

Attacking the XNU Kernel in El Capitan

Luca Todesco (@qwertyoruiop)

<me@qwertyoruiop.com>

BlackHat EU 2015

About Me

- Independent vulnerability researcher from Venice, Italy
- Focusing on Apple's products, particularly attracted by jailbreaking techniques
- Author of several XNU Kernel-related CVEs and exploits
 - “vpwn” (< 10.10.2 LPE) - CVE-2015-1140 / CVE-2015-5865
 - “tpwn” (< 10.11 LPE) - CVE-2015-5932 / CVE-2015-5847 / CVE-2015-5864
 - “npwn” (10.11 SIP bypass) - CVE-2015-6974

Why attack XNU?

- XNU has been a target primarily for iOS jailbreaking
- Yosemite enforces KEXT signatures
- El Capitan introduces “System Integrity Protection”
 - System-wide, kernel-enforced sandbox profile that prevents access to system resources
- Attacking the kernel is a viable way to bypass rootless and sandbox

the xnu heap

A quick overview

The XNU Heap: Zone Allocator (zalloc)

- `zinit(...)` / `zalloc(zone)` / `zfree(zone, ptr)`
- Each zone has a LIFO linked list containing free chunks
- Allocations in a zone are same-sized
- When allocating from a zone without free chunks, a new page is mapped in, page is split in chunks and each chunk is added to the free list
- Discussed in detail in countless talks by Stefan Esser

The XNU Heap: Zone Allocator (zalloc)

- No inline metadata for allocated chunks, free list metadata on free chunks
- Free list metadata is not an interesting target due to hardening
- Application metadata is the only target
- Different zones use different areas of memory, so cross-zone attacks aren't feasible
- This does not apply to large allocations

The XNU Heap: Zone Allocator (kalloc)

- `kalloc(size)`, `kfree(ptr, size)`
- Wrapper around `zalloc`
- Registers several generic zones with various sizes
- Essentially provides a `malloc`-like interface, but lack of metadata in allocated chunks requires passing “size” to `kfree`

The XNU Heap: Zone Allocator (kalloc)

zone name	elem size	cur size	max size	cur #elts	max #elts	cur inuse	alloc size	alloc count	
kalloc.16	16	1664K	1751K	106496	112100	95001	4K	256	C
kalloc.32	32	2272K	2627K	72704	84075	58856	4K	128	C
kalloc.48	48	4256K	5911K	90794	126113	83520	4K	85	C
kalloc.64	64	9172K	13301K	146752	212816	87246	4K	64	C
kalloc.80	80	20672K	29927K	264601	383068	255865	4K	51	C
kalloc.96	96	1736K	2335K	18517	24911	13912	8K	85	C
kalloc.128	128	7672K	8867K	61376	70938	59846	4K	32	C
kalloc.160	160	1552K	1556K	9932	9964	9123	8K	51	C
kalloc.256	256	23680K	29927K	94720	119709	91884	4K	16	C
kalloc.288	288	2300K	2594K	8177	9226	8068	20K	71	C
kalloc.512	512	52740K	101004K	105480	202009	99398	4K	8	C
kalloc.1024	1024	24132K	29927K	24132	29927	22996	4K	4	C
kalloc.1280	1280	768K	768K	614	615	475	20K	16	C
kalloc.2048	2048	9572K	19951K	4786	9975	4181	4K	2	C
kalloc.4096	4096	5052K	13301K	1263	3325	1261	4K	1	C
kalloc.8192	8192	6432K	7882K	804	985	799	8K	1	C

kalloc zones on 10.11

(output of “zprint kalloc” as root)

(for some reason “zprint kalloc” segfaults in 10.11, but “zprint | grep kalloc” works)

vm_map_copy corruption

A quick overview of 10.10 techniques

The XNU Heap: `vm_map_copy` in 10.10

- Introduced as an easy way to do data-only memory leaks by Tarjei Mandt and Mark Dowd's HITB2012KUL "iOS 6 Security" presentation
- `vm_map_copy` is a structure used to hold a copy of some data
- For small amounts of data the kernel heap is used
- Targeted by an endless amount of kernel exploits

The XNU Heap: `vm_map_copy` in 10.10

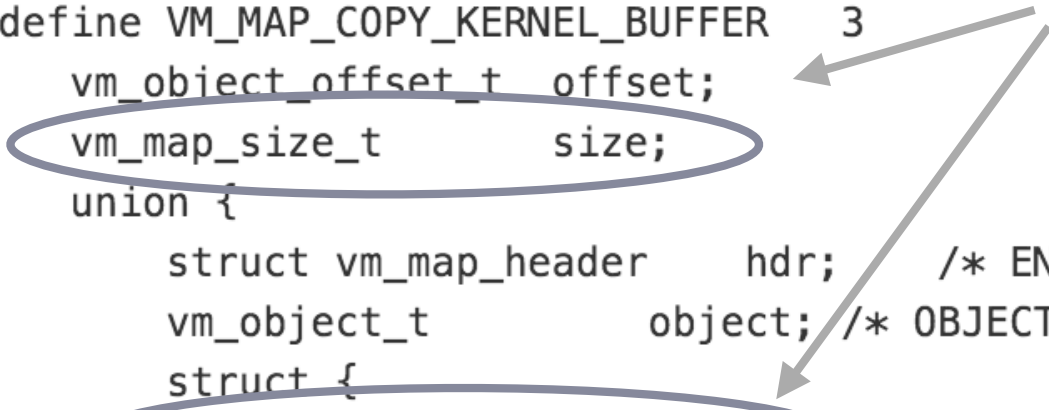
- Allocated with `kalloc(sizeof(struct vm_map_copy) + data_size)`
 - **Controlled size!**
- Can be created and accessed easily via OOL `mach_msg` data
- Completely unaffected by sandboxing

The XNU Heap: vm_map_copy in 10.10

10.10 source:

```
struct vm_map_copy {
    int          type;
#define VM_MAP_COPY_ENTRY_LIST      1
#define VM_MAP_COPY_OBJECT        2
#define VM_MAP_COPY_KERNEL_BUFFER  3
    vm_object_offset_t  offset;
    vm_map_size_t      size;
    union {
        struct vm_map_header  hdr;      /* ENTRY_LIST */
        vm_object_t          object; /* OBJECT */
        struct {
            void              *kdata; /* KERNEL_BUFFER */
            vm_size_t        kalloc_size; /* size of this copy_t */
        } c_k;
    } c_u;
};
```

Usual info-leak targets



x86_64 sizeof(struct vm_map_copy) = 0x58

tpwn: a 10.10 kernel exploit

tpwn: a 10.10 kernel exploit

- Released in Aug 2015
- 0-day at the time
 - CVE-2015-5932 / CVE-2015-5847 / CVE-2015-5864
- Core issue is a type confusion in handling mach ports in `io_service_open_extended`
- Ports passed as “task” with a non-`IKOT_TASK` type would cause `NULL` to be passed as pointer to task struct to `IOUserClients` (CVE-2015-5932)

tpwn: __PAGEZERO strikes again

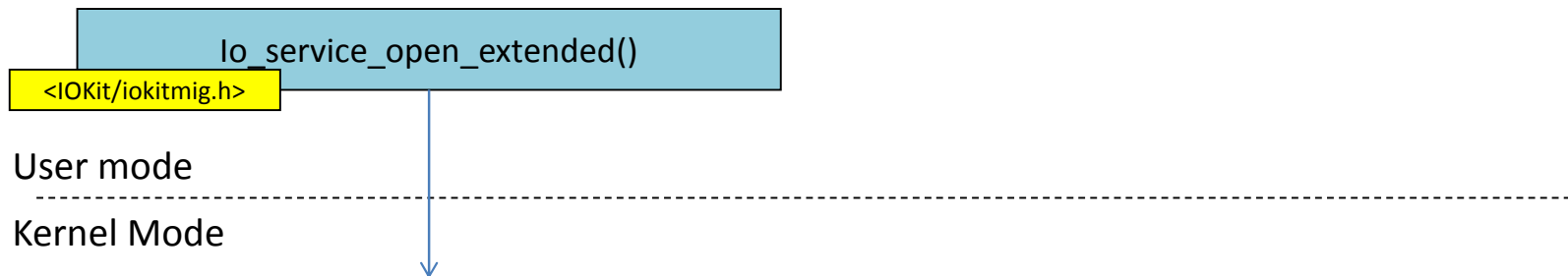
- The Mach-O format defines __PAGEZERO as a guard area
 - 32-bit: 4K, used to trap NULL pointer dereferences
- Apple enforces “hard page zero” to prevent mapping NULL
- But

```
#if __x86_64__  
    /*  
     * On x86, for compatibility, don't enforce the hard page-zero restriction for 32-bit binaries.  
     */  
    if ((imgp->ip_flags & IMGPF_IS_64BIT) == 0) {  
        enforce_hard_pagezero = FALSE;  
    }  
#endif
```

Page zero is left wide open in 32-bit binaries!

tpwn: a 10.10 kernel exploit

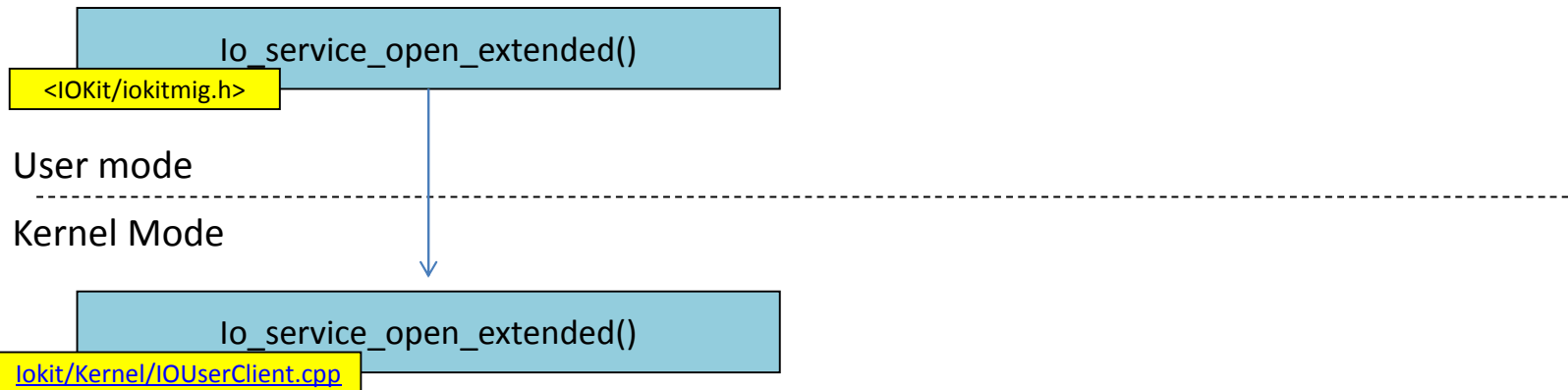
(service, owningTask, connect_type, ndr, properties, propertiesCnt, *result, *connection)



`io_service_open_extended` is one of several undocumented MIG functions to communicate with IOKit drivers from user mode

tpwn: a 10.10 kernel exploit

(service, owningTask, connect_type, ndr, properties, propertiesCnt, *result, *connection)



```
/* Routine io_service_open_ndr */
kern_return_t is_io_service_open_extended(
    io_object_t _service,
    task_t owningTask,
    uint32_t connect_type,
    NDR_record_t ndr,
    io_buf_ptr_t properties,
    mach_msg_type_number_t propertiesCnt,
    kern_return_t * result,
    io_object_t *connection )
{
    IOUserClient * client = 0;
    kern_return_t err = KERN_SUCCESS;
    IOReturn res = kIOReturnSuccess;
    OSDictionary * propertiesDict = 0;
    bool crossEndian;
    bool disallowAccess;

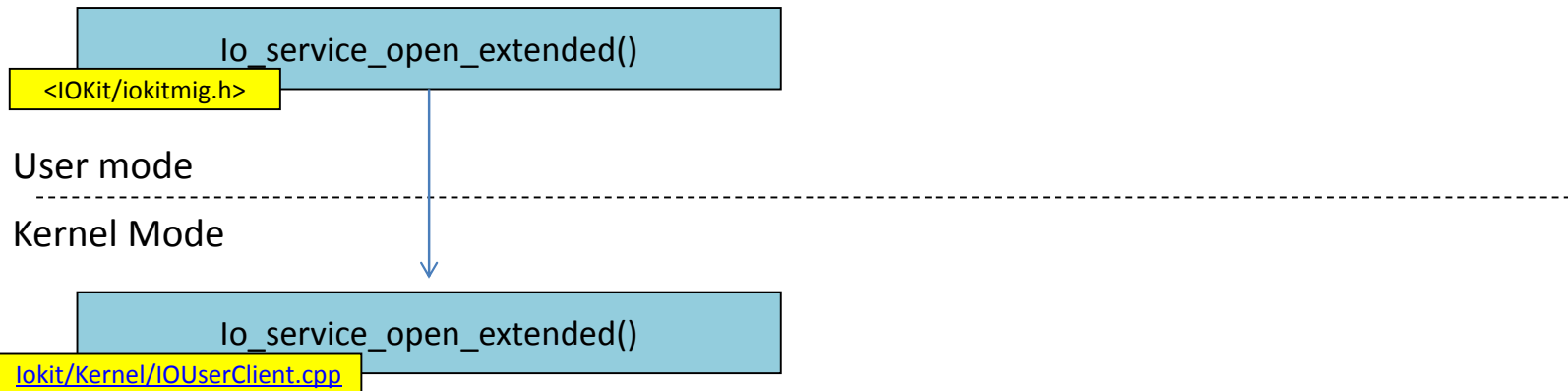
    CHECK( IOService, _service, service );

    do
    {
        if (properties)
        {
            ...
        }
    }
    ...
}
```

Note NO CHECK ON owningTask!

tpwn: a 10.10 kernel exploit

(service, owningTask, connect_type, ndr, properties, propertiesCnt, *result, *connection)



```
/* Routine io_service_open_ndr */
kern_return_t is_io_service_open_extended(
    io_object_t _service,
    task_t owningTask,
    uint32_t connect_type,
    NDR_record_t ndr,
    io_buf_ptr_t properties,
    mach_msg_type_number_t propertiesCnt,
    kern_return_t * result,
    io_object_t *connection )
{
    IOUserClient * client = 0;
    kern_return_t err = KERN_SUCCESS;
    IOReturn res = kIOReturnSuccess;
    OSDictionary * propertiesDict = 0;
    bool crossEndian;
    bool disallowAccess;

    CHECK( IOService, _service, service );

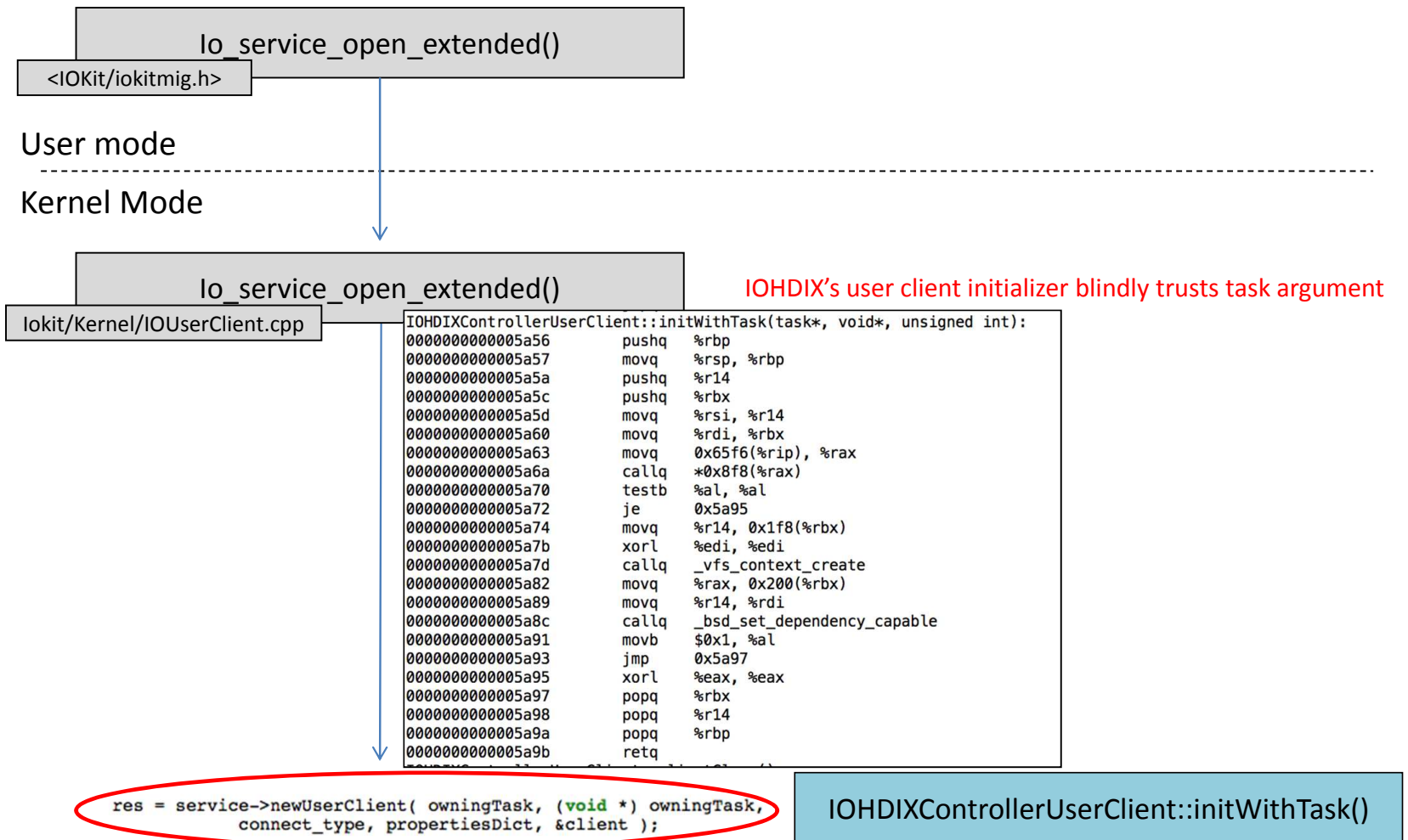
    do
    {
        if (properties)
        {
            . . .

            res = service->newUserClient( owningTask, (void *) owningTask,
                connect_type, propertiesDict, &client );
        }
    }
}
```

← owningTask then gets passed to User Clients

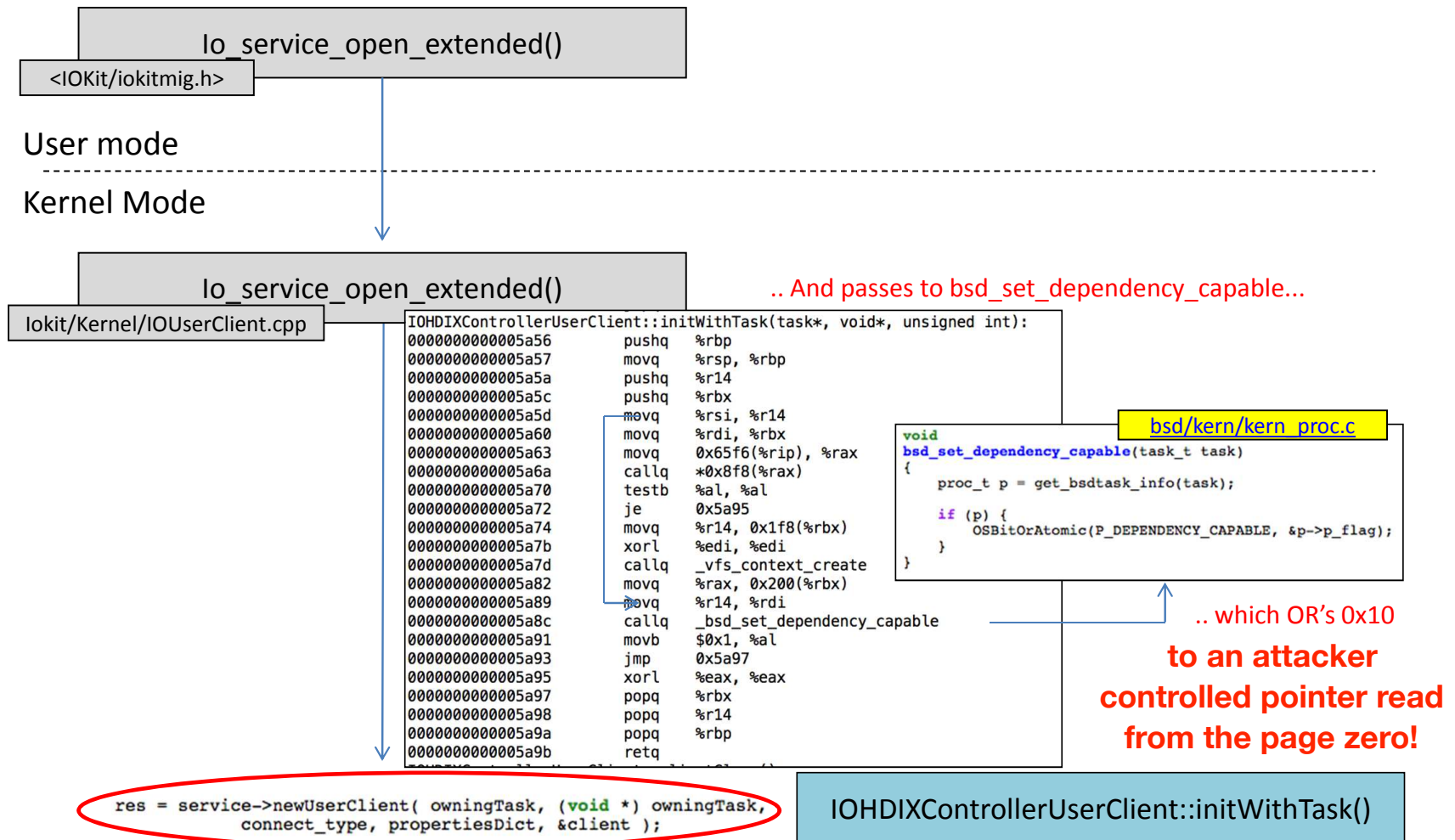
tpwn: a 10.10 kernel exploit

(service, owningTask, connect_type, ndr, properties, propertiesCnt, *result, *connection)



tpwn: a 10.10 kernel exploit

(service, owningTask, connect_type, ndr, properties, propertiesCnt, *result, *connection)



tpwn: a 10.10 kernel exploit

- Using an heap info leak (CVE-2015-5864) we can locate a C++ object in `kalloc.1024`
- We need to locate a `vm_map_copy` and make sure it's adjacent to a C++ object
- Corrupt the size of the `vm_map_copy` to read the C++ object's memory
- Derive kASLR slide from there
- Gain instruction pointer control, pivot the stack

tpwn: controlling the heap layout

1



KALLOC.1024 (FRAGMENTED HEAP)



ALLOCATED



FREE HOLE



IOAudioEngineUserClient



vm_map_copy

tpwn: controlling the heap layout



ALLOCATED



FREE HOLE



IOAudioEngineUserClient



vm_map_copy

tpwn: controlling the heap layout



ALLOCATED



FREE HOLE

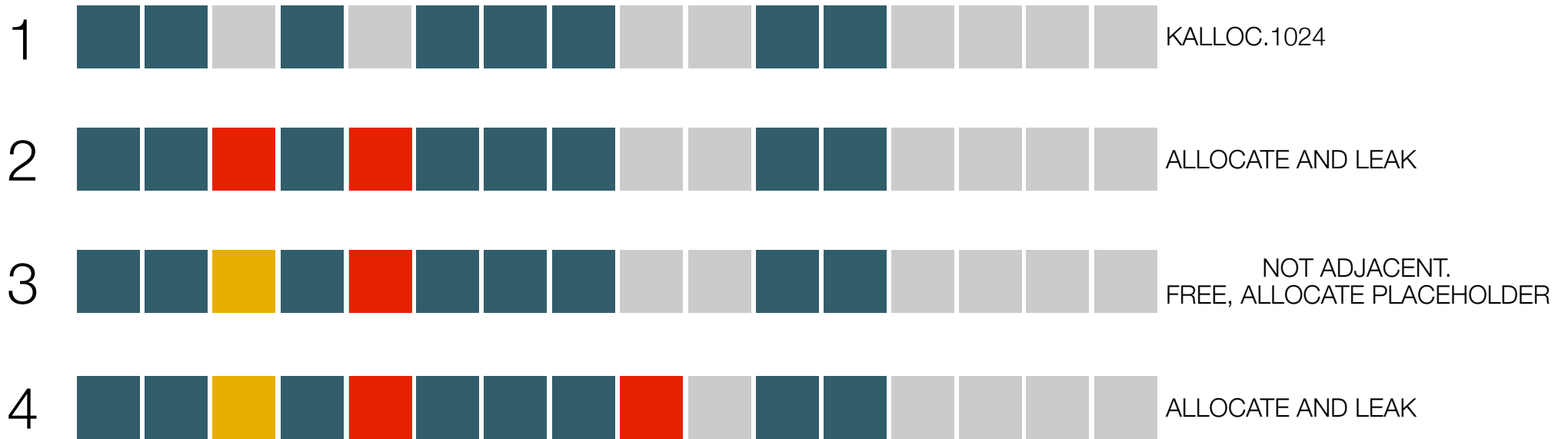


IOAudioEngineUserClient

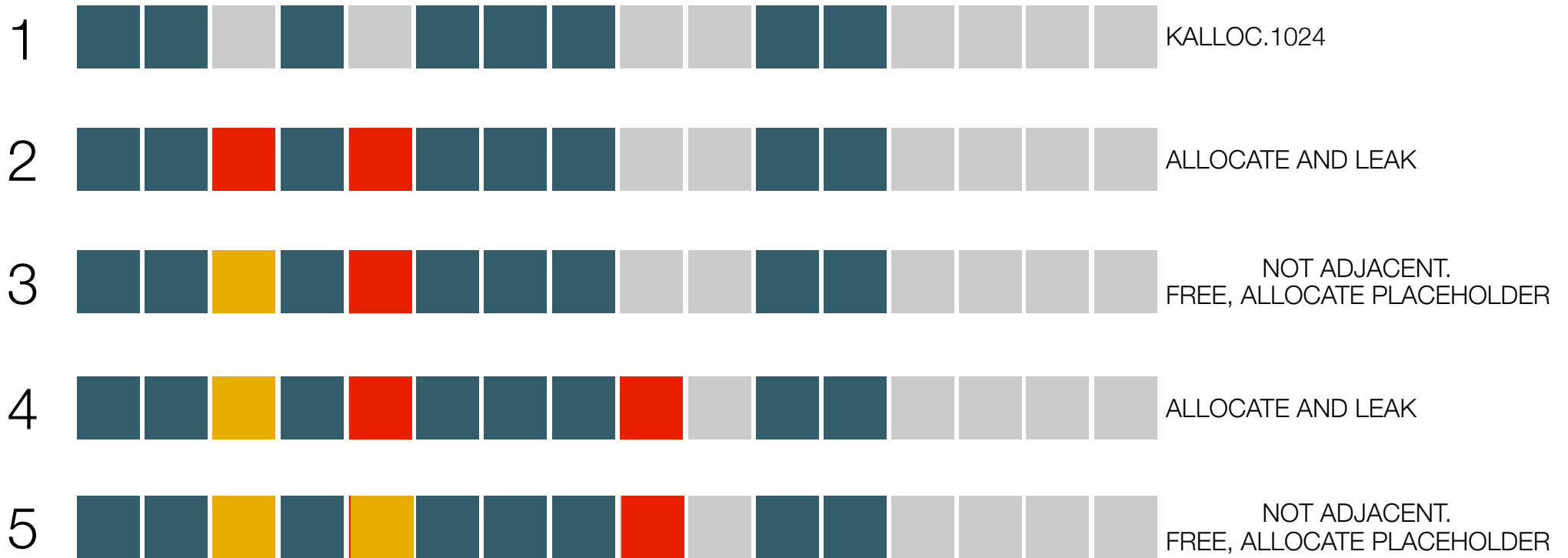


vm_map_copy

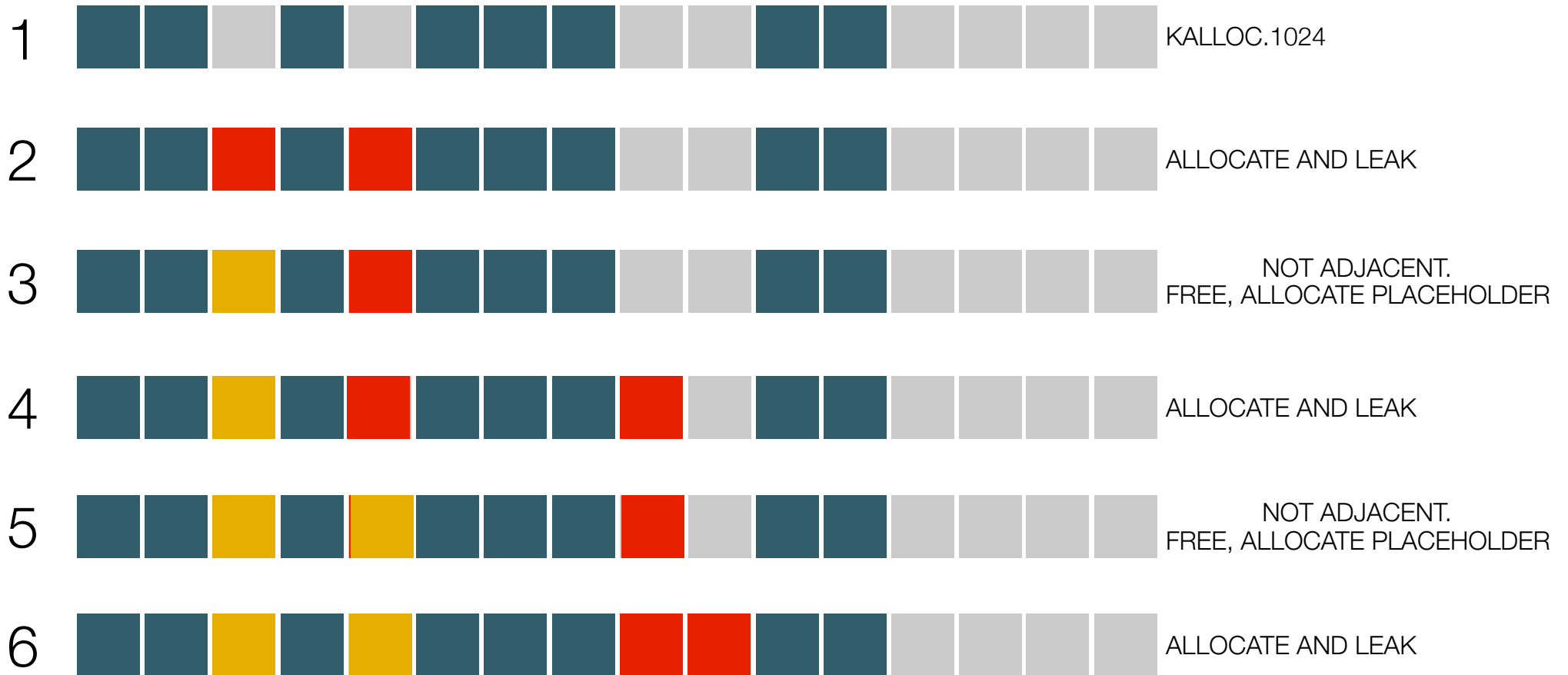
tpwn: controlling the heap layout



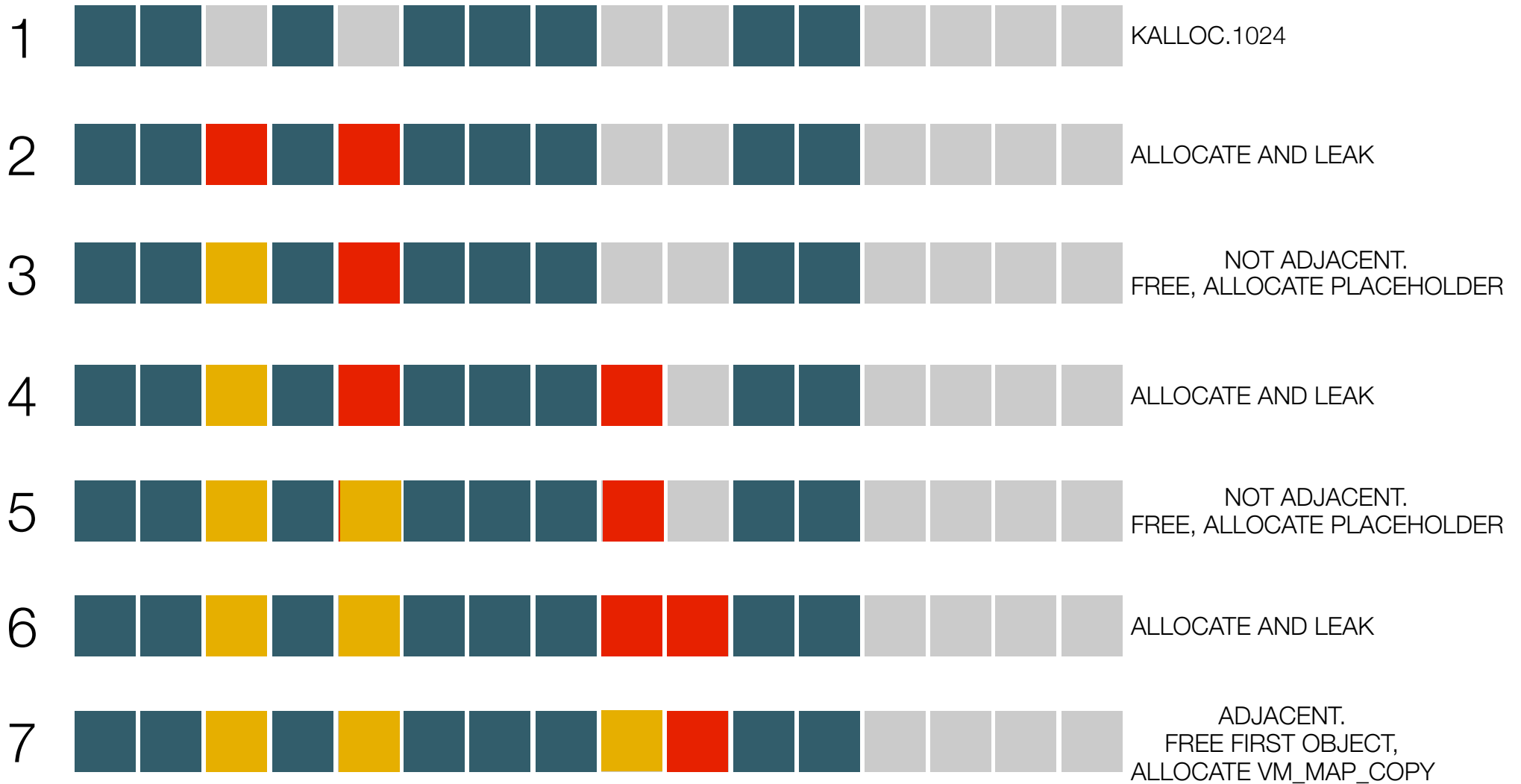
tpwn: controlling the heap layout



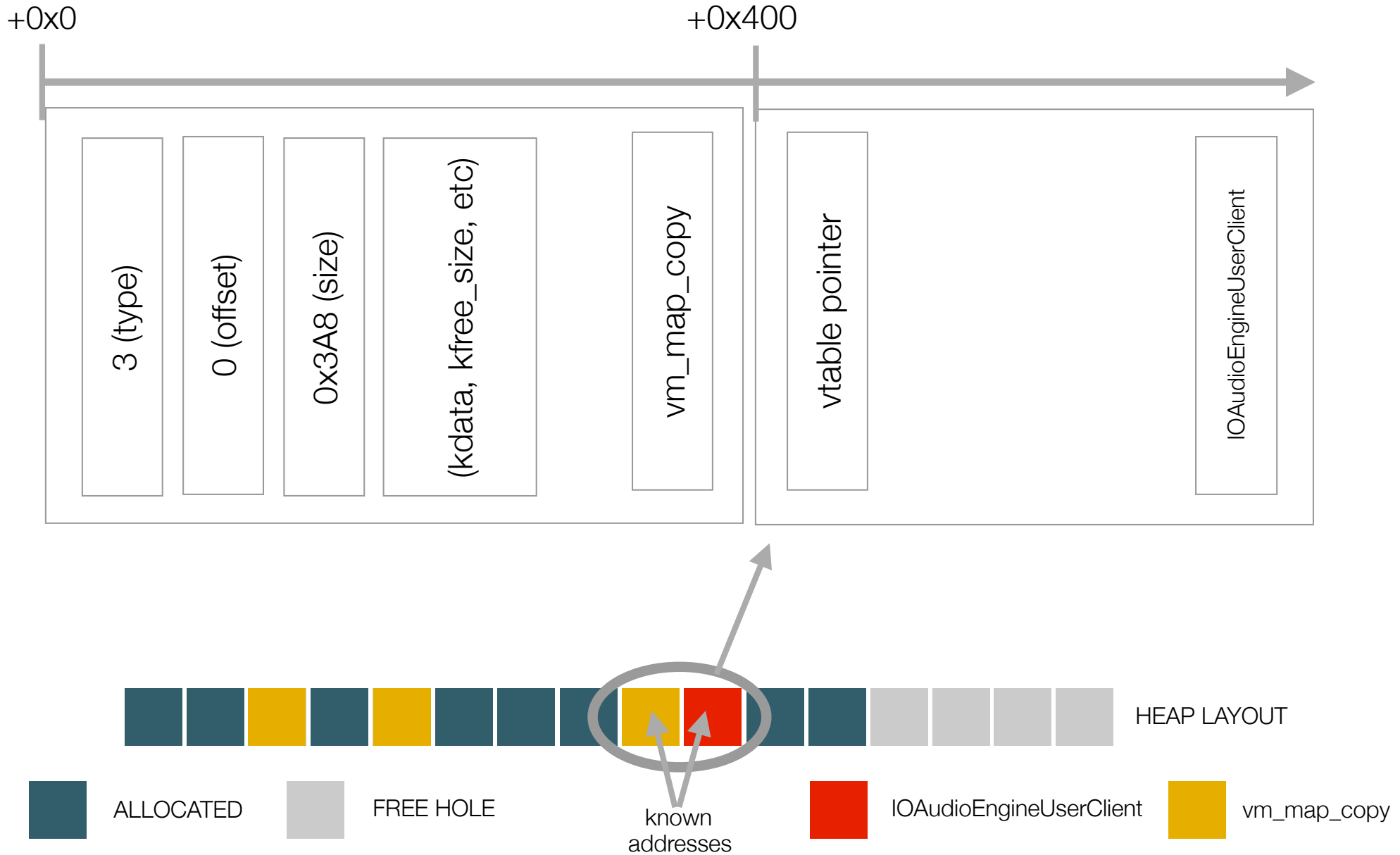
tpwn: controlling the heap layout



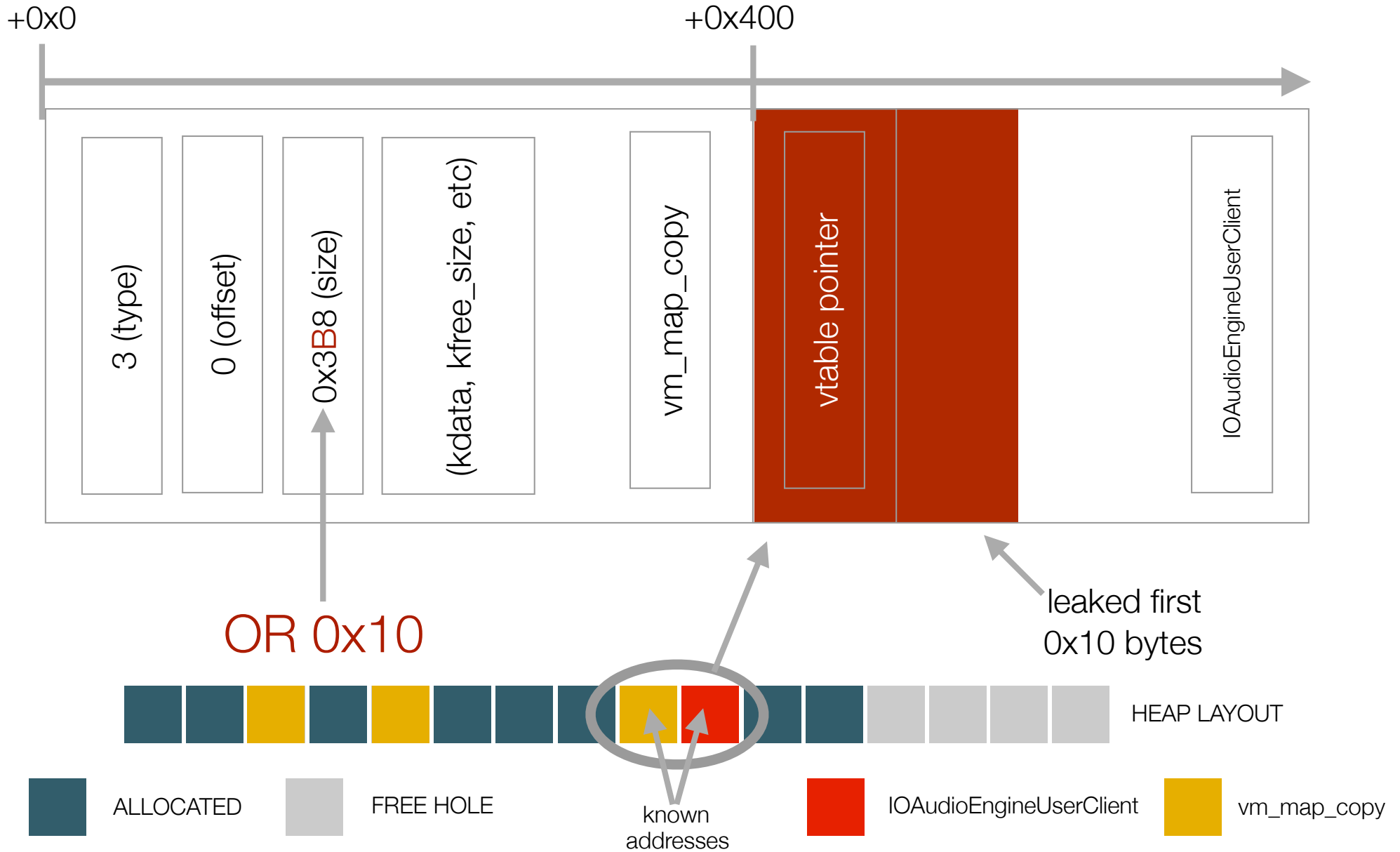
tpwn: controlling the heap layout



tpwn: 10.10 kASLR leaking strategy



tpwn: 10.10 kASLR leaking strategy



tpwn: a 10.10 kernel exploit

- Result:

```
[qwertyoruiop@qwertyoruiops-iMac:~/xnux/x]$ make
gcc *.m -o tpwn -framework IOKit -framework Foundation -m32 -Wl,-pagezero_size,0
-03
strip tpwn
[qwertyoruiop@qwertyoruiops-iMac:~/xnux/x]$ ./tpwn
leaked kaslr slide, @ 0x0000000000000000
sh-3.2# uname -a
Darwin qwertyoruiops-iMac.local 14.4.0 Darwin Kernel Version 14.4.0: Thu May 28
11:35:04 PDT 2015; root:xnu-2782.30.5~1/RELEASE_X86_64 x86_64
sh-3.2#
```

<https://github.com/kpwn/tpwn>

(fairly straightforward code)

vm_map_copy corruption

10.11 Info Leaking Strategies

The XNU Heap: vm_map_copy in 10.11

- Structure has been changed in 10.11
- On x86_64 sizeof(vm_map_copy) is 0x18 now

10.11 debug kernel:

```
struct vm_map_copy
{
    int type;
    vm_object_offset_t offset;
    vm_map_size_t size;
    vm_map_copy::$30C14F0EB10F809AE5F27A96BE564370 c_u;
};

union vm_map_copy::$30C14F0EB10F809AE5F27A96BE564370
{
    vm_map_header hdr;
    vm_object_t object;
    uint8_t_0 kdata[];
};
```

10.10 source:

```
struct vm_map_copy {
    int type;
#define VM_MAP_COPY_ENTRY_LIST 1
#define VM_MAP_COPY_OBJECT 2
#define VM_MAP_COPY_KERNEL_BUFFER 3
    vm_object_offset_t offset;
    vm_map_size_t size;
    union {
        struct vm_map_header hdr; /* ENTRY_LIST */
        vm_object_t object; /* OBJECT */
        struct {
            void *kdata; /* KERNEL_BUFFER */
            vm_size_t kalloc_size; /* size of this copy_t */
        } c_k;
    } c_u;
};
```

The XNU Heap: `vm_map_copy` in 10.11

- Size to `kfree` and data size have been unified
 - Cannot read adjacent memory without corrupting it, since increasing data size past heap allocation boundaries will free into the wrong zone
- Pointer to data has been removed
 - Can't read data pointer off adjacent `vm_map_copy`
 - Can't swap data pointer to leak arbitrary memory
- New techniques are needed

vm_map_copy: Leaking adjacent data in 10.11

- Leaking adjacent bytes can now be done only by first reading and corrupting, then writing back the read data
 - Not as reliable as corrupting data size since it involves a re-allocation

Leaking heap pointers in 10.11

- You can't read the data pointer off a `vm_map_copy` to leak heap pointers since it has been removed from the structure
- Heap address leaks are useful since they allow you to locate controlled data in the kernel heap
- Just use another structure containing heap pointers
- The free list is an easy target

Leaking heap pointers in 10.11

- Allocate two adjacent `vm_map_copy` structures
- Free the second
- Corrupt the first to increase size
- Read the first (leaking adjacent memory)
- Allocate a new `vm_map_copy` with the leaked data
- Allocate two `vm_map_copy` structures in the same zone, second you allocate will be located at the pointer you've leaked off the free list

Leaking arbitrary memory in 10.11

- You can't swap the data pointer off a `vm_map_copy` to get arbitrary memory leaks since it has been removed from the structure
- `OSData` is a kernel C++ object used to represent generic data. On `x86_64` it lives in `kalloc.48`
- Use `io_service_open_extended`'s `OSUnserializeXML` to create `OSData` objects
 - Although dated, the “iOS Kernel Heap Armageddon” talk by Esser explains more about `OSUnserializeXML` and `libkern` objects

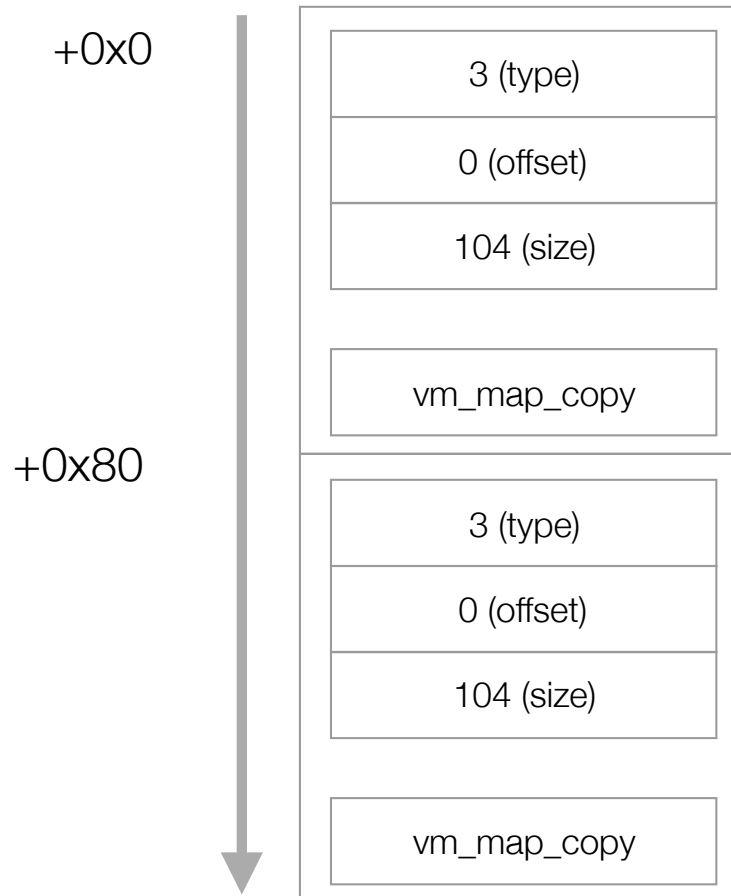
Leaking arbitrary memory in 10.11



- Allocate two adjacent `vm_map_copy` structures
- Corrupt the first one's size
- Read out the data, change the second structure's size to 24, write it back
- Read the second `vm_map_copy` out, causing a wrong free to the `kalloc.48` zone
- Allocate `OSData`

Leaking arbitrary memory in 10.11

- OSData object now overlaps vm_map_copy's data
- Can read/write to it in userland
- vtable pointer leaks kASLR slide
- Data pointer leaks a pointer to arbitrary user-controlled data
- Changing the data pointer and setting capacity to 0xFFFFFFFF allows arbitrary memory leaks on 10.11 -> Just use IORegistryEntryCreateCFProperties to retrieve data

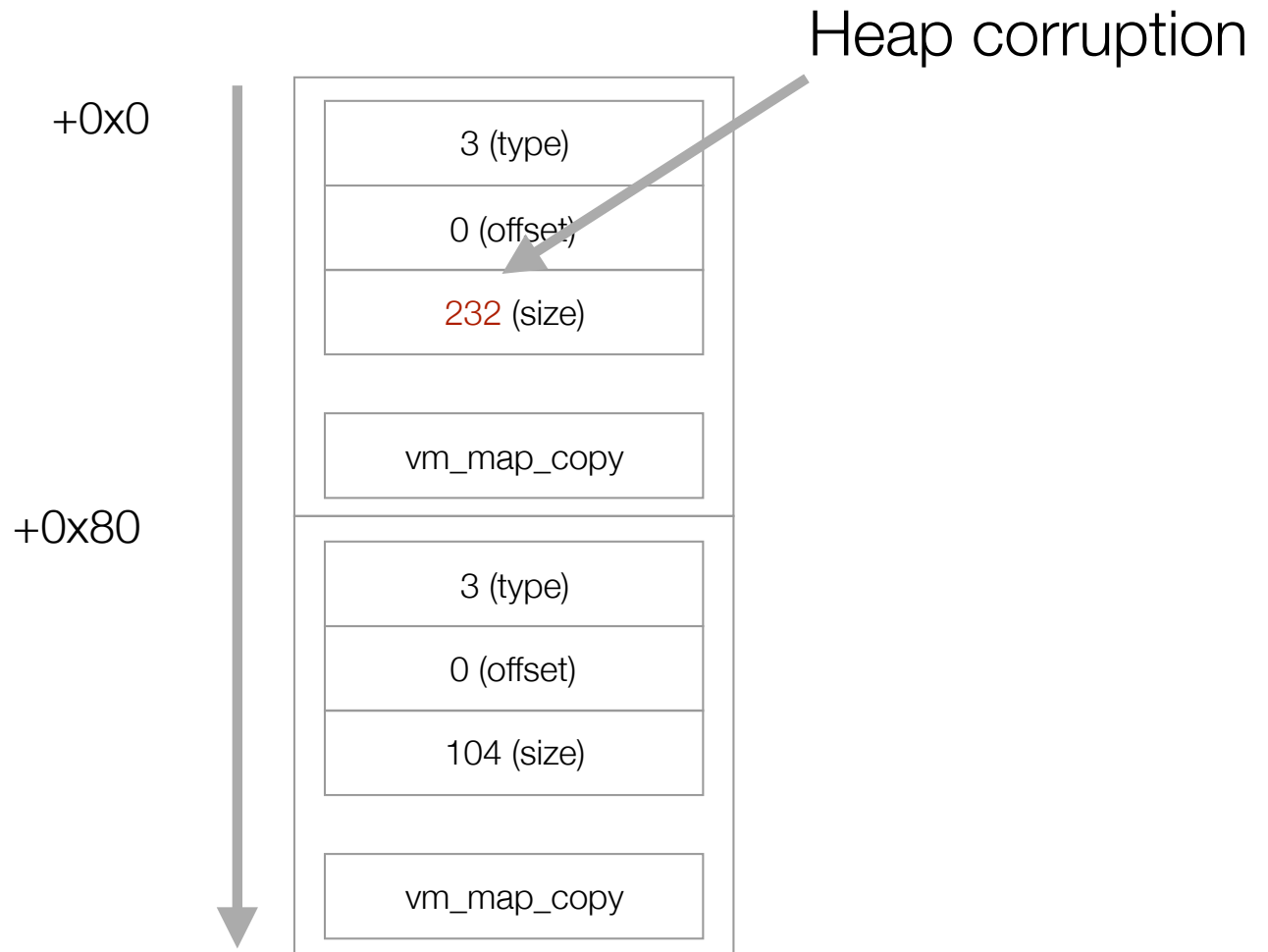
Leaking arbitrary memory in 10.11



 overlapping heap chunk  heap chunk

(assuming kalloc.128)

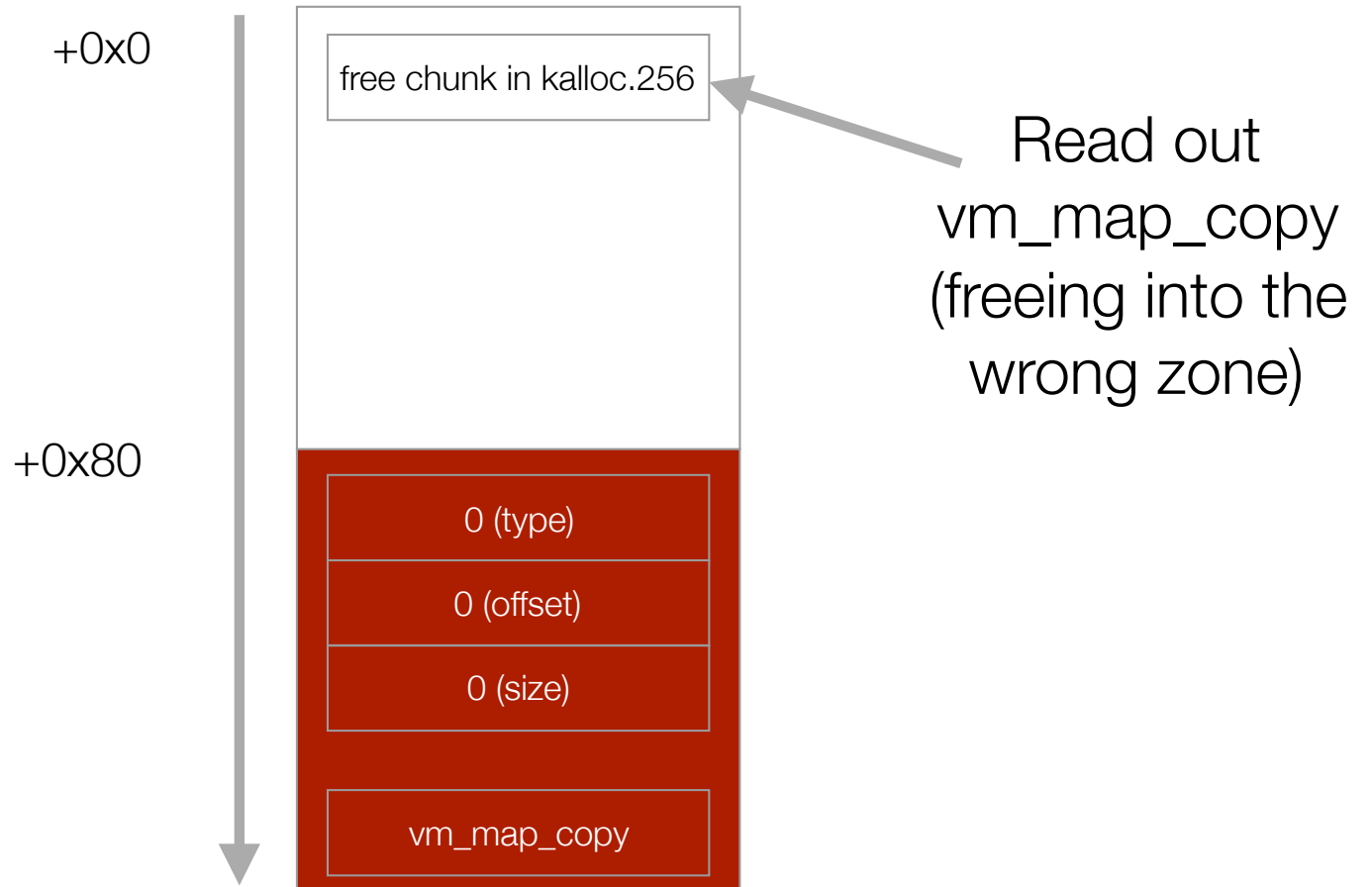
Leaking arbitrary memory in 10.11



 overlapping heap chunk  heap chunk

(assuming `kalloc.128`)

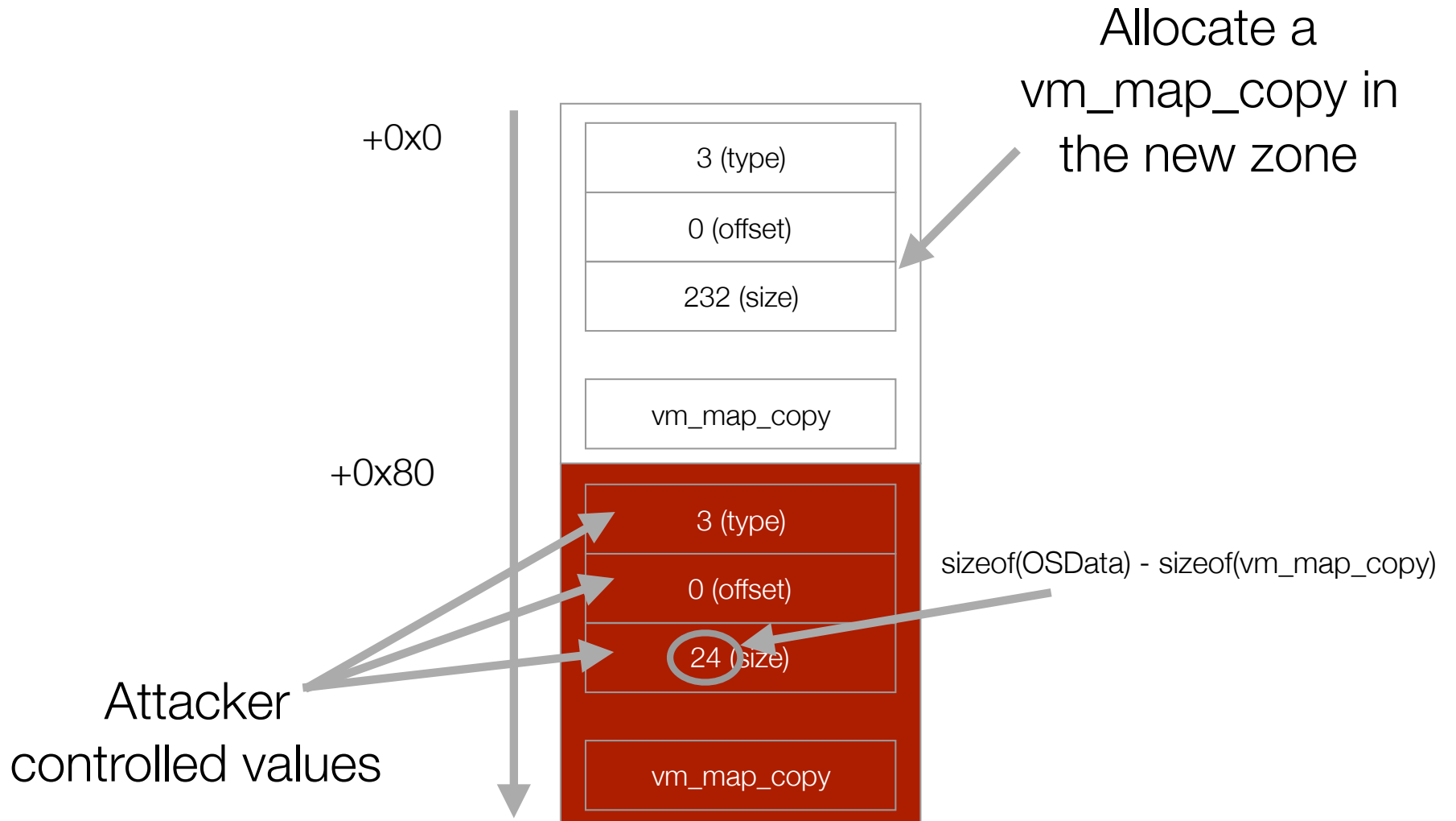
Leaking arbitrary memory in 10.11



 overlapping heap chunk  heap chunk

(assuming kalloc.128)

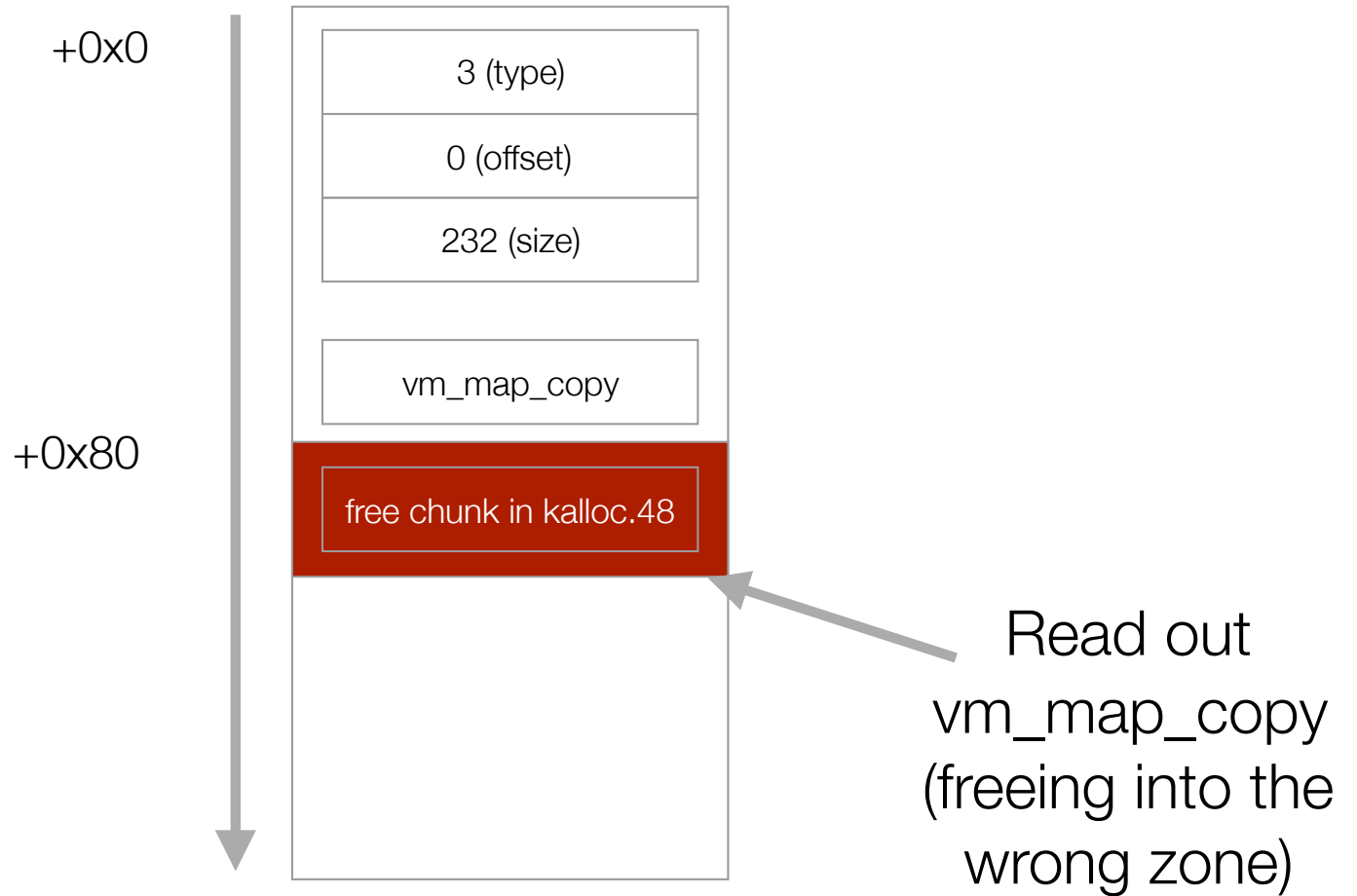
Leaking arbitrary memory in 10.11



 overlapping heap chunk  heap chunk

(assuming `kalloc.128`)

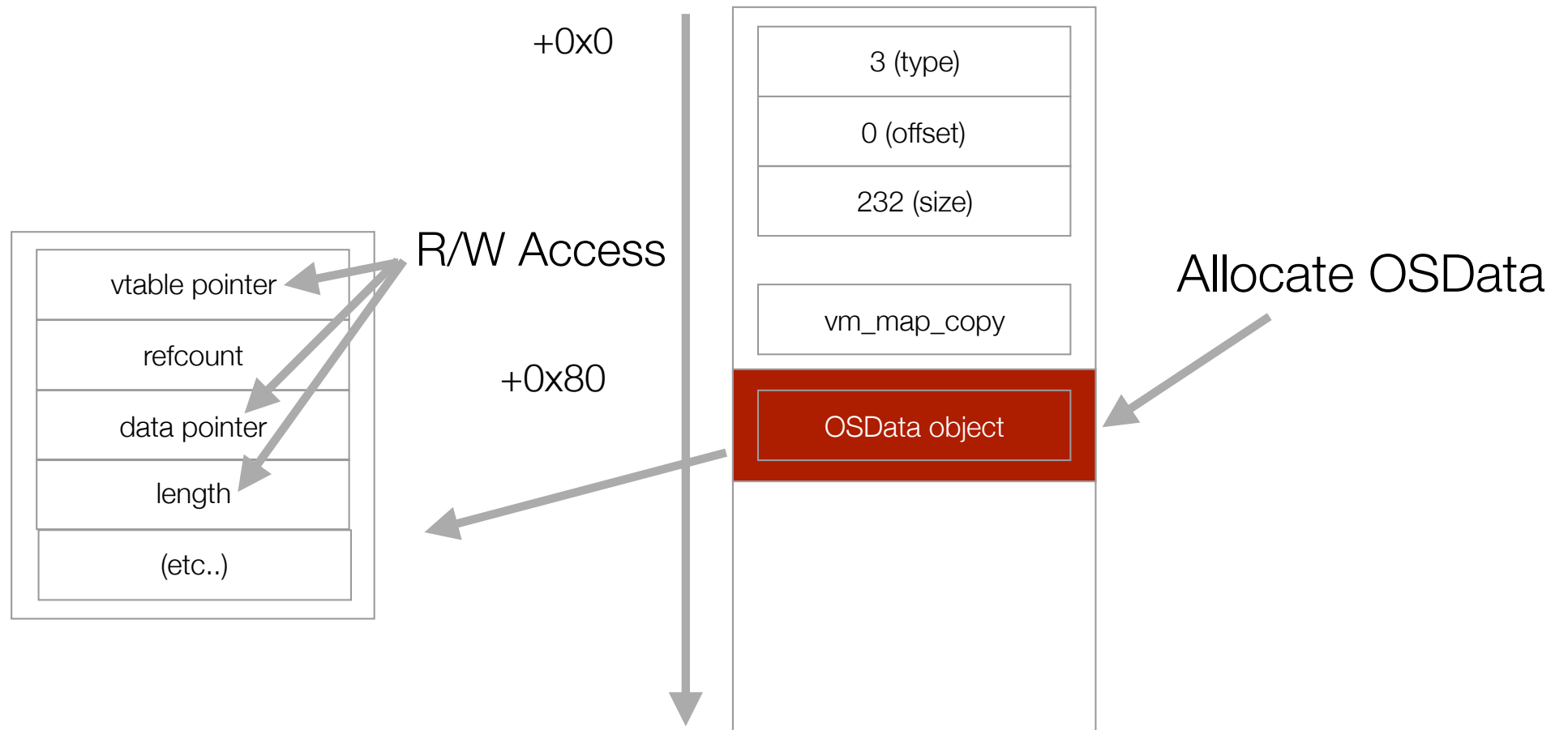
Leaking arbitrary memory in 10.11



 overlapping heap chunk  heap chunk

(assuming kalloc.128)

Leaking arbitrary memory in 10.11



 overlapping heap chunk  heap chunk

(assuming kalloc.128)

zalloc() timing attack

**A new technique to increase heap feng
shui reliability**

zalloc() Timing Attack

- Most heap attacks require adjacent allocations of some sort
- You can get adjacent allocations fairly easily by emptying the free list since the layout of allocations in newly mapped pages is deterministic
- However you don't get to know exactly when a particular free list runs out unless `uid=0` and `PE_i_can_has_debugger()` returns 1
- You can try to guess by picking an “high enough” number of allocations, but this yields to probabilistic exploits (which are good enough for e.g. jailbreaking)

zalloc() Timing Attack

- You can get adjacent allocations fairly easily by emptying the free list since **the layout of allocations in newly mapped pages is deterministic**
- Mapping pages is expensive
- Expensive enough to detect it in userland?

zalloc() Timing Attack

- In kalloc.1024, using a heap info leak to verify adjacency

time of execution of a
mach_msg call with OOL
data

vm_map_copyin
(newly mapped page)

```
timing attack: 1089 [0xffffffff8066f1c00]
timing attack: 343 [0xffffffff8066f16800]
timing attack: 334 [0xffffffff8066f16400]
timing attack: 436 [0xffffffff8066f16000]
timing attack: 1457 [0xffffffff8066f1c00]
timing attack: 386 [0xffffffff8066f18800]
timing attack: 369 [0xffffffff8066f18400]
timing attack: 360 [0xffffffff8066f18000]
timing attack: 1293 [0xffffffff8066f1c00]
timing attack: 353 [0xffffffff8066f19800]
timing attack: 362 [0xffffffff8066f19400]
timing attack: 350 [0xffffffff8066f19000]
timing attack: 1199 [0xffffffff8066f1c00]
timing attack: 346 [0xffffffff8066f1d800]
timing attack: 333 [0xffffffff8066f1d400]
timing attack: 346 [0xffffffff8066f1d000]
timing attack: 1897 [0xffffffff8066f1c00]
timing attack: 349 [0xffffffff8066f1e800]
timing attack: 334 [0xffffffff8066f1e400]
timing attack: 353 [0xffffffff8066f1e000]
timing attack: 1169 [0xffffffff8066f1c00]
timing attack: 347 [0xffffffff8066f1f800]
timing attack: 401 [0xffffffff8066f1f400]
timing attack: 389 [0xffffffff8066f1f000]
timing attack: 1293 [0xffffffff8066f2c00]
timing attack: 369 [0xffffffff8066f22800]
timing attack: 351 [0xffffffff8066f22400]
timing attack: 400 [0xffffffff8066f22000]
timing attack: 1130 [0xffffffff8066f24c00]
```

zalloc() Timing Attack

- You can get adjacent allocations fairly easily by emptying the free list since the layout of allocations in **newly mapped pages** is deterministic
- Mapping pages is expensive
- Expensive enough to detect it in userland? Yes!

zalloc() Timing Attack

- A good target to time is `vm_map_copyin`
- Create a bunch of `vm_map_copy` structs via `mach_msg`
- Read them out
- Recreate them, timing and keeping an average
- You are guaranteed that the average doesn't represent newly mapped memory
- Keeping those allocated, allocate more, timing `mach_msg`

zalloc() Timing Attack

- Once you get a mach_msg taking more time than the average * 1.5, a new page has just been mapped in
- Number of free list entries added = PAGE_SIZE/zone size
- Do more mach_msg timing
- A time spike is expected to happen after “number of free list entries added” allocations
- If it does, for additional reliability, do it again for another page

zalloc() Timing Attack

- Once you have pages filled with adjacent `vm_map_copy` structures, you can easily craft the heap layout by poking holes and reallocating the objects that most suit your needs
- Limit the number of allocations to some reasonable number to avoid running out of kernel memory
- On failure you can just fall back to a probabilistic approach

zalloc() Timing Attack: A practical use case

- In some rare cases extremely precise heap layout control is required to have any form of meaningful reliability
- An example is IOHIDFamily's CVE-2015-6974
- Fixed in 10.11.1, found independently by multiple parties*
- Used by Pangu9 and npwn
- Required uid=0 on OS X, container sandbox escape on iOS.
- Terminating an IOHIDUserDevice after creating one drops the reference count without setting pointers to it to NULL

*so far I'm aware of me, @panguteam and @cererdlong

CVE-2015-6974: A textbook Use-After-Free

```
IOReturn IOHIDResourceDeviceUserClient::terminateDevice()  
{  
    if (_device) {  
        _device->terminate();  
    }  
    OSSafeRelease(_device); ← Free  
  
    return kIOReturnSuccess;  
}
```

```
if ( arguments->scalarInput[0] )  
    AbsoluteTime_to_scalar(&timestamp) = arguments->scalarInput[0];  
else  
    clock_get_uptime( &timestamp );  
  
if ( !arguments->asyncWakePort ) {  
    ret = _device->handleReportWithTime(timestamp, report); ← Use  
    report->release();  
} else {  
    return value passed to userland  
    return ret;
```

Both of these functions are IOExternalMethods

CVE-2015-6974: OS"notso"SafeRelease

what Apple did



```
IOReturn IOHIDResourceDeviceUserClient::terminateDevice()  
{  
    if (_device) {  
        _device->terminate();  
    }  
    OSSafeRelease(_device);  
  
    return kIOReturnSuccess;  
}
```

```
/*! @function OSSafeRelease  
 * @abstract Release an object if not <code>NULL</code>.  
 * @param inst Instance of an OSObject, may be <code>NULL</code>.  
 */  
#define OSSafeRelease(inst) do { if (inst) (inst)->release(); } while (0)  
  
/*! @function OSSafeReleaseNULL  
 * @abstract Release an object if not <code>NULL</code>, then set it to <code>NULL</code>.  
 * @param inst Instance of an OSObject, may be <code>NULL</code>.  
 */  
#define OSSafeReleaseNULL(inst) do { if (inst) (inst)->release(); (inst) = NULL; } while (0)
```

what Apple really wanted to do



CVE-2015-6974

```
ret = _device->handleReportWithTime(timestamp, report);
```

vcall on free'd object at vtable+0x948

The bug allows you to control the vtable pointer used for this call

1st argument: pointer to UaF'd allocation

2nd argument: controlled 64 bit value

By controlling the vtable pointer you can get code exec easily with these constraints:

- on non-SMEP OS X you can point the vtable in userland and jump to user memory
- on non-SMAP OS X you can point the vtable in userland and ROP with a kASLR info leak
- on iOS and SMAP OS X you need to use an heap info leak as well as a kASLR info leak

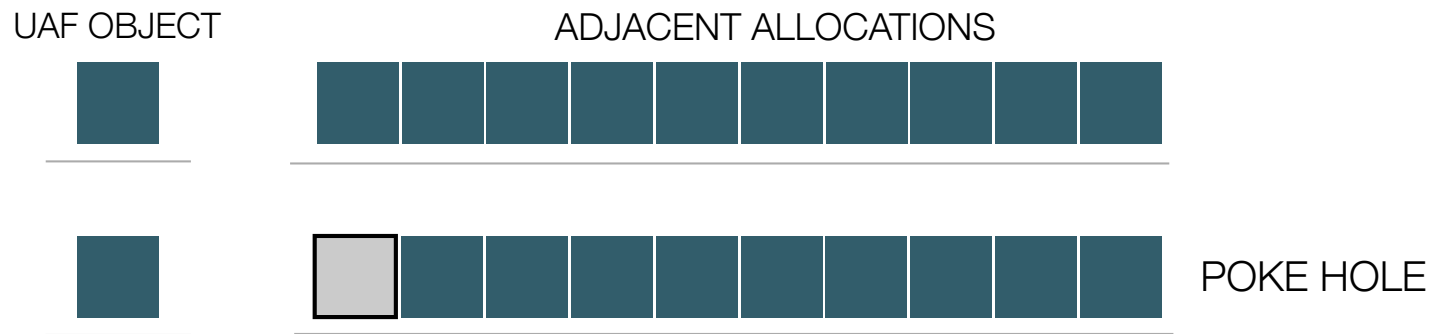
CVE-2015-6974

An alternate avenue for exploitation for SMAP / iOS requires a **tightly controlled heap layout**.
The vtable index for the vcall is 0x948 and the object lives in kalloc.256.



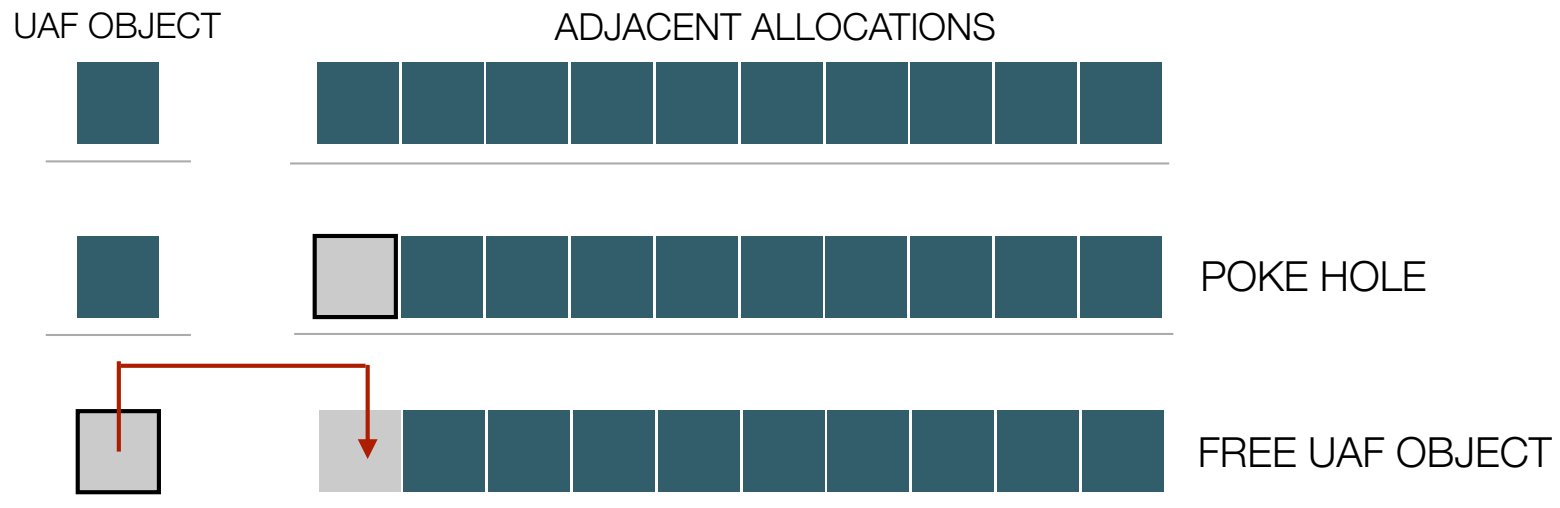
CVE-2015-6974

An alternate avenue for exploitation for SMAP / iOS requires a **tightly controlled heap layout**.
The vtable index for the vcall is 0x948 and the object lives in kalloc.256.



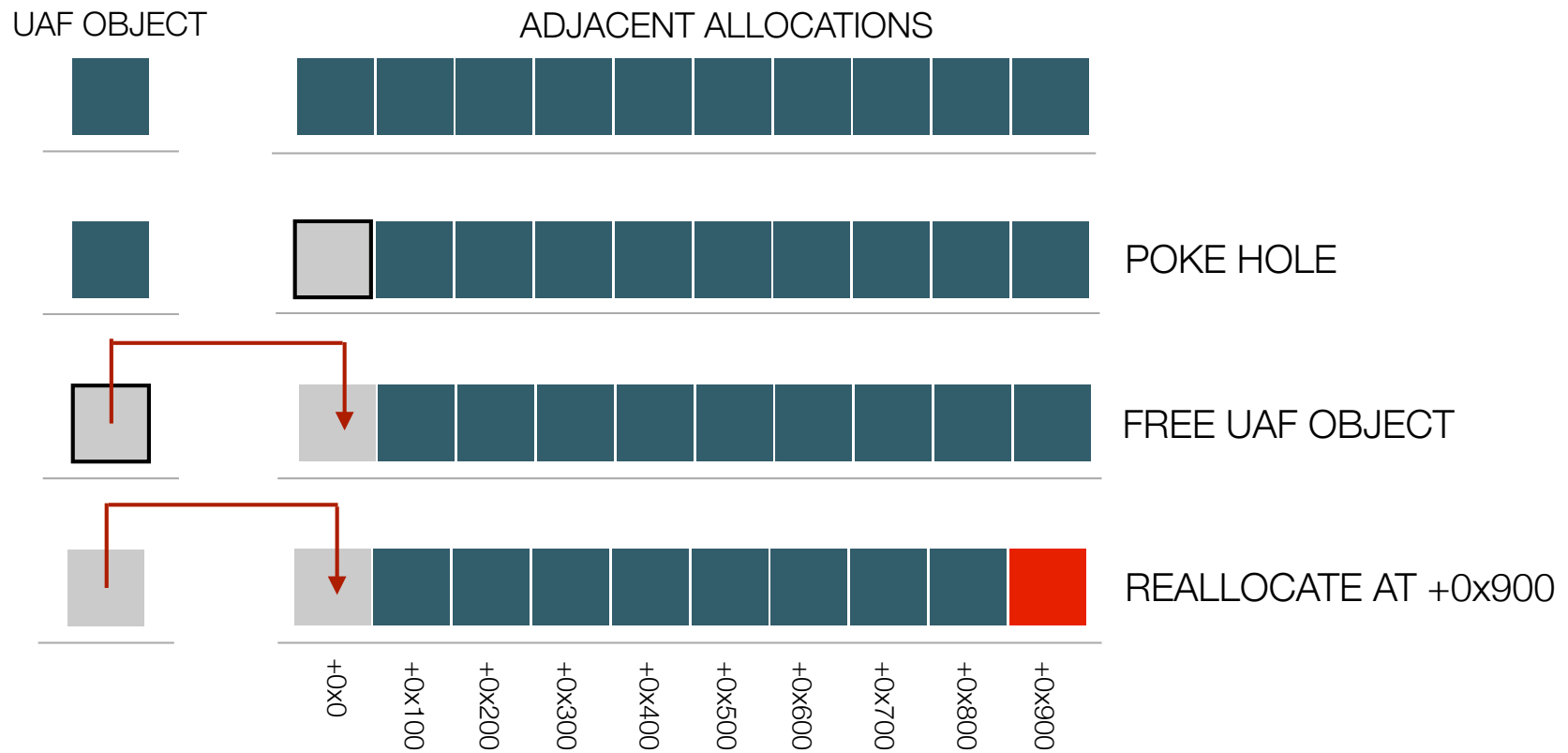
CVE-2015-6974

An alternate avenue for exploitation for SMAP / iOS requires a **tightly controlled heap layout**. The vtable index for the vcall is 0x948 and the object lives in kalloc.256.



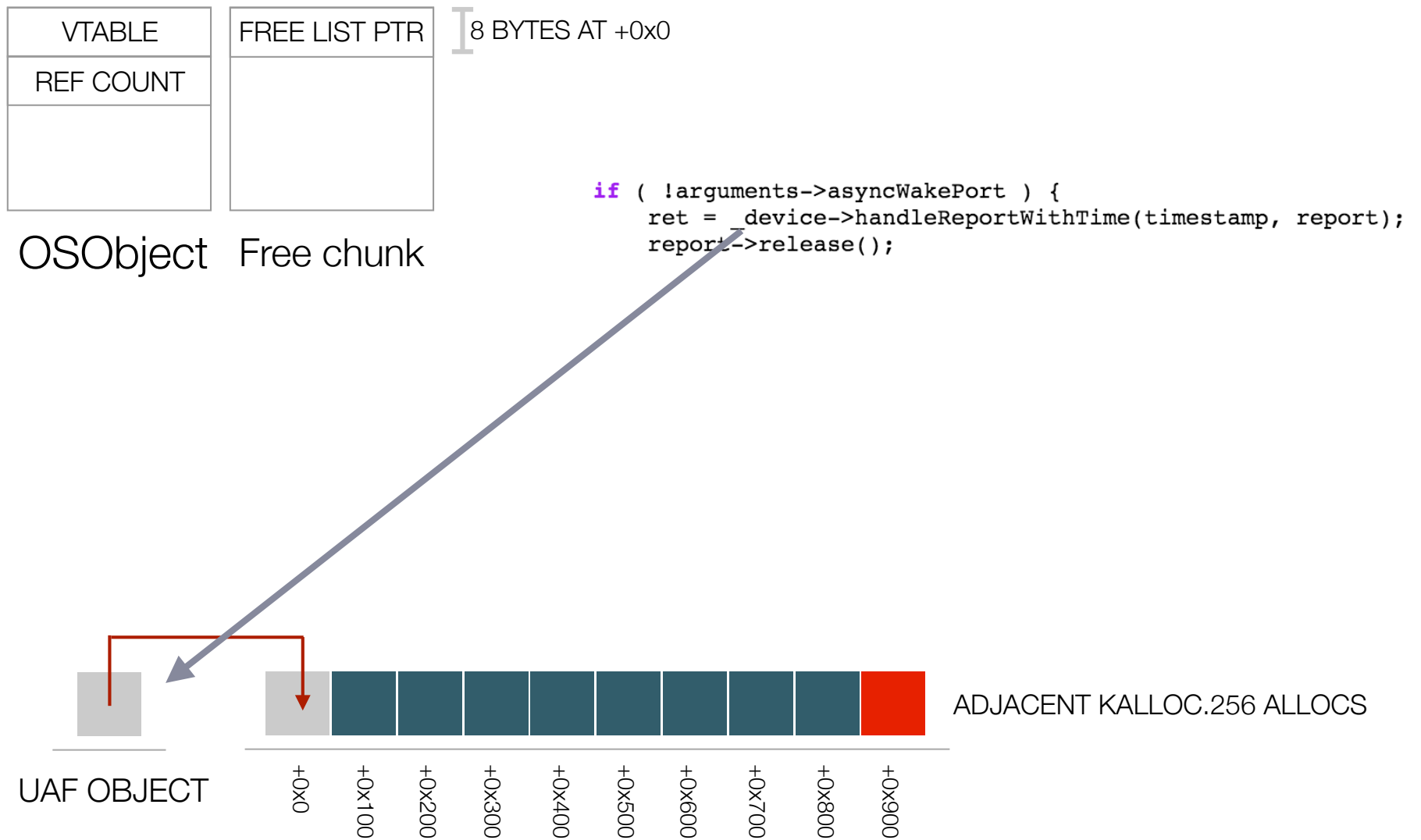
CVE-2015-6974

An alternate avenue for exploitation for SMAP / iOS requires a **tightly controlled heap layout**. The vtable index for the vcall is 0x948 and the object lives in kalloc.256.



CVE-2015-6974

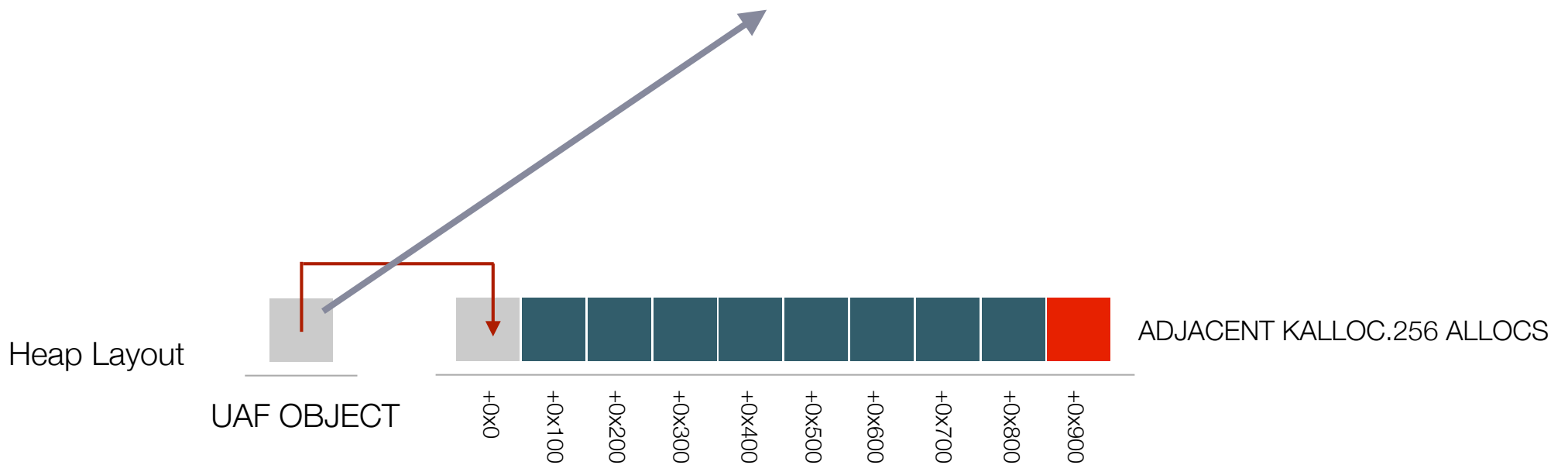
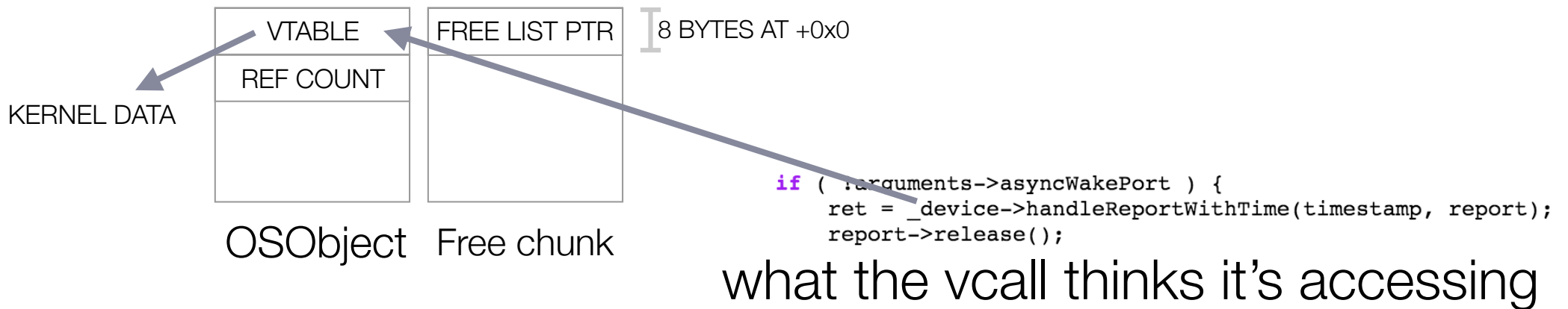
An alternate avenue for exploitation requires a tightly controlled heap layout.
The vtable index for the vcall is 0x948 and the object lives in kalloc.256.



CVE-2015-6974

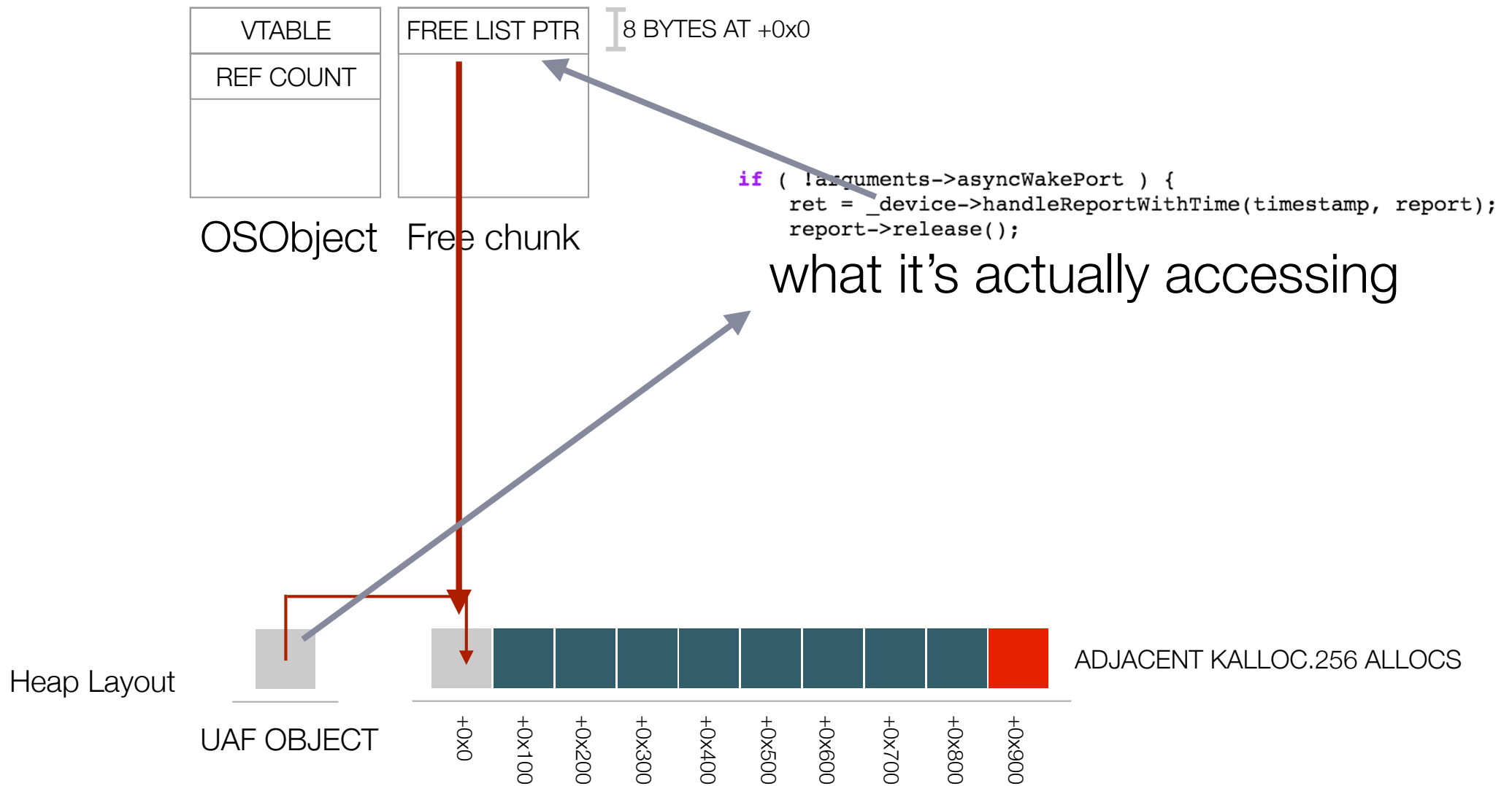
An alternate avenue for exploitation requires a tightly controlled heap layout.

The vtable index for the vcall is 0x948 and the object lives in kalloc.256.



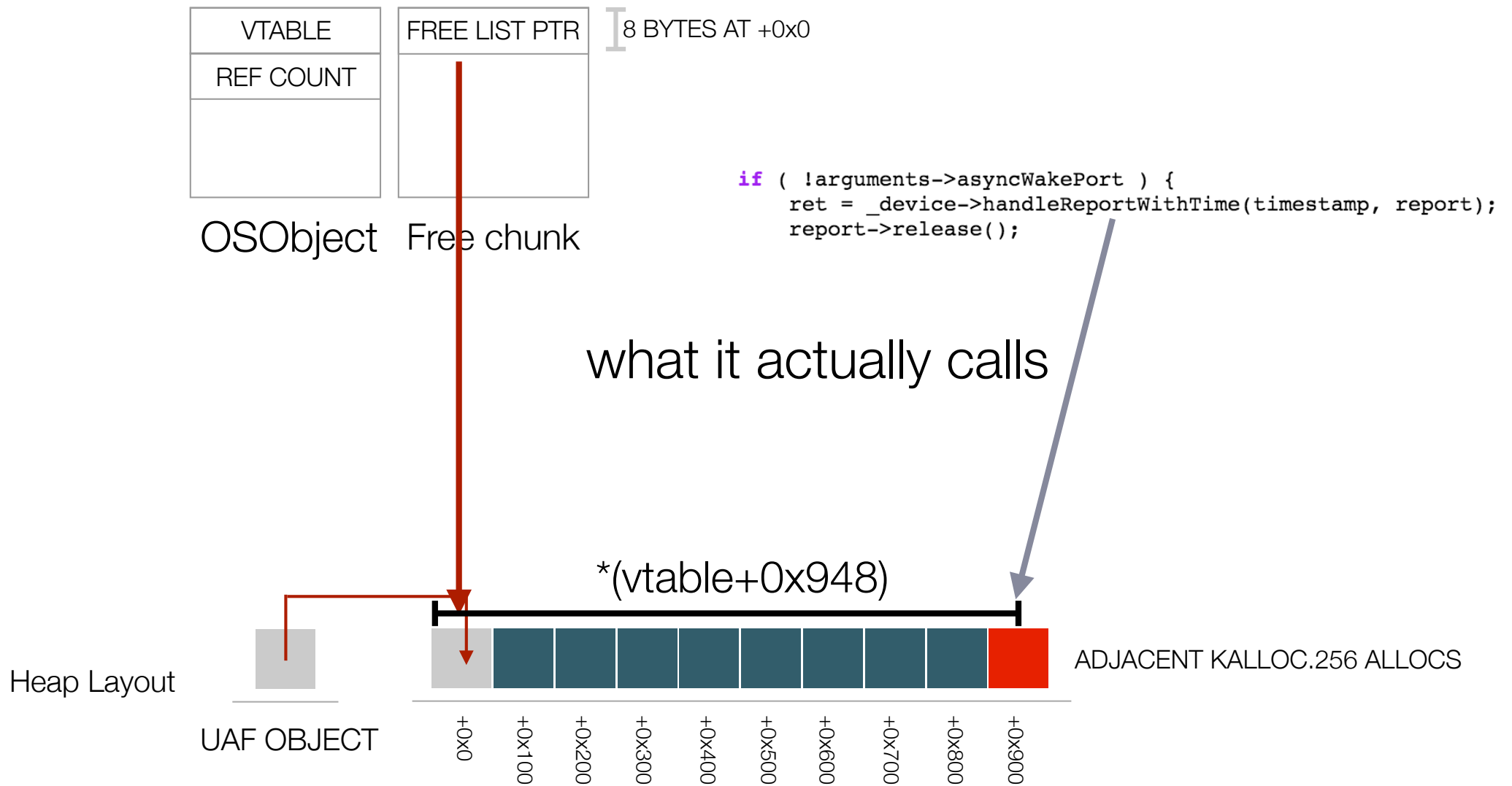
CVE-2015-6974

An alternate avenue for exploitation requires a tightly controlled heap layout.
The vtable index for the vcall is 0x948 and the object lives in kalloc.256.



CVE-2015-6974

An alternate avenue for exploitation requires a tightly controlled heap layout.
The vtable index for the vcall is 0x948 and the object lives in kalloc.256.



CVE-2015-6974

- We can now control the instruction pointer and the 2nd argument
- First argument is a pointer to the UaF'd allocation
- kASLR slide not leaked yet
 - In npwn I used “kas_info”, which could be considered cheating but is still allowed on SIP-protected 10.11.1
 - Alternative kASLR leaking strategy (used by Pangu9): abuse the UaF like a type confusion

Disabling System Integrity Protection

- Pedro Vilaça (@osxreverser) discussed `_csr_set_allow_all` for his “rootfool” kernel extension
- We can just redirect the `vcall` to `_csr_set_allow_all`
- As long as the first argument is non-NULL, it'll disable SIP for good
- ROP is not needed at all

Demo!

Black Hat Sound Bytes

- The rapid growth in use of sandboxing technology is pushing many attackers to kernel attacks.
- Apple has been trying to harden the kernel heap for years now but it's still fairly easy to carry out attacks.
- The zalloc timing attack can prove useful in many situations

Questions?

Twitter: @qwertyoruiop

Mail: me at qwertyoruiop dot com

Thanks to:

- windknown (@windknown) & Pangu Team (@PanguTeam)
 - Pangu9 was amazing stuff!
- Steven De Franco (@iH8sn0w)
- Filippo Bigarella (@FilippoBiga)
- Joshua Hill (@p0sixninja)
- Nicholas Allegra (@comex)
- Jonathan Levin (@Technologeeks / <http://newosxbook.com/>)
- Stefan Esser (@i0n1c)
 - Make sure you've read his XNU exploitation papers!
- Pedro Vilaça (@osxreverser)
- Mark Dowd (@mdowd)
- Tarjei Mandt (@kernelpool)