Microsoft

Pursuing Durably Safe Systems Software

Matt Miller (@epakskape)
Microsoft Security Response Center (MSRC)

SSTIC 2020
June 3rd, 2020

This presentation is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

This presentation represents my personal opinions

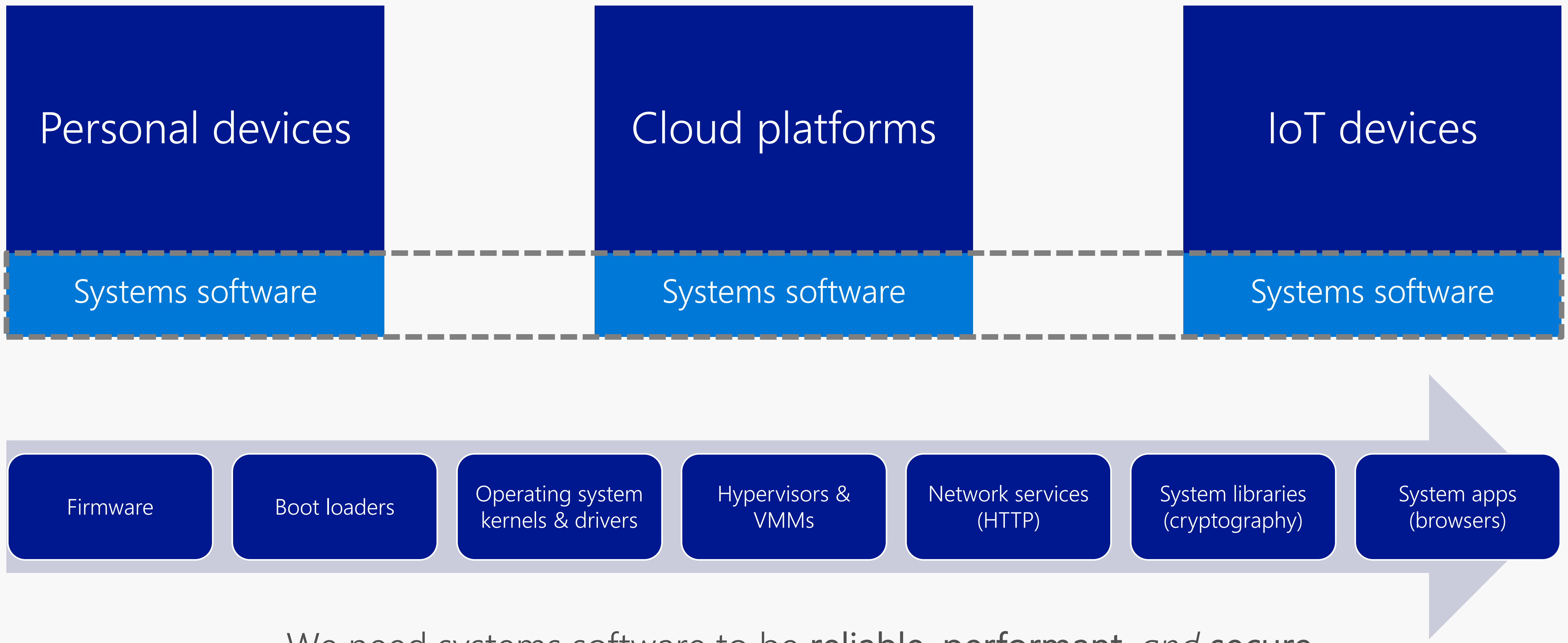
It is intended to provoke thoughtful discussion

It is not intended to broadly represent the views of Microsoft

☺

The malleable foundation of modern technology

Systems software provides the core platforms on which other software is built



We need systems software to be **reliable**, **performant**, *and* **secure**

What properties should safe systems software uphold?

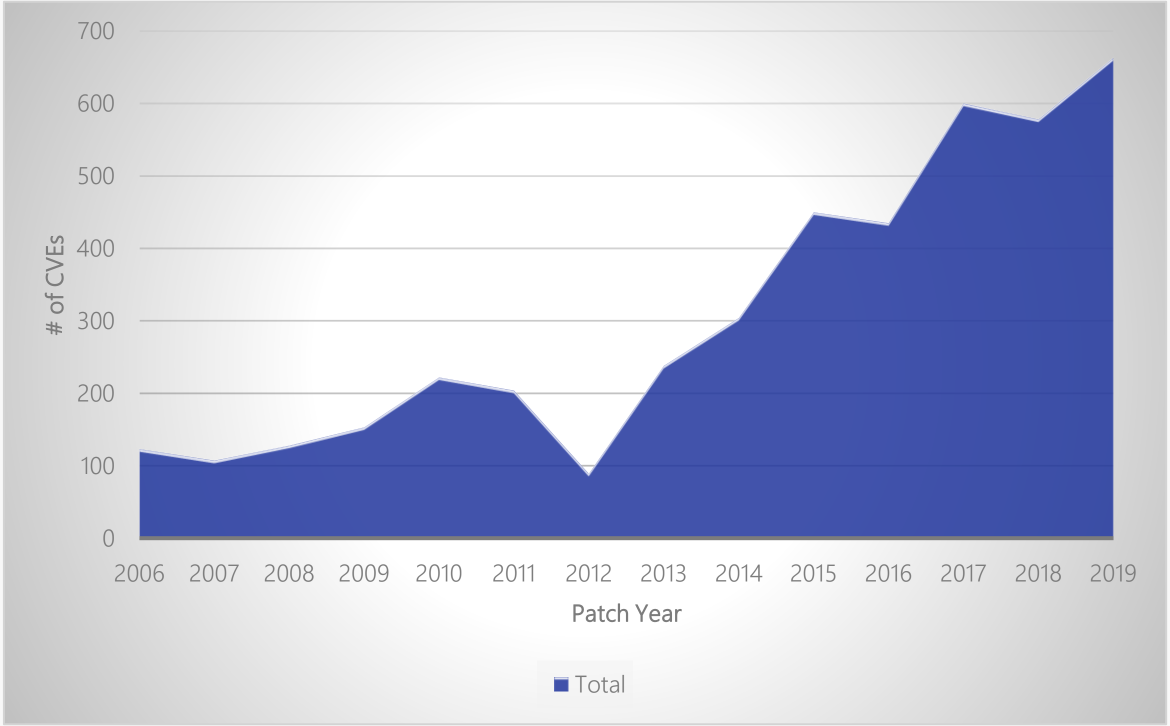
Hard to do the unsafe thing	Developers must be intentional about unsafety
Easy to verify that the safe thing happens	Safety is verifiable by developers and downstream consumers of the software
Productivity is maximized	Developers and downstream consumers of the software are maximally productive
Inherently viable	Performance, compatibility, and other tenets must be upheld

Systems software security

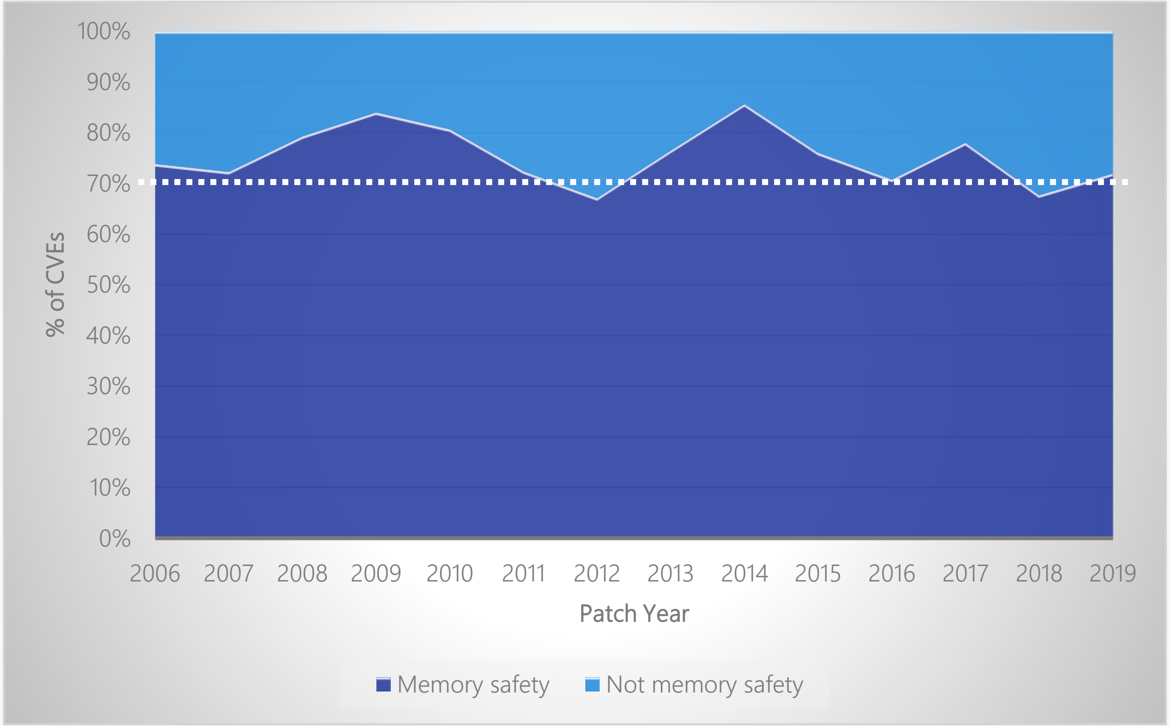
Today

It is too easy to do the unsafe thing today

For systems software[1] at Microsoft, memory safety is the most common vulnerability class



Vulnerabilities reported & fixed per year is increasing



~70% of vulnerabilities are memory safety year over year

Memory safety is an industry challenge
[2,3,4,5]

~66% of iOS 12 vulnerabilities

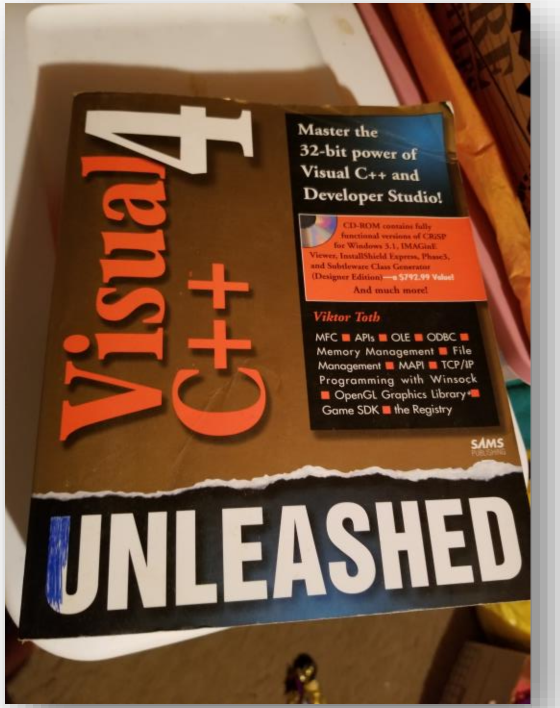
~72% of macOS 10.14 vulnerabilities

~60% of high severity vulnerabilities in Chrome

~90% of Android vulnerabilities

Why is it easy to do the unsafe thing today?

Most systems software is currently written in unsafe languages such as C and C++



These are great languages, but developers need to consciously do the safe thing

And it is easy to make a mistake ☹

```
case 'c':
{char feel[25];

cout << "Hello, how are you? " << endl;
cin >> feel;
```

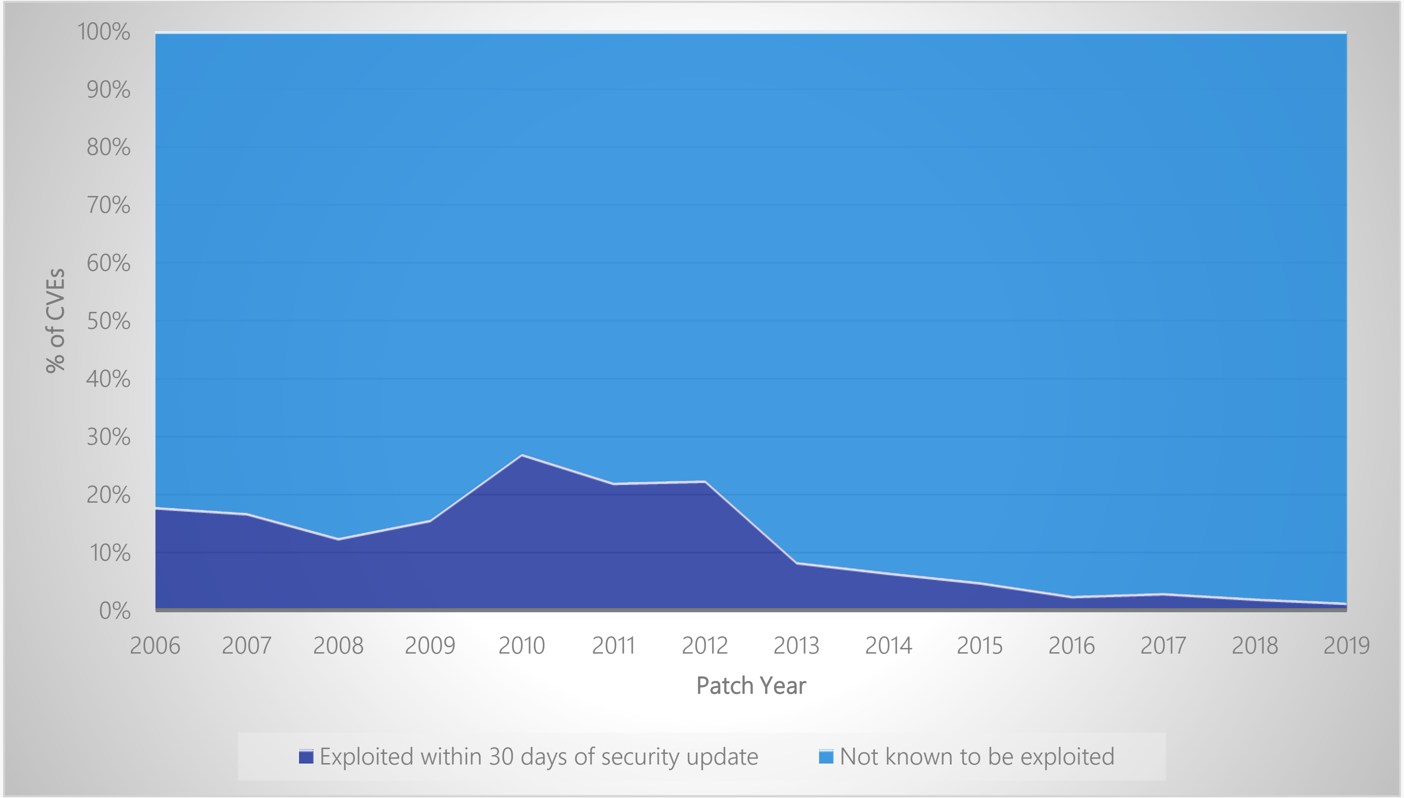
Some code I wrote when first learning C++ in 1995

Fast forward 2 decades & millions of lines of C/C++ ...

CVE-2019-1345

A portable executable (PE) parsing memory safety vulnerability[6] found by @j00ru that I introduced into the Windows kernel in 2016

Safety has improved, but vulnerabilities remain



For systems software[1] at Microsoft

Most vulnerabilities are not known to be exploited in the wild*

If a vulnerability is exploited, it is most likely to first be exploited as zero day in a targeted attack

Broad exploitation has become uncommon

Customer safety has meaningfully improved

What changed?

Exploiting vulnerabilities has become more expensive
→ Alphabet soup of exploit mitigations, sandboxes, and other controls have increased costs

Many attackers have pivoted to alternative tactics with better ROI
→ Social engineering (phishing for credential theft, ransomware, etc)

* Acknowledging that we have imperfect visibility

It's hard to verify that the safe thing happens

How do we know if systems software is free of various vulnerability classes?

We typically do not know today, so we leverage tools[25,26] to help us find vulnerabilities

Static analysis	Dynamic analysis	Fuzzing	Code review

Typically not automated and/or not broadly enabled by default

Typically no guarantee that an entire vulnerability class does not exist

Findings are valid for a specific point in time – new code may introduce issues

Downstream consumers cannot verify the completeness of these efforts

All of these tools have merit, but they do not satisfy the properties outlined earlier

And what about dependencies?

Software is increasingly dependent on a broad ecosystem of open source developers

No standardized way to know & enforce that dependencies implement specific security controls

Which compiler security features were enabled?

Which static analysis warnings have been eliminated?

Which vulnerability classes are guaranteed to not exist?

Did developers use MFA for commits?

What security controls were enabled in the CI/CD pipeline?

Productivity is not being maximized

Developers expend non-trivial energy debugging & fixing memory safety issues	Downstream consumers can experience vulnerability management costs
Reproducing the issue	Performing a risk analysis on vulnerable dependencies
Determining the root cause	Performing validation of security updates to mitigate regression risk
Developing and validating the fix	Ingesting security fixes from dependent components
Deploying updates with the fix	Deploying security updates to affected systems in a timely fashion
×	×
# of vulnerabilities	# of vulnerabilities

The upstream & downstream costs to productivity can be significant

Systems software security

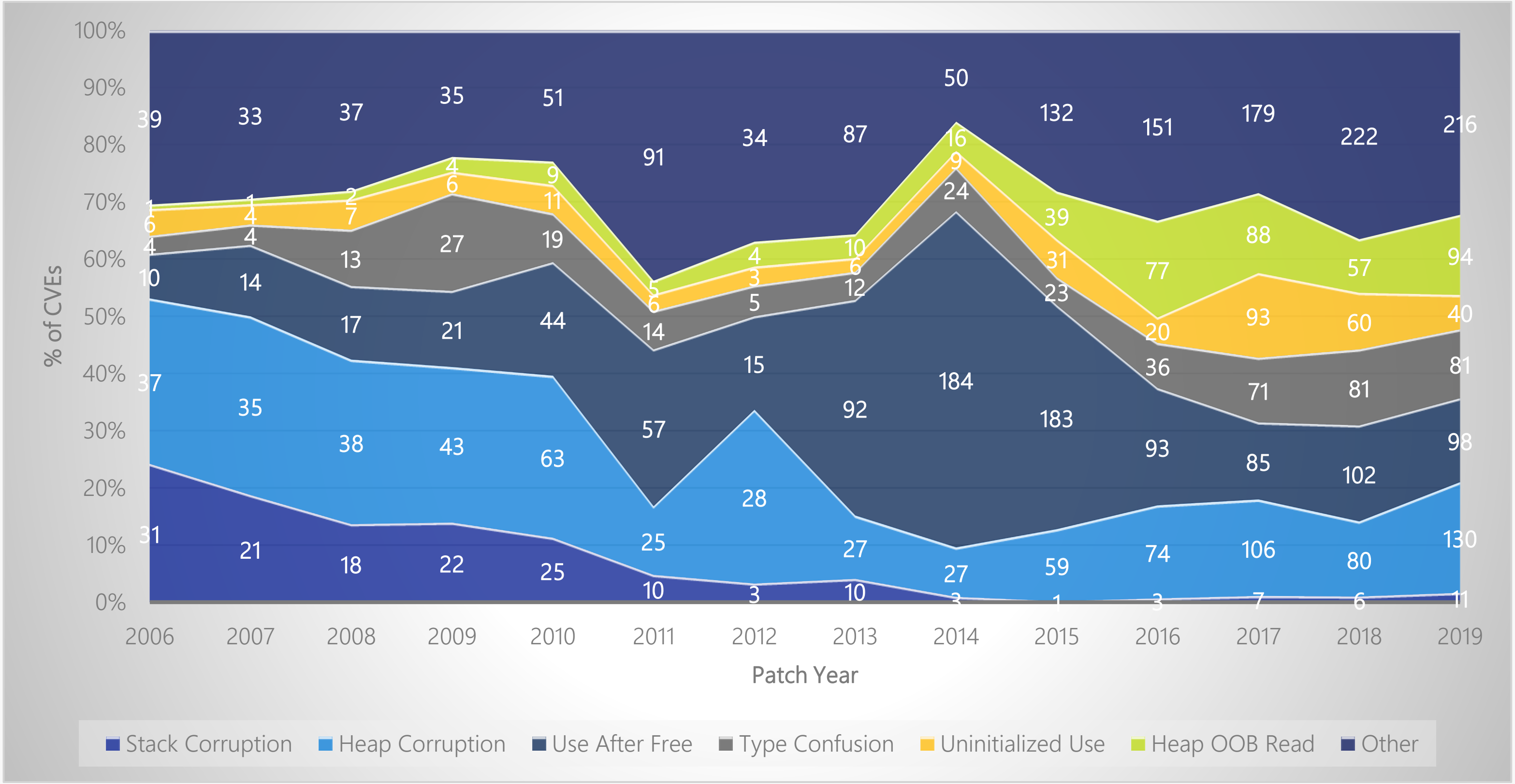
Pursuing durable safety

What options can we consider?

Make unsafe code safer	Durably & verifiably eliminate common classes of vulnerabilities in unsafe code
Transition to safer languages	Adopt safer languages such as C# and Rust where it matters
Safer hardware extensions	Pursue hardware security features that help eliminate vulnerability classes

Making unsafe code safer

Finding ways to make C and C++ code durably & verifiably safer is attractive but challenging



Top vulnerability classes in systems software[1] at Microsoft (2016 through 2019)

#1 – heap out-of-bounds

#2 – use after free

#3 – type confusion

#4 – uninitialized use

How can we approach eliminating the most common classes of vulnerabilities?

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

Eliminating common C/C++ vulnerability classes [1/3]

Vulnerability category	Vulnerability class	Durable safety solution	Completeness?	Enforceability?	Verifiability?	Developer friction?
Spatial safety	Heap out-of-bounds read/write	Use gsl::span<T> and do not index raw pointers or perform pointer arithmetic on raw pointers[7]	🙂	😐	☹️	😐
	Stack out-of-bounds read/write					
	Global out-of-bounds read/write					
Temporal safety	Heap uninitialized use	Always initialize members in constructors[9]	☹️	😐	☹️	🙂
		Use a memory allocator that initializes by default	🙂	☹️	☹️	🙂
	Stack uninitialized use	Always initialize members in constructors[9]	☹️	😐	☹️	🙂
		Always initialize local variables before use[8,18]	🙂	☹️	☹️	🙂

Eliminating common C/C++ vulnerability classes [2/3]

Vulnerability category	Vulnerability class	Durable safety solution	Completeness?	Enforceability?	Verifiability?	Developer friction?
Temporal safety	Heap use after free	Use RAII, owner<T>, unique_ptr<T>, and shared_ptr<T> instead of raw pointers or references to objects[10, 11, 12]	😐	😐	☹	☹
	Stack use after free					
Concurrency safety	Memory access race condition	Unknown[13]	☹	☹	☹	☹

Object lifetime and concurrency vulnerabilities are challenging to categorically eliminate

Eliminating common C/C++ vulnerability classes [3/3]

2nd order vulnerability category	Vulnerability class	Durable safety solution	Completeness?	Enforceability?	Verifiability?	Developer friction?
Type confusion	Illegal static down cast	Use dynamic cast or similar runtime verification[14,17]	🙂	😐	☹️	☹️
	Union field type confusion	Use std::variant[15]	🙂	☹️	☹️	😐
Arithmetic errors	Integer overflow or underflow	Use safe integer manipulation libraries[16]	🙂	☹️	☹️	☹️

2nd order vulnerability classes can give rise to memory safety vulnerabilities

Observations: making unsafe code safer

Making C and C++ code durably safer is possible

No clear line-of-sight to solutions for all common classes

Some solutions have build-time rules, but most are off by default

Some solutions can have non-trivial developer friction

Enforcing and verifying that solutions are in place is not easy today

Transitioning to safer languages

Safer languages can categorically eliminate most of the common vulnerability classes

C#	Rust
Type-safe	Type-safe
Memory-safe (with GC)	Memory-safe (without GC)
Interoperable with C/C++	Interoperable with C/C++

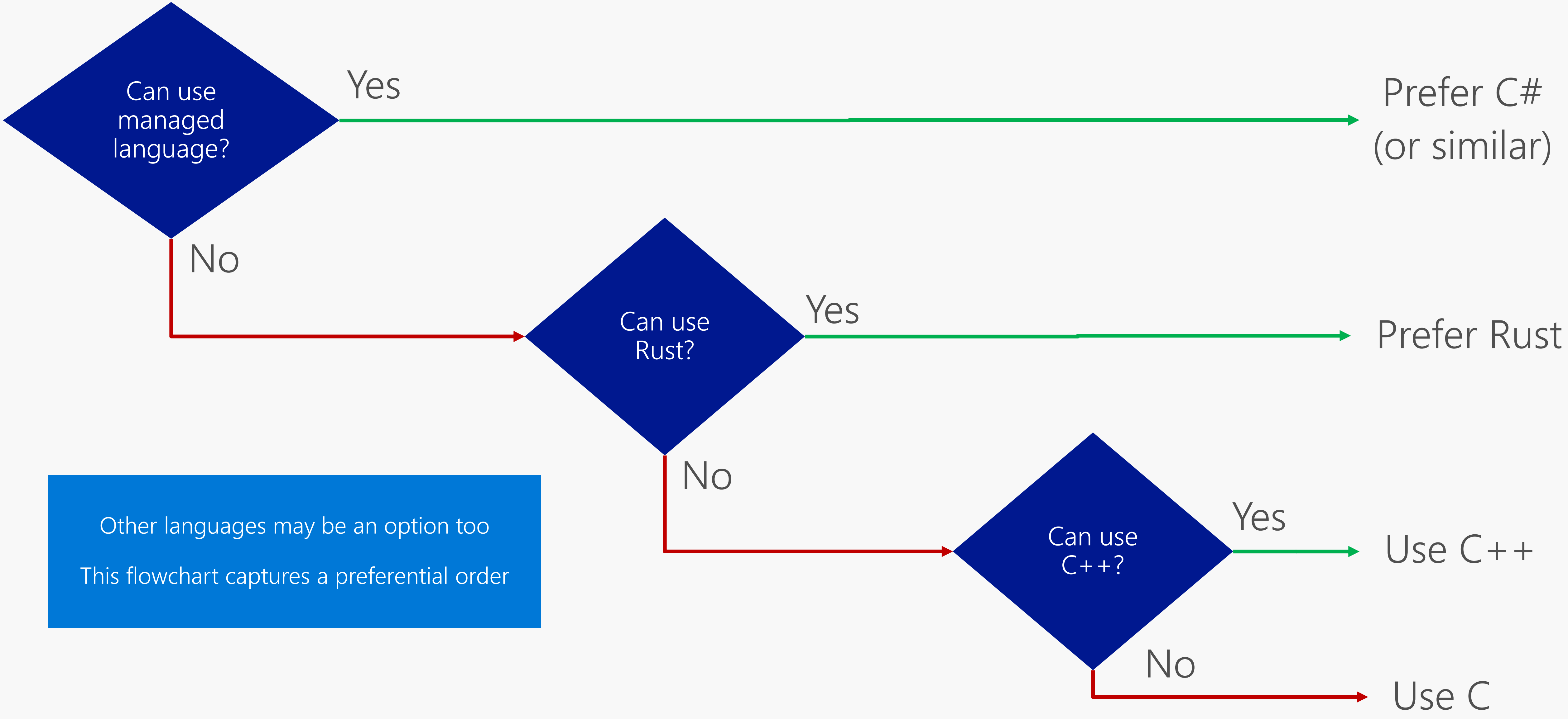
C# is a wonderful language, but it is not suitable in many systems contexts

C# and Rust eliminate many vulnerability classes

Vulnerability category	Vulnerability class	C# Completeness	Rust Completeness
Spatial safety	Heap out-of-bounds read/write	🙂	🙂
	Stack out-of-bounds read/write	🙂	🙂
	Global out-of-bounds read/write	🙂	🙂
Temporal safety	Heap uninitialized use	🙂	🙂
	Stack uninitialized use	🙂	🙂
	Heap use after free	🙂	🙂
	Stack use after free	🙂	🙂
Concurrency safety	Memory access race condition	😐	🙂
Type confusion	Illegal static down cast	🙂	🙂
	Union field type confusion	🙂	🙂
Arithmetic errors	Integer overflow or underflow	🙂	😐

Both C# and Rust have an unsafe keyword, but developers must intentionally use it

Proposed systems software language selection flow chart



Observations: transition to safer languages

Safer languages can provide strong durable safety

Reduced cognitive load enables developers to be more productive

Rewriting existing code in a safer language can be high friction

Interop and compatibility with existing code and tools is important

Opportunities exist to innovate in safer languages (Verona) [24]

Safer hardware extensions

Systems software can leverage CPU architecture extensions to eliminate classes of vulnerabilities

Memory Tagging

CHERI

These features can durably eliminate some vulnerability classes & may also make exploitation more difficult

Memory tagging

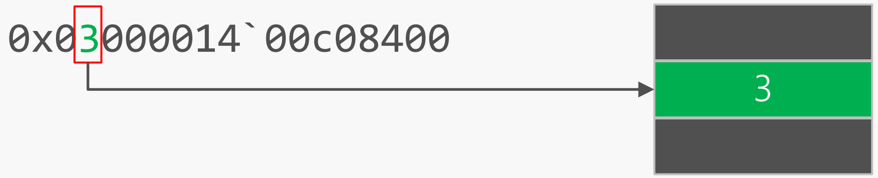
Armv8.5 Memory Tagging Extension (MTE) basics[19]

16-byte aligned memory regions have a 4-bit "memory" tag

Pointers have a 4-bit "address" tag in reserved virtual address bits

When pointers are accessed, the tags are compared

If they don't match, an exception is raised



MTE's impact on various vulnerability classes[20]

Deterministic protection for

- Adjacent out-of-bounds access

Probabilistic protection for

- Non-adjacent out-of-bounds access
- Use after free

Situational protection for

- Uninitialized use

Completeness?	Enforceability?	Verifiability?	Developer friction?
😐	🙂	☹️	🙂

Observations: memory tagging

Can help discover many different types of vulnerabilities at scale

Durably eliminates a common vulnerability class (adjacent out-of-bounds)

Probabilistically mitigates many vulnerability classes, but with low entropy

Probabilistic protection may not be effective in all cases due to side channels

Durability of probabilistic protection is an open question

CHERI

Capability Hardware Enhanced RISC Instructions[21,28]

Unforgeable capabilities enable fine-grained memory access control[22]

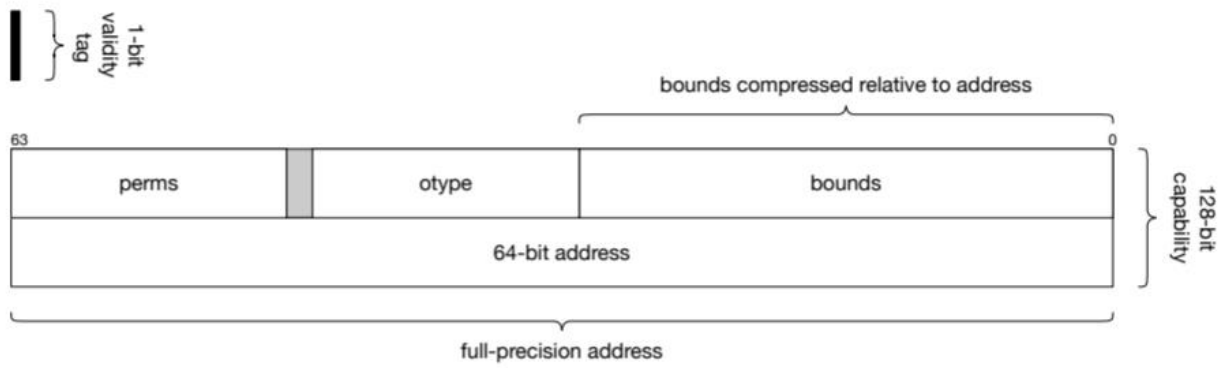


Figure 2.1: 128-bit CHERI Concentrate capability representation: 64-bit address and metadata in addressable memory and 1-bit tag out of band.

CHERI's impact on various classes of vulnerabilities

Deterministic protection for

- Out-of-bounds memory access

Non-deterministic protection for

- Temporal safety (work in progress[23])

Completeness?	Enforceability?	Verifiability?	Developer friction?
😐	🙂	☹️	😐

Observations: CHERI

Durably eliminates most spatial safety vulnerabilities

The inability to forge capabilities may make exploitation more difficult

Most existing C and C++ code is expected to be compatible with CHERI

Non-trivial cost to support the OS platform for the CHERI architecture

Which paths should we pursue?

Why not all of them? ☺

Prefer safer languages such as C# and Rust where possible	Adopt safer C++ practices and security features where possible	Explore ISA extensions that could help address safety gaps in C and C++

Enforce and verify the correct use of these security controls

These approaches can help us work toward achieving the properties we outlined earlier

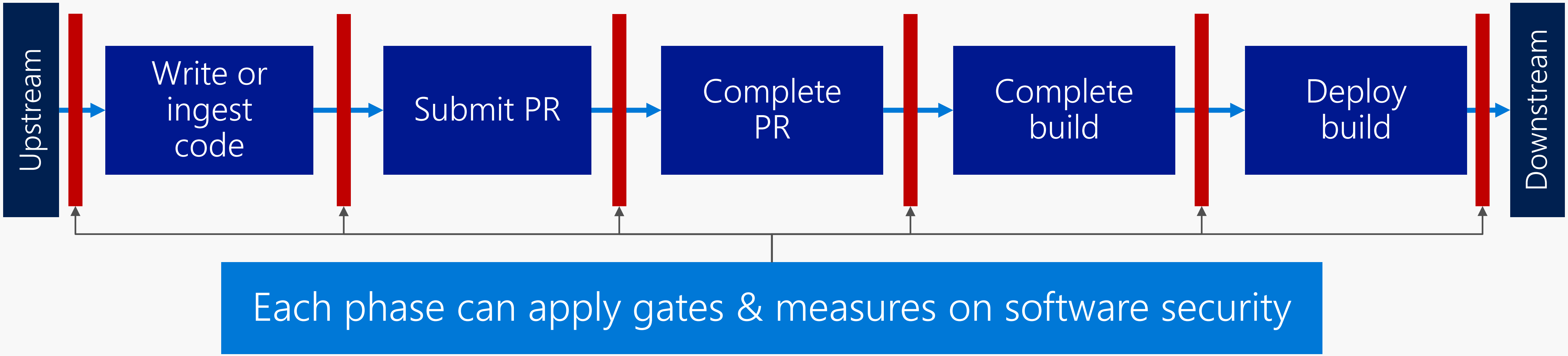
- ✔ Hard to do the unsafe thing
- ✔ Easy to verify that the safe thing happens
- ✔ Productivity is maximized
- ✔ Inherently viable

But what about the broad software supply chain?

How might we pervasively achieve our desired properties?

Enforcing safety for the broader software supply chain

Approaches like Software Bill of Materials (SBOM) could help enforce *transitive* software security controls[27]



Measures	Gates
Compile-time security features enabled in the code	Prohibit the use of unsafe code without approval
Safety-relevant compile-time warnings present in the code	Require that vulnerability class X, Y, Z not exist
Authentication method used for commits (e.g. MFA)	Require the use of MFA for commits
Attested health state of devices (dev, build, deployment)	Require devices used in supply chain be healthy
Versions of dependencies consumed	No known vulnerable dependencies are allowed
…	…

Wrapping up

Systems software forms the foundation of modern technology

Durable safety for systems software is imperative for society

How much progress can we make toward this in the next 5-10 years?

I’m looking forward to seeing what we can achieve together ☺

A huge **THANK YOU** to everyone at Microsoft & across the industry who is working to durably improve systems software security

© 2020 Microsoft. All rights reserved. Microsoft, Windows and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries. The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.

References

[1] Vulnerabilities in Microsoft Windows, Office, Internet Explorer and Edge with a security impact of Remote Code Execution (RCE), Elevation of Privilege (EOP), or Information Disclosure (ID)

[2] Introduction to Memory Unsafety for VPs of Engineering. https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/

[3] Memory Unsafety in Apple's Operating Systems. https://langui.sh/2019/07/23/apple-memory-safety/

[4] GWP-ASan: Sampling heap memory error detection in-the-wild. https://security.googleblog.com/2019/11/gwp-asan-sampling-heap-memory-error.html

[5] Queue the Hardening Enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html

[6] Issue 1909: Windows Kernel out-of-bounds read in nt!MiParseImageLoadConfig while parsing malformed PE file. https://bugs.chromium.org/p/project-zero/issues/detail?id=1909

[7] C++ Core Guidelines Bounds safety profile. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#probounds-bounds-safety-profile

[8] C++ Core Guideline ES.20: always initialize an object. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Res-always

[9] C++ Core Guidelines type.6: always initialize a member variable. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#SS-type

[10] C++ Core Guidelines Pro.lifetime: lifetime safety profile. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#SS-lifetime

[11] C++ Core Guidelines R: resource management. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#r-resource-management

[12] ES.65: Don't dereference an invalid pointer. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Res-deref

[13] C++ Core Guidelines CP: concurrency and parallelism. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#cp-concurrency-and-parallelism

[14] C++ Core Guidelines C.146: Use dynamic_cast where class hierarchy navigation is unavoidable. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rh-dynamic_cast

[15] C++ Core Guidelines C.181: Avoid "naked" union s. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Ru-naked

[16] C++ Core Guidelines ES.100 - ES.107: expressions and statements related to arithmetic rules. https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#S-expr

[17] Control Flow Integrity Design Documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html

[18] Killing Uninitialized Memory: Protecting the OS Without Destroying Performance. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_09_CppCon/CppCon2019%20-%20Killing%20Uninitialized%20Memory.pdf

[19] Armv8.5-A Memory Tagging Extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

References

[20] Security analysis of memory tagging. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf

[21] Capability Hardware Enhanced RISC Instructions (CHERI). https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

[22] An Introduction to CHERI. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[23] Addressing Temporal Memory Safety. https://www.cst.cam.ac.uk/blog/tmj32/addressing-temporal-memory-safety

[24] Verona. https://github.com/microsoft/verona

[25] Mitigating vulnerabilities in endpoint network stacks. https://www.microsoft.com/security/blog/2020/05/04/mitigating-vulnerabilities-endpoint-network-stacks/

[26] Keeping Windows secure. https://www.youtube.com/watch?v=NlfZG2wTPZU

[27] Collaborating to Improve Open Source Security: How the Ecosystem Is Stepping Up. https://www.rsaconference.com/usa/agenda/collaborating-to-improve-open-source-security-how-the-ecosystem-is-stepping-up

[28] Digital Security by Design: Capability Hardware Enhanced RISC Instructions Architecture and Software Model. https://vimeo.com/376177222