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 **obvious** bugs/vulnerabilities in that well tested component?
Probably not

\_(ツ)_/\n
¯\_\_(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::CSimpleStringT<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 0x3e1660 in FunctionTest::Run C:\JobAI\testunits\Tester.cpp:71
#9 0x3e1660 in Tester::RunTest C:\JobAI\testunits\Tester.cpp:186
#10 0x5880e5 in TestSerVal::DecryptString C:\JobAI\advinst\msicomp\serval\SerValUtil.cpp:85
#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
Due to the nature of delivery medium & streaming delays, I prefer to take questions at the end.
Keynote:
Refactoring Superpowers:
Make Your IDE Do Your Work, Faster and More Safely

Clare Macrae
Safe, How?

IDE

Tests

Version control
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

recent compiler(s)
[conformant/strict]

perf profiler
test framework

static analyzer
dynamic analyzer
(runtime)

CI/CD service
code reviews platform
+ 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


VS 16.7
## Standard C/C++ rule sets

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

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

Static Analysis

Visual Studio integrates with
- MSVC Code Analysis [https://aka.ms/cpp/ca/bg](https://aka.ms/cpp/ca/bg)
- Clang-tidy [https://aka.ms/cpp/clangtidy](https://aka.ms/cpp/clangtidy)
- Visual Studio Code Linters [https://aka.ms/cpp/linter](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
clang-tidy

~ 300 checks

clang.llvm.org/extra/clang-tidy/checks/list.html
- 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
- `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-*`
• 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

<table>
<thead>
<tr>
<th>Check</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>bugprone-argument-comment</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-assert-side-effect</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-bool-pointer-implicit-conversion</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-branch-clone</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-copy-constructor-init</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-dangling-handle</td>
<td>On</td>
</tr>
<tr>
<td>bugprone-forwarding-reference-overload</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-inaccurate-erase</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-incorrect-roundings</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-integer-division</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-lambda-function-name</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-macro-parentheses</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-macro-repeated-side-effects</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-misplaced-operator-in-strlen-in-alloc</td>
<td>Off</td>
</tr>
<tr>
<td>bugprone-misplaced-widening-cast</td>
<td>Off</td>
</tr>
</tbody>
</table>

The **bugprone-dangling-handle** check is selected to detect dangling references in value handles like `std::experimental::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.
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.

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.

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

clang -Wlifetime Experimental

Lifetime profile v1.0

Lifetime safety: Preventing common dangling

```cpp
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

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

```cpp
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 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

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

Use checks from: CustomChecks
Predefined Checks: Select
Custom Checks: modernize-* ➔ your custom checks
Header filter: *
Custom executable: C:\dev\llvm\bin\clang-tidy.exe ➔ your custom
clang-tidy build
Format after Tidy: ✔️
Tidy on save: ❌
Tidy file config: Export
Write **custom** checks for your needs (project specific)

Run them regularly!
Explore Further

Explore Further

https://www.youtube.com/watch?v=JPnN2c2odNY
What About Developer Workflow?

😊 + 📱

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

Exploring Clang Tooling, Part 1: Extending Clang-Tidy

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**

**Exploring Clang Tooling: Using Build Tools with clang-tidy**
Explore Further

More blog articles by Stephen Kelly

**Future Developments in clang-query**

**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
<table>
<thead>
<tr>
<th>Workloads</th>
<th>Individual components</th>
<th>Language packs</th>
<th>Installation locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compilers, build tools, and runtimes

- [ ] C++ Clang Compiler for Windows (11.0.0)
- [ ] C++ Clang-cl for v142 build tools (x64/x86)
Visual Studio 2019 since v16.2

clang-cl.exe
Visual Studio 2019
since v16.4

clang-tidy

code analysis

Visual Studio 2019 since v16.4
Visual Studio 2019
since v16.4

clang-tidy warnings

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
Clang Power Tools

www.clangpowertools.com

LLVM

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

Visual Studio

2015 / 2017 / 2019

Free/OSS
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
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+

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](https://github.com/google/sanitizers)
Meeting C++ Community Survey

Which sanitizers do you use in your builds? (n=1302)

- none
- Address
- Leak
- Memory
- UB
- Thread

meetingcpp.com/mcpp/survey/?q=19
Common Vulnerabilities and Exposures

Memory safety continues to dominate
Address Sanitizer (ASan)

De facto standard for detecting memory safety issues

It’s important for basic correctness and sometimes true vulnerabilities

[Link: 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](https://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](https://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/
Address Sanitizer + Fuzzing + VS2019

jradigan@Microsoft.com

Visual Studio 2019 launch

Address Sanitizer + Fuzzing + VS20149

Video Sponsorship Provided By:

AURORA

CppCon 2019: Jim Radigan C++ Sanitizers and Fuzzing for the Windows Platform Using New Compilers...

https://www.youtube.com/watch?v=0EsqxGgYOQU
Visual Studio 2019 since v16.4

Installation details

- Desktop development with C++
  - Includes
    - C++ core desktop features
    - IntelliCode
- Optional
  - MSVC v142 - VS 2019 C++ x64/x86 build tools (...)
  - Windows 10 SDK (10.0.18362.0)
  - Just-In-Time debugger
  - C++ profiling tools
  - C++ CMake tools for Windows
  - C++ ATL for latest v142 build tools (x86 & x64)
  - Test Adapter for Boost.Test
  - Test Adapter for Google Test
  - Live Share
- C++ AddressSanitizer (Experimental)
  - IntelliTrace
  - C++ MFC for latest v142 build tools (x86 & x64)
  - C++/CLI support for v142 build tools (14.24)
  - C++/CLI modules for v142 build tools (x64/x86)
  - C++ Clang tools for Windows (8.0.1 - x64/x86)
## Visual Studio 2019

**since v16.4**

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Individual components</th>
<th>Language packs</th>
<th>Installation locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>sanitizer</td>
<td>☑ C++ AddressSanitizer (Experimental)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Debugging and testing

- **C++ AddressSanitizer (Experimental)**

  AddressSanitizer (ASAN) is a tool for detecting memory errors in C/C++ code. ASAN uses instrumentation to check memory accesses and report any memory safety issues. This feature is experimental and should not be used outside of testing environments.
Visual Studio 2019 since v16.4

Tech Preview
October 2019
Visual Studio 2019 since v16.7

Tech Preview
August 2020

x64 & Debug builds
Visual Studio 2019 since v16.7

since v16.7 August 2020

+ x64 & Debug builds

support all Debug runtimes: /MTd /MDd

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
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)
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/
ASAN is out of Experimental => GA

devblogs.microsoft.com/cppblog/address-sanitizer-for-msvc-now-generally-available
Visual Studio 2019

v16.9

March 2021
Visual Studio 2019
v16.9

- expanded RtlAllocateHeap support (fixed compatibility issue with RtlCreateHeap and RtlAllocateHeap interceptors when creating executable memory pools)
Visual Studio 2019

v16.9

- expanded \texttt{RtlAllocateHeap} support (fixed compatibility issue with \texttt{RtlCreateHeap} and \texttt{RtlAllocateHeap} interceptors when creating executable memory pools)
- support for the legacy \texttt{GlobalAlloc} and \texttt{LocalAlloc} family of memory functions
  \( \texttt{ASAN\_OPTIONS=windows\_hook\_legacy\_allocators=true} \)
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
  - \( \text{ASAN\_OPTIONS} = \text{windows\_hook\_legacy\_allocators=true} \)
- explicit error messages for shadow memory interleaving and interception failure
expanding \texttt{RtlAllocateHeap} support (fixed compatibility issue with \texttt{RtlCreateHeap} and \texttt{RtlAllocateHeap} interceptors when creating executable memory pools)

- support for the legacy \texttt{GlobalAlloc} and \texttt{LocalAlloc} family of memory functions
  \begin{verbatim}
  ASAN\_OPTIONS=windows\_hook\_legacy\_allocators=true
  \end{verbatim}

- explicit \textbf{error messages} for shadow memory interleaving and interception failure

- IDE integration can now handle the complete collection of \textbf{exceptions} which ASan can report
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
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

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 strncpy 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
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

Structured Exception Handling (SEH) /EHa

non-standard C++
Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

Structured Exception Handling (SEH) /EHa

AV traps 0xc0000005
Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

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

- Structured Exception Handling (SEH) /EHa
- AV traps 0xc0000005
- vast amount of legacy code (really, really, really OLD code)
- COM

non-standard C++
Challenges bringing ASan to Windows

the surface area of the Microsoft platform is enormous

Structured Exception Handling (SEH) /EH

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

Structured Exception Handling (SEH) /EHa

AV traps \texttt{0xc0000005}

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

COM

Managed C++

ASan runtime interop with managed code (.NET)

non-standard C++
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
ASan Visual Studio integration:

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

[aka.ms/asan](aka.ms/asan)
Visual Studio ASan CMake

CMakeSettings.json

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

> build with /fsanitize:address
Address Sanitizer (ASan)

```cpp
#include <iostream>

int main()
{
    int* array = new int[100];
    array[100] = 1; // Exception Unhandled
}
```

Exception Unhandled

Address Sanitizer Error: Heap buffer overflow

Full error details can be found in the output window
Address Sanitizer (ASan)

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

ASan logging information => Output window
AddressSanitizer: stack-use-after-scope on address 0x0055fc68 at pc 0x793d62de bp 0x0055fb4 sp 0x0055fb8

==27748==ERROR: AddressSanitizer: stack-use-after-scope on address 0x0055fc68
d:\work\5\vlvm\projects\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:764
#0 0x793d62f6 in __asan_wrap_memset d:\_work\5\vlvm\projects\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:76
#1 0x77dd46e7 in __asan_wrap_memset d:\_work\5\vlvm\projects\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:76
#2 0x77dd4ce1 in __asan_wrap_memset d:\_work\5\vlvm\projects\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:76
#3 0x75d408fe in __asan_wrap_memset d:\_work\5\vlvm\projects\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:76
#4 0xa52da7 in common_exit minkernel\crts\ucrt\src\appcr\startup\exit.cpp:28
#5 0xa52da7 in exit minkernel\crts\ucrt\src\appcr\startup\exit.cpp:28
#6 0xa52f6 in exit minkernel\crts\ucrt\src\appcr\startup\exit.cpp:28
#7 0x75ef6358 in __asan_wrap_memset d:\_work\5\vlvm\projects\compiler-rt\lib\sanitizer_common\sanitizer_common_interceptors.inc:76

Address 0x0055fc68 is located in stack of thread T0

Shadow bytes around the buggy address:
0x300abf30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x300abf70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x300abf80: 00 00 00 00 00 00 00 00 00 00 00 00 [00] 00 00 0
0x300abf90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0x300abfd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0

Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable:          0
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.

Changes the way you report a bug, in general

+ Live Share
Exception Unhandled

ASAN Error: Stack Buffer Overflow

Full error details can be found in the output window

Copy Details  Start collaboration session...

Exception Settings

Loaded
How does it work?
ASan is just Malware, used for Good
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
==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:1
#1 0x7ff6e3a66ce8 in invoke_main D:\agent\_work\9\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl:78
#2 0x7ff6e3a66bdc 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 0x7fffee9a76fd3 in BaseThreadInitThunk+0x13 (C:\WINDOWS\System32\KERNEL32.DLL+0x180016fd3)
#6 0x7ffeeda97cec0 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 0x7fffe83be7e91 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 0x7ff6e3a66bdc 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 0x7fffee9a76fd3 in BaseThreadInitThunk+0x13 (C:\WINDOWS\System32\KERNEL32.DLL+0x180016fd3)
#7 0x7ffeeda97cec0 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:
  0x04d981eefe0: 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 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 00 00 00 00 00 00
  =>0x04d981ef0030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  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
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

Process Memory

Shadow Memory

my allocated memory

Poisoned memory

Red zones
if (ShadowByte::IsBad(p))
Assembly::Report(p, sz)

*p = 0xbadf00d

If the shadow byte is poisoned,
ASAN runtime reports the problem and crashes the application
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
Lookups into shadow memory need to be very fast

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

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

Location of shadow region in memory

Stack use after scope
Shadow Mapping

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

*p = 0xf00d
Shadow Mapping

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

*p = 0xbadf00d
Heap Red Zones

malloc()

| alloc 1 | alloc 2 | alloc 3 | alloc 4 | alloc 5 |

ASAN malloc()

| alloc 1 | alloc 2 | alloc 3 | alloc 4 | alloc 5 |
Heap Red Zones

**ASAN malloc()**

```
alloc 1  alloc 2  alloc 3  alloc 4  alloc 5
```

**Shadow Memory**

```
alloc 1  alloc 2  alloc 3  alloc 4  alloc 5
```

Poisoned memory
Heap Red Zones

ASAN malloc()

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

Detect:
- heap underflows/overflows
- use-after-free & double free

Shadow Memory
Stack Red Zones

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

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

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

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

left red zone

mid red zone

right red zone
AddressSanitizer ContainerOverflow

`std::vector<T>`

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

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

std::vector<T>

begin() end() poisoned memory 0x0fc

capacity() 0x0fc

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];

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 \(\sim 2x\)

[More information](https://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
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:

```
libcruntime.lib (C:\\Computer\\User\\SomeProject) error LNK2021 unresolved external symbol _flt_divide referenced in function "_somefunc" unresolved symbol _somefunc referenced in function "_someotherfunc" unresolved symbol _someotherfunc referenced in function "_anotherfunc" unresolved symbol _anotherfunc referenced in function "_yetanotherfunc" unresolved symbol _yetanotherfunc referenced in function "_lastfunc"
```

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:

![Exception Settings](image)

Uncheck **Access violation**.
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/](devblogs.microsoft.com/cppblog/asan-for-windows-x64-and-debug-build-support/)
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
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
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)'
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]
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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
> libcrt.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]
>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
In general, if you have **overrides** for:

```c
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

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());
Somewhere inside

```cpp
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 0x194821 in Tester::RunTest<__cdecl>(__cdecl)\(void)> C:\JobAI\testunits\common_crt\CommonCrtTests.cpp:93
    #3 0x194b65 in main C:\JobAI\testunits\common_crt\main.cpp:22
    #4 0x1cc142 in invoke_main crt\vcstartup\src\startup\exe_common.inl:78
    #5 0x1cc046 in __scrt_common_main seh crt\vcstartup\src\startup\exe_common.inl:288
    #6 0x1cc046 in __scrt_common_main seh crt\vcstartup\src\startup\exe_common.inl:288
    #7 0x1cc046 in __scrt_common_main seh crt\vcstartup\src\startup\exe_common.inl:288
    #8 0x1cc046 in __scrt_common_main seh crt\vcstartup\src\startup\exe_common.inl:288
    #9 0x7645fa28 in BaseThreadInitThunk+0x18 (C:\WINDOWS\System32\KERNEL32.DLL+0x6b81fa28)
   #10 0x773e7683 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 overlaps 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));
...

2021 Victor Ciura | @ciura_victor - Address Sanitizer on Windows
Naive Test Unit

```cpp
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 0x7ff6da711d67 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 0x7ff6da96d8b0 in FunctionTest::Run C:\JobAI\testunits\Tester.cpp:71
   #5 0x7ff6da98fb05 in Tester::RunTest C:\JobAI\testunits\Tester.cpp:186
   #6 0x7ff6da99b3b4 in Tester::ExecuteCommandLine C:\JobAI\testunits\Tester.cpp:558
   #7 0x7ff6da97b59e in main C:\JobAI\testunits\comps\TestComponents.cpp:2236
   #8 0x7ff6da6b196d 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

```cpp
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

```cpp
unsigned char uChars[] = { 0x41, 0x42, 0x43, 0x44 };

const basic_string<unsigned char> kTextString3(uChars);

wstring
if (StringUtil::AsciiToUnicode(kTextString3, result))
  return -1;
return 0;

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

Search Online

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

```c
/**
 * 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, 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);
Just enough to wet your appetite

Go explore on your own...
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 your program to crash it.
Sanitizers + Fuzzing

Case study at Microsoft Windows scale

A tough job...

5.7 Million
Source Code Files

440
Official Branches of Windows

1100
Pull Requests per day

3600+
Developers commit to Windows

https://sched.co/e7C0
Sanitizers + Fuzzing

Case study at Microsoft Windows scale

https://sched.co/e7C0
Sanitizers + Fuzzing
Case study at Microsoft Windows scale

Windows CVEs

CVE Count by Patch Year

An estimated 80% of cases are found with fuzzing

https://sched.co/e7C0
Workflow

Compile + Asan RT  Fuzzing
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

Laptop

VS2019
ASAN Compiler + RT IDE

Teams IM

Remote Debug Unique failure

BUCKET
unique errors

Snapshot Files for VS

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
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
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

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
{ 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
```

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 :)  😈

```c
int main() {
    #ifdef __SANITIZE_ADDRESS__
    printf("Address sanitizer enabled\n");
    #else
    printf("Address sanitizer not enabled\n");
    #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