Enhancing spatial safety: Better array-bounds checking in C (and Linux)

Gustavo A. R. Silva
gustavoars@kernel.org
fosstodon.org/@gustavoars

Supported by
The Linux Foundation & Alpha-Omega

SSTIC
June 7, 2024
Rennes, France

Who am I?

Who am I?

- **Upstream first** – 8 years.
- Upstream Linux Kernel Engineer.
    - Focused on security.

Who am I?

- **Upstream first** – 8 years.
- Upstream Linux Kernel Engineer.
    - Focused on security.

- Kernel Self-Protection Project (**KSPP**).
- Google Open Source Security Team (**GOSST**).
    - Linux Kernel division.

Agenda

- **Introduction**
  - Fixed-size arrays & trailing arrays
  - Flex arrays, flex structures & flex-array transformations
- **Better array-bounds checking**
  - memcpy() hardening and -fstrict-flex-arrays
  - The new __counted_by__ attribute
  - __builtin_dynamic_object_size()
  - Bleeding-edge kernel hardening
- **Conclusions**

Fixed-size arrays

- Simple declaration of an array of fixed size.
- C doesn’t enforce array’s boundaries.
- It’s up to the developers to enforce them.
- Size determined at **compile-time**.

```
int fixed_size_array[10];
```

Trailing arrays

- Arrays declared at the end of a structure.
- Size determined at **compile-time**.

```
struct foo {
        ...
        some members;
        ...
        int trailing_array[10];
};
```

Flexible arrays & flexible structures

- Flexible array
  - **Trailing** array whose size is determined at **run-time**.
- Flexible structure
  - Structure that contains a **flexible array**.

```
struct flex_struct {
        ...
        size_t count;
        struct foo flex_array[];
};
```

Flexible arrays & flexible structures

- We use a flexible array when we know the size of the trailing array is going to be dynamic.

```
struct flex_struct {
        ...
        size_t count;
        struct foo flex_array[];
};
```

Flexible-Array Members

- FAMs were introduced in C99.

```
struct flex_struct {
        ...
        size_t count;
        struct foo flex_array[];
};
```

Flexible-Array Members

- FAMs were introduced in C99.
- A proper way to declare a flexible array in a struct.

```
struct flex_struct {
        ...
        size_t count;
        struct foo flex_array[];
};
```

Flexible-Array Members

- FAMs were introduced in C99.
- A proper way to declare a flexible array in a struct.
- Before C99 people would use [1] & [0]

```
struct flex_struct {
        ...
        size_t count;
        struct foo flex_array[];
};
```

Flexible-Array Members

- FAMs were introduced in C99.
- A proper way to declare a flexible array in a struct.
- Before C99 people would use [1] & [0]
- The flex struct usually contains a ***counter*** member.

```
struct flex_struct {
        ...
        size_t count;
        struct foo flex_array[];
};
```

Flexible arrays & flexible structures

Example:

```
struct flex_struct {
    ...
    size_t count;
    struct foo flex_array[];
} *p;

total_size = sizeof(*p) + sizeof(struct foo) * items;
p = kzalloc(total_size, GFP_KERNEL);
if (!p)
    return;

p->count = items;
```

Flexible arrays & flexible structures

Example:

```
struct flex_struct {
    ...
    size_t count;
    struct foo flex_array[];
} *p;

total_size = sizeof(*p) + sizeof(struct foo) * items;
p = kzalloc(struct_size(p, flex_array, items), GFP_KERNEL);
if (!p)
    return;

p->count = items;
```

Flexible-array transformations (FATs)

Kick-off of FATs in the Kernel Self-Protection Project

Flexible-array transformations (FATs)

Kick-off of FATs in the Kernel Self-Protection Project

```
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;
};
```

Flexible-array transformations (FATs)

Kick-off of FATs in the Kernel Self-Protection Project

- Undefined Behavior – **The bug**.
- e48f129c2f20 ("[SCSI] cxgb3i: convert cdev->l2opt to ...")

```
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;
};
```

Flexible-array transformations (FATs)

Kick-off of FATs in the Kernel Self-Protection Project

- Undefined Behavior – **The bugfix**.
- 76497732932f ("cxgb3/l2t: Fix undefined behavior")

```
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[];
};
```

Flexible-array transformations (FATs)

Kick-off of FATs in the Kernel Self-Protection Project

- Undefined Behavior – **The bugfix**.
- 76497732932f ("cxgb3/l2t: Fix undefined behavior")
- **8-year-old bug** introduced in **2011**, and 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[];
};
```

Better array-bounds checking

The BleedingTooth vulnerabilities

BadVibes

- **`last_adv_data`** is overflowed and list_head pointers corrupted.

```
#define HCI_MAX_AD_LENGTH      31

struct hci_dev {
   ...
   struct discovery_state {
        ...
        u8 last_adv_data[HCI_MAX_AD_LENGTH];
        ...
    } discovery;
    ...
    struct list_head {
        struct list_head *next;
        struct list_head *prev;
    } mgmt_pending;
    ...
};
```

```
static void store_pending_adv_report(..., u8
*data, u8 len)
{
    struct discovery_state *d = ...;
    ...
    memcpy(d->last_adv_data, data, len);
    ...
}
```

The BleedingTooth vulnerabilities

BadVibes

- **`last_adv_data`** is overflowed and list_head pointers corrupted.

```
#define HCI_MAX_AD_LENGTH      31

struct hci_dev {
   ...
   struct discovery_state {
        ...
        u8 last_adv_data[HCI_MAX_AD_LENGTH];
        ...
    } discovery;
    ...
    struct list_head {
        struct list_head *next;
        struct list_head *prev;
    } mgmt_pending;
    ...
};
```

```
static void store_pending_adv_report(..., u8
*data, u8 len)
{
    struct discovery_state *d = ...;
    ...
    memcpy(d->last_adv_data, data, len);
    ...
}
```

The BleedingTooth vulnerabilities

BadVibes

- **`last_adv_data`** is overflowed and list_head pointers corrupted.

```
#define HCI_MAX_AD_LENGTH      31

struct hci_dev {
   ...
   struct discovery_state {
        ...
        u8 last_adv_data[HCI_MAX_AD_LENGTH];
        ...
    } discovery;
    ...
    struct list_head {
        struct list_head *next;
        struct list_head *prev;
    } mgmt_pending;
    ...
};
```

- `len` is not sanity-checked before calling memcpy().

```
static void store_pending_adv_report(..., u8
*data, u8 len)
{
    struct discovery_state *d = ...;
    ...
    memcpy(d->last_adv_data, data, len);
    ...
}
```

The BleedingTooth vulnerabilities

BadVibes

- **`last_adv_data`** is overflowed and list_head pointers corrupted.

```
#define HCI_MAX_AD_LENGTH      31

struct hci_dev {
   ...
   struct discovery_state {
        ...
        u8 last_adv_data[HCI_MAX_AD_LENGTH];
        ...
    } discovery;
    ...
    struct list_head {
        struct list_head *next;
        struct list_head *prev;
    } mgmt_pending;
    ...
};
```

- `len` is not sanity-checked before calling memcpy().
- That's not great. :/

```
static void store_pending_adv_report(..., u8
*data, u8 len)
{
    struct discovery_state *d = ...;
    ...
    memcpy(d->last_adv_data, data, len);
    ...
}
```

The BleedingTooth vulnerabilities

BadVibes

- **`last_adv_data`** is overflowed and list_head pointers corrupted.

```
#define HCI_MAX_AD_LENGTH      31

struct hci_dev {
   ...
   struct discovery_state {
        ...
        u8 last_adv_data[HCI_MAX_AD_LENGTH];
        ...
    } discovery;
    ...
    struct list_head {
        struct list_head *next;
        struct list_head *prev;
    } mgmt_pending;
    ...
};
```

"The parser can theoretically receive and route a packet **up to 255 bytes** to this method. If that is possible, we could **overflow *last_adv_data* and corrupt members up to offset 0xbaf**."

```
static void store_pending_adv_report(..., u8
*data, u8 len)
{
    struct discovery_state *d = ...;
    ...
    memcpy(d->last_adv_data, data, len);
    ...
}
```

What’s the problem with **memcpy**()?

What’s the problem with memcpy()?

- memcpy() doesn’t know about your true intentions.
- You can overread and overwrite data without any restriction.
- It’s up to the developers to enforce boundaries for **src** and **dst** before calling memcpy().

```
void *memcpy(void *dst, const void *src, size_t size)
```

So, what was the fix for **BadVibes**?

What’s the problem with memcpy()?

- a2ec905d1e16 (“Bluetooth: fix kernel oops in store_...”)

```
 static void store_pending_adv_report(...)
 {
   struct discovery_state *d = ...;

+  if (len > HCI_MAX_AD_LENGTH)
+     return;
+
   ...
   memcpy(d->last_adv_data, data, len);
   ...
}
```

What’s the problem with memcpy()?

- a2ec905d1e16 (“Bluetooth: fix kernel oops in store_...”)

```
 static void store_pending_adv_report(...)
 {
   struct discovery_state *d = ...;

+  if (len > HCI_MAX_AD_LENGTH)
+     return;
+
   ...
   memcpy(d->last_adv_data, data, len);
   ...
 }
```

Hardening memcpy()

“Fortified” memcpy() (before BadVibes)

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 0);
        size_t src_size = __builtin_object_size(src, 0);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

Hardening memcpy()

“Fortified” memcpy() (before BadVibes)

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 0);
        size_t src_size = __builtin_object_size(src, 0);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

Hardening memcpy()

“Fortified” memcpy() (before BadVibes)

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 0);
        size_t src_size = __builtin_object_size(src, 0);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

Hardening memcpy()

“Fortified” memcpy() (before BadVibes)

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 0);
        size_t src_size = __builtin_object_size(src, 0);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

Hardening memcpy()

“Fortified” memcpy() (before BadVibes)

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 0);
        size_t src_size = __builtin_object_size(src, 0);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

Hardening memcpy()

“Fortified” memcpy() (before BadVibes)

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 0);
        size_t src_size = __builtin_object_size(src, 0);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

Hardening memcpy()

__builtin_object_size(OBJ, MODE)

- MODE 0: bytes to the end of the *entire struct*
- MODE 1: bytes to the end of *struct member*

```
struct foo {
        int count;      /* 4 bytes */
        char name[8];   /* 8 bytes */
        int secret;     /* 4 bytes */
        char blob[];    /* flexible array */
} *instance;            /* 16 bytes total */

__builtin_object_size(&instance->count, 0) == 16
__builtin_object_size(instance->name,   0) == 12
__builtin_object_size(instance->blob,   0) == -1

__builtin_object_size(&instance->count, 1) ==  4
__builtin_object_size(instance->name,   1) ==  8
__builtin_object_size(instance->blob,   1) == -1
```

Hardening memcpy()

__builtin_object_size(OBJ, MODE)

- MODE 0: bytes to the end of the *entire struct*
- MODE 1: bytes to the end of *struct member*

```
struct foo {
        int count;      /* 4 bytes */
        char name[8];   /* 8 bytes */
        int secret;     /* 4 bytes */
        char blob[];    /* flexible array */
} *instance;            /* 16 bytes total */

__builtin_object_size(&instance->count, 0) == 16
__builtin_object_size(instance->name,   0) == 12
__builtin_object_size(instance->blob,   0) == -1

__builtin_object_size(&instance->count, 1) ==  4
__builtin_object_size(instance->name,   1) ==  8
__builtin_object_size(instance->blob,   1) == -1
```

Hardening memcpy()

__builtin_object_size(OBJ, MODE)

- MODE 0: bytes to the end of the *entire struct*
- MODE 1: bytes to the end of *struct member*

```
struct foo {
        int count;      /* 4 bytes */
        char name[8];   /* 8 bytes */
        int secret;     /* 4 bytes */
        char blob[];    /* flexible array */
} *instance;            /* 16 bytes total */

__builtin_object_size(&instance->count, 0) == 16
__builtin_object_size(instance->name,   0) == 12
__builtin_object_size(instance->blob,   0) == -1

__builtin_object_size(&instance->count, 1) ==  4
__builtin_object_size(instance->name,   1) ==  8
__builtin_object_size(instance->blob,   1) == -1
```

Hardening memcpy()

__builtin_object_size(OBJ, MODE)

- MODE 0: bytes to the end of the *entire struct*
- MODE 1: bytes to the end of *struct member*

```
struct foo {
        int count;      /* 4 bytes */
        char name[8];   /* 8 bytes */
        int secret;     /* 4 bytes */
        char blob[];    /* flexible array */
} *instance;            /* 16 bytes total */

__builtin_object_size(&instance->count, 0) == 16
__builtin_object_size(instance->name,   0) == 12
__builtin_object_size(instance->blob,   0) == -1

__builtin_object_size(&instance->count, 1) ==  4
__builtin_object_size(instance->name,   1) ==  8
__builtin_object_size(instance->blob,   1) == -1
```

Hardening memcpy()

__builtin_object_size(d->**last_adv_data**, **0**)

- __bos() returns the number of bytes from **last_adv_data** to the end of **struct hci_dev**

```
struct hci_dev {
   ...
   struct discovery_state {
        ...
        u8 last_adv_data[HCI_MAX_AD_LENGTH];
        ...
    } discovery;
    ...
    struct list_head {
        struct list_head *next;
        struct list_head *prev;
    } mgmt_pending;
    ...
};
```

```
static void store_...(struct hci_dev *hdev, ...)
{
    struct discovery_state *d = &hdev->discovery;
    ...
    memcpy(d->last_adv_data, data, len);
    ...
}
```

Hardening memcpy()

__builtin_object_size(d->**last_adv_data**, **0**)

- __bos() returns the number of bytes from **last_adv_data** to the end of **struct hci_dev**

```
struct hci_dev {
    ...
    struct discovery_state {
        ...
        u8 last_adv_data[HCI_MAX_AD_LENGTH];
        ...
    } discovery;
    ...
    struct list_head {
        struct list_head *next;
        struct list_head *prev;
    } mgmt_pending;
    ...
};
```

```
static void store_...(struct hci_dev *hdev, ...)
{
    struct discovery_state *d = &hdev->discovery;
    ...
    memcpy(d->last_adv_data, data, len);
    ...
}
```

What do we do about it?

Hardening memcpy()

What do we do about it?

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 0);
        size_t src_size = __builtin_object_size(src, 0);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

Hardening memcpy()

What do we do about it?

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 0);
        size_t src_size = __builtin_object_size(src, 0);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

Hardening memcpy()

What do we do about it?

- Replace __bos(0) with __bos(1)

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

This is enough to prevent **BadVibes-like vulnerabilities**. :)

OK, but…

OK, but…

what about **intentional cross-member overflows**?

Intentional cross-member overflows

- Linux had many intentional cross-member **memcpy**() overflows

```
struct mwl8k_cmd_set_key {
        ...

                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];

        ...
};

keymlen = MAX_ENCR_KEY_LENGTH + 2 * MIC_KEY_LENGTH;
...
memcpy(cmd->key_material, key->key, keymlen);
```

Intentional cross-member overflows

Linux had many intentional cross-member **memcpy**() overflows

```
struct mwl8k_cmd_set_key {
        ...

                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];

        ...
};

keymlen = MAX_ENCR_KEY_LENGTH + 2 * MIC_KEY_LENGTH;
...
memcpy(cmd->key_material, key->key, keymlen);
```

Intentional cross-member overflows

- Linux had many intentional cross-member **memcpy**() overflows

```
struct mwl8k_cmd_set_key {
        ...

                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];

        ...
};

keymlen = MAX_ENCR_KEY_LENGTH + 2 * MIC_KEY_LENGTH;
...
memcpy(cmd->key_material, key->key, keymlen);
```

Intentional cross-member overflows

- Linux had many intentional cross-member **memcpy**() overflows

```
struct mwl8k_cmd_set_key {
        ...

                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];

        ...
};

keymlen = MAX_ENCR_KEY_LENGTH + 2 * MIC_KEY_LENGTH;
...
memcpy(cmd->key_material, key->key, keymlen);
```

Intentional cross-member overflows

Linux had many intentional cross-member **memcpy**() overflows

```
struct mwl8k_cmd_set_key {
        ...

                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];

        ...
};

keymlen = MAX_ENCR_KEY_LENGTH + 2 * MIC_KEY_LENGTH;
...
memcpy(cmd->key_material, key->key, keymlen);
```

Intentional cross-member overflows

Could be fixed by simply adding a named sub-struct

```
struct mwl8k_cmd_set_key {
        ...
        struct {
                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];
        } tkip;
        ...
};

keymlen = MAX_ENCR_KEY_LENGTH + 2 * MIC_KEY_LENGTH;
...
memcpy(cmd->tkip,          key->key, keymlen);
```

Intentional cross-member overflows

Could be fixed by simply adding a named sub-struct

```
struct mwl8k_cmd_set_key {
        ...
        struct {
                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];
        } tkip;
        ...
};

keymlen = MAX_ENCR_KEY_LENGTH + 2 * MIC_KEY_LENGTH;
...
memcpy(cmd->tkip,          key->key, keymlen);
```

Intentional cross-member overflows

But now everything must include the name of the sub-struct

```
struct mwl8k_cmd_set_key {
        ...
        struct {
                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];
        } tkip;
        ...
};

diff ...
-    do_something_with(cmd->key_material);
+    do_something_with(cmd->tkip.key_material);
```

Intentional cross-member overflows

struct_group() was invented to provide both

```
struct mwl8k_cmd_set_key {
        ...
        struct_group(tkip,
                __u8 key_material[MAX_ENCR_KEY_LENGTH];
                __u8 tkip_tx_mic_key[MIC_KEY_LENGTH];
                __u8 tkip_rx_mic_key[MIC_KEY_LENGTH];
        };
        ...
};

/* Accessible either way: */
do_something_with(cmd->key_material);
do_something_with(cmd->tkip.key_material);
```

The **struct_group**() helper macro

Created by Kees Cook and Keith Packard

```
#define struct_group(NAME, MEMBERS...)    \
        union {                           \
                struct { MEMBERS };       \
                struct { MEMBERS } NAME; \
        }
```

The **struct_group**() helper macro

Created by Kees Cook and Keith Packard

- struct_group_tagged(), struct_group_attr() & __struct_group()

```
#define struct_group(NAME, MEMBERS...)    \
        union {                           \
                struct { MEMBERS };       \
                struct { MEMBERS } NAME; \
        }
```

The **struct_group**() helper macro

Created by Kees Cook and Keith Packard

- struct_group_tagged(), struct_group_attr() & __struct_group()
- Access each member directly or through the named struct.

```
#define struct_group(NAME, MEMBERS...)    \
        union {                           \
                struct { MEMBERS };       \
                struct { MEMBERS } NAME; \
        }
```

The **struct_group**() helper macro

Created by Kees Cook and Keith Packard

- struct_group_tagged(), struct_group_attr() & __struct_group()
- Access each member directly or through the named struct.
- Gain bounds-checking on the group as a whole.

```
#define struct_group(NAME, MEMBERS...)    \
        union {                           \
                struct { MEMBERS };       \
                struct { MEMBERS } NAME;  \
        }
```

With **struct_group**() we avoid false positives, gain bounds-checking and can use __**bos**(1)

memcpy() & -fstrict-flex-arrays

- Now we can use __bos(1).

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        ...
}
```

memcpy() & -fstrict-flex-arrays

- Now we can use __bos(1). Life's beautiful. **^.^**

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        ...
}
```

Well...

__bos() & flexible arrays

```
struct foo {
        int count;      /* 4 bytes */
        char name[8];   /* 8 bytes */
        int secret;     /* 4 bytes */
        char blob[];    /* flexible array */
} *instance;            /* 16 bytes total */

__builtin_object_size(&instance->count, 0) == 16
__builtin_object_size(instance->name,   0) == 12
__builtin_object_size(instance->blob,   0) == -1

__builtin_object_size(&instance->count, 1) ==  4
__builtin_object_size(instance->name,   1) ==  8
__builtin_object_size(instance->blob,   1) == -1
```

memcpy() & flexible arrays

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {      /* Compile-time */
                if (dst_size < size)
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        ...
        return __underlying_memcpy(dst, src, size);
}
```

```
struct flex_struct {
    ...
    size_t count;
    struct foo flex_array[];
} *p;
...
memcpy(p->flex_array, &source, SOME_SIZE);
```

memcpy() & flexible arrays

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);  == -1 /* __bos() returns -1 */
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {      /* Compile-time */
                if (dst_size < size)
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        ...
        return __underlying_memcpy(dst, src, size);
}
```

```
struct flex_struct {
    ...
    size_t count;
    struct foo flex_array[];
} *p;
...
memcpy(p->flex_array, &source, SOME_SIZE);
```

memcpy() & flexible arrays

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);  == -1 /* __bos() returns -1 */
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {      /* Compile-time */
                if (dst_size < size)
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        ...
        return __underlying_memcpy(dst, src, size);
}
```

```
struct flex_struct {
    ...
    size_t count;
    struct foo flex_array[];
} *p;
...
memcpy(p->flex_array, &source, SOME_SIZE);
```

- FAMs are objects of incomplete type.

memcpy() & flexible arrays

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);  == -1 /* __bos() returns -1 */
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {      /* Compile-time */
                if (dst_size < size)  /* in this case, the condition is always false */
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        ...
        return __underlying_memcpy(dst, src, size);
}
```

```
struct flex_struct {
    ...
    size_t count;
    struct foo flex_array[];
} *p;
...
memcpy(p->flex_array, &source, SOME_SIZE);
```

- FAMs are objects of incomplete type.

memcpy() & flexible arrays

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);  == -1 /* __bos() returns -1 */
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {      /* Compile-time */
                if (dst_size < size)  /* in this case, the condition is always false */
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        ...
        return __underlying_memcpy(dst, src, size);
}
```

```
struct flex_struct {
    ...
    size_t count;
    struct foo flex_array[];
} *p;
...
memcpy(p->flex_array, &source, SOME_SIZE);
```

- FAMs are objects of incomplete type.
- Bounds-checking is not possible in this case.

memcpy() & flexible arrays

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);  == -1 /* __bos() returns -1 */
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {      /* Compile-time */
                if (dst_size < size)  /* in this case, the condition is always false */
                     __write_overflow();
                if (src_size < size)
                     __read_overflow2();
        }
        ...
        return __underlying_memcpy(dst, src, size);
}
```

```
struct flex_struct {
    ...
    size_t count;
    struct foo flex_array[];
} *p;
...
memcpy(p->flex_array, &source, SOME_SIZE);
```

- FAMs are objects of incomplete type.
- Bounds-checking is not possible in this case.

memcpy() & flexible arrays

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_object_size(dst, 1);  == -1 /* __bos() returns -1 */
        size_t src_size = __builtin_object_size(src, 1);

        if (__builtin_constant_p(size)) {      /* Compile-time */
                if (dst_size < size)  /* in this case, the condition is always false */
                     __write_overflow();
                if (src_size < size)
                     __read_overflow2();
        }
        ...
        return __underlying_memcpy(dst, src, size);
}
```

```
struct flex_struct {
     ...
     size_t count;
     struct foo flex_array[];
} *p;
...
memcpy(p->flex_array, &source, SOME_SIZE);
```

- FAMs are objects of incomplete type.
- Bounds-checking is not possible in this case.
- All this is expected behavior.

__builtin_object_size() & trailing arrays

- However, __**bos(1)** was returning -1 for any trailing array.

```
__builtin_object_size(any_struct->any_trailing_array, 1) == -1
```

__builtin_object_size() & trailing arrays

- However, __**bos(1)** was returning -1 for any trailing array.

```
struct foo {
        ...
        some members;
        ...

        int trailing_array[10];
} *p;
```

__builtin_object_size() & trailing arrays

- However, __**bos(1)** was returning -1 for any trailing array.

```
struct foo {
        ...
        some members;
        ...

        int trailing_array[10];
} *p;
```

```
__builtin_object_size(p->trailing_array, 1) == -1
```

__builtin_object_size() & trailing arrays

- However, __**bos(1)** was returning -1 for any trailing array.

```
__builtin_object_size(any_struct->any_trailing_array, 1) == -1
```

__builtin_object_size() & trailing arrays

- However, __**bos(1)** was returning -1 for any trailing array.

```
__builtin_object_size(any_struct->any_trailing_array, 1) == -1
```

Under this scenario *memcpy*() is not able to sanity-check trailing arrays of any size at all.

__builtin_object_size() & trailing arrays

- However, __**bos(1)** was returning -1 for any trailing array.

```
__builtin_object_size(any_struct->any_trailing_array, 1) == -1
```

Under this scenario *memcpy*() is not able to sanity-check trailing arrays of any size at all.

But why, exactly?

__builtin_object_size() & trailing arrays

- BSD **sockaddr** (sys/socket.h)

```
struct sockaddr {
   unsigned char   sa_len;         /* total length */
   sa_family_t     sa_family;      /* address family */
   char            sa_data[14]; /* actually longer; */
};

/* longest possible addresses */
#define SOCK_MAXADDRLEN   255
```

__builtin_object_size() & trailing arrays

- BSD **sockaddr** (sys/socket.h)

```
struct sockaddr {
    unsigned char   sa_len;         /* total length */
    sa_family_t     sa_family;      /* address family */
    char            sa_data[14];    /* actually longer; */
};

/* longest possible addresses */
#define SOCK_MAXADDRLEN   255
```

__builtin_object_size() & trailing arrays

- BSD **sockaddr** (sys/socket.h)

```
struct sockaddr {
    unsigned char   sa_len;         /* total length */
    sa_family_t     sa_family;      /* address family */
    char            sa_data[14]; /* actually longer; */
};

/* longest possible addresses */
#define SOCK_MAXADDRLEN   255
```

“A feature, not a bug”

- https://reviews.llvm.org/D126864

“**Some code consider that trailing arrays are flexible, whatever their size**. Support for these **legacy** code has been introduced in f8f632498307d22e10fab0704548b270b15f1e1e but **it prevents evaluation of builtin_object_size and builtin_dynamic_object_size in some legit cases**.”

“A feature, not a bug”

- https://reviews.llvm.org/D126864

“Some code consider that trailing arrays are flexible, whatever their size. Support for these legacy code has been introduced in f8f632498307d22e10fab0704548b270b15f1e1e but it prevents evaluation of builtin_object_size and builtin_dynamic_object_size in some legit cases.”

“A feature, not a bug”

- https://reviews.llvm.org/D126864

“Some code consider that trailing arrays are flexible, whatever their size. Support for these legacy code has been introduced in f8f632498307d22e10fab0704548b270b15f1e1e but **it prevents evaluation of builtin_object_size and builtin_dynamic_object_size in some legit cases.**”

“A feature, not a bug”

- https://reviews.llvm.org/D126864

“**Some code consider that trailing arrays are flexible, whatever their size**. Support for these **legacy** code has been introduced in f8f632498307d22e10fab0704548b270b15f1e1e but **it prevents evaluation of builtin_object_size and builtin_dynamic_object_size in some legit cases**.”

__builtin_object_size() & trailing arrays

So, what do we do about it?

memcpy() & -fstrict-flex-arrays

- **Compiler side**: Fix it and make it enforce FAMs.

- **Kernel side**: Make flex-array declarations **unambiguous**.

memcpy() & -fstrict-flex-arrays

- **Compiler side**: Fix it and make it enforce FAMs.
  - Fix **__builtin_object_size()**
  - Add new option **-fstrict-flex-arrays**[=n]
  - Enforcing FAMs as the only way to declare flex arrays.

- **Kernel side**: Make flex-array declarations **unambiguous**.

memcpy() & -fstrict-flex-arrays

- **Compiler side**: Fix it and make it enforce FAMs.
  - Fix **__builtin_object_size()**
  - Add new option **-fstrict-flex-arrays**[=n]
  - Enforcing FAMs as the only way to declare flex arrays.

- **Kernel side**: Make flex-array declarations **unambiguous**.
  - Get rid of **fake** flexible arrays ([1] & [0]).
  - Only C99 **flexible-array members** should be used as flexible arrays.

memcpy() & -fstrict-flex-arrays=3

-fstrict-flex-arrays[=n]

memcpy() & -fstrict-flex-arrays=3

-fstrict-flex-arrays[=n] – Released in **GCC-13** and **Clang-16.**

memcpy() & -fstrict-flex-arrays=3

- **-fstrict-flex-arrays[=n]** – Released in **GCC-13** and **Clang-16.**
  - -fstrict-flex-arrays**=3**
    - Only C99 flexible-array members (**[ ]**) are treated VLOs.

```
__builtin_object_size(flex_struct->flex_array_member, 1) == -1
```

memcpy() & -fstrict-flex-arrays=3

-fstrict-flex-arrays[=n] – Released in **GCC-13** and **Clang-16.**

- -fstrict-flex-arrays**=3**
  - Only C99 flexible-array members (**[ ]**) are treated VLOs.

```
__builtin_object_size(flex_struct->flex_array_member, 1) == -1

__bos(any_struct->any_non_flex_array, 1) == sizeof(any_non_flex_array)
```

With this **ALL trailing arrays of fixed-size gain** bounds-checking.

memcpy() & -fstrict-flex-arrays=3

-fstrict-flex-arrays[=n] – Released in **GCC-13** and **Clang-16.**

- -fstrict-flex-arrays**=3**
  - Only C99 flexible-array members (**[ ]**) are treated VLOs.

```
__builtin_object_size(flex_struct->flex_array_member, 1) == -1

__bos(any_struct->any_non_flex_array, 1) == sizeof(any_non_flex_array)
__bos(any_struct->one_element_array,  1) == sizeof(one_element_array)
__bos(any_struct->zero_length_array,  1) == sizeof(zero_length_array) == 0
```

With this **ALL trailing arrays of fixed-size gain** bounds-checking. Including [1] & [0], of course.

memcpy() & -fstrict-flex-arrays=3

Fortified **memcpy()** and **-fstrict-flex-arrays=3**

- Globally enabled in **Linux 6.5**. Yeeiii!!

memcpy() & -fstrict-flex-arrays=3

Fortified **memcpy()** and **-fstrict-flex-arrays=3**

- Globally enabled in **Linux 6.5**. Yeeiii!!
- Only C99 flexible-array members are considered to be dynamically sized.
- **The trailing array ambiguity is gone.**

memcpy() & -fstrict-flex-arrays=3

Fortified **memcpy()** and **-fstrict-flex-arrays=3**

- Globally enabled in **Linux 6.5**. Yeeiii!!
- Only C99 flexible-array members are considered to be dynamically sized.
- **The trailing array ambiguity is gone.**

Therefore, we've gained bounds-checking on **trailing arrays of fixed-size.**

Great, but what about bounds-checking on **flexible-array members**?

The new __counted_by__ attribute

The new __counted_by__ attribute

- __attribute__((__**counted_by**__(member)))
- Coming soon in **GCC-15** (bugzilla id=108896) (Qing Zhao)

```
struct bounded_flex_struct {
   ...
   size_t count;
   struct foo array[] __attribute__((__counted_by__(count)));
};
```

The new __counted_by__ attribute

- __attribute__((__**counted_by**__(member)))
- Coming soon in **GCC-15** (bugzilla id=108896) (Qing Zhao)
- Just released in **Clang-18 !!!** (LLVM id=76348) (Bill Wendling)

```
struct bounded_flex_struct {
    ...
    size_t count;
    struct foo array[] __attribute__((__counted_by__(count)));
};
```

The new __counted_by__ attribute

- __attribute__((__**counted_by**__(member)))
- Coming soon in **GCC-15** (bugzilla id=108896) (Qing Zhao)
- Just released in **Clang-18 !!!** (LLVM id=76348) (Bill Wendling)

“Clang now supports the C-only attribute counted_by. When applied to a struct’s flexible array member, **it points to the struct field that holds the number of elements in the flexible array member**. This information **can improve the results of the array bound sanitizer and the __builtin_dynamic_object_size builtin**.”

https://releases.llvm.org/18.1.0/tools/clang/docs/ReleaseNotes.html

The new __counted_by__ attribute

- __attribute__((__**counted_by**__(member)))
- Coming soon in **GCC-15** (bugzilla id=108896) (Qing Zhao)
- Just released in **Clang-18 !!!** (LLVM id=76348) (Bill Wendling)

“Clang now supports the C-only attribute counted_by. When applied to a struct’s flexible array member, **it points to the struct field that holds the number of elements in the flexible array member**. This information can improve the results of the array bound sanitizer and the __builtin_dynamic_object_size builtin.”

https://releases.llvm.org/18.1.0/tools/clang/docs/ReleaseNotes.html

The new __counted_by__ attribute

- __attribute__((__**counted_by**__(member)))
- Coming soon in **GCC-15** (bugzilla id=108896) (Qing Zhao)
- Just released in **Clang-18 !!!** (LLVM id=76348) (Bill Wendling)

“Clang now supports the C-only attribute counted_by. When applied to a struct’s flexible array member, it points to the struct field that holds the number of elements in the flexible array member. This information **can improve the results of the array bound sanitizer and the __builtin_dynamic_object_size builtin.**”

https://releases.llvm.org/18.1.0/tools/clang/docs/ReleaseNotes.html

The new __counted_by__ attribute

- __attribute__((__**counted_by**__(member)))
- Coming soon in **GCC-15** (bugzilla id=108896) (Qing Zhao)
- Just released in **Clang-18 !!!** (LLVM id=76348) (Bill Wendling)

“Clang now supports the C-only attribute counted_by. When applied to a struct’s flexible array member, **it points to the struct field that holds the number of elements in the flexible array member**. This information **can improve the results of the array bound sanitizer and the __builtin_dynamic_object_size builtin**.”

https://releases.llvm.org/18.1.0/tools/clang/docs/ReleaseNotes.html

The new __counted_by__ attribute

- __attribute__((__**counted_by**__(member)))
- Coming soon in **GCC-15** (bugzilla id=108896) (Qing Zhao)
- Just released in **Clang-18 !!!** (LLVM id=76348) (Bill Wendling)

```
#if __has_attribute(__counted_by__)
# define __counted_by(member) __attribute__((__counted_by__(member)))
#else
# define __counted_by(member)
#endif
```

The new __counted_by__ attribute

- __attribute__((__**counted_by**__(member)))
- Coming soon in **GCC-15** (bugzilla id=108896) (Qing Zhao)
- Just released in **Clang-18 !!!** (LLVM id=76348) (Bill Wendling)

```
struct bounded_flex_struct {
   ...
   size_t count;
   struct foo array[] __counted_by(count);
};
```

__builtin_dynamic_object_size()

Fortified **memcpy()** and **__builtin_dynamic_object_size**()

__builtin_dynamic_object_size()

Fortified **memcpy()** and **__builtin_dynamic_object_size**()

- __bdos() **replaced** __builtin_object_size()
- __bdos() adds **run-time coverage.**
- It **gets hints from** __alloc_size__ and from **__counted_by__**
- Greater fortification for **memcpy**().
- **CONFIG_FORTIFY_SOURCE=y** benefits from all this.

__builtin_dynamic_object_size()

Fortified **memcpy()** and **__builtin_dynamic_object_size**()

- __bdos() **replaced** __builtin_object_size()
- __bdos() adds **run-time coverage.**
- It **gets hints from** __alloc_size__ and from **__counted_by__**
- Greater fortification for **memcpy**().
- **CONFIG_FORTIFY_SOURCE=y** benefits from all this.

With __counted_by__ & __bdos(1), we gain bounds-checking on **flex-arrays.**

__builtin_dynamic_object_size()

Fortified **memcpy()** and **__builtin_dynamic_object_size()**

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_dynamic_object_size(dst, 1);
        size_t src_size = __builtin_dynamic_object_size(src, 1);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                    __write_overflow();
                if (src_size < size)
                    __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

__builtin_dynamic_object_size()

Fortified **memcpy()** and **__builtin_dynamic_object_size()**

```
__FORTIFY_INLINE void *memcpy(void *dst, const void *src, size_t size)
{
        size_t dst_size = __builtin_dynamic_object_size(dst, 1);
        size_t src_size = __builtin_dynamic_object_size(src, 1);

        if (__builtin_constant_p(size)) {     /* Compile-time */
                if (dst_size < size)
                        __write_overflow();
                if (src_size < size)
                        __read_overflow2();
        }
        if (dst_size < size || src_size < size) /* Run-time */
                fortify_panic(__func__);
         return __underlying_memcpy(dst, src, size);
}
```

OK, we’re done with memcpy(), __counted_by__ & __bdos(1)

:)

Bleeding-edge upstream kernel hardening

- **-Wflex-array-member-not-at-end** (GCC-14)

Bleeding-edge upstream kernel hardening

-Wflex-array-member-not-at-end (GCC-14)

```
struct flex_struct {
    ...
    size_t count;
    struct something flex_array[] __counted_by(count);
};

struct composite_struct {
    ...

    struct flex_struct flex_in_the_middle; /* suspicious ¬.¬ */

    ...
};
```

Bleeding-edge upstream kernel hardening

-Wflex-array-member-not-at-end (GCC-14)

- We had **~60,000 warnings** in total.

```
struct flex_struct {
    ...
    size_t count;
    struct something flex_array[] __counted_by(count);
};

struct composite_struct {
    ...

    struct flex_struct flex_in_the_middle; /* suspicious ¬.¬ */

    ...
};
```

Bleeding-edge upstream kernel hardening

-Wflex-array-member-not-at-end (GCC-14)

- We had **~60,000 warnings** in total. Only **650** unique.

```
struct flex_struct {
    ...
    size_t count;
    struct something flex_array[] __counted_by(count);
};

struct composite_struct {
    ...

    struct flex_struct flex_in_the_middle; /* suspicious ¬.¬ */

    ...
};
```

-Wflex-array-member-not-at-end

Four different categories of False Positives

- C1: Some **FAMs not used at all.**
  - commit f4b09b29f8b4
- C2: **FAMs never accessed**.
  - commit 5c4250092fad
- C3: **Implicit unions** between FAMs and fixed-size arrays.
  - commit 38aa3f5ac6d2
- C4: The same as case 3 but **on-stack**.
  - commit 34c34c242a1b

-Wflex-array-member-not-at-end

Patches landed in mainline already

- -Wflex-array-member-not-at-end patches in mainline.
- From 650 to **less than 400 warnings now!** :D
- **~30%** of warnings have been addressed in the last couple of months.

Conclusions

Conclusions

Problem:

- **Unintentional** cross-member overflows.
- BadVibes-like bugs.

Conclusions

Problem:

- **Unintentional** cross-member overflows.
- BadVibes-like bugs.

Solution:

- Use __builtin_dynamic_object_size(1)

Conclusions

Problem:

- **Intentional** cross-member overflows.

Conclusions

Problem:

- **Intentional** cross-member overflows.

Solution:

- Use the **struct_group**() family of helpers.

Conclusions

Problem:

- Trailing array **ambiguity**.
- Lack of bounds-checking on trailing **arrays of fixed-size.**

Conclusions

Problem:

- Trailing array **ambiguity**.
- Lack of bounds-checking on trailing **arrays of fixed-size.**

Solution:

- Flexible-array **transformations.**
- **-fstrict-flex-arrays=3**

Conclusions

Problem:

- Lack of bounds-checking on **flexible arrays.**

Conclusions

Problem:

- Lack of bounds-checking on **flexible arrays.**

Solution:

- Annotate FAMs with __**counted_by**__
- __builtin_dynamic_object_size(1)

Conclusions

Problem:

- Flexible arrays **in the middle.**

Conclusions

Problem:

- Flexible arrays **in the middle.**

Solution:

- Enable **-Wflex-array-member-not-at-end**
- Clean up your codebase.
- (Take a look at the commit IDs ;-))

Conclusions

- Clear strategy to enable **-Wflex-array-member-not-at-end** in mainline, soon.
- Build your kernel with **CONFIG_FORTIFY_SOURCE** & **CONFIG_UBSAN_BOUNDS (-fsanitize=bounds)**.
- **The security of the kernel is being significantly improved. :)**

Merci beaucoup, Rennes!

Gustavo A. R. Silva
gustavoars@kernel.org
fosstodon.org/@gustavoars