Toolchain security features status update

Kees Cook <keescook@chromium.org>
Qing Zhao <qing.zhao@oracle.com>
Bill Wendling <morbo@google.com>

https://outflux.net/slides/2023/lpc/features.pdf
## 2022 Security Features Review

<table>
<thead>
<tr>
<th>Feature</th>
<th>GCC</th>
<th>Clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero call-used registers</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Structure layout randomization</td>
<td>plugin</td>
<td>yes</td>
</tr>
<tr>
<td>Stack protector guard location</td>
<td>arm64 arm32 riscv ppc</td>
<td>arm64 arm32 riscv ppc</td>
</tr>
<tr>
<td>Forward edge CFI</td>
<td>CPU inline hash</td>
<td>CPU inline hash</td>
</tr>
<tr>
<td>Backward edge CFI</td>
<td>CPU</td>
<td>CPU SCS:arm64</td>
</tr>
<tr>
<td><code>-fstrict-flex-arrays</code></td>
<td>in progress</td>
<td>workable</td>
</tr>
<tr>
<td>Counted by attribute</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Integer overflow protection</td>
<td>broken</td>
<td>broken</td>
</tr>
</tbody>
</table>
### 2023: security features review

<table>
<thead>
<tr>
<th>Feature</th>
<th>GCC</th>
<th>Clang</th>
<th>RustC</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero call-used registers</td>
<td>yes</td>
<td>yes</td>
<td>needed</td>
</tr>
<tr>
<td>structure layout randomization</td>
<td>plugin</td>
<td>yes</td>
<td>needed</td>
</tr>
<tr>
<td>stack protector guard location</td>
<td>arm64, arm32, riscv, ppc</td>
<td>arm64, arm32, riscv, ppc</td>
<td>N/A</td>
</tr>
<tr>
<td>forward edge CFI</td>
<td>CPU inline hash</td>
<td>CPU inline hash</td>
<td>in progress</td>
</tr>
<tr>
<td>backward edge CFI</td>
<td>CPU SCS:arm64</td>
<td>CPU SCS:arm64</td>
<td>SCS:arm64</td>
</tr>
<tr>
<td><code>-fstrict-flex-arrays</code></td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>counted_by attribute</td>
<td>in progress</td>
<td>in progress</td>
<td>???</td>
</tr>
<tr>
<td>integer overflow protection</td>
<td>broken</td>
<td>broken</td>
<td>exists</td>
</tr>
</tbody>
</table>
New compiler to consider: RustC

- With Rust in the Linux kernel, we need to keep RustC at parity with Clang and GCC so we avoid cross-language attacks.

- Areas where Rust hardening needs attention:
  - zero call-used regs needs to happen in Rust code too
  - randstruct needs to work with Rust or structs aren’t ordered correctly
  - kCFI is in progress (slide 58)
  - counted_by attribute needs to be investigated
  - arithmetic overflow handling exists, but how to wire up traps consistently vs UBSan?
Parity reached: -fstrict-flex-arrays=3

- -fstrict-flex-arrays=3
  - Implemented in GCC 13+.
  - Implemented in Clang 16+.
- Includes logic changes for -fsanitize=bounds and __builtin_dynamic_object_size()
- Linux kernel enabled it globally in v6.5.
## Work needed: stack protector guard location (no progress)

<table>
<thead>
<tr>
<th>Arch</th>
<th>Linux Kernel Options</th>
<th>GCC</th>
<th>Clang</th>
</tr>
</thead>
</table>
| x86_64 & ia32 | -mstack-protector-guard-reg=fs
              -mstack-protector-guard-symbol=__stack_chk_guard | yes (8.1+) | yes (16+)  |
| arm64       | -mstack-protector-guard=sysreg
              -mstack-protector-guard-reg=sp_el0
              -mstack-protector-guard-offset=...TSK_STACK_CANARY... | yes (9.1+) | yes (14+)  |
| arm32       | -mstack-protector-guard=tls
              -mstack-protector-guard-offset=...TSK_STACK_CANARY... | yes (13.1+) | yes (15+)  |
| riscv       | -mstack-protector-guard=tls
              -mstack-protector-guard-reg=tp
              -mstack-protector-guard-offset=...TSK_STACK_CANARY... | yes (12.1+) | needed     |
| powerpc     | -mstack-protector-guard=tls
              -mstack-protector-guard-reg=r13 | yes (7.1+) | needed?    |
Work needed: forward edge CFI

- **CPU hardware support (coarse-grain: marked entry point matching) at parity**
  - x86 ENDBR instruction, GCC & Clang (CONFIG_X86_KERNEL_IBT):
    - `-fcf-protection=branch`
  - arm64 BTI instruction, GCC & Clang (CONFIG_ARM64_BTI_KERNEL):
    - `-mbranch-protection=bti`
    - `__attribute__((target("branch-protection=bti")))`
    - GCC bug [still open](#)

- **Software (fine-grain: per-function-prototype matching)**
  - Clang: inline hash checking: `-fsanitize=kcfi` (arm64 and x86_64)
  - GCC: inline hash checking needed (earlier [arm64 effort](#) needs more attention)

- **Exploitation of func pointers easier than ever via automated gadget discovery**
  - [https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei](https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei)
Work needed: backward edge CFI

- CPU hardware support at parity
  - x86 Shadow Stack CPU feature bit and implicit operation: no compiler support needed
    - Kernel support landed finally (Shadow Stack systems available for 3 years now)!
    - In-kernel Shadow Stack still not explored yet.
  - arm64 PAC instructions, GCC and Clang (CONFIG_ARM64_PTR_AUTH_KERNEL):
    - -mbranch-protection=pac-ret[+leaf]
    - __attribute__((target("branch-protection=pac-ret[+leaf]")))  

- Software (shadow stack)
  - x86: inline hash checking (like kCFI) would be nice to have in both Clang and GCC
  - arm64 shadow call stack: GCC (12.1+) and Clang (CONFIG_SHADOW_CALL_STACK):
    - -fsanitize=shadow-call-stack
In progress: bounds-checked Flexible Array Members

New attribute to annotate bounds of FAMs to enable flexible array bounds checking at runtime:

```c
struct object {
    int items;
    int flex[] __attribute__((__counted_by__(items)));
};
```

Use new attribute for array bounds check of flexible arrays (via -fsanitize=bounds) and __builtin_dynamic_object_size() too (for FORTIFY_SOURCE).
GCC Status: counted_by attribute (*The Current Plan*)

1. Provide counted_by attribute to flexible array member (FAM).
2. Use the attribute in __builtin_dynamic_object_size (subobject only).
3. Use the attribute in array bounds sanitizer.
4. Improve __builtin_dynamic_object_size to use the attribute for whole-object.
5. Emit warnings when the user breaks the requirements for the new attribute.

We planned to finish 1-3 in GCC14, and then 4-5 in GCC15. Have submitted 3rd version for patches 1-3 to GCC upstream on 8/29/2023. Due to a missing data dependency issue raised during review, have to postpone all 1-5 to GCC15.
The call to `__bdos` at line 8 will use `obj->size` at line 7. This implicit data dependency is missing in the source code. Compiler might reorder these two statements or apply other wrong optimizations without the data dependency presenting.
The solution

- a new GCC internal function to carry this data dependency.
  `.ACCESS_WITH_SIZE (REF_TO_OBJ, REF_TO_SIZE, ...)`
- replace every reference to a FAM field with `counted_by` with this function.

```c
7  obj->size = sz;
    tmp = .ACCESS_WITH_SIZE (obj->buf, &obj->size, ...)
8  return __builtin_dynamic_object_size (tmp, 1);
```
GCC Status: counted_by attribute (*The User Interface*)

counted_by (COUNT)

The number of the elements of the FAM is given by the field named COUNT in the same structure.

```c
struct P {
    size_t count;
    char array[] __attribute__ ((counted_by (count)));
} *p;
```

Two Requirements:
1. `p->count` should be initialized before the first reference to `p->array`.
2. `p->array` has at least `p->count` number of elements available all the time.

One important feature:
A ref to the FAM will use the latest value assigned to the size field:

```c
    p->count = val1; ref1 (p->array);
    p->count = val2; ref2 (p->array);
    ref1 uses val1, ref2 uses val2.
```
GCC Status: counted_by attribute (*A Small Example*)

test.h:

```c
struct annotated {
    size_t count;
    char other;
    char array[] __attribute__((counted_by (count)));
};

/* Compute the minimum # of bytes needed to hold a structure "annotated",
   whose # of elements of "array" is COUNT. */
#define MAX(A, B) (A > B) ? (A) : (B)
#define ALLOC_SIZE_ANNOTATED(COUNT) \
    MAX(sizeof (struct annotated), \
        offsetof(struct annotated, array[0]) + (COUNT) * sizeof(char))

/* Allocate the memory for the structure with FAM,
   update "count" with the # of elements "index". */
static struct annotated * __attribute__((__noinline__)) alloc_buf (int index) {
    struct annotated *p;
    p = (struct annotated *) malloc (ALLOC_SIZE_ANNOTATED(index));
    p->count = index;
    return p;
}
```
GCC Status: counted_by attribute (*A Small Example*)

Use counted_by in bound sanitizer:

test.c:
#include "test.h"
int main ()
{
    struct annotated *p_annotated = alloc_buf (10);
    p_annotated->array[11] = 0; // out-of-bounds access, can GCC detect it?
    return 0;
}

Yes, it can with the counted_by attribute:

$ my_gcc -O2 -fsanitize=bounds test.c && ./a.out
test.c:22:21: runtime error: index 11 out of bounds for type 'int [*]'

GCC Status: counted_by attribute (A Small Example)

Use counted_by in __bdos for sub-object size:

test.c:
#include "test.h"
#include <stdio.h>

int main ()
{
    struct annotated *p = alloc_buf (10);
    printf ("The max __bdos sub-object is %lu\n",
            __builtin_dynamic_object_size (p->array, 1));
    // Can GCC compute the sub-object size now?
    return 0;
}

Yes, it can with the counted_by attribute:

$ my_gcc -O2 test.c && ./a.out
The max __bdos sub-object is 10
GCC Status: counted_by attribute (*Further Improvement*)

Improve __bdos for whole-object size!!

In general, given a structure with fixed-size trailing array:

```c
struct fixed {
    size_t count;
    char array[10];
};
```

```c
struct fixed *p = alloc_fixed ();
__builtin_dynamic_object_size(p->array, 0)???
```

Q: Can the compiler use the TYPE of “struct fixed” for the whole object size?

A: Theoretically, NO, since `p` might point to an array of “struct fixed”.

But, given a structure with FAM:

```c
struct annotated {
    size_t count;
    char array[]__attribute__((counted_by (count)));
};
```

```c
struct annotated *q = alloc.annotated (10);
__builtin_dynamic_object_size(q->array, 0)???
```

Q: can the compiler use the TYPE and “counted_by” for the whole object size?

A: Yes. Since a structure with FAM can not be an element of an array, so, “q” must point to an single object with “struct annotated”
GCC Status: counted_by attribute (*Further Improvement*)

Issue warnings when user requirements are violated:

1. p->count should be initialized before the first reference to p->array.

```c
struct annotated *p;
p = (struct annotated *) malloc (ALLOC_SIZE_ANNOTATED(n));
p->count = n;
```

2. p->array has at least p->count number of elements available all the time.

```c
struct annotated *p;
p = (struct annotated *) malloc (ALLOC_SIZE_ANNOTATED(n));
p->array[n + 1] = 10; // out-of-bound will not be detected.
p->count = n + SIZE_BUMP;
```

Compilation time: -Wcounted-by
Run time: -fsanitizer=counted-by
GCC Status: counted_by attribute *(Future Work)*

- Add the counted_by attribute for FAM first; (GCC15?)
- Extend the counted_by attribute to general pointers;
- Add more attributes later if needed (sized_by, ended_by, etc);
- Integrate the array bounds information for FAM and general pointers into language syntax and TYPE system.
- The potential to integrate the `-fbounds-safety proposal` into GCC.
Clang Status: counted_by attribute (Current Status)

Working closely with GCC on the implementation. One change from GCC's implementation. Borrowing from Qing's slide:

1. Provide counted_by attribute to flexible array member (FAM).
2. Use the attribute in __builtin_dynamic_object_size (sub-object only).
3. Use the attribute in array bounds sanitizer.
4. Improve __builtin_dynamic_object_size to use the attribute for whole-object.
5. Emit warnings when the user breaks the requirements for the new attribute.
test.h:
struct annotated {
    size_t count;
    char other;
    char array[] __attribute__((counted_by (count)));
};

/* ... MAX and ALLOC_SIZE_ANNOTATED definitions ... */

/* Allocate the memory for the structure with a FAM, 
   update “count” with the # of elements “count”. */
static struct annotated *__attribute__((__noinline__)) alloc_buf(int count) {
    struct annotated *p;
    p = (struct annotated *) malloc(ALLOC_SIZE_ANNOTATED(index));
    p->count = count;
    return p;
}
Clang Status: counted_by attribute (Examples)

```
$ cat test.c
#include <stdio.h>
#include <stdlib.h>
#include "test.h"

extern void foo(char c);

int main () {
    struct annotated *p = alloc_buf (10);

    /* Sanitizer: Out-of-bounds index. */
    foo(p->array[42]);
    return 0;
}
```

$ clang -O2 -fstrict-flex-arrays=3 test.c && ./a.out
test.c:11:9: runtime error: index 42 out of bounds for type 'char *'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior test.c:11:9 in The value is 0.
Clang Status: counted_by attribute (*Examples*)

```
$ cat test.c
#include <stdio.h>
#include <stdlib.h>
#include "test.h"

int main () {
    struct annotated *p = alloc_buf (10);

    /* Size of a flexible array member. */
    printf("The max __bdos(p->array, 1) == %lu.
", __builtin_dynamic_object_size(p->array, 1));
    return 0;
}

$ clang -O2 -fstrict-flex-arrays=3 test.c && ./a.out
The max __bdos(p->array, 1) == 10.
```
Clang Status: counted_by attribute (Examples)

```c
#include <stdio.h>
#include <stdlib.h>
#include "test.h"

int main () {
    struct annotated *p = alloc_buf (10);

    /* Size of pointer within a flexible array member. */
    printf("The max __bdos(&p->array[3]) == %lu.\n",
            __builtin_dynamic_object_size(&p->array[3], 1));
    return 0;
}
```

```bash
$ clang -O2 -fstrict-flex-arrays=3 test.c && ./a.out
The max __bdos(&p->array[3], 1) == 7.
```
Clang Status: counted_by attribute *(Examples)*

```c
$ cat test.c
#include <stdio.h>
#include <stdlib.h>
#include "test.h"

int main () {
    struct annotated *p = alloc_buf (10);

    /* Size of a flexible array member with out-of-bounds indices. */
    printf("The max __bdos(&p->array[-1], 1) == %lu.\n",
            __builtin_dynamic_object_size(&p->array[-1], 1));
    printf("The max __bdos(&p->array[42], 1) == %lu.\n",
            __builtin_dynamic_object_size(&p->array[42], 1));
    return 0;
}

$ clang -O2 -fstrict-flex-arrays=3 test.c && ./a.out
The max __bdos(&p->array[-1], 1) == 0.
The max __bdos(&p->array[42], 1) == 0.
```
Clang Status: counted_by attribute (Examples)

$ cat test.c
#include <stdio.h>
#include <stdlib.h>
#include "test.h"

int main () {
    struct annotated *p = alloc_buf (10);

    /* Size of struct with a flexible array member. */
    printf("The max __bdos(p, 1) == %lu.\n",
           __builtin_dynamic_object_size(p, 1));
    return 0;
}

$ clang -O2 -fstrict-flex-arrays=3 test.c && ./a.out
The max __bdos(p, 1) == 19.
Clang Status: -fbounds-safety (*Future Work*)

- Adopt GCC's data dependency workaround and new flags
- Work with Apple and GCC to implement Apple's bounds safety features:
  - Pointers to a single object: __single
    - Pointer arithmetic is a compile time error
  - External bounds annotations: __counted_by(N), __sized_by(N), and __ended_by(P)
  - Internal bounds annotations (i.e. "Rubenesque" pointers): __bidi_indexable and __indexable
  - Sentinel-delimited arrays: __null_terminated and __terminated_by(T)
  - Annotation for interoperating with bounds-unsafe code: __unsafe_indexable

Work needed: bounds checking for general pointers

- Two types of arrays
  - Fixed-sized bounds in TYPE
  - Dynamically-sized
    - Variable-length array (VLA) bounds in TYPE
    - Flexible array member (FAM) bounds in attribute
    - Pointer offset

- The -fbounds-safety extension offers bounds annotations that can be attached to pointers in general.

(Apple’s RFC for LLVM[1] on May 24, 2023)
Work needed: -fbounds-safety proposal from Apple

- A superset of counted_by attribute
- Covers all the pointers and arrays (including FAM)
- More effort and burden when adopting existing C applications
- We might consider to add this later
Work needed: other aspects of bounds checking

- Handling nested structures ending in a Flexible Array Member (Clang)
  - [https://github.com/llvm/llvm-project/issues/72032](https://github.com/llvm/llvm-project/issues/72032)

- `-Warray-bounds` false positives (GCC, due to jump threading)
  - [https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109071](https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109071)

- Language extension to support Flexible Array Members in Unions

```c
union u {
    int foo;
    char bar[0];
};
```
Work needed: arithmetic overflow protection

● Technically working …
  ○ GCC & Clang:  -fsanitize={signed-integer-overflow,pointer-overflow}
  ○ Clang:  -fsanitize=unsigned-integer-overflow

● … but there are some significant behavioral caveats related to -fwrapv and -fwrapv-pointer (enabled via kernel’s use of -fno-strict-overflow)
  ○ “It’s not an undefined behavior to wrap around.”

● More than avoiding “undefined behavior”, we want no “unexpected behavior”.
  ○ Like run-time bounds checking, need arithmetic overflow to be handled as a trap or “warn and continue with wrapped value” and a way to optionally allow wrap-around.
  ○ It would be nice to have a “warn and continue with saturated value” mode instead, to reduce the chance of denial of service and reach normal error checking.

● Clarify language for “overflow” vs “wrap around”
Questions / Comments?

Thank you for your attention!

Kees Cook <keescook@chromium.org>
Qing Zhao <qing.zhao@oracle.com>
Bill Wendling <morbo@google.com>
Bonus Slides…
counted_by may track logical (instead of allocated) size

```c
struct annotated {
    unsigned short allocated;
    unsigned short usable;
    ...
    struct foo array[] __attribute__((counted_by (usable)));
};

struct annotated *p;
int i = 0;

p = malloc(sizeof(*p) + sizeof(p->array[0]) * max_item_queue_size);
p->allocated = max_item_queue_size;
p->usable = 0;
while (items_available() && i < p->allocated) {
    p->usable ++;
    memcpy(&p->array[i++], next_item(), sizeof(p->array[0]));
}
```
Work needed: Link Time Optimization

- Toolchain support is at parity
  - GCC: -flto
  - Clang: -flto or -flto=thin

- Linux kernel support is only present with Clang
- No recent patches sent to LKML
- Latest development branch (against v5.19) appears to be Jiri Slaby’s, continuing Andi Kleen’s work:
  - [https://git.kernel.org/pub/scm/linux/kernel/git/jirislaby/linux.git/log/?h=lto](https://git.kernel.org/pub/scm/linux/kernel/git/jirislaby/linux.git/log/?h=lto)
Work needed: Spectre v1 mitigation

- GCC: wanted? no open bug...
- Clang:
  - -mspeculative-load-hardening
  - __attribute__((speculative_load_hardening))
  - [https://llvm.org/docs/SpeculativeLoadHardening.html](https://llvm.org/docs/SpeculativeLoadHardening.html)

- Performance impact is relatively high, but lower than using lfence everywhere.
- Really needs some kind of “reachability” logic to reduce overhead.

- Does anyone care about this?