Faking user space pointers

Pedro Tammela

Jun 29 2019

I've come across an interesting technique used by the Linux kernel, across a lot of modules but especially in the in-house variants of the sockets system calls, to fake kernel pointers as user space pointers and call system calls directly.

Let's see the original code.

```
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;
}
```

`kernel_setsockopt()` should behave exactly as `setsockopt()`. A major concern is that all system calls expects user space pointers to be sent and this is explicit told to the compiler using the macro `__user`.

This can be pretty annoying when you are developing near the user space, a real world example is a in-kernel proxy server. All socket system calls are ported to the kernel space, exactly as the code above shows, with proper pointer space transformation and error handling, but what if we want to use other features such as `poll()`, `epoll()`, `select()` or any other system call?. These system calls have no kernel counter part, which is totally frustrating for our in-kernel proxy server.

Looking our code example, we can see that to force a pointer transformation we must do this:

1. Create the desired pointer with the `__user` macro.
2. Assign it the cast of our kernel pointer.
3. Cast our kernel pointer using the macros `__user __force`

So what's magical about these macros?

Nothing.

All these macros are for semantics notation for the sparse tool, written for kernel developers to mitigate incorrect pointer handling.

You might be asking yourself, "So it was all a lie?"

In some sense, yes. Let me show you.

The actual trick is performed on the horrible named functions `get_fs()` and `set_fs()`. These functions modifies the `current` maximum address space in physical memory. This means that when we are in process context, context is set to the maximum user space address and will break any `*_from_user()` as kernel addresses are not in range.

```
/**
 * access_ok: - Checks if a user space pointer is valid
 * @type: Type of access: %VERIFY_READ or %VERIFY_WRITE.  Note that
 *        %VERIFY_WRITE is a superset of %VERIFY_READ - if it is safe
 *        to write to a block, it is always safe to read from it.
 * @addr: User space pointer to start of block to check
 * @size: Size of block to check
 *
 * Context: User context only. This function may sleep if pagefaults are
 *          enabled.
 *
 * Checks if a pointer to a block of memory in user space is valid.
 *
 * Returns true (nonzero) if the memory block may be valid, false (zero)
 * if it is definitely invalid.
 *
 * Note that, depending on architecture, this function probably just
 * checks that the pointer is in the user space range - after calling
 * this function, memory access functions may still return -EFAULT.
 */
#define access_ok(type, addr, size)                 \
({                                      \
    WARN_ON_IN_IRQ();                       \
    likely(!__range_not_ok(addr, size, user_addr_max()));       \
})
```

```
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;
}
```

Using `set_fs(KERNEL_DS)` sets the maximum address to the last memory address possible, therefore bypassing all address range checks and turning the functions into fanciers versions of `memcpy()`.