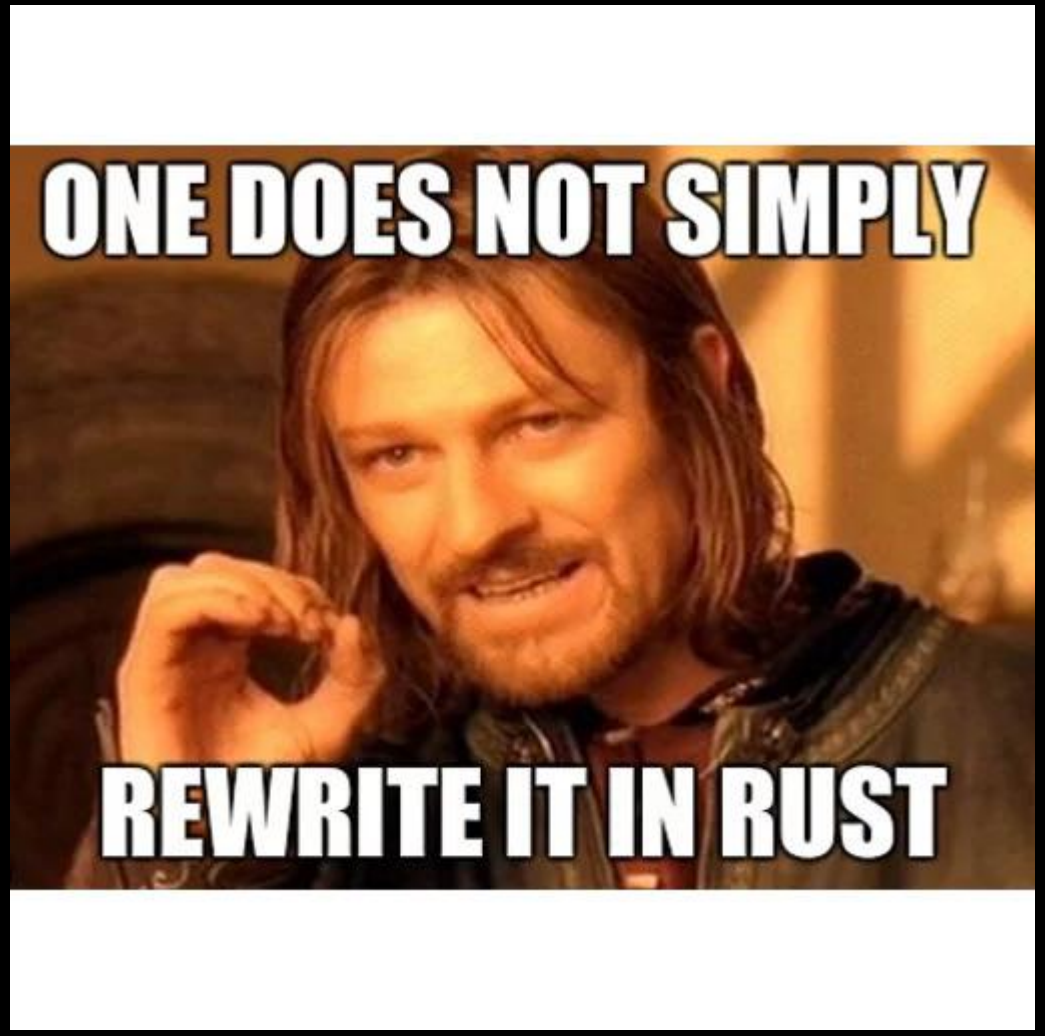
The time is now – practical mem safety

David “dwizzzle” Weston

Microsoft, VP OS Security and Enterprise



Securing the “world’s computer”

A tough job…

5.7 Million
Source Code Files

1100
Pull Requests per day

440
Official Branches of Windows

3600+
Developers committing to Windows

Windows

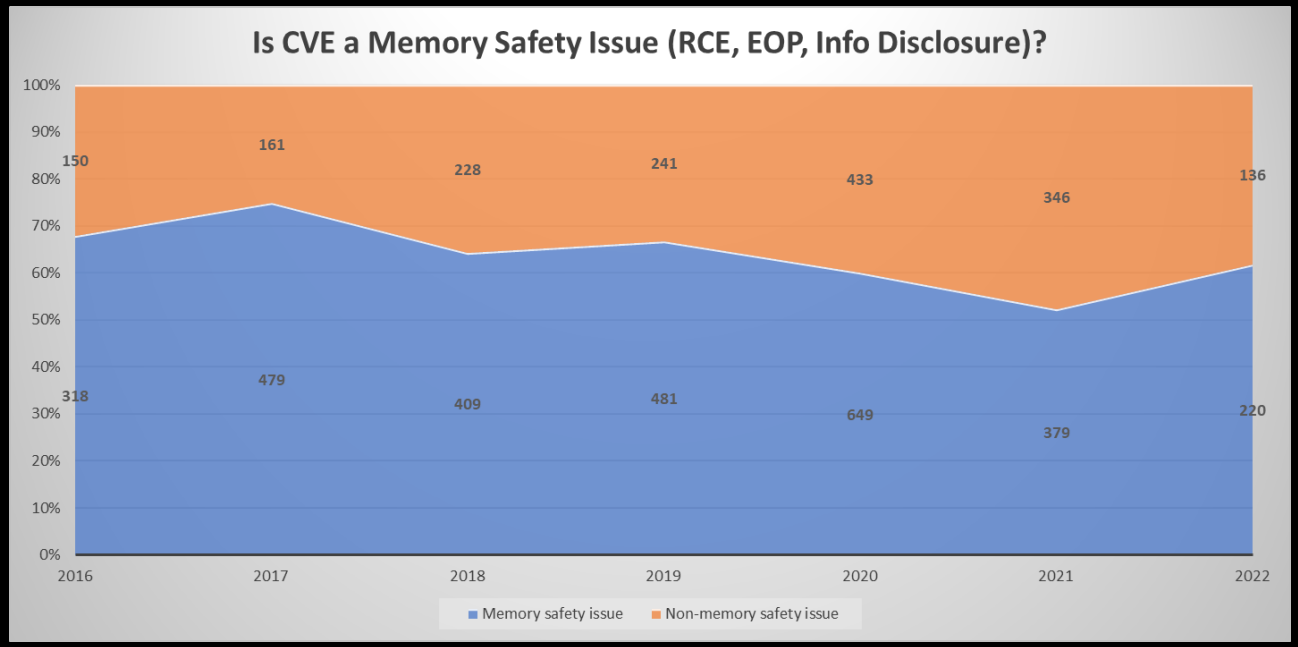
Microsoft and Memory Corruption

Shift-left from mitigation to elimination

Reduced investment in exploit mitigations (CFG/XFG)

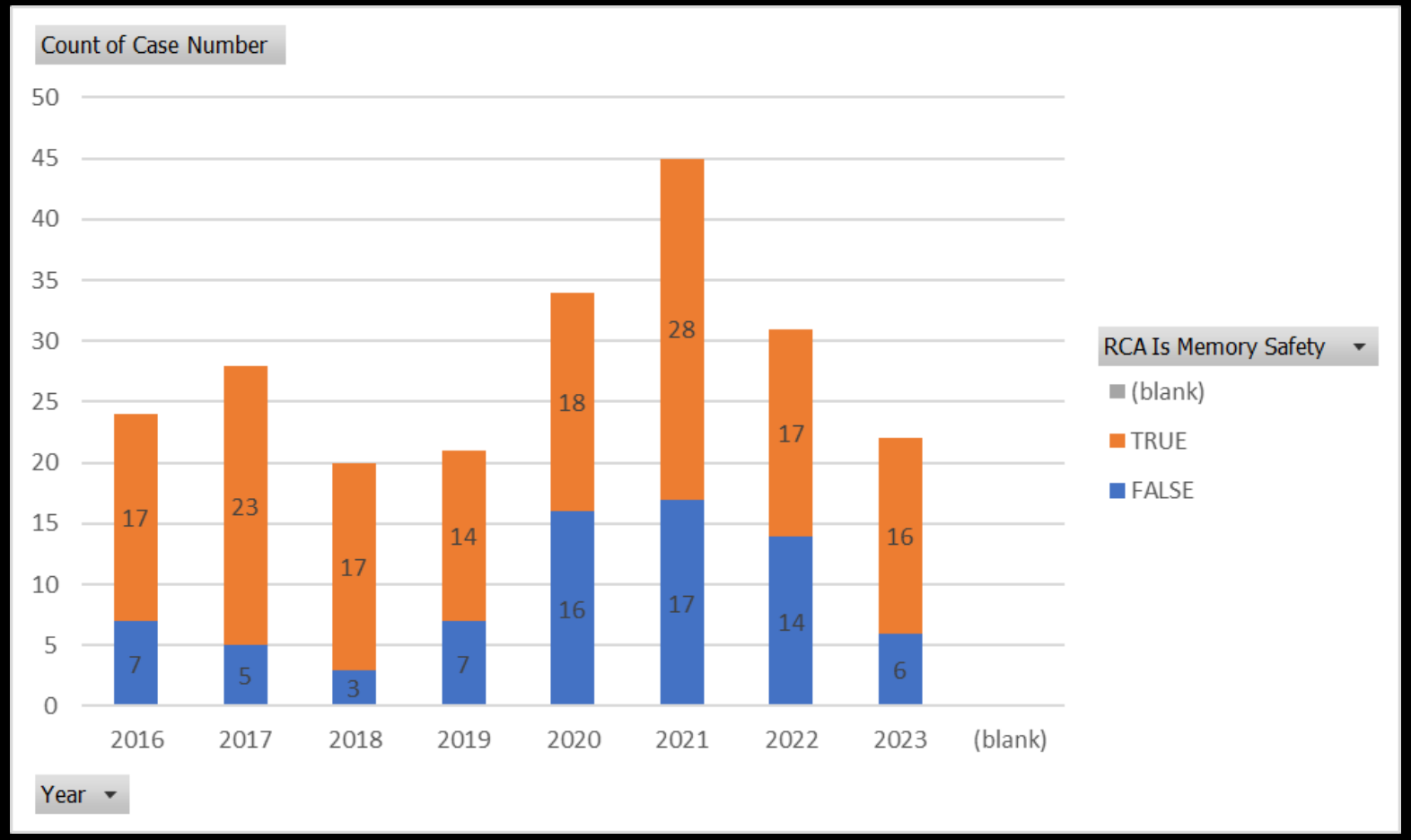
Increased investment in bug class elimination

Learning with systems level memory safe languages



Exploit mitigations have not eliminated 0days

2023 on-pace for record



Bugs vs. Killing Exploit Techniques

Mitigating exploit techniques has ambiguous long-term value.

Mitigations are typically far enough away from actual bug that bugs are still exploitable using different techniques.

Tradeoffs between performance, compatibility, and mitigation durability are becoming increasingly difficult.

Unclear how many more practical opportunities there are for building meaningful exploit mitigations.

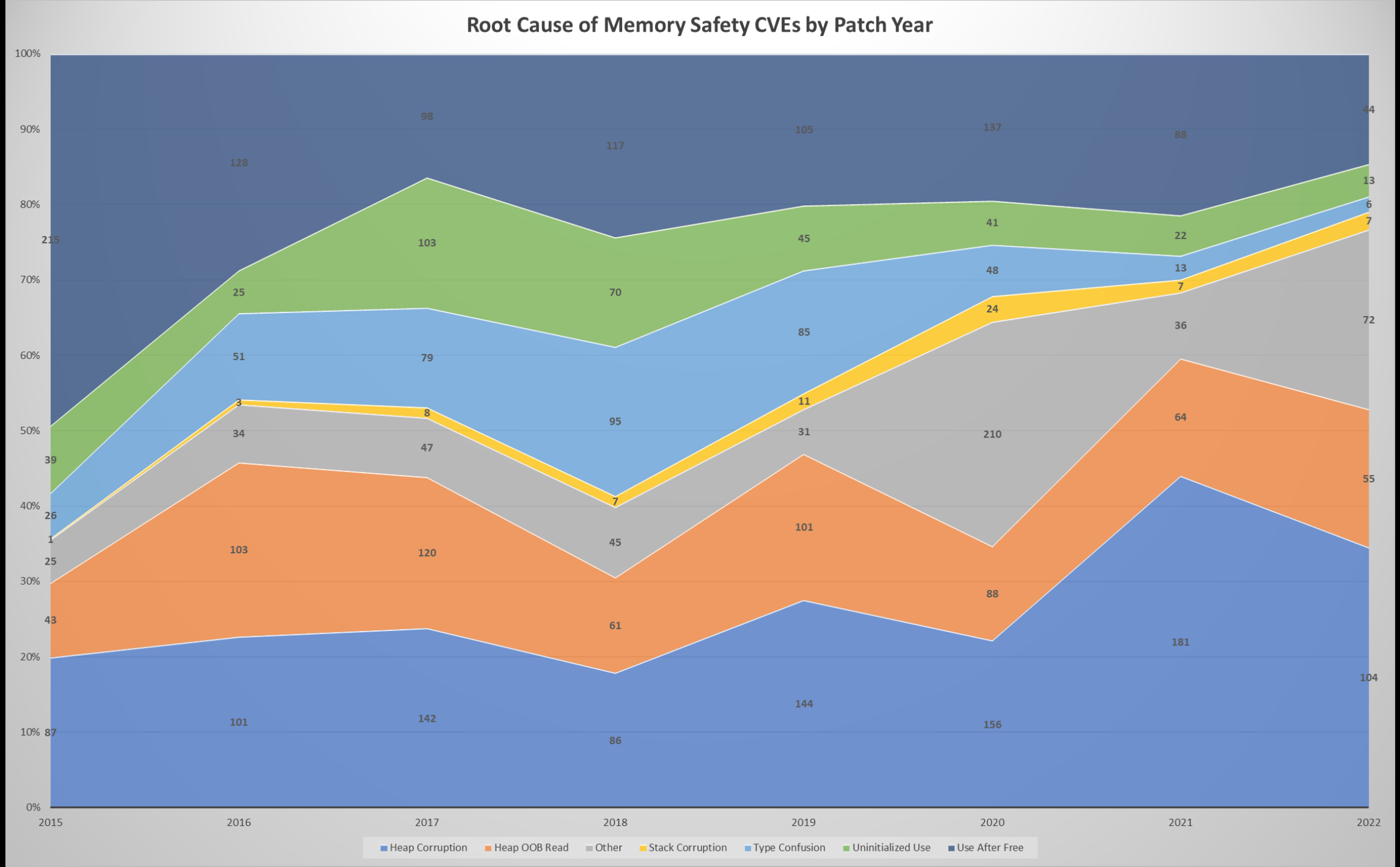
Killing Bugs vs. Killing Exploit Techniques

Microsoft increasingly focused on eliminating vulnerability classes, removing attack surface, and sandboxing code, and memory safe programming languages.

Hyper-V vPCI component refactored from a C kernel component to C++ (w/ GSL) user component.

Microsoft investigation of Rust and other safer systems languages, and use of managed languages.

CLFS blocked from sandboxes, Redirection Guard, etc.


Root Cause of Memory Safety CVEs by Patch Year
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
2015
2016
2017
2018
2019
2020
2021
2022
Heap Corruption
Heap OOB Read
Other
Stack Corruption
Type Confusion
Uninitialized Use
Use After Free

Microsoft Strategy

1. Memory Safe Languages
2. CPU Architectural Changes
3. Safer Language Subset

Memory Safe Languages

Our Engineering Advances

- Identity
- Strengthening identity protection and governance

- Secure by Design
- Transforming software development with automation and AI

- Secure by Default
- Embedding more security defaults into our products for out-of-the-box protection

- Response & Transparency
- Setting a new standard for faster incident response, security updates, and transparency

Transforming software development with automation and AI

What we will do:

- Making a step-change in our Security Development Lifecycle operations and making additional investments to meet the evolving needs of cloud and emerging technologies
- Completing our deployment of CodeQL integrated with GitHub Copilot learnings, by the end of 2024.
- Standardizing on RUST and other memory safe languages (MSLs) across engineering efforts
- Contribute $1M to support the work of the RUST Foundation, and to assist developers making the transition from C/C++ to Rust, we will invest $10M in Rust developer tooling.

Systems Language Overview

Most Difficult

Least Difficult

	Rust	C++	C
Object Lifetime	Statically Enforced	Not Enforced, unclear path forward.	No hope
Type Safety	Statically Enforced	Not enforced, unclear path forward.	No hope
Bounds Safety	Enforced at runtime when needed	Could be enforced for STL containers.	No hope
Uninitialized Safety	Statically Enforced	Not enforced, could be enforced w/ breaking change.	Stack could be enforced w/ breaking change.

Rust offers performance on-par with C/C++ but with full memory safety guarantees. It is seeing increased industry adoption.

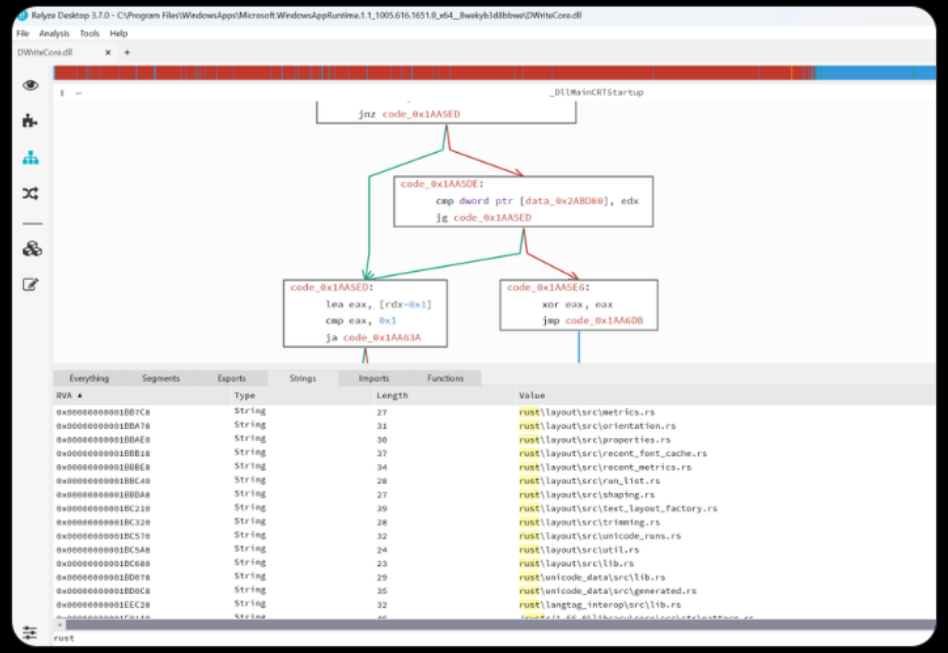
Microsoft needs to avoid C/C++ and use Rust or memory safe languages wherever possible.

Azure has selected Rust as the preferred Memory Safe Systems Programming Language.

1

David Weston (DWIZZZLE)
@dwizzzleMSFT

dwrite font parsing ported to Rust? 😲😲😲
learn.microsoft.com/en-us/windows/...



2:46 AM · Oct 8, 2022

2

David Weston (DWIZZZLE)
@dwizzzleMSFT

Windows is putting Rust in the kernel 🤯 learn more at my @BlueHatIL talk.

9:06 PM · Mar 16, 2023 · **106.6K** Views

Arden White, Christopher Leung, and many others are to thank for this work

Hardening C/C++ code

Forward for C/C++ Code

Four high-level bug classes responsible for majority of memory safety vulnerabilities.

- Buffer Overflow & Out-of-Bounds Accesses (i.e. attacker controls array index)
- Uninitialized Memory
- Type Confusion
- Use-After-Free

Hardening Existing C/C++ Code

Long Term: Combination of software and hardware mitigations to eliminate and detect the most common memory safety issues classes.

Short Term: Tactical efforts to eliminate attack surface, block exploit techniques, and statically analyze vulnerabilities, fuzz, etc.

- CLFS Signing, Heap Mitigations, ASLR, CFG, UMFD, etc.

Strategy and overall outlook is shaped by ~20 years of Microsoft trying to squash memory safety vulnerabilities.

Long-Term C/C++ Mitigation Investments

- InitAll / Pool Zeroing – Zero initialize stack variables and kernel pool allocations.
  - ToDo: User-mode heap strategy.
- CastGuard – Prevent illegal stack downcasts (type confusion).
  - ToDo: Research for additional types of type confusion.
- Memory Tagging – Broad impact to a variety of bug classes, hardware feature.
- These are cross-Microsoft and cross-Industry efforts.

Type Confusion

Come in many flavors..

- Illegal static downcast (down-casting to the wrong derived type in a class hierarchy).
- Improper union use.
- Illegal reinterpret_cast (i.e. cast an object of some type to totally different type).
- Generic logic issues (i.e. using fields incorrectly).

Offer extremely powerful primitives to attackers and can often lead to breaking mitigations such as Memory Tagging.

Many forms of type confusion are not possible to generically solve ☹.

Illegal Static Downcasts

```
struct Animal {
        virtual void WhoAmI() {cout << “Animal”;}
};

struct Dog : public Animal {
        virtual void WhoAmI() {cout << “Dog”;}
};

struct Cat : public Animal {
        virtual void WhoAmI() {cout << “Cat”;}
};

Animal* myAnimal = new Dog();
static_cast<Cat>(myAnimal)->WhoAmI();   // Illegal down-cast
```

Concept

To protect against illegal downcast, object needs a type identifier that can be checked.

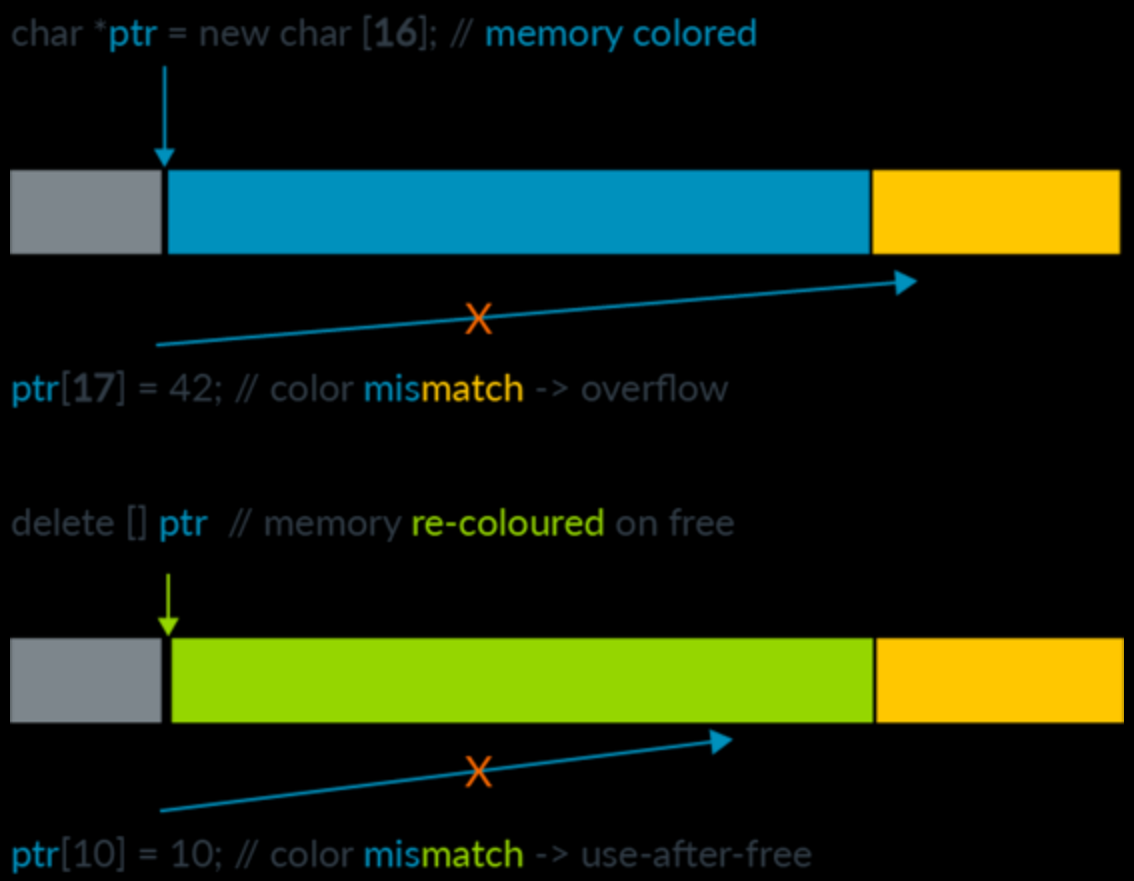
We cannot change object layout or we break the world.

Objects with a vftable already have an identifier, the vftable.

Automatically convert all static_cast on classes with vftables in to CastGuard protected casts.

Hardening C/C++ code

Memory Tagging



- Pointer stores a 4-bit “tag”
- Bitmap used to track the “tag” for the process address space
- “Tag” for an allocation gets set on allocation, cleared on free
- CPU checks pointer tag matches bitmap tag

Memory Tagging

- Helps developers catch bugs (like a hardware ASAN), stops bugs from being exploitable if they ship to customers.
- Non-trivial CPU and memory overhead, but low enough to enable-by-default in production.
- Google will deploy to Android soon, expect Apple to also deploy.
- Microsoft is actively working on Memory Tagging designs w/ silicon partners that are scalable from small client devices to the largest Azure servers.
- Goal: Enable by default for Windows to make a more reliable and secure OS. Support in Azure so Windows and Linux guests can take advantage.

Bug Classes vs. Mitigations

We can make progress on these (both detection and exploitability).

Vulnerability Class	Deterministic or Probabilistic Mitigation	% of Memory Safety Issue CVE's
Heap Linear Overflow	Deterministic	9.5%
Heap Non-Linear Overflow	Probabilistic	12%
Use-After-Free	Probabilistic	26%
Heap Linear Overread	Deterministic	5.3%*
Heap Non-Linear Overread	Probabilistic	14%*
Uninitialized Memory	Deterministic	12%
Type Confusion	Not Mitigated	14%

We can mostly solve these. Uninitialized doesn't require memory tagging.

CastGuard solves some type confusion, others have no apparent solution.

No investments for most stack safety bugs, but these are now a minority

CHERI

- Capability based architecture – creates a hardware representation of a pointer (not just interchangeable with a 64-bit integer)
- Pointers contains bounds information, double in size to 128-bits
- Can be combined with memory tagging
- Provides the most robust memory safety guarantees possible for C/C++ code
- Extremely difficult and expensive to deploy on Windows, will be used for other purposes (Pluton). See CHERIoT

Next steps with memory safety

Driven by community engagement

Broader Rust investment

$10 million spend, Azure Standard, shipping Dwrite, win32, hypervisor VMM,

Safer Native Code

Continue with compiler based mitigations

CPU architecture

CheriIOT, Memory Tagging, and other approaches being investigated for broad memory safety strategy

THANKS!

Huge thanks to Joe Bialek, Arlie Davis, Christopher Leung, Tony Chen, and all the other folks who's work is represented here