AddressSanitizer

Purpose

AddressSanitizer (ASan) is a fast compiler-based tool for detecting memory bugs in native code. It is comparable to Valgrind (Memcheck tool), but, unlike it, ASan:

- + detects overflows on stack and global objects
- - does not detect uninitialized reads and memory leaks
- + is much faster (two-three times slowdown compared to Valgrind’s 20-100x)
- + has less memory overhead

This document describes how to build and run parts of the Android platform with AddressSanitizer. If you are looking to build a standalone (i.e. SDK/NDK) application with AddressSanitizer, see the AddressSanitizerOnAndroid (https://github.com/google/sanitizers/wiki/AddressSanitizerOnAndroid) public project site instead.

AddressSanitizer consists of a compiler (`external/clang`) and a runtime library (`external/compiler-rt/lib/asan`).

Note: Use the current master branch to gain access to the SANITIZE_TARGET (#sanitize_target) feature and the ability to build the entire Android platform with AddressSanitizer at once. Otherwise, you are limited to using `LOCAL_SANITIZE`.

Building with Clang

As a first step to building an ASan-instrumented binary, make sure that your code builds with Clang. This is done by adding `LOCAL_CLANG:=true` to the build rules. Clang may find bugs in your code that GCC missed.

Building executables with AddressSanitizer

Add `LOCAL_SANITIZE:=address` to the build rule of the executable. This requires: `LOCAL_CLANG:=true`

```
LOCAL_CLANG:=true
LOCAL_SANITIZE:=address
```

When a bug is detected, ASan prints a verbose report both to the standard output and to `logcat` and then crashes the process.

Building shared libraries with AddressSanitizer

Due to the way ASan works, a library built with ASan cannot be used by an executable that's built without ASan.

Note: In runtime situations where an ASan library is loaded into an incorrect process, you will see unresolved symbol messages starting with `_asan` or `_sanitizer`.

To sanitize a shared library that is used in multiple executables, not all of which are built with ASan, you'll need two copies of the library. The recommended way to do this is to add the following to `Android.mk` for the module in question:

```
LOCAL_CLANG:=true
LOCAL_SANITIZE:=address
LOCAL_MODULE_RELATIVE_PATH := asan
```

This puts the library in `/system/lib/asan` instead of `/system/lib`. Then, run your executable with: `LD_LIBRARY_PATH=/system/lib/asan`

For system daemons, add the following to the appropriate section of `/init.rc` or `/init.$device$.rc`.

```
setenv LD_LIBRARY_PATH /system/lib/asan
```

Warning: The `LOCAL_MODULE_RELATIVE_PATH` setting **moves** your library to `/system/lib/asan`, meaning that clobbering and rebuilding from scratch will result in the library missing from `/system/lib`, and probably an unbootable image. That's an unfortunate limitation of the current build system. Don't clobber; do `make -j $N` and `adb sync`.

Verify the process is using libraries from `/system/lib/asan` when present by reading `/proc/$PID/maps`. If it's not, you may need to disable SELinux, like so:

```
$ adb root
$ adb shell setenforce 0
# restart the process with adb shell kill $PID
# if it is a system service, or may be adb shell stop; adb shell start.
```

Better stack traces

AddressSanitizer uses a fast, frame-pointer-based unwinder to record a stack trace for every memory allocation and deallocation event in the program. Most of Android is built without frame pointers. As a result, you will often get only one or two meaningful frames. To fix this, either rebuild the library with ASan (recommended!), or with:

```
LOCAL_CFLAGS:=-fno-omit-frame-pointer
LOCAL_ARM_MODE:=arm
```

Or set `ASAN_OPTIONS=fast_unwind_on_malloc=0` in the process environment. The latter can be very CPU-intensive, depending on the load.

Symbolization

Initially, ASan reports contain references to offsets in binaries and shared libraries. There are two ways to obtain source file and line information:

- Ensure llvm-symbolizer binary is present in `/system/bin`. Llvm-symbolizer is built from sources in: `third_party/llvm/tools/llvm-symbolizer`
- Filter the report through the `external/compiler-rt/lib/asan/scripts/symbolize.py` script.

The second approach can provide more data (i.e. file:line locations) because of the availability of symbolized libraries on the host.

AddressSanitizer in the apps

AddressSanitizer cannot see into Java code, but it can detect bugs in the JNI libraries. For that, you'll need to build the executable with ASan, which in this case is `/system/bin/app_process(`*32|64*`)`. This will enable ASan in all apps on the device at the same time, which is a bit stressful, but nothing that a 2GB RAM device cannot handle.

Add the usual `LOCAL_CLANG:=true, LOCAL_SANITIZE:=address` to the app_process build rule in `frameworks/base/cmds/app_process`. Ignore the `app_process__asan` target in the same file for now (if it is still there at the time you read this). Edit the Zygote record in `system/core/rootdir/init.zygote(`*32|64*`).rc` to add the following lines:

```
setenv LD_LIBRARY_PATH /system/lib/asan:/system/lib
setenv ASAN_OPTIONS
allow_user_segv_handler=true
```

Build, adb sync, fastboot flash boot, reboot.

Using the wrap property

The approach in the previous section puts AddressSanitizer into every application in the system (actually, into every descendant of the Zygote process). It is possible to run only one (or several) applications with ASan, trading some memory overhead for slower application startup.

This can be done by starting your app with the "wrap." property, the same one that's used to run apps under Valgrind. The following example runs the Gmail app under ASan:

```
$ adb root
$ adb shell setenforce 0  # disable SELinux
$ adb shell setprop wrap.com.google.android.gm "asanwrapper"
```

In this context, asanwrapper rewrites `/system/bin/app_process` to `/system/bin/asan/app_process`, which is built with AddressSanitizer. It also adds `/system/lib/asan` at the start of the dynamic library search path. This way ASan-instrumented libraries from `/system/lib/asan` are preferred to normal libraries in `/system/lib` when running with asanwrapper.

Again, if a bug is found, the app will crash, and the report will be printed to the log.

SANITIZE_TARGET

The master branch has support for building the entire Android platform with AddressSanitizer at once.

Run the following commands in the same build tree.

```
$ make -j42
$ make USE_CLANG_PLATFORM_BUILD:=true SANITIZE_TARGET=address -j42
```

In this mode, `userdata.img` contains extra libraries and must be flashed to the device as well. Use the following command line:

```
$ fastboot flash userdata && fastboot flashall
```

At the moment of this writing, hammerhead-userdebug and shamu-userdebug boot to the UI in this mode.

This works by building two sets of shared libraries: normal in `/system/lib` (the first make invocation), ASan-instrumented in `/data/lib` (the second make invocation). Executables from the second build overwrite the ones from the first build. ASan-instrumented executables get a different library search path that includes `/data/lib` before `/system/lib` through the use of "/system/bin/linker_asan" in PT_INTERP.

The build system clobbers intermediate object directories when the `$SANITIZE_TARGET` value has changed. This forces a rebuild of all targets while preserving installed binaries under `/system/lib`.

Some targets cannot be built with ASan:

- Statically linked executables.
- `LOCAL_CLANG:=false` targets
- `LOCAL_SANITIZE:=undefined`; will not be ASan'd for `SANITIZE_TARGET=address`

Executables like these are skipped in the SANITIZE_TARGET build, and the version from the first make invocation is left in `/system/bin`.

Libraries like this are simply built without ASan. They can contain some ASan code anyway from the static libraries they depend upon.

Supporting documentation

AddressSanitizerOnAndroid (https://github.com/google/sanitizers/wiki/AddressSanitizerOnAndroid) public project site

AddressSanitizer and Chromium (https://www.chromium.org/developers/testing/addresssanitizer)

Other Google Sanitizers (https://github.com/google/sanitizers)

Last updated March 27, 2017.