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
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 😞
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

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

What changed?

Exploited within 30 days of security update
Not known to be exploited
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
- Typically not automated and/or not broadly enabled by default

Dynamic analysis
- Typically no guarantee that an entire vulnerability class does not exist

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

Code review
- 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**

- Reproducing the issue
- Determining the root cause
- Developing and validating the fix
- Deploying updates with the fix

**Downstream consumers can experience vulnerability management costs**

- Performing a risk analysis on vulnerable dependencies
- Performing validation of security updates to mitigate regression risk
- Ingesting security fixes from dependent components
- Deploying security updates to affected systems in a timely fashion

![Crossmark](https://via.placeholder.com/15)

# 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]

<table>
<thead>
<tr>
<th>Vulnerability category</th>
<th>Vulnerability class</th>
<th>Durable safety solution</th>
<th>Completeness?</th>
<th>Enforceability?</th>
<th>Verifiability?</th>
<th>Developer friction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial safety</td>
<td>Heap out-of-bounds read/write</td>
<td>Use gsl::span&lt;T&gt; and do not index raw pointers or perform pointer arithmetic on raw pointers[7]</td>
<td>☑️</td>
<td>☐️</td>
<td>☐️</td>
<td>☐️</td>
</tr>
<tr>
<td></td>
<td>Stack out-of-bounds read/write</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Global out-of-bounds read/write</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temporal safety</td>
<td>Heap uninitialized use</td>
<td>Always initialize members in constructors[9]</td>
<td>☐️</td>
<td>☐️</td>
<td>☐️</td>
<td>☑️</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use a memory allocator that initializes by default</td>
<td>☑️</td>
<td>☐️</td>
<td>☐️</td>
<td>☑️</td>
</tr>
<tr>
<td></td>
<td>Stack uninitialized use</td>
<td>Always initialize members in constructors[9]</td>
<td>☐️</td>
<td>☐️</td>
<td>☐️</td>
<td>☑️</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Always initialize local variables before use[8,18]</td>
<td>☑️</td>
<td>☐️</td>
<td>☐️</td>
<td>☑️</td>
</tr>
</tbody>
</table>
## Eliminating common C/C++ vulnerability classes [2/3]

<table>
<thead>
<tr>
<th>Vulnerability category</th>
<th>Vulnerability class</th>
<th>Durable safety solution</th>
<th>Completeness?</th>
<th>Enforceability?</th>
<th>Verifiability?</th>
<th>Developer friction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporal safety</td>
<td>Heap use after free</td>
<td>Use RAII, owner&lt;T&gt;, unique_ptr&lt;T&gt;, and shared_ptr&lt;T&gt; instead of raw pointers or references to objects[10, 11, 12]</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td></td>
<td>Stack use after free</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concurrency safety</td>
<td>Memory access race condition</td>
<td>Unknown[13]</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
<td>😞</td>
</tr>
</tbody>
</table>

Object lifetime and concurrency vulnerabilities are challenging to categorically eliminate.
Eliminating common C/C++ vulnerability classes [3/3]

<table>
<thead>
<tr>
<th>2nd order vulnerability category</th>
<th>Vulnerability class</th>
<th>Durable safety solution</th>
<th>Completeness?</th>
<th>Enforceability?</th>
<th>Verifiability?</th>
<th>Developer friction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type confusion</td>
<td>Illegal static down cast</td>
<td>Use dynamic cast or similar runtime verification[14,17]</td>
<td>🌝</td>
<td>🛑</td>
<td>🛑</td>
<td>🛑</td>
</tr>
<tr>
<td></td>
<td>Union field type confusion</td>
<td>Use std::variant[15]</td>
<td>🌝</td>
<td>🛑</td>
<td>🛑</td>
<td>🛑</td>
</tr>
<tr>
<td>Arithmetic errors</td>
<td>Integer overflow or underflow</td>
<td>Use safe integer manipulation libraries[16]</td>
<td>🌝</td>
<td>🛑</td>
<td>🛑</td>
<td>🛑</td>
</tr>
</tbody>
</table>

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

Transitioning to safer languages

Safer languages can categorically eliminate most of the common vulnerability classes.

<table>
<thead>
<tr>
<th>C#</th>
<th>Rust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-safe</td>
<td>Type-safe</td>
</tr>
<tr>
<td>Memory-safe (with GC)</td>
<td>Memory-safe (without GC)</td>
</tr>
<tr>
<td>Interoperable with C/C++</td>
<td>Interoperable with C/C++</td>
</tr>
</tbody>
</table>

C# is a wonderful language, but it is not suitable in many systems contexts.
C# and Rust eliminate many vulnerability classes

<table>
<thead>
<tr>
<th>Vulnerability category</th>
<th>Vulnerability class</th>
<th>C# Completeness</th>
<th>Rust Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial safety</td>
<td>Heap out-of-bounds read/write</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>Stack out-of-bounds read/write</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>Global out-of-bounds read/write</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Temporal safety</td>
<td>Heap uninitialized use</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>Stack uninitialized use</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>Heap use after free</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>Stack use after free</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Concurrency safety</td>
<td>Memory access race condition</td>
<td>☹</td>
<td>☺</td>
</tr>
<tr>
<td>Type confusion</td>
<td>Illegal static down cast</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td></td>
<td>Union field type confusion</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Arithmetic errors</td>
<td>Integer overflow or underflow</td>
<td>☺</td>
<td>☹</td>
</tr>
</tbody>
</table>

Both C# and Rust have an `unsafe` keyword, but developers must intentionally use it
Proposed systems software language selection flow chart

- Can use managed language?
  - Yes: Prefer C# (or similar)
  - No: Can use Rust?
    - Yes: Prefer Rust
    - No: Can use C++?
      - Yes: Use C++
      - No: Use C

Other languages may be an option too
This flowchart captures a preferential order
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

---

<table>
<thead>
<tr>
<th>Completeness?</th>
<th>Enforceability?</th>
<th>Verifiability?</th>
<th>Developer friction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>☹️</td>
<td>☀️</td>
<td>☹️</td>
<td>☀️</td>
</tr>
</tbody>
</table>
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
Unforgeable capabilities enable fine-grained memory access control[22]

CHERI
Capability Hardware Enhanced RISC Instructions[21,28]

Deterministic protection for
• Out-of-bounds memory access

Non-deterministic protection for
• Temporal safety (work in progress[23])

<table>
<thead>
<tr>
<th>Completeness?</th>
<th>Enforceability?</th>
<th>Verifiability?</th>
<th>Developer friction?</th>
</tr>
</thead>
<tbody>
<tr>
<td>☹</td>
<td>☻</td>
<td>☹</td>
<td>☹</td>
</tr>
</tbody>
</table>
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].

Each phase can apply gates & measures on software security

### Measures
- Compile-time security features enabled in the code
- Safety-relevant compile-time warnings present in the code
- Authentication method used for commits (e.g. MFA)
- Attested health state of devices (dev, build, deployment)
- Versions of dependencies consumed

### Gates
- Prohibit the use of unsafe code without approval
- Require that vulnerability class X, Y, Z not exist
- Require the use of MFA for commits
- Require devices used in supply chain be healthy
- 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
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)


References


