AddressSanitizer on Windows

Victor Ciura

Abstract

Clang-tidy is the go-to assistant for most C++ programmers looking to improve their code, whether to modernize it or to find hidden bugs with its built-in checks. Static analysis is great, but you also get tons of false positives.

Now that you're hooked on smart tools, you have to try dynamic/runtime analysis. After years of improvements and successes for Clang and GCC users, LLVM AddressSanitizer (ASan) is finally available on Windows, in the latest Visual Studio 2019 versions. Let's find out how this experience is for MSVC projects.

We'll 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 in Visual Studio, to pinpoint the failure.

Want to unleash the memory vulnerability beast? Put your test units on steroids, by spinning fuzzing jobs with ASan in Azure, leveraging the power of the Cloud from the comfort of your Visual Studio IDE.

Do you think you have
good unit tests & coverage
on your project ?

Probably not...

I have yet to find a team
happy about this topic

But I reckon you have
at least one component
that you're pretty confident about

Would you be surprised

to find out there are <u>obvious</u> bugs/vulnerabilities

in that well tested component ?

Probably not

¯_(ツ)_/¯

I bet you'd like to quickly dig up something like this:

```
heap-buffer-overflow on address 0x0a2301b4 at pc 0x005b7a35 bp 0x011df078 sp 0x011df06c
READ of size 5 at 0x0a2301b4 thread T0

    #0 0x5b7a4d in __asan_wrap_strlen crt\asan\llvm\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:365
    #1 0x278eeb in ATL::CSimpleStringT<char,0>::StringLength MSVC\14.28.29333\atlmfc\include\atlsimpstr.h:726
    #2 0x278a35 in ATL::CSimpleStringT<char,0>::SetString MSVC\14.28.29333\atlmfc\include\atlsimpstr.h:602
    #3 0x274d69 in ATL::CSimpleStringT<char,0>::operator= MSVC\14.28.29333\atlmfc\include\atlsimpstr.h:314
    #4 0x274d99 in ATL::CStringT<char,ATL::StrTraitATL<char,ATL::ChTraitsCRT<char>>>::operator=
                   MSVC\14.28.29333\atlmfc\include\cstringt.h:1315
    #5 0x27469c in ATL::CStringT<char,ATL::StrTraitATL<char,ATL::ChTraitsCRT<char>>>::CStringT
                   MSVC\14.28.29333\atlmfc\include\cstringt.h:1115
    #6 0x27641a in SerValUtil::DecryptString C:\JobAI\advinst\msicomp\serval\SerValUtil.cpp:85
    #7 0x3e1660 in TestSerVal C:\JobAI\testunits\serval\SerValTests.cpp:60
    #8 0x5880e5 in FunctionTest::Run C:\JobAI\testunits\Tester.cpp:71
    #9 0x5889b1 in Tester::RunTest C:\JobAI\testunits\Tester.cpp:186
    #10 0x586ddb in Tester::ExecuteCommandLine C:\JobAI\testunits\Tester.cpp:558
    #11 0x5798d1 in main C:\JobAI\testunits\comps\TestComponents.cpp:2236

0x0a2301b4 is located 0 bytes to the right of 4-byte region [0x0a2301b0,0x0a2301b4)
allocated by thread T0
```

Stay with me for this 90 minute infomercial
and I'll show you how easy it is

Address Sanitizer on Windows

accu 2021

@ciura_victor

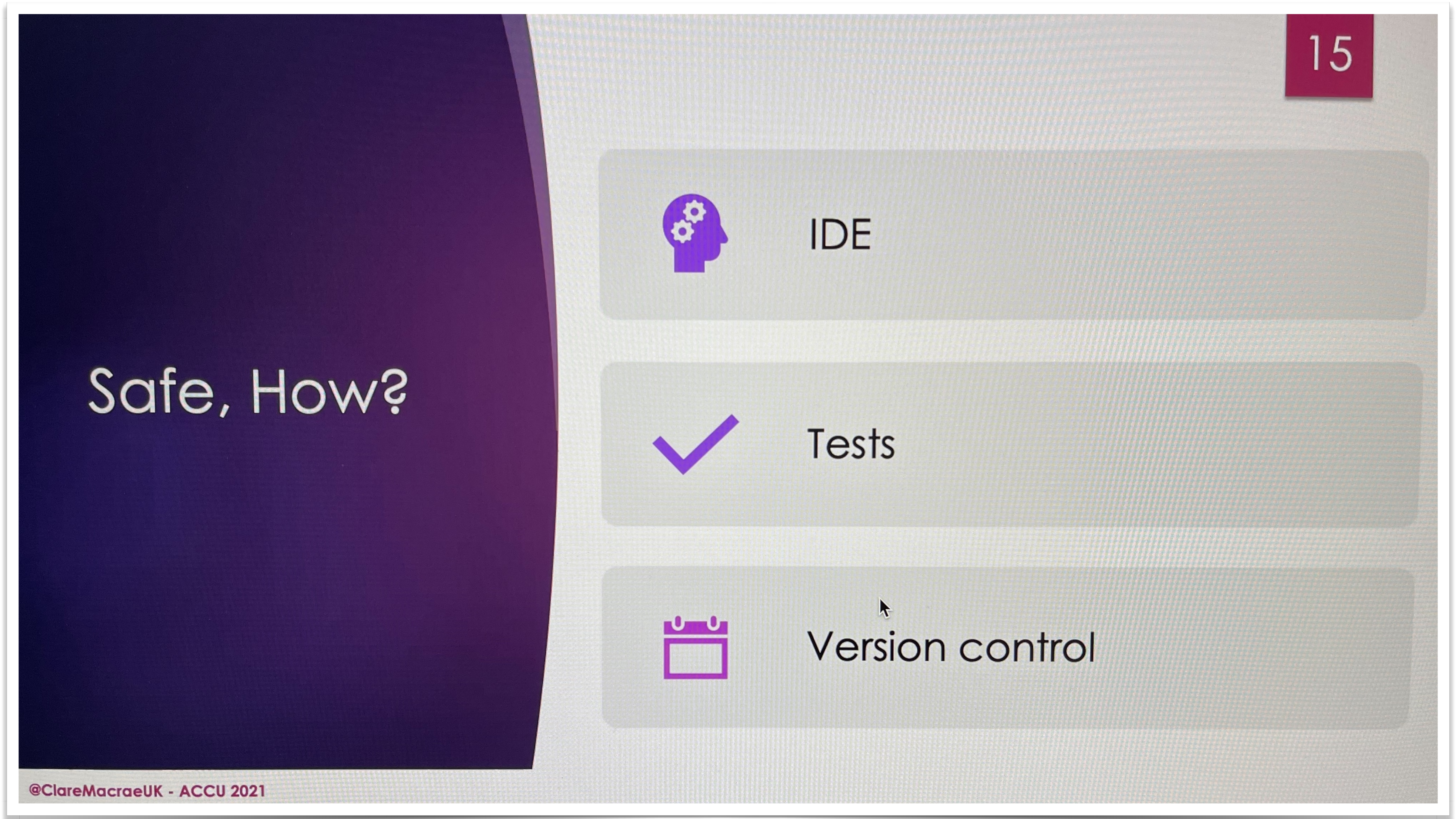
Victor Ciura
Principal Engineer

CAPHYON

Due to the nature of delivery medium &
streaming delays, I prefer to take questions at the end.

Q & A


accu 2021
VIRTUAL EVENT
Bloomberg
Engineering
undo
mosaic
CONSULTANTS TO FINANCIAL SERVICES
Keynote:
Refactoring Superpowers:
Make Your IDE Do Your Work, Faster and More Safely
Clare Macrae


15
Safe, How?
IDE
Tests
Version control
@ClareMacraeUK - ACCU 2021

Humans Depend on Tools

Get to know your tools

well

Programmers Depend on Tools

good code editor
(or IDE)

linter/formatter

powerful (visual) debugger

automated refactoring tools

build system

package manager

SCM client

CI/CD service

code reviews platform

recent compiler(s)
[conformant/strict]

perf profiler

test framework

static analyzer

dynamic analyzer
(runtime)

+ fuzzing

I'm a tool maker

Advanced Installer

Clang Power Tools

Free/OSS

@ciura_victor

Vignette in 3 parts

Static Analysis

Dynamic Analysis

Warm Fuzzy Feelings

Part I

Static Analysis

C++ Core Guidelines Checker

docs.microsoft.com/en-us/cpp/code-quality/quick-start-code-analysis-for-c-cpp

docs.microsoft.com/en-us/cpp/code-quality/code-analysis-for-cpp-corecheck

devblogs.microsoft.com/cppblog/new-safety-rules-in-c-core-check/

VS 16.7

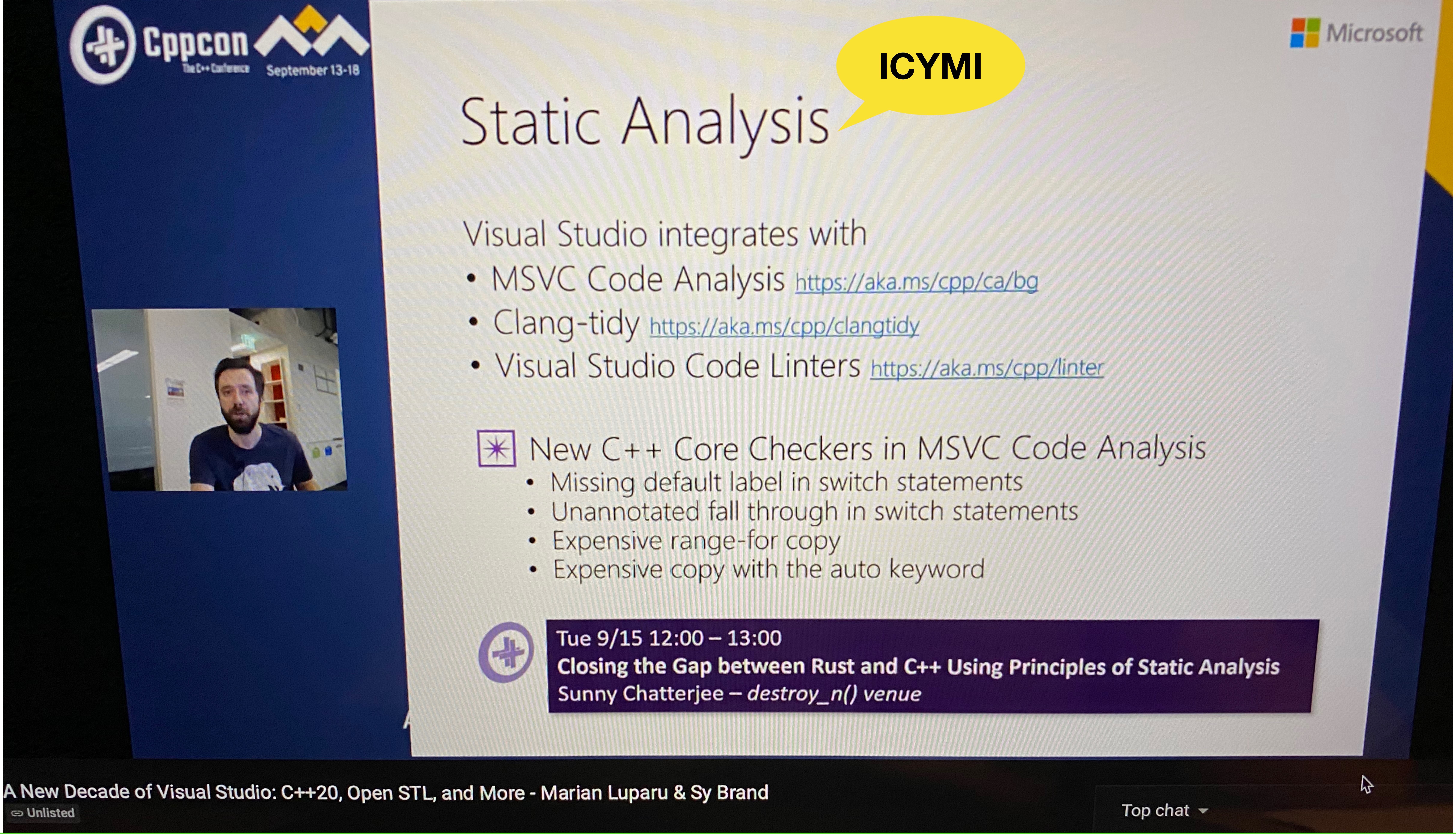
Standard C/C++ rule sets

Visual Studio includes these standard sets of rules for native code:

Rule Set	Description
C++ Core Check Arithmetic Rules	These rules enforce checks related to arithmetic operations from the C++ Core Guidelines.
C++ Core Check Bounds Rules	These rules enforce the Bounds profile of the C++ Core Guidelines.
C++ Core Check Class Rules	These rules enforce checks related to classes from the C++ Core Guidelines.
C++ Core Check Concurrency Rules	These rules enforce checks related to concurrency from the C++ Core Guidelines.
C++ Core Check Const Rules	These rules enforce const-related checks from the C++ Core Guidelines.
C++ Core Check Declaration Rules	These rules enforce checks related to declarations from the C++ Core Guidelines.
C++ Core Check Enum Rules	These rules enforce enum-related checks from the C++ Core Guidelines.
C++ Core Check Experimental Rules	These rules collect some experimental checks. Eventually, we expect these checks to be moved to other rulesets or removed completely.
C++ Core Check Function Rules	These rules enforce checks related to functions from the C++ Core Guidelines.
C++ Core Check GSL Rules	These rules enforce checks related to the Guidelines Support Library from the C++ Core Guidelines.

● ● ●

docs.microsoft.com/en-us/cpp/code-quality/code-analysis-for-cpp-corecheck


Cppcon
The C++ Conference
September 13-18
Microsoft
ICYMI
Static Analysis
Visual Studio integrates with
• MSVC Code Analysis https://aka.ms/cpp/ca/bg
• Clang-tidy https://aka.ms/cpp/clangtidy
• Visual Studio Code Linters https://aka.ms/cpp/linter
New C++ Core Checkers in MSVC Code Analysis
• Missing default label in switch statements
• Unannotated fall through in switch statements
• Expensive range-for copy
• Expensive copy with the auto keyword
Tue 9/15 12:00 – 13:00
Closing the Gap between Rust and C++ Using Principles of Static Analysis
Sunny Chatterjee – destroy_n() venue
A New Decade of Visual Studio: C++20, Open STL, and More - Marian Luparu & Sy Brand
Unlisted
Top chat

clang-tidy

~ 300 checks

clang.llvm.org/extra/clang-tidy/checks/list.html

clang-tidy

- modernize-use-nullptr
- modernize-loop-convert
- modernize-use-override
- readability-redundant-string-cstr
- modernize-use-emplace
- modernize-use-auto
- modernize-make-shared & modernize-make-unique
- modernize-use-equals-default & modernize-use-equals-delete

clang-tidy

- `modernize-use-default-member-init`
- `readability-redundant-member-init`
- `modernize-pass-by-value`
- `modernize-return-braced-init-list`
- `modernize-use-using`
- `cppcoreguidelines-pro-type-member-init`
- `readability-redundant-string-init & misc-string-constructor`
- `misc-suspicious-string-compare & misc-string-compare`
- `misc-inefficient-algorithm`
- `cppcoreguidelines-*`

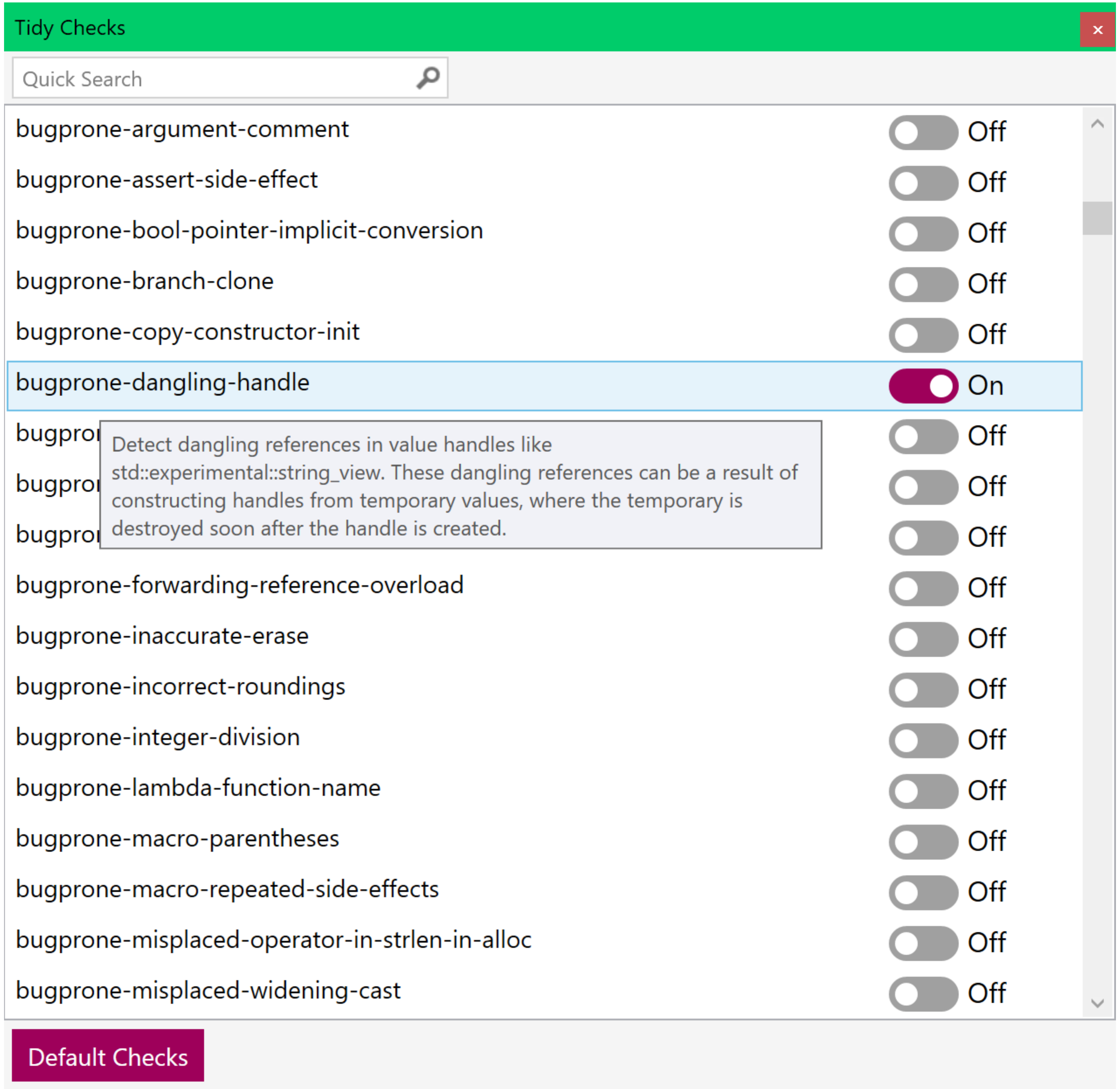
clang-tidy

- `abseil-string-find-startswith`
- `boost-use-to-string`
- `bugprone-string-constructor`
- `bugprone-string-integer-assignment`
- `bugprone-string-literal-with-embedded-nul`
- `bugprone-suspicious-string-compare`
- `modernize-raw-string-literal`
- `performance-faster-string-find`
- `performance-inefficient-string-concatenation`
- `readability-redundant-string-cstr`
- `readability-redundant-string-init`
- `readability-string-compare`

`string` checks

`clang-tidy`

checks



`clang-tidy` `bugprone-dangling-handle`

" Detect dangling references in value handles like `std::string_view`

These dangling references can be a result of constructing handles from ***temporary*** values, where the temporary is destroyed **soon** after the handle is created.

Options:

`HandleClasses`

A semicolon-separated list of class names that should be treated as handles. By default only `std::string_view` is considered.

https://clang.llvm.org/extra/clang-tidy/checks/bugprone-dangling-handle.html

`Lifetime profile v1.0`

Lifetime safety: Preventing common dangling

This is important because it turns out to be ***easy*** to convert **[by design]**

a `std::string` to a `std::string_view`,

or a `std::vector/array` to a `std::span`,

so that dangling is almost the default behavior.

CppCoreGuidelines

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf

Lifetime profile v1.0

Lifetime safety: Preventing common dangling

```
void example()
{
  std::string_view sv = std::string("dangling"); // A
  std::cout << sv;
}
```

`clang -Wlifetime` **Experimental**

CppCoreGuidelines

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf

Lifetime profile v1.0

Lifetime safety: Preventing common dangling

```
void example()
{
  std::string_view sv = std::string("dangling"); // A
  std::cout << sv;            // ERROR (lifetime.3): ‘sv’ was invalidated when
}                             // temporary was destroyed (line A)
```

`clang -Wlifetime` **Experimental**

CppCoreGuidelines

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf

Lifetime safety: Preventing common dangling

`[-Wdangling-gsl]` **diagnosed by default in Clang 10**

warning: initializing pointer member to point to a temporary object whose lifetime is shorter than the lifetime of the constructed object

```
void example()
{
  std::string_view sv = std::string("dangling");


  std::cout << sv;
}
```

https://clang.llvm.org/docs/DiagnosticsReference.html#wdangling-gsl

Lifetime safety: Preventing common dangling

[-Wdangling-gsl] **diagnosed <u>by default</u> in Clang 10**

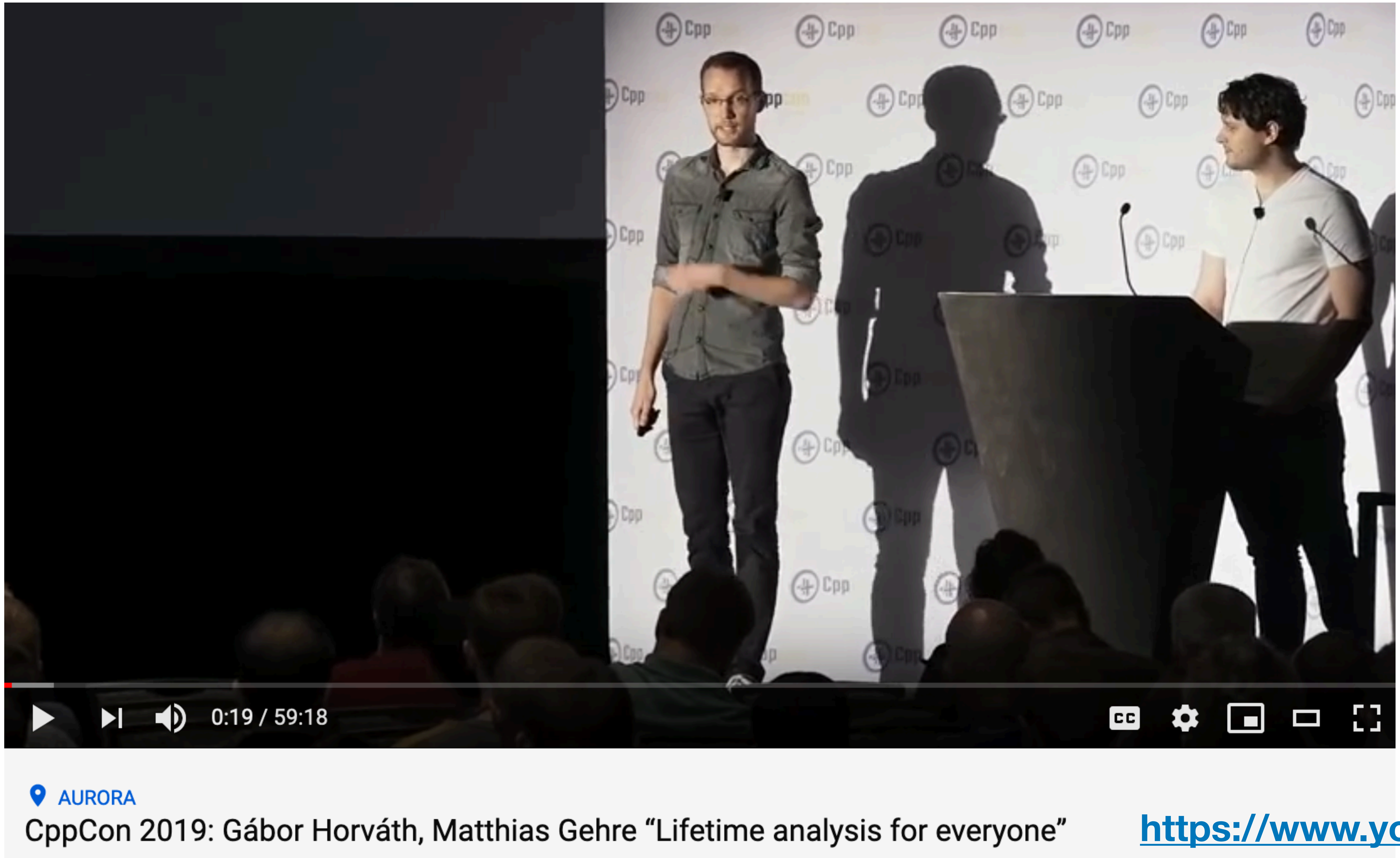
warning: initializing pointer member to point to a temporary object whose lifetime is shorter than the lifetime of the constructed object

```
void example()
{
  std::string_view sv = std::string("dangling");
                 // warning: object backing the pointer will be destroyed
                 // at the end of the full-expression [-Wdangling-gsl]
  std::cout << sv;
}
```

https://clang.llvm.org/docs/DiagnosticsReference.html#wdangling-gsl

Lifetime profile

https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetime.pdf



https://www.youtube.com/watch?v=d67kfSnhbpA

`clang-tidy`

Checks are organized in **modules**, which can be linked into clang-tidy with minimal or no code changes in clang-tidy

`clang-tidy`

Checks are organized in **modules**, which can be linked into clang-tidy with minimal or no code changes in clang-tidy

Checks can plug into the analysis on the **preprocessor** level using **PPCallbacks** or on the AST level using **AST Matchers**

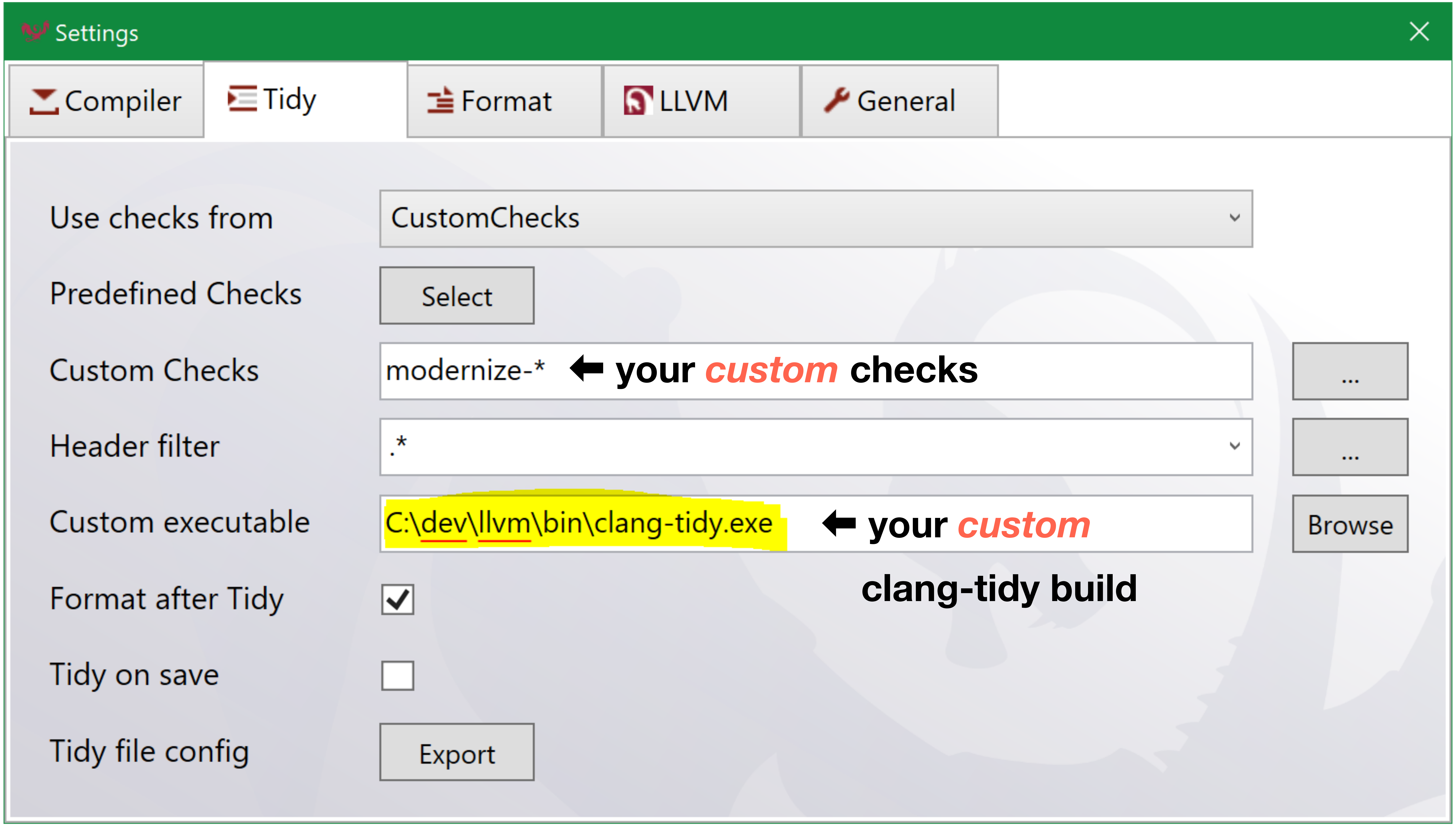
clang-tidy

Checks are organized in **modules**, which can be linked into clang-tidy
with minimal or no code changes in clang-tidy

Checks can plug into the analysis on the **preprocessor** level using **PPCallbacks**
or on the AST level using **AST Matchers**

Checks can **report** issues in a similar way to how Clang diagnostics work.
A **fix-it** hint can be attached to a diagnostic message

Custom clang-tidy checks



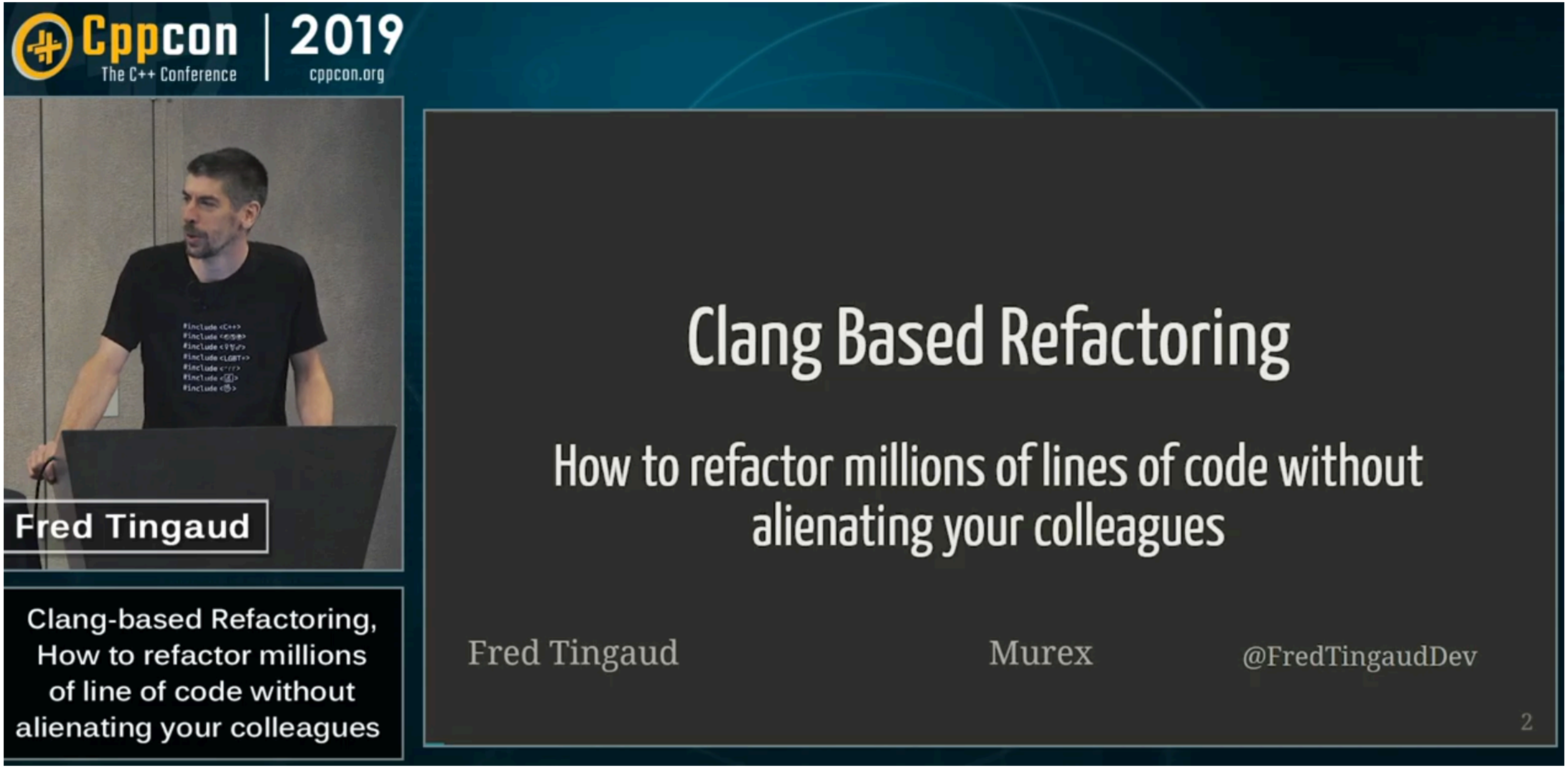
Write *custom* checks for your needs
(project specific)

Run them regularly !

Explore Further

https://steveire.wordpress.com/2019/01/02/refactor-with-clang-tooling-at-codedive-2018/

Explore Further



https://www.youtube.com/watch?v=JPnN2c2odNY

www.youtube.com/watch?v=Iz4C29yul2U

Explore Further

A new series of blog articles on **Visual C++ Team blog** by **Stephen Kelly**

Exploring Clang Tooling, Part 0: Building Your Code with Clang

https://blogs.msdn.microsoft.com/vcblog/2018/09/18/exploring-clang-tooling-part-0-building-your-code-with-clang/

Exploring Clang Tooling, Part 1: Extending Clang-Tidy

https://blogs.msdn.microsoft.com/vcblog/2018/10/19/exploring-clang-tooling-part-1-extending-clang-tidy/

Exploring Clang Tooling, Part 2: Examining the Clang AST with clang-query

https://blogs.msdn.microsoft.com/vcblog/2018/10/23/exploring-clang-tooling-part-2-examining-the-clang-ast-with-clang-query/

Explore Further

A new series of blog articles on **Visual C++ Team blog** by **Stephen Kelly**

Exploring Clang Tooling, Part 3: Rewriting Code with clang-tidy

https://blogs.msdn.microsoft.com/vcblog/2018/11/06/exploring-clang-tooling-part-3-rewriting-code-with-clang-tidy/

Exploring Clang Tooling: Using Build Tools with clang-tidy

https://blogs.msdn.microsoft.com/vcblog/2018/11/27/exploring-clang-tooling-using-build-tools-with-clang-tidy/

Explore Further

More blog articles by **Stephen Kelly**

Future Developments in clang-query

https://steveire.wordpress.com/2018/11/11/future-developments-in-clang-query/

Composing AST Matchers in clang-tidy

https://steveire.wordpress.com/2018/11/20/composing-ast-matchers-in-clang-tidy/

Visual Studio 2019
since v16.2

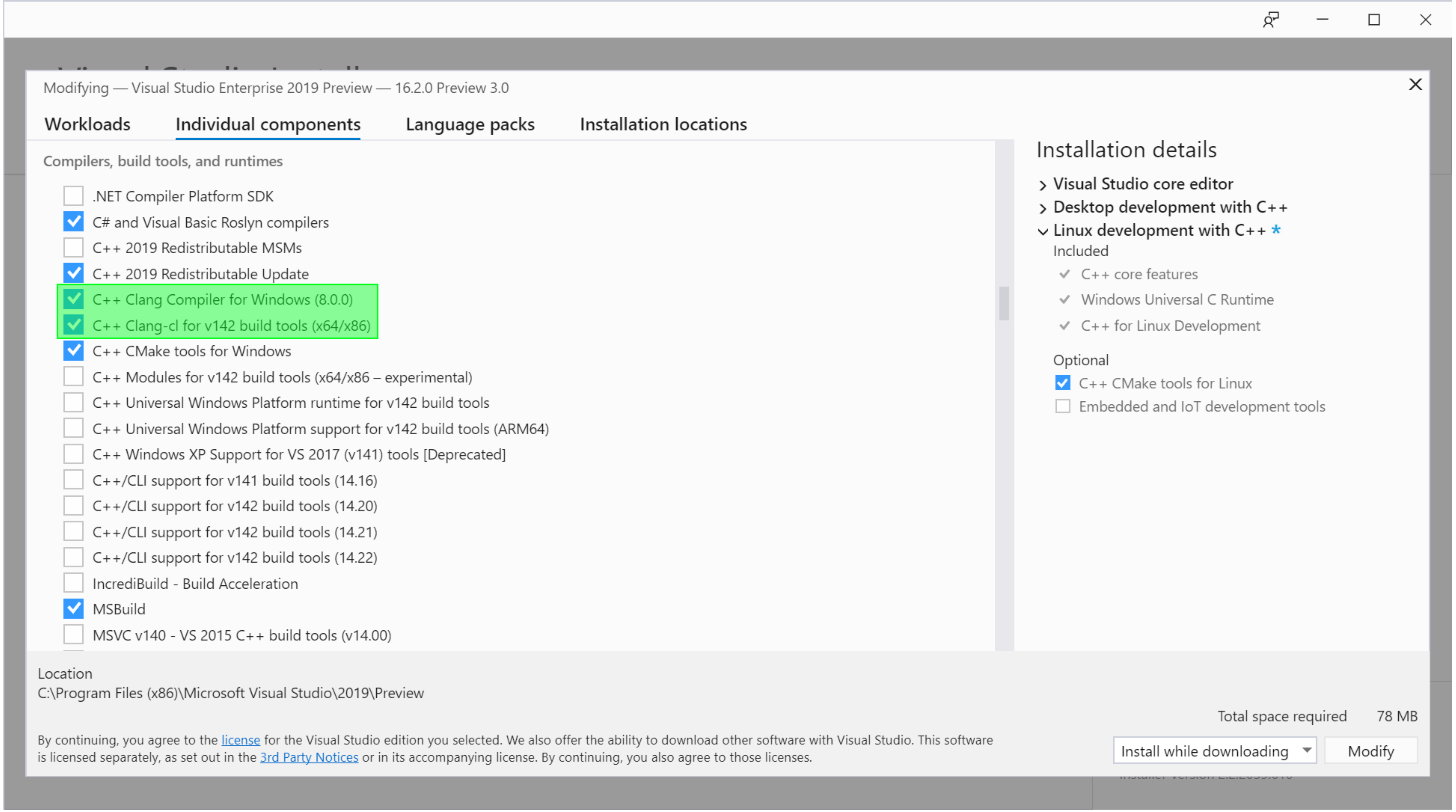
`Clang/LLVM` support for `MSBuild` & `CMake` Projects

Ships with `Clang` (as optional component)

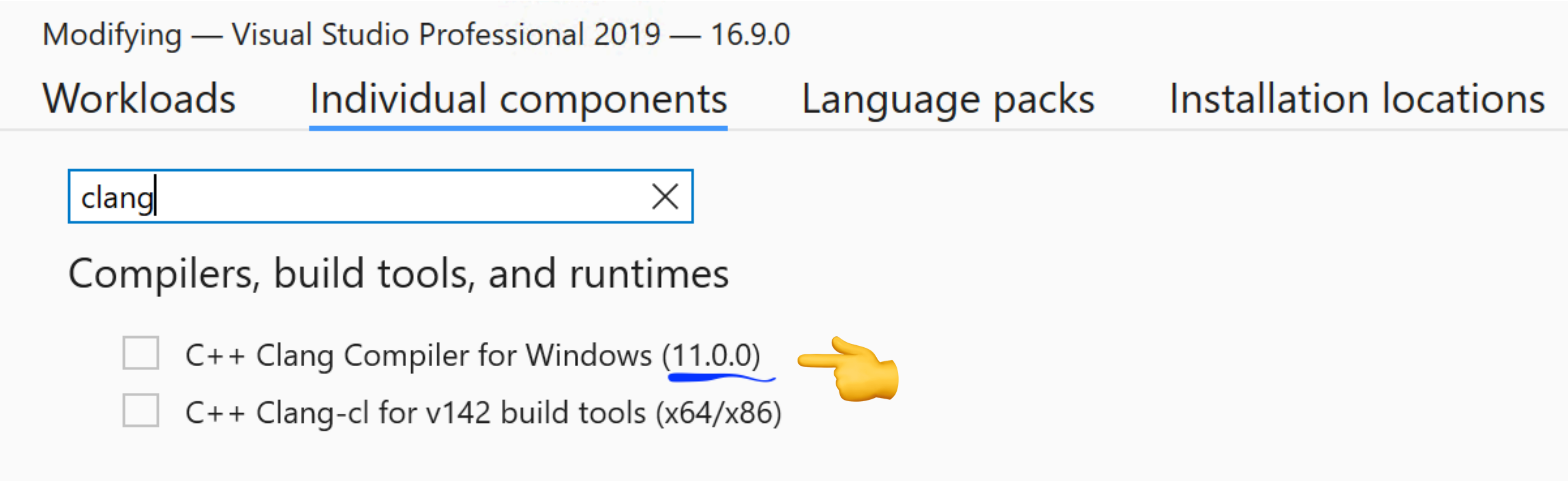
`clang-cl.exe`

📖 https://devblogs.microsoft.com/cppblog/clang-llvm-support-for-msbuild-projects/

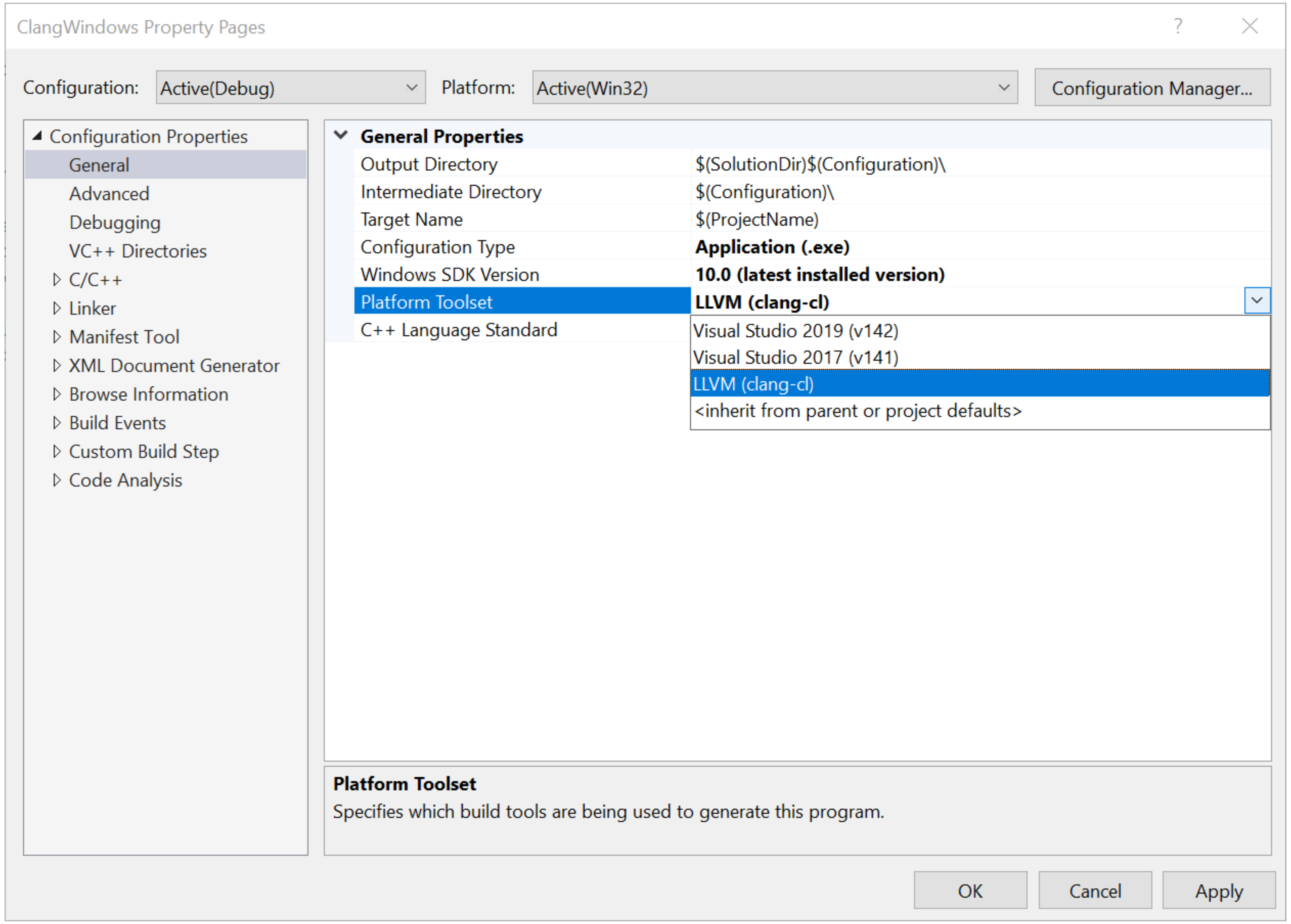
Visual Studio 2019
since v16.2



Visual Studio 2019
v16.9



Visual Studio 2019
since v16.2



`clang-cl.exe`

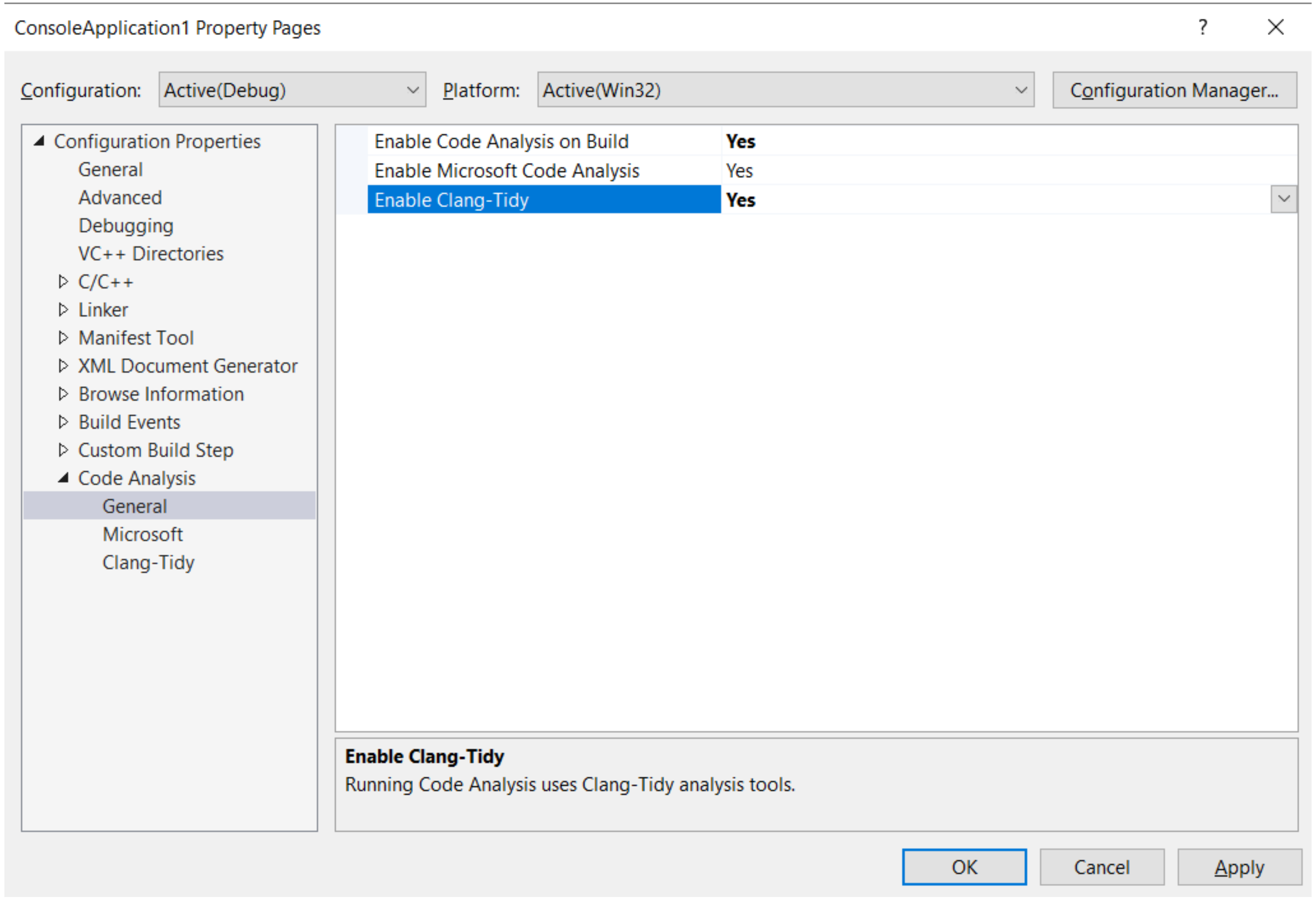
Visual Studio 2019
since v16.4

`clang-tidy`

code analysis

📖 https://devblogs.microsoft.com/cppblog/code-analysis-with-clang-tidy-in-visual-studio/

Visual Studio 2019
since v16.4



Visual Studio 2019
since v16.4

`clang-tidy` **warnings**

Error List

Entire Solution | 0 Errors | 10 Warnings | 0 Messages | Build + IntelliSense

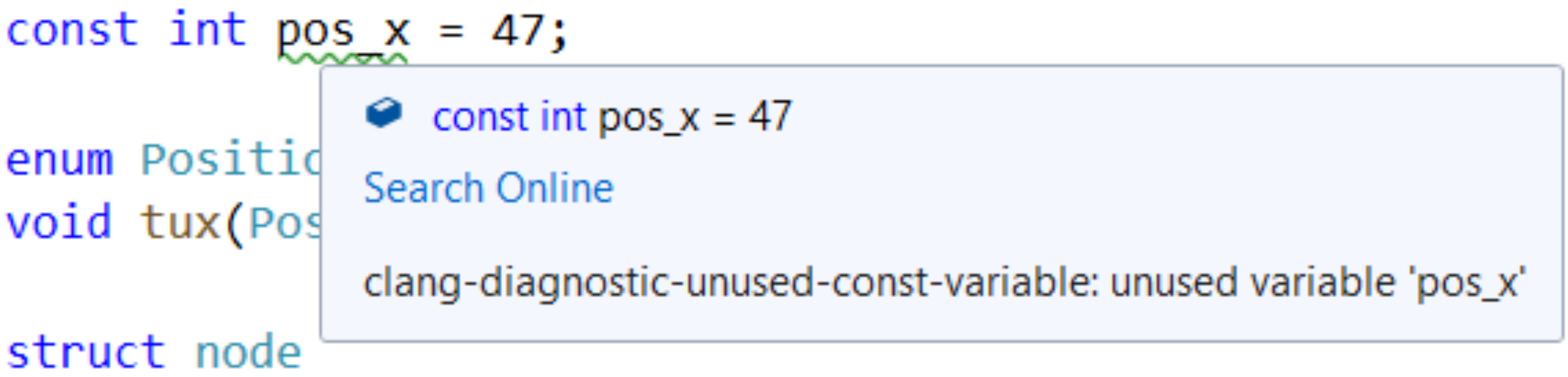
Code	Description	File	Line	Col	Category
readability-isolate-declaration	multiple declarations in a single statement reduces readability	CMAKEDEMO.CPP	23	2	readability
modernize-use-nullptr	use nullptr	CMAKEDEMO.CPP	31	7	modernize
cppcoreguidelines-macro-usage	macro 'TRUE' used to declare a constant; consider using a 'constexpr' constant	CMAKEDEMO.CPP	35	9	cppcoreguidelines
clang-diagnostic-unused-variable	unused variable 'local'	CMAKEDEMO.CPP	50	13	clang-diagnostic
clang-diagnostic-unused-const-variable	unused variable 'pos_x'	CMAKEDEMO.CPP	36	11	clang-diagnostic
clang-diagnostic-uninitialized	variable 'numLives' is uninitialized when used here	CMAKEDEMO.CPP	24	3	clang-diagnostic
clang-diagnostic-return-type	control reaches end of non-void function	CMAKEDEMO.CPP	32	1	clang-diagnostic
clang-analyzer-core.NullDereference	Dereference of undefined pointer value	CMAKEDEMO.CPP	24	12	clang-analyzer

Error List Output

https://devblogs.microsoft.com/cppblog/code-analysis-with-clang-tidy-in-visual-studio/

Visual Studio 2019
since v16.4

`clang-tidy` **warnings also display as in-editor squiggles**



Code Analysis runs automatically in the background

NOT on

Visual Studio 2019 v16.4+ yet ?

No problem

= ->

Free/OSS

Clang Power Tools

www.clangpowertools.com

LLVM

clang-tidy
clang++
clang-format
clang-check/query

Visual Studio

2015 / 2017 / 2019

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!

Part II

Dynamic Analysis

ICYMI

Control Flow Guard

`/guard:cf`

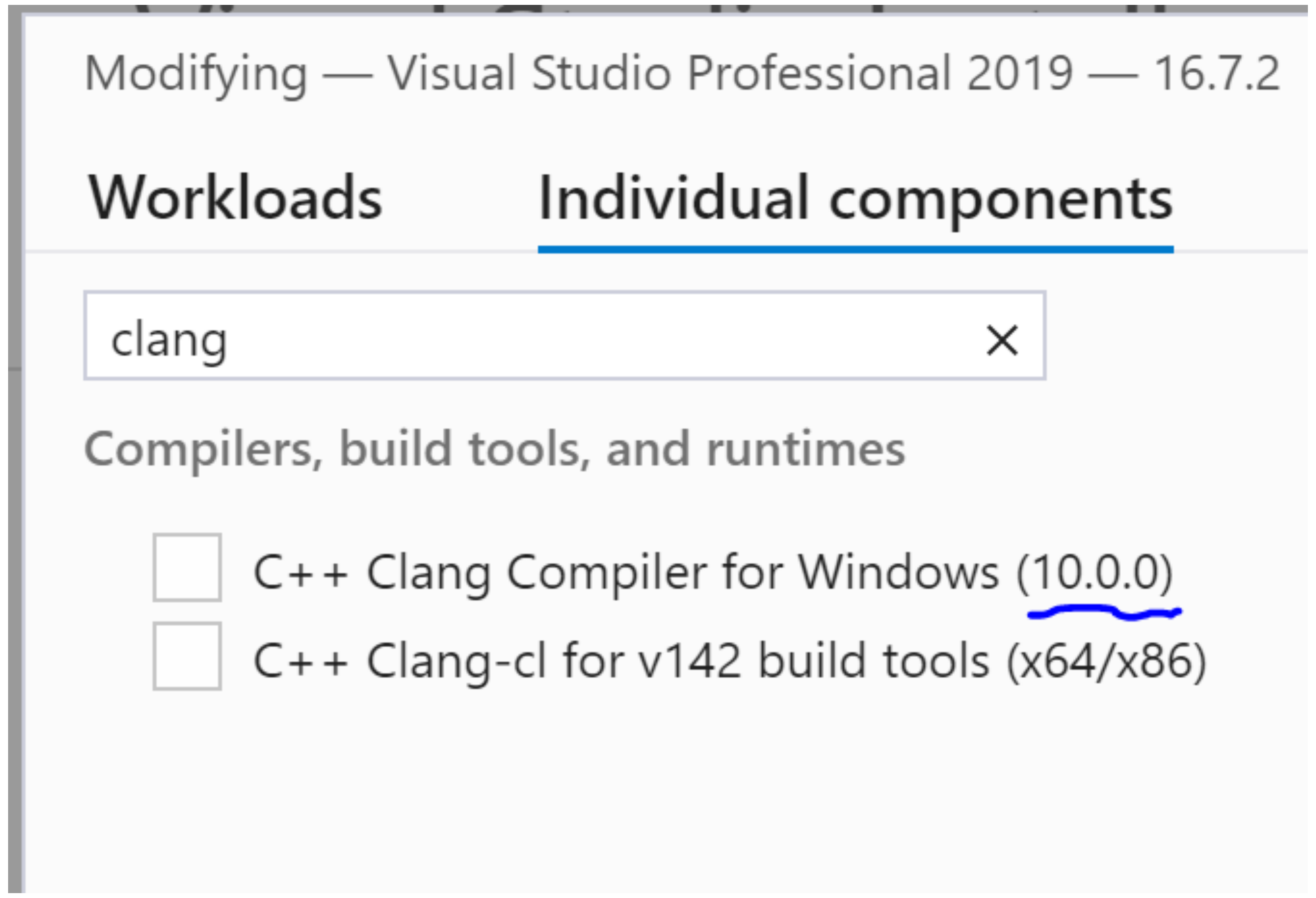
Enforce control flow integrity (Windows 8.1 & Windows 10)

CFG is complementary to other exploit mitigations, such as:

- Address Space Layout Randomization (**ASLR**)
- Data Execution Prevention (**DEP**)

MSVC

CFG is now supported in **LLVM 10+**



C++ & Rust

https://aka.ms/cpp/cfg-llvm

Sanitizers

Sanitizers

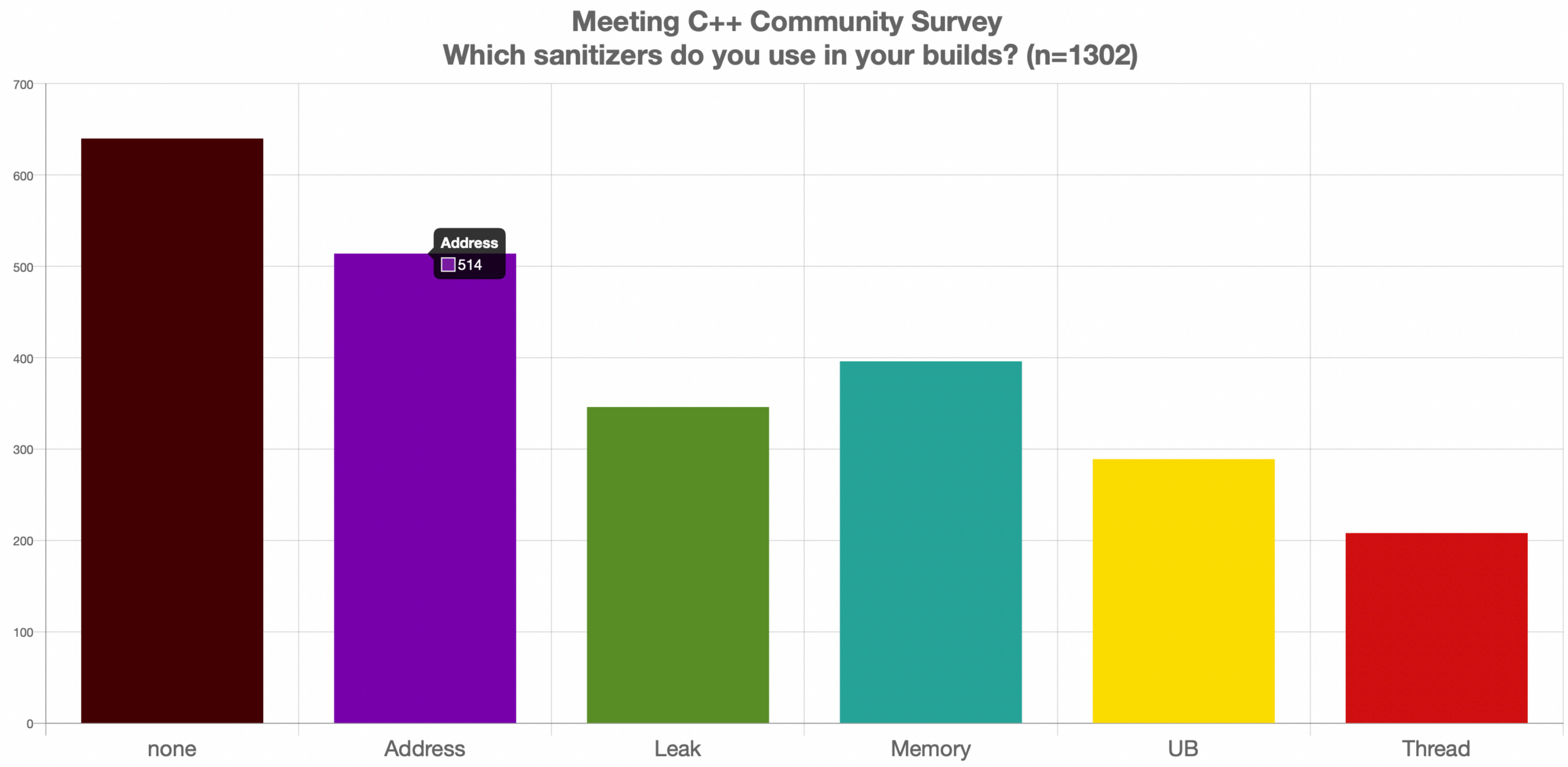
- `AddressSanitizer` - detects addressability issues
- `LeakSanitizer` - detects memory leaks
- `ThreadSanitizer` - detects data races and deadlocks
- `MemorySanitizer` - detects use of uninitialized memory
- `HWASAN` - hardware-assisted AddressSanitizer (consumes less memory)
- `UBSan` - detects Undefined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

github.com/google/sanitizers

Meeting C++ Community Survey

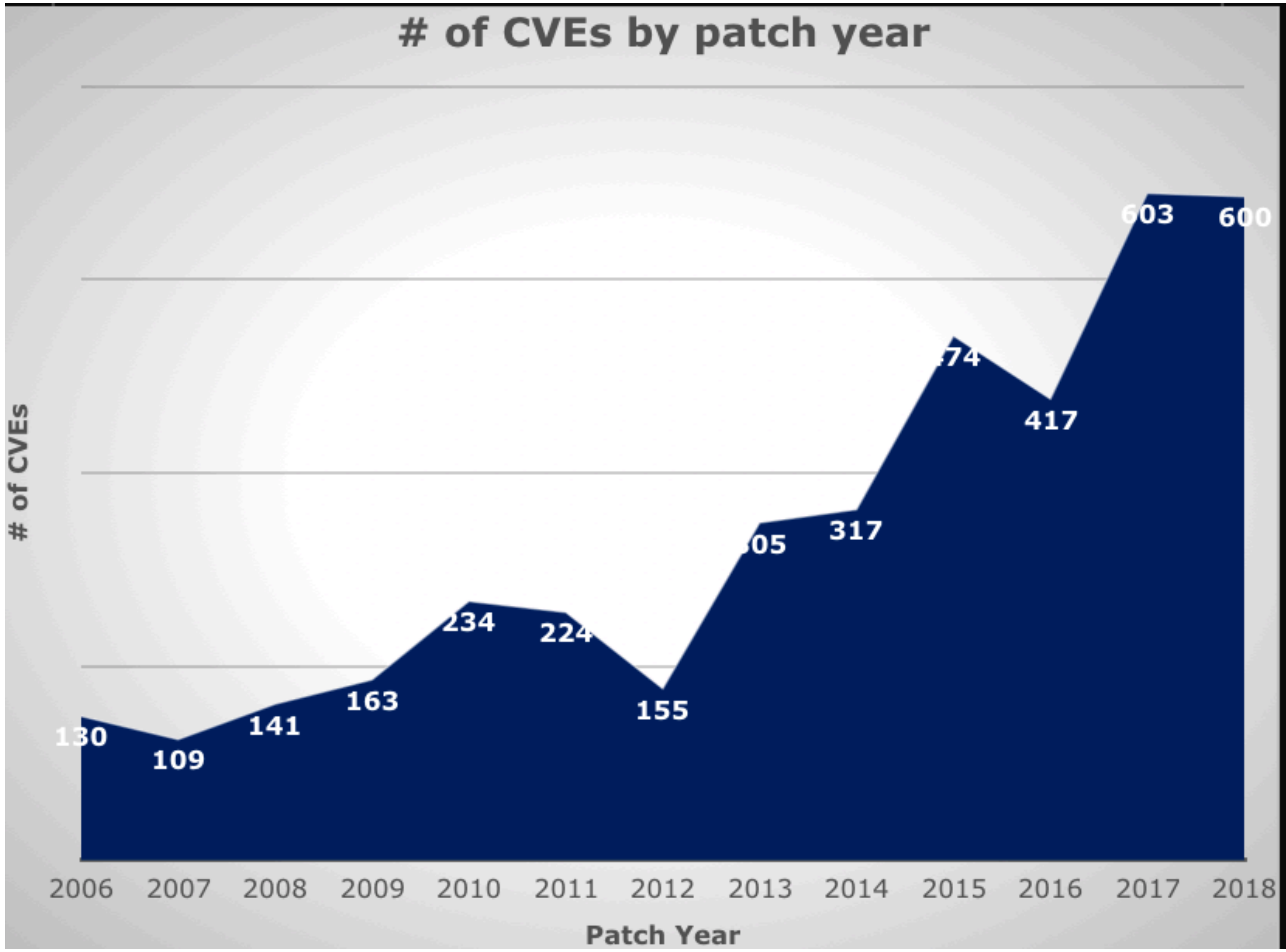
meetingcpp.com/mcpp/survey/?q=19

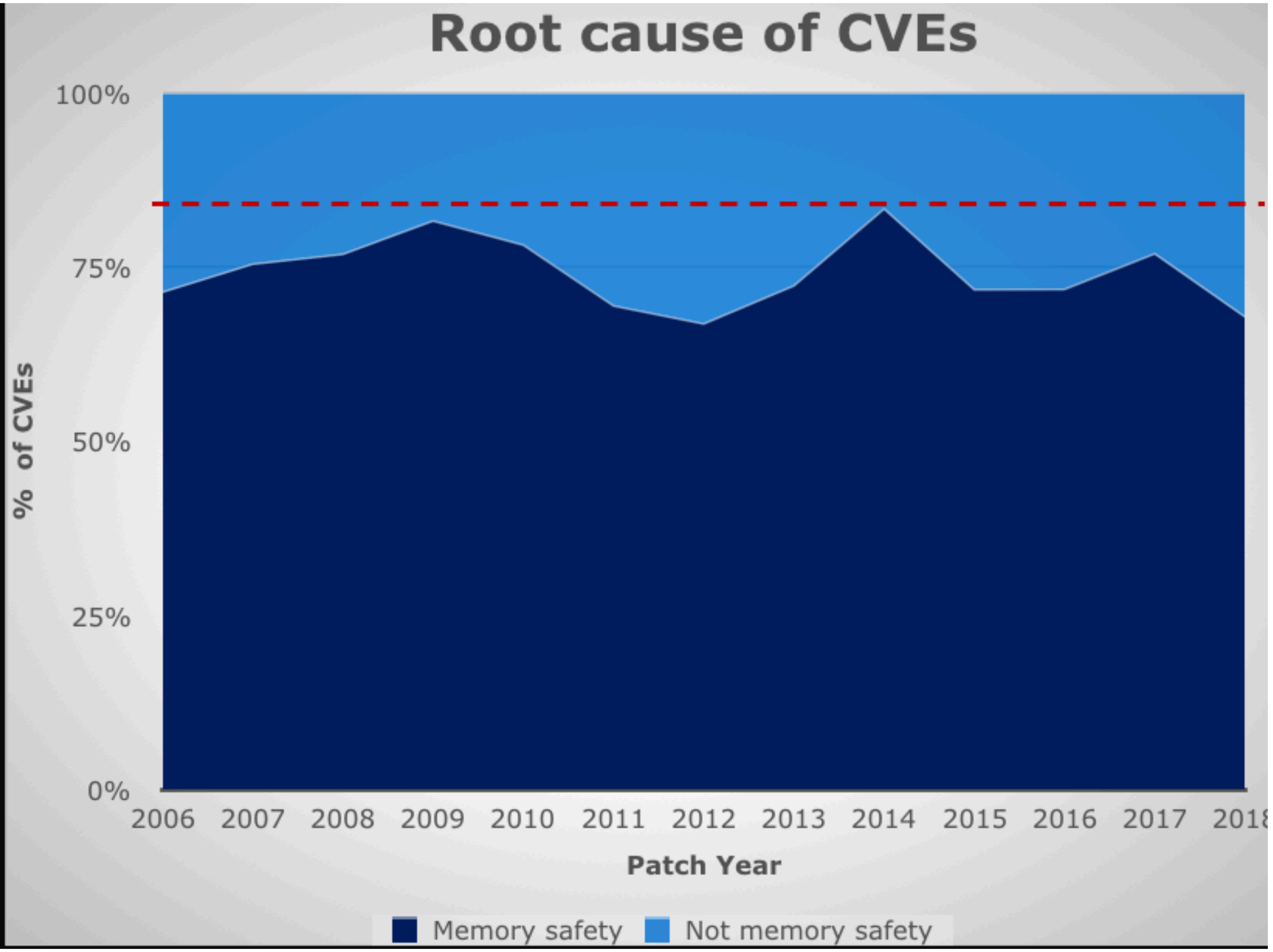
Next Question | Survey results



Common Vulnerabilities and Exposures

Memory safety continues to dominate





youtube.com/watch?v=0EsqxGgYOQU

Address Sanitizer (ASan)

De facto standard for detecting memory safety issues

It's important for basic correctness and sometimes true vulnerabilities

github.com/google/sanitizers/wiki/AddressSanitizer

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** (late 2019, exp.)

Visual Studio 2019
since v16.4

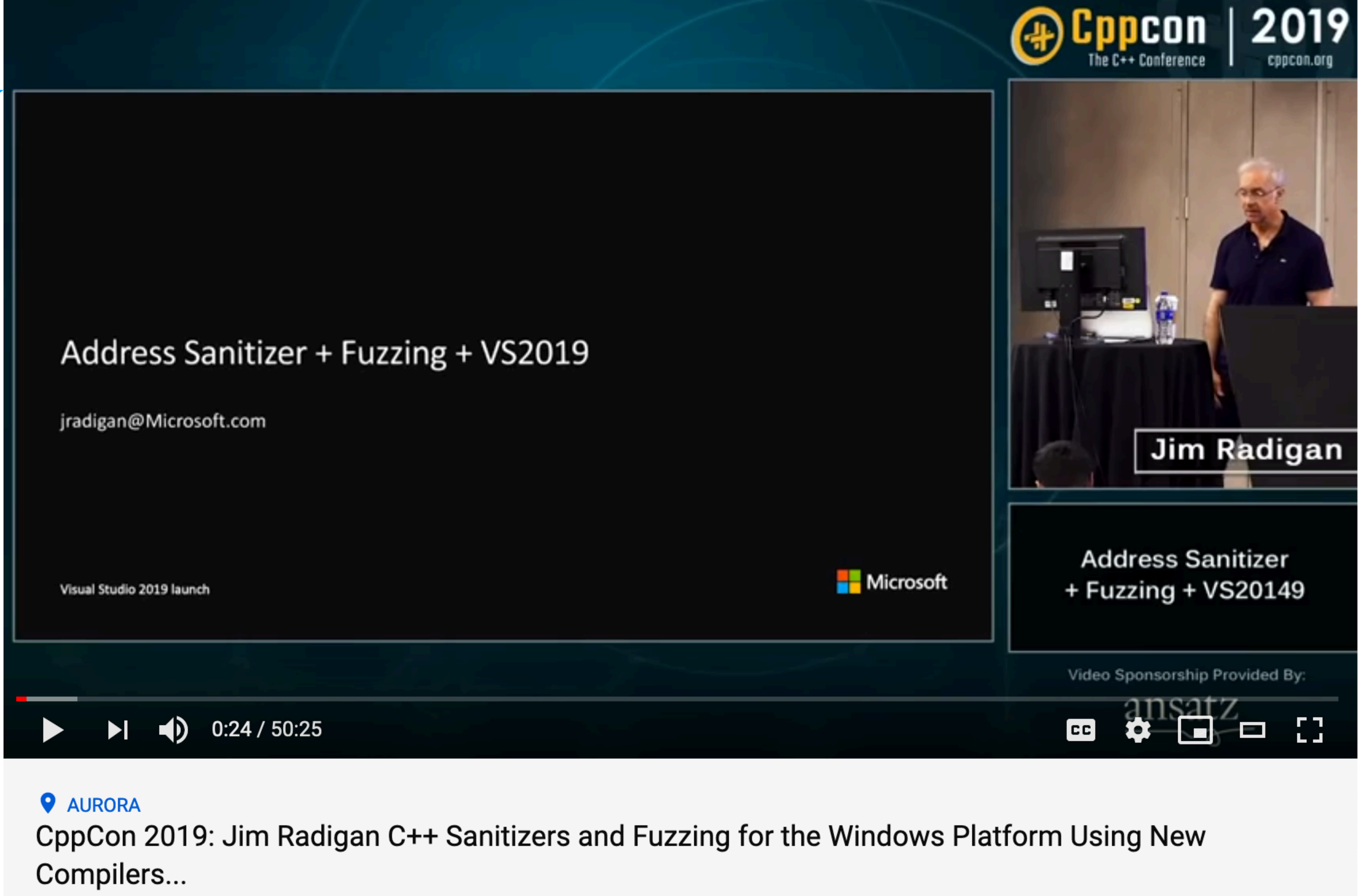
October 2019

Address Sanitizer
(ASan)

🎉

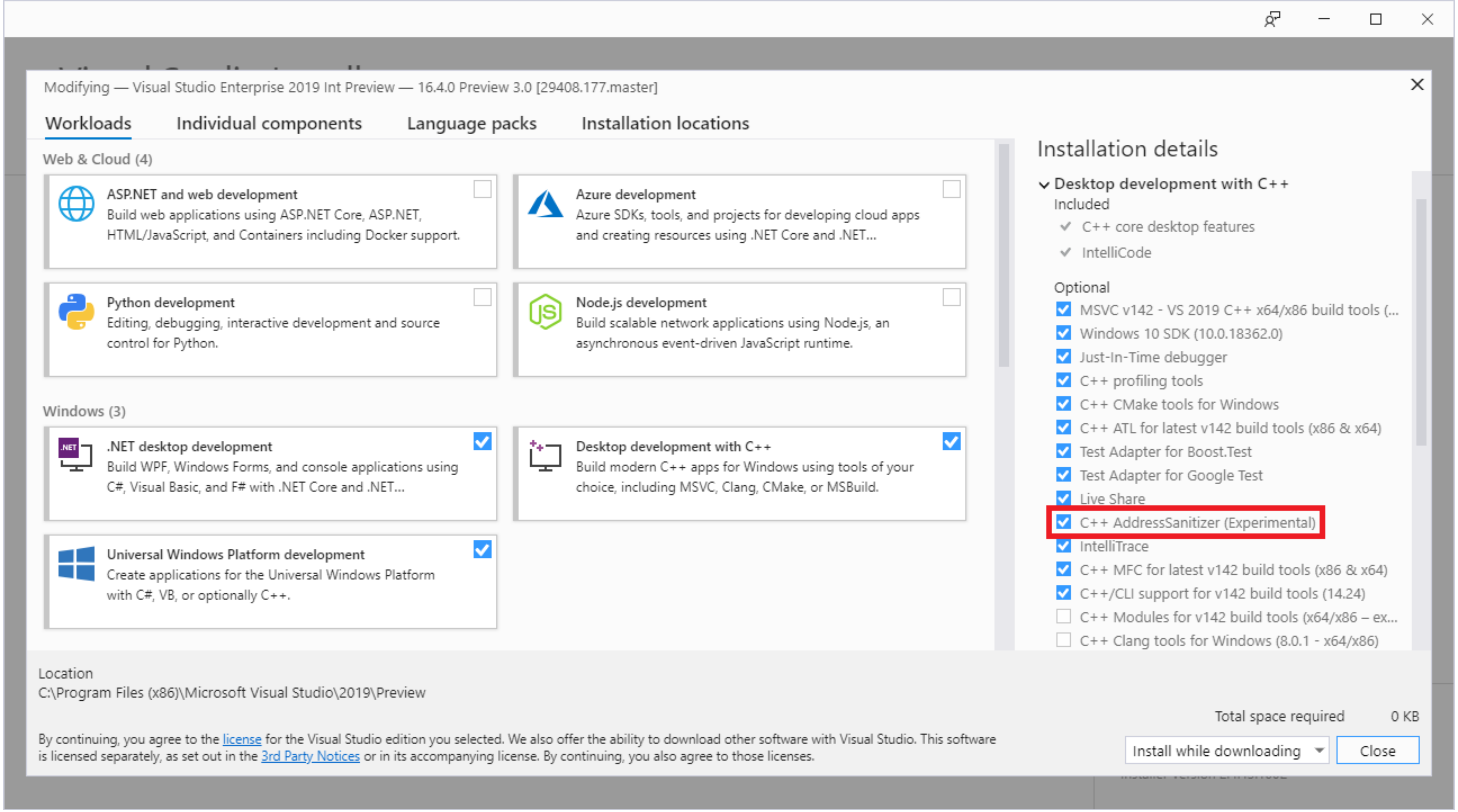
📖 devblogs.microsoft.com/cppblog/addresssanitizer-asan-for-windows-with-msvc/

sneak peek

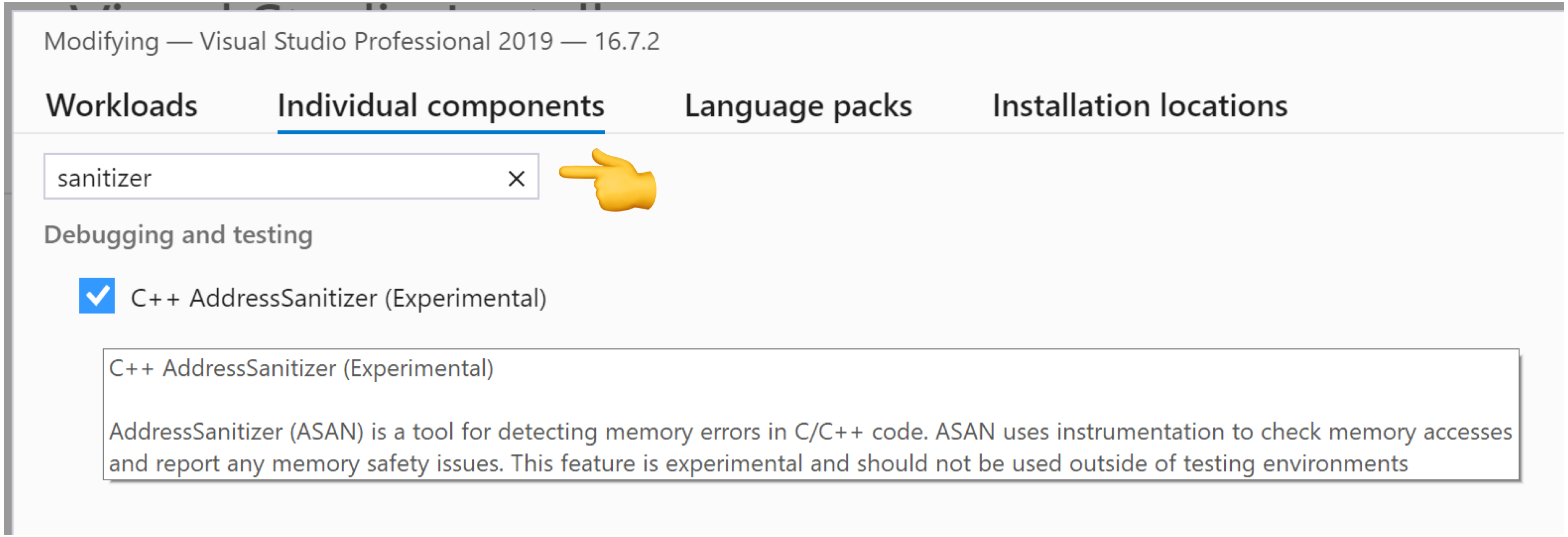


https://www.youtube.com/watch?v=0EsqxGgYOQU

Visual Studio 2019
since v16.4

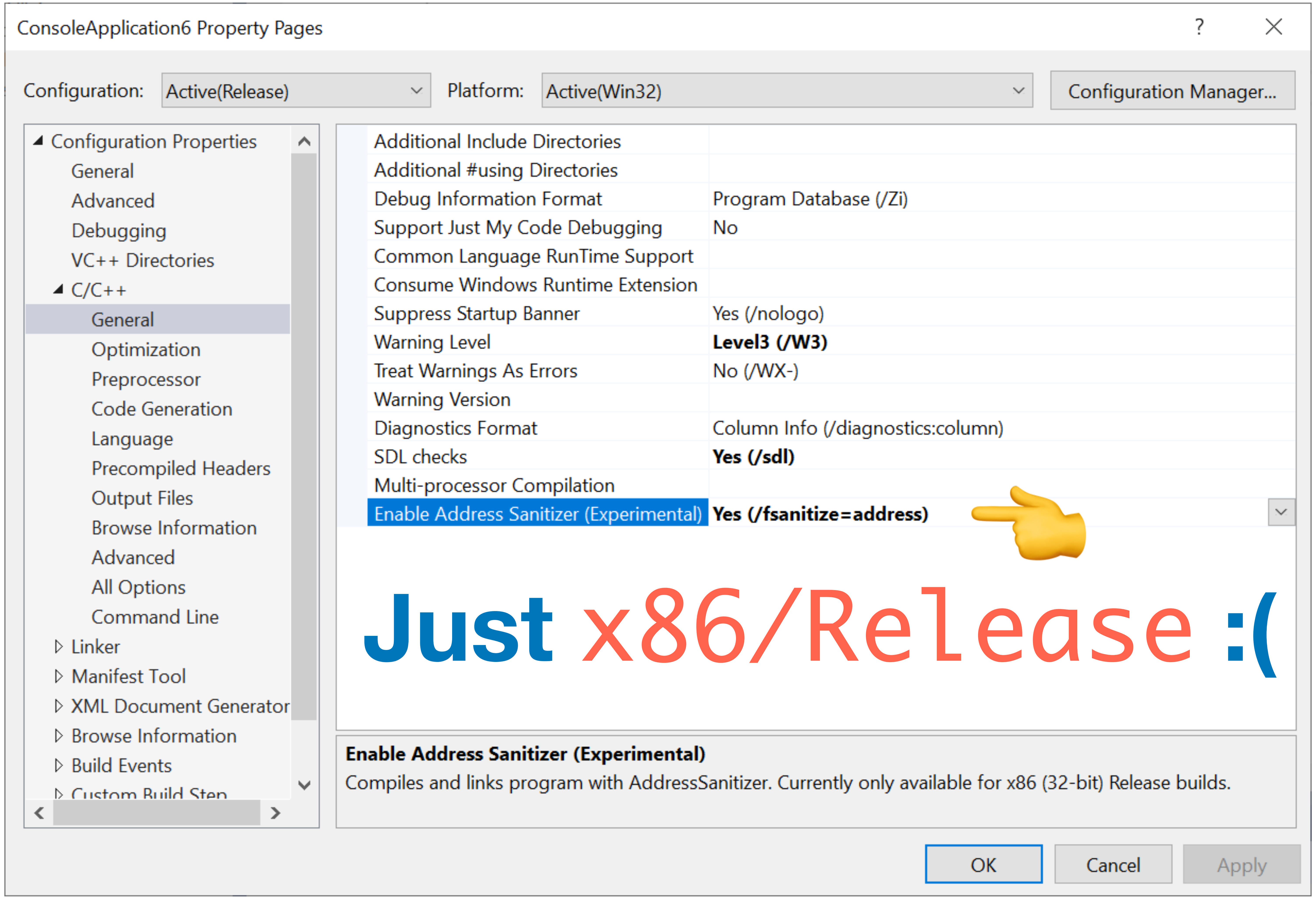


Visual Studio 2019
since v16.4



Visual Studio 2019

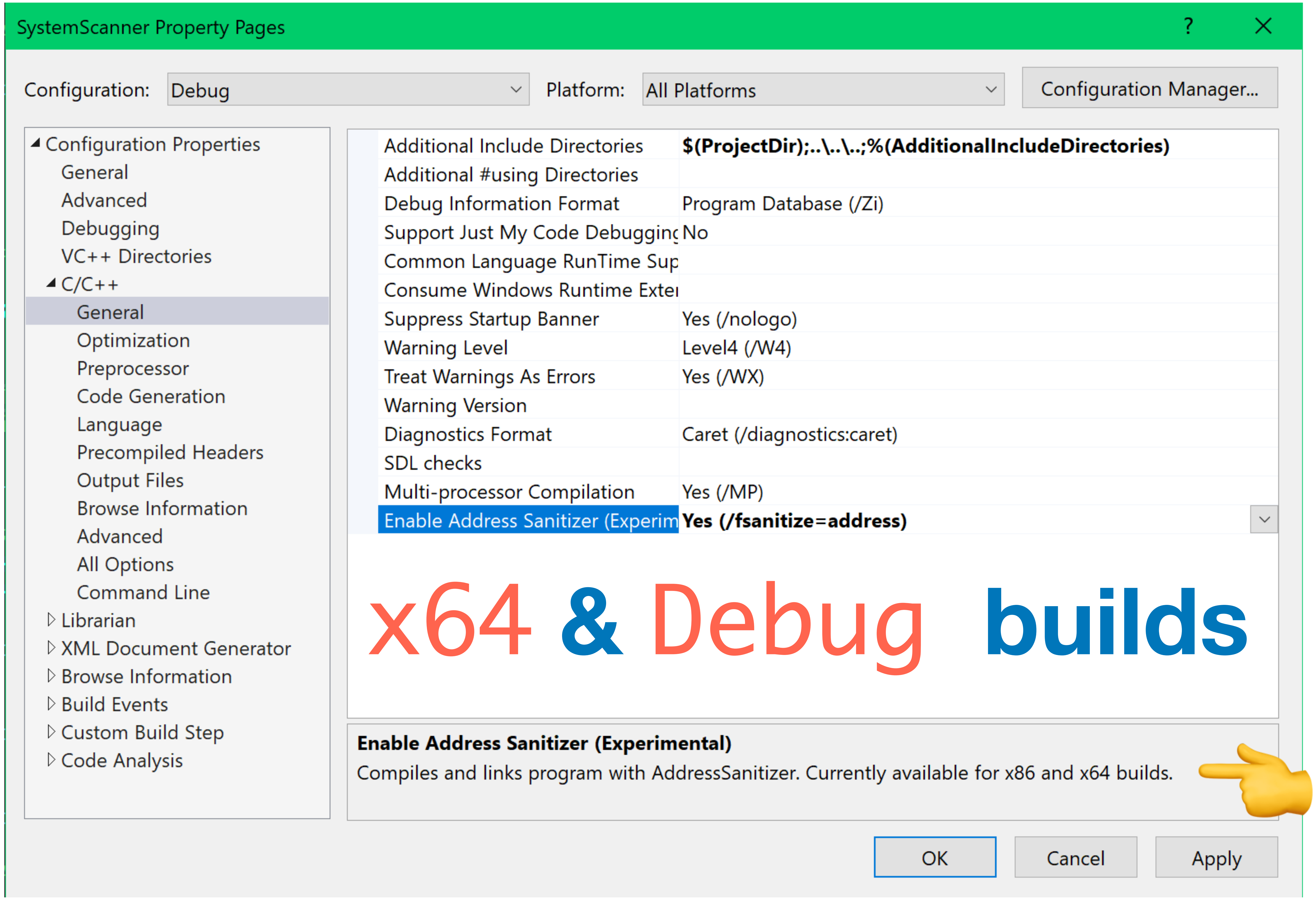
since v16.4



Tech Preview

October 2019

Visual Studio 2019
since v16.7



Tech Preview

August 2020

Visual Studio 2019

since v16.7 August 2020

+ x64 & Debug builds

support all Debug runtimes: `/MTd` `/MDd`

Tech Preview

docs.microsoft.com/en-us/visualstudio/releases/2019/release-notes#16.7.0

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`

Visual Studio 2019
v16.8-9

New 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)

NEW

March 2021

Visual Studio 2019
v16.9

Available today:

Visual Studio 2019 v16.9 and v16.10 Preview

- Address Sanitizer support for Windows
- C++ conformance
- Improved call stack handling
- New memory dump analyzers
- Improvements to GitHub Actions tooling
- .NET productivity enhancements

Learn what's new

devblogs.microsoft.com/visualstudio/vs2019-v16-9-and-v16-10-preview-1/

NEW

March 2021

Visual Studio 2019
v16.9

`ASAN is out of Experimental => GA`

🎉

devblogs.microsoft.com/cppblog/address-sanitizer-for-msvc-now-generally-available

NEW

March 2021

Visual Studio 2019

v16.9

NEW

March 2021

Visual Studio 2019
v16.9

- expanded `RtlAllocateHeap` support (fixed compatibility issue with `RtlCreateHeap` and `RtlAllocateHeap` interceptors when creating executable memory pools)

NEW

March 2021

Visual Studio 2019
v16.9

- expanded `RtlAllocateHeap` support (fixed compatibility issue with `RtlCreateHeap` and `RtlAllocateHeap` interceptors when creating executable memory pools)
- support for the legacy `GlobalAlloc` and `LocalAlloc` family of memory functions (`ASAN_OPTIONS=windows_hook_legacy_allocators=true`)

NEW

March 2021

Visual Studio 2019
v16.9

- expanded `RtlAllocateHeap` support (fixed compatibility issue with `RtlCreateHeap` and `RtlAllocateHeap` interceptors when creating executable memory pools)
- 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

NEW

March 2021

Visual Studio 2019 v16.9

- expanded `RtlAllocateHeap` support (fixed compatibility issue with `RtlCreateHeap` and `RtlAllocateHeap` interceptors when creating executable memory pools)
- 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

NEW

March 2021

Visual Studio 2019
v16.9

- expanded `RtlAllocateHeap` support (fixed compatibility issue with `RtlCreateHeap` and `RtlAllocateHeap` interceptors when creating executable memory pools)
- 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

NEW

March 2021

- AddressSanitizer
  - **AddressSanitizer overview**
  - Build and language reference
  - Runtime reference
  - Debugger integration
  - Shadow bytes
  - Cloud or distributed testing
  - AddressSanitizer error examples
    - AddressSanitizer error examples
    - alloc-dealloc-mismatch error
    - allocation-size-too-big error
    - calloc-overflow error
    - double-free error
    - dynamic-stack-buffer-overflow error
    - global-buffer-overflow error
    - heap-buffer-overflow error
    - heap-use-after-free error
    - invalid-allocation-alignment error
    - memcpy-param-overlap error
    - new-delete-type-mismatch error
    - stack-buffer-overflow error

AddressSanitizer

03/05/2021 • 7 minutes to read •

Overview

The C & C++ languages are powerful, but can suffer from a class of bugs that affect program correctness and program security. Starting in Visual Studio 2019 version 16.9, the Microsoft C/C++ compiler (MSVC) and IDE supports the *AddressSanitizer*. AddressSanitizer (ASan) is a compiler and runtime technology that exposes many hard-to-find bugs with **zero** false positives:

- Alloc/dealloc mismatches and new/delete type mismatches
- Allocations too large for the heap
- calloc overflow and alloca overflow
- Double free and use after free
- Global variable overflow
- Heap buffer overflow
- Invalid alignment of aligned values
- memcpy and strncat parameter overlap
- Stack buffer overflow and underflow
- Stack use after return and use after scope
- Memory use after it's poisoned

Use AddressSanitizer to reduce your time spent on:

- Basic correctness
- Cross platform portability
- Security
- Stress testing
- Integrating new code

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

Visual Studio ASan

Very tall order to bring ASAN to Windows

😅

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

non-standard C++

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

non-standard C++

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

non-standard C++

Structured Exception Handling (SEH) `/EHa`

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

non-standard C++

Structured Exception Handling (SEH) `/EHa`

AV traps `0xc0000005`

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

non-standard C++

Structured Exception Handling (SEH) `/EHa`

AV traps `0xc0000005`

vast amount of legacy code (really, really, really OLD code)

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

non-standard C++

Structured Exception Handling (SEH) `/EHa`

AV traps `0xc0000005`

vast amount of legacy code (really, really, really OLD code)

COM

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

non-standard C++

Structured Exception Handling (SEH) `/EHa`

AV traps `0xc0000005`

vast amount of legacy code (really, really, really OLD code)

COM

Managed C++

Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

non-standard C++

Structured Exception Handling (SEH) `/EHa`

AV traps `0xc0000005`

vast amount of legacy code (really, really, really OLD code)

COM

Managed C++

ASan runtime interop with managed code (.NET)

Visual Studio ASan

"Thank you" to `Microsoft` team

tirelessly working on this

🙏

Everyone will continue to invest heavily in this area (sanitizers)

just because it’s **so effective** at just finding correctness issues

Microsoft is contributing back to LLVM
all the work they've done to make ASan runtime work on Windows

github.com/llvm/llvm-project/tree/master/compiler-rt

Visual Studio 2019

ASan Visual Studio integration:

- **MSBuild** & **CMake** support for both Windows & Linux
- **Debugger** integration for MSVC and Clang/LLVM

aka.ms/asan

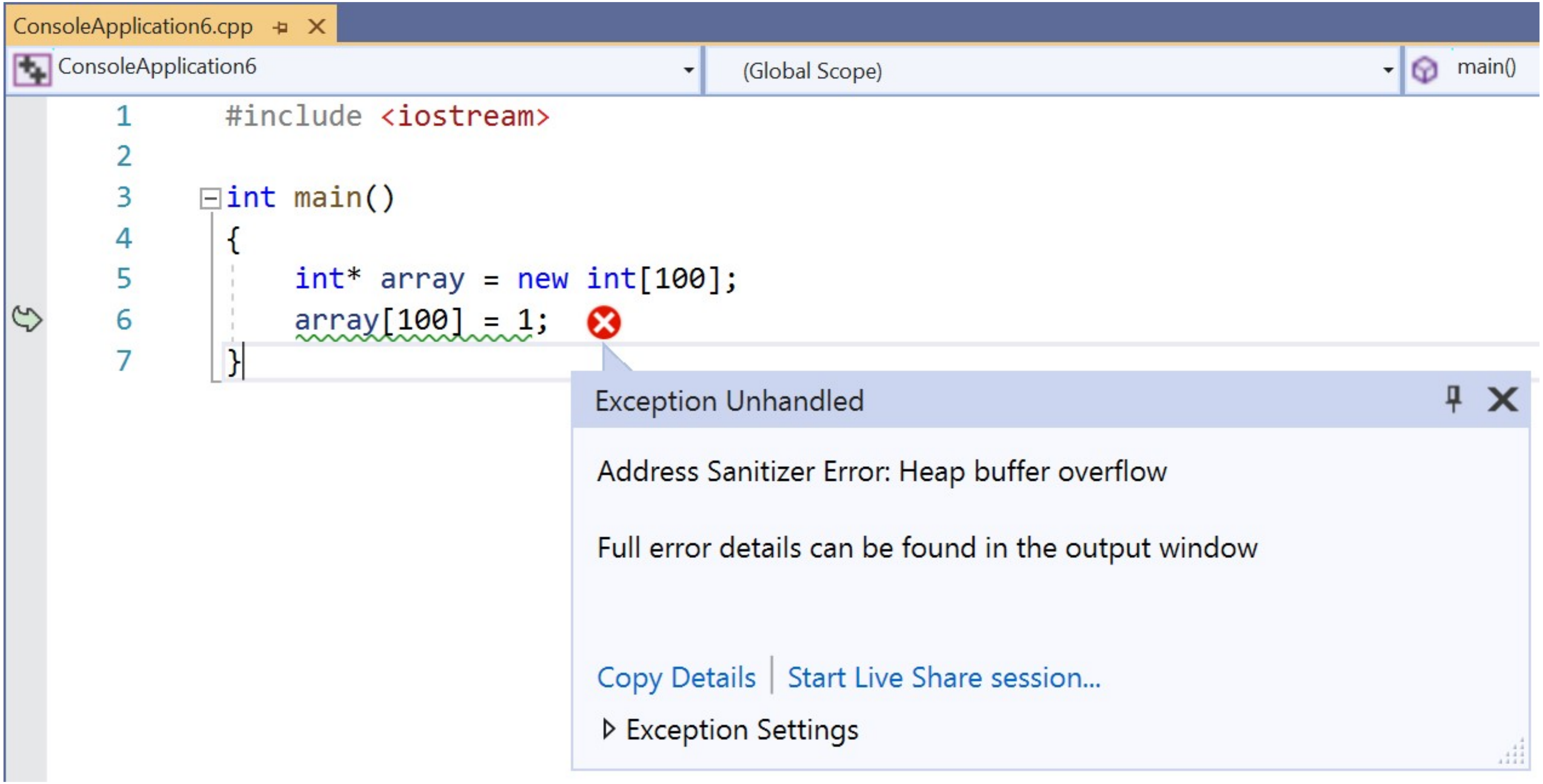
Visual Studio ASan
CMake

`CMakeSettings.json`

```
// eg. under the x86-Release configuration
{
  "addressSanitizerEnabled": true
}
```

> build with `/fsanitize:address`

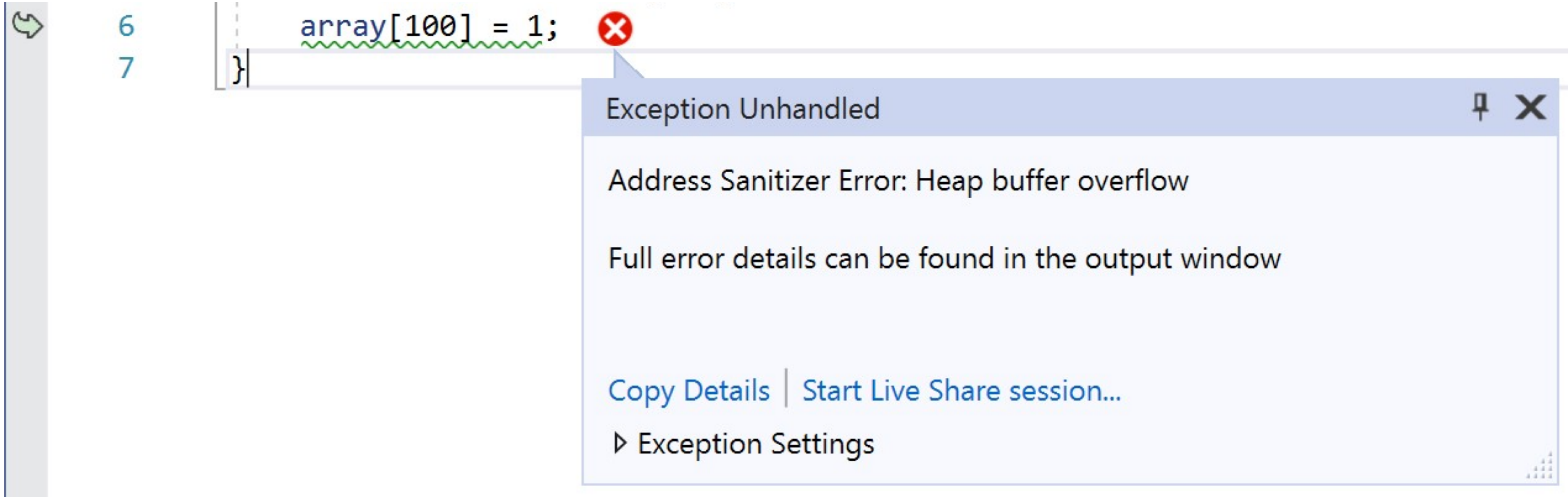
Address Sanitizer (ASan)



Address Sanitizer (ASan)

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

ASan logging information => Output window



Clang/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

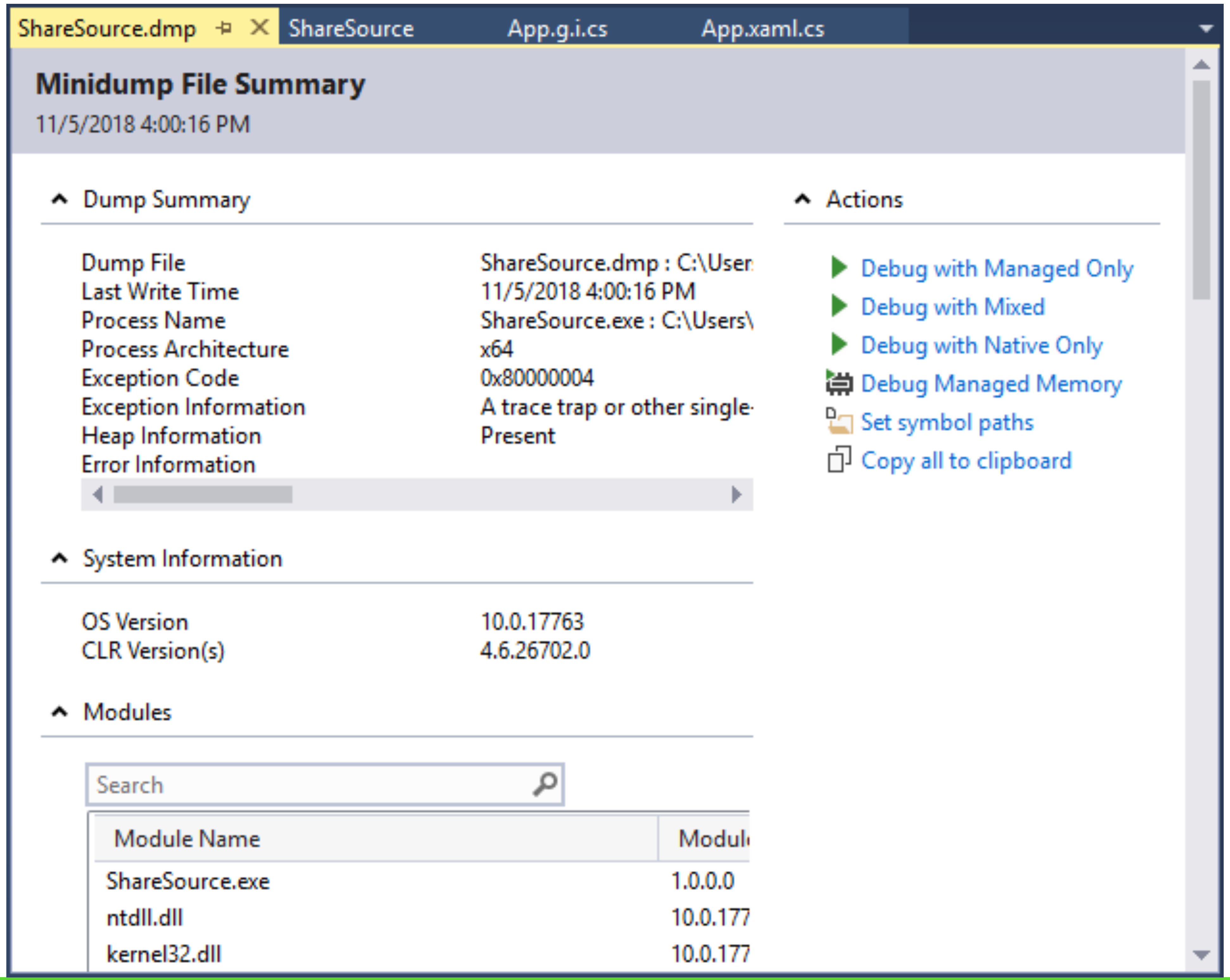
Game changer!

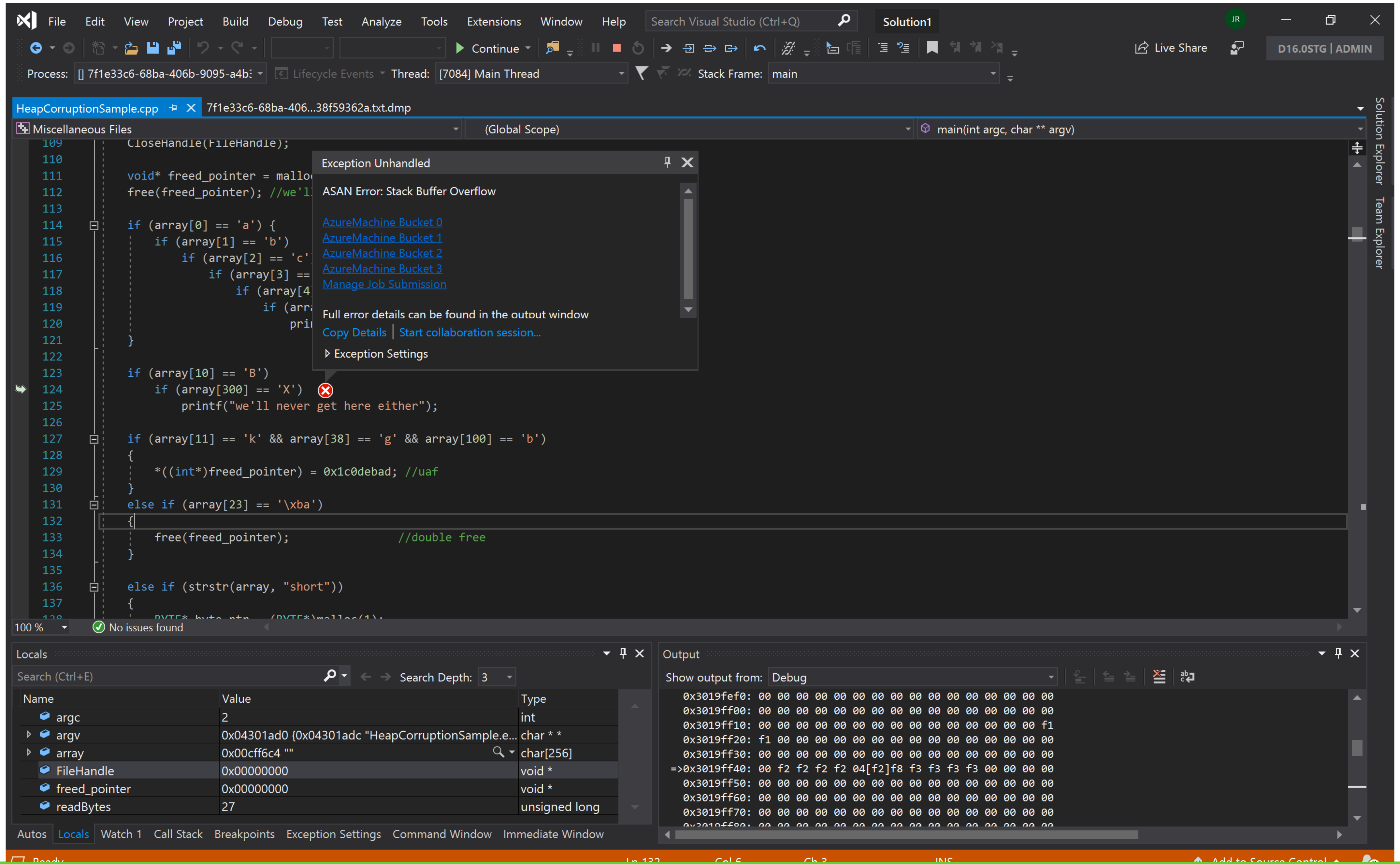
Minidump file (*.dmp) <= Windows snapshot process (program virtual memory/heap + metadata)

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

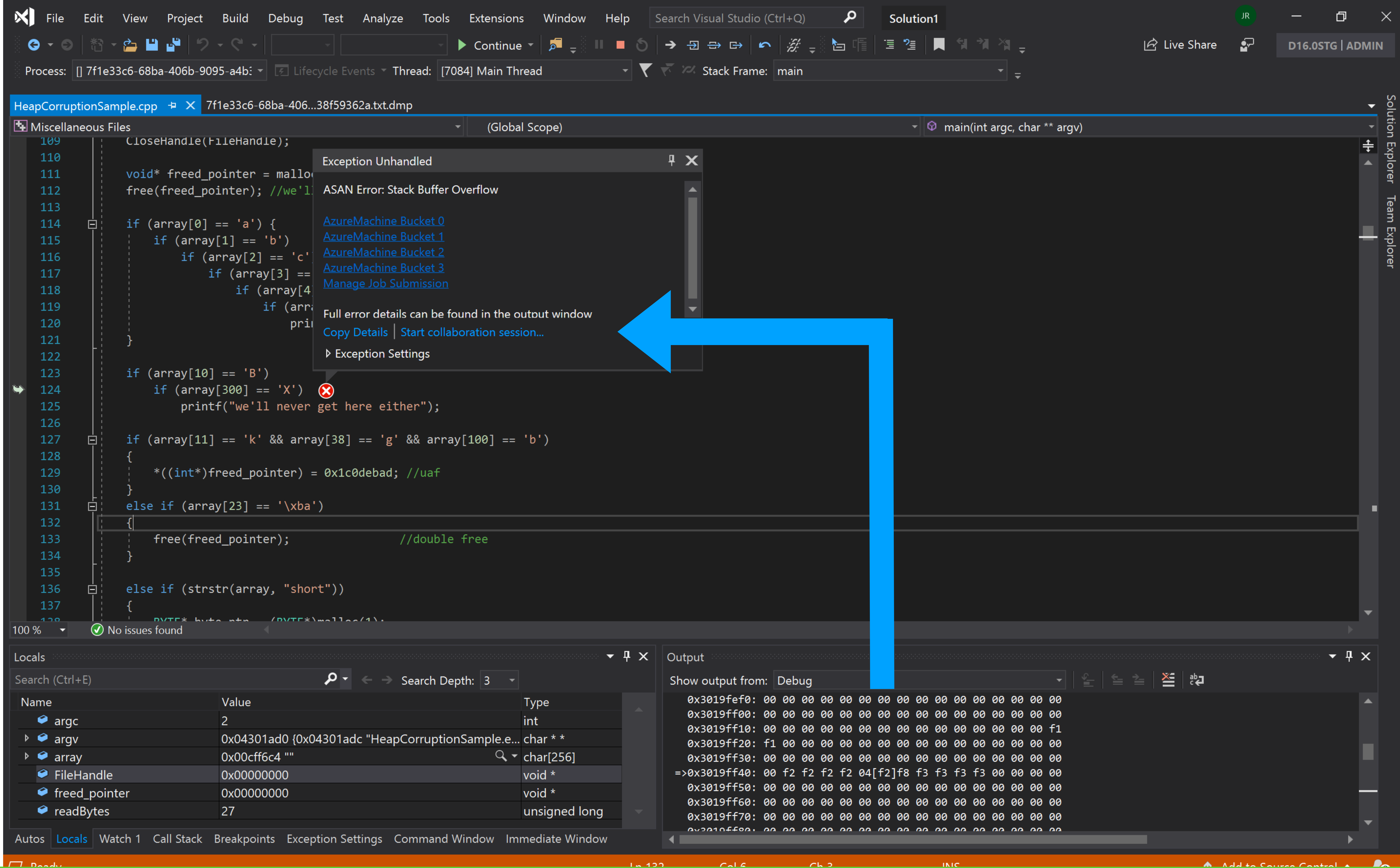
+ Live Share

Changes the way you report a bug, in general





Snapshot Loaded



How does it work ?

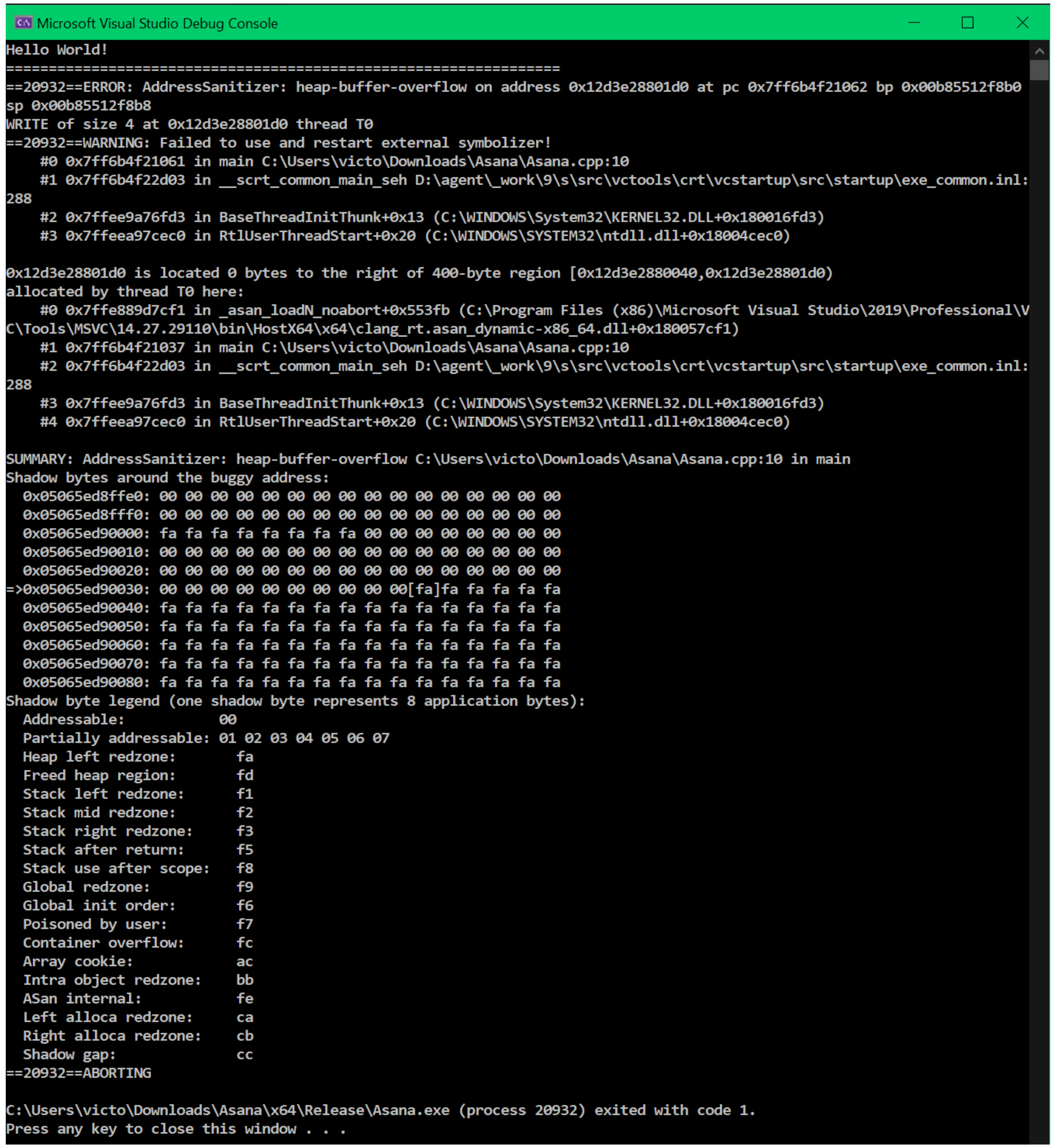
ASan is just Malware, used for Good

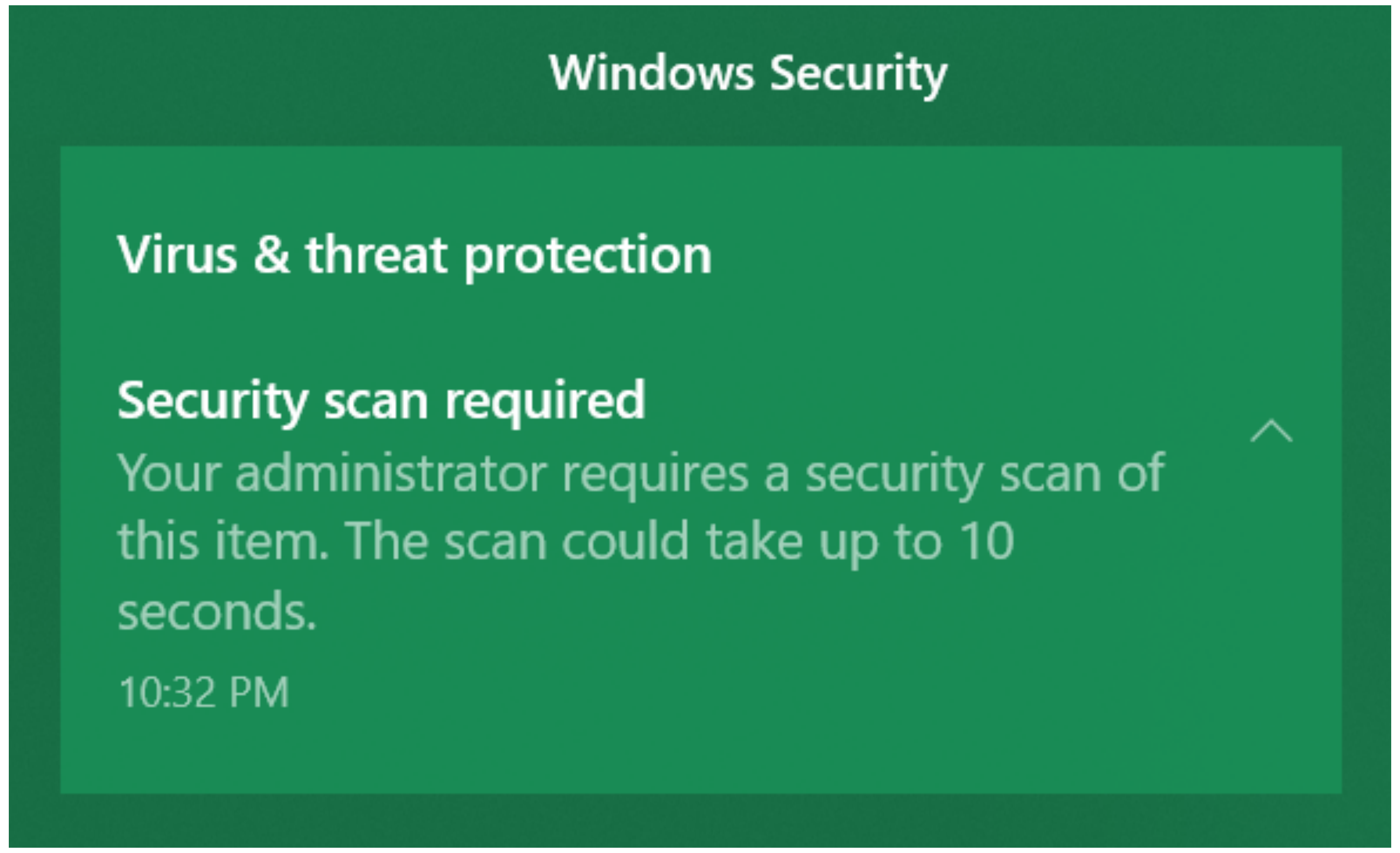
Microsoft Visual Studio Debug Console

```
Hello World!
=================================================================
==20932==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x12d3e28801d0 at pc 0x7ff6b4f21062 bp 0x00b85512f8b0
sp 0x00b85512f8b8
WRITE of size 4 at 0x12d3e28801d0 thread T0
==20932==WARNING: Failed to use and restart external symbolizer!
    #0 0x7ff6b4f21061 in main C:\Users\victo\Downloads\Asana\Asana.cpp:10
    #1 0x7ff6b4f22d03 in __scrt_common_main_seh D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:
288
    #2 0x7ffee9a76fd3 in BaseThreadInitThunk+0x13 (C:\WINDOWS\System32\KERNEL32.DLL+0x180016fd3)
    #3 0x7ffeea97cec0 in RtlUserThreadStart+0x20 (C:\WINDOWS\SYSTEM32\ntdll.dll+0x18004cec0)

0x12d3e28801d0 is located 0 bytes to the right of 400-byte region [0x12d3e2880040,0x12d3e28801d0)
allocated by thread T0 here:
    #0 0x7ffe889d7cf1 in _asan_loadN_noabort+0x553fb (C:\Program Files (x86)\Microsoft Visual Studio\2019\Professional\V
C\Tools\MSVC\14.27.29110\bin\HostX64\x64\clang_rt.asan_dynamic-x86_64.dll+0x180057cf1)
    #1 0x7ff6b4f21037 in main C:\Users\victo\Downloads\Asana\Asana.cpp:10
    #2 0x7ff6b4f22d03 in __scrt_common_main_seh D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:
288
    #3 0x7ffee9a76fd3 in BaseThreadInitThunk+0x13 (C:\WINDOWS\System32\KERNEL32.DLL+0x180016fd3)
    #4 0x7ffeea97cec0 in RtlUserThreadStart+0x20 (C:\WINDOWS\SYSTEM32\ntdll.dll+0x18004cec0)

SUMMARY: AddressSanitizer: heap-buffer-overflow C:\Users\victo\Downloads\Asana\Asana.cpp:10 in main
Shadow bytes around the buggy address:
  0x05065ed8ffe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x05065ed8fff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x05065ed90000: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
  0x05065ed90010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x05065ed90020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x05065ed90030: 00 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa
  0x05065ed90040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x05065ed90050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x05065ed90060: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x05065ed90070: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x05065ed90080: 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
==20932==ABORTING

C:\Users\victo\Downloads\Asana\x64\Release\Asana.exe (process 20932) exited with code 1.
Press any key to close this window . . .
```

😈

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()`, `free()`, `memset()`, etc.
- error analysis and reporting
- does not require complete recompile => great for **interop**
- zero false positives

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
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
```

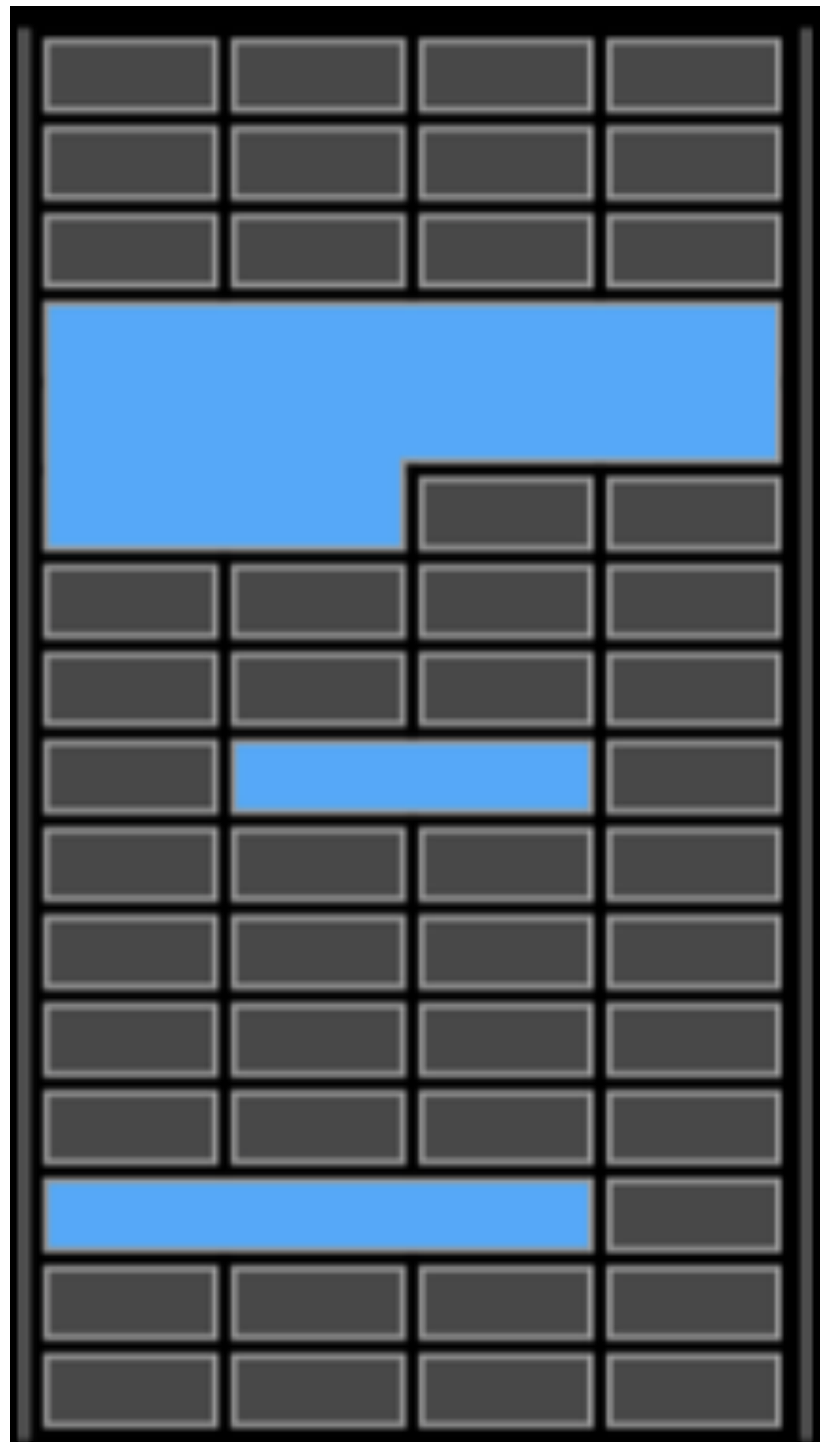
(of the 8 application bytes, how many are accessible)

issues & markers

Shadow byte legend

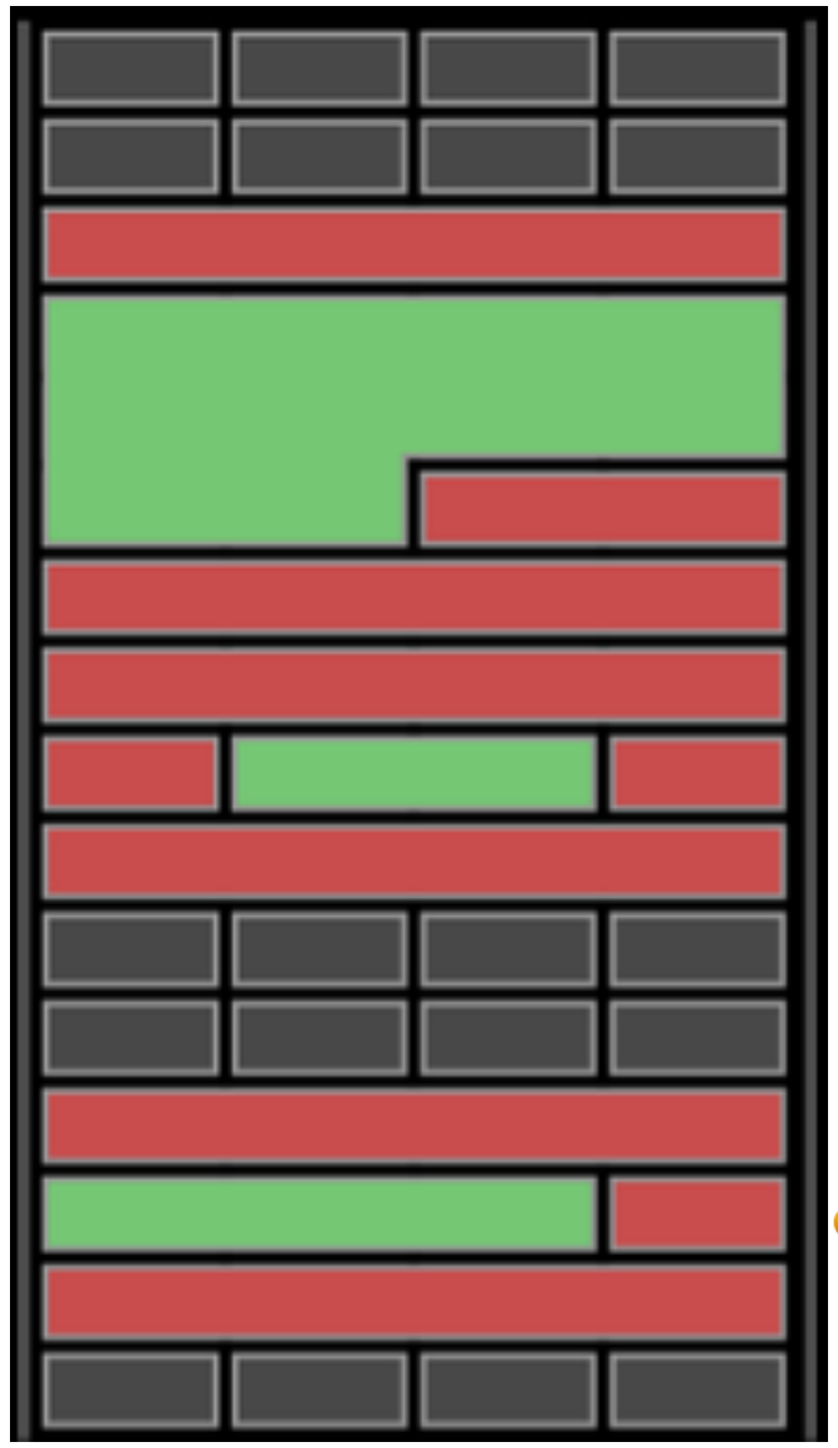
(one shadow byte represents 8 application bytes)

Shadow Mapping



my allocated memory

Process Memory



Poisoned memory

Red zones

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

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;
```

Stack use after scope

Code Generation

(simplified)

Lookups into shadow memory need to be very fast

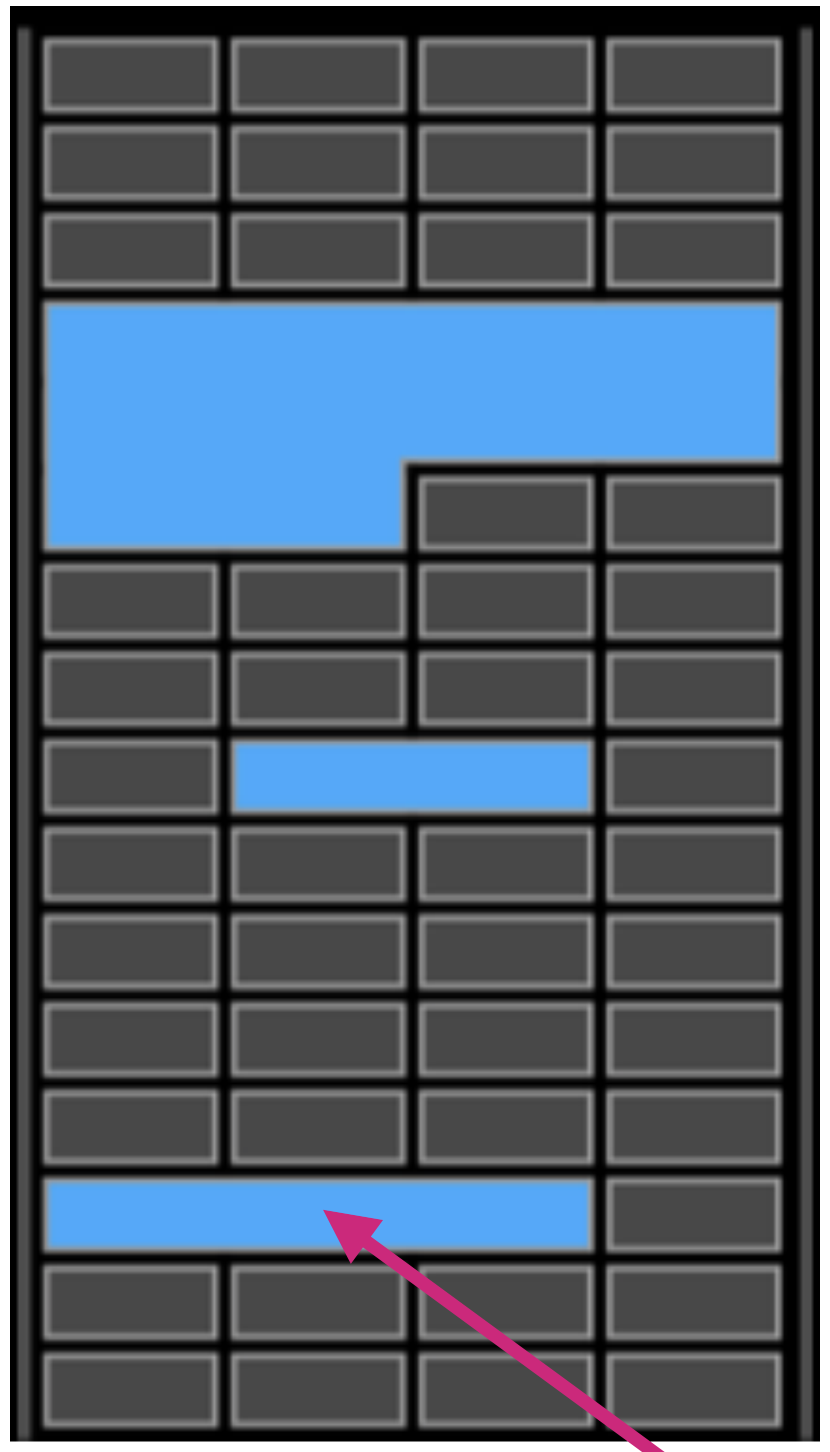
```
bool ShadowByte::IsBad(Addr) // is poisoned ?
{
  Shadow = Addr >> 3 + Offset;
  return (*Shadow) != 0;
}
```

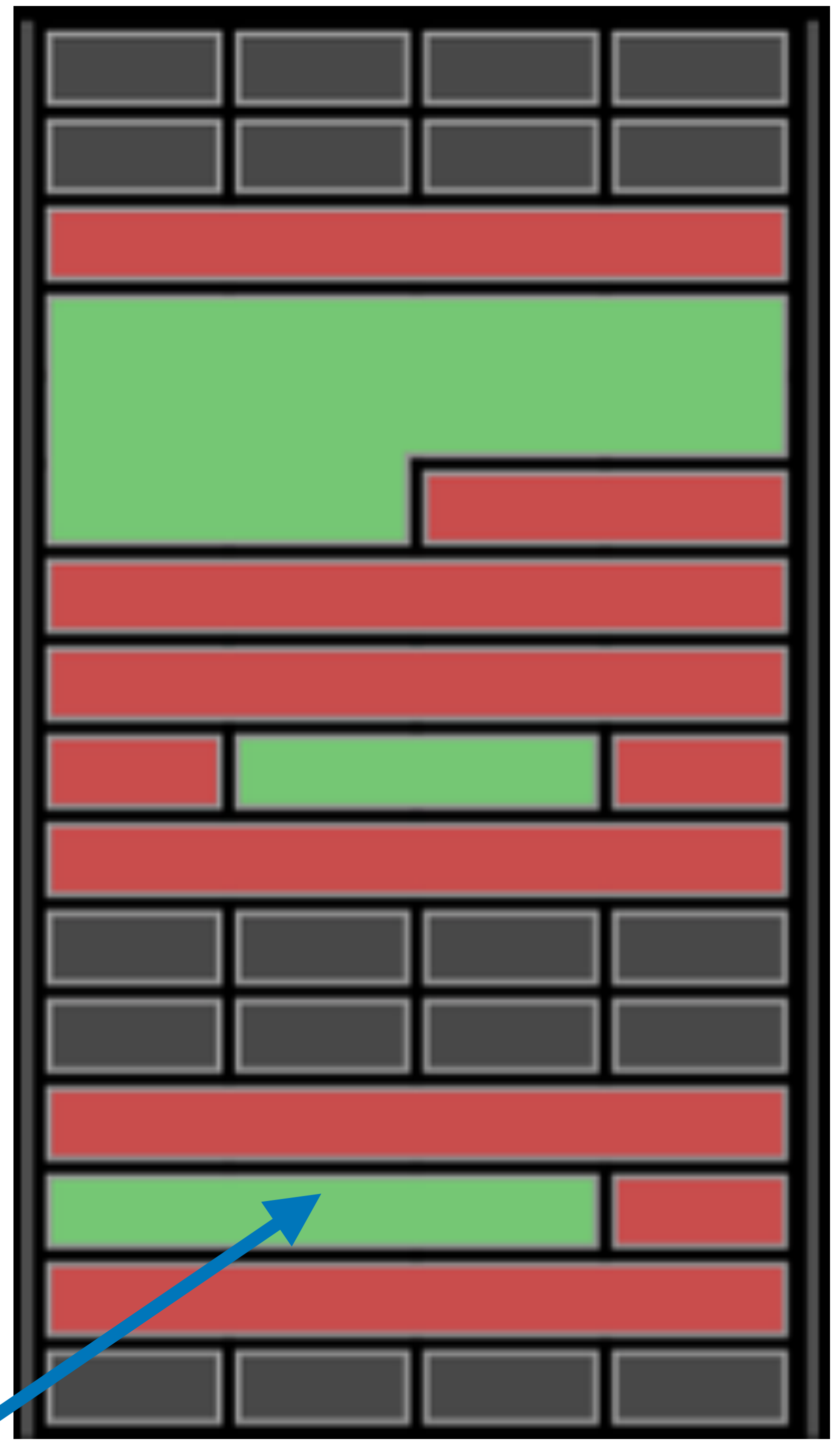
Location of shadow region in memory

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

Stack use after scope

Shadow Mapping



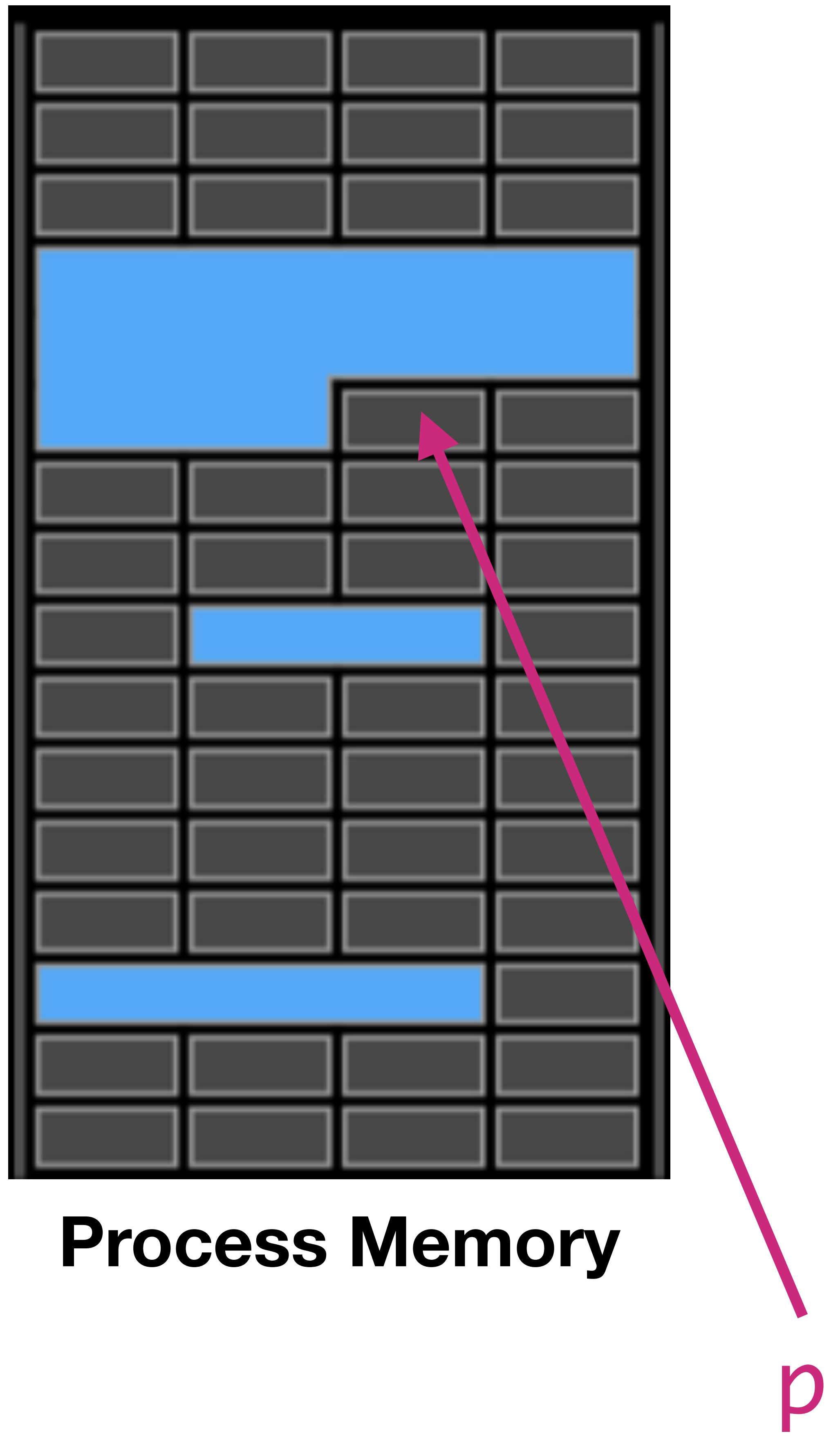



```
if (ShadowByte::IsBad(p))
  AsanRt::Report(p, sz);

*p = 0xf00d
```

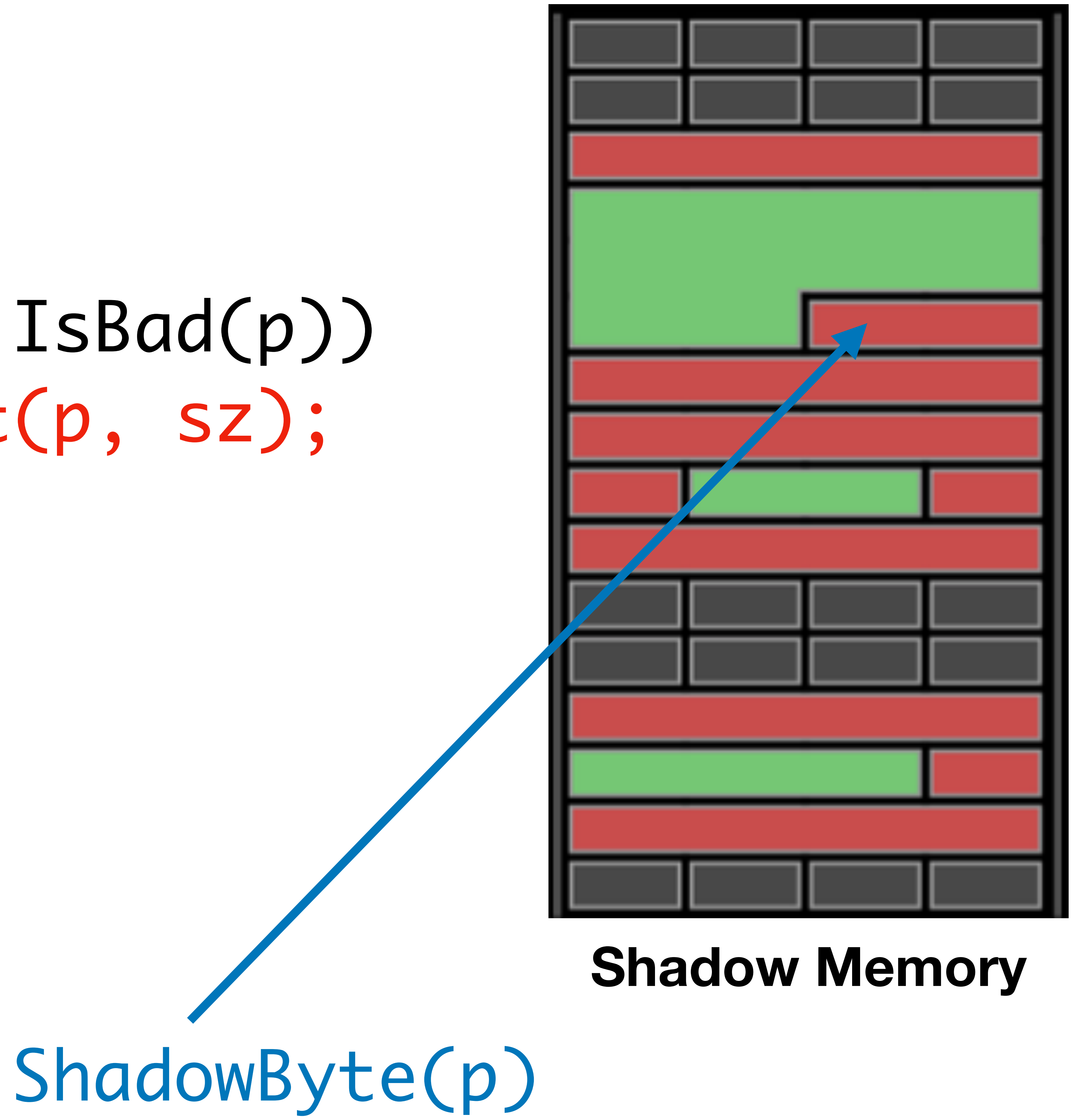
p

ShadowByte(p)

Shadow Mapping




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

*p = 0xbadf00d
```

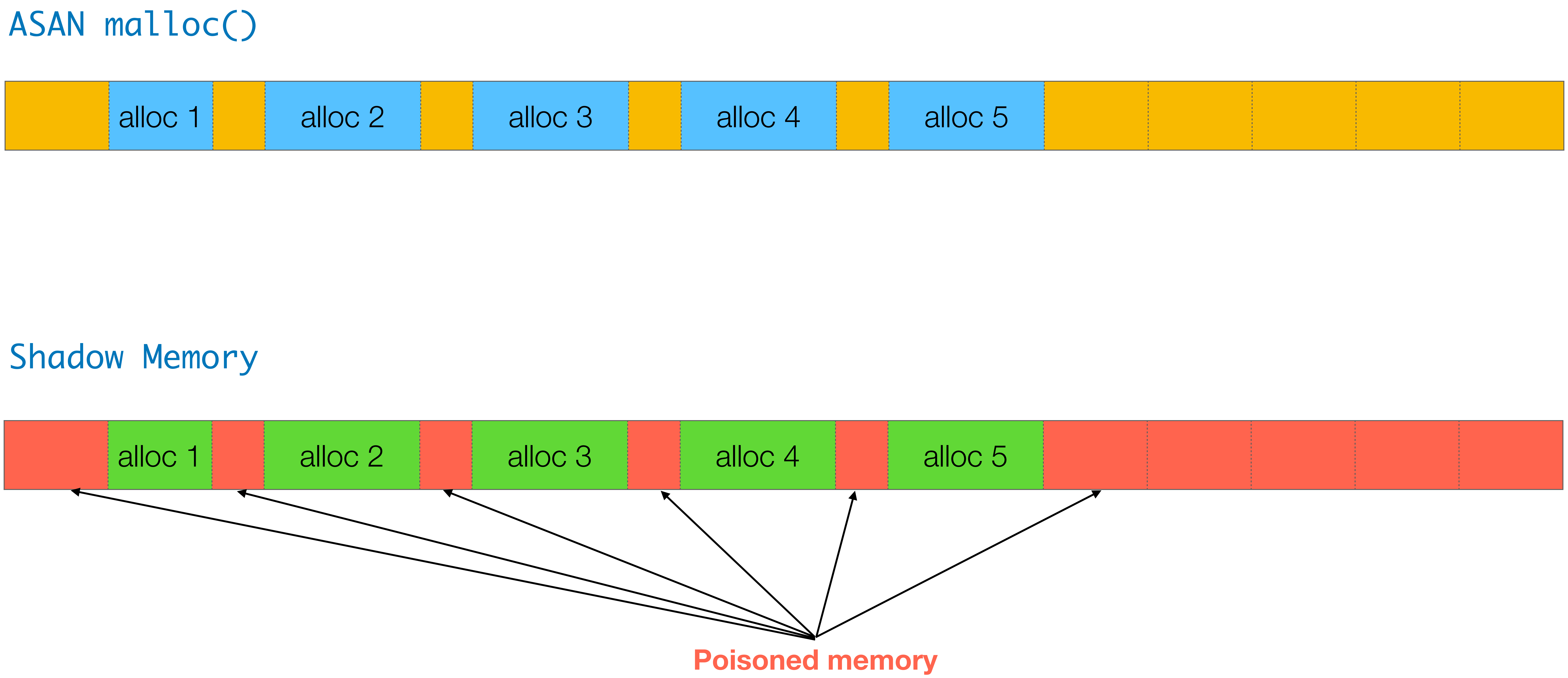


Heap Red Zones

`malloc()`

`ASAN malloc()`

Heap Red Zones

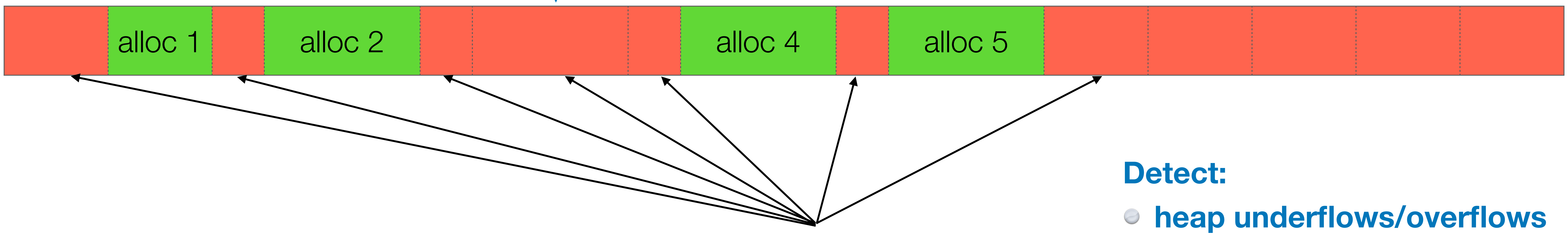


Heap Red Zones

ASAN malloc()

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

Shadow Memory

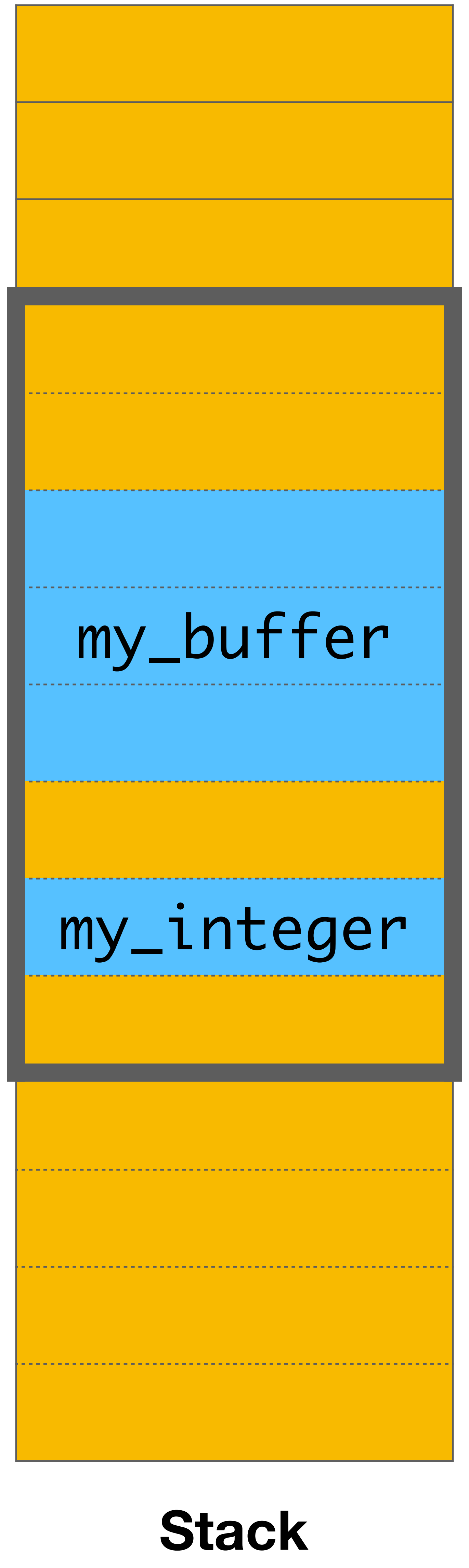


Poisoned memory

Detect:

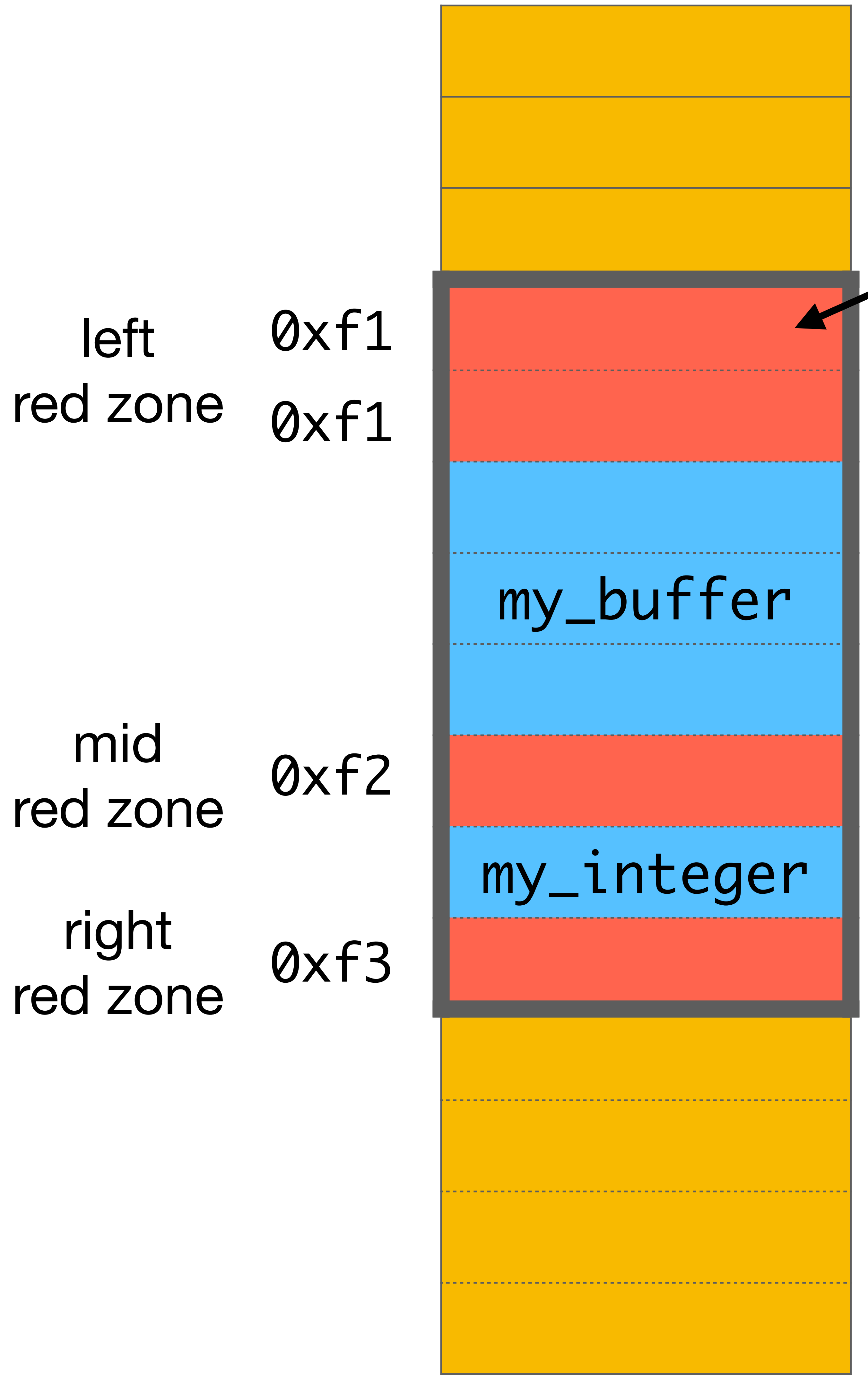
- **heap underflows/overflows**
- **use-after-free & double free**

Stack Red Zones




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

Stack Red Zones



Stack

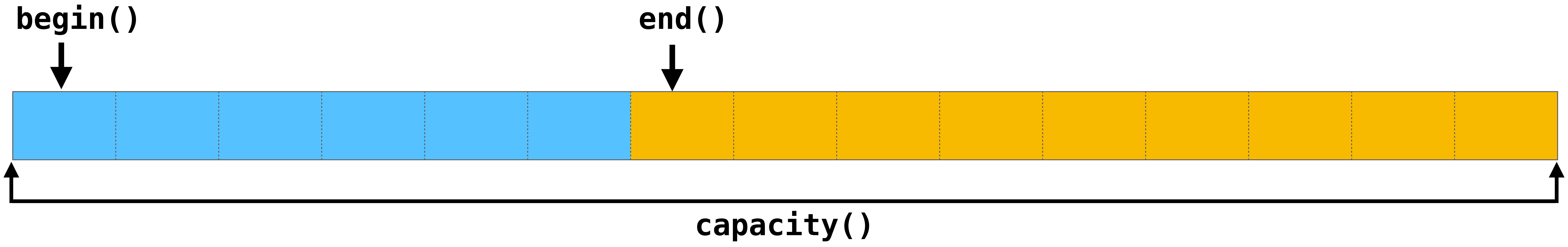
at runtime, the stack is **poisoned** when entering the function

```
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**

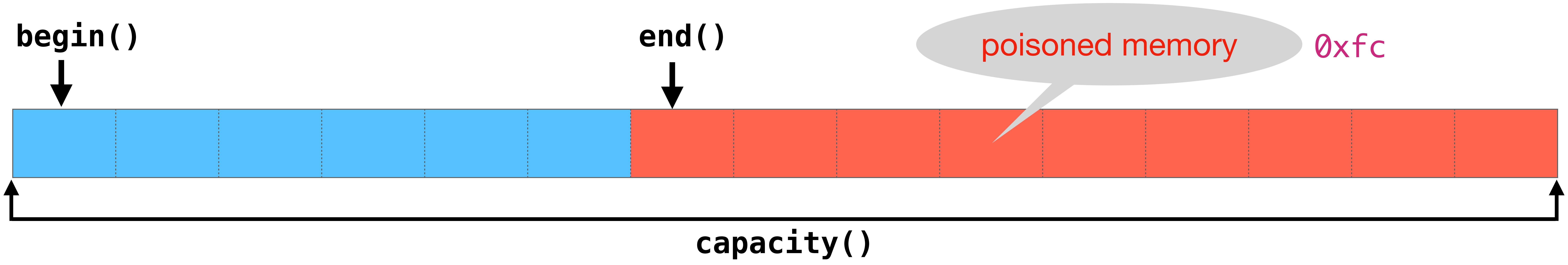
libc++

libstdc++

https://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

VS 16.7-16.9

Compiling/Linking from command-line

Manual CLI compile/link can be tedious
(choosing the correct **ASan libraries** to link against)

Check here for all the details:

devblogs.microsoft.com/cppblog/asan-for-windows-x64-and-debug-build-support/

Eg.

- **Compiling a single static EXE**
  link the static runtime `asan-i386.lib` and the cxx library

- **Compiling an EXE with /MT runtime which will use ASan-instrumented DLLs**
  the EXE needs to have `asan-i386.lib` linked and
  the DLLs need the `clang_rt.asan_dll_thunk-i386.lib`

- **When compiling with the /MD dynamic runtime**
  all EXE and DLLs with instrumentation should be linked with
  `asan_dynamic-i386.lib` and `clang_rt.asan_dynamic_runtime_thunk-i386.lib`
  At runtime, these libraries will refer to the
  `clang_rt.asan_dynamic-i386.dll` shared ASan runtime.

`/fsanitize:address`

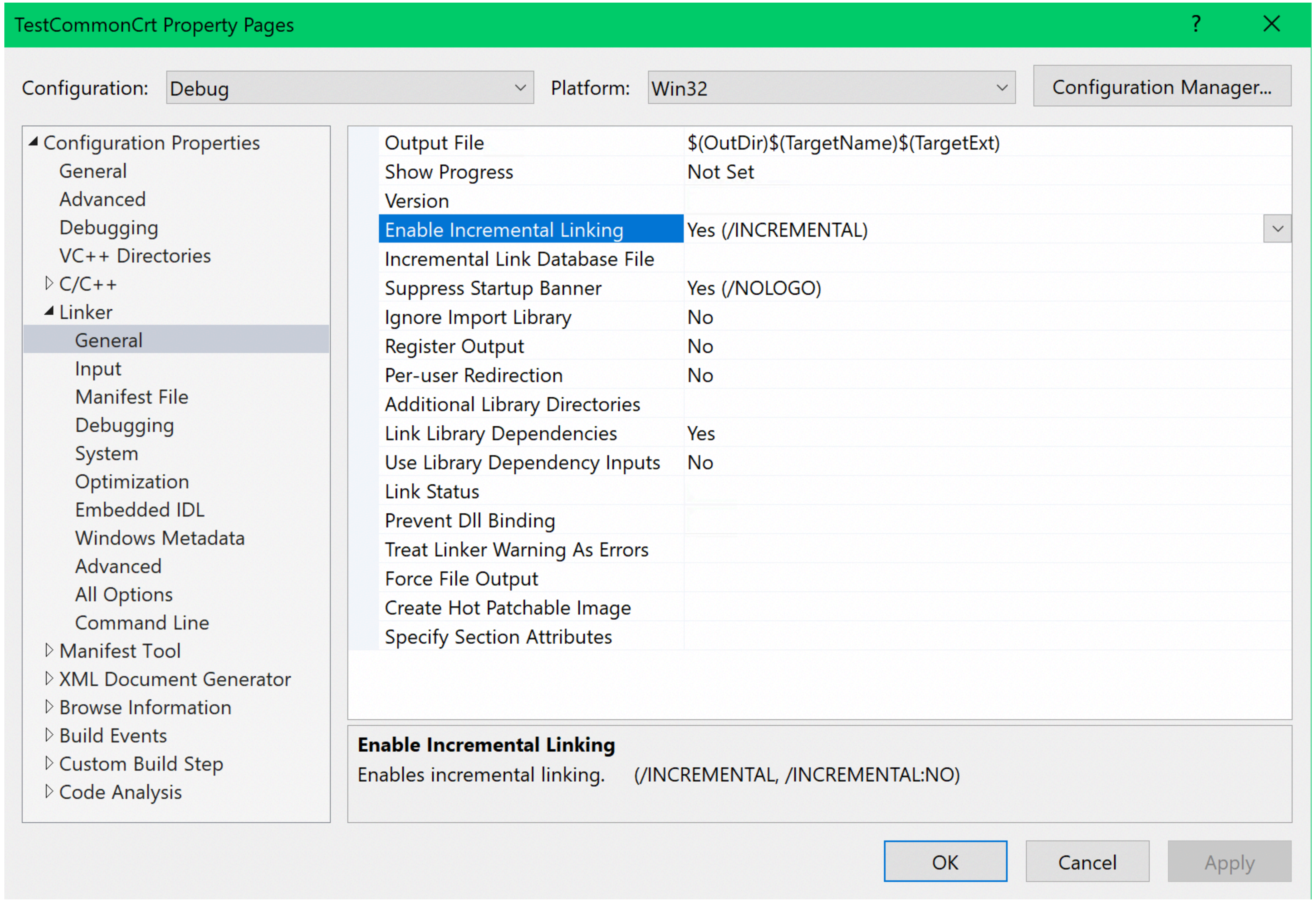
fixed in **v16.9**

/ZI
Edit and Continue (Debug)

```
error MSB8059:

-fsanitize=address (Enable Address Sanitizer) is incompatible with option
'edit-and-continue' debug information /ZI
```

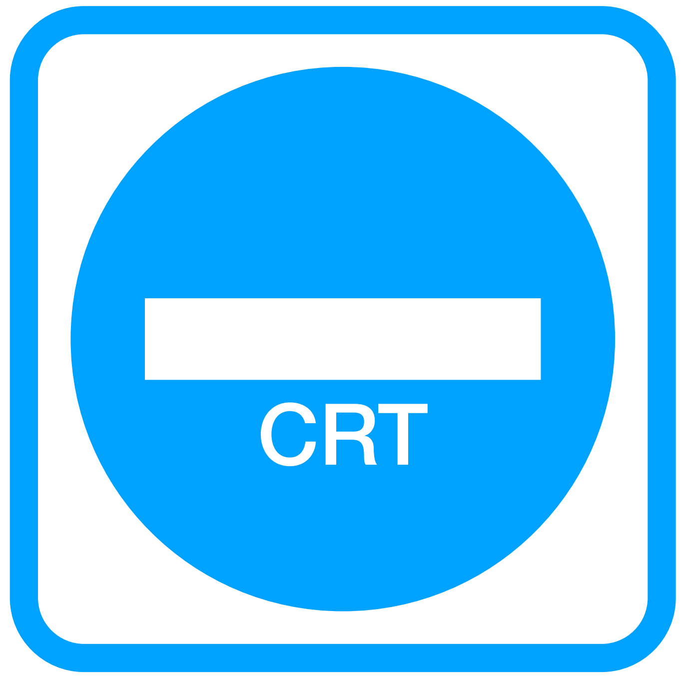
`Link /INCREMENTAL`

Debug builds

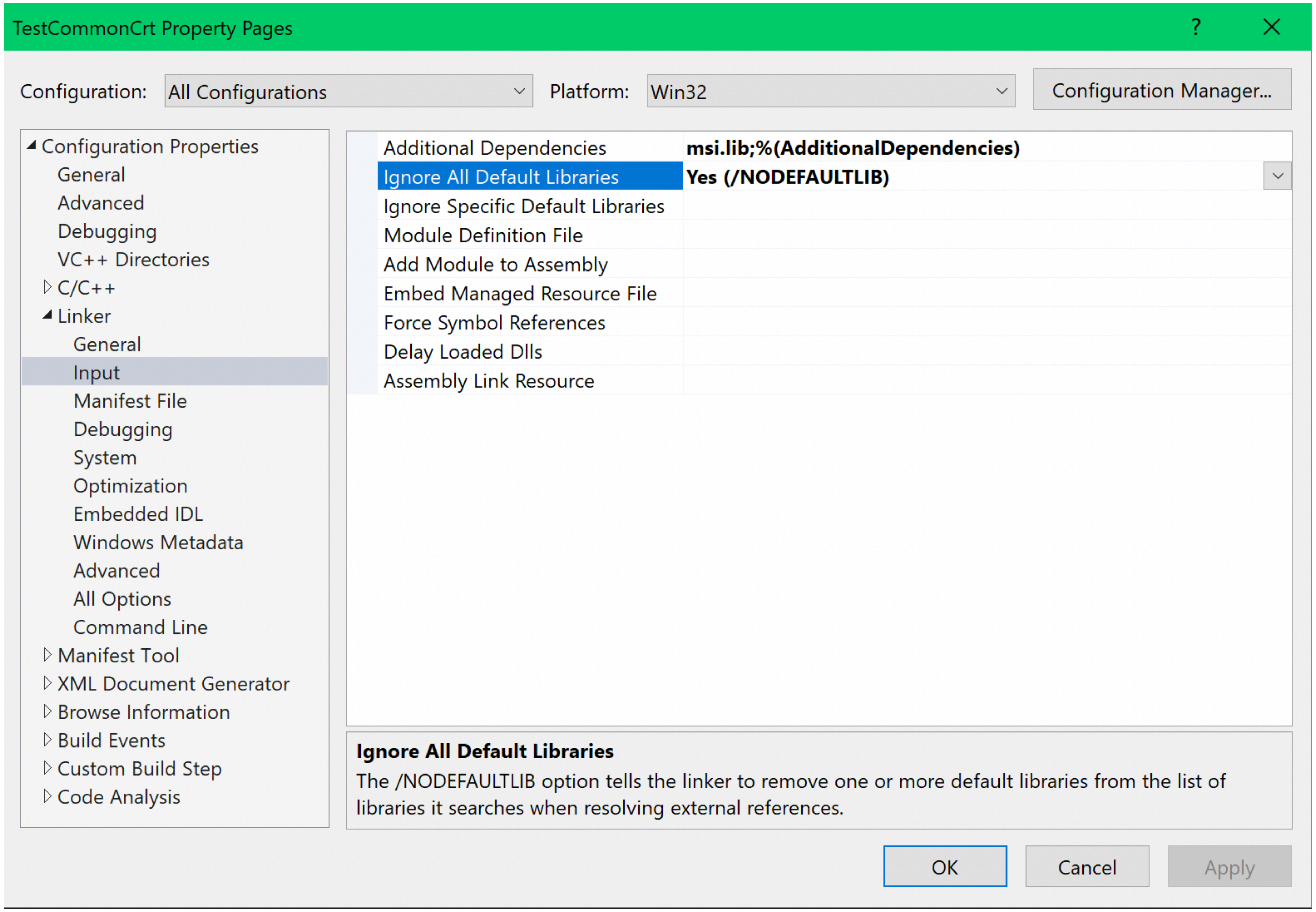


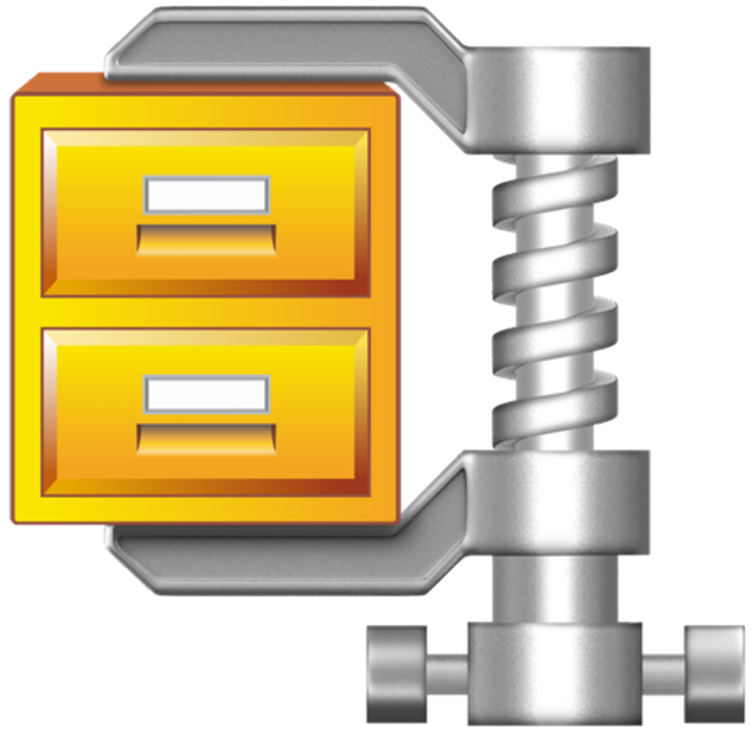

```
error MSB8059:
-fsanitize=address (Enable Address Sanitizer) is incompatible with option
'incremental linking (/INCREMENTAL)'
```

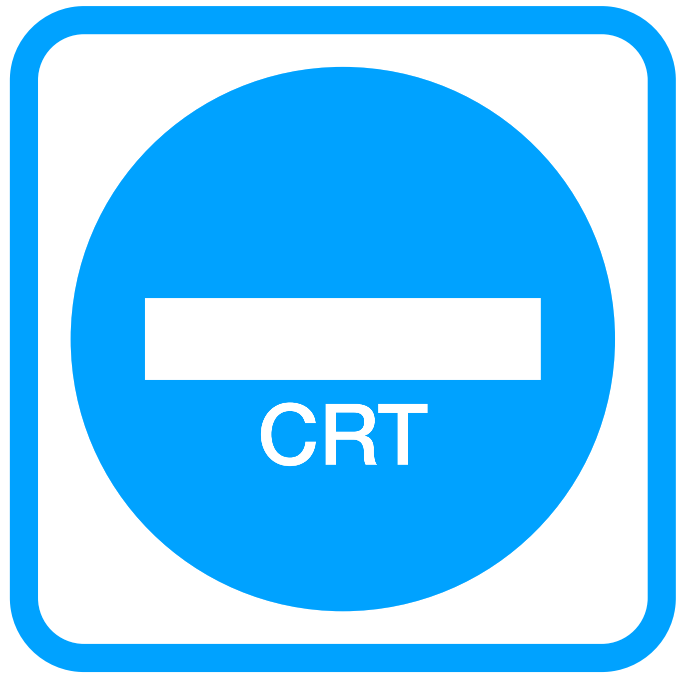
ASan + /NODEFAULTLIB



The linker will be very mad at you







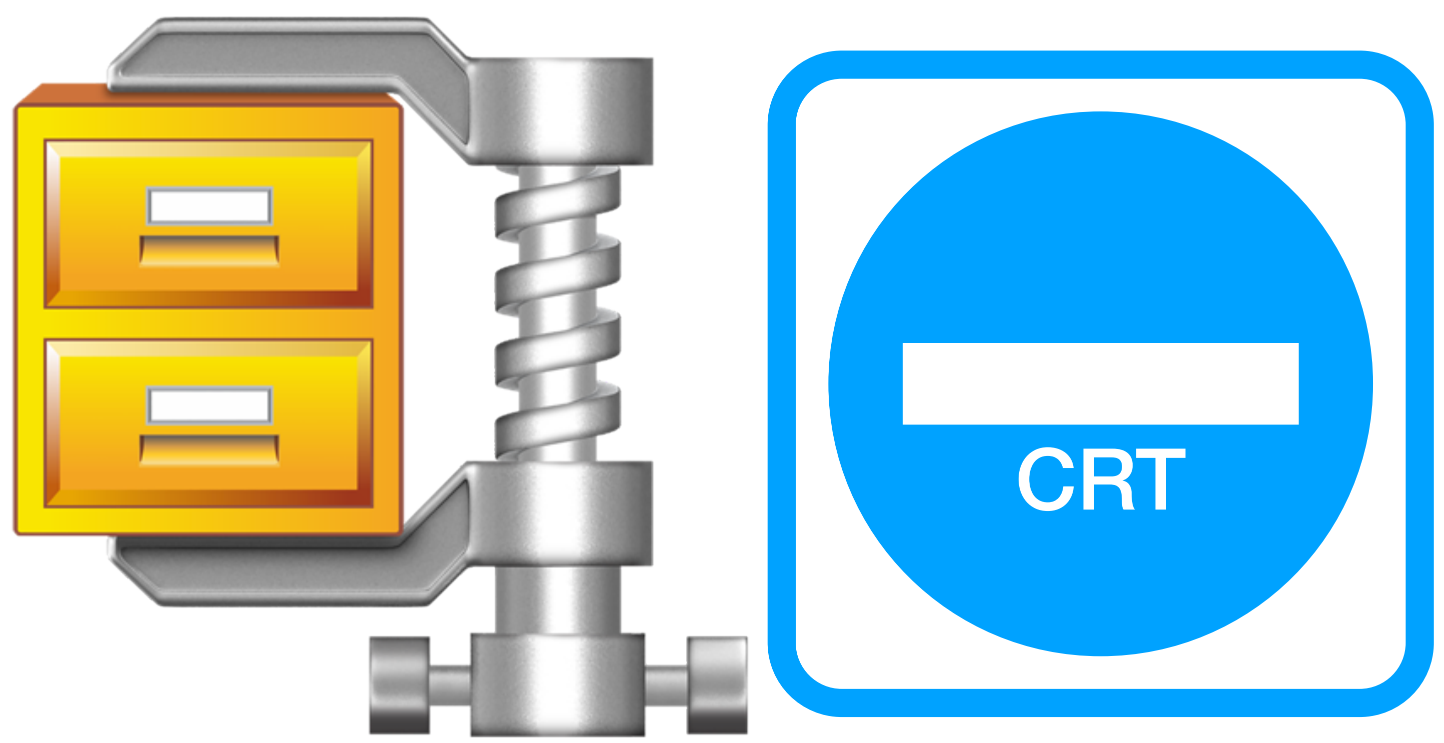
ASan + /NODEFAULTLIB

The linker will be very mad at you:

```
1>clang_rt.asan_dbg-i386.lib(ubsan_value.cc.obj) : error LNK2019: unresolved external symbol __allshl referenced in function "public: __int64 __thiscall __ubsan::Value::getSIntValue(void)const " (?getSIntValue@Value@__ubsan@@QBE_JXZ)
1>clang_rt.asan_dbg-i386.lib(ubsan_value.cc.obj) : error LNK2019: unresolved external symbol __allshr referenced in function "public: __int64 __thiscall __ubsan::Value::getSIntValue(void)const " (?getSIntValue@Value@__ubsan@@QBE_JXZ)
1>clang_rt.asan_dbg-i386.lib(ubsan_value.cc.obj) : error LNK2001: unresolved external symbol __fltused
1>clang_rt.asan_dbg-i386.lib(ubsan_diag.cc.obj) : error LNK2001: unresolved external symbol __fltused
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2001: unresolved external symbol __fltused
1>clang_rt.asan_dbg-i386.lib(asan_interceptors.cc.obj) : error LNK2001: unresolved external symbol __fltused
1>clang_rt.asan_dbg-i386.lib(ubsan_flags.cc.obj) : error LNK2019: unresolved external symbol _getenv referenced in function "char const * __cdecl __ubsan::GetFlag(char const *)" (?GetFlag@__ubsan@@YAPBDPBD@Z)
1>clang_rt.asan_dbg-i386.lib(ubsan_diag.cc.obj) : error LNK2019: unresolved external symbol ___stdio_common_vsprintf_s referenced in function _sprintf_s
1>clang_rt.asan_dbg-i386.lib(sanitizer_unwind_win.cc.obj) : error LNK2019: unresolved external symbol _memset referenced in function "private: void __thiscall __sanitizer::BufferedStackTrace::UnwindSlow(unsigned long,void *,unsigned int)
1>clang_rt.asan_dbg-i386.lib(interception_win.cc.obj) : error LNK2001: unresolved external symbol _memset
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2001: unresolved external symbol _memset
1>clang_rt.asan_dbg-i386.lib(asan_interceptors.cc.obj) : error LNK2001: unresolved external symbol _memset
1>clang_rt.asan_dbg-i386.lib(sanitizer_symbolizer_win.cc.obj) : error LNK2019: unresolved external symbol _wcsrchr referenced in function "void __cdecl __sanitizer::InitializeDbgHelpIfNeeded(void)" (?InitializeDbgHelpIfNeeded@__sanitizer
1>clang_rt.asan_dbg-i386.lib(sanitizer_symbolizer_win.cc.obj) : error LNK2019: unresolved external symbol _wcscat_s referenced in function "void __cdecl __sanitizer::InitializeDbgHelpIfNeeded(void)" (?InitializeDbgHelpIfNeeded@__sanitize
1>clang_rt.asan_dbg-i386.lib(sanitizer_allocator_checks.cc.obj) : error LNK2019: unresolved external symbol __errno referenced in function "void __cdecl __sanitizer::SetErrnoToENOMEM(void)" (?SetErrnoToENOMEM@__sanitizer@@YAXXZ)
1>clang_rt.asan_dbg-i386.lib(sanitizer_win.cc.obj) : error LNK2001: unresolved external symbol __errno
1>clang_rt.asan_dbg-i386.lib(asan_allocator.cc.obj) : error LNK2001: unresolved external symbol __errno
1>clang_rt.asan_dbg-i386.lib(sanitizer_win.cc.obj) : error LNK2019: unresolved external symbol _atexit referenced in function "int __cdecl __sanitizer::RunAtexit(void)" (?RunAtexit@__sanitizer@@YAHXZ)
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2001: unresolved external symbol _atexit
1>clang_rt.asan_dbg-i386.lib(sanitizer_win.cc.obj) : error LNK2019: unresolved external symbol _qsort referenced in function "void __cdecl __sanitizer::DumpProcessMap(void)" (?DumpProcessMap@__sanitizer@@YAXXZ)
1>clang_rt.asan_dbg-i386.lib(sanitizer_win.cc.obj) : error LNK2019: unresolved external symbol __alldiv referenced in function "unsigned __int64 __cdecl __sanitizer::NanoTime(void)" (?NanoTime@__sanitizer@@YA_KXZ)
1>clang_rt.asan_dbg-i386.lib(sanitizer_win.cc.obj) : error LNK2019: unresolved external symbol __allmul referenced in function "unsigned __int64 __cdecl __sanitizer::NanoTime(void)" (?NanoTime@__sanitizer@@YA_KXZ)
1>clang_rt.asan_dbg-i386.lib(sanitizer_win.cc.obj) : error LNK2019: unresolved external symbol __chkstk referenced in function "public: void __thiscall __sanitizer::ListOfModules::init(void)" (?init@ListOfModules@__sanitizer@@QAEXXZ)
1>clang_rt.asan_dbg-i386.lib(sanitizer_printf.cc.obj) : error LNK2019: unresolved external symbol __aulldvrm referenced in function "int __cdecl __sanitizer::AppendNumber(char * *,char const *,unsigned __int64,unsigned char,unsigned char
1>clang_rt.asan_dbg-i386.lib(interception_win.cc.obj) : error LNK2019: unresolved external symbol __stricmp referenced in function "bool __cdecl __interception::OverrideImportedFunction(char const *,char const *,char const *,unsigned lo
1>clang_rt.asan_dbg-i386.lib(asan_win.cc.obj) : error LNK2019: unresolved external symbol __except_handler3 referenced in function "void __cdecl __asan::InitializePlatformInterceptors(void)" (?InitializePlatformInterceptors@__asan@@YAXX
1>clang_rt.asan_dbg-i386.lib(asan_win.cc.obj) : error LNK2019: unresolved external symbol __except_handler4 referenced in function "void __cdecl __asan::InitializePlatformInterceptors(void)" (?InitializePlatformInterceptors@__asan@@YAXX
1>libvcasand.lib(vcasan.obj) : error LNK2001: unresolved external symbol __except_handler4
1>clang_rt.asan_dbg-i386.lib(asan_win.cc.obj) : error LNK2019: unresolved external symbol __tls_array referenced in function "void * __cdecl __asan::AsanTSDGet(void)" (?AsanTSDGet@__asan@@YAPAXXZ)
1>clang_rt.asan_dbg-i386.lib(asan_win.cc.obj) : error LNK2019: unresolved external symbol __tls_index referenced in function "void * __cdecl __asan::AsanTSDGet(void)" (?AsanTSDGet@__asan@@YAPAXXZ)
1>clang_rt.asan_dbg-i386.lib(asan_rtl.cc.obj) : error LNK2019: unresolved external symbol __aullrem referenced in function "void __cdecl __asan::InitializeHighMemEnd(void)" (?InitializeHighMemEnd@__asan@@YAXXZ)
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol __invalid_parameter referenced in function "void * __cdecl std::_Allocate_manually_vector_aligned<struct std::_Default_allocate_tra
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol __CrtDbgReport referenced in function "void * __cdecl std::_Allocate_manually_vector_aligned<struct std::_Default_allocate_traits>(
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol "public: __thiscall std::_Lockit::_Lockit(int)" (??0_Lockit@std@@QAE@H@Z) referenced in function "public: __thiscall std::_List_ite
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol "public: __thiscall std::_Lockit::~_Lockit(void)" (??1_Lockit@std@@QAE@XZ) referenced in function "public: __thiscall std::_List_it
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol ___std_exception_copy referenced in function "public: __thiscall std::bad_alloc::bad_alloc(class std::bad_alloc const &)" (??0bad_a
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol ___std_exception_destroy referenced in function "public: virtual __thiscall std::bad_array_new_length::~bad_array_new_length(void)"
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol "void __cdecl std::_Xlength_error(char const *)" (?_Xlength_error@std@@YAXPBD@Z) referenced in function "protected: struct std::pa
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol "void __cdecl std::_Xout_of_range(char const *)" (?_Xout_of_range@std@@YAXPBD@Z) referenced in function "public: void * __thiscall
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol __Mtx_init_in_situ referenced in function "void __cdecl `dynamic initializer for 'GlobalHeapMutex''(void)" (??__EGlobalHeapMutex@@
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol __Mtx_destroy_in_situ referenced in function "void __cdecl `dynamic atexit destructor for 'GlobalHeapMutex''(void)" (??__FGlobalHea
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol __Mtx_lock referenced in function "public: void * __thiscall MoveableMemoryManager::Alloc(unsigned long,unsigned int)" (?Alloc@Move
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol __Mtx_unlock referenced in function "public: void * __thiscall MoveableMemoryManager::Alloc(unsigned long,unsigned int)" (?Alloc@Mo
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol "void __cdecl std::_Throw_C_error(int)" (?_Throw_C_error@std@@YAXH@Z) referenced in function "public: void * __thiscall MoveableMem
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol __CxxThrowException@8 referenced in function __catch$??$_Emplace_reallocate@ABQAVMoveableAllocEntry@@@?$vector@PAVMoveableAllocEnt
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol ___CxxFrameHandler3 referenced in function __ehhandler$??$_Emplace_reallocate@ABQAVMoveableAllocEntry@@@?$vector@PAVMoveableAllocEn
1>libvcasand.lib(vcasan.obj) : error LNK2001: unresolved external symbol ___CxxFrameHandler3
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol __ftoui3 referenced in function "protected: struct std::pair<struct std::_List_node<struct std::pair<void * const,class MoveableAll
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol _ceil referenced in function "protected: struct std::pair<struct std::_List_node<struct std::pair<void * const,class MoveableAlloc
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol _memcpy referenced in function "public: void * __thiscall MoveableMemoryManager::ReallocFixedToHandle(void *,bool)" (?ReallocFixed
1>clang_rt.asan_dbg-i386.lib(asan_interceptors.cc.obj) : error LNK2001: unresolved external symbol _memcpy
1>libvcasand.lib(vcasan.obj) : error LNK2001: unresolved external symbol _memcpy
1>clang_rt.asan_dbg-i386.lib(asan_malloc_win_moveable.cc.obj) : error LNK2019: unresolved external symbol _memmove referenced in function "class MoveableAllocEntry * * * __cdecl std::_Copy_memmove<class MoveableAllocEntry * * *,class Mov
```

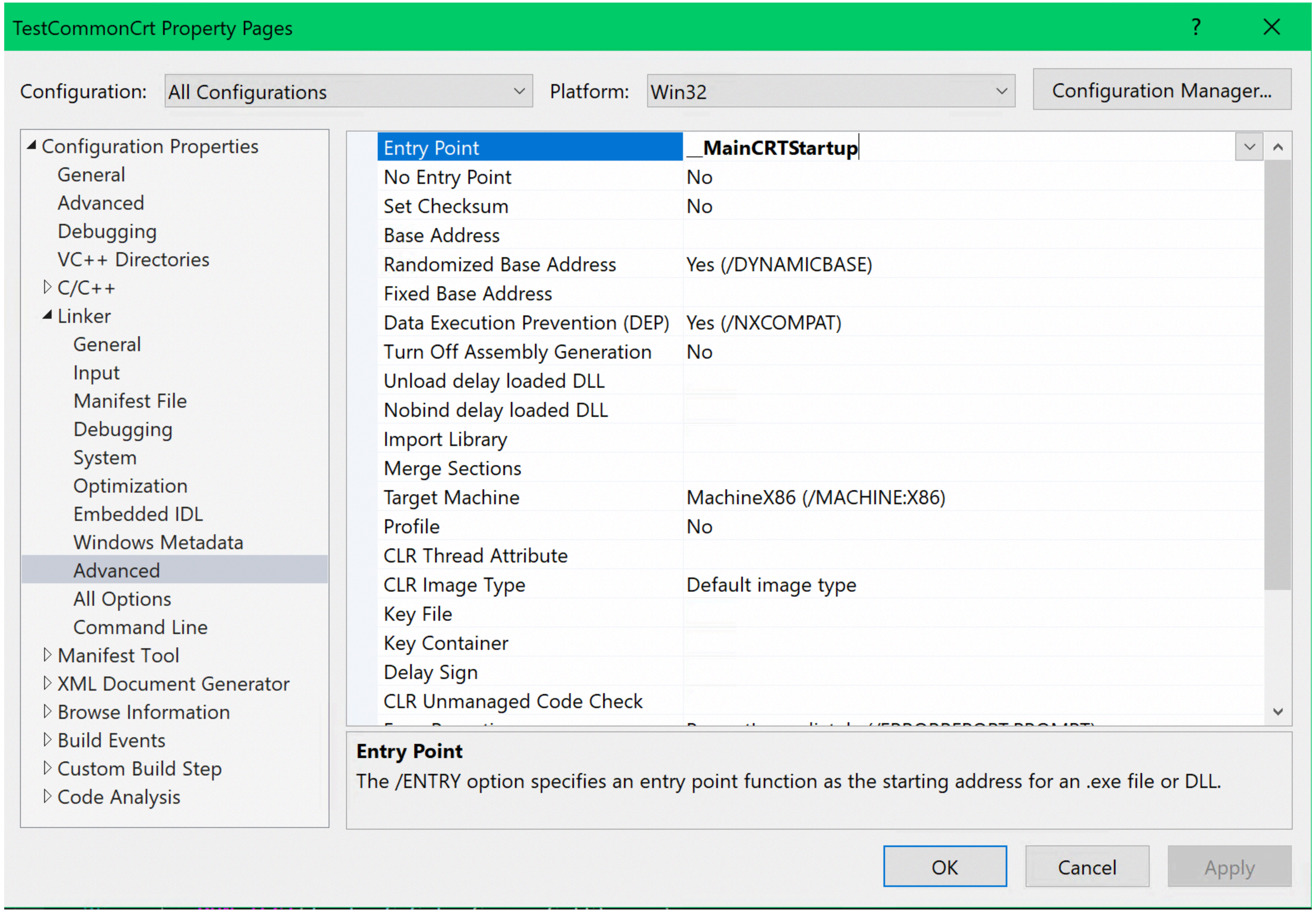
ASan runtime assumes **CRT** is linked

ASan + /NODEFAULTLIB



The linker will be very mad at you

if you have a custom entry point
(bypass CRT main)



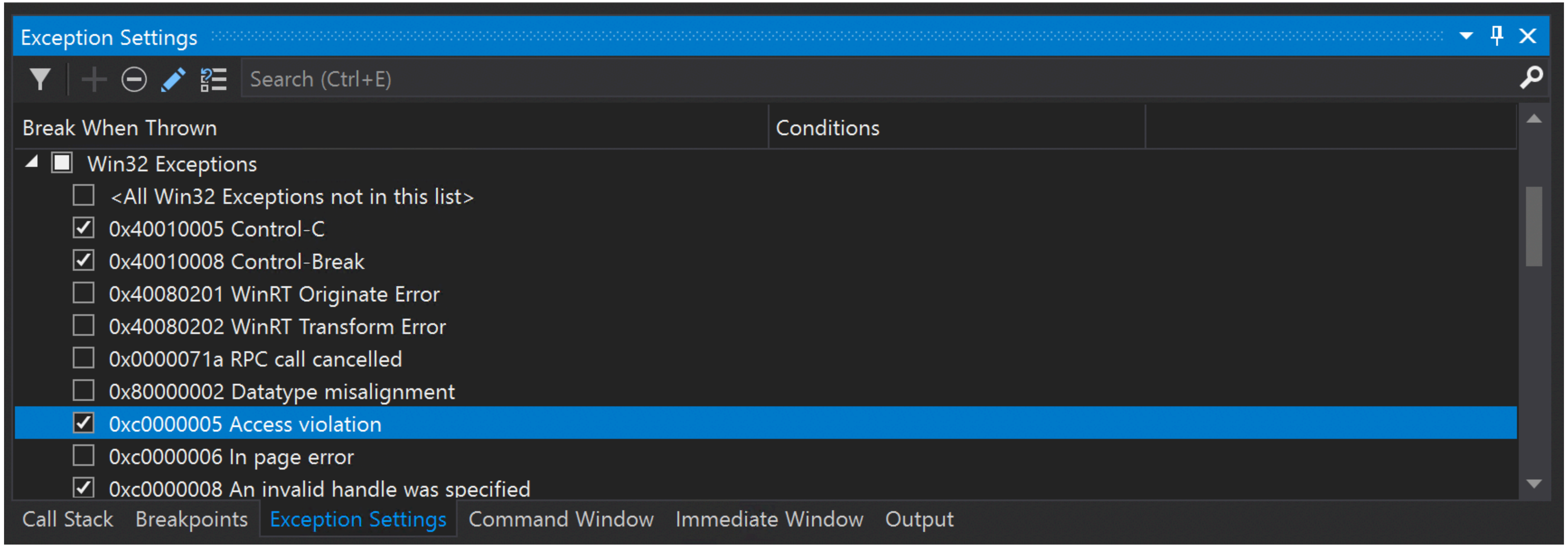
Access Violation Exceptions

Debugger may break frequently and you may see a lot of SEH access violation exceptions

This is normal (x64). It's how ASAN traps memory allocations to instrument its own *shadow memory*

Just tell the *Debugger* to stop breaking on this type of exception:

uncheck



Mixing ASan & non-ASan modules

Problem:

A non-ASan built executable can NOT call `LoadLibrary()` on a DLL built with ASAN.

Reason:

ASan runtime is tracking memory and the non-ASan executable might have done something like `HeapAlloc()`

This limitation is a problem if you're building a plugin (DLL)

MSVC team is considering dealing with this issue in a later release

devblogs.microsoft.com/cppblog/asan-for-windows-x64-and-debug-build-support/

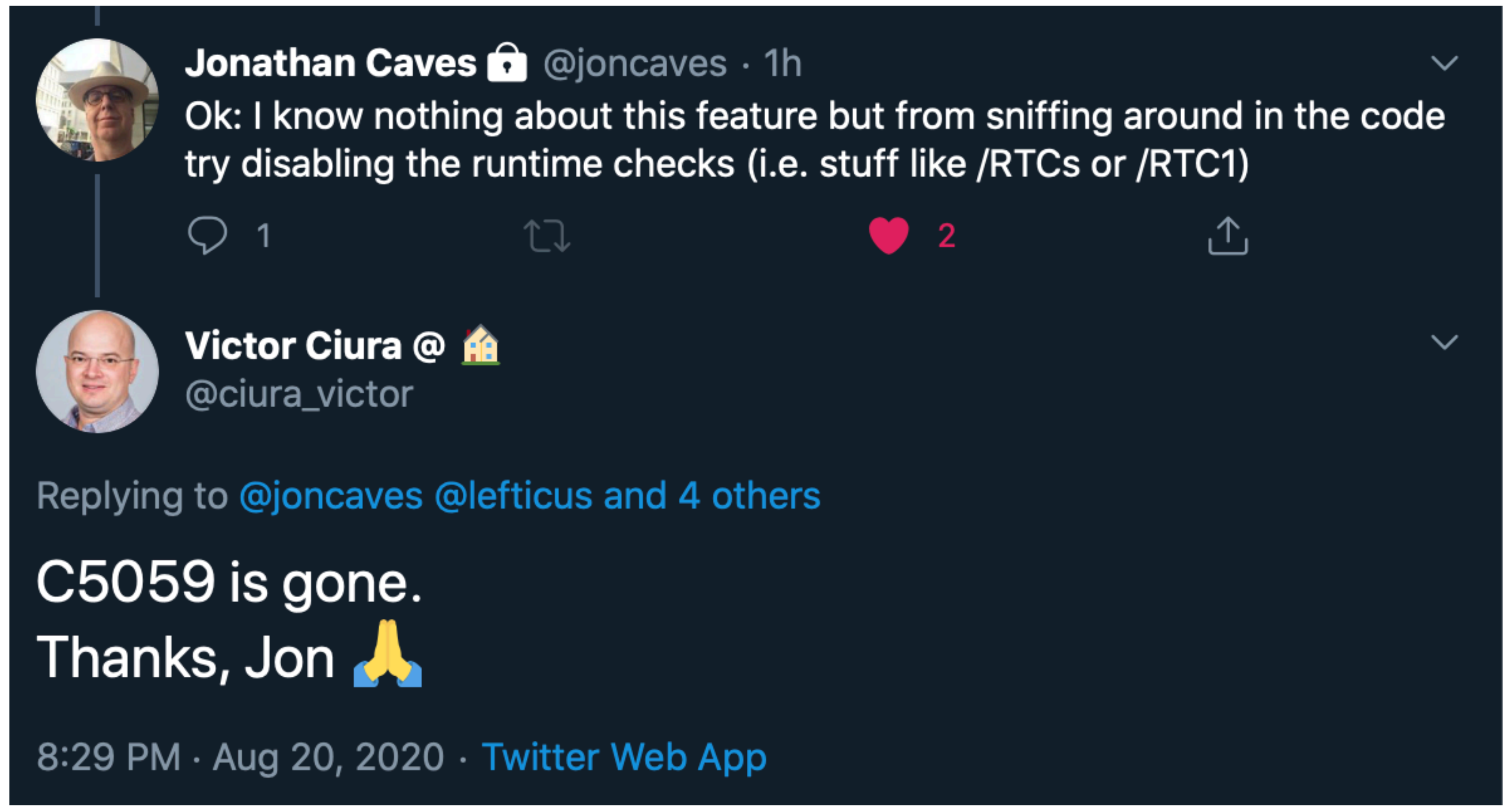
/RTCs and /RTC1 Runtime Checks

`warning C5059:`

runtime checks and address sanitizer is not currently supported - disabling runtime checks

If you use `/WX` this harmless/informative warning becomes a build blocker **:(**

=> we had to disable `/RTCs` and `/RTC1` so we could do the ASan experiments



twitter.com/ciura_victor/status/1296499633825492992

Missing PDBs from VS

v16.7

It appears some ASan runtime PDBs were not included in the VS installer:

```
[Debug]
vcasand.lib(vcasan.obj) : warning LNK4099: PDB 'vcasand.pdb' was not found with 'vcasand.lib(vcasan.obj)'
linking object as if no debug info


[Release]
vcasan.lib(vcasan.obj) : warning LNK4099: PDB 'vcasan.pdb' was not found with 'vcasan.lib(vcasan.obj)'
linking object as if no debug info
```

Building an EXE

fixed in **v16.9**

Missing PDBs from VS

v16.7

It appears some PDBs were not included in the VS installer:

```
[Debug]
libvcasand.lib(vcasan.obj) : warning LNK4099: PDB 'libvcasand.pdb' was not found with
'libvcasand.lib(vcasan.obj)


[Release]
libvcasan.lib(vcasan.obj) : warning LNK4099: PDB 'libvcasan.pdb' was not found with
'libvcasan.lib(vcasan.obj)'
```

Building a static LIB, linked into an EXE

fixed in **v16.9**

`vcasan(d).lib`

- creates **metadata** the **IDE** will parse to support error reporting in its sub-panes
- metadata is stored in **.dmp** files produced when a program is terminated by ASan

IDE integration for ASan-reported exceptions now handles the complete collection of reportable ASan exceptions

Linker Trouble?

Building a static LIB, linked into an EXE

```
[Debug | x64]
>libucrtd.lib(debug_heap.obj) : warning LNK4006: _calloc_dbg already defined in clang_rt.asan_dbg-x86_64.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: _expand_dbg already defined in clang_rt.asan_dbg-x86_64.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: _free_dbg already defined in clang_rt.asan_dbg-x86_64.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: _malloc_dbg already defined in clang_rt.asan_dbg-x86_64.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: _realloc_dbg already defined in clang_rt.asan_dbg-x86_64.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: _recalloc_dbg already defined in clang_rt.asan_dbg-x86_64.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(expand.obj)     : warning LNK4006: _expand already defined in clang_rt.asan_dbg-x86_64.lib(asan_malloc_win.cc.obj); second definition ignored

[Debug | x86]
>libucrtd.lib(debug_heap.obj) : warning LNK4006: __calloc_dbg already defined in clang_rt.asan_dbg-i386.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: __expand_dbg already defined in clang_rt.asan_dbg-i386.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: __free_dbg already defined in clang_rt.asan_dbg-i386.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: __malloc_dbg already defined in clang_rt.asan_dbg-i386.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: __realloc_dbg already defined in clang_rt.asan_dbg-i386.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(debug_heap.obj) : warning LNK4006: __recalloc_dbg already defined in clang_rt.asan_dbg-i386.lib(asan_malloc_win.cc.obj); second definition ignored
>libucrtd.lib(expand.obj)     : warning LNK4006: __expand already defined in clang_rt.asan_dbg-i386.lib(asan_malloc_win.cc.obj); second definition ignored
```

+ ASan

```
>uafxcw.lib(afxmem.obj) : error LNK2005: "void * __cdecl operator new(unsigned int)" (??2@YAPAXI@Z) already defined in clang_rt.asan_cxx-i386.lib(asan_new_delete.cc.obj)

>uafxcw.lib(afxmem.obj) : error LNK2005: "void __cdecl operator delete(void *)" (??3@YAXPAX@Z) already defined in clang_rt.asan_cxx-i386.lib(asan_new_delete.cc.obj)

>uafxcw.lib(afxmem.obj) : error LNK2005: "void * __cdecl operator new[](unsigned int)" (??_U@YAPAXI@Z) already defined in clang_rt.asan_cxx-i386.lib(asan_new_delete.cc.obj)

>uafxcw.lib(afxmem.obj) : error LNK2005: "void __cdecl operator delete[](void *)" (??_V@YAXPAX@Z) already defined in clang_rt.asan_cxx-i386.lib(asan_new_delete.cc.obj)
```

⚠️ **if you link statically to MFC lib**

developercommunity.visualstudio.com/content/problem/1144525/mfc-application-fails-to-link-with-address-sanitiz.html

+ ASan

In general, if you have overrides for:

```
void* operator new(size_t size);
```

Workarounds:

- set `/FORCE:MULTIPLE` in the linker command line (settings)
- temporarily set your MFC application to link to **shared** MFC DLLs for testing with ASan

ASAN Finds bugs

Really !

```
AddressSanitizer: heap-buffer-overflow on address 0x0a2301b4 pc 0x005b7a35 bp 0x011df078 sp 0x011df06c
READ of size 5 at 0x0a2301b4 thread T0

    #0 0x5b7a4d in __asan_wrap_strlen crt\asan\llvm\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:365
    #1 0x278eeb in ATL::CSimpleStringT<char,0>::StringLength MSVC\14.28.29333\atlmfc\include\atlsimpstr.h:726
    #2 0x278a35 in ATL::CSimpleStringT<char,0>::SetString MSVC\14.28.29333\atlmfc\include\atlsimpstr.h:602
    #3 0x274d69 in ATL::CSimpleStringT<char,0>::operator= MSVC\14.28.29333\atlmfc\include\atlsimpstr.h:314
    #4 0x274d99 in ATL::CStringT<char,ATL::StrTraitATL<char,ATL::ChTraitsCRT<char>>>::operator=
                   MSVC\14.28.29333\atlmfc\include\cstringt.h:1315
    #5 0x27469c in ATL::CStringT<char,ATL::StrTraitATL<char,ATL::ChTraitsCRT<char>>>::CStringT
                   MSVC\14.28.29333\atlmfc\include\cstringt.h:1115
    #6 0x27641a in SerValUtil::DecryptString C:\JobAI\advinst\msicomp\serval\SerValUtil.cpp:85
    #7 0x3e1660 in TestSerVal C:\JobAI\testunits\serval\SerValTests.cpp:60
    #8 0x5880e5 in FunctionTest::Run C:\JobAI\testunits\Tester.cpp:71
    #9 0x5889b1 in Tester::RunTest C:\JobAI\testunits\Tester.cpp:186
    #10 0x586ddb in Tester::ExecuteCommandLine C:\JobAI\testunits\Tester.cpp:558
    #11 0x5798d1 in main C:\JobAI\testunits\comps\TestComponents.cpp:2236

0x0a2301b4 is located 0 bytes to the right of 4-byte region [0x0a2301b0,0x0a2301b4)
allocated by thread T0
```

Fun with ATL::CString

```
ATL::CSimpleArray<BYTE> decrypted;
X::DecryptString(encrypted, decrypted);

ATL::CStringA decryptedStr(&decrypted[0]);
decryptedStr.ReleaseBufferSetLength(decrypted.GetSize());
```

Fun with `ATL::CString`

```
ATL::CSimpleArray<BYTE> decrypted;
X::DecryptString(encrypted, decrypted);

ATL::CStringA decryptedStr(&decrypted[0]);
decryptedStr.ReleaseBufferSetLength(decrypted.GetSize());
```

Fun with `ATL::CString`

Somewhere inside

```
ATL::CString::ReleaseBufferSetLength(int nLength)
{

   GetData()->nDataLength = nLength;
   m_pszData[nLength] = 0;
   ...
```

Fun with `ATL::CString`

Classic story: null-terminated string.

Array of chars to string class - size has a different meaning, because of the ending `\0`

Easy fix

```
ATL::CSimpleArray<BYTE> decrypted;
X::DecryptString(encrypted, decrypted);

ATL::CStringA decryptedStr(decrypted.GetData(), decrypted.GetSize());
```

It's actually more efficient, too.

```
AddressSanitizer: stack-buffer-overflow on address 0x00b3f766 at pc 0x00181b07 bp 0x00b3f6bc sp
0x00b3f6b0
WRITE of size 2 at 0x00b3f766 thread T0

    #0 0x181b06 in CommonCrt::ItoaT<wchar_t> C:\JobAI\platform\util\CommonCrt.h:402
    #1 0x183e02 in CommonCrt::Itoa C:\JobAI\platform\util\CommonCrt.cpp:119
    #2 0x190696 in TestCommonCrtItoa C:\JobAI\testunits\common_crt\CommonCrtTests.cpp:93
    #3 0x194821 in Tester::RunTest<int (__cdecl*)(void)> C:\JobAI\testunits\common_crt\tester\Tester.h:55
    #4 0x194b65 in main C:\JobAI\testunits\common_crt\main.cpp:22
    #5 0x1cc142 in invoke_main crt\vcstartup\src\startup\exe_common.inl:78
    #6 0x1cc046 in __scrt_common_main_seh crt\vcstartup\src\startup\exe_common.inl:288
    #7 0x1cbeec in __scrt_common_main crt\vcstartup\src\startup\exe_common.inl:330
    #8 0x1cc1a7 in mainCRTStartup crt\vcstartup\src\startup\exe_main.cpp:16
    #9 0x7645fa28 in BaseThreadInitThunk+0x18 (C:\WINDOWS\System32\KERNEL32.DLL+0x6b81fa28)
    #10 0x773e76b3 in RtlGetAppContainerNamedObjectPath+0xe3 (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2e76b3)
    #11 0x773e7683 in RtlGetAppContainerNamedObjectPath+0xb3 (C:\WINDOWS\SYSTEM32\ntdll.dll+0x4b2e7683)

Address 0x00b3f766 is located in stack of thread T0 at offset 30 in frame
    #0 0x1905ef in TestCommonCrtItoa C:\JobAI\testunits\common_crt\CommonCrtTests.cpp:84

  This frame has 2 object(s):
    [16, 30) 'result1' <== Memory access at offset 30 overflows this variable
    [32, 46) 'result2' <== Memory access at offset 30 underflows this variable
```

Naive Test Unit

```
const LONG    kNumber1 = 21474835;
TCHAR         result1[kMaxSize];
const TCHAR * compare1 = L"21474835";
const LONG    kNumber2 = -2100;
TCHAR         result2[kMaxSize];
const TCHAR * compare2 = L"-2100";

CommonCrt::Itoa(kNumber1, result1);

ASSERT_EQ(CompareStrings(result1, compare1));
...
```

Naive Test Unit

```
const LONG    kNumber1 = 21474835;
TCHAR         result1[kMaxSize];
const TCHAR * compare1 = L"21474835";
const LONG    kNumber2 = -2100;
TCHAR         result2[kMaxSize];
const TCHAR * compare2 = L"-2100";

CommonCrt::Itoa(kNumber1, result1);

ASSERT_EQ(CompareStrings(result1, compare1));
...
```

```
AddressSanitizer: stack-buffer-overflow on address 0x00843b3ae544 at pc 0x7ff6da711d86 bp 0x00843b3ae180
sp 0x00843b3ae188
READ of size 1 at 0x00843b3ae544 thread T0

#0 0x7ff6da711d85 in std::_Char_traits<unsigned char,long>::length MSVC\14.28.29333\include\xstring:143
    #1 0x7ff6da711667 in std::basic_string<unsigned char,std::char_traits<unsigned char>,std::allocator<unsigned char> >::assign
                         MSVC\14.28.29333\include\xstring:3062
    #2 0x7ff6da70af94 in std::basic_string<unsigned char...> MSVC\14.28.29333\include\xstring:2417
    #3 0x7ff6da70c163 in TestStringUtilAsciiToUnicode C:\JobAI\testunits\strings\StringEncodingTests.cpp:26
    #4 0x7ff6da98db80 in FunctionTest::Run C:\JobAI\testunits\Tester.cpp:71
    #5 0x7ff6da98fb05 in Tester::RunTest C:\JobAI\testunits\Tester.cpp:186
    #6 0x7ff6da98b3b4 in Tester::ExecuteCommandLine C:\JobAI\testunits\Tester.cpp:558
    #7 0x7ff6da97b59e in main C:\JobAI\testunits\comps\TestComponents.cpp:2236
    #8 0x7ff6dac2a8d8 in invoke_main d:\agent\_work\63\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:78

Address 0x00843b3ae544 is located in stack of thread T0 at offset 564 in frame
    #0 0x7ff6da70badf in TestStringUtilAsciiToUnicode C:\JobAI\testunits\strings\StringEncodingTests.cpp:14

This frame has 12 object(s):
    [32, 72) 'result1'
    [48, 88) 'kTextString1'
    [64, 104) 'result2'
    [80, 120) 'kTextString3'
    [96, 136) 'result3'
    [112, 152) 'compiler temporary'
    [128, 144) 'compiler temporary'
    [144, 160) 'compiler temporary'
    [160, 164) 'uChars'
    [176, 177) 'compiler temporary'
    [192, 216) 'compiler temporary'
    [208, 232) 'compiler temporary' <== Memory access at offset 564 overflows this variable
```

Naive Test Unit

```
unsigned char                     uChars[] = { 0x41, 0x42, 0x43, 0x44 };
const basic_string<unsigned char> kTextString3(uChars);
wstring                           result3 = wstring(kTextString3.begin(), kTextString3.end());
if (StringUtil::AsciiToUnicode(kTextString3) != result3)
  return -1;
```

Naive Test Unit

```
unsigned char                  uChars[] = { 0x41, 0x42, 0x43, 0x44 };
const basic_string<unsigned char> kTextString3(uChars);
wstring                        resu                                    );
if (StringUtil::AsciiToUnicode(kTextStr
  return -1;
return 0;
```

(local variable) const std::basic_string<unsigned char> kTextString3

Search Online

C6054: String 'uChars' might not be zero-terminated.

Naive Test Unit

It's worth paying attention to your squiggles !

VS analyzer does a pretty good job keeping you safe.

```
AddressSanitizer: global-buffer-overflow on address 0x00c158ca at pc 0x00838b91 bp 0x016fef98 sp
0x016fef8c
READ of size 2 at 0x00c158ca thread T0

    #0 0x838b90 in StringUtil::StoreNULLSeparatedStrings C:\JobAI\platform\util\strings\StringProcessing.cpp:430
    #1 0x67edfb in TestStringUtilStoreNULLSeparatedStrings C:\JobAI\testunits\strings\StringProcessingTests.cpp:563
    #2 0x7e8035 in FunctionTest::Run C:\JobAI\testunits\Tester.cpp:71
    #3 0x7e8901 in Tester::RunTest C:\JobAI\testunits\Tester.cpp:186
    #4 0x7e6d2b in Tester::ExecuteCommandLine C:\JobAI\testunits\Tester.cpp:558
    #5 0x7d9821 in main C:\JobAI\testunits\comps\TestComponents.cpp:2236
    #6 0x9d92f2 in invoke_main crt\vcstartup\src\startup\exe_common.inl:78
    #7 0x9d91f6 in __scrt_common_main_seh crt\vcstartup\src\startup\exe_common.inl:288
    #8 0x9d909c in __scrt_common_main crt\vcstartup\src\startup\exe_common.inl:330
    #9 0x9d9357 in mainCRTStartup crt\vcstartup\src\startup\exe_main.cpp:16

0x00c158ca is located 0 bytes to the right of global variable '<C++ string literal>' defined in
'StringProcessingTests.cpp:561:9' (0xc158a0) of size 42

SUMMARY:
AddressSanitizer: global-buffer-overflow StringProcessing.cpp:430 in StringUtil::StoreNULLSeparatedStrings
```

Use the full power of your Debugger

Use the full power of your Debugger

Excessive Test Unit

```
...

buff = L"token0\0token1\0token2\0";

list.clear();
StringUtil::StoreNULLSeparatedStrings(buff, list);

if (list.size() != 3)
  return -1;
if (list[2] != L"token2")
  return -1;

...
```

Excessive Test Unit

```
...

buff = L"token0\0token1\0token2\0";

list.clear();
StringUtil::StoreNULLSeparatedStrings(buff, list);

if (list.size() != 3)
  return -1;
if (list[2] != L"token2")
  return -1;

...
```

Excessive Test Unit

```
/**
 * Creates a vector with strings that are separated by \0
 * aBuff - buffer containing NULL separated strings
 * aLen - the length of buffer
 * aSection - vector that contains the strings from aBuff
 */
void StoreNULLSeparatedStrings(const wchar_t * aBuff, DWORD aLen,
                               vector<wstring> & aStringList);

/**
 * Creates a vector with strings that are separated by \0 and end with \0\0
 * aBuff - buffer containing NULL separated strings
 * aSection - vector that contains the strings from aBuff
 */
void StoreNULLSeparatedStrings(const wchar_t * aBuff, vector<wstring> & aStringList);
```

Excessive Test Unit

```
/**
 * Creates a vector with strings that are separated by \0
 * aBuff - buffer containing NULL separated strings
 * aLen - the length of buffer
 * aSection - vector that contains the strings from aBuff
 */
void StoreNULLSeparatedStrings(const wchar_t * aBuff, DWO
                               vector<wstring> & aString

/**
 * Creates a vector with strings that are separated by \0 and end with \0\0
 * aBuff - buffer containing NULL separated strings
 * aSection - vector that contains the strings from aBuff
 */
void StoreNULLSeparatedStrings(const wchar_t * aBuff, vector<wstring> & aStringList);
```

OUT OF CONTRACT CALL

Just enough to wet your appetite

Go explore on your own...

🔍

📖 Explore Further

AddressSanitizer (ASan) for Windows with MSVC

devblogs.microsoft.com/cppblog/addresssanitizer-asan-for-windows-with-msvc/

AddressSanitizer for Windows: x64 and Debug Build Support

devblogs.microsoft.com/cppblog/asan-for-windows-x64-and-debug-build-support/

by Augustin Popa

@augustin_popa

Part III

Warm Fuzzy Feelings

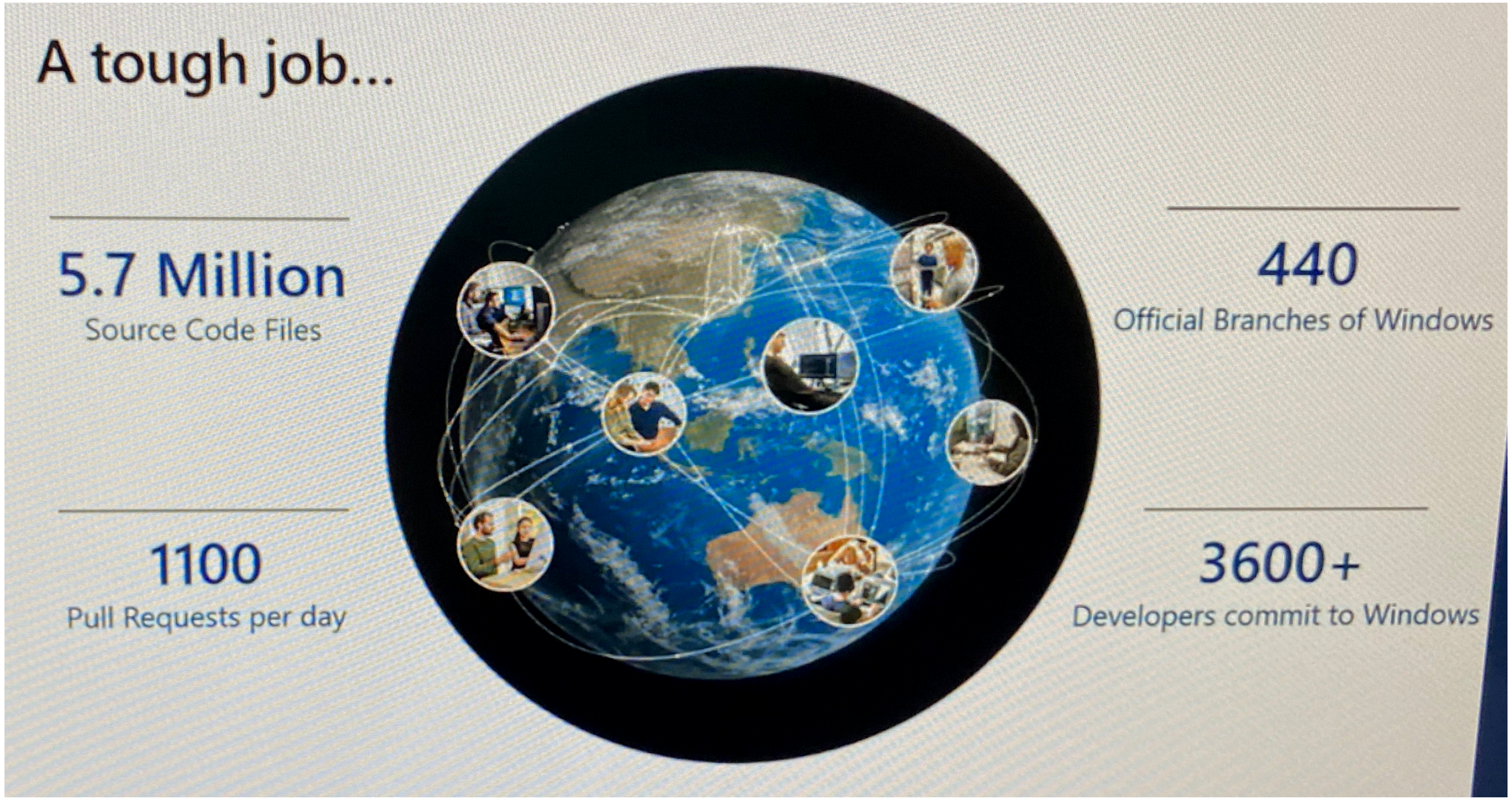
Sanitizers + Fuzzing

💪

Automatically generate inputs to you program to crash it.

Sanitizers + Fuzzing

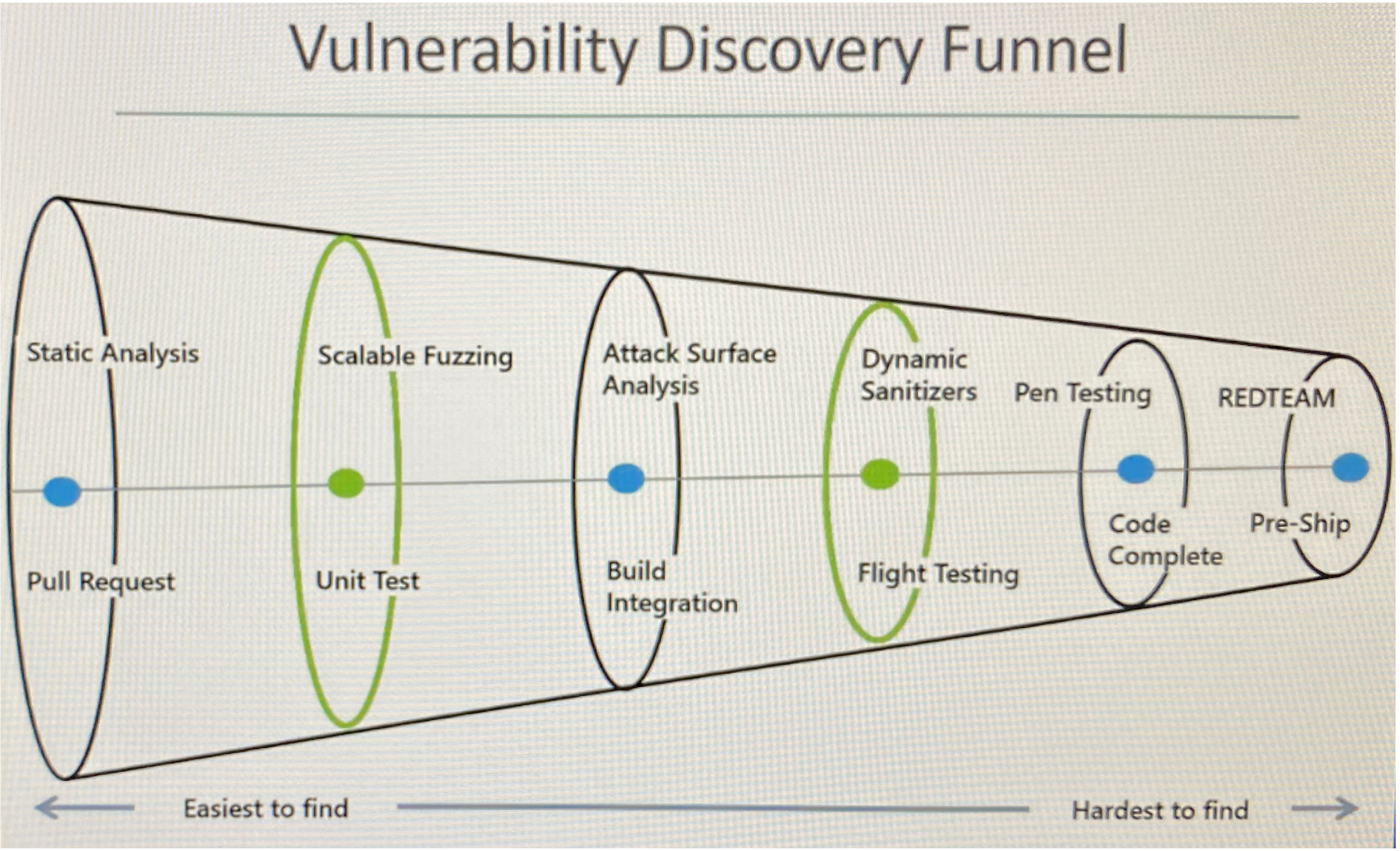
Case study at Microsoft Windows scale



https://sched.co/e7C0

Sanitizers + Fuzzing

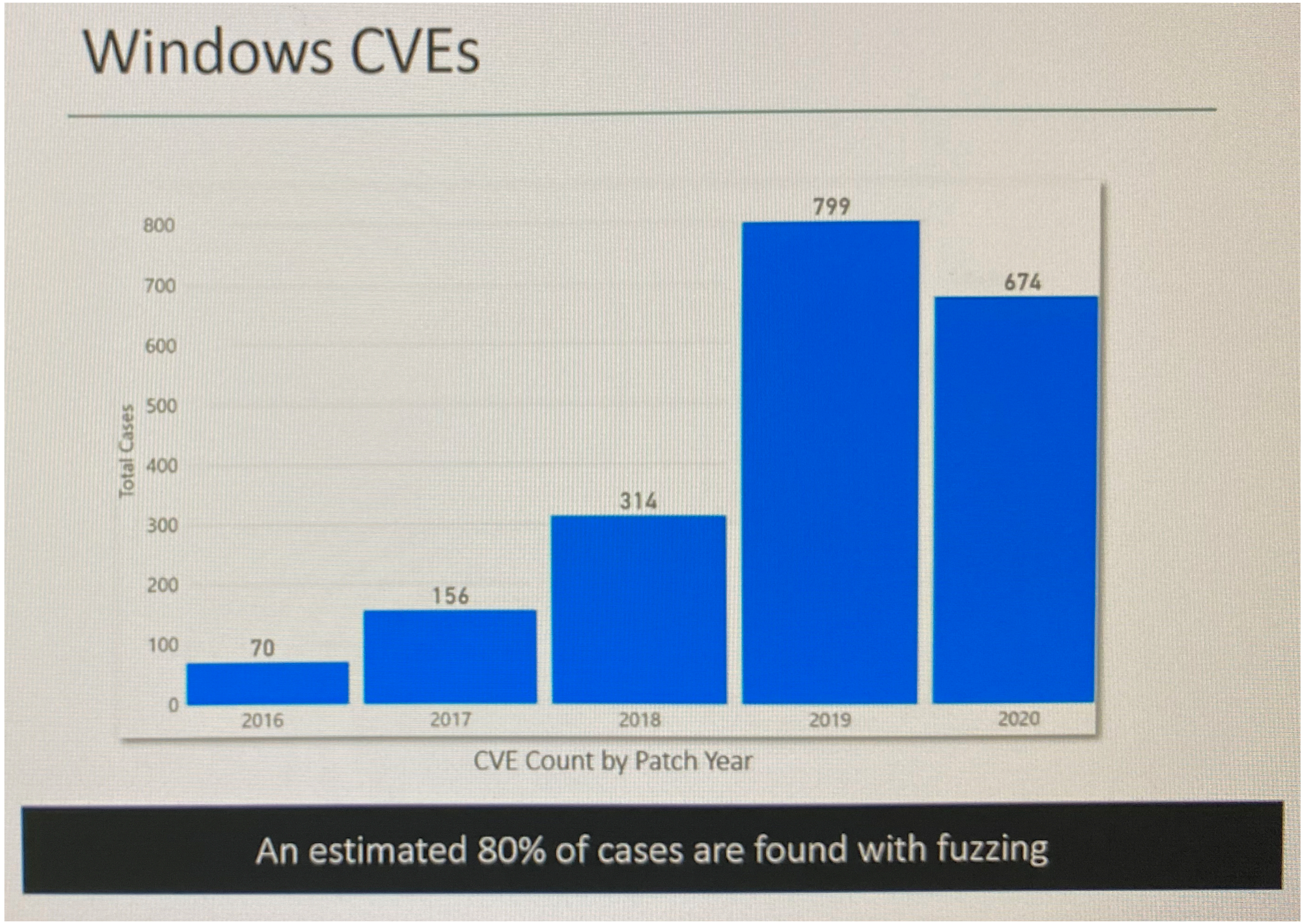
Case study at Microsoft Windows scale



https://sched.co/e7C0

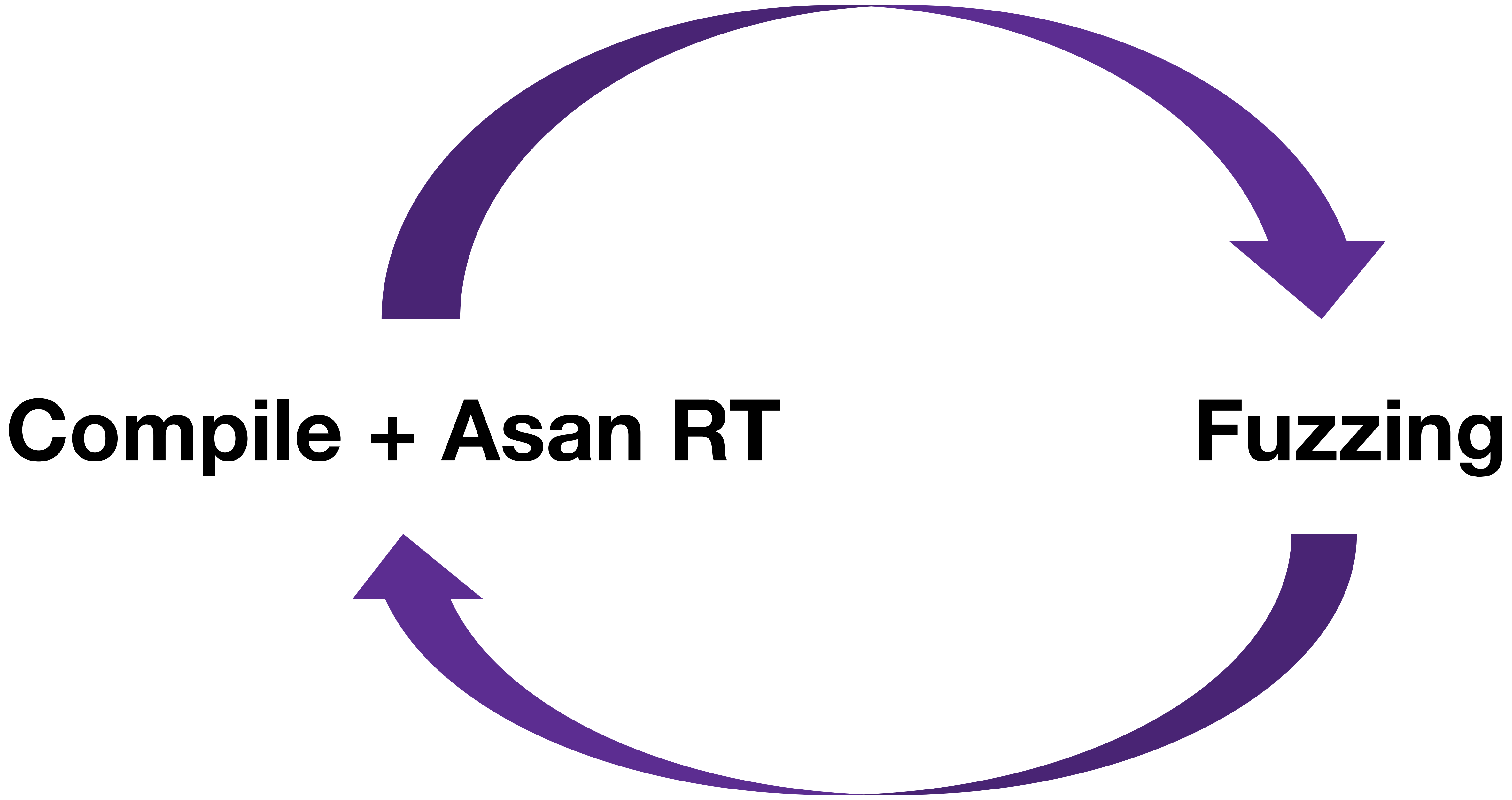
Sanitizers + Fuzzing

Case study at Microsoft Windows scale



https://sched.co/e7C0

Workflow



{ ASan + Fuzzing } => Azure

What is Microsoft Security Risk Detection?

Security Risk Detection is Microsoft's unique fuzz testing service for finding security critical bugs in software. Security Risk Detection helps customers quickly adopt practices and technology battle-tested over the last 15 years at Microsoft.

READ SUCCESS STORIES >

"Million dollar" bugs

Security Risk Detection uses "Whitebox Fuzzing" technology which discovered 1/3rd of the "million dollar" security bugs during Windows 7 development.

Battle tested tech

The same state-of-the-art tools and practices honed at Microsoft for the last decade and instrumental in hardening Windows and Office — with the results to prove it.

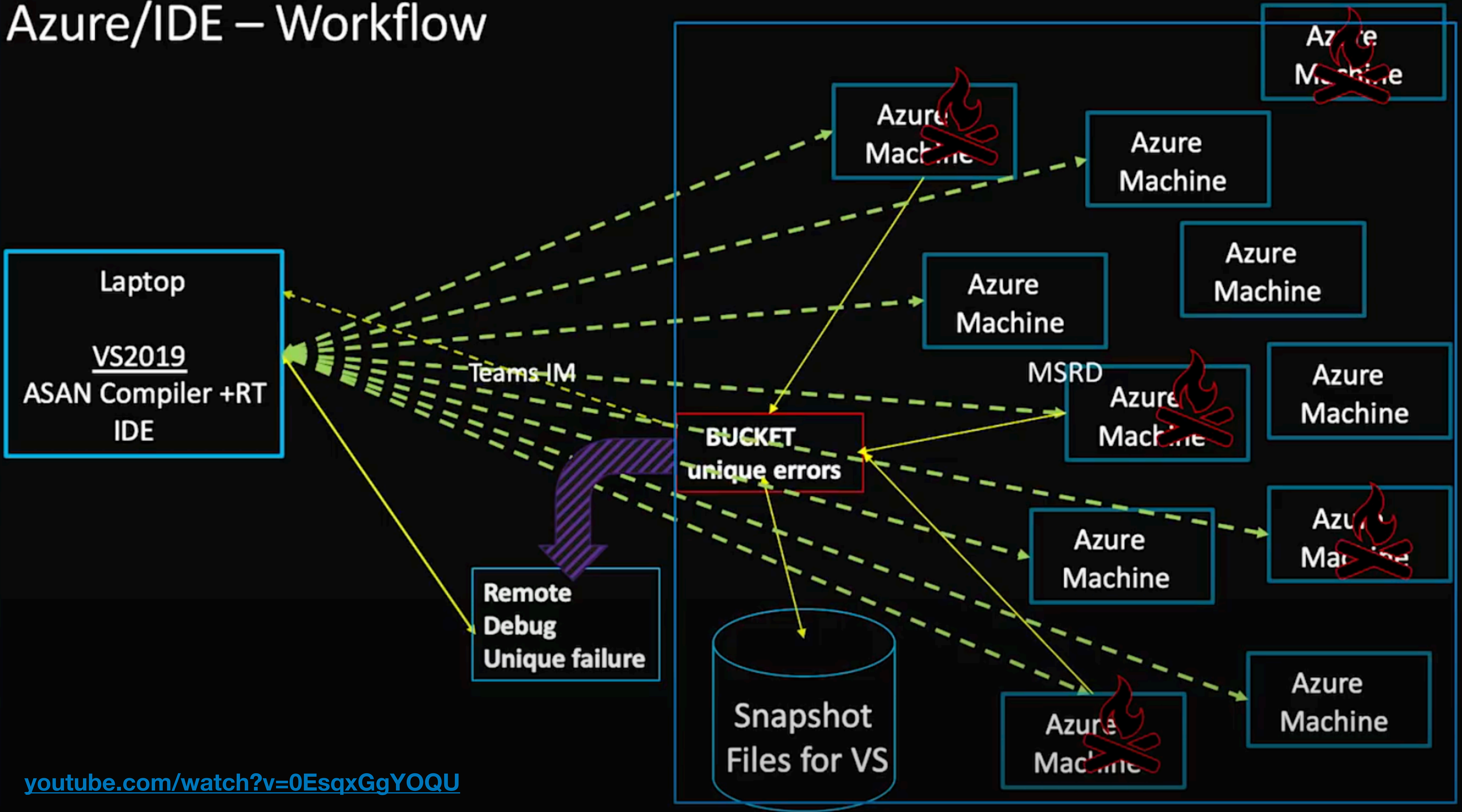
Scalable fuzz lab in the cloud

One click scalable, automated, Intelligent Security testing lab in the cloud.

Cross-platform support

Linux Fuzzing is now available. So, whether you're building or deploying software for Windows or Linux or both, you can utilize our Service.

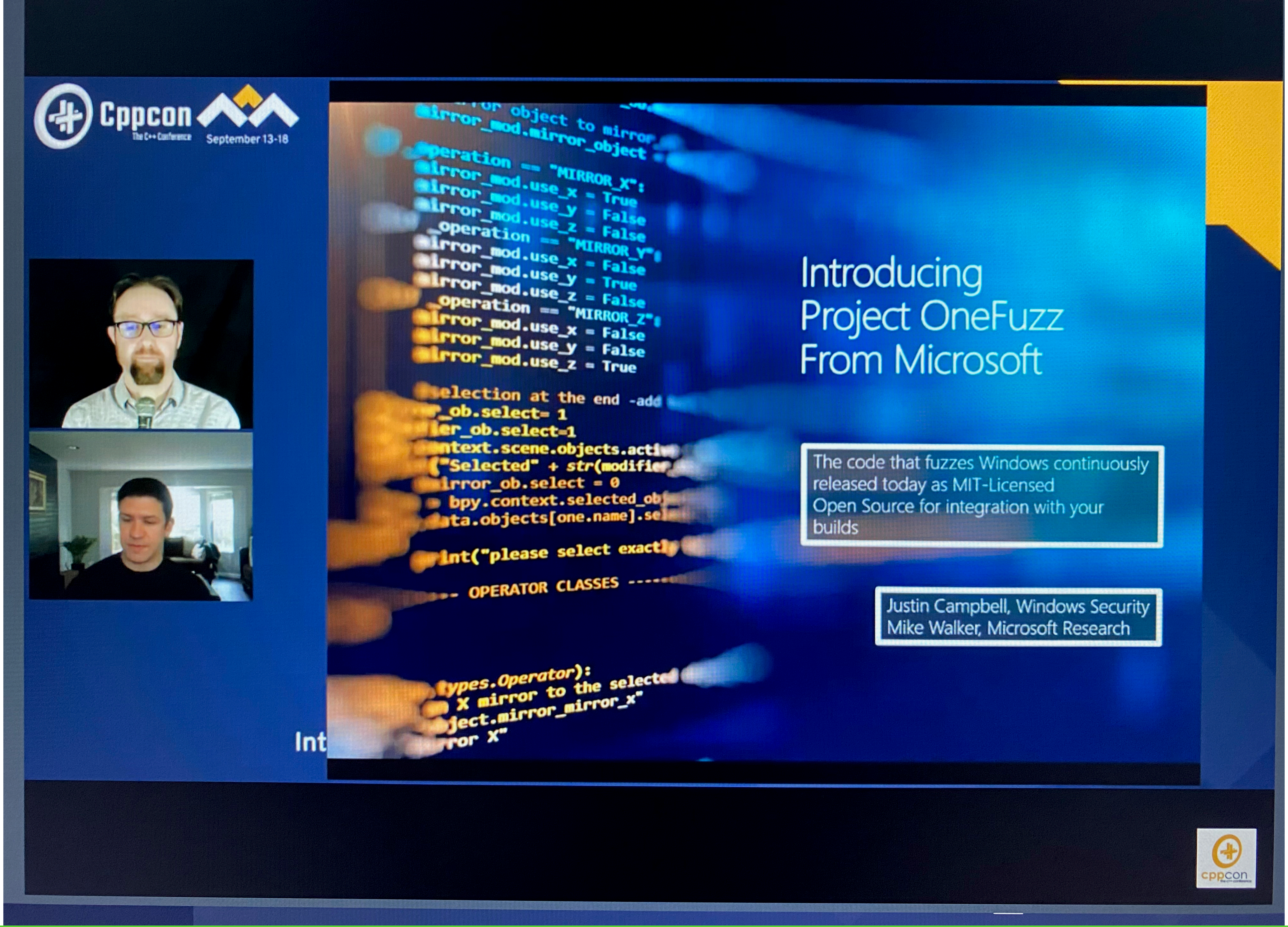
Azure/IDE – Workflow



youtube.com/watch?v=0EsqxGgYOQU

Microsoft OneFuzz

a platform you will be able to download from Github and run fuzzing on premise or in Azure


Cppcon
The C++ Conference
September 13-18
Introducing
Project OneFuzz
From Microsoft
The code that fuzzes Windows continuously
released today as MIT-Licensed
Open Source for integration with your
builds
Justin Campbell, Windows Security
Mike Walker, Microsoft Research
cppcon

Project OneFuzz

September 15, 2020

Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale

Justin Campbell   Principal Security Software Engineering Lead, Microsoft Security

Mike Walker   Senior Director, Special Projects Management, Microsoft Security

A self-hosted Fuzzing-As-A-Service platform

microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/

A self-hosted Fuzzing-As-A-Service platform

github.com/microsoft/onefuzz

Project OneFuzz
CI/CD

🔥🔥🔥

New unique crashes create notifications:

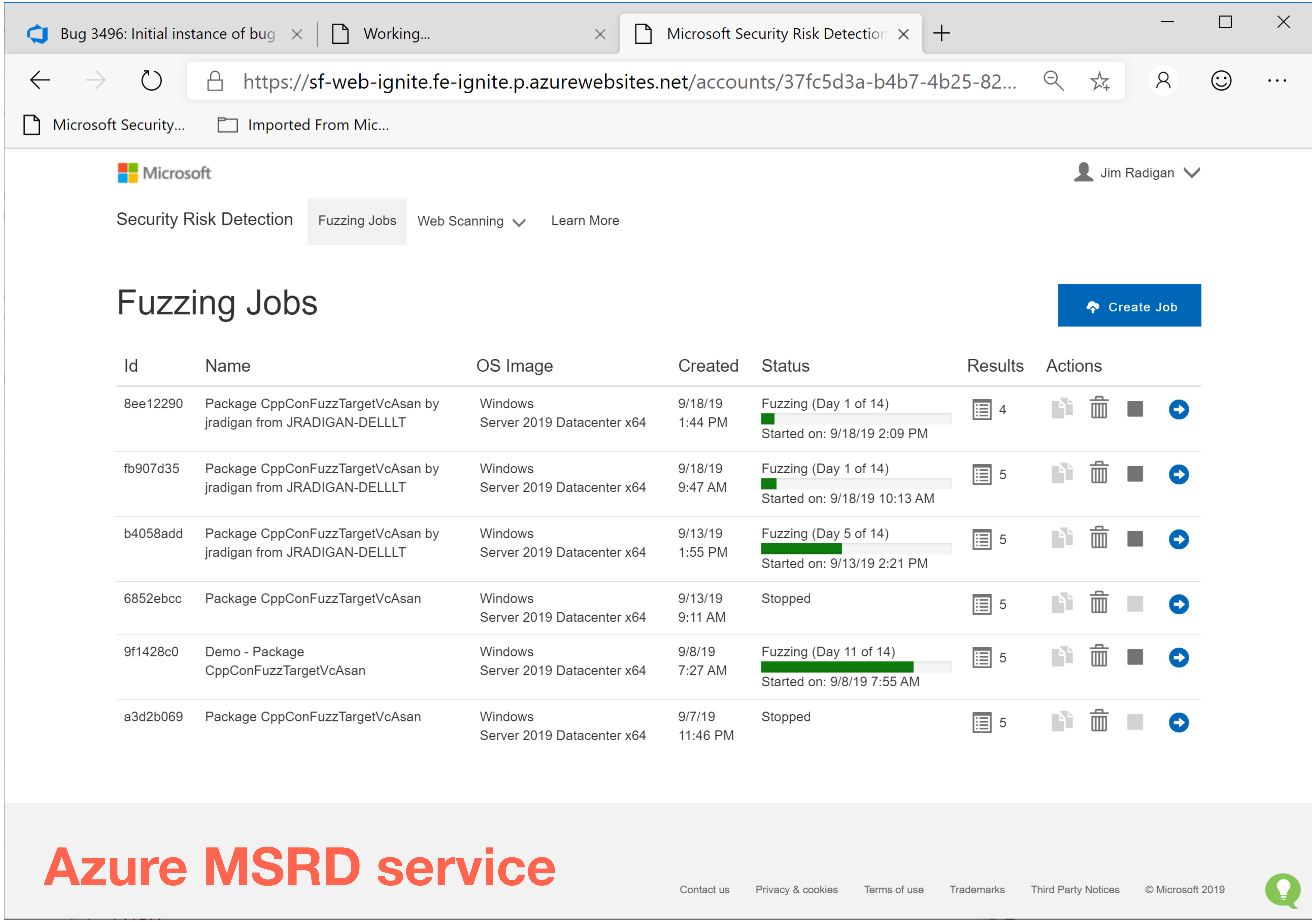
- **Teams**
- **ADO work items**

Azure DevOps Pipeline

GitHub Actions

github.com/microsoft/onefuzz-samples

{ ASan + Fuzzing } => Azure



Bug 3496: Initial instance of bug | Working... | Microsoft Security Risk Detection

https://sf-web-ignite.fe-ignite.p.azurewebsites.net/accounts/37fc5d3a-b4b7-4b25-82...

Microsoft Security... | Imported From Mic...

Microsoft — Jim Radigan

Security Risk Detection | Fuzzing Jobs | Web Scanning | Learn More

Fuzzing Jobs

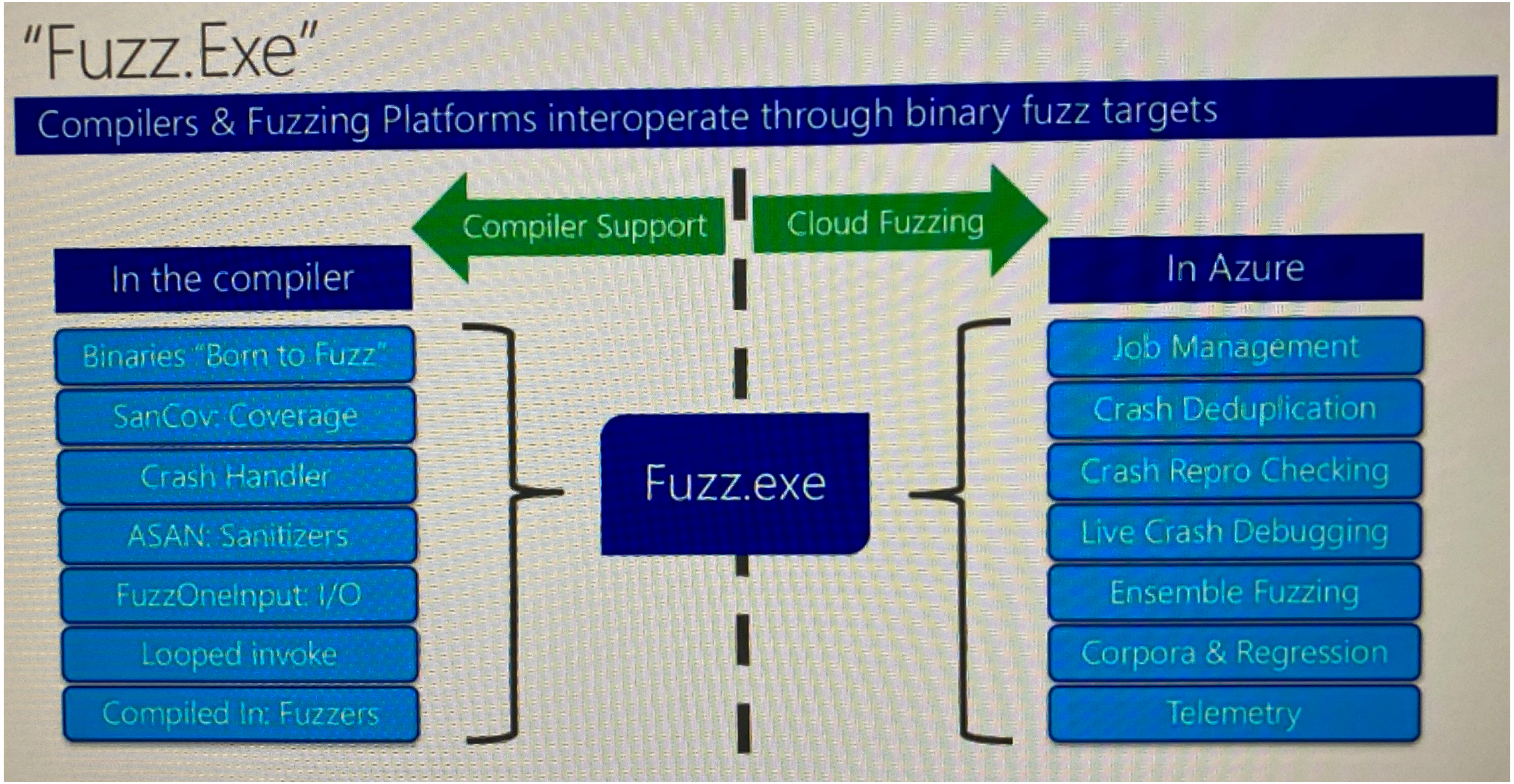
Create Job

Id	Name	OS Image	Created	Status	Results	Actions
8ee12290	Package CppConFuzzTargetVcAsan by jradigan from JRADIGAN-DELLLT	Windows Server 2019 Datacenter x64	9/18/19 1:44 PM	Fuzzing (Day 1 of 14) Started on: 9/18/19 2:09 PM	4	
fb907d35	Package CppConFuzzTargetVcAsan by jradigan from JRADIGAN-DELLLT	Windows Server 2019 Datacenter x64	9/18/19 9:47 AM	Fuzzing (Day 1 of 14) Started on: 9/18/19 10:13 AM	5	
b4058add	Package CppConFuzzTargetVcAsan by jradigan from JRADIGAN-DELLLT	Windows Server 2019 Datacenter x64	9/13/19 1:55 PM	Fuzzing (Day 5 of 14) Started on: 9/13/19 2:21 PM	5	
6852ebcc	Package CppConFuzzTargetVcAsan	Windows Server 2019 Datacenter x64	9/13/19 9:11 AM	Stopped	5	
9f1428c0	Demo - Package CppConFuzzTargetVcAsan	Windows Server 2019 Datacenter x64	9/8/19 7:27 AM	Fuzzing (Day 11 of 14) Started on: 9/8/19 7:55 AM	5	
a3d2b069	Package CppConFuzzTargetVcAsan	Windows Server 2019 Datacenter x64	9/7/19 11:46 PM	Stopped	5	

Contact us | Privacy & cookies | Terms of use | Trademarks | Third Party Notices | © Microsoft 2019

Azure MSRD service

{ ASan + Fuzzing }



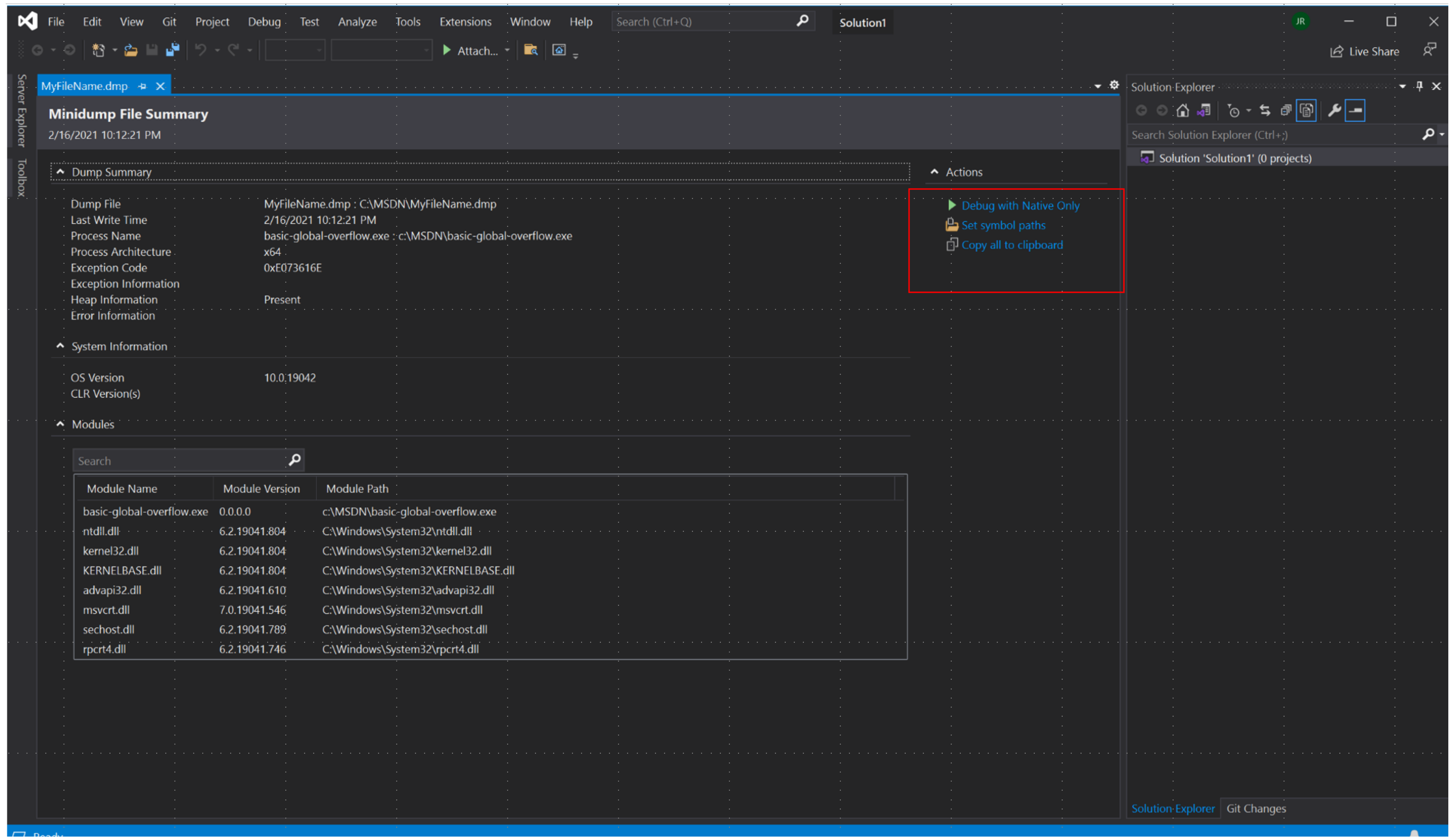
https://sched.co/e7C0

ASAN cloud / distributed testing

You can create the **dump** on test or production infrastructure
where the failure occurs, and debug it later on your developer PC

Crash dumps are created upon AddressSanitizer failures
by setting the following environment variable:

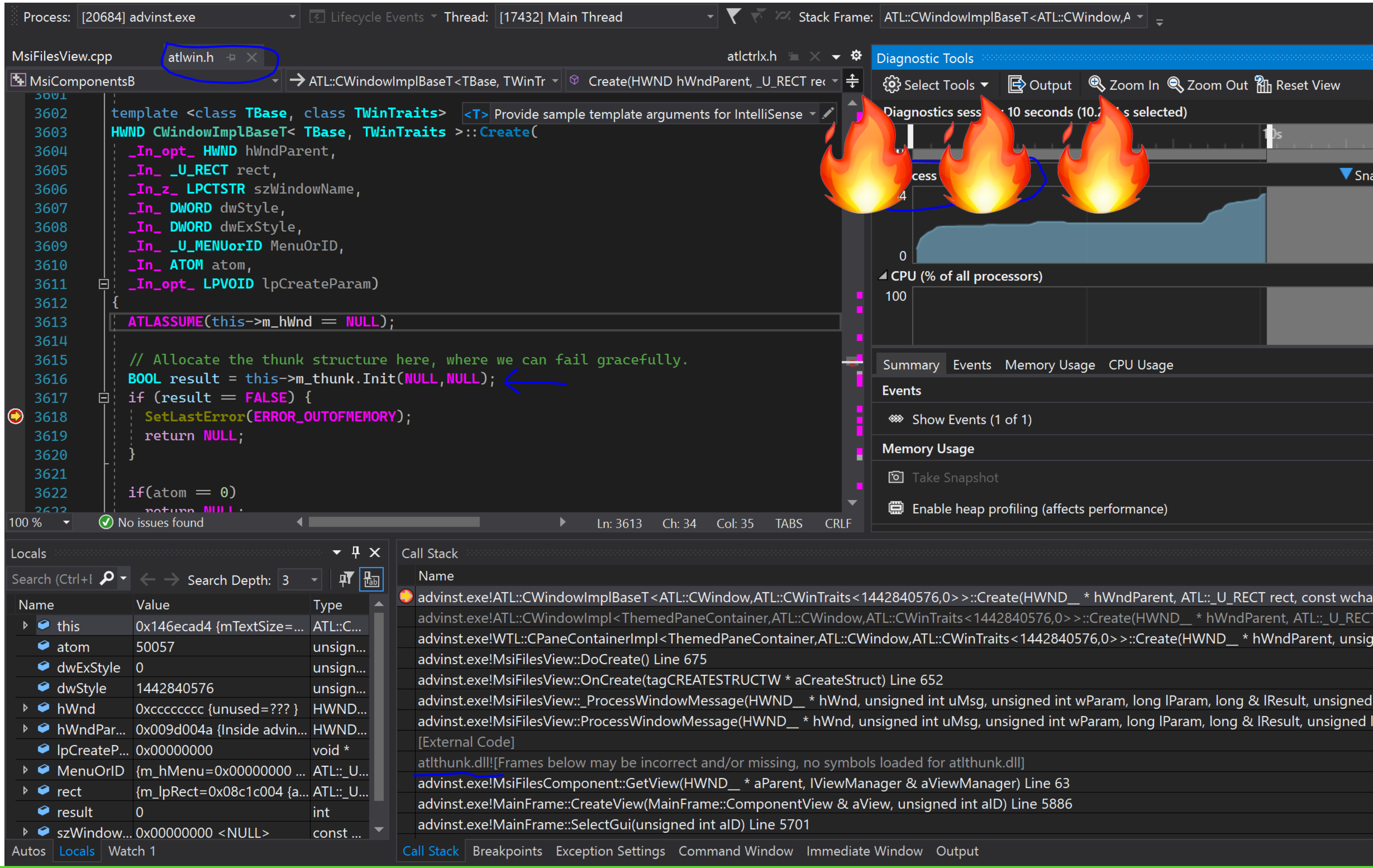
```
set ASAN_SAVE_DUMPS=MyFileName.dmp
```



docs.microsoft.com/en-us/cpp/sanitizers/asan-offline-crash-dumps

I hope you're now as excited as I am for leveraging the power of ASan on Windows

Looking forward to many days of bug-fixing ahead 😬



ASan Testing 🚗💨 Dieselgate style :)

```
int main() {
    #ifdef __SANITIZE_ADDRESS__
        printf("Address sanitizer enabled");
    #else
        printf("Address sanitizer not enabled");
    #endif
    return 1;
}
```

```
__declspec(no_sanitize_address)
void test1() {
    int x[100];
    x[100] = 5; // ASan exception not caught
}

void test2() {
    __declspec(no_sanitize_address) int x[100];
    x[100] = 5; // ASan exception not caught
}

__declspec(no_sanitize_address) int g[100];
void test3() {
    g[100] = 5; // ASan exception not caught
}
```

😈

Q & A

Address Sanitizer on Windows

accu 2021

@ciura_victor

Victor Ciura
Principal Engineer
CAPHYON