poc`中的`taskUser.c`和`task.h`。

相关的细节可以查看《Mac OS X Internals: A Systems Approach》一书的**Section 9.6**中的描述。

1.2 内核中的内存管理

描述内核中堆内存的管理相关内容可以参考这个slides，比较简单明了的说清楚了内核中内存的基本结构iOS 10 Kernel Heap Revisited。

0x02 调试过程

2.1 core文件分析

在运行`POC`之后系统崩溃，查看崩溃的调用栈。

```
(lldb) bt
* thread #1: tid = 0x0000, 0xffffff80049c0f01 kernel`hw_lock_to + 17, stop reason = signal SIGSTOP
  * frame #0: 0xffffff80049c0f01 kernel`hw_lock_to + 17
    frame #1: 0xffffff80049c5cb3 kernel`usimple_lock(l=0xdeadbeefdeadbef7) + 35 at locks_i386.c:365 [opt]
    frame #2: 0xffffff80048c991c kernel`ipc_port_release_send [inlined] lck_spin_lock(lck=0xdeadbeefdeadbef7) + 44 at locks_i386.c:269 [opt]
    frame #3: 0xffffff80048c9914 kernel`ipc_port_release_send(port=0xdeadbeefdeadbeef) + 36 at ipc_port.c:1567 [opt]
    frame #4: 0xffffff80048e22d3 kernel`mach_ports_register(task=<unavailable>, memory=0xffffff800aad4270, portsCnt=3) + 547 at ipc_tt.c:1097 [opt]
    frame #5: 0xffffff8004935b3f kernel`_Xmach_ports_register(InHeadP=0xffffff800b2c297c, OutHeadP=0xffffff800e5c4b90) + 111 at task_server.c:647 [opt]
    frame #6: 0xffffff80048df2c3 kernel`ipc_kobject_server(request=0xffffff800b2c2900) + 259 at ipc_kobject.c:340 [opt]
    frame #7: 0xffffff80048c28f8 kernel`ipc_kmsg_send(kmsg=<unavailable>, option=<unavailable>, send_timeout=0) + 184 at ipc_kmsg.c:1443 [opt]
    frame #8: 0xffffff80048d26a5 kernel`mach_msg_overwrite_trap(args=<unavailable>) + 197 at mach_msg.c:474 [opt]
    frame #9: 0xffffff80049b8eca kernel`mach_call_munger64(state=0xffffff800f7e6540) + 410 at bsd_i386.c:560 [opt]
    frame #10: 0xffffff80049ecd86 kernel`hndl_mach_scall64 + 22
```

简单的分析一下调用栈

`_Xmach_ports_register`就是`taskSever.c`中对应的函数。

出问题的最关键的是`#4`，`#5`两个栈。

通过分析`mach_ports_register`函数的源码

```

    ...

    for (i = 0; i < TASK_PORT_REGISTER_MAX; i++) {
        ipc_port_t old;

        old = task->itk_registered[i];
        task->itk_registered[i] = ports[i];
        ports[i] = old;
    }

    itk_unlock(task);

    for (i = 0; i < TASK_PORT_REGISTER_MAX; i++)
        if (IP_VALID(ports[i]))
            ipc_port_release_send(ports[i]); <--#5b崩溃的地方

     if (portsCnt != 0)
        kfree(memory,<--释放memory
              (vm_size_t) (portsCnt * sizeof(mach_port_t)));
     ...
```

这里是对`ports`的数组中的参数调用`ipc_port_release_send`，出发的崩溃。

查看`ports`中的值，如下，

```
(lldb) f 4
kernel was compiled with optimization - stepping may behave oddly; variables may not be available.
frame #4: 0xffffff80048e22d3 kernel`mach_ports_register(task=<unavailable>, memory=0xffffff800aad4270, portsCnt=3) + 547 at ipc_tt.c:1097 [opt]
(lldb) p ports
(ipc_port_t [3]) $0 = {
  [0] = 0xffffff800b890680
  [1] = 0xdeadbeefdeadbeef
  [2] = 0x6c7070612e6d6f63
}
```

因为源码中会使用`kfree`去释放`memory`,接下来就去动态的调试吧。

2.2 动态调试

2.2.1 mach_ports_register

因为`mach_ports_register`这个函数在一些其他流程中都会有有调用，如果直接在内核中的`mach_ports_register`下断点，会有很多其他的调用会被断到，这里我的做法是先用`lldb`启动`r3gister`程序并断在`mach_ports_register`处，并运行。

```
➜ lldb r3gister
(lldb) target create "r3gister"
Current executable set to 'r3gister' (x86_64).
(lldb) b mach_ports_register
Breakpoint 1: 2 locations.
(lldb) r
Process 425 launched: '/Users/mrh/mach_port_register/r3gister' (x86_64)
Process 425 stopped
* thread #1: tid = 0x10fd, 0x00000001000012a7 r3gister`mach_ports_register(target_task=259, init_port_set=0x00007fff5fbffa98, init_port_setCnt=3) + 39 at taskUser.c:690, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000012a7 r3gister`mach_ports_register(target_task=259, init_port_set=0x00007fff5fbffa98, init_port_setCnt=3) + 39 at taskUser.c:690
   687             Reply Out;
   688         } Mess;
   689
-> 690         Request *InP = &Mess.In;
   691         Reply *Out0P = &Mess.Out;
   692
   693         mach_msg_return_t msg_result;
```

当已经断在这里的时候，在内核上下断点。

```
(lldb) b mach_ports_register
Breakpoint 1: where = kernel.development`mach_ports_register + 40 at ipc_tt.c:1060, address = 0xffffff8009686568
(lldb) c
Process 1 resuming
```

在内核的断点设置成功后，继续执行`r3gister`,就会触发内核中的断点。

```
(lldb) bt
* thread #1: tid = 0x0001, 0xffffff8009686568 kernel.development`mach_ports_register(task=0xffffff8012933640, memory=0xffffff800f7da660, portsCnt=3) + 40 at ipc_tt.c:1060, stop reason = breakpoint 1.1
  * frame #0: 0xffffff8009686568 kernel.development`mach_ports_register(task=0xffffff8012933640, memory=0xffffff800f7da660, portsCnt=3) + 40 at ipc_tt.c:1060 [opt]
    frame #1: 0xffffff80096e43ff kernel.development`_Xmach_ports_register(InHeadP=0xffffff8013c7937c, OutHeadP=0xffffff8014efaf90) + 111 at task_server.c:647 [opt]
    frame #2: 0xffffff8009683443 kernel.development`ipc_kobject_server(request=0xffffff8013c79300) + 259 at ipc_kobject.c:340 [opt]
    frame #3: 0xffffff800965ef03 kernel.development`ipc_kmsg_send(kmsg=<unavailable>, option=<unavailable>, send_timeout=0) + 211 at ipc_kmsg.c:1443 [opt]
    frame #4: 0xffffff8009675985 kernel.development`mach_msg_overwrite_trap(args=<unavailable>) + 197 at mach_msg.c:474 [opt]
    frame #5: 0xffffff800977f000 kernel.development`mach_call_munger64(state=0xffffff801278eb60) + 480 at bsd_i386.c:560 [opt]
    frame #6: 0xffffff80097b4de6 kernel.development`hndl_mach_scall64 + 22
```

2.2.2 memory的zone分析

通过`lldb`调试器查看`memory`处的内存，如下。

```
(lldb) memory read --format x --size 8 memory
0xffffff800f7da660: 0xffffff80140e7580 0xdeadbeefdeadbeef
0xffffff800f7da670: 0xffffff800f7dab00 0xfacadea23d1ec085
0xffffff800f7da680: 0x0000000000000000 0xffffffff00000000
0xffffff800f7da690: 0x0000000000000000 0x0000000000001000
```

通过一点小技巧可以查看`memory`是在哪一个`zone`上分配的，也可以继续跟踪代码，在后面的`kfree`中调用`zfree`函数的流程中会出现相关的转换代码。

```
...
if (zone->use_page_list) {
        struct zone_page_metadata *page_meta = get_zone_page_metadata((struct zone_free_element *)addr);
        if (zone != page_meta->zone) {
...
```

其实就是`get_zone_page_metadata`这个函数的实现了，这里的`addr`就是`memory`。

```
(lldb) p *(zone_page_metadata*)0xffffff80140e7000
(zone_page_metadata) $1 = {
  pages = {
    next = 0xffffff8012db4000
    prev = 0xffffff801109f000
  }
  elements = 0xffffff80140e76d0
  zone = 0xffffff800f480ba0
  alloc_count = 12
  free_count = 1
}
```

在查看`zone`的具体数据，可以得知`memory`被分配在哪个`zone`当中。

```
...
zone_name = 0xffffff8009d35bac "kalloc.16"
...
```

可以得知，因为`port`的个数被设置为1个，所以只需要一个指针，而在调用`kfree`的时候，`kfree`的`size`是3个指针的长度，所以是试图在`kalloc.24`中释放内存,这就会造成错误的`kfree`，但是苹果有一段神奇的代码，尝试修复这个问题。

```
if (zone->use_page_list) {
        struct zone_page_metadata *page_meta = get_zone_page_metadata((struct zone_free_element *)addr);
        if (zone != page_meta->zone) {
            /*
             * Something bad has happened. Someone tried to zfree a pointer but the metadata says it is from
             * a different zone (or maybe it's from a zone that doesn't use page free lists at all). We can repair
             * some cases of this, if:
             * 1) The specified zone had use_page_list, and the true zone also has use_page_list set. In that case
             *    we can swap the zone_t
             * 2) The specified zone had use_page_list, but the true zone does not. In this case page_meta is garbage,
             *    and dereferencing page_meta->zone might panic.
             * To distinguish the two, we enumerate the zone list to match it up.
             * We do not handle the case where an incorrect zone is passed that does not have use_page_list set,
             * even if the true zone did have this set.
             */
            zone_t fixed_zone = NULL;
            int fixed_i, max_zones;

            simple_lock(&all_zones_lock);
            max_zones = num_zones;
            fixed_zone = first_zone;
            simple_unlock(&all_zones_lock);

            for (fixed_i=0; fixed_i < max_zones; fixed_i++, fixed_zone = fixed_zone->next_zone) {
                if (fixed_zone == page_meta->zone && fixed_zone->use_page_list) {
                    /* we can fix this */
                    printf("Fixing incorrect zfree from zone %s to zone %s\n", zone->zone_name, fixed_zone->zone_name);
                    zone = fixed_zone;
                    break;
                }
            }
        }
    }
```

用代码修复数据结构的错误本身就是一件很危险的事情，而这里更危险的是如果不能修复这个错误的话，代码没有任何报错或提示，这里就会有很多隐患。

2.2.3 堆内存简单分析

这里简单的分析一下内核中堆的数据结构，写到这里的时候我重启了一次虚拟机和调试器，所以地址和之前会对不上。

这一次看到的`kalloc.16`的`zone`如下。

```
p *(zone*)0xffffff80213caea0
(zone) $5 = {
  free_elements = 0x0000000000000000
  pages = {
    any_free_foreign = {
      next = 0xffffff80213caea8
      prev = 0xffffff80213caea8
    }
    all_free = {
      next = 0xffffff80213caeb8
      prev = 0xffffff80213caeb8
    }
    intermediate = {
      next = 0xffffff8025e06000
      prev = 0xffffff8025706000
    }
    all_used = {
      next = 0xffffff8025226000
      prev = 0xffffff8025d29000
    }
  }
  count = 29951
  countfree = 37
  lock_attr = (lck_attr_val = 0)
  lock = {
    lck_mtx_sw = {
      lck_mtxd = {
        lck_mtxd_owner = 0
         = {
           = {
            lck_mtxd_waiters = 0
            lck_mtxd_pri = 0
            lck_mtxd_ilocked = 0
            lck_mtxd_mlocked = 0
            lck_mtxd_promoted = 0
            lck_mtxd_spin = 0
            lck_mtxd_is_ext = 0
            lck_mtxd_pad3 = 0
          }
          lck_mtxd_state = 0
        }
        lck_mtxd_pad32 = 4294967295
```

```
      }
      lck_mtxi = {
        lck_mtxi_ptr = 0x0000000000000000
        lck_mtxi_tag = 0
        lck_mtxi_pad32 = 4294967295
      }
    }
  }
  lock_ext = {
    lck_mtx = {
      lck_mtx_sw = {
        lck_mtxd = {
          lck_mtxd_owner = 0
           = {
             = {
              lck_mtxd_waiters = 0
              lck_mtxd_pri = 0
              lck_mtxd_ilocked = 0
              lck_mtxd_mlocked = 0
              lck_mtxd_promoted = 0
              lck_mtxd_spin = 0
              lck_mtxd_is_ext = 0
              lck_mtxd_pad3 = 0
            }
            lck_mtxd_state = 0
          }
          lck_mtxd_pad32 = 0
        }
        lck_mtxi = {
          lck_mtxi_ptr = 0x0000000000000000
          lck_mtxi_tag = 0
          lck_mtxi_pad32 = 0
        }
      }
    }
    lck_mtx_grp = 0x0000000000000000
    lck_mtx_attr = 0
    lck_mtx_pad1 = 0
    lck_mtx_deb = (type = 0, pad4 = 0, pc = 0, thread = 0)
    lck_mtx_stat = 0
    lck_mtx_pad2 = ([0] = 0, [1] = 0)
  }
  cur_size = 479808
  max_size = 531441
  elem_size = 16
  alloc_size = 4096
  page_count = 119
  sum_count = 235587
  exhaustible = 0
```

```
  collectable = 1
  expandable = 1
  allows_foreign = 0
  doing_alloc_without_vm_priv = 0
  doing_alloc_with_vm_priv = 0
  waiting = 0
  async_pending = 0
  zleak_on = 0
  caller_acct = 0
  doing_gc = 0
  noencrypt = 0
  no_callout = 0
  async_prio_refill = 0
  gzalloc_exempt = 0
  alignment_required = 0
  use_page_list = 1
  _reserved = 0
  index = 12
  next_zone = 0xffffff80213ca120
  zone_name = 0xffffff801ef3af0d "kalloc.16"
  zleak_capture = 0
  zp_count = 0
  prio_refill_watermark = 0
  zone_replenish_thread = 0x0000000000000000
  gz = {
    gzfc_index = 0
    gzfc = 0xdeadbeefdeadbeef
  }
}
```

这里就主要的分析一下`page`和`page`内的内存的分布。

```
p *(zone*)0xffffff80213caea0
(zone) $5 = {
  free_elements = 0x0000000000000000
  pages = {
    any_free_foreign = {
      next = 0xffffff80213caea8
      prev = 0xffffff80213caea8
    }
    all_free = {
      next = 0xffffff80213caeb8
      prev = 0xffffff80213caeb8
    }
    intermediate = {
      next = 0xffffff8025e06000
      prev = 0xffffff8025706000
    }
    all_used = {
      next = 0xffffff8025226000
      prev = 0xffffff8025d29000
    }
  }
  ...
```

简单的看一下4个`pages`的队列中的`intermediate`，在这个队列中的`page`里都会有一些未被使用的内存,通过`pages`里面的`next`和`prev`构成了一个双向链表，如下所示。

```
(lldb) p *(zone_page_metadata*)0xffffff8025e06000
(zone_page_metadata) $6 = {
  pages = {
    next = 0xffffff8025548000
    prev = 0xffffff80213caec8
  }
  elements = 0xffffff8025e06750
  zone = 0xffffff80213caea0
  alloc_count = 252
  free_count = 1
}

(lldb) p *(zone_page_metadata*)0xffffff8025548000
(zone_page_metadata) $12 = {
  pages = {
    next = 0xffffff8025c96000
    prev = 0xffffff8025e06000
  }
  elements = 0xffffff8025548d50
  zone = 0xffffff80213caea0
  alloc_count = 252
  free_count = 1
}

(lldb) p *(zone_page_metadata*)0xffffff8025c96000
(zone_page_metadata) $13 = {
  pages = {
    next = 0xffffff8025cb3000
    prev = 0xffffff8025548000
  }
  elements = 0xffffff8025c968e0
  zone = 0xffffff80213caea0
  alloc_count = 252
  free_count = 5
}
```

`elements`就是第一个可以`alloc`的内存，通过lldb观察内存布局

```
(lldb) memory read --format x --size 8 0xffffff8025c968e0
0xffffff8025c968e0: 0xffffff8025c96670 0xfacade04d7b687dd

(lldb) memory read --format x --size 8 0xffffff8025c96670
0xffffff8025c96670: 0xffffff8025c96680 0xfacade04d7b6872d

(lldb) memory read --format x --size 8 0xffffff8025c96680
0xffffff8025c96680: 0xffffff8025c96a40 0xfacade04d7b68bed

(lldb) memory read --format x --size 8 0xffffff8025c96680
0xffffff8025c96680: 0xffffff8025c96a40 0xfacade04d7b68bed
```

`freeelement`是通过单向链表随机的串联在`page`中，在前面[iOS 10 Kernel Heap Revisited]中提到的。

可以看到前面的8个字节控制链表的，后面8个字节是`freeelement`的存储空间，`0xfacade`就是堆中的cookies。

```
    zp_poisoned_cookie &= 0x000000FFFFFFFFFF;
    zp_poisoned_cookie |= 0x0535210000000000; /* 0xFACADE */
```

了解了`freeelement`的内存布局，再看一看已经被分配了的内存，也就是`memory`。

```
(lldb) memory read --format x --size 8 memory
0xffffff8025e06300: 0xffffff8029828430 0xdeadbeefdeadbeef
```

阅读源码中的`zalloc_internal`函数的实现，可以得知，在kalloc.16的堆中分配申请内存时，会将申请出来的内存会被写入`0xdeadbeefdeadbeef`。

```
vm_offset_t *primary  = (vm_offset_t *) addr; //addr == memory
vm_offset_t *backup   = get_backup_ptr(inner_size, primary);

*primary = ZP_POISON;
*backup  = ZP_POISON;
```

2.2.4 ipc_port_release_send

在导致崩溃的函数处下断点

```
(lldb) b ipc_tt.c :1097
Breakpoint 2: where = kernel.development`mach_ports_register + 521 at ipc_tt.c:1097, address = 0xffffff801e886749
(lldb) c
Process 1 resuming

Process 1 stopped
* thread #2: tid = 0x0002, 0xffffff801e886749 kernel.development`mach_ports_register(task=<unavailable>, memory=0xffffff8025e06300, portsCnt=3) + 521 at ipc_tt.c:1097, stop reason = breakpoint 2.1
    frame #0: 0xffffff801e886749 kernel.development`mach_ports_register(task=<unavailable>, memory=0xffffff8025e06300, portsCnt=3) + 521 at ipc_tt.c:1097 [opt]

(lldb) p ports
(ipc_port_t [3]) $15 = {
  [0] = 0xffffff80279d8190
  [1] = 0x0000000000000000
  [2] = 0x0000000000000000
}
```

发现`ports`的值只有ports[0]有值，这是因为这里的`ports`是从旧的`port`中替换来的

```
    /*
     *  Replace the old send rights with the new.
     *  Release the old rights after unlocking.
     */

    for (i = 0; i < TASK_PORT_REGISTER_MAX; i++) {
        ipc_port_t old;

        old = task->itk_registered[i];
        task->itk_registered[i] = ports[i];
        ports[i] = old;
    }
```

据悉执行后，`r3gister`会再次被断住，第二次调用`mach_ports_register`后，内核断点，再看`ports`，如下

```
* thread #2: tid = 0x0002, 0xffffff801e886749 kernel.development`mach_ports_register(task=<unavailable>, memory=0xffffff80253d3e70, portsCnt=3) + 521 at ipc_tt.c:1097, stop reason = breakpoint 2.1
    frame #0: 0xffffff801e886749 kernel.development`mach_ports_register(task=<unavailable>, memory=0xffffff80253d3e70, portsCnt=3) + 521 at ipc_tt.c:1097 [opt]
(lldb) p ports
(ipc_port_t [3]) $16 = {
  [0] = 0xffffff8029828430
  [1] = 0xdeadbeefdeadbeef
  [2] = 0xffffff80252f1460
}
```

从而导致在服务器的后续代码中触发了崩溃

```
    for (i = 0; i < TASK_PORT_REGISTER_MAX; i++)
        if (IP_VALID(ports[i]))
            ipc_port_release_send(ports[i]);//<--第二个ports就是0xdeadbeefdeadbeef

    /*
     *  Now that the operation is known to be successful,
     *  we can free the memory.
     */

    if (portsCnt != 0)
        kfree(memory,
              (vm_size_t) (portsCnt * sizeof(mach_port_t)));
```

0x03 小结

这里只分析了`POC`的触发，如果修改`mach_ports_register`的参数，将`port`的个数改为2个，port[1]就会变成`0x0000000000000000`，从而避免了对`0xdeadbeefdeadbeef`调用函数出发崩溃，而且zone会变成iports的一个专用的`zone`，而不是`kalloc.16`,所以这个漏洞值得研究的地方还有很多。希望本文能为大家继续研究这个漏洞提供一些帮助；-）。

参考

1、[OS X/iOS multiple memory safety issues in mach_ports_register]

2、[iOS 10 Kernel Heap Revisited]