Attacking the iOS Kernel: A Look at ‘evasi0n’

Tarjei Mandt
tm@azimuthsecurity.com
@kernelpool

About Me

- Senior Security Researcher at Azimuth Security
- Recent focus on Apple iOS/OSX
- Previously done research on Windows
    - Windows 8 Heap Internals (w/ Chris Valasek)
    - http://mista.nu/blog
- In the program committee of a few conferences
    - WISA 2013 (http://www.wisa.or.kr)
    - NSC (http://www.nosuchcon.org)
- MSc in Information Security from GUC ☺

iOS 6

- Apple released iOS 6 in September 2012
- Large focus on security improvements
    - E.g. offers kernel address space layout randomization (KASLR)
- Primarily targets strategies employed in «jailbreaks»
- Additional security improvements in iOS 6.1
    - E.g. service hardening (plist signing)

evasi0n Jailbreak

- First public jailbreak on iOS 6
  - Released February 2013
  - http://www.evasi0n.com
- Allows users to run unsigned code without sandbox restrictions
- Comprises several components
  - Injection vector, persistence (survive reboot), etc.
- Kernel exploit used to gain full control of the operating system

Talk Outline

- Part 1: iOS 6 Kernel Security
  - Kernel Address Space Layout Randomization
  - Kernel Address Space Protection
  - Information Leak Mitigations
- Part 2: evasi0n Kernel Exploit
  - Vulnerability
  - Information Leaking Strategies
  - Gaining Arbitrary Code Execution
  - Exploitation Techniques

Recommended Reading

- Presentations/Papers
  - iOS 6 Kernel Security: A Hacker's Guide
  - Dion Blazakis – The Apple Sandbox
  - Charlie Miller – Breaking iOS Code Signing
  - Various iOS talks by Stefan Esser
- Books
  - iOS Hacker's Handbook
  - A Guide to Kernel Exploitation: Attacking the Core
  - OS X and iOS Kernel Programming
  - Mac OSX and iOS Internals: To the Apple's Core

iOS 6 Kernel Security

Attacking the iOS Kernel

Kernel ASLR

- Goal
  - Prevent attackers from modifying/utilizing data at known addresses
- Strategy is two-fold
  - Randomize kernel image base
  - Randomize base of kernel_map

Kernel ASLR – Kernel Image

- Kernel base randomized by boot loader (iBoot)
  - Random data generated
  - SHA-1 hash of data taken
  - Byte from SHA-1 hash used to calculate kernel slide
- Kernel is rebased using the formula: 0x01000000 + (slide_byte * 0x00200000)
  - If byte is 0, static offset of 0x21000000 is used

Kernel ASLR – Kernel Image

- Calculated value added to the kernel preferred base later on
  - Adjusted base = 0x80000000 + slide
- Kernel can be rebased at 256 possible locations
  - Base addresses are 2MB apart (ARM cache optimization)
  - Example: 0x81200000, 0x81400000, … 0xA1000000
- Adjusted base passed to kernel via boot argument structure

Kernel ASLR – Kernel Map

- Used for kernel allocations of all types
  - kalloc( ), kernel_memory_allocate( ), etc.
- Spans all of kernel space
  - 0x80000000 -> 0xFFFEFFFF
- Kernel-based maps are submaps of kernel_map
  - zone_map, ipc_kernel_map, etc.
- Initialized by kmem_init()

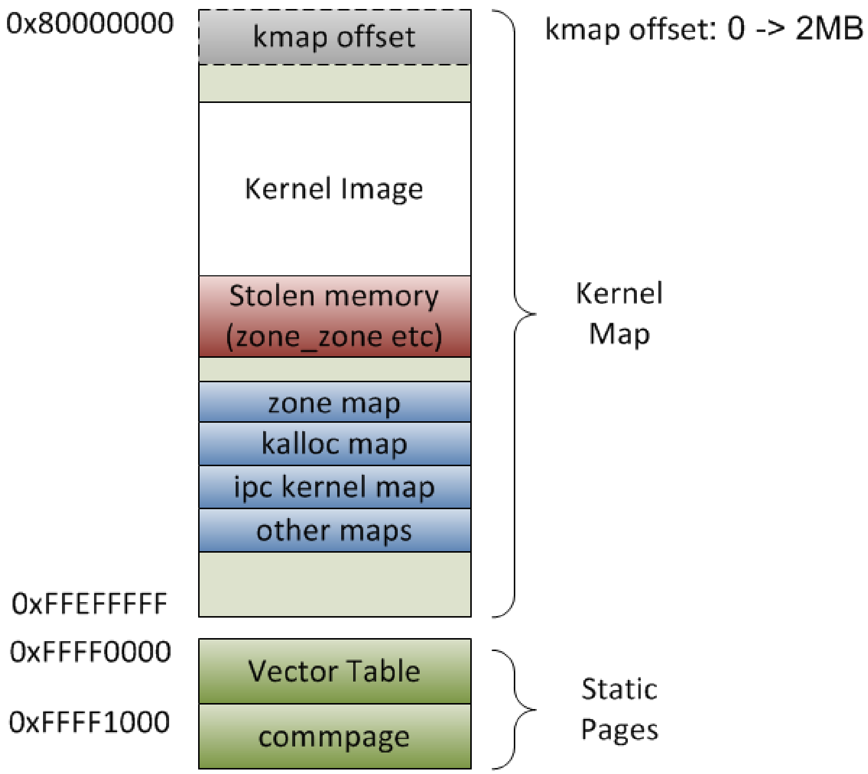
Kernel ASLR – Kernel Map

- Goal: Make kernel map allocations less predictable
- Strategy: Randomize the base of the kernel map
  - Random 9-bit value generated
  - Multiplied by page size
  - Resulting value used for initial kernel_map allocation
  - 9 bits = 512 different allocation size possibilities

Kernel ASLR – Kernel Map

- Subsequent kernel_map (including submap) allocations pushed forward by random amount
  - Allocation silently removed after first garbage collection
- Behavior can be overridden with «kmapoff» boot parameter

Kernel ASLR – Kernel Map



iOS 6 Kernel Memory Layout

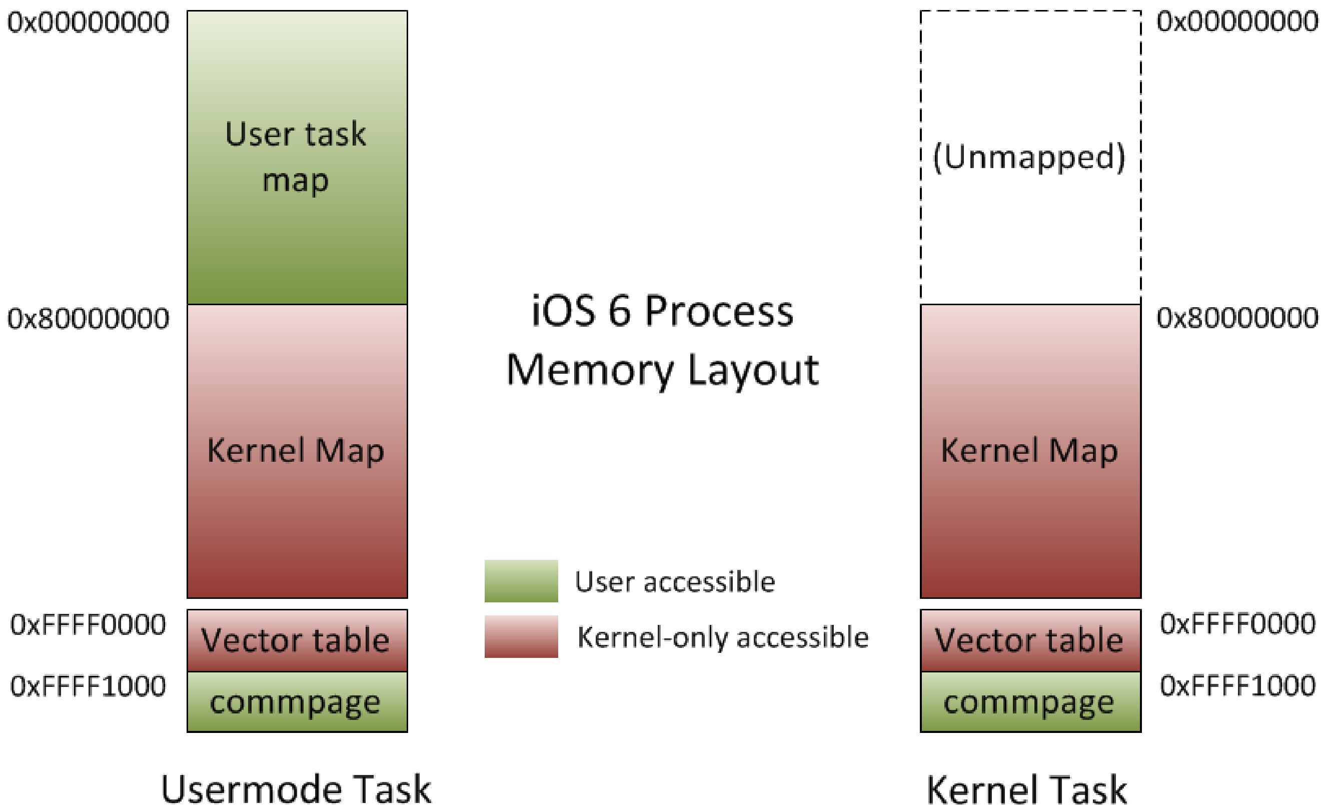
Kernel Address Space Protection

- Goal: Prevent user-mode dereference vulnerabilities (from kernel)
  - E.g. offset-to-null
- Previously, kernel and user shared address space
- NULL-dereferences were prevented by forcing binaries to have __PAGE_ZERO section
  - Does not prevent dereferences above this section

Kernel Address Space Protection

- In iOS 6, the kernel task has its own address space while executing
  - Transitioned to with interrupt handlers
  - Switched between during copyin() / copyout()
- Also configurable on 64-bit OSX with the no_shared_cr3 boot argument
- User-mode pages therefore not accessible while executing in kernel mode

Kernel Address Space Protection



Kernel Address Space Protection

- ARMv6+ has two translation table base registers
  - TTBR0: process specific addresses
  - TTBR1: OS (kernel) and I/O addresses
- On iOS 6, TTBR1 is mirrored to TTBR0 while the kernel is executing
- TTBR0 is set to process table during copyin() / copyout()
  - Also switches ASID to prevent cache leaks

Kernel Address Space Protection

- Memory is no longer RWX
  - Kernel code cannot be directly patched
  - Heap is non-executable
  - Stack is non-executable
- Syscall table is no longer writable
  - Moved into DATA const section

Information Leaking Mitigations

- Goals
  - Prevent disclosure of kernel base
  - Prevent disclosure of kernel heap addresses
- Strategies
  - Disables some APIs
  - Obfuscate kernel pointers for some APIs
  - Zero out pointers for others

Information Leaking Mitigations

- Previous attacks relied on zone allocator status disclosure
  - host_zone_info() / mach_zone_info()
- Allowed attacker to determine the number of allocations needed to fill a particular zone
  - Used to defragment a heap
- APIs now require debug access (configured using boot argument)

Information Leaking Mitigations

- Several APIs disclose kernel object pointers
  - mach_port_kobject( )
  - mach_port_space_info( )
  - vm_region_recurse( )
  - vm_map_region_recurse( )
  - proc_info(…)
  - fstat( ) (when querying pipes)
  - sysctl(net.inet.* .pcblist)

Information Leaking Mitigations

- Need these APIs for lots of reasons
  - Often, underlying APIs rather than those previously listed
- Some pointer values are used as unique identifiers to user mode
  - E.g. pipe inode number
- Strategy: Obfuscate pointers
  - Generate random value at boot time
  - Add random value to real pointer

Information Leaking Mitigations

```
/*
 * Initialize the global used for permuting kernel
 * addresses that may be exported to userland as tokens
 * using VM_KERNEL_ADDRPERM(). Force the random number
 * to be odd to avoid mapping a non-zero
 * word-aligned address to zero via addition.
 */
vm_kernel_addrperm = (vm_offset_t)early_random() | 1;
```

Generate random value at boot time

Macro for obfuscating kernel pointers

```
#define   VM_KERNEL_ADDRPERM(_v)              \
          (((vm_offset_t)(_v) == 0) ?         \
                     (vm_offset_t)(0) :       \
                     (vm_offset_t)(_v) + vm_kernel_addrperm)
```

Example use: obfuscated pipe object pointer

```
/*
 * Return a relatively unique inode number based on the current
 * address of this pipe's struct pipe.  This number may be recycled
 * relatively quickly.
 */
sb->st_ino = (ino_t)VM_KERNEL_ADDRPERM((uintptr_t)cpipe);
```

Information Leaking Mitigations

- Other APIs disclose pointers unnecessarily
  - Zero them out
- Used to mitigate some leaks via sysctl()
  - E.g. known process structure info leak

Heap / Stack Hardening

- Cookie introduced to the kernel stack
  - Aims to mitigate return address overwrite
- Multiple hardenings to the kernel heap
  - Pointer validation
  - Block poisoning
  - Freelist integrity verification
- Described in more detail in «iOS 6 Kernel Security: A Hacker's Guide»

evasi0n Kernel Exploit

Attacking the iOS Kernel

evasi0n

- Uses a kernel vulnerability to gain full control of the OS kernel
  - com.apple.iokit.IOUSBDeviceFamily
- Primarily required to evade sandbox restrictions and code signing enforcement
- Arguably the most complex public kernel exploit seen to date on iOS
  - Written by David Wang (@planetbeing)

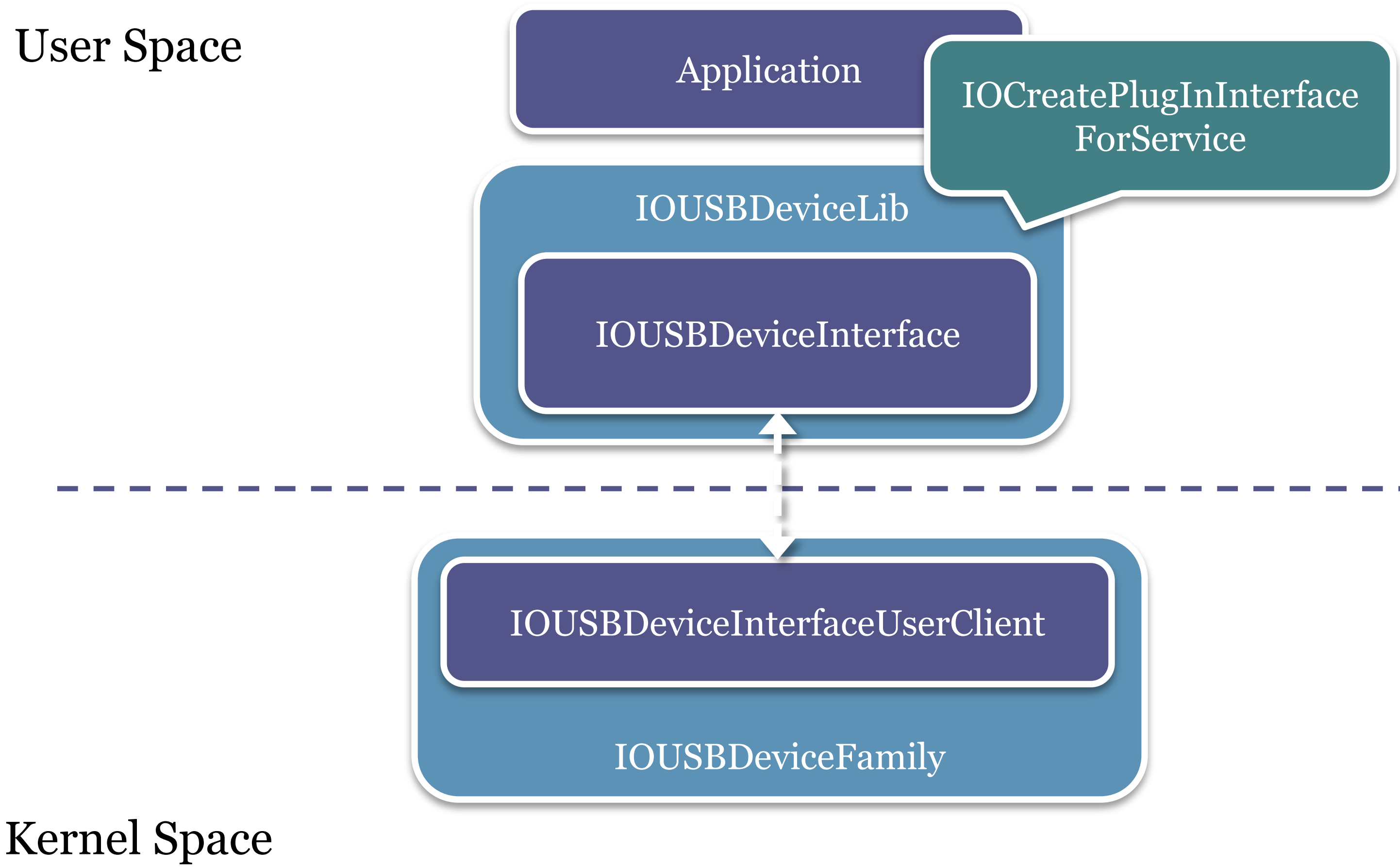
IOUSBDeviceFamily

- Kernel extension enabling a device to communicate with a host over USB
  - E.g. to iTunes or accessory port devices
- Used by various applications and daemons
  - Picture-transport-protocol daemon
  - Media server daemon (usb audio streaming)
- Represents the device end, whereas IOUSBFamily (OSX) represents the host end

IOUSBDeviceInterface

- IOKit class used to represent a USB interface on a device
- Provides a user client for user space access
  - IOUSBDeviceInterfaceUserClient
  - Exposes various methods to support USB interaction
- Commonly accessed from a user-space library
  - IOUSBDeviceFamily.kext/PlugIns/IOUSBDeviceLib.plugin
  - Implemented as a CFPlugIn extension
- Accessible to tasks with the USB entitlement (com.apple.security.device.usb)

IOUSBDeviceInterface Interaction



Pipe Translation

- A *pipe* is the communication channel between a host and a device endpoint
- Applications normally access pipes by their index value
  - Index 0: default control pipe
  - GetNumEndpoints() on interface object
- Value passed in as argument to user client
  - Translates pipe index to real pipe object
  - Performs operation with pipe object

Pipe Translation in IOUSBFamily (OSX)

```
IOReturn
IOUSBInterfaceUserClientV2::ResetPipe(UInt8 pipeRef)
{
    IOUSBPipe                                *pipeObj;
    IOReturn                                 ret;

    ...

    if (fOwner && !isInactive())
    {
               pipeObj = GetPipeObj(pipeRef);
               if (pipeObj)
               {
                      ret = pipeObj->Reset();
                      pipeObj->release();
               }
               else
                      ret = kIOUSBUnknownPipeErr;
    }
```

User client takes pipe index (pipeRef) as input

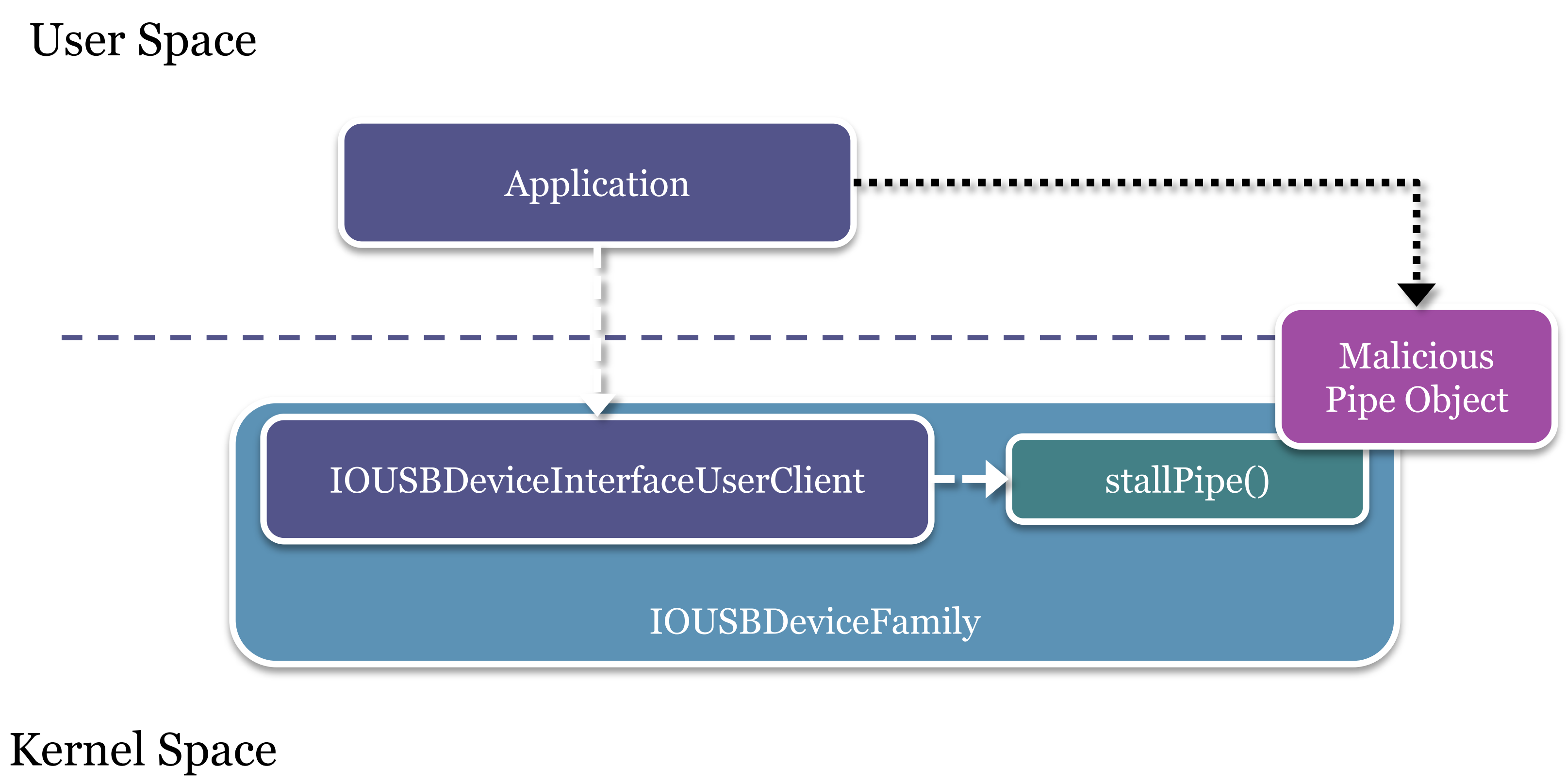
Pipe index translated to pipe object

IOUSBFamily-540.4.1\IOUSBUserClient\Classes\IOUSBInterfaceUserClient.cpp

IOUSBDeviceFamily Vulnerability

- The IOUSBDeviceInteface user client <u>does not</u> operate with pipe index values
  - Pipe object pointers passed in directly from user mode
- Methods exposed by the user client only check if the pipe object pointer is non-null
  - E.g. read/writePipe, abortPipe, and stallPipe
- An attacker can connect to the user client and specify an arbitrary pipe pointer

IOUSBDeviceFamily Vulnerability



stallPipe() Disassembly #1

```
0000:80660EE8 ; unsigned int stallPipe(int interface, int pipe)
0000:80660EE8
0000:80660EE8          PUSH           {R7,LR}
0000:80660EEA          MOVW           R0, #0x2C2
0000:80660EEE          MOV            R7, SP
0000:80660EF0          MOVT.W         R0, #0xE000
0000:80660EF4          CMP            R1, #0          // is pipe object pointer null?
0000:80660EF6          IT EQ
0000:80660EF8          POPEQ          {R7,PC}         // return if null
0000:80660EFA          MOV            R0, R1
0000:80660EFC          BL             __stallPipe     // pass in as arg if non-null
0000:80660F00          MOVS           R0, #0
0000:80660F02          POP            {R7,PC}
```

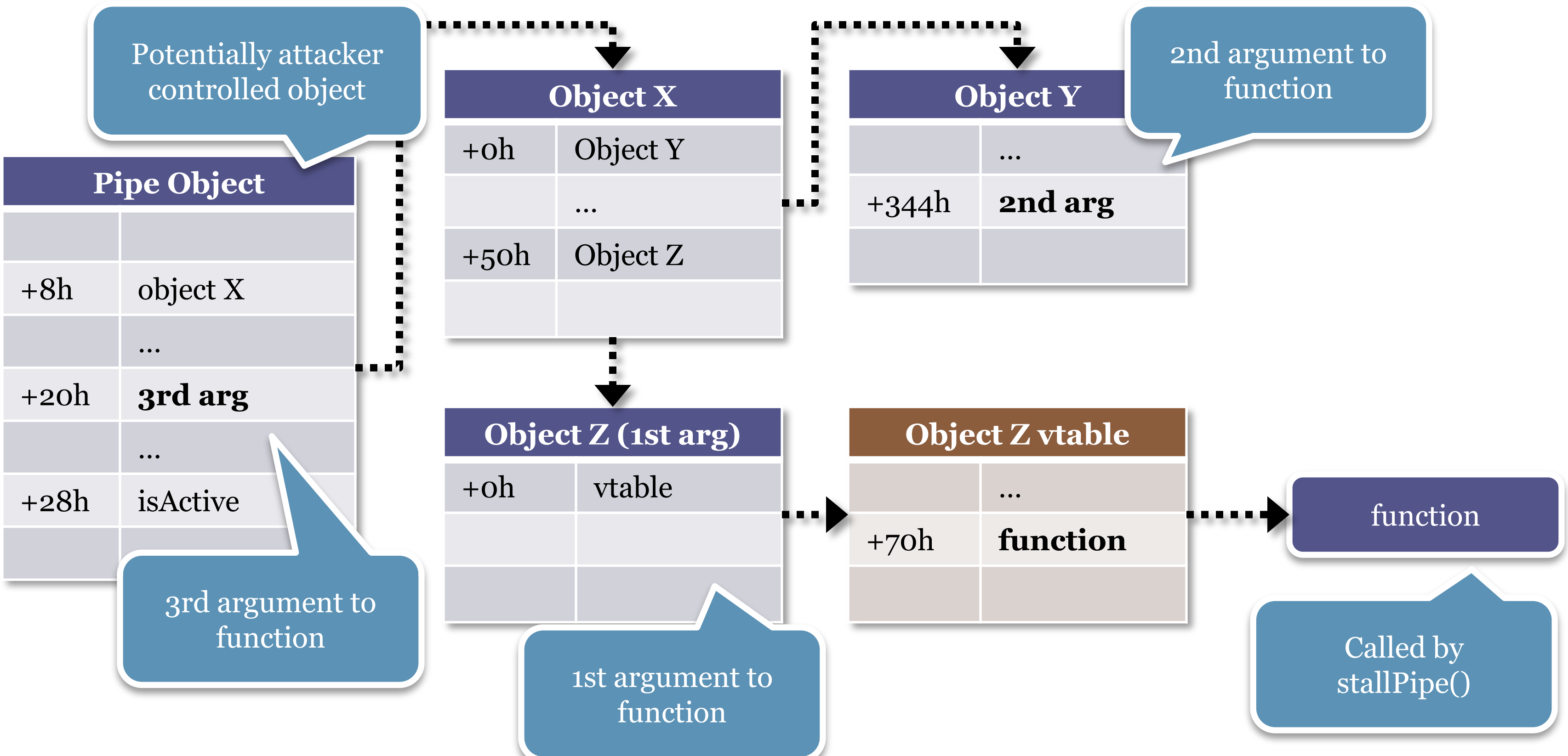
stallPipe() Disassembly #2

```
0000:8065FC60 __stallPipe
0000:8065FC60         LDR         R1, [R0,#0x28]
0000:8065FC62         CMP         R1, #1           // check if active
0000:8065FC64         IT NE
0000:8065FC66         BXNE        LR
0000:8065FC68         LDR         R2, [R0,#8]      // get object X from pipe object
0000:8065FC6A         LDR         R1, [R0,#0x20]   // get value from pipe object
0000:8065FC6C         MOV         R0, R2
0000:8065FC6E         MOVS        R2, #1
0000:8065FC70         B.W         sub_80661B70
```

stallPipe() Disassembly #3

```
0000:80661B70 ; int sub_80661B70(int interface)
0000:80661B70
0000:80661B70        PUSH        {R7,LR}
0000:80661B72        MOV         R7, SP
0000:80661B74        SUB         SP, SP, #8
0000:80661B76        LDR.W       R9, [R0]         // get object Y from object X
0000:80661B7A        MOV         R12, R2
0000:80661B7C        LDR         R0, [R0,#0x50]   // get object Z from X (1st arg)
0000:80661B7E        MOV         R2, R1           // 3rd arg
0000:80661B80        LDR.W       R1, [R9,#0x344]  // get value from Y (2nd arg)
0000:80661B84        LDR         R3, [R0]         // object Z vtable
0000:80661B86        LDR.W       R9, [R3,#0x70]   // get function from Z vtable
0000:80661B8A        MOVS        R3, #0
0000:80661B8C        STR         R3, [SP,#0x10+var_10]
0000:80661B8E        STR         R3, [SP,#0x10+var_C]
0000:80661B90        MOV         R3, R12
0000:80661B92        BLX         R9               // call function
0000:80661B94        ADD         SP, SP, #8
0000:80661B96        POP         {R7,PC}
```

stallPipe() Object Handling



Exploitation

- An attacker who is able to control the referenced memory can control execution
- On iOS 5, the attacker could allocate memory in user-mode in order to fully control the object
  - Easy win
- On iOS 6, user/kernel address space separation does not allow this
  - Evasion must find a way to inject user controlled data into kernel memory

Attack Strategy

- Inject user controlled data into kernel memory
  - Need to control the values of the fake pipe object
- Learn the location of user controlled data
  - Typically requires an information disclosure
- Learn the base address of the kernel
  - Required in order to patch sandbox and code signing checks
- Build read and write primitives
  - Arbitrary read/write to kernel memory

Information Disclosure

- An application can request a memory mapping when interacting with IOUSBDeviceInterface
  - Selector method 18 – createData()
  - Produces an IOMemoryMap kernel object
- The IOMemoryMap object address is returned to the user as a «map token»
  - Object addresses typically used as handles/identifiers
  - kalloc(68) -> allocated in the kalloc.88 zone

Information Disclosure

```
uint64_t length = 1024;
uint64_t output[3];
uint32_t outputCnt = 3;

rc = IOConnectCallScalarMethod( dataPort, 18, &length, 1, output, &outputCnt );

if ( KERN_SUCCESS != rc )
{
    printf( "Unable to map memory\n" );
    return 0;
}

printf( "data ptr: %x\n", (uint32_t) output[0] );
printf( "capacity: %x\n", (uint32_t) output[1] );
printf( "map token: %x\n", (uint32_t) output[2] );
```

IOUSBDeviceInterface user client

```
data ptr: 446c000
capacity: 1000
map token: a48fb948
```

Address in kalloc.88 zone

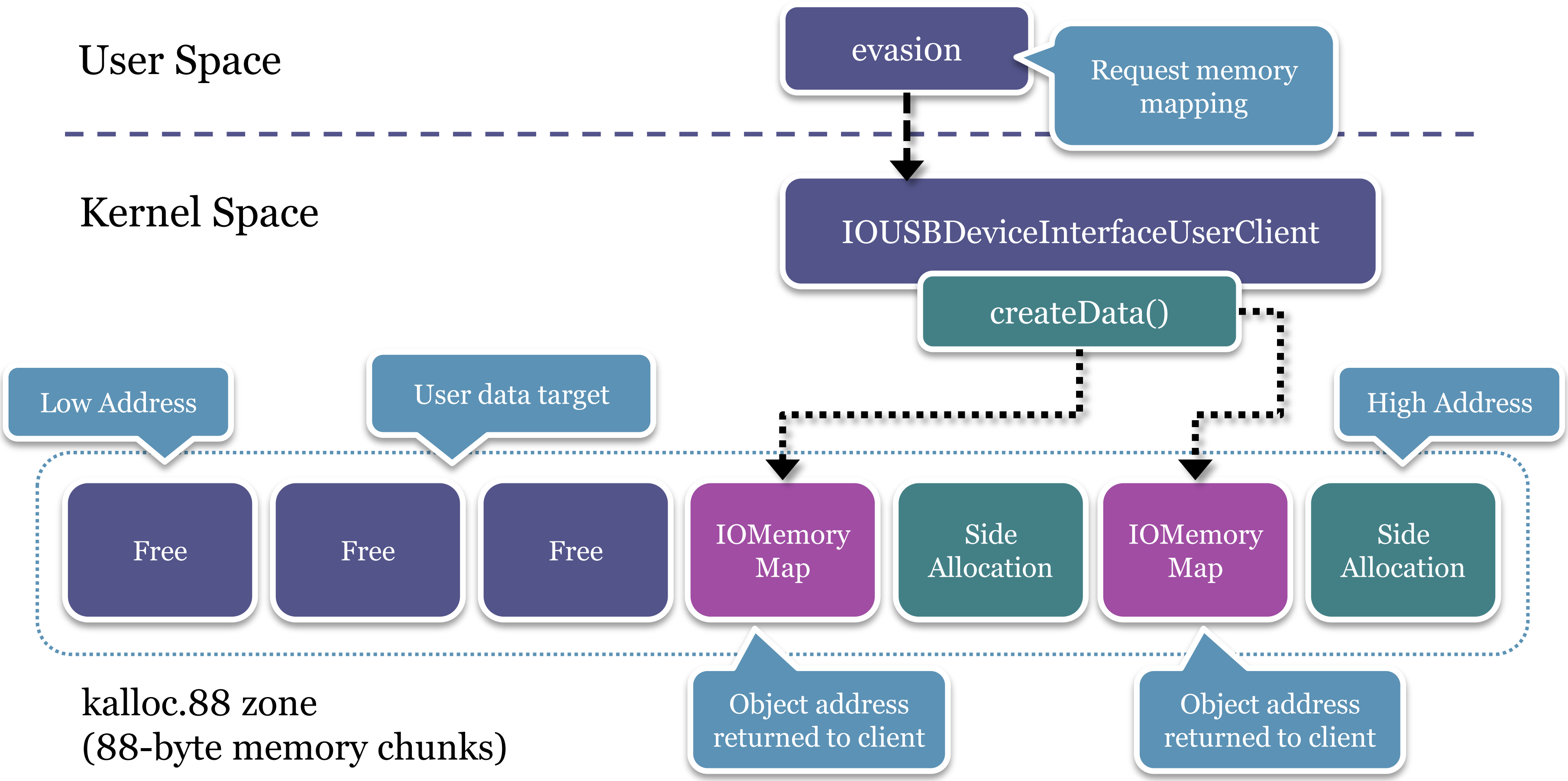
Defragmenting the Kernel Heap

- Information disclosure is more useful with a predictable kernel heap
  - Can be used to infer the location of user data
- A defragmented (filled) heap is more predictable
  - New pages used for subsequent allocations
    - Divided into equally sized chunks
    - E.g. 88 bytes for kalloc.88 zone
  - New chunks served in a sequential manner

Defragmenting the Kernel Heap

- evasion requests memory mappings until the kernel heap is defragmented
  - Waits until it has 9 sequentially positioned IOMemoryMap objects
- Subsequent allocations assumed to fall directly next to the last IOMemoryMap object
  - Target for user data injection

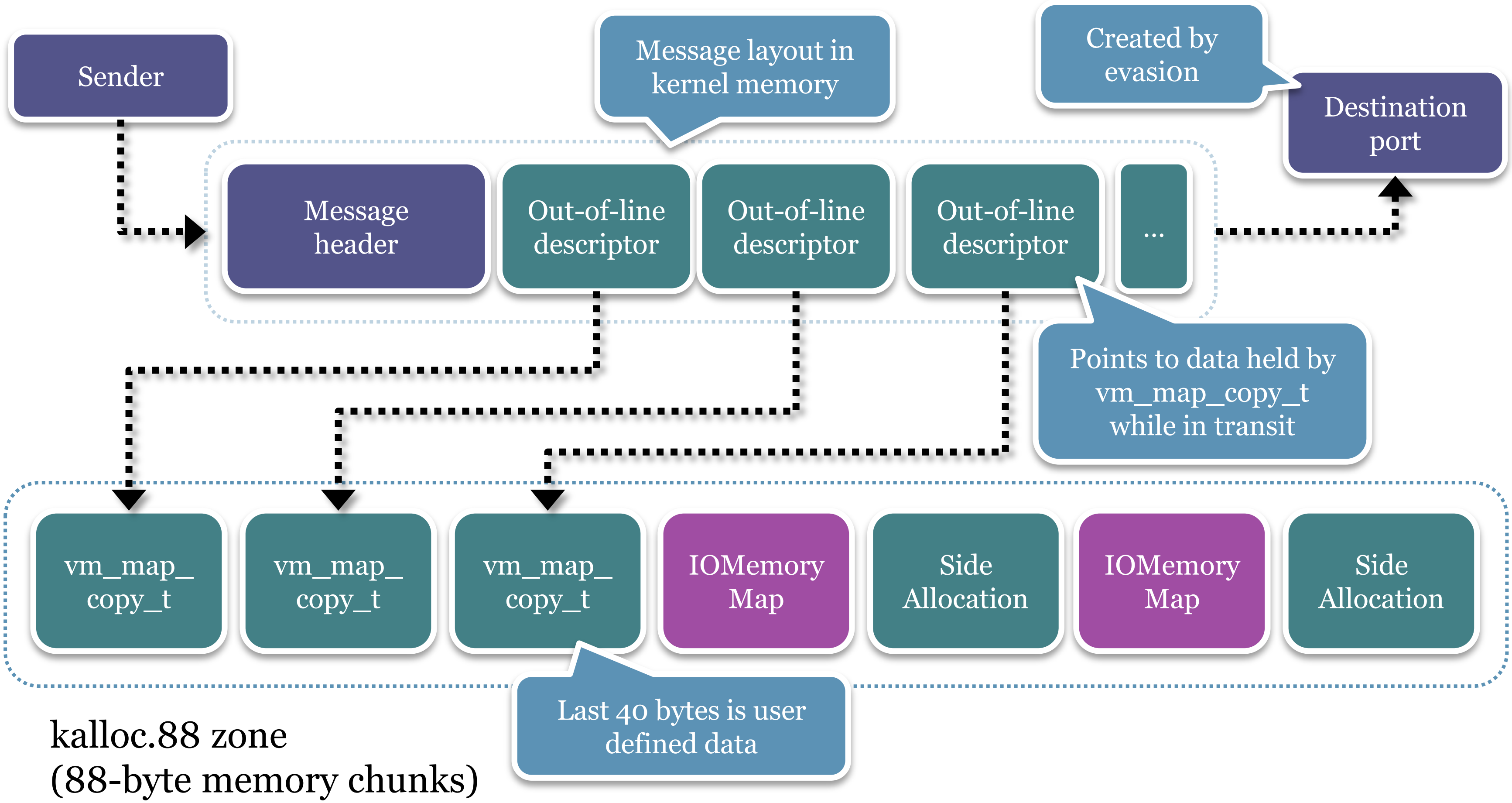
Defragmenting the Kernel Heap



Injecting User Controlled Data

- Mach message used to set the contents of the bordering free data
- Message holds 20 «out-of-line descriptors»
  - Allows arbitrary sized data to be passed between a sender and receiver
  - 40 bytes of user controlled data in each descriptor
- While in transit, ool descriptor data is internally wrapped by a «vm_map_copy_t» structure
  - kalloc(48 + 40 bytes data) -> kalloc.88 zone

Injecting User Controlled Data



Controlling the Program Counter

- evasion can now find its user controlled data in kernel memory
  - Relative offset from IOMemoryMap object
- Used to gain control of execution
  - Crafts a fake pipe object in user data
  - Provides its pointer to stallPipe()
  - Fully controls called function pointer and args (...)
- Needs to find a useful function to call
  - Heap is non-executable

Finding the Kernel Image Base

- Kernel address space is not entirely randomized
- ARM exception vectors located at a fixed address
  - 0xFFFF0000
- Can call the data abort handler directly to generate a user exception
- Allows retrieval of all the CPU registers at the time of exception

Offset	Handler
00h	Reset
04h	Undefined Instruction
08h	Supervisor Call (SVC)
0Ch	Prefetch Abort
10h	Data Abort
14h	(Reserved)
18h	Interrupt (IRQ)
1Ch	Fast Interrupt (FIQ)

ARM vector table
at 0xffff0000

Finding the Kernel Image Base

- evasi0n calls the data abort handler to record the address of the «faulting» instruction
  - Sets up an exception state identity handler
- Address used to reveal the base address of com.apple.iokit.IOUSBDeviceFamily
  - Located at a fixed offset from the kernel itself
- Retrieves the offset to the kernel image using OSKextCopyLoadedKextInfo()
  - Used to compute the kernel image base address

Arbitrary Read and Write

- Ultimate goal of any kernel exploit
- Allows necessary locations in memory to be patched
  - E.g. sandbox settings
- evasi0n is no exception
  - Needs to locate functions in memory
  - Needs to patch variables in memory

Arbitrary Kernel Memory Read

- Can also leak 4 bytes using exception technique
  - Controls the memory read into R1 («object Y»)
- Non-ideal method
  - Requires the heap data to be updated every time
  - Message must be received and re-sent
- Instead, finds a pointer to memmove()
  - Scans from the kernel code section base
  - Follows branching instructions
  - Looks for a specific bytecode signature

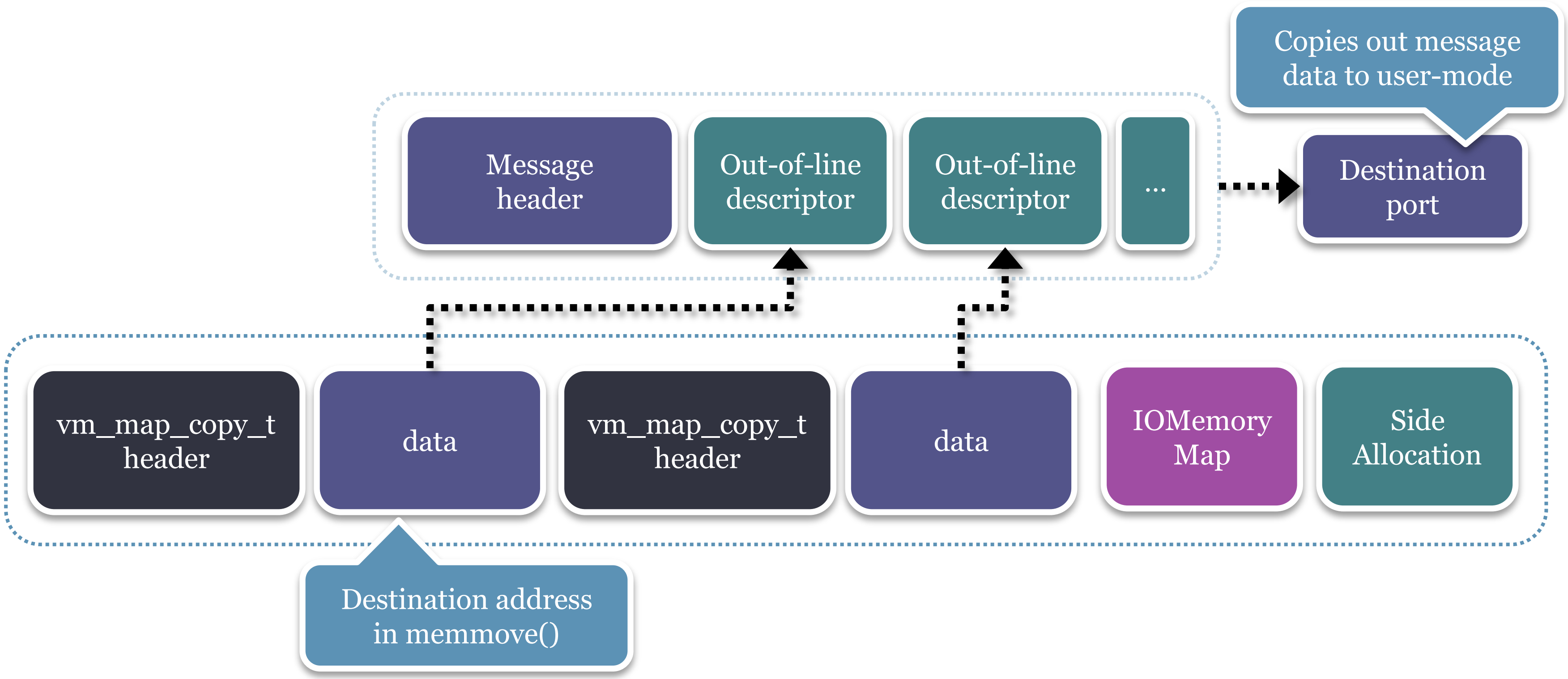
Arbitrary Kernel Memory Read

```
__TEXT:__text:80088744
__TEXT:__text:80088744                     ; =============== S U B R O U T I N E =======================================
__TEXT:__text:80088744
__TEXT:__text:80088744                     ; Attributes: bp-based frame
__TEXT:__text:80088744
__TEXT:__text:80088744                     ; void *memmove(void *, const void *, size_t)
__TEXT:__text:80088744                                     EXPORT _memmove
__TEXT:__text:80088744                     _memmove                                ; CODE XREF: sub_800047F4+8A↑p
__TEXT:__text:80088744                                                             ; sub_800047F4+E4↑p ...
__TEXT:__text:80088744 000 00 00 52 E3                     CMP             R2, #0  ; _memcpy
__TEXT:__text:80088748 000 01 00 50 11                     CMPNE           R0, R1
__TEXT:__text:8008874C 000 1E FF 2F 01                     BXEQ            LR
__TEXT:__text:80088750 000 B1 40 2D E9                     STMFD           SP!, {R0,R4,R5,R7,LR}
__TEXT:__text:80088754 014 0C 70 8D E2                     ADD             R7, SP, #0xC
__TEXT:__text:80088758 014 01 30 40 20                     SUBCS           R3, R0, R1
__TEXT:__text:8008875C 014 00 30 41 30                     SUBCC           R3, R1, R0
__TEXT:__text:80088760 014 02 00 53 E1                     CMP             R3, R2
__TEXT:__text:80088764 014 51 00 00 3A                     BCC             loc_800888B0
__TEXT:__text:80088768
```

Arbitrary Kernel Memory Read

- Uses memmove() to read memory back into the ool descriptor data buffer
  - Always pointed to by the first argument
  - memmove(objectZ, source, length)
  - Source and length is attacker controlled
- Can be copied out to user-mode by receiving the sent message
- Limited to 24 bytes
  - Copy starts 16 bytes into the buffer

Arbitrary Kernel Memory Read



Arbitrary Kernel Memory Read

- Different approach needed for reads > 24 bytes
- Corrupts a vm_map_copy_t structure in order to leak arbitrary sized data
  - A size larger than 24 bytes corrupts the next vm_map_copy_t structure
- Technique presented by Azimuth Security at Hack In the Box / Breakpoint last year
  - iOS 6 Kernel Security: A Hacker's Guide

Data Structure: vm_map_copy_t

Size of data

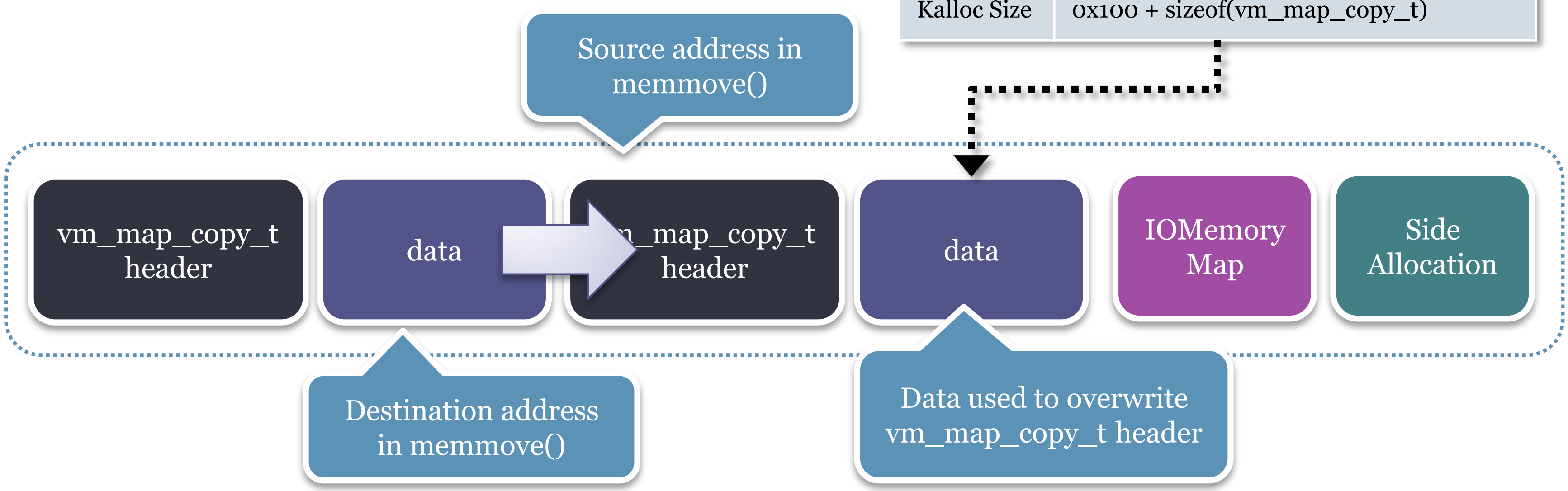
Pointer to data

Data always follows the header structure

Type	VM_MAP_COPY_KERNEL_BUFFER
Offset	0
Size	0x100
Kdata	<pointer>
Kalloc Size	0x100 + sizeof(vm_map_copy_t)
Data	AAAA.... (0x100 bytes)

Data Structure Corruption

Type	VM_MAP_COPY_KERNEL_BUFFER
Offset	0
Size	New size
Kdata	New address to copy out to user
Kalloc Size	0x100 + sizeof(vm_map_copy_t)



Arbitrary Kernel Memory Write

- Cannot use memmove() technique for patching
  - evasion does not fully control the destination pointer
- Instead, searches for an STR R1, [R2], BX LR instruction sequence in memory
  - Writes four bytes (R1) into the location pointed to by R2
  - First argument is irrelevant
- Used for subsequent kernel patches

Patching the Kernel

- Various patches made to the kernel
  - Disable mandatory code signing
  - Disable sandbox checks
  - Enable task_for_pid(0) -> kernel task
  - Enable RWX protection
  - Disable service (plist) signing
- Code pages are initially read/executable
  - Made writable by patching the phyiscal memory map (kernel_pmap)

Conclusion

Attacking the iOS Kernel

Vulnerability Fix

- Apple has addressed the IOUSBDeviceFamily vulnerability in iOS 6.1.3
  - Vulnerable APIs have been disabled
- Also addresses the ARM exception vector information leak
  - Checks the caller of the data abort handler
- Still possible to leak the address of IOMemoryMap objects

Closing Notes

- KASLR and address space separation greatly complicate kernel exploitation
    - iOS 5 was a walk in the park ☺
- Address space information leaks are now paramount to the attacker
    - Data injection may also be necessary
- Sandboxing reduces attack surface
    - Vulnerability can only be triggered by a less restrictive sandbox (i.e. not from MobileSafari)

Thanks!

- Questions?
- http://blog.azimuthsecurity.com/2013/02/from-usr-to-svc-dissecting-evasion.html
- E-mail
  - tm@azimuthsecurity.com
  - kernelpool@gmail.com