New features in AddressSanitizer

LLVM developer meeting
Nov 7, 2013
Alexey Samsonov, Kostya Serebryany

Agenda

- AddressSanitizer (ASan): a quick reminder
- New features:
  - Initialization-order-fiasco
  - Stack-use-after-scope
  - Stack-use-after-return
  - Leaks
- ASan for Linux Kernel
- Misc: compile time, MPX, ASM, libs
- BOF at 2:00pm today

ASan: a quick reminder

- Dynamic testing tool, finds memory bugs
  - Buffer overflows, use-after-free
  - Found 5000+ bugs everywhere, including LLVM
- Compiler instrumentation + run-time library
- ~2x slowdown
- In LLVM since 3.1
- Siblings:
  - ThreadSanitizer (TSan): data races
  - MemorySanitizer (MSan): uses of uninitialized memory

ASan: a quick reminder (cont.)

- Every 8 bytes of application memory are associated with 1 byte of “shadow” memory
- Redzones are created around buffers; freed memory is put into quarantine
- Shadow of redzones and freed memory is “poisoned”
- On every memory access compiler-injected code checks if shadow is poisoned

ASan report example: heap-use-after-free

```
int main(int argc, char **argv) {
   int *array = new int[100];
   delete [] array;
   return array[argc];  // BOOM
}
% clang++ -O1 -g -fsanitize=address a.cc && ./a.out
==30226== ERROR: AddressSanitizer heap-use-after-free
READ of size 4 at 0x7faa07fce084 thread T0
   #0 0x40433c in main a.cc:4
0x7faa07fce084 is located 4 bytes inside of 400-byte
region
freed by thread T0 here:
   #0 0x4058fd in operator delete[](void*) _asan_rtl_
   #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
   #0 0x405579 in operator new[](unsigned long) _asan_rtl_
   #1 0x4042f3 in main a.cc:2
```

New Features

ASan report example: init-order-fiasco

```
// i1.cc
extern int B;
int A = B;
int main() {
  return A;
}
```

```
// i2.cc
#include <stdlib.h>
int B = atoi("123");
```

```
% clang -g -fsanitize=address i1.cc i2.cc
% ASAN_OPTIONS=check_initialization_order=1 ./a.out
```

```
==19504==ERROR: AddressSanitizer: initialization-order-fiasco
READ of size 4 at 0x000001aaff60 thread T0
    #0 0x414fa3 in __cxx_global_var_init  i1.cc:2
    #1 0x415015 in global constructors keyed to a  i1.cc:5

0x000001aaff60 is located 0 bytes inside
  of global variable 'B' from 'i2.cc' (0x1aaff60) of size 4
```

Detecting init-order-fiasco

- Frontend knows which globals are dynamically initialized

- Instrumented code registers globals

```
struct __asan_global {
  void *address;
  size_t size; <...>
  const char *module_name;
  bool has_dynamic_initializer;
}
// asan.module_ctor has the highest priority.
asan.module_ctor() { <...>
  __asan_register_globals(globals, n);
}
```

Detecting init-order-fiasco (cont.)

```
// All globals from the translation unit are
// initialized here.
_GLOBAL__I_a() {
    // Poison shadow memory for {uninitialized, all}
    // globals in another TUs.
    __asan_before_dynamic_init(module_name);
    __cxx_global_var_init1();
    <...>
    __cxx_global_var_initN();
    // Unpoison shadow memory for all the globals.
    __asan_after_dynamic_init();
}
```

Init-order fiasco detector modes

```
// Poison shadow memory for {uninitialized [1], all [2]}
// globals in another TUs.
__asan_before_dynamic_init(module_name);
```

[1] `ASAN_OPTIONS=check_initialization_order=true`

[2] `ASAN_OPTIONS=strict_init_order=true` (has false positives).

```
struct Foo {
  Foo() { if (!initialized) value = get_value(); }
  int get() { if (!initialized) value = get_value();
              return value; }
  int value;
  static bool initialized;
};
```

Init-order fiasco status

- Works on Linux.

- OFF by default :(May bark on globals with no-op constructors, user has to blacklist them.

- Still worth using:
  - Strict mode ON by default for Google code, hundreds of errors are fixed.
  - Good for large code bases, which are difficult to be made `-Wglobal-constructors`-clean.
  - Finds potentials errors (LTO).

ASan report example: stack-use-after-scope

```
int main() {
  int *p;
  { int x = 0; p = &x; }
  return *p;
}
% clang -g -fsanitize=address,use-after-scope a.cc ; ./a.
out

==15839==ERROR: AddressSanitizer: stack-use-after-scope
READ of size 4 at 0x7fffe06c20a0 thread T0
    #0 0x46103d in main a.cc:4

Address is located in stack of thread T0 at offset 160 in
frame
    #0 0x460daf in main a.cc:1

  This frame has 4 object(s):
    [96, 104) 'p'
    [160, 164) 'x' <== Memory access at offset 160 is
inside this variable
```

Detecting stack-use-after-scope

Use `llvm.lifetime` intrinsics to generate calls to ASan runtime:

```
llvm.lifetime.start(size, ptr) ->
__asan_unpoison_stack_memory(ptr, size)

llvm.lifetime.end(size, ptr) ->
__asan_poison_stack_memory(ptr, size)
```

Stack-use-after-scope status

- Still at a prototype stage.
- Clang doesn't yet emit llvm.lifetime intrinsics for temporaries:

```
const char *s = FunctionReturningStdString().c_str();
char c = s[0];  // BOOM.
```

- Need to optimize redundant calls to ASan runtime (static analysis).
- Stack-use-after-scope will be bundled with stack-use-after-return (discussed further).

ASan report example: stack-use-after-return

```
int *g;                         int main() {
void LeakLocal() {                LeakLocal();
  int local;                      return *g;
  g = &local;                   }
}
```

```
% clang -g -fsanitize=address a.cc
% ASAN_OPTIONS=detect_stack_use_after_return=1 ./a.out
```

```
==19177==ERROR: AddressSanitizer: stack-use-after-return
READ of size 4 at 0x7f473d0000a0 thread T0
    #0 0x461ccf in main   a.cc:8

Address is located in stack of thread T0 at offset 32 in frame
    #0 0x461a5f in LeakLocal()  a.cc:2
  This frame has 1 object(s):
    [32, 36) 'local' <== Memory access at offset 32
```

Stack-use-after-return instrumentation

```
// Function entry
char frame[N];
char *fake_frame = &frame[0];
if (__asan_option_detect_stack_uar)
  fake_frame = asan_stack_malloc(N, frame);
…
// Function exit
if (fake_frame != frame)
  asan_stack_free(fake_frame, N);
```

Stack-use-after-return allocator

```
char *asan_stack_malloc(
    size_t N, char *real_frame);
void asan_stack_free(
    char *fake_frame, size_t N);
```

- Fast thread-local malloc-like allocator
- Has quarantine for freed chunks
- Uses a fixed size mmap-ed buffer
- If allocation fails, returns the original frame

ASan report example: memory leak

```
int *g = new int;
int main() {
  g = 0; // Lost the pointer.
}

% clang -g -fsanitize=address a.cc
% ASAN_OPTIONS=detect_leaks=1 ./a.out

==19894==ERROR: AddressSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x44a3b1 in operator new(unsigned long)
    #1 0x414f66 in __cxx_global_var_init  leak.cc:1
```

LeakSanitizer (ASan's leak detector)

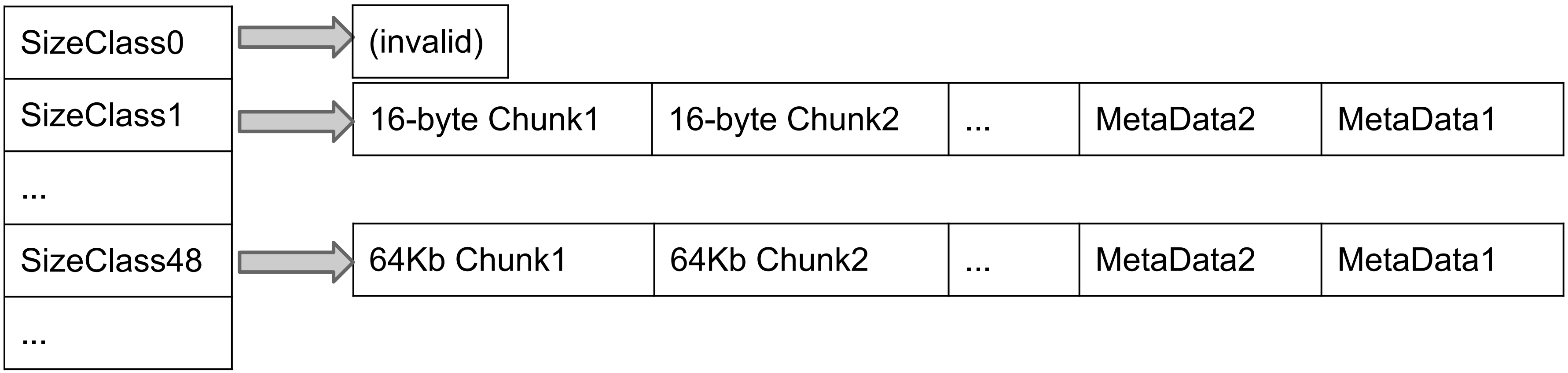
- Similar to other tools: tcmalloc, valgrind, etc
- Faster than any of those
  - No extra overhead on top of ASan at run-time
  - Small overhead at shutdown
- Based on the ASan/MSan/TSan allocator
- Can be bundled with ASan/MSan/TSan or used as a standalone tool
  - Currently, supported only in ASan or standalone
- Requires StopTheWorld() -- today Linux only

ASan/TSan/MSan allocator

- Full malloc/free API
  - thread-local caches, similar to tcmalloc

- Extra features for the tool:
  - Associate metadata with every heap chunk:
    - Stack trace of malloc/free
    - Other tool-specific metadata
  - ASan keeps metadata in the redzone
  - TSan/MSan: metadata is not adjacent to the chunk
  - Fast mapping “address => chunk => metadata”

ASan/TSan/MSan allocator (cont.)



- Memory is allocated from a fixed addr. range
  - ASan: [0x600000000000, 0x640000000000) -- 4Tb
- 64 regions; each allocates its own size class
  - Chunks are allocated left to right. Metadata: right to left.
- Fast “address=>chunk=>metadata”
  - Simple arithmetic
  - Lock-free

MISC

ASan for Linux Kernel

- … has nothing to do with LLVM :(
  - Our *early prototype* uses GCC's TSan module
  - Instrumentation is a bit different
  - Run-time is different (inside the kernel)
- Found 12 bugs already, 5 fixed!
- Want to use Clang for better instrumentation
  - Clang issues are resolved (?)
  - Still some issues in the Kernel code
- Want to test another kernel? Talk to us!

Intel MPX

- Intel MPX: Memory Protection Extensions
  - Published on July’13, HW available in ~ 2 years
  - Additional instructions to find buffer overflows
  - Expensive instructions touch two cache lines
  - Requires lots of memory
  - Slow for programs with graphs, lists and trees.
  - Does not detect use-after-free
  - Has false positives
  - *Biased* comparison against ASan: goo.gl/RrhZIz

- Still worth supporting in LLVM!
  - Finds intra-object buffer overflows
  - Very fast for long loops that traverse simple arrays

Compile time with ASan

- ASan and MSan create more control flow
- Some LLVM passes downstream explode
- Example: PR17409 (quadratic?)
    - llvm::SpillPlacement::addLinks
    - InlineSpiller::propagateSiblingValue

Why instrument all libs?

- ASan: stack unwinding with frame pointers
- TSan: catching synchronization via atomics
- MSan: avoid false positives
- All tools: more coverage

- Status: can build 50+ libs used by Chromium on Linux
- Help is welcome!

We also want to instrument ASM!

- MSan: avoid false positives
  - Ex.: FD_ZERO on Linux is inline asm
  - Ex.: optimized libraries (openssl, libjpeg_turbo)
- All tools: more coverage (same as libs)

Ideas

- Pattern matching for simple cases
- An MC Pass
- Use MCLayer

Summary

- ASan keeps getting new features
  - Initialization-order-fiasco: done (Linux)
  - Stack-use-after-scope: work-in-progress
  - Stack-use-after-return: beta
  - Memory Leaks: done (Linux)
- Lots of work to do
  - Libs, ASM, Kernel, MPX, compile time
  - Better support for non-Linux-x86_64