Security countermeasure strikes at several levels:

- hardware (buy a new machine)
- microcode (update your kernel)
- compiler (pick compiler flags)
- codebase (update your code)
ALWAYS TWO THERE ARE

Focusing on C/C++:

- GCC Toolchain (mostly gcc + ld.bfd)
- LLVM Toolchain (mostly clang + lld)
SECURITY RELATED FLAGS: A COMMON GCC/LLVM VIEW

Serge « sans paille » Guelton

Compiler Engineer / Wood Craft Lover / Red Hat employee

LPC’20 — 26th of August 2020
DEFAULT FEDORA FLAGS
(X86_64)

-o2 -g -pipe -Wall -Werror=format-security \
-Wp,-D_FORTIFY_SOURCE=2 -Wp,-D_GLIBCXX_ASSERTIONS \
-fexceptions -fstack-protector-strong \
-grecord-gcc-switches \
-specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 \
-specs=/usr/lib/rpm/redhat/redhat-annobin-cc1 -m64 \
-mtune=generic -fasynchronous-unwind-.tables \
-fstack-clash-protection -fstr-preservation \
-Wl,-z,relro -Wl,--as-needed -Wl,-z,now \
-specs=/usr/lib/rpm/redhat/redhat-hardened-ld
DEFAULT DEBIAN FLAGS (X86_64)

- g -O2 \\
- fdebug-prefix-map=/home/sylvestre/dev/debian/pkg-llvm/\\
/llvm-toolchain/branches=. -fstack-protector-strong \ 
- Wformat -Werror=format-security -Wl,-z,relro

* https://sources.debian.org/src/gcc-10/10.1.0-1/debian/rules.patch
COMMON LIBRARY EXPLOITATION

Attack: Exploit standard C/C++ functions misuse

Countermeasure: Provide fortified version of these functions

Flag: -D_FORTIFY_SOURCE (gcc, clang for builtin supports), -D_GLIBCXX_ASSERTIONS

Overhead: low (fortify) to high (asserts)

Artifact: nm a.out | grep __strcpy_chk
COMMON FORMATTING ATTACKS

**Attack:** Exploit user-controlled formating arguments

**Countermeasure:** Warn about dubious patterns

**Flag:** `-Werror=format-security (gcc, clang)`

**Overhead:** nop (compile time)
**COMMON CODE OVERFLOWS**

*Attack*: Exploit buffer overflow

*Countermeasure*: Range analysis

*Flag*: `-Werror=array-bounds` *(gcc, clang)*

*Overhead*: nop (compile time)
**UNINITIALIZED STACK VARIABLES**

*Attack*: Use uninitialized variable to leak previous state

*Countermeasure*: Always initialize stack variable

*Flag*: `-ftrivial-auto-var-init=pattern` (clang)

*Overhead*: yes (?)
GOT / PLT OVERWRITE

**Attack:** Overwrite the GOT/PLT to overwrite executable sections

**Countermeasure:** Load everything then mark GOT/PLT read-only

**Flag:** `-Wl,-z,relro,-Wl,-z,now` (ld.bfd, lld)

**Overhead:** increased startup time

**Artifact:** `readelf -a now | grep BIND_NOW`
**EXECUTABLE STACK**

*Attack:* Overwrite an executable stack with malicious code

*Countermeasure:* Mark the stack as non-executable

*Flag:* `-Wl,-z,noexecstack` (*ld.bfd, lld*)

*Overhead:* `nop (?)`

*Artifact:* `readelf -e a.out | { ! grep -E 'GNU_STACK.*RWE' ; }`
SECURITY THROUGH DIVERSITY

**Attack:** Use hardcoded address in shellcodes/others

**Countermeasure:** Randomize process addresses (ASLR)

**Flag:** `-pie -fPIE or -fPIC (gcc/ld.bfd, clang/ld) + /proc/sys/kernel/randomize_va_space`

**Overhead:** relative jump computation

**Artefact:** `readelf -e a.out | grep 'DYN (Shared object file)'`
STACK CLASH

*Attack:* Make the stack and the heap grow so that they overlap

*Countermeasure:* Probe each page to trigger the kernel page guard

*Flag:* `-fstack-clash-protector (gcc, clang)`

*Overhead:* only for functions with large / dynamic stack alloc

*Artefact:* `objdump -S a.out | grep 'subq 4096, %rsp'`
**STACK SMASH**

*Attack:* Modify the stack thanks to an overflow

*Countermeasure:* Stack Canary, Split Stack

*Flag:* `-fstack-protector-strong` (gcc, clang), `-fsanitize=safe-stack` (clang)

*Overhead:* one check per function, user-controlled granularity

*Artefact:* `nm a.out | grep __stack_chk_fail`
AND NOW FOR SOMETHING DIFFERENT

All these slides were pretty classic, right?
SPECTRE V1

Attack: Trick branch prediction into filling the cache with secret data

Countermeasure: create a data dependency between data access and predicate state

Flag: -mspeculative-load-hardening (clang)

Overhead: non-negligible
**SPECTRE V2**

*Attack*: Trick branch prediction into executing a controlled function pointer

*Countermeasure*: Use return prediction instead of branch prediction

*Flag*: `-mretpoline (clang) -mindirect-branch, -mfunction-return (gcc)`

*Overhead*: non-neglectible
RETURN ORIENTED PROGRAMING

*Attack:* Execute arbitrary code through a chain of gadget

*Countermeasure:* Check Control Flow Integrity / Intel CET, ARM BTI

*Flag:* `-fsanitize=cfi (clang) -fno-control-protection`
`-fno-control-protection (clang, gcc)`
CERTIFICATION

Want to double-check the flags used in the build process?

- `-fplugin=annobin (gcc, clang)`
- `-fgrecord-gcc-switches (gcc)`

**Artefact:** `readelf a.out -p .GCC.command.line`  
| `grep record-gcc-switches`
**POST-COMPILATION CHECK**

For each compiler flag, test for hardening artefacts, à la hardening-check.

EXAMPLE: STACK CLASH PROTECTION

- LLVM implem using the GCC implem as reference
- Different Test beds (GCC: compiler report, LLVM: assembly reference)
- Paths to explore
  - instrumentation-based verification of distance invariant?
  - Static verification?
FOLLOW-UPS

- Convergence of options names is ~OK
- But beside names, implementation differ!
  - Discussing implementation across mlist (or on a common medium?)
  - Sharing compiler-agnostic test beds?
- Thanks to Adrien Guinet, Juan Manuel Martinez Sylvestre Ledru and Florian Weimer!