iOS 6 Kernel Security: A Hacker's Guide

by Mark Dowd and Tarjei Mandt
mdowd@azimuthsecurity.com
tm@azimuthsecurity.com

azimuth SECURITY

Introduction

- iOS 6 recently released
- Large focus on security improvements – particularly kernel hardening
- Primarily targets strategies employed in “jailbreaks”
- This talk provides an overview of the new kernel-based mitigations
- Explores new techniques for attacking iOS 6

Topics Covered

- Part 1 – Defense
  - Heap Hardening Strategies
  - Stack Cookies
  - Information Leaking Mitigations
  - Address Space Layout Randomization (ASLR)
  - User/Kernel address space hardening
- Part 2 – Offense
  - Information Leaking
  - Heap Strategies

Randomization Algorithm

- First, a word on randomness…
- Used to derive random numbers for stack cookie, heap cookies, kernel map ASLR, and pointer obfuscation
- Random seed generated (or retrieved) during boot loading (iBoot)
- Combined with current time to get random value

Randomization Algorithm

```
unsigned long long
GetRandomValue(unsigned long long time, unsigned long long seed)
{
    unsigned int time_low       = time & 0xFFFFFFFF;
    unsigned int time_high      = (time >> 32) & 0xFFFFFFFF;
    unsigned int result_low;
    unsigned int result_high;
    unsigned int tmp;

    // calculate low DWORD of output

    tmp = (time_low & 0xFF) << 8;
    result_low = (time_low ^ tmp) ^ (time_low << 16);

    // calculate high DWORD of output

    tmp = (seed & 0xFF) << 16;
    result_high = tmp ^ (result_low ^ time_high);

    tmp = (result_low >> 8) ^ 0xFF;
    result_high = ROTATE_RIGHT(result_high, tmp);

    // done

    return (result_high << 32) | result_low;
}
```

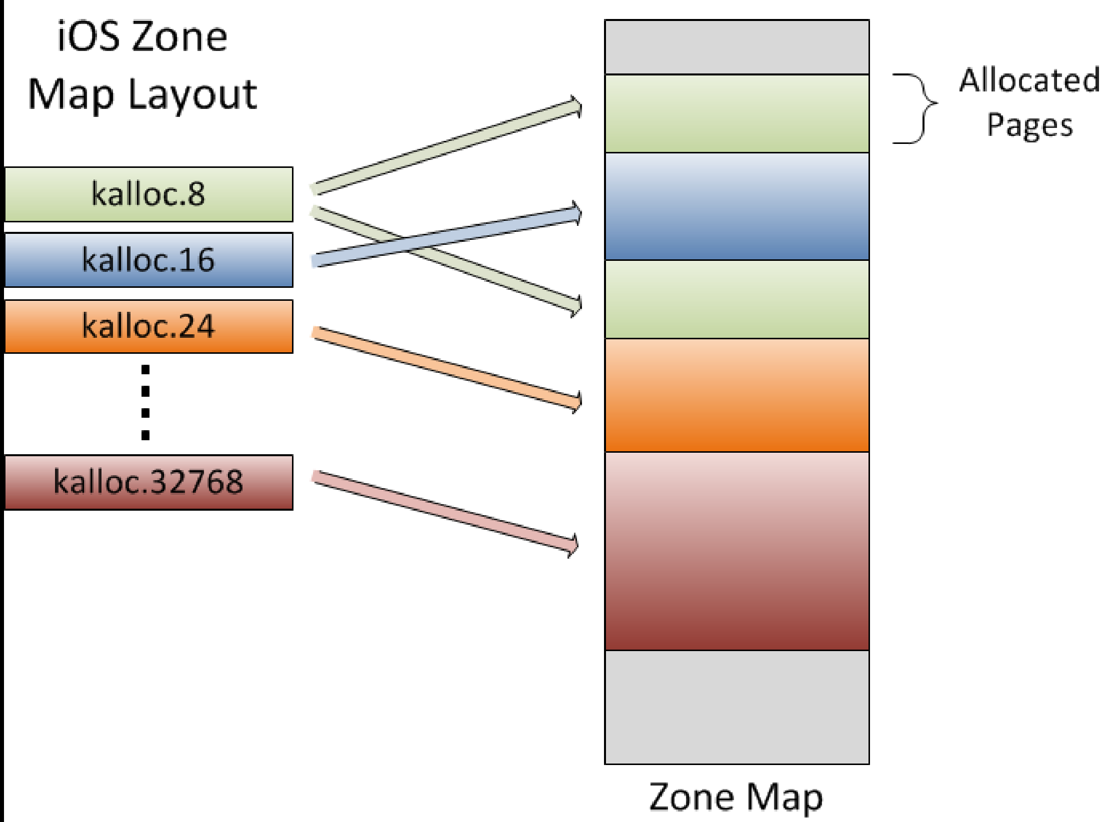
Heap Hardening

- Heap has been hardened to prevent well-known attack strategies
- Three mitigations put in place
  - Pointer validation
  - Block poisoning
  - Freelist integrity verification
- Specific to the zone allocator (zalloc( ), used by kalloc( ), MALLOC( ), MALLOC_ZONE( ))

Heap Hardening - Recap

- Quick recap of old exploitation techniques required
  - Covered in the past extensively by Stefan Esser, Nemo, probably others
- Zone allocations divided in to fixed-size zones (kalloc.8, kalloc.16, ... kalloc.32768)
  - Specialized zones also utilized for specific tasks (eg. Pmap_zone, vm_map_copy_zone, etc)
- Zone allocates more pages on demand

Heap Hardening - Recap

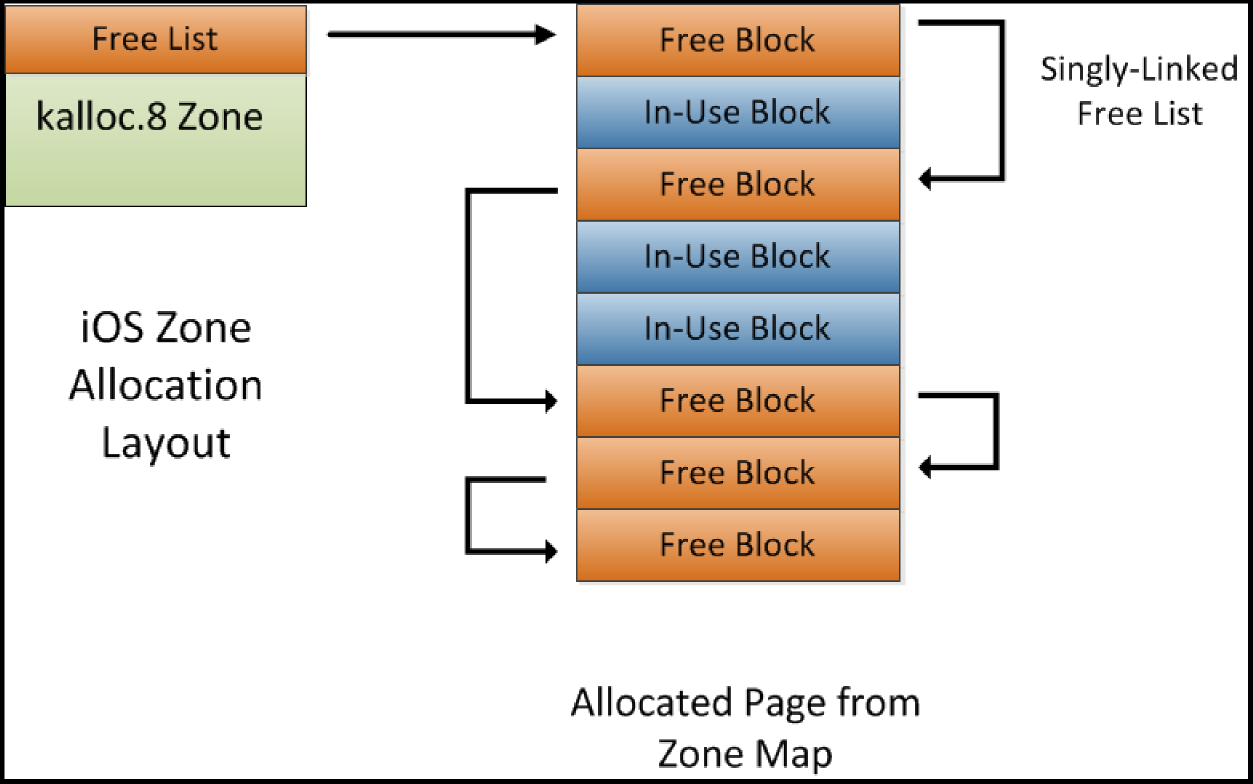


azimuth SECURITY

Heap Hardening - Recap

- Zone allocates blocks of pages on demand
  - Divides memory in to element-size blocks
  - All blocks initially added to zone's free list
- Zone free list maintained as singly linked list
  - First DWORD of free block overwritten with "next" pointer when it is freed
- Allocations simply remove elements from the free list

Heap Hardening - Recap



Heap Hardening - Recap

- Previous exploitation techniques rely on overwriting free list pointers in free blocks
  - Future allocation can return arbitrary memory block
- Typical strategy: Add a pointer to sysent
  - Add new system call
  - Invoke new system call
  - Profit

Heap Hardening – Pointer Validation

- Goal: Prevent invalid pointers being entered in to kalloc( ) zone's freelist
- Additional checks performed on pointers passed to zfree( )
  - Also performed as part of validation on pointers in freelist during allocation (zalloc( ))

Heap Hardening – Pointer Validation

- Pointer verified to be in kernel memory (0x80000000 < ptr < 0xFFFEFFFF)
- If allows_foreign is set in zone, no more validation performed
  - Currently event_zone, vm_map_entry_reserved_zone, vm_page_zone
- If pointer is within kernel image, allow (??)
- Otherwise, ensure pointer is within zone_map

Heap Hardening – Block Poisoning

- Goal: Prevent UAF-style attacks
- Strategy involves filling blocks with sentinel value (0xdeadbeef) when being freed
  - Performed by add_to_zone( ) called from zfree( )
- Only performed on selected blocks
  - Block sizes smaller than cache line size of processor (e.g. 32 bytes on A5/A5X devices)
  - Can override with “-zp”, “-no-zp”, “zp-factor” boot parameters

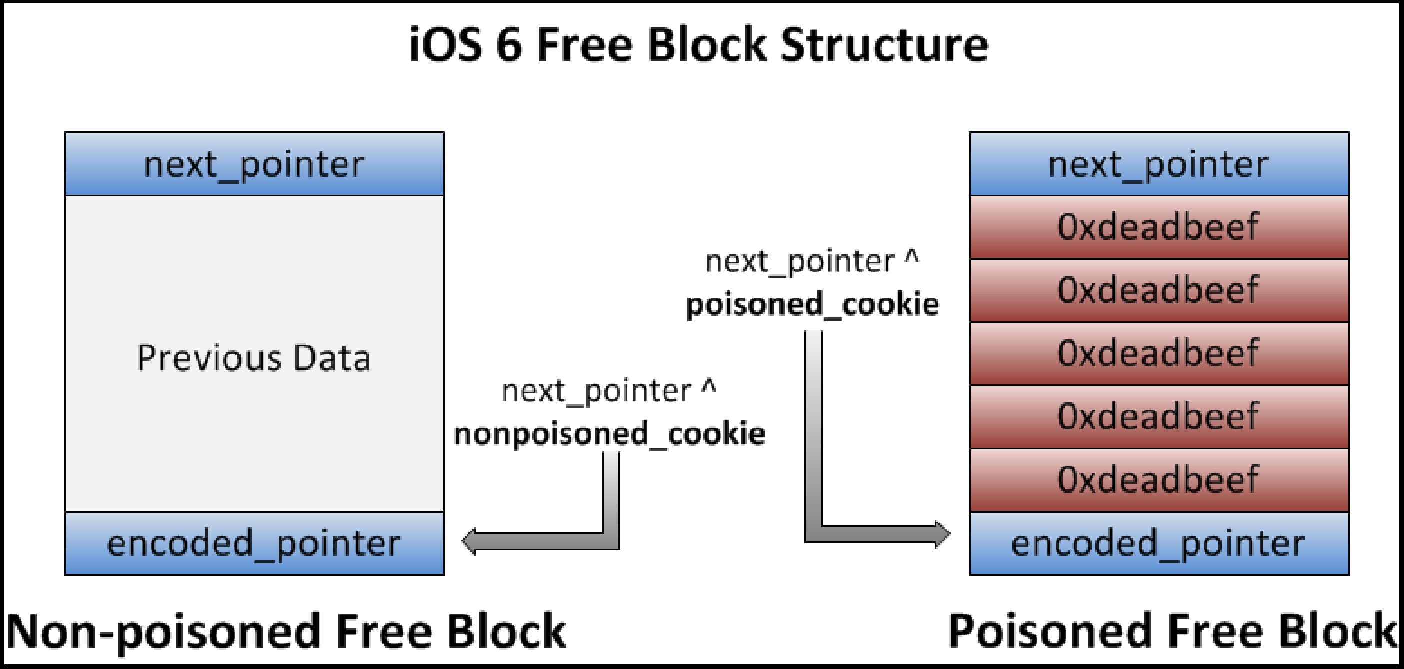
Heap Hardening – Freelists

- Goal: Prevent heap overwrites from being exploitable
- Two random values generated at boot time (zone_bootstrap( ))
  - 32-bit cookie for "poisoned blocks"
  - 31-bit cookie for "non-poisoned blocks"
    - Low bit is clear
- Values serve as validation cookies

Heap Hardening – Freelists

- Freelist pointers at the top of a free block are now validated by zalloc( )
  - Work performed by alloc_from_zone( )
- Encoded next pointer placed at end of block
  - XOR'd with poisoned_cookie or nonpoisoned_cookie

Heap Hardening – Freelists



Heap Hardening – Freelists

- zalloc( ) ensures next_pointer matches encoded pointer at end of block
  - Tries both cookies
  - If poisoned cookie matches, check whole block for modification of sentinel (0xdeadbeef) values
  - Cause kernel panic if either check fails
- Next pointer and cookie replaced by 0xdeadbeef when allocated
  - Possible information leak protection

Heap Hardening – Primitives

- OSUnserializeXML( ) could previously be used to perform kernel heap feng shui
  - Technique presented by Stefan Esser in «iOS Kernel Heap Armageddon» at SyScan 2012
- Allowed precise allocation and freeing of kalloc zone data
- Also possible to force persistent allocations by wrapping the reference count

Heap Hardening - Primitives

```
<plist version="1.0">
<dict>
  <key>AAAA</key>
  <array ID="1" CMT="IsNeverFreedTooManyReferences">...</array>
  <key>REFS</key>
<array>
<x IDREF="1"/><x IDREF="1"/><x IDREF="1"/><x IDREF="1"/>
<x IDREF="1"/><x IDREF="1"/><x IDREF="1"/><x IDREF="1"/>
<x IDREF="1"/><x IDREF="1"/><x IDREF="1"/><x IDREF="1"/>
...
<x IDREF="1"/><x IDREF="1"/><x IDREF="1"/><x IDREF="1"/>
</array>
</dict>
</plist>
```

Heap Hardening - Primitives

- Duplicate dictionary keys no longer result in freeing of the original key/value
- Dictionary entries can no longer be pinned to memory using multiple references
- In both cases, the plist dictionary is considered invalid

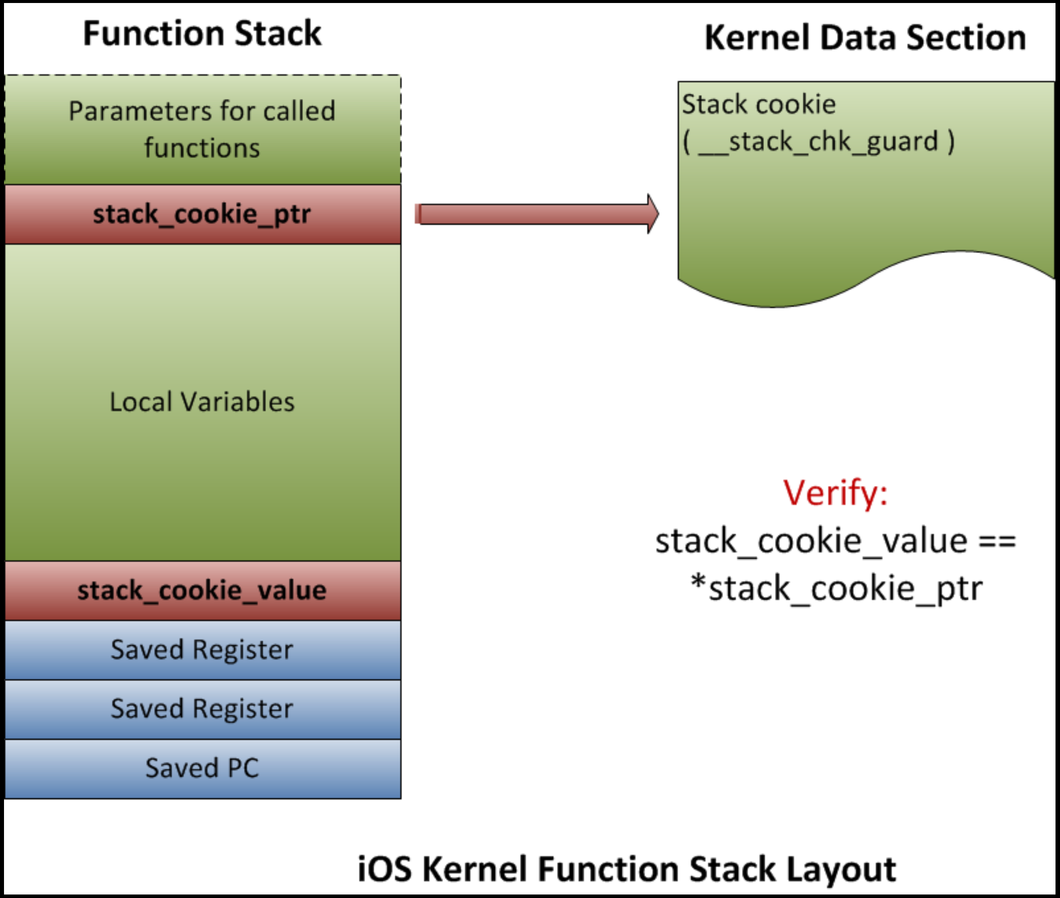
Stack Cookies

- Goal: Prevent stack overflow exploitation
- Only applied to functions with structures/buffers
- Random value generated during early kernel initialization (arm_init( ))
- 24-bit random value
  - 32-bit value really, but 2nd byte zeroed out
  - Presumably string copy prevention

Stack Cookies

- Generated stack cookie placed directly after saved registers at bottom of stack frame
- Pointer to cookie saved at top of stack frame
  - Or in a register if convenient
  - Space above stack cookie pointer used for called functions if necessary

Stack Cookies



Stack Cookies

- Function epilog verifies saved stack cookie
  - Generated value found by following saved pointer
- Verification failure results in kernel panic

```
__TEXT:__text:800051FC __epilog                                   ; CODE XREF: sub_80004F98+2B4↓j
__TEXT:__text:800051FC                                            ; sub_80004F98+486↓j
__TEXT:__text:800051FC                 LDR     R0, [SP,#0x2CC+stack_cookie_ptr]
__TEXT:__text:800051FE                 LDR     R0, [R0]
__TEXT:__text:80005200                 LDR     R1, [SP,#0x2CC+stack_cookie]
__TEXT:__text:80005202                 CMP     R0, R1          ; check stack cookie validity
__TEXT:__text:80005204                 ITTTT EQ
__TEXT:__text:80005206                 MOVEQ   R0, R4
__TEXT:__text:80005208                 ADDEQ.W SP, SP, #0x2B4
__TEXT:__text:8000520C                 POPEQ.W {R8,R10,R11}
__TEXT:__text:80005210                 POPEQ   {R4-R7,PC}
__TEXT:__text:80005212                 BL      ___stack_chk_fail
```

azimuth SECURITY

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( )
  - Stefan Esser described using this for heap “feng shui” (https://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_Slides.pdf)
- APIs now require PE_i_can_has_debugger() access

Information Leaking Mitigations

- Several APIs disclose kernel object pointers
  - mach_port_kobject( )
  - mach_port_space_info( )
  - vm_region_recurse( ) / vm_map_region_recurse( )
  - vm_map_page_info( )
  - proc_info (PROC_PIDREGIONINFO, PROC_PIDREGIONPATHINFO, PROC_PIDFDPIPEINFO, PROC_PIDFDSOCKETINFO, PROC_PIDFILEPORTSOCKETINFO)
  - fstat( ) (when querying pipes)
  - sysctl(net.inet.*.pcblist)

Information Leaking Mitigations

- Need these APIs for lots of reasons
  - Often, underlying APIs rather than exposed ones listed previously
- Strategy: Obfuscate pointers
  - Generate 31 bit random value at boot time
    - lowest bit always 1
  - Add random value to real pointer

Information Leaking Mitigations

```
int
fill_pipeinfo(struct pipe * cpipe, struct pipe_info * pinfo)
{

        ... code ...

        pinfo->pipe_handle = (uint64_t)((uintptr_t)cpipe);
        pinfo->pipe_peerhandle = (uint64_t)((uintptr_t)(cpipe->pipe_peer));
        pinfo->pipe_status = cpipe->pipe_state;

        PIPE_UNLOCK(cpipe);

        return (0);
}
```

Information Leaking Mitigations

```
__TEXT:__text:801EC942                 MOV             R1, (heap_random_value_ptr - 0x801EC94E) ; heap_random_value_ptr
__TEXT:__text:801EC94A                 ADD             R1, PC ; heap_random_value_ptr
__TEXT:__text:801EC94C                 LDR             R1, [R1]
__TEXT:__text:801EC94E                 STR             R3, [R5,#0x20]
__TEXT:__text:801EC950                 ASRS            R6, R3, #0x1F
__TEXT:__text:801EC952                 STR             R6, [R5,#0x24]
__TEXT:__text:801EC954                 STMIA.W         R4, {R0,R2,R3,R6}
__TEXT:__text:801EC958                 ADD.W           R4, R5, #0x38
__TEXT:__text:801EC95C                 STMIA.W         R4, {R0,R2,R3,R6}
__TEXT:__text:801EC960                 MOVS            R0, #0
__TEXT:__text:801EC962                 LDR             R2, [R1] ; R2 = heap_random_value
__TEXT:__text:801EC964                 ADD.W           R6, R2, R8 ; R6 = (unsigned long) cpipe + heap_random_value;
__TEXT:__text:801EC968                 STR.W           R6, [R5,#0x88] ; set pipe_handle to cpipe + heap_random_value
__TEXT:__text:801EC96C                 STR.W           R0, [R5,#0x8C] ; set pipe_handle_peer to NULL
__TEXT:__text:801EC970                 LDR.W           R2, [R8,#0x30]
```

Information Leaking Mitigations

- Other APIs disclose pointers unnecessarily
  - Zero them out
- Used to mitigate some leaks via sysctl
  - Notably, known proc structure infoleak

Kernel ASLR

- Goal: Prevent attacker's from modifying/utilizing data at known (fixed) addresses
- Strategy is two-fold
  - Randomize kernel image base
  - Randomize base of kernel_map (sort of)

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 slide is 0, static offset of 0x21000000 is used

Kernel ASLR – Kernel Image

```
ROM:5FF19CF8
ROM:5FF19CF8 loc_5FF19CF8                            ; CODE XREF: Image3_RelocateImage+3A↑j
ROM:5FF19CF8                 ADD     R0, SP, #0x3C+slide ; address of output buffer
ROM:5FF19CFA                 MOVS    R4, #0
ROM:5FF19CFC                 MOVS    R1, #1          ; number of random bytes required
ROM:5FF19CFE                 STRB.W  R4, [SP,#0x3C+slide]
ROM:5FF19D02                 MOV     R5, R9
ROM:5FF19D04                 BL      iBoot_GetRandomBytes
ROM:5FF19D08                 CBZ     R0, loc_5FF19D10
ROM:5FF19D0A                 LDR     R0, =0x5FF44EB8
ROM:5FF19D0C                 STR     R4, [R0]        ; failed to generate random bytes, just make slide 0
ROM:5FF19D0E                 B       loc_5FF19D28
ROM:5FF19D10 ; ---------------------------------------------------------------------------
ROM:5FF19D10
ROM:5FF19D10 loc_5FF19D10                            ; CODE XREF: Image3_RelocateImage+54↑j
ROM:5FF19D10                 LDR     R2, =0x5FF44EB8
ROM:5FF19D12                 MOV.W   R0, #0x21000000
ROM:5FF19D16                 LDRB.W  R1, [SP,#0x3C+slide]
ROM:5FF19D1A                 CMP     R1, #0
ROM:5FF19D1C                 ITT NE
ROM:5FF19D1E                 MOVNE.W R0, #0x1000000
ROM:5FF19D22                 ADDNE.W R0, R0, R1,LSL#21 ; slide = (slide_byte << 21) + 0x00100000
ROM:5FF19D26                 STR     R0, [R2]        ; store slide value for later use
```

Kernel ASLR – Kernel Image

- Calculated value added to kernel preferred base later on
- Result:
  - Kernel can be rebased at 1 of 256 possible locations
  - Base addresses are 2MB apart
    - Example: 0x81200000, 0x81400000, … 0xA1000000
- Adjusted base passed to kernel in boot args structure (offset 0x04)

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

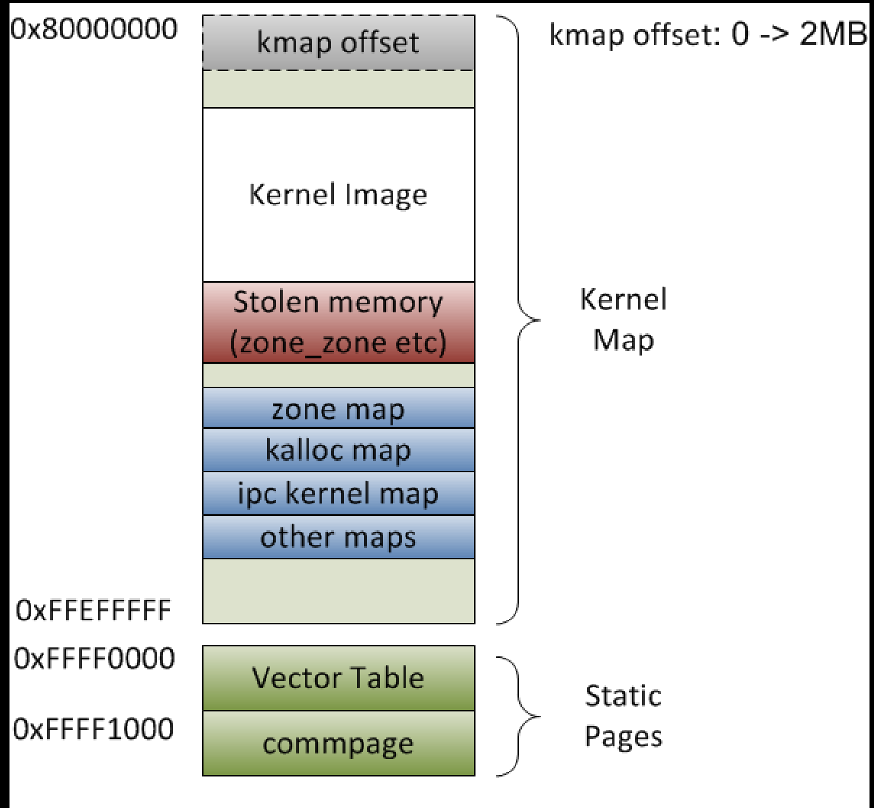
Kernel ASLR – Kernel Map

- Strategy involves randomizing the base of kernel_map
  - Random 9-bit value generated right after kmem_init( ) (which establishes kernel_map)
  - Multiplied by page size
  - Resulting value used as size for initial kernel_map allocation
  - 9 bits = 512 different allocation size possibilities

Kernel ASLR – Kernel Map

- Future kernel_map (including submap) allocations pushed forward by random amount
  - Allocation silently removed after first garbage collection (and reused)
- Behavior can be overridden with “kmapoff” boot parameter

Kernel ASLR – Kernel Map



iOS 6 Kernel Memory Layout

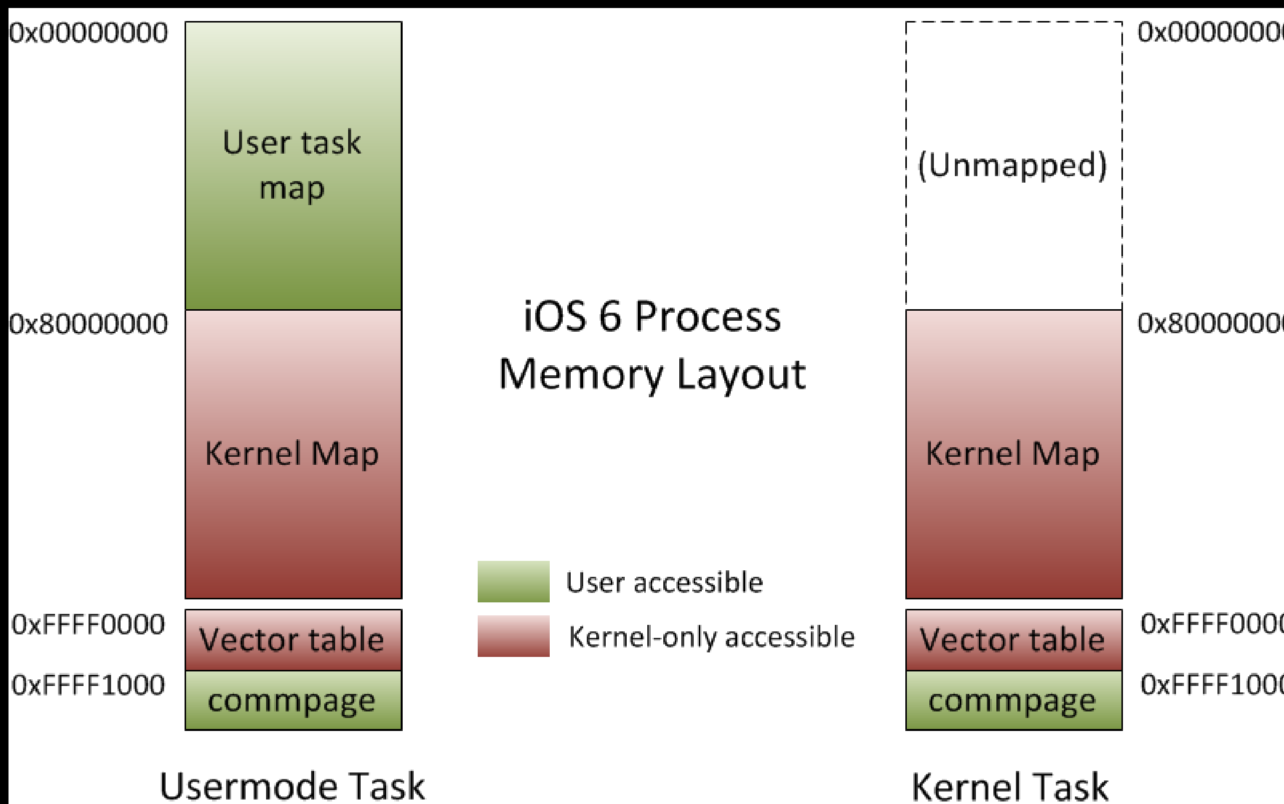
Kernel Address Space Protection

- Goal: Prevent NULL/offset-to-NULL dereference vulnerabilities
- Previously, kernel mapped in to user-mode address space
- NULL-dereferences were prevented by forcing binaries to have __PAGE_ZERO section
  - Does not prevent offset-to-NULL problems

Kernel Address Space Protection

- kernel_task now has its own address space while executing
  - Transitioned to with interrupt handlers
  - Switched between during copyin( ) / copyout( )
- User-mode pages therefore not accessible while executing in kernel mode
- Impossible to accidentally access them

Kernel Address Space Protection



Kernel Address Space Protection

- BUG – iOS 5 and earlier had very poor user/kernel validation in copyin( ) / copyout( )
  - Only validation: usermode pointer < 0x80000000
  - Length not validated
- Pointer + length can be > 0x80000000 (!)
  - Can potentially read/write to kernel memory
- Limitation: Device must have > 512M to map 0x7FFFF000
  - iPad 3 / iPhone 5

Kernel Address Space Protection

```
              EXPORT _copyout
_copyout                                          ; CODE XREF: sub_8000D64C+C↑p
                                                  ; __TEXT:__text:800132F6↑p ...
              CMP             R2, #0
              MOVEQ           R0, #0
              BXEQ            LR
              CMP             R1, #0x80000000
              BCS             _error
              STMFD           SP!, {R4}
              ADR             R3, fault_handler_routine
              MRC             p15, 0, R12,c13,c0, 4
              LDR             R4, [R12,#0x220]
              STR             R3, [R12,#0x220]
              CMP             R2, #0x10
              BLT             bytewise_copy
              ORR             R3, R0, R1
              TST             R3, #3
              BNE             bytewise_copy
              SUB             R2, R2, #8
```

Kernel Address Space Protection

- iOS 6 added security checks
  - Integer overflow/signedness checks
  - Conservative maximum length
  - Pointer + length < 0x80000000
- iOS 6 still vulnerable!
  - If copy length <= 0x1000, pointer + length check not performed
  - Can read/write to first page of kernel memory

Kernel Address Space Protection

```
_copyout                                        ; CODE XREF: sub_8000E490+C↑p
                                                ; __mach_trap_vm_allocate+4A↑p ...

var_8           = -8

                CMP             R2, #0
                MOVEQ           R0, #0
                BXEQ            LR
                CMP             R1, #0x80000000
                BCS             loc_80088278
                CMP             R2, #0x1000
                BLS             do_copy
                STMFD           SP!, {R4-R7,LR}
                MOV             R4, R0
                MOV             R5, R1
                MOV             R6, R2
                ADD             R7, SP, #0x14+var_8
                BLX             copy_validate
                CMP             R0, #0
                LDMNEFD         SP!, {R4-R7,PC}
                MOV             R0, R4
                MOV             R1, R5
                MOV             R2, R6
                LDMFD           SP!, {R4-R7,LR}
```

Kernel Address Space Protection

- Is anything in the first page of memory?
  - Initially contains kmap offset allocation, but that is removed after first garbage collection
  - Some things allocate to kernel map directly
    - HFS
    - kalloc() blocks of >= 256k
- Create a pipe, specify buffers > 0x7FFFF000
- Bonus: If memory is not mapped, kernel will not panic (safely return EFAULT)

Kernel Address Space Protection

- Memory is no longer RWX
  - Kernel code cannot be directly patched
  - Heap is non-executable
  - Stack is non-executable

Kernel Attacks: Overview

- Protections kill most of the known attack strategies
  - Syscall table overwrites
  - Patching kernel code
  - Attacking key data structures (randomized locations)
- Need something new!

Kernel Attacks: Overview

- Generally, exploit will require information leaking followed by corruption
- Corruption primitives dictate strategy
  - Write in to adjacent buffer (overflow)
  - Write to relative location from buffer
  - Write to arbitrary location
- Different types of primitives will be considered separately

Kernel Attacks: KASLR

- Leaking the kernel base is really useful
- Kext_request( ) allows applications to request information about kernel modules
  - Divided into active and passive operations
- Active operations (load, unload, start, stop, etc.) require privileged (root) access
  - Secure kernels (i.e. iOS) remove ability to load kernel extensions

Kernel Attacks: KASLR

- Passive operations were originally unrestricted in < iOS 6
  - Allowed unprivileged users to query kernel and module base addresses

```
    result = kOSKextReturnNotPrivileged;
    if (hostPriv == HOST_PRIV_NULL) {
        if (!predicate->isEqualTo(kKextRequestPredicateGetLoaded) &&
            !predicate->isEqualTo(kKextRequestPredicateGetKernelImage) &&

!predicate->isEqualTo(kKextRequestPredicateGetKernelLoadAddress)) {

            goto finish;
        }
    }
```

Kernel Attacks: KASLR

- iOS 6 inadvertently removed some limitations
  - Only load address requests disallowed

```
    result = kOSKextReturnNotPrivileged;
    if (hostPriv == HOST_PRIV_NULL) {
        if (sPrelinkBoot) {
            hideTheSlide = true;

            /* must be root to use these kext requests */
            if
(predicate->isEqualTo(kKextRequestPredicateGetKernelLoadAddress) ) {
                OSKextLog(/* kext */ NULL,
                          kOSKextLogErrorLevel |
                          kOSKextLogIPCFlag,
                          "Access Failure - must be root user.");
                goto finish;
            }
        }
    }
```

Kernel Attacks: KASLR

- We can use kKextRequestPredicateGetLoaded
  - Returns load addresses and mach-o header dumps (base64 encoded)
  - Load address / Mach-O segment headers are obscured to hide ASLR slide
  - Mach-O section headers are not!
  - Reveals virtual addresses of loaded kernel sections

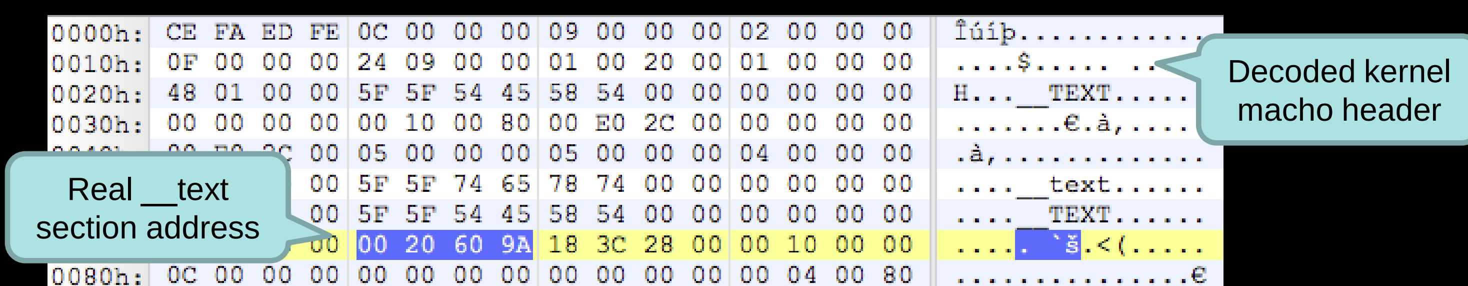
Kernel Attacks: KASLR

Request

<dict><key>Kext Request Predicate</key><string>Get Loaded Kext Info</string></dict>

Response

<dict ID="0"><key>__kernel__</key><dict
ID="1"><key>OSBundleMachOHeaders</key><data
ID="2">zvrt/gwAAAAJAAAAAgA…AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMhQ
CIAAAAAAJgAAABAAAAAwPDIAwEUAAA==</data>
…
<key>OSBundleLoadAddress</key><integer size="64" ID="9">0x80001000</integer>



Kernel Attacks: Heap Corruption

- Standard heap overflow tricks no longer work
  - Overwriting freelist pointer results in validation step failing
- Exploitation requires new strategies
  - Information leak to find heap address/cookies
  - Control structure manipulation
- Depends on corruption primitives

Kernel Attacks: Heap Overflows

- Overflowing metadata is useful
  - Various control structures can be targeted instead
  - Requires some heap grooming (may or may not be difficult depending on block size)
- Various heap attacking primitives can be combined to gain code execution

Kernel Attacks: Heap Overflows

- Introducing vm_map_copy_t

```
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;
};
```

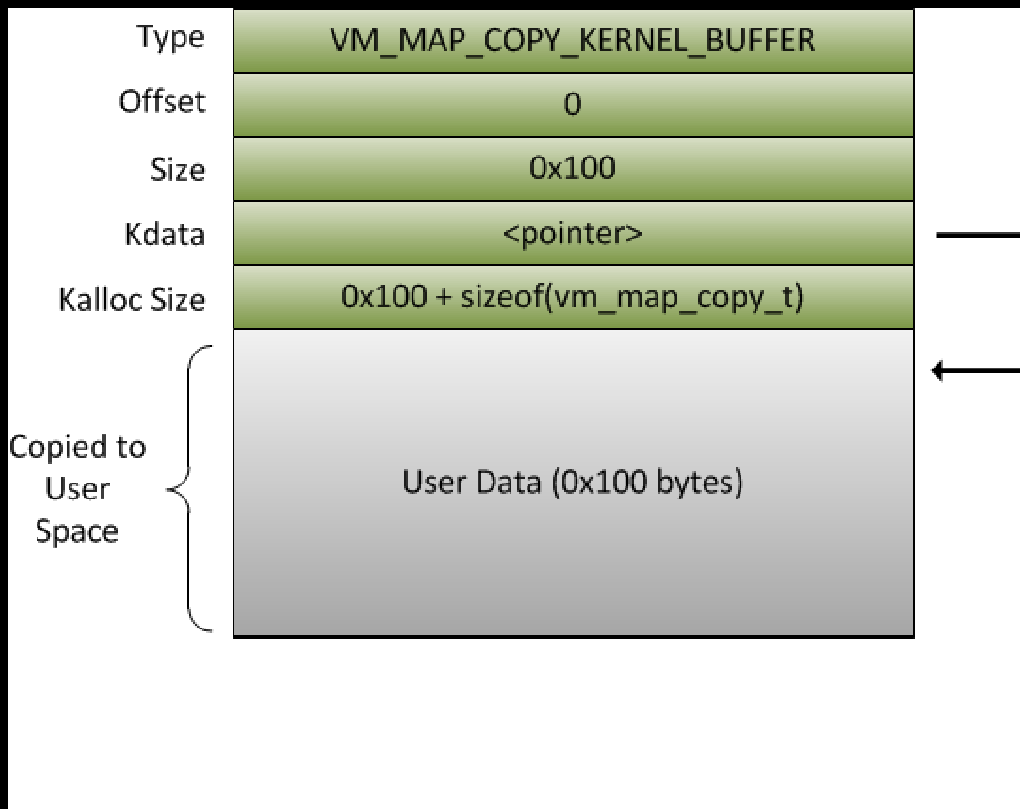
Kernel Attacks: Heap Overflows

- Kernel buffers allocated by vm_map_copyin() if size < 4096
- Creating them is easy
  - Send messages to a mach port with ool_descriptors in them
  - They are persistent until the message is received
- Corrupting these structures are useful for information leaking and exploitation

Kernel Attacks: Heap Overflows

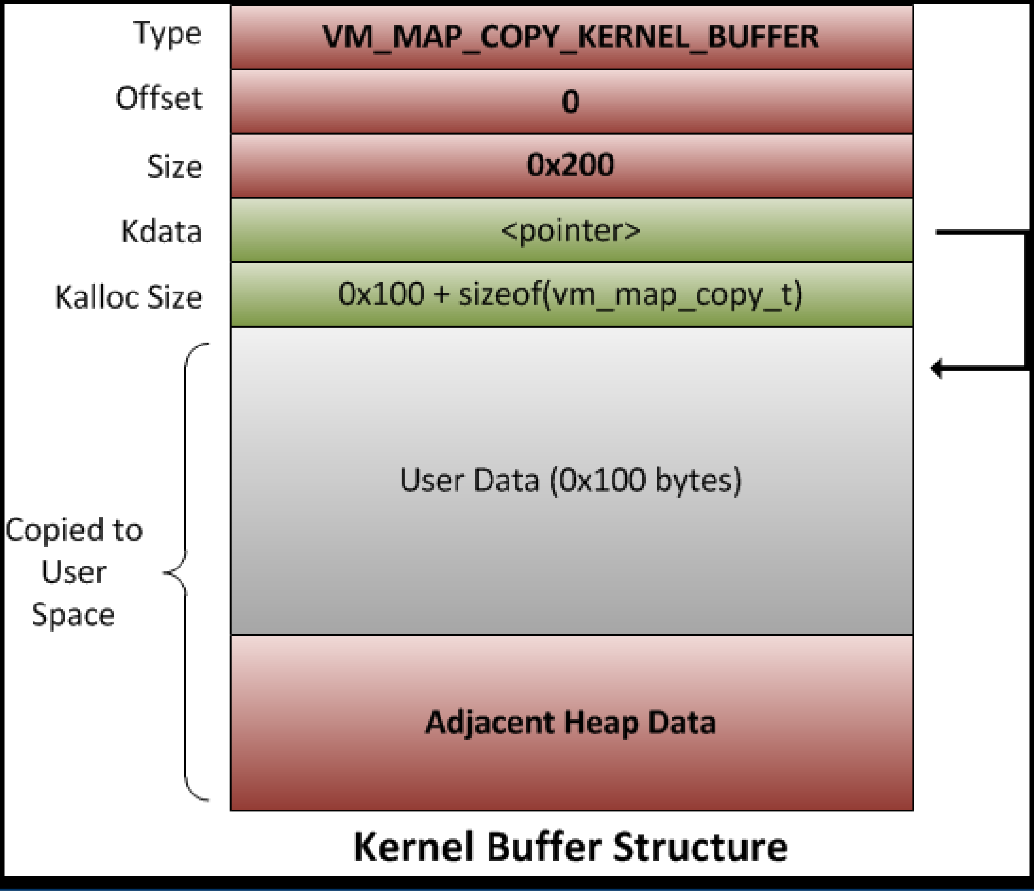
- Primitive 1: Adjacent Disclosure
  - Overwrite size parameter of vm_map_copy_t
  - Receive the message corresponding to the map
  - Returns memory past the end of your allocated buffer
- Bonus: Overwritten size is not used in kfree()
  - No side effects

Kernel Attacks: Heap Overflows



Kernel Buffer Structure

Kernel Attacks: Heap Overflows



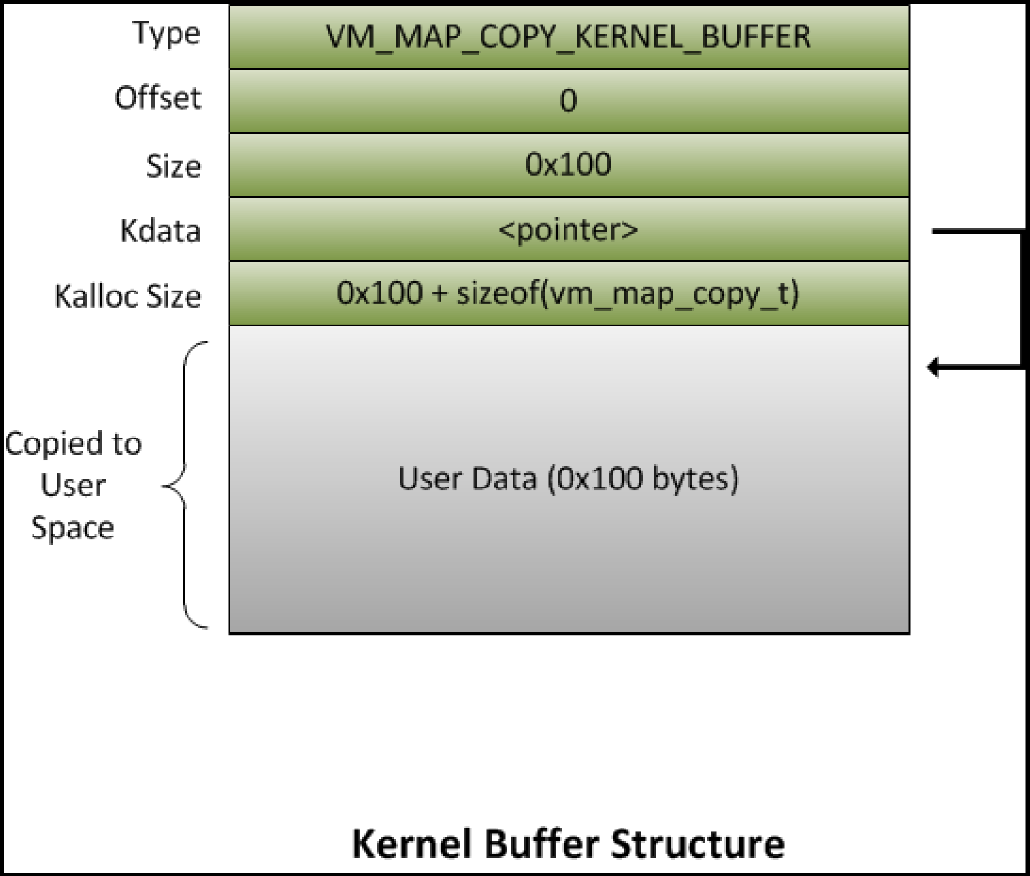
Kernel Attacks: Heap Overflows

- Primitive 2: Arbitrary Memory Disclosure
  - Overwrite size and pointer of adjacent vm_map_copy_t
  - Receive message, read arbitrary memory from kernel
- No side effects
  - Data pointer (cpy_kdata) is never passed to kfree() (the vm_map_copy_t is)
  - Leave kalloc_size alone!

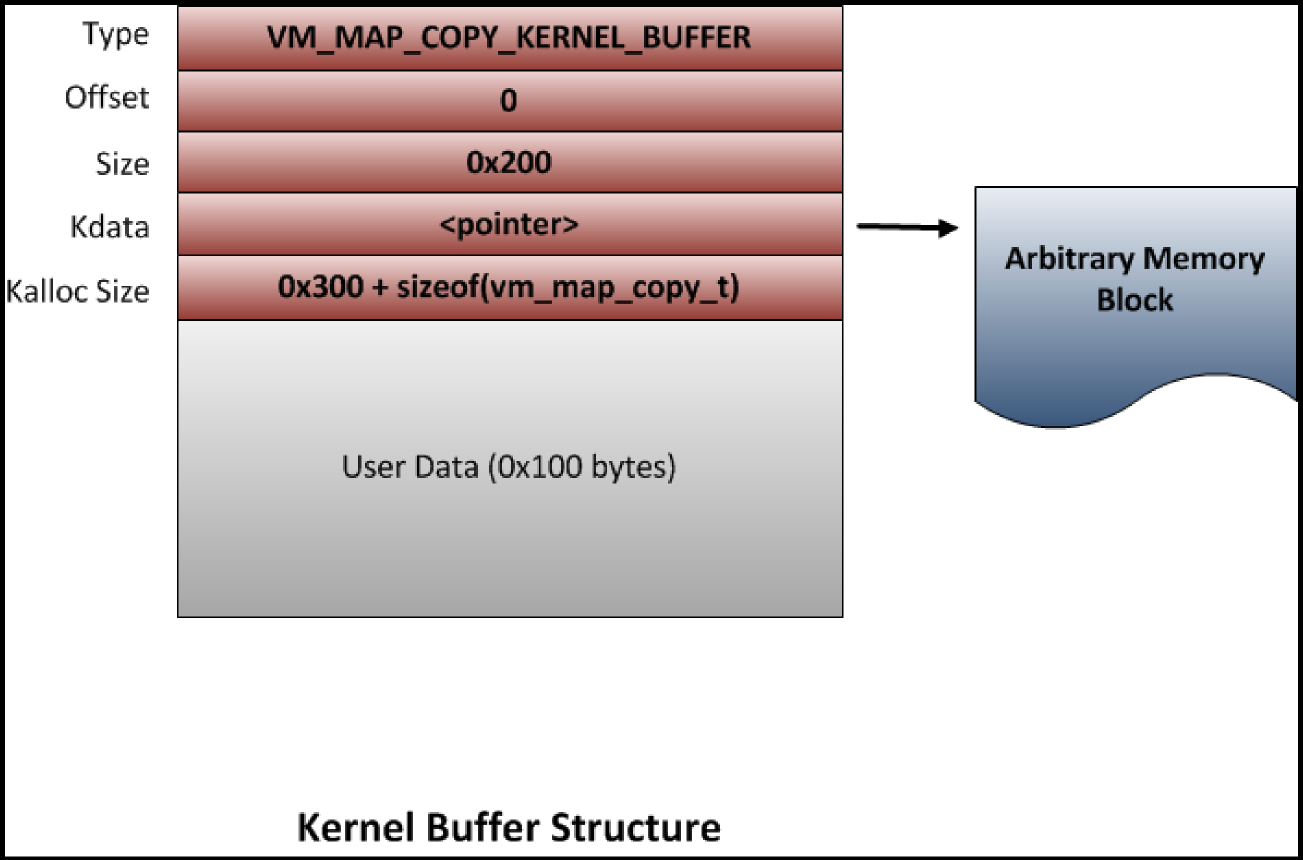
Kernel Attacks: Heap Overflows

- Primitive 3: Extended Overflow
  - Overwrite kalloc_size with larger value
  - Passed to kfree() – block entered in to wrong zone (eg. kalloc.256 instead of kalloc.128)
  - Allocate block from poisoned zone
  - Profit

Kernel Attacks: Heap Overflows



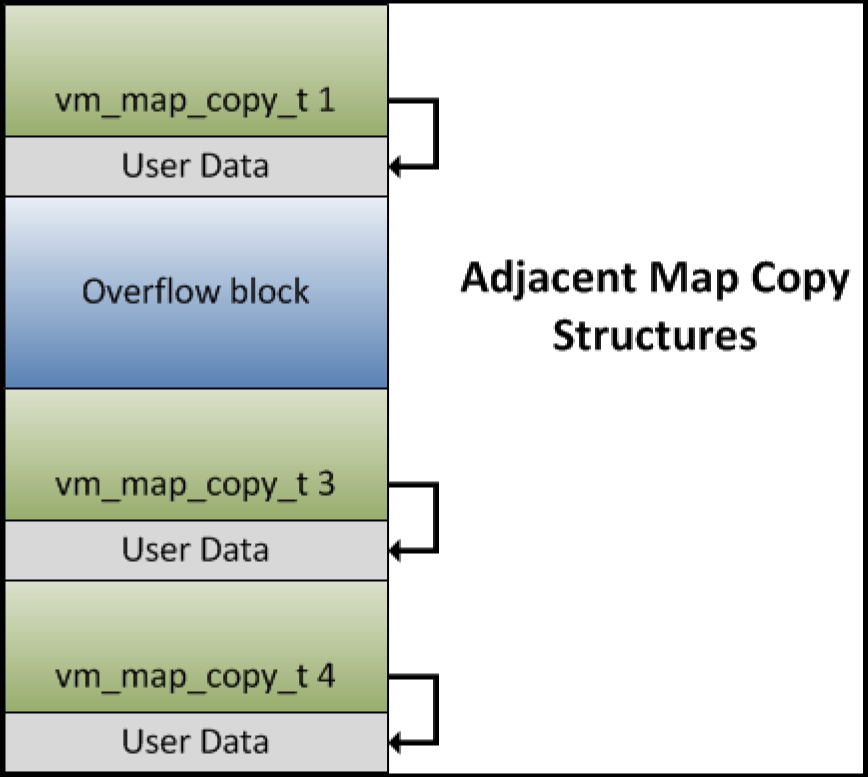
Kernel Attacks: Heap Overflows



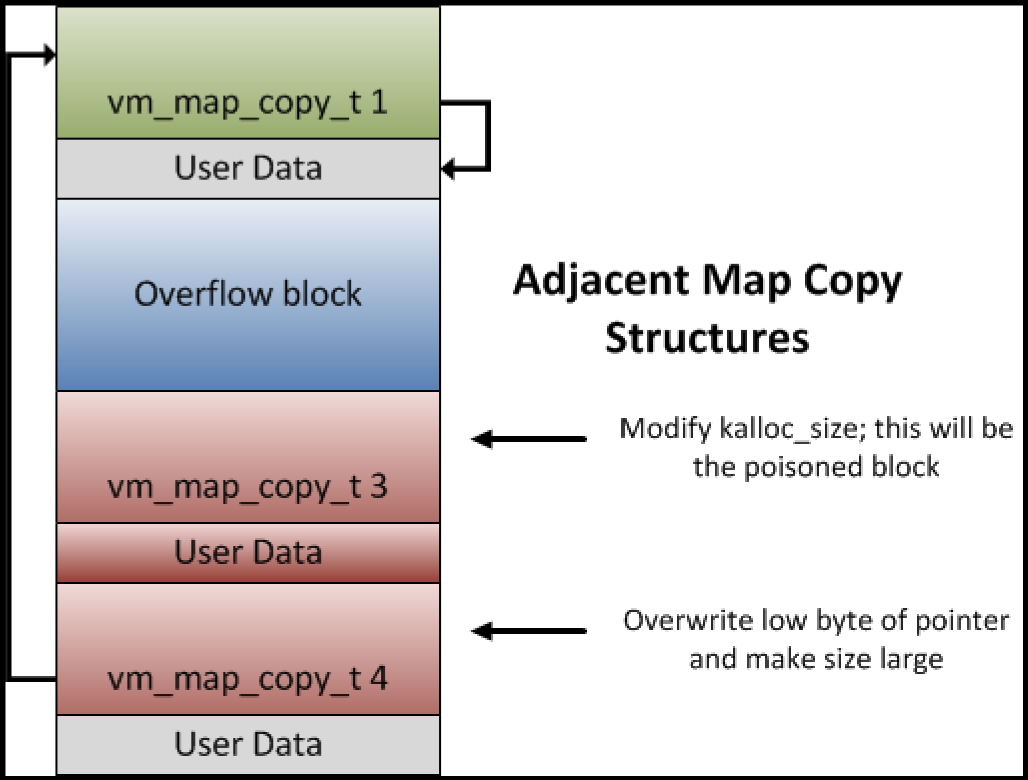
Kernel Attacks: Heap Overflows

- Primitive 4: Find our own address + Overflow
  - Mix and match primitive 1 and 3
  - Overwrite one whole vm_map_copy_t, changing kalloc_size to be suitably large
  - Overflow in to adjacent vm_map_copy_t, partially overwriting pointer / length
  - Free second copy (revealing pointers to itself)
  - Free first copy, creating poisoned kalloc block at known location

Kernel Attacks: Heap Overflows



Kernel Attacks: Heap Overflows



Conclusion

- iOS 6 mitigations significantly raise the bar
  - Many of the old tricks don’t work
  - A variety of bugs likely to be (reliably) unexploitable now
- Presented strategies provide useful mechanisms for exploiting iOS 6
- Thanks!