latest news

- 5th InfoCom Security Conference
- 5th InfoCom Mobile World Conference
- Project Heapbleed
- Presentation at FORTH Institute of Computer Science

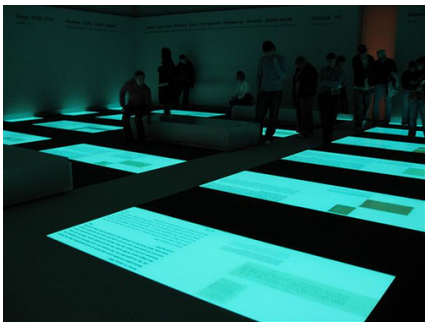
blog posts

- The road to efficient Android fuzzing
- Introducing wifiphisher — BSides London 2015
- Fuzzing Objects d'ART — Hack In The Box 2015 Amsterdam
- OR'LYEH? The Shadow over Firefox (INFILTRATE 2015)

The Linux kernel memory allocators from an exploitation perspective

[posted by argp on 03.01.2012]

In anticipation of Dan Rosenberg's talk on exploiting the Linux kernel's SLOB memory allocator at the Infiltrate security conference and because I recently had a discussion with some friends about the different kernel memory allocators in Linux, I decided to write this quick introduction. I will present some of the allocators' characteristics and also provide references to public work on exploitation techniques.



At the time of this writing, the Linux kernel has three different memory allocators in the official code tree, namely SLAB, SLUB and SLOB. These allocators are on a memory management layer that is logically on top of the system's low level page allocator and are mutually exclusive (i.e. you can only have one of them enabled/compiled in your kernel). They are used when a kernel developer calls `kmalloc()` or a similar function. Unsurprisingly, they can all be found in the mm directory. All of them follow, to various extends and by extending or simplifying, the traditional slab allocator design (notice the lowercase "slab"; that's the term for the general allocator design approach, while "SLAB" is a slab implementation in the Linux kernel). Slab allocators allocate prior to any request, for example at kernel boot time, large areas of virtual memory (called "slabs", hence the name). Each one of these slabs is then associated to a kernel structure of a specific type and size. Furthermore, each slab is divided into the appropriate number of slots for the size of the kernel structure it is associated with. As an example consider that a slab for the structure `task_struct` has 31 slots. The size of a `task_struct` is 1040 bytes, so assuming that a page is 4096 bytes (the default) then a `task_struct` slab is 8 pages long. Apart from the structure-specific slabs, like the one above for `task_struct`, there are also the so called general purpose slabs which are used to serve arbitrary-sized `kmalloc()` requests. These requests are adjusted by the allocator for alignment and assigned to a suitable slab.

Let's take a look at the slabs of a recent Linux kernel:

```
$ cat /proc/slabinfo
slabinfo - version: 2.1
...
fat_inode_cache       57     57    416   19    2 : tunables    0    0    0 : slabdata
3      3      0
fat_cache            170    170     24  170    1 : tunables    0    0    0 : slabdata
1      1      0
VMBlockInodeCache      7      7   4480    7    8 : tunables    0    0    0 : slabdata
1      1      0
blockInfoCache         0      0   4160    7    8 : tunables    0    0    0 : slabdata
0      0      0
AF_VMCI                0      0    704   23    4 : tunables    0    0    0 : slabdata
0      0      0
fuse_request          80     80    400   20    2 : tunables    0    0    0 : slabdata
4      4      0
fuse_inode         21299  21690    448   18    2 : tunables    0    0    0 : slabdata
1205   1205      0
...
kmalloc-8192          94     96   8192    4    8 : tunables    0    0    0 : slabdata
24     24      0
kmalloc-4096         118    128   4096    8    8 : tunables    0    0    0 : slabdata
16     16      0
kmalloc-2048         173    208   2048   16    8 : tunables    0    0    0 : slabdata
13     13      0
kmalloc-1024         576    640   1024   16    4 : tunables    0    0    0 : slabdata
40     40      0
kmalloc-512          904    992    512   16    2 : tunables    0    0    0 : slabdata
62     62      0
kmalloc-256          540    976    256   16    1 : tunables    0    0    0 : slabdata
61     61      0
kmalloc-128          946   1408    128   32    1 : tunables    0    0    0 : slabdata
44     44      0
kmalloc-64         13013  13248     64   64    1 : tunables    0    0    0 : slabdata
```

latest advisories

- Oracle WebCenter information exposure vulnerability
- libpurple OTR information leakage
- Netvolution referer header SQL injection vulnerability
- FreeBSD kernel NFS client local vulnerabilities
- Monkey HTTPd improper input validation vulnerability
- CoreHTTP web server off-by-one buffer overflow vulnerability

job openings

- Lead Penetration Tester

```
207    207      0
kmalloc-32         23624  27264     32  128    1 : tunables    0    0    0 : slabdata
213    213      0
kmalloc-16          3546   3584     16  256    1 : tunables    0    0    0 : slabdata
14     14      0
kmalloc-8           4601   4608      8  512    1 : tunables    0    0    0 : slabdata
9      9      0
kmalloc-192         3659   4620    192   21    1 : tunables    0    0    0 : slabdata
220    220      0
kmalloc-96         10137  11340     96   42    1 : tunables    0    0    0 : slabdata
270    270      0
kmem_cache            32     32    128   32    1 : tunables    0    0    0 : slabdata
1      1      0
kmem_cache_node      256    256     32  128    1 : tunables    0    0    0 : slabdata
2      2      0
```

Here you can see structure-specific slabs, for example `fuse_inode`, and general purpose slabs, for example `kmalloc-96`.

When it comes to the exploitation of overflow bugs in the context of slab allocators, there are three general approaches to corrupt kernel memory:

- Corruption of the adjacent objects/structures of the same slab.
- Corruption of the slab allocator's management structures (referred to as *metadata*).
- Corruption of the adjacent *physical* page of the slab your vulnerable structure is allocated on.

The ultimate goal of the above approaches is of course to gain control of the kernel's execution flow and divert/hijack it to your own code. In order to be able to manipulate the allocator and the state of its slabs, arranging structures on them to your favor (i.e. next to each other on the same slab, or at the end of a slab), it is nice (but not strictly required ;) to have some information on the allocator's state. The proc filesystem provides us with a way to get this information. Unprivileged local users can simply `cat /proc/slabinfo` (as shown above) and see the allocator's slabs, the number of used/free structures on them, etc. Is your distribution still allowing this?

For each one of the Linux kernel's allocators I will provide references to papers describing practical attack techniques and examples of public exploits.

SLAB

Starting with the oldest of the three allocators, SLAB organizes physical memory frames in caches. Each cache is responsible for a specific kernel structure. Also, each cache holds slabs that consist of contiguous pages and these slabs are responsible for the actual storing of the kernel structures of the cache's type. A SLAB's slab can have both allocated (in use) and deallocated (free) slots. Based on this and with the goal of reducing fragmentation of the system's virtual memory, a cache's slabs are divided into three lists; a list with full slabs (i.e slabs with no free slots), a list with empty slabs (slabs on which all slots are free), and a list with partial slabs (slabs that have slots both in use and free).

A SLAB's slab is described by the following structure:

```
struct slab
{
    union
    {
        struct
        {
            struct list_head list;
            unsigned long colouroff;
            void *s_mem;            /* including colour offset */
            unsigned int inuse;     /* num of objs active in slab */
            kmem_bufctl_t free;
            unsigned short nodeid;
        };

        struct slab_rcu __slab_cover_slab_rcu;
    };
};
```

The `list` variable is used to place the slab in one of the lists I described above. Coloring and the variable `colouroff` require some explanation in case you haven't seen them before. Coloring or cache coloring is a performance trick to reduce processor L1 cache hits. This is accomplished by making sure that the first "slot" of a slab (which is used to store the slab's `slab` structure, i.e. the slab's metadata) is not placed at the beginning of the slab (which is also at the start of a page) but an offset `colouroff` from it. `s_mem` is a pointer to the first slot of the slab that stores an actual kernel structure/object. `free` is an index to the next free object of the slab.

As I mentioned in the previous paragraph, a SLAB's slab begins with a `slab` structure (the slab's metadata) and is followed by the slab's objects. The stored objects on a slab are contiguous, with no metadata in between them, making easier the exploitation approach of corrupting adjacent objects. Easier means that when we overflow from one object to its adjacent we don't corrupt management data that could lead to making the system crash.

By manipulating SLAB through controlled allocations and deallocations from userland that affect the kernel (for example via system calls) we can arrange that the overflow from a vulnerable structure will corrupt an adjacent structure of our own choosing. The fact that SLAB's allocations and deallocations work in a LIFO manner is of course to our advantage in arranging structures/objects on the slabs. As qobaiashi has presented in his paper "The story of exploiting kmalloc() overflows", the system calls `semget()` and `semctl(..., ..., IPC_RMID)` is one way to make controlled allocations and deallocations respectively. The term "controlled" here refers to both the size of the allocation/deallocation and the fact that we can use them directly from userland. Another requirement that these system calls satisfy is that the structure they allocate can help us in our quest for code execution when used as a victim object and corrupted from a vulnerable object. Other ways/system calls that satisfy all the above requirements do exist.

Another resource on attacking SLAB is "Exploiting kmalloc overflows to 0wn j00" by amnesia and clflush. In that presentation the authors explained the development process for a reliable exploit for vulnerability CVE-2004-0424 (which was an integer overflow leading to a kernel heap buffer overflow found by ihaquer and cliph). Both the presentation and the exploit are not public as far as I know. However, a full exploit was published by twiz and sgrakkyu in Phrack #64 (castity.c).

SLUB

SLUB is currently the default allocator of the Linux kernel. It follows the SLAB allocator I have already described in its general design (caches, slabs, full/empty/partial lists of slabs, etc.), however it has introduced simplifications in respect to management overhead to achieve better performance. One of the main differences is that SLUB has no metadata at the beginning of each slab like SLAB, but instead it has added it's metadata variables in the Linux kernel's `page` structure to track the allocator's data on the physical pages.

The following excerpt includes only the relevant parts of the `page` structure, see here for the complete version.

```
struct page
{
    ...

    struct
    {
        union
        {
            pgoff_t index;          /* Our offset within mapping. */
            void *freelist;         /* slub first free object */
        };

        ...

        struct
        {
            unsigned inuse:16;
            unsigned objects:15;
            unsigned frozen:1;
        };

        ...
    };

    ...

    union
    {
        ...
        struct kmem_cache *slab;        /* SLUB: Pointer to slab */
        ...
    };

    ...
};
```

Since there are no metadata on the slab itself, a `page`'s `freelist` pointer is used to point to the first free object of the slab. A free object of a slab has a small header with metadata that contain a pointer to the next free object of the slab. The `index` variable holds the offset to these metadata within a free object. `inuse` and `objects` hold respectively the allocated and total number of objects of the slab. `frozen` is a flag that specifies whether the page can be used by SLUB's list management functions. Specifically, if a page has been frozen by a CPU core, only this core can retrieve free objects from the page, while the other available CPU cores can only add free objects to it. The last interesting for our discussion variable is `slab` which is of type `struct kmem_cache` and is a pointer to the slab on the page.

The function `on_freelist()` is used by SLUB to determine if a given object is on a given page's `freelist` and provides a nice introduction to the use of the above elements. The following snippet is an example invocation of `on_freelist()` (taken from here):

```
slab_lock(page);

if(on_freelist(page->slab, page, object))
{
    object_err(page->slab, page, object, "Object is on free-list");
    rv = false;
}
else
{
    rv = true;
}

slab_unlock(page);
```

Locking is required to avoid inconsistencies since `on_freelist()` makes some modifications and it could be interrupted. Let's take a look at an excerpt from `on_freelist()` (the full version is here):

```
static int on_freelist(struct kmem_cache *s, struct page *page, void *search)
{
    int nr = 0;
    void *fp;
    void *object = NULL;
    unsigned long max_objects;

    fp = page->freelist;
    while(fp && nr <= page->objects)
    {
        if(fp == search)
            return 1;
        if(!check_valid_pointer(s, page, fp))
        {
            if(object)
            {
```

```
                object_err(s, page, object, "Freechain corrupt");
                set_freepointer(s, object, NULL);
                break;
            }
            else
            {
                slab_err(s, page, "Freepointer corrupt");
                page->freelist = NULL;
                page->inuse = page->objects;
                slab_fix(s, "Freelist cleared");
                return 0;
            }

            break;
        }

        object = fp;
        fp = get_freepointer(s, object);
        nr++;
    }

    ...
}
```

The function starts with a simple piece of code that walks the `freelist` and demonstrates the use of SLUB internal variables. Of particular interest is the call of the `check_valid_pointer()` function which verifies that a `freelist`'s object's address (variable `fp`) is within a slab page. This is a check that safeguards against corruptions of the `freelist`.

This brings us to attacks against the SLUB memory allocator. The attack vector of corrupting adjacent objects on the same slab is fully applicable to SLUB and largely works like in the case of the SLAB allocator. However, in the case of SLUB there is an added attack vector: exploiting the allocator's metadata (the ones responsible for finding the next free object on the slab). As twiz and sgrakkyu have demonstrated in their book on kernel exploitation, the slab can be misaligned by corrupting the least significant byte of the metadata of a free object that hold the pointer to the next free object. This misalignment of the slab allows us to create an in-slab fake object and by doing so to a) satisfy safeguard checks as the one I explained in the previous paragraph when they are used, and b) to hijack the kernel's execution flow to our own code.

An example of SLUB metadata corruption and slab misalignment is the exploit for vulnerability CVE-2009-1046 which was an off-by-two kernel heap overflow. In this blog post, sgrakkyu explained how by using only an one byte overflow turned this vulnerability into a reliable exploit (tiocl_houdini.c). If you're wondering why an one byte overflow is more reliable than a two byte overflow think about little-endian representation.

A public example of corrupting adjacent SLUB objects is the exploit i-can-haz-modharden.c by Jon Oberheide for vulnerability CVE-2010-2959 discovered by Ben Hawkes. In this blog post you can find an overview of the exploit and the technique.

SLOB

Finally, SLOB is a stripped down kernel allocator implementation designed for systems with limited amounts of memory, for example embedded versions/distributions of Linux. In fact its design is closer to traditional userland memory allocators rather than the slab allocators SLAB and SLUB. SLOB places all objects/structures on pages arranged in three linked lists, for small, medium and large allocations. Small are the allocations of size less than `SLOB_BREAK1` (256 bytes), medium those less than `SLOB_BREAK2` (1024 bytes), and large are all the other allocations:

```
#define SLOB_BREAK1 256
#define SLOB_BREAK2 1024
static LIST_HEAD(free_slob_small);
static LIST_HEAD(free_slob_medium);
static LIST_HEAD(free_slob_large);
```

Of course this means that in SLOB we can have objects/structures of different types and sizes on the same page. This is the main difference between SLOB and SLAB/SLUB. A SLOB page is defined as follows:

```
struct slob_page
{
    union
    {
        struct
        {
            unsigned long flags;    /* mandatory */
            atomic_t _count;        /* mandatory */
            slobidx_t units;        /* free units left in page */
            unsigned long pad[2];
            slob_t *free;           /* first free slob_t in page */
            struct list_head list;  /* linked list of free pages */
        };

        struct page page;
    };
};
```

The function `slob_alloc()` is SLOB's main allocation routine and based on the requested size it walks the appropriate list trying to find if a page of the list has enough room to accommodate the new object/structure (the full function is here):

```
static void *slob_alloc(size_t size, gfp_t gfp, int align, int node)
{
    struct slob_page *sp;
```

```
    struct list_head *prev;
    struct list_head *slob_list;
    slob_t *b = NULL;
    unsigned long flags;

    if (size < SLOB_BREAK1)
        slob_list = &free_slob_small;
    else if (size < SLOB_BREAK2)
        slob_list = &free_slob_medium;
    else
        slob_list = &free_slob_large;

    ...

    list_for_each_entry(sp, slob_list, list)
    {
        ...
```

I think this is a good place to stop since I don't want to go into too many details and because I really look forward to Dan Rosenberg's talk.

Edit: Dan has published a whitepaper to accompany his talk with all the details on SLOB exploitation; you can find it here.

Notes

Wrapping this post up, I would like to mention that there are other slab allocators proposed and implemented for Linux apart from the above three. SLQB and SLEB come to mind, however as the benevolent dictator has ruled they are not going to be included in the mainline Linux kernel tree until one of the existing three has been removed.

Exploitation techniques and methodologies like the ones I mentioned in this post can be very helpful when you have a vulnerability you're trying to develop a reliable exploit for. However, you should keep in mind that every vulnerability has its own set of requirements and conditions and therefore every exploit is a different story/learning experience. Understanding a bug and actually developing an exploit for it are two very different things.

Thanks to Dan and Dimitris for their comments.

References

The following resources were not linked directly in the discussion, but would be helpful in case you want to look more into the subject.

http://lxr.linux.no/linux+v3.1.6/
http://lwn.net/Articles/229984/
http://lwn.net/Articles/311502/
http://lwn.net/Articles/229096/
http://phrack.org/issues.html?issue=66&id=15#article
http://phrack.org/issues.html?issue=66&id=8#article

tags: exploitation, heap, kernel, linux, slab, slob, slub

Add post to: