google / **sanitizers**

AddressSanitizer

Evgeniy Stepanov edited this page 6 days ago · 23 revisions

Introduction

AddressSanitizer (aka ASan) is a memory error detector for C/C++. It finds:

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

This tool is very fast. The average slowdown of the instrumented program is ~2x (see AddressSanitizerPerformanceNumbers).

The tool consists of a compiler instrumentation module (currently, an LLVM pass) and a run-time library which replaces the `malloc` function.

The tool works on x86, ARM, MIPS (both 32- and 64-bit versions of all architectures), PowerPC64. The supported operation systems are Linux, Darwin (OS X and iOS Simulator), FreeBSD, Android:

OS	x86	x86_64	ARM	ARM64	MIPS	MIPS64	PowerPC64
Linux	yes	yes			yes	yes	yes
OS X	yes	yes					
iOS Simulator	yes	yes					
FreeBSD	yes	yes					
Android			yes	yes			

Other OS/arch combinations may work as well, but aren't actively developed/tested.

See also:

- AddressSanitizerAlgorithm -- if you are curious how it works.
- AddressSanitizerComparisonOfMemoryTools

Getting AddressSanitizer

AddressSanitizer is a part of LLVM starting with version 3.1 and a part of GCC starting with version 4.8 If you prefer to build from source, see AddressSanitizerHowToBuild.

So far, AddressSanitizer has been tested only on Linux Ubuntu 12.04, 64-bit (it can run both 64- and 32-bit programs), Mac 10.6, 10.7 and 10.8, and AddressSanitizerOnAndroid 4.2+.

Pages 74

Find a Page…

Home
AddressSanitizer
AddressSanitizerAlgorithm
AddressSanitizerAndDebugger
AddressSanitizerAndroidPlatform
AddressSanitizerAsDso
AddressSanitizerBasicBlockTracing
AddressSanitizerCallStack
AddressSanitizerClangVsGCC (3.8 vs 6.0)
AddressSanitizerClangVsGCC (5.0 vs 7.1)
AddressSanitizerComparisonOfMemoryTools
AddressSanitizerCompileTimeOptimizations
AddressSanitizerContainerOverflow
AddressSanitizerExampleGlobalOutOfBounds
AddressSanitizerExampleHeapOutOfBounds
Show 59 more pages…

Clone this wiki locally
https://github.com/google

Using AddressSanitizer

In order to use AddressSanitizer you will need to compile and link your program using `clang` with the `-fsanitize=address` switch. To get a reasonable performance add `-O1` or higher. To get nicer stack traces in error messages add `-fno-omit-frame-pointer`. Note: Clang 3.1 release uses another flag syntax.

```
% cat tests/use-after-free.c
#include <stdlib.h>
int main() {
  char *x = (char*)malloc(10 * sizeof(char*));
  free(x);
  return x[5];
}
% ../clang_build_Linux/Release+Asserts/bin/clang -fsanitize=address -O1 -fno-
omit-frame-pointer -g   tests/use-after-free.c
```

Now, run the executable. AddressSanitizerCallStack page describes how to obtain symbolized stack traces.

```
% ./a.out
==9901==ERROR: AddressSanitizer: heap-use-after-free on address 0x60700000dfb5 at
pc 0x45917b bp 0x7fff4490c700 sp 0x7fff4490c6f8
READ of size 1 at 0x60700000dfb5 thread T0
    #0 0x45917a in main use-after-free.c:5
    #1 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-
start.c:226
    #2 0x459074 in _start (a.out+0x459074)
0x60700000dfb5 is located 5 bytes inside of 80-byte region
[0x60700000dfb0,0x60700000e000)
freed by thread T0 here:
    #0 0x4441ee in __interceptor_free projects/compiler-rt/lib
/asan/asan_malloc_linux.cc:64
    #1 0x45914a in main use-after-free.c:4
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-
start.c:226
previously allocated by thread T0 here:
    #0 0x44436e in __interceptor_malloc projects/compiler-rt/lib
/asan/asan_malloc_linux.cc:74
    #1 0x45913f in main use-after-free.c:3
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-
start.c:226
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```

Interaction with other tools

gdb

See AddressSanitizerAndDebugger

ulimit -v

The `ulimit -v` command makes little sense with ASan-ified binaries because ASan consumes 20 terabytes of virtual memory (plus a bit).

You may try more sophisticated tools to limit your memory consumption, e.g. https://en.wikipedia.org/wiki/Cgroups

Flags

See the separate AddressSanitizerFlags page.

Call stack

See the separate AddressSanitizerCallStack page.

Incompatibility

Sometimes an AddressSanitizer build may behave differently than the regular one. See AddressSanitizerIncompatiblity for details.

Turning off instrumentation

In some cases a particular function should be ignored (not instrumented) by AddressSanitizer:

- Ignore a very hot function known to be correct to speedup the app.
- Ignore a function that does some low-level magic (e.g. walking through the thread's stack bypassing the frame boundaries).
- Don't report a known problem. In either case, be **very careful**.

To ignore certain functions, one can use the **no_sanitize_address** attribute supported by Clang (3.3+) and GCC (4.8+). You can define the following macro:

```
#if defined(__clang__) || defined (__GNUC__)
# define ATTRIBUTE_NO_SANITIZE_ADDRESS __attribute__((no_sanitize_address))
#else
# define ATTRIBUTE_NO_SANITIZE_ADDRESS
#endif
...
ATTRIBUTE_NO_SANITIZE_ADDRESS
void ThisFunctionWillNotBeInstrumented() {...}
```

Clang 3.1 and 3.2 supported `__attribute__((no_address_safety_analysis))` instead.

You may also ignore certain functions using a blacklist: create a file `my_ignores.txt` and pass it to AddressSanitizer at compile time using `-fsanitize-blacklist=my_ignores.txt` (This flag is new and is only supported by Clang now):

```
# Ignore exactly this function (the names are mangled)
fun:MyFooBar
# Ignore MyFooBar(void) if it is in C++:
fun:_Z8MyFooBarv
# Ignore all function containing MyFooBar
fun:*MyFooBar*
```

FAQ

- Q: Can AddressSanitizer continue running after reporting first error?
-

A: Yes it can, AddressSanitizer has recently got continue-after-error mode. This is somewhat experimental so may not yet be as reliable as default setting (and not as timely supported). Also keep in mind that errors after the first one may actually be spurious. To enable continue-after-error, compile with `-fsanitize-recover=address` and then run your code with `ASAN_OPTIONS=halt_on_error=0`.

- Q: Why didn't ASan report an obviously invalid memory access in my code?
- A1: If your errors is too obvious, compiler might have already optimized it out by the time Asan runs.
- A2: Another, C-only option is accesses to global common symbols which are not protected by Asan (you can use -fno-common to disable generation of common symbols and hopefully detect more bugs).
- A3: If _FORTIFY_SOURCE is enabled, ASan may have false positives, see below.
- Q: When I link my shared library with -fsanitize=address, it fails due to some undefined ASan symbols (e.g. asan_init_v4)?
- A: Most probably you link with -Wl,-z,defs or -Wl,--no-undefined. These flags don't work with ASan unless you also use -shared-libasan (which is the default mode for GCC, but not for Clang).
- Q: My malloc stacktraces are too short or do not make sense?
- A: Try to compile your code with -fno-omit-frame-pointer or set ASAN_OPTIONS=fast_unwind_on_malloc=0 (the latter would be a performance killer though unless you also specify malloc_context_size=2 or lower). Note that frame-pointer-based unwinding does not work on Thumb.
- Q: My new() and delete() stacktraces are too short or do not make sense?
- A: This may happen when the C++ standard library is linked statically. Prebuilt libstdc++/libc++ often do not use frame pointers, and it breaks fast (frame-pointer-based) unwinding. Either switch to the shared library with the -shared-libstdc++ flag, or use ASAN_OPTIONS=fast_unwind_on_malloc=0. The latter could be very slow.
- Q: I'm using dynamic ASan runtime and my program crashes at start with "Shadow memory range interleaves with an existing memory mapping. ASan cannot proceed correctly.".
- A1: If you are using shared ASan DSO, try LD_PRELOAD'ing Asan runtime into your program.
- A2: Otherwise you are probably hitting a known limitation of dynamic runtime. Libasan is initialized at the end of program startup so if some preceeding library initializer did lots of memory allocations memory region required for ASan shadow memory could be occupied by unrelated mappings.
- Q: The PC printed in ASan stack traces is consistently off by 1?
- A: This is not a bug but rather a design choice. It is hard to compute exact size of preceding instruction on CISC platforms. So ASan just decrements 1 which is enough for tools like addr2line or readelf to symbolize addresses.
- Q: I've ran with ASAN_OPTIONS=verbosity=1 and ASan tells something like

```
==30654== Parsed ASAN_OPTIONS: verbosity=1
==30654== AddressSanitizer: failed to intercept 'memcpy'
```

- A: This warning is false (see https://gcc.gnu.org/bugzilla/show_bug.cgi?id=58680 for details).

- Q: I've built my main executable with ASan. Do I also need to build shared libraries?
- A: ASan will work even if you rebuild just part of your program. But you'll have to rebuild all components to detect all errors.
- Q: I've built my shared library with ASan. Can I run it with unsanitized executable?
- A: Yes! You'll need to build your library with dynamic version of ASan and then run executable with LD_PRELOAD=path/to/asan/runtime/lib.
- Q: I've compiled my code with -D_FORTIFY_SOURCE flag and ASan, or -D_FORTIFY_SOURCE is enabled by default in my distribution. Now ASan misbehaves (either produces false warnings, or does not find some bugs).
- A: Currently ASan (and other sanitizers) doesn't support source fortification, see https://github.com/google/sanitizers/issues/247. The fix should most likely be on the glibc side, see the (stalled) discussion here.
- Q: On Linux I am seeings a crash at startup with something like this

```
ERROR: AddressSanitizer failed to allocate 0x400000000 (17179869184) bytes at
address 67fff8000 (errno: 12)
```

- A: Make sure you don't have `2` in `/proc/sys/vm/overcommit_memory`
- Q: I'm working on a project that uses bare-metal OS with no pthread (TLS) support and no POSIX syscalls and want to use ASan, but its code depends on some stuff (e.g. **dlsym**) that is unavailable on my platform. Does ASan support bare-metal targets?
- A: Out of the box we don't have support for your use case. The easiest for you would be to rip off everything you don't have and rebuild the ASan run-time. However, there have been many attempts in applying ASan to bare-metal and at least some were successful. E.g. http://events.linuxfoundation.org/sites/events/files/slides/Alexander_Popov-KASan_in_a_Bare-Metal_Hypervisor_0.pdf and also grep for "bare-metal" and similar stuff in https://groups.google.com/forum/#!forum/address-sanitizer group.
- Q: Can I run AddressSanitizer with more aggressive diagnostics enabled?
- A: Yes! In particular you may want to enable

```
 CFLAGS += -fsanitize-address-use-after-scope

ASAN_OPTIONS=strict_string_checks=1:detect_stack_use_after_return=1:check_initializa
```

check [Flags wiki] (https://github.com/google/sanitizers/wiki/AddressSanitizerFlags) for more details on this.

- Q: My library crashes with SIGABRT while calling `free` . What's going wrong?
- A: Most probably you are dlopening your library with **RTLD_DEEPBIND** flag. ASan doesn't support **RTLD_DEEPBIND**, see issue #611 for details.

Talks and papers

- Watch the presentation from the LLVM Developer's meeting (Nov 18, 2011): Video, slides.
- Read the USENIX ATC '2012 paper.

Comments?

Send comments to address-sanitizer@googlegroups.com or in Google+.