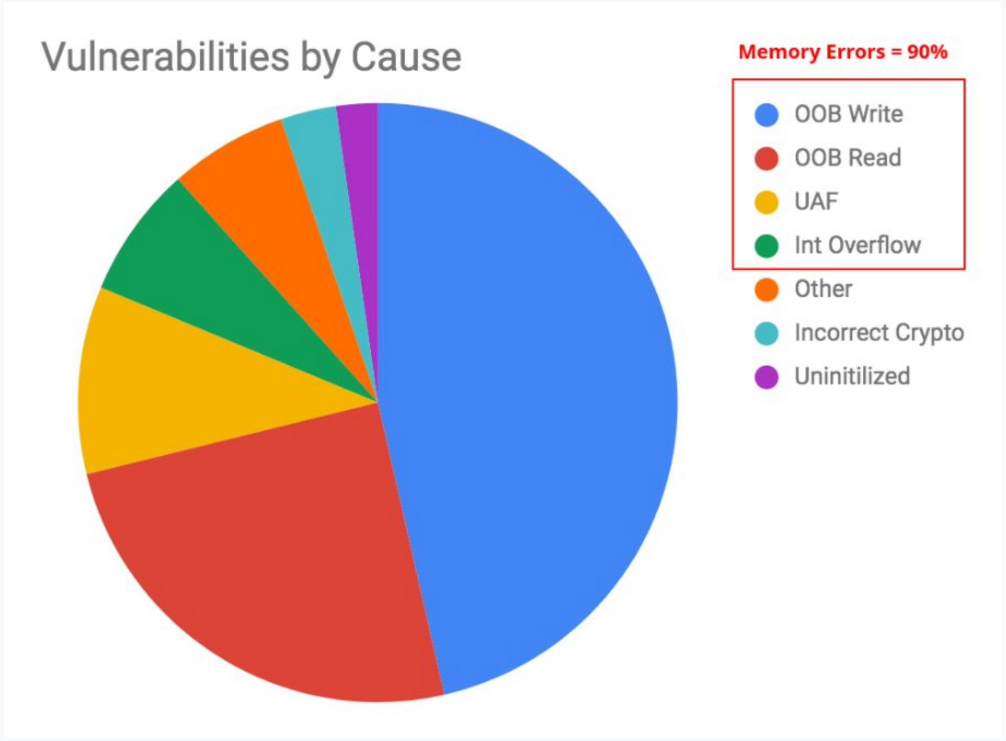
Integer Overflow Prevention BoF

LPC 24

Dan Carpenter
Kees Cook
Justin Stitt

Prevalence of Integer Overflows



Integer overflows are seen as the 4th largest class of vulnerabilities, but they often go unrecognized as the root cause of Out-of-Bounds (OOB) reads and writes (e.g. buffer overflows), so their true prevalence is higher.

Example: Buffer overflow masking an integer overflow

```
u8 i = 0, num_elems = 0;
...
nla_for_each_nested(nl_elems, attrs, rem_elems)
    num_elems++; /* Truncates when >255 elements: an integer overflow */
elems = kzalloc(struct_size(elems, elem, num_elems), GFP_KERNEL);
...
nla_for_each_nested(nl_elems, attrs, rem_elems) {
    elems->elem[i++] = nla_data(nl_elems); /* Observed as OOB Write */
    ...
}
```

Fixed in 6311071a0562

Google

Types of Integer overflows (and truncations)

Necessary

Intentional

Unintentional but harmless

Buggy

Idiom Exclusions (Necessary overflows)

There exists some overflow-dependent code patterns that rely on the results of well-defined arithmetic wrap-around (even for pointers and signed types via `-fno-strict-overflow`).

- `if (a + b < a) ...`
- `while (i--)`
- `-1UL, -2UL, -{N}UL;`
- `end = start + length - 1;`

Intentional overflows

```
/* Align to pointer size and check for overflows */
tmp = ALIGN(data_size, sizeof(void *)) +
        ALIGN(offsets_size, sizeof(void *));
if (tmp < data_size || tmp < offsets_size)
        return 0;
```

or

```
crc += byte;
```

Unintentional overflows

```
p = xdr_inline_decode(xdr,
            rc_list->rcl_nrefcalls * 2 * sizeof(uint32_t));
....
rc_list->rcl_refcalls = kmalloc_array(rc_list->rcl_nrefcalls,
                        sizeof(*rc_list->rcl_refcalls),
                        GFP_KERNEL);
```

Implicit hidden code-flow choices

All arithmetic operations produce 2 potential code-flows:

- Path 1: no overflow has happened, carry on
- Path 2: overflow has happened! What do?

C does not provide a choice, and forces one path by default. (Some languages use exceptions to access other path – C must use the sanitizers to do that...)

x = a + b; // both paths exist – cannot be distinguished, forces Path 1.
x = -1UL; // Path 1 cannot exist, forces Path 2.
if (a + b < a) { /* path 2 */ } else { /* path 1 */ } // paths distinguished!

Silencing false positives

Wrappers?

```
x = overflows_ok(a + b);
```

Attributes?

```
size_t x __wraps;
x = a + b;

typedef unsigned long __nowraps size_t;
size_t y;
y = a + b;
```

ALIGN and round_up()

```
nctx_len = ALIGN(struct_size(nctx, ctx, val_len), sizeof(void *));
if (nctx_len > *uctx_len)
        return -E2BIG;

nctx = kzalloc(nctx_len, GFP_KERNEL);
```

PTR_ERR()

```
static inline long __must_check PTR_ERR(__force const void *ptr)
{
	return (long) ptr;
}

int function(...)
{
	...
	if (IS_ERR(result))
		return PTR_ERR(result); // negative long truncated to negative int
	...
}
```

But PTR_ERR_OR_ZERO() is actually an int return!