



# Microsoft's strategy and technology improvements toward mitigating arbitrary native code execution

David Weston, Windows Offensive Security Research (OSR)  
Matt Miller, Microsoft Security Response Center (MSRC)

@dwizzleMSFT  
@epakskape

March, 2017

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



# Agenda

How we think about mitigating software vulnerabilities

Our progress toward mitigating native code execution

Impact, lessons learned, and future plans

# Imagine you could change the laws of physics

Vertical engineering is a super power when it comes to defense



Change or enable apps to better protect themselves

Microsoft Azure

Change the cloud platform to protect online services & data



Change the OS to protect apps, data, & devices

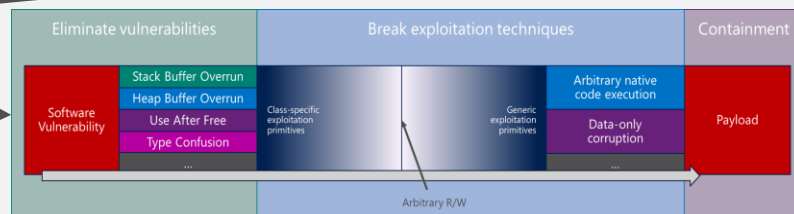
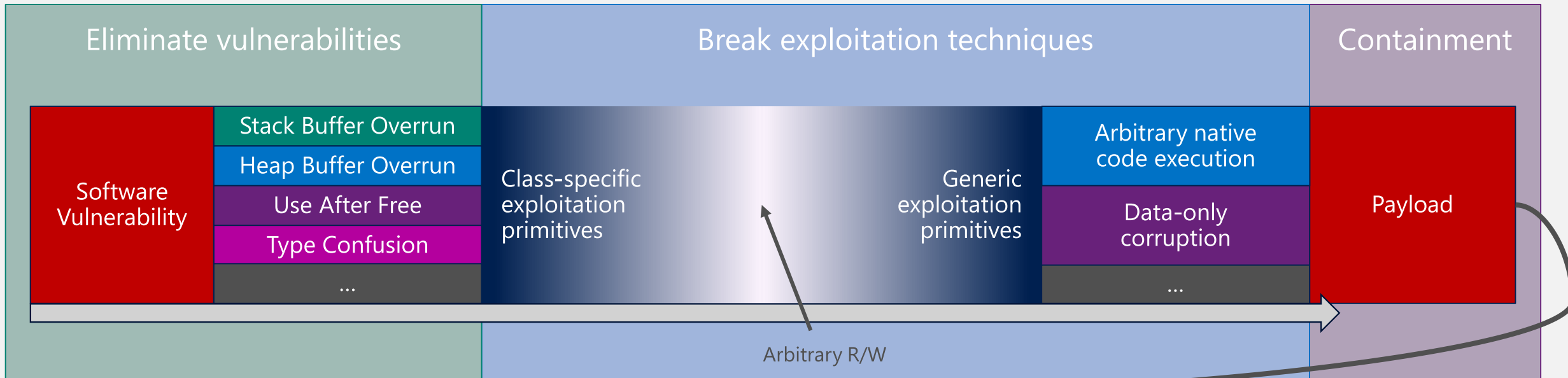


Change the compiler to produce safer code

Collaborate with hardware partners to improve the foundation

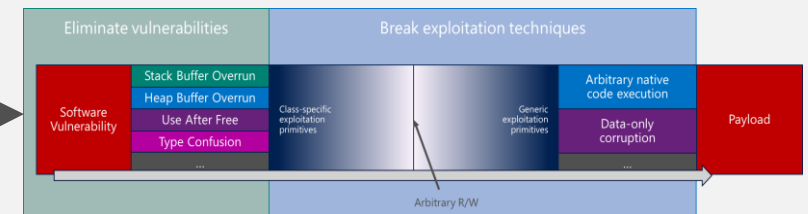
# How we think about mitigating software vulnerabilities

Attackers transform software vulnerabilities into tools for delivering a payload to a target device



**Attackers typically need to elevate privileges**

This means applying the same defenses to privileged attack surfaces



**At some point, we lose containment as a defense**

This leaves eliminating vulnerabilities & breaking techniques

# Layered, data-driven software defense in Windows 10

## Our Strategy

Make it difficult & costly to find, exploit, and leverage software vulnerabilities

## Our Tactics

Eliminate entire classes of vulnerabilities

Break exploitation techniques

Contain damage & prevent persistence

Limit the window of opportunity to exploit

Today, we'll be focusing on one aspect of breaking exploitation techniques

# Chokepoints for breaking exploitation techniques

All exploits rely on combining various primitives to enable the delivery of a payload

Break exploitation techniques

Class-specific  
exploitation  
primitives

*Weak mitigations*

Generic  
exploitation  
primitives

Arbitrary native  
code execution

Data-only  
corruption

...

Chokepoint #2

**Break generic primitives** with the assumption that an attacker has arbitrary R/W.

Goal: make it difficult or impossible to deliver a payload independent of the type of vulnerability.

Arbitrary native code execution

- Generic, vulnerability-independent primitive
- Universal & flexible way to execute a payload
- Ubiquitously relied upon by exploits

Good candidate for mitigations 😊

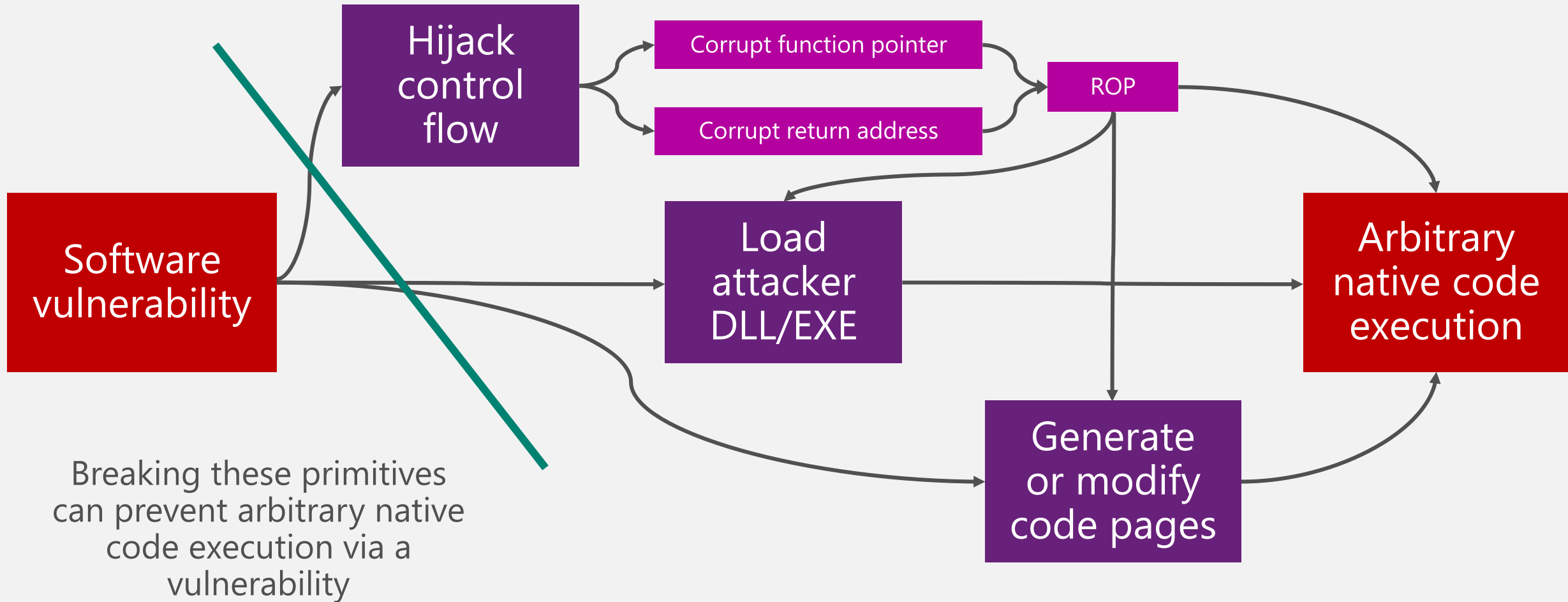
Chokepoint #1

**Break class-specific techniques** for transforming a vulnerability into generic primitives.

Goal: make it difficult or impossible to exploit certain types of vulnerabilities.

# The paths to arbitrary native code execution

There are a finite number of ways to transform a vulnerability into arbitrary native code execution



Mitigating arbitrary native  
code execution



# Defining our threat model & assumptions

It is critical to define the key aspects of the threat model that we want to defend against

## **A vulnerability exists**

There is a memory corruption vulnerability that could potentially be exploited

## **Interactive runtime is present**

The attacker is able to leverage an interactive runtime (such as a script engine) in conjunction with their exploit

## **Arbitrary R/W at arbitrary times**

The attacker can read/write arbitrary memory locations, with controlled data, at arbitrary points in time

## **Address space layout is known**

The attacker knows where all discoverable memory regions are located – traditional ASLR is a non-factor

This threat model sets a very high bar, but it is grounded by real-world data & expectations

# Technologies for mitigating code execution

Prevent  
arbitrary code  
generation

## Code Integrity Guard

Images must be signed and load  
from valid places

## Arbitrary Code Guard

Prevent dynamic code generation,  
modification, and execution

Prevent  
control-flow  
hijacking

## Control Flow Guard

Enforce control flow integrity  
on indirect function calls

???

Enforce control flow integrity on  
function returns

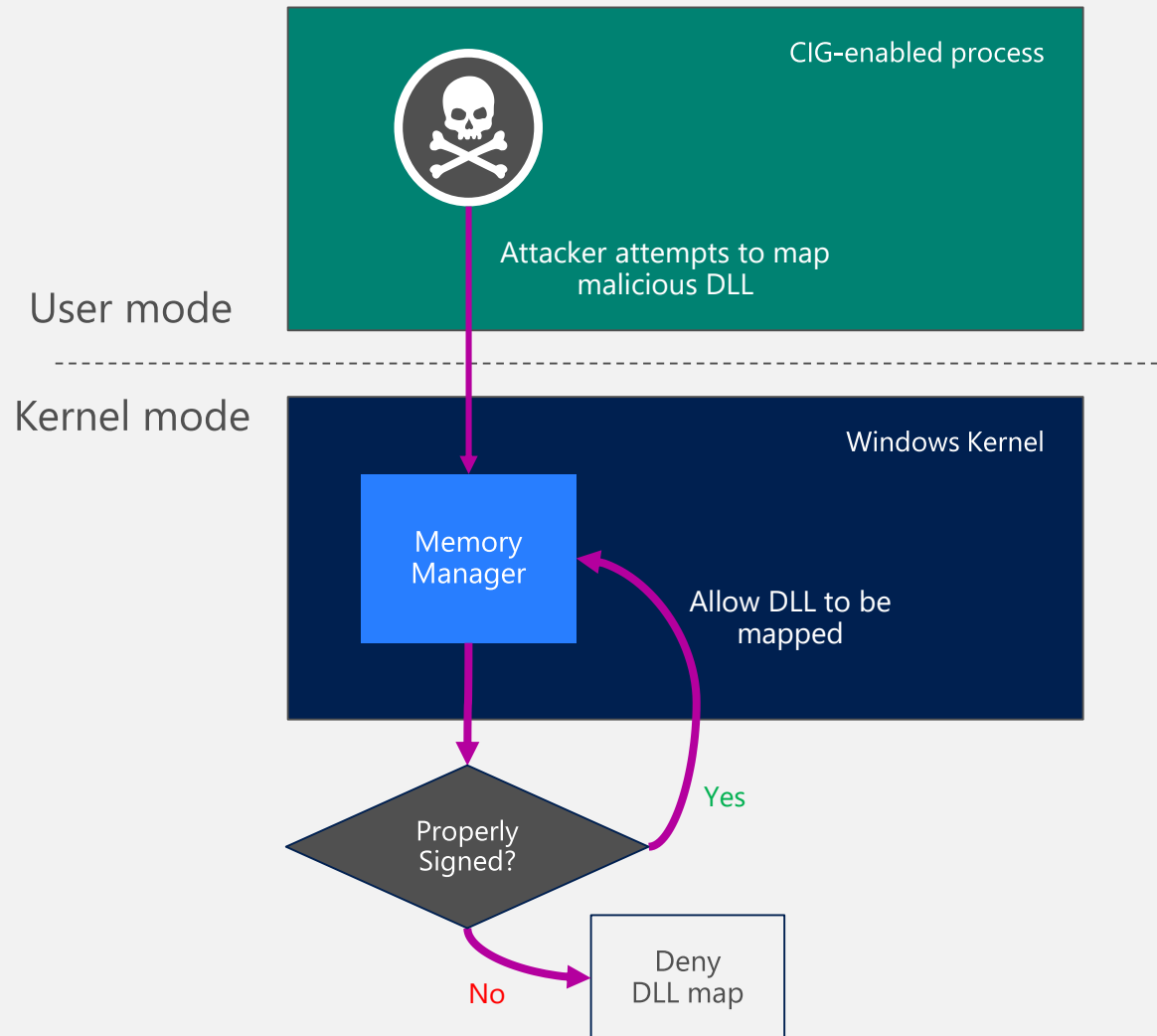
✓ Only valid, signed code pages can  
be mapped by the app

✓ Code pages are immutable and  
cannot be modified by the app

✓ Code execution stays “on the rails”  
per the control-flow integrity policy

# Code Integrity Guard (CIG)

CIG leverages code signing restrictions to prevent attacker DLLs from loading



Example of such an attack [provided by Yang Yu](#) @ Black Hat USA 2014

## “LoadLibrary” via JavaScript

1. Download a DLL by XMLHttpRequest object, the file will be temporarily saved in the cache directory of IE;
2. Use "Scripting.FileSystemObject" to search the cache directory to find that DLL;
3. Use "Scripting.FileSystemObject" to create a directory named "System32", copy the DLL into that directory, and named it as "shell32.dll";
4. Modify the "SystemRoot" environment variable of current process via "WScript.Shell" object to the upper directory of the "System32" directory created just now;
5. Create "Shell.Application" object, trigger to loading "%SystemRoot%\System32\shell32.dll".



## Code Integrity Policies

Only allow Microsoft-signed DLLs

Only allow Microsoft, Store, or WHQL signed DLLs

Configurable via Device Guard policy

# Considerations for enabling CIG for an app

**3<sup>rd</sup> party improperly signed binaries that need to be loaded**

- Trade-off between security and functionality
- Microsoft Edge will currently disable CIG when 3<sup>rd</sup> party IMEs are installed

# Arbitrary Code Guard (ACG)

Exploits often rely on creating or modifying executable pages to execute their payload

ACG enables two kernel-enforced W^X policies

- ✓ Code is immutable
- ✓ Data cannot become code



Code Integrity Guard (CIG)

- ✓ Only properly signed code pages can be mapped

The following will fail with `ERROR_DYNAMIC_CODE_BLOCKED`

```
VirtualProtect(codePage, ..., PAGE_EXECUTE_READWRITE)
```

```
VirtualProtect(codePage, ..., PAGE_READWRITE)
```

```
VirtualAlloc(..., PAGE_EXECUTE*)
```

```
VirtualProtect(dataPage, ..., PAGE_EXECUTE*)
```

```
MapViewOfFile(hPagefileSection, FILE_MAP_EXECUTE, ...)
```

```
WriteProcessMemory(codePage, ...)
```

```
...
```

ACG + CIG ensure  
the integrity of code  
pages

# Considerations for enabling ACG for an app

**Just-in-Time (JIT)  
compilation no longer  
works in-proc**

- JITs need to move out-of-process or be disabled
- Microsoft Edge has moved JITs out-of-process in the Creators Update of Windows 10

**3<sup>rd</sup> party binaries may be  
incompatible**

- Binaries must enable /DYNAMICBASE, cannot have RWX sections, and must not merge their import table into a code section
- BinSkim has checks for this, see <https://github.com/Microsoft/binskim>
- Microsoft Edge will currently disable ACG when incompatible graphics drivers may be present

**Does not prevent code  
injection from privileged  
processes**

- Privileged code running outside of the app can still inject dynamic code into an ACG-enabled process

# Control Flow Guard (CFG)

Modern exploits typically rely on hijacking control-flow through an indirect call

Example control-flow hijack via indirect call to a ROP gadget[1]

```
/* Corrupt sound object vtable ptr */
while (1)
{
    if (this.s[index][j] == 0x00010c00 && this.s[index][j+0x09] == 0x1234)
    {
        soundobjref = this.s[index][j+0x0A];
        dec = soundobjref-cvaddr-1;
        this.s[index][dec/4-2] = cvaddr+2*4+4*4;
        break;
    }

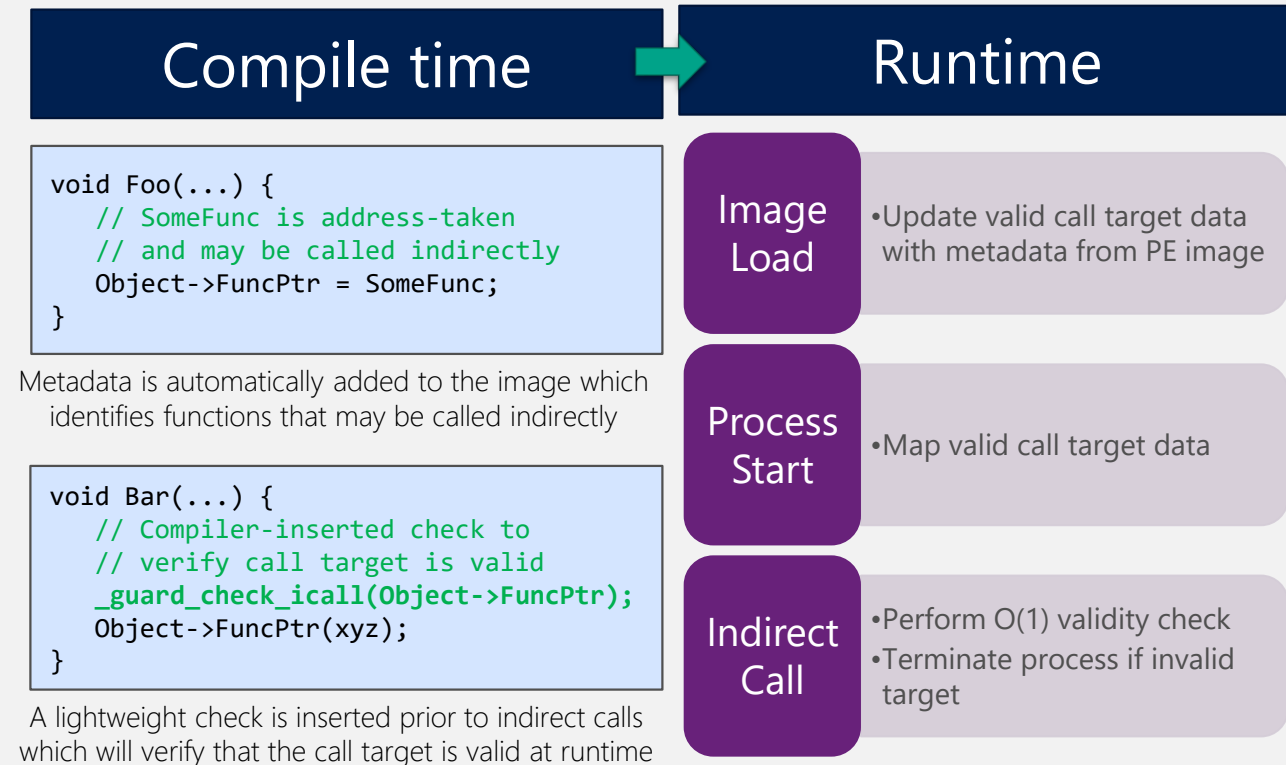
    j++;
}

/* Run Payload */
this.sound.toString();
```

Transfers control to a stack pivot ROP gadget

With CFG in place, traditional ROP gadgets and other invalid functions cannot be called indirectly

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



[1] <https://github.com/rapid7/metasploit-framework/blob/abd76c50000e75bcac0616b96cd8583e1df3927f/external/source/exploits/CVE-2014-0322/AsXploit.as>

# Control Flow Guard Bypasses & Enhancements

Like all mitigations, CFG has by design limitations that place constraints on its overall effectiveness

- ✓ Return addresses are not protected
- ✓ Valid functions can be called out of context
- ✓ “Fail-open” design for compatibility

Since shipping CFG, researchers have identified bypasses and additional enhancements have been made

Bypass	Status
Non-enlightened Just-in-Time (JIT) compilers	Mitigated in latest version of Edge on Windows 10 (Chakra, Adobe Flash, and WARP)
Multiple non-instrumented indirect calls reported to our <a href="#">Mitigation Bypass Bounty</a>	Mitigated in latest version of Edge on Windows 10
Calling sensitive APIs out of context	NtContinue/longjmp – mitigated for all CFG enabled apps on Windows 10
	VirtualProtect/VirtualAlloc – mitigated in latest version of Microsoft Edge on Windows 10
	LoadLibrary – mitigated in latest version of Microsoft Edge on Windows 10 via CIG
	WinExec – mitigated in Edge on Windows 10 anniversary edition via child process policy
	All exports – mitigated in Edge on Windows 10 Creators Update via export suppression
Corrupting mutable read-only memory	Known limitation that we are exploring solutions for



# What about return addresses?

With CFG in place, we have observed attackers shifting to target return addresses

Protecting return addresses is a decades old (and very difficult) problem

- ✓ Stack cookies (/GS, etc)
  - Only mitigates stack buffer overruns
- ✓ Shadow/split stack solutions
  - E.g. SafeStack (<http://clang.llvm.org/docs/SafeStack.html>)
- ✓ Fine-grained CFI solutions
  - E.g. RAP ([https://www.grsecurity.net/rap\\_announce.php](https://www.grsecurity.net/rap_announce.php))

Why have none of the generic solutions achieved mainstream adoption yet?

Because software engineering is hard ☺

Protecting return addresses on Windows means satisfying many engineering requirements

<b>Security</b>	Must be robust against our threat model
<b>Performance</b>	Cost must be within reason and proportional to value
<b>Compatibility</b>	Don't break legacy apps
<b>Interoperability</b>	Binary compatibility with previous <i>and</i> future versions of Windows
<b>ABI compliant</b>	We can't rebuild the world
<b>Agility</b>	Don't paint ourselves into a corner
<b>Developer friction</b>	Cost for developers to enable should be minimal (ideally zero)

...and satisfying all of these requirements is *very* hard ☺

# Return address protection

We have worked with Intel on designing a hardware-assisted solution for return address protection

Our red team also came up with a clever solution that works on existing x86\_64 hardware

- ✓ Control-flow Enforcement Technology (CET)
  - Indirect branch tracking via ENDBRANCH
  - Return address protection via a shadow stack
- ✓ Hardware-assists for helping to mitigate control-flow hijacking & ROP
- ✓ Robust against our threat model

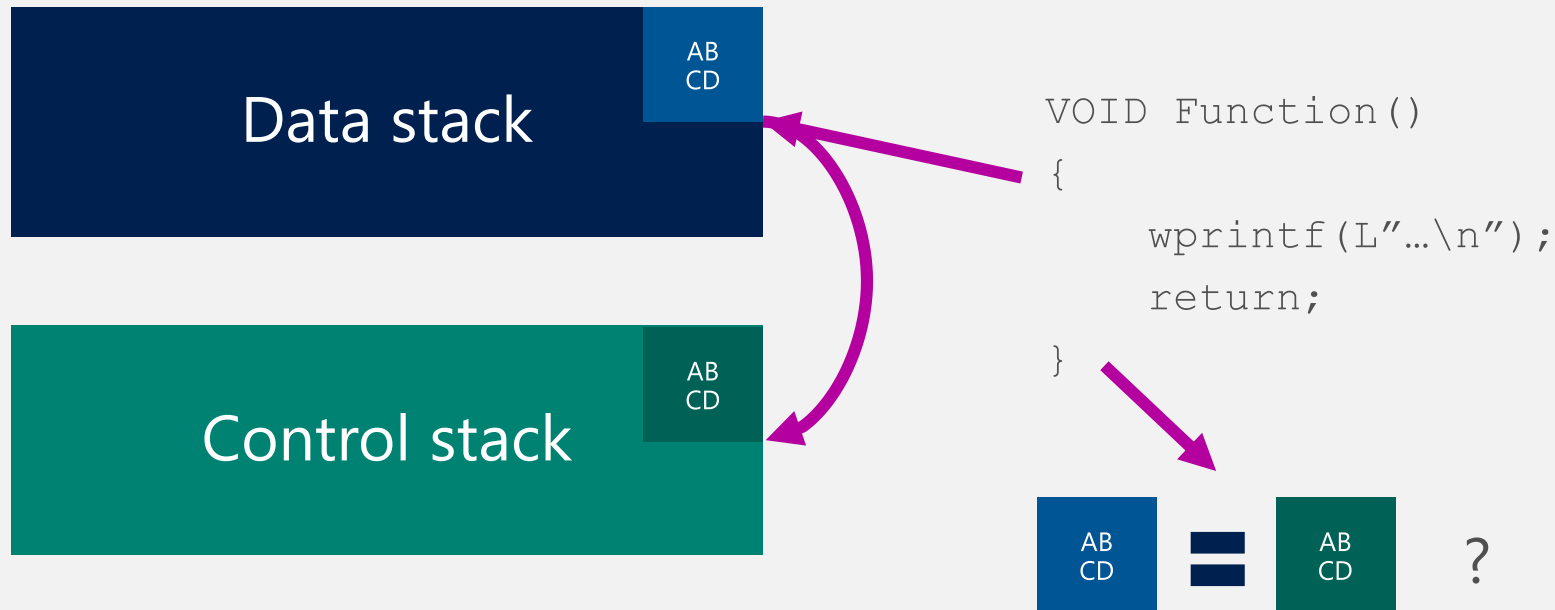
Preview specification:

<https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>

Return Flow Guard  
(RFG)

# Return Flow Guard (RFG)

RFG is an ABI-compliant shadow stack design that leverages x86\_64 segmentation to hide the shadow stack

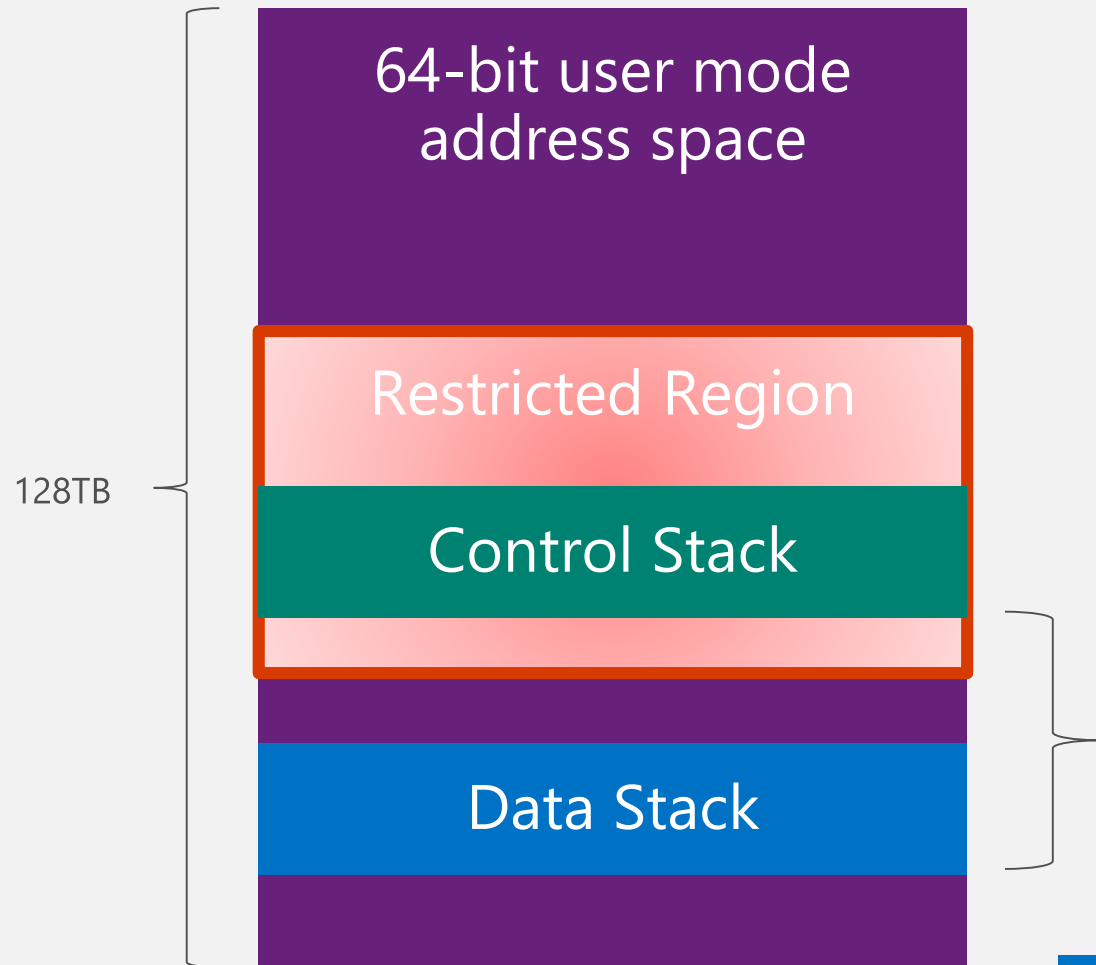


On function entry, a copy of the return address is stored in the control stack

On function exit, the return address in both stacks is compared

If they mismatch, terminate the process

# RFG key design concepts



Per-function instrumentation is materialized at page-in

Prologue (9 bytes)

488b0424	mov	rax,qword ptr [rsp]
6448890424	mov	qword ptr fs:[rsp],rax

Epilogue (15 bytes)

644c8b1c24	mov	r11,qword ptr fs:[rsp]
4c3b1c24	cmp	r11,qword ptr [rsp]
0f8518a00300	jne	DLL!__guard_ss_verify_failure

FS segment base set to displacement between control & data stack

Thread->UserFsBase = ControlStack.BaseAddress - DataStack.BaseAddress

Context  
switch

```
mov    eax, ThUserFsBase[rsi] ; load the botton of the control stack delta FS base address
mov    edx, ThUserFsBase + 4[rsi] ; load the top of the control stack delta FS base address
mov    ecx, MSR_FS_BASE      ; get the FS base MSR number
wrmsr                                ; write the control stack delta FS base address
```

No pointers to a control stack exist in user mode memory

# RFG appeared to meet all of our requirements

✖	<b>Security</b>	Must be robust against our threat model
✓	<b>Performance</b>	Cost must be within reason and proportional to value
✓	<b>Compatibility</b>	Don't break legacy apps
✓	<b>Interoperability</b>	Binary compatibility with previous <i>and</i> future versions of Windows
✓	<b>ABI compliant</b>	We can't rebuild the world
✓	<b>Agility</b>	Don't paint ourselves into a corner
✓	<b>Developer friction</b>	Cost for developers to enable should be minimal (ideally zero)

But there was just one big problem...

**Our red team found a critical weakness**

# Using real vulnerabilities to evaluate RFG

## Array.prototype.slice() Type Confusion

Takes a JavaScriptArray as input, returns a newly created JavaScriptArray

```
var fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];  
var citrus = fruits.slice(1, 3);  
  
// fruits contains ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango']  
// citrus contains ['Orange', 'Lemon']
```

MS16-084	Scripting Engine Memory Corruption Vulnerability	CVE-2016-3260	Jordan Rabet of Microsoft Offensive Security Research Team
----------	---	---------------	--

<https://github.com/Microsoft/ChakraCore/commit/17f3d4a4852dcc9e48de7091685b1862afb9f307>

# Exploit Sequence

- Create a NativeFloatArray fake\_obj\_arr which contains a fake object
- Use CVE-2016-3260 to leak its address
- Use CVE-2016-3260 to create a JavaScript variable fake\_obj pointing to fake\_obj\_arr + data\_offset
- ...and a few other steps we won't be talking about today 😊

```
// build a fake dataview object
// could do this with a typedbuffer but would need to have the vtable address probably
fake_object_arr[0] = hex2float(0) // vtable
fake_object_arr[1] = hex2float(fake_type_addr) // type
fake_object_arr[2] = hex2float(0) // auxSlots
fake_object_arr[3] = hex2float(0) // objectArray / flags
fake_object_arr[4] = hex2float(0xffffffff) // length
fake_object_arr[5] = hex2float(locate_object(real_ab)) // arrayBuffer
fake_object_arr[6] = hex2float(0) // byteOffset
fake_object_arr[7] = hex2float(0xdeadbabedad) // buffer pointer
// create fake type for dataview
fake_object_arr[fake_type_offset] = hex2float(56) // TypeIds_DataView
fake_object_arr[fake_type_offset + 1] = hex2float(fake_object_start_addr) // javascriptLibrary : needs to be valid ptr

var fake_dv = create_offsetted_object(fake_object_arr, arr_inlineslots_offset)
```

Exploit was used as a platform to confirm a design-level bypass for RFG

# Sticking to our threat model

## The attack against RFG violated the assumptions of our threat model

- ✓ Accepting a weaker threat model would mean significant drop in ROI
- ✓ Long-term impact on cost to exploit would not be high enough to warrant long-term cost of feature
- ✓ Red team was key to vetting the solution
- ✓ @zer0mem also reported some interesting attacks 😊

## RFG remains a research project; CET not affected

- ✓ Reminder: Microsoft has an ongoing \$100,000 USD bounty for defensive ideas 😊
- ✓ <https://technet.microsoft.com/en-us/security/dn425049.aspx>
- ✓ <https://blogs.technet.microsoft.com/srd/2015/09/08/what-makes-a-good-microsoft-defense-bounty-submission/>

**The Creators Update of Windows 10 supports ACG, CIG, and CFG – RFG is not supported**





# Enabling code execution mitigations

Windows 10 provides multiple ways to enable mitigations for an app

Opt-in method	Details	Precedence
Runtime API	See <code>SetProcessMitigationPolicy</code>	4
EXE flag	Mitigation-specific flag in PE headers	3
Process creation attribute	See <code>UpdateProcThreadAttribute</code> with <code>PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY</code>	2
Image File Execution Option (IFEO)	<code>Set MitigationOptions (REG_BINARY)</code> IFEO for an EXE  Same bits as process creation attribute	1

Mitigation	Relevant policies	Runtime API	EXE bit	Process creation attribute	IFEO
CIG	<code>ProcessSignaturePolicy</code>	Yes	No	Yes	Yes
ACG	<code>ProcessDynamicCodePolicy</code>	Yes	No	Yes	Yes
CFG	<code>ProcessControlFlowGuardPolicy</code>	No	Yes (/guard:cf)	Yes	Yes
No Child Process	<code>CHILD_PROCESS_POLICY</code> process creation attribute	No	No	Yes	No



# Protected apps in Windows 10 Creators Update

	 Microsoft Edge	 Hyper-V VMWP
ACG	✓	✓
CIG	✓	✓
CFG	✓	✓
No Child Process	✓	✓

- For Microsoft Edge, ACG and CIG are disabled in the presence of certain 3<sup>rd</sup> party extensions (e.g. IMEs, legacy graphics drivers). ACG is currently enabled only on 64-bit.
- For Hyper-V VMWP, the no child process mitigation is disabled when certain features are used (vTPM)

# What about the Windows kernel?

Windows 10 leverages Hyper-V Virtualization Based Security (VBS) to enable CI, ACG, and CFG for the Windows kernel

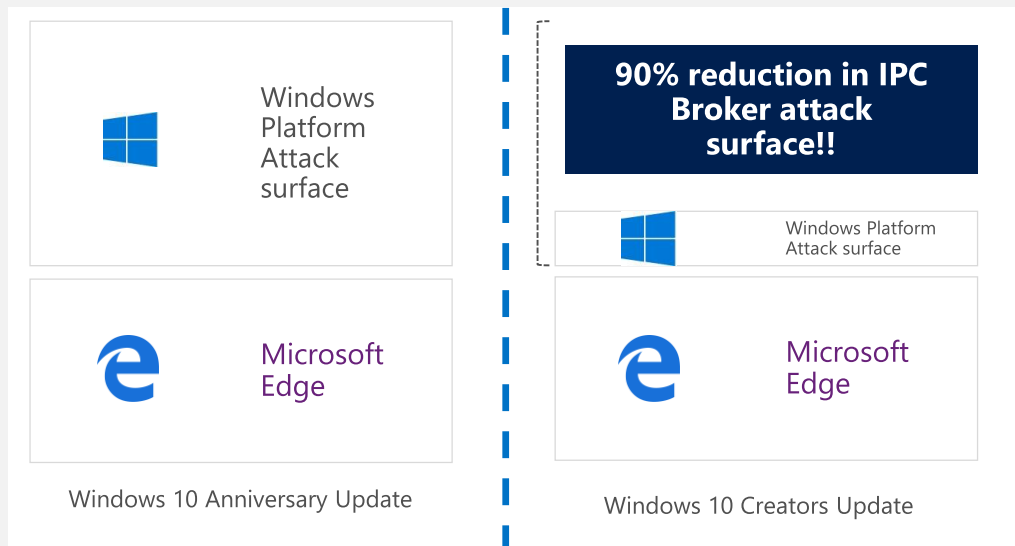
Kernel Mode		 Windows 10 Creators Update
	ACG	 Enabled with Hypervisor-Enforced Code Integrity (HVCI) via Device Guard
	CIG	
	CFG	

# Conclusion & next steps

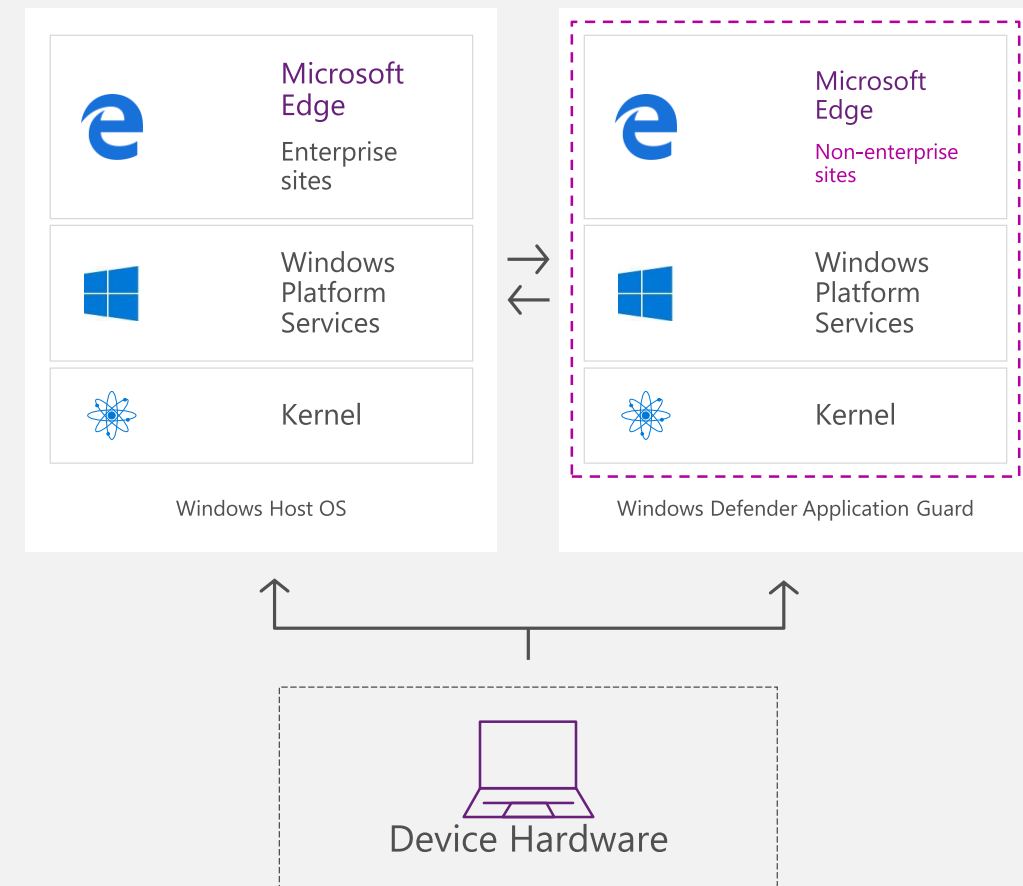
# It's not RCE mitigations *or* sandboxing – it's both

We are continuing to invest in improved isolation technologies as part of our layered strategy

## Improved software isolation (Microsoft Edge AppContainer Profile)



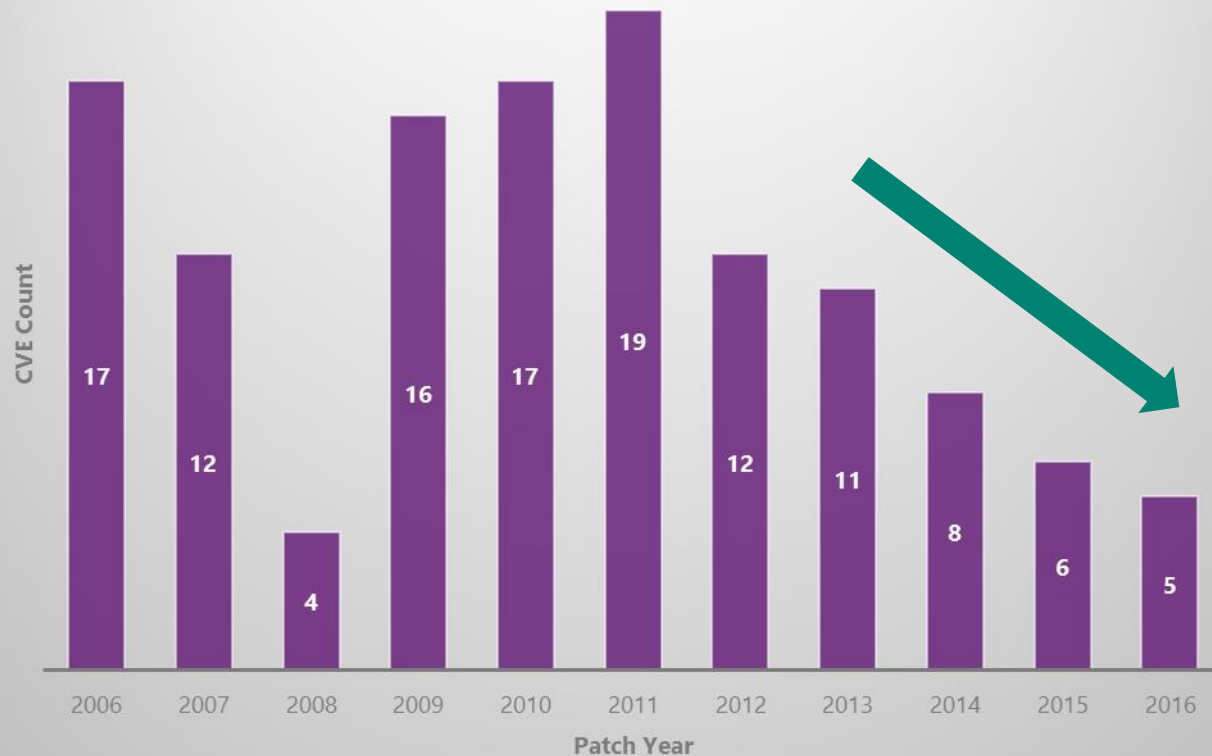
## Virtualized Isolation (Application Guard)



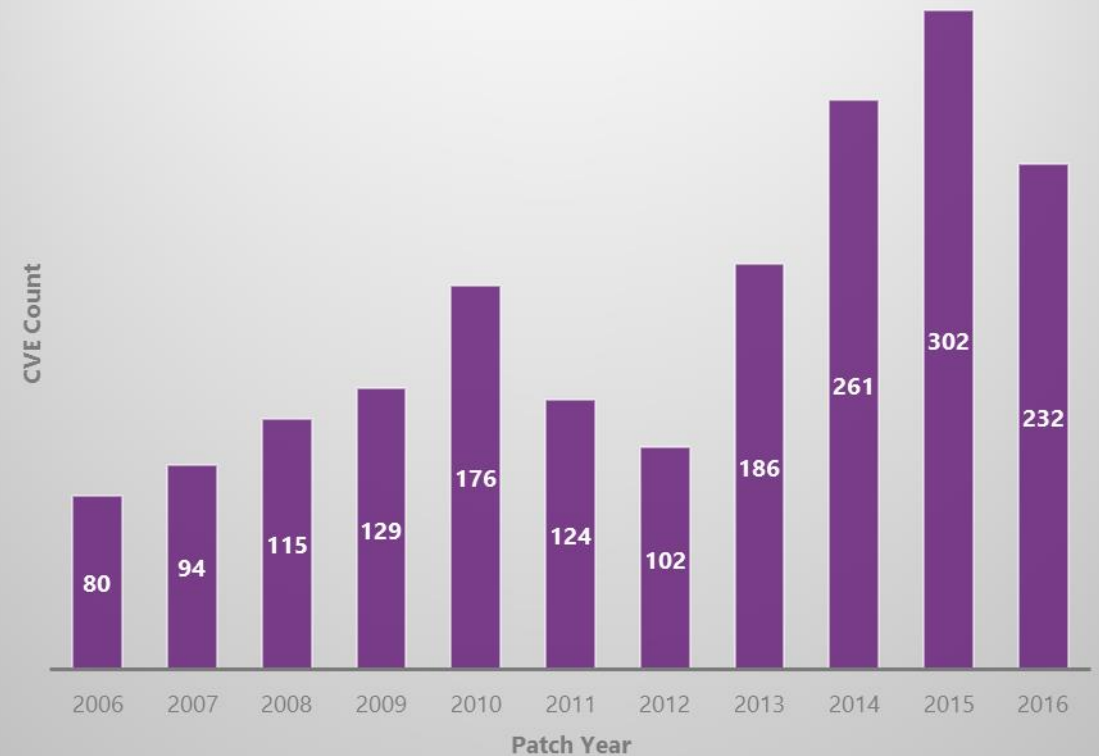
# Our work is having a measurable impact

# of known zero day RCE exploits has declined despite increase in known vulnerabilities

**# of Microsoft Remote Code Execution CVEs exploited as zero day by year patched**



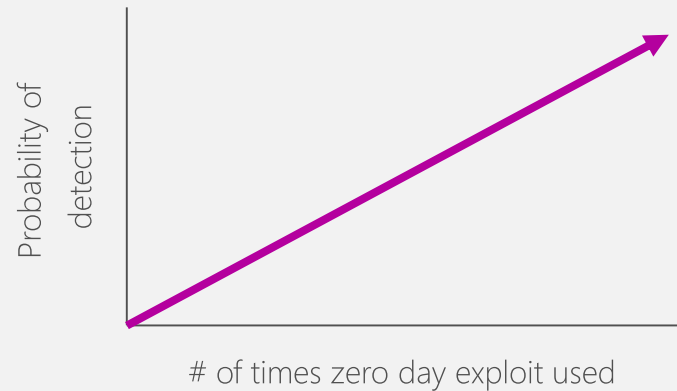
**# of Microsoft Remote Code Execution CVEs by year patched**



# What about the zero days we don't know about?

We cannot effectively measure the number of unknown zero day exploits, but we don't really need to

Hypothesis: increased exploit costs drive selective use



- ✓ Probability of detection increases with zero day use
  - Attackers are incentivized to minimize use
  - Targets that detect zero day may alert vendor
- ✓ Selective use reduces downstream supply
  - Many actors lack means and capability to acquire

Assertion: economics of the zero day market have shifted

- ✓ Windows 10 is always up to date
  - Poor ROI for exploiting patched vulnerabilities
  - Rapid evolution of defensive technologies
- ✓ Mass-market exploit kits struggling to maintain supply
  - Decrease in reusable & public exploits
  - Cost to acquire exceeds expected ROI
- ✓ Market has shifted toward social engineering
  - Macros, tech support scams, so on

**MAGNITUDE ACTOR ADDS A SOCIAL ENGINEERING SCHEME FOR WINDOWS 10**

MARCH 08, 2017 Kafeine <https://www.proofpoint.com/us/threat-insight/post/magnitude-actor-social-engineering-scheme-windows-10>

# Lessons learned

**3<sup>rd</sup> party binary extensions are security's nemesis**

- Very difficult to maintain compatibility and interoperability with 3<sup>rd</sup> party binary extensions
- Apps need to tightly control their binary extension points to maintain security agility

**We're not done yet with mitigating arbitrary native code execution**

- CET should provide robust protection for return addresses once available

**Layered, data-driven, and red-team-assisted defense is key**

- No silver bullets exist in security
- Focus investments on defensible positions



# What's next?

We believe our strategy is working & are continuing to execute on it

Kill more bug  
classes

Drive down  
attack surface

Break more  
exploitation  
techniques

Improve least  
privilege  
containment

Lots of exciting things in store. Stay tuned to the WIP Fast Ring!

Fascinated by what you saw? Want to help us make the online world safer?



[aka.ms/bugbounty](https://aka.ms/bugbounty)

Report vulnerabilities &  
mitigation bypasses via our  
bounty programs!

<https://aka.ms/bugbounty>

Or come work with us. We're hiring 😊

<https://aka.ms/cesecurityopenjobs>

<https://aka.ms/wdgsecurityjobs>

# Acknowledgements

Tons of people worked hard to help make this possible!

Thanks to the following groups & teams for their contributions:

- Windows kernel and Hyper-V teams
- Visual C++ team
- Microsoft Edge security team
- WDG Offensive Security Research team
- Microsoft Security Response Center
- And everyone else who contributed!

