Faking kernel pointers as user space pointers

The Linux kernel has routines that emulate system calls within the kernel. The most obvious ones are the socket routines from the net subsystem as they are widely used by many other subsystems. They are pretty handy and most are just wrappers around the functions that do the heavy lifting of the system calls.

```
int
kernel_setsockopt(struct socket *sock,
    int level, int optname, char *optval,
     unsigned int optlen)
{
  mm_segment_t oldfs = get_fs();
  char __user *uoptval;
  int err;

  uoptval = (char __user __force *)
      optval;

  set_fs(KERNEL_DS);
  if (level == SOL_SOCKET)
    err = sock_setsockopt(sock, level,
        optname, uoptval, optlen);
  else
    err = sock->ops->setsockopt(sock,
        level, optname, uoptval, optlen);
  set_fs(oldfs);
  return err;
}
```

Listing 1: *setsockopt* in-kernel variant[1]

Listing 1 is the actual implementation of the in-kernel variant of the *setsockopt* system call. When the kernel interacts with the user space via a system call, it may need to copy data from user space to do something useful. In case of *setsockopt*, the buffer pointed to by *optval* may have a length of size *optlen* and may refer to the many options available via *optname*.

Some may point out that there must be some sort of special type cast with the macros *__user __force*. In fact, this special type cast has the purpose to do a semantic notation for a tool called 'smatch'[2]and it's not actually "faking" the kernel pointer into a user pointer. Also note that the cast will not impose any performance penalty, as through common sub-expression elimination, the cast is likely to disappear. The actual "faking" occurs on the call *set_fs(KERNEL_DS)*.

```
static inline unsigned long
_copy_from_user(void *to, const void
    __user *from, unsigned long n)
{
  unsigned long res = n;
  might_fault();
  if (likely(access_ok(VERIFY_READ, from,
       n))) {
    kasan_check_write(to, n);
    res = raw_copy_from_user(to, from,
      n);
  }
  if (unlikely(res))
    memset(to + (n - res), 0, res);
  return res;
}
```

Listing 2: Implementation of *_copy_from_user*[3]

When copying from a user buffer, the kernel will use a function called *_copy_from_user*. The implementation checks if the user buffer belongs to the user portion of the virtual address space, if that's the case it may proceed with the copy. The check is performed by the macro *access_ok* and its implementation is in Listing 3[4].

```
#define access_ok(type, addr, size)      \
({                                       \
  WARN_ON_IN_IRQ();                      \
  likely(!__range_not_ok(addr, size,     \
        user_addr_max()));               \
})
```

Listing 3: Implementation of *access_ok*[5]

The call *set_fs(KERNEL_DS)* will set the maximum user address of the current running thread to the maximum address possible, therefore bypassing the *access_ok* check in a kernel buffer. After calling the system call, the previous user address is restored via *set_fs(oldfs)*.

In modern operating systems with paging based virtual memory, the virtual address space is split between two parts (user/kernel), introducing more semantics to virtual memory pointers. On 32-bit, the user space virtual address gets most of the virtual address space, this usually accounts to 3GiB of the total 4GiB address space. The rest is left to the kernel. Checking whether a pointer is from user space or kernel space needs just a simple arithmetic operation.

In the Linux kernel, the split address can be set via the configuration system using the *CONFIG_PAGE_OFFSET* option. Some predefined virtual address space layouts can also be found in the configuration system.

[1]Code style adapted.
[2]A tool for static analysis of C code.
[3]Code style adapted.
[4]Original comments removed.
[5]The kernel may also use high memory mappings when under memory pressure.