Security Feature Parity: GCC and Clang

Kees Cook <keescook@chromium.org>
skipping lots of “at parity”(?) features

- stack canaries: -fstack-protector -fstack-protector-strong
- uninitialized variable analysis: -Wuninitialized -Wmaybe-uninitialized
- format string safety analysis: -Wformat -Wformat-security
- read-only relocations: -Wl,-z,relro
- immediate bindings: -Wl,-z/bindnow
- Position Independent Executable to use ASLR: -Wl,-z,pie -fPIE
- Variable Length Array analysis: -Wvla
- Spectre v2:
  - gcc: -mindirect-branch -mfunction-return
  - clang: -mretpoline
## features needing attention

<table>
<thead>
<tr>
<th>Feature</th>
<th>gcc</th>
<th>clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Time Optimization</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>stack utilization probing</td>
<td>yes</td>
<td>x86 yes</td>
</tr>
<tr>
<td>stack protector guard location</td>
<td>arm64 yes, riscv proposed</td>
<td>no</td>
</tr>
<tr>
<td>Spectre v1 mitigation</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>caller-saved register wiping</td>
<td>proposed</td>
<td>no</td>
</tr>
<tr>
<td>stack variable auto-initialization</td>
<td>plugin</td>
<td>yes</td>
</tr>
<tr>
<td>structure layout randomization</td>
<td>plugin</td>
<td>no</td>
</tr>
<tr>
<td>signed overflow protection</td>
<td>yes, usability issues</td>
<td>yes, usability issues</td>
</tr>
<tr>
<td>unsigned overflow protection</td>
<td>no</td>
<td>yes, usability issues</td>
</tr>
<tr>
<td>backward edge CFI</td>
<td>hardware only</td>
<td>hardware w/ arm64 soft</td>
</tr>
<tr>
<td>forward edge CFI</td>
<td>hardware only</td>
<td>yes</td>
</tr>
</tbody>
</table>
# flashback! 2019’s features needing attention

<table>
<thead>
<tr>
<th>Feature</th>
<th>gcc</th>
<th>clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Time Optimization</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>stack utilization probing</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>stack protector guard location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spectre v1 mitigation</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>caller-saved register wiping</td>
<td>patch</td>
<td>no</td>
</tr>
<tr>
<td>stack variable auto-initialization</td>
<td>plugin</td>
<td>yes</td>
</tr>
<tr>
<td>structure layout randomization</td>
<td>plugin</td>
<td>no</td>
</tr>
<tr>
<td>signed overflow protection</td>
<td>yes, usability issues</td>
<td>yes, usability issues</td>
</tr>
<tr>
<td>unsigned overflow protection</td>
<td>no</td>
<td>yes, usability issues</td>
</tr>
<tr>
<td>backward edge CFI</td>
<td>hardware only</td>
<td>hardware w/ arm64 soft</td>
</tr>
<tr>
<td>forward edge CFI</td>
<td>hardware only</td>
<td>yes</td>
</tr>
</tbody>
</table>
## features needing attention

<table>
<thead>
<tr>
<th>Feature</th>
<th>gcc</th>
<th>clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Time Optimization</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>stack utilization probing</td>
<td>yes</td>
<td>x86 yes</td>
</tr>
<tr>
<td>stack protector guard location</td>
<td>arm64 yes, riscv proposed</td>
<td>no</td>
</tr>
<tr>
<td>Spectre v1 mitigation</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>caller-saved register wiping</td>
<td>proposed</td>
<td>no</td>
</tr>
<tr>
<td>stack variable auto-initialization</td>
<td>plugin</td>
<td>yes</td>
</tr>
<tr>
<td>structure layout randomization</td>
<td>plugin</td>
<td>no</td>
</tr>
<tr>
<td>signed overflow protection</td>
<td>yes, usability issues</td>
<td>yes, usability issues</td>
</tr>
<tr>
<td>unsigned overflow protection</td>
<td>no</td>
<td>yes, usability issues</td>
</tr>
<tr>
<td>backward edge CFI</td>
<td>hardware only</td>
<td>hardware w/ arm64 soft</td>
</tr>
<tr>
<td>forward edge CFI</td>
<td>hardware only</td>
<td>yes</td>
</tr>
</tbody>
</table>
Link Time Optimization

- gcc: -flto
- clang: -flto or -flto=thin

• Required for software-based forward edge Control Flow Integrity.
• Lots of pain to update kernel build tooling but Sami Tolvanen is keeping it working and grinding through getting it upstream, but only Clang is being tested.
  - https://github.com/samitolvanen/linux/commits/clang-lto
stack utilization probing

- gcc: -fstack-clash-protection
- clang: x86 supported, other architectures needed

- Defense against giant VLAs/alloca()s
- Kernel removed all VLA usage, so this is mainly a concern for userspace.
stack protector guard location

- gcc: arm64 supported, riscv proposed
  -mstack-protector-guard=sysreg
  -mstack-protector-guard-reg=sp_el0
  -mstack-protector-guard-offset=0

- clang: needed

- Provides per-thread stack canaries in the kernel (otherwise the canary is a per-boot global value for all threads)

- (x86 is already supported via its existing Thread Local Storage implementation)
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.
zero caller-saved regs on func return

- **gcc**: proposed `-fzero-call-used-regs=[skip|used-gpr|all-gpr|used|all]`
  
earlier patch for `-mzero-caller-saved-regs=used`
  
  [link](https://github.com/clearlinux-pkgs/gcc/blob/master/0001-x86-Add-mzero-caller.patch)

- **clang**: needed

- 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.

- [link](https://github.com/KSPP/linux/issues/84)
stack variable auto-initialization

- gcc: *kernel plugin*
- clang:
  - `-ftrivial-auto-var-init=pattern`
  - `-ftrivial-auto-var-init=zero`

- Linus wants to be able to **depend on zeroing** in the kernel
- The zeroing mode is now **enabled by default** in Android, Chrome OS, and XNU via Clang, and the Windows kernel via VC++’s similar option
- IIUC, this feature has been getting discussed in the GCC universe, but I can’t find public references ...
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
signed overflow protection

- fsanitize=signed-integer-overflow

• gcc: working!
• clang: working!

• There are, however, some behavioral caveats related to
  -fno-strict-overflow (which implies -fwrapv-pointer and -fwrapv)
• Also, it would be nice to have a “warn and continue with saturated value”
  mode instead of either “die” or “warn and continue with wrapped value”.
unsigned overflow detection

- `fsanitize=unsigned-integer-overflow`
  - gcc: needed
  - clang: working!

- This one isn’t technically “undefined behavior”, but it certainly leads to exploitable (or at least unexpected) conditions.
- Same thoughts as signed overflow:
  - behavioral caveats related to `-fno-strict-overflow`
  - would be nice to have a “warn and continue with saturated value” mode
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 (wait for CET?)
  - arm64:
    - gcc: needed
    - clang: -fsanitize=shadow-call-stack
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=bti`
      - `__attribute__((target("branch-protection=bti")))`

- **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
Thank you; stay safe!

Thoughts? Questions?

Kees ("Case") Cook

keescook@chromium.org
keescook@google.com
kees@outflux.net

@kees_cook