Linters-Sanitizers Tutorial HY-255

Giorgos Xanthakis - gxanth@csd.uoc.gr

Giorgos Kelantonakis – csdp1224@csd.uoc.gr

Linters

- A linter is a tool that scans your code and finds issues that can lead to
- Bugs
- Inconsistencies
  - Code Health
  - Style
- Some linters can even help you to fix the issues mentioned above!

Clang-tidy

- Installation
  - sudo apt-get install clang-tidy
- Documentation
  - https://clang.llvm.org/extra/clang-tidy/
- Checking
  - clang-tidy testhy255.c -checks="*"
- Fixing
  - clang-tidy testhy255.c -checks="*" -fix

Name prefix	Description
abseil-	Checks related to Abseil library.
altera-	Checks related to OpenCL programming for FPGAs.
android-	Checks related to Android.
boost-	Checks related to Boost library.
bugprone-	Checks that target bugprone code constructs.
cert-	Checks related to CERT Secure Coding Guidelines.
clang-analyzer-	Clang Static Analyzer checks.
concurrency-	Checks related to concurrent programming (including threads, fibers, coroutines, etc.).
cppcoreguidelines-	Checks related to C++ Core Guidelines.
darwin-	Checks related to Darwin coding conventions.
fuchsia-	Checks related to Fuchsia coding conventions.
google-	Checks related to Google coding conventions.
hicpp-	Checks related to High Integrity C++ Coding Standard.
linuxkernel-	Checks related to the Linux Kernel coding conventions.
llvm-	Checks related to the LLVM coding conventions.
llvmlibc-	Checks related to the LLVM-libc coding standards.
misc-	Checks that we didn't have a better category for.
modernize-	Checks that advocate usage of modern (currently "modern" means "C++11") language constructs.
mpi-	Checks related to MPI (Message Passing Interface).
objc-	Checks related to Objective-C coding conventions.
openmp-	Checks related to OpenMP API.
performance-	Checks that target performance-related issues.
portability-	Checks that target portability-related issues that don't relate to any particular coding style.
readability-	Checks that target readability-related issues that don't relate to any particular coding style.
zircon-	Checks related to Zircon kernel coding conventions.

Cppcheck

- Installation
  - sudo apt-get install cppcheck
- Documentation
  - https://sourceforge.net/p/cppcheck/wiki/Home/
- Checking
  - cppcheck --language=c . --enable=all --inconclusive --max-ctu-depth=1024 --force

Sanitizers

- Sanitizers are tools provided by compilers that check code at runtime
- There are different sanitizers that each handles a category of bugs

1. Address Sanitizer
   - https://clang.llvm.org/docs/AddressSanitizer.html
2. Leak Sanitizer
   - https://clang.llvm.org/docs/LeakSanitizer.html
3. Undefined Behavior Sanitizer
   - https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
4. Memory Sanitizer
   - https://clang.llvm.org/docs/MemorySanitizer.html
5. Thread Sanitizer
   - https://clang.llvm.org/docs/ThreadSanitizer.html

How to use Sanitizers

1. You need a recent gcc or clang version
   - GCC >= 5
   - Clang >= 3.1
   - The same compiler flags apply for both compilers
2. Address Sanitizer
   - gcc –fsanitize=address -fno-omit-frame-pointer –g
   - ASAN_OPTIONS=abort_on_error=1:detect_stack_use_after_return=1:strict_init_order=1 gdb a.out
3. Leak Sanitizer
   - gcc –fsanitize=leak -fno-omit-frame-pointer -g
4. Undefined Behavior Sanitizer
   - gcc –fsanitize=undefined -fno-omit-frame-pointer -g
5. Memory Sanitizer
   - clang –fsanitize=memory -fno-omit-frame-pointer -g
6. Thread Sanitizer
   - gcc –fsanitize=thread -fno-omit-frame-pointer -g

Example #1

```
//Address
void buffer_overflow(void)
{
        char x = 'a';
        char * a = malloc(sizeof(char) * 10);
        memcpy(a, &x, 11);
        free(a);
}
```

Reading Address Sanitizer Output

```
❯ ./a.out
=================================================================
==866152==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffc6255fd21 at pc 0x7fd0a0315e4b bp 0x7ffc6255fce0 sp 0x7ffc6255f488
READ of size 11 at 0x7ffc6255fd21 thread T0
    #0 0x7fd0a0315e4a in __interceptor_memcpy /usr/src/debug/gcc/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:827
    #1 0x55f527a622eb in buffer_overflow /home/gxanth/Downloads/sanitizers.c:15
    #2 0x55f527a6255e in main /home/gxanth/Downloads/sanitizers.c:47
    #3 0x7fd0a00f830f in __libc_start_call_main (/usr/lib/libc.so.6+0x2d30f)
    #4 0x7fd0a00f83c0 in __libc_start_main@GLIBC_2.2.5 (/usr/lib/libc.so.6+0x2d3c0)
    #5 0x55f527a62134 in _start (/home/gxanth/Downloads/a.out+0x1134)

Address 0x7ffc6255fd21 is located in stack of thread T0 at offset 33 in frame
    #0 0x55f527a62218 in buffer_overflow /home/gxanth/Downloads/sanitizers.c:12

  This frame has 1 object(s):
    [32, 33) 'x' (line 13) <== Memory access at offset 33 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /usr/src/debug/gcc/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:827 in __interceptor_memcpy
Shadow bytes around the buggy address:
  0x10000c4a3f50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000c4a3f60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000c4a3f70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000c4a3f80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000c4a3f90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x10000c4a3fa0: f1 f1 f1 f1[01]f3 f3 f3 00 00 00 00 00 00 00 00
  0x10000c4a3fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000c4a3fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000c4a3fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000c4a3fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10000c4a3ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==866152==ABORTING
```

1

2

3

Example #2

```
//Leak
struct list *memory_leak(int x)
{
        struct list *node = malloc(sizeof(struct list));
        return NULL;
}
```

Example #3

```
//Undefined
void overflow(int argc)
{
        int k = 0x7fffffff;
        k += argc;
}
```

Example #4

```
//Memory
void unitialized(int argc)
{
        int a[10];

        a[5] = 0;

        if (a[argc])
                printf("xx\n");
}
```