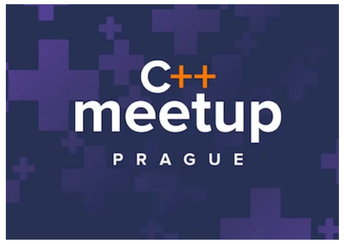
Casting Out Code Goblins: ASan's 🎃Halloween Guard



October 31 👻
2023

🐦@ciura_victor
🦣 @ciura_victor@hachyderm.io

Victor Ciura
Principal Engineer
Visual C++

Casting Out Code Goblins: ASan's 🎃Halloween Guard

In the code where the digital goblins play,
And memory monsters lurk, ready to sway,
ASan's the wizard, with spells so neat,
Banishing bugs, making errors retreat.

Pumpkins glow, and the night's full of haze,
But with ASan, no app goes astray.
Witches and warlocks might cackle and scheme,
But ASan ensures a software dream.

On All Hallows' Eve, when codes might feel eerie,
With ASan on watch, we're never weary.
For amidst the spook, and the bytes' haunted fun,
ASan's the shield, making dangers undone!

Let's see how AddressSanitizer works behind the scenes (compiler and ASAN runtime) and analyze the instrumentation impact, both in perf and memory footprint. We'll examine a handful of examples diagnosed by ASAN and see how easy it is to read memory snapshots to pinpoint the failure.

Casting Out Code Goblins: ASan's 🎃Halloween Guard

In the code where the digital goblins play,
And memory monsters lurk, ready to sway,
ASan's the wizard, with spells so neat,
Banishing bugs, making errors retreat.

Pumpkins glow, and the night's full of haze,
But with ASan, no app goes astray.
Witches and warlocks might cackle and scheme,
But ASan ensures a software dream.

On All Hallows' Eve, when codes might feel eerie,
With ASan on watch, we're never weary.
For amidst the spook, and the bytes' haunted fun,
ASan's the shield, making dangers undone!

Let's see how AddressSanitizer works behind the scenes (compiler and ASAN runtime) and analyze the instrumentation impact, both in perf and memory footprint. We'll examine a handful of examples diagnosed by ASAN and see how easy it is to read memory snapshots to pinpoint the failure.

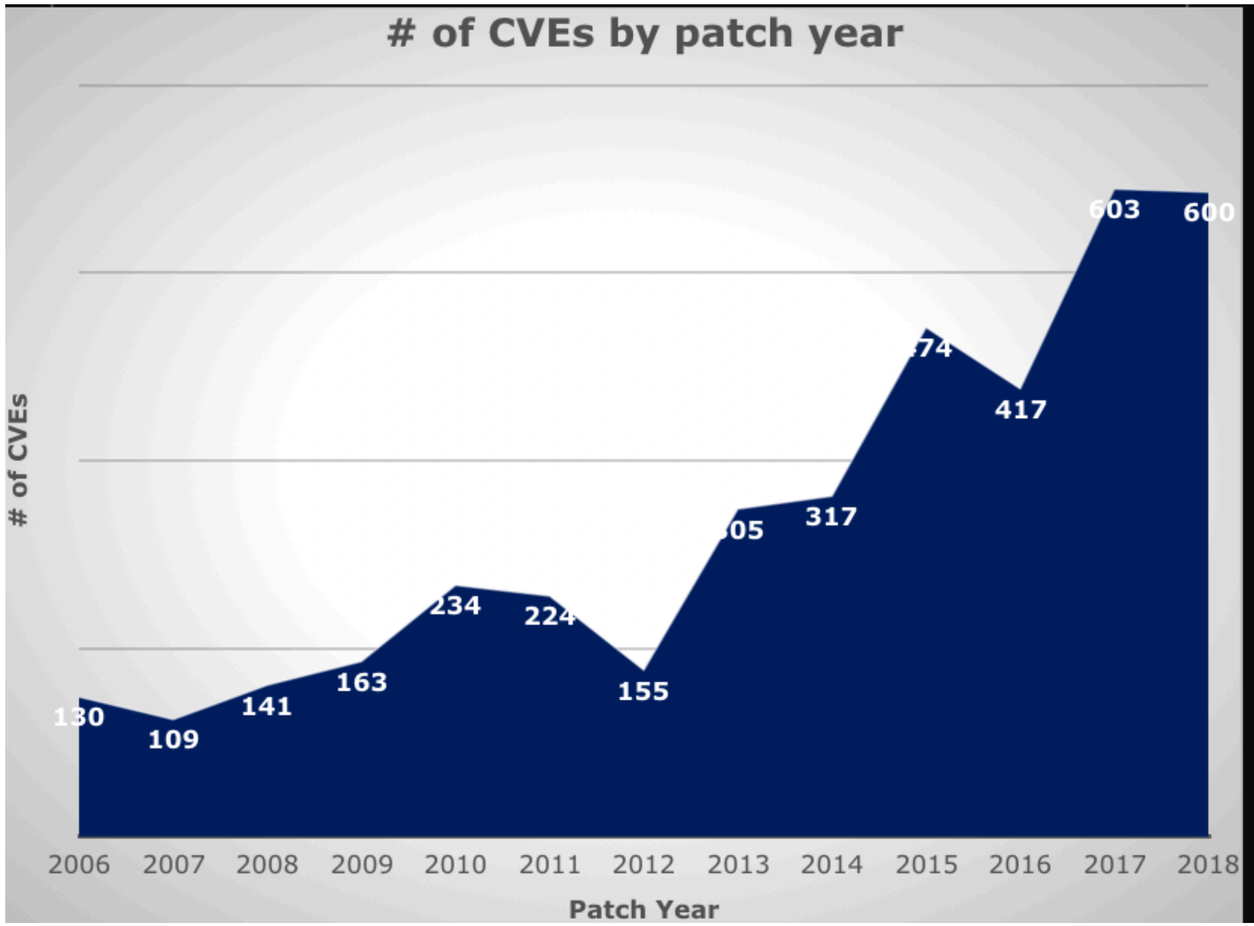
2021 CWE Top 25

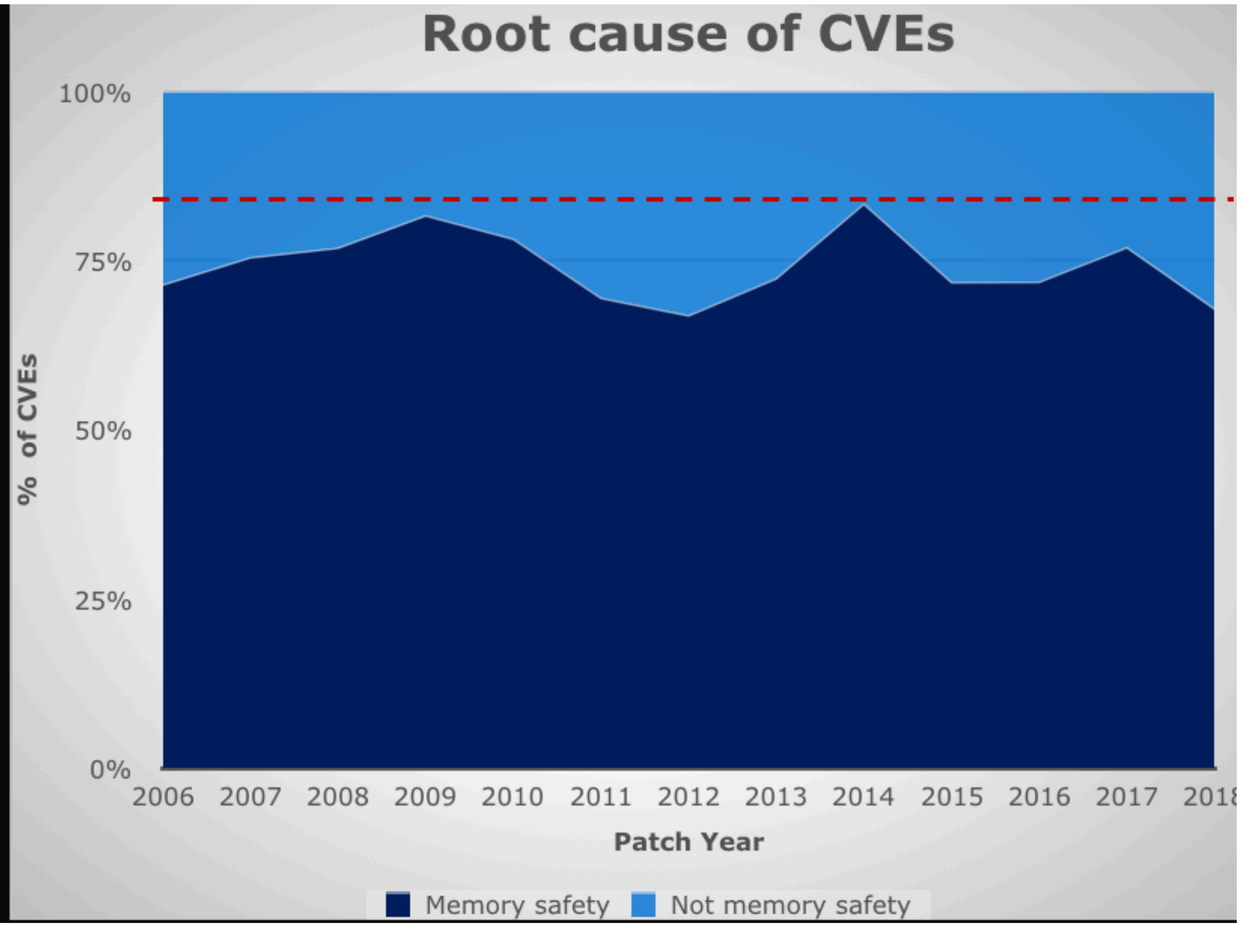
Rank	ID	Name	Score	2020 Rank Change
[1]	CWE-787	Out-of-bounds Write	65.93	+1
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.84	-1
[3]	CWE-125	Out-of-bounds Read	24.9	+1
[4]	CWE-20	Improper Input Validation	20.47	-1
[5]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	19.55	+5
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	19.54	0
[7]	CWE-416	Use After Free	16.83	+1
[8]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.69	+4
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	14.46	0
[10]	CWE-434	Unrestricted Upload of File with Dangerous Type	8.45	+5
[11]	CWE-306	Missing Authentication for Critical Function	7.93	+13
[12]	CWE-190	Integer Overflow or Wraparound	7.12	-1
[13]	CWE-502	Deserialization of Untrusted Data	6.71	+8
[14]	CWE-287	Improper Authentication	6.58	0
[15]	CWE-476	NULL Pointer Dereference	6.54	-2
[16]	CWE-798	Use of Hard-coded Credentials	6.27	+4
[17]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	5.84	-12

Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Weaknesses

Common Vulnerabilities and Exposures

Memory safety continues to dominate





Opportunistic exploits 😈

ROP - return oriented programming
DOP - data oriented programming
BOP - block oriented programming
DDM - direct memory manipulation

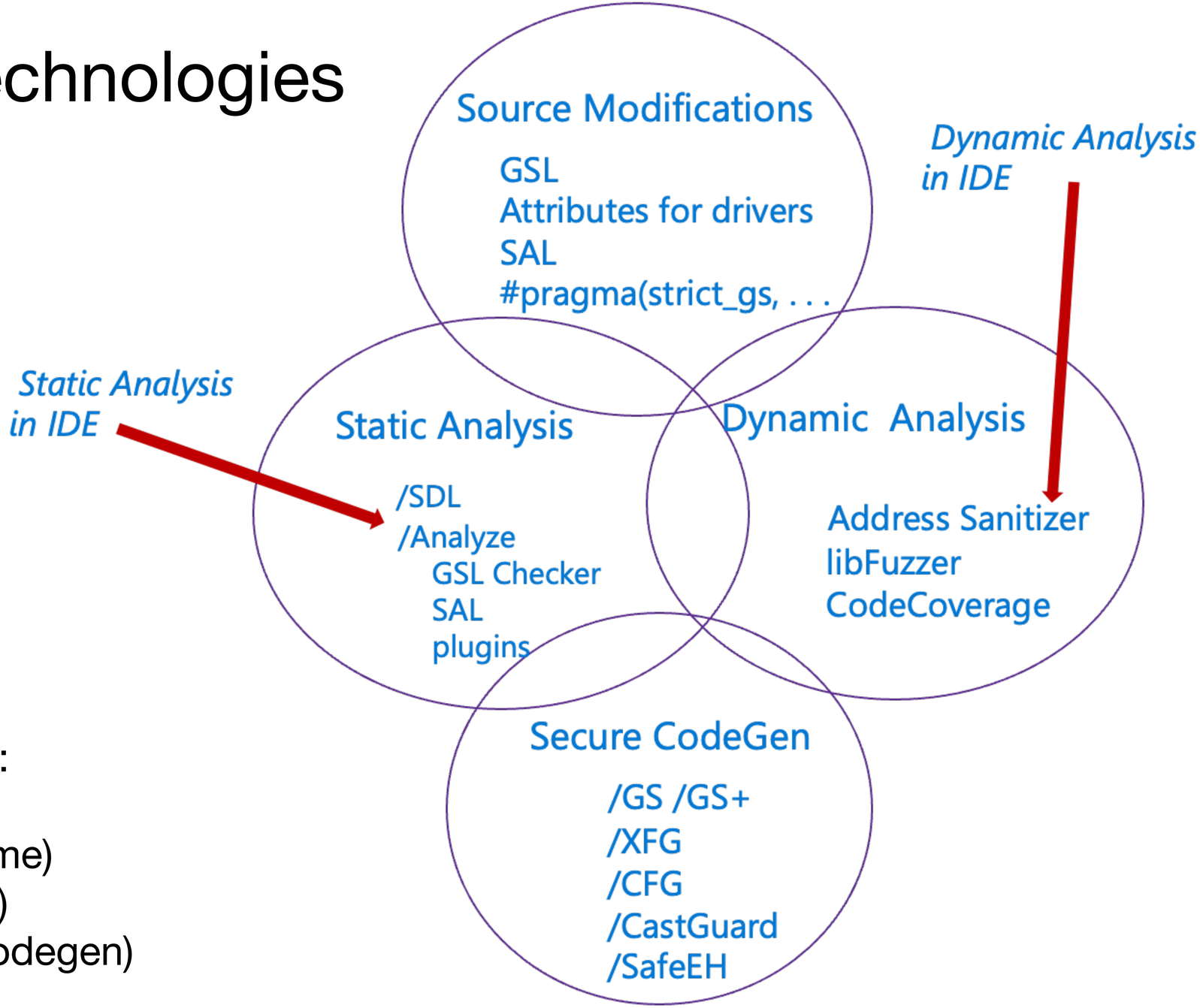
➡️ they all exploit memory safety errors

C++ developers

C++ developers

C++ developers

C++ Security Technologies



Delivering safe C++ requires:

- static analysis (compile-time)
- dynamic analysis (runtime)
- code hardening (secure codegen)

Quick primer

Static vs Dynamic Analysis

Static Analysis

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code
- is intimately linked to the used programming language

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code
- is intimately linked to the used programming language
- can detect a lot of semantic issues

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code
- is intimately linked to the used programming language
- can detect a lot of semantic issues
- can yield a lot of false positive results (sometimes you go on a wild goose chase)

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code
- is intimately linked to the used programming language
- can detect a lot of semantic issues
- can yield a lot of false positive results (sometimes you go on a wild goose chase)
- very poor at whole program analysis (follow connections in different TUs)

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code
- is intimately linked to the used programming language
- can detect a lot of semantic issues
- can yield a lot of false positive results (sometimes you go on a wild goose chase)
- very poor at whole program analysis (follow connections in different TUs)
- almost helpless around virtual functions (difficult to de-virtualize calls)

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code
- is intimately linked to the used programming language
- can detect a lot of semantic issues
- can yield a lot of false positive results (sometimes you go on a wild goose chase)
- very poor at whole program analysis (follow connections in different TUs)
- almost helpless around virtual functions (difficult to de-virtualize calls)
- weak analysis ability around global pointers

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code
- is intimately linked to the used programming language
- can detect a lot of semantic issues
- can yield a lot of false positive results (sometimes you go on a wild goose chase)
- very poor at whole program analysis (follow connections in different TUs)
- almost helpless around virtual functions (difficult to de-virtualize calls)
- weak analysis ability around global pointers
- pointer aliasing makes it hard to prove things (alias analysis is hard problem)

Static Analysis

- offline (out of the normal compilation cycle) => can take longer to process source code
- is intimately linked to the used programming language
- can detect a lot of semantic issues
- can yield a lot of false positive results (sometimes you go on a wild goose chase)
- very poor at whole program analysis (follow connections in different TUs)
- almost helpless around virtual functions (difficult to de-virtualize calls)
- weak analysis ability around global pointers
- pointer aliasing makes it hard to prove things (alias analysis is hard problem)
- vicious cycle: type propagation <> alias analysis

Dynamic Analysis

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode
- runtime overhead (performance impact: depending on tool, from **2x** up to **10x**)

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode
- runtime overhead (performance impact: depending on tool, from **2x** up to **10x**)
- extra-memory usage (for memory related tools/instrumentation), 2x or more

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode
- runtime overhead (performance impact: depending on tool, from **2x** up to **10x**)
- extra-memory usage (for memory related tools/instrumentation), 2x or more
- sometimes difficult to map error reports into source code for Release/optimized builds (symbols info, line numbers, inlined functions)

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode
- runtime overhead (performance impact: depending on tool, from **2x** up to **10x**)
- extra-memory usage (for memory related tools/instrumentation), 2x or more
- sometimes difficult to map error reports into source code for Release/optimized builds (symbols info, line numbers, inlined functions)
- some tools require to recompile the whole program in instrumented mode

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode
- runtime overhead (performance impact: depending on tool, from **2x** up to **10x**)
- extra-memory usage (for memory related tools/instrumentation), 2x or more
- sometimes difficult to map error reports into source code for Release/optimized builds (symbols info, line numbers, inlined functions)
- some tools require to recompile the whole program in instrumented mode
- must integrate runtime analysis with Test Units

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode
- runtime overhead (performance impact: depending on tool, from **2x** up to **10x**)
- extra-memory usage (for memory related tools/instrumentation), 2x or more
- sometimes difficult to map error reports into source code for Release/optimized builds (symbols info, line numbers, inlined functions)
- some tools require to recompile the whole program in instrumented mode
- must integrate runtime analysis with Test Units
- must ensure good code coverage for the runtime analysis (all possible scenarios)

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode
- runtime overhead (performance impact: depending on tool, from **2x** up to **10x**)
- extra-memory usage (for memory related tools/instrumentation), 2x or more
- sometimes difficult to map error reports into source code for Release/optimized builds (symbols info, line numbers, inlined functions)
- some tools require to recompile the whole program in instrumented mode
- must integrate runtime analysis with Test Units
- must ensure good code coverage for the runtime analysis (all possible scenarios)
- the biggest impact when combined with fuzzing

Dynamic Analysis

- sometimes intrusive: you need to compile the program in a special mode
- runtime overhead (performance impact: depending on tool, from **2x** up to **10x**)
- extra-memory usage (for memory related tools/instrumentation), 2x or more
- sometimes difficult to map error reports into source code for Release/optimized builds (symbols info, line numbers, inlined functions)
- some tools require to recompile the whole program in instrumented mode
- must integrate runtime analysis with Test Units
- must ensure good code coverage for the runtime analysis (all possible scenarios)
- the biggest impact when combined with fuzzing

0 false positives!

Sanitizers

Sanitizers

- `AddressSanitizer` - detects addressability issues
- `LeakSanitizer` - detects memory leaks
- `ThreadSanitizer` - detects data races and deadlocks
- `MemorySanitizer` - detects use of uninitialized memory
- `HWASAN` - hardware-assisted AddressSanitizer (consumes less memory)
- `UBSan` - detects Undefined Behavior

github.com/google/sanitizers

Address Sanitizer (ASan)

De-facto standard for detecting memory safety issues

It's important for basic code correctness and true vulnerabilities

github.com/google/sanitizers/wiki/AddressSanitizer

"Why did the programmer use ASan?"

"Why did the programmer use ASan?"

"To address all the bugs!" 😄

"Why did the programmer use ASan?"

"To address all the bugs!" 😄

-- credit ChatGPT4

Address Sanitizer (ASan)

Detects:

- **Use after free** (dangling pointer dereference)
- **Heap buffer overflow**
- **Stack buffer overflow**
- **Global buffer overflow**
- **Use after return**
- **Use after scope**
- **Initialization order bugs**
- **Memory leaks**

github.com/google/sanitizers/wiki/AddressSanitizer

Address Sanitizer (ASan)

Started in **LLVM** by a team @ Google

and quickly took off as a *de facto* industry standard

for runtime program analysis

github.com/google/sanitizers/wiki/AddressSanitizer

Address Sanitizer (ASan)

LLVM starting with version **3.1** (2012)

GCC starting with version **4.8** (2013)

MSVC starting with VS **16.4** (2019)

ASan features:

- stack-use-after-scope
- stack-buffer-overflow
- stack-buffer-underflow
- heap-buffer-overflow (no underflow)
- heap-use-after-free
- calloc-overflow
- dynamic-stack-buffer-overflow (alloca)
- global-overflow (C++ source code)
- new-delete-type-mismatch
- memcpy-param-overlap
- allocation-size-too-big
- invalid-aligned-alloc-alignment
- use-after-poison
- intra-object-overflow
- initialization-order-fiasco
- double-free
- alloc-dealloc-mismatch

docs.microsoft.com/en-us/cpp/sanitizers/asan

ASan features:

- `global 'C' variables`
  (in C a global can be declared many times, and each declaration can be of a different type and size)

- `__declspec(no_sanitize_address)`
  (**opt-out** of instrumenting entire functions or specific variables)

- `automatically link appropriate ASan libs`
  (eg. when building from command-line with `/fsanitize:address`)

- `use-after-return (opt-in)`
  (requires code gen that utilizes two stack frames for each function)

ASan features:

ASan features:

- expanded `RtlAllocateHeap` support

ASan features:

- expanded `RtlAllocateHeap` support
- support for the legacy `GlobalAlloc` and `LocalAlloc` family of memory functions

```
ASAN_OPTIONS=windows_hook_legacy_allocators=true
```

ASan features:

- expanded `RtlAllocateHeap` support
- support for the legacy `GlobalAlloc` and `LocalAlloc` family of memory functions

  ```
  ASAN_OPTIONS=windows_hook_legacy_allocators=true
  ```

- explicit error messages for shadow memory interleaving and interception failure

ASan features:

- expanded `RtlAllocateHeap` support
- support for the legacy `GlobalAlloc` and `LocalAlloc` family of memory functions

  ```
  ASAN_OPTIONS=windows_hook_legacy_allocators=true
  ```

- explicit error messages for shadow memory interleaving and interception failure
- IDE integration can now handle the complete collection of exceptions which ASan can report

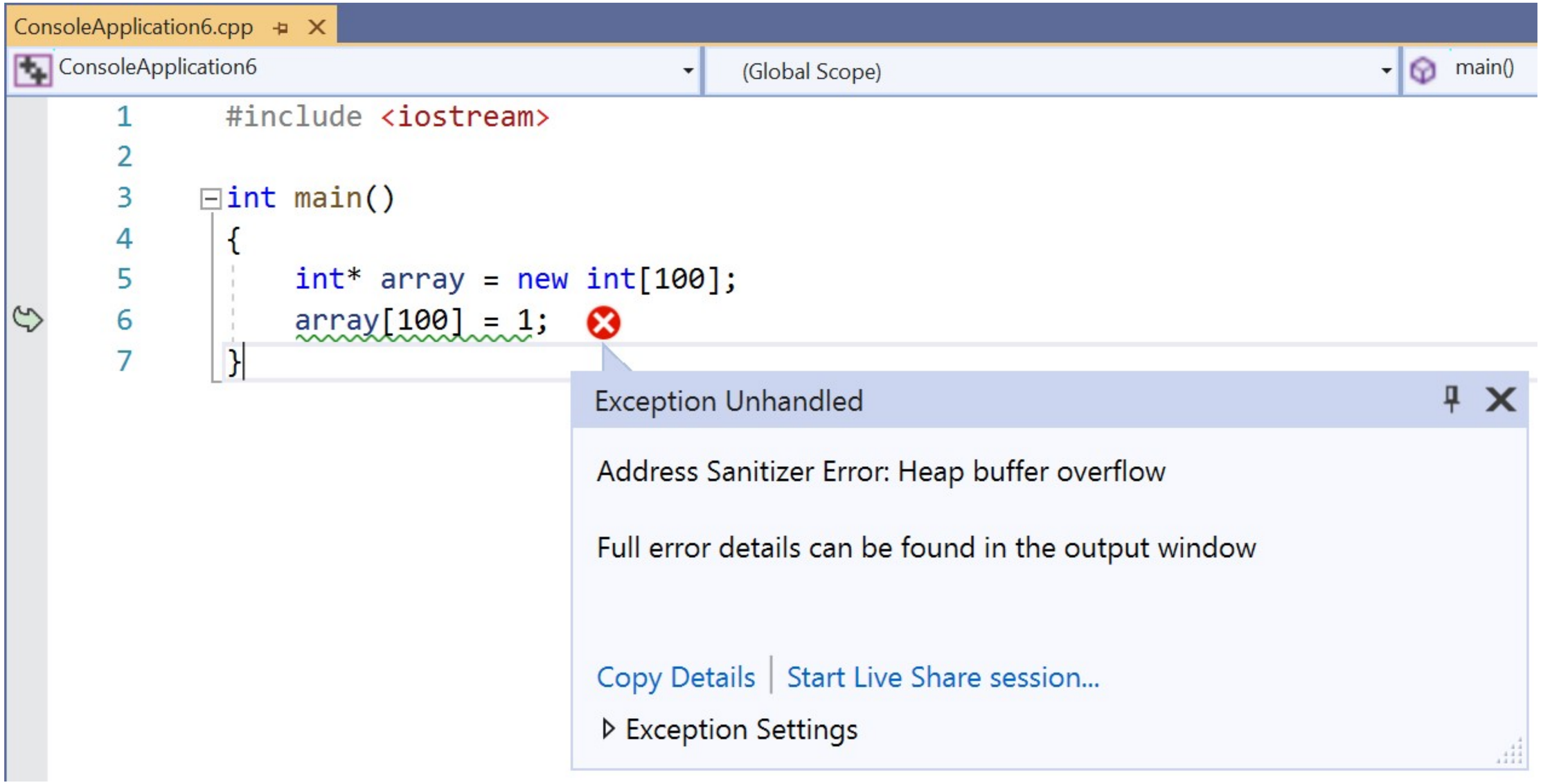
ASan features:

- expanded `RtlAllocateHeap` support
- support for the legacy `GlobalAlloc` and `LocalAlloc` family of memory functions

  ```
  ASAN_OPTIONS=windows_hook_legacy_allocators=true
  ```

- explicit error messages for shadow memory interleaving and interception failure
- IDE integration can now handle the complete collection of exceptions which ASan can report
- compiler/linker will suggest emitting debug information when building with ASan

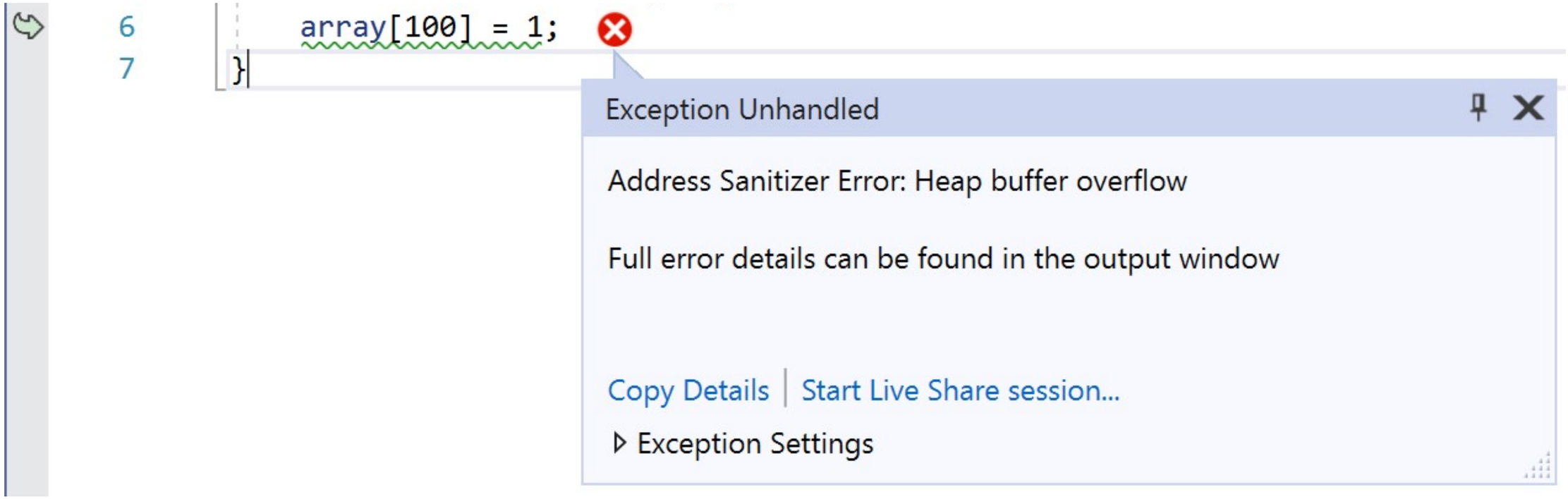
Address Sanitizer (ASan)



Address Sanitizer (ASan)

IDE Exception Helper will be displayed when an issue is encountered
=> program execution will stop

ASan logging information => Output window



LLVM

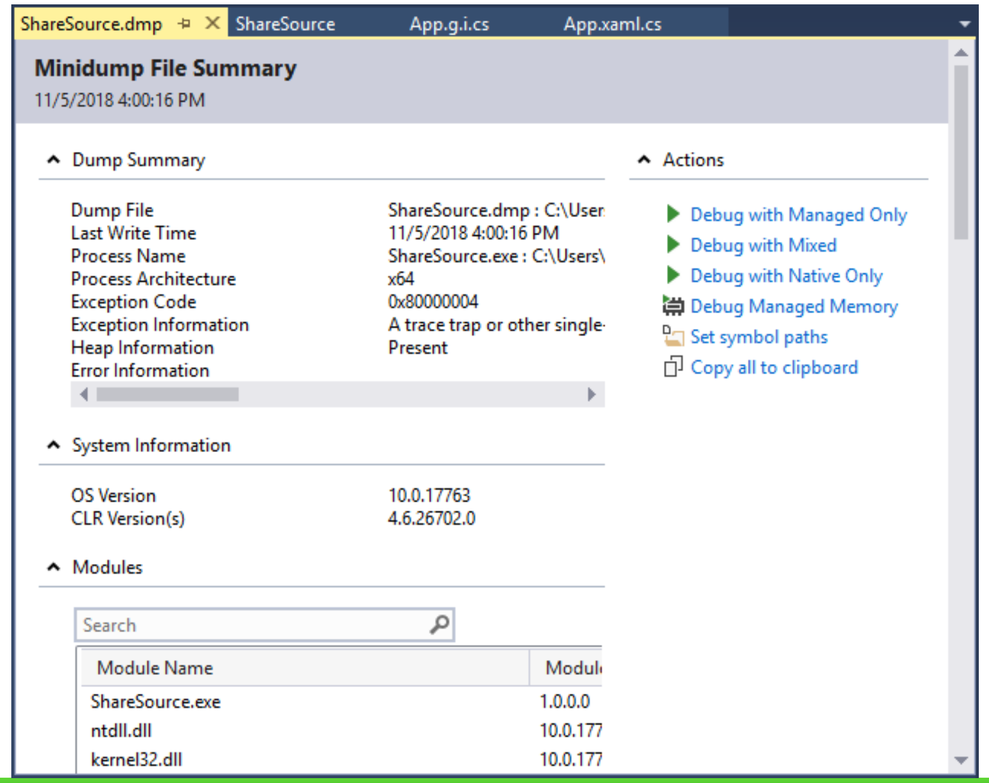
```
==27748==ERROR: AddressSanitizer: stack-use-after-scope on address 0x0055fc68 at pc 0x793d62de bp 0x0055fbf4 sp 0x0055fbe8
WRITE of size 80 at 0x0055fc68 thread T0
    #0 0x793d62f6 in __asan_wrap_memset d:\_work\5\s\llvm\projects\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:764
    #1 0x77dd46e7  (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2c46e7)
    #2 0x77dd4ce1  (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2c4ce1)
    #3 0x75d408fe  (C:\WINDOWS\System32\KERNELBASE.dll+0x100f08fe)
    #4 0xa5ada0 in try_get_first_available_module minkernel\crts\ucrt\src\appcrt\internal\winapi_thunks.cpp:271
    #5 0xa5ae99 in try_get_function minkernel\crts\ucrt\src\appcrt\internal\winapi_thunks.cpp:326
    #6 0xa5b028 in __acrt_AppPolicyGetProcessTerminationMethodInternal minkernel\crts\ucrt\src\appcrt\internal\winapi_thunks.cpp:737
    #7 0xa606ad in __acrt_get_process_end_policy minkernel\crts\ucrt\src\appcrt\internal\win_policies.cpp:84
    #8 0xa52dcb in exit_or_terminate_process minkernel\crts\ucrt\src\appcrt\startup\exit.cpp:134
    #9 0xa52da7 in common_exit minkernel\crts\ucrt\src\appcrt\startup\exit.cpp:280
    #10 0xa52fb6 in exit minkernel\crts\ucrt\src\appcrt\startup\exit.cpp:293
    #11 0xa2deb3 in _scrt_common_main_seh d:\agent\_work\2\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:295
    #12 0x75ef6358  (C:\WINDOWS\System32\KERNEL32.DLL+0x6b816358)
    #13 0x77df7a93  (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2e7a93)

Address 0x0055fc68 is located in stack of thread T0
SUMMARY: AddressSanitizer: stack-use-after-scope d:\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:764 in __asan_wrap_memset
Shadow bytes around the buggy address:
  0x300abf30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x300abf70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x300abf80: 00 00 00 00 00 00 00 00 00 00 00 00 00[f8]00 00
  0x300abf90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x300abfd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
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
==27748==ABORTING
```

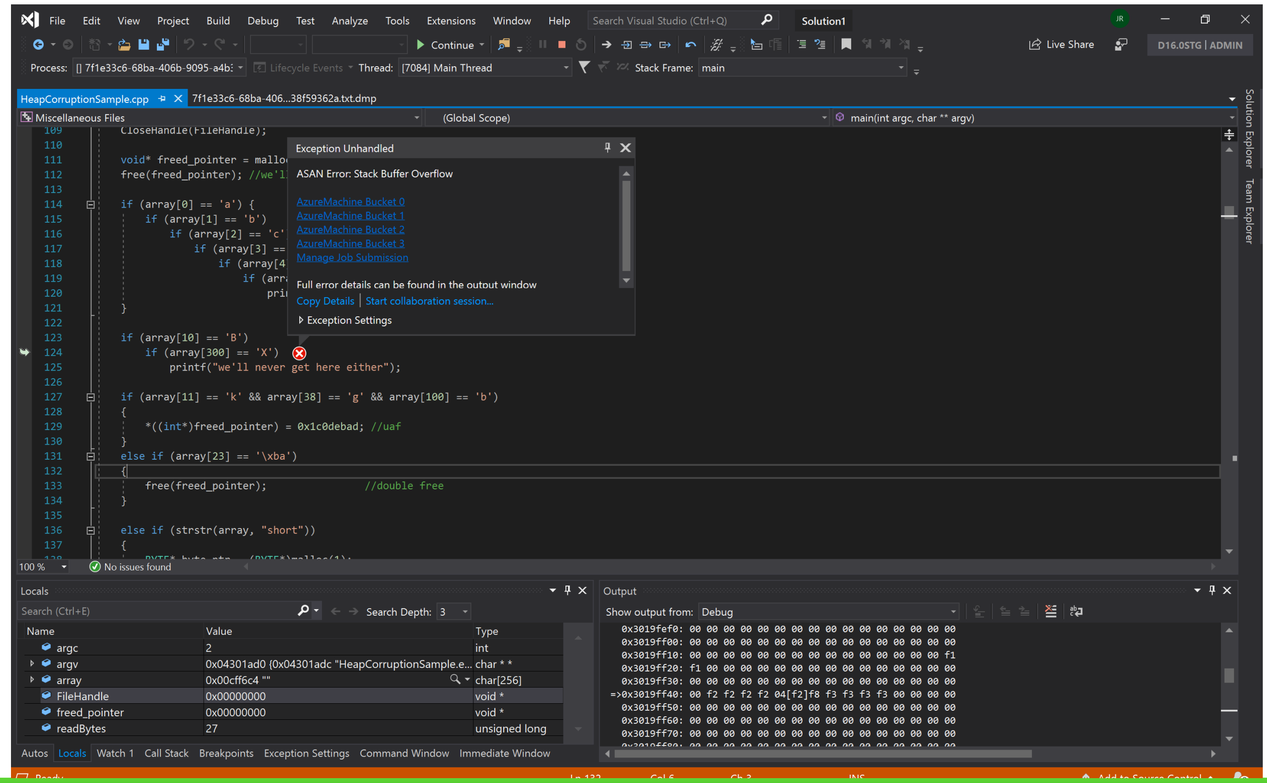
Snapshot File

Minidump file (*.dmp) <= Windows snapshot process (program virtual memory/heap + metadata)

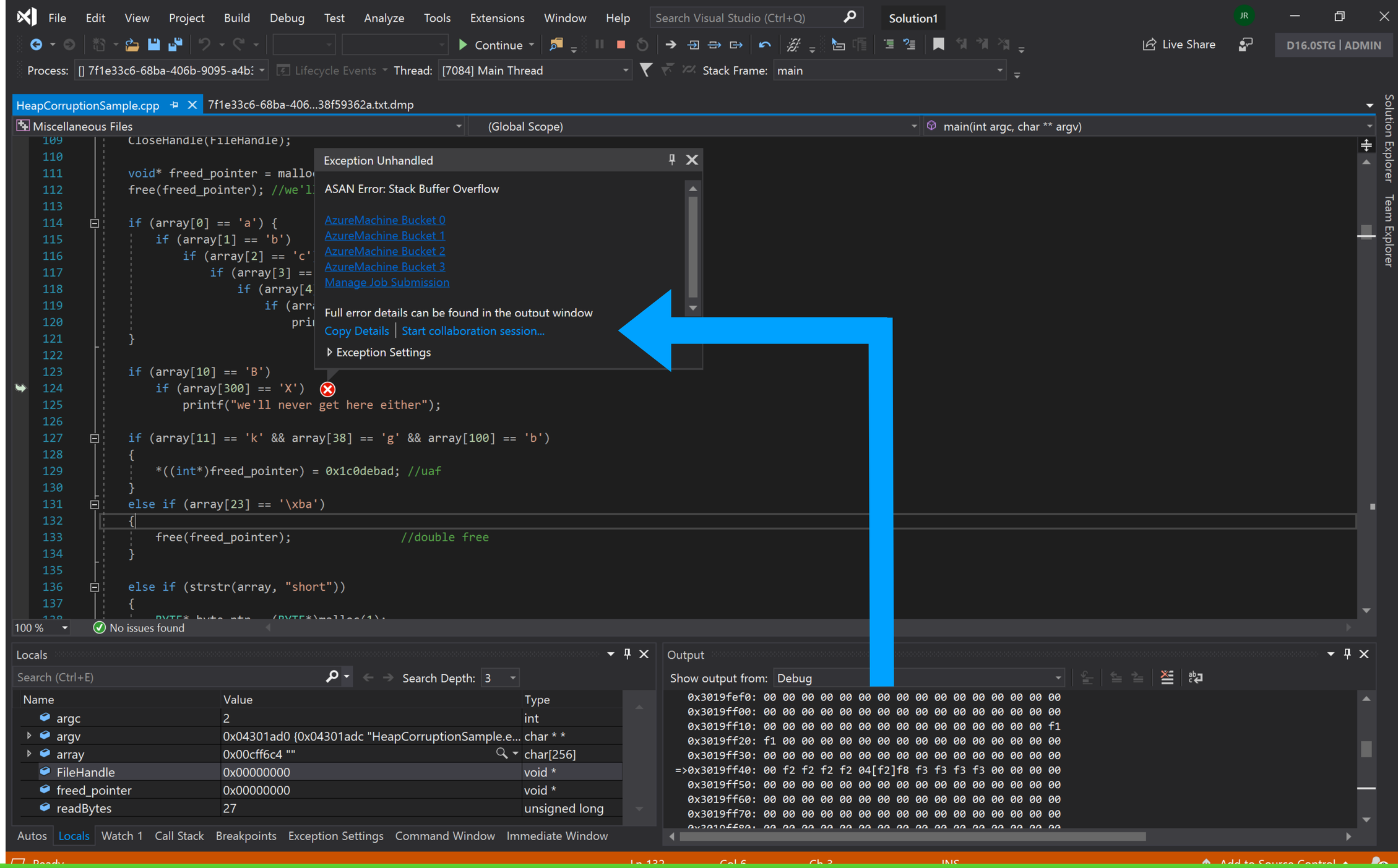
VS can parse & open this => Points at the location the error occurred.

Changes the way you report a bug, in general





Snapshot Loaded



How does it work ?

ASan is just Malware, used for Good 😈

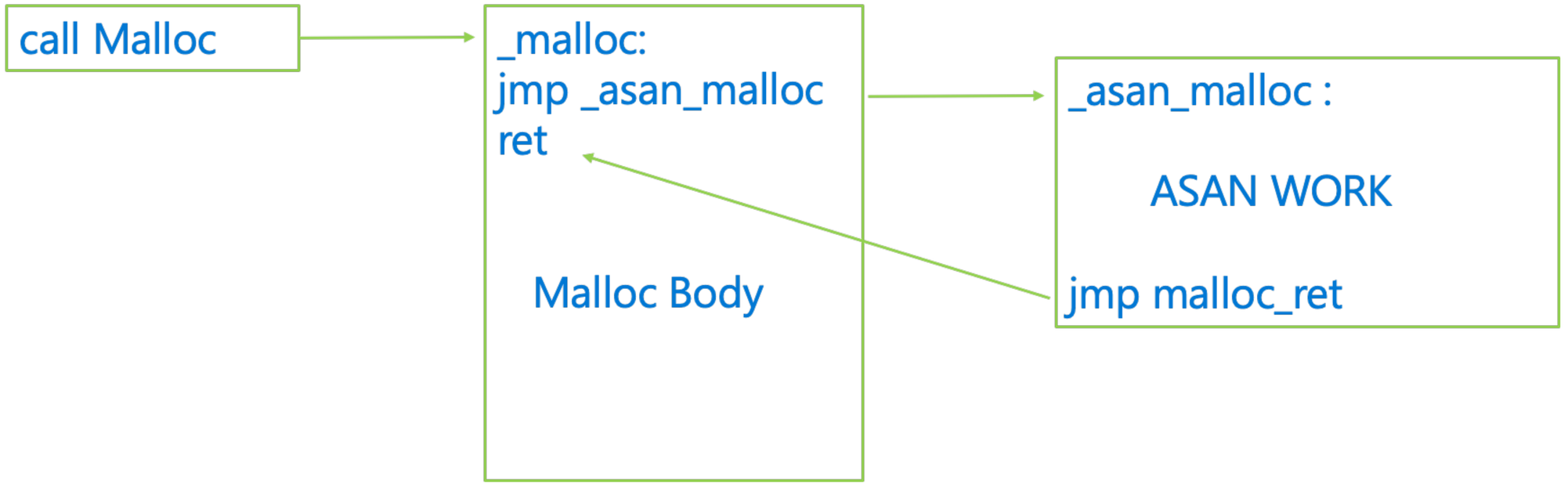
Address Sanitizer (ASan)

Compiler

- instrumentation code, stack layout, and calls into runtime
- meta-data in OBJ for the runtime

Sanitizer Runtime

- hooking `malloc(), memset(), memcpy(), strncpy(), RtlAllocate(), ...`
- error analysis and reporting
- does not require complete recompile => great for **interop**
- zero false positives


call Malloc
_malloc:
jmp _asan_malloc
ret
Malloc Body
_asan_malloc :
ASAN WORK
jmp malloc_ret

ASan Report

```
==23364==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x12ac01b801d0 at
pc 0x7ff6e3a627be bp 0x0097d4b4fac0 sp 0x0097d4b4fac8
WRITE of size 4 at 0x12ac01b801d0 thread T0
#0 0x7ff6e3a627bd in main C:\Asana\Asana.cpp:10
#1 0x7ff6e3a66ce8 in invoke_main D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:78
#2 0x7ff6e3a66bcd in __scrt_common_main_seh D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:288
#3 0x7ff6e3a66a8d in __scrt_common_main D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:330
#4 0x7ff6e3a66d78 in mainCRTStartup D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_main.cpp:16
#5 0x7ffee9a76fd3 in BaseThreadInitThunk+0x13 (C:\WINDOWS\System32\KERNEL32.DLL+0x180016fd3)
#6 0x7ffeea97cec0 in RtlUserThreadStart+0x20 (C:\WINDOWS\SYSTEM32\ntdll.dll+0x18004cec0)

0x12ac01b801d0 is located 0 bytes to the right of 400-byte region [0x12ac01b80040,0x12ac01b801d0)
allocated by thread T0 here:
#0 0x7ffe83be7e91 in _asan_loadN_noabort+0x55555 (...\bin\HostX64\x64\clang_rt.asan_dbg_dynamic-x86_64.dll+0x180057e91)
#1 0x7ff6e3a62758 in main C:\Asana\Asana.cpp:9
#2 0x7ff6e3a66ce8 in invoke_main D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:78
#3 0x7ff6e3a66bcd in __scrt_common_main_seh D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:288
#4 0x7ff6e3a66a8d in __scrt_common_main D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:330
#5 0x7ff6e3a66d78 in mainCRTStartup D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_main.cpp:16
#6 0x7ffee9a76fd3 in BaseThreadInitThunk+0x13 (C:\WINDOWS\System32\KERNEL32.DLL+0x180016fd3)
#7 0x7ffeea97cec0 in RtlUserThreadStart+0x20 (C:\WINDOWS\SYSTEM32\ntdll.dll+0x18004cec0)
```

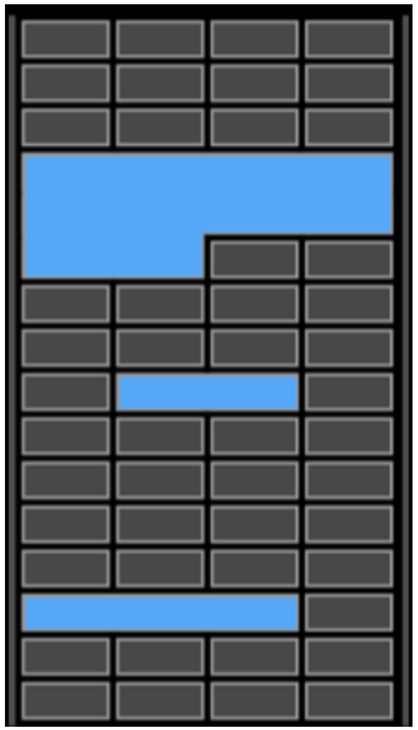
```
SUMMARY: AddressSanitizer: heap-buffer-overflow C:\Asana\Asana.cpp:10 in main()

Shadow bytes around the buggy address:
  0x04d981eeffe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x04d981eefff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x04d981ef0000: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
  0x04d981ef0010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x04d981ef0020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x04d981ef0030: 00 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa
  0x04d981ef0040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x04d981ef0050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x04d981ef0060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x04d981ef0070: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x04d981ef0080: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Addressable:	00 👍	
Partially addressable:	01 02 03 04 05 06 07	(of the 8 application bytes, how many are accessible)
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	issues & markers
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	

Shadow byte legend

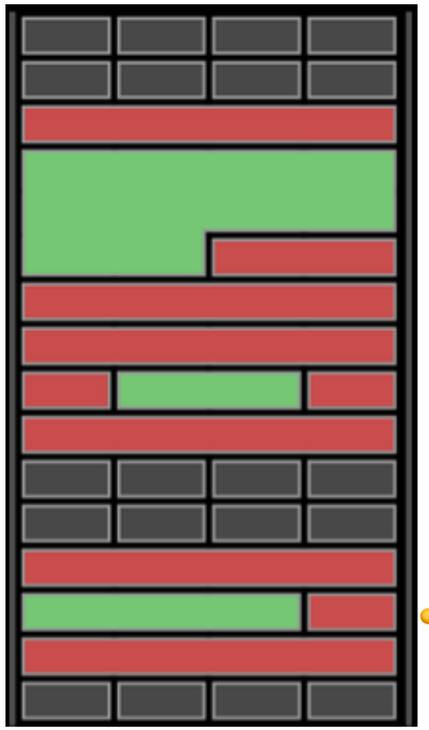
(one shadow byte represents 8 application bytes)

Shadow Mapping



my allocated memory

➡️



🧪☣️ Poisoned memory

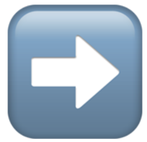
👉 Red zones

Process Memory

Shadow Memory

Code Generation
(simplified)

```
*p = 0xbadf00d
```



```
if (ShadowByte::IsBad(p))
  AsanRt::Report(p, sz)

*p = 0xbadf00d
```

If the shadow byte is poisoned,

ASAN runtime **reports** the problem and **crashes*** the application

* unless `continue_on_error` is enabled

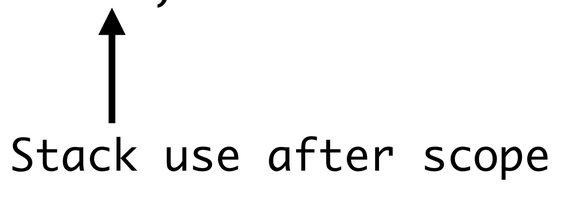
Code Generation
(simplified)

Lookups into shadow memory need to be very fast

ASAN maintains a lookup table where every **8 bytes** of user memory are tracked by **1 shadow byte**

=> **1/8** of the address space (shadow region)

A Shadow Byte: `*( (User_Address >> 3) + 0x30000000 ) = 0xF8;`



Code Generation
(simplified)

Lookups into shadow memory need to be very fast

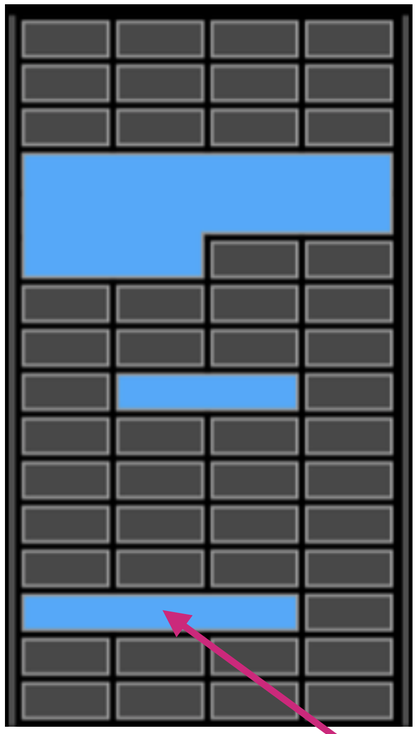
```
bool ShadowByte::IsBad(Addr) // is poisoned ?
{
  Shadow = Addr >> 3 + Offset;
  return (*Shadow) != 0;
}
```

Location of shadow region in memory

A Shadow Byte: `*( (User_Address >> 3) + 0x30000000 ) = 0xF8;`

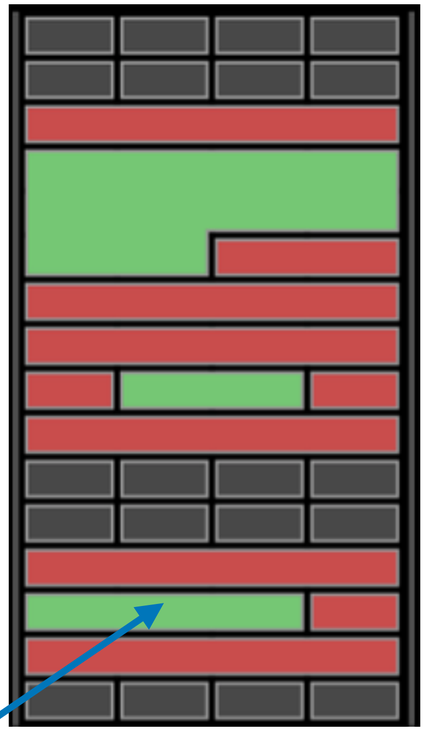
Stack use after scope

Shadow Mapping



Process Memory

```
if (ShadowByte::IsBad(p))
  AsanRt::Report(p, sz);

*p = 0xf00d
```

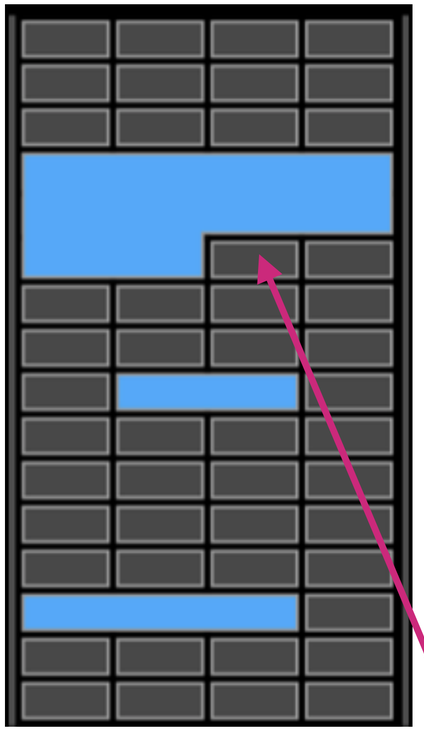


Shadow Memory

p

ShadowByte(p)

Shadow Mapping

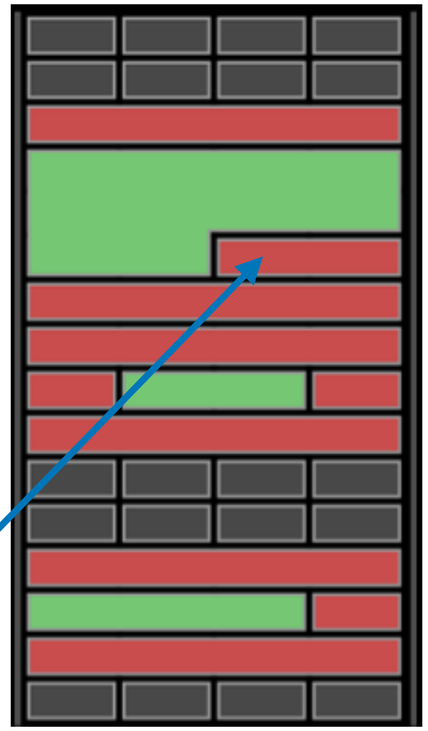


Process Memory

```
if (ShadowByte::IsBad(p))
  AsanRt::Report(p, sz);

*p = 0xbadf00d
```

p

ShadowByte(p)



Shadow Memory

Heap Red Zones

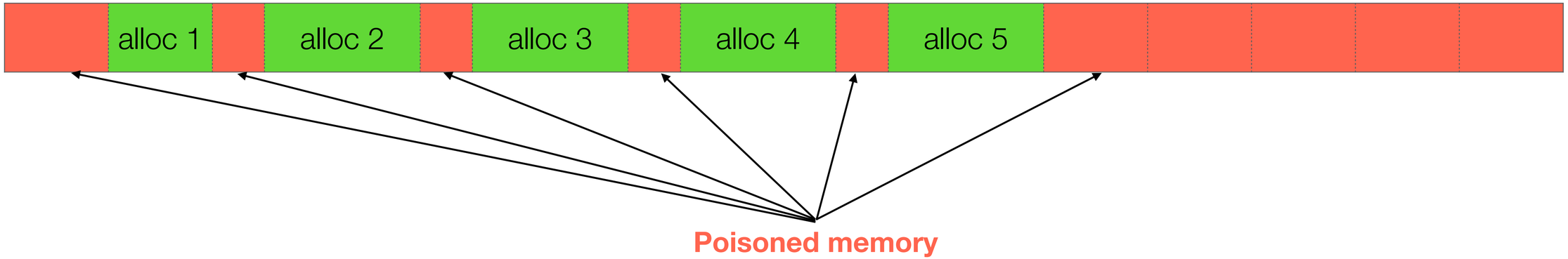
`malloc()`

`ASAN malloc()`

Heap Red Zones

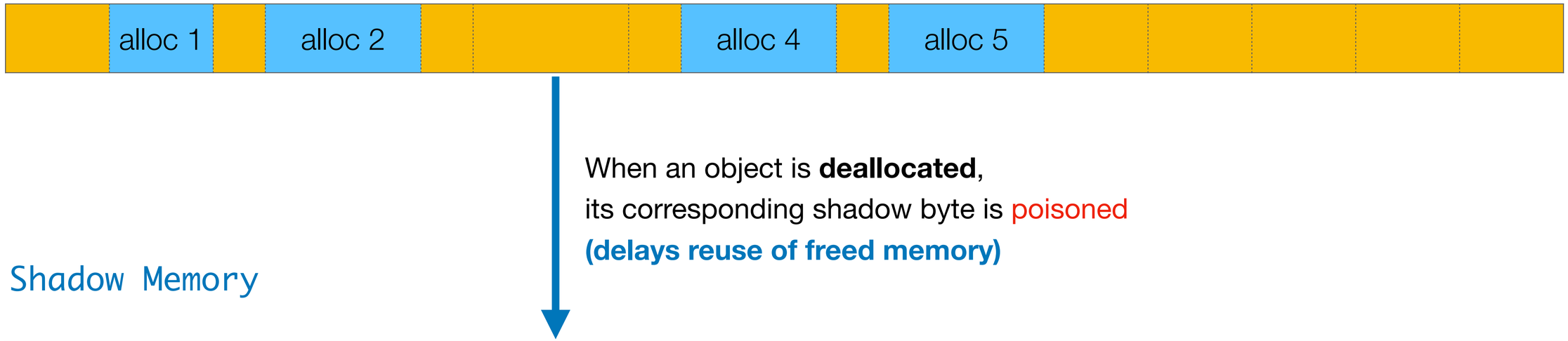
ASAN malloc()

Shadow Memory



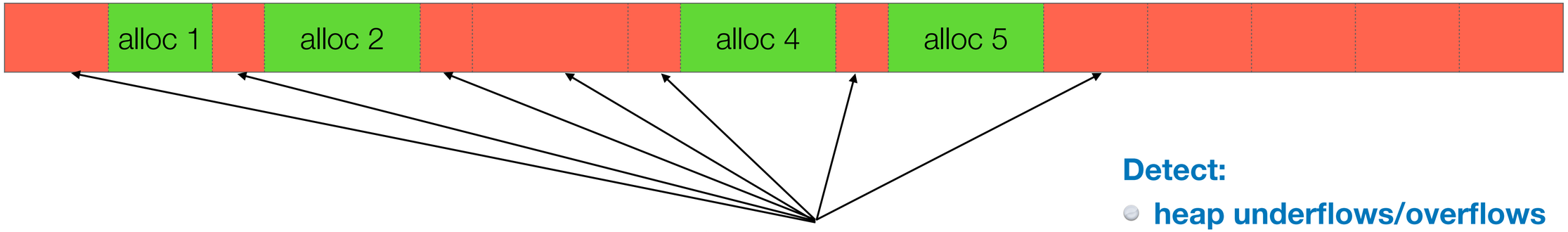
Heap Red Zones

ASAN malloc()



When an object is **deallocated**,
its corresponding shadow byte is poisoned
(delays reuse of freed memory)

Shadow Memory

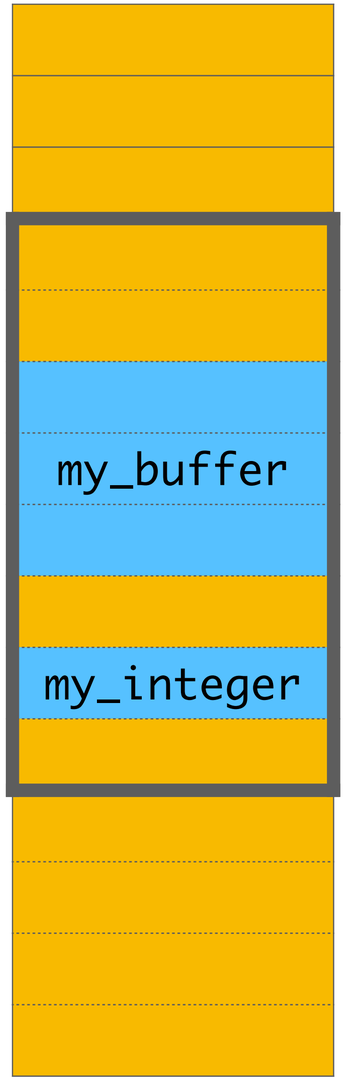


Poisoned memory

Detect:

- **heap underflows/overflows**
- **use-after-free & double free**

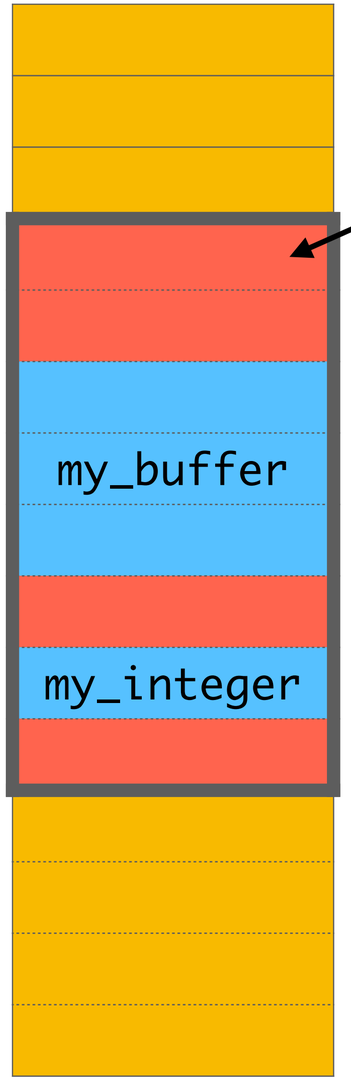
Stack Red Zones



Stack

```
void Func()
{
  std::byte my_buffer[12];
  int my_integer = 5;
  ...
  ...
  ...
  ...
  my_buffer[12] = 0;
}
```

Stack Red Zones



Stack

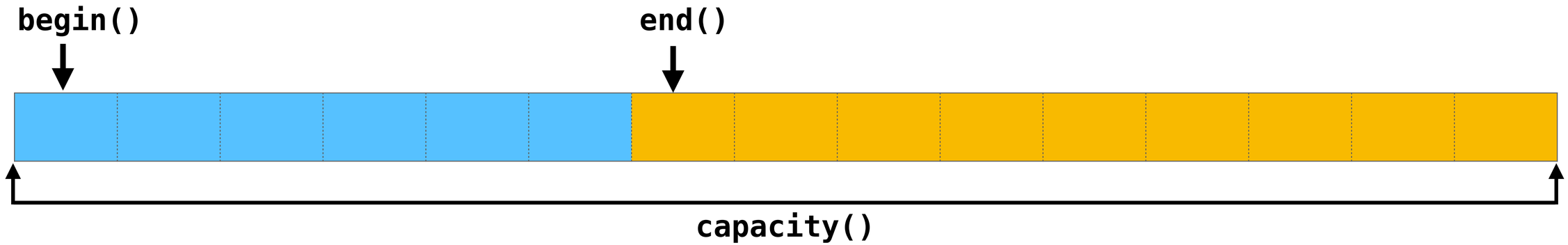
at runtime, the stack is **poisoned** when entering the function

```
void Func()
{
  std::byte my_buffer[12];
  int my_integer = 5;
  ...

  if (AsanRt::IsPoisoned(&my_buffer[12]))
    AsanRt::Report(my_buffer);
  my_buffer[12] = 0;
}
```

stack red zones are **un-poisoned** when exiting the function

AddressSanitizer ContainerOverflow

`std::vector<T>`



with the help of **code annotations** in **std::vector**

libc++

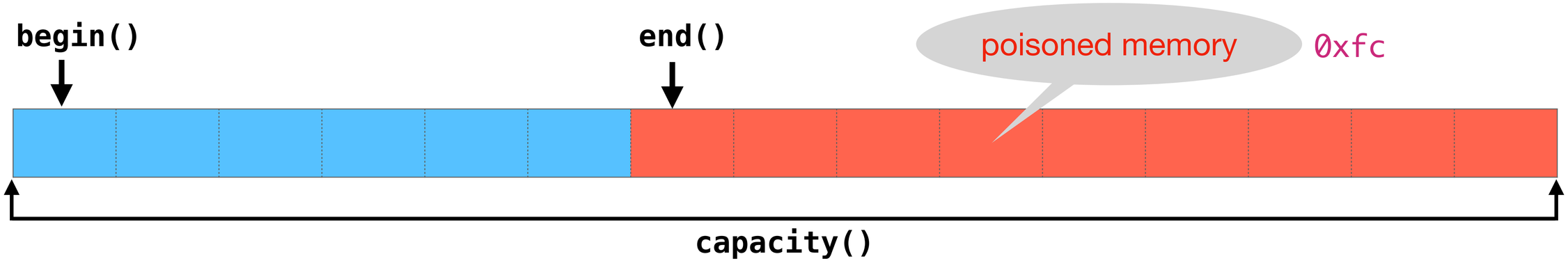
libstdc++

MSVC STL

github.com/google/sanitizers/wiki/AddressSanitizerContainerOverflow

AddressSanitizer ContainerOverflow

`std::vector<T>`




```
std::vector<int> v;
v.push_back(0);
v.push_back(1);
v.push_back(2);
assert(v.capacity() >= 4);
assert(v.size() == 3);

T * p = &v[0];
std::cout << p[3];
```

container-overflow

0xfc

`v[3]` could be detected by simple checks in `std::vector`

https://github.com/google/sanitizers/wiki/AddressSanitizerContainerOverflow

Address Sanitizer (ASan)

Very fast instrumentation

The average slowdown of the instrumented program is ~2x

github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers

Problems & Gotchas

stuff you need to know

The (ASan) Trap

One-n-Done problem

Blows up large test labs:

Eg.

- 36 hour builds
- 200,000+ tests
- 100+ distributed test machines

```
if (ShadowByte::IsBad(p))
  AsanRt::ReportAndAbort(p, sz)
```

```
*p = 0xbadf00d
```

ASan `continue_on_error`

C++ memory-safe-checked-build

- Return control back to app, after reporting every error
- Move ASan internal heap meta-data (“bad” writes clobber ASan internals with COE)
- Summarize unique errors (changed error reporting)

```
if (ShadowByte::IsBad(p))
  AsanRt::ReportContinue(p, sz)

*p = 0xbadf00d
```

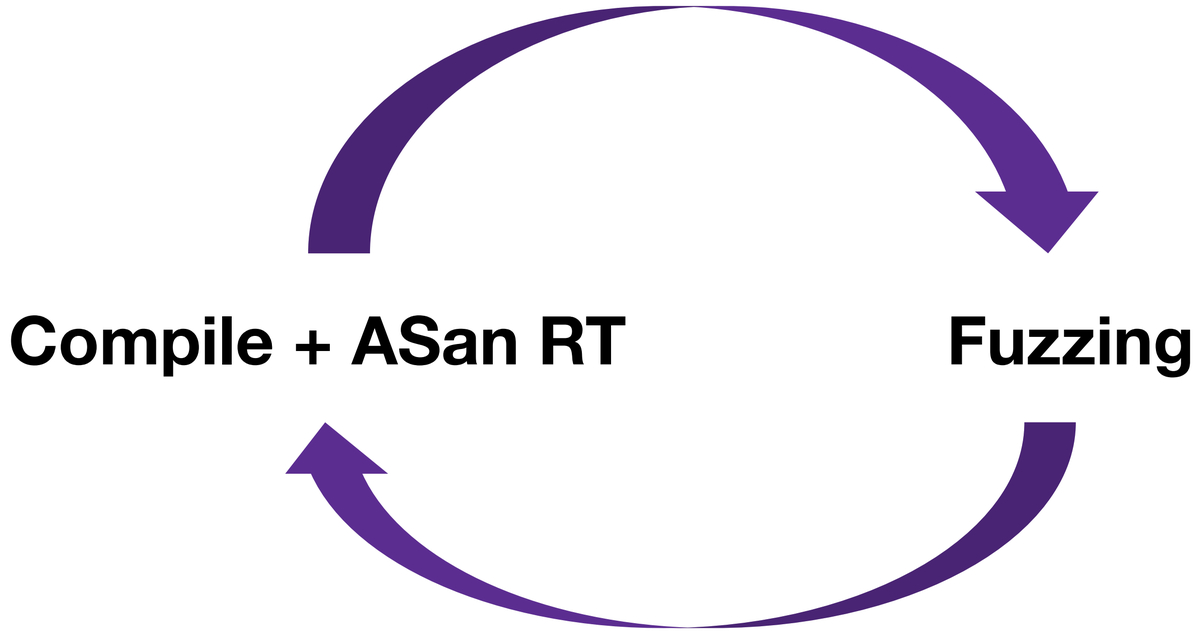
since VS2022 v17.6

Warm Fuzzy Feelings

Sanitizers + Fuzzing

Automatically generate inputs to your program, to crash it

Workflow

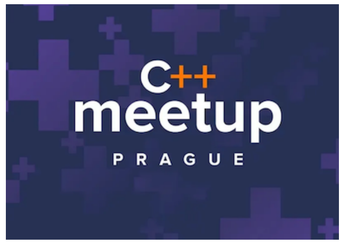


Go Fuzz & Sanitize !

ASan finds bugs

Really !

Casting Out Code Goblins: ASan's 🎃Halloween Guard



October 31 👻
2023

 @ciura_victor
🦣 @ciura_victor@hachyderm.io

Victor Ciura
Principal Engineer
Visual C++