

# Coccinelle Users Day – Exercises

November 25, 2009

These exercises are presented roughly in order of increasing difficulty. Often there are multiple possible solutions to the exercise, which may produce different sets of reports and different sets of false positives.

## 1 Device Data

The Linux file `include/linux/device.h` defines the following functions

```
static inline void *dev_get_drvdata(const struct device *dev)
{
    return dev->driver_data;
}

static inline void dev_set_drvdata(struct device *dev, void *data)
{
    dev->driver_data = data;
}
```

These functions are used 1364 and 439 times in the Linux 2.6.30 kernel, but are not used everywhere they could be.

Write a semantic patch to remove all explicit references to the `driver_data` field of a `device` structure.

Running this rule on the entire Linux kernel takes some time. Testing it on `drivers/net` and `drivers/ieee1394` is sufficient to obtain some results.

Check that there are no occurrences of the `driver_data` field that are not covered by your rule.

## 2 Zeroed Memory Allocation

### 2.1 Part 1

In Linux, the basic memory allocation function is `kmalloc`. A common pattern, however, is to allocate a region of memory and then zero its elements using the function `memset`. An example of this pattern is as follows, from the file `drivers/staging/meilhaus/me0600_device.c`:

```
me0600_device = kmalloc(sizeof(me0600_device_t), GFP_KERNEL);
if (!me0600_device) {
    PERROR("Cannot get memory for device instance.\n");
    return NULL;
}
memset(me0600_device, 0, sizeof(me0600_device_t));
```

Rather than having first the call to `kmalloc` and then the call to `memset`, it was judged to be better to encapsulate these operations in a single function, `kzalloc`, that both allocates and zeros the memory. Using `kzalloc`, the above code would be written as:

```
me0600_device = kzalloc(sizeof(me0600_device_t), GFP_KERNEL);
if (!me0600_device) {
    PERROR("Cannot get memory for device instance.\n");
    return NULL;
}
```

Write a semantic patch to perform this transformation throughout the Linux kernel. To reduce the running time, it would be sufficient to test your rule on `drivers/staging`.

### 2.2 Part 2

When the memory to be allocated is to be used as an array, it is better to use the function `kcalloc`. This function does essentially the same thing as `kzalloc`, but it checks the dimensions of the array (number of elements and size of each element) for possible overflow before multiplying these values to compute the total size. An example of code that could be written using `kcalloc` is as follows, from `drivers/staging/go7007/go7007-usb.c`:

```
gofh->bufs = kmalloc(count * sizeof(struct go7007_buffer), GFP_KERNEL);
if (!gofh->bufs) {
    up(&go->hw_lock);
    goto unlock_and_return;
}
memset(gofh->bufs, 0, count * sizeof(struct go7007_buffer));
```

This code should be rewritten as:

```
gofh->bufs = kcalloc(count, sizeof(struct go7007_buffer), GFP_KERNEL);
if (!gofh->bufs) {
    up(&go->hw_lock);
    goto unlock_and_return;
}
```

Extend your semantic patch from part 1 to perform this transformation.

### 3 NULL pointer dereferences

Last summer Brad Spengler constructed a Linux kernel exploit that was enabled by the following fragment of code in `drivers/net/tun.c`:

```
static unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;
    ...
}
```

The problem is that the variable `tun` is dereferenced before it is checked for being `NULL`, and it can actually be `NULL` in practice. The solution is to move the initialization of `sk` below the `NULL` test.

Write a semantic patch to perform this transformation. There are many ways to do this, and those that are more general also tend to find more false positives. One approach, which should have no false positives, is to try to stay very close to the pattern illustrated by the example above.

## 4 More NULL pointer dereferences

Often when an error is detected, one would like to print out some information about the context in which the error occurs. It turns out that when the error is that some pointer is NULL, it is a common mistake to try to print out some information that should have been stored under that pointer. An example is as follows, from `drivers/media/dvb/firewire/firedtv-1394.c`:

```
if (!fdtv) {
    dev_err(fdtv->device, "received at unknown iso channel\n");
    goto out;
}
```

Write a semantic patch to detect dereferences under a NULL test. Note that such a dereference may occur any number of times.

## 5 Unchecked memory allocation

### 5.1 Part 1

Memory allocation using `kmalloc`, `kzalloc`, or `kcalloc` can fail, in which case the result is NULL. The calling code should thus always check that it has received a valid pointer. Nevertheless, some code does not perform this check, as illustrated by the following, from `drivers/staging/stlc45xx/stlc45xx.c`.

```
entry = kmalloc(sizeof(*entry), GFP_ATOMIC);
entry->start = pos;
```

Write a semantic patch to find cases where the result of calling `kmalloc` is dereferenced with no previous NULL pointer test.

### 5.2 Part 2

Sometimes a helper function is defined to perform the memory allocation. In the simplest case, this function just returns the result of calling `kmalloc`. The result of calling such a helper function should be tested for NULL as well before being dereferenced. Write a semantic patch to find cases where the result of calling such a helper function is dereferenced with no previous NULL pointer test.

## 6 & on function names

Some Linux code uses & in front of a function name when the function name is used as an expression, even though this is not necessary.

```
i = request_irq(dev->irq, &el3_interrupt, 0, dev->name, dev);
```

Although this code is not incorrect, it is odd to do this in the case where everywhere else in the file, function names are used as expressions directly. Write a semantic patch to do the following:

1. Count the number of occurrences of a function name with & and the number of occurrences without &.
2. Detect the case where there is only one occurrence of a function name with & and many occurrences without &.
3. Transform the occurrence of a function name with & so that it does not use &.