Address Sanitizer(ASan)

Introduction

AddressSanitizer 是一个快速的内存错误探测器。 它由一个编译检测模块和一个运行时库组成。它可以检测到一下类型的bug:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (to some extent)//未测试
- Double-free, invalid free
- Memory leaks (experimental)//未测试

Address Sanitizer重写了malloc和free,是一种非常高效的内存检查方式,使用Address Sanitizer后程序执行仅仅慢了两倍。

AddressSanitizer是一对工具：首先是编译扩展，然后是运行库。ASan的运行库会分配一个影子内存：一个超大块的内存，用于为每8字节内存记录一字节信息。默认情况下，所有内存的影子字节会被设置为0，表明它不能被访问。当内存被分配后，影子字节设置为其它值（用于记录字的哪些字节被分配、谁分配了它们），它还会重载分配器来跟踪内存的分配和释放。

这样，每次发生内存访问时，运行库将检查相应影子字节的值，如果访问是不被允许的，ASan就会中止程序的执行： ASan在第一个错误发生时崩溃程序，它强制程序必须是ASan检查通过的。

编译扩展的主要作用是将对内存的每一次访问封装在一个小分支中，通过检查影子内存的内容确认访问是否允许。不过因为是在编译器中进行处理，所以它可以访问大量的信息，比如正在访问什么内存，变量或结构体成员的布局是怎样的，…并且它还能可以改变这些。这正是ASan让人眼前一亮的地方：它可以在全局的变量间或栈上的变量间添加红色区域，使得对这些变量的错误访问更容易被检查到。

用法

使用-fsanitize=address 简单的编译链接你的程序。AdressSanitizer运行时库应当最后被被链接,所以确保在链接的最后一步使用clang(not ld)。当链接共享的库文件的时候，Address Sanitizer 运行时库不会被链接，因此,-wl,-z,defs可能会引起链接错误（不要和Address Sanitizer一起使用这些命令）。为了获取合理的性能,可以添加-o1或者更高的编译参数.

```
% clang -fsanitize=address  main.c
```

fno-omit-frame-pointer:可以获得更加详细的信息

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.

如果你的进程是在OSX 10.10及以后的沙盒环境中运行,你需要设置DYLD*INSERT*LIBRARIES环境变量,并将它指向和编译器打包在一起的ASan库。

符号化输出

为了让AddressSanitizer将其输出符号化为你需要的内容，你需要将 `ASAN_SYMBOLIZER_PATH` 环境变量设置为为llvm-symbolizer库(或者将llvm-symbolizer加到你的$PATH中)。

```
% ASAN_SYMBOLIZER_PATH=/usr/local/bin/llvm-symbolizer ./a.out
```

如果这个没有效果,你可以使用一个单独的脚本来离线的解析你的输出结果。(在线解析可以使用ASAN_OPTIONS=symbolize=0 来强制禁用)。

```
% ASAN_OPTIONS=symbolize=0 ./a.out 2> log
% projects/compiler-rt/lib/asan/scripts/asan_symbolize.py / < log | c++filt
```

Note:在OSX中你可能需要运行dsymutil。(Note that on OS X you may need to run dsymutil on your binary to have the file:line info in the AddressSanitizer reports.)

Example:

```
void MyFunction(int nSize);
int main(int argc, char * argv[]) {
    int *a = (int*)malloc(3 * sizeof(int));
    a [4] = 1;//Out-of-bounds accesses to heap, stack and globals
    free(a);
    a [6] = 1;//Use-after-free
    free(a);//Double-free, invalid free
}
```

```
jianghaideMac-mini:Desktop jianghai$ clang -fsanitize=address main.c
jianghaideMac-mini:Desktop jianghai$ ./a.out
=================================================================
==2616==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000eebc at pc 0x000108690ef2 bp 0x7fff5756fbd0 sp 0x7fff5756fbc8
WRITE of size 4 at 0x60200000eebc thread T0
    #0 0x108690ef1 in atos[2617]: [fatal] 'pid_for_task' failed: (os/kern) failure (5) (+0x100000ef1)
==2616==WARNING: Can't write to symbolizer at fd 3
    #1 0x7fff8bbbb5ac in atos[2620]: [fatal] 'pid_for_task' failed: (os/kern) failure (5) (+0x35ac)
    #2 0x0  (<unknown module>)

0x60200000eebc is located 0 bytes to the right of 12-byte region [0x60200000eeb0,0x60200000eebc)
allocated by thread T0 here:
==2616==WARNING: Can't write to symbolizer at fd 3
    #0 0x1086e19c0 in atos[2623]: [fatal] 'pid_for_task' failed: (os/kern) failure (5) (+0x489c0)
==2616==WARNING: Can't write to symbolizer at fd 3
    #1 0x108690e9d in atos[2626]: [fatal] 'pid_for_task' failed: (os/kern) failure (5) (+0x100000e9d)
==2616==WARNING: Can't read from symbolizer at fd 3
==2616==WARNING: Failed to use and restart external symbolizer!
    #2 0x7fff8bbbb5ac in start (/usr/lib/system/libdyld.dylib+0x35ac)
    #3 0x0  (<unknown module>)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/Users/jianghai/Desktop/./a.out+0x100000ef1) in main
Shadow bytes around the buggy address:
  0x1c0400001d80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400001d90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400001da0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400001db0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400001dc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x1c0400001dd0: fa fa fa fa fa fa 00[04]fa fa 00 06 fa fa 00 00
  0x1c0400001de0: fa fa 00 04 fa fa 00 00 fa fa 00 00 fa fa 00 fa
  0x1c0400001df0: fa fa fd fd fa fa fd fd fa fa fd fd fa fa fd fa
  0x1c0400001e00: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400001e10: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400001e20: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
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
  Heap right redzone:      fb
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack partial redzone:   f4
  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
```