链滴

AddressSanitizer VS Valgrind

作者：Hanseltu

原文链接：https://ld246.com/article/1557304039375

来源网站：链滴

原文链接 AddressSanitizer VS Valgrind

前言

C/C++等底层语言在提供强大功能及性能的同时，其灵活的内存访问也带来了各种纠结的问题。如果crash的地方正是内存使用错误的地方，说明你人品好。如果crash的地方内存明显不是consistent的，者内存管理信息都已被破坏，并且还是随机出现的，那就比较麻烦了。当然，裸看code打log是一个法，但其效率不是太高，尤其是在运行成本高或重现概率低的情况下。另外，静态检查也是一类方法有很多工具（lint, cppcheck, klockwork, splint, o, etc.）。但缺点是误报很多，不适合针对性问题另外好点的一般还要钱。最后，就是动态检查工具。下面介绍几个Linux平台下主要的运行时内存检工具。绝大多数都是开源免费且支持x86和ARM平台的。

首先，比较常见的内存问题有下面几种：

- memory overrun：写内存越界
- double free：同一块内存释放两次
- use after free：内存释放后使用
- wild free：释放内存的参数为非法值
- access uninitialized memory：访问未初始化内存
- read invalid memory：读取非法内存，本质上也属于内存越界
- memory leak：内存泄露
- use after return：caller访问一个指针，该指针指向callee的栈内内存
- stack overflow：栈溢出

针对上面的问题，主要有以下几种方法：

- （1） 为了检测内存非法使用，需要hook内存分配和操作函数。hook的方法可以是用C-preprocesor，也可以是在链接库中直接定义（因为Glibc中的malloc/free等函数都是weak symbol），或是用L_PRELOAD。另外，通过hook strcpy(),memmove()等函数可以检测它们是否引起buffer overflow。
- （2）为了检查内存的非法访问，需要对程序的内存进行bookkeeping，然后截获每次访存操作并测是否合法。bookkeeping的方法大同小异，主要思想是用shadow memory来验证某块内存的合法。至于instrumentation的方法各种各样。有run-time的，比如通过把程序运行在虚拟机中或是通过biary translator来运行；或是compile-time的，在编译时就在访存指令时就加入检查操作。另外也可通过在分配内存前后加设为不可访问的guard page，这样可以利用硬件(MMU)来触发SIGSEGV，从提高速度。
- （3）为了检测栈的问题，一般在stack上设置canary，即在函数调用时在栈上写magic number或随机值，然后在函数返回时检查是否被改写。另外可以通过mprotect()在stack的顶端设置guard pag，这样栈溢出会导致SIGSEGV而不至于破坏数据。

两个典型的内存检测工具

工具1： AddressSanitizer

AddressSanitizer是Google用于检测内存各种buffer overflow(Heap buffer overflow, Stack buffer overflow, Global buffer overflow)的一个非常有用的工具。该工具是一个LLVM的Pass，现已集成至lvm中，要是用它可以通过-fsanitizer=address选项使用它。AddressSanitizer的源码位于/lib/Transforms/Instrumentation/AddressSanitizer.cpp中，Runtime-library的源码在llvm的另一个项目compi

er-rt的/lib/asan文件夹中。

AddressSanitizer早期只能在Clang编译器下使用，后来加入到GCC中，GCC4.8及以上版本可使用此译选项。4.8版本GCC对AddressSanitizer支持有限，功能不太完善，输出的错误信息也不够友好，用不太方便，建议使用4.9及以上版本。

使用实例：

选取Google提供的一段小代码，来说明AddressSanitizer的用法。

编写源代码：use-after-free.c

```
#include <stdlib.h>
int main() {
  char *x = (char*)malloc(10 * sizeof(char*));
  free(x);
  return x[5];
}
```

编译源代码

```
clang -fsanitize=address -O1 -fno-omit-frame-pointer -g   use-after-free.c
```

运行程序可得以下输出结果

```
=================================
==7332==ERROR: AddressSanitizer: heap-use-after-free on address 0x607000000095 at pc 0
00010532cef4 bp 0x7ffeea8d38d0 sp 0x7ffeea8d38c8
READ of size 1 at 0x607000000095 thread T0
 # 0 0x10532cef3 in main use-after-free.c:5
 #1 0x7fff7c0a13d4 in start (libdyld.dylib:x86_64+0x163d4)
0x607000000095 is located 5 bytes inside of 80-byte region [0x607000000090,0x6070000000
0)
freed by thread T0 here:
 #0 0x10538c20d in wrap_free (libclang_rt.asan_osx_dynamic.dylib:x86_64h+0x5c20d)
 #1 0x10532ceba in main use-after-free.c:4
 #2 0x7fff7c0a13d4 in start (libdyld.dylib:x86_64+0x163d4~~~~
previously allocated by thread T0 here:
 #0 0x10538c053 in wrap_malloc (libclang_rt.asan_osx_dynamic.dylib:x86_64h+0x5c053)
 #1 0x10532ceaf in main use-after-free.c:3
 #2 0x7fff7c0a13d4 in start (libdyld.dylib:x86_64+0x163d4)
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 in main
Shadow bytes around the buggy address:
 0x1c0dffffffc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x1c0dffffffd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x1c0dffffffe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x1c0dfffffff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x1c0e00000000: fa  fa  fa  fa 00 00 00 00 00 00 00 00 03 fa  fa  fa
=>0x1c0e00000010: fa  fa[fd]fd  fd  fd  fd  fd  fd  fd  fd  fd  fa  fa  fa  fa
0x1c0e00000020: fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa
0x1c0e00000030: fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa
0x1c0e00000040: fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa
0x1c0e00000050: fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa  fa
0x1c0e00000060: 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
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
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
 Shadow gap: cc
==7332==ABORTING
Abort trap: 6
```

由提示 SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 in main可以知道main函数的第5行存在heap-use-after-free的内存错误。

工具2：Valgrind

Valgrind是一套Linux下，开放源代码（GPL V2）的仿真调试工具的集合。Valgrind由内核（core）及基于内核的其他调试工具组成。内核类似于一个框架（framework），它模拟了一个CPU环境，并供服务给其他工具；而其他工具则类似于插件 (plug-in)，利用内核提供的服务完成各种特定的内存试任务。

由于Valgrind没有和编译器集成，只能另行安装，安装方法就不说明，在此只简单说明其用法。

使用实例

为了使valgrind发现的错误更精确，如能够定位到源代码行，建议在编译时加上-g参数，编译优化选请选择O0，虽然这会降低程序的执行效率。

生成可执行程序，如

```
gcc -g -O0 test.c -o test
```

生成可执行程序test之后，使用以下操作让Valgrind来生成内存的记录文件：

```
valgrind --leak-check=full --log-file=test_valgrind.log --num-callers=30 ./test
```

- --log-file 后面的test_valgrind.log是指定生成的日志文件名称。
- --num-callers 后面的60是生成的每个错误记录的追踪行数。30是随便设定的，如果没指定，默认12行（有可能有的追踪行就没显示）。
- --leak-check=full 表示开启详细的内存泄露检测器。

编写test.cpp

```
#include <iostream>
using namespace std;
int main()
{
 int *a = new int[10];
 a[11] = 0;
 cout << a[11]<< endl;
 return 0;
}
```

编译程序

```
gcc –g –o test test.cpp
```

使用valgrind分析程序

```
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes test
```

输出结果如下

```
==2051== Memcheck, a memory error detector.
==2051== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==2051== Using LibVEX rev 1804, a library for dynamic binary translation.
==2051== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==2051== Using valgrind-3.3.0, a dynamic binary instrumentation framework.
==2051== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==2051== For more details, rerun with: -v
==2051==
==2051== Invalid write of size 4
==2051== at 0x4009C6: main (test.cpp:7)
==2051== Address 0x4a2005c is 4 bytes after a block of size 40 alloc'd
==2051== at 0x490581B: operator new[](unsigned long) (vg_replace_malloc.c:274)
==2051== by 0x4009B9: main (test.cpp:6)
==2051==
==2051== Invalid read of size 4
==2051== at 0x4009D4: main (test.cpp:8)
==2051== Address 0x4a2005c is 4 bytes after a block of size 40 alloc'd
==2051== at 0x490581B: operator new[](unsigned long) (vg_replace_malloc.c:274)
==2051== by 0x4009B9: main (test.cpp:6)
0
==2051==
==2051== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 9 from 4)
==2051== malloc/free: in use at exit: 40 bytes in 1 blocks.
==2051== malloc/free: 1 allocs, 0 frees, 40 bytes allocated.
==2051== For counts of detected errors, rerun with: -v
==2051== searching for pointers to 1 not-freed blocks.
==2051== checked 198,560 bytes.
==2051==
==2051==
==2051== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2051== at 0x490581B: operator new[](unsigned long) (vg_replace_malloc.c:274)
==2051== by 0x4009B9: main (test.cpp:6)
==2051==
==2051== LEAK SUMMARY:
==2051== definitely lost: 40 bytes in 1 blocks.
```

原文链接：AddressSanitizer VS Valgrind

```
==2051== possibly lost: 0 bytes in 0 blocks.
==2051== still reachable: 0 bytes in 0 blocks.
==2051== suppressed: 0 bytes in 0 blocks.
```

从 LEAK SUMMARY: 来看，确定有一块40byte的内存泄漏。

AddressSanitizer和Valgrind综合对比

关于AddressSanitizer和Valgrind的具体实现和源码分析在后续的文章中会详细说明，以下仅做功能的对比。

| --- | --- | --- | --- |

| | AddressSanitizer | Valgrind

| technology | CTI | DBI |

| ARCH| x86, ARM, PPC | x86, ARM, PPC, MIPS, S390X, TILEGX |

| OS| Linux, OS X, Windows, FreeBSD, Android, iOS Simulator| Linux OS X, Solaris, Android |

| Slowdown | **2x**| **20x** |

[Heap OOB](https://github.com/google/sanitizers/wiki/AddressSanitizerExampleHeapOutOfBounds)| yes| yes|
[Stack OOB](https://github.com/google/sanitizers/wiki/AddressSanitizerExampleStackOutOfBounds)	yes	no
[Global OOB](https://github.com/google/sanitizers/wiki/AddressSanitizerExampleGlobalOutOfBounds)	yes	no
[UAF](https://github.com/google/sanitizers/wiki/AddressSanitizerExampleUseAfterFree)	yes	yes
[UAR](https://github.com/google/sanitizers/wiki/AddressSanitizerExampleUseAfterReturn)	yes	no
UMR	no	yes
Leaks	yes	yes

各简写含义如下：

DBI: dynamic binary instrumentation（动态二进制插桩）

CTI: compile-time instrumentation （编译时插桩）

UMR: uninitialized memory reads （读取未初始化的内存）

UAF: use-after-free (aka dangling pointer) （使用释放后的内存）

UAR: use-after-return （使用返回后的值）

OOB: out-of-bounds （溢出）

x86: includes 32- and 64-bit.

另外还有一些其他内存检测工具对比和详细说明可参考AddressSanitizerComparisonOfMemoryTools

部分参考：Linux中的常用内存问题检测工具