Intro

Security countermeasure strikes at several level:

- 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 -fcf-protection \
-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 Formating 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

Coutermeasure: 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 adress in shellcodes/others

Countermeasure: Randomize process adresses (ASLR)

Flag: `-pie -fPIE` or `-fPIC` (gcc/ld.bfd, clang/lld) + `/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-neglectible

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) `-fcf-protection` (clang, gcc)

Certification

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

- `-fplugin=annobin` (gcc, clang)
- `-[fg]record-gcc-switches` (gcc)

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

Post-Compilation Check

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

https://github.com/serge-sans-paille/hardening-artefacts

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!