Sanitizer 在字节跳动 C++ 业务中的实践

王留帅、徐明杰
字节跳动 STE 团队

ByteDance 字节跳动

目录

C++ @ ByteDance

Common Bugs In C++ Programming:

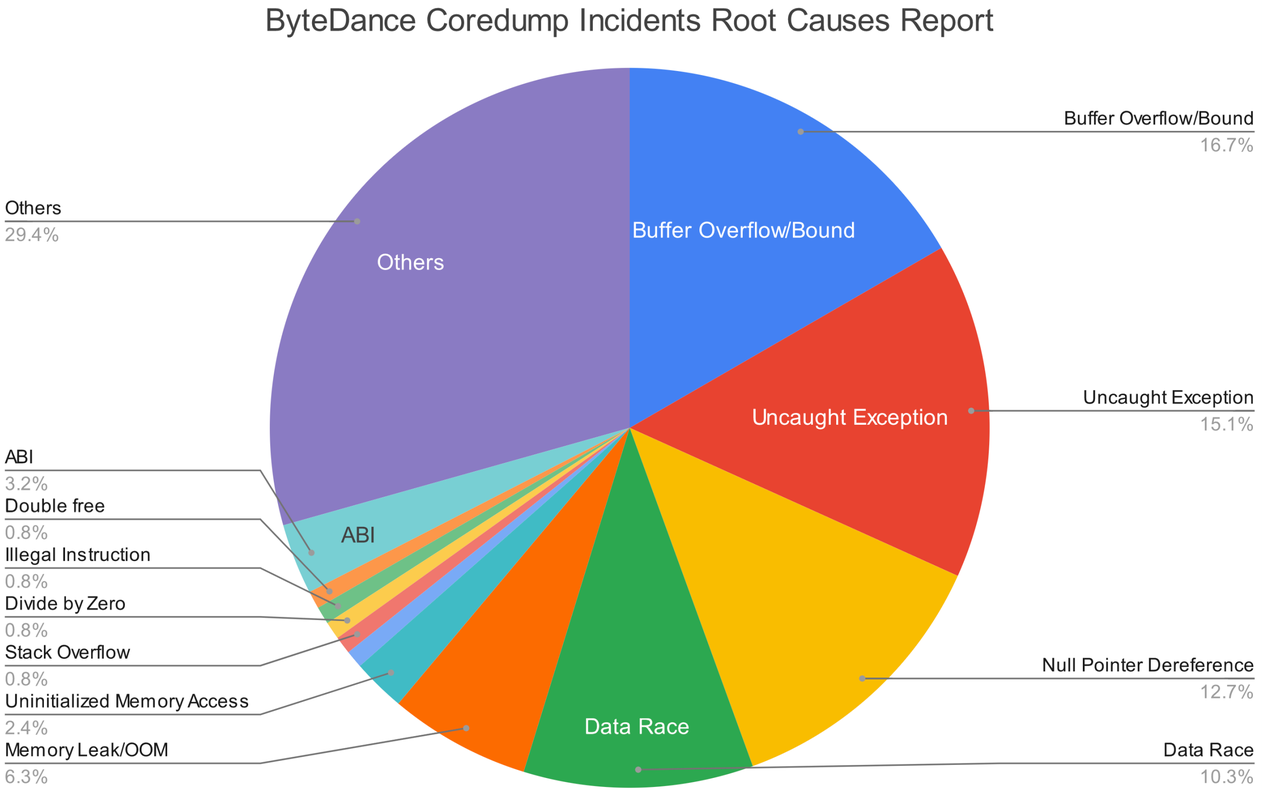
- Memory safety bugs
  - Spatial errors
  - Temporal errors
- Thread safety bugs
  - Deadlock
  - Data race
- Memory leak
- Use of uninitialized memory
- ODR violation
- Undefined Behaviors
- …

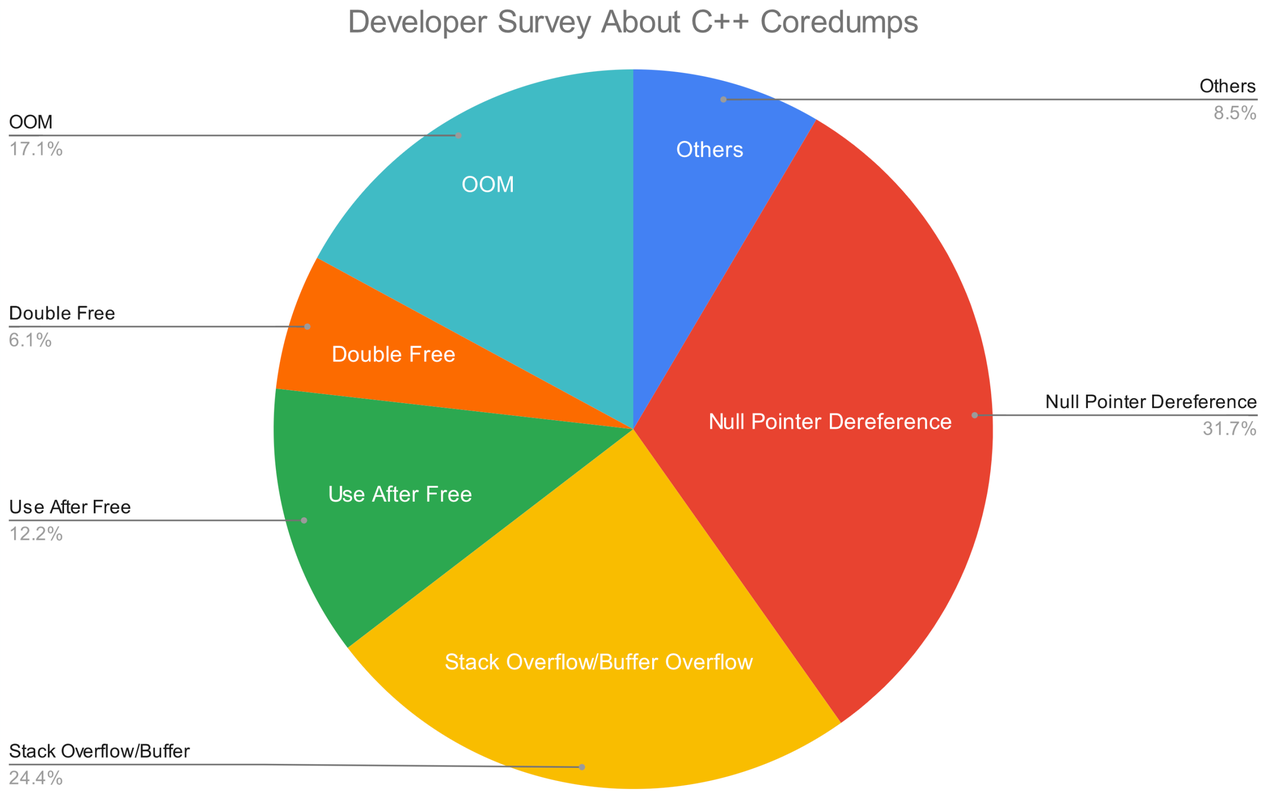
C++ @ ByteDance

Safer C++ is a dream that many people share:

- Safer Usage Of C++.
- [RFC] Lifetime annotations for C++.
- [RFC] C++ Buffer Hardening.
- Enumerating Core Undefined Behavior.

C++ @ ByteDance





For C++ programs, more than half of the incidents are caused by memory errors.

目录

Introduction To Sanitizer

Sanitizers: open-source tools for dynamic code analysis

- AddressSanitizer: a memory error detector
- LeakSanitizer: a memory leak detector (can be combined with AddressSanitizer)
- MemorySanitizer: a detector of uninitialized reads
- ThreadSanitizer: a data race and deadlock detector
- UndefinedBehaviorSanitizer: a undefined behavior detector

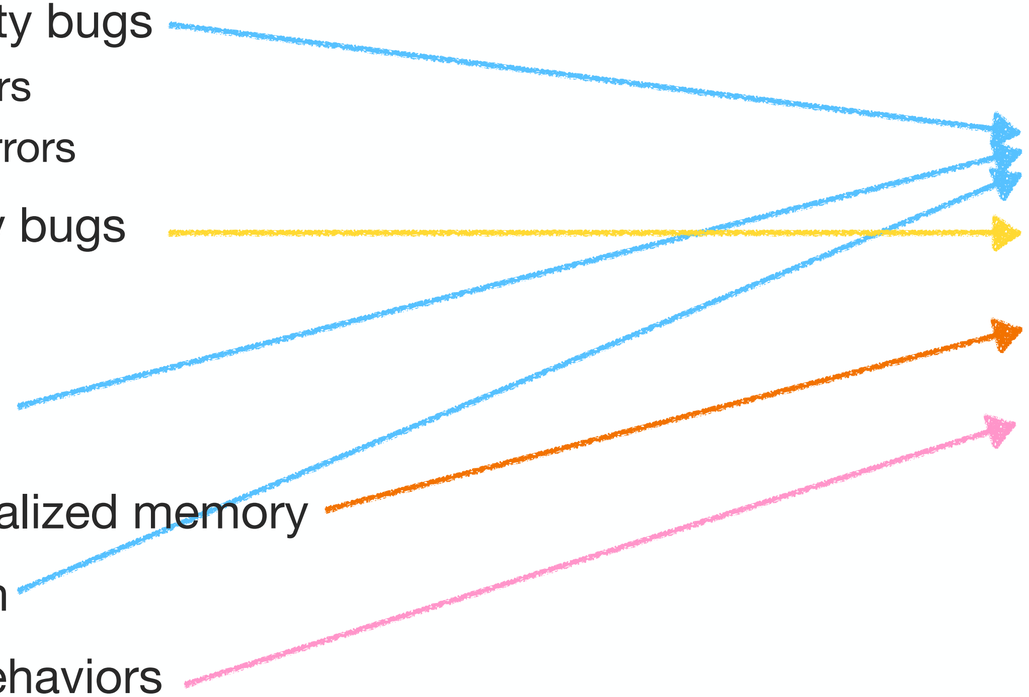
```
% cat example_UseAfterFree.cpp
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}

# Compile and link
% clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer example_UseAfterFree.cpp
```

Introduction To Sanitizer

Common Bugs In C++ Programming:

- Memory safety bugs
  - Spatial errors
  - Temporal errors
- Thread safety bugs
  - Deadlock
  - Data race
- Memory leak
- Use of uninitialized memory
- ODR violation
- Undefined Behaviors
- …



Sanitizers

- AddressSanitizer (ASan)
- ThreadSanitizer (TSan)
- MemorySanitizer (MSan)
- UndefinedBehaviorSanitizer (UBSan)

Introduction To Sanitizer

	AddressSanitizer	**Valgrind/Memcheck**	Dr. Memory	Mudflap	Guard Page	gperftools
technology	CTI	DBI	DBI	CTI	Library	Library
ARCH	x86, ARM, PPC	x86, ARM, PPC, MIPS, S390X, TILEGX	x86	all(?)	all(?)	all(?)
OS	Linux, OS X, Windows, FreeBSD, Android, iOS	Linux, OS X, Solaris, Android	Windows, Linux	Linux, Mac(?)	All (1)	Linux, Windows
Slowdown	**2x**	20x	10x	2x-40x	?	?
Detects:						
Heap OOB	**yes**	yes	yes	yes	some	some
Stack OOB	**yes**	no	no	some	no	no
Global OOB	**yes**	no	no	?	no	no
UAF	**yes**	yes	yes	yes	yes	yes
UAR	**yes** (see AddressSanitizerUseAfterReturn)	no	no	no	no	no
UMR	no (see MemorySanitizer)	yes	yes	?	no	no
Leaks	**yes** (see LeakSanitizer)	yes	yes	?	no	yes

1. https://github.com/google/sanitizers/wiki/AddressSanitizerComparisonOfMemoryTools
2. https://developers.redhat.com/blog/2021/05/05/memory-error-checking-in-c-and-c-comparing-sanitizers-and-valgrind

Introduction To Sanitizer

Sanitizer Internals

- Shadow memory
- Compile-time instrumentation
- Run-time library

Introduction To Sanitizer

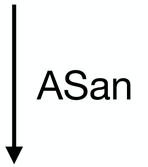
Sanitizer Internals (ASan)

- **Shadow memory**
  - maps 8 bytes of the application memory into 1 byte of the shadow memory.

- **Compile-time instrumentation**
  - instruments all loads/stores
  - inserts redzones around Stack and Global Variables

- **Run-time library**
  - malloc replacement (redzones, quarantine)
  - Bookkeeping for error messages

```
int64_t *addr;
*addr; // 8-byte access
```




```
char *shadow = MemToShadow(addr);
if (*shadow != 0)
  ReportError(addr);
*addr; // 8-byte access
```

目录

Sanitizer @ ByteDance — Observation

Memory safety bugs found

- **Temporal errors (~70%).** accesses memory when that memory is not valid at the time of the access.
- **Spatial errors (~30%).** memory access occurs out-of-bounds of a known object.

According to the sanitizer reports in ByteDance, **temporal errors** account for 59% !

```
std::unordered_map<int64_t, int64_t>
get_int64_int64_map_std_unordered(std::string name);

void foo(int64_t id) {
  auto it = get_int64_int64_map_std_unordered("some_map").find(id);
  bar(it->second);
}
```

Sanitizer @ ByteDance — Observation

Thread safety bugs found

- **Data Race on STL containers.**
- **…**

According to our experience with core-dump root cause analysis in ByteDance:
the complex core-dump/crash occurred in the production environment more or less related to **data race** !

```
void thread1() {
  vec.push_back(x);
}

void thread2() {
  vec.push_back(y);
}
```

Sanitizer @ ByteDance — Observation

Memory leak occurs when heap-allocated objects are not freed at appropriate time. It is manifested in two forms:

1. **Unreachable leak**, in which an allocated object is no longer reachable from the root objects such as global and stack variables.
2. **Forgotten leak**, in which an allocated object is still reachable but no longer accessed.

LeakSanitizer

- Atexit(DoLeakCheck)
- Unreachable leak 🙂
- Forgotten leak 😵

Sanitizer @ ByteDance — Deployment Status

Solution, Not Tools

Issues:

- *-fsanitize=address*, *-static-libasan*, *-shared-libasan*. *Oh My !*
- "ASan runtime does not come first in initial library list; you should either link runtime to your application or manually preload it with ***LD_PRELOAD***"
- "You are trying to dlopen a xxx.so shared library with ***RTLD_DEEPBIND*** flag which is incompatible with sanitizer runtime *(see https://github.com/google/sanitizers/issues/611 for details)*"
- "WARNING: ASan is ignoring requested ***__asan_handle_no_return***: False positive error reports may follow *(see https://github.com/google/sanitizers/issues/189 for details)*"
- "Your application is linked against incompatible ASan runtimes."
- "AddressSanitizer CHECK failed"
- ...

Sanitizer @ ByteDance — Deployment Status

Sanitizers need instrument all libs

- **ASan**: works even if you rebuild just part of your program, but have to rebuild all components to detect all errors.
- **TSan**: catching synchronization via atomics.
- **MSan**: avoid false positives. In particular, need MSan-instrumented C++ standard library.
- **All Tools**: work properly and not produce any false positives

Sanitizer @ ByteDance — Deployment Status

Tests Can Not Catch Everything

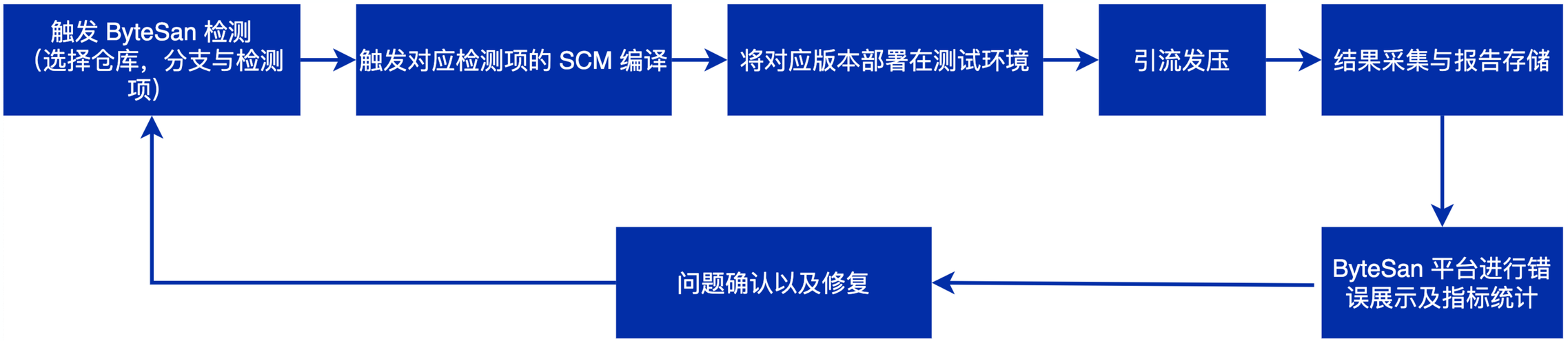
We should deploy sanitizers to the ***production environment !***

ASan is highly effective and one of the lowest overhead instrumentations available that detects the errors that it does; however, it still incurs an average **2-3x** performance and memory overhead.
Near-impossible for production environment.

GWP-ASan: **Sampling** heap memory error detection for ***production environment !***

Sanitizer @ ByteDance — Deployment Status

Solution, Not Tools → **ByteSan**



Sanitizer @ ByteDance — Improvement

ASan excessive stack usage

- Sanitizer CHECK failed

  `llvm-project/compiler-rt/lib/sanitizer_common/sanitizer_stacktrace.cpp:51 ((stack_top)) > ((stack_bottom)) (291338927, 291339221)`

- AddressSanitizer CHECK failed

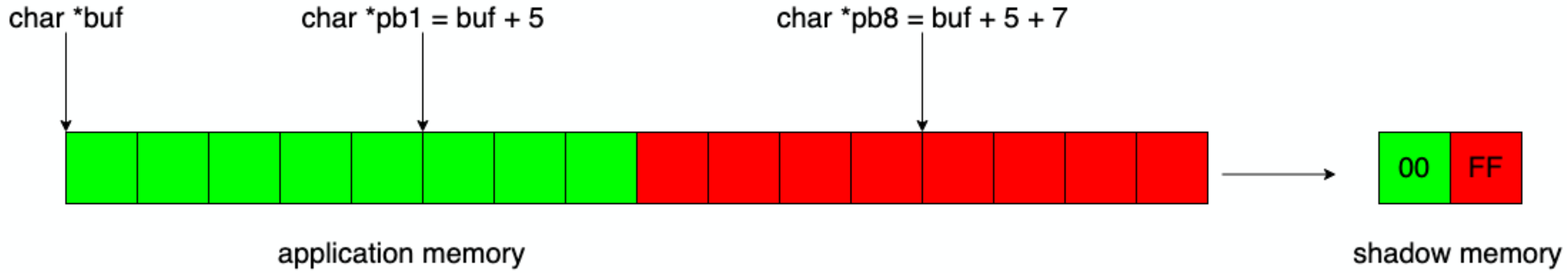
  `gcc/libsanitizer/sanitizer_common/sanitizer_allocator_local_cache.h:55 "((c->count)) > ((0))" (0x0, 0x0)`

```
$ ASAN_OPTIONS="help=1" ./a.out 2>&1 >/dev/null | grep adjusted_thread_stack_size
  adjusted_thread_stack_size
    - Experimental flag (WARNING: USE AT YOUR OWN RISK!). This is the thread stack size in bytes. If non-zero,
 the thread stack size will be set to adjusted_thread_stack_size, asan-only (Current Value: 0x0)
```

Sanitizer @ ByteDance — Improvement

ASan unaligned partially out-of-bound accesses

```
int main(int argc, char **argv) {
  char buf[8]; // 8 bytes: [0, 7]
  long long *p = (long long*)(buf + 5);
  *p = 1; // Access to range [5-12], BOOM
  return 0;
}
```



MemToShadow(pb1) = (pb1 >> SHADOW_SCALE) + SHADOW_OFFSET
= ((buf + 5) >> 3) + SHADOW_OFFSET
= (buf >> 3) + SHADOW_OFFSET
= MemToShadow(buf)

MemToShadow(pb8) = (pb8 >> SHADOW_SCALE) + SHADOW_OFFSET
= ((buf + 5 + 7) >> 3) + SHADOW_OFFSET
= (buf >> 3) + SHADOW_OFFSET + 1
= MemToShadow(buf) + 1

Sanitizer @ ByteDance — Improvement

TSan [failed to restore the stack]

- WARNING: ThreadSanitizer: data race (pid=1415044)
  - Read of size 4 at 0x7b10002ccc40 by thread T318:
    #0 memcpy sanitizer_common_interceptors.inc:808:5
  - Previous write of size 8 at 0x7b2c000000e0 by main thread:
    [failed to restore the stack]

```
$ TSAN_OPTIONS="help=1" ./a.out 2>&1 >/dev/null | grep  history_size
 history_size
  - Per-thread history size, controls how many previous memory accesses are remembered per thread.
  Possible values are [0..7]. history_size=0 amounts to 16K memory accesses.  Each next value doub
les the amount of memory accesses, up to history_size=7 that amounts to 2M memory accesses.  The d
efault value is 3 (128K memory accesses). (Current Value: 3)
```

```
$ TSAN_OPTIONS="help=1" ./a.out 2>&1 >/dev/null | grep  history_size
 history_size
  - Per-thread history size, controls how many previous memory accesses are remembered per thread.
Possible values are [0..8]. history_size=0 amounts to 16K memory accesses.  Each next value doubles
the amount of memory accesses, up to history_size=8 that amounts to 4M memory accesses.  The default
 value is 3 (128K memory accesses). (Current Value: 3)
```


https://groups.google.com/g/thread-sanitizer/c/VzSWE7UxhIE

Sanitizer @ ByteDance — Results

ByteSan detected bugs

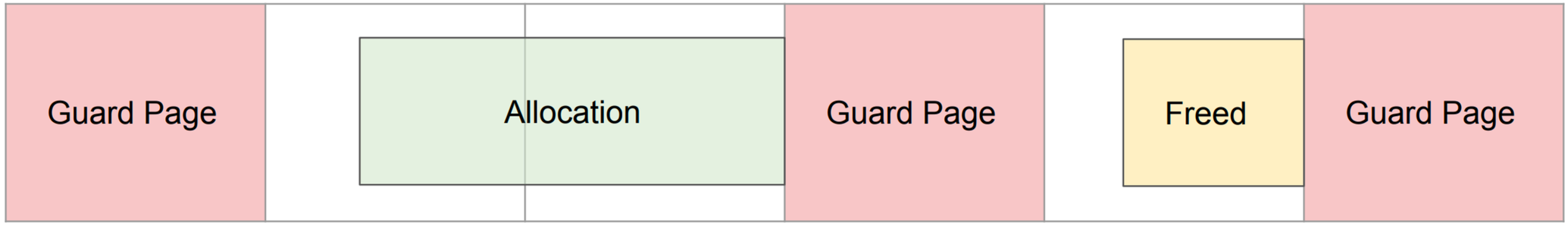
Bug Type	Count
Heap Use After Free	75 (50 has been fixed)
Double Free	19 (14 has been fixed)
Heap Buffer Overflow	38 (25 has been fixed)
Stack Overflow	6 (all fixed)
Global Buffer Overflow	2 (all fixed)
Stack Use After Scope	8 (7 has been fixed)
Null Ptr Dereference	19 (13 has been fixed)
Stack Buffer Overflow	2 (all fixed)
Alloc Dealloc Mismatch	1 (all fixed)
ABI	1 (all fixed)
Div By Zero	1 (all fixed)

- **Temporal errors account for 59%**
- Spatial errors account for 24%
- Double free errors mainly caused by data race
- ASan has certain stack overhead, more likely lead to stack overflow
- **Most problems are caused by STL container misuse, such as iterator invalidation and concurrency issue**
- **ByteSan** running on **~60** C++ server-side applications, including search, recommendation, and online advertising

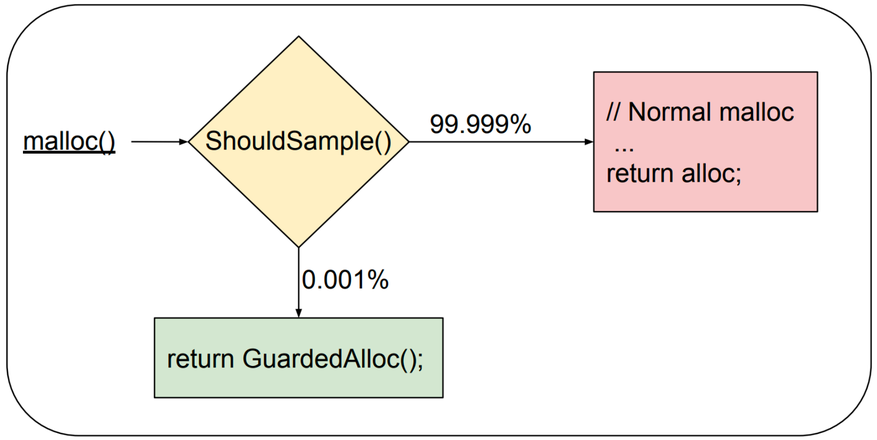
Sanitizer @ ByteDance — Next

GWP-ASan: Sampling heap memory error detection for production environment

- Detects heap-buffer-overflows using guard pages.
- Detects use-after-frees by mprotect-ing freed memory.



- Randomly guard a tiny fraction of allocations (e.g. 1/100,000).



Sanitizer @ ByteDance — Next

GWP-ASan: Sampling heap memory error detection for production environment

GWP-ASan implementations:

- TCMalloc ✅ https://google.github.io/tcmalloc/gwp-asan.html
- Chromium ✅ https://chromium.googlesource.com/chromium/src/+/lkgr/docs/gwp_asan.md
- LLVM-project ✅ https://llvm.org/docs/GwpAsan.html
- Android ✅ https://developer.android.com/ndk/guides/gwp-asan
- Jemalloc ❌

Implment GWP-ASan on jemalloc:

```
// cat uaf.cpp
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

```
$ clang++ uaf.cpp -L`~/jemalloc/output/bin/jemalloc-config --libdir` -Wl,-rpath,`~/jemalloc/output
/bin/jemalloc-config --libdir` -ljemalloc `~/jemalloc/output/bin/jemalloc-config --libs`
$ GWP_ASAN_OPTIONS='SampleRate=1' ./a.out
*** GWP-ASan detected a memory error ***
Use After Free at 0x7fb3fd77fe74 (4 bytes into a 400-byte allocation at 0x7fb3fd77fe70) by thread
2206616 here:
  #0 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x774b9) [0x7fb3fe9d84b9]
  #1 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x77920) [0x7fb3fe9d8920]
  #2 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x77dc4) [0x7fb3fe9d8dc4]
  #3 /lib/x86_64-linux-gnu/libpthread.so.0(+0x12730) [0x7fb3fe613730]
  #4 ./a.out(+0x1197) [0x56399a214197]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7fb3fe46209b]
  #6 ./a.out(+0x108a) [0x56399a21408a]

0x7fb3fd77fe74 was deallocated by thread 2206616 here:
  #0 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x774c9) [0x7fb3fe9d84c9]
  #1 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x7833c) [0x7fb3fe9d933c]
  #2 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x78df2) [0x7fb3fe9d9df2]
  #3 ./a.out(+0x118f) [0x56399a21418f]
  #4 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7fb3fe46209b]
  #5 ./a.out(+0x108a) [0x56399a21408a]

0x7fb3fd77fe74 was allocated by thread 2206616 here:
  #0 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x774c9) [0x7fb3fe9d84c9]
  #1 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x7833c) [0x7fb3fe9d933c]
  #2 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(+0x78aa8) [0x7fb3fe9d9aa8]
  #3 /data00/home/xumingjie.enna1/jemalloc/output/lib/libjemalloc.so.2(_Znam+0x89) [0x7fb3fe9d81f9
]
  #4 ./a.out(+0x1170) [0x56399a214170]
  #5 /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xeb) [0x7fb3fe46209b]
  #6 ./a.out(+0x108a) [0x56399a21408a]

*** End GWP-ASan report ***
```

Sanitizer @ ByteDance — Next

Sanitizer & BOLT

- **Sanitizers** primarily rely on the compiler for instrumentation, limiting their visibility into **assembly** and **pre-compiled** third-party code. Loads and stores missed during instrumentation can lead to **false positives** and **false negatives** in the tool output.
- **BOLT** is a **post-link optimizer** developed to speed up large applications. It achieves the improvements by optimizing application's code layout based on execution profile gathered by sampling profiler, such as Linux perf tool.
- **BOLT** can **add missing instrumentation** and provide a better experience running sanitizers. Prototype: https://reviews.llvm.org/D129225

目录

Summary

- Has deployed **ASan** for ~60 C++ server-side applications in test environment
- **TSan**/**MSan** not widely deployed due to pre-compiled library missing sanitizer instrumentation
- Working on **jemalloc** based **GWP-ASan**, sampling-based heap memory error detector for production environment
- Will try **BOLT** to add missing **sanitizer** instrumentation for pre-compiled library

THANKS

wangliushuai@bytedance.com
xumingjie.enna1@bytedance.com
from 字节跳动 STE 基础库与编译工具链团队
欢迎更多同学加入我们

ByteDance 字节跳动