iOS 10 - Kernel Heap Revisited

<stefan.esser@sektioneins.de>

Singapore, August 2016
Motivation behind this talk

- my talk about the iOS kernel heap was around time of iOS 5
- however many details have changed between iOS 5 and iOS 9
- there are a number of tweets/blog posts about some of these changes but not all (e.g. from Azimuth about iOS 7 page lists)
- some iOS kernel talks at BlackHat do not even mention that there were changes
- and no central talk trying to discuss all these changes
And then iOS 10 beta happened …

- When this talk was submitted, **no iOS 10 beta was publicly available**.
- It was expected that iOS 10 would again **slightly** change the heap.
- Expectation was wrong because the 1st iOS 10 beta showed bigger changes.
- However, iOS 10 is still in beta, so details might change until final release.

**WARNING:** Because any kind of kernel research on iOS is harder than on OS X/MacOS, the preliminary analysis of new features was performed with debugging kernel extensions on MacOS and then manually compared to decompilation of the iOS 10 kernel.
Agenda

- **Part I:** What is the iOS kernel heap?
- **Part II:** iOS kernel heap around iOS 5
- **Part III:** Changes to the iOS kernel heap between iOS 6 and 9
- **Part IV:** Upcoming changes to the iOS kernel heap in iOS 10
Part I

What is the iOS Kernel Heap?
Kernel Zone Heap Allocator

- most used heap allocator in kernel
- memory is divided into zones
- zones group allocations of same type/size together
- all allocations in one zone are same size
Zone Allocator Usage

- caller decides what zone is allocated from / freed to
  - \( \text{ptr} = \text{zalloc} (\text{zone}) \)
  - \( \text{zfree} (\text{zone}, \text{ptr}) \)
- size of allocated block depends on zone
- no variable length allocations
List of Zones

```bash
$ sudo zprint
Password:

<table>
<thead>
<tr>
<th>zone name</th>
<th>elem size</th>
<th>cur size</th>
<th>max size</th>
<th>cur #elts</th>
<th>max #elts</th>
<th>cur inuse</th>
<th>alloc size</th>
<th>alloc count</th>
</tr>
</thead>
<tbody>
<tr>
<td>zones</td>
<td>288</td>
<td>64K</td>
<td>54K</td>
<td>227</td>
<td>192</td>
<td>215</td>
<td>20K</td>
<td>71</td>
</tr>
<tr>
<td>vm.objects</td>
<td>240</td>
<td>35280K</td>
<td>44286K</td>
<td>150528</td>
<td>188956</td>
<td>134502</td>
<td>4K</td>
<td>17</td>
</tr>
<tr>
<td>vm.object.hash.entries</td>
<td>40</td>
<td>4928K</td>
<td>5832K</td>
<td>126156</td>
<td>149299</td>
<td>120377</td>
<td>4K</td>
<td>102</td>
</tr>
<tr>
<td>maps</td>
<td>248</td>
<td>92K</td>
<td>90K</td>
<td>379</td>
<td>371</td>
<td>329</td>
<td>8K</td>
<td>33</td>
</tr>
<tr>
<td>VM.map.entries</td>
<td>80</td>
<td>5832K</td>
<td>7776K</td>
<td>74649</td>
<td>99532</td>
<td>58296</td>
<td>20K</td>
<td>256</td>
</tr>
<tr>
<td>Reserved.VM.map.entries</td>
<td>80</td>
<td>356K</td>
<td>2560K</td>
<td>150528</td>
<td>188956</td>
<td>134502</td>
<td>4K</td>
<td>17</td>
</tr>
<tr>
<td>VM.map.copies</td>
<td>88</td>
<td>16K</td>
<td>24K</td>
<td>186</td>
<td>279</td>
<td>0</td>
<td>8K</td>
<td>93</td>
</tr>
<tr>
<td>VM.map.holes</td>
<td>32</td>
<td>236K</td>
<td>16K</td>
<td>7552</td>
<td>512</td>
<td>5750</td>
<td>4K</td>
<td>128</td>
</tr>
<tr>
<td>pmap</td>
<td>368</td>
<td>124K</td>
<td>144K</td>
<td>345</td>
<td>400</td>
<td>317</td>
<td>4K</td>
<td>11</td>
</tr>
<tr>
<td>pagetable.anchors</td>
<td>4096</td>
<td>1328K</td>
<td>1751K</td>
<td>332</td>
<td>437</td>
<td>317</td>
<td>4K</td>
<td>1</td>
</tr>
<tr>
<td>pv_list</td>
<td>48</td>
<td>21864K</td>
<td>27217K</td>
<td>466432</td>
<td>580648</td>
<td>466200</td>
<td>12K</td>
<td>256</td>
</tr>
<tr>
<td>vm.pages</td>
<td>64</td>
<td>127336K</td>
<td>0K</td>
<td>2037376</td>
<td>0</td>
<td>2034534</td>
<td>4K</td>
<td>64</td>
</tr>
<tr>
<td>kalloc.16</td>
<td>16</td>
<td>1136K</td>
<td>1167K</td>
<td>72704</td>
<td>74733</td>
<td>52626</td>
<td>4K</td>
<td>256</td>
</tr>
<tr>
<td>kalloc.32</td>
<td>32</td>
<td>1348K</td>
<td>1751K</td>
<td>43136</td>
<td>56050</td>
<td>40037</td>
<td>4K</td>
<td>128</td>
</tr>
<tr>
<td>kalloc.48</td>
<td>48</td>
<td>3308K</td>
<td>3941K</td>
<td>70570</td>
<td>84075</td>
<td>59071</td>
<td>4K</td>
<td>85</td>
</tr>
<tr>
<td>kalloc.64</td>
<td>64</td>
<td>6128K</td>
<td>8867K</td>
<td>98048</td>
<td>141877</td>
<td>84972</td>
<td>4K</td>
<td>64</td>
</tr>
<tr>
<td>kalloc.80</td>
<td>80</td>
<td>1776K</td>
<td>1751K</td>
<td>22732</td>
<td>22420</td>
<td>21808</td>
<td>4K</td>
<td>51</td>
</tr>
<tr>
<td>kalloc.96</td>
<td>96</td>
<td>1748K</td>
<td>2335K</td>
<td>18645</td>
<td>24911</td>
<td>18276</td>
<td>8K</td>
<td>85</td>
</tr>
<tr>
<td>kalloc.128</td>
<td>128</td>
<td>5352K</td>
<td>5911K</td>
<td>42816</td>
<td>47292</td>
<td>32835</td>
<td>4K</td>
<td>32</td>
</tr>
<tr>
<td>kalloc.160</td>
<td>160</td>
<td>1340K</td>
<td>1556K</td>
<td>8576</td>
<td>9664</td>
<td>6495</td>
<td>8K</td>
<td>51</td>
</tr>
<tr>
<td>kalloc.256</td>
<td>256</td>
<td>9328K</td>
<td>13301K</td>
<td>37312</td>
<td>53204</td>
<td>24442</td>
<td>4K</td>
<td>16</td>
</tr>
<tr>
<td>kalloc.288</td>
<td>288</td>
<td>1660K</td>
<td>2594K</td>
<td>5902</td>
<td>9226</td>
<td>5563</td>
<td>20K</td>
<td>71</td>
</tr>
<tr>
<td>kalloc.512</td>
<td>512</td>
<td>19324K</td>
<td>19951K</td>
<td>38648</td>
<td>39903</td>
<td>35135</td>
<td>4K</td>
<td>8</td>
</tr>
<tr>
<td>kalloc.1024</td>
<td>1024</td>
<td>6024K</td>
<td>8867K</td>
<td>6024</td>
<td>8867</td>
<td>4945</td>
<td>4K</td>
<td>4</td>
</tr>
<tr>
<td>kalloc.1280</td>
<td>1280</td>
<td>480K</td>
<td>512K</td>
<td>384</td>
<td>410</td>
<td>186</td>
<td>20K</td>
<td>16</td>
</tr>
<tr>
<td>kalloc.2048</td>
<td>2048</td>
<td>6164K</td>
<td>8867K</td>
<td>3082</td>
<td>4433</td>
<td>2784</td>
<td>4K</td>
<td>2</td>
</tr>
<tr>
<td>kalloc.4096</td>
<td>4096</td>
<td>12968K</td>
<td>13301K</td>
<td>3242</td>
<td>3325</td>
<td>1768</td>
<td>4K</td>
<td>1</td>
</tr>
<tr>
<td>kalloc.8192</td>
<td>8192</td>
<td>6440K</td>
<td>7882K</td>
<td>805</td>
<td>985</td>
<td>412</td>
<td>8K</td>
<td>1</td>
</tr>
<tr>
<td>mem_obj_control</td>
<td>16</td>
<td>1956K</td>
<td>2187K</td>
<td>125184</td>
<td>139968</td>
<td>120377</td>
<td>4K</td>
<td>256</td>
</tr>
</tbody>
</table>

...
Zone Memory (Region)

• kernel reserves a memory region called **zone_map** for zone allocator

• default size of this region is 1/4 of the physical memory

• reserved very early on during kernel start
Zone Memory (Pages)

- Pages from `zone_map` are assigned to zones.
- Zones are grown by their allocation size.
- Kernel pages are 16kb on new 64 bit device with lots of memory.
- All other devices have 4kb pages.
- To waste less space, zones use optimal allocation sizes >= 1 page.
Dynamic Length Allocations?

• zone allocator is not suited for dynamic length allocations

• but for dynamic length allocations various wrappers exist, e.g.:

  • `kalloc(size) / kfree(ptr, size)` - wrapper around `zalloc()`
    dynamic length but caller must remember length

  • `MALLOC(size) / FREE(ptr)` - wrapper around `kalloc()`
    dynamic length and uses meta data to remember length
• iOS has kernel boot arguments to enable heap allocation debugging

• for us the interesting boot arguments are

  • -zc selects zone corruption logging mode
  • zlog=<zonename> selects ONE zone to log
  • zrecs=<number> controls how many log entries should be kept

• because iOS does not allow to control kernel boot arguments these features need to be activated by means of a kernel exploit
Kernel Heap Allocation Debugging (II)

- kernel zone allocator **corruption logging** logs for all (de)allocations
  - if allocation or deallocation
  - **pointer** allocated / deallocated
  - kernel **backtrace** with up to 15 elements
- data is collected via **btlog** routines in kernel
- researcher need to create their own routine to extract data from kernel
Kernel Heap Allocation Debugging (III)
Part II

iOS Kernel Heap around iOS 5
Zone Structure

- information about a zone is stored in `zone` struct
- zone structs are stored in heap zone “zones”

```c
struct zone {
    int count; /* Number of elements used now */
    vm_offset_t free_elements;
    decl_lck_mtx_data,(lock) /* zone lock */
    lck_mtx_ext_t lock_ext; /* placeholder for indirect mutex */
    lck_attr_t lock_attr; /* zone lock attribute */
    lck_grp_t lock_grp; /* zone lock group */
    lck_grp_attr_t lock_grp_attr; /* zone lock group attribute */
    vm_size_t cur_size; /* current memory utilization */
    vm_size_t max_size; /* how large can this zone grow */
    vm_size_t elem_size; /* size of an element */
    vm_size_t alloc_size; /* size used for more memory */
    uint64_t sum_count; /* count of allocs (life of zone) */
    unsigned int
        exhaustible :1, /* (F) merely return if empty? */
        collectable :1, /* (F) garbage collect empty pages */
        expandable :1, /* (T) expand zone (with message)? */
        allows_foreign :1,/* (F) allow non-zalloc space */
        doing_alloc :1, /* is zone expanding now? */
        waiting :1, /* is thread waiting for expansion? */
        async_pending :1, /* asynchronous allocation pending? */
        caller_acct: 1, /* do we account allocation/free to the caller? */
        doing_gc :1, /* garbage collect in progress? */
        noencrypt :1,
        no_callout:1,
        async_prio_refill:1;
    int index; /* index into zone_info arrays for this zone */
    struct zone * next_zone; /* Link for all-zones list */
    call_entry_data_t call_async_alloc; /* callout for asynchronous alloc */
    const char *zone_name; /* a name for the zone */
    vm_size_t prio_refill_watermark;
    thread_t zone_replenish_thread;
};
```

DISCLAIMER: this struct layout if taken from OS X 10.7.5 - the iOS 5 layout might have been slightly different
Free Memory Blocks

- free elements are kept in a **single linked freelist** per zone
- zone structure has **free_elements** pointer to head of freelist
- free elements have a **pointer to next free** element in beginning
- there is **no pointer(-chain) back** to zone struct
  - **not possible** to know what zone an element (a page) **belongs to**
  - **slow garbage collection**
Allocation (I)

- allocation returns next element from freelist
- when free list is empty zone grows by allocation size
- all elements in added pages are added to free list
- last element will be first in freelist (LIFO)
Allocation (II)

- allocation returns next element from freelist
- when free list is empty zone grows by allocation size
- all elements in added pages are added to free list
- last element will be first in freelist (LIFO)
Allocation (III)

- allocation returns next element from freelist
- when free list is empty zone grows by allocation size
- all elements in added pages are added to free list
- last element will be first in freelist (LIFO)

“new memory is returned in backward order by heap allocator”
Allocation (IV)

- allocation returns next element from freelists
- when free list is empty zone grows by allocation size
- all elements in added pages are added to free list
- last element will be first in freelists (LIFO)

"new memory is returned in backward order by heap allocator"
Dynamic Length Allocations - kalloc()

- to handle dynamic lengths kalloc() registers multiple zones
- named kalloc.<number>
  - e.g. kalloc.128 for 128 byte allocations
- allocations will be put into the next larger zone
  - e.g. 97 byte allocations in kalloc.128 (31 bytes waste)
- different iOS versions define different zones
  (also depending on 32 bit vs. 64 bit)
- caller needs to remember allocation size so that
  kfree(ptr, size) can put it back into the right zone
Dynamic Length Allocations - MALLOC()

- to handle dynamic lengths MALLOC() stores the size as meta data
- internally uses `kalloc()` / `kfree()`
- stores the size in front of data

<table>
<thead>
<tr>
<th>size</th>
<th>data</th>
<th>waste due to zone size</th>
</tr>
</thead>
</table>
How attackers abused the iOS 5 Zone Allocator

• Heap-Feng-Shui

  • exploit specific allocation and deallocation primitives (e.g. opening/closing NDRV sockets)

  • generic allocation deallocation primitives via `OSUnserializeXML` (controlling memory layout via many IOKit API functions using XML - and filling it with objects and object pointers)

• Corruption Targets

  • zone allocator freelist `next_ptr` pointers (control ptr next allocation will return and overwrite with attacker controlled data)

  • `size fields` of elements allocated via `MALLOC()` or a wrapper (tricks `kfree()` to put element into a zone that is for bigger elements - next allocation will result in a larger bufferoverflow)

  • `application data` (control objects/vtable pointers, object pointers, size fields, …)
Part II

Changes to the iOS kernel heap between iOS 6 and 9
iOS 6 Heap Changes/Hardening

- Apple made some changes to `OSUnserializeXML` that had little impact on usability for heap feng shui
- Location of `zone_map` is randomized due to KASLR
- Single linked freelist now protected by `heap cookies/canaries`
- Small free memory blocks are now poisoned
iOS 6 Heap Cookies

- iOS creates two random cookies
  - `zp_nopoison_cookie` (lowest bit cleared)
  - `zp_poisoned_cookie` (lowest bit set)
- Last bytes of a free element will be overwritten with: `ptr ^ cookie`
- Allocation code will detect illegal cookie values and `panic()`
- Protection against overflows and other `ptr` corruption
iOS 6 Heap Cookie Leak Protection

- both `ptr` and `ptr^cookie` get overwritten when block is allocated
- value `0xdeadbeef` is written over sensitive values
- protects against potential information leak after memory block is returned
iOS 6 Heap Poisoning

- small blocks when freed get overwritten with **0xdeadbeef**
- indicated by use of **zp_poisoned_cookie**
- on allocation **0xdeadbeef** is verified - **panic()** if modified
How attackers abused the iOS 6 Zone Allocator

- **Heap-Feng-Shui**
  - with release of iOS 6 creating `vm_map_copy_t` structures by sending `mach messages with OOL data` became heap-feng-shui method of choice
  - previously used methods less often seen but still possible

- **Corruption Targets**
  - because zone allocator freelist got protected all public exploits seem to target `vm_map_copy_t` structure overwrites (could be used for arbitrary info leak and `kfree()` confusion)
  - other size fields of e.g. `MALLOC()` and application data still targeted
• poisoning of larger blocks every $X$ frees

• introduction of new heap pagelist feature (for easier GC)
iOS 7 Large Block Poisoning

- `zp_factor` debugging feature *(that already existed before)* is now always activated
- randomly set once at boot time to the value **15** (25%), **16** (50%) or **17** (25%)
- controls after every how many frees in a zone a single block is poisoned (regardless of size)
- counter is zone specific in the `zp_count` field of the zone structure
iOS 7 Zone Pagelist Feature

- iOS 7 introduces **new pagelist feature**
- activated via **bit** in zone struct (only **subset of zones** use new feature)
- adds **meta data at end of all pages** inside a zone
- keeps all pages in one of four **double linked lists (queue)**
iOS 7 Zone Page Meta Data

- backpointer to zone
- free elements in this page (page local freelist)
- forward and backward pointer to other pages
- alloc_count - max number of elements in page
- free_count - number of free elements in page

```c
struct zone_page_metadata {
    queue_chain_t pages;
    struct zone_free_element *elements;
    zone_t zone;
    uint16_t alloc_count;
    uint16_t free_count;
};
```
iOS 7 Zone Pagelists

- Allocator defines four queues for every zone
  - `any_free_foreign` - for the few zones that allow foreign elements
  - `intermediate` - for pages that are partially allocated at the moment
  - `all_free` - for pages that are completely free at the moment
  - `all_used` - for pages that are completely allocated at the moment
- during allocation or free the allocator ensures that page is always on right queue
iOS 7 Allocation under Page Lists

- allocation now traverses the page queues in the following order
  - any_free_foreign
  - intermediate
  - all_free
- first queue with a usable page is used
- then the first free element from this page’s freelist is returned
- if no usable page found - system adds an all_free page and retries
iOS 7 Freeing under Page Lists

- freeing an **element** adds it to its page’s **freelist**
- allocator ensures that page is still on **right queue**
- page could move
  - from **intermediate** to **all_free**
  - from **all_used** to **intermediate**
Impact of iOS7 Page Lists

- Garbage collection now super easy
  
  *(just give back all pages from the all_free page queue)*

- Freeing of memory is **local to each page’s freelist**
  
  *(less interruption for heap-feng-shui)*

- Allocations are **local to current front page**
  
  *(not really a change because compatible exploits always had to concentrate on staying in one page)*

- Meta data contained **double linked list without any exploit mitigation**
  
  *(iOS 7 made heap overflows very easy to exploit)*

- **NEW ATTACK:**
  
  Confusing allocator to free elements from page list zones in freelist zones
Was there a memory corruption? Yes? Continue!

- Apple added code that detects for page list zones if a corruption happened
- when they detect a corruption they try to repair it <—— NEVER EVER DO THAT!
- when they cannot repair they ignore that they detected a memory corruption <—— APPLE REALLY???

```c
if (zone->use_page_list) {
    struct zone_page_metadata *page_meta = get_zone_page_metadata((struct zone_free_element *)addr);
    if (zone != page_meta->zone) {
        /* Something bad has happened. Someone tried to zfree a pointer but the metadata says it is from
         * a different zone (or maybe it's from a zone that doesn't use page free lists at all). We can repair
         * some cases of this, if:
         * 1) The specified zone had use_page_list, and the true zone also has use_page_list set. In that case
         *    we can swap the zone_t
         * 2) The specified zone had use_page_list, but the true zone does not. In this case page_meta is garbage,
         *    and dereferencing page_meta->zone might panic.
         * To distinguish the two, we enumerate the zone list to match it up.
         * We do not handle the case where an incorrect zone is passed that does not have use_page_list set,
         * even if the true zone did have this set.
         */
        zone_t fixed_zone = NULL;
        int fixed_i, max_zones;

        simple_lock(&all_zones_lock);
        max_zones = num_zones;
        fixed_zone = first_zone;
        simple_unlock(&all_zones_lock);

        for (fixed_i=0; fixed_i < max_zones; fixed_i++, fixed_zone = fixed_zone->next_zone) {
            if (fixed_zone == page_meta->zone && fixed_zone->use_page_list) {
                /* we can fix this */
                printf("Fixing incorrect zfree from zone %s to zone %s\n", zone->zone_name, fixed_zone->zone_name);
                zone = fixed_zone;
                break;
            }
        }
    }
}
```
How attackers abused the iOS 7 Zone Allocator

- **Heap-Feng-Shui**
  - same methods as before were used

- **Corruption Targets**
  - all previously targeted areas still work
  - but double linked lists introduced by pagelist feature easiest target
    (target the next/prev pointers for arbitrary writes)
iOS 8 Heap Changes/Hardening

- pagelist queue hardening
- poisoning of larger blocks made less frequent (depending on size)
iOS 8 Queue Hardening

- iOS 8 adds safe unlink protection to its queue macros
- double linked lists added to iOS 7 heap are now protected
- heap overflows can no longer go after the queue pointers in metadata
iOS 8 Less Frequent Large Block Poisoning

- Introduction of `zp_scale` feature with a default value of 4.
- Poisoning of large blocks follows now the following formula:
  \[ zp\_factor + element\_size \gg zp\_scale \]
- This means the larger elements are in a zone the less often they are poisoned.
- Example: \((15/16/17) + 256 \gg 4 = (31/32/33)\)
How attackers abused the iOS 8 Zone Allocator

- Heap-Feng-Shui
  - same methods as before were used

- Corruption Targets
  - targeting double linked lists (pagelists) not possible due to `safe_unlink`
  - all other previously targeted areas still work
iOS 9 Heap Changes/Hardening

- `vm_map_copy_t` "hardening"
- repositioning of page metadata
- randomization of initial freelist
iOS 9 `vm_map_copy_t` “hardening”

- `vm_map_copy_t` structure for kernel buffers was stripped down
  - `data pointer removed` because data should be after header
  - secondary `size field` removed because that is `headerlen + size`
- smaller structure **allows controlling** heap in **smaller zones**
- when overwritten can **still lead to zone confusion and info leaks**
- but info leaks are **limited to 4k because hardcoded limit** in code
- and removal of pointer **removes possibility for arbitrary info leaks**
- some places try to verify that size is not overwritten
iOS 9 moves the page meta data to the beginning of the page

possible reasoning: protect meta data against overflows
when new memory is added to a zone elements within are freed one by one into the freelist

starting with iOS 9.2 the elements are no longer added linear from first to last

instead every time a next element is added a random decision is made if the first or last element should be added
• when new memory is added to a zone elements within are freed one by one into the freelist

• starting with iOS 9.2 the elements are no longer added linear from first to last

• instead every time a next element is added a random decision is made if the first or last element should be added
iOS 9.2 Initial Freelist Randomization (III)

- when new memory is added to a zone, elements within are freed one by one into the freelist.
- starting with iOS 9.2 the elements are no longer added linear from first to last.
- instead every time a next element is added a random decision is made if the first or last element should be added.
• when new memory is added to a zone elements within are freed one by one into the freelist

• starting with iOS 9.2 the elements are no longer added linear from first to last

• instead every time a next element is added a random decision is made if the first or last element should be added
iOS 9.2 Initial Freelist Randomization (V)

- When new memory is added to a zone, elements within are freed one by one into the freelist.
- Starting with iOS 9.2, the elements are no longer added linear from first to last.
- Instead, every time a next element is added, a random decision is made if the first or last element should be added.
iOS 9.2 Initial Freelist Randomization (VI)

- when new memory is added to a zone, elements within are freed one by one into the freelist.
- starting with iOS 9.2 the elements are no longer added linear from first to last.
- instead, every time a next element is added, a random decision is made if the first or last element should be added.
iOS 9.2 Initial Freelist Randomization (VII)

- when new memory is added to a zone elements within are freed one by one into the freelist
- starting with iOS 9.2 the elements are no longer added linear from first to last
- instead every time a next element is added a random decision is made if the first or last element should be added
How attackers abused the iOS 9 Zone Allocator

• **Heap-Feng-Shui**
  
  • same methods as before were used
    (vm_map_copy_t after “hardening” still usable for heap-feng-shui)

• **Corruption Targets**
  
  • targeting `vm_map_copy_t` still allows zone confusion and limited info leaks
    (only larger info leaks and arbitrary pointer info leaks stopped by “hardening”)
  
  • all other previously targeted areas still work
Part III

Upcoming changes to the iOS kernel heap in iOS 10
Changes in upcoming iOS 10

• more `vm_map_copy_t` "hardening"
• fixed zone structure array
• zone page metadata completely revamped
• page freelist pointer leak protection
• `zalloc()` wrappers add no inbound metadata anymore
• zone allocator debugging features revamped
iOS 10 Fixed Zonestructure Array

- zone structures are no longer allocated on zone allocator heap
- instead stored in fixed zone_array in __DATA::__bss
- seems to be a bad idea because address of zone structure is now fixed (relative to kernelbase)

```plaintext
__DATA::__bss:FFFFFFF007552170 public zone_metadata_region_min
__DATA::__bss:FFFFFFF007552170 ; vm_offset_t zone_metadata_region_min
__DATA::__bss:FFFFFFF007552170 zone_metadata_region_min dq 0 ; DATA XREF: zone_element_size+19r
__DATA::__bss:FFFFFFF007552170 ; get_zone_page_metadata+2Fr ...
__DATA::__bss:FFFFFFF007552178 public zone_metadata_region_max
__DATA::__bss:FFFFFFF007552178 ; vm_offset_t zone_metadata_region_max
__DATA::__bss:FFFFFFF007552178 zone_metadata_region_max dq 0 ; DATA XREF: zone_init+F3w
__DATA::__bss:FFFFFFF007552180 public zone_array
__DATA::__bss:FFFFFFF007552180 ; zone zone_array[256]
__DATA::__bss:FFFFFFF007552180 zone_array zone 100h dup(0) ; DATA XREF: panic_display_zprint+4Bo
__DATA::__bss:FFFFFFF007552180 ; zone_element_size+5Do ...
```
in iOS 10 ALL zones make use of page metadata

a new zone_metadata_region is utilised for that

region is reserved in the zone_map

used to keep meta data for every single page in the zone_map
iOS 10 page metadata

- **zindex**: index of zone in zone_array (instead of back pointer)
- **page_count**: number of pages in allocation size
- **free_count**: number of free elements in page
- **freelist_offset**: offset of first free element starting from this page’s address
- **real_metadata_offset**: offset of this secondary meta data from page 0 metadata
iOS 10 page freelists

- page metadata does not contain pointer to head of freelist anymore
- instead `freelist_offset` determines byte offset of first free element in page
- `ptr` to next free block is always XORed against `nopoison_cookie`
- backup `ptr` is `ptr` value XORed against selected cookie (noposition/poisioned)

Clever change from before even if whole memory leaks it is not possible to determine cookie values unless you know exact address of next `ptr`.

 ALSO PROTECTION AGAINST TYPE CONFUSION OF EMPTY BLOCKS AS IOKIT OBJECTS
iOS 10 Metadata vs. Wrong Zone Frees

- every zone uses the new metadata
- this allows determining correct zone by element
- freeing elements into the wrong zone can be reliably detected
- when this happens `zfree()` panics the kernel
- abuse seems not possible anymore
iOS 10 Wrappers and Metadata

- new meta data can be used to determine size of elements on heap
- heap allocation wrappers like `MALLOC()` and `kern_os_malloc()` no longer need to store the size
- both become simpler wrappers around `kalloc()` and are therefore inlined in several places
- allocations are now only application data
  (more codepaths can now be used for heap layout control)
iOS 10 Kernel Heap Allocation Debugging (I)

- kernel heap allocation logging was revamped
- no longer limited to a single zone
- code now limited to logging in max 5 zones
- zones selected by new kernel boot args
  - `zlog1`, `zlog2`, `zlog3`, `zlog4`, `zlog5`
  - usage: `-zc zlog1=kalloc.128 zlog2=kalloc.256`
• bitfield in zone struct contains new bit 15 to select if `zone_logging`
• zone struct contains `zlog_btlog` variable instead of global variable
• `btlog_t` structure heavily modified
• log is no longer a simple list but a hash table

```c
unsigned __int32 gzalloc_exempt : 1;
unsigned __int32 alignment_required : 1;
unsigned __int32 zone_logging : 1;
unsigned __int32 zone_replenishing : 1;
unsigned __int32 _reserved : 15;
int index;
const char *zone_name;
uint32_t_0 zleak_capture;
uint32_t_0 zp_count;
vm_size_t prio_refill_watermark;
thread_t zone_replenish_thread;
gzalloc_data_t gz;
btlog_t *zlog_btlog;
};
```
How attackers will abuse the iOS 10 Zone Allocator

- **Heap-Feng-Shui**
  - for simple feng-shui all previous methods should still be usable (beside continued hardening of vm_map_copy_t)
  - code paths that use MALLOC() are now also usable for heap-feng-shui (e.g. posix_spawn())

- **Corruption Targets**
  - application data like object and mach port pointers very interesting
  - **BUT** size field corruption to confuse free into wrong zone will trigger `panic()`
Time for questions...

Questions...?