关于 ASAN

gnutgcy
学会学习

1 ASAN 简介

ASAN（Address Sanitizer）是针对 C/C++ 的快速内存错误检测工具，在运行时检测 C/C++ 代码中的多种内存错误。

ASAN 早先是 LLVM 中的特性，后被集成到 GCC 4.8 中，在 4.9 版本中加入了对 ARM 平台的支持。

ASAN 目前支持的平台有 X86/X86_64/ARM/ARM64，对于 ARM64 平台，Android 官方推荐使用 HWASan （HWAddress Sanitizer）。

2 ASAN 使用

ASAN 特性需要编译器支持，对于采用 GCC 编译器的项目需要 GCC 4.8 以上的版本（最好 GCC 4.9 及以上的版本，GCC 4.8 版本对 ASAN 支持不完善）。

可通过在编译时设置 CFLAGS，指定相关的编译参数来使能 ASAN 特性。

对于 Android 系统，因为 wrap.sh 仅适用于 API 级别 27 及更高级别，所以从 API 级别 27（Android O MR 1）开始，Android NDK 可支持 ASAN。

ASAN 使用 shadow memory 跟踪哪些字节为正常内存，哪些字节为中毒内存。字节可以标记为完全正常（shadow memory 值为 0）、完全中毒（shadow memory 值为负值）或前面 k 个字节未中毒（shadow memory 值为 k）。如果 shadow memory 显示某个字节中毒，则 ASAN 会使程序崩溃，并输出有用的调试信息，包括调用堆栈、影子内存映射、内存违例类型、读取或写入的内容、导致违例的计算机以及内存内容。

2.1 GCC 编译选项

> -fsanitize=address：启用快速内存错误检测器 ASAN。 内存访问指令用于检测out-of-bounds 和 use-after-free 错误。该选项启用 -fsanitize-address-use-after-scope。有关详细信息，请参阅 <github.com/google/sanit...>。可以使用 ASAN_OPTIONS 环境变量影响运行时行为。当设置为 "help=1" 时，可用选项会在检测程序启动时显示。参阅<github.com/google/sanit...> 获取支持的选项列表。该选项不能与 -fsanitize=thread 和/或 -fcheck-pointer-bounds 结合使用。--param asan-globals=0 --param asan-stack=0 --param asan-instrument-reads=0 --param asan-instrument-writes=0 --param asan-memintrin=0 --param asan-use-after-return=0 --param asan-instrumentation-with-call-threshold=0 --param use-after-scope-direct-emission-threshold=256
> -fsanitize=kernel-address：为 Linux 内核启用 ASAN。有关详细信息，请参阅 <github.com/google/kasan...>。该选项不能与 -fcheck-pointer-bounds 结合使用。
> -fno-sanitize=all：此选项禁用所有以前启用的 sanitizers。 -fsanitize=all 是不允许的，因为某些 sanitizers 不能一起使用。
> -fasan-shadow-offset=number：此选项强制 GCC 在 AddressSanitizer 检查中使用自定义阴影偏移。它对于在 Kernel AddressSanitizer 中试验不同的 shadow memory 布局很有用。
> -fsanitize-sections=s1,s2,...：Sanitize 选定用户定义部分中的全局变量。si 可能包含通配符。
> -fsanitize-recover[=opts]：-fsanitize-recover= 控制以逗号分隔的 opts 列表中提到的 sanitizer 的错误恢复模式。为 sanitizer 组件启用此选项会使它尝试继续运行程序，就好像没有发生错误一样。这意味着可以在单个程序运行中报告多个运行时错误，即使已报告错误，程序的退出代码也可能指示程序运行成功。-fno-sanitize-recover= 选项可用于更改此行为：仅报告第一个检测到的错误，然后程序以非零退出代码退出。目前此功能仅适用于 -fsanitize=undefined（及其子选项，除了 -fsanitize=unreachable 和 -fsanitize=return）、-fsanitize=float-cast-overflow、-fsanitize=float-divide-by-zero、-fsanitize= bounds-strict、-fsanitize=kernel-address 和 -fsanitize=address。由于此功能是实验性的，所以除 -fsanitize=address 外，sanitizers 默认情况下会打开错误恢复。-fsanitize-recover=all 和 -fno-sanitize-recover=all 也被接受，前者启用所有支持它的 sanitizer 的恢复，后者禁用所有支持它的 sanitizer 的恢复。即使在编译器端开启了恢复模式，也需要在运行时库端开启，否则失败仍然是致命的。对于 TSAN 和 UBSAN，运行时库默认为 "halt_on_error=0"，而 ASAN 的默认值为 "halt_on_error=1"。这可以通过在相应的环境变量中设置 "halt_on_error"标志来覆盖。不推荐使用没有显式 opts 参数的语法。它相当于指定一个 opts 列表：undefined,float-cast-overflow,float-divide-by-zero,bounds-strict。
> -fsanitize-address-use-after-scope：启用局部变量的 sanitization 以检测 use-after-scope 错误。该选项将 -fstack-reuse 设置为 none。
> -fsanitize-undefined-trap-on-error：该选项指示编译器使用 "__builtin_trap" 而不是 "libubsan" 库例程来报告未定义的行为。这样做的好处是不需要 "libubsan" 库，也不需要链接，因此即使在独立环境中也可以使用。
> -fsanitize-coverage=trace-pc：启用覆盖率指导的模糊代码检测。在每个基本块中插入对 "__sanitizer_cov_trace_pc" 的调用。

2.2 运行时标志（run-time flags）：ASAN_OPTIONS

> https://github.com/google/sanitizers/wiki/AddressSanitizerFlags#run-time-flags

大多数运行时标志通过环境变量 ASAN_OPTIONS 传递给 ASAN，类似：

```
ASAN_OPTIONS=verbosity=1:help=1 ./a.out
```

也可以通过修改 GCC 的 __asan_default_options 函数在源代码中嵌入默认标志。

运行时标志支持参数可通过执行如下命令获取：

```
ASAN_OPTIONS=help=1 ./a.out
```

常用运行时标志：

> quarantine_size_mb：指定检测 use-after-free 错误的隔离区大小，较低的值可减少内存的使用，但会增加 use-after-free 错误的误报几率。
> redzone：堆对象周围红区的最小大小（以字节为单位）。 要求：redzone >= 16，是 2 的幂。
> max_redzone：堆对象周围红区的最大大小（以字节为单位）。
> log_path：将日志写入 "log_path.pid"。特殊值是 "stdout" 和 "stderr"。默认值为 "stderr"。
> log_exe_name：报告错误时提及可执行文件的名称并将可执行文件名称附加到日志（如 "log_path.exe_name.pid" 中）。
> log_to_syslog：除其他记录方式外，将所有 sanitizer 输出写入 syslog。
> Available flags for AddressSanitizer:
> quarantine_size
> - Deprecated, please use quarantine_size_mb.
> quarantine_size_mb
> - Size (in Mb) of quarantine used to detect use-after-free errors. Lower value may reduce memory usage but increase the chance of false negatives.
> redzone
> - Minimal size (in bytes) of redzones around heap objects. Requirement: redzone >= 16, is a power of two.
> max_redzone
> - Maximal size (in bytes) of redzones around heap objects.
> debug
> - If set, prints some debugging information and does additional checks.
> report_globals
> - Controls the way to handle globals (0 - don't detect buffer overflow on globals, 1 - detect buffer overflow, 2 - print data about registered globals).
> check_initialization_order
> - If set, attempts to catch initialization order issues.
> replace_str
> - If set, uses custom wrappers and replacements for libc string functions to find more errors.
> replace_intrin
> - If set, uses custom wrappers for memset/memcpy/memmove intinsics.
> mac_ignore_invalid_free
> - Ignore invalid free() calls to work around some bugs. Used on OS X only.
> detect_stack_use_after_return
> - Enables stack-use-after-return checking at run-time.
> min_uar_stack_size_log
> - Minimum fake stack size log.
> max_uar_stack_size_log
> - Maximum fake stack size log.
> uar_noreserve
> - Use mmap with 'noreserve' flag to allocate fake stack.
> max_malloc_fill_size
> - ASan allocator flag. max_malloc_fill_size is the maximal amount of bytes that will be filled with malloc_fill_byte on malloc.
> malloc_fill_byte
> - Value used to fill the newly allocated memory.
> allow_user_poisoning
> - If set, user may manually mark memory regions as poisoned or unpoisoned.
> sleep_before_dying

- Number of seconds to sleep between printing an error report and terminating the program. Useful for debugging purposes (e.g. when one needs to attach gdb).
check_malloc_usable_size
- Allows the users to work around the bug in Nvidia drivers prior to 295.*.
unmap_shadow_on_exit
- If set, explicitly unmaps the (huge) shadow at exit.
protect_shadow_gap
- If set, mprotect the shadow gap
print_stats
- Print various statistics after printing an error message or if atexit=1.
print_legend
- Print the legend for the shadow bytes.
atexit
- If set, prints ASan exit stats even after program terminates successfully.
print_full_thread_history
- If set, prints thread creation stacks for the threads involved in the report and their ancestors up to the main thread.
poison_heap
- Poison (or not) the heap memory on [de]allocation. Zero value is useful for benchmarking the allocator or instrumentator.
poison_partial
- If true, poison partially addressable 8-byte aligned words (default=true). This flag affects heap and global buffers, but not stack buffers.
poison_array_cookie
- Poison (or not) the array cookie after operator new[].
alloc_dealloc_mismatch
- Report errors on malloc/delete, new/free, new/delete[], etc.
new_delete_type_mismatch
- Report errors on mismatch betwen size of new and delete.
strict_init_order
- If true, assume that dynamic initializers can never access globals from other modules, even if the latter are already initialized.
start_deactivated
- If true, ASan tweaks a bunch of other flags (quarantine, redzone, heap poisoning) to reduce memory consumption as much as possible, and restores them to original values when the first instrumented module is loaded into the process. This is mainly intended to be used on Android.
detect_invalid_pointer_pairs
- If non-zero, try to detect operations like <, <=, >, >= and - on invalid pointer pairs (e.g. when pointers belong to different objects). The bigger the value the harder we try.
detect_container_overflow
- If true, honor the container overflow annotations. See code.google.com/p/addre...
detect_odr_violation
- If >=2, detect violation of One-Definition-Rule (ODR); If ==1, detect ODR-violation only if the two variables have different sizes
dump_instruction_bytes
- If true, dump 16 bytes starting at the instruction that caused SEGV
suppressions
- Suppressions file name.
halt_on_error
- Crash the program after printing the first error report (WARNING: USE AT YOUR OWN RISK!)
symbolize
- If set, use the online symbolizer from common sanitizer runtime to turn virtual addresses to file/line locations.
external_symbolizer_path
- Path to external symbolizer. If empty, the tool will search $PATH for the symbolizer.
allow_addr2line
- If set, allows online symbolizer to run addr2line binary to symbolize stack traces (addr2line will only be used if llvm-symbolizer binary is unavailable.
strip_path_prefix
- Strips this prefix from file paths in error reports.
fast_unwind_on_check
- If available, use the fast frame-pointer-based unwinder on internal CHECK failures.
fast_unwind_on_fatal
- If available, use the fast frame-pointer-based unwinder on fatal errors.

fast_unwind_on_malloc
- If available, use the fast frame-pointer-based unwinder on malloc/free.
handle_ioctl
- Intercept and handle ioctl requests.
malloc_context_size
- Max number of stack frames kept for each allocation/deallocation.
log_path
- Write logs to "log_path.pid". The special values are "stdout" and "stderr". The default is "stderr".
log_exe_name
- Mention name of executable when reporting error and append executable name to logs (as in "log_path.exe_name.pid").
log_to_syslog
- Write all sanitizer output to syslog in addition to other means of logging.
verbosity
- Verbosity level (0 - silent, 1 - a bit of output, 2+ - more output).
detect_leaks
- Enable memory leak detection.
leak_check_at_exit
- Invoke leak checking in an atexit handler. Has no effect if detect_leaks=false, or if __lsan_do_leak_check() is called before the handler has a chance to run.
allocator_may_return_null
- If false, the allocator will crash instead of returning 0 on out-of-memory.
print_summary
- If false, disable printing error summaries in addition to error reports.
check_printf
- Check printf arguments.
handle_segv
- If set, registers the tool's custom SIGSEGV/SIGBUS handler.
handle_abort
- If set, registers the tool's custom SIGABRT handler.
handle_sigfpe
- If set, registers the tool's custom SIGFPE handler.
allow_user_segv_handler
- If set, allows user to register a SEGV handler even if the tool registers one.
use_sigaltstack
- If set, uses alternate stack for signal handling.
detect_deadlocks
- If set, deadlock detection is enabled.
clear_shadow_mmap_threshold
- Large shadow regions are zero-filled using mmap(NORESERVE) instead of memset(). This is the threshold size in bytes.
color
- Colorize reports: (always|never|auto).
legacy_pthread_cond
- Enables support for dynamic libraries linked with libpthread 2.2.5.
intercept_tls_get_addr
- Intercept __tls_get_addr.
help
- Print the flag descriptions.
mmap_limit_mb
- Limit the amount of mmap-ed memory (excluding shadow) in Mb; not a user-facing flag, used mosly for testing the tools
hard_rss_limit_mb
- Hard RSS limit in Mb. If non-zero, a background thread is spawned at startup which periodically reads RSS and aborts the process if the limit is reached
soft_rss_limit_mb
- Soft RSS limit in Mb. If non-zero, a background thread is spawned at startup which periodically reads RSS. If the limit is reached all subsequent malloc/new calls will fail or return NULL (depending on the value of allocator_may_return_null) until the RSS goes below the soft limit. This limit does not affect memory allocations other than malloc/new.
can_use_proc_maps_statm
- If false, do not attempt to read /proc/maps/statm. Mostly useful for testing sanitizers.
coverage

- If set, coverage information will be dumped at program shutdown (if the coverage instrumentation was enabled at compile time).
coverage_pcs
- If set (and if 'coverage' is set too), the coverage information will be dumped as a set of PC offsets for every module.
coverage_order_pcs
- If true, the PCs will be dumped in the order they've appeared during the execution.
coverage_bitset
- If set (and if 'coverage' is set too), the coverage information will also be dumped as a bitset to a separate file.
coverage_counters
- If set (and if 'coverage' is set too), the bitmap that corresponds to coverage counters will be dumped.
coverage_direct
- If set, coverage information will be dumped directly to a memory mapped file. This way data is not lost even if the process is suddenly killed.
coverage_dir
- Target directory for coverage dumps. Defaults to the current directory.
full_address_space
- Sanitize complete address space; by default kernel area on 32-bit platforms will not be sanitized
print_suppressions
- Print matched suppressions at exit.
disable_coredump
- Disable core dumping. By default, disable_core=1 on 64-bit to avoid dumping a 16T+ core file. Ignored on OSes that don't dump core bydefault and for sanitizers that don't reserve lots of virtual memory.
use_madv_dontdump
- If set, instructs kernel to not store the (huge) shadow in core file.
symbolize_inline_frames
- Print inlined frames in stacktraces. Defaults to true.
symbolize_vs_style
- Print file locations in Visual Studio style (e.g: file(10,42): ...
stack_trace_format
- Format string used to render stack frames. See sanitizer_stacktrace_printer.h for the format description. Use DEFAULT to get default format.
no_huge_pages_for_shadow
- If true, the shadow is not allowed to use huge pages.
strict_string_checks
- If set check that string arguments are properly null-terminated
intercept_strstr
- If set, uses custom wrappers for strstr and strcasestr functions to find more errors.
intercept_strspn
- If set, uses custom wrappers for strspn and strcspn function to find more errors.
intercept_strpbrk
- If set, uses custom wrappers for strpbrk function to find more errors.
intercept_memcmp
- If set, uses custom wrappers for memcmp function to find more errors.
strict_memcmp
- If true, assume that memcmp(p1, p2, n) always reads n bytes before comparing p1 and p2.
decorate_proc_maps
- If set, decorate sanitizer mappings in /proc/self/maps with user-readable names
exitcode
- Override the program exit status if the tool found an error
abort_on_error
- If set, the tool calls abort() instead of _exit() after printing the error report.
include
- read more options from the given file
include_if_exists
- read more options from the given file (if it exists)

3 ASAN 工具主要组成部分

3.1 编译时插桩模块

使用 ASAN 相关编译参数进行编译，代码的所有内存访问（读写）操作会被编译器修改，访问内存前，判断即将被访问的内存是否为中毒状态，如果为中毒状态则输出报错信息，否则正常访问内存。

对于读操作

```
... = *address;
```

替换为

```
if (IsPoisoned(address)) {
    ReportError(address, kAccessSize, kIsWrite);
}
... = *address;
```

对于写操作

```
*address = ...;
```

替换为

```
if (IsPoisoned(address)) {
    ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```

函数 IsPoisoned 和函数 ReportError 的实现需要注重代码执行效率和对编译目标的大小影响。

编译时会被插桩的函数可通过下载对应版本的GCC代码，查看函数 InitializeAsanInterceptors 获得。

```
memmove
memset
memcpy
strcat
strchr
strcpy
strlen
wcslen
strncat
strncpy
strdup
strnlen
index
atoi
atol
strtol
atoll
strtoll
longjmp
sigaction
bsd_signal
signal
swapcontext
_longjmp
siglongjmp
__cxa_throw
pthread_create
pthread_join
```

```
__cxa_atexit
fork
```

3.2 运行时库

运行时库（libasan.so）接管函数 malloc 和函数 free，接管对函数 malloc 分配的内存前后产生影响，其前后被 ASAN 所用，称为红区（redzone），redzone 被标记为中毒状态，在使用函数 free 释放内存时，所释放的内存被隔离开来（暂时不会被分配出去），并被标记为中毒状态。

4 ASAN 支持检测哪些内存问题

1. use-after-free
2. use-after-return（无效的栈上内存，运行时标记 ASAN_OPTIONS=detect_stack_use_after_return=1）
3. use-after-scope（作用域外访问，clang标记-fsanitize-address-use-after-scope）Stack use outside scope：在某个局部变量的作用域之外，使用其指针。
4. heap-buffer-overflow/heap-buffer-underflow
5. stack-buffer-overflow/stack-buffer-underflow
6. global-buffer-overflow/global-buffer-underflow
7. Memory leaks（检测内存泄漏的 LeakSanitizer 是集成在 AddressSanitizer 中的一个相对独立的工具，它工作在检查过程的最后阶段。不是所有的平台都默认检测内存泄露，可以指定通过环境变量 ASAN_OPTIONS 开启：ASAN_OPTIONS=detect_leaks=1 yourapp，不是所有的平台都支持检测内存泄露，比如 ARM：==1901==AddressSanitizer: detect_leaks is not supported on this platform.）
8. double-free
9. Initialization order bugs

5 ASAN 防护缓冲区溢出的基本步骤

1. 在被保护的全局变量、堆、栈前后创建 redzone，并将 redzone 标记为中毒状态。
2. 将缓冲区和 redzone 每 8 字节对应 1 字节的映射方式建立影子内存区（影子内存区使用函数 MemToShadow 获取）。
3. 出现对 redzone 的访问（读写执行）行为时，由于 redzone 对应的影子内存区被标记为中毒状态触发报错。
4. 报错信息包含发生错误的进程号、错误类型、出错的源文件名、行号、函数调用关系、影子内存状态。其中影子内存状态信息中出错的部分用中括号标识出来。
5. 中毒状态：内存对应的 shadow 区标记该内存不能访问的状态。

6 ASAN 内存泄漏检测原理

1. ASAN 接管内存申请接口（用户使用的内存全部由 ASAN 管理）。
2. 在进程退出时触发 ASAN 内存泄漏检测（可通过复位、重启等可使进程正常退出的方法触发 ASAN 对内存泄漏的检测）。
3. 触发 ASAN 内存泄漏检测后，ASAN 遍历当前所有已分配给用户但没有释放的堆内存，扫描这些内存是否被某个指针（可能是全局变量、局部变量或者是堆内存里面的指针）引用，没有被引用则表示发生了内存泄漏。
4. 输出所有泄漏的内存信息，包含内存大小和内存申请的调用栈信息等。

7 ASAN 内存泄漏误报

1. 结构体非 4 字节对齐：报错提示结构体 A 内存泄漏，A 内存的指针存放在结构体 B 中，A 内存指针在结构体 B 中的偏移量非 4 的整数倍，由于 ASAN 扫描内存时是按照 4 字节偏移进行，从而扫描不到 A 内存指针导致误报。解决方法：对非4字节对齐的结构体进行整改。
2. 信号栈内存：该内存是在信号处理函数执行时做栈内存用的，其指针会保存在内核中，所以在用户态的 ASAN 扫描不到，产生误报；
3. 内存指针偏移后保存：
4. 存在ASAN未监控的内存接口：
5. 越界太离谱，越界访问的地址不在 buffer 的 redzone 内：
6. 对于memcpy的dest和src是在同一个malloc的内存块中时，内存重叠的情况无法检测到。
7. ASAN对于overflow的检测依赖于安全区，而安全区总归是有大小的。它可能是64bytes，128bytes或者其他什么值，但不管怎么样终归是有限的。如果某次踩踏跨过了安全区，踩踏到另一片可寻址的内存区域，ASAN同样不会报错。这是ASAN的另一种漏检。
8. ASAN对于UseAfterFree的检测依赖于隔离区，而隔离时间是非永久的。也就意味着已经free的区域过一段时间后又会重新被分配给其他人。当它被重新分配给其他人后，原先的持有者再次访问此块区域将不会报错。因为这一块区域的shadow memory不再是0xfd。所以这算是ASAN漏检的一种情况。

8 实践经验

1. 项目的构建方案应当有编译选项可以随时启用/关闭ASAN
2. 项目送测阶段可以打开ASAN以帮助暴露更多的低概率诡异问题
3. 请勿在生产版本中启用ASAN, 其会降低程序运行速度大概2-5倍 (特殊情况除外)
4. 实际开发测试过程中通过ASAN扫出的常见问题有：多线程下临界资源未加保护导致同时出现读写访问, 解决方案一般是对该资源恰当地加锁即可；内存越界, 如申请了N字节的内存却向其内存地址拷贝大于N字节的数据, 这种情况在没有开启ASAN的情况下一般都很难发现。
5. 一些显而易见的访问无效内存操作可能会被编译器优化而会漏报。

9 性能影响

1. ASan 支持 arm 和 x86 平台，使用 ASan 时，APP 性能会变慢且内存占用会飙升。
2. CPU 开销约为 2 倍，代码大小开销在 50% 到 2 倍之间，内存开销很大，约为 2 倍，具体取决于分配模式。

10 参考链接

> https://developer.android.com/ndk/guides/asan
> https://developer.android.com/ndk/guides/hwasan
> https://github.com/google/sanitizers/wiki/AddressSanitizer
> https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm
> https://clang.llvm.org/docs/AddressSanitizer.html
> https://clang.llvm.org/docs/SanitizerCoverage.html#tracing-pcs-with-guards

编辑于 2022-05-18 16:59

Linux　debug小技巧　内存（RAM）