Windows

The Evolution of CFI Attacks and Defenses

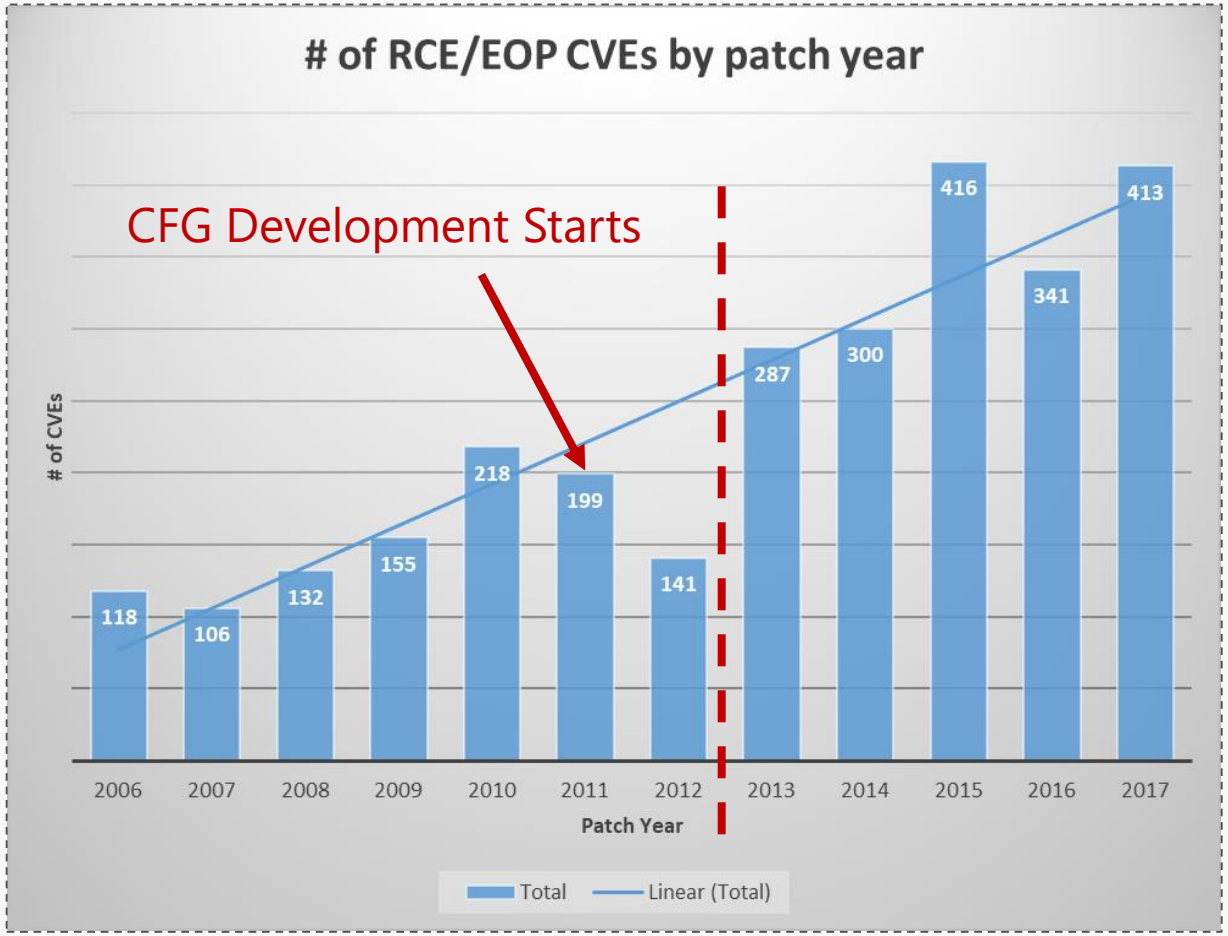
Joe Bialek (@JosephBialek) – MSRC Vulnerabilities & Mitigations Team

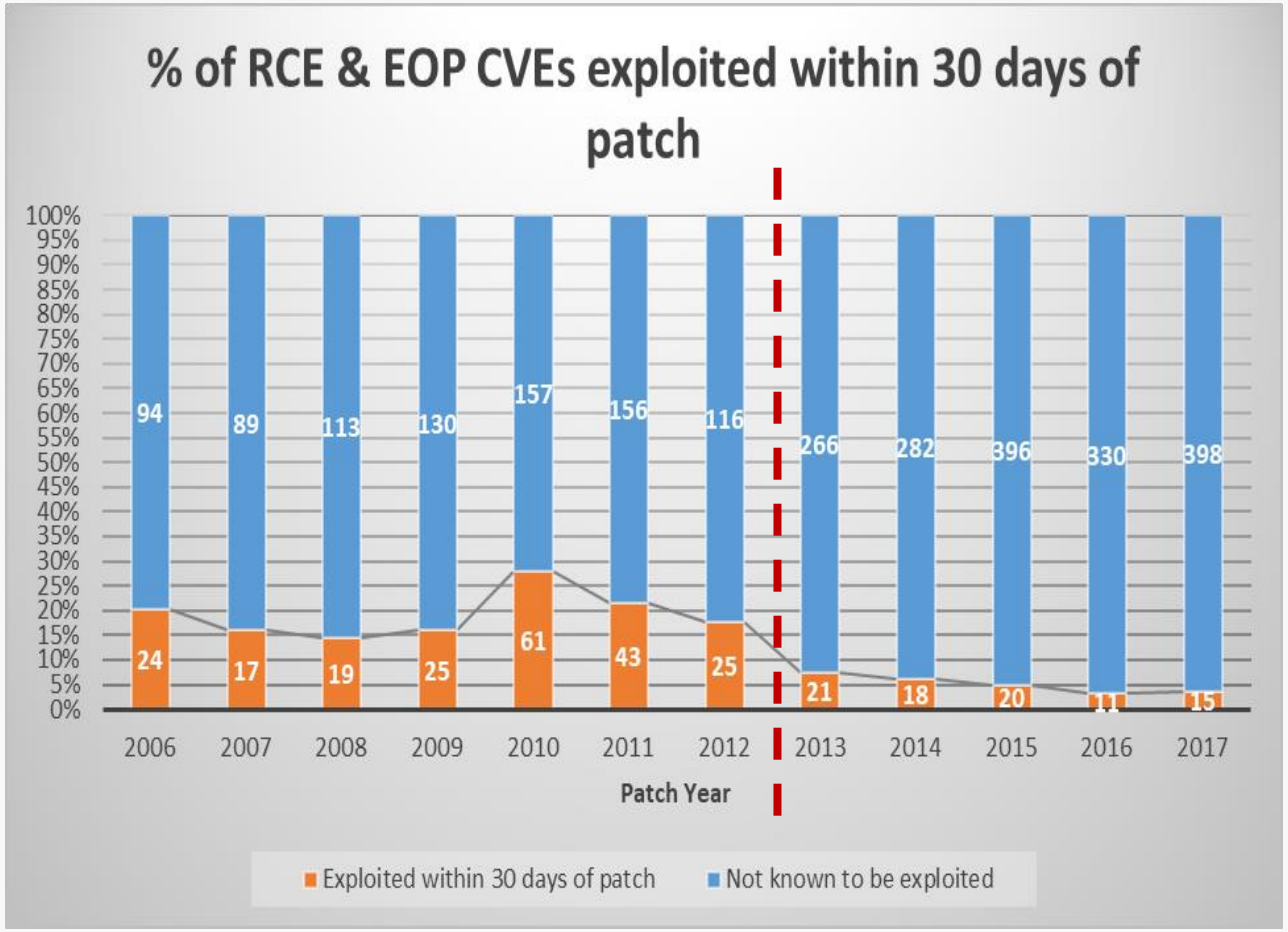
Microsoft

Historical Context (circa 2011-2012)

Problem: Vulnerabilities Impacting Customers

Vulnerabilities and exploitation are on the rise, Microsoft needs to protect customers





Realities (circa 2011-2012)

What we are doing isn't working

We cannot "find all the bugs"

- SDL
- Pentesting / fuzzing
- Static Analysis

Simplistic mitigations aren't robust against strong primitives

- DEP
- ASLR
- Heap metadata hardening

We have limited options

Solutions must work for our most attacked targets (IE/Office)

Hardcore sandboxes not an option for IE/Office due to:

- Compatibility
- Huge engineering cost + lack of developer resources

All changes have compatibility & performance hurdles to jump, must invest wisely

CFI Strategy

Theory

Microsoft's overarching goal is to make exploitation financially infeasible or impossible

All RCE memory corruption exploits found in-the-wild hijack control flow

Attackers often follow "path of least resistance", breaking them means increasing cost of exploitation

Constraining control flow to "legitimate" paths breaks all of these exploits as-written

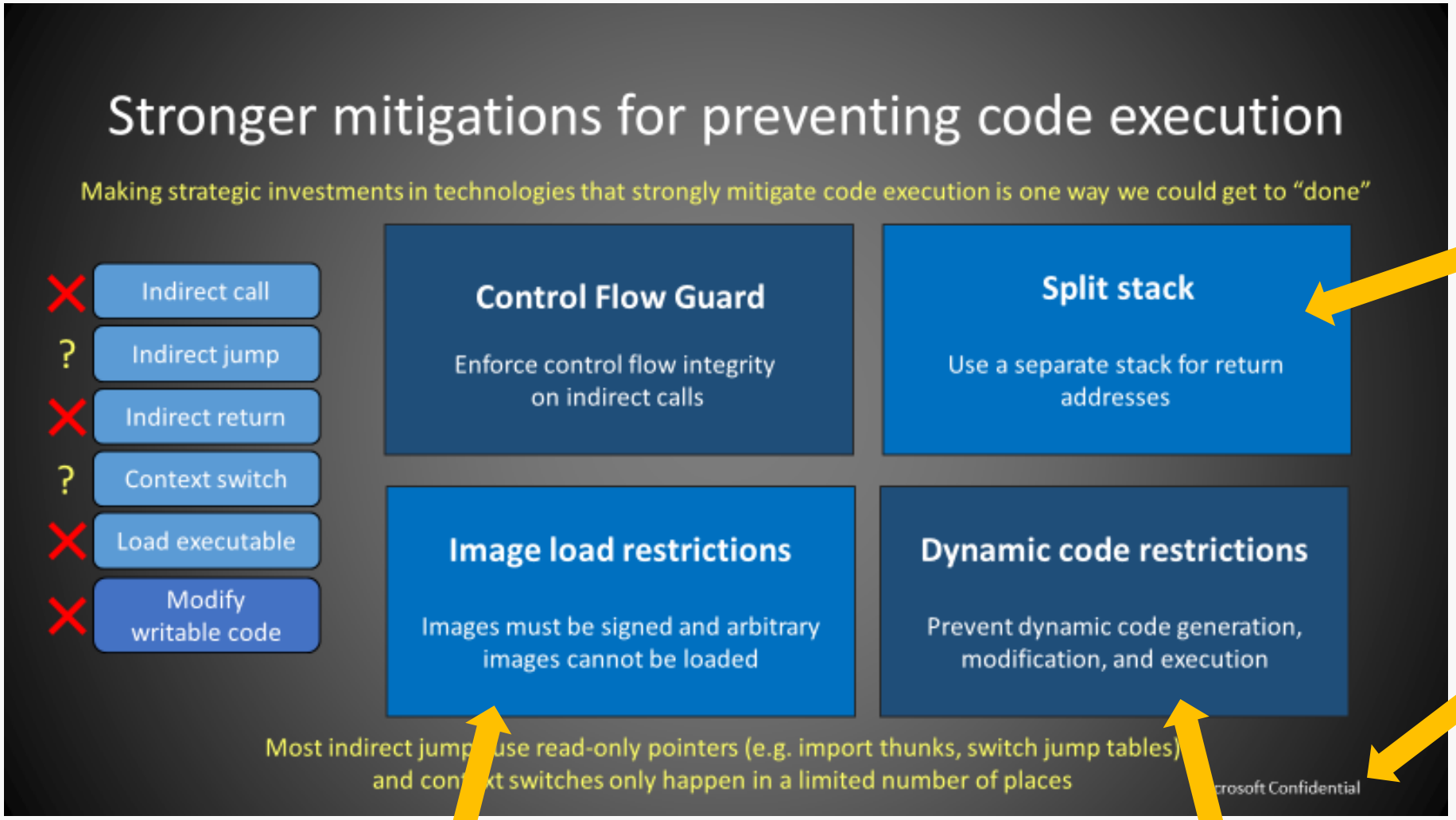
After some formal thought, we believe CFI will robustly mitigate against stronger primitives

Security teams are well positioned to drive these changes

CFG had no formal threat model during very early development. Thought of as a way to kill ROP.

Hindsight is 20/20, but we did have formal thought around future exploit trends. See [1]

2012 Strategy Slide Deck



Intel CET (hardware shadow stack)

Not Anymore ☺

Code Integrity Guard (CIG) + NoChildProc + NoLowLabel + NoRemoteImage

Arbitrary Code Guard (ACG)

"Since we will be the first major software vendor to deploy a control-flow integrity solution at scale, we will have the great distinction of being the first to experience the school of hard knocks."

– Internal CFG Security Impact Assessment Document

“The School Of Hard Knocks”

Or: How our understanding of CFI robustness has evolved over time

How Did We Get Here?

Mitigation	In scope	Out of scope
Control Flow Guard(CFG)	Techniques that make it possible to gain control of the instruction pointer through an indirect call in a process that has enabled CFG.	• Hijacking control flow viare turn address corruption • Bypasses related to limitations of coarse-grained CFI (e.g. calling functions out of context) • Leveraging non-CFG images • Bypasses that rely on modifying or corrupting read-only memory • Bypasses that rely on CONTEXT record corruption • Bypasses that rely on race conditions or exception handling • Bypasses that rely on thread suspension • Instances of missing CFG instrumentation prior to an indirect call

How Does CFG Work?

CFG implements **coarse-grained control-flow integrity** for indirect calls

Compile time

```
void Foo(...) {
   // SomeFunc is address-taken
   // and may be called indirectly
   Object->FuncPtr = SomeFunc;
}
```

Metadata is automatically added to the image which identifies functions that may be called indirectly

```
void Bar(...) {
   // Compiler-inserted check to
   // verify call target is valid
   _guard_check_icall(Object->FuncPtr);
   Object->FuncPtr(xyz);
}
```

A lightweight check is inserted prior to indirect calls which will verify that the call target is valid at runtime

Runtime

Process Start
- Map valid call target data

Image Load
- Update valid call target data with metadata from PE image

Indirect Call
- Perform O(1) validity check
- Terminate process if invalid target
- Jmp if target is valid

CFG is a deterministic mitigation, its security is not dependent on keeping secrets.

For C/C++ code, CFG requires no source code changes.

```
ntdll!LdrpDispatchUserCallTarget:
00007ffb`4e100e10 4c8b1d59e50d00  mov     r11,qword ptr
[ntdll!LdrSystemDllInitBlock+0xb0]
00007ffb`4e100e17 4c8bd0          mov     r10,rax
00007ffb`4e100e1a 49c1ea09        shr     r10,9
00007ffb`4e100e1e 4f8b1cd3        mov     r11,qword ptr [r11+r10*8]
00007ffb`4e100e22 4c8bd0          mov     r10,rax
00007ffb`4e100e25 49c1ea03        shr     r10,3
00007ffb`4e100e29 a80f            test    al,0Fh
00007ffb`4e100e2b 7509            jne     ntdll!LdrpDispatchUserCallTarget+0x26
ntdll!LdrpDispatchUserCallTarget+0x1d:
00007ffb`4e100e2d 4d0fa3d3        bt      r11,r10
00007ffb`4e100e31 7303            jae     ntdll!LdrpDispatchUserCallTarget+0x26
ntdll!LdrpDispatchUserCallTarget+0x23:
00007ffb`4e100e33 48ffe0          jmp     rax
…
…
```

Known Limitations Prior To Shipping

Stack Corruption	• No protection for return addresses • X86 mixed calling conventions can lead to stack imbalance • http://labs.bromium.com/2015/09/28/an-interesting-detail-about-control-flow-guard/
Compat	• New executable pages default to: all addresses valid icall targets • Non-CFG binaries can be loaded in CFG enabled processes • Some binaries aren't fully instrumented (assembly, libs)
Coarse Grained	• Calling functions out of context can lead to stack address leaks • Legitimate functions like VirtualProtect can be called
Dangerous Functions	• Functions like NtContinue & longjmp directly set RIP/RSP/RBP
Data Corruption	• Attacker can modify parameters to WinExec, LoadLibrary, etc.

Prior to CFG shipping, we were evaluating it in terms of an arbitrary read/write primitive

CFG Ships to the World

- November 2014: Optional rollup for Windows 8.1
  - Flash is compiled with CFG
  - IE/Flash JITs are not enlightened (all JIT memory valid icall targets)
  - WARP D3D JIT not enlightened
- Late 2014: two flash CVEs under active exploitation
  - CVE-2014-0311 and CVE-2014-0313
  - Exploit doesn't bypass CFG, customers with the latest updates are protected ☺
- February 2015: CFG is enabled by default for Windows 8.1

2015 Learning: External Exploits

Pwn2Own

- IE Entries
  - Indirectly call WinExec with controlled parameters
  - Leak stack addresses and call NtContinue
    - Stack address leaked from jscript9!StackPinnedBase::global

- Flash Entries
  - Leak stack address from the flash heap, call NtContinue
  - Global in flash.ocx contains pointer to RWX memory (used to store ATL thunks)
    - CRT already updated to fix this, Flash was using old CRT ☹

Exploits in-the-wild

- Not covered in this slide deck because they follow the same patterns as Pwn2Own entries & our own internal research

2015 Learnings and Observations

- Internal prototyping on “CFG+”, finer grained CFG
  - No line-of-sight to truly fine grained CFG on Windows due to compat ☹
  - The granularity we could achieve (number of function arguments) not worth the development cost
  - Note: Fine grain CFG was considered in the original design but wasn’t pursued for version 1
- Internal red team identifies “stack spray” attack, predictable way to locate stacks
  - MM uses the same allocator for stacks & VirtualAlloc
  - Make large allocations in JS, free them, cause thread creation. VirtualAlloc puts the thread stack where the previous large JS allocation was
- DLL’s providing scripting engines (PowerShell, VBScript) can be loaded into CFG processes

What We Did: Windows 10 TH1 (July 2015)

Strategic Changes:

- Edge replaces IE as the default Microsoft browser
  - Edge JIT supports CFG (only entry points valid indirect calls, JIT pages RX)
  - Edge has a much better sandbox

Tactical Changes:

- "Explicit Suppression": Dangerous functions (e.g. WinExec, NtContinue, longjmp) no longer valid indirect call targets
- A number of "stack address leaks" are fixed
  - Found using both static analysis and dynamic analysis
  - We quickly determine this is a rats nest and not viable to pursue long term
- Instrumentation gaps fixed (found via internal tooling & mitigation bounty)
- Improvements to WOW64 with CFG

What We Did: Windows 10 TH2 (Nov 2015)

Strategic Changes:

- Edge prevents image loading from remote shares (kernel enforced)
- Edge only loads Microsoft signed binaries (CIG)

Tactical Changes:

- Edge opts in to additional explicit suppression
  - Edge can be more aggressive than the system-wide explicit suppression shipped in TH1
- Stack isolation / TEB / PEB isolation
- Eliminated some “dangerous wrappers” (valid call targets that wrap explicitly suppressed call targets such as SetProcessValidCallTargets)
  - Found using static analysis

2016 Learnings: External Exploits

Pwn2Own (Edge Browser):

- Edge: Call setjmp to locate stack, corrupt return address
  - ROP payload uses VirtualProtect to make memory RWX and deliver the final payload
- Flash: Leak a stack address from flash internal structures, corrupt return address

PwnFest (Edge Browser):

- Edge #1: Out-of-bounds write on the stack used to corrupt return address. Second stage payload delivered in a RWX page setup using ROP.
- Edge #2: Stack address leaked from heap, return address corrupted. Second stage payload delivered in a RWX page setup using ROP.

All exploit competition entries hijack control flow by directly corrupting the stack

2016 Learnings: Bounty Reports

Memory Protection Issues

- Attacks against read-only memory
  - Make the page storing ntdll! __guard_dispatch_icall_fptr writable, set it to _guard_dispatch_icall_nop
- Race condition: Modify JIT'd code before it's made read-execute
- Wrappers around explicitly suppressed functions that are valid icall targets
  - Example: Wrapper around VirtualProtect allows making arbitrary virtual address RWX

Compiler Bugs

- DLL generated with writable Import Address Table (IAT)
- Compiler generated thunks that make unprotected icalls were marked as valid icall targets

Data Only Attacks

- Modify a global variable containing a DLL path
  - Didn't really work against Edge due to CIG + NoRemoteImages

Misc.

- Stack address leaks
- Uninstrumented indirect calls
- Coarse grained CFI limitations [3]

2016 Learnings: Observations & Issues

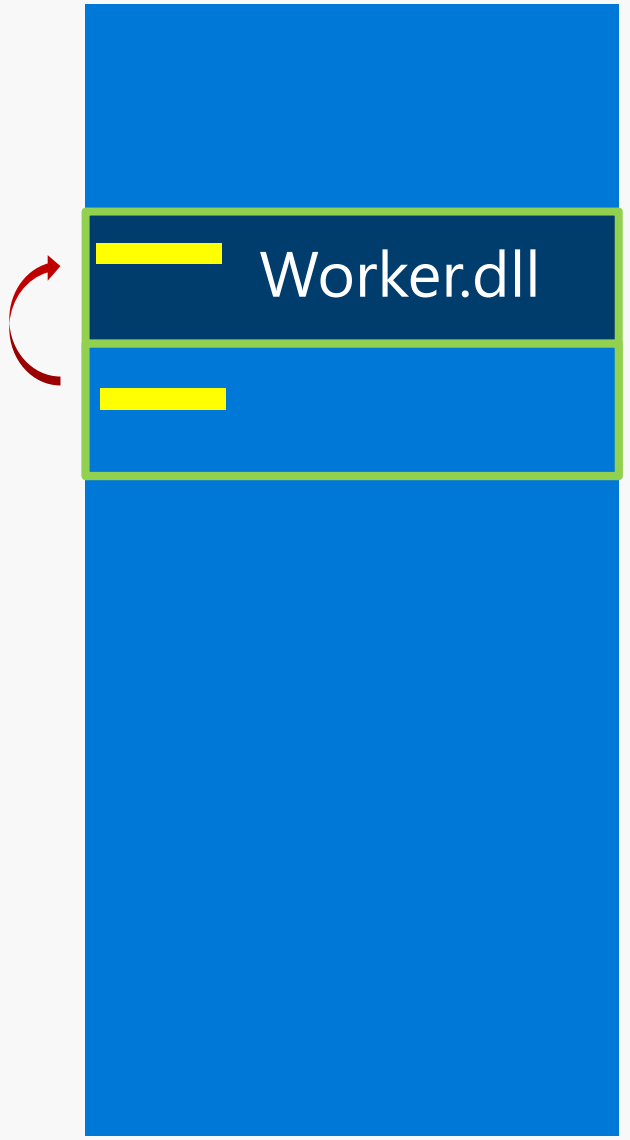
- External & internal research indicates that data-only corruption is sufficient to perform arbitrary actions (e.g. system calls with controlled parameters)
  - While there may be cases where this isn't true, it's difficult to reason about so we must assume the worst
  - Still makes writing payloads more expensive
  - Emphasizes that CFI goes hand-in-hand with sandboxing, CFI makes breaking the sandbox much more painful
- Binary downgrade attacks
  - Attacker loads MS signed binary with known CFG bypasses

2016 Learnings: Code Replacement Attacks

Foo::StartWork has called the function Worker::DoWork.

These functions are in different DLL's, both DLL's are signed by Microsoft.

2016 Learnings: Code Replacement Attacks



Attacker forces Bar.dll to be unloaded. We assume this is possible via data corruption attacks.

2016 Learnings: Code Replacement Attacks

Attacker forces Bar.dll to be loaded in the virtual address space that previous mapped Foo.dll.

Bar.dll is signed by Microsoft.

2016 Learnings: Code Replacement Attacks

Worker.dll::DoWork eventually returns using the return pointer stored on it's stack. This used to point in to Foo.dll::StartWork but because Foo.dll was unloaded, the return pointer is now a dangling pointer.

DoWork ends up returning to a ROP gadget in Bar.dll.

The Stack Conundrum

- Most CFG improvements will provide little value-add until stack protection lands, attackers are unanimously corrupting the stack
- Microsoft would prefer not to wait for CET
  - Even if it was released tomorrow, market penetration would take 6+ years
- Should we shift away from CFI in the short-term?
- If only there was a performant, compatible, and robust way to protect returns in software…

RFG – Return Flow Guard

RFG was our compatible, ABI compliant, performant software shadow stack

Compile Time

NOP's added to the prolog & epilog of all functions

Metadata added to the image to locate the prolog and epilog NOP bytes

Runtime

Process Start

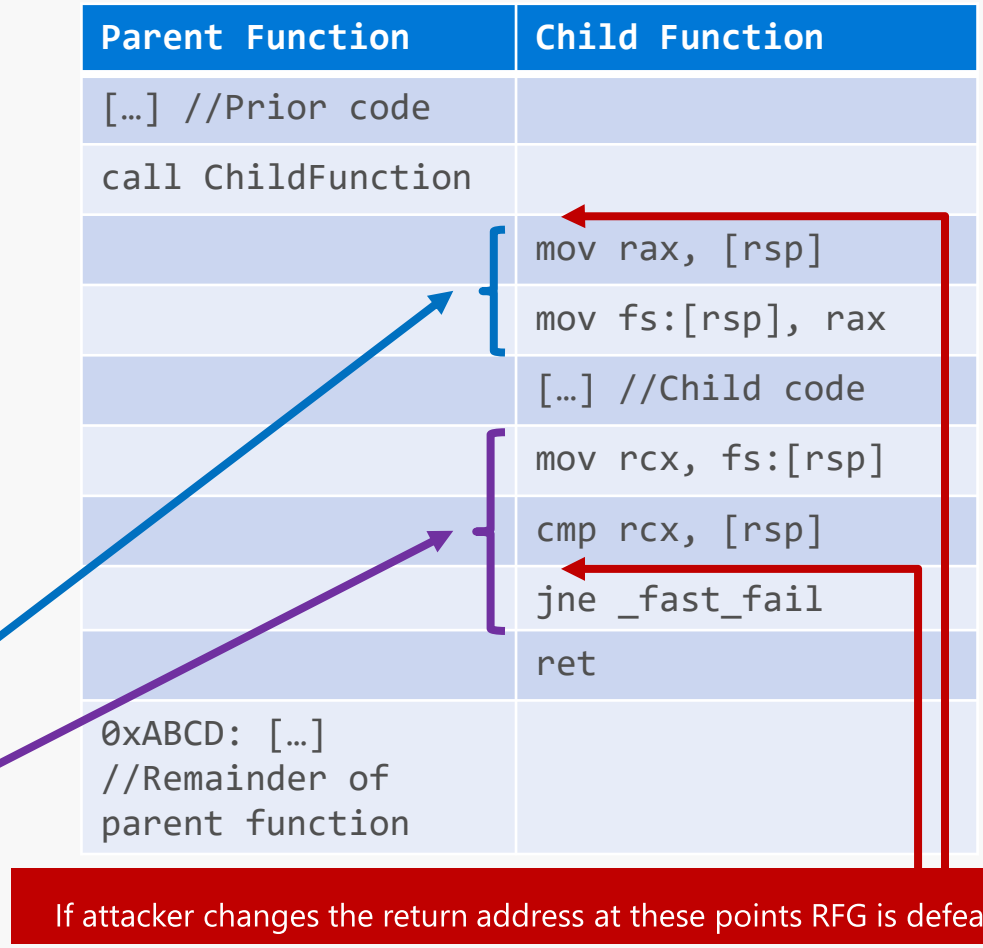
- 1TB shadow stack region created
- Region cannot be queried
- A/V's in region are fatal
- FS segment points to the shadow stack of the current thread

Image Load

- If process enables RFG: patch NOP's with RFG prolog/epilog

Function Calls

- Prolog: Push return address to shadow stack
- Epilog: Fast fail if return address on stack and shadow stack are mismatched



RFG relies on a secret: the shadow stack's virtual address

RFG Learnings: Secrets Are Bad!

- AnC attack [2] could successfully leak where shadow stacks were mapped
  - POC took several minutes in Edge, but attacks never get slower

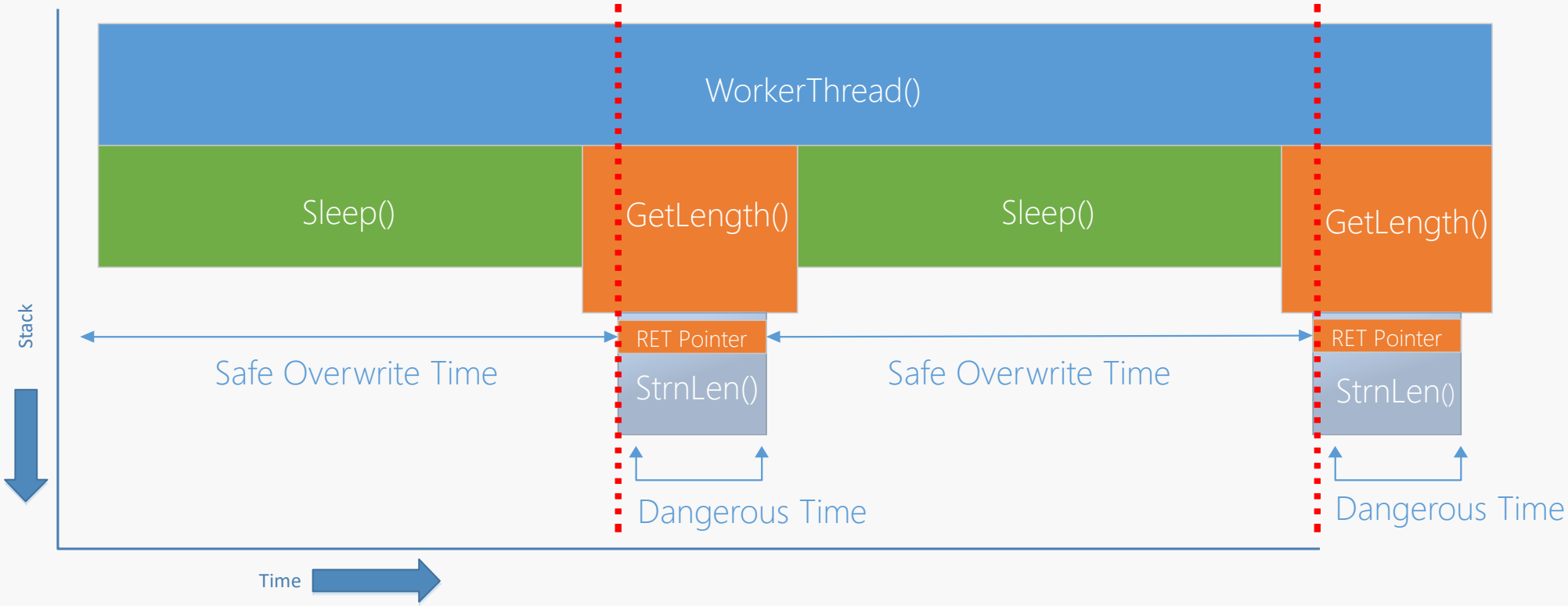
- CFG originally designed to avoid using secrets, we thought we could get away with it here, but we couldn’t

RFG Learnings: The By-Design Race

- RFG had a by-design race condition that we thought was unexploitable

- Theory: If you attempt to "win the race" and corrupt the return address of another thread, you'll likely:
  - Corrupt the return address too late and the process will fast fail
  - Do your corruption at the wrong time and corrupt the stack frame of an unexpected function

RFG Learnings: Winning the Race



Thread 1: In the above sleep() -> GetLength() -> sleep() loop
Thread 2: Constantly writing to the virtual address of “RET Pointer” of the strnlen function

By attacking a leaf function, 99.99% of the “writes” are harmless. When the leaf function is entered, you have a very high probability of winning the race.

Credit goes to Reid Borsuk for discovering this technique

Exploitable Races + CFI = A New Threat Model

- Techniques exist for extending race windows
  - Find races that cross cache line boundaries
  - SuspendThread/GetThreadContext
- More races exist
  - CFG check function (code replacement attack after check passes but before indirect call is made)
  - Unwinder
- CFI must be threat modeled against: Arbitrary read/write at arbitrary times
  - This is one of the most powerful primitives an attacker could achieve
- There is no CFI solution that is robust against this threat model today
  - But we think CFG + CET can be

Windows RS1 Improvements (August 2016)

Strategic Changes:

- "No Child Process" mitigation policy enabled for Edge

Tactical Changes:

- Longjmp hardening
  - Instruction pointer protected in a coarse-grained fashion
  - Compiler determines were setjmp is called and long allows longjmps to corresponding locations
  - Stack address can become unaligned
  - Not safe from races

2017 Learnings

Pwn2Own (Edge Browser)

- All exploits follow the pattern:
  - Corrupt the stack
  - Use ROP to allocate RWX memory
  - Execute stage 2 payload from RWX memory

Bounty

- Most bounty cases: binaries missing instrumentation
- Some implementation issues with ACG
  - Thanks to Alex Ionescu [4] & Ivan Fratric [5]

RS2&3 Improvements (April/October 2017)

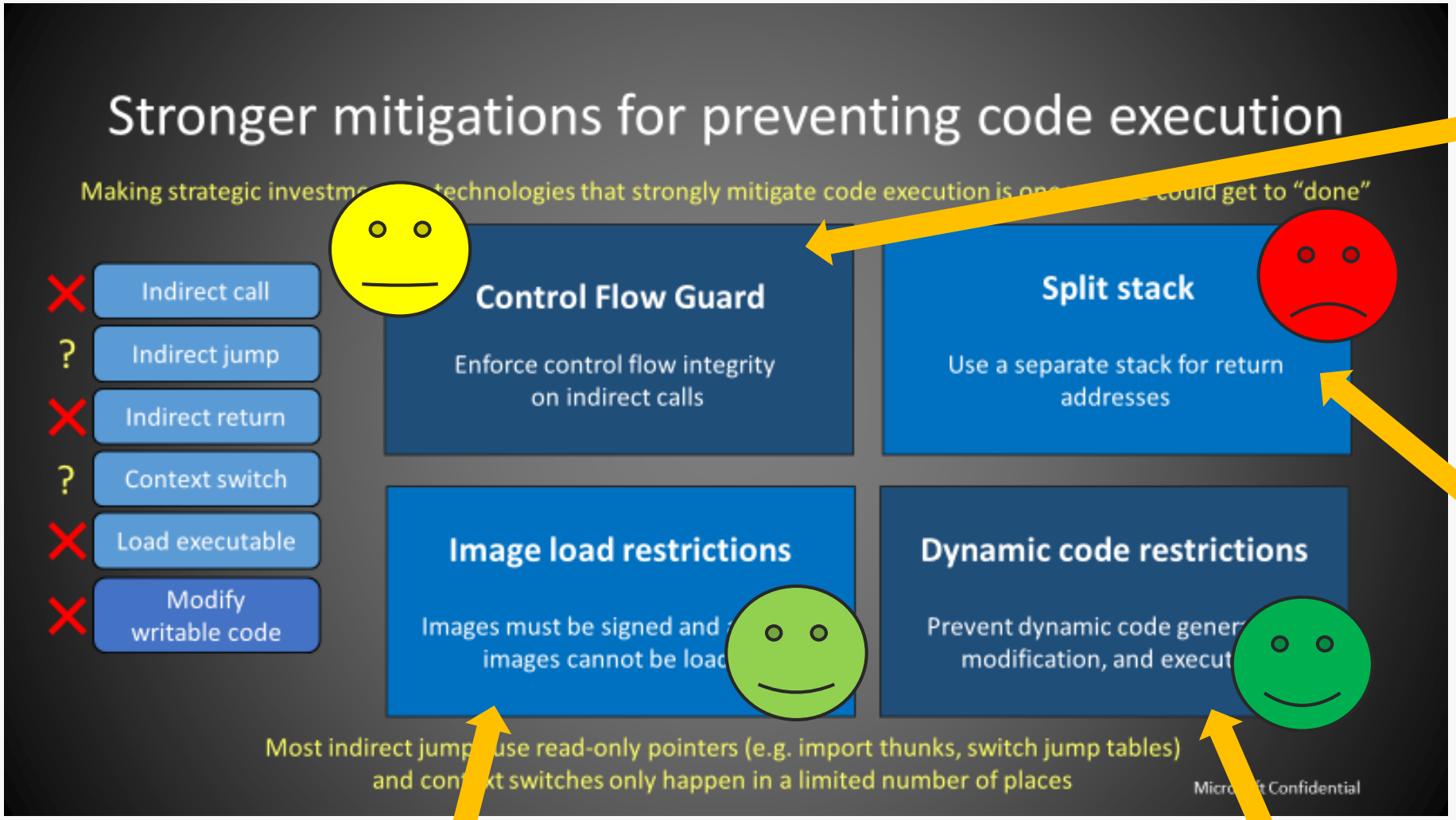
Strategic Changes:

- Arbitrary Code Guard enabled for Edge
  - Exploits will no longer be able to create RWX memory in Edge
- NoLowMandatoryLabelImages enabled for Edge
  - Prevents images with a Low mandatory label from being loaded
  - Helps prevent downgrade attacks
- Support for CFG “strict mode” added to Windows
  - Only binaries that were compiled with CFG can be loaded
  - Issue: Binaries compiled with non-CFG lib’s or non-CFG assembly can be loaded

Tactical Changes:

- Export suppression added (& enabled in Edge)
  - DLL exports are not valid icall targets unless GetProcAddress is called on them
  - Makes exploits slightly more difficult, but bypassable by-design

Where Are We Now (Edge specific)?



CFG has shipped, outstanding design issues

No Intel CET yet

Code Integrity Guard (CIG) + NoChildProc + NoLowLabel /NoRemote Images

Arbitrary Code Guard (ACG)

What's Left To Do?

Code Integrity Guard	• Needs anti-rollback protection • Need to control what binaries are allowed to load (i.e. no scripting engines like PowerShell)
Arbitrary Code Guard	• Design is working as intended
CET	• Need hardware to be released • Microsoft is planning on using CET for backwards edge protection
CFG	Code Replacement Attacks • Broker only allows executable pages to be unmapped when safe Race Conditions • Unwinder needs to be out-of-process • CFG check race killed with "code replacement attack" mitigation X86 Stack Misbalancing & Generic Stack Address Leaks • Mitigated by CET Read-Only Memory Attacks • Certain regions must be permanently read-only and not unmappable. Broker decides when it is safe to allow unmap • Operations that require write access to this memory (i.e. when delay loading a DLL) must be OOP Coarse Grained CFI Limitations • Look for opportunities to make CFG more fine grained

Takeaways

Building a robust CFI solution is quite difficult

For CFI to be robust in all environments, it must be modeled against arbitrary read/write at arbitrary times

No CFI solution currently available is robust against this primitive, Microsoft thinks we can get there

To maximize the impact of CFI, it must be combined with sandboxing

Even non-robust CFI does increase exploit complexity & not all environments give attackers as much control as web browsers

Bounties @ Microsoft

Bounty	Max Payout
Mitigation Bounty	$100,000
Hyper-V Bounty	$250,000
WDAG (Windows Defender Application Guard)	$30,000
WIP (Windows Insider Preview)	$15,000

See all bounties at: https://aka.ms/BugBounty

References

1. Modeling the exploitation and mitigation of memory safety vulnerabilities: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2012_10_Breakpoint/BreakPoint2012_Miller_Modeling_the_exploitation_and_mitigation_of_memory_safety_vulnerabilities.pdf
2. AnC Attack: https://www.vusec.net/projects/anc/
3. From read-write anywhere to controllable calls: https://medium.com/@mxatone/mitigation-bounty-from-read-write-anywhere-to-controllable-calls-ca1b9c7c0130
4. The Bird that Killed Arbitrary Code Guard: https://www.youtube.com/watch?v=gu_i6LYuePg
5. ACG Bypass Using Duplicate Handle: https://bugs.chromium.org/p/project-zero/issues/detail?id=1299