Variable Length Arrays[VLA]

WHAT ARE VLA'S?

- Variable-length arrays (VLAs) have a non-constant size that is determined (and which can vary) at run time.
- C supports variable sized arrays from C99 standard.

- Ex:

```
void fun(int n)
{
    int arr[n];
    //…
    //....
}
int main()
{
    fun(10);
}
```

VLA’S & LINUX KERNEL

- Use of VLAs in the kernel has long been discouraged but not prohibited.
- A recent push was made by Kees Cook to remove VLAs from the kernel.
- Ex:

```
static int tls_read_size(struct strparser *strp, struct sk_buff *skb)
{
        struct tls_context *tls_ctx = tls_get_ctx(strp->sk);
        struct tls_sw_context *ctx = tls_sw_ctx(tls_ctx);
        char header[tls_ctx->rx.prepend_size];
        struct strp_msg *rxm = strp_msg(skb);
size_t cipher_overhead; size_t data_len = 0; int ret;
```

- This effort let to an interesting problem.

THE PROBLEM:

```
int btree_merge(struct btree_head *target, struct btree_head *victim, struct btree_geo
*geo, gfp_t gfp)
{
     unsigned long key[geo->keylen];
     unsigned long dup[geo->keylen];
     void *val;
     int err;
```

The length of both the `key` and `dup` arrays is determined by the value stored in the `keylen` field of the passed-in `geo` structure. The compiler cannot know what that value will be at compile time, so those arrays must be allocated at run time.

Why Remove VLA's?
1. They add a bit of run-time overhead, since the size of a
VLA must be calculated every time that the function declaring it is called
2. Stacks in the kernel are small.

THE PROBLEM: CONT...

- Kees Cook found with –Wvla, GCC was issuing warning for arrays that were meant to be of constant size. Eg: lib/vsprintf.c

```
#define RSRC_BUF_SIZE ((2 * sizeof(resource_size_t)) + 4)
#define FLAG_BUF_SIZE (2 * sizeof(res->flags))
#define DECODED_BUF_SIZE sizeof("[mem - 64bit pref window disabled]")
#define RAW_BUF_SIZE sizeof("[mem - flags 0x]")

char sym[max(2*RSRC_BUF_SIZE + DECODED_BUF_SIZE,
             2*RSRC_BUF_SIZE + FLAG_BUF_SIZE + RAW_BUF_SIZE)];
```

- The length of sym is clearly a constant and can be determined at compile time, but GCC wars about VLA. The problem turns out to be kernel's max()macro.
- The Problem is, ISO C90 requires an array size to be not a constant value, but a constant expression.

MAX()

- Simplest version of it from K & R book:

```
#define max(A,B) ((A) > (B) ? (A) : (B))
```

- Kernel's version:

```
#define __max(t1, t2, max1, max2, x, y) ({ \
t1 max1 = (x); \
t2 max2 = (y); \
(void) (&max1 == &max2); \
max1 > max2 ? max1 : max2; })

#define max(x, y) \
    __max(typeof(x), typeof(y), \
                __UNIQUE_ID(max1_), __UNIQUE_ID(max2_), \
                x, y)
```

MAX() CONT…

Kees’s version:

```
#define __single_eval_max(t1, t2, max1, max2, x, y) ({      \
     t1 max1 = (x);                         \
     t2 max2 = (y);                         \
     (void) (&max1 == &max2);                    \
     max1 > max2 ? max1 : max2; })

#define __max(t1, t2, x, y)                         \
     __builtin_choose_expr(__builtin_constant_p(x) &&          \
                           __builtin_constant_p(y),            \
                           (t1)(x) > (t2)(y) ? (t1)(x) : (t2)(y),       \
                           __single_eval_max(t1, t2,           \
                      __UNIQUE_ID(max1_), \
                      __UNIQUE_ID(max2_), \
                      x, y))

#define max(x, y) __max(typeof(x), typeof(y), x, y)
```

THE HACK!

#define ICE_P(x) (sizeof(int) == sizeof(*(1 ? ((void*)((x) * 0l)) : (int*)1)))

Breaking it down:

sizeof(*(1 ? ((void*)((x) * 0l)) : (int*)1))

Left hand side always returns 1:

1 ? ((void*)((x) * 0l)) : (int*)1

As Linus explains here: https://lkml.org/lkml/2018/3/20/845, when x is a interger constant expression, Left hand side becomes NULL.

– when x is ICE

1 ? ((void*)(NULL)) : (int*)1

and result: sizeof(int) == sizeof(*(int *))– when x is ICE

– when is not an ICE

1 ? ((void*)(NOT-NULL)) : (int*)1

and result: sizeof(int) == sizeof(*(void *))

For non-ICEs the cast to void * will result in a non-NULL pointer.

Dereferencing (void*) is not valid, but sizeof(*(void*)) works resulting in 1.

MAX() NOW:

```
/*
 * This returns a constant expression while determining if an argument is
 * a constant expression, most importantly without evaluating the argument.
 * Glory to Martin Uecker <Martin.Uecker@med.uni-goettingen.de> */
#define __is_constexpr(x) \
     (sizeof(int) == sizeof(*(8 ? ((void *)((long)(x) * 0l)) : (int *)8)))

#define __no_side_effects(x, y) \
     (__is_constexpr(x) && __is_constexpr(y))

#define __safe_cmp(x, y) \
     (__typecheck(x, y) && __no_side_effects(x, y))

#define __cmp(x, y, op) ((x) op (y) ? (x) : (y))

#define __cmp_once(x, y, unique_x, unique_y, op) ({ \
typeof(x) unique_x = (x); \
typeof(y) unique_y = (y); \
__cmp(unique_x, unique_y, op); })

#define __careful_cmp(x, y, op) \
__builtin_choose_expr(__safe_cmp(x, y), \
__cmp(x, y, op), \
__cmp_once(x, y, __UNIQUE_ID(__x), __UNIQUE_ID(__y), op))

#define max(x, y) __careful_cmp(x, y, >)
```

QUESTIONS??

REFERENCES:

- https://lkml.org/lkml/2018/3/20/845
- https://lwn.net/Articles/750306/
- https://lwn.net/Articles/749064/
- https://lwn.net/Articles/749093/