The Quest to Memory Safety

Programming Languages

Sebastian Fernandez
@snfernandez

Microsoft Security Response Center
Vulnerabilities & Mitigations

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

Who am I?

- Vulnerabilities and Mitigations team in MSRC
- Analyze and promote the use of safer languages
- Bug hunts and sometimes writes exploits

Defining the problem

Software vulnerabilities incur a high cost for everyone

For vendors

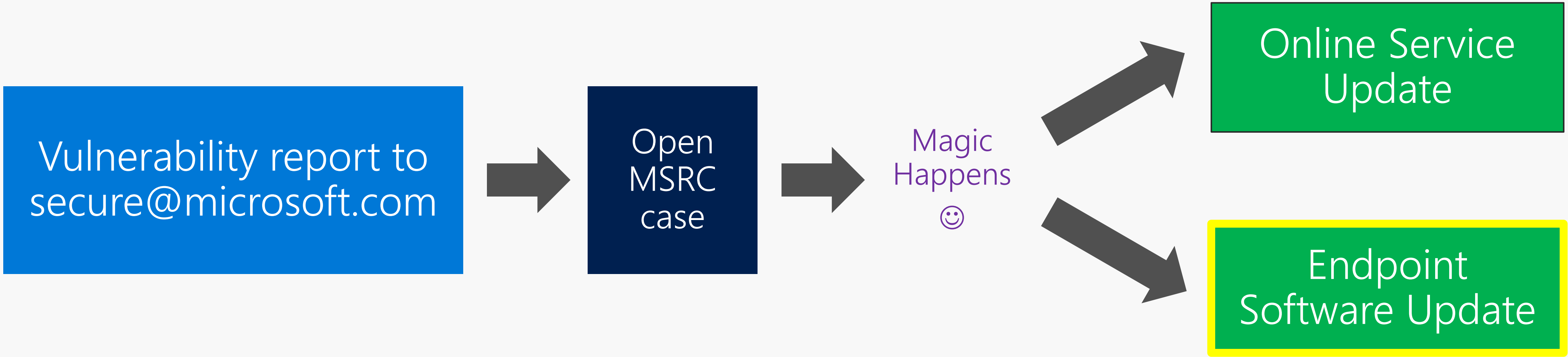
- Enforcing secure development practices (SDL)
- Fixing, verifying and shipping security patches
- Bad PR

For customers

- Vulnerable software leaves computers open to attacks
- Extra layers of security
- Disruptive security updates

Defining our scope

Vulnerabilities reported to Microsoft are typically addressed in one of two ways


Vulnerability report to secure@microsoft.com

Open MSRC case

Magic Happens ☺

Online Service Update

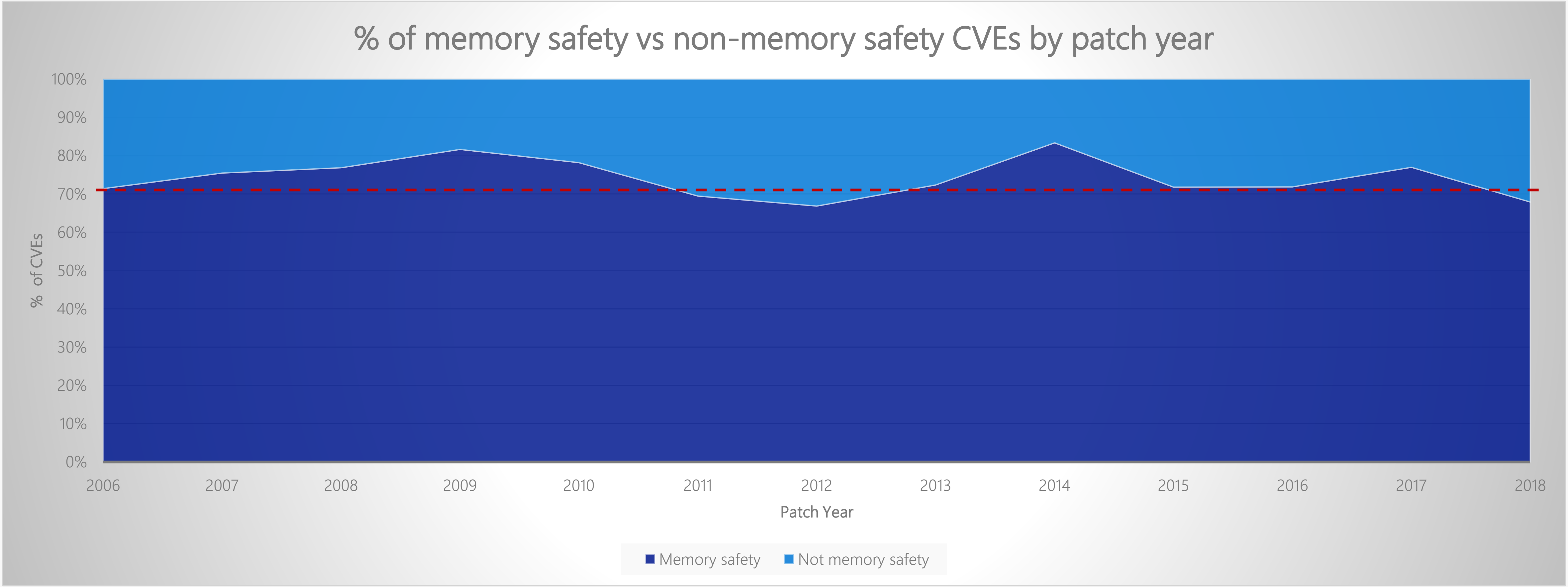
Endpoint Software Update

In 2018, ~54% of reported vulnerabilities were addressed via a software update

~85% of those vulnerabilities were Remote Code Execution (RCE), Elevation of Privilege (EOP), or Information Disclosure (ID)

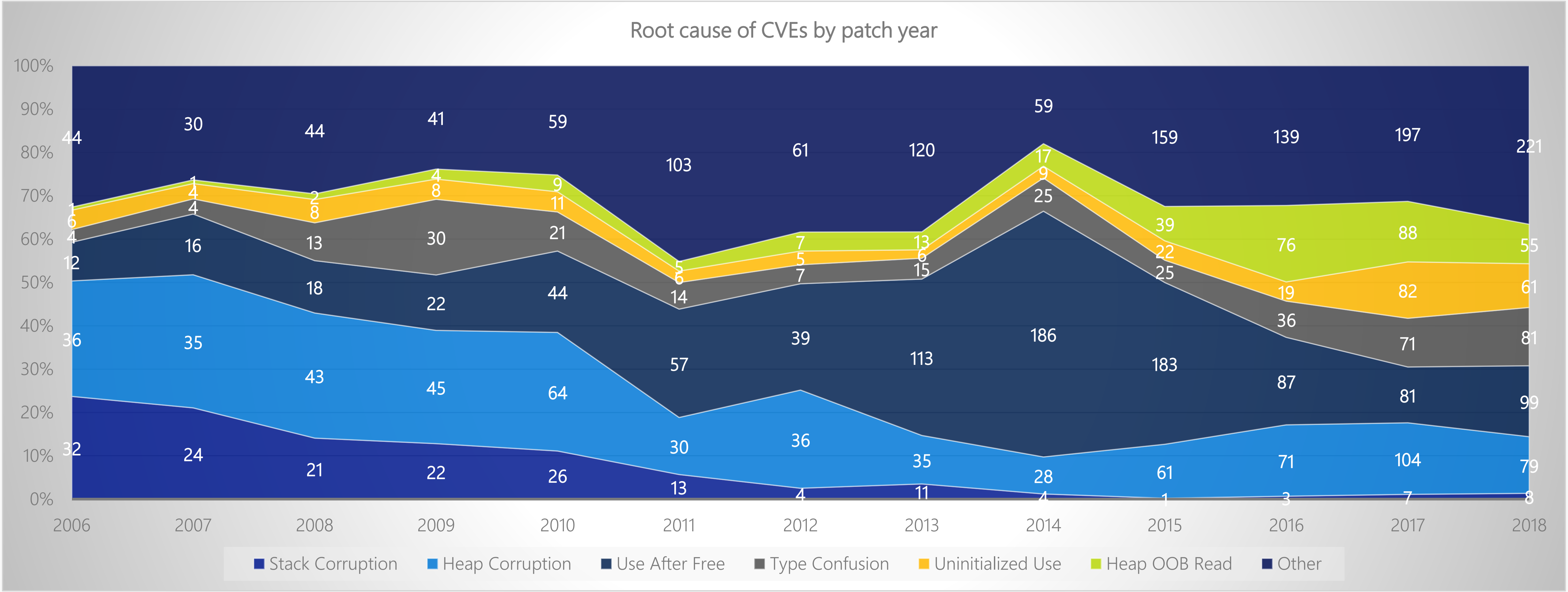
Memory safety issues remain dominant

We closely study the root cause trends of vulnerabilities & search for patterns



~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues

Drilling down into root causes



Root cause of CVEs by patch year

Top root causes since 2016:

#1: heap out-of-bounds

#2: use after free

#3: type confusion

#4: uninitialized use

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

And we are not alone

"Most of Android's vulnerabilities occur in the media and Bluetooth components. ***Use-after-free (UAF), integer overflows, and out of bounds (OOB) reads/writes comprise 90% of vulnerabilities*** *with OOB being the most common." – Jeff Vander Stoep about bugs in* Android during 2018

"Across the entirety of iOS 12 Apple has fixed 261 CVEs, 173 of which were memory unsafety. ***That's 66.3% of all vulnerabilities****.*
Across the entirety of Mojave Apple has fixed 298 CVEs, 213 of which were memory unsafety. ***That's 71.5% of all vulnerabilities****."* – Paul Kehrer about bugs in Apple software

"There's a significant overlap between memory vulnerabilities and severe security problems. ***Of the 34 critical/high bugs, 32 were memory-related.****"* – Diane Hosfelt about bugs in the Firefox's styling code

Microsoft’s vulnerability mitigation strategy

Break exploitation techniques	• StackCookies, DEP, Heap Isolation, ACG, CFG, CIG
Eliminate vulnerabilities	• NULL deref protection, MemGC, InitAll
Contain damage & prevent persistence	• AppContainer, Virtualization
Promote safer programming languages	• .NET, Rust, C++17 & Core Guidelines

Transitioning to safer languages

```
string &not_fun(vector<char>& vec, Ending *ending) {
  string str = s
  for(auto it = vec.begin(); it != vec.end(); ++it){
    auto v = *it;
    if (v >= 65) {
      str.push_back(v+12);
    } else {
      vec.erase(it);
    }
  }
  str.push_back(reinterpret_cast<AsciiEnding*>(ending)->get_char());
  return str;
}
```

Not a smart pointer

Might be a dangling pointer

Dangling pointer

Undefined behavior if signed char overflows

Invalidates iterator

Hmmm....

No NULL check

Invalid cast

Returns a local dangling reference

Systems programming

Systems programming aims to produce software and software platforms which provide services to other software, are performance constrained, or both

Desired features

- Raw control over memory access and control flow
- Easily interoperate with existing ABIs
- Little runtime overhead

Software examples

- Kernels and Hypervisors
- Runtime libraries
- Low latency (servers, game engines, browsers)

Which are our options?

C++ >= 17
& Core Guidelines

- New features to help the user stay memory safe
- Easiest to integrate with existing C/C++ codebases
- No boundaries between unsafe and safe code

Not strong enough safety guarantees

Garbage Collected languages (C#, F#, TypeScript, Go, etc)

- Memory is managed by the runtime
- Applies bounds checks at runtime
- Impact on performance predictability

Doesn't meet all "SPL" criteria

Rust

- Memory safe by default
- Fearless concurrency: data race safe by default
- Allows memory unsafety in a well-defined scope
- Performance comparable to C++

Memory safety

Spatial memory safety

- Pointer indexing
- Buffer overflows
- Pointers lack size information
- Time of check vs time of use
- Arithmetic operations

Temporal memory safety

- Use after free
- Double free
- No ownership model
- Lack of lifetime enforcement

Type safety

- Incorrect casting
- Undefined behavior
- Unsafe operations
- Undefined states

Spatial Memory Safety

Spatial memory safety

Allocates memory

```
 *ptr1;
alloc(1)
memset(mem, 0, 2)
```

Copies more t n allocated

Checked memory access

Transfers the responsibility of verifying the bounds of the objects to the runtime or enforcing them statically. Safeguards the user from making mistakes checking sizes

Fat pointers (pointer+size)

- Rust *std::slices<T>*
- C++ *gsl::span<T>*

Arbitrary pointer dereferencing

- No ☺. Well, you can still use *unsafe*

Temporal memory safety

Temporal memory safety

Allocates memory

Frees memo

Copies pointer

Use fr
memory

```
char *ptr1;
      = alloc(..)
 ar *ptr2;
 tr2 = ptr1;
      tr1);
memset(ptr2, ..)
```

Enter memory ownership

A system that allows the developer to enforce when memory can be freed, transferred or moved, thus preventing them from accidentally creating and dereferencing dangling pointers

Single ownership

- Rust lifetimes enforced statically
- C++ *gsl::owner<T*>* (and *std::unique_ptr<T>*)

Shared ownership (Reference counted pointers)

- Rust *std::Rc<T>/std::Arc<T>*
- C++ *std::shared_ptr<T>/std::weak_ptr<T>*

Let’s move to safer languages!

The stages of software

	Hypotheses	
New software	• Usually open to new technologies	• Use safe language
Software actively developed	• Must interoperate with existing code • Prohibitive to introduce a different runtime	• Use safe language if possible • Adopt safer development practices
Software in maintenance	• Only fixes • Must be careful with regressions	• Rewrite is expensive • Adopt safer practices if new code is added

Mitigations!

Conclusion

If possible, switch to safer languages like Rust or .NET

Transition to C++ 17 and enforce the Core Guidelines

In 20 years we shouldn’t be introducing any memory safety vulnerabilities ☺

Report vulnerabilities &
mitigation bypasses via our
bounty programs!

https://aka.ms/bugbounty

© 2019 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.