black hat
EUROPE 2018

DECEMBER 3-6, 2018
EXCEL LONDON / UNITED KINGDOM

Eternal War in XNU Kernel Objects

Min(Spark) Zheng, Xiaolong Bai, Hunter
Alibaba Orion Security Lab

whoami

```
iPhone#
iPhone# uname -a
Darwin iPhone 17.5.0 Darwin Kernel Version 17.5.0: Tue Mar 13
21:32:11 PDT 2018; root:xnu-4570.52.2-8/RELEASE_ARM64_S8000 iP
hone8,1
iPhone# id
uid=0(root) gid=0(wheel) egid=501(mobile) groups=0(wheel),1(da
emon),2(kmem),3(sys),4(tty),5(operator),8(procview),9(procmod)
,20(sta      29(certusers),80(admin)
iPhon         rootfs remounted and JB by Spark and Bxl" > /Ove
rSk
iP
```

Min(Spark) Zheng
@SparkZheng

Tweets	Following	Followers
203	72	24.6K

- SparkZheng @ Twitter，蒸米spark @ Weibo
- Alibaba Security Expert
- CUHK PhD, Blue-lotus and Insight-labs
- Gave talks at RSA, BlackHat, DEFCON, HITB, ISC, etc

Xiaolong Bai
@bxl1989 Follows you

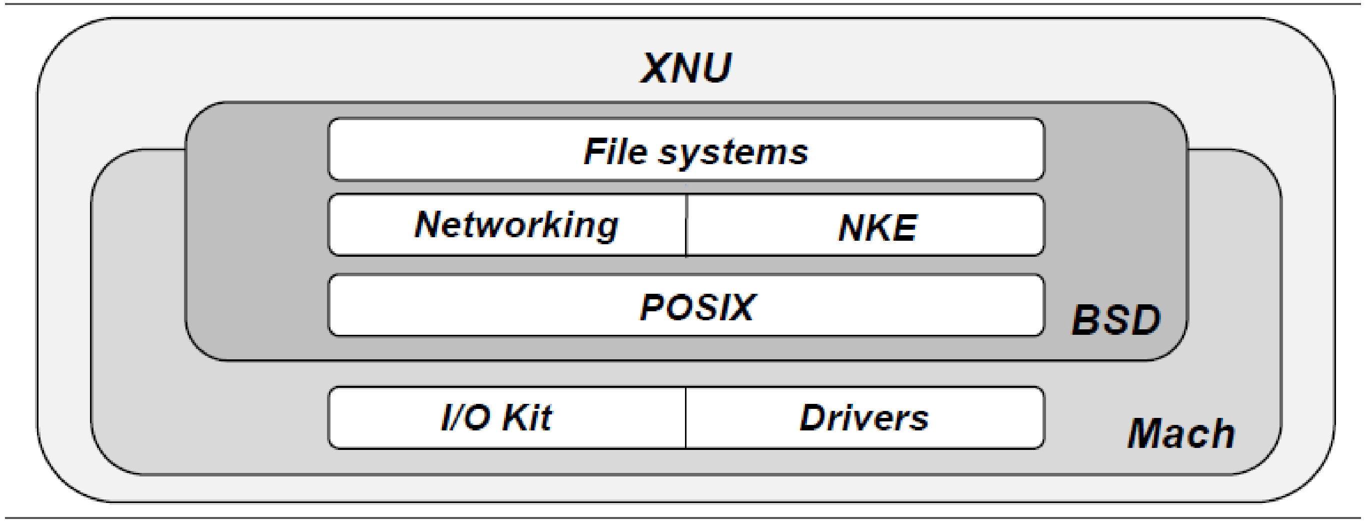
Tweets	Following	Followers
97	188	2,036

- Xiaolong Bai (bxl1989 @ Twitter&Weibo)
- Alibaba Security Engineer
- Ph.D. graduated from Tsinghua University
- Published papers on S&P, Usenix Security, CCS, NDSS

Apple Devices & Jailbreaking

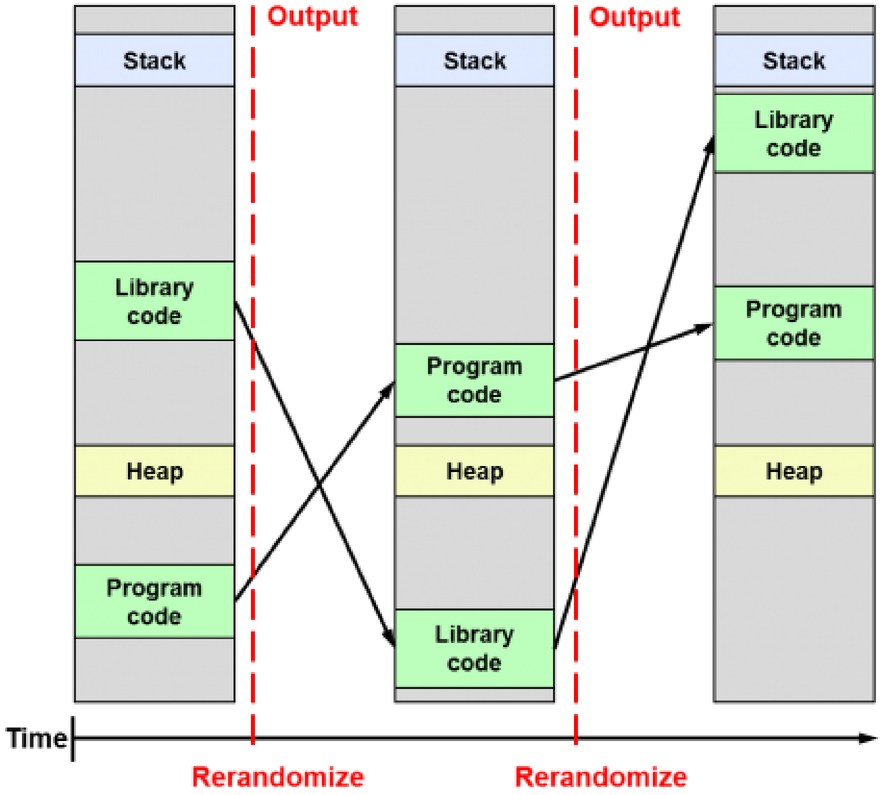
- **Jailbreaking** in general means breaking the device out of its "jail".
- Apple devices (e.g., iPhone, iPad) are most famous "**jail**" devices among the world.
- iOS, macOS, watchOS, and tvOS are **operating systems** developed by Apple Inc and used in Apple devices.

XNU



- All systems deploy a same hybrid kernel structure called **XNU**.
- There are cases that **kernel vulnerabilities** have been used to escalate the privileges of attackers and get full control of the system (hence jailbreak the device).
- Accordingly, Apple has deployed multiple **security mechanisms** that make the exploitation of the device harder.

Mitigation - DEP/KASLR

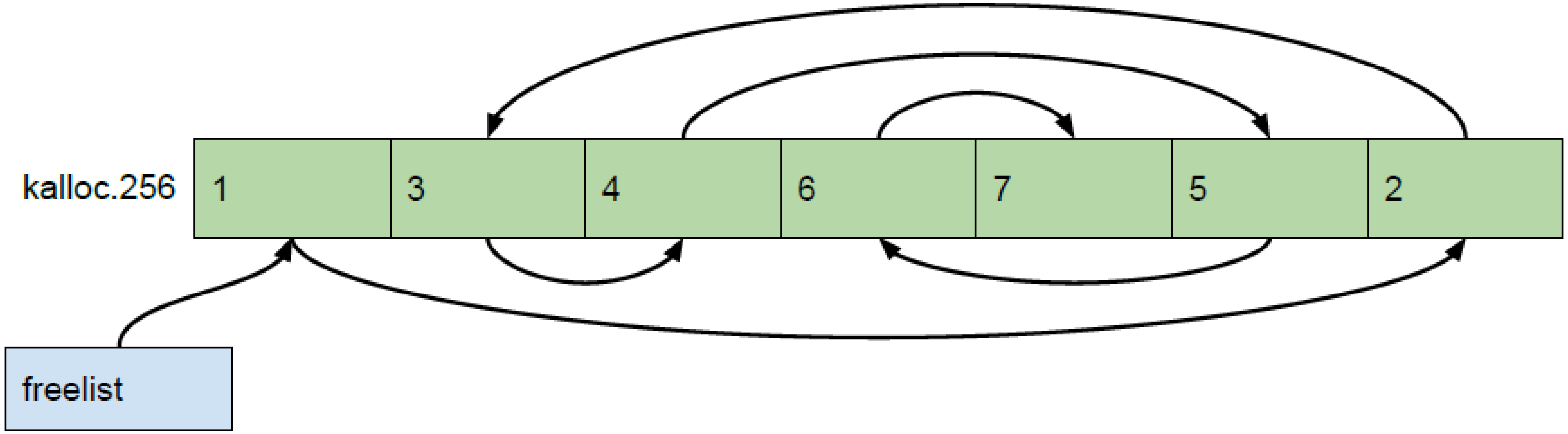


- Apple deployed Data Execution Prevention (**DEP**) and Kernel Address Space Layout Randomization (**KASLR**) from iOS 6 and macOS 10.8.

- **DEP** enables the system to mark relevant pages of memory as non-executable to prevent code injection attack. To break the DEP protection, code-reuse attacks (e.g., ROP) were proposed.

- To make these addresses hard to predict, **KASLR** memory protection randomizes the locations of various memory segments. To bypass KASLR, attackers usually need to leverage information leakage bugs.

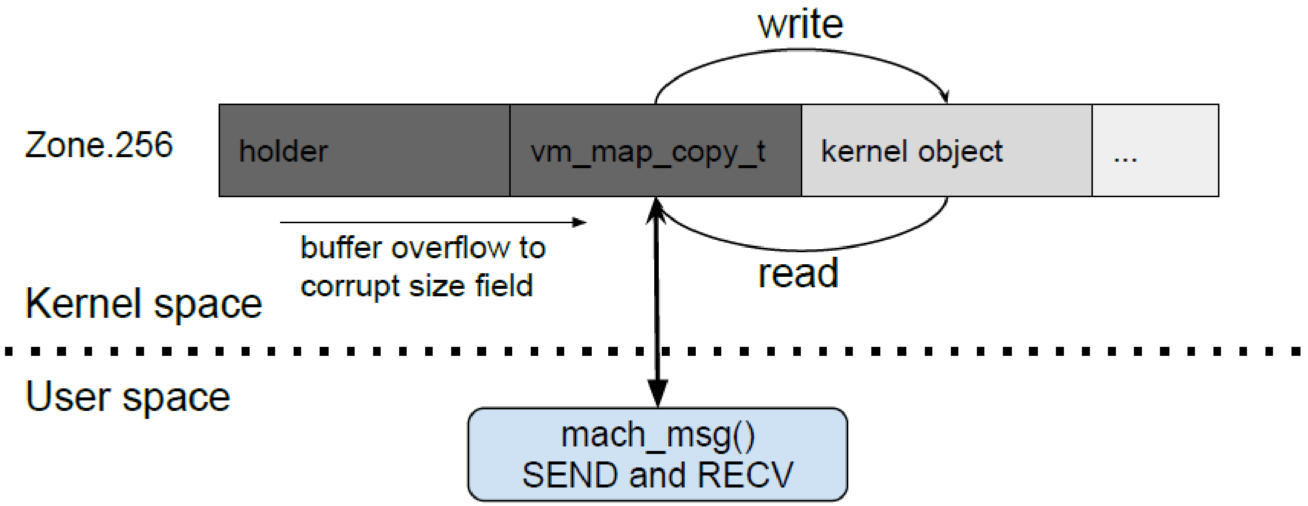
Mitigation - Freelist Randomization



- In previous XNU, the freelist that contains all the freed kernel objects inside a zone uses the **LIFO** (last-in-first-out) policy.

- To make the adjacent object hard to predict, Apple deployed a mitigation called freelist **randomization** in iOS 9.2. When a kernel object is freed, the XNU will randomly choose the first or last position of the freelist to store the freed element.

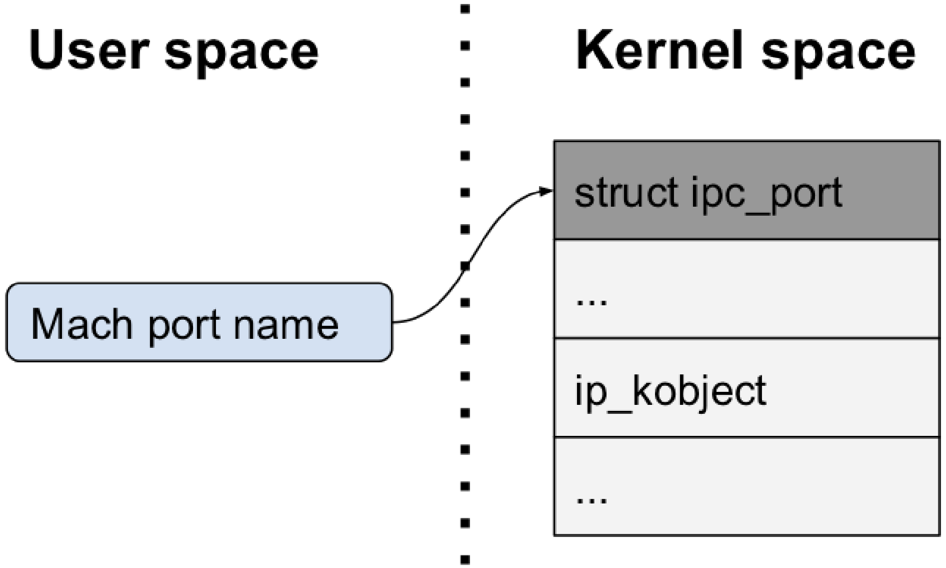
Mitigation - Wrong Zone Free Protection



- An attacker can use a memory corruption vulnerability to change the size value of a kernel object to a **wrong** size (e.g., 512) and receive (free) the object. After that, the attacker can allocate a new kernel object with the changed size (e.g., 512) into the original kalloc.256 zone.

- To mitigate this attack, Apple added a new **zone_metadata_region** structure for each zone in iOS 10.

New Target - Mach Port in User Space



- A Mach port in XNU is a kernel controlled **communication channel**. It provides basic operations to pass messages between threads.

- **Ports** are used to represent resources, services, and facilities (e.g., hosts, tasks, threads, memory objects, and clocks) thus providing **object-style** access to these abstractions.

- In user space, Mach ports are **integer numbers** like handlers for kernel objects.

New Target – Struct ipc_port in Kernel Space

struct ipc_port	
io_bits	io_references
io_lock_data	
…	
struct ipc_space *receiver;	
ipc_kobject_t ip_kobject;	
…	
mach_vm_address_t ip_context;	
…	

- In the kernel, a Mach port is represented by a pointer to an **ipc_port** structure.

- There are **40** types of ipc_port objects in XNU and **io_bits** field defines the type of it. io_references field counts the reference number of the object. Locking related data is stored in the io_lock_data field.

- Receiver field is a pointer that points to receiver’s **IPC** space (e.g. ipc_space_kernel). **ip_kobject** field points to a kernel data structure according to the kernel object type.

(Mach) Port-oriented Programming (POP)

- The main goal is to obtain multiple **primitives** to read/write kernel memory and execute arbitrary kernel code, even in the case that multiple **mitigations** are deployed in the system.

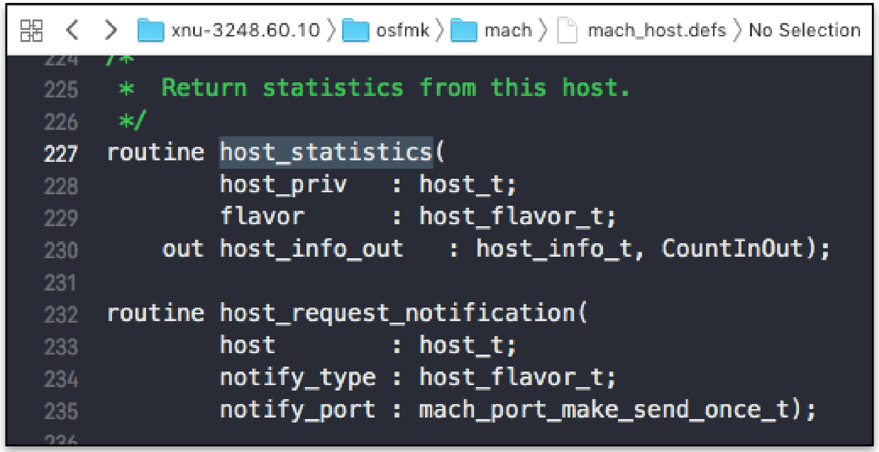
(MACH) PORT-ORIENTED PROGRAMMING

- Attackers leverage a special kernel object, i.e., ipc_port, to obtain multiple primitives, including kernel read/write and arbitrary code execution, by issuing system calls in user mode. Since the proposed method is mainly based on the ipc_port kernel object, we call it **(Mach) Port-oriented Programming (POP)**.

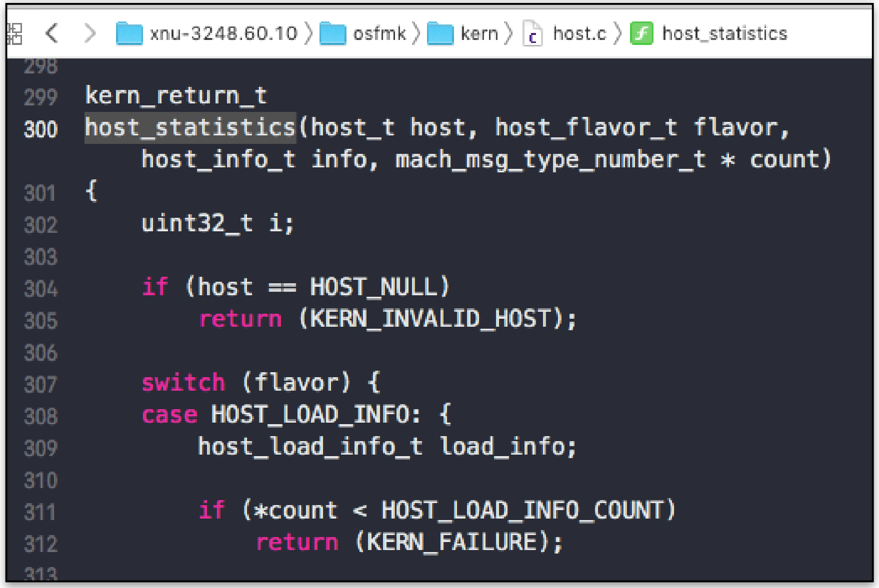
- Note that POP technology was **not created** by **us**. We saw it in many **public** exploits and then **summarize** this code reuse attack technique for systematic study.

MIG in Source Code

```c
const struct mig_subsystem *mig_e[] = {
        (const struct mig_subsystem *)&mach_vm_subsystem,
        (const struct mig_subsystem *)&mach_port_subsystem,
        (const struct mig_subsystem *)&mach_host_subsystem,
        (const struct mig_subsystem *)&host_priv_subsystem,
        (const struct mig_subsystem *)&host_security_subsystem,
        (const struct mig_subsystem *)&clock_subsystem,
        (const struct mig_subsystem *)&clock_priv_subsystem,
        (const struct mig_subsystem *)&processor_subsystem,
        (const struct mig_subsystem *)&processor_set_subsystem,
        (const struct mig_subsystem *)&is_iokit_subsystem,
    (const struct mig_subsystem *)&lock_set_subsystem,
    (const struct mig_subsystem *)&task_subsystem,
    (const struct mig_subsystem *)&thread_act_subsystem,
#ifdef VM32_SUPPORT
    (const struct mig_subsystem *)&vm32_map_subsystem,
#endif
    (const struct mig_subsystem *)&UNDReply_subsystem,
    (const struct mig_subsystem *)&mach_voucher_subsystem,
    (const struct mig_subsystem *)&mach_voucher_attr_control_subsystem,

#if     XK_PROXY
        (const struct mig_subsystem *)&do_uproxy_xk_uproxy_subsystem,
#endif /* XK_PROXY */
#if     MACH_MACHINE_ROUTINES
        (const struct mig_subsystem *)&MACHINE_SUBSYSTEM,
#endif  /* MACH_MACHINE_ROUTINES */
#if     MCMSG && iPSC860
    (const struct mig_subsystem *)&mcmsg_info_subsystem,
#endif  /* MCMSG && iPSC860 */
};
```



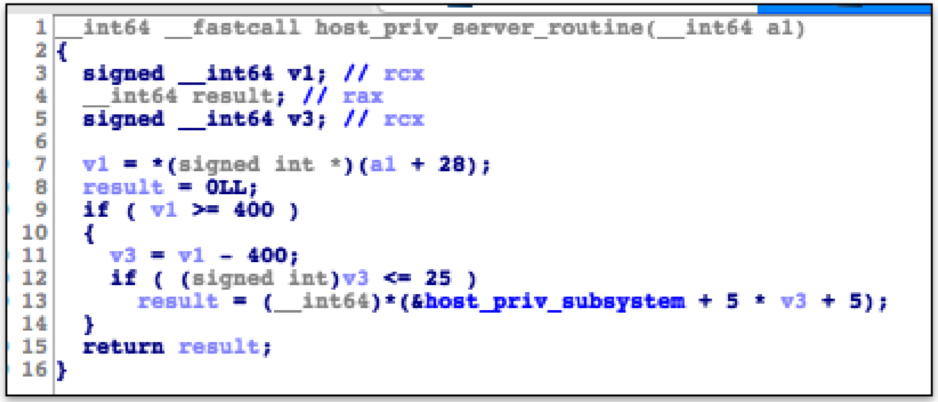
```
/*
 *  Return statistics from this host.
 */
routine host_statistics(
        host_priv   : host_t;
        flavor      : host_flavor_t;
    out host_info_out   : host_info_t, CountInOut);

routine host_request_notification(
        host        : host_t;
        notify_type : host_flavor_t;
        notify_port : mach_port_make_send_once_t);
```



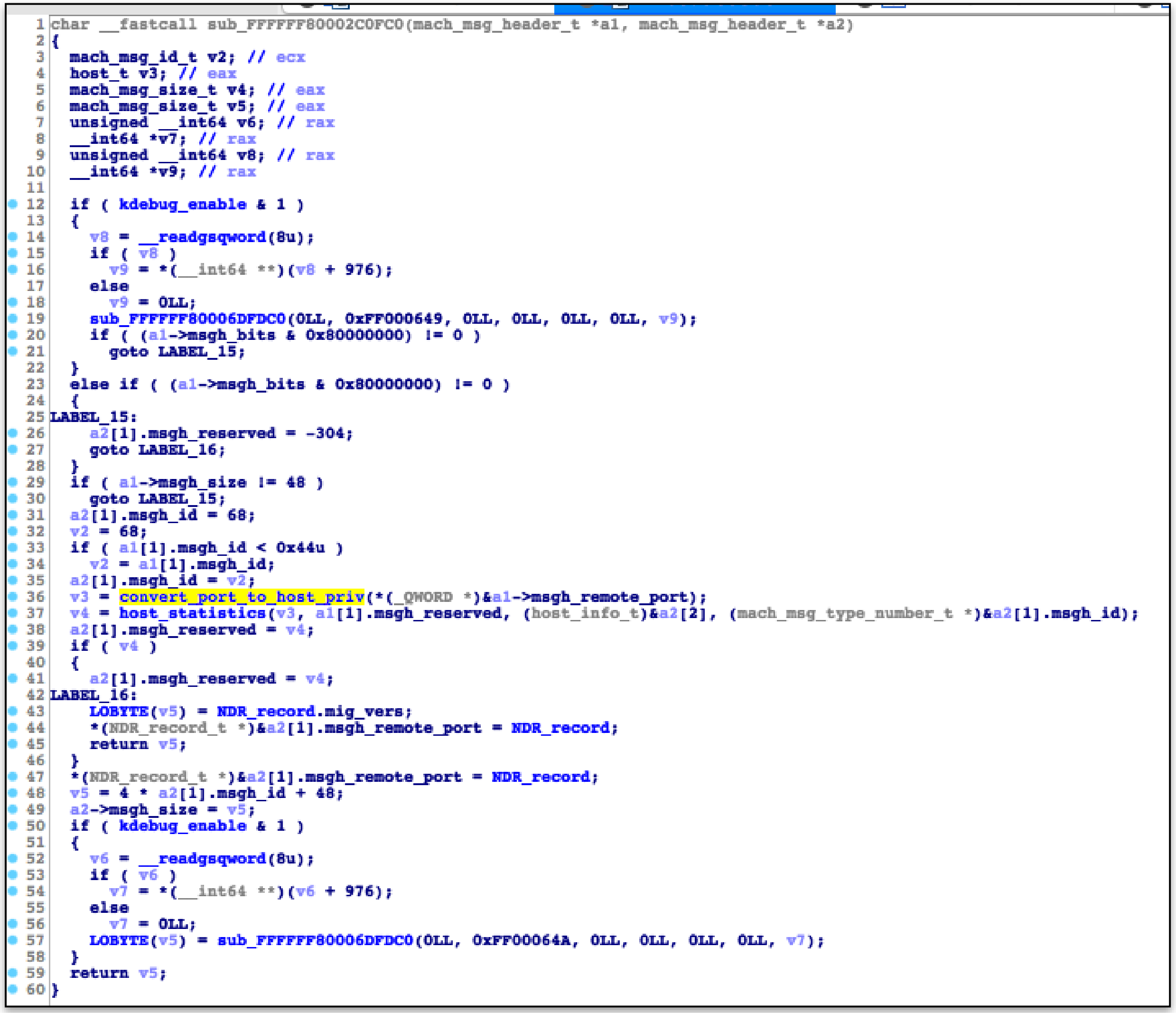
```c
kern_return_t
host_statistics(host_t host, host_flavor_t flavor,
    host_info_t info, mach_msg_type_number_t * count)
{
    uint32_t i;

    if (host == HOST_NULL)
        return (KERN_INVALID_HOST);

    switch (flavor) {
    case HOST_LOAD_INFO: {
        host_load_info_t load_info;

        if (*count < HOST_LOAD_INFO_COUNT)
            return (KERN_FAILURE);
```

MIG in Kernel Cache

```
__constdata:FFFFFF8000C5FD00                 public _host_priv_subsystem
__constdata:FFFFFF8000C5FD00 _host_priv_subsystem dq offset _host_priv_server_routine
__constdata:FFFFFF8000C5FD00                                         ; DATA XREF:  host priv
__constdata:FFFFFF8000C5FD00                                         ;  host priv server+44↑o
__constdata:FFFFFF8000C5FD08                 db  90h
__constdata:FFFFFF8000C5FD09                 db    1
__constdata:FFFFFF8000C5FD0A                 db    0
__constdata:FFFFFF8000C5FD0B                 db    0
__constdata:FFFFFF8000C5FD0C                 db 0AAh
__constdata:FFFFFF8000C5FD0D                 db    1
__constdata:FFFFFF8000C5FD0E                 db    0
__constdata:FFFFFF8000C5FD0F                 db    0
__constdata:FFFFFF8000C5FD10                 db  34h ; 4
__constdata:FFFFFF8000C5FD11                 db  10h
__constdata:FFFFFF8000C5FD12                 db    0
__constdata:FFFFFF8000C5FD13                 db    0
__constdata:FFFFFF8000C5FD14                 db    0
__constdata:FFFFFF8000C5FD15                 db    0
__constdata:FFFFFF8000C5FD16                 db    0
__constdata:FFFFFF8000C5FD17                 db    0
__constdata:FFFFFF8000C5FD18                 db    0
__constdata:FFFFFF8000C5FD19                 db    0
__constdata:FFFFFF8000C5FD1A                 db    0
__constdata:FFFFFF8000C5FD1B                 db    0
__constdata:FFFFFF8000C5FD1C                 db    0
__constdata:FFFFFF8000C5FD1D                 db    0
__constdata:FFFFFF8000C5FD1E                 db    0
__constdata:FFFFFF8000C5FD1F                 db    0
__constdata:FFFFFF8000C5FD20                 db    0
__constdata:FFFFFF8000C5FD21                 db    0
__constdata:FFFFFF8000C5FD22                 db    0
__constdata:FFFFFF8000C5FD23                 db    0
__constdata:FFFFFF8000C5FD24                 db    0
__constdata:FFFFFF8000C5FD25                 db    0
__constdata:FFFFFF8000C5FD26                 db    0
__constdata:FFFFFF8000C5FD27                 db    0
__constdata:FFFFFF8000C5FD28                 dq offset sub_FFFFFF80002C0DA0
__constdata:FFFFFF8000C5FD30                 db    2
```

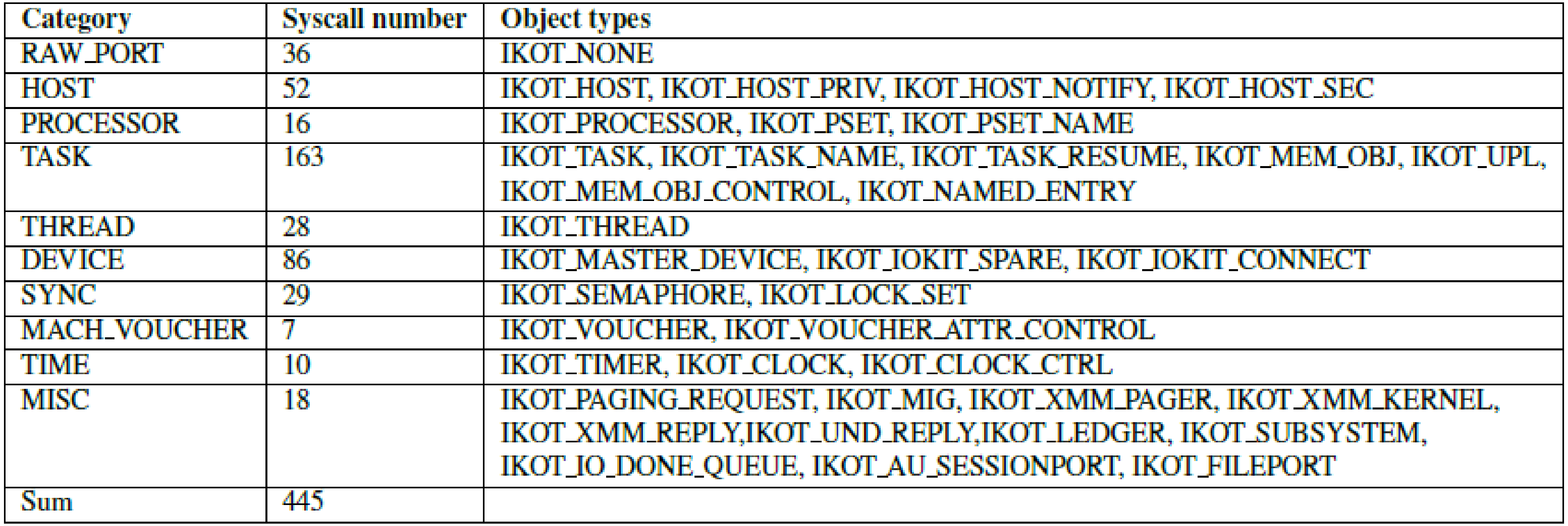


```
__int64 __fastcall host_priv_server_routine(__int64 a1)
{
  signed __int64 v1; // rcx
  __int64 result; // rax
  signed __int64 v3; // rcx

  v1 = *(signed int *)(a1 + 28);
  result = 0LL;
  if ( v1 >= 400 )
  {
    v3 = v1 - 400;
    if ( (signed int)v3 <= 25 )
      result = (__int64)*(&host_priv_subsystem + 5 * v3 + 5);
  }
  return result;
}
```



```
char __fastcall sub_FFFFFF80002C0FC0(mach_msg_header_t *a1, mach_msg_header_t *a2)
{
  mach_msg_id_t v2; // ecx
  host_t v3; // eax
  mach_msg_size_t v4; // eax
  mach_msg_size_t v5; // eax
  unsigned __int64 v6; // rax
  __int64 *v7; // rax
  unsigned __int64 v8; // rax
  __int64 *v9; // rax

  if ( kdebug_enable & 1 )
  {
    v8 = __readgsqword(8u);
    if ( v8 )
      v9 = *(__int64 **)(v8 + 976);
    else
      v9 = 0LL;
    sub_FFFFFF80006DFDC0(0LL, 0xFF000649, 0LL, 0LL, 0LL, 0LL, v9);
    if ( (a1->msgh_bits & 0x80000000) != 0 )
      goto LABEL_15;
  }
  else if ( (a1->msgh_bits & 0x80000000) != 0 )
  {
LABEL_15:
    a2[1].msgh_reserved = -304;
    goto LABEL_16;
  }
  if ( a1->msgh_size != 48 )
    goto LABEL_15;
  a2[1].msgh_id = 68;
  v2 = 68;
  if ( a1[1].msgh_id < 0x44u )
    v2 = a1[1].msgh_id;
  a2[1].msgh_id = v2;
  v3 = convert_port_to_host_priv(*(_QWORD *)&a1->msgh_remote_port);
  v4 = host_statistics(v3, a1[1].msgh_reserved, (host_info_t)&a2[2], (mach_msg_type_number_t *)&a2[1].msgh_id);
  a2[1].msgh_reserved = v4;
  if ( v4 )
  {
    a2[1].msgh_reserved = v4;
LABEL_16:
    LOBYTE(v5) = NDR_record.mig_vers;
    *(NDR_record_t *)&a2[1].msgh_remote_port = NDR_record;
    return v5;
  }
  *(NDR_record_t *)&a2[1].msgh_remote_port = NDR_record;
  v5 = 4 * a2[1].msgh_id + 48;
  a2->msgh_size = v5;
  if ( kdebug_enable & 1 )
  {
    v6 = __readgsqword(8u);
    if ( v6 )
      v7 = *(__int64 **)(v6 + 976);
    else
      v7 = 0LL;
    LOBYTE(v5) = sub_FFFFFF80006DFDC0(0LL, 0xFF00064A, 0LL, 0LL, 0LL, 0LL, v7);
  }
  return v5;
}
```

General Purpose Primitives



Category	Syscall number	Object types
RAW_PORT	36	IKOT_NONE
HOST	52	IKOT_HOST, IKOT_HOST_PRIV, IKOT_HOST_NOTIFY, IKOT_HOST_SEC
PROCESSOR	16	IKOT_PROCESSOR, IKOT_PSET, IKOT_PSET_NAME
TASK	163	IKOT_TASK, IKOT_TASK_NAME, IKOT_TASK_RESUME, IKOT_MEM_OBJ, IKOT_UPL, IKOT_MEM_OBJ_CONTROL, IKOT_NAMED_ENTRY
THREAD	28	IKOT_THREAD
DEVICE	86	IKOT_MASTER_DEVICE, IKOT_IOKIT_SPARE, IKOT_IOKIT_CONNECT
SYNC	29	IKOT_SEMAPHORE, IKOT_LOCK_SET
MACH_VOUCHER	7	IKOT_VOUCHER, IKOT_VOUCHER_ATTR_CONTROL
TIME	10	IKOT_TIMER, IKOT_CLOCK, IKOT_CLOCK_CTRL
MISC	18	IKOT_PAGING_REQUEST, IKOT_MIG, IKOT_XMM_PAGER, IKOT_XMM_KERNEL, IKOT_XMM_REPLY,IKOT_UND_REPLY,IKOT_LEDGER, IKOT_SUBSYSTEM, IKOT_IO_DONE_QUEUE, IKOT_AU_SESSIONPORT, IKOT_FILEPORT
Sum	445	

- The Mach subsystem receives incoming **Mach messages** and processes them by performing the requested **operations** to multiple **resources** such as processors, tasks and threads. This approach allows attackers to achieve general and useful **primitives** through Mach messages **without** hijacking the control flow.

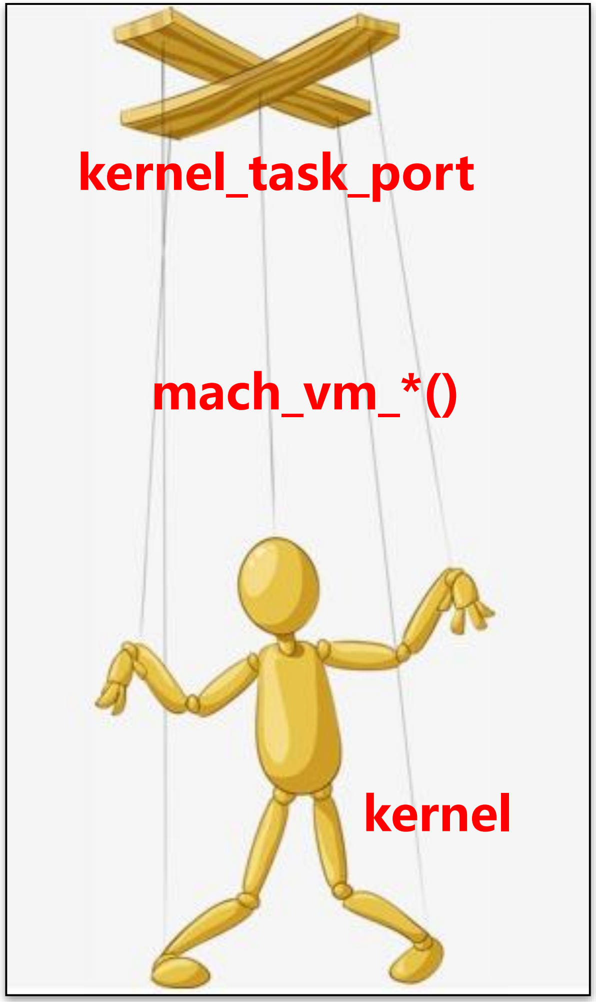
General Purpose Primitives for Host

- Mach represents the overall computer system as a **host** object.

- Through **host_*()** system calls, a userspace app can retrieve information (e.g., host_info()) or set properties (e.g.,host_set_multiuser_config_flags()) for a host.

- Moreover, with a send right to **host_priv** port (like **root** user) and related system calls like host_processor_set_priv(), an attacker can gain send rights to other **powerful ports** (e.g.,processor_set port).

General Purpose Primitives for VM



Virtual memory management:

- XNU provides a powerful set of routines, **mach_vm_*()** system calls, to userspace apps for manipulating task **memory** spaces.
- With an information leak vulnerability or an arbitrary kernel memory read primitive, the attacker could retrieve other tasks' map pointers and craft **fake** tasks to manage other processes' memory space (especially for **kernel**'s memory space).

Querying Primitives

```
kern_return_t
mach_port_kobject(
        ipc_space_t                    space,
        mach_port_name_t               name,
        natural_t                      *typep,
        mach_vm_address_t              *addrp)
{
 ...

    *typep = (unsigned int) ip_kotype(port);
    *addrp =  VM_KERNEL_ADDRPERM\
                 (VM_KERNEL_UNSLIDE(kaddr));
    ip_unlock(port);
    return KERN_SUCCESS;
}
```

- Querying primitives have a characteristic that the **return value** of the system call could be used to **leak** kernel information, e.g., speculating executed code paths.

- For example, mach_port_kobject() is a system call retrieve the type and **address** of the kernel object.

- Both Pangu and TaiG's jailbreaks used it to break KASLR in iOS 7.1 - 8.4, until Apple **removed** the address querying code in the release version (*addrp = 0;).

Querying Primitives

```
kern_return_t clock_sleep_trap(
struct clock_sleep_trap_args *args)
{
mach_port_name_t clock_name = args->clock_name;
…
if (clock_name == MACH_PORT_NULL)
        clock = &clock_list[SYSTEM_CLOCK];
else
        clock = port_name_to_clock(clock_name);
…
if (clock != &clock_list[SYSTEM_CLOCK])
        return (KERN_FAILURE);
…
return KERN_SUCCESS;
}
```

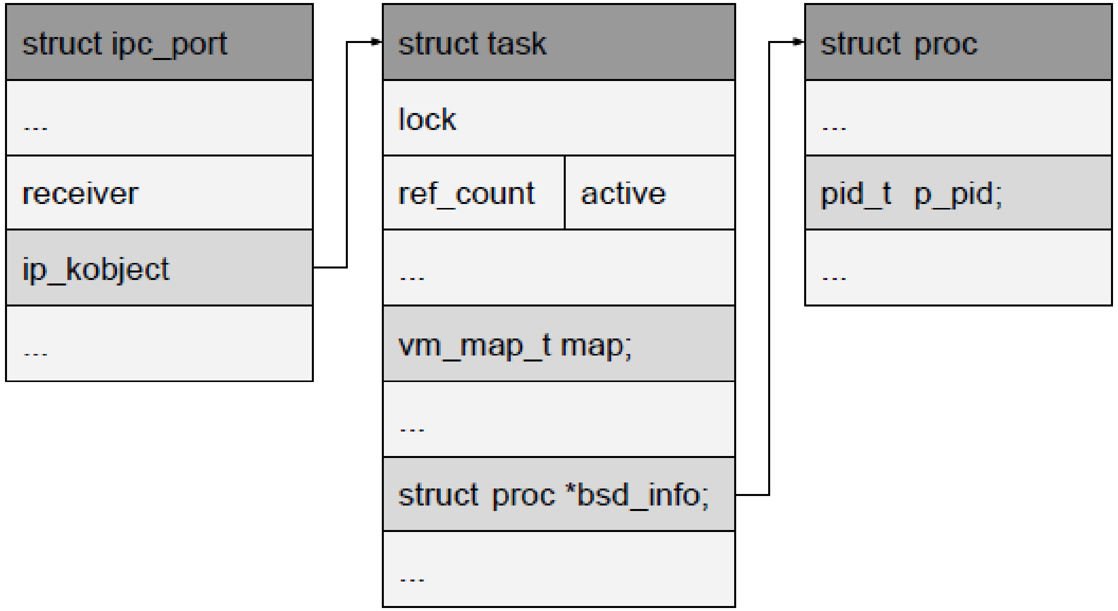
- clock_sleep_trap() is a system call expecting its first argument (if not NULL) to be a send right to the **global** system clock, and it will return **KERN_SUCCESS** if the port name is correct.
- If the attacker can manipulate an ipc_port kernel object and **change** its **ip_kobject** field, a side channel attack could be launched to break KASLR.

Memory Interoperation Primitives

```
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;
…
t1 = port_name_to_task_inspect(t);
…
p = get_bsdtask_info(t1);
if (p) {
            pid  = proc_pid(p);
            err = KERN_SUCCESS;
}
copyout(&pid, pid_addr, sizeof(int));
…
}
```

- By using **type confusion** attack, we can leverage some system calls to copy sensitive data between kernel space and user space. Specifically, some memory interoperation primitives are not used for the original **original intention** of the design.
- **pid_for_task()** is such a system call which returns the **PID** number corresponding to a particular Mach **task**.

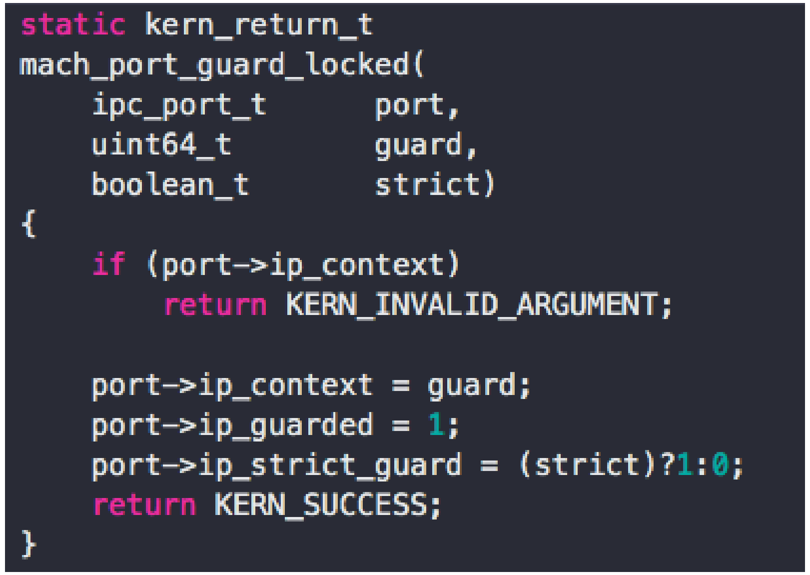
Memory Interoperation Primitives



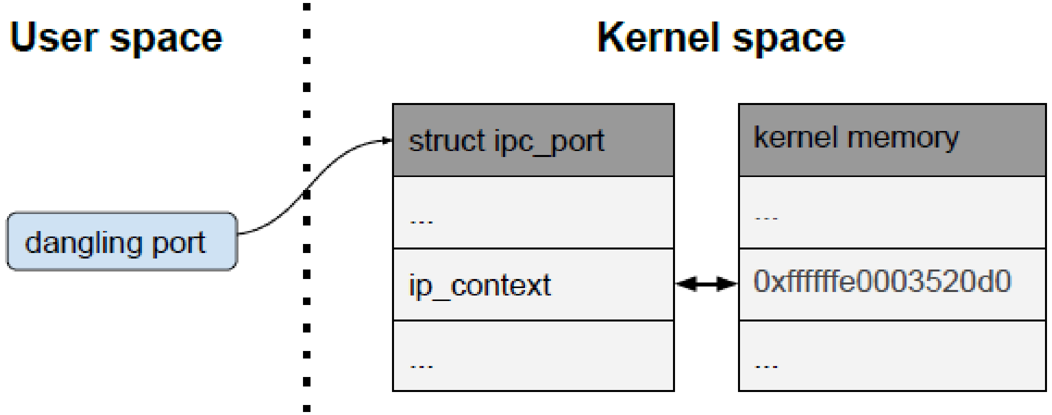
- The function calls port_name_to_task() to get a Mach **task** object, then invokes get_bsdtask_info() to get the **bsd_info** of the Mach task. After getting bsd_info, the function calls proc_pid() to get **PID** number of the Mach task and uses **copyout()** to transmit the PID number to userspace.

- However, the function does not check the validity of the task, and directly returns the value of **task -> bsd_info -> p_pid** to user space after calling get_bsdtask_info() and proc_pid().

Memory Interoperation Primitives

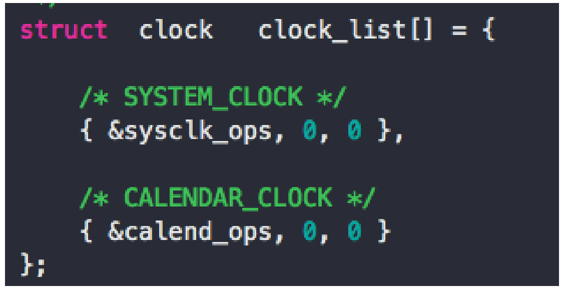


```
static kern_return_t
mach_port_guard_locked(
    ipc_port_t         port,
    uint64_t           guard,
    boolean_t          strict)
{
    if (port->ip_context)
        return KERN_INVALID_ARGUMENT;

    port->ip_context = guard;
    port->ip_guarded = 1;
    port->ip_strict_guard = (strict)?1:0;
    return KERN_SUCCESS;
}
```

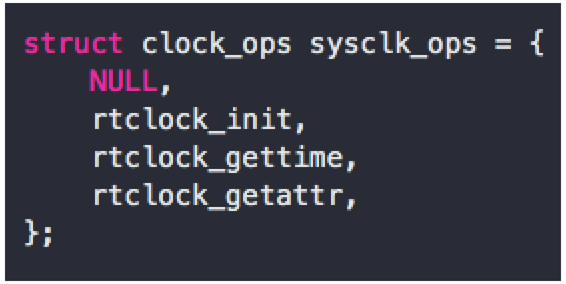


- A port referring to a freed ipc_port object is called a **dangling port**.
- System calls like mach_port_set/get_*(), mach_port_guard/unguard() are used to write and read the **member fields** of the ipc_port object.
- ip_context field in the ipc_port object is used to associate a userspace pointer with a port. By using mach_port_set/get_context() to a dangling port, the attacker can **retrieve** and **set** 64-bits value in the kernel space.

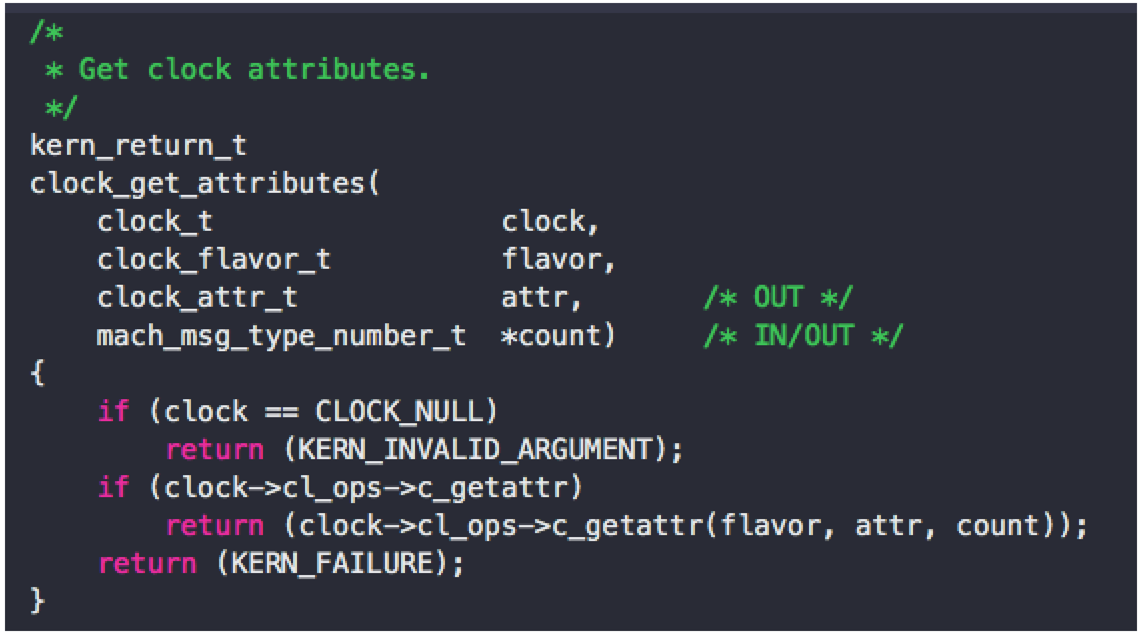
Arbitrary Code Execution Primitives



```
struct  clock   clock_list[] = {

    /* SYSTEM_CLOCK */
    { &sysclk_ops, 0, 0 },

    /* CALENDAR_CLOCK */
    { &calend_ops, 0, 0 }
};
```



```
struct clock_ops sysclk_ops = {
    NULL,
    rtclock_init,
    rtclock_gettime,
    rtclock_getattr,
};
```



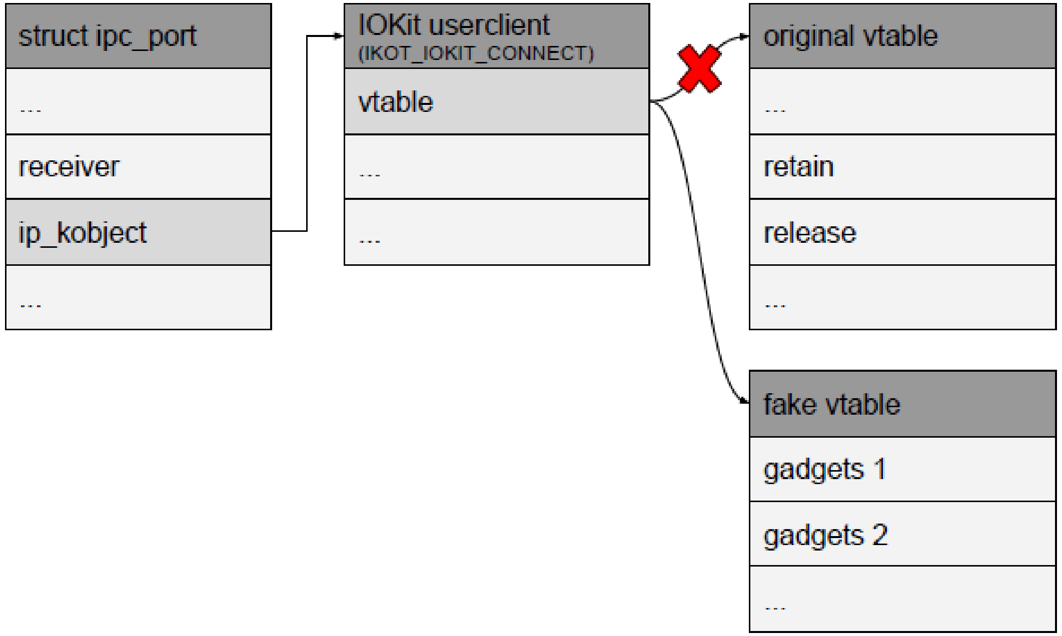
```
/*
 * Get clock attributes.
 */
kern_return_t
clock_get_attributes(
    clock_t                 clock,
    clock_flavor_t          flavor,
    clock_attr_t            attr,       /* OUT */
    mach_msg_type_number_t  *count)     /* IN/OUT */
{
    if (clock == CLOCK_NULL)
        return (KERN_INVALID_ARGUMENT);
    if (clock->cl_ops->c_getattr)
        return (clock->cl_ops->c_getattr(flavor, attr, count));
    return (KERN_FAILURE);
}
```

- This type of primitives can be used to execute kernel code (e.g., a ROP chain or a kernel function) in **arbitrary** addresses.

- clock_get_attributes() is a system call to get attributes of target clock object. An attack can change the **global** function pointers or **fake** an object to hijack the control flow.

- This technique was used in the **Pegasus** APT attack in iOS 9.3.3.

Arbitrary Code Execution Primitives



struct ipc_port
...
receiver
ip_kobject
...

IOKit userclient
(IKOT_IOKIT_CONNECT)
vtable
...
...

original vtable
...
retain
release
...

fake vtable
gadgets 1
gadgets 2
...

- IOKit is an object-oriented device driver framework in XNU that uses a subset of **C++** as its language.

- If the attacker has the kernel write primitives, then he can change the **vtable** entry of an I/OKit userclient to hijack the control flow to the address of a ROP gadget to achieve a kernel code execution primitive.

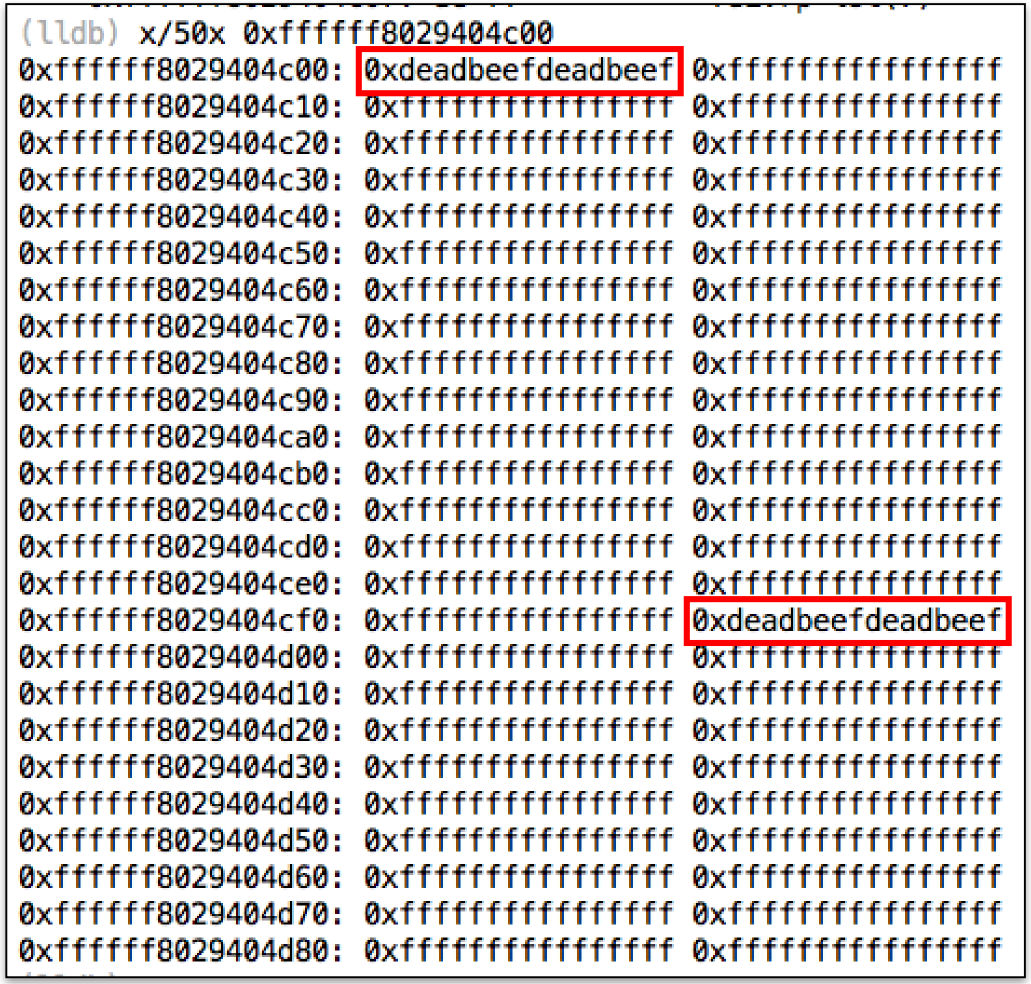
Practical Case Study: Yalu Exp

```
kern_return_t
mach_voucher_extract_attr_recipe_trap(
struct mach_voucher_..._args *args)
{
...
 mach_msg_type_number_t sz = 0;

 copyin(args->recipe_size, (void *)&sz, \
       sizeof(sz));
...
 uint8_t *krecipe = kalloc((vm_size_t)sz);
...
 //args->recipe_size should be sz
 copyin(args->recipe, (void *)krecipe, \
       args->recipe_size)
...
}
```

- CVE-2017-2370 is a heap **buffer overflow** in mach_voucher_extract_attr_recipe_trap().
- The function first copies 4 bytes from the user space pointer args->recipe_size to the **sz** variable. After that, it calls kalloc(sz).
- The function then calls copyin() to copy args->recipe_size sized data from the user space to the **krecipe** (should be **sz**) sized kernel heap buffer. Consequently, it will cause a buffer overflow.

Practical Case Study: Yalu Exp

- **Before heap overflow**

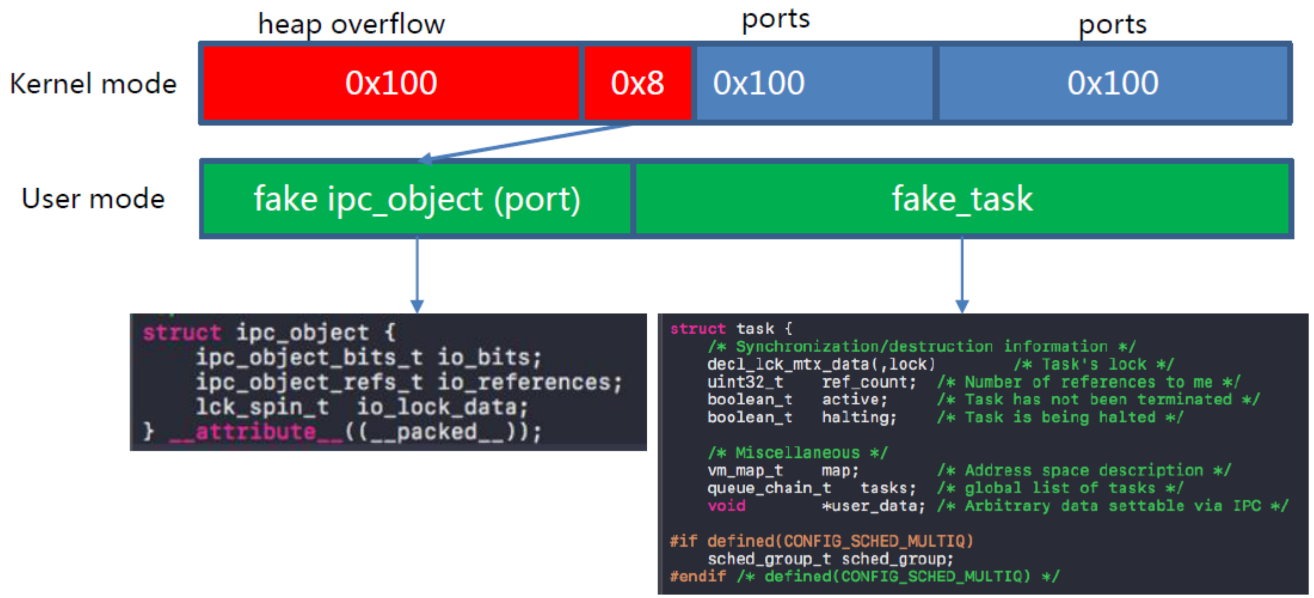


```
(lldb) x/50x 0xffffff8029404c00
0xffffff8029404c00: 0xdeadbeefdeadbeef 0xffffffffffffffff
0xffffff8029404c10: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404c20: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404c30: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404c40: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404c50: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404c60: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404c70: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404c80: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404c90: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404ca0: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404cb0: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404cc0: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404cd0: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404ce0: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404cf0: 0xffffffffffffffff 0xdeadbeefdeadbeef
0xffffff8029404d00: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d10: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d20: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d30: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d40: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d50: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d60: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d70: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d80: 0xffffffffffffffff 0xffffffffffffffff
```

- **After heap overflow**

```
(lldb) x/50x 0xffffff8029404c00
0xffffff8029404c00: 0x4141414141414141 0x4141414141414141
0xffffff8029404c10: 0x4141414141414141 0x4141414141414141
0xffffff8029404c20: 0x4141414141414141 0x4141414141414141
0xffffff8029404c30: 0x4141414141414141 0x4141414141414141
0xffffff8029404c40: 0x4141414141414141 0x4141414141414141
0xffffff8029404c50: 0x4141414141414141 0x4141414141414141
0xffffff8029404c60: 0x4141414141414141 0x4141414141414141
0xffffff8029404c70: 0x4141414141414141 0x4141414141414141
0xffffff8029404c80: 0x4141414141414141 0x4141414141414141
0xffffff8029404c90: 0x4141414141414141 0x4141414141414141
0xffffff8029404ca0: 0x4141414141414141 0x4141414141414141
0xffffff8029404cb0: 0x4141414141414141 0x4141414141414141
0xffffff8029404cc0: 0x4141414141414141 0x4141414141414141
0xffffff8029404cd0: 0x4141414141414141 0x4141414141414141
0xffffff8029404ce0: 0x4141414141414141 0x4141414141414141
0xffffff8029404cf0: 0x4141414141414141 0x4141414141414141
0xffffff8029404d00: 0x4242424242424242 0x4242424242424242
0xffffff8029404d10: 0x4242424242424242 0x4242424242424242
0xffffff8029404d20: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d30: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d40: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d50: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d60: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d70: 0xffffffffffffffff 0xffffffffffffffff
0xffffff8029404d80: 0xffffffffffffffff 0xffffffffffffffff
```

Practical Case Study: Yalu Exp

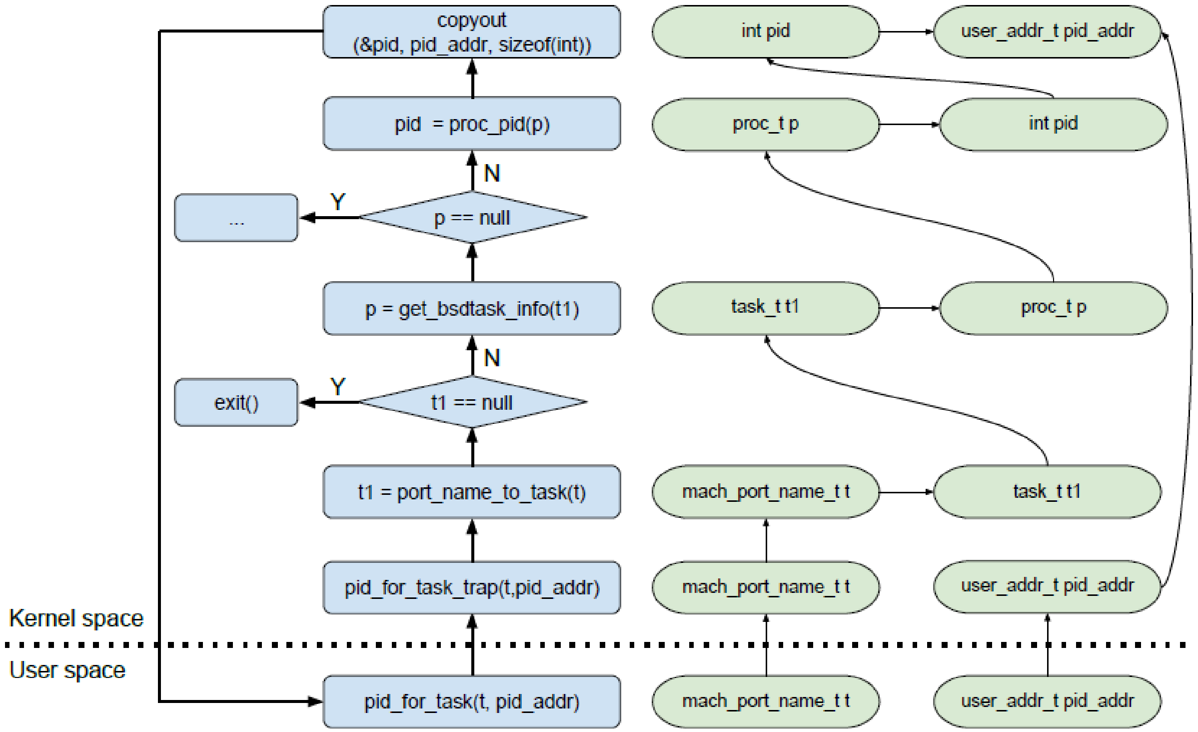


```
struct ipc_object {
    ipc_object_bits_t io_bits;
    ipc_object_refs_t io_references;
    lck_spin_t  io_lock_data;
} __attribute__((__packed__));
```

```
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) */
```

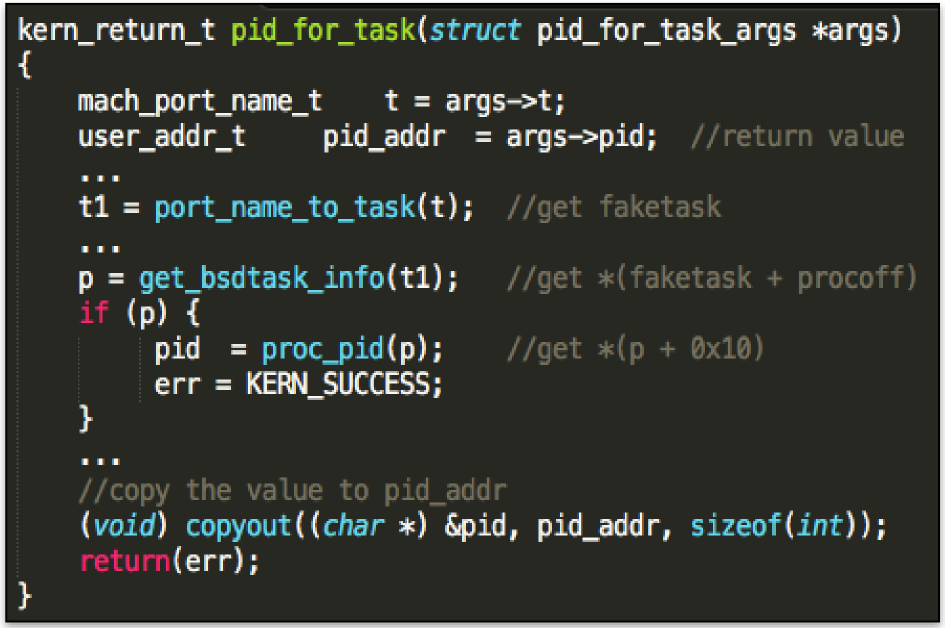
- The exploit overflow those pointers and modify one ipc_object **pointer** to point to a **fake** ipc_object in user mode. The exploit creates a **fake** task in user mode for the fake port as well.
- After that, the exploit chain calls clock_sleep_trap() system call to **brute force** the address of the **global** system clock.

Practical Case Study: Yalu Exp

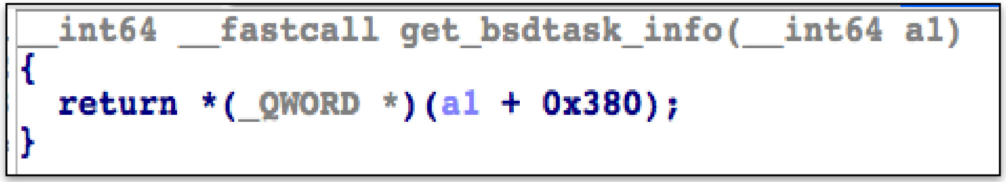


- The exploit sets io_bits of the fake ipc_object to IKOT_TASK and craft a **fake** task for the fake port. By setting the value at the faketask + bsdtask offset, an attacker could **read arbitrary** 32 bits kernel memory through pid_for_task() **without** break KASLR.

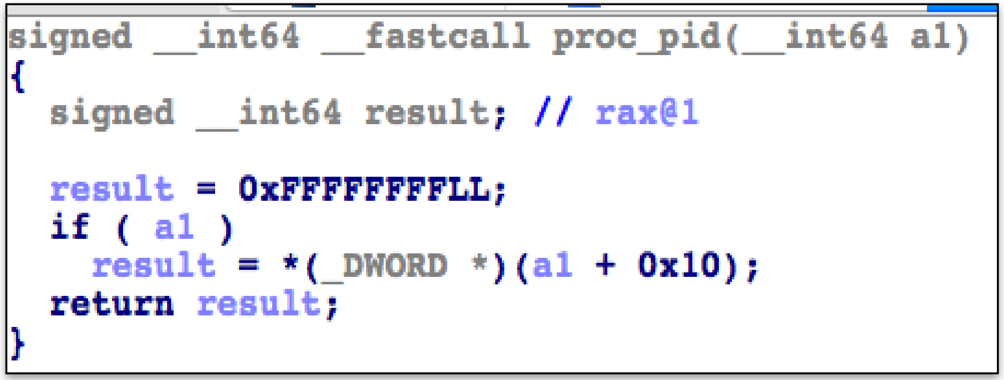
Practical Case Study: Yalu Exp



```
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));
```

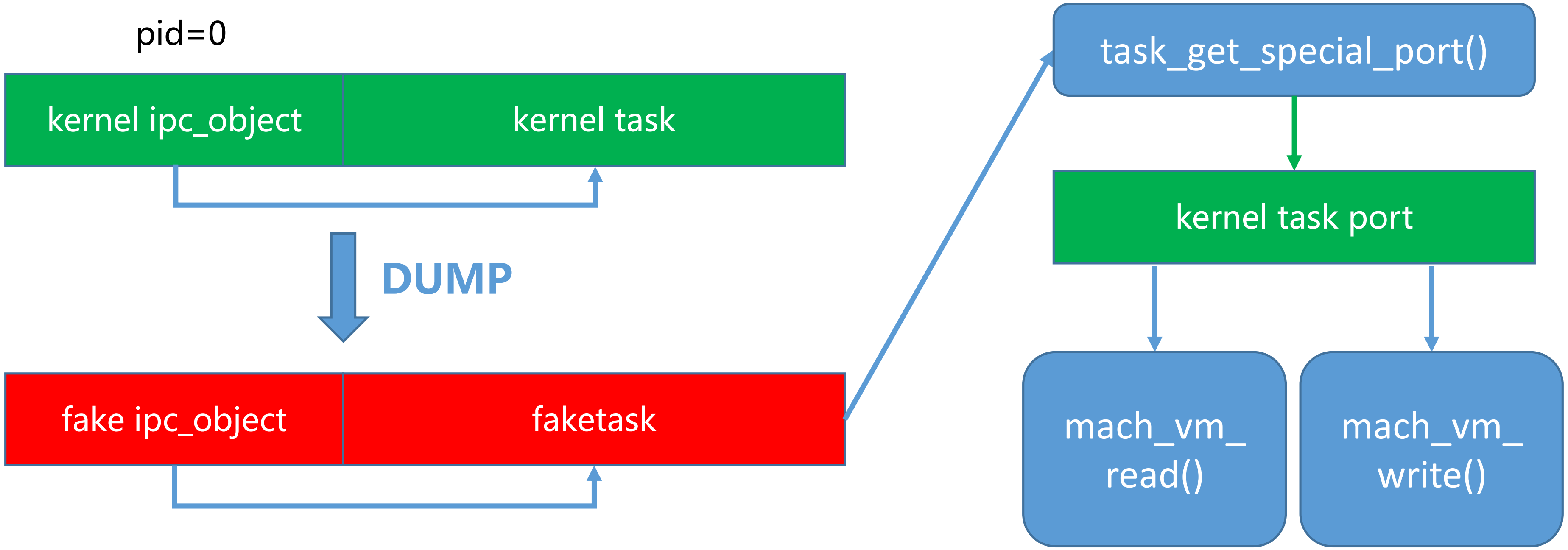
```
read 0xffffff800cc00000 : 0xfeedfacf
```

- As we mentioned before, the function doesn’t check the **validity** of the task, and just return the value of *(*(faketask + 0x380) + 0x10).

Practical Case Study: Yalu Exp

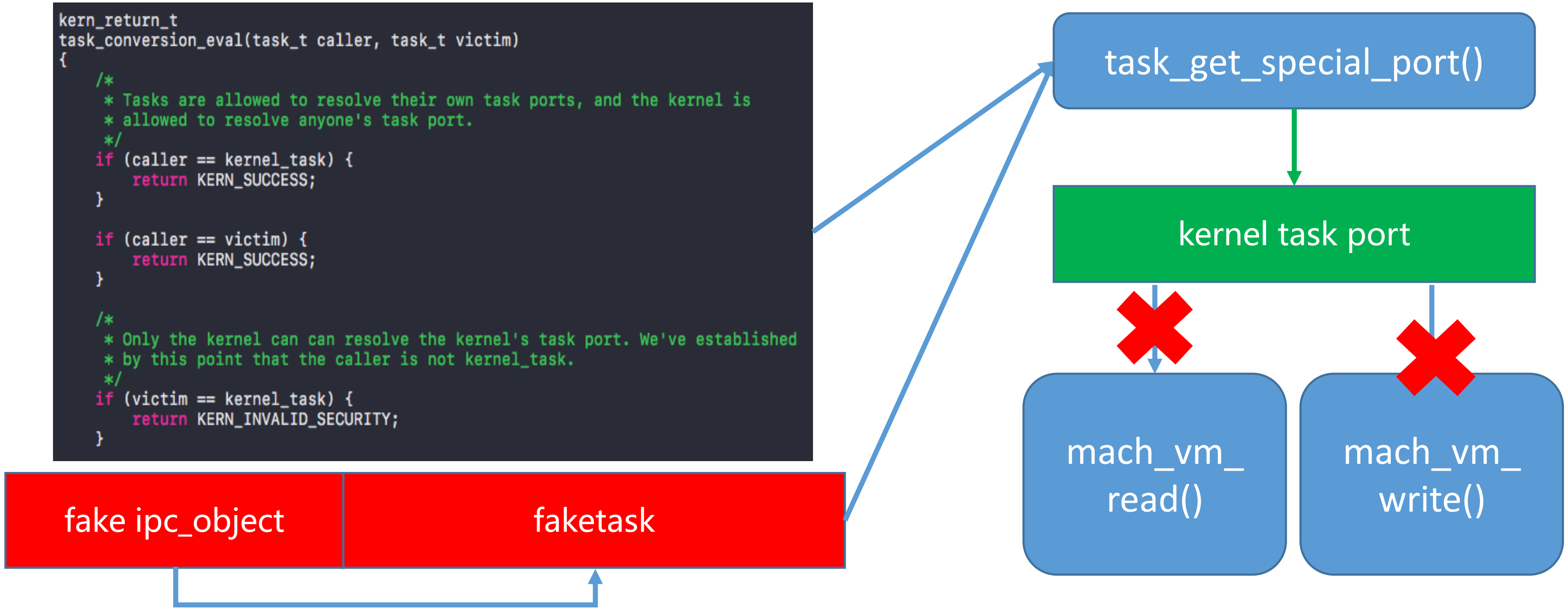
- The attacker **dumps** kernel ipc_object and kernel task to a fake ipc_object and a fake task. By using task_get_special_port() to the fake ipc_object and task, the attacker could get the **kernel task port**.

- Kernel task port can be used to do **arbitrary** kernel memory read and write.



iOS 11 Kernel Task Mitigation

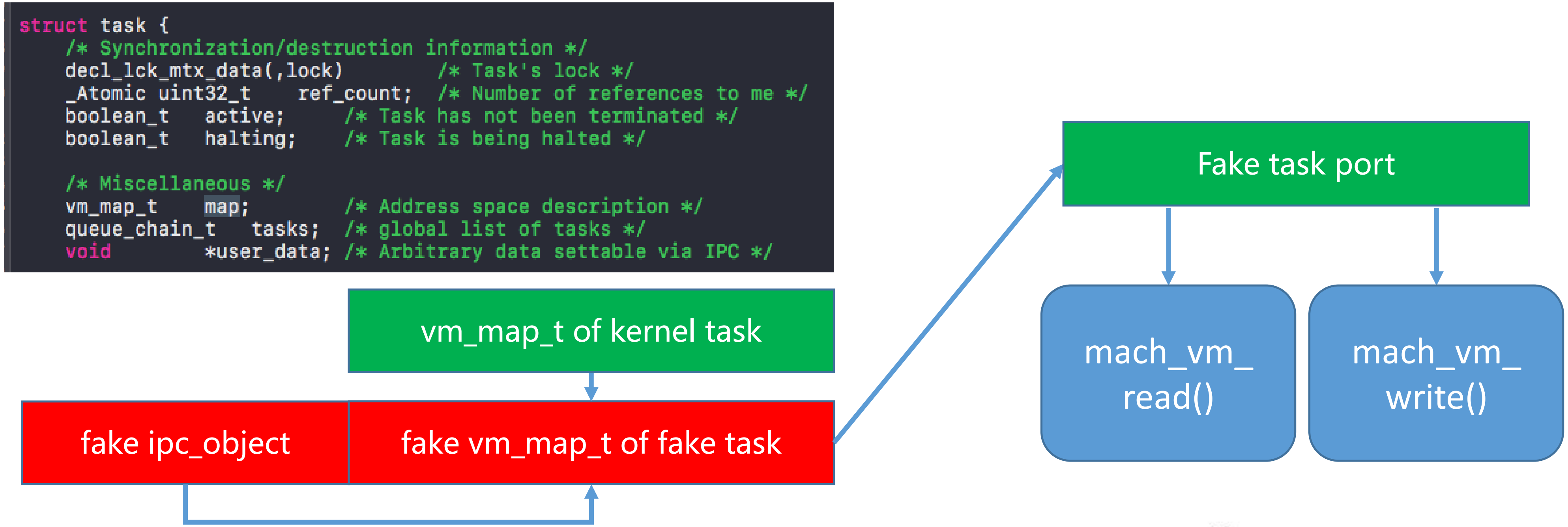
- iOS 11 added a new **mitigation** that only the kernel can resolve the kernel’s task port.
- We **cannot** use the task_get_special_port() trick on iOS 11.



```
kern_return_t
task_conversion_eval(task_t caller, task_t victim)
{
    /*
     * Tasks are allowed to resolve their own task ports, and the kernel is
     * allowed to resolve anyone's task port.
     */
    if (caller == kernel_task) {
        return KERN_SUCCESS;
    }

    if (caller == victim) {
        return KERN_SUCCESS;
    }

    /*
     * Only the kernel can can resolve the kernel's task port. We've established
     * by this point that the caller is not kernel_task.
     */
    if (victim == kernel_task) {
        return KERN_INVALID_SECURITY;
    }
```

Mitigation bypass in Async_wake Exp

- The attacker cannot use a real kernel task port. But the attacker can copy reference pointer of kernel’s **vm** to the fake task.
- Now the fake port has a same **address space** as the kernel task port. It’s enough for the attacker to do arbitrary kernel read/write.



```
struct task {
    /* Synchronization/destruction information */
    decl_lck_mtx_data(,lock)        /* Task's lock */
    _Atomic 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 */
```

vm_map_t of kernel task

fake ipc_object

fake vm_map_t of fake task

Fake task port

mach_vm_read()

mach_vm_write()

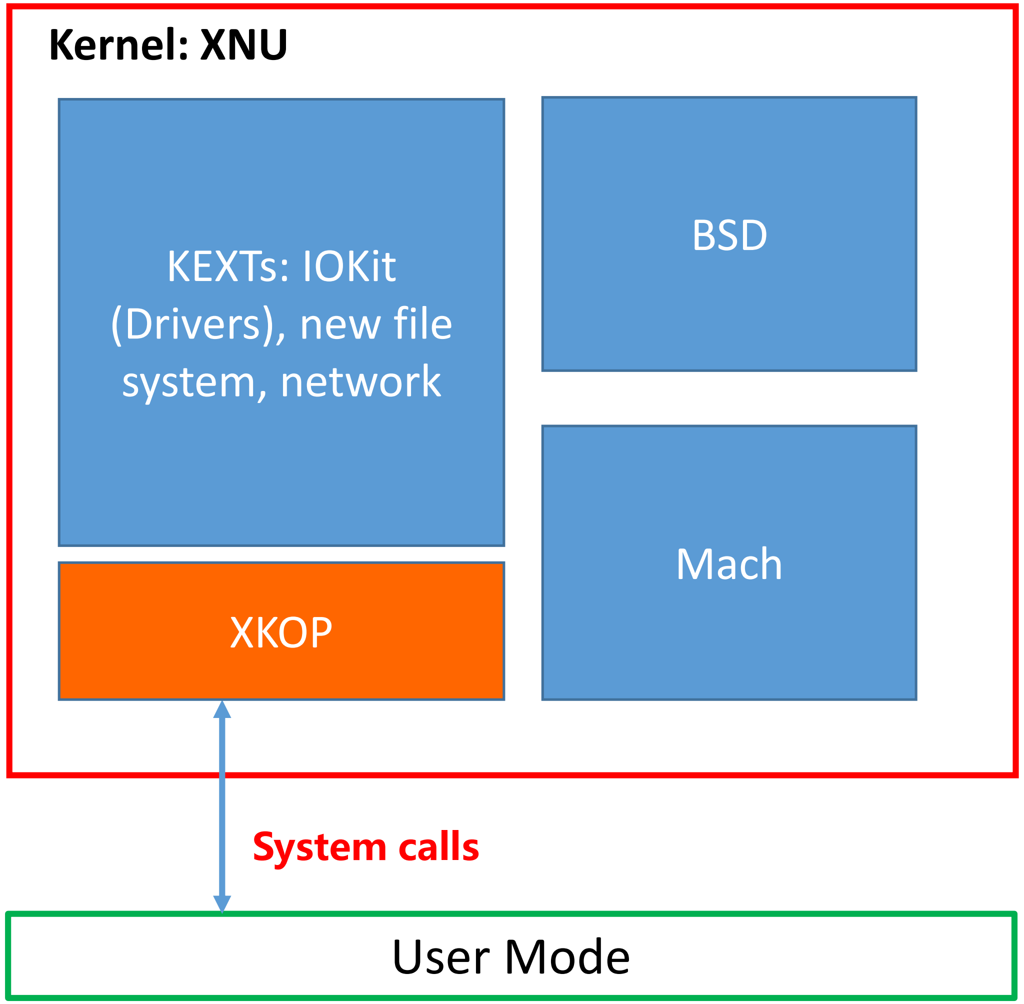
Enterprise Computer Security

Pic from time.com

- Lots of companies (e.g., Alibaba Inc and Tencent) offer Macbooks as work computers to their employees.
- **Problems:**

  1. macOS is not forced to upgrade like iOS.
  2. Less hardware based protections (e.g., AMCC and PAC) on Macbooks.
  3. Less secure sandbox rules than iOS.
- Hard to defend against advanced persistent threat (APT). Enterprise computers need a more **secure** system.

XNU Kernel Object Protector



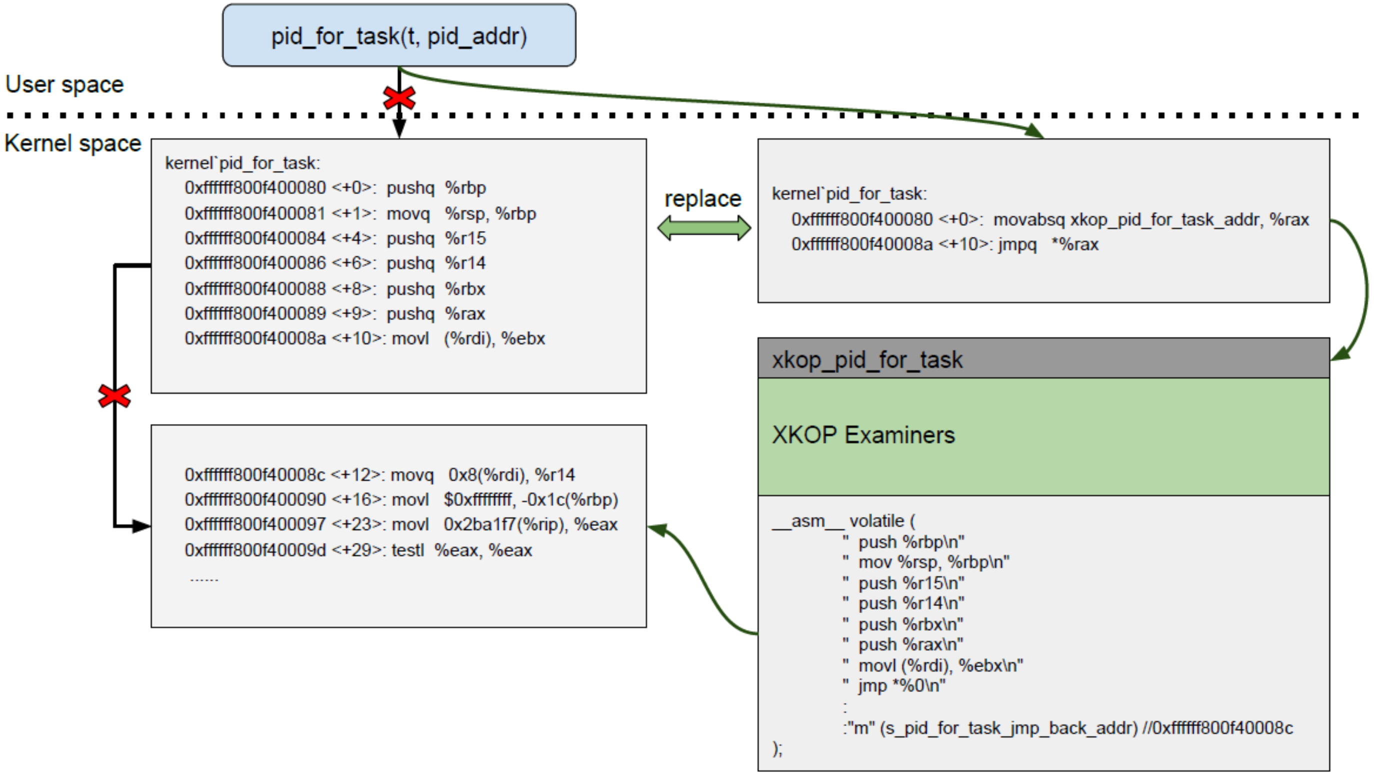
- To mitigate the APT and POP attack, we propose a framework called ***XNU Kernel Object Protector*** (XKOP).
- **Basic idea**: a kernel extension to implement inline hooking for specific system calls and deploy integrity check for ipc_port kernel objects.
- In addition, XKOP could bring **new** mitigations to **old** macOS versions.

Inline Hooking

```
/**
  @file mac_policy.h
  @brief Kernel Interfaces for MAC policy modules

  This header defines the list of operations that are defined by the
  TrustedBSD MAC Framwork on Darwin.  MAC Policy modules register
  with the framework to declare interest in a specific set of
  operations.  If interest in an entry point is not declared, then
  the policy will be ignored when the Framework evaluates that entry
  point.
*/

#ifndef _SECURITY_MAC_POLICY_H_
#define _SECURITY_MAC_POLICY_H_

#ifndef PRIVATE
#warning "MAC policy is not KPI, see Technical Q&A QA1574, this header
#endif

#include <security/_label.h>

struct attrlist;
struct auditinfo;
struct bpf_d;
struct cs_blob;
struct devnode;
struct exception_action;
struct fileglob;
struct ifnet;
struct inpcb;
struct ipq;
struct label;
struct mac_module_data;
```

- Our system needs to find reliable code points that the **examiners** could be executed.
- **KAuth** kernel subsystem exports a KPI that allows third-party developers to authorize actions within the kernel. However, the operation set is very limited.
- **MAC** framework is private and can only be used by Apple. In addition, the rules are hardcoded in the code of the XNU kernel.
- Finally, we choose **inline hooking**.

Inline Hooking



```
kernel`pid_for_task:
    0xffffff800f400080 <+0>:  pushq  %rbp
    0xffffff800f400081 <+1>:  movq   %rsp, %rbp
    0xffffff800f400084 <+4>:  pushq  %r15
    0xffffff800f400086 <+6>:  pushq  %r14
    0xffffff800f400088 <+8>:  pushq  %rbx
    0xffffff800f400089 <+9>:  pushq  %rax
    0xffffff800f40008a <+10>: movl   (%rdi), %ebx
```

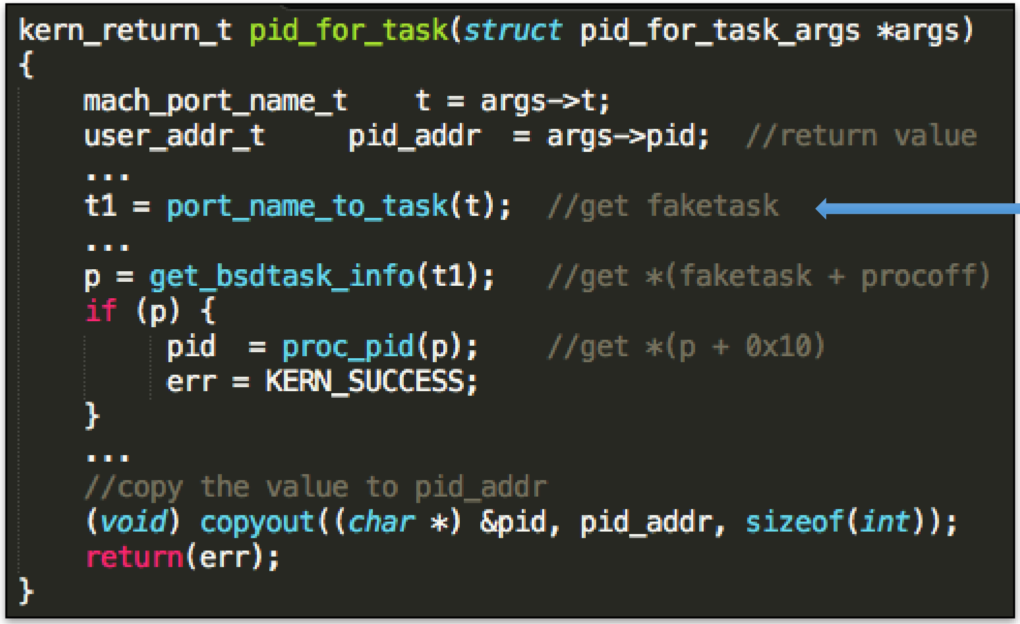
```
    0xffffff800f40008c <+12>: movq   0x8(%rdi), %r14
    0xffffff800f400090 <+16>: movl   $0xffffffff, -0x1c(%rbp)
    0xffffff800f400097 <+23>: movl   0x2ba1f7(%rip), %eax
    0xffffff800f40009d <+29>: testl  %eax, %eax
     ......
```

```
kernel`pid_for_task:
    0xffffff800f400080 <+0>:  movabsq xkop_pid_for_task_addr, %rax
    0xffffff800f40008a <+10>: jmpq   *%rax
```

```
__asm__ volatile (
            "  push %rbp\n"
            "  mov %rsp, %rbp\n"
            "  push %r15\n"
            "  push %r14\n"
            "  push %rbx\n"
            "  push %rax\n"
            "  movl (%rdi), %ebx\n"
            "  jmp *%0\n"
            :
            :"m" (s_pid_for_task_jmp_back_addr) //0xffffff800f40008c
);
```

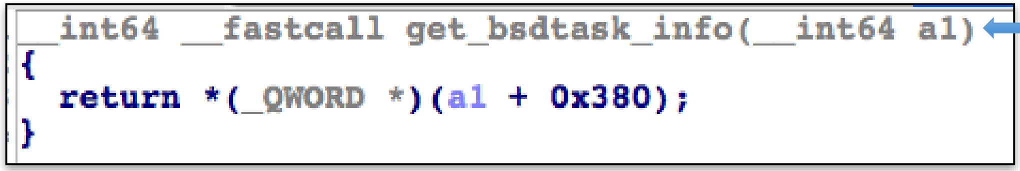
- Based on the examiners, XKOP replaces the original code entry of the target system call into a **trampoline**. The trampoline jumps to the **examiner** stored in the XKOP kernel extension. Then, the examiner verifies the **integrity** of the target kernel object.

Examiners



```
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);
}
```

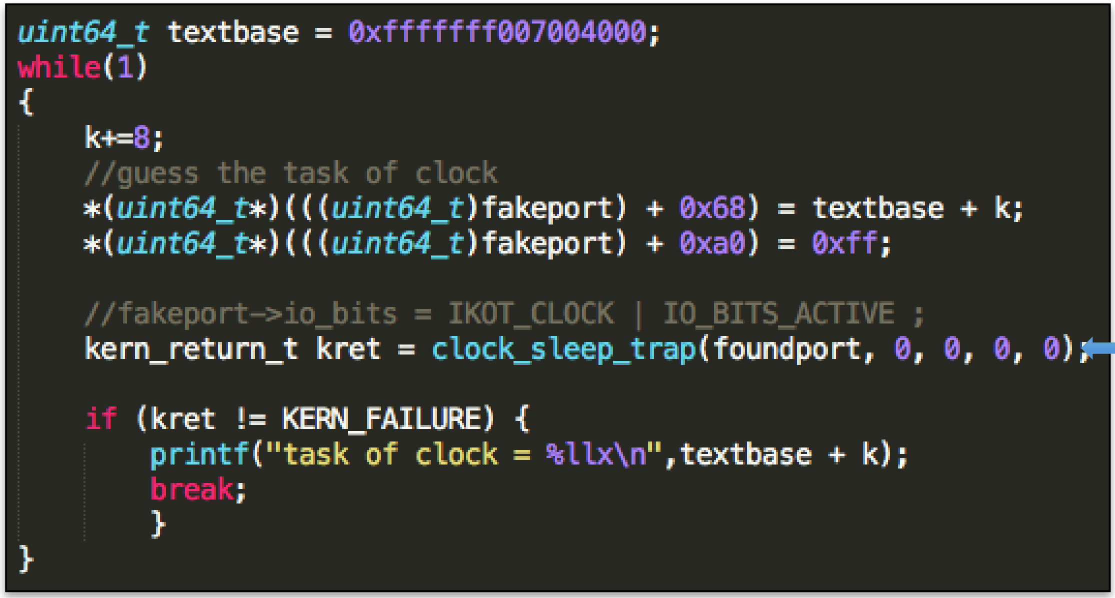
Kernel object address checker:
t1 should not be in the user space address. Must break KASLR first and put the payload into kernel. Just like a soft SMAP for old devices.



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

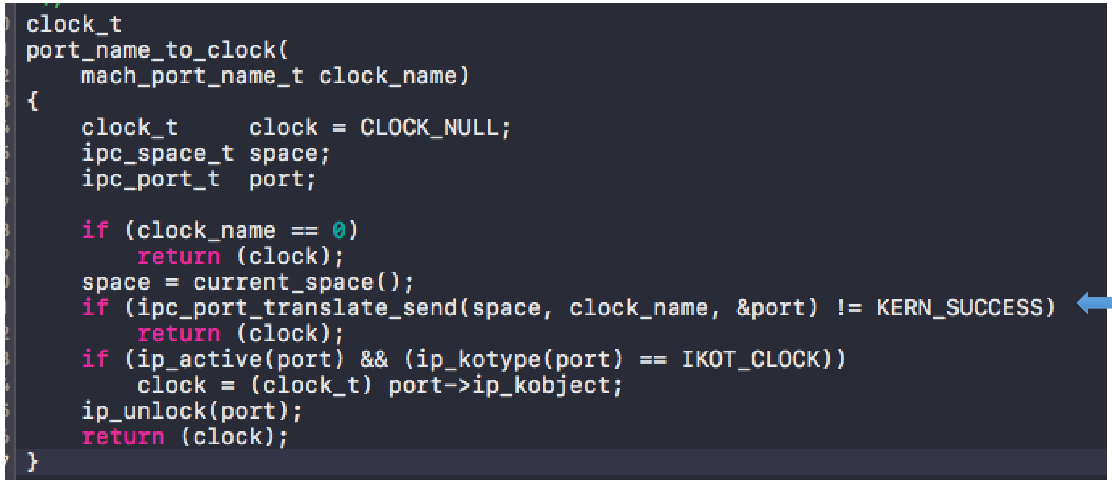
Kernel object type examiner:
a1 should be a real badtask_info structure with a valid pid number.

Examiners



```
uint64_t textbase = 0xfffffff007004000;
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;
        }
}
```

Through brute force attacks, clock_sleep_trap() can be used to guess the address of global clock object and break the KASLR.

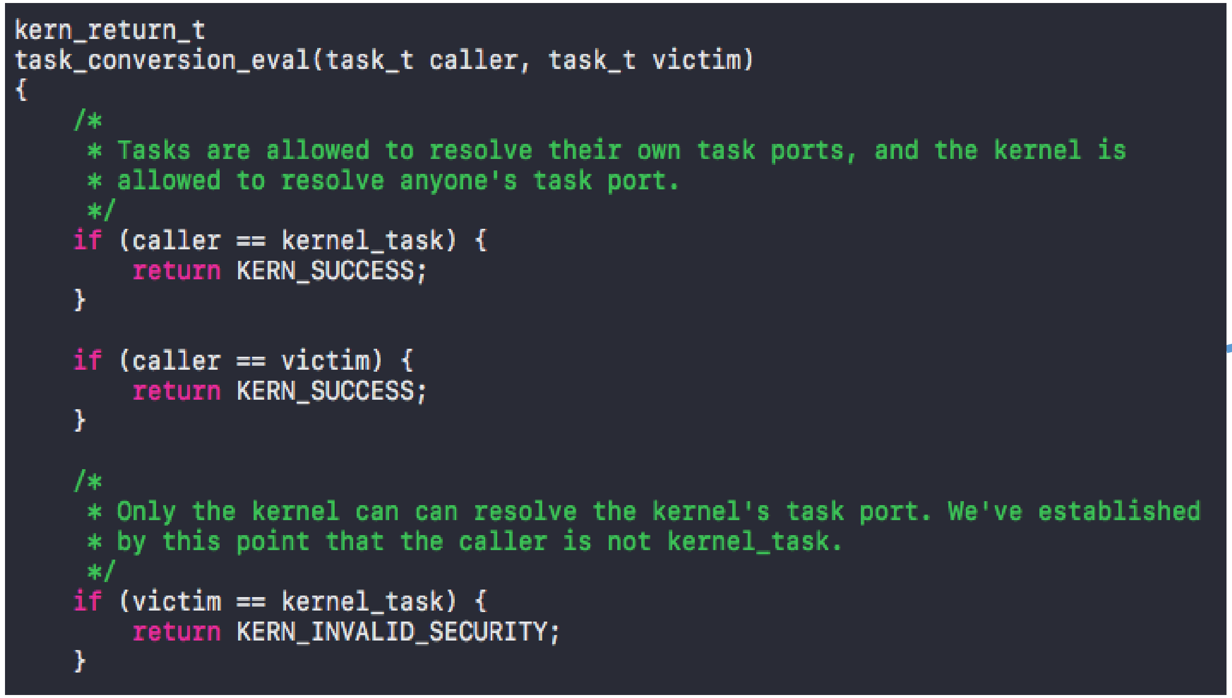


```
clock_t
port_name_to_clock(
    mach_port_name_t clock_name)
{
    clock_t     clock = CLOCK_NULL;
    ipc_space_t space;
    ipc_port_t  port;

    if (clock_name == 0)
        return (clock);
    space = current_space();
    if (ipc_port_translate_send(space, clock_name, &port) != KERN_SUCCESS)
        return (clock);
    if (ip_active(port) && (ip_kotype(port) == IKOT_CLOCK))
        clock = (clock_t) port->ip_kobject;
    ip_unlock(port);
    return (clock);
}
```

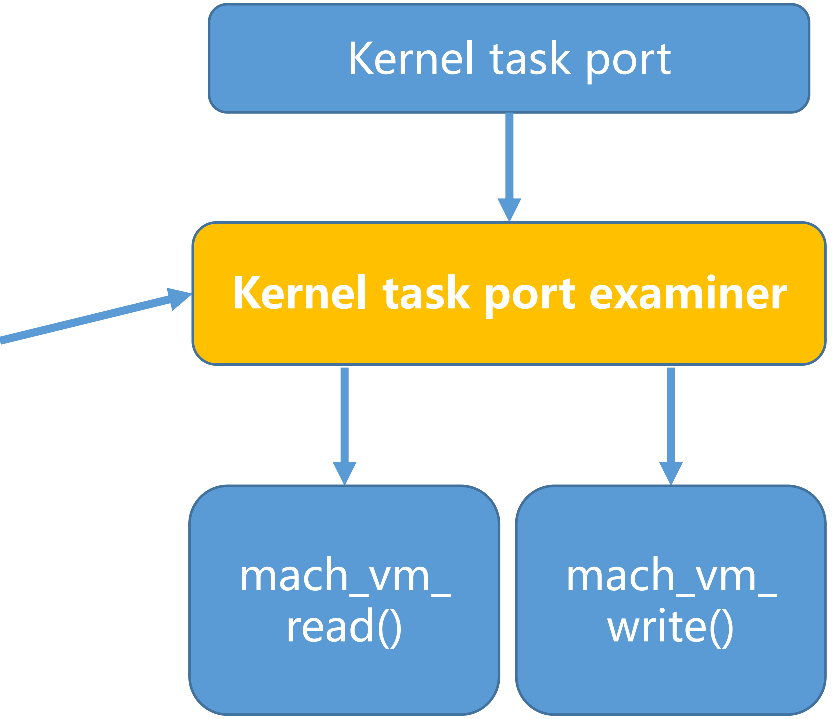
Kernel object querying examiner: if the function returns too many errors, warning the user or panic according to the configuration.

Examiners

- **Kernel task port examiner**: firstly, bring task_conversion_eval(task_t caller, task_t victim) mitigation to old macOS system versions. Only the kernel can resolve the kernel's task port.

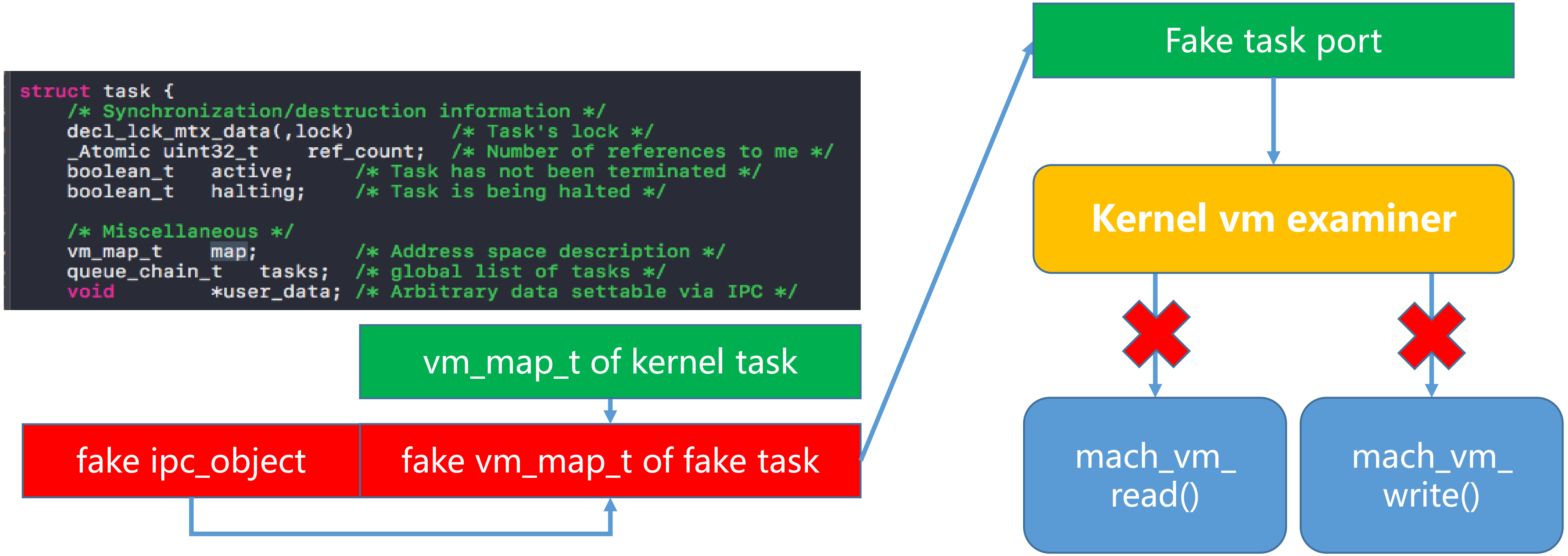


```
kern_return_t
task_conversion_eval(task_t caller, task_t victim)
{
    /*
     * Tasks are allowed to resolve their own task ports, and the kernel is
     * allowed to resolve anyone's task port.
     */
    if (caller == kernel_task) {
        return KERN_SUCCESS;
    }

    if (caller == victim) {
        return KERN_SUCCESS;
    }

    /*
     * Only the kernel can can resolve the kernel's task port. We've established
     * by this point that the caller is not kernel_task.
     */
    if (victim == kernel_task) {
        return KERN_INVALID_SECURITY;
    }
```



Examiners

- **Kernel vm examiner** for mach_vm_*(): if the caller process does not belong to kernel (pid == 0) and the target ipc_port object has the same map structure with the one of a kernel task, the examiner will trigger configured operations, e.g., error return or panic.



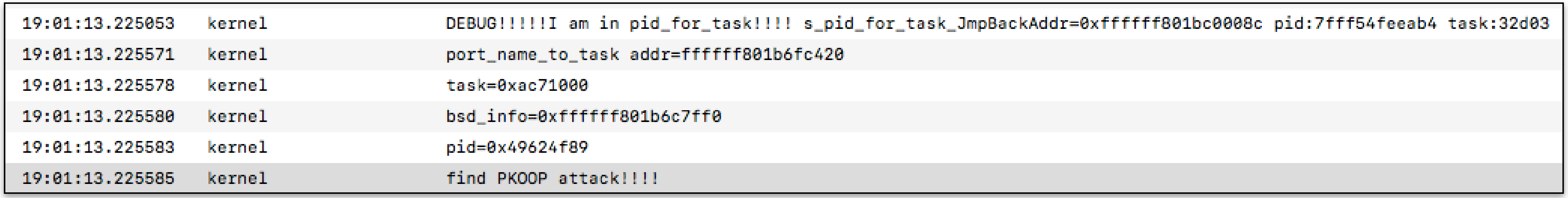
```
struct task {
    /* Synchronization/destruction information */
    decl_lck_mtx_data(,lock)        /* Task's lock */
    _Atomic uint32_t    ref_count;  /* Number of references to me */
    boolean_t   active;     /* Task has not been terminated */
    boolean_t   halting;    /* Task is being halted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

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

Evaluation

- We selected 4 kernel vulnerabilities (two for each version of macOS) and available exploits to **evaluate** the effectiveness of our system.

macOS version	Vulnerability (CVE)	XKOP Protection
10.12	CVE-2016-4669	YES
	CVE-2017-2370	YES
10.13	CVE-2017-13861	YES
	CVE-2018-4241	YES

- We first ensure that the **exploits** work on the corresponding systems, and then we deploy the XKOP framework and run the exploits again to check whether our system detects and blocks the attack.



```
19:01:13.225053    kernel                DEBUG!!!!!I am in pid_for_task!!!! s_pid_for_task_JmpBackAddr=0xffffff801bc0008c pid:7fff54feeab4 task:32d03
19:01:13.225571    kernel                port_name_to_task addr=ffffff801b6fc420
19:01:13.225578    kernel                task=0xac71000
19:01:13.225580    kernel                bsd_info=0xffffff801b6c7ff0
19:01:13.225583    kernel                pid=0x49624f89
19:01:13.225585    kernel                find PKOOP attack!!!!
```

- The experiment result shows that XKOP provides **deterministic** protection for every vulnerability and blocks each attempt to exploit the system.

Discussion

- Unfortunately, XKOP cannot mitigate **all** kinds of POP primitives:

  (1). Querying primitives use error return values to gain an extra source of information which is very similar to the **side-channel** attack.

  (2). No protection for arbitrary code execution primitives. Without hardware support, software-based CFI implementation can be very **expensive**. In addition, modern kernel could be patched by **pure data** which means kernel memory read and write primitives are enough for attackers to accomplish the aim.

- We may miss some potential vulnerabilities that can bypass XKOP protection. As an imperfect solution, XKOP supports **extensible** examiners to prevent new threats in the first place.

Conclusion

- We discuss the **mitigation** techniques in the XNU kernel, i.e., the kernel of iOS and macOS, and how these mitigations make the traditional exploitation technique ineffective.

- We summarize a new attack called **POP** that leverages multiple ipc_port kernel objects to bypass these mitigations.

- A defense mechanism called XNU Kernel Object Protector (**XKOP**) is proposed to protect the integrity of the kernel objects in the XNU kernel.

Contact information:

- weibo@蒸米spark
- twitter@SparkZheng

Reference

- *OS Internals & Jtool: http://newosxbook.com/
- A Brief History of Mitigation: The Path to EL1 in iOS 11, Ian Beer
- Yalu: https://github.com/kpwn/yalu102, qwertyoruiopz and marcograssi
- iOS 10 Kernel Heap Revisited, Stefan Esser
- Port(al) to the iOS Core, Stefan Esser
- iOS/MacOS kernel double free due to IOSurfaceRootUserClient not respecting MIG ownership rules, Google. https://bugs.chromium.org/p/project-zero/issues/detail?id=1417
- mach voucher buffer overflow. https://bugs.chromium.org/p/projectzero/issues/detail?id=1004
- Mach portal: https://bugs.chromium.org/p/project-zero/issues/detail?id=965
- PassiveFuzzFrameworkOSX: https://github.com/SilverMoonSecurity/PassiveFuzzFrameworkOSX

black hat

EUROPE 2018

DECEMBER 3-6, 2018

EXCEL LONDON / UNITED KINGDOM

Thank you!

#BHEU / @BLACK HAT EVENTS