12

Ceci n'est pas une "heap"*: Kernel Memory Management

Thankfully, kernel KPI users seldom have to deal with the physical layer, and most calls can remain in the virtual layer. We detail how the kernel manages its own `vm_map` - the `kernel_map` - through `kmem_alloc*` and `kalloc*`. We next present the very important abstractions of kernel zones. Special care is given to the nooks and crannies of the zone allocator, due to their significance over the years in exploitation.

Before concluding, we reintroduce memory pressure - also presented in I/8, but described here from the kernel's perspective. The gentle `memorystatus` of MacOS and the ruthless `jetsam` of *OS are both detailed. This continues with a discussion of XNU's proprietary purgeable memory, which helps deal with memory pressure automatically. Lastly, we conclude with a rough map of the kernel's address space, and consider the kernel slide.

---

* - Kernel memory is often incorrectly refered to as the the "kernel heap". This term (apparently used within Apple as well) is technically wrong - the heap refers to the data strcuture used in user-mode in backing `malloc(3)`, as opposed to the automatic, thread-local memory of the stack (although the pedantic will correctly argue that user mode "heaps" no longer use the heap structure as well...). Though the kernel does make use of stack memory (for its threads and user-mode threads when in-kernel), the heap data structure is not used in any way in maintaining kernel memory, which is merely segmented into zones, similar to the Linux Slab allocator.

XNU makes use of multiple memory allocation facilities in the kernel. We have already touched on those of BSD earlier in Chapter 6 ("BSD Memory Zones"), and we will discuss those of IOKit in Chapter 13. Let us now consider then, those of Mach.

The `kernel_map`

The `kernel_map` is the `vm_map` representing the kernel's addressable virtual memory space. It is set up in `kmem_init()` (osfmk/vm/vm_kern.c, called from `vm_mem_bootstrap()`). The map is then `vm_map_create()`d to span `VM_MIN_KERNEL_AND_KEXT_ADDRESS` to `VM_MAX_KERNEL_ADDRESS`. On MacOS, the range spanned is 0xffffff7f80000000-0xfffffffffffe000, and in *OS it is 0xffffffe000000000-0xfffffff3fffffff. These ranges allow 513GB and "merely" 79GB (respectively), although the map naturally contains wide holes.

All subsequent kernel memory allocations inevitably end up as allocations inside the `kernel_map`. This is either by allocating directly in it (using a `kmem_alloc*()` variant), or carving out a sub map (`kmem_suballoc()`. With few notable exceptions (namely, `kmem_alloc_pageable[_external]()`), all memory in the `kernel_map` is wired - i.e. resident in physical memory.

`kmem_alloc()` and friends

This is where the `kmem_alloc*()` family of functions comes into play. The various functions in the family (with the exception of `kmem_alloc_pages()`) all funnel to `kernel_memory_allocate()`, although with different parameter and flag settings. `kmem_alloc_pages()` is different, in that it allocates virtual pages using `vm_page_alloc()`, but does not commit any of them.

`kernel_memory_allocate()`

The `kernel_memory_allocate()` function is thus a common handler for nearly all memory allocations. It proceeds as follows:

- Round allocation size to page multiple. The allocation size is also checked to not be zero, or not exceed sane boundaries.
- If guard pages were requested (indicated by `KMA_GUARD_[FIRST|LAST]` flags) they are allocated. This is done by a call to `vm_page_grab_guard()`, which sets page protections but only maps them fictitiously.
- If actual wired memory was requested (indicated by the absence of `KMA_VAONLY` and/or `KMA_PAGEABLE`), it is allocated using `vm_page_grab[lo]()`. Low memory is only grabbed if the `KMA_LOMEM` flag was specified.
- If the `KMA_KOBJECT` flag was specified, the `kernel_object` is referenced. If `KMA_COMPRESOR` was specified, the `compressor_object` is referenced. Otherwise, a new memory object is created using a call to `vm_object_allocate()`.
- `vm_map_find_space()` is called to find a suitable map address. This populates both the `map_addr` and the `entry`. The `vm_map_entry_t` is then linked to the object and offset.
- Any `KMA_GUARD_FIRST` guard pages are `vm_page_insert()`ed to the map, followed by wired pages (with `vm_page_insert_wired()`). For the wired page, `PMAP_ENTER_OPTIONS` is called, trying a fast path (`PMAP_OPTIONS_NOWAIT`). Should that fail - a slow path with `PMAP_ENTER` is tried, and must succeed. Any `KMA_GUARD_LAST` pages are then appended to the map, again through `vm_page_insert()`.
- For allocations using the `kernel_object` or `compressor_object`, a call to `vm_map_simplify()` coalesces neighboring regions. Otherwise, the memory object is deallocated.

kmem_suballoc()

The kmem_suballoc() function creates and maps a submap inside a parent map (usually the kernel_map). Important suballocations include the kalloc_map (for kalloc.* zones), ipc_kernel[_copy]_map, the g_kext_map (for kext mappings), the zone_map (and zone tag maps, if #VM_MAX_TAG_ZONES), IOKit pageable spaces, and BSD's bufferhdr_map, mb_map, and the bsd_pageable_map.

The flow of kmem_suballoc() is a simple application of vm_map_enter() to reserve the memory in the parent map, followed by vm_map_create() for the submap, and vm_map_submap() to link the two together.

kmem_realloc()

It's quite rare, but there are certain cases wherein a reallocation of existing memory might be required. Examples of this are the ensureCapacity() methods of libkern's OSSerialize and OSData objects, and vm_purgeable_token_add(). For these cases, kmem_realloc() is used.

There are two fine points with kmem_realloc(). The first is, that it does not free the original address range - meaning that it must be explicitly kmem_free()ed. The second, is that it does not bzero() the new memory allocated (if larger than the original allocation), leading to a potential kernel memory disclosure.

kalloc()

The simplest KPI for memory allocation provided by the kernel is that of kalloc variants. The simplest of these macros, defined in osfmk/kern/kalloc.h, is the kernel-mode equivalent of user-mode's malloc(3). Its variants allow for additional options, like specifying a memory tag or controlling whether the operation can block or not. Table 12-1 shows these macros:

Table 12-1: The various kalloc() macros, defined in osfmk/kern/kalloc.h

Macro variant	Provides
kalloc(*size*)	Basic allocation of *size* bytes
kalloc_tag(*size*,itag)	As above, but with memory tag
kalloc_tag_bt(*size*,itag)	As above, but with tag determined from caller
kalloc_noblock*	All the above, guaranteed to return immediately (but may fail)
kallocp*	All the above, taking *size* by reference and updating actual size allocated

The macros all expand to a call to kalloc_canblock(). Similar to the BSD MALLOC macro, they first set up a vm_allocation_site, and then call the underlying function, which takes the size argument by reference, a boolean as to whether or not it may block, and the site by reference. The reason the size is passed by reference is because kalloc_canblock() reports the actual size of the element allocated, which may be larger than the size requested. The need for this is rare, but kallocp* variants exists to allow passing the *size* argument by reference if necessary. libkern's kern_os_malloc() is one example of this feature's usefulness, as are libkern's calls to kallocp_container(), which ensure the returned memory is entirely bzero()ed up to the size allocated, even if more than initially requested.

kalloc()ed memory is automatically tagged as VM_KERN_MEMORY_KALLOC (#13), though a different tag may be specified with the _tag variants of the macro. As of XNU-3248, it's also possible to request an allocation to be tagged according to the backtrace leading up to it. This is done by tagging the allocation site with a special VM_TAG_BT value (0x800), as is done by the kalloc[/p_/noblock]_tag_bt variants. When the tag is retrieved, vm_tag_bt() (in osfmk/vm/vm_resident.c) walks the current thread's kernel stack, attempting to locate the most recent return address and then calls OSKextGetAllocationSiteForCaller. The function searches for the address in the kernel extension accounting data (described in Chapter 3), and retrieves the associated tag from the allocation site.

`kalloc.###` zones

Behind the scenes, `kalloc_canblock` tries not to block, by providing the allocation from one of several `kalloc.###` "zones". These zones are pre-allocated regions of memory, spanning an integer number of pages, and "carved" into element slots. Each zone uses a fixed element size, as indicated by the zone name (e.g. "kalloc.16" uses 16-byte elements). The zones are set up early during kernel initialization, by a call to `kalloc_init` (osfmk/kern/kalloc.c) from `vm_mem_bootstrap()`. The `kalloc` operation finds the first zone capable of containing the allocation, unless the allocation size exceeds that of the largest zone (presently, 8192 for MacOS, and 32,768 elsewhere), in which case `kmem_alloc_flags()` is used instead. XNU originally provided kalloc zones with sizes at exact powers of two, starting at 1 (2^0) bytes, but over time Apple has established 16 bytes as the smallest possible allocation, and added additional zones (48,80,96,160,192,224,288,368,400,576,768,1152,1280,1664,6144). If a zone can be found to satisfy the allocation, `kalloc_canblock()` calls `zalloc_canblock_tag()`, to perform the allocation from the zone and tag it according to the `vm_allocation_site`.

The Kalloc DLUT

With so many zones, it's important to keep the operation of finding the right one as quick and as efficient as possible. The **Direct LookUp Table** ("DLUT") was introduced in XNU-2050 as a means to quickly locate the right zone for an element.

Listing 12-2: The Direct LookUp Table of `kalloc` (from osfmk/kern/kalloc.c)

```
/*
 * Many kalloc() allocations are for small structures containing a few
 * pointers and longs - the k_zone_dlut[] direct lookup table, indexed by
 * size normalized to the minimum alignment, finds the right zone index
 * for them in one dereference.
 */

#define INDEX_ZDLUT(size)       \
                  (((size) + KALLOC_MINALIGN - 1) / KALLOC_MINALIGN)
#define N_K_ZDLUT        (2048 / KALLOC_MINALIGN)
                  /* covers sizes [0 .. 2048 - KALLOC_MINALIGN] */
#define MAX_SIZE_ZDLUT   ((N_K_ZDLUT - 1) * KALLOC_MINALIGN)

static int8_t k_zone_dlut[N_K_ZDLUT];    /* table of indices into k_zone[] */

/*
 * If there's no hit in the DLUT, then start searching from k_zindex_start.
 */
static int k_zindex_start;

static zone_t k_zone[MAX_K_ZONE];

..

/*
 * Given an allocation size, return the kalloc zone it belongs to.
 * Direct LookUp Table variant.
 */
static __inline zone_t
get_zone_dlut(vm_size_t size)
{
        long dindex = INDEX_ZDLUT(size);
        int zindex = (int)k_zone_dlut[dindex];
        return (k_zone[zindex]);
}
```

The slow path

As its name implies, there are circumstances in which `kalloc_canblock` does block. If the allocation cannot fit in one of the many `kalloc.*` zones (and `canblock` is TRUE), a slow path is taken instead. A call to `kmem_alloc_flags()` attempts to allocate the requested memory - first from the `kalloc_map`, and then - as a fallback - the `kernel_map`. The `kalloc_map` is a dedicated `vm_map` for large allocations, which is `kmem_suballoc()`ed by `kalloc_init()` to span virtual memory between `kalloc_map_min` and `kalloc_map_max`. The difference between the two (`kalloc_map_size`) is normally set to the `1/32` of the kernel's `sane_size`, or 128MB for 32-bit kernels.

What makes the slow path so slow is that `kmem_alloc_flags()` calls `kernel_memory_allocate()`, which needs to eventually call `vm_page_grab()` in order to obtain physical memory pages, to back the allocation. This operation will block if there are no pages immediately available, and can only be satisfied after page fault handling, which may take thousands of cycles, if not more. The current thread will surely block, making this path unsuitable and unsafe in atomic contexts. `kmem_alloc_flags()` is described later in this chapter.

OSMalloc*

Yet another memory allocation facility offered by the kernel is **`OSMalloc`**. The function prototypes are declared in libkern/libkern/OSMalloc.h (making them technically part of the Libkern KPI), but their implementation is in osfmk/kern/kalloc.c.

The main advantage of using `OSMalloc` is its support of **memory tags**. The function takes an additional `OSMallocTag` argument, which is a pointer to a structure defined as follows:

Listing 12-3: The `OSMalloc` tag, from libkern/libkern/OSMalloc.h

```
typedef struct _OSMallocTag_ {
    queue_chain_t   OSMT_link;
    uint32_t        OSMT_refcnt;
    uint32_t        OSMT_state; // OSMT_VALID or OSMT_RELEASED
    uint32_t        OSMT_attr;
    char            OSMT_name[OSMT_MAX_NAME];  // 64
} * OSMallocTag;

#define OSMT_DEFAULT   0x00
#define OSMT_PAGEABLE  0x01

extern OSMallocTag OSMalloc_Tagalloc(
    const char * name,
    uint32_t    flags);
..

void *
OSMalloc(
        uint32_t                        size,
        OSMallocTag                     tag)
```

Using a tag offers not only the ability to provide a meaningful name, but to also count the number of times it is used. When a tag is created with `OSMalloc_Tagalloc()`, subsequent allocations using `OSMalloc` increase its reference count. Further, flagging a tag with `OSMT_PAGEABLE` causes the `OSMalloc()` to call `kmem_alloc_pageable_external()`, rather than `kallog_tag_bt()`. Note, however, that the `OSMalloc` tags do not map to that `kalloc_tag*`, for which only `VM_KERN_MEMORY_KALLOC` is used. Unfortunately, there is no way to obtain the `OSMallocTag` of `OSMalloc()`ed memory, nor is there a way (outside of inspecting kernel memory) to obtain the list of tags.

The zones used by kalloc are only several of many more, used by the underlying **Zone Allocator**. Zone allocators are very popular across operating system kernels, though they are sometimes known by other names, for example, Linux's Slabs. A **zone** is defined as one or more sets of preallocated, virtually contiguous memory pages. Each zone has a predefined element size, and its pages can be used for obtaining elements of that stated size, but no other: zalloc() and its variants accept only the zone, but no size arguments. Preallocating pages and allocating elements in them allows compaction of memory space, especially in cases where the element size is much smaller than a page size - as the alternative would have been to consume a page per element, which would have been very wasteful. When elements are no longer needed, they are zfree()d back into their zone, and may be reused.

The kernel allocates a large chunk of its virtual memory for zones in zone_map, through a call to kmem_suballoc() made in zone_init() (osfmk/kern/zalloc.c). Just how large the zone_map is varies by pointer size and physical memory. vm_mem_bootstrap (which calls zone_init()) takes the zsize boot argument (if specified, in GB) as a base, or defaults to one quarter of physical memory, but no less than CONFIG_ZONE_MAP_MIN (specified in config/MASTER). For 64-bit architectures, whichever value used will further be increased by 50%, but clamped to no more than half of RAM (In other words, the zone_map is usually 3/8 of available RAM). For 32-bit architectures, it is not increased and further clamped by 1.5GB. Note, however, that the allocation is virtual, not physical - i.e. not all pages of the zone_map are guaranteed to be resident (until actually used).

Individual zones can then be allocated inside the zone_map. Each zone is initially allocated as one set of pages, using zinit() (also from osfmk/kern/zalloc.c). The function takes four arguments - the *size* of the element, the *max*imum size the zone can grow to, the *alloc_size* used for the initial allocation or during expansion, and a human readable name for the zone. The alloc_size is chosen so as to divide as cleanly as possible by the number of elements. The name is required so that mach_zone_info users (notably, the zprint(1) utility) can distinguish between the many dozens of zones*.

The zinit() call returns a zone_t, which is a pointer to a struct zone. This is a relatively simple structure, used to maintain the zone's free_elements list, pages metadata, its lock, zone_name, and a bitmap of its associated attributes. These are normally set by default during zone creation (i.e., in zinit()), but remain modifiable (only before the zone is used) through flags specified to zone_change().

Table 12-4: Flags settable by zone_change()

Attribute	Default	Meaning
Z_EXHAUST (1)	false	When zone is full no more elements can be allocated.
Z_COLLECT (2)	true	Zone applicable for garbage collection.
Z_EXPAND (3)	true	Zone may grow with subsequent allocations.
Z_FOREIGN (4)	false	May collect foreign elements, outsize zone map.
Z_CALLERACCT (5)	true	Allocations accounted to the caller.
Z_NOENCRYPT (6)	false	Mark zone memory as not requiring encryption.
Z_NOCALLOUT (7)	false	No asynchronous allocations.
Z_ALIGNMENT_REQUIRED (8)	false	Mark as alignment required (for CONFIG_KASAN)
Z_GZALLOC_EXEMPT (9)	false	Mark as untracked by the Guard Zone Allocator (CONFIG_GZALLOC)
Darwin 17		
Z_KASAN_QUARANTINE (10)	true	CONFIG_KASAN: Quarantine zfree()d elements
Z_TAGS_ENABLED (11)	false	VM_MAX_TAG_ZONES: Enable element tags (also requires -zt boot-arg)
Darwin 18		
Z_CACHING_ENABLED (12)	no	Enables zone to be used with new Zone Cache mechanism
Darwin 19		
Z_CLEARMEMORY (13)	no	bzero() new chunks (KMA_ZERO)

* - The human readable name as a fourth argument to a function both serves to uniquely symbolicate zinit(), as well as all of its callers - a useful method used by jtool2 --analyze when operating on kernelcaches.

Most zones are named after their element name (i.e. the corresponding `struct`), or very similarly (e.g. "tasks" containing `struct task`). Table 12-5 shows some of the important zones. Those initialized by `kmeminit()` are all Mach zones corresponding to BSD layer zones.

Table 12-5: Some of the zones defined as of XNU 4570

Zone Name	Allocated by	Used for
`kalloc.###`	`kalloc_init()` (osfmk/kern/zalloc.c)	Miscellaneous `kalloc` allocations
`tasks`	`task_init()` osfmk/kern/task.c)	Mach Task objects (q.v. Chapter 9)
`thread`	`thread_init()` osfmk/kern/thread.c)	Mach thread shuttles (q.v. Chapter 9)
`ipc spaces`	`ipc_init()` (osfmk/ipc/ipc_init.c)	Task `ipc_space` objects
`ipc ports`		`ipc_port` objects
`proc`	`kmeminit()` (bsd/kern/kern_malloc.c)	BSD processes (q.v. Chapter 6)
`uthread`		BSD threads (q.v. Chapter 6)
`mount`		Mounted filesystems (q.v. Chapter 7)
`fileglob`		Kernel file representation(q.v. Chapter 6)
`fileproc`		Process files (q.v. Chapter 6)
`file desc`		Process file descriptors (q.v. Chapter 6)
`vnodes`		Vnodes (q.v. Chapter 7)

Once a zone is `zinit()`ed, elements can easily be obtained from it by a call to `zalloc()`. The function takes a single argument - the zone from which to allcoate, and thus ensures the size is as was predetermined by `zinit()`. `zalloc()` calls `zalloc_internal()`, which also controls whether the allocation `canblock`, needs to wait for new pages (`nopagewait`), a `reqsize` indicating how much of the allocation will actually be used, and a memory `tag`. Multiple `zalloc_*` variants exist to wrap these arguments, although their use is uncommon.

When a zone nears exhaustion or is empty, it may be expanded (if `expandable`). Zones marked `async_prio_refill` (presently, `Reserved.VM.map.entries` and `VM.map.holes`) are replenished asynchronously by the `zone_replenish_thread()` when they fall below `prio_refill_watermark`. These must be also be marked to `allow_foreign`, as memory might be allocated for them outside the zone map, if the zone map is out of space. All other zones marked `expandable` may be expanded in `zalloc_internal()`.

Both expansion and repleneshing operations involve `kernel_memory_allocate()`ing a new chunk of the specified zone's `alloc_size` (or, if memory is low, at least one element, rounded up to a page size), and then "cramming" it into the zone, using the `zcram()` routine, which also updates the metadata for the new pages accordingly. There is also a `zfill()` function, which `kernel_memory_allocate()`s a chunk large enough to fill a requested number of elements. The `os_reason_init()` routine (in bsd/kern/sys_reason.c) uses this to ensure the `os_reason_zone` has enough memory, even during jetsam events (which require specifying an `os_reason` object).

The `zprint(1)` command is a highly useful utility to glean information about the state of the zones. `zprint(1)` uses the `mach_[zone|memory]_info` (`mach_host` subsystem #220 and #227) and `task_zone_info` (#3428 in the `task` subsystem) MIG routines. The former MIG calls produce the zone listing, and although part of `mach_host` and not `host_priv` they nonetheless requires `root` privileges. There is also a `mach_zone_info_for_zone()` MIG routine (`mach_host` #231).

`zprint(1)` is part of the system_cmds project, but is not part of the *OS binpack because Apple already provides a signed (and entitled!) binary on *OS, intended to be used as part of the `sysdiagnose(1)` process. As of later *OS variants, however, Apple restricts `mach_memory_info()` through a `#if CONFIG_DEBUGGER_FOR_ZONE_INFO`, which refuses this functionality unless `PE_i_can_has_debugger(NULL)` (=`debug_enabled`) is true, rendering the command useless.

Using `zprint(1)` is a great way to find the `sizeof()` of some common structures, especially those which keep changing in between Darwin versions.

Zone Management

It used to actually take a zone in order to manage zones. Prior to setting up the kernel zones, a special "zone of zones" was created by a call to `zone_bootstrap()`, prior even to `zone_init()`, which sets up the `zone_map`. This practice also supported "fake zones", which were memory regions (e.g. kernel stacks) that `mach_zone_info()` would report to user mode.

As of Darwin 16, the `zones` zone has been removed, and `zone_bootstrap()` instead sets up a `zone_array`, which is statically allocated to accommodate up to `MAX_ZONES struct zone` entries. The value was initially 256, but has grown (around Darwin 17.2) to 320, where it remains at this time. The `zone_empty_bitmap` tracks which zones are empty (i.e., it is initially set to all '1's), allowing destroyed zones to be reused. The number of used entries (= '0' bits) in the bitmap tracked with the `num_zones_in_use` variable, and the number of overall zones created with `num_zones`. The zone names themselves are stored (NULL-terminated and concatenated to one another) on a dedicated page, allocated by `kmem_alloc_kobject` and pointed to by `zone_names_start`. An additional pointer, `zone_names_next`, points to the end of the last zone name in the page.

Zone memory is taken from the `zone_map`. This is a `vm_map` which spans from `zone_map_min_address` to `zone_map_max_address`. At the beginning of the `zone_map` is a special area called the **zone metadata region**, through which individual pages can be tracked to their allocating zones. The metadata spans from `zone_metadata_region_min` (usually equal to `zone_map_min_address`) to `zone_metadata_region_max`.

Table 12-6 shows all the zone-related variables, including those described above, and others we will encounter soon. MacOS XNU exports these symbols, but the *OS kernels do not - although all are readily identified by `joker`.

Table 12-6: Zone related variables (all defined in osfmk/kern/zalloc.c)

Variable	Holds
`zone_map`	A `vm_map_t` holding the actual virtual memory used by the all zones
`all_zones_lock`	A `simple_lock` guarding access to the `zone_array`, `num_zones[_in_use]` and the `zone_empty_bitmap`
`zone_map_[min/max]_address`	`vm_offset_t`s specifying beginning and end of zone map
`zone_metadata_region_[min/max]`	`vm_offset_t`s specifying beginning and end of metadata
`zone_metadata_region_lck`	A `mtx_lck_t` protecting the zone metadata region contents
`zone_array`	An array of up to `MAX_ZONES` (presently, 320) `struct zone` entries
`num_zones`	Tracks zones created in `zone_array` (incremented by `zinit`)
`num_zones_in_use`	Tracks non-empty zones (as `num_zones`, decremented by `zdestroy`)
`zone_empty_bitmap`	Bitmap of `MAX_ZONES` (in practice, `num_zones`) bits, tracking use

As an example (from MacOS 14.3), consider the following:

```
bash-3.2# xnoop dump _zone_map_min_address,8
0xffffff801449c950    0xffffff801a038000
bash-3.2# xnoop dump _zone_map_max_address,8
0xffffff801449c958    0xffffff804a038000
bash-3.2# xnoop dump _sane_size,8
0xffffff80145e7408 00 00 00 80 00 00 00 00
```

Working the math and taking the difference between the `zone_map_max_address` and the `zone_map_min_address` we'd get 0x30000000. This is in line with what would be expected from the source, since this value is a three eighths of `sane_size` (as set up by `vm_mem_bootstrap()` for `__LP64__` kernels.

The `zone_metadata_region`

Maintaining metadata for zones is a daunting challenge. On the one hand, it must be done as efficiently as possible. On the other, zones are frequent target for exploitation in controlled kernel memory overwrites, and therefore efficiency should not come at the cost of security. Apple has continuously been modifying zone management, with the latest redesign in Darwin 16 and above.

As of Darwin 16, per page zone metadata has been moved into its very own `zone_metadata_region`. Bound between `zone_metadata_region_min` and `..max`, this is a large array of `struct zone_page_metadata`. Each of these is a fixed size element, so it follows that a formula can be used, given a memory address, to find its zone metadata. First, the page index needs to be found, for which the `PAGE_INDEX_FOR_ELEMENT` macro is used. Listing 12-7 shows this macro:

Listing 12-7: The `PAGE_INDEX` macros in osfmk/kern/zalloc.c

```
/* Macro to get page index (within zone_map) of page containing element */
#define PAGE_INDEX_FOR_ELEMENT(element)                                     \
  (((vm_offset_t)trunc_page(element) - zone_map_min_address) / PAGE_SIZE)

/* Macro to get page for given page index in zone_map */
#define PAGE_FOR_PAGE_INDEX(index)                                          \
  (zone_map_min_address + (PAGE_SIZE * (index)))
```

Recall, that the zone map includes its own pages - and all other zones, whose pages are contiguous in virtual memory. A page's index can therefore be found by taking the rounded page address of the element (bitmasked with the inverse of `PAGE_MASK`, as is performed by the `trunc_page()` macro) and subtracting it from `zone_map_min_address`, then dividing that difference by the `PAGE_SIZE`. Of course, this operation is fully reversible, so the inverse macro, `PAGE_FOR_PAGE_INDEX`, is used in those cases.

Index at hand, finding the metadata is as straightforward as looking at the entry at that index in the array starting at `zone_metadata_region_min` whose elements are `struct zone_page_metadata`. Indeed, this is what the `PAGE_METADATA_FOR_PAGE_INDEX` macro does. This, too, has an inverse operation, and both are shown in Listing 12-8:

Listing 12-8:

```
/* Macro to get metadata structure given a page index in zone_map */
#define PAGE_METADATA_FOR_PAGE_INDEX(index)                                 \
  (zone_metadata_region_min + ((index) * sizeof(struct zone_page_metadata)))

/* Macro to get index (within zone_map) for given metadata */
#define PAGE_INDEX_FOR_METADATA(page_meta)                                  \
 (((vm_offset_t)page_meta - zone_metadata_region_min) / sizeof(struct zone_page_metadata))
```

Now let's continue our example with an arbitrary address - say, that of the `kernel_task`:

```
bash-3.2# jtool -S /System/Library/Kernels/kernel | grep kernel_task$
ffffff8000c9c218 S _kernel_task
# Apply slide:
bash-3.2# xnoop sdump 0xffffff8000c9c218,8
0xffffff801449c218     0xffffff801aa4d280
```

The `kernel_task` export is, once slid, at 0xffffff801449c218 - which is outside the zone map. This is as it should be, because the `_kernel_task` export is in the `__DATA.__common`. But the `kernel_task` is a `task_t` - i.e. a pointer, and the `struct task` it points to is at `0xffffff801aa4d280` - well within the zone map. To find the page index of this address, we need to manually apply the `PAGE_INDEX_FOR_ELEMENT` macro calculation:

```
PAGE_INDEX_FOR_ELEMENT(0xffffff801aa4d280) =
        (0xffffff801aa4d000 - 0xffffff801a038000) / 0x1000 = 0xa15
```

Getting the metadata for a zone element at a given address is therefore a two step operation, although it can be combined into a larger macro - which is exactly what `PAGE_METADATA_FOR_ELEMENT` achieves.

Continuing our example of locating the page holding the `kernel_task` structure, which was in page 0xa15. Where is the metadata for this zone? First, we find the metadata region, which conveniently overlaps with the beginning of the zone map*:

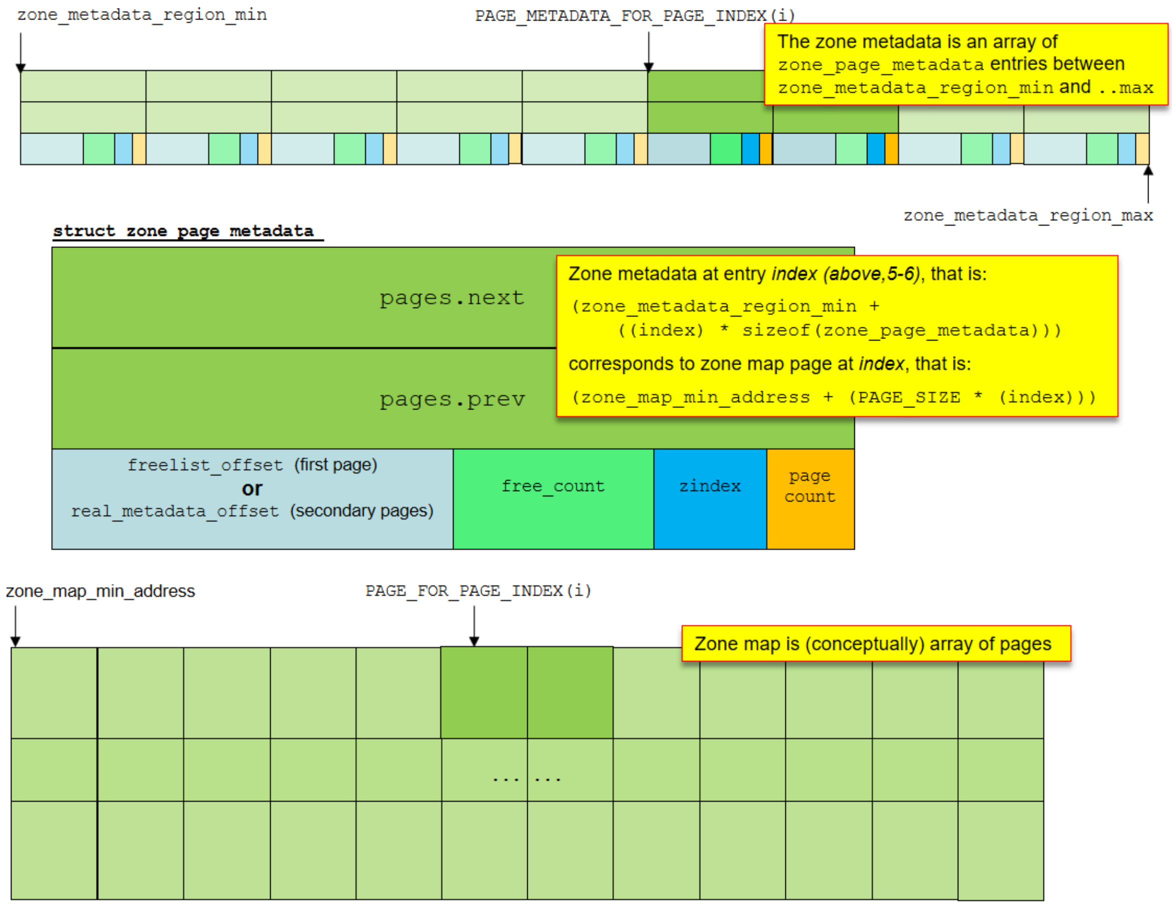
```
bash-3.2# xnoop dump _zone_metadata_region_min,8
0xffffff801449c960    0xffffff801a038000
bash-3.2# xnoop dump _zone_metadata_region_max,8
0xffffff801449c968    0xffffff801a4b8000
```

The `sizeof(struct zone_page_metadata)` is 24 (= 0x18). This means that the metadata entry for page 0xa15 can be found by:

```
PAGE_METADATA_FOR_PAGE_INDEX(0x1a5) =
= 0xffffff801a038000 + (0x1a5 * 0x18) = 0xffffff801a03a778
```

Figure 12-9 shows a graphic example. In it, pages #5 and #6 of the zone map (counting from 0) are highlighted, as is their metadata. Finding the metadata and the page, given its index, is a direct application of the formulae shown in this section.

Figure 12-9: Zone metadata and the zone map



There is an important exception to this type of metadata maintenance: Foreign allocations (which, you'll recall, are from outside the `zone_map`). Though rare, these may be allocated by the `zone_replenish_thread()` can't allocate from the `zone_map` so it falls back to the `kernel_map`. Such allocations obviously won't work with this scheme (as there is no valid `PAGE_INDEX_FOR_ELEMENT`), so metadata instead is stored on the allocation itself, which is limited to one page.

---

* - Actually, not that conveniently.. Attempting to read from the very beginning of the zone metadata region will fail, and may panic the kernel! In fact, pages #5 and #6 shown in the illustration and chosen for simplicity, are also similarly unreadable, with the metadata becoming safe to read only later on (in a case handled by `get_zone_page_metadata()`). The reason why is left to ponder as a review question.

The zone metadata

Thus far, we've established pages belonging to zones (and only those pages) have corresponding metadata elements in the `zone_metadata_region`. When a zone claims pages (during its creation or expansion), the metadata of these pages is made resident (by `kernel_memory_populate()`). But what exactly is maintained in the metadata? Listing 12-10 shows the structure definition (from osfmk/kern/zalloc.c), as did Figure 12-9 (in the previous page).

Listing 12-10: The `struct zone_page_metadata` (from osfmk/kern/zalloc.c)

```
struct zone_page_metadata {
  queue_chain_t   pages; /* linkage pointer for metadata lists */

  /* Union maintaining start of element free list and real metadata (for multipage allocations) */
 union {
  /*
  * The start of the freelist can be maintained as a 32-bit offset instead of a pointer because
  * the free elements would be at max ZONE_MAX_ALLOC_SIZE bytes away from the metadata. Offset
  * from start of the allocation chunk to free element list head.
  */
  uint32_t   freelist_offset;
  /*
  * This field is used to lookup the real metadata for multipage allocations, where we mark the
  * metadata for all pages except the first as "fake" metadata using MULTIPAGE_METADATA_MAGIC.
  * Offset from this fake metadata to real metadata of allocation chunk (-ve offset).
  */
  uint32_t   real_metadata_offset;
  };

  /*
  * For the first page in the allocation chunk, this represents the total number of
  * free elements in the chunk.
  */
  uint16_t    free_count;
  unsigned    zindex     : ZINDEX_BITS;    /* Zone index within the zone_array */
  unsigned    page_count : PAGECOUNT_BITS; /* Count of pages within the allocation chunk */
};
```

Prior to Darwin 17 the number of bits in zindex was fixed to 8 - which caused a problem with zone indices greater than 254. After increasing `MAX_ZONES` past 256 (in Darwin 17), `zindex` was allowed to "borrow" two bits from `page_count` (i.e. `ZINDEX_BITS` is now 10), which is fine considering that zones chunks have a small number of pages.

The `pages` queue chain is a pointer to the next (and previous) metadata for another zone chunk, or (if there are no more) a pointer to the `struct zone`'s corresponding chunk list. Each `struct zone` presently maintains four such lists (`queue_head_t`s):

- **`any_free_foreign`**: A list of foreign pages crammed into the zone. These are outside the zone map, and therefore have the metadata embedded in them. This is only applicable for zones which explicitly `allow_foreign` (i.e. have `Z_FOREIGN` set via `zone_change()`.
- **`all_free`**: A list of chunks that are either freshly allocated or, over time, had all their elements `zfree()`d. These are candidates to be picked up in the next garbage collection.
- **`intermediate`**: A list of chunks in which at least one element is free, but also at least one element is in use.
- **`all_used`**: A list of chunks in which all elements are used. These chunks have a `free_count` of 0, and a `freelist_offset` of `0xffffffff`.

Chunks are moved between the list based on the `free_count`, which is incremented on `try_alloc_from_zone()` and decremented on `free_to_zone()`.

Element Free Lists

Even if we assume for a minute that zones start up empty and elements are added linearly (one after the other), soon enough (as elements are freed) it is inevitable that "holes" form in zones over the freed elements. To maintain efficiency, these free elements need to be tracked so that they can be reallocated. The way to do that is to maintain element **free lists**. The zone metadata maintains a 32-bit `freelist_offset`, from the start of the allocation chunk to the freelist's head.

Using an offset instead of a pointer is advantageous in that it saves four bytes per element. For other pages within the same chunk, there's no benefit in saving any freelist reference - as it can be walked from the first page anyway. There is, however, a need to quickly find the real metadata. Once again, this is best served by an offset. Thus, for subsequent pages inside the same allocation chunk, this field is repurposed (via a union) to point to the `real_metadata_offset`.

Also, assuming linear addition of elements is no longer valid. After allocating a zone chunk, `zcram()` introduces freelist randomization: `random_free_to_zone()` is called to free the chunk's elements and splice the free list by progressively adding elements from the beginning or the end of the page. Entropy is generated using `random_bool_gen_bits()` (implemented in osfmk/prng/prng_random.c, which is the same PRNG used in the IPC space (port name) entropy.

Listing 12-11: The `random_free_to_zone()` from XNU-6153's osfmk/kern/zalloc.c

```
#define MAX_ENTROPY_PER_ZCRAM          4

static void
random_free_to_zone(
        zone_t          zone,
        vm_offset_t     newmem,
        vm_offset_t     first_element_offset,
        int             element_count,
        unsigned int    *entropy_buffer)
{
        vm_offset_t     last_element_offset;
        vm_offset_t     element_addr;
        vm_size_t       elem_size;
        int             index;

        assert(element_count && element_count <= ZONE_CHUNK_MAXELEMENTS);
        elem_size = zone->elem_size;
        last_element_offset = first_element_offset + ((element_count * elem_size) - elem_size);
        for (index = 0; index < element_count; index++) {
                assert(first_element_offset <= last_element_offset);
                if (
#if DEBUG || DEVELOPMENT
                    leak_scan_debug_flag || __improbable(zone->tags) ||
#endif /* DEBUG || DEVELOPMENT */
                    random_bool_gen_bits(&zone_bool_gen, entropy_buffer, MAX_ENTROPY_PER_ZCRAM, 1))
                    element_addr = newmem + first_element_offset;
                    first_element_offset += elem_size;
               } else {
                    element_addr = newmem + last_element_offset;
                    last_element_offset -= elem_size;
                }
                if (element_addr != (vm_offset_t)zone) {
                    zone->count++;  /* compensate for free_to_zone */
                    free_to_zone(zone, element_addr, FALSE);
                }
                zone->cur_size += elem_size;
        }
}
```

The Listing above shows how `random_free_to_zone()` determines where to free to, but the actual splicing of the free list is performed by `free_to_zone()`. This routine adds the `zfree()`d element to the head of the chunk free list, with or without a "poison", as discussed shortly.

Experiment: Viewing zones and metadata in memory

The following example illustrates a simple zone layout in memory. To remain efficient, the example uses `xnoop(j)`, but can also be followed with the help of a simple memory dumper (per a D-script in MacOS or with a `kernel_task` `mach_vm_read()`) in jailbroken *OS, following `jtool2 --analyze` for symbolication.

We start by getting the preliminary values we'll need to perform some zone-arithmetics:

Output 12-12-a: Preliminary values needed for zone math

```
root@Bifröst (~)# xnoop syms slid | grep zone_array
0xffffff8001aa3520 14 _zone_array
root@Bifröst (~)# xnoop dump _zone_metadata_region_min,16
0xffffff8001aa3500    0xffffff802f05f000  # _zone_metadata_region_min
0xffffff8001aa3508    0xffffff803145f000  # _zone_metadata_region_max
```

Let's look at the zone at index 6. Given the size of a zone in MacOS 15 is 0x130, that will have us at `0xffffff8001aa3c40`. The reader is highly encouraged to follow along this example with the `zonedump.d` D-Script (available in the Book's website as a listing), or even on *OS, crafting a zone dumper as further exercise.

Output 12-12-b: Showing the `pmap` zone data structure, as an example

```
root@Bifröst (~) # xnoop dump 0xffffff8001aa3c40 zone
zone@0xffffff8001aa3c40:
   zcache(@0x0): NULL            # Not cached
   free_elements(@0x8):  NULL  # Unused
   pages.any_free_foreign.next(@0x10): 0xffffff8001aa3c50  # empty (!allow_foreign)
   pages.any_free_foreign.prev(@0x18): 0xffffff8001aa3c50  # empty (!allow_foreign)
   pages.all_free.next(@0x20):         0xffffff8001aa3c60  # empty
   pages.all_free.prev(@0x28):         0xffffff8001aa3c60  # empty
   pages.intermediate.next(@0x30):     0xffffff802f3f5ca8
   pages.intermediate.prev(@0x38):     0xffffff802f3b0000
   pages.all_used.next(@0x40):         0xffffff8001aa3c80  # Empty
   pages.all_used.prev(@0x48):         0xffffff8001aa3c80  # Empty
   count(@0x50): 542            countfree(@0x54): 98
   count_all_free_pages(@0x58): 0
   cur_size(@0xb8): 286720      max_size(@0xc0): 270336
   elem_size(@0xc8): 448        alloc_size(@0xd0): 28672
   page_count(@0xd8): 0x46      sum_count(@0xe0): 0x50a67
   exhaustible(@0xe8): false    collectable(@0xe8): true
   expandable(@0xe8): true      allows_foreign(@0xe8): false
   doing_alloc_without_vm_priv(@0xe8): false   doing_alloc_with_vm_priv(@0xe8): false
   waiting(@0xe8): false                async_pending(@0xe8): false
   zleak_on(@0xe9): false               caller_acct(@0xe9): true
   noencrypt(@0xe9): true               no_callout(@0xe9): false
   async_prio_refill(@0xe9): false gzalloc_exempt(@0xe9): false
   alignment_required(@0xe9): false zone_logging(@0xe9): false
   zone_replenishing(@0xea): false kasan_quarantine(@0xea): true
   tags(@0xea): false           tags_inline(@0xea): false
   tag_zone_index(@0xea): 0     zone_valid(@0xeb): true
   cpu_cache_enable_when_ready(@0xeb): false    cpu_cache_enabled(@0xeb): false
   clear_memory(@0xeb): false   zone_destruction(@0xeb): false
   index(@0xf0): 6              zone_name(@0xf8): pmap
   ...
```

We see that the zone is `pmap`, and serendipitously it is a pretty simple zone, as it only has the `intermediate` list active (which is why it was chosen for the example), with the other lists pointing to their own address (at offsets +0x10, +0x20 and +0x40). We can walk the intermediate list, again using `xnoop`:

Output 12-12-c: Walking the `pmap` intermediate list

```
root@Bifröst (~) # xnoop  walk zone | grep pmap
Zone @0xffffff8001aa3c40: pmap Index: 6
     Size: 0x46000/0x42000, Element Size: 448, alloc_size: 0x7000 Using 70 pages
pmap Intermediate: 0xffffff802f3f5ca8 (0xffffff80554e6000-0xffffff80554ed000) - 29 free
pmap Intermediate: 0xffffff802f2af260 (0xffffff8050e43000-0xffffff8050e4a000) - 8 free
pmap Intermediate: 0xffffff802f260de0 (0xffffff80446f3000-0xffffff80446fa000) - 4 free
pmap Intermediate: 0xffffff802f15f938 (0xffffff8039b6c000-0xffffff8039b73000) - 14 free
pmap Intermediate: 0xffffff802f1cac00 (0xffffff803e2df000-0xffffff803e2e6000) - 4 free
pmap Intermediate: 0xffffff802f291110 (0xffffff8046715000-0xffffff804671c000) - 5 free
pmap Intermediate: 0xffffff802f38bd60 (0xffffff8050e43000-0xffffff8050e4a000) - 12 free
pmap Intermediate: 0xffffff802f3e38a0 (0xffffff80548bb000-0xffffff80548c2000) - 10 free
pmap Intermediate: 0xffffff802f096fc8 (0xffffff80315b2000-0xffffff80315b9000) - 6 free
pmap Intermediate: 0xffffff802f3b0000 (0xffffff805265f000-0xffffff8052666000) - 6 free
```

Note, that even though the intermediate list chunks are not in ascending order, the math adds up. The count of all free elements across chunks is the same as `countfree`. Each chunk is 0x7000 bytes, and so the total number of chunks is 70, same as the zone's `page_count`.

Garbage Collection

As we've established, freeing zone elements results in holes. If such holes are large enough so as to encompass an entire page, the zone can be compacted and the page freed for re-use, possibly by another zone. There is thus a need to collect "garbage" pages periodically, or on low memory conditions.

Garbage collection is performed by calling `consider_zone_gc()`, with a boolean to `consider_jetsams`. The "consideration" is allowing for a special case reclaiming early boot memory (from `kmapoff_kaddr`, as discussed in Chapter 5) and otherwise checking that `zone_gc_allowed` is set to TRUE, which it always is. The calls to `consider_zone_gc()` are made from the vm_pageout thread's `vm_pageout_garbage_collect`, and from `vm_page_find_contiguous()`, on failure to find pages.

The actual garbage collection is performed by `zone_gc()`, which may first call `kill_process_in_largest_zone()` if `consider_jetsams` was true, as discussed later. `zone_gc()` then acquires the `zone_gc_lock` and iterates over all zones marked as `collectable` in the `zone_array`, calling `drop_free_elements()` if they have pages in their `all_free` queue. Cacheable zones (as of Darwin 18) also have their depots drained, as discussed later. The garbage collection occurs under the `zone_gc_lock`, which ensures only one concurrent garbage collection can take place. Throughout the process the thread's `options` flags `TH_OPT_ZONE_GC`, marking the thread as garbage collecting (and also avoid a potential deadlock with the zone replenish thread).

`drop_free_elements()` locks the zone it operates on, in order to "snatch" its `all_free` queue, replacing it with an empty queue. The (now detached) queue is iterated over once to determine its size and element count, and then the zone is locked again briefly so it can be adjusted accordingly. The detached queue is then iterated over again, this time dequeueing each page chunk in turn, and then calling `kmem_free()` to free the page from the `zone_map`.

Iterating over all the zones' free lists in this manner can be a very long operation, so after every such free operation a call to `thread_yield_to_preemption()` is made (allows possible preemption (for a pending `AST_PREEMPT`, as discussed in Chapter 9). Since `drop_free_elements()` may be called from `zdestroy`, the preemption check is made only as part of a `zone_gc()`, as determined by the aforementioned `TH_OPT_ZONE_GC` option.

GC and UAF

Garbage collection, however, also proves to be an instrumental step in exploitation, as part of the "Feng Shui" required to channel the Qi of exploitation. Rather than thinking about it as "garbage collection" in this context, it helps to consider "memory recycling" - The memory freed following garbage collection can be reused by the system for entirely different purposes. Memory which previously represented a given object in some zone may be repurposed and later used by another object (of the same or of a different type) in some other (or the same) zone.

If a Use-After-Free condition can be triggered, a user mode attacker can cause the reference count of an object to drop to zero, while still nonetheless holding a reference to the object in user mode. If the attacker further has the ability to control write operations to the object's memory after it is repurposed (for example, by spraying fake content in a Mach message OOL descriptor), the object (usually, an `ipc_port`, or `IOUserClient`) can be entirely controlled. Specific examples of these attacks can be found throughout Volume III.

What made this type of exploitation far easier was that garbage collection could be trigged from user mode, by calling `mach_zone_force_gc` (MIG message #221 of the `mach_host` subsystem). The call was synchronous, so it was guaranteed any reclamation of pages would be complete when it returned. Apple eventually figured out this is a security concern, and removed the call (outside of `DEBUG`/`DEVELOPMENT`) as of Darwin 17. Garbage collection can still be triggered, however, by causing the rapid allocation of many kernel objects, or sending (but not receiving) many Mach messages. Doing so will first fill up any `intermediate` lists, then lead to more chunk alocations. Destroying the objects or receiving the messages results in freeing the respective zone elements, and reclaims the entire free pages, after which they may be repurposed. Although the operation is nowadays performed asynchronously, the interested caller can simply delay execution sufficiently for collection to reliably complete.

Battling zone corruption

In a perfect world, zone metadata, free lists and allocations could be trusted, since they are in kernel memory. In the cruel, far-from-perfect world of XNU, however, nefarious hackers and jailbreakers find new vulnerabilities leading to kernel memory corruption. Apple has reworked the metadata several times, before settling (for now) on the Darwin 16 and later approach described earlier.

A common exploitation method (exploited, for example, by the TaiG jailbreak, as discussed in III/18) involved freeing an element (using `zfree()` or, as of Darwin 18, the quicker `zfree_direct()`) into the wrong zone. Either variant requires both the zone pointer and the element to be freed, but it is only from Darwin 16 that such `zfree()` operations (but not `zfree_direct()`s, used by the zone cache) are reliably intercepted, thanks to the new zone metadata layout.

During the free operation the element may or may not be "poisoned", by `memset()`ing with `ZP_POISON` (`0xdeadbeefdeadbeef`). The `zfree()` variants both call element poisoning code (refactored in Darwin 18 into the `zfree_poison_element()` routine). Doing so for every element, however, is quite costly, so concessions have to be made:

- Zones whose element sizes is equal to or less than `zp_tiny_zone_limit` always get poisoned. This value is set by `zp_init()` to the CPU's cache line size (`cpu_info.cache_line_size`). The `-no-zp` boot argument sets this value to zero, thereby disabling poisoning altogether.
- For larger zones, `zp_factor` and `zp_scale` govern the frequency of poisoning. These are initially set to `ZP_DEFAULT_[SAMPLING/SCALE]_FACTOR` (16 and 4, respectively), and the default factor is further permuted in one out of every two cases by +/-1 (as determined by two bits from `early_random()`. The `zp-factor` and `zp-scale` boot arguments (with dashes, not underscores..) can override these values. Whichever way they are set, `sample_counter()` tracks the freed zone's `zp_count`, and possibly poisons according to them, with the `zp_scale` providing a right logical shift for the element size. This means that larger elements are less likely to be poisoned, and the `zp_scale` can control the frequency of poisoning. Setting the `zp_factor` to 0 effectively disables this poisoning, and setting it to 1 (or setting the `-zp` boot argument, which does so as well) poisons every operation.

There are further integity checks for zone pointers in the free list, rolled up into `is_sane_zone_ptr()`. The current criteria mandate alignment to pointer boundary, a kernel address, and pointing to somewhere in the zone map (unless the zone allows foreign elements).

Darwin 19 adds a significant improvement with `zone_require (`*`address`*`, `*`zindex`*`)`. Prior to dereferencing an object pointer, this call ensures that the *address* belongs to the zone at *zindex*. This effectively eliminates a common technique of UaF/GC in which fake objects (mostly `ipc_port`s, but potentially tasks, procs, etc) could be constructed (by parking `mach_msg`s and OOL descriptors in kernel). iOS 13.2 further laces most port to kobject conversion checks with calls to `zone_require()`.*

The Guard Mode Zone Allocator (MacOS)

MacOS `#define`s the `CONFIG_GZALLOC` setting, which enables the "Guard Mode" zone allocator. When set, this makes `zalloc_internal` first call to `gzalloc_alloc()`, rather than the zone cache or the traditional zone allocation. Guarded zone allocations then behave very similarly to the way `libgmalloc(3)` (Guard Malloc) allocations do in user mode, to detect use of uninitialized data or potential overflows, but in kernel mode. It does so by `memset()`ting a pattern ('g') on free and adjoining guard pages (protected to `PROT_NONE`) to allocations.

---

* - The author is befuddled by the fact that `zone_require()` in *OS up to and including 13.3 does not `panic()` if the address is not in a zone, despite the open sources (of XNU-6153.11.26) seeming to indicate it does. At any rate, this "minor" oversight enabled Brandon Azad's "oob_timestamp" exploit technique for 13.3.[1]

Similar to libgmalloc(3), the Guard Mode zone allocator can be configured to detect underflows, rather than overflows, and other aspects of allocation behavior can be tweaked. This is done by passing the following boot arguments:

Table 12-13: The boot arguments processed by the Guard Mode zone allocator

-[no]gzalloc_mode	Enables the allocator on all zones, or disables. Default is disabled
gzalloc_[min|max]	Target only zones with elements between min and max (default: none)
gzalloc_uf_mode	Protect underflows, rather than overflows
gzalloc_fc_size	Free element cache size
gzname	Zone to target, by name
-gzalloc_wp	Set guard page permissions to allow read
-gzalloc_no_dfree_check	Disable double free check (default: check)
-gzalloc_noconsistency	Disable consistency checks (default: check)

The guard zone allocator also adds gzalloc_data_t metadata to every zone. This is a simple structure, containing an array of addresses (gzfc), and an index holding its active size. Using this allocator is far more reliable than the probabilistic poisoning, but does waste significant memory. It therefore only applies to gzalloc_tracked() zones, which presently consist of zones whose element sizes fall between the gzalloc_[min|max], or the particular zone matched by the gzname argument. This is only if the zones in questions aren't marked by gzalloc_exempt (through the Z_GZALLOC_EXEMPT flag of zone_change()). The default value of gzalloc_min is greater than the max, so unless changed (or -gzalloc_mode is explicitly stated), no zones (but the possibly named one) will be tracked.

The Zone Cache (Darwin 18+)

Darwin 18 adds a new layer on top of the zone allocator, called the **zone cache**. The layer draws on academic research, and aims to make zone allocations more efficient and scalable across multiple CPUs. Listing 12-14 (next page) shows the verbose description of the zone caching mechanism, from osfmk/kern/zcache.h. Readers remembering the discussion of the user mode magazine allocator (I/8) will likely be able to find the strong parallels between the two.

Zone caching is contingent on CONFIG_ZCACHE being defined, though that is true across all Darwin 18 flavors. Additionally, it requires either specific zone opt-in by specifying the zcc_enable_for_zone_name= boot-arg, global enablement by -zcache_all, or specific opt-in by calling zone_change() with the Z_CACHING_ENABLED flag (presently set only for ipc_kmsg_zone). When caching is enabled for a zone, zcache_init() is called on it, initializing a per-CPU cache for it and setting the zone's cpu_cache_enabled() field. Zones zinit()ed before the zone cache is ready are tagged through their cpu_cache_enable_when_ready field, so that zone_bootstrap() picks up the marking and zcache_init()s them.

zone_caching_enabled(*zone*) checks the criteria on a per zone basis which is that the zone is marked with cpu_cache_enabled, and that the zone is not tagged or followed by zleaks. If met, zalloc_internal() is diverted to zcache_alloc_from_cpu_cache, and likewise zfree() is diverted to call zcache_free_to_cpu_cache().

The magazines are kept in their own zone (zcc_magazine_zone). The zone cache also uses its own zcache_canary, with an early_random() value set by zcache_bootstrap(). The canary is added at the beginning and end of each element when freed to the CPU cache (or when the magazine is filled), and validated when elements are allocated from the cache (or the magazine is drained). This provides another way to intercept potential use after free. When draining the magazine, after the canary is validated the element is freed through zfree_direct(), as a lightweight version of zfree() which skips the cumbersome checks of zfree()ing to the wrong zone.

Listing 12-14: Zone caches (from XNU 4903's osfmk/kern/zcache.h)

```
/*      Below is a diagram of the caching system. This design is based of the
 * paper "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and
 * Arbitrary Resources" by Jeff Bonwick and Jonathan Adams. It is divided into 3
 * layers: the Per-cpu Layer, the Depot Layer, and the Zone Allocator. The
 * Per-CPU and Depot layers store elements using arrays we call magazines.
 *
 *      Magazines function like a stack (we push and pop elements) and can be
 *  moved around for bulk operations.
 *  _________         _________         _________
 * |  CPU 1  |       |  CPU 2  |       |  CPU 3  |
 * |  _   _  |       |  _   _  |       |  _   _  |
 * | |#| | | |       | | | |#| |       | |#| |#| |        Per-CPU Layer
 * | |#| |_| |       | |_| |#| |       | |#| |#| |
 * |_________|       |_________|       |_________|
 *
 *  ______________________________________________
 * |           _   _   _   _   _   _               |
 * |          |#| |#| |#| | | | | | |              |     Depot Layer
 * |          |#| |#| |#| |_| |_| |_|              |
 * |_______________________________________________|
 *
 *  _______________________________________________
 * | # | # | # | # | # | # | # | # | # | # | # | # |   Zone Allocator
 * |_______________________________________________|
 *
 *      The top layer is the per-cpu cache and consists of a current and
 * previous magazine for each CPU. The current magazine is the one we always try
 * to allocate from and free to first. Only if we are unable, do we check the
 * previous magazine. If the previous magazine can satisfy the allocate or free,
 * then we switch the two and allocate from the new current magazine. This layer
 * requires no locking, so we can access multiple CPU's caches concurrently.
 * This is the main source of the speedup.
 *
 *      We have two magazines here to prevent thrashing when swapping magazines
 * with the depot layer. If a certain pattern of alloc and free are called we
 * can waste a lot of time swapping magazines to and from the depot layer. We
 * prevent this by dividing the per-cpu cache into two separate magazines.
 *
 *      The middle layer is the magazine depot. This layer consists of a
 * collection of full and empty magazines. These are used to reload the per-cpu
 * caches when needed. This is implemented as an array of magazines which are
 * initially all empty and as we fill up magazines we increment the index to
 * point at the first empty magazine. Since this layer is per-zone, it allows us
 *  to balance the cache between cpus, but does require taking a lock.
 *
 *      When neither the current nor previous magazine for a given CPU can
 * satisfy the free or allocation, we look to the depot layer. If there are
 * magazines in the depot that can satisfy the free or allocation we swap
 * that magazine into the current position. In the example below, to allocate on
 * the given CPU we must lock the depot layer and swap magazine A with magazine
 * B and decrement the depot index.
 *
 *      _____________________       _______________________________________
 *     |    Per-CPU Cache    |     |              Depot Layer              |
 *     |                     |     |                                       |
 *     |   A___      ____    |     |     ____      B___      ____     ____ |
 *     |  |    |    |    |   |     |    | ## |    | ## |    |    |   |    ||
 *     |  |    |    |    |   |     |    | ## |    | ## |    |    |   |    ||
 *     |  |    |    |    |   |     |    | ## |    | ## |    |    |   |    ||
 *     |  |    |    |    |   |     |    | ## |    | ## |    |    |   |    ||
 *     |  |____|    |____|   |     |    |_##_|    |_##_|    |____|   |____||
 *     | Current   Previous  |     |                                       |
 *     |_____________________|     |_______________________________________|
 *
 *      The bottom layer is the Zone Allocator. This is already implemented in
 *  XNU and will remain mostly unchanged. Implementation for this can be found
 * in zalloc.c and zalloc.h. We will only use the zone if all other layers are
 * unable to satisfy the allocation or free. When we do use the zone, we will
 * try to allocate an entire magazine of elements or free an entire magazine of
 * elements at once.
 *
 *      Caching must be enabled explicitly, by calling zone_change() with the
 * Z_CACHING_ENABLED flag, for every zone you want to cache elements for. Zones
 * which are good candidates for this are ones with highly contended zone locks.
 * Some good potential candidates are kalloc.16, kalloc.48, Vm objects, VM map
 * entries, ipc vouchers, and ipc ports.
 * ..
 */
```

The user mode perspective of memory pressure conditions, which occur when the system is low on physical memory, was discussed in III/8, which also introduced MacOS's **memorystatus**, and the *OS Jetsam. Whereas the former is a gentle, opt-in mechanism (thanks to the abundant availability of swap space), the latter is a cruel and harsh overlord, which will not hesitate to kill for the most minor of transgressions. The `memorystatus_do_kill()` routine (in bsd/kern/kern_memorystatus.c) takes a `uint32_t` *cause* argument, which is a `kMemoryStatusKilled*` constant from the following:

Table 12-15: The many causes of untimely death by Jetsam/Memorystatus (from sys/kern_memorystatus.h)

kMemorystatusKilled..	**Reason**
(--)	Jettisoned
`Hiwat`	High Water Mark (maximum memory utilization)
`Vnodes`	Maximum vnode utilization (vnode table full)
`VMPageShortage`	Overall free page shortage
`ProcThrashing`	Process thrashing
`FCThrashing`	File Cache thrashing
`PerProcessLimit`	Per-process page limit exceeded
`DiskSpaceShortage`	Low disk space
`IdleExit`	Idle exit (memory status)
`ZoneMapExhaustion`	Zone map nearing exhaustion
`VMCompressorThrashing`	Compressor thrashing (excessive operations due to memory handling)
`VMCompressorSpaceShortage`	Compressor overall space shortage
`LowSwap`	Low swap space

The various reasons have corresponding `memorystatus_kill_on_..` functions, and those funnel to `memorystatus_kill_process_sync()`, which proceeds to kill either the PID specified, or the top process according to the jetsam priority bands (discussed in I/9). Execution is swift and merciless: `memorystatus_do_kill()` calls the "no-frills", no saving throw `jetsam_do_kill()`, which uses `exit_with_reason()` to smite the process with a `SIGKILL`. `memorytstatus_do_kill()` then triggers memory compaction, to free as much memory as possible.

The most common reason for riling Jetsam is `kMemoryStatusKilledHiwat`, which is common on *OS when a process commits the deadly sin of gluttony, consuming too much memory. Jetsam can be made to warn (`memorystatus_warn_process()`) through `memorystatus_on_ledger_footprint_exceeded()`. This code path, however, is no longer active as of Darwin 16. Instead, `consider_vm_pressure_events()` defers to `memorystatus_update_vm_pressure()`, which dispatches a knote, which Jetsam-fearing apps can respond to (per `didReceiveMemoryWarning()`, q.v. I/9-32).

Senseless bloodshed must be avoided if possible, so `memorystatus_kill_proc()` also attempts a call to `vm_purgeable_purge_task_owned()`, to see if it can purge some of the task's memory. This is done for all causes, save for vnode or zone map exhaustion.

Purgeable memory

Purgeable memory is a non-standard extension provided by XNU. Such memory is marked at the `vm_object` level, and may be discarded at the kernel's discretion. As explained in I/8, `libmalloc` supports `malloc_make_[non]purgeable`() calls, and `libcache` (or the higher level `Foundation.framework`'s `NSCache`) make use of this facility. Purgeable memory may also be allocated directly, with the `VM_FLAGS_PURGABLE`* flag when calling `[mach_]vm_allocate()` (or through the `fd` argument of `mmap(2)`, when using `MAP_ANON`).

* - The correct spelling is "purg**e**able", yet portions of the kernel (primarily in osfmk/mach/vm_purgable.h) are spelled "purgable". This is not only an unfortunate mistake, it can get downright frustrating, especially when both spellings are used in the same routine. The comment in osfmk/vm/vm_purgeable_internal.h expects to eventually "change this on occasion", (perhaps by the simple solution of aliasing through macros?) but the occasion has yet to arrive.

The purgeable facility is accessible through MIG routines in several subsystems:

- `task_purgable_info` (#3438): returning a `struct vm_purgeable_info` on the counts of purgeable memory for this task.
- `[mach_]vm_purgable_control()` (#4818/3830): a `VM_PURGABLE_GET_STATE`, `SET_STATE` or `PURGE_ALL` operation.
- `memory_entry_purgable_control()` (#4900): Starting with Darwin 18, XNU provides a new MIG subsystem, `memory_entry` (#4900), whose first routine is `memory_entry_purgable_control()`. This is the same as the VM-level ones, but at a memory entry granularity.

In kernel mode, purgeable memory is maintained at the `vm_object` level, in the `purgable` field. This can exist in one of three states: `VM_PURGABLE_VOLATILE`, `..NONVOLATILE`, or `..DENY`. The high order bits of the state hold the ordering (`VM_PURGABLE_BEHAVIOR_[FIFO/LIFO]`) and grouping of the page, as documented in osfmk/mach/vm_purgable.h:

Listing 12-16: The purgeable state bits (from osfmk/vm/vm_purgeable.h)

```
/*
 * Purgeable state:
 *
 *  31 15 14 13 12 11 10 8 7 6 5 4 3 2 1 0
 * +-----+--+-----+--+----+-+-+---+---+---+
 * |     |NA|DEBUG|  | GRP| |B|ORD|   |STA|
 * +-----+--+-----+--+----+-+-+---+---+---+
 * " ": unused (i.e. reserved)
 * STA: purgeable state
 *      see: VM_PURGABLE_NONVOLATILE=0 to VM_PURGABLE_DENY=3
 * ORD: order
 *      see:VM_VOLATILE_ORDER_*
 * B: behavior
 *      see: VM_PURGABLE_BEHAVIOR_*
 * GRP: group
 *      see: VM_VOLATILE_GROUP_*
 * DEBUG: debug
 *      see: VM_PURGABLE_DEBUG_*
 * NA: no aging
 *      see: VM_PURGABLE_NO_AGING*
 */
```

Purgable status maintenance is performed through `vm_object_purgable_control()`, which is also what the higher level `..purgable_control()` MIG routines call. The VM subsystem calls look up the corresponding `vm_map_entry`, and through it resolve the `VME_OBJECT`. Possible operations are `VM_PURGABLE_[GET/SET]_STATE`, or `.._PURGE_ALL`.

When setting the state to `VM_PURGABLE_VOLATILE`, the object is disconnected from its physical page, and added to one of the `purgeable_queues`. The facility presently maintains three `purgeable_queues`: `PURGEABLE_Q_TYPE_OBSOLETE` (deprecated, FIFO), `.._FIFO` and `...LIFO`, corresponding to the `VM_PURGABLE_BEHAVIOR_*` bits. The queues (defined in osfmk/vm/vm_purgeable_internal.h) further support sub-groups, allowing objects to be sub-classified. The `..FIFO` and `..LIFO` queues also support tokens:

Listing 12-17: The `purgeable_q` type (from osfmk/vm/vm_purgeable_internal.h)

```
#define NUM_VOLATILE_GROUPS 8
struct purgeable_q {
        token_idx_t token_q_head;    /* first token */
        token_idx_t token_q_tail;    /* last token  */
        token_idx_t token_q_unripe;  /* first token which is not ripe */
        int32_t new_pages;
        queue_head_t objq[NUM_VOLATILE_GROUPS];
        enum purgeable_q_type type;
};
```

Thus, when `vm_object_purgable_control()` is called with `.._PURGE_ALL`, the routine calls `vm_purgeable_object_purge_all()` (from osfmk/vm/vm_purgeable.c), cycling through all queues, over each of the groups, and calling `vm_object_purge()` to dispose of it. All of this is done under up to three locks - `vm_purgeable_queue_lock`, the purgeable `vm_object`'s own lock, and the `vm_page_queue_lock` (when removing the page). Purging the object is performed by force-moving the physical pages of the object to the free queue (unless busy, paged or wired). The object is then marked as "empty".

Kernel Memory Layout

With all the types of kernel memory figured out at this point, we can draw a rough atlas of kernel memory. Certain areas in kernel memory - especially the `zone_map` - are quite volatile, as processes, threads, kexts and other objects pop in and out of existence. At a higher level, however, i.e. one that considers the `zone_map` and other sub maps opaque, the layout is fairly stable, owing to the kernel's deterministic startup and fixed allocations.

The `kernel_map` regions

The `kernel_map` is just a special case of a `vm_map`, so have `kernel_task`, will travel: Using the `mach_vm_region_*` MIG routines, the `kernel_map`'s individual mappings can be retrieved through `vm_region_*_info` flavors. This can actually be accomplished without holding the task, as well, thanks to the powerful `proc_info` system call (#336). This is the method employed by `procexp(j)` when displaying `regions` for PID 0, and requires no entitlement - only `root` privileges. The output of `procexp(j)` will gives similar results similar to Output 12-18, though over time will get cluttered further with kext allocations and other kernel allocations, which may fill the numerous holes.

Output 12-18: The regions of XNU

```
# Display kernel regions through proc_info, weeding out those with dynamic
# tags, which belong to individual kernel extensions
#
root@Zephyr(~)# procexp 0 regions |
pipe>           grep -v ^Tag
Untagged (0)    0x00000000 ffffff7f80000000-ffffff7f97400000 [ 372M]---/--- NUL
Kext            0x00000000 ffffff7f97400000-ffffff8000000000 [   1G]rw-/rwx NUL   # __PRELINK_TEXT
#
# Kernel hiding here in plain sight... (from __TEXT through __LAST, with KLD section jettisoned.
# In *OS this would be in a 4-16G ---/---- mapping, to make it "harder" to figure out slide
Untagged (0)    0x00000000 ffffff8000000000-ffffff80176c2000 [ 374M]---/--- NUL
Untagged (0)    0x00e6e1c9 ffffff80176c2000-ffffff8017830000 [   1M]rw-/rwx PRV # __LINKEDIT (jettisoned)
Untagged (0)    0x00000000 ffffff8017830000-ffffff8021786000 [ 159M]---/--- NUL
..
PMAP            0x00e50640 ffffff8021931000-ffffff8026a5d000 [  81M]rw-/rwx S/A  # pmap structure from pmap_init()
ZONE            0x00000000 ffffff8026a5d000-ffffff80e6a5d000 [   3G]rw-/rwx NUL  # zone_map_[min-max]_address
OSFMK           0x00e50640 ffffff80e6a5d000-ffffff80e6a5e000 [   4K]rw-/rwx S/A  # zone names
kalloc          0x00000000 ffffff80e6a5e000-ffffff80f6a5e000 [ 256M]rw-/rwx NUL  # kalloc_map
....
IPC             0x00000000 ffffff80f6ad3000-ffffff80f6bd3000 [1024K]rw-/rwx NULL # ipc_kernel_map
IPC             0x00000000 ffffff80f6bd3000-ffffff80f73d3000 [   8M]rw-/rwx NULL # ipc_kernel_copy_map
..
OSKext          0x00c73cc9 ffffff811763d000-ffffff8117642000 [  20K]r--/rwx PRV  # gLoadedKextSummaries
compressor      0x00e50640 ffffff811fcce000-ffffff811fccf000 [   4K]rw-/rwx S/A
compressor      0x00000000 ffffff8123069000-ffffff91232a9000 [  64G]rw-/rwx NUL  # compressor_map
..
```

It's possible to determine the semantics of the memory regions thanks to the memory tags assigned by the kernel and the individual kexts: Calls to `kernel_memory_allocate()`, `kmem_suballoc()` and friends take a **tag** value, and the tags are listed in osfmk/mach/vm_statistics.h as `VM_KERNL_MEMORY_*` constants (all `KERNEL_PRIVATE`, so they are not visible in the user mode header). Note, that some tags are used only in the context of `kalloc_tag[_bt]` calls, and will thus not be visible in region information. Table 12-20 highlights the tags and - more importantly - their callers:

Table 12-20: The `VM_KERN_MEMORY_*` tags in osfmk/kern/memory_statistics.h

#	`VM_KERN_MEMORY_*`	caller
0	`_NONE`	The kernel's own Mach-O is tagged this way
1	`_OSFMK`	Miscellaneous in osfmk/vm/*
2	`_BSD`	Temporary mappings for sysctl(8), Mach-O loading, and execve(2)
3	`_IOKIT`	`gIOKitPageableMaps`, serializer data, etc
4	`_LIBKERN`	tags used by libkern `kalloc_tag[_bt]` calls
5	`_OSKEXT`	OSKext structures (primarily, `gLoadedKextSummaries`, etc.
6	`_KEXT`	Loaded Kext Mach-Os (`__PRELINK_TEXT`)
7	`_IPC`	`ipc_kernel_map` and `ipc_kernel_copy_map`
8	`_STACK`	Thread stack space
9	`_CPU`	per-CPU data
10	`_PMAP`	MacOS: `pv_*_tables` (from `pmap_init()`
11	`_PTE`	Used as `vm_object` tag in `vm_page_wire/vm_page_insert_wired()`

Table 12-20 (cont.): The `VM_KERN_MEMORY_*` tags in osfmk/kern/memory_statistics.h (cont.)

#	VM_KERN_MEMORY_*	caller
12	`_ZONE`	The kernel zone map
13	`_KALLOC`	The `kalloc_map`
14	`_COMPRESSOR`	The `compressor_map`
15	`_COMPRESSED_DATA`	Unused
16	`_PHANTOM_CACHE`	Ghost pages in phantom cache
17	`_WAITQ`	`global_waitqs` and (`#if CONFIG_WAITQ_STATS`) `g_waitq_stats`
18	`_DIAG`	kdebug & telemetry buffers etc.
19	`_LOG`	`os_log kernel_firehose_addr`
20	`_FILE`	Kernel UPLs, VFS buffers
21	`_MBUF`	The `mbmap` for allocating network `mbuf`s
22	`_UBC`	Reserved for Unified Buffer Cache, but unused
23	`_SECURITY`	Apple Protect Pager, `mac_wire`, etc.
24	`_MLOCK`	`mlock(2)`ed and/or `mach_vm_wire()`d memory
25	`_REASON`	`OSReason kcdata` bufs (`kalloc_tag[_bt]` only)
26	`_SKYWALK`	Skywalk subsystem memory (arenas, etc)
27	`_LTABLE`	Darwin 18+: Lockless Link tables
28+	`_DYNAMIC`	Dynamic tags by specific kexts, first come first served

The Kernel Slide

The Kernel Address Space Layout Randomization (KASLR) was introduced in Darwin 12 in an effort to raise the bar on kernel exploitation. Determining the target address space is an important step in successfully overwriting memory or obtaining code execution. The idea behind KASLR, therefore, is to add a random **slide** value into the kernel base, so as to make what are otherwise fixed virtual memory addresses harder to determine, and thus exploit.

The kernel slide is set by the boot loader (EFI/`iBoot`), prior to loading the kernel. It can then be determined during kernel boot (in `[i386/arm]_vm_init`), by taking the fixed kernel address and subtracting it from the virtual base address passed in the Platform Expert's `boot_args` struct. It is then cached in the `vm_kernel_slide` global.

The `kern.slide` sysctl MIB is set to 1 if the kernel is slid, and the `kas_info` syscall (#439) returns the kernel slide value to user mode. On the *OS `SECURE_KERNEL` this is naturally unimplemented (`ENOTSUP`). In MacOS, it requires both `root` privileges and the agreement of the MACF policies hooking `mac_system_check_kas_info()`, which is enforced by the Sandbox.kext when SIP is enabled. If SIP is disabled, numerous ways of retrieving the value exist, as simple as a one-line DTrace script dumping the value from `PE_state->kslide`.

It is absolutely imperative to "unslide" any kernel addresses reported back to user mode through debugging interfaces. There are generally two methods of doing so. The first is simply unsliding - i.e. subtracting the `vm_kernel_slide` value, so the address returned is the same as can be found in the kernel's Mach-O. This is usually the case during backtraces, as it both serves to hide the slide and make the stack traces easy to symbolicate. The second is permuting the address, so that it remains unique but not easy to associate with its original value. This is the case when returning unique object addresses from zones or elsewhere in the `kernel_map` - for example the `iin_object`s of `mach_port_space_info`.

Two macros are commonly used - `VM_KERNEL_UNSLIDE[_OR_PERM]`. The latter acts unslides `VM_KERNEL_IS_SLID` addresses, and applies the permutation to other kernel addresses. The permutation is an arbitrary `vm_kernel_addrperm` value, `read_random()`ly during the `kernel_bootstrap_thread()`, and then added to the result of applying the `VM_KERNEL_STRIP_PTR` macro on the pointer (Unfortunately, the macro merely returns the original pointer). Darwin 17 and later add (but, as of yet, do not use) a third macro, `VM_KERNEL_ADDRHASH`, which uses a proper hash (presently, SHA-256), along with a much needed `vm_kernel_addrhash_salt`, also `read_random()`ly during startup.

Review Questions

1. Prior to Darwin 16, Apple tested other locations for the zone metadata, including putting it in the beginning and end of every page. What is an advantage and a disadvantage of the present solution?
2. What other, possibly simpler way, to get a zone element's metadata by its index, could you consider in place of the `PAGE_METADATA_FOR_PAGE_INDEX` macro? Why is a macro preferred?
3. Looking back at the footnote in the section discussing the zone metadata region, you will note that attempting to read the very beginning of the metadata region (which is also the very beginning of the zone map) will fail and is prone to panic on *OS through `mach_vm_read()` of the `kernel_task`. Why is that?
4. Following on the previous question, what is the formula to determine the first valid metadata entry in the `zone_metadata_region`, which is also safe to read from kernel memory? What is the minimal page index to which this applies?
5. What other concern would one encounter when trying to sequentially read kernel memory from the `zone_metadata_region`? How could that issue be solved?
6. Why is the `MAX_ENTROPY_PER_ZCRAM` set to 4?
7. How is it that the `pmap` zone's `cur_size` may exceed its `max_size` (as in 12-12-c)?
8. What is the difference between `Z_EXHAUST` and `!Z_EXPANDABLE`?
9. What could have been the rationale for `zone_require()` *not* `panic()`ing on an address outside the `zone_map`? Why is this incorrect? And how could the routine be properly reimplemented so as to cover all cases?
10. In older versions of Darwin (and even the present day, for foreign allocations) the zone metadata could be embedded in the element page. Why is this a bad idea?
11. What are the similarities and differences between the user mode magazine allocator and the new kernel mode zone allocator of Darwin 18?
12. Why is it absolutely vital to empty the `vm_object` after force-freeing its pages during `vm_object_purge()`?
13. Where are the **two(!) obvious(!!)** KASLR memory disclosures in `procexp 0 regions` in MacOS (at least up to 15)? Which one of those is (at least up to iOS 13, maybe later) in *OS as well?
14. Why is the salt an absolute requirement for the `VM_KERNEL_ADDRHASH` scenario?
15. How could ledgers (from Chapter 9) be used to augment the defenses against zone corruption attacks and fake objects?

References

1. Brandon Azad (Google Project Zero) - "oob_timestamp" (CVE-2020-3837) - https://bugs.chromium.org/p/project-zero/issues/detail?id=1986#c4