“Hey Cortana – Do We Still Care About Binary Exploitation?”

Connor McGarr [@33y0re]

WWHF 2022 – Deadwood

About Me

- Connor McGarr
  - Software Engineer @ CrowdStrike
- Blog/Contact
  - https://connormcgarr.github.io
  - @33y0re on Twitter
- I like C, ASM, and development!

Agenda

- Exploitation – The Golden Years
- A Post-DEP-and-ASLR World
- The “New-Era” of Exploitation

Exploitation – The Golden Years

Exploitation – The Golden Years

- Memory corruption exploits have been prominent for some time
  - *Smashing The Stack For Fun And Profit* from Phrack goes back to 1996

Current issue : **#49** | **Release date** : **1996-11-08** | **Editor** : **daemon9**

Title : Smashing The Stack For Fun And Profit

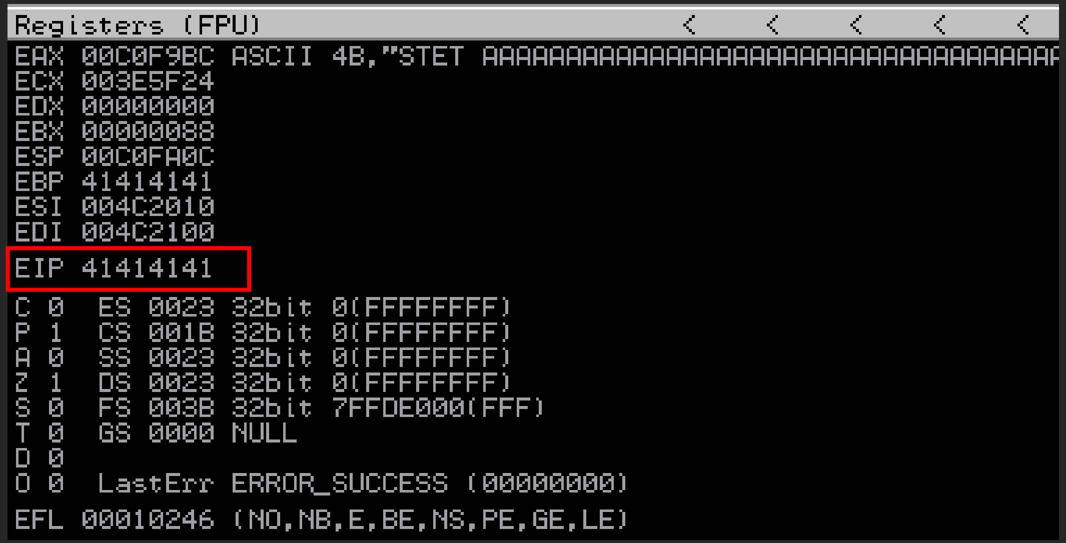
Author : Aleph1

```
.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org
```

Exploitation – The Golden Years

- Exploits at this time are not as documented/accessible – but are very trivial to exploit (by today's standards)
- Memory is readable/writable/executable/static and attackers have it easy

1. Overflow the stack with data and shellcode
2. Memory is static – so we can overwrite a return address with the hardcoded address of shellcode
3. Jump to the fake return address & profit!
   - Nothing stops this/us!



Exploitation – The Golden Years

- This led to the implementation of Data Execution Protection (DEP) on Windows systems in ~2005
  - The stack, along with other data segments of memory, are no longer executable – meaning we can no longer execute shellcode from the stack!

Command

```
0:021> u rip L1
ntdll!DbgBreakPoint:
00007ffb`7d2c7c40 ffe4            jmp     rsp
0:021> dps rsp L1
00000011`c7cff808  90909090`90909090
0:021> t
00000011`c7cff808 90              nop
0:021> t
(6bfc.6a5c): Access violation - code c0000005 (first chance)
```

Exploitation – The Golden Years

- DEP led to attackers to start using code-reuse attacks, as a first-stage payload, to mark the stack as executable
  - Most notably done in the form of ROP, or Return-Oriented Programming
    - We now re-use existing code that is executable!
- DEP – problem solved!
  - …But we are relying on static memory

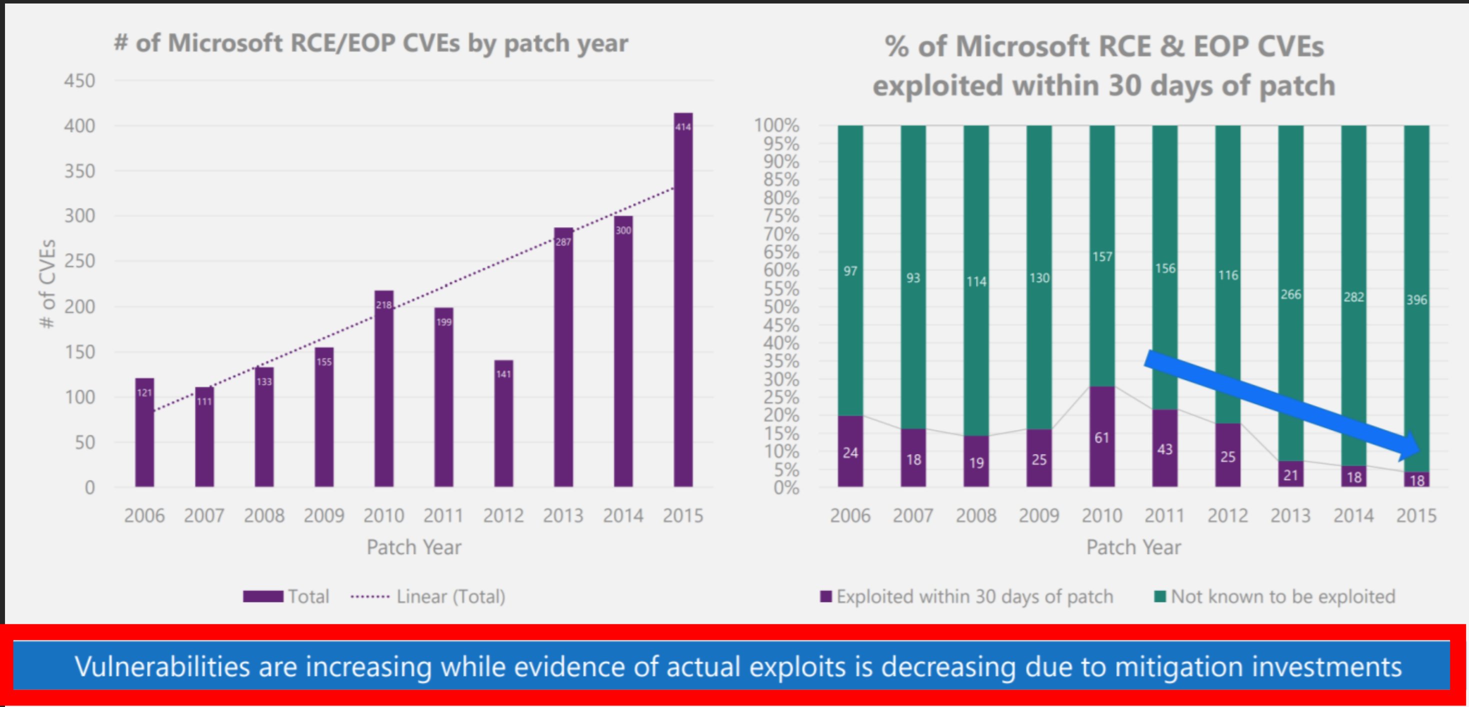
pop rcx; ret (0x7ff112233440: app.dll)
lpAddress (SHELLCODE)
pop rdx; ret (0x7ff112233448: app.dll)
dwSize (sizeof(SHELLCODE))
pop r8; ret (0x7ff112233448: app.dll)
flnewProtect (PAGE_EXECUTE_READWRITE)
pop r9; ret (0x7ff112233448: app.dll)
lpflOldProtect (Any writable pointer)
ret (0x7ff112233448: app.dll)
KERNELBASE!VirtualProtect

Exploitation – The Golden Years

- Static ROP gadgets led to the implementation of Address Space Layout Randomization (ASLR) in ~2007
  - Memory is now randomized on a per-boot basis
    - This means that the pop rcx gadget located at 0x7ff11223344 will be located somewhere else upon reboot
  - ASLR means we can no longer bypass DEP with hardcoded ROP gadget pointers anymore!
    - However, DLLs still need to compile with ASLR – so attackers can still abuse non-ASLR'd DLLs to use static memory
      - Internet Explorer 8 used an old version of MSVCR71.dll that didn't have ASLR when specific versions of Java are used
- For modern exploits we now rely on information disclosure vulnerabilities to leak the layout of memory in order to bypass ASLR

Exploitation – The Golden Years

- Issues with DEP and ASLR?
  - Statistics show in the era of <u>only</u> DEP/ASLR, exploitation was still rampant
    - ~2010-2011
    - What has caused this drop-off in exploitation within just the last few years if not ASLR and/or DEP?



A Post-DEP-and-ASLR World

A Post-DEP-and-ASLR World

- It turns out that only DEP and ASLR don’t solve our problems
  - Exploitation isn’t as trivial but there are two distinct characteristics that allow exploits to manifest (even with DEP and ASLR)
    1. Hijacking control-flow
    2. Execution of unsigned-code/shellcode
- DEP and ASLR make exploitation more tedious – but they aren’t resilient against a modern attacker
  - Once we bypass ASLR we then can return to the “old way” of doing exploitation (after first using ROP to mark shellcode as executable)

A Post-DEP-and-ASLR World

- Hijacking control-flow
  - Primarily done in one of two ways (not a comprehensive list)
    1. Function pointer overwrite
    2. Return address overwrite

A Post-DEP-and-ASLR World

- Function pointer invocation
  - Also known as an “indirect call”
  - Call to a *non-absolute* memory address
    - call rax or call [qword ptr *ADDR*] for example
      - Is RAX, at call time, legitimate or malicious?
      - What does *ADDR* point to? How is it possible to verify this at runtime?
  - If an attacker can corrupt what a function pointer is pointing to – when said function pointer is invoked, it will call into attacker-controlled memory

A Post-DEP-and-ASLR World

- Example
- Browsers are popular exploit targets and are written in C++
  - Copious usage of virtual functions
  - Virtual function tables inherently store an array of virtual functions
    - …Also known as an array of function pointers

1. Locate a virtual function pointer that can be invoked by an accessible method in JavaScript
2. Overwrite the function pointer with attacker-controlled memory
3. Invoke the method in JavaScript, which internally will invoke the function pointer which has now been overwritten

A Post-DEP-and-ASLR World

- Overwriting a function pointer

```
Command
0:000> dps chakracore!Js::DataView::`vftable'
00007ffd`27d16c08  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c10  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c18  00007ffd`27a7d9f0 chakracore!FinalizableObject::Mark
00007ffd`27d16c20  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c28  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c30  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c38  00007ffd`279cd6d0 chakracore!Js::DynamicTypeHandler::AllPropertiesAreEnumerable
00007ffd`27d16c40  00007ffd`27bdc0d0 chakracore!Js::DynamicObject::GetPropertyId
```

```
00007ffd`27d173e8  00007ffd`2793ac60 chakracore!Js::DynamicObject::GetItemQuery
00007ffd`27d173f0  00007ffd`2793abf0 chakracore!Js::DynamicObject::GetItemSetter
00007ffd`27d173f8  00007ffd`2793ab80 chakracore!Js::DynamicObject::SetItem
00007ffd`27d17400  00007ffd`27bdbf30 chakracore!Js::DynamicObject::DeleteItem
```

A Post-DEP-and-ASLR World

- Overwriting a function pointer

Command

```
0:000> dps chakracore!Js::DataView::`vftable'
00007ffd`27d16c08  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c10  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c18  00007ffd`27a7d9f0 chakracore!FinalizableObject::Mark
00007ffd`27d16c20  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c28  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c30  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c38  00007ffd`279cd6d0 chakracore!Js::DynamicTypeHandler::AllPropertiesAreEnumerable
00007ffd`27d16c40  00007ffd`27bdc0d0 chakracore!Js::DynamicObject::GetPropertyId
```

```
00007ffd`27d173e8  00007ffd`2793ac60 chakracore!Js::DynamicObject::GetItemQuery
00007ffd`27d173f0  00007ffd`2793abf0 chakracore!Js::DynamicObject::GetItemSetter
00007ffd`27d173f8  41414141`41414141
00007ffd`27d17400  00007ffd`27bdbf30 chakracore!Js::DynamicObject::DeleteItem
```

A Post-DEP-and-ASLR World

- How do we stop something like this?
  - Control flow integrity!
- Control flow integrity, or CFI, is a way to ensure that the control-flow of an application/target isn’t hijacked or tampered with
  - Windows supports both *forward-edge* and *backwards-edge* control flow integrity

A Post-DEP-and-ASLR World

- Forward-edge inspection
  - Allows for inspection of indirect calls and jumps
- Microsoft's implementation of forward-edge inspection is known as Control Flow Guard, or CFG
  - We can now inspect *call [qword ADDR]*!

A Post-DEP-and-ASLR World

- CFG has support in both user-mode and kernel-mode
  - CFG works by creating a bitmap that is comprised of all known indirect call targets at compile time
    - By keeping a “source of truth” (the bitmap) with all the functions that should be used in valid control-flow transfers, we can inspect each indirect function call and determine if a legitimate function is being called
  - When a call happens, we now check to see if the target function exists in the bitmap. If it is, it is a valid call target. If it isn’t, the process crashes
    - Checks are performed by __guard_dispatch_icall_fptr which points to ntdll!LdrpDispatchUserCallTarget

Command

```
0:023> dps msedge!_guard_dispatch_icall_fptr
00007fff`f4c3f008  00007ff8`972b3720 ntdll!LdrpDispatchUserCallTarget
00007fff`f4c3f010  00007ff8`972b35e0 ntdll!LdrpValidateUserCallTarget
```

A Post-DEP-and-ASLR World

- With CFG, when we call into 0x4141414141414141 we validate if this is a valid call target

Command

```
0:000> dps chakracore!Js::DataView::`vftable'
00007ffd`27d16c08  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c10  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c18  00007ffd`27a7d9f0 chakracore!FinalizableObject::Mark
00007ffd`27d16c20  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c28  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c30  00007ffd`27872390 chakracore!DListBase<Memory::ArenaData * __ptr64,FakeCount>::~DListBase<Memory::ArenaData *
00007ffd`27d16c38  00007ffd`279cd6d0 chakracore!Js::DynamicTypeHandler::AllPropertiesAreEnumerable
00007ffd`27d16c40  00007ffd`27bdc0d0 chakracore!Js::DynamicObject::GetPropertyId
```

```
00007ffd`27d173e8  00007ffd`2793ac60 chakracore!Js::DynamicObject::GetItemQuery
00007ffd`27d173f0  00007ffd`2793abf0 chakracore!Js::DynamicObject::GetItemSetter
00007ffd`27d173f8  41414141`41414141
00007ffd`27d17400  00007ffd`27bdbf30 chakracore!Js::DynamicObject::DeleteItem
```

A Post-DEP-and-ASLR World

- Since 0x4141414141414141 isn't a valid call target, the process now crashes
  - R11 contains bitmap address
  - RAX contains the address to be checked
    - 0x4141414141414141 wasn't found in the bitmap, and a crash occurs

Command

```
0:023> u rip
ntdll!LdrpDispatchUserCallTarget:
00007ff8`972b3720 4c8b1d81bc0f00  mov     r11,qword ptr [ntdll!LdrSystemDllInitBlock+0xb8 (00007ff8`973af3a8)]
00007ff8`972b3727 4c8bd0          mov     r10,rax
00007ff8`972b372a 49c1ea09        shr     r10,9
00007ff8`972b372e 4f8b1cd3        mov     r11,qword ptr [r11+r10*8]
00007ff8`972b3732 4c8bd0          mov     r10,rax
00007ff8`972b3735 49c1ea03        shr     r10,3
00007ff8`972b3739 a80f            test    al,0Fh
00007ff8`972b373b 7509            jne     ntdll!LdrpDispatchUserCallTarget+0x26 (00007ff8`972b3746)
0:023> r rax
rax=4141414141414141
0:023> t
(30c4.1df8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
ntdll!LdrpDispatchUserCallTarget+0xe:
00007ff8`972b372e 4f8b1cd3        mov     r11,qword ptr [r11+r10*8] ds:010582fa`a1440500=????????????????
```

A Post-DEP-and-ASLR World

- CFG stops us in the previous scenario!
  - However, CFG doesn’t solve *all* our problems…

A Post-DEP-and-ASLR World

- Consider that application.exe is compiled with CFG
  - CFG only knows about functions within application.exe that are used in control-flow transfers at compile time
  - What if application.exe needs to make a call to kernelbase!VirtualAlloc?
    - VirtualAlloc is found in kernelbase.dll – *not* in application.exe and is instead imported
      - This creates an issue with intermodular calls as VirtualAlloc isn't technically a valid call target in application.exe

A Post-DEP-and-ASLR World

- Due to these compatibility issues, CFG essentially allows *any* exported function (except for a few explicitly-suppressed functions) to be used in a valid control-flow transfer
- This means that CFG *does* prevent overwriting a function pointer with something like a heap address, but we can still:
  1. Overwrite function pointers with pointers from kernelbase.dll, kernel32.dll, and ntdll.dll
  2. Overwrite function pointers with *other* valid call targets
     - E.g., msedge!FUNC1 can be overwritten with msedge!FUNC2
- This is an issue as CFG should allow *only* the developer-intended function pointer to be called
- How do we improve on this in order to make CFG "better"?

A Post-DEP-and-ASLR World

- eXtended Flow Guard (XFG)
  - "What CFG always should have been"
- XFG works by hashing the prototype of a function to create a signature
  - This includes the number and type of parameters, and the return value
  - This information *should* be unique to a given function and can be used to "identify" each function
- An "XFG hash" is placed 8 bytes above every call target and is used as an additional check before control-flow transfer

A Post-DEP-and-ASLR World

- _guard_xfg_dispatch_icall_fptr is the new check function
  - The XFG hash is placed in R10 during the check
  - Since each function now has a unique identifier, we can now ensure that only developer-intended functions are used in control-flow transfers!

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

var_18= qword ptr -18h

sub     rsp, 38h
mov     eax, 8
imul    rax, 0
lea     rcx, cfgTest1
mov     rax, [rcx+rax]
mov     [rsp+38h+var_18], rax
mov     r10, 85F13E9656DA4870h
mov     rax, [rsp+38h+var_18]
call    cs:__guard_xfg_dispatch_icall_fptr
main endp
```

A Post-DEP-and-ASLR World

- Issues?
- XFG is currently instrumented, *but it is not enforced*
  - In the current implementation, if an XFG check determines that a function-pointer overwrite has occurred, XFG defaults to CFG – meaning that currently XFG is essentially just a trampoline into CFG
  - All CFG bypasses/limitations at the current moment are valid – although this is subject to change when XFG becomes <u>fully</u> enforced

A Post-DEP-and-ASLR World

- We have talked about inspecting calls and other forward-edge control-flow transfers, but what about *backwards-edge* control flow
  - What about returns?

A Post-DEP-and-ASLR World

- A well-known limitation of CFG/XFG are return control-flow transfers

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

A Post-DEP-and-ASLR World

- Returns?
  - When a call instruction happens a return address is pushed onto the stack
    - This return address is used by the callee (the function being called) to determine where to return control-flow/execution after the function work has been performed
- What if we can leak the stack and corrupt the return address *before* the return address it is executed
  - This would allow us to achieve control-flow hijacking via a return address corruption!

A Post-DEP-and-ASLR World

- Example

Command

```
ModLoad: 00007fff`44380000 00007fff`443eb000   C:\Windows\System32\oleacc.dll
ModLoad: 00007fff`52950000 00007fff`52960000   C:\Windows\system32\msimtf.dll
ModLoad: 00007fff`4fc20000 00007fff`4fca8000   C:\Windows\system32\directmanipulation.dll
ModLoad: 00007fff`53250000 00007fff`532b9000   C:\Windows\SYSTEM32\Bcp47Langs.dll
ModLoad: 00007fff`4b5d0000 00007fff`4b5e4000   C:\Windows\System32\Windows.System.Profile.PlatformDiagnosticsAndUsageDataSet
ModLoad: 00007fff`45ab0000 00007fff`45ae8000   C:\Windows\System32\smartscreenps.dll
ModLoad: 00007fff`4fcb0000 00007fff`4fe38000   C:\Windows\SYSTEM32\windows.globalization.dll
(1b98.150c): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007fff`5ba68d70 cc              int     3
0:034> g
onecoreuap\inetcore\urlmon\zones\zoneidentifier.cxx(366)\urlmon.dll!00007FFF505D6790: (caller: 00007FFF505D642D) ReturnHr(2)
(1b98.2010): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
chakra!amd64_CallFunction+0x9a:
00007fff`3d1c303a c3              ret
0:017> dqs rsp
000000dd`66afae18  41414141`41414141
000000dd`66afae20  000002c2`05c2a120
000000dd`66afae28  00000000`00000010
000000dd`66afae30  ffffffff`fffffffe
000000dd`66afae38  00007fff`3d0b0a0b chakra!Js::ScriptContext::OnScriptStart+0xbb
000000dd`66afae40  000000dd`66afb2a0
000000dd`66afae48  00000000`00000000
000000dd`66afae50  000002c2`05c29cd0
000000dd`66afae58  00000000`00000002
000000dd`66afae60  000002c2`092a0700
000000dd`66afae68  000002c2`05c2a120
000000dd`66afae70  000000dd`66afb148
000000dd`66afae78  00007fff`3cfecb10 chakra!Js::JavascriptFunction::CallRootFunctionInternal+0x100
000000dd`66afae80  000002c2`092a0720
000000dd`66afae88  00007fff`3d1c3190 chakra!Js::JavascriptFunction::DeferredParsingThunk
000000dd`66afae90  000000dd`66afaed0
```

A Post-DEP-and-ASLR World

- Attackers can still control execution even with XFG/CFG enabled by overwriting a return address on the stack!
  - Due to this, Microsoft tried to implement a software mitigation to protect return addresses known as Return Flow Guard, or RFG
    - RFG was deprecated due a design issue where attackers could still locate protected return addresses
  - Because of this a hardware mitigation on Intel chips, known as Intel Control-Flow Enforcement Technology (CET), is used instead to protect return addresses

A Post-DEP-and-ASLR World

- Intel CET
  - Hardware-assisted control flow integrity
  - Inspects both forward-edge and backward-edge control-flow transfers
    - Forward-edge inspection is known as Indirect Branch Tracking, or IBT
    - Backwards-edge inspection is known as Shadow Stack
- Since Microsoft already has a forward-edge solution (CFG/XFG), only the shadow stack portion of CET is used

A Post-DEP-and-ASLR World

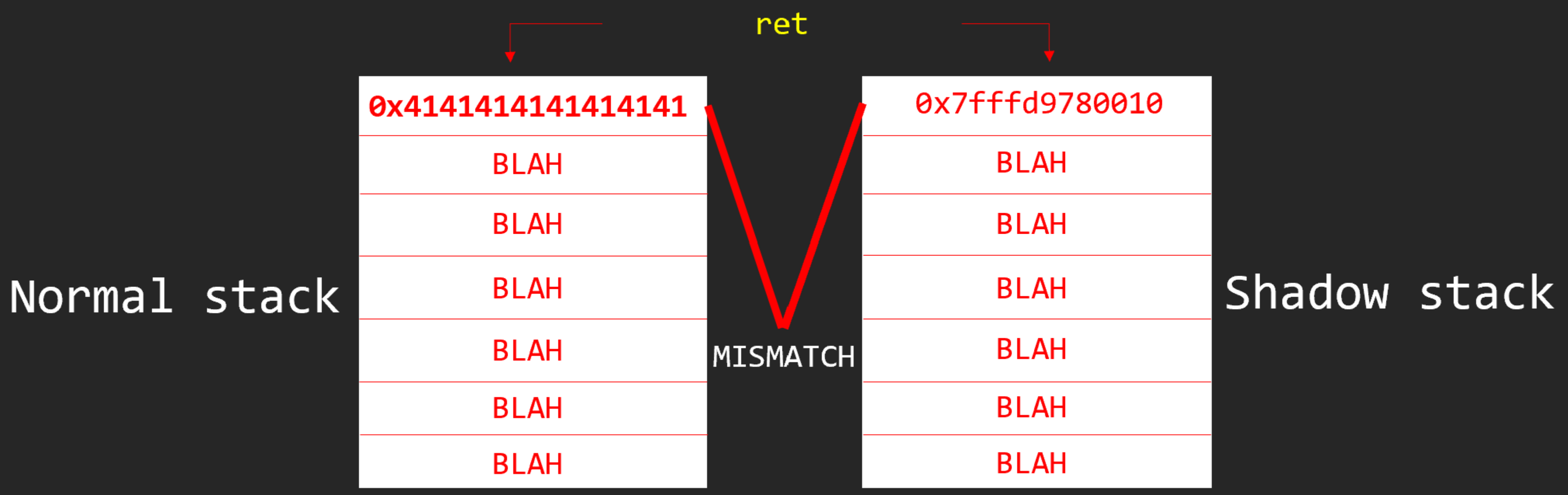
- The shadow stack contains only return addresses (and no parameters)
  - When a call happens, with shadow stack enabled, a return address is pushed onto the normal stack *and* the shadow stack

`call foo()`

Normal stack	Shadow stack
0x7fffd9780010	0x7fffd9780010
BLAH	BLAH
BLAH	BLAH
BLAH	BLAH
BLAH	BLAH
BLAH	BLAH
BLAH	BLAH

A Post-DEP-and-ASLR World

- When a return happens, the shadow stack copy of the return address is compared to the normal stack
  - The shadow stack is protected by hardware and is immutable
    - If they are different, a crash occurs



A Post-DEP-and-ASLR World

- This means ROP is mitigated
  - ROP relies on flooding the stack with fake return addresses
    - Fake return addresses are now detected and prevented with a shadow stack copy of the stack which is inaccessible by software

A Post-DEP-and-ASLR World

- Windows CFI conclusion:
  - XFG mitigates function pointer overwrites
  - CET mitigates return address overwrites
    - Need to find other ways to control execution!
    - More time will need to pass for greater adoption in the public for XFG and CET

A Post-DEP-and-ASLR World

- Control flow integrity is great, but it isn’t resilient
    - Your machine is only as safe as your CFI is good!
    - If an attacker bypasses CFI, exploitation can still occur “the old way”
- We still need to address the other element of exploitation, and that is shellcode execution

A Post-DEP-and-ASLR World

- Execution of unsigned-code/shellcode
  - Achieved by violating memory permissions
  - DEP separates data and code
    - Data should be read/write (data doesn't need to be executed)
    - Code should be read/execute (code *only* needs to execute)
  - Violating DEP means either code becomes writable, or data becomes executable – allowing an attacker to place unsigned-code in executable memory

A Post-DEP-and-ASLR World

- To combat the issue of attackers manually marking shellcode as executable, typically via ROP, Microsoft implemented Arbitrary Code Guard (ACG)
  - ACG is the Windows implementation of W^X
    - Memory can be writable, or executable – but never both
  - How does it work?

A Post-DEP-and-ASLR World

- The stack, for instance, is a data-only and stores arguments and return addresses

Command

```
0:000> dps rsp
00000076`727af2b0  00000246`69bb7c50
00000076`727af2b8  00007ffd`0000001b
00000076`727af2c0  00000246`69bb74b0
00000076`727af2c8  00007ffd`cba083ba ntdll!RtlpFreeHeapInternal+0x7ca
00000076`727af2d0  00000000`00000009
00000076`727af2d8  00000000`00000000
00000076`727af2e0  00000000`00000000
00000076`727af2e8  00000000`00000000
00000076`727af2f0  00000000`00000000
00000076`727af2f8  00000000`00000000
00000076`727af300  00000000`00000000
00000076`727af308  00000000`00000000
00000076`727af310  00000000`00000000
00000076`727af318  00000000`00000000
00000076`727af320  00000000`00000000
00000076`727af328  00000000`00000000
```

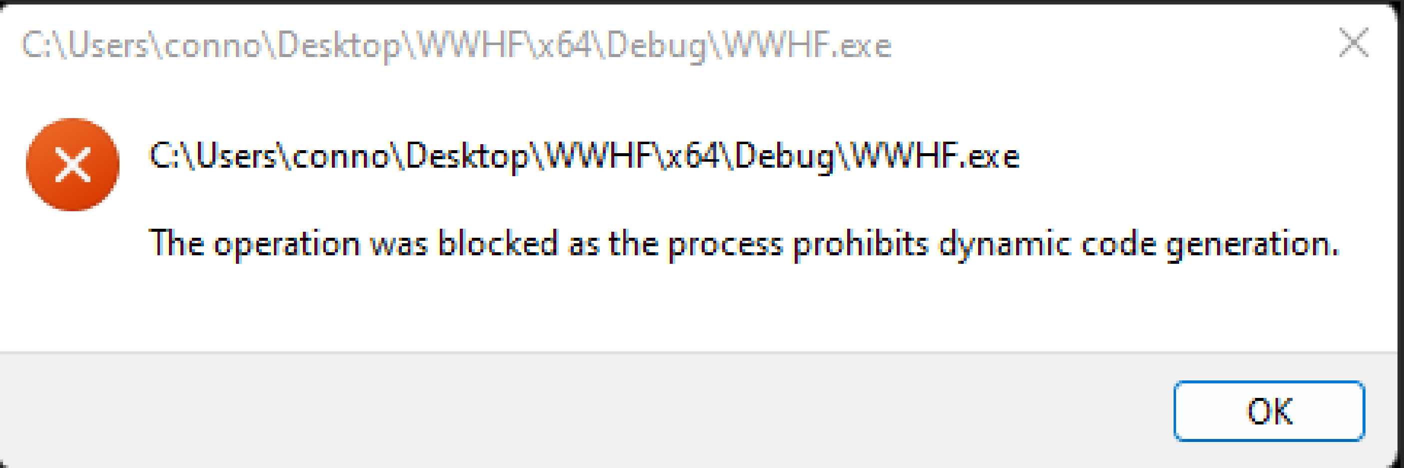
A Post-DEP-and-ASLR World

- Data-segments of memory like the stack never need to execute code!
- Additionally, code segments never need to be writable and only should execute code
  - ACG enforces that:
    1. Data can *NEVER* become code
    2. Code can *NEVER* become writable

A Post-DEP-and-ASLR World

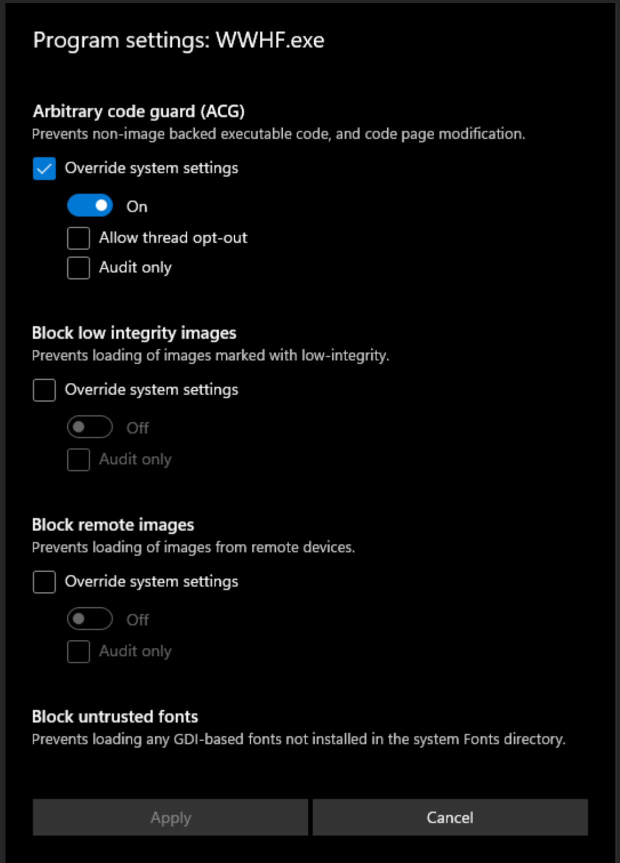
- It's now impossible to generate dynamic code!

```c
//
// Call VirtualAlloc to create RW memory.
//
buffer = VirtualAlloc(NULL,
                      size,
                      MEM_COMMIT | MEM_RESERVE,
                      PAGE_READWRITE);

//
// Error handling.
//
if (buffer == NULL)
{
    goto Exit;
}

//
// Call VirtualProtect to make the memory executable.
//
result = VirtualProtect(buffer,
                        size,
                        PAGE_EXECUTE_READ,
                        &oldProtect);

//
// Error handling.
//
if (!result)
{
    goto Exit;
}
```



A Post-DEP-and-ASLR World

- How can ACG be enabled?
  - Exploit protection settings on Windows or SetProcessMitigationPolicy API




```
BOOL result;
PROCESS_MITIGATION_DYNAMIC_CODE_POLICY policy = { 0 };

//
// Set the policy.
//
policy.ProhibitDynamicCode = 0x1;        // Enable ACG

//
// Call SetProcessMitigationPolicy.
//
result = SetProcessMitigationPolicy(ProcessDynamicCodePolicy,
                                    &policy,
                                    sizeof(PROCESS_MITIGATION_DYNAMIC_CODE_POLICY));

//
// Error handling.
//
if (!result)
{
    goto Exit;
}
```

A Post-DEP-and-ASLR World

- It is important to note that ACG is a <u>user-mode mitigation only</u>
  - The actual mitigation is implemented in the kernel
    - ACG's integrity is protected by the user/security boundary
  - nt!MiArbitraryCodeBlocked checks if a process has ACG enabled when page permission changes occur (in ACG-related situations)

If the mitigation is enabled for the process and the current thread isn't allowed to opt-out of ACG (which is possible) ACG enforcement occurs

```
__int64 __fastcall MiArbitraryCodeBlocked(_EPROCESS *CurrentProcess)
{
  struct _KTHREAD *CurrentThread; // rcx
  unsigned int MitigationFlags; // edx

  CurrentThread = KeGetCurrentThread();
  MitigationFlags = CurrentProcess->MitigationFlags;
  if ( (MitigationFlags & 0x100) == 0 || _bittest((const signed __int32 *)&CurrentThread[1].SwapListEntry + 2, 0x12u) )
  {
    if ( (MitigationFlags & 0x800) != 0 && !_bittest((const signed __int32 *)&CurrentThread[1].SwapListEntry + 2, 0x12u) )
      EtwTimLogProhibitDynamicCode(1i64, CurrentProcess);
    EtwTraceMemoryAcg(0i64);
    return 0i64;
  }
  else
  {
    EtwTraceMemoryAcg(0x80000000i64);
    EtwTimLogProhibitDynamicCode(2i64, CurrentProcess);
    return 0xC0000604i64;
  }
}
```

STATUS_DYNAMIC_CODE_BLOCKED

A Post-DEP-and-ASLR World

- Attackers can no longer create unsigned code/shellcode!
  - …with some interesting caveats
- ACG was created for Microsoft Edge as an exploit mitigation
  - Edge, however, has a problem (like all browsers) in that it has a just-in-time (JIT) engine
    - JIT engines, by nature, constantly create dynamic executable code
  - What is the answer?
    - Edge uses an out-of-process JIT server
      - Essentially a separate "JIT" process injects the JIT'd code into an ACG-protected content/renderer process (which is what users interface with)
        - The JIT process *doesn't* have ACG enabled while the content process *does*

A Post-DEP-and-ASLR World

- This means that ACG had to be designed, because of Edge, to allow non-ACG protected processes to inject into ACG-protected processes!
  - Remember, the JIT process isn't protected by ACG but needs to inject code into an ACG-protected process (the content process)
- Additionally, an ACG-protected process *cannot* <u>directly</u> inject into a non-ACG protected process!
    - This is because if an attacker can compromise the content process (ACG enabled) they could, instead of compromising the local process, inject into the JIT server process where ACG is not enabled
- tl;dr – ACG is a great exploit mitigation for what it intends to do
  - ACG attempts to prevent an exploited ACG-protected process for creating dynamic executable memory, or migrating to a non-ACG protected process

A Post-DEP-and-ASLR World

- We have talked about ACG as a user-mode mitigation, but what about implementing kernel-mode ACG to prevent unsigned-code execution in the kernel?
  - The issue here is that the kernel is the highest security boundary on a system
  - If kernel ACG was implemented in the kernel, but we assume attackers are already in the kernel, they could just disable the mitigation (like in the case of DEP in user mode)
    - The kernel "can't defend against itself"

A Post-DEP-and-ASLR World

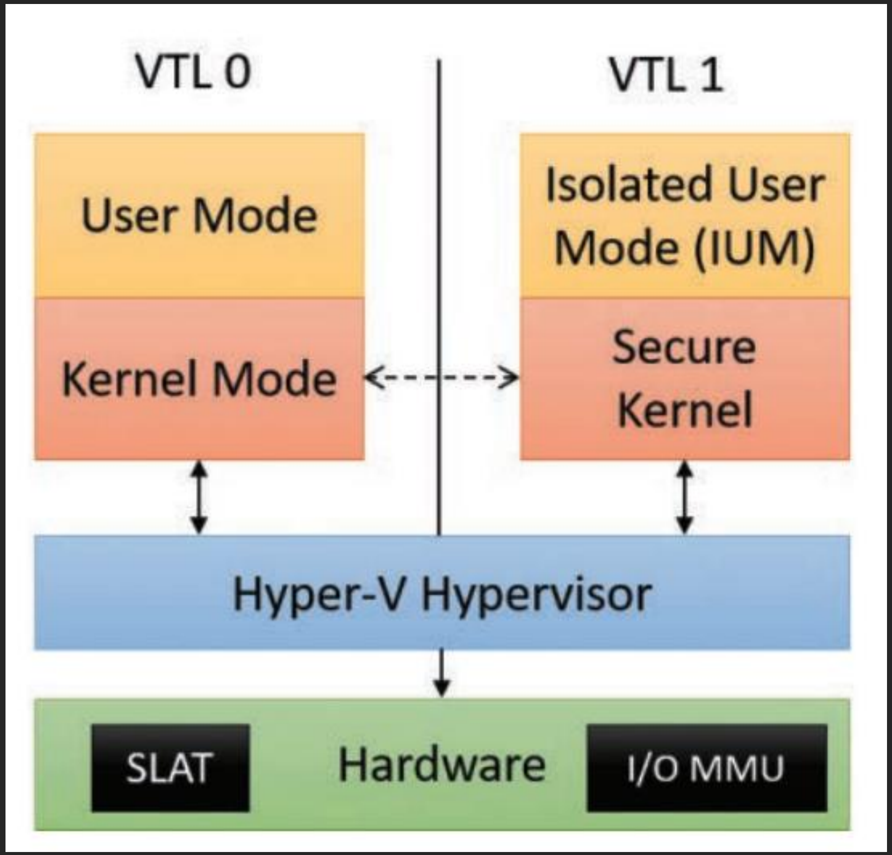
- What can we do about this?
  - Microsoft produced a brilliant and quite novel way to address this
    - Let’s use a higher security boundary than the kernel – the hypervisor!

A Post-DEP-and-ASLR World

- Virtualization-Based Security
  - Provided by Hyper-V, the Microsoft hypervisor
- Windows is now broken up into two “virtual trust levels” (VTLs) with the hypervisor having the ability to configure these VTLs
  - VTL 1 – “Secure mode”
  - VTL 0 – “Normal mode”
    - VTLs are essentially VMs in the sense that their memory is isolated from one another, but there is no virtual disk, networking, etc.
    - VTLs “abuse” the technologies used to isolate VMs
      - Second Layer Address Translation (SLAT)/Extended Page Tables (EPTs)

A Post-DEP-and-ASLR World

- The OS is now split into (currently two) VTLs



A Post-DEP-and-ASLR World

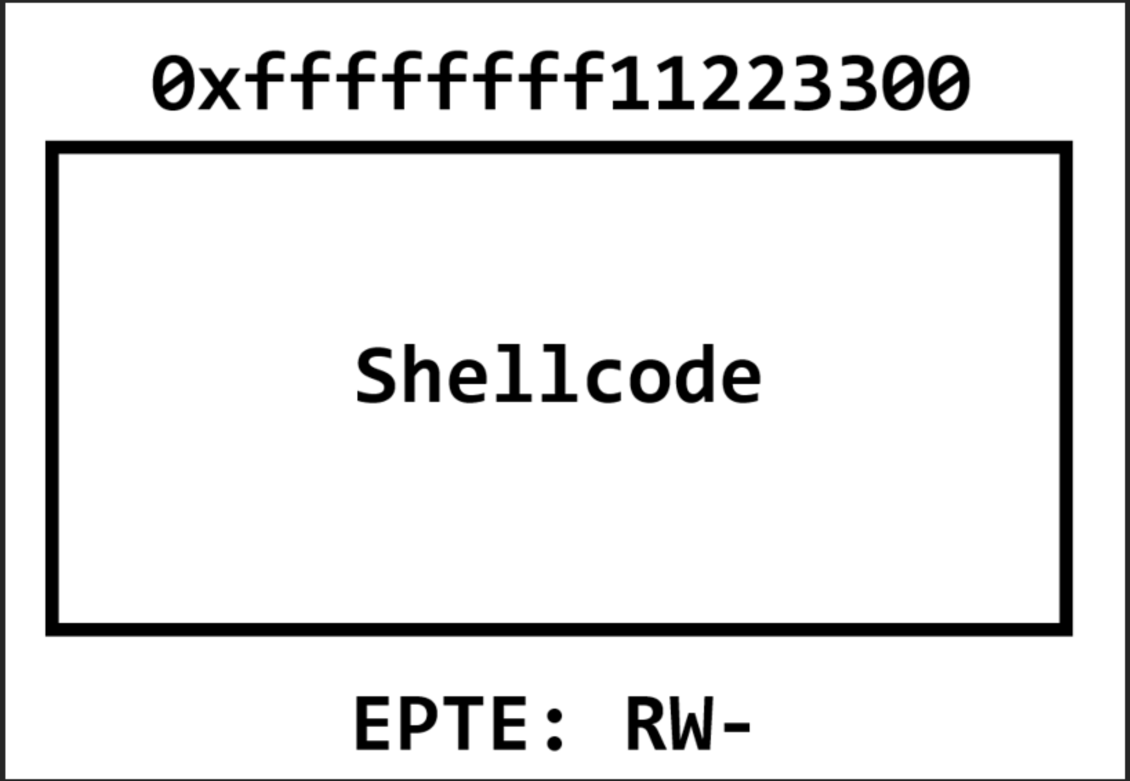
- How is this leveraged?
- Page table entries (PTEs) are stored in the kernel
  - PTEs contain metadata that describes page permissions (RWX, etc.)
    - If an attacker can locate the PTE corresponding to attacker-controlled memory, it can be manipulated to make read-only memory writable and executable as an example
- VTL 1 (secure mode) can combat this by configuring memory permissions to its liking
  - These permissions are stored in the extended page tables (EPTs)
    - The EPTs are managed by the hypervisor and are *immutable* from the rest of the OS
  - We can now use the EPTs to protect the PTE metadata!

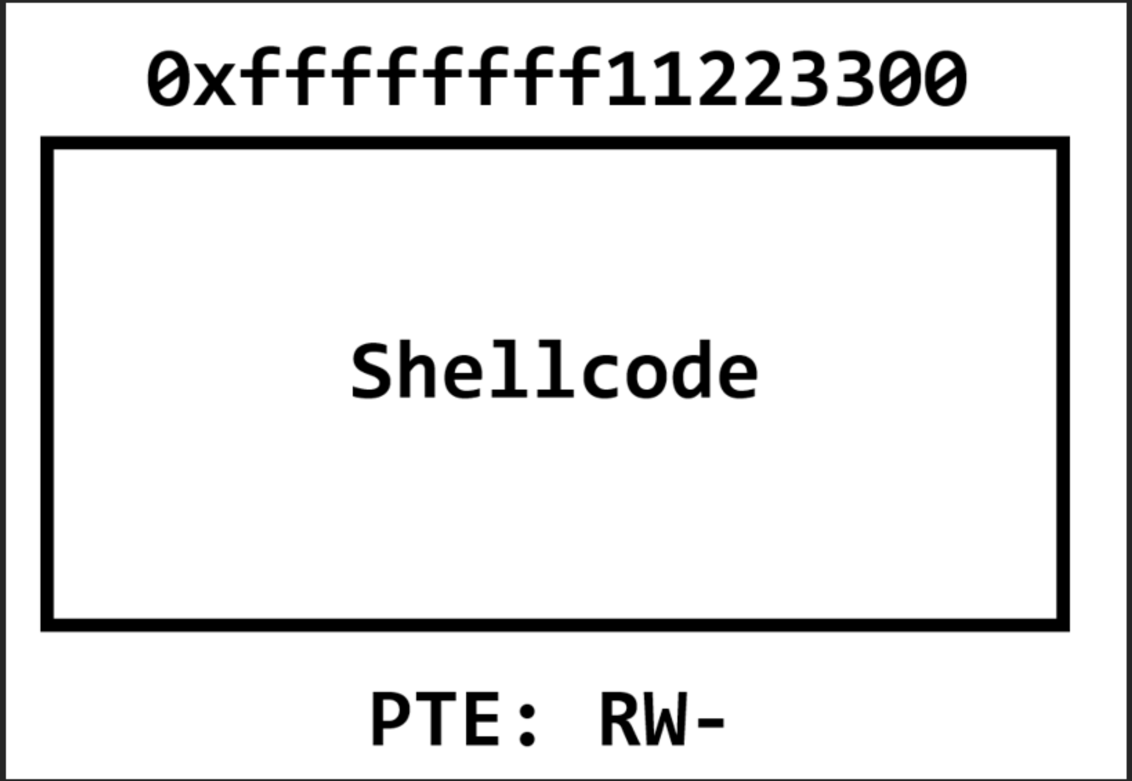
A Post-DEP-and-ASLR World

- When VTL 1 configures the extended page tables they are configured as W ^ X – meaning no memory is ever both able to be writable and executable (required for shellcode)
  - This “policy” is known as the mitigation Hypervisor-Protected Code Integrity (HVCI)
    - Let’s look at an example of why this is useful

A Post-DEP-and-ASLR World

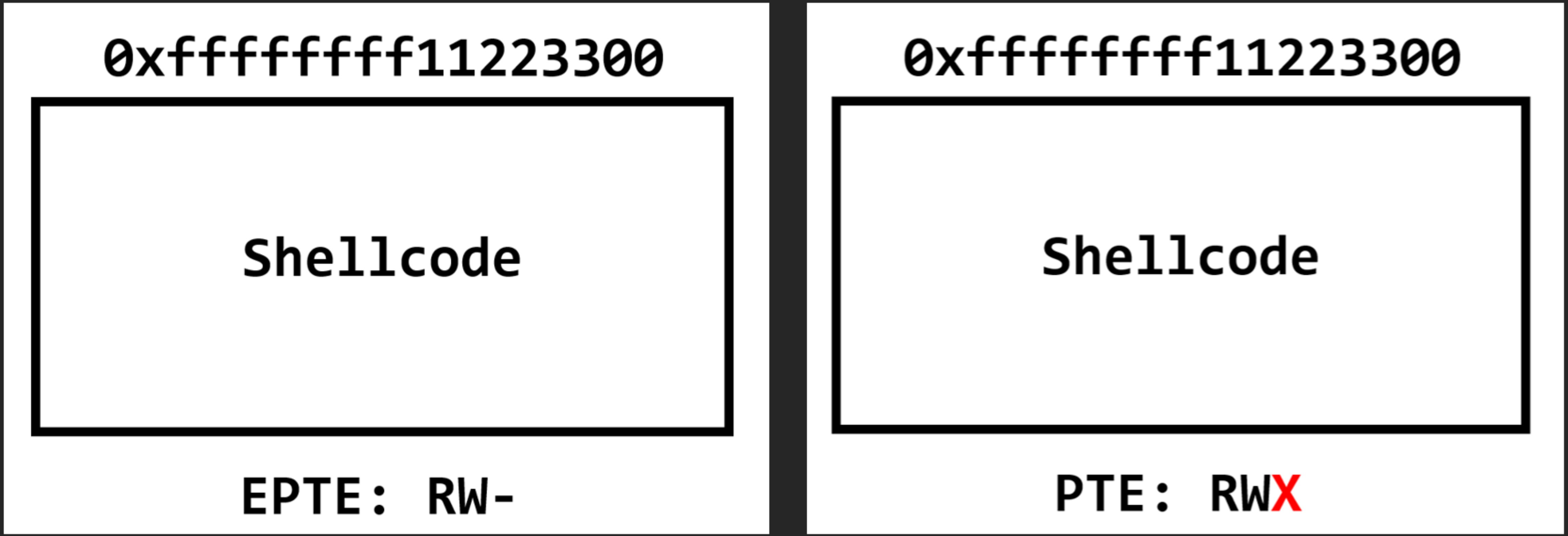
- Example





A Post-DEP-and-ASLR World

- Example:



A Post-DEP-and-ASLR World

- With HVCI enabled it is impossible to generate dynamically-created executable code in the kernel
    - This is because the "true" memory permissions are managed by the hypervisor and cannot be corrupted/tampered with even from the kernel
- Hence why HVCI is often called the Arbitrary Code Guard of the kernel

The “New-Era” of Exploitation

The "New-Era" of Exploitation

- ASLR/DEP made exploitation tedious, but didn't really "change" how exploitation occurs
  - We now just need to know the memory layout and use ROP to execute shellcode
- With the advent of CFI and W^X, we must adapt our exploitation techniques

The "New-Era" of Exploitation

- The first way we can go about this is just "playing by the rules of the mitigation"
  - Although ACG and HVCI won't let you use ROP to violate DEP, it is possible to build an entire payload in ROP

```
//
// ZwOpenProcess
//
write64(inHandle, retAddr, ntBase + 0x2bda97);              // 0x2bda97: pop rcx ; ret ; \x40\x59\xc3 (1 found)
write64(inHandle, retAddr + 0x8, &defenderprocHandle);      // HANDLE (to receive MsMpEng.exe process handle)
write64(inHandle, retAddr + 0x10, ntBase + 0x6398a1);       // 0x6398a1: pop rdx ; ret ; \x5a\x46\xc3 (1 found)
write64(inHandle, retAddr + 0x18, PROCESS_ALL_ACCESS);      // PROCESS_ALL_ACCESS
write64(inHandle, retAddr + 0x20, ntBase + 0x2f7161);       // 0x2f7161: pop r8 ; ret ; \x41\x58\xc3 (1 found)
write64(inHandle, retAddr + 0x28, &objAttrs);               // OBJECT_ATTRIBUTES
write64(inHandle, retAddr + 0x30, ntBase + 0x42b023);       // 0x42b023: pop r9 ; ret ; \x41\x59\xc3 (1 found)
write64(inHandle, retAddr + 0x38, &clientId);               // CLIENT_ID
write64(inHandle, retAddr + 0x40, ntBase + 0x73a941);       // 0x73a941: pop rax ; ret ; \x58\xc3 (1 found)
write64(inHandle, retAddr + 0x48, ntBase + 0x4140c0);       // nt!ZwOpenProcess
write64(inHandle, retAddr + 0x50, ntBase + 0xab4408);       // 0xab4408: jmp rax; \x48\xff\xe0 (1 found)
```

The “New-Era” of Exploitation

- Additionally, we can use data-only attacks
  - We can’t hijack control-flow of a target, nor can we execute shellcode – but nothing stops us from targeting crucial data structures!
- It is worthwhile to note that there are a few mitigations that target data-only corruption, like Kernel Data Protection (KDP)

The "New-Era" of Exploitation

- KDP works by protecting kernel data against data-only attack by "using the infrastructure as HVCI"
  - We can use the extended page tables to mark data regions of memory as read-only so that attackers can't tamper with said data
- Example: CI!g_CiOptions
  - Flag used to determine what code integrity options are enabled. Using a data-only attack, an adversary will try to patch this global variable to disable driver signature requirements, etc.
  - CI!g_CiOptions is now protected by KDP and is read-only via the EPTEs
    - Accomplished via nt!MmProtectDriverSection which requests the EPTE entry

```
if ( (g_CiOptions & 0x10) != 0 || KdDebuggerEnabled != 1 || KdDebuggerNotPresent )
  MmProtectDriverSection(&g_CiOptions, 0i64, 1i64);
```

The "New-Era" of Exploitation

- KDP currently comes in *static* and *dynamic* format*
  - CI!g_CiOptions is a static global variable and therefore is protected by static KDP
  - However, only memory allocated/placed in the "secure pool" is currently protected by dynamic KDP
    - What if we target *other* dynamic data, like a process object (EPROCESS), which isn't allocated in the secure pool?
- Example of corrupting EPROCESS?
  - Consider that most kernel exploits want to achieve privilege escalation
    - E.g., we want to become NT AUTHORITY\SYSTEM

The “New-Era” of Exploitation

- Instead of doing this via shellcode, it is possible to do this via a data-only attack
  - System process on Windows
    - Responsible for execution of most kernel-mode threads
    - Runs with SYSTEM privileges (obviously)
  - Each process is represented in the executive of the kernel as an EPROCESS object
    - EPROCESS.Token is a writable field that contains a process token
      - It is possible to use data-only to locate the EPROCESS object of the System process and the exploit process, and overwriting the exploit process Token with the System token

The "New-Era" of Exploitation

```
Command

0: kd> dt nt!_EPROCESS Token
   +0x4b8 Token : _EX_FAST_REF
0: kd> dx -g @$cursession.Processes.Where(p => p.Name == "System").Select(p => new { Name = p.Name, EPROCESS = &p.KernelO
=================================================================
=            = Name      = (+) EPROCESS          = Token               =
=================================================================
= [0x4]      - System    - 0xffffb903302aa040    - 0xffffcc8118c0804e  =
=================================================================
0: kd> dx -g @$cursession.Processes.Where(p => p.Name == "cmd.exe").Select(p => new { Name = p.Name, EPROCESS = &p.Kernel
=================================================================
=            = Name      = (+) EPROCESS          = Token               =
=================================================================
= [0x14d4]   - cmd.exe   - 0xffffb903386c60c0    - 0xffffcc811f25406b  =
=================================================================
0: kd> ep 0xffffb903386c60c0+0x4b8 0xffffcc8118c0804e
0: kd> g
```

```
C:\WINDOWS\system32\cmd.exe

Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\User>whoami
windev2202eval\user

C:\Users\User>whoami
nt authority\system
```

The “New-Era” of Exploitation

- Conclusion
  - Exploitation is less widespread because many of the new mitigations address the root problem
  - Exploitation has evolved and may be harder to detect as a side effect
    - Detecting RWX memory being executed from a heap allocation not backed by disk is easy to detect
    - A data-only attack (which is what we are forced to do now)? Not so much
  - Research has now shifted to other tradecraft such social engineering, C2 frameworks, etc. as a result
    - Dare I say this was year of the C2?
  - Lastly, Enable your exploit mitigations! Windows offers cutting edge mitigations that come for free with your OS
- Questions?