Jon Oberheide

Linux Kernel CAN SLUB Overflow

Ben Hawkes discovered a vulnerability in the Controller Area Network (CAN) packet family in the Linux kernel that results in a controllable overflow of a SLUB-allocated structure. As there's not a whole lot of modern, public examples of SLUB overflow exploits, I'll describe my exploit of the CAN vulnerability in detail.

The Vulnerability

Ben provides the full details of the vulnerability in his blog post:

> *A controller area network is backed by the AF_CAN datagram socket type. This socket is enabled by the CONFIG_CAN kernel configuration option, so any kernel compiled with CONFIG_CAN and CONFIG_CAN_BCM options were vulnerable. This included at least Ubuntu 10.04 and Debian 5.0 (i'm told a pre-release version of Red Hat was also affected).*
>
> *The bug is an integer overflow in the sendmsg implementation for BCM (broadcast manager) AF_CAN sockets which results in controlled corruption of a kmalloc heap chunk. No physical CAN device is required to trigger the overflow.*

The bcm_sendmsg function in net/can/bcm.c reads in a bcm_msg_head structure from a user-supplied iovec:

```
struct bcm_msg_head {
     __u32 opcode;
     __u32 flags;
     ...
     canid_t can_id;
     __u32 nframes;
     struct can_frame frames[0];
};
```

The opcode field dictates the type of message processing that should be performed by bcm_sendmsg. The vulnerability is in the RX_SETUP operation, which is backed by the bcm_rx_setup function in net/can/bcm.c (comments marked with BH):

```
#define CFSIZ sizeof(struct can_frame)

static int bcm_rx_setup(struct bcm_msg_head *msg_head, ...
 // BH: the ifindex parameter is set to zero if
 // BH: msg->msg_name is NULL
 ...

 op = bcm_find_op(&bo->rx_ops, msg_head->can_id,
                  ifindex);
 // BH: by setting can_id to 0xdeadbeef, a NULL op
 // BH: is returned
 if (op) {
     ...
 }
 else {
    op = kzalloc(OPSIZ, GFP_KERNEL);

    if (!op)
       return -ENOMEM;

    op->can_id    = msg_head->can_id;
    op->nframes   = msg_head->nframes;
    // BH: nframes is controlled by the attacker

    if (msg_head->nframes > 1) {
       op->frames = kmalloc(
                  msg_head->nframes * CFSIZ,
                  GFP_KERNEL);
       // BH: integer overflow here, large nframes
       // BH: wraps around to cause a small alloc
       ...

    }...

    if (msg_head->nframes) {
       err = memcpy_fromiovec((u8 *)op->frames,
                    msg->msg_iov,
                    msg_head->nframes * CFSIZ);
       // BH: size field overflows to same value as
       // BH: the allocation, no corruption
                  ...
```

```
    } ...

    do_rx_register = 1
 }

 ...
 if (do_rx_register) {
    if (ifindex) {
       // BH: ifindex is zero, as noted above
       ...
    } else
       err = can_rx_register(NULL, op->can_id,
                        REGMASK(op->can_id),
                        bcm_rx_handler, op, "bcm");

        // BH: can_rx_register explicitly
        // BH: allows registering to a NULL device

        if (err) {
           ...
        }
    }
...
```

Now at this point no memory corruption has occurred, but there is a bcm_op structure registered for the NULL device under a can_id of 0xdeadbeef with a large value for nframes (e.g. nframes = 268435458) and a small allocation for the frames buffer (e.g. 32 bytes). If the RX_SETUP operation is called again on this operation structure, but this time with a mid-sized nframes value (e.g. nframes = 512), then the following 'update' code in bcm_rx_setup is invoked:

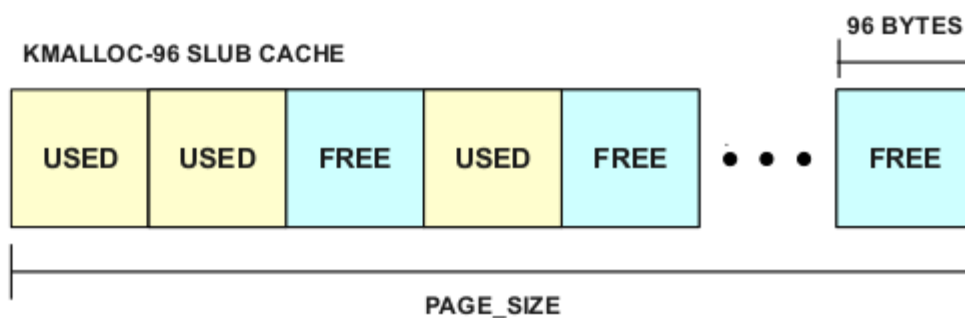
```
op = bcm_find_op(&bo->rx_ops, msg_head->can_id, ifindex);
// BH: op struct for 0xdeadbeef is returned

if (op) {
   // BH: 512 < 268435458
   if (msg_head->nframes > op->nframes)
      return -E2BIG;

   if (msg_head->nframes) {
      // BH: writes 8192 attacker-controlled bytes
      // BH: in to a 32-byte buffer
      err = memcpy_fromiovec((u8 *)op->frames,
                       msg->msg_iov,
                       msg_head->nframes * CFSIZ);
      ...
   }
}
```

This means that it's possible to corrupt any amount of data contiguous to the original 'frames' kmalloc chunk with an attacker-controlled value. The tough part from here is finding a good way of normalizing the heap layout to get a consistent (aka exploitable) crash. Needless to say that, with a bit of work, you can get an arbitrary kernel-space write.

SLUB Overflows

Exploiting a SLUB overflow requires a little knowledge about the SLUB allocator and how its caches are structured. The SLUB allocator manages many of the dynamic allocations and deallocations of internal kernel memory and is a descendant of the SLAB allocator. The kernel maintains a number of SLUB caches, distinguished by size for allocation efficiency. Some caches are general-purpose (eg. the "kmalloc-64" cache holds allocations that are of size <= 64 bytes but > 32 bytes) while others are explicitly defined for commonly allocated structures (eg. the "task_struct" cache contains the allocations for the kernel structure task_struct).

Each "slab" in a cache contains a number of contiguous allocations of some object. For example, the "kmalloc-32" cache contains 128 objects each with a maximum size of 32 bytes (32*128=4096, which is PAGE_SIZE on many systems). The allocator keeps track of free slots in each slab, so a typical slab may have both used and free slots and look something like the following:



Overflowing a SLUB Allocation

A key aspect of the SLUB allocator is that objects in a slab are allocated contiguously. Therefore if we can write past the intended bound of an allocation, we may be able to influence adjacently allocated objects:

can-slub-2

So, if we can overflow into an adjacent allocation, how can we convert that control of data into controlling the execution flow of the kernel? Ideally, we'd like to find a structure allocated in our SLUB cache that contains function pointers. If we can overwrite a function pointer in the kernel with a value we control, we can easily redirect control flow to an address of our choosing and escalate privileges.

However, in many cases, the allocated structure might not itself contain a function pointer, but may reference a number of additional structures, one of which contains a function pointer. A popular target for SLUB overflows that satisfies this condition is shmid_kernel, an internal kernel structure used to track

POSIX shared memory segments:

``` {style="text-align: left;"} struct shmid_kernel / *private to the kernel* / { struct kern_ipc_perm shm_perm; struct file *shm_file; unsigned long shm_nattch; unsigned long shm_segsz; time_t shm_atim; time_t shm_dtim; time_t shm_ctim; pid_t shm_cprid; pid_t shm_lprid; struct user_struct* mlock_user; };

```
It is an attractive target since its allocations are easily controlled
from unprivileged userspace applications (via shmget(2), shmat(2),
shmctl(2), etc), the allocations stay active even after process
termination, and it references a chain of structures that eventually
leads to a function pointer that is controllable and triggerable by the
attacker. If we construct fake versions of these structures and
reference them during the overflow, we can control the function
pointer. For shmid_kernel, the chain of structures that leads to a
controllable mmap function pointer looks like:

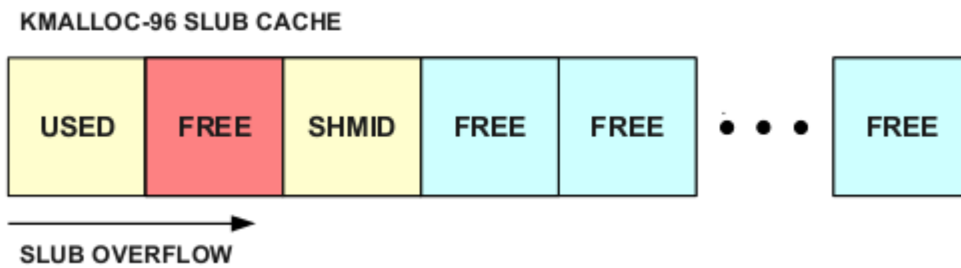
``` {style="text-align: left;"}
struct shmid_kernel {
    .shm_file = struct file {
        .f_op = struct file_operations = {
            .mmap = ATTACKER_ADDRESS
        }
    }
}
```

It's important to note that shmid_kernel is not the only eligible structure for SLUB overflows. For vulnerabilities where we can control the size of the allocation, shmid_kernel works great since we can ensure that our allocation is sized properly to be co-located in the same cache as shmid_kernel. However, for vulnerabilities where the size of the overflowed allocation is not in our control, it is necessary to find an alternate structure allocated in the same cache that meets the desired properties.

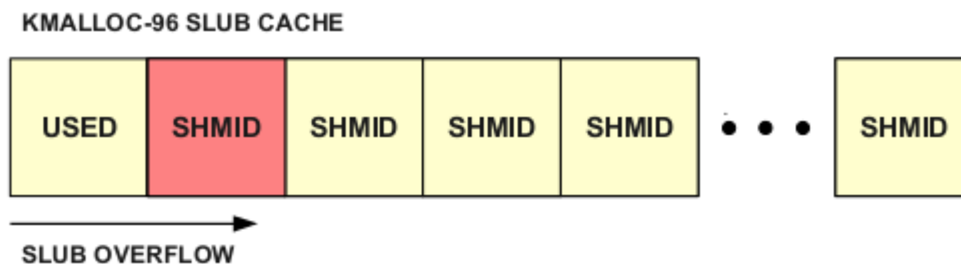
SLUB Feng Shui

As is common with exploiting overflows in dynamic memory allocators, the state of the cache and slab is critical to exploitation. If we cannot reliably control the state of the slab during our allocations and overflow, we will likely crash the target machine.

A slab state that is fragmented and contains many holes is not ideal for exploitation since we might overflow into an unused slot or an allocation other than our target one:



The easiest way to massage the SLUB into a more friendly state is to force a large number of shmid_kernel allocations, reducing the potential for fragmentation and increasing the likelihood that our overflowing allocation will be adjacent to a shmid_kernel allocation:



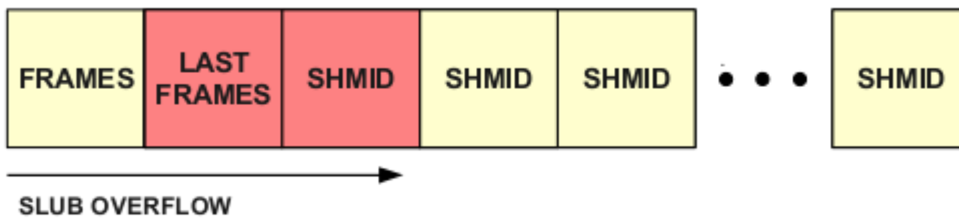
To further improve reliability, we can use /proc/slabinfo if it is available and accessible to an unprivileged user, to ensure we're dealing with a fresh non-fragmented slab and achieve allocations of our shmid_kernel structure adjacent to our overflowed structure.

The Exploit

While we've already covered much of the necessary material to exploit a generic SLUB overflow, there are some aspects of the CAN BCM vulnerability that make it interesting and are worth further discussion.

First, the allocation pattern of the CAN BCM module gives us some desirable properties for smashing the SLUB. We control the kmalloc with a 16-byte granularity allowing us to place our allocation in the SLUB cache of our choosing. As described above, we'll specify a 96-byte allocation so that it will be allocated from the kmalloc-96 cache, the same cache used to allocate shmid_kernel structures. The allocation can also be made in its own discrete stage before the overwrite which allows us to be a bit more conservative in ensuring the proper layout of our SLUB cache.

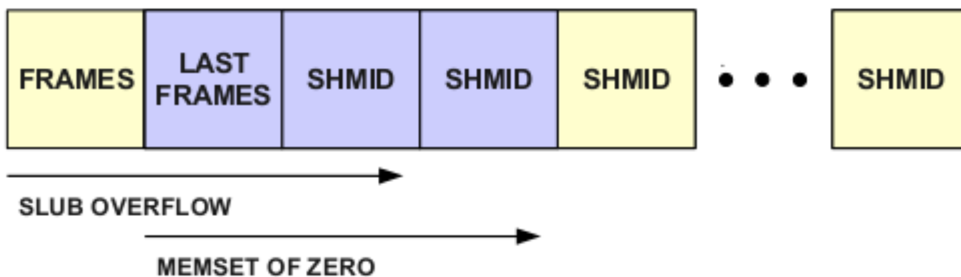
To exploit the vulnerability, we first create a BCM RX op with a crafted nframes value to trigger the integer overflow during the kmalloc. On the second call to update the existing RX op, we bypass the E2BIG check since the stored nframes in the op is large, yet has an insufficiently sized allocation associated with it. We then have a controlled write into the adjacent shmid_kernel object in the 96-byte SLUB cache:



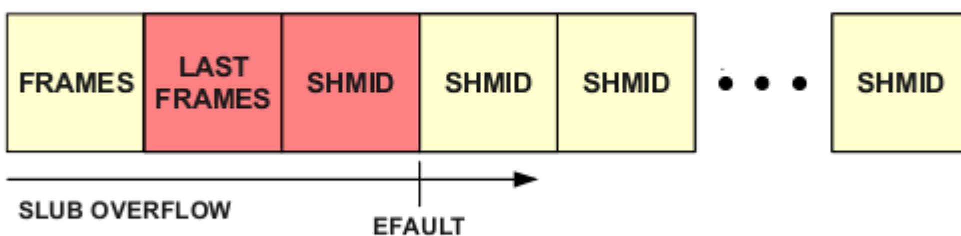
However, while we control the length of the SLUB overwrite via a memcpy_fromiovec operation, there exists a memset operation that directly follows:

```
/* update can_frames content */
err = memcpy_fromiovec((u8 *)op->frames,
                       msg->msg_iov,
                       msg_head->nframes * CFSIZ);
if (err < 0)
    return err;

/* clear last_frames to indicate 'nothing received' */
memset(op->last_frames, 0, msg_head->nframes * CFSIZ);
```

This memset which zeros out last_frames, highly likely to be an adjacent allocation, with the same malformed length, effectively nullifying our shmid smash:



To work around this, we take advantage of the fact that copy_from_user can perform partial writes on x86 and trigger an EFAULT by setting up a truncated copy for the memcpy_fromiovec operation, allowing us to smash the necessary amount of memory and then pop out and return early before the memset operation occurs:



We then perform a dry-run and detect the shmid smash via an EIDRM errno from shmat() caused by an invalid ipc_perm sequence number. Once we're sure we have a shmid_kernel under our control we re-smash it with the malformed version. By invoking invoking shmat(2) on the smashed shmid, we cause the kernel to dereference the mmap function pointer which is now under our control, as seen in ipc/shm.c:

```
static int shm_mmap(struct file *file, struct vm_area_struct* vma)
{
    ...
    ret = sfd->file->f_op->mmap(sfd->file, vma);
    if (ret != 0)
        return ret;
    ...
}
```

Which redirects control flow to our credential modifying function mapped in user space, escalating our privileges:

```
int __attribute__((regparm(3)))
kernel_code(struct file *file, void *vma)
{
    commit_creds(prepare_kernel_cred(0));
    return -1;
}
```

The full exploit is available. It is targeted for 32-bit Ubuntu Lucid 10.04 (2.6.32-21-generic), but ports easily to other vulnerable kernels/distros.

Posted Fri 10 September 2010 by jono

Copyright © 2018 - Jon Oberheide