Exploiting Stack Overflows in the Linux Kernel

In this post, I'll introduce an exploitation technique for kernel stack overflows in the Linux kernel. Keep in mind this does not refer to buffer overflows on the kernel stack (whose exploitability is well understood), but rather the improper expansion of the kernel stack causing it to overlap with critical structures which may be subsequently corrupted. This is a vulnerability class in the Linux kernel that I do not believe have been exploited publicly in the past, but is relevant due to a recent vulnerability in the Econet packet family.

Kernel Stack Layout

On Linux, every thread on your system has a corresponding kernel stack allocated in kernel memory. Linux kernel stacks on x86 are either 4096 or 8192 bytes in size, depending on your distribution. While this size may seem small to contain a full call chain and associated local stack variables, in reality the kernel call chains are relatively shallow and kernel functions are discouraged from abusing the precious space with large local stack variables when efficient allocators such as the SLUB are available.

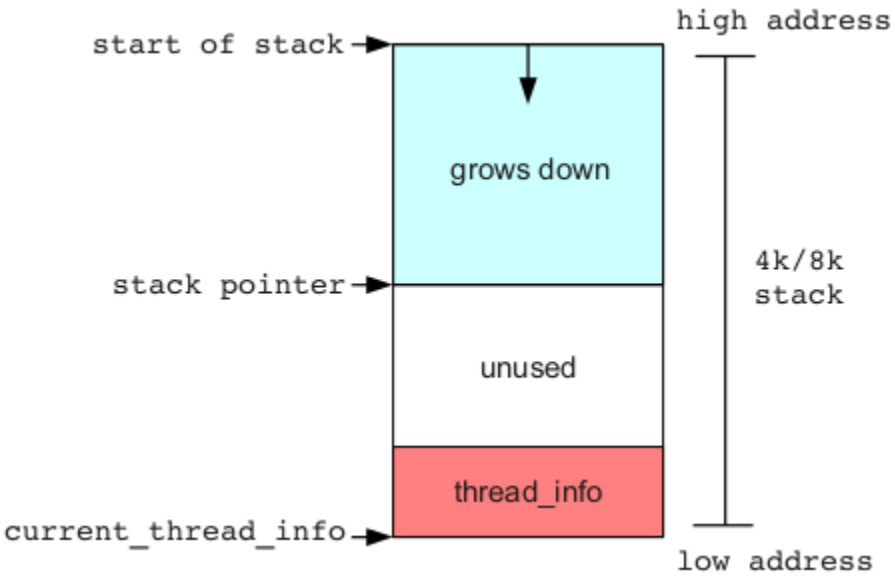
The stack shares the 4k/8k total size with the thread_info structure, which contains some metadata about the current thread, as seen in include/linux/sched.h:

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

The thread_info structure has the following definition on x86 from arch/x86/include/asm/thread_info.h:

```
struct thread_info {
    struct task_struct  *task;
    struct exec_domain  *exec_domain;
    __u32                flags;
    __u32                status;
    __u32                cpu;
    int                  preempt_count;
    mm_segment_t         addr_limit;
    struct restart_block restart_block;
    void __user         *sysenter_return;
#ifdef CONFIG_X86_32
    unsigned long        previous_esp;
    __u8                 supervisor_stack[0];
#endif
    int                  uaccess_err;
};
```

Visually, a kernel stack looks like the following in memory:



So what happens when a function in the kernel requires more than 4k/8k worth of stack space or a long call chain exceeds the available stack space? Well, normally an overflow of the stack will occur and cause the kernel to crash if the thread_info structure or critical memory beyond it becomes corrupted. However, if the moons align and we have a situation where we can actually control the data that is written to the stack and beyond, we may have an exploitable condition.

Exploiting a Stack Overflow

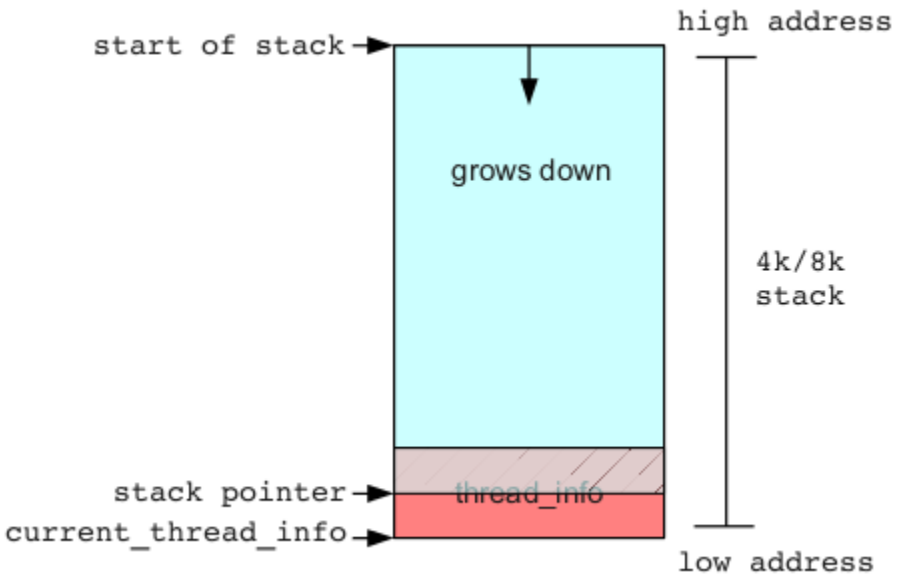
Let's look at a simple example to see how overflowing the stack and clobbering the thread_info structure can result in an exploitable privilege escalation:

```
static int blah(int __user *vals, int __user count)
{
    int i;
    int big_array[count];
    for (i = 0; i < count; ++count) {
        big_array[i] = vals[i];
    }
}
```

In the above code, we have a variable length array on the stack (big_array) whose size is based on an attacker-controlled count. Variable length arrays are allowed in C99 and supported by GCC. GCC will simply calculate the necessary size at runtime and decrement the stack pointer appropriately to allocate space on the stack for the array.

However, if the attacker provides a sufficiently large count, the stack may extend down past the boundary of thread_info, allowing the attacker to subsequently write arbitrary values into the

thread_info structure. Extending the stack pointer past the thread_info boundary would look like the following:



So what is in the thread_info structure that may be useful for an attacker to control? Ideally, we'd like to find something with a function pointer that we can overwrite and redirect control flow to an address of our choosing.

Let's take a deeper look at one promising member of thread_info: restart_block. restart_block is a per-thread structure used to track information and arguments for restarting system calls. System calls that are interrupted by signals can either abort and return EINTR or automatically restart themselves if SA_RESTART is specified in sigaction(2). restart_block is defined as follows in include/linux/thread_info.h:

```
struct restart_block {
    long (*fn)(struct restart_block *);
    union {
        struct {
            ...
        };
        /* For futex_wait and futex_wait_requeue_pi */
        struct {
            ...
        } futex;
        /* For nanosleep */
        struct {
            ...
        } nanosleep;
        /* For poll */
        struct {
            ...
        } poll;
    };
```

```
};
```

Hey, that fn function pointer sure looks promising! Where in the kernel does that function pointer actually get invoked? Why, right there in the restart_syscall system call in kernel/signal.c:

```
SYSCALL_DEFINE0(restart_syscall)
{
    struct restart_block *restart = &current_thread_info()->restart_block;
    return restart->fn(restart);
}
```

The restart_syscall system call is defined in arch/x86/kernel/syscall_table_32.S:

```
.long sys_restart_syscall   /* 0 - old "setup()" system call, used for
restarting */
```

That's right, there's actually system call number zero. We couldn't ask for anything easier! We can trivially invoke its functionality from userspace via:

```
syscall(SYS_restart_syscall);
```

Thereby causing the kernel to call the function pointer contained in the restart_block structure.

So there we have it: if we can clobber the function pointer in the restart_block member of thread_info, we can point it to a function in userspace under our control, trigger its execution by invoking sys_restart_syscall, and escalate privileges.

Econet Vulnerability

As I mentioned previously, I'm not aware of any exploits that have used such a technique in the past. And while the simple example above may appear a bit contrived, there is a real-world example of this type of vulnerability recently discovered by Nelson Elhage in the Econet packet family.

In a forthcoming post, I'll describe the Econet vulnerability and exploit in further detail. Until then, patch up!