Compiler Features for Kernel Security

Kees Cook <keescook@chromium.org>
Qing Zhao <qing.zhao@oracle.com>

skipping various common features

- stack canaries: `-fstack-protector -fstack-protector-strong`
- uninitialized variable analysis: `-Wuninitialized -Wmaybe-uninitialized`
- format string safety analysis: `-Wformat -Wformat-security`
- Position Independent Executable to use ASLR: `-Wl,-z,pie -fPIE`
- Variable Length Array analysis: `-Wvla`
- Spectre v2:
  - GCC: `-mindirect-branch -mfunction-return`
  - Clang: `-mretpoline`
# Flashback! 2020’s Features Needing Attention

<table>
<thead>
<tr>
<th>Feature</th>
<th>GCC</th>
<th>Clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack protector guard location</td>
<td>arm64 riscv arm32</td>
<td>arm64 riscv arm32</td>
</tr>
<tr>
<td>Zero call-used registers</td>
<td>proposed</td>
<td>no</td>
</tr>
<tr>
<td>Stack variable auto-initialization</td>
<td>plugin</td>
<td>yes</td>
</tr>
<tr>
<td>Array bounds checking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signed overflow protection</td>
<td>conflicts with other options</td>
<td>conflicts with other options</td>
</tr>
<tr>
<td>Unsigned overflow protection</td>
<td>no</td>
<td>conflicts with other options</td>
</tr>
<tr>
<td>Link Time Optimization</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Forward edge CFI</td>
<td>hardware only</td>
<td>yes</td>
</tr>
<tr>
<td>Backward edge CFI</td>
<td>hardware only</td>
<td>hardware w/ arm64 soft</td>
</tr>
<tr>
<td>Lvalue introspection builtin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure layout randomization</td>
<td>plugin</td>
<td>no</td>
</tr>
<tr>
<td>Spectre v1 mitigation</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

| Flashback! 2020’s Features Needing Attention |
|----------------------------------------------|--------------|---------------|
| Stack protector guard location               | arm64 riscv arm32 | arm64 riscv arm32 |
| Zero call-used registers                     | proposed     | no            |
| Stack variable auto-initialization           | plugin       | yes           |
| Array bounds checking                        |              |               |
| Signed overflow protection                   | conflicts with other options | conflicts with other options |
| Unsigned overflow protection                 | no           | conflicts with other options |
| Link Time Optimization                       | yes          | yes           |
| Forward edge CFI                             | hardware only | yes           |
| Backward edge CFI                            | hardware only | hardware w/ arm64 soft |
| Lvalue introspection builtin                |              |               |
| Structure layout randomization               | plugin       | no            |
| Spectre v1 mitigation                        | no           | yes           |
# features needing attention

<table>
<thead>
<tr>
<th>Feature</th>
<th>GCC</th>
<th>Clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack protector guard location</td>
<td>arm64 riscv arm32</td>
<td>arm64 riscv arm32</td>
</tr>
<tr>
<td>zero call-used registers</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>stack variable auto-initialization</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>array bounds checking</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>signed overflow protection</td>
<td>conflicts with other options</td>
<td>conflicts with other options</td>
</tr>
<tr>
<td>unsigned overflow protection</td>
<td>no</td>
<td>conflicts with other options</td>
</tr>
<tr>
<td>Link Time Optimization</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>forward edge CFI</td>
<td>hardware only</td>
<td>yes</td>
</tr>
<tr>
<td>backward edge CFI</td>
<td>hardware only</td>
<td>hardware w/ arm64 soft</td>
</tr>
<tr>
<td>lvalue introspection builtin</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>structure layout randomization</td>
<td>plugin</td>
<td>no</td>
</tr>
<tr>
<td>Spectre v1 mitigation</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
stack protector guard location

- GCC: supported on arm64 & riscv, **needed** on arm32
- Clang: supported on arm64, **needed** on riscv & arm32

- `mstack-protector-guard=sysreg`
- `mstack-protector-guard-reg=sp_el0`
- `mstack-protector-guard-offset=0`

- Provides per-thread stack canaries in the kernel (otherwise the canary is a per-boot global value for all threads)
- (x86 & powerpc are already supported via its existing Thread Local Storage implementation)
- Canary value is leaky :(  See https://github.com/KSPP/linux/issues/29

5/25
zero call-used regs on func return

- GCC: since version 11
  -fzero-call-used-regs=[skip|used-gpr|all-gpr|used|all]
  (open issues: possible arm32 ICE and a request to always use XOR)

- Clang: needed

- Supported in the kernel since v5.15 as CONFIG_ZERO_CALL_USED_REGS (only using used-gpr)
- Virtually no performance impact (register self-xor is highly pipelined), and strongly frustrates ROP gadget utility. Also makes sure those register contents cannot be used for speculation-style attacks.
- https://github.com/KSPP/linux/issues/84
stack variable auto-initialization

- GCC: added in version 12
- Clang: supported

- `ftrivial-auto-var-init=zero`
- `ftrivial-auto-var-init=pattern`

- Not intended to remove `-Wuninitialized` coverage.
- Linus wants to be able to depend on zeroing in the kernel.
- The zeroing mode is enabled by default in Android, Chrome OS, and XNU via Clang, and the Windows kernel via VC++’s similar option.
array bounds checking: goals

• Kernel has been converting all legacy 0-element and 1-element arrays to flexible arrays to gain sane bounds checking:
  – Keep 0-element arrays out of the kernel source (except in legacy UAPI headers)
  – Warn about overlapping 0-element arrays (to make sure no bad UAPI use creeps in):
    – Never access beyond array size ...
      • warn if size and index are known at compile-time
      • freak out if run-time index is larger than size
array bounds checking
(no 0-element arrays)

• Keep 0-element arrays out of the kernel source:

```c
struct legacy {
    unsigned long flags;
    size_t count;
    int elements[0]; /* <- change to "int elements[];" */
};
```

- Clang has `-Wzero-length-array` (except that UAPI must keep them forever)
- GCC feature has been requested
- Both need a struct attribute to ignore certain structures declarations (UAPI will have 0-element arrays for a long time)
array bounds checking
(warn on overlap)

• Warn about using 0-element arrays when they overlap with other members (i.e. make sure no bad UAPI use continues)
  
  ```c
  struct legacy {
      unsigned long flags;
      union {
          int weird[0];
          struct stuff not_weird;
      }
  } instance;
  ...
  instance.weird[0] = something;
  ```

- GCC: `-Wzero-length-bounds`
- Clang should likely gain this coverage
array bounds checking
(check for index overflow ...)

• Never index beyond array size
  – No current way in C to deal with flexible arrays, but some great proposals for language extensions:

```
struct variable_size {
    size_t count;
    ...
    int elements[.count];
};
```

  – For everything else, coverage is possible now when the array size is known at compile time:

```
struct something instance[8]; /* size is 8: indexes can be 0 to 7. */
```

• When index is known at compile time, warn: -Warray-bounds

```
instance[12] = ... /* build warning */
```

• When index is only known at run-time, perform check at run-time: -fsanitize=bounds

```
instance[index] = ... /* run-time freak out when index < 0 or index > 7 */
```
array bounds checking
... at compile time

• GCC and Clang: `-Warray-bounds` *(with caveats noted below)*
  
  ```c
  struct something {
      ...
      int elements[1];
  } instance, *ptr;
  ```

  – Clang pretends 0-element and 1-element arrays are flexible arrays, and does not enforce checks on such members:
    
    ```c
    instance.elements[3] = ...; /* no warning! :( */
    ptr->elements[3] = ...;     /* no warning! :( */
    ```

  – GCC pretends *dereferences* to 0/1-element arrays are flexible arrays and does not enforce checks:
    
    ```c
    instance.elements[3] = ...; /* warning :) */
    ptr->elements[3] = ...;     /* no warning! :( */
    ```

  • worse: `__builtin_object_size()` thinks all trailing arrays have unknown size, breaking FORTIFY_SOURCE depending on struct layout!

  – Both compilers need an option for “no legacy flexible array handling”
array bounds checking
... at run time

- GCC and Clang: `-fsanitize=bounds` (with similar caveats)
  - Clang has more knobs: `-fsanitize=bounds` contains two options:
    - `fsanitize=array-bounds`
    - `fsanitize=local-bounds` (but is only trapping?!)  
- But, of course, both pretend 0/1-element arrays are flexible arrays
  - GCC can disable this with `-fsanitize=bounds-strict`
  - Clang needs this (or perhaps just the new option proposed on prior slide)
- How should the kernel freak out on run-time bounds failure?
  - Warn (doesn't stop the overflow)
  - Trap (i.e. `BUG()`, denial of service)
  - Exception handling (needs to be done manually in C)
bonus: __builtin_dynamic_object_size

- FORTIFY_SOURCE is implemented mainly through the use of __builtin_object_size (with the various bugs above), but lacks any visibility into run-time sizes (usually via alloc_size function attribute).

- Expand coverage to run time with __builtin_dynamic_object_size
  - Clang: implemented
  - GCC: discussed

    thing->obj = kmalloc(alloc_size, GFP_KERNEL);

    ... 

    if (write_size > __builtin_dynamic_object_size(thing->obj, 1)) {
        /* freak out */
    }

- Yes, yes, “why not check alloc_size?”, but this is desired for use in helpers that only have visibility into thing and write_size but not alloc_size (think memcpy(), and similarly expanded FORTIFY_SOURCE coverage).
signed overflow protection

• GCC and Clang: technically working ...
  - `fsanitize=signed-integer-overflow`

• There are, however, some significant behavioral caveats related to `-fwrapv` and `-fwrapv-pointer` (which are enabled by `-fno-strict-overflow`)
  - “It's not an undefined behavior to wrap around.”

• Like run-time bounds checking, arithmetic overflow can be handled as a Trap, or “Warn and continue with wrapped value”
  - 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.
unsigned overflow detection

- GCC: needed
- Clang: working, with similar problems as in prior slide ...
  
  `-fsanitize=unsigned-integer-overflow`

- This one isn’t technically “undefined behavior”, but it certainly leads to exploitable (or at least unexpected) conditions.
- Similar issues as signed overflow:
  - behavioral caveats related to `-fno-strict-overflow`
  - would be nice to have a “warn and continue with saturated value” mode
Link Time Optimization

- GCC: `-flto`
- Clang: `-flto` or `-flto=thin`

- Required for software-based forward edge Control Flow Integrity.
- Works with the kernel, but *only with Clang*.
  - Most recent GCC LTO series hasn’t been sent to LKML in a long time, but continues to be worked on by Andi Kleen:
CFI (forward edge: indirect calls)

- hardware (coarse-grain: entry points)
  - x86: ENDBR instruction
    - GCC and Clang: -fcf-protection=branch
  - arm64: BTI instruction
    - GCC and Clang:
      ```
      -mbranch-protection=bt
      __attribute__((target("branch-protection=bt")))
      ```

- software (fine-grain: per-function-prototype)
  - GCC: needed (though there is -fvtable-verify=[std|preinit|none] for C++)
  - Clang: -fsanitize=cfi

- We really need fine-grain forward edge CFI: stops automated gadget exploitation
  - [https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei](https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei)
CFI (backward edge: returns)

- **hardware**
  - x86: CET CPU feature bit and implicit operation: no compiler support needed!
  - arm64: PAC instructions, supported by both GCC and Clang:
    - `mbranch-protection=pac-ret[+leaf]`
    - `__attribute__((target("branch-protection=pac-ret[+leaf]")))`

- **software shadow stack**
  - x86: none (Want CET! Please, test the series and review it. Linux is behind)
  - arm64:
    - GCC: needed
    - Clang: `-fsanitize=shadow-call-stack`
lvalue introspection builtin

- GCC and Clang: not implemented
- Needed to build a type-aware allocator drop-in replacement to minimize the impact of type-confused use-after-free flaws. Unlikely to convince folks to rewrite the existing idiom from:
  ```c
  instance = kmalloc(size, GFP_KERNEL);
  ```
  into:
  ```c
  kmalloc(instance, size, GFP_KERNEL);
  ```
  - If `size` is `sizeof(*instance)`, allocation can live in `typeof(*instance)` bucket
  - otherwise, it's a flexible array: allocation can live in “flexible `typeof(*instance)`” bucket
  - `kmalloc()` macro side of assignment has no visibility into the type of `instance`. :(  
- Perhaps something like `__builtin_lvalue()` that resolves to the lvalue, or `__builtin_lvalue_type()`?
structure layout randomization

__attribute__((randomize_layout))

- GCC: kernel plugin
- Clang: proposed but stalled needing work
  - Fun for really paranoid builds
  - Most users of the features are highly interested in build diversity
  - Used by at least one phone vendor
Spectre v1 mitigation

- GCC: wanted? no open bug...
- Clang:
  
  -mspeculative-load-hardening
  __attribute__((speculative_load_hardening))
  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.
What’s next for GCC

● known issues for `-fzero-call-used-regs`
  - Always use XOR (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101891)
  - ICE with `-mthumb` (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100775)

● known issues for `-ftrivial-auto-var-init`
  - Missing `-Wuninitialized` warning for address taken variables
  - Spurious warning (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=102276)
  - ICEs
    http://gcc.gnu.org/bugzilla/show_bug.cgi?id=102285
    https://gcc.gnu.org/bugzilla/show_bug.cgi?id=102281
What’s next for GCC

• New tasks:
  - Adjust signed integer overflow detector to work with \( -fwrapv \)
    https://gcc.gnu.org/bugzilla/show_bug.cgi?id=102317
  - Provide an option to turn off the GCC heuristic “all trailing arrays are flexible array”:
    https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101836
  - Unsigned overflow detection;
    \((-fsanitize=unsigned-integer-overflow)\)
  - What else?
Thank you; stay safe!

Thoughts? Questions?


Kees (“Case”) Cook
keescook@chromium.org
@kees_cook

Qing Zhao
qing.zhao@oracle.com