Port(al) to the iOS Core

Introduction to a previously private iOS Kernel Exploitation Technique

March, 2017

Who am I?

- Stefan Esser
- from Germany
- in Information Security since 1998
- SektionEins GmbH from (2007 - 2016)
- Antid0te UG (2013 - now)

What is this talk about?

- a “new” (set of) iOS kernel exploitation technique(s)
- previously only discussed in my iOS Kernel Exploitation trainings
- part of teaching material since around 2015
- trainee from Dec 2016 leaked it within one month to developers of Yalu
- who then distributed an iOS 10.2 jailbreak using this technique in Jan 2017

Previous iOS Kernel Heap Feng Shui / Exploitation Techniques

- **BlackHat 2012 - iOS Kernel Heap Armageddon Revisited**
  - Author: Stefan Esser
  - Idea: Fill kernel heap with C++ objects via OSUnserializeXML() and overwrite them
  - Status: Apple mitigated but a slightly modified technique still usable in iOS 10

- **Hack In The Box 2012 - iOS 6 Security**
  - Author(s): Mark Dowd / Tarjei Mandt
  - Idea: Fill heap with vm_copy_t structures and get information leaks and extended buffer overflows from overwriting them
  - Status: Apple added mitigations so that technique got less and less valuable

Status of public iOS Kernel Exploitation

- everybody is using the public heap feng shui techniques
- bugs are often overflows or UAF
- exploitation often targets **vm_map_copy_t** or kernel **C++ objects**
- Apple keeps adding mitigations against the publicly seen techniques
- public techniques become less and less usable

➡ we need a different / new technique

Ingredients of our Kernel Exploitation Technique(s)

1. idea for a different / new kernel data structure to attack
2. way to fill the kernel heap with this structure or pointers to it
3. strategy how to continue once overwritten

What Kernel Data Structure should we attack?

- there are for sure many data structures in the kernel
- but when you look at the Mach part of the kernel
- one data structure jumps into your face immediately

➡ mach ports!

What are Mach Ports?

What are Mach Ports?

- likely the most important data structure in Mach part of kernel
- have multiple purposes
  - act like handles to kernel objects / subsystems
  - allow sending / receiving messages for IPC
- stored internally in **ipc_port_t** structure

ipc_port_t (I)

IPC ports are internally hold in the following structure defined in /osfmk/ipc/ipc_port.h

```c
struct ipc_port {

    /*
     * Initial sub-structure in common with ipc_pset
     * First element is an ipc_object second is a
     * message queue
     */
    struct ipc_object ip_object;
    struct ipc_mqueue ip_messages;

    union {
        struct ipc_space *receiver;
        struct ipc_port *destination;
        ipc_port_timestamp_t timestamp;
    } data;

    union {
        ipc_kobject_t kobject;
        ipc_importance_task_t imp_task;
        uintptr_t alias;
    } kdata;

    struct ipc_port *ip_nsrequest;
    struct ipc_port *ip_pdrequest;
    struct ipc_port_request *ip_requests;
    struct ipc_kmsg *ip_premsg;

    mach_port_mscount_t ip_mscount;
    mach_port_rights_t ip_srights;
    mach_port_rights_t ip_sorights;
```

ipc_port_t (II)

IPC ports are internally hold in the following structure

defined in /osfmk/ipc/ipc_port.h

```
    natural_t ip_sprequests:1,  /* send-possible requests outstanding */
          ip_spimportant:1, /* ... at least one is importance donating */
          ip_impdonation:1, /* port supports importance donation */
          ip_tempowner:1,   /* dont give donations to current receiver */
          ip_guarded:1,         /* port guarded (use context value as guard) */
          ip_strict_guard:1,    /* Strict guarding; Prevents user manipulation of context values directly */
          ip_reserved:2,
          ip_impcount:24;   /* number of importance donations in nested queue */

    mach_vm_address_t ip_context;


#if MACH_ASSERT
#define IP_NSPARES      4
#define IP_CALLSTACK_MAX    16
    queue_chain_t   ip_port_links;  /* all allocated ports */
    thread_t    ip_thread;  /* who made me?  thread context */
    unsigned long   ip_timetrack;   /* give an idea of "when" created */
    uintptr_t   ip_callstack[IP_CALLSTACK_MAX]; /* stack trace */
    unsigned long   ip_spares[IP_NSPARES]; /* for debugging */
#endif  /* MACH_ASSERT */
};
```

ipc_object_t (I)

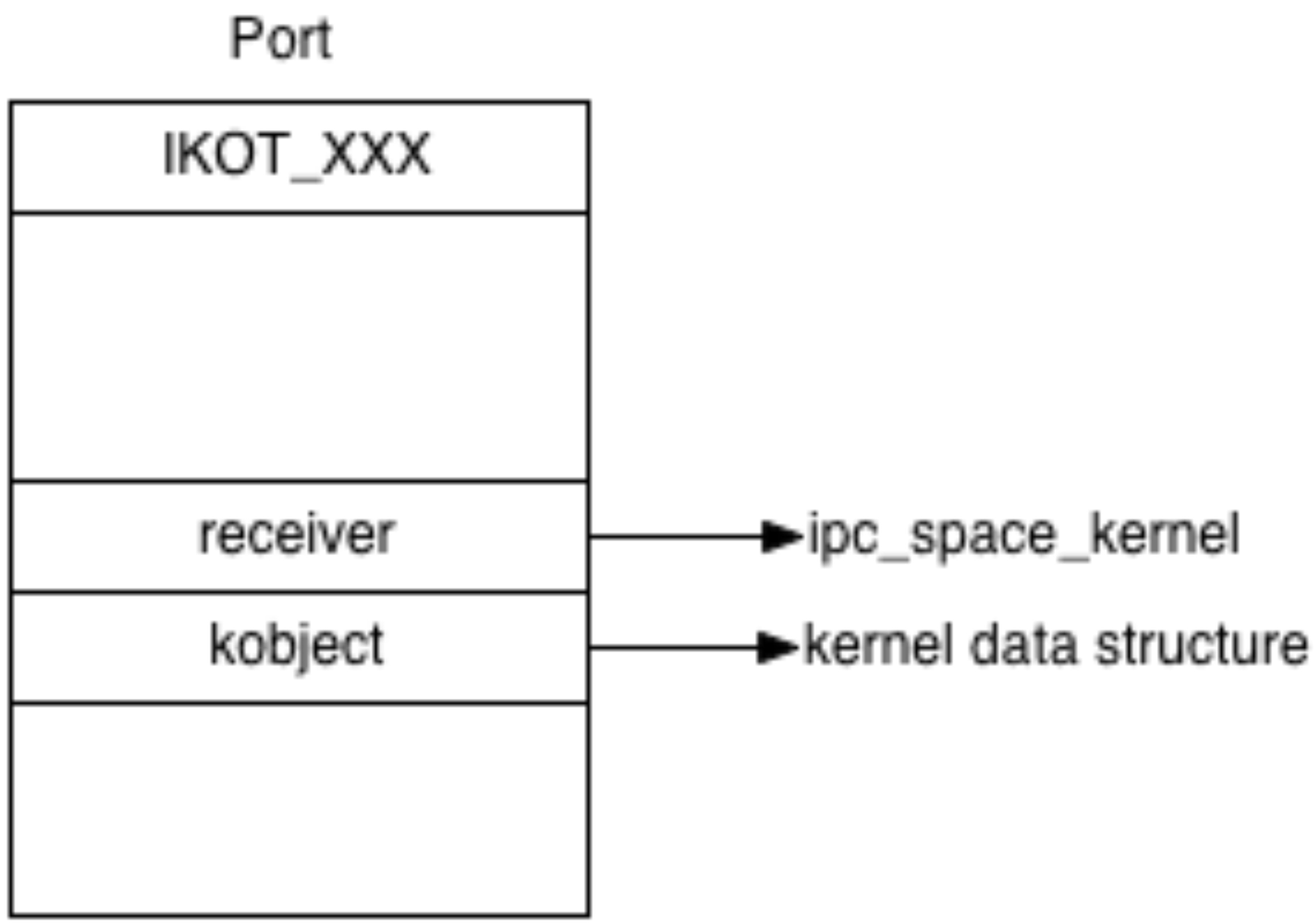
common data structure for IPC objects like ports

defined in /osfmk/ipc/ipc_object.h

```
/*
 * The ipc_object is used to both tag and reference count these two data
 * structures, and (Noto Bene!) pointers to either of these or the
 * ipc_object at the head of these are freely cast back and forth; hence
 * the ipc_object MUST BE FIRST in the ipc_common_data.
 *
 * If the RPC implementation enabled user-mode code to use kernel-level
 * data structures (as ours used to), this peculiar structuring would
 * avoid having anything in user code depend on the kernel configuration
 * (with which lock size varies).
 */
struct ipc_object {
    ipc_object_bits_t io_bits;
    ipc_object_refs_t io_references;
    lck_spin_t  io_lock_data;
};
```

Ports as Handles to Kernel Objects / Data Structures

- **io_bits** field filled with **kobject type**
- **receiver** field points to **ipc_space_kernel**
- **kobject** field points to kernel data structure



Possible Kobject Types

IPC Kobject types are defined in

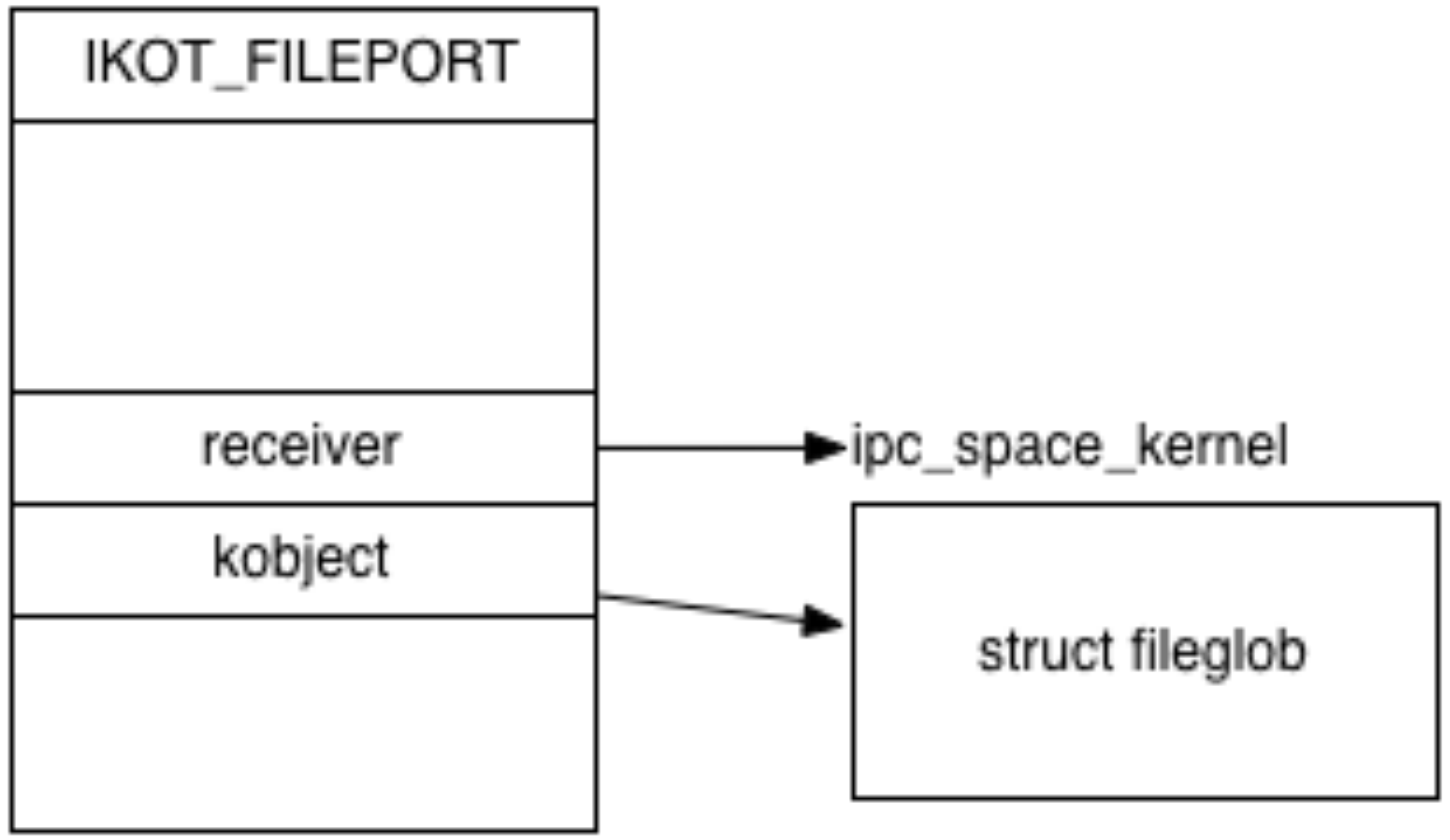
/osfmk/ipc/ipc_kobject.h

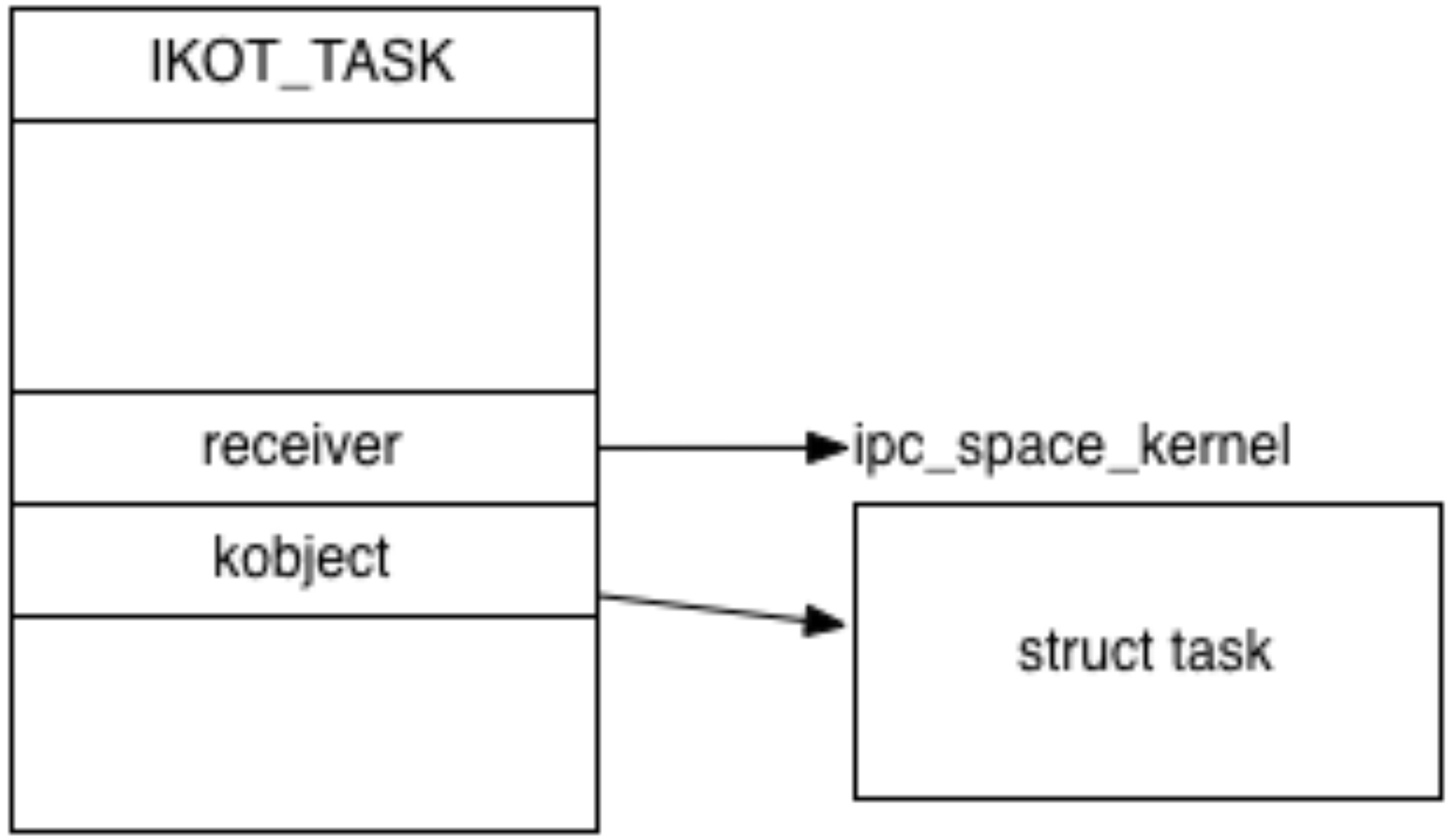
```
#define IKOT_NONE                 0
#define IKOT_THREAD               1
#define IKOT_TASK                 2
#define IKOT_HOST                 3
#define IKOT_HOST_PRIV            4
#define IKOT_PROCESSOR            5
#define IKOT_PSET                 6
#define IKOT_PSET_NAME            7
#define IKOT_TIMER                8
#define IKOT_PAGING_REQUEST       9
#define IKOT_MIG                  10
#define IKOT_MEMORY_OBJECT        11
#define IKOT_XMM_PAGER            12
#define IKOT_XMM_KERNEL           13
#define IKOT_XMM_REPLY            14
#define IKOT_UND_REPLY            15
#define IKOT_HOST_NOTIFY          16
#define IKOT_HOST_SECURITY        17
#define IKOT_LEDGER               18
#define IKOT_MASTER_DEVICE        19
```

```
#define IKOT_TASK_NAME            20
#define IKOT_SUBSYSTEM            21
#define IKOT_IO_DONE_QUEUE        22
#define IKOT_SEMAPHORE            23
#define IKOT_LOCK_SET             24
#define IKOT_CLOCK                25
#define IKOT_CLOCK_CTRL           26
#define IKOT_IOKIT_SPARE          27
#define IKOT_NAMED_ENTRY          28
#define IKOT_IOKIT_CONNECT        29
#define IKOT_IOKIT_OBJECT         30
#define IKOT_UPL                  31
#define IKOT_MEM_OBJ_CONTROL      32
#define IKOT_AU_SESSIONPORT       33
#define IKOT_FILEPORT             34
#define IKOT_LABELH               35
#define IKOT_TASK_RESUME          36
#define IKOT_VOUCHER              37
#define IKOT_VOUCHER_ATTR_CONTROL 38
```


Examples

- **kobject** always points to an **IKOT** specified data structure





What are Mach Messages?

What are Mach Messages?

- data structures sent to or received from Mach Ports
  - header with routing information for kernel
  - optionally descriptors for COMPLEX messages
  - data that is only between sender and receiver
- used for IPC and the Mach API
- sent to kernel via mach traps

Simple vs. Complex Messages

- simple messages are just data blobs

```
typedef struct
{
        mach_msg_header_t       header;
        char                    body[];
} mach_msg_simple_t;
```

- complex messages contain descriptors with special meaning for kernel
  - **MACH_MSG_PORT_DESCRIPTOR** - embedding a port in a message
  - **MACH_MSG_OOL_DESCRIPTOR** - attaching OOL data to message
  - **MACH_MSG_OOL_PORTS_DESCRIPTOR** - attaching OOL ports array to message

```
typedef struct
{
        mach_msg_header_t       header;
        mach_msg_body_t         body;
        mach_msg_descriptor_t   desc[x];
        char                    data[];
} mach_msg_complex_t;
```


Mach Message Header

```
typedef struct
{
  mach_msg_bits_t     msgh_bits;
  mach_msg_size_t     msgh_size;
  mach_port_t         msgh_remote_port;
  mach_port_t         msgh_local_port;
  mach_port_name_t    msgh_voucher_port;
  mach_msg_id_t       msgh_id;
} mach_msg_header_t;
```

where to send to

where to get reply from

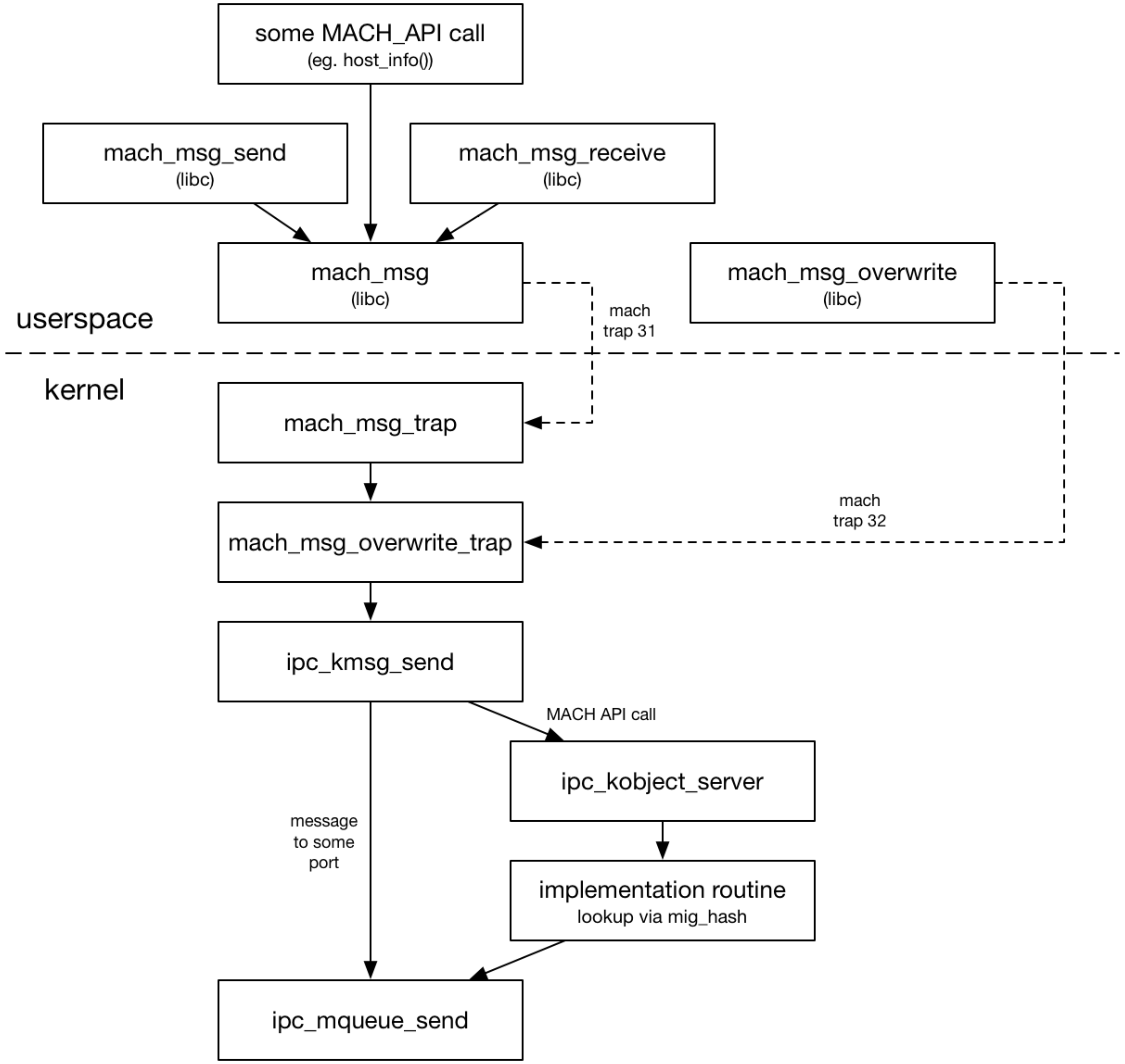
id between sender and receiver

Sending and Receiving Messages

- Mach messages are sent via mach traps

```
mach_msg_return_t
mach_msg(msg, option, send_size, rcv_size, rcv_name, timeout, notify)
    mach_msg_header_t *msg;
    mach_msg_option_t option;
    mach_msg_size_t send_size;
    mach_msg_size_t rcv_size;
    mach_port_t rcv_name;
    mach_msg_timeout_t timeout;
    mach_port_t notify;

mach_msg_return_t
mach_msg_overwrite(msg, option, send_size, rcv_limit, rcv_name, timeout,
           notify, rcv_msg, rcv_scatter_size)
    mach_msg_header_t *msg;
    mach_msg_option_t option;
    mach_msg_size_t send_size;
    mach_msg_size_t rcv_limit;
    mach_port_t rcv_name;
    mach_msg_timeout_t timeout;
    mach_port_t notify;
    mach_msg_header_t *rcv_msg;
    mach_msg_size_t rcv_scatter_size;
```

Sending Mach Messages (Function Overview)



What is the Mach API?

- programming interface offering huge number of functions
- internally converts C style function calls into messages
- first parameter is always the kernel object port to send message to
- usually they manipulate the objects behind the kernel object ports

- special code path detects if **receiver=ipc_space_kernel**
- header’s id field selects what API is called

Mach API Example: vm_write()

- C level call to **vm_write()** automatically converted into Mach message
  - **target_task** set as remote port
  - **id** set to 3807

```
kern_return_t vm_write
      (vm_task_t                target_task,
       vm_address_t             address,
       pointer_t                data,
       mach_msg_type_number_t   data_count);
```

```
typedef struct {
    mach_msg_header_t Head;
    /* start of the kernel processed data */
    mach_msg_body_t msgh_body;
    mach_msg_ool_descriptor_t data;
    /* end of the kernel processed data */
    NDR_record_t NDR;
    vm_address_t address;
    mach_msg_type_number_t dataCnt;
} __Request__vm_write_t;
```


Heap-Feng-Shui for Ports?

Ports as Target

- **kernel object ports** point to kernel data structures
- overwriting/replacing them would allow calling APIs on fake data structures
- wide variety of **IKOT** types means many types to choose from
  - **IKOT_FILEPORT** - fileglob structure has function pointer list
  - **IKOT_IOKIT_CONNECT** - C++ object with vtable pointer
  - ...

Filling the Heap with Ports?

- so should we create a lot of ports to fill the heap?
- would be possible but ports are stored in their own memory zone
- memory corruptions usually involve other memory zones
- cross zone attacks are possible but not KISS

➡ let's add a level of indirection

Filling the Heap with Pointers to Ports?

- instead of filling the heap with **ipc_port_t** structures fill it with pointers
- overwriting a pointer to an **ipc_port_t** still allows to create a fake port
- idea is that pointers are likely allocated in same memory zones as buffers
- when in same memory zone exploitation gets a lot easier

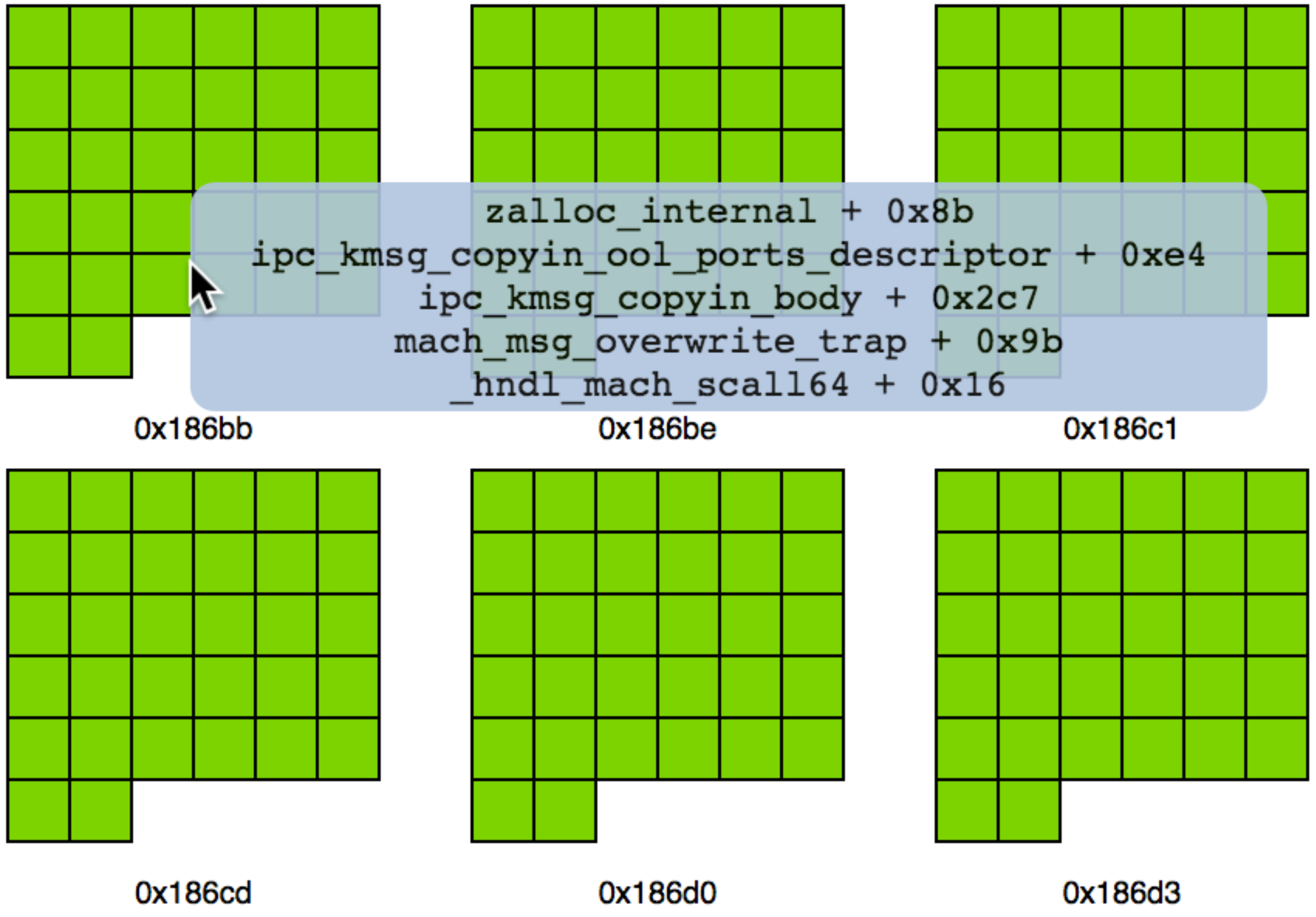
How to fill the memory with Port pointers?

- we can fill the memory with pointers to ports by Mach messages
- we use **MACH_MSG_OOL_PORTS_DESCRIPTOR** for this
- kernel will allocate memory via **kalloc()** to store pointers in memory
- arbitrary sized allocations by sending right amount of ports

```
/* calculate length of data in bytes, rounding up */
ports_length = count * sizeof(mach_port_t);
names_length = count * sizeof(mach_port_name_t);

if (ports_length == 0) {
    return user_dsc;
}

data = kalloc(ports_length);
```


How to fill the memory with Port pointers? (II)

- sending enough messages will fill up the heap pretty quickly
- we can send **MACH_PORT_NULL** or **MACH_PORT_DEAD**



Poking holes...

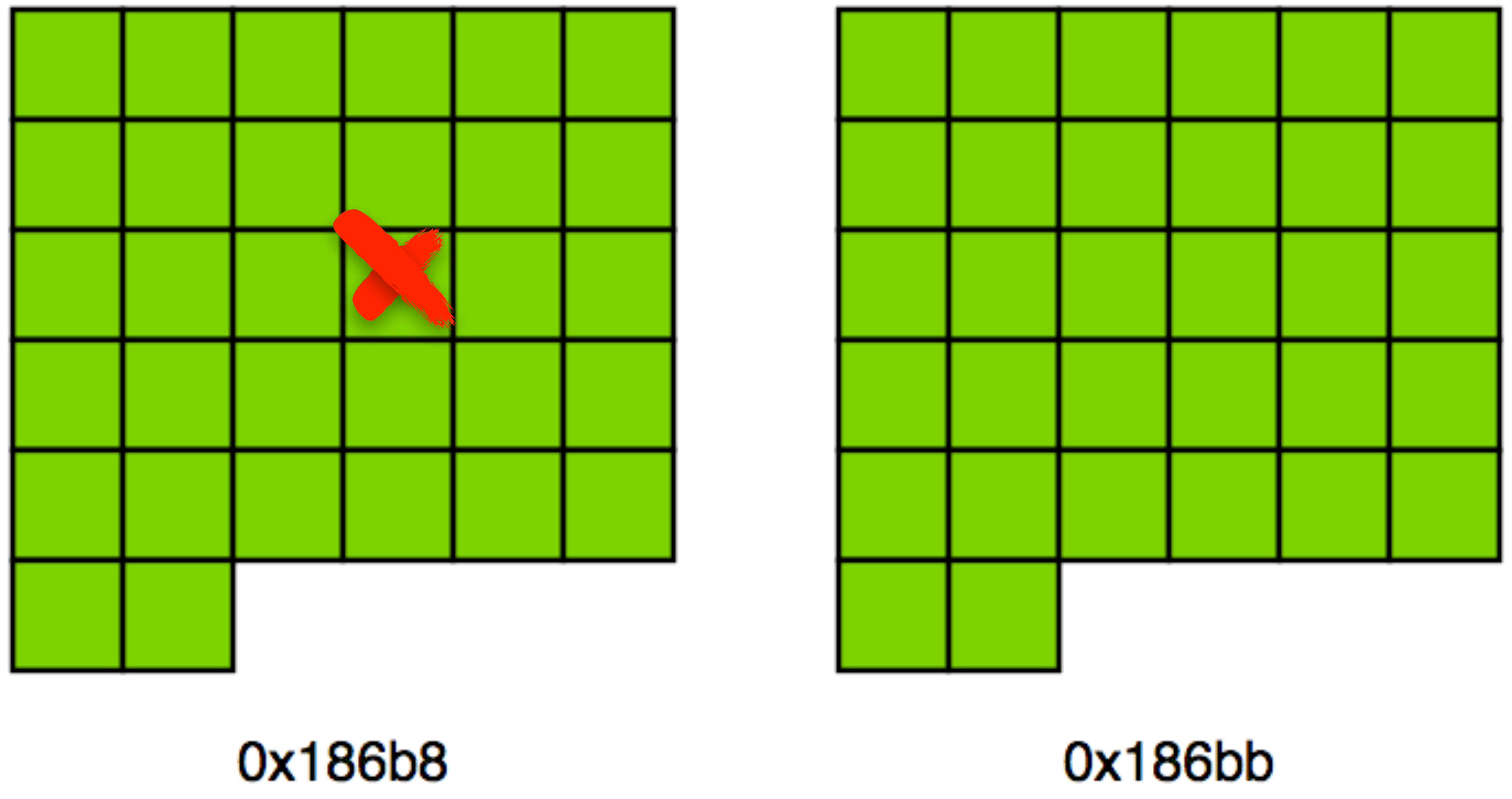
- poking holes in the allocation is done by receiving selected messages
- kernel code will free the previously allocated memory
- deallocation is fine grained because we select what messages to receive
- keep in mind the heap randomization since iOS 9.2

```
/* copyout to memory allocated above */
void *data = dsc->address;
if (copyoutmap(map, data, rcv_addr, names_length) != KERN_SUCCESS)
   *mr |= MACH_MSG_VM_SPACE;
kfree(data, ports_length);
```


Corrupting Port Pointers

- when messages are received all ports within are registered in IPC space
- corrupting any of the allocated pointer lists allows injecting a fake port
- user space can access the fake port

```
/* copyout port rights carried in the message */

for ( i = 0; i < count ; i++) {
    ipc_object_t object = (ipc_object_t)objects[i];

    *mr |= ipc_kmsg_copyout_object(space, object,
               disp, &names[i]);
}
```



Faking Ports

How to fake a port? (I)

- fake pointer must point to something that looks like a port
- we need to setup a number of fields for our port to work
  - **io_bits** - select one of the possible types and make it active
  - **io_references** - better give it some references
  - **io_lock_data** - must be valid lock data
  - **kobject** - pointer to a fake data structure
  - **receiver** - we cannot fill out because we don’t know **ipc_space_kernel**

How to fake a port? (II)

- fake port and fake data must be in attacker controlled memory
- it is required to know address of that memory
- easy to do for 64 bit devices (except iPhone 7) because of user land dereferences
- requires additional information leaks for iPhone 7 and 32 bit devices (unless already privileged outside the sandbox)

What can we do with such a faked port?

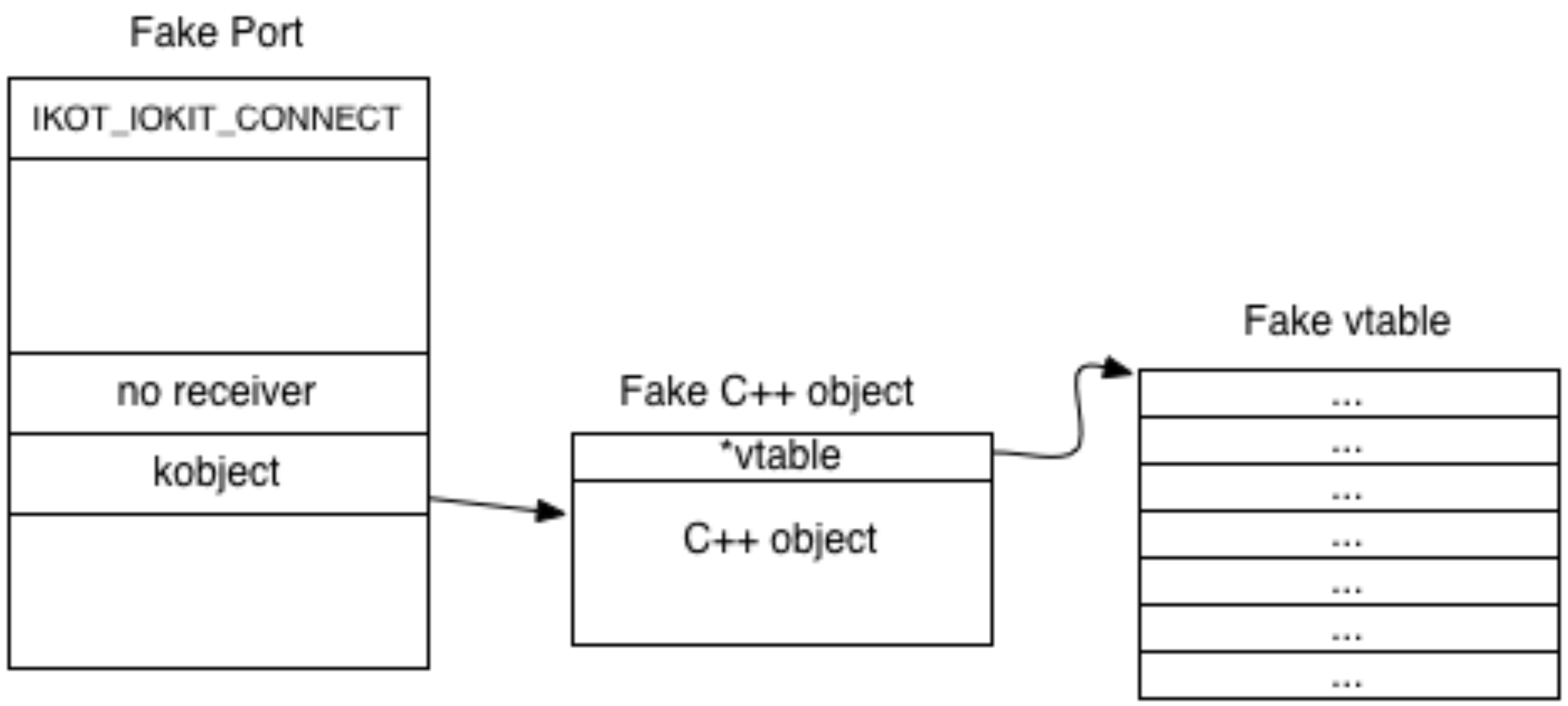
- because we cannot fill in **receiver** not a fully usable port
- it works fine when used as argument to
  - syscalls
  - mach traps
  - as additional parameter Mach API (not 1st argument)
- but it will NOT work as first argument to a MachAPI
  (for this we need the receiver to be ipc_space_kernel)

Some Examples

- some examples of ports we can fake
  - **IKOT_IOKIT_CONNECT** - driver connection to a IOUserClient derived object
  - **IKOT_CLOCK** - clock object
  - **IKOT_TASK** - task object

Faking IOKit Driver Connection Ports

- ports of type **IKOT_IOKIT_CONNECT** can be used via **iokit_user_client_trap()**
- **kobject** pointer points to a C++ object
- good target because it allows control of the **method table**
- see “HITB2013 - Tales from iOS 6 Exploitation” for example



Faking Clock Ports (I)

- ports of type **IKOT_CLOCK** can be used via **clock_sleep_trap()**
- **kobject** pointer points to a struct clock
- looks like a good target because there is a function pointer list

```
/*
 * Actual clock object data structure. Contains the machine
 * dependent operations list and clock operation ports.
 */
struct  clock {
    clock_ops_t         cl_ops;         /* operations list */
    struct ipc_port     *cl_service;    /* service port */
    struct ipc_port     *cl_control;    /* control port */
};
```

pointer to list of
function pointers

Faking Clock Ports (II)

- code of **clock_sleep_internal()** will not allow a fake **clock struct**
- only the valid **SYSTEM_CLOCK** pointer is accepted
- otherwise function errors out - triggering code execution not possible

```
if (clock == CLOCK_NULL)
    return (KERN_INVALID_ARGUMENT);

if (clock != &clock_list[SYSTEM_CLOCK])
    return (KERN_FAILURE);

/*
 * Check sleep parameters. If parameters are invalid
 * return an error, otherwise post alarm request.
 */
(*clock->cl_ops->c_gettime)(&clock_time);

chkstat = check_time(sleep_type, sleep_time, &clock_time);
if (chkstat < 0)
    return (KERN_INVALID_VALUE);
```

validation against our
fake clock struct pointer

Faking Clock Ports (III)

- wait a second!
- a wrong clock pointer will lead to **KERN_FAILURE**
- a good pointer with bad other arguments leads to **KERN_INVALID_VALUE**

```
if (clock == CLOCK_NULL)
    return (KERN_INVALID_ARGUMENT);

if (clock != &clock_list[SYSTEM_CLOCK])
    return (KERN_FAILURE);

/*
 * Check sleep parameters. If parameters are invalid
 * return an error, otherwise post alarm request.
 */
(*clock->cl_ops->c_gettime)(&clock_time);

chkstat = check_time(sleep_type, sleep_time, &clock_time);
if (chkstat < 0)
    return (KERN_INVALID_VALUE);
```

error if our pointer is not pointing to the SYSTEM_CLOCK

error if our pointer was okay but other arguments bad

Faking Clock Ports (IV)

- if we can change **kobject** we can bruteforce the **SYSTEM_CLOCK** address
- userland dereference makes this easy on 64 bit pre iPhone 7
- this reveals pointer inside kernel image and therefore breaks KASLR

```
our_fake_port->io_bits = IKOT_CLOCK | IO_BITS_ACTIVE;
our_fake_port->kobject = low_kernel_address;

while (1) {
    our_fake_port->kobject+= 8;
    kret = clock_sleep_trap(magicport, 0x12345, 0, 0, NULL);

    if (kret != KERN_FAILURE) {
        break;
    }
}
```


Faking Task Ports (I)

- ports of type **IKOT_TASK** have **kobject** pointer pointing to a **task struct**
- unfortunately cannot be used directly in task Mach API functions
- but there are other usages like

pid_for_task() - return the pid for a given task

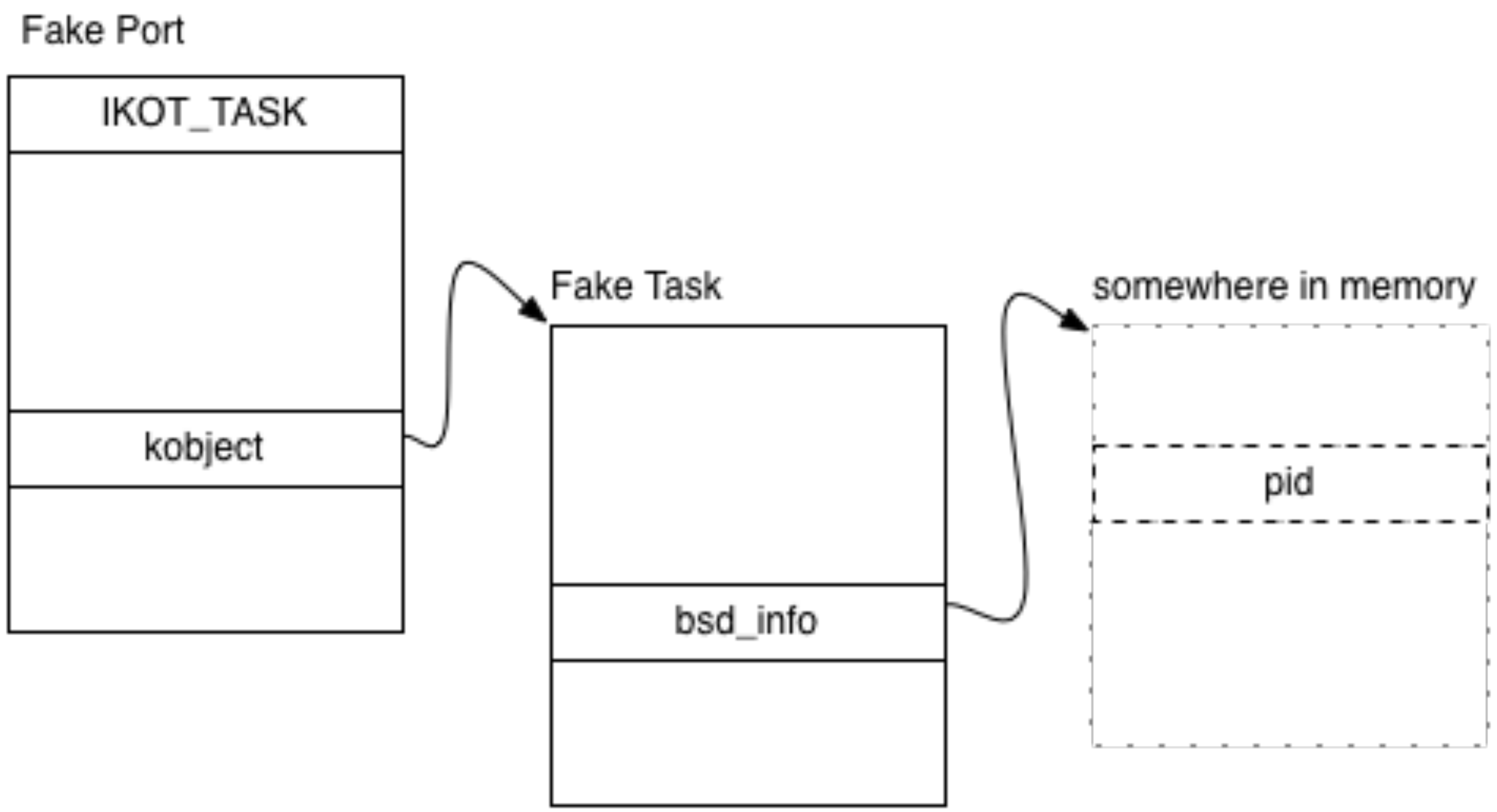
```
t1 = port_name_to_task(t);

if (t1 == TASK_NULL) {
   err = KERN_FAILURE;
   goto pftout;
} else {
   p = get_bsdtask_info(t1);
   if (p) {
      pid  = proc_pid(p);
```

reads pointer to struct proc
from our fake struct task

returns pid from
without struct proc

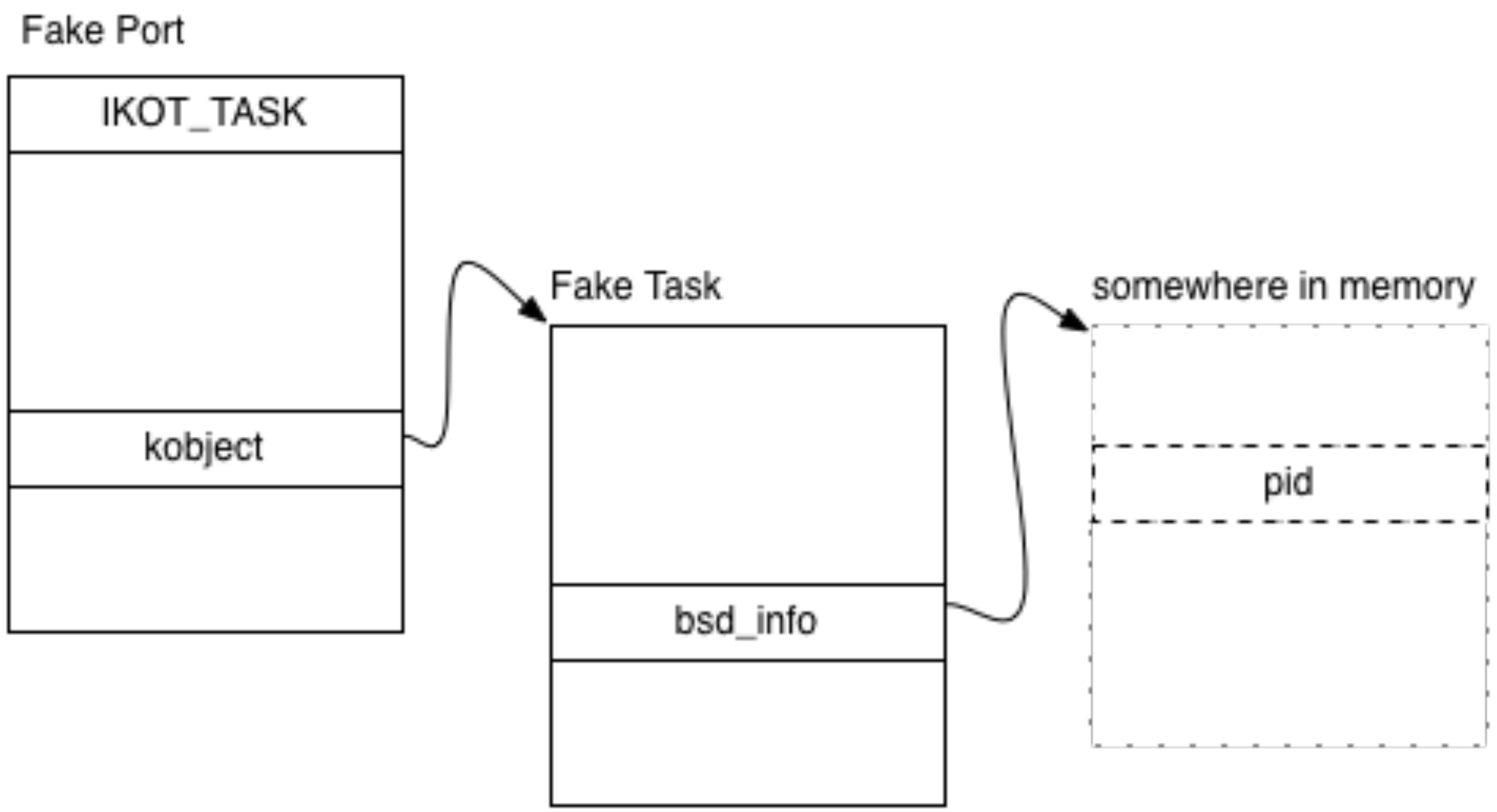
Faking Task Ports (II)

- our fake **IKOT_TASK** port points to a fake **task struct**
- the **bsd_info** fields points anywhere in memory
- **pid_for_task()** will read back at offset 0x10
- allows to read from anywhere in kernel memory



Faking Task Ports (III)

- if we can change **kobject** we can read everything
- userland dereference makes this easy on 64 bit pre iPhone 7
- this allows to read important variables like **ipc_space_kernel**
- this means afterwards we can use Mach API with our fake port



Kernel Task Port

Our Port(al) to the Core

Kernel Task Port

- among all the ports in an iOS system the **kernel task port** is the holy grail
- with access to the **kernel task port** we can manipulate the kernel memory
  - **vm_read** - allows reading kernel memory
  - **vm_write** - allows writing kernel memory
  - **vm_allocate** - allows allocating memory inside the kernel address space
  - …
- whoever has access to the kernel task port more or less controls the system
- to turn out fake task port into a kernel task port we need to know **kernel_task** and **ipc_space** kernel

Attack Plan for 64 bit devices (except iPhone 7)

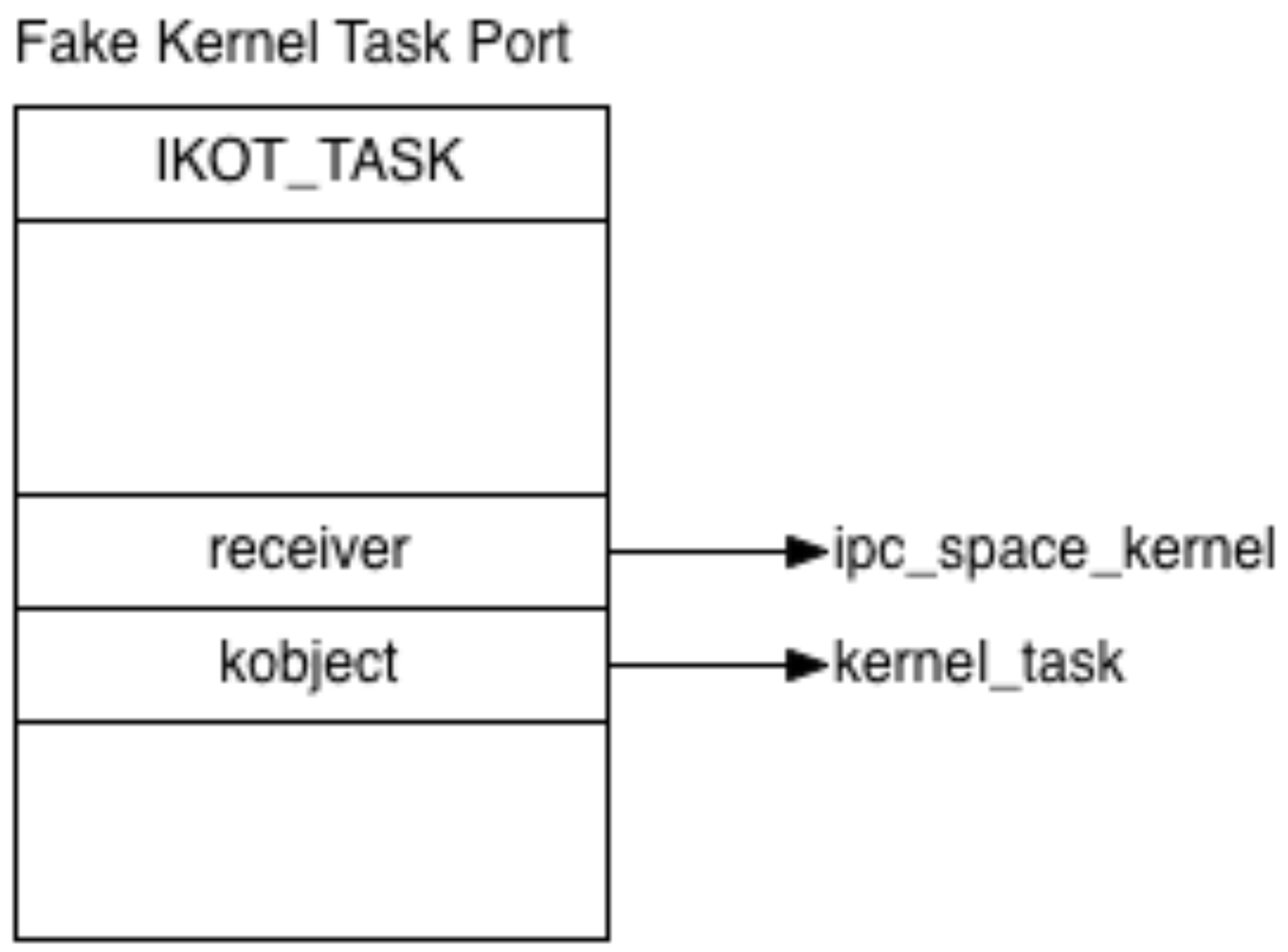
- **Corruption Phase**
  - perform heap feng shui with OOL_PORTS_DESCRIPTORS
  - corrupt any of the “sprayed” port pointers
  - receive all messages to get access to port

- **Post Corruption Phase**
  - fake a CLOCK port to break KASLR - via bruteforce of clock address
  - fake a TASK port and TASK struct to have arbitrary kernel read
  - read ipc_space_kernel and kernel_task
  - fake a kernel TASK port

Faking the KERNEL TASK Port (I)

- with **ipc_space_kernel** our fake ports can be used in Mach API
- with **kernel_task** we can fake a kernel task port
- mach API gives us read/write access to kernel memory

- **Game Over!**



Conclusion

Conclusion

- overwriting port pointers
  - allows to gain code execution
  - or full read write access to kernel memory

- heap feng shui with mach messages and **OOL_PORTS_DESCRIPTOR**
  - gives fine grained control over heap
  - fills heap with port pointers that when corrupted

- post corruption code is fully reusable for different corruptions (64bit before i7)

Questions ?

www.antid0te.com
stefan@antid0te.com