Progress On Bounds Checking in C and the Linux Kernel

Kees Cook & Gustavo A. R. Silva

Linux Security Summit, North America 2023

https://outflux.net/slides/2023/lss-na/bounds-checking.pdf

Agenda

- Goal: memory safety
- 50 years of missing bounds checking
- Problems with existing work-arounds
- Current mitigations lack sufficient coverage
- Improve coverage: Refactor for unambiguous arrays
- Improve coverage: Annotate dynamic array sizes
- Compiler work
- Metrics!

Goal: memory safety

Bounds checking is incomplete in C

One of the more tractable topics under the umbrella of “memory safety” is simple bounds checking: programs must not access outside a specified range of memory. A huge portion of historical security flaws fall under this basic category, and it *could* be solved by the compiler, but the standard C language is too ambiguous.

Protection can be added for arrays, as C effectively treats “`array`” as a pointer to an associated fixed-size region:

```
char array[16]; /* 16 bytes */
```

But if the size isn’t compile-time fixed, there is only a bare pointer type available:

```
char *pointer;  /* unknown region size */
```

Dynamically sized structures are needed

A common data storage pattern is a header followed by some number of the same data structure, but addressing and iterating is cumbersome if they're not part of the struct:

```
struct message {
    unsigned long flags;
    char urgency;
    int item_count;
};

struct item item1;
struct item item2;
...
```

Header

Element count

Elements

```
struct message *p;
struct item *item;

p = kzalloc(sizeof(*p) +
            sizeof(struct item) * count,
            GPF_KERNEL);
item = (struct item *)(p + 1);

for (int i; i < count; i++) {
    do_something(item);
    item = item + 1;
}
```

Pointers for dynamically sized structures are wasteful

Standard C only provides a pointer, and pointing to the area of memory immediately after the struct wastes space and requires explicit initialization:

```
struct message {
    unsigned long flags;
    char urgency;
    int item_count;
    struct item *items;
};
```

Header

Element count

Ptr to elements

```
struct item item1;
struct item item2;
...
```

Elements

```
struct message *p;


p = kzalloc(sizeof(*p) +
            sizeof(struct item) * count,
            GPF_KERNEL);
p->items = (struct item *)(p + 1);

for (int i; i < count; i++)
    do_something(&p->items[i]);
```

Dynamically sized arrays are needed

Just having an array with a size that isn't known in advance is what is needed for this code pattern:

```
struct message {
    unsigned long flags;
    char urgency;
```
Header

```
    int item_count;
```
Element count

```
    struct item items[???];
```
Elements

```
};
```

```
struct message *p;


p = kzalloc(struct_size(p, items, count),
            GPF_KERNEL);



for (int i; i < count; i++)
    do_something(&p->items[i]);
```

50 years of missing bounds checking

Working around lack of trailing dynamically sized arrays

- one-element array hack:

```
struct item elements[1];
```

  - Standard C says an array cannot be zero-sized, so C developers learning from/working on code from the last millennium were forced to lie to the compiler and manually manage bounds.
- C90 GNU extension, zero-length arrays:

```
struct item elements[0];
```

  - Still not standard C, still lying to the compiler: it’s not zero-sized, so diagnostics can go wrong.
- C99 flexible-array member:

```
struct item elements[];
```

  - Not technically lying anymore, but still forcing compiler to be blind to actual size.
- And "oops, this used to be fixed-sized but now it's variable length” arrays

```
struct item elements[N...];
```

Problems with existing work-arounds

Problems with 1-element arrays

- Always “contribute” with **size-of-one-element** to the size of the enclosing structure.
- Developers have to remember to subtract **1** from **count**, or **sizeof**(**struct** foo) from **sizeof**(**struct** ancient).
- Prone to **off-by-one** problems.

```
struct ancient {
        ...
        size_t count;
        struct foo array[1];
} *p;
```

```
alloc_size = sizeof(*p) + sizeof(struct foo) * (p->count - 1);
```

Problems with 1-element arrays

- Tons of `-Warray-bounds` false positives.

```
struct ancient {
        ...
        size_t count;
        struct foo array[1];
} *p;

for(i = 0; i < p->count; i++)
        p->array[i];
```

i == 0 is fine :)
i >= 1 is not :/

```
warning: array subscript 1 is above array bounds of
‘struct foo[1]’ [-Warray-bounds]
```

C90 GNU extension: zero-length arrays

- Not part of the C standard.
- They **don't** contribute to the size of the flexible struct.
- Slightly less buggy, but still…
- Be aware of **sizeof(p->array) == 0**

```
struct old {
        ...
        size_t count;
        struct foo array[0];
} *p;
```

```
alloc_size = sizeof(*p) + sizeof(struct foo) * p->count;
```

Undefined Behavior

- The compiler cannot detect dangerous code like this.
  - “Overlapping” members do not trigger compiler warnings.
- e48f129c2f20 ("[SCSI] cxgb3i: convert cdev->l2opt to use…")

```
struct l2t_data {
        unsigned int nentries;
        struct l2t_entry *rover;
        atomic_t nfree;
        rwlock_t lock;
        struct l2t_entry l2tab[0];
+       struct rcu_head rcu_head;
};
```

Undefined Behavior

- 76497732932f ("cxgb3/l2t: Fix undefined behavior")
- **Kick-off** of flexible array transformation in the KSPP.
- Bug introduced in 2011. Fixed in 2019.

```
struct l2t_data {
        unsigned int nentries;
        struct l2t_entry *rover;
        atomic_t nfree;
        rwlock_t lock;
-       struct l2t_entry l2tab[0];
        struct rcu_head rcu_head;
+       struct l2t_entry l2tab[];
};
```

Undefined Behavior

- f5823fe6897c ("qed: Add ll2 option to limit the number of bds per packet")
- Fake flex-array transformation **from [18] to [1]**.

```
struct qed_ll2_tx_packet {
         ...
+        /* Flexible Array of bds_set determined by max_bds_per_packet */
         struct {
                 struct core_tx_bd *txq_bd;
                 dma_addr_t tx_frag;
                 u16 frag_len;
-        } bds_set[ETH_TX_MAX_BDS_PER_NON_LSO_PACKET];
+        } bds_set[1];
 };

#define ETH_TX_MAX_BDS_PER_NON_LSO_PACKET       18
```

Undefined Behavior

- f5823fe6897c ("qed: Add ll2 option to limit the number of bds per packet")
- **`struct qed_ll2_tx_packet`** now contains a fake flex-array ([1] array).

```
struct qed_ll2_tx_queue {
        ...
-       struct qed_ll2_tx_packet *descq_array;
+       void *descq_mem; /* memory for variable sized qed_ll2_tx_packet*/
        struct qed_ll2_tx_packet *cur_send_packet;
        struct qed_ll2_tx_packet cur_completing_packet;
        ...
        u16 cur_completing_frag_num;
        bool b_completing_packet;
};
```

Undefined Behavior

- f5823fe6897c ("qed: Add ll2 option to limit the number of bds per packet")
- **`struct qed_ll2_tx_packet`** now contains a fake flex-array ([1] array).

```
struct qed_ll2_tx_queue {
        ...
-       struct qed_ll2_tx_packet *descq_array;
+       void *descq_mem; /* memory for variable sized qed_ll2_tx_packet*/
        struct qed_ll2_tx_packet *cur_send_packet;
        struct qed_ll2_tx_packet cur_completing_packet;
        ...
        u16 cur_completing_frag_num;
        bool b_completing_packet;
};
```

Undefined Behavior

- f5823fe6897c ("qed: Add ll2 option to limit the number of bds per packet")
- **`struct qed_ll2_tx_packet`** now contains a fake flex-array ([1] array).

```
struct qed_ll2_tx_queue {
        ...
-       struct qed_ll2_tx_packet *descq_array;
+       void *descq_mem; /* memory for variable sized qed_ll2_tx_packet*/
        struct qed_ll2_tx_packet *cur_send_packet;
        struct qed_ll2_tx_packet cur_completing_packet;    forgot to move this
        ...                                                to the end
        u16 cur_completing_frag_num;
        bool b_completing_packet;
};
```

Undefined Behavior

- a93b6a2b9f46 ("qed/qed_ll2: Replace one-element array with flexible … ")
- Bug introduced in 2017. Fixed in 2020.

```
struct qed_ll2_tx_packet {
        struct core_tx_bd *txq_bd;
        dma_addr_t tx_frag;
        u16 frag_len;
-   } bds_set[1];
+   } bds_set[];
};
```

```
struct qed_ll2_rx_queue {
    ...
-   struct qed_ll2_tx_packet cur_completing_packet;
    ...
    u16 cur_completing_frag_num;
    bool b_completing_packet;
    ...
+   struct qed_ll2_tx_packet cur_completing_packet;
};
```

The tale of `sizeof`() and the three trailing arrays. :)

The tale of `sizeof()` and the three trailing arrays. :)

- Of course, `sizeof()` returns different results.

```
sizeof(flex_struct->one_element_array) == size-of-element-type
sizeof(flex_struct->zero_length_array) == 0
```

The tale of `sizeof()` and the three trailing arrays. :)

- Of course, `sizeof()` returns different results.
- And that's another source of problems.
- Found multiple issues in the kernel.

```
sizeof(flex_struct->one_element_array) == size-of-element-type
sizeof(flex_struct->zero_length_array) == 0
sizeof(flex_struct->flex_array_member) == ?   /* error! */
```

```
error: invalid application of 'sizeof' to incomplete type
```

Trailing fixed-sized arrays of variable-length ;)

- BSD sockaddr (sys/socket.h)

```
/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
        unsigned char   sa_len;         /* total length */
        sa_family_t     sa_family;      /* address family */
        char            sa_data[14];    /* actually longer; address value */
};
#define SOCK_MAXADDRLEN    255         /* longest possible addresses */
```

Current mitigations lack sufficient coverage

Existing compiler features for array bounds checking

- `-Warray-bounds` (always almost ready)
  - Compile-time only: depends on compiler's internal determination of array sizes and *index variable value tracking*
- `-fsanitize=bounds` (`CONFIG_UBSAN_BOUNDS`)
  - Depends on compiler's internal determination of array sizes...
- `__builtin_object_size()` (`CONFIG_FORTIFY_SOURCE`)
  - Only for fixed-size known at compile-time, similar to sizeof()
- `__builtin_dynamic_object_size()` (`CONFIG_FORTIFY_SOURCE`)
  - Gains run-time size from hints like `__attribute__((__alloc_size__(...)))`

🤕 ***None*** of these work correctly for ***trailing*** arrays, since the compiler is forced to assume all trailing arrays are of an unknown size.

Compiler diagnostics blind to *all* trailing arrays

Mitigations from prior slide work for "`array`" as long as it isn't the last element in the structure:

```
struct foo {
    int something;
    int array[6];
    int more;
};
```

- Fixed size.
- Not trailing.
- Protected.

```
struct bar {
    int something;
    int array[6];
};
```

- Fixed size.
- *Trailing*.
- ***Not protected***.

```
struct baz {
    int something;
    int array[];
};
```

- *Unknown size*.
- *Trailing*.
- ***Not protected***.

GCC 13 and Clang 16: `-fstrict-flex-arrays=3`

- Makes the ambiguity of ***trailing*** fixed-sized arrays go away; they are their declared size:

```
struct foo {
    int something;
    int array[6];
    int more;
};
```

- Fixed size.
- Protected.

```
struct bar {
    int something;
    int array[6];
};
```

- Fixed size.
- Protected.

```
struct baz {
    int something;
    int array[];
};
```

- *Unknown size.*
- ***Not protected.***

Future compiler feature: `element_count` attribute

```
#define __counted_by(member)  \
        __attribute__((__element_count__(member)))
```

```
struct baz {
    int something;
    int count;
    int array[];
};
```

- *Unknown size.*
- ***Not protected.***

```
struct yay {
    int something;
    int count;
    int array[] __counted_by(count);
};
```

- Run-time sized by “count” member.
- Protected.

Improve coverage:
Refactor array sizes to be unambiguous

Flexible array transformation refactoring

- The general case.
- Flexible arrays in Unions (and helpers).
- The case of UAPI.

The general case - [1] to [ ]

Before

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

```
p = kmalloc(sizeof(*p) + sizeof(struct bar) * (p->count - 1), GFP_KERNEL);
copy some data into p->array through memcpy()
for (i = 0; i < p->count; i++) {
    do something with p->array[i] and live happily ever after :)
}
```

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[];
} *p;
```

After

```
p = kmalloc(sizeof(*p) + sizeof(struct bar) * p->count, GFP_KERNEL);
copy some data into p->array through memcpy()
for (i = 0; i < p->count; i++) {
    do something with p->array[i] and live happily ever after :)
}
```

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof`(*p)
- `sizeof`(p->array)

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`
- `sizeof(struct bar)`

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`
- `sizeof(struct bar)`

- Identify the **count** member and look for instances of **count - 1**.

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`
- `sizeof(struct bar)`

- Identify the **count** member and look for instances of **count - 1**.
- What if we don’t find **count - 1** but only **count**?

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`
- `sizeof(struct bar)`

- Identify the **count** member and look for instances of **count - 1**.
- What if we don't find **count - 1** but only **count**? Was that intentional? Is that a bug? :p

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`
- `sizeof(struct bar)`

- Identify the **count** member and look for instances of **count - 1**.
- What if we don't find **count - 1** but only **count**? Was that intentional? Is that a bug? :p
- What if the element type is `uint8_t` or `char` or any type of size 1 byte?

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`
- `sizeof(struct bar)`

- Identify the **count** member and look for instances of **count - 1**.
- What if we don't find **count - 1** but only **count**? Was that intentional? Is that a bug? :p
- What if the element type is `uint8_t` or `char` or any type of size 1 byte?
  - Look for instances of '**- 1**'.

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`
- `sizeof(struct bar)`

- Identify the **count** member and look for instances of **count - 1**.
- What if we don't find **count - 1** but only **count**? Was that intentional? Is that a bug? :p
- What if the element type is `uint8_t` or `char` or any type of size 1 byte?
  - Look for instances of '**- 1**'. (how fun!) D:

The general case - [1] to [ ]

```
struct foo {
        ...
        size_t count;
        struct bar array[1];
} *p;
```

Audit instances of

- `sizeof(*p)`
- `sizeof(p->array)`
- `sizeof(struct foo)`
- `sizeof(struct bar)`

- Identify the **count** member and look for instances of **count - 1**.
- What if we don't find **count - 1** but only **count**? Was that intentional? Is that a bug? :p
- What if the element type is `uint8_t` or `char` or any type of size 1 byte?
  - Look for instances of '**- 1**'. (how fun!) D:
- Lastly, what if there is any struct containing `struct foo` as a member?

The general case - [0] to [ ]

- Pretty much straightforward. Pay attention to any build warnings, though.
- Wait and see if we broke anything in user-space (UAPI).

```
struct foo {
        ...
        size_t count;
        struct bar array[0];
} *p;
```

```
struct foo {
        ...
        size_t count;
        struct bar array[];
} *p;
```

```
alloc_size = sizeof(*p) + sizeof(struct bar) * (p->count);
```

The general case - [0] to [ ]

- Most of the transformations from [0] to [] were done with the following Coccinelle script:

```
@@
identifier S, member, array;
type T1, T2;
@@
 struct S {
       ...
       T1 member;
       T2 array[
-                0
                 ];
 };
```

Flexible arrays in Unions

- Use **range** when nr_range == 1
- Use **ranges** when nr_range > 1

```
/*
 ...
 * @nr_range: number of ranges to be mapped
 * @range: range to be mapped when nr_range == 1
 * @ranges: array of ranges to be mapped when nr_range > 1
 */
struct dev_pagemap {
        ...
        int nr_range;
        union {
                struct range range;
                struct range ranges[0];
        };
};
```

Flexible arrays in Unions

- DECLARE_FLEX_ARRAY() for **flex-arrays in unions** (or alone in a struct).

```
/*
 ...
 * @nr_range: number of ranges to be mapped
 * @range: range to be mapped when nr_range == 1
 * @ranges: array of ranges to be mapped when nr_range > 1
 */
struct dev_pagemap {
        ...
        int nr_range;
        union {
                struct range range;
                DECLARE_FLEX_ARRAY(struct range, ranges);
        };
};
```

The case of UAPI ([1] to [ ] - first attempts)

- Duplicate the original struct within a union.
- Flexible-array for **kernel-space** and one-element array for **user-space**.
- 2d3e5caf96b9 (“net/ipv4: Replace one-element array with flexible-array member”)

```
 struct ip_msfilter {
-       __be32          imsf_multiaddr;
-       __be32          imsf_interface;
-       __u32           imsf_fmode;
-       __u32           imsf_numsrc;
-       __be32          imsf_slist[1];
+       union {
+               struct {
+                       __be32          imsf_multiaddr_aux;
+                       __be32          imsf_interface_aux;
+                       __u32           imsf_fmode_aux;
+                       __u32           imsf_numsrc_aux;
+                       __be32          imsf_slist[1];
+               };
+               struct {
+                       __be32          imsf_multiaddr;
+                       __be32          imsf_interface;
+                       __u32           imsf_fmode;
+                       __u32           imsf_numsrc;
+                       __be32          imsf_slist_flex[];
+               };
+       };
 };
```

The case of UAPI ([1] to [ ] - better code)

- __DECLARE_FLEX_ARRAY() for **flex-arrays in unions** (or alone in a struct).
- The bad news is that the sizeof(flex_struct) will remain the same.
- 5854a09b4957 (“net/ipv4: Use __DECLARE_FLEX_ARRAY() helper”)

```
struct ip_msfilter {
        __be32          imsf_multiaddr;
        __be32          imsf_interface;
        __u32           imsf_fmode;
        __u32           imsf_numsrc;
        union {
                __be32          imsf_slist[1];
                __DECLARE_FLEX_ARRAY(__be32, imsf_slist_flex);
        };
};
```

The case of (“breaking”) UAPI

- **Breaking** user-space (android-tools 33.0.3). :P
- https://github.com/nmeum/android-tools/issues/74

Build failure with Kernel >= 6.0: error: flexible array member ‘usbdevfs_urb::iso_frame_desc’ not at end of ‘struct usb_handle’

Closed **mdartmann** opened this issue on Sep 28, 2022 · 12 comments · Fixed by #85

mdartmann commented on Sep 28, 2022

On Kernel 6.0-rc6 with gcc 12.1, building android-tools 33.0.3 fails with the following error:

```
/usr/lib/ccache/bin/g++ -DADB_HOST=1 -D_FILE_OFFSET_BITS=64 -D_GNU_SOURCE -I/home/mae/.cache/kiss/proc/121205/b
In file included from /home/mae/.cache/kiss/proc/121205/build/android-tools/vendor/adb/client/usb_linux.cpp:28:
/usr/include/linux/usbdevice_fs.h:134:41: error: flexible array member ‘usbdevfs_urb::iso_frame_desc’ not at er
  134 |         struct usbdevfs_iso_packet_desc iso_frame_desc[];
      |                                         ^~~~~~~~~~~~~~
/home/mae/.cache/kiss/proc/121205/build/android-tools/vendor/adb/client/usb_linux.cpp:76:18: note: next member
   76 |     usbdevfs_urb urb_out;
      |                  ^~~~~~
/home/mae/.cache/kiss/proc/121205/build/android-tools/vendor/adb/client/usb_linux.cpp:61:8: note: in the defini
   61 | struct usb_handle {
      |        ^~~~~~~~~~
```

Assig
No o

Labe
None

Proje
None

Miles
No m

The case of (“breaking”) UAPI

- Breaking user-space. :P
- 94dfc73e7cf4 (“treewide: uapi: Replace zero-length arrays with flexible-array members”)
- **Kernel-space code.**

```
struct usbdevfs_urb {
    ...
    void __user *usercontext;
-   struct usbdevfs_iso_packet_desc iso_frame_desc[0];
+   struct usbdevfs_iso_packet_desc iso_frame_desc[];
};
```

The case of (“breaking”) UAPI

- Breaking user-space. :P
- struct usb_handle
- **User-space code.**

```
struct usb_handle {
      std::string path;
      ...

      usbdevfs_urb urb_in;
      usbdevfs_urb urb_out;

      ...
      pthread_t reaper_thread = 0;
};
```

The case of (“breaking”) UAPI

- The fix. :)
- 2247053 (“Update usage of usbdevfs_urb to match new kernel UAPI”)

GustavoARSilva commented on Oct 10, 2022

> I wonder if there is a way to patch `struct usb_handle {}` instead and make it compatible with both the old and the new struct definition?

One way to patch that user-space code is to turn these two interior members

```
usbdevfs_urb urb_in;
usbdevfs_urb urb_out;
```

into pointers:

```
usbdevfs_urb *urb_in;
usbdevfs_urb *urb_out;
```

and of course change the related code, accordingly.

The case of (“breaking”) UAPI

- The fix. :)
- 2247053 (“Update usage of usbdevfs_urb to match new kernel UAPI”)

```
struct usb_handle {
      std::string path;
      ...

-     usbdevfs_urb urb_in;
-     usbdevfs_urb urb_out;
+     usbdevfs_urb *urb_in;
+     usbdevfs_urb *urb_out;

      ...
      pthread_t reaper_thread = 0;
}
```

Improve coverage:
Annotate dynamic array sizes

Annotate allocators with `__alloc_size`

Usually Easy!

```
void *kmalloc(size_t bytes, gfp_t flags);
```

into

```
void *kmalloc(size_t bytes, gfp_t flags) __alloc_size(1);
```

Tricky bit is there are a lot of allocation wrappers or fixed-size helpers:

```
kcalloc, kvalloc, devm_kmalloc, kmem_cache_alloc, ...
```

And the attribute gets lost across inlines.

Using allocators with `__alloc_size`

Knowledge limited to the function calling the allocator:

```
struct foo *p = kzalloc(struct_size(p, array, count),
                        GFP_KERNEL);
...
for (int i = 0; i <= count; i++)
    do_something(p->array[i]); /* size information available! */
...
return p; /* size information lost */
```

Or internally to protected `memcpy()` implementation, we can check:

```
__builtin_dynamic_object_size(p, 1);
```

Annotate flexible array structures with `__counted_by`

Usually easy! Normally there is an obvious naming convention:

```
struct vexpress_syscfg_func {
    ...
    int num_templates;
    u32 template[];
};
```

Manually annotate flex-array structs with `__counted_by`

Usually easy! Normally there is an obvious naming convention:

```
struct vexpress_syscfg_func {
    ...
    int num_templates;
    u32 template[] __counted_by(num_templates);
};
```

Automatically annotate structs with `__counted_by`

Can use Coccinelle to find allocate/assign patterns.

Find the allocation:

```
@allocated@
identifier STRUCT, ARRAY, COUNTER, CALC, COUNT;
struct STRUCT *PTR;
identifier ALLOC =~ "[kv][cvzm]alloc";
@@
 PTR = ALLOC(..., struct_size(PTR, ARRAY, COUNT), ...);
 ...
 PTR->COUNTER = COUNT;
```

Automatically annotate structs with `__counted_by`

Can use Coccinelle to find allocate/assign patterns.

Add the annotation:

```
@annotate@
type COUNTER_TYPE, ARRAY_TYPE;
identifier allocated.STRUCT;
identifier allocated.ARRAY;
identifier allocated.COUNTER;
attribute name __counted_by;
@@
```

```
 struct STRUCT {
     ...
     COUNTER_TYPE COUNTER;
     ...
     ARRAY_TYPE ARRAY[]
+        __counted_by(COUNTER)
     ;
     ...
 };
```

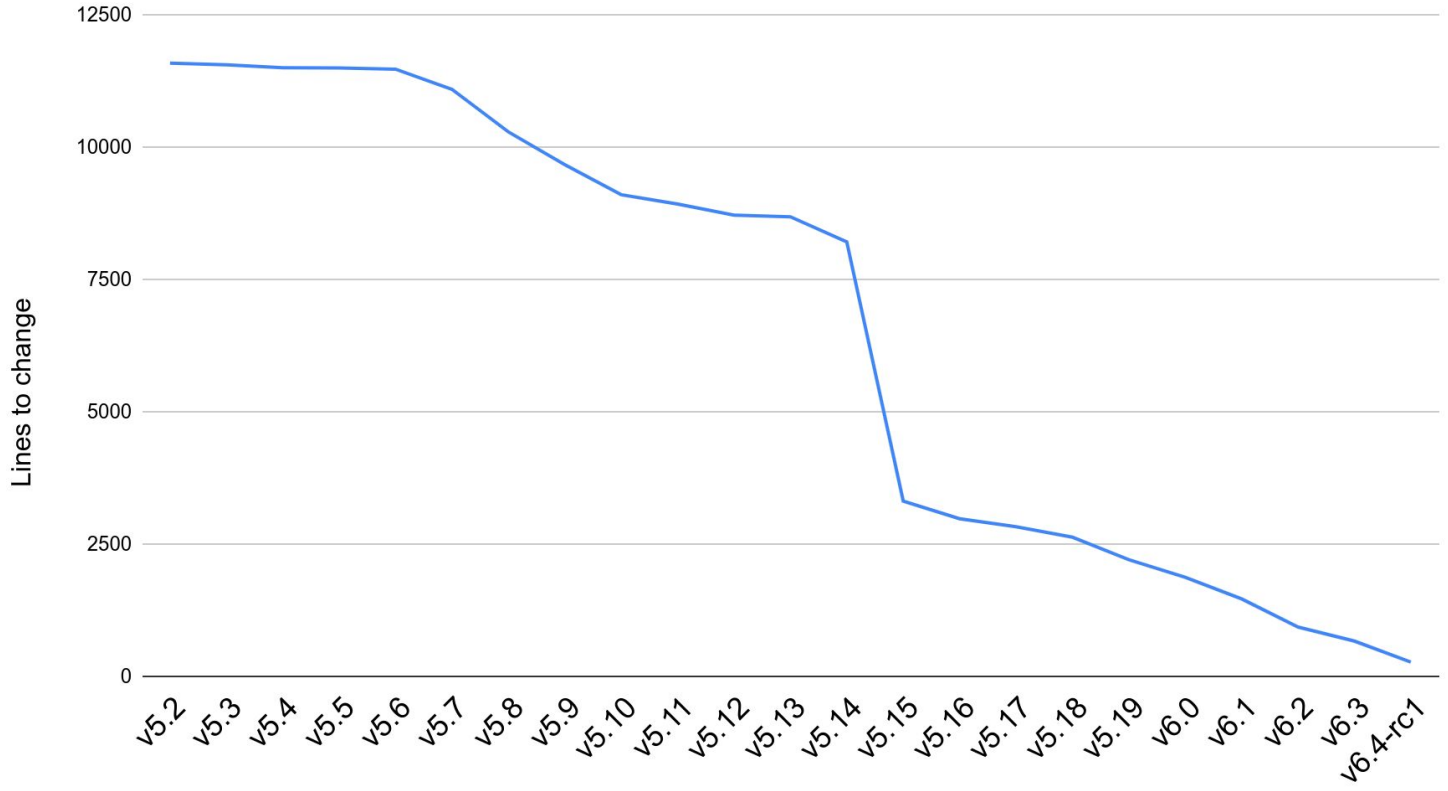
Compiler work

Ongoing work in GCC and Clang

- False positives with GCC's `-Warray-bounds`
  - jump threading: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109071
- Nested flexible array structure visibility to `__builtin_object_size()`
  - GCC: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101832
  - Clang: needed
- Solve `-fsanitize=bounds` vs `-fsanitize=object-size` (the latter has codegen issues)
  - `CONFIG_UBSAN_OBJECT_SIZE` was removed from Linux kernel
- Coordinate `__counted_by` attribute between compilers
  - Clang: https://reviews.llvm.org/D148381
  - GCC: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108896

Metrics!

Refactoring to use flexible array member (4 years)



Coverage for `memcpy()` bounds checking

For an x86_64 `defconfig` bound with `CONFIG_FORTIFY_SOURCE=y`, the counts of `memcpy()` mitigation:

- Linux v6.1: 46.6% coverage
  - no `__alloc_size`, no `-fstrict-flex-arrays=3`
  - 4165 total (fixed-size: 1940, unknown: 2225)
- Linux v6.3: 54.4% coverage
  - yes `__alloc_size`, no `-fstrict-flex-arrays=3`
  - 3969 total (fixed-size: 1833, dynamic: 325, unknown: 1807)
- Linux v6.4-rc1: 56.7% coverage
  - yes `__alloc_size`, *forced* `-fstrict-flex-arrays=3` with `KCFLAGS`
  - 3993 total (fixed-size: 1936, dynamic: 328, unknown: 1729)
- Future: add `__counted_by`, convert "unknown" to "dynamic"!

Questions/Thoughts?

Thanks for your attention!

Kees Cook <keescook@chromium.org>

https://fosstodon.org/@kees

Gustavo A. R. Silva <gustavoars@kernel.org>

https://fosstodon.org/@gustavoars

https://outflux.net/slides/2023/lss-na/bounds-checking.pdf