Deprecated Interfaces, Language Features, Attributes, and Conventions

In a perfect world, it would be possible to convert all instances of some deprecated API into the new API and entirely remove the old API in a single development cycle. However, due to the size of the kernel, the maintainership hierarchy, and timing, it's not always feasible to do these kinds of conversions at once. This means that new instances may sneak into the kernel while old ones are being removed, only making the amount of work to remove the API grow. In order to educate developers about what has been deprecated and why, this list has been created as a place to point when uses of deprecated things are proposed for inclusion in the kernel.

__deprecated

While this attribute does visually mark an interface as deprecated, it does not produce warnings during builds any more because one of the standing goals of the kernel is to build without warnings and no one was actually doing anything to remove these deprecated interfaces. While using *__deprecated* is nice to note an old API in a header file, it isn't the full solution. Such interfaces must either be fully removed from the kernel, or added to this file to discourage others from using them in the future.

BUG() and BUG_ON()

Use WARN() and WARN_ON() instead, and handle the "impossible" error condition as gracefully as possible. While the BUG()-family of APIs were originally designed to act as an "impossible situation" assert and to kill a kernel thread "safely", they turn out to just be too risky. (e.g. "In what order do locks need to be released? Have various states been restored?") Very commonly, using BUG() will destabilize a system or entirely break it, which makes it impossible to debug or even get viable crash reports. Linus has very strong feelings about this.

Note that the WARN()-family should only be used for "expected to be unreachable" situations. If you want to warn about "reachable but undesirable" situations, please use the pr_warn()-family of functions. System owners may have set the *panic_on_warn* sysctl, to make sure their systems do not continue running in the face of "unreachable" conditions. (For example, see commits like this one.)

open-coded arithmetic in allocator arguments

Dynamic size calculations (especially multiplication) should not be performed in memory allocator (or similar) function arguments due to the risk of them overflowing. This could lead to values wrapping around and a smaller allocation being made than the caller was expecting. Using those allocations could lead to linear overflows of heap memory and other misbehaviors. (One exception to this is literal values where the compiler can warn if they might overflow. However, the preferred way in these cases is to refactor the code as suggested below to avoid the open-coded arithmetic.)

For example, do not use `count * size` as an argument, as in:

```
foo = kmalloc(count * size, GFP_KERNEL);
```

Instead, the 2-factor form of the allocator should be used:

```
foo = kmalloc_array(count, size, GFP_KERNEL);
```

Specifically, kmalloc() can be replaced with kmalloc_array(), and kzalloc() can be replaced with kcalloc().

If no 2-factor form is available, the saturate-on-overflow helpers should be used:

```
bar = vmalloc(array_size(count, size));
```

Another common case to avoid is calculating the size of a structure with a trailing array of others structures, as in:

```
header = kzalloc(sizeof(*header) + count * sizeof(*header->item),
                 GFP_KERNEL);
```

Instead, use the helper:

```
header = kzalloc(struct_size(header, item, count), GFP_KERNEL);
```

> **Note**
>
> If you are using struct_size() on a structure containing a zero-length or a one-element array as a trailing array member, please refactor such array usage and switch to a flexible array member instead.

For other calculations, please compose the use of the size_mul(), size_add(), and size_sub() helpers. For example, in the case of:

```
foo = krealloc(current_size + chunk_size * (count - 3), GFP_KERNEL);
```

Instead, use the helpers:

```
foo = krealloc(size_add(current_size,
                        size_mul(chunk_size,
                                 size_sub(count, 3))), GFP_KERNEL);
```

For more details, also see array3_size() and flex_array_size(), as well as the related check_mul_overflow(), check_add_overflow(), check_sub_overflow(), and check_shl_overflow() family of functions.

simple_strtol(), simple_strtoll(), simple_strtoul(), simple_strtoull()

The simple_strtol(), simple_strtoll(), simple_strtoul(), and simple_strtoull() functions explicitly ignore overflows, which may lead to unexpected results in callers. The respective kstrtol(), kstrtoll(), kstrtoul(), and kstrtoull() functions tend to be the correct replacements, though note that those require the string to be NUL or newline terminated.

strcpy()

strcpy() performs no bounds checking on the destination buffer. This could result in linear overflows beyond the end of the buffer, leading to all kinds of misbehaviors. While *CONFIG_FORTIFY_SOURCE=y* and various compiler flags help reduce the risk of using this function, there is no good reason to add new uses of this function. The safe replacement is strscpy(), though care must be given to any cases where the return value of strcpy() was used, since strscpy() does not return a pointer to the destination, but rather a count of non-NUL bytes copied (or negative errno when it truncates).

strncpy() on NUL-terminated strings

Use of strncpy() does not guarantee that the destination buffer will be NUL terminated. This can lead to various linear read overflows and other misbehavior due to the missing termination. It also NUL-pads the destination buffer if the source contents are shorter than the destination buffer size, which may be a needless performance penalty for callers using only NUL-terminated strings. The safe replacement is strscpy(), though care must be given to any cases where the return value of strncpy() was used, since strscpy() does not return a pointer to the destination, but rather a count of non-NUL bytes copied (or negative errno when it truncates). Any cases still needing NUL-padding should instead use strscpy_pad().

If a caller is using non-NUL-terminated strings, strncpy() can still be used, but destinations should be marked with the __nonstring attribute to avoid future compiler warnings.

strlcpy()

strlcpy() reads the entire source buffer first (since the return value is meant to match that of strlen()). This read may exceed the destination size limit. This is both inefficient and can lead to linear read overflows if a source string is not NUL-terminated. The safe replacement is strscpy(), though care must be given to any cases where the return value of strlcpy() is used, since strscpy() will return negative errno values when it truncates.

%p format specifier

Traditionally, using "%p" in format strings would lead to regular address exposure flaws in dmesg, proc, sysfs, etc. Instead of leaving these to be exploitable, all "%p" uses in the kernel are being printed as a hashed value, rendering them unusable for addressing. New uses of "%p" should not be added to the kernel. For text addresses, using "%pS" is likely better, as it produces the more useful symbol name instead. For nearly everything else, just do not add "%p" at all.

Paraphrasing Linus's current guidance:

- If the hashed "%p" value is pointless, ask yourself whether the pointer itself is important. Maybe it should be removed entirely?
- If you really think the true pointer value is important, why is some system state or user privilege level considered "special"? If you think you can justify it (in comments and commit log) well enough to stand up to Linus's scrutiny, maybe you can use "%px", along with making sure you have sensible permissions.

If you are debugging something where "%p" hashing is causing problems, you can temporarily boot with the debug flag "no_hash_pointers".

Variable Length Arrays (VLAs)

Using stack VLAs produces much worse machine code than statically sized stack arrays. While these non-trivial performance issues are reason enough to eliminate VLAs, they are also a security risk. Dynamic growth of a stack array may exceed the remaining memory in the stack segment. This could lead to a crash, possible overwriting sensitive contents at the end of the stack (when built without *CONFIG_THREAD_INFO_IN_TASK=y*), or overwriting memory adjacent to the stack (when built without *CONFIG_VMAP_STACK=y*)

Implicit switch case fall-through

The C language allows switch cases to fall through to the next case when a "break" statement is missing at the end of a case. This,

however, introduces ambiguity in the code, as it's not always clear if the missing break is intentional or a bug. For example, it's not obvious just from looking at the code if *STATE_ONE* is intentionally designed to fall through into *STATE_TWO*:

```
switch (value) {
case STATE_ONE:
        do_something();
case STATE_TWO:
        do_other();
        break;
default:
        WARN("unknown state");
}
```

As there have been a long list of flaws due to missing "break" statements, we no longer allow implicit fall-through. In order to identify intentional fall-through cases, we have adopted a pseudo-keyword macro "fallthrough" which expands to gcc's extension __attribute__((__fallthrough__)). (When the C17/C18 *[[fallthrough]]* syntax is more commonly supported by C compilers, static analyzers, and IDEs, we can switch to using that syntax for the macro pseudo-keyword.)

All switch/case blocks must end in one of:

- break;
- fallthrough;
- continue;
- goto <label>;
- return [expression];

Zero-length and one-element arrays

There is a regular need in the kernel to provide a way to declare having a dynamically sized set of trailing elements in a structure. Kernel code should always use "flexible array members" for these cases. The older style of one-element or zero-length arrays should no longer be used.

In older C code, dynamically sized trailing elements were done by specifying a one-element array at the end of a structure:

```
struct something {
        size_t count;
        struct foo items[1];
};
```

This led to fragile size calculations via sizeof() (which would need to remove the size of the single trailing element to get a correct size of the "header"). A GNU C extension was introduced to allow for zero-length arrays, to avoid these kinds of size problems:

```
struct something {
        size_t count;
        struct foo items[0];
};
```

But this led to other problems, and didn't solve some problems shared by both styles, like not being able to detect when such an array is accidentally being used _not_ at the end of a structure (which could happen directly, or when such a struct was in unions, structs of structs, etc).

C99 introduced "flexible array members", which lacks a numeric size for the array declaration entirely:

```
struct something {
        size_t count;
        struct foo items[];
};
```

This is the way the kernel expects dynamically sized trailing elements to be declared. It allows the compiler to generate errors when the flexible array does not occur last in the structure, which helps to prevent some kind of undefined behavior bugs from being inadvertently introduced to the codebase. It also allows the compiler to correctly analyze array sizes (via sizeof(), *CONFIG_FORTIFY_SOURCE*, and *CONFIG_UBSAN_BOUNDS*). For instance, there is no mechanism that warns us that the following application of the sizeof() operator to a zero-length array always results in zero:

```
struct something {
        size_t count;
        struct foo items[0];
};

struct something *instance;

instance = kmalloc(struct_size(instance, items, count), GFP_KERNEL);
instance->count = count;

size = sizeof(instance->items) * instance->count;
memcpy(instance->items, source, size);
```

At the last line of code above, `size` turns out to be `zero`, when one might have thought it represents the total size in bytes of the dynamic memory recently allocated for the trailing array `items`. Here are a couple examples of this issue: link 1, link 2. Instead, flexible array members have incomplete type, and so the sizeof() operator may not be applied, so any misuse of such operators will be immediately noticed at build time.

With respect to one-element arrays, one has to be acutely aware that such arrays occupy at least as much space as a single object of the type, hence they contribute to the size of the enclosing structure. This is prone to error every time people want to calculate the total size of dynamic memory to allocate for a structure containing an array of this kind as a member:

```
struct something {
        size_t count;
        struct foo items[1];
};

struct something *instance;

instance = kmalloc(struct_size(instance, items, count - 1), GFP_KERNEL);
instance->count = count;

size = sizeof(instance->items) * instance->count;
memcpy(instance->items, source, size);
```

In the example above, we had to remember to calculate `count - 1` when using the struct_size() helper, otherwise we would have -- unintentionally-- allocated memory for one too many `items` objects. The cleanest and least error-prone way to implement this is through the use of a *flexible array member*, together with struct_size() and flex_array_size() helpers:

```
struct something {
        size_t count;
        struct foo items[];
};

struct something *instance;

instance = kmalloc(struct_size(instance, items, count), GFP_KERNEL);
instance->count = count;

memcpy(instance->items, source, flex_array_size(instance, items, instance->count));
```