



Modern Exploit Mitigations

Swiss Cyber Storm 2017, 18.10.2017

Matthias Ganz & Antonio Barresi
xorlab AG



About



Antonio Hüseyin Barresi
antonio.barresi@xorlab.com
@AntonioHBarresi



Matthias Ganz
matthias.ganz@xorlab.com
@GanzMatthias



Swiss IT Security startup providing IT Security solutions to defend against cyber attacks.

Outline

- > History and overview of mitigations
- > Modern mitigations
 - > EMET, ROP mitigations
 - > Control-Flow Guard and Return-Flow Guard
 - > Intel Control-flow Enforcement Technology
- > Conclusion

Memory corruption & exploitation

How old is the problem?



Morris Worm

The Morris Internet Worm source code

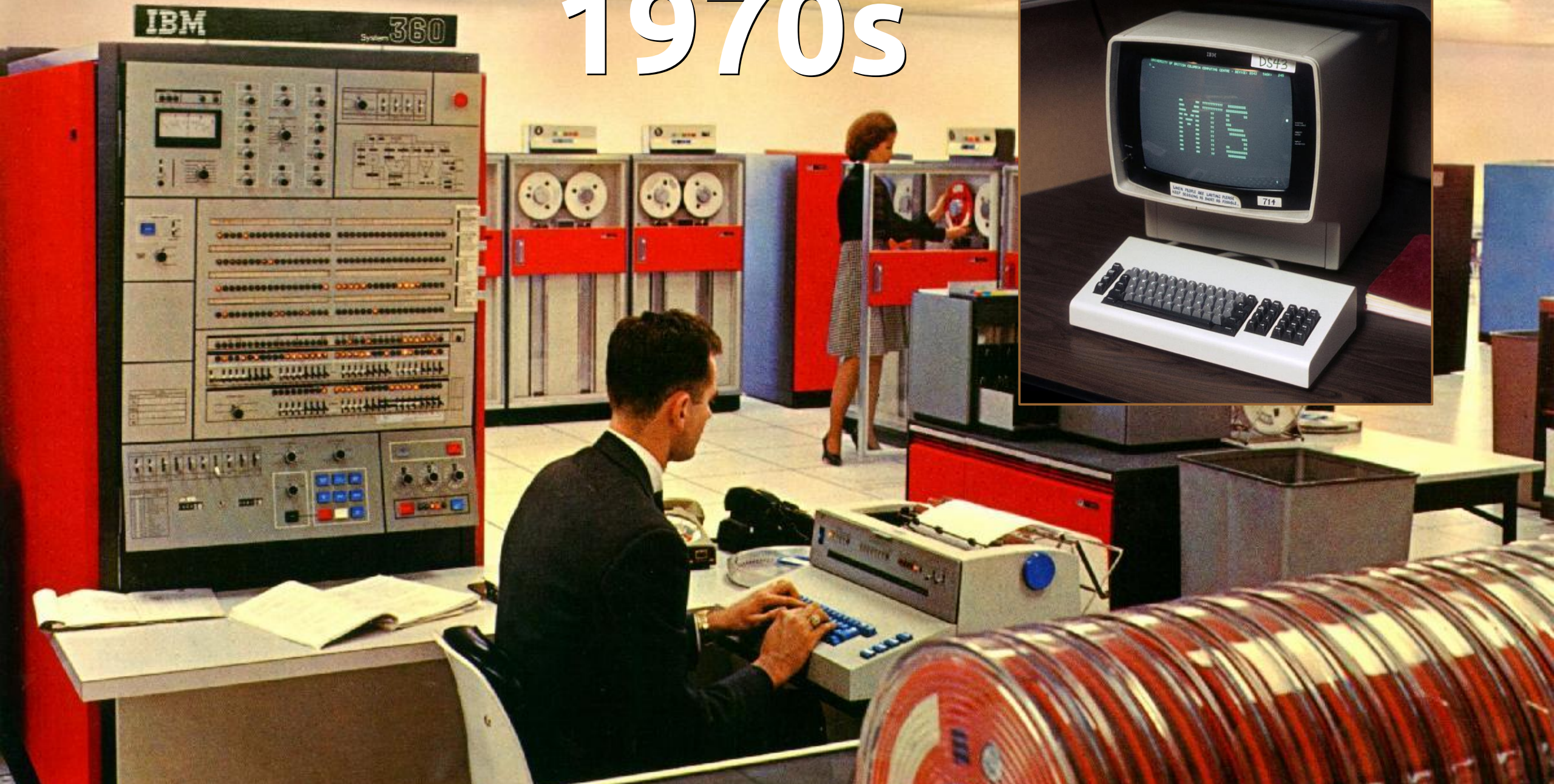
This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.

The worm was the first of many intrusive programs that use the Internet to spread.



Internet Worm -
Source code
X1294.96 A-D

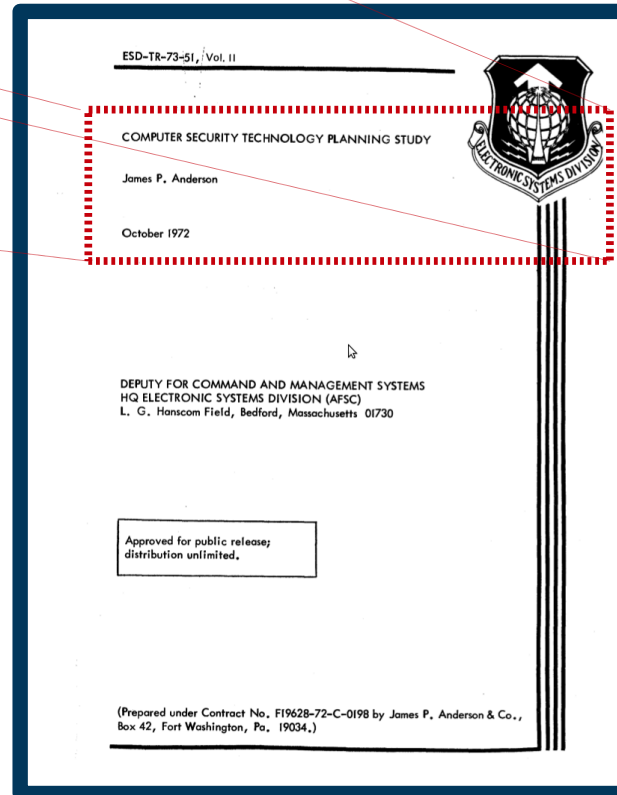
1970s



Long long ago - “The Anderson Report”



- October 1972
- Time-sharing
- ARPANET 1969
- C 1972, UNIX 1973



1.1 Background

In recent years the Air Force has become increasingly aware of the problem of computer security. This problem has intruded upon virtually every aspect of USAF operations and administration. The problem arises from a combination of factors that includes: greater reliance on the computer as a data processing and decision making tool in sensitive functional areas; the need to realize economies by consolidating ADP resources thereby integrating or co-locating previously separate data processing operations; the emergence of complex resource sharing computer systems providing users with capabilities for sharing data and processes with other users; the extension of resource sharing concepts to networks of computers; and the slowly growing recognition of security inadequacies of currently available computer systems.



That's cloud computing!

1.5.2 Study Tasks

Specific tasks called for within this scope included:

- a. A study and analysis of the security penetration threats and techniques as well as the effectiveness of current technology in meeting these threats, and the extent of research and development required to improve the current computer security technology.

Offensive research!

Mr. James P. Anderson,
Deputy Chairman

Dr. Melvin Conway

Mr. Daniel J. Edwards (NSA)

Miss Hilda Faust (NSA)

Mr. Steven Lipner (MITRE)
(Chairman, Requirements
Working Group)

Dr. Eldred Nelson (TRW)

Mr. Bruce Peters (SDC)*

Dr. Charles Rose
(Case Western Reserve)

Mr. Clark Weissman (SDC)



The major vulnerability to be guarded against in HOL-only systems is the possibility that the user (programmer) of the system may escape from the higher order language to enter or execute arbitrary machine code of his choice, and defeat or bypass the run-time package.

In the discussion to follow, we refer to FORTRAN because it is a common language and serves the purposes of illustration. The primary technical problem is whether FORTRAN user can break out of the FORTRAN envelope into data areas and be able to execute arbitrary instructions planted in the program as data.

In order to break out of the FORTRAN envelope it is necessary to execute references outside of those defined by the FORTRAN program. These would include references beyond the upper limit of the program, branching to an unlabeled area, or being able to write beyond the defined area. The ability to write beyond the defined area is the ability to break out of the FORTRAN confinement.

Considering these problems, the secure higher order language must:

- There are no constants.
- All references are to locations within the program.
- All transfers of control lie within the program.
- All input-output is that authorized by the program.

Exploit mitigation recommendations in the 70s

Arbitrary

Execution

data

Data separation / DEP

Data pointer integrity

DEP / Control-Flow Integrity

Bounds checking

<http://csrc.nist.gov/publications/history/ande72.pdf>

Pages 36 & 37 / "6.2.2 Security Requirements for HOL-only Systems / HOL: Higher Order Language"

45 years later?

Pwn2own 2017

\$ 833'000 awarded
51 zero-day bugs found

Exploited

- VMWare
- Microsoft Edge
- Adobe Flash
- Safari
- MacOS
- ...

The third and final day of the largest Pwn2Own shapes up with three entries and the awarding of Master of Pwn. It's a tight race with multiple teams still in the running. Here's the schedule for Day Three:

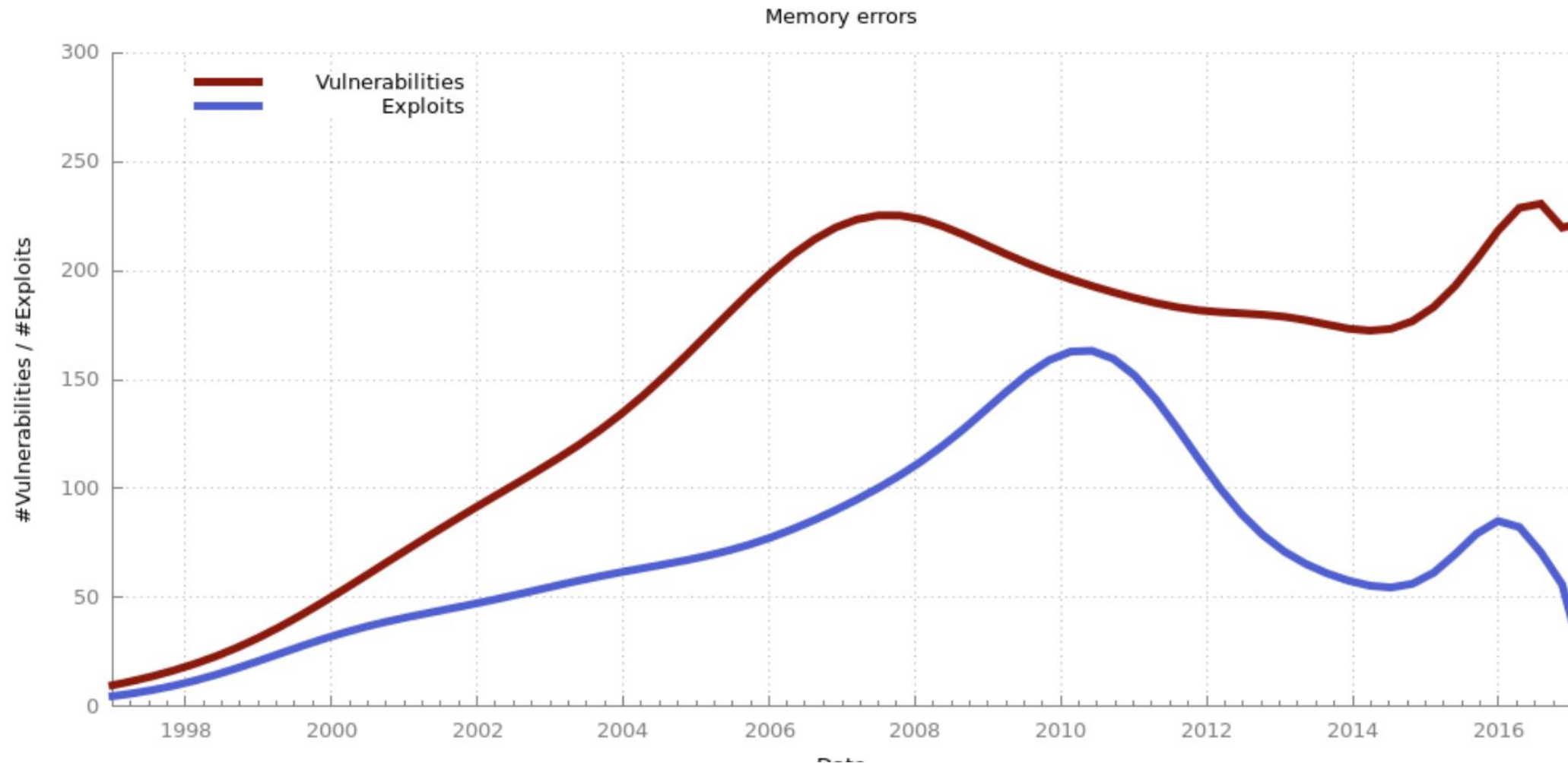
9:00am – 360 Security (@mj011sec) targeting Microsoft Edge with a SYSTEM-level escalation and a virtual machine escape

SUCCESS: The 360 Security (@mj011sec) team used a used heap overflow in Microsoft Edge, a type confusion bug in the Windows kernel, and an uninitialized buffer in VMware for a complete virtual machine escape. They more than earn \$105,000 and 27 Master of Pwn points.

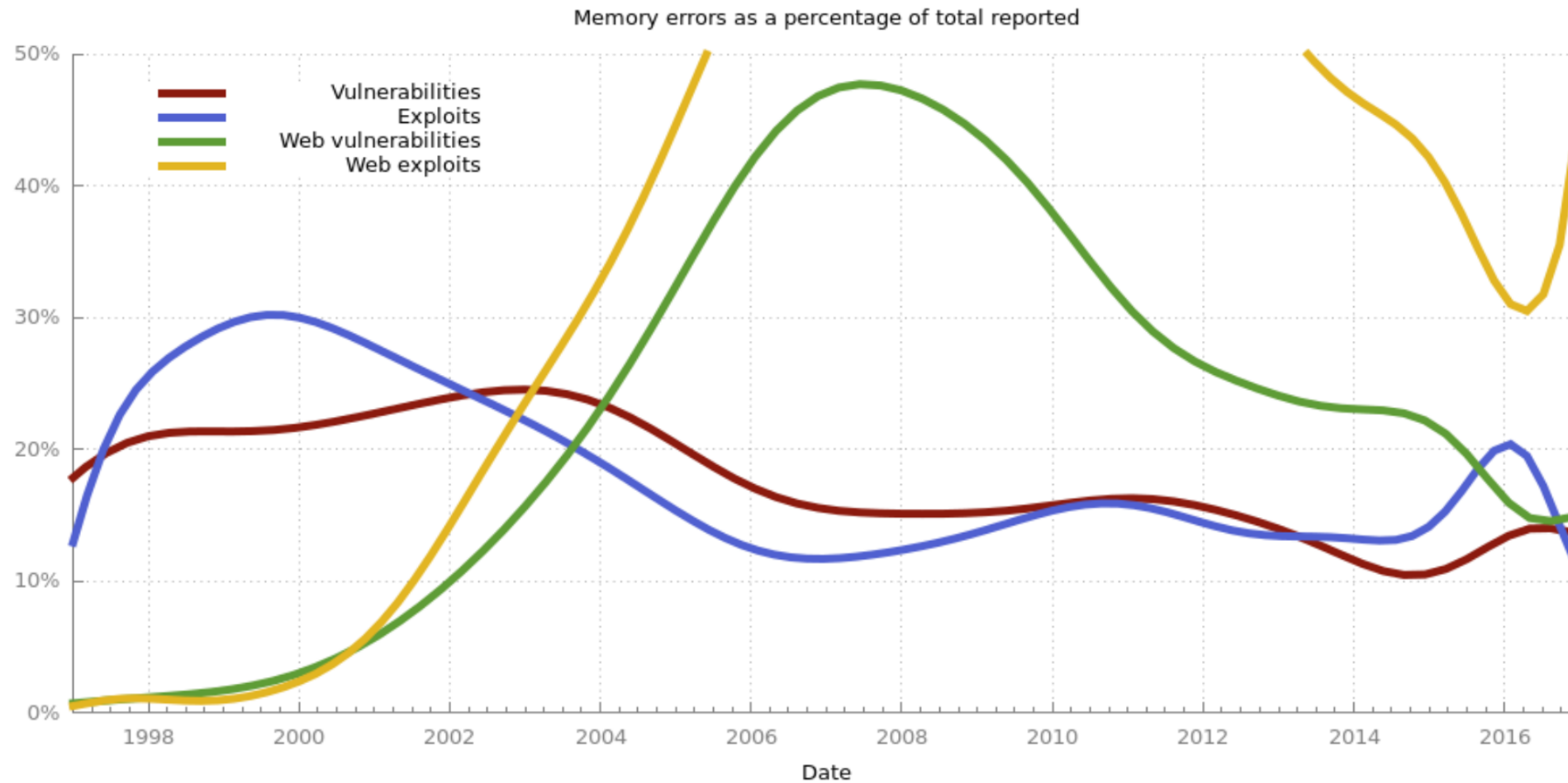
Next up, Samuel Groß and Niklas Baumstark earned some style points by leaving a special message on the touch bar of the targeted Mac. They employed a use-after-free (UAF) in Safari combined with three logic bugs and a null pointer dereference to exploit Safari and elevate their privileges to root in macOS. Unfortunately, the UAF had already been corrected in the beta version of the browser, but this bug chain still netted them a partial win, garnering them \$28,000 and 9 Master of Pwn points.

SUCCESS: Tencent Security – Team Sniper (Keen Lab and PC Mgr) successfully exploits Adobe Flash via a UAF and escalates to SYSTEM with a UAF in the Windows kernel. This earned them \$40,000 and 12 points for Master of Pwn.

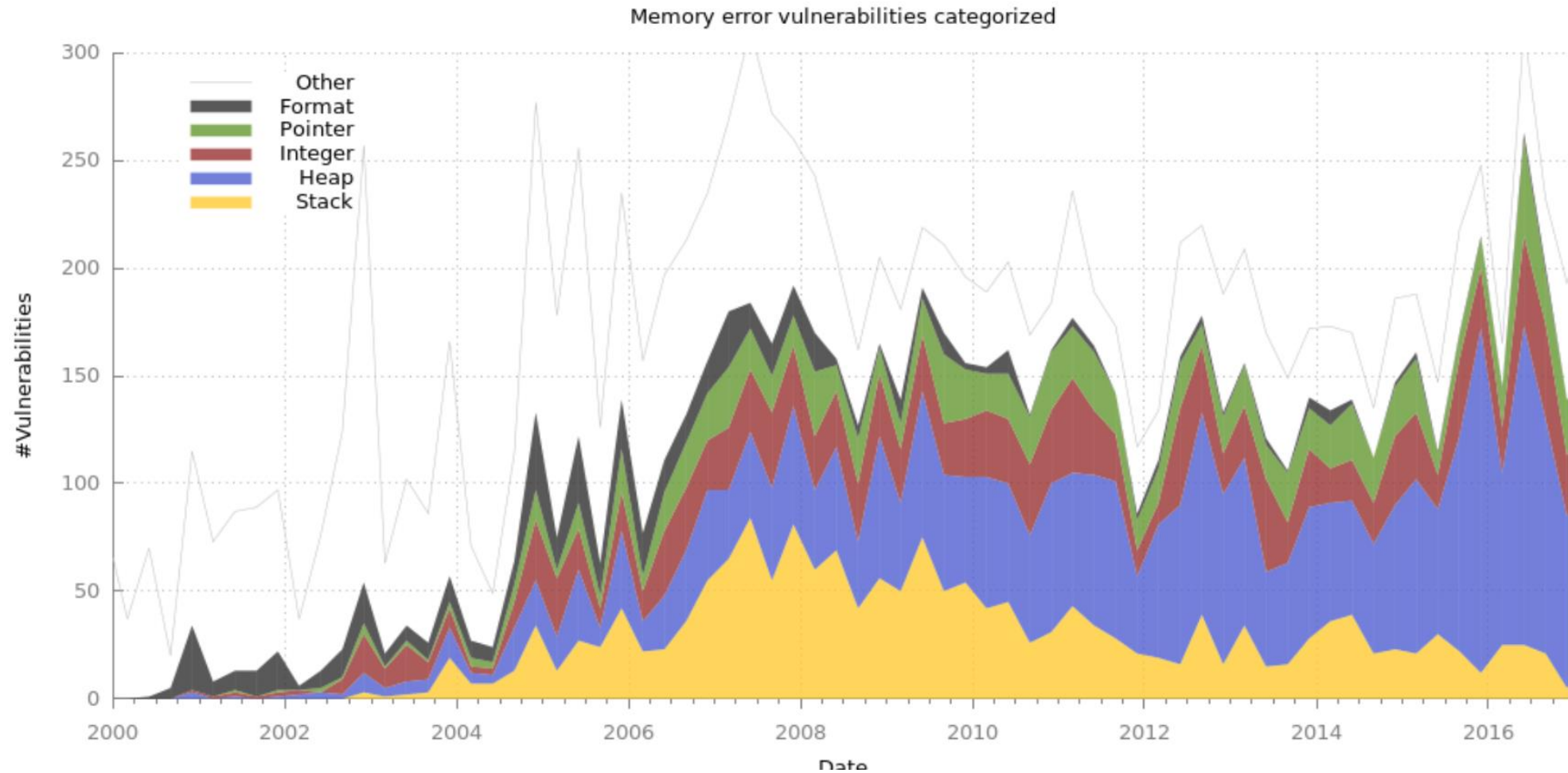
Memory error vulnerabilities



Memory error vulnerabilities



Memory error vulnerabilities

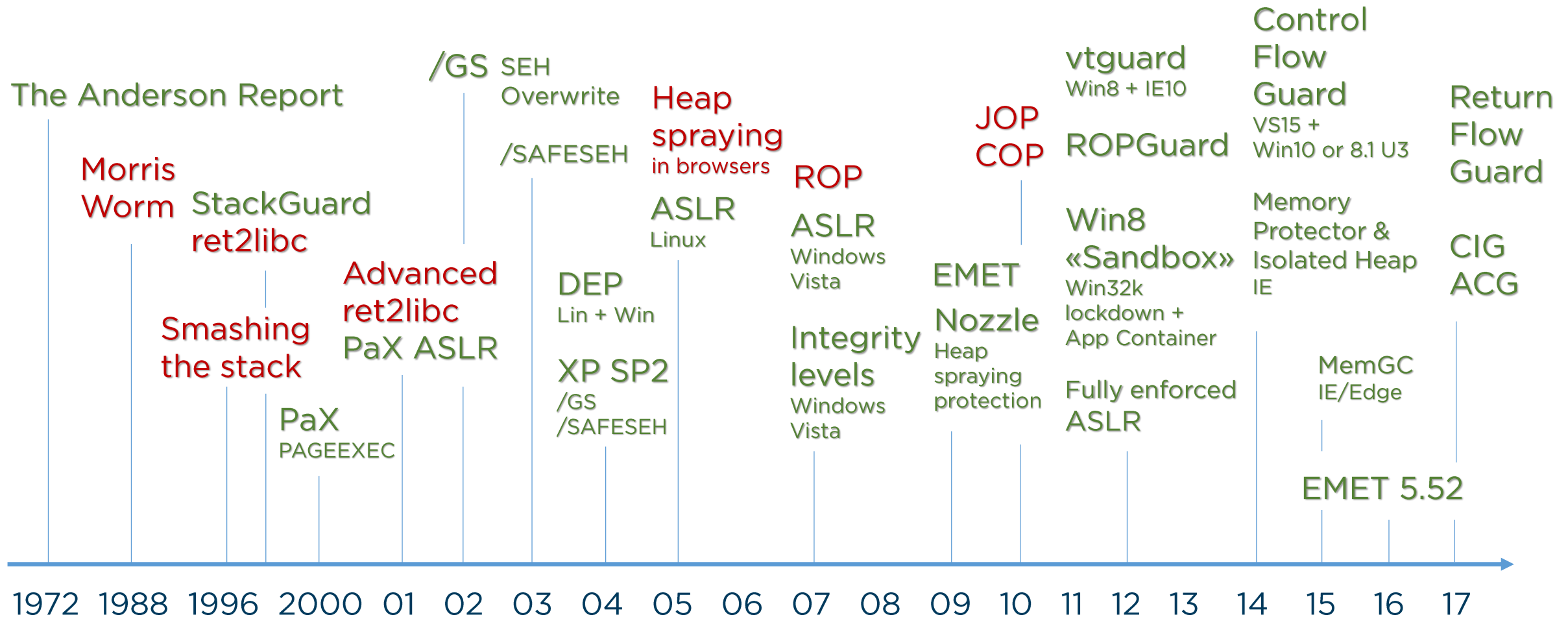


We can't eliminate all bugs...
so what should we do?

Eliminate vulnerability classes
or exploitation techniques...

...or at least make exploitation more difficult!

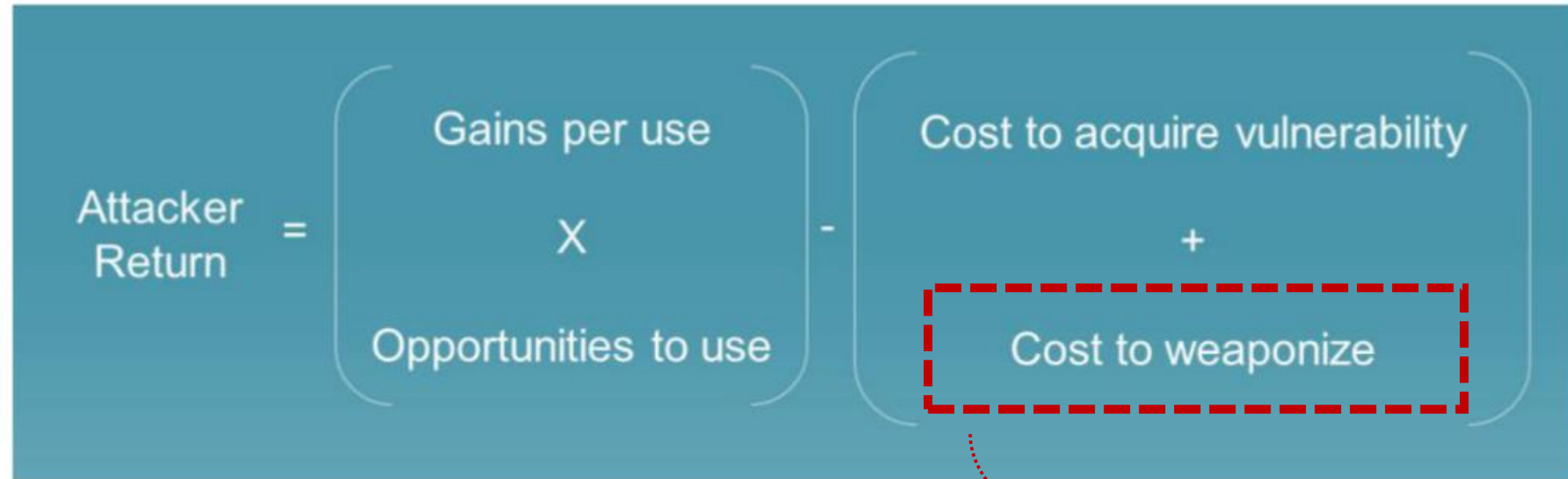
Exploit mitigations since the 90s



Attacker Return

$$\text{Attacker Return} = \left(\begin{array}{c} \text{Gains per use} \\ \times \\ \text{Opportunities to use} \end{array} \right) - \left(\begin{array}{c} \text{Cost to acquire vulnerability} \\ + \\ \text{Cost to weaponize} \end{array} \right)$$

Attacker Return

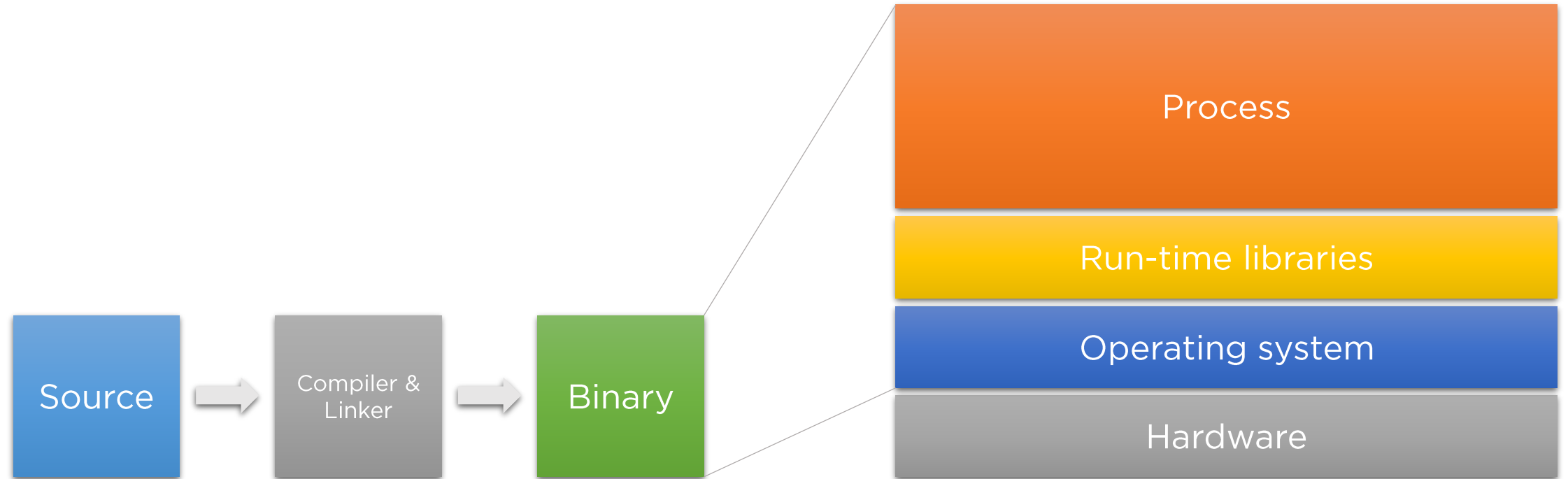


The diagram illustrates the formula for Attacker Return. It is presented on a teal background. On the left, the text "Attacker Return" is followed by an equals sign. To the right of the equals sign is a large white bracket containing the expression "Gains per use" multiplied by "Opportunities to use". This is followed by a minus sign and another large white bracket. Inside this second bracket, the text "Cost to acquire vulnerability" is followed by a plus sign and the text "Cost to weaponize". The "Cost to weaponize" text is enclosed in a red dashed rectangular box. A red dotted arrow points from this box down towards the text "Increase costs" located below the diagram.

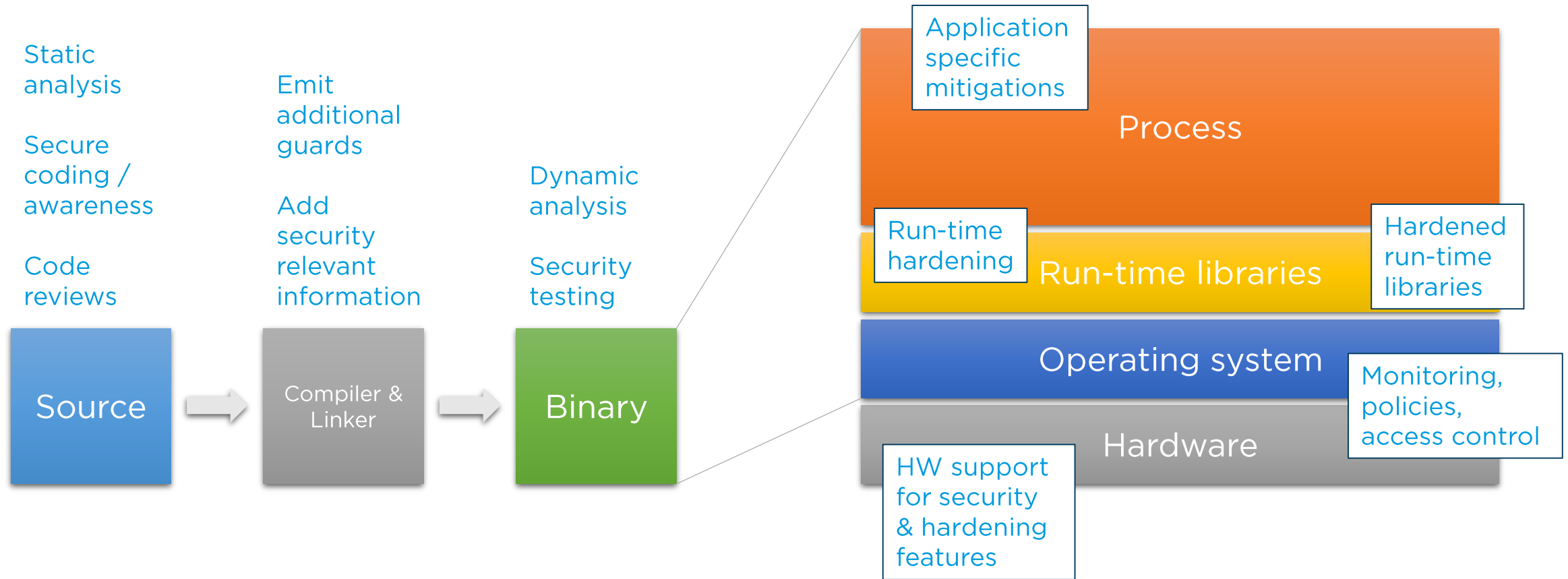
$$\text{Attacker Return} = \left(\text{Gains per use} \times \text{Opportunities to use} \right) - \left(\text{Cost to acquire vulnerability} + \text{Cost to weaponize} \right)$$

Increase costs

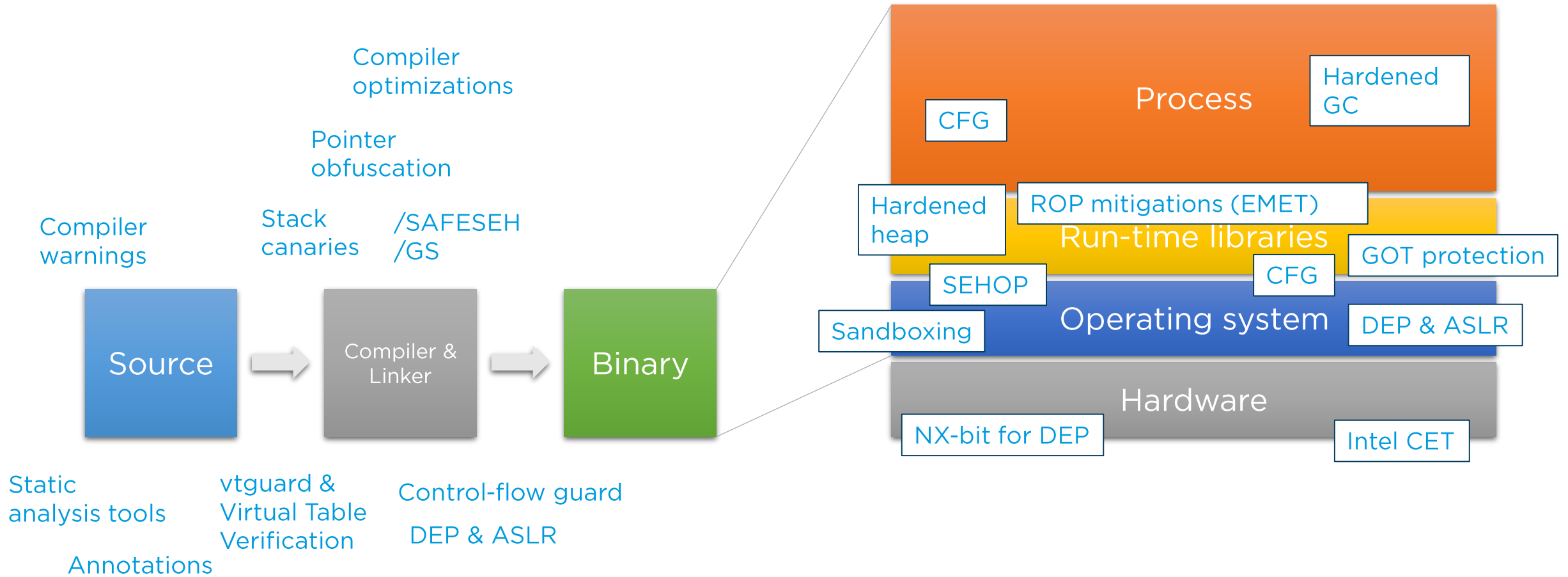
Hardening value chain



Hardening value chain



Hardening value chain



Firefox's adoption of compiler flags

2008

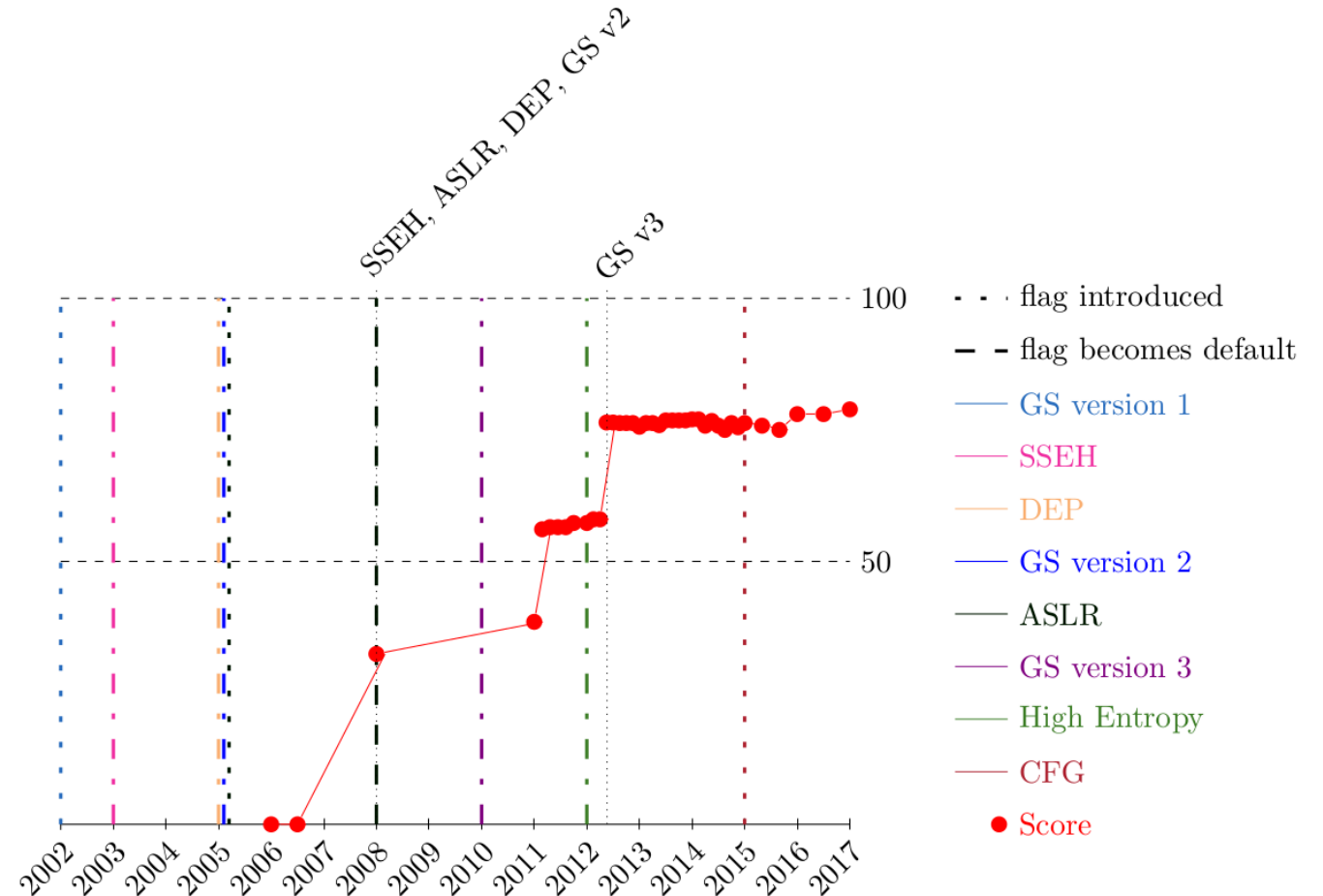
- > Visual Studio 2005
- > + DEP & ASLR partial

2011

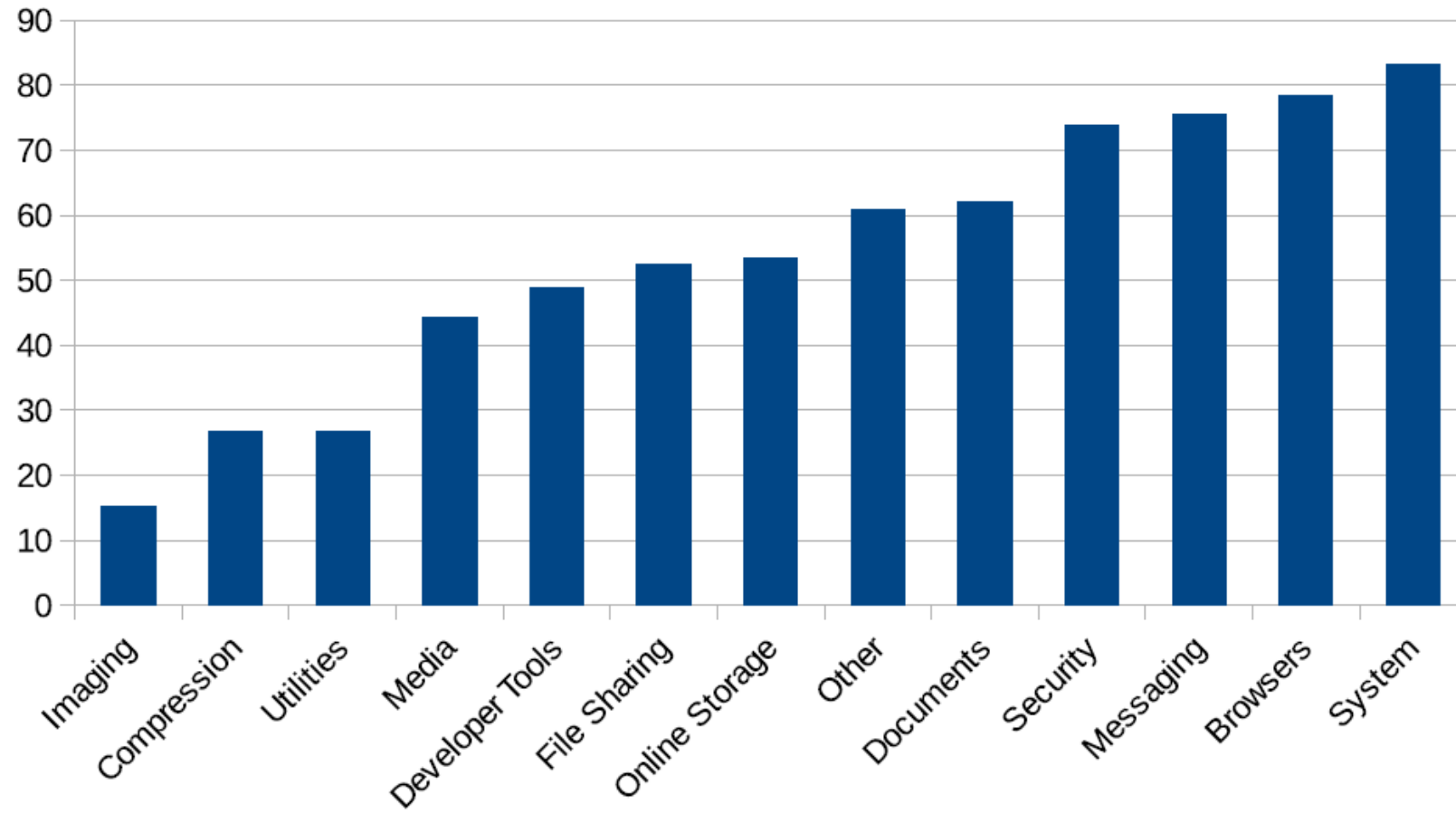
- > + DEP & ASLR full

2012

- > Visual Studio 2010
- > + GS v3



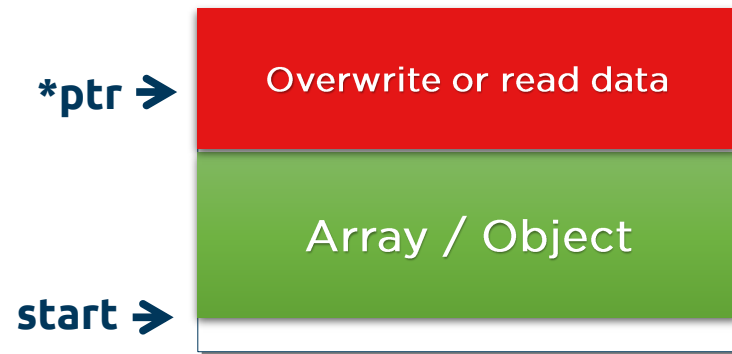
Application classes



Refresher

Types of memory errors

Spatial error



Temporal error

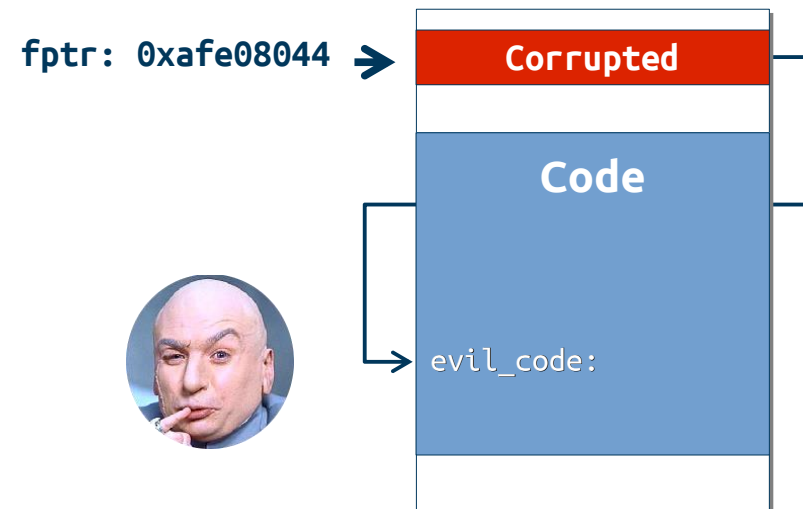
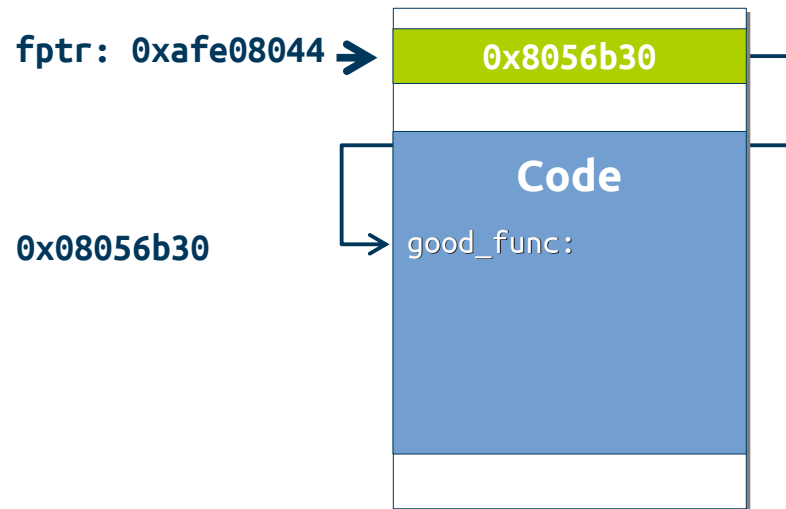


Control-flow hijack attack

Most ISAs support indirect branch instructions

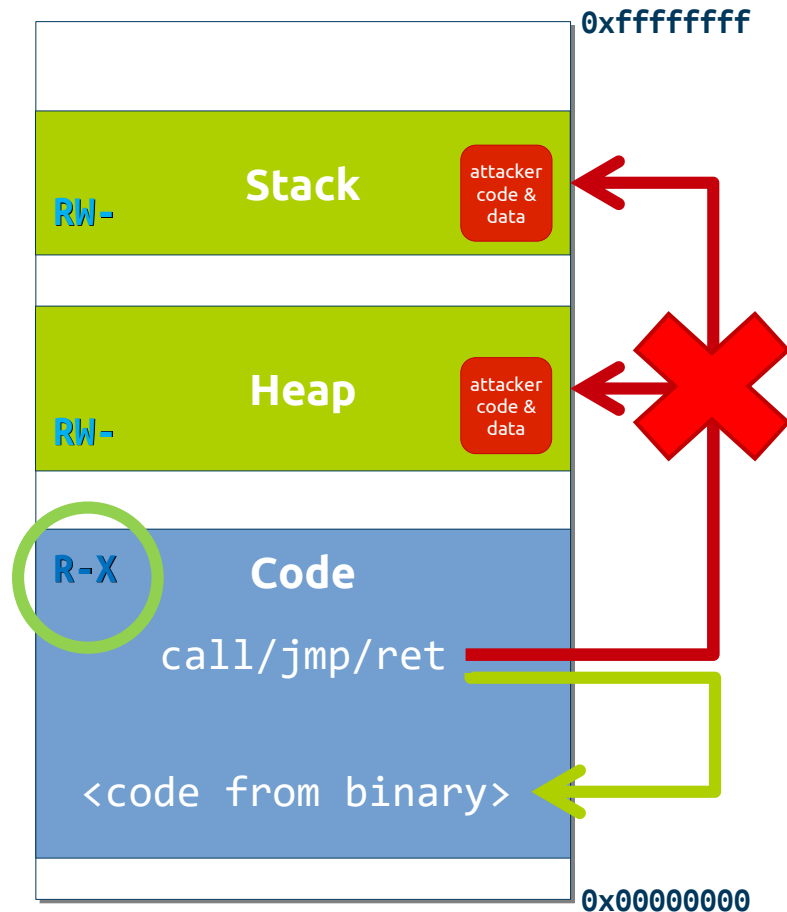
E.g., x86 **ret**, indirect **jmp**, indirect **call**

branch *fptr



DEP & ASLR

DEP (Data Execution Prevention)



Bypassing DEP

Code-reuse attack

- ret2libc, ret2bin, ret2*
- Return-oriented programming (ROP)
- Jump/Call-oriented programming

Use code-reuse technique to change protection flags

Make memory executable

- mprotect/VirtualProtect
- mmap/VirtualAlloc



ASLR (Address Space Layout Randomization)

Randomize memory layout of processes to make address prediction/guessing hard

What can be randomized?

- OS: Stack, heap and memory mapping base addresses
- OS, compiler, linker: Executables and libraries
 - Position-independent or relocatable code

Bypassing ASLR

- Low entropy
 - Brute-force addresses (multiple attempts required)
- Memory leaks (information disclosure)
 - Leak addresses to derive base addresses
 - Construct and enforce a leak by memory corruption
- Application and vulnerability specific attacks
 - Force predictable memory state
 - Heap-spraying / Heap massaging
 - Pointer inference
 - Use a side-channel
 - Avoid using exact addresses
 - Only corrupt least significant bytes i.e. offsets

Current state of attack techniques

- > Use memory error to construct primitives
...ideally a write and read primitive
- > Inject machine code and ROP chain
- > Hijack control-flow and execute ROP chain
...ROP chain will make code executable
- > Execute injected machine code... done!



What else can we do?

Post-ASLR/DEP... what else can we do?

- > Assume attacker can read and write memory
- > Don't allow run-time code generation
- > Don't allow all control-flow transfers
 - > Not every source target pair is needed

New kids on the block (Windows & Intel)

- > EMET & ROP Mitigations
- > Code Integrity Guard & Arbitrary Code Guard
- > Control-Flow Guard & Return-Flow Guard
- > Intel Control-flow Enforcement Technology (CET)

Let's look at

- > EMET & ROP Mitigations
- > Code Integrity Guard & Arbitrary Code Guard
- > Control-Flow Guard & Return-Flow Guard
- > Intel Control-flow Enforcement Technology (CET)

EMET

Enhanced Mitigation Experience Toolkit

- Released in 2009, EMET 1.x (27.10.2009)
 - 2.x 2.9.2010
 - 3.x 25.5.2012
 - 4.x 18.4.2013
 - 5.x 31.7.2014
 - 5.5 29.1.2016
 - 5.52 14.11.2016

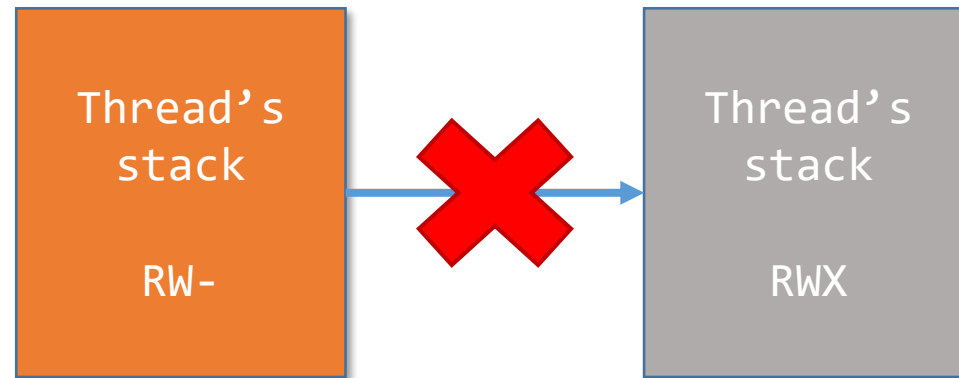
EMET

- Implements various features and hardening techniques
- Several ROP mitigations (at least for 32bit)
 - Memory Protection
 - Caller Check
 - Simulate Execution Flow
 - Stack Pivot
 - EAF, EAF+

Memory Protection checks



- Disallow making the stack area executable
- Prevents placing shellcode on stack

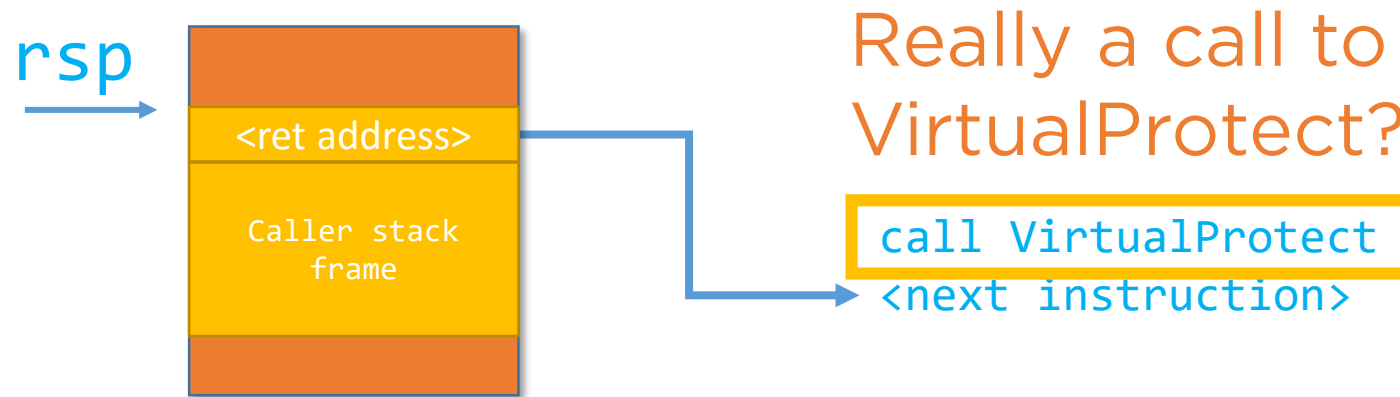


Caller Check



- During an ongoing ROP attack functions are not really called but returned to
- Check call site
 - Check the instruction before the return address on the stack
 - Is it really a call?
 - Does it call the hooked API?

Caller Check for VirtualProtect



Stack Pivot check



- Checks if the **stack pointer** is within bounds
- Detects if ESP/RSP points to the heap
- ROP chain can not be on heap when the API is called

EMET bypasses

- There has been a lot of work on bypassing EMET
- Bypass EMET 4.1
<https://labs.bromium.com/2014/02/24/bypassing-emet-4-1/>
<https://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf>
- EMET 5.1 Armor or Curtain? Rene Freingruber
ZeroNights 2014, 31C3
- Bypass EMET 5.2 hooks by jumping over them
<http://casual-scrutiny.blogspot.ch/2015/03/defeating-emet-52.html>
- Disable EMET 5.2 by calling a cleanup function reachable via emet.dll!DllMain
https://www.fireeye.com/blog/threat-research/2016/02/using_emet_to_disabl.html
- EAF disabling by clearing HW breakpoints by Piotr Bania
http://piotrbania.com/all/articles/anti_emet_eaf.txt

EMET

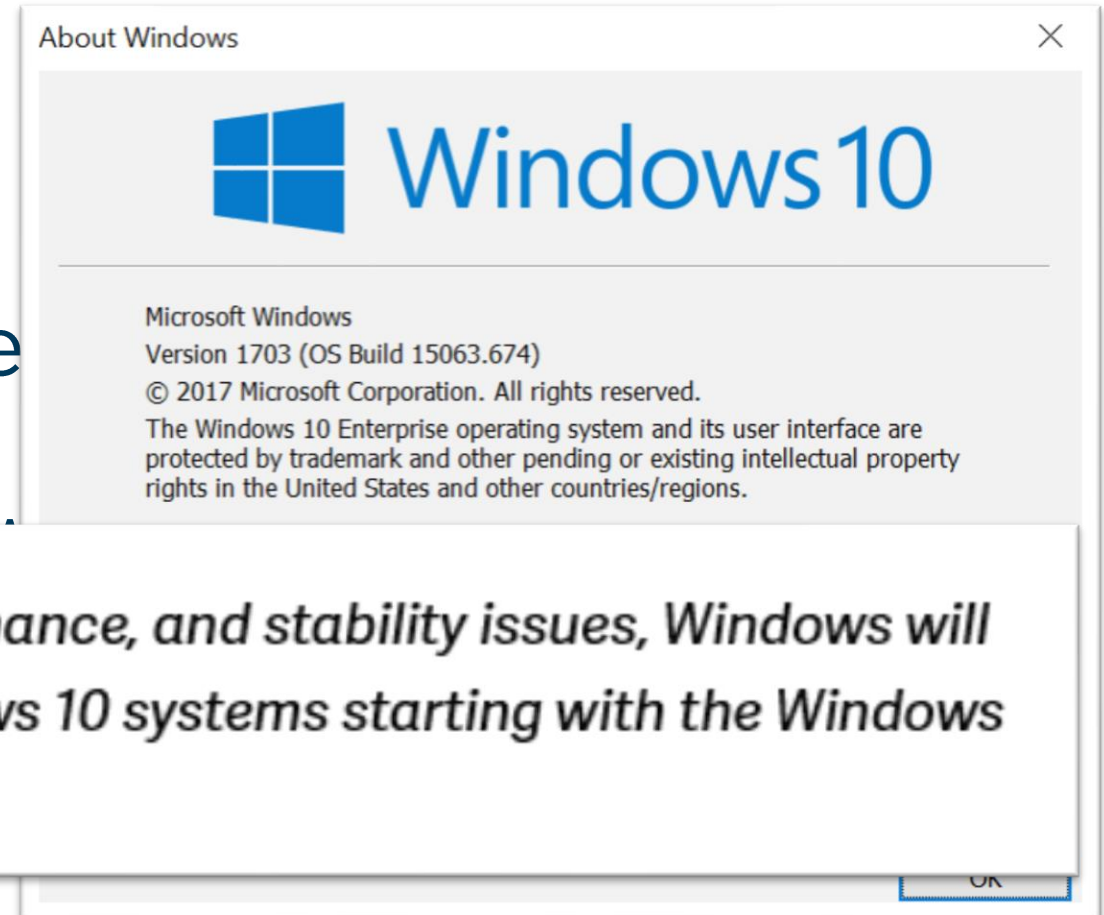
- > All mitigations can be bypassed
- > Makes exploitation harder
- > For x64 not all mitigations implemented
- > More effective for old x86 applications

EMET & Windows 10

> Checks got integrated into Windows 10

> But not all of them!

> EMET is blocked on new



Note: To prevent possible compatibility, performance, and stability issues, Windows will automatically block or remove EMET on Windows 10 systems starting with the Windows 10 Fall Creators Update.

EMET & Windows 10

- > Heap spray, EAF, EAF+ discontinued
- > LoadLib, MemProt at Process/Thread creation
- > ROP mitigations «replaced» by CFG

EMET & Windows 10 summary

- > EMET «replaced» by new compiler options & API
- > Some features might be found in WD Exploit Guard
- > Better compatibility
- > Responsibility shifted to developers
- > Bad news for legacy software



Control-Flow Guard

Control-flow integrity (CFI)

- Original publication in 2005
 - «Control-Flow Integrity – Principles, Implementations, and Applications»
Abadi, Budiu, Erlingsson, Ligatti, CCS'05
- Many CFI implementations proposed since then
 - Compiler-based, binary-only (static rewriting)
- Check indirect control-flow transfers and limit the set of allowed targets

Control-flow guard (CFG)

- First adoption of a practical Control-Flow Integrity (CFI) implementation
- Restricts indirect call/jmp targets to a static global set of locations
- Requires recompilation and OS support
 - VS15 + Windows 10 or 8.1

Compiling with /guard:cf (32bit)

NO /guard:cf

```
0131170E 8B F4      mov     esi,esp
01311710 68 AC 9D 31 01    push    1319DACH
01311715 FF 55 E8      call    dword ptr [func_ptr]
01311718 83 C4 04      add     esp,4
0131171B 3B F4      cmp     esi,esp
0131171D E8 DD F9 FF FF    call    __RTC_CheckEsp (013110FFh)
```

WITH /guard:cf

```
002D2478 8B F4      mov     esi,esp
002D247A 68 AC 9D 2D 00    push    2D9DACH
002D247F 8B 4D E4      mov     ecx,dword ptr [func_ptr]
002D2482 89 4D B0      mov     dword ptr [ebp-50h],ecx
002D2485 8B FC      mov     edi,esp
002D2487 8B 4D B0      mov     ecx,dword ptr [ebp-50h]
002D248A FF 15 00 F0 2D 00 call    dword ptr [__guard_check_icall_fptr (02DF000h)]
002D2490 3B FC      cmp     edi,esp
002D2492 E8 99 EE FF FF    call    __RTC_CheckEsp (02D1330h)
002D2497 FF 55 B0      call    dword ptr [ebp-50h]
002D249A 83 C4 04      add     esp,4
002D249D 3B F4      cmp     esi,esp
002D249F E8 8C EE FF FF    call    __RTC_CheckEsp (02D1330h)
```

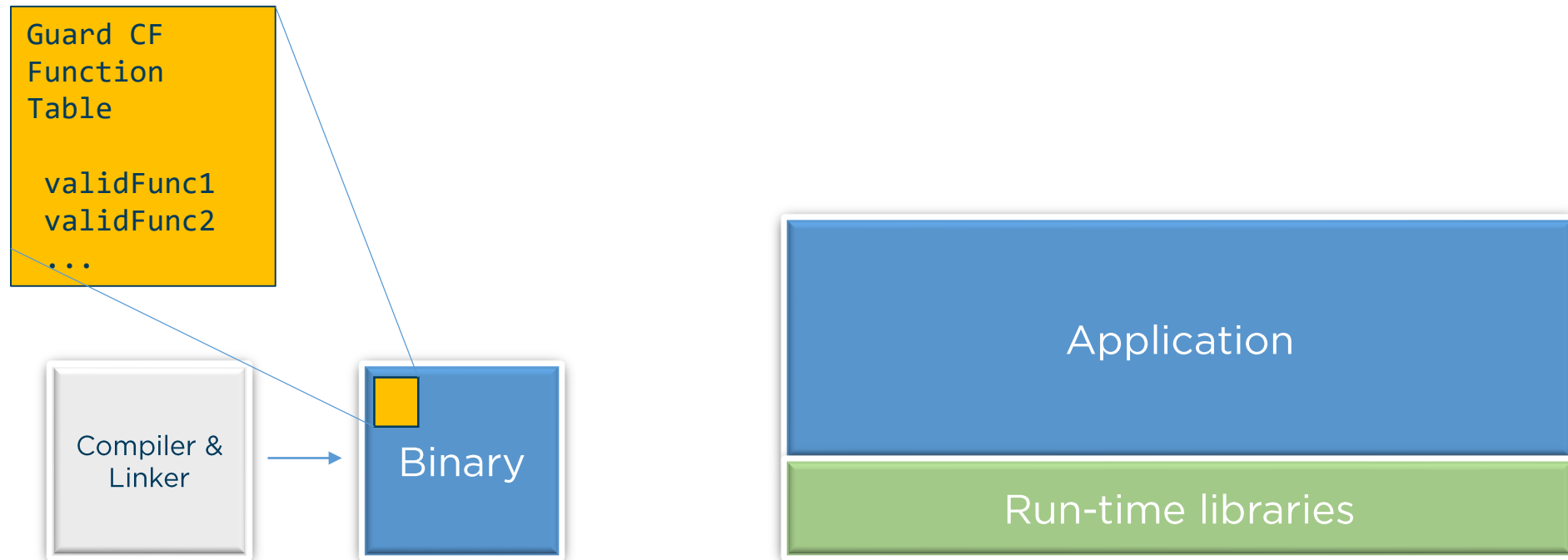
Added by
/guard:cf

- `call dword ptr [__guard_check_icall_fptr (...)]`
 - calls `LdrpValidateUserCallTarget` located in `kernel132.dll`

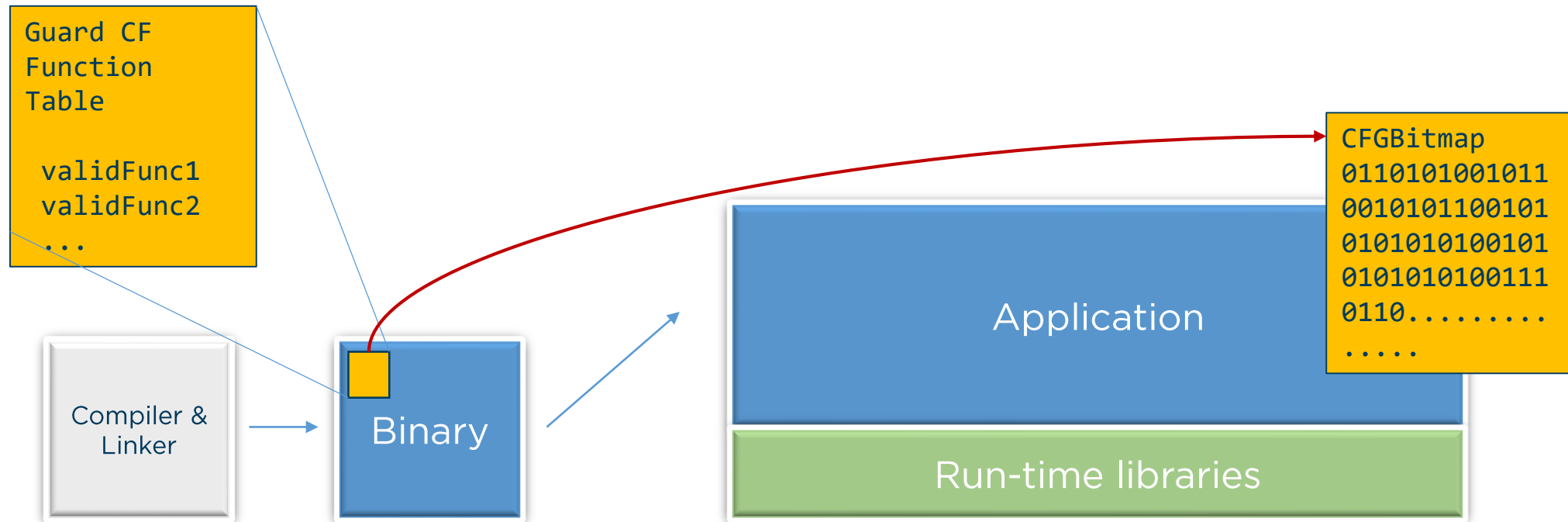
Control-flow guard (CFG)

- `call dword ptr [_guard_check_icall_fptr (...)]`
- Verifies if indirect control-flow transfer target is valid
 - Valid according to a global list of allowed targets
- Yes, this check is done for all indirect calls
 - And some indirect jmps
- Introduces some run-time overhead

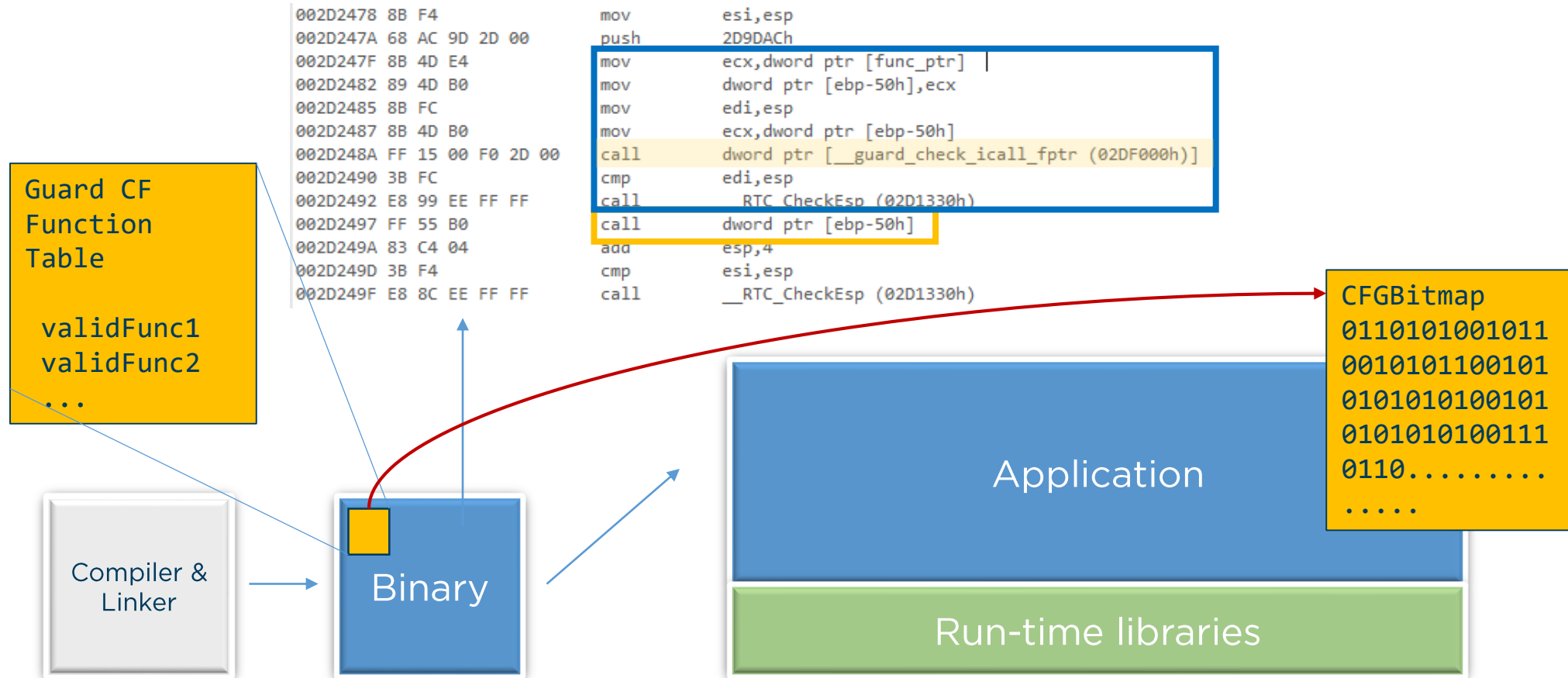
Control-flow guard (CFG)



Control-flow guard (CFG)



Control-flow guard (CFG)



Bypassing CFG

- Quite some work was already done
- Bypass on Windows 8.1, Francisco Falcon, March 2015
<https://blog.coresecurity.com/2015/03/25/exploiting-cve-2015-0311-part-ii-bypassing-control-flow-guard-on-windows-8-1-update-3/>
- Bypass CFG by Zhang Yunhai, Black Hat US 2015
<https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively-wp.pdf>
- Never Let your Guard Down, Sun et al., Black Hat Asia 2017
<https://www.blackhat.com/docs/asia-17/materials/asia-17-Sun-Never-Let-Your-Guard-Down-Finding-Unguarded-Gates-To-Bypass-Control-Flow-Guard-With-Big-Data.pdf>

Bypassing CFG - Implementation issues

- Non-CFG enabled modules
- `__guard_check_icall_fptr`
can be made writable in some cases
- Memory based indirect calls (RO function pointers)
- Unprotected JIT Code

Dynamic code generation

- All targets valid by default

```
void *mem = VirtualAlloc(NULL, size, MEM_COMMIT | MEM_RESERVE, PAGE_NOACCESS);
```

- New memory protection introduced

PAGE_EXECUTE

PAGE_EXECUTE_READ

- SetProcessExplicitAppUserModelID

[https://msdn.microsoft.com/library/windows/desktop/dn934202\(v=vs.85\).aspx](https://msdn.microsoft.com/library/windows/desktop/dn934202(v=vs.85).aspx)

JITs need to be
CFG aware

Bypassing CFG - Design issues

- Ret instructions not protected
- Call valid functions directly WinExec (see Falcon)
- There are still valid gadgets that can be called
 - Imprecision of bitmap
 - Just stay within the legitimate CFG

Return-Flow Guard

Return-Flow Guard

- > Mitigates rop attacks with a shadow stack
- > Compile-time technique, requires OS support
- > At compile-time reserve space with NOPs
- > At load time put RFG checks at reserved locations

Return-Flow Guard

> Function prologue

Copy stack top to shadow stack top

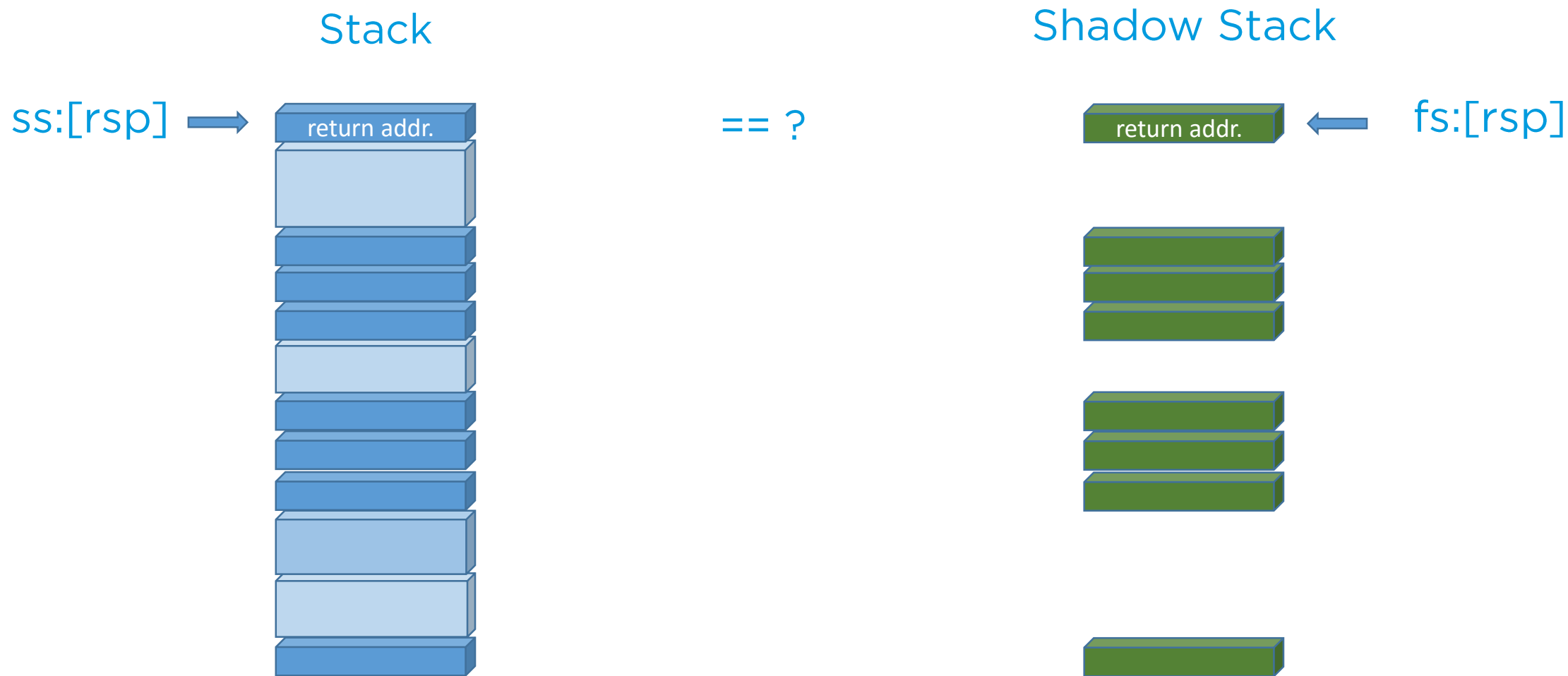
```
0:000> u calc!wWinMain
calc!wWinMain:
00007ff7`91ca176c 488b0424      mov     rax,qword ptr [rsp]
00007ff7`91ca1770 6448890424    mov     qword ptr fs:[rsp],rax
```

> Function epilogue

Compare stack top with shadow stack top

```
0:000> u calc!_guard_ss_common_verify_stub
calc!_guard_ss_common_verify_stub:
00007ff7`91ca25bc 644c8b1c24    mov     r11,qword ptr fs:[rsp]
00007ff7`91ca25c1 4c3b1c24      cmp     r11,qword ptr [rsp]
00007ff7`91ca25c5 0f85f5000000 jne     calc!_guard_ss_verify_failure (00007ff7`91ca26c0)
00007ff7`91ca25cb c3            ret
```

Return-Flow Guard



Return-Flow Guard status



Matt Miller

@epakskape

Heads up researchers
15031 removes sup
Guard. Bypass bou
[technet.microsoft.com](https://technet.microsoft.com/en-us/security/dn425049.aspx)

8:06 AM - 13 Feb 2017

81 Retweets 56 Likes



1 81 56

REVISION HISTORY

- Jan 31, 2017: Return Flow Guard experimental mitigation was removed from the list of in scope mitigations

6 7

RFG's Status

After I finished my research, I waited for an official Windows 10 version (Creators Update), hoping it will include Return Flow Guard.

Unfortunately, in Jan 31, 2017, shortly after it was announced, Microsoft updated the bug bounty page and excluded RFG from the program. They later added that their Red Team found a flaw in the mechanism, and that Microsoft chose to wait to Intel's hardware implementation of the Shadow Stack.

Intel CET

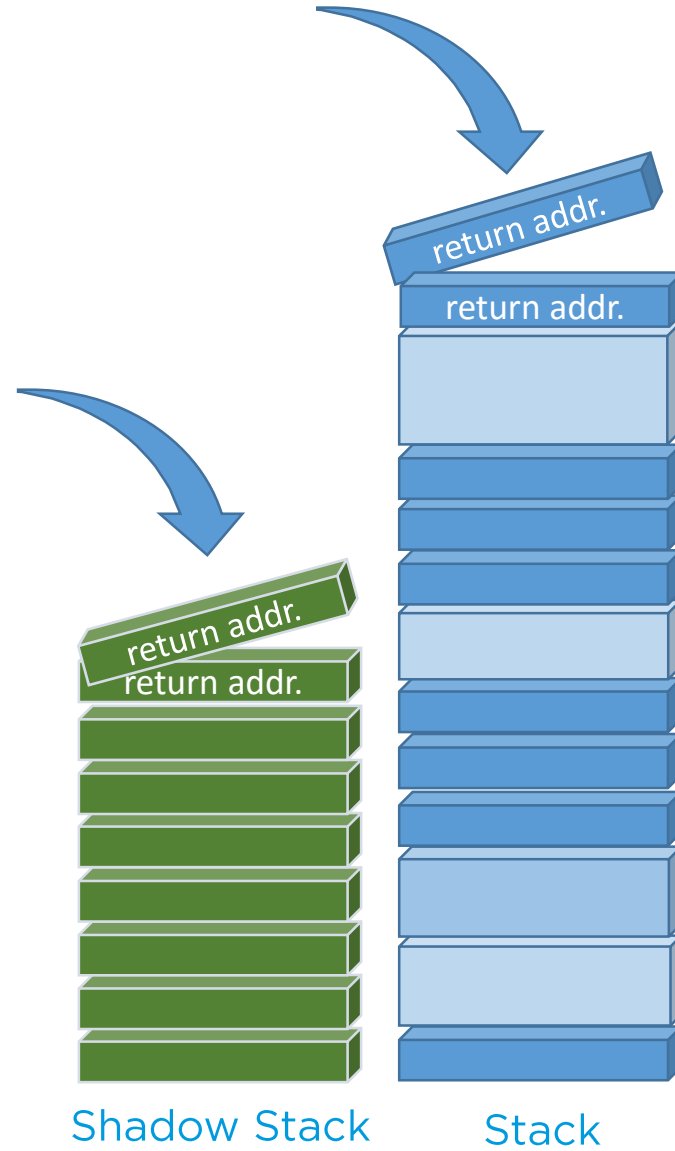
Control-flow Enforcement Technology (CET)

- > Hardware extension
- > Aims to defeat Return/Call/Jmp-oriented programming
- > Shadow stack for ret instructions
 - > Replaces RFG
- > ENDBRANCH for indirect calls/jmps
 - > Replaces CFG



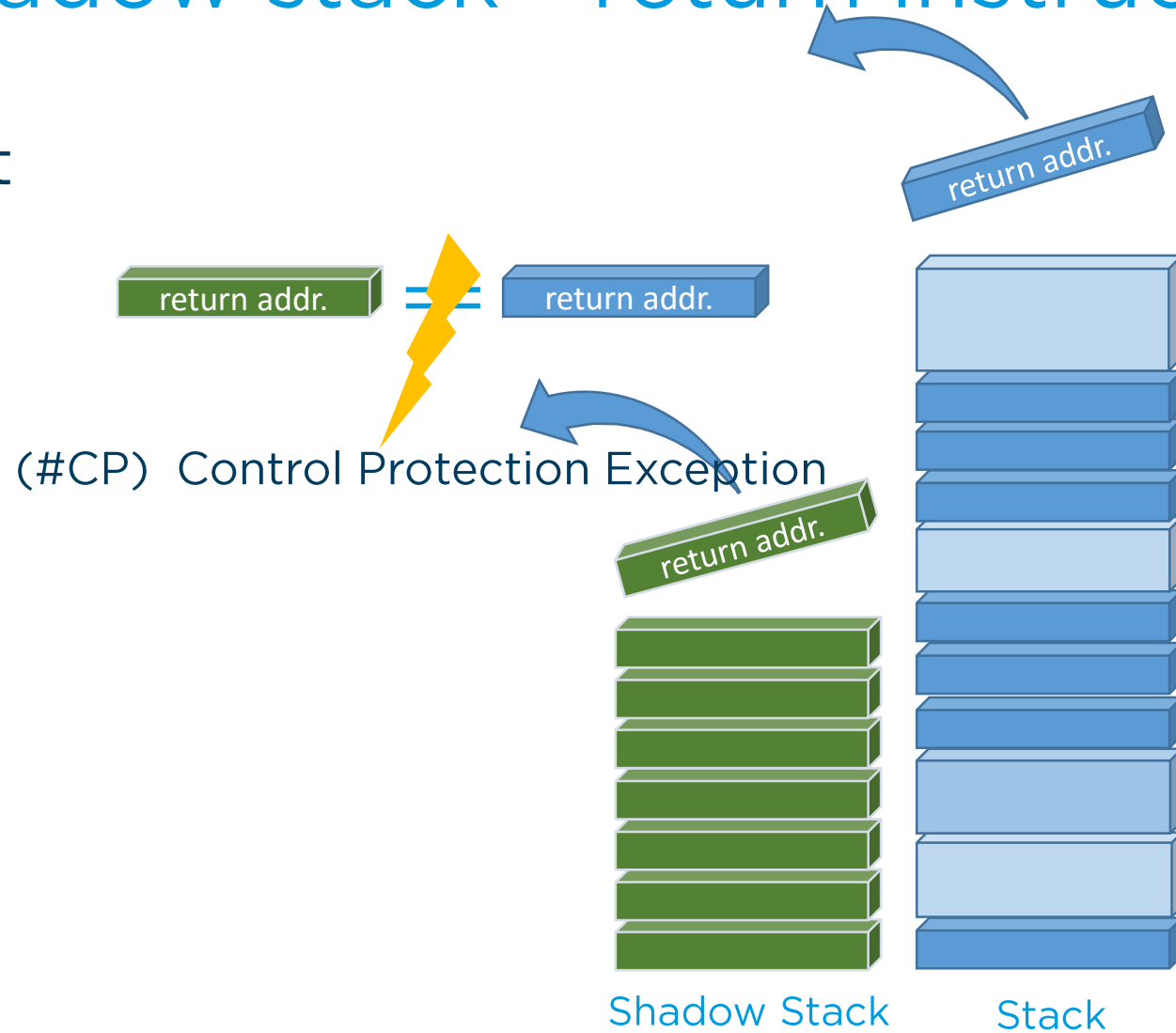
Shadow stack

call printf



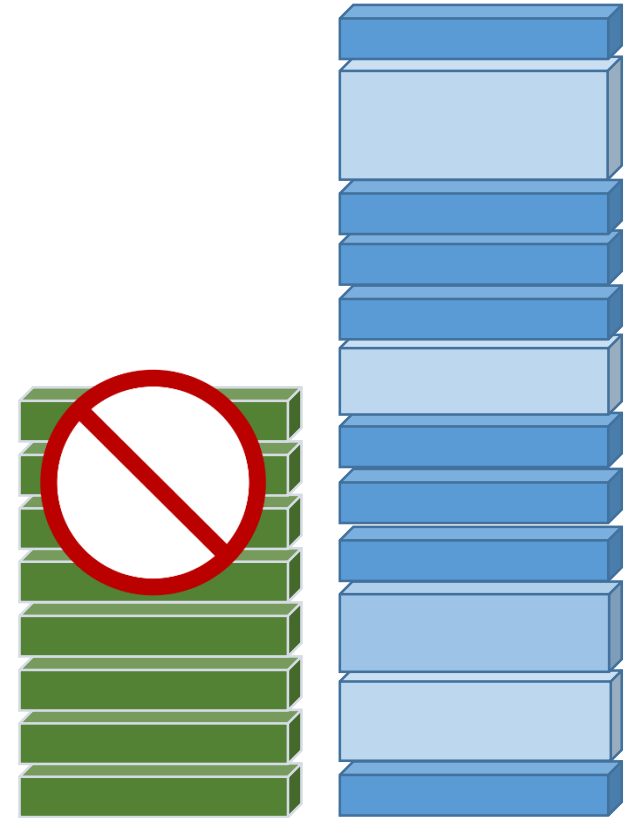
Shadow stack - return instructions

ret



Shadow stack protection

- > New page protection flag: Shadow Stack
- > Read/write only via
 - > shadow stack instructions
 - > call & ret
- > Push, mov, XSAVE fails with CP Exception



Indirect Branch Tracking



Idea: Global set of valid jmp/call targets

ENDBRANCH instruction marks **indirect call/jmp** targets

NOP opcode on legacy systems

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 1E FA	ENDBR64	NP	Valid	Valid	Terminate indirect branch in 64 bit mode.

Opcode	Instruction	Op/ En	64- Bit Mode	Compat/ Leg Mode	Description
F3 0F 1E FB	ENDBR32	NP	Valid	Valid	Terminate indirect branch in 32 bit and compatibility mode.

Indirect Branch Tracking

- > Mitigates hijacking of virtual calls
- > Jmp/Call-oriented programming
- > Substitution for Microsoft CFG (/guard:cf)

Indirect Branch Tracking (how it might look)

```
class Hello {  
public:  
    virtual void say_hello() {  
        printf("hello world\n");  
    }  
};  
  
int main(int argc, char *argv[]) {  
    Hello *h = new Hello();  
    h->say_hello();  
    return 0;  
}
```

```
mov rax, qword ptr [h]           # Load object h  
mov rax, qword ptr [rax]         # Load Hello v-table  
mov rcx, qword ptr [h]           # Prepare object h as Arg0  
call    qword ptr [rax]          # perform indirect call  
...
```

```
Hello_say_hello:  
    endbr                         # call target  
...
```

CET – it's a long way

- > CET Specification in revision 2.0 (June 2017)
- > Implement hardware
- > OS Support
- > Compiler support
- > Client program adoption
- > Customer adoption
 - > Buy new hardware
 - > Update OS
 - > Update software



Conclusion

Conclusion

- > We are definitely in the Post-DEP/ASLR era
 - > Modern systems have adopted DEP/ASLR
- > Control-flow integrity based mitigations are here
- > Intel CET will significantly raise the bar
 - > But adoption will require some years

xorlab

<https://www.xorlab.com>
contact@xorlab.com

