AddressSanitizer

Introduction

AddressSanitizer is a fast memory error detector. It consists of a compiler instrumentation module and a run-time library. The tool can detect the following types of bugs:

Out-of-bounds accesses to heap, stack and globals
Use-after-free
Use-after-return (runtime flag *ASAN_OPTIONS=detect_stack_use_after_return=1*)
Use-after-scope (clang flag *-fsanitize-address-use-after-scope*)
Double-free, invalid free
Memory leaks (experimental)

Typical slowdown introduced by AddressSanitizer is **2x**.

How to build

Build LLVM/Clang with **CMake**.

Usage

Simply compile and link your program with -fsanitize=address flag. The AddressSanitizer run-time library should be linked to the final executable, so make sure to use clang (not ld) for the final link step. When linking shared libraries, the AddressSanitizer run-time is not linked, so -Wl,-z,defs may cause link errors (don't use it with AddressSanitizer). To get a reasonable performance add -O1 or higher. To get nicer stack traces in error messages add -fno-omit-frame-pointer. To get perfect stack traces you may need to disable inlining (just use -O1) and tail call elimination (-fno-optimize-sibling-calls).

```
% cat example_UseAfterFree.cc
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

```
# Compile and link
% clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer example_UseAfterFree.cc
```

or:

```
# Compile
% clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer -c example_UseAfterFree.cc
# Link
% clang++ -g -fsanitize=address example_UseAfterFree.o
```

If a bug is detected, the program will print an error message to stderr and exit with a non-zero exit code. AddressSanitizer exits on the first detected error. This is by design:

> This approach allows AddressSanitizer to produce faster and smaller generated code (both by ~5%).
> Fixing bugs becomes unavoidable. AddressSanitizer does not produce false alarms. Once a memory corruption occurs, the program is in an inconsistent state, which could lead to confusing results and potentially misleading subsequent reports.

If your process is sandboxed and you are running on OS X 10.10 or earlier, you will need to set DYLD_INSERT_LIBRARIES environment variable and point it to the ASan library that is packaged with the compiler used to build the executable. (You can find the library by searching for dynamic libraries with asan in their name.) If the environment variable is not set, the process will try to re-exec. Also keep in mind that when moving the executable to another machine, the ASan library will also need to be copied over.

Symbolizing the Reports

To make AddressSanitizer symbolize its output you need to set the ASAN_SYMBOLIZER_PATH environment variable to point to the llvm-symbolizer binary (or make sure llvm-symbolizer is in your $PATH):

```
% ASAN_SYMBOLIZER_PATH=/usr/local/bin/llvm-symbolizer ./a.out
==9442== ERROR: AddressSanitizer heap-use-after-free on address 0x7f7ddab8c084 at pc 0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8
READ of size 4 at 0x7f7ddab8c084 thread T0
   #0 0x403c8c in main example_UseAfterFree.cc:4
   #1 0x7f7ddabcac4d in __libc_start_main ??:0
0x7f7ddab8c084 is located 4 bytes inside of 400-byte region [0x7f7ddab8c080,0x7f7ddab8c210)
freed by thread T0 here:
   #0 0x404704 in operator delete[](void*) ??:0
   #1 0x403c53 in main example_UseAfterFree.cc:4
   #2 0x7f7ddabcac4d in __libc_start_main ??:0
previously allocated by thread T0 here:
   #0 0x404544 in operator new[](unsigned long) ??:0
   #1 0x403c43 in main example_UseAfterFree.cc:2
   #2 0x7f7ddabcac4d in __libc_start_main ??:0
==9442== ABORTING
```

If that does not work for you (e.g. your process is sandboxed), you can use a separate script to symbolize the result offline (online symbolization can be force disabled by setting ASAN_OPTIONS=symbolize=0):

```
% ASAN_OPTIONS=symbolize=0 ./a.out 2> log
% projects/compiler-rt/lib/asan/scripts/asan_symbolize.py / < log | c++filt
==9442== ERROR: AddressSanitizer heap-use-after-free on address 0x7f7ddab8c084 at pc 0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8
READ of size 4 at 0x7f7ddab8c084 thread T0
   #0 0x403c8c in main example_UseAfterFree.cc:4
   #1 0x7f7ddabcac4d in __libc_start_main ??:0
...
```

Note that on OS X you may need to run dsymutil on your binary to have the file:line info in the AddressSanitizer reports.

Additional Checks

Initialization order checking

AddressSanitizer can optionally detect dynamic initialization order problems, when initialization of globals defined in one translation unit uses globals defined in another translation unit. To enable this check at runtime, you should set environment variable ASAN_OPTIONS=check_initialization_order=1.

Note that this option is not supported on OS X.

Memory leak detection

For more information on leak detector in AddressSanitizer, see **LeakSanitizer**. The leak detection is turned on by default on Linux, and can be enabled using ASAN_OPTIONS=detect_leaks=1 on OS X; however, it is not yet supported on other platforms.

Issue Suppression

AddressSanitizer is not expected to produce false positives. If you see one, look again; most likely it is a true positive!

Suppressing Reports in External Libraries

Runtime interposition allows AddressSanitizer to find bugs in code that is not being recompiled. If you run into an issue in external libraries, we recommend immediately reporting it to the library maintainer so that it gets addressed. However, you can use the following suppression mechanism to unblock yourself and continue on with the testing. This suppression mechanism should only be used for suppressing issues in external code; it does not work on code recompiled with AddressSanitizer. To suppress errors in external libraries, set the ASAN_OPTIONS environment variable to point to a suppression file. You can either specify the full path to the file or the path of the file relative to the location of your executable.

```
ASAN_OPTIONS=suppressions=MyASan.supp
```

Use the following format to specify the names of the functions or libraries you want to suppress. You can see these in the error report. Remember that the narrower the scope of the suppression, the more bugs you will be able to catch.

```
interceptor_via_fun:NameOfCFunctionToSuppress
interceptor_via_fun:-[ClassName objCMethodToSuppress:]
interceptor_via_lib:NameOfTheLibraryToSuppress
```

Conditional Compilation with __has_feature(address_sanitizer)

In some cases one may need to execute different code depending on whether AddressSanitizer is enabled. **__has_feature** can be used for this purpose.

```
#if defined(__has_feature)
#  if __has_feature(address_sanitizer)
// code that builds only under AddressSanitizer
#  endif
#endif
```

Disabling Instrumentation with __attribute__((no_sanitize("address")))

Some code should not be instrumented by AddressSanitizer. One may use the function attribute __attribute__((no_sanitize("address"))) (which has deprecated synonyms *no_sanitize_address* and *no_address_safety_analysis*) to disable instrumentation of a particular function. This attribute may not be supported by other compilers, so we suggest to use it together with __has_feature(address_sanitizer).

Suppressing Errors in Recompiled Code (Blacklist)

AddressSanitizer supports src and fun entity types in **Sanitizer special case list**, that can be used to suppress error reports in the specified source files or functions. Additionally, AddressSanitizer introduces global and type entity types that can be used to suppress error reports for out-of-bound access to globals with certain names and types (you may only specify class or struct types).

You may use an init category to suppress reports about initialization-order problems happening in certain source files or with certain global variables.

```
# Suppress error reports for code in a file or in a function:
src:bad_file.cpp
# Ignore all functions with names containing MyFooBar:
fun:*MyFooBar*
# Disable out-of-bound checks for global:
global:bad_array
# Disable out-of-bound checks for global instances of a given class ...
type:Namespace::BadClassName
# ... or a given struct. Use wildcard to deal with anonymous namespace.
type:Namespace2::*::BadStructName
# Disable initialization-order checks for globals:
global:bad_init_global=init
type:*BadInitClassSubstring*=init
src:bad/init/files/*=init
```

Suppressing memory leaks

Memory leak reports produced by **LeakSanitizer** (if it is run as a part of AddressSanitizer) can be suppressed by a separate file passed as

```
LSAN_OPTIONS=suppressions=MyLSan.supp
```

which contains lines of the form *leak:<pattern>*. Memory leak will be suppressed if pattern matches any function name, source file name, or library name in the symbolized stack trace of the leak report. See **full documentation** for more details.

Limitations

AddressSanitizer uses more real memory than a native run. Exact overhead depends on the allocations sizes. The smaller the allocations you make the bigger the overhead is.
AddressSanitizer uses more stack memory. We have seen up to 3x increase.
On 64-bit platforms AddressSanitizer maps (but not reserves) 16+ Terabytes of virtual address space. This means that tools like ulimit may not work as usually expected.
Static linking is not supported.

Supported Platforms

AddressSanitizer is supported on:

Linux i386/x86_64 (tested on Ubuntu 12.04)
OS X 10.7 - 10.11 (i386/x86_64)
iOS Simulator
Android ARM
FreeBSD i386/x86_64 (tested on FreeBSD 11-current)

Ports to various other platforms are in progress.

Current Status

AddressSanitizer is fully functional on supported platforms starting from LLVM 3.1. The test suite is integrated into CMake build and can be run with make check-asan command.

More Information

https://github.com/google/sanitizers/wiki/AddressSanitizer