Understanding Windows Kernel Exploitation

Braden Hollembaek & Adam Pond

Motivation

- Got bugs, now what
- Check to see if older techniques still work on Win10 x64
  - Cesar Cerrudo “Easy Local Windows Kernel Exploitation” 2012
  - Mateusz ‘j00ru’ Jurczyk & Gynvael Coldwind “SMEP: What is it, and how to beat it on Windows” 2011
  - Cedric Halbronn “Exploiting CVE-2015-2426, and How I Ported it to a Recent Windows 8.1 64-bit”

What We Will Be Covering

- Discretionary Access Controls
  - Tokens and their place in exploitation
  - Security Descriptors and their place in exploitation
- Mandatory Access Controls
  - Integrity Levels and their place in exploitation

What We Will Be Covering

- Kernel Code Execution Prevention
  - Bypassing various protections to get our code running in the kernel
- Supervisor Mode Execution Prevention (SMEP)
  - Bypassing SMEP to execute code as kernel in userland

What We WIll Not Be Covering

- How to find vulnerabilities
  - We did a talk about this at DerbyCon last month
- What vulnerabilities look like
  - You can find lots of examples via bing

Assumptions Going Forward

- We have already found vulnerabilities that give us kernel reads and writes
- For demos, we wrote a custom kernel driver that gives us the ability to read and write arbitrary kernel memory
  - Not so far fetched

Goals

- Elevate our effective privileges from non-administrative user to Administrator or greater
- Keep the system in a stable state (avoid BSOD)
- Make the exploits reliable

Windows Security Model Overview

- Discretionary Access Controls
  - Tokens
  - Security Descriptors
- Mandatory Access Controls
  - Integrity Levels
- Kernel Protections
  - Protection Rings + SMEP
  - Code signing
  - IUM/SKM
  - *Guard: PatchGuard && DeviceGuard && CredentialGuard

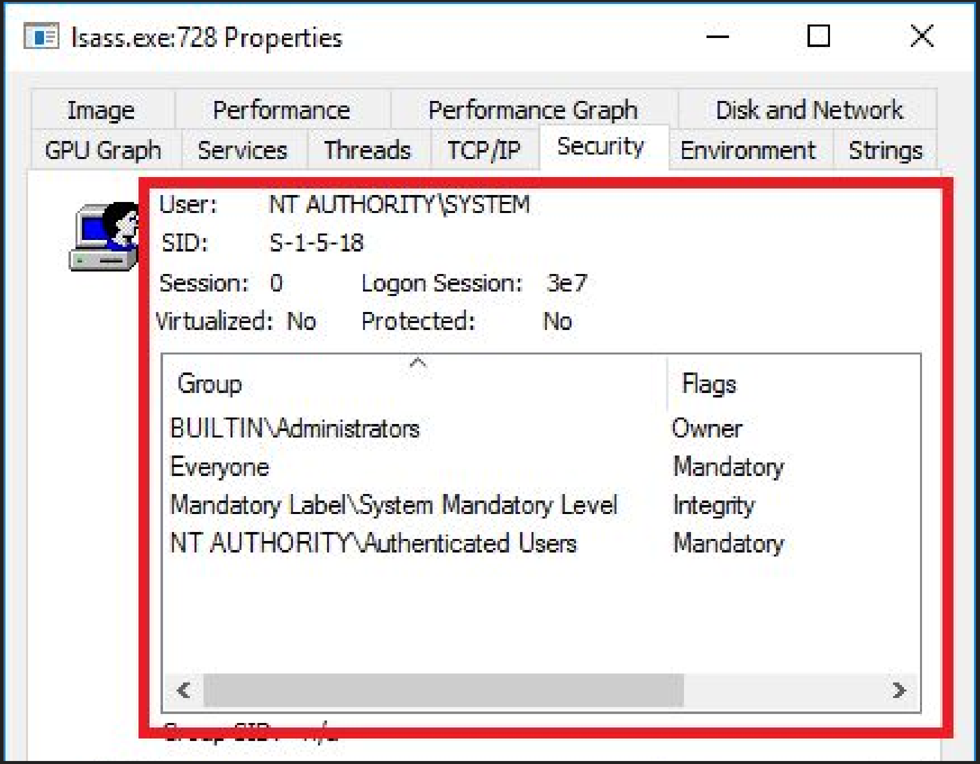
Discretionary Access Controls - Tokens

- Access Tokens
  - Describes security context of the process or thread
  - Used to determine whether or not we can access an object
  - Contains important identity and privilege information:
    - Security Identifiers (SIDs) identifying the user and their groups
    - Privileges indicating rights that the token bearer holds

Discretionary Access Controls - Tokens

- Security Identifiers (SIDs)
  - Used in tokens to identify the user and their groups
  - Used in security descriptors to identify the owner of the object
    - We will discuss security descriptors shortly
  - Used in Access Control Entries (ACEs) to grant/restrict access

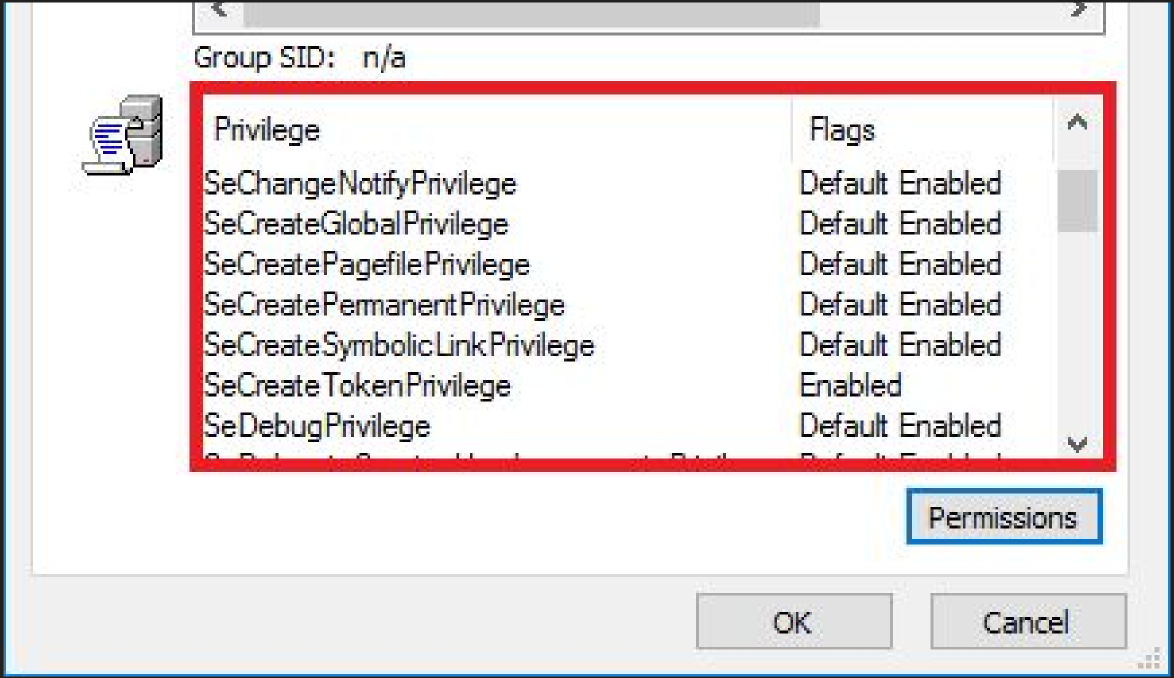
Discretionary Access Controls - Tokens



Discretionary Access Controls - Tokens

- Privileges
  - Used to grant access to system resources and system tasks
  - Some of the more interesting privileges:
    - SeTcbPrivilege - Act as part of the operating system/trusted computing base
    - SeTakeOwnershipPrivilege - Take ownership of an object
    - SeSystemEnvironmentPrivilege - Access environment variables in firmware
    - SeLoadDriverPrivilege - Load drivers
    - SeDebugPrivilege - Debug other processes
    - SeSecurityPrivilege - Control security audit logs

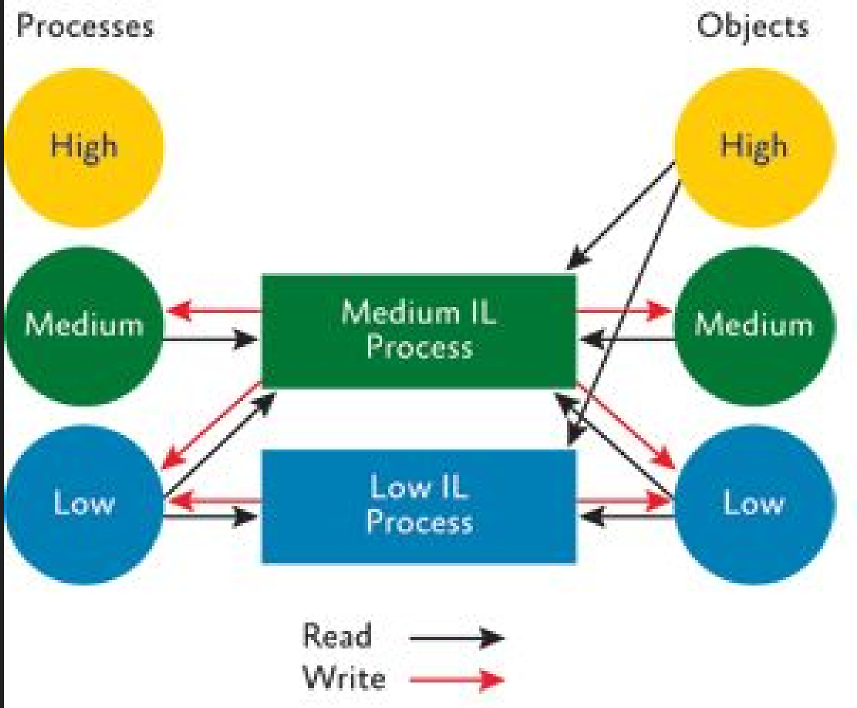
Discretionary Access Controls - Tokens



Mandatory Access Controls - Integrity Level

- Integrity Level
  - Restricts the access permissions of applications that are running under the same user account and that are less trustworthy
    - Useful in sandboxes, like the ones used by (some) web browsers
  - Levels: Untrusted, Low, Medium, High, System

Mandatory Access Controls - Integrity Level



https://technet.microsoft.com/en-us/library/2007.06.uac.aspx

Mandatory Access Controls - Integrity Level

- Represented as a SID in the token

Integrity level SID	Name
S-1-16-4096	Mandatory Label\Low Mandatory Level
S-1-16-8192	Mandatory Label\Medium Mandatory Level
S-1-16-12288	Mandatory Label\High Mandatory Level
S-1-16-16384	Mandatory Label\System Mandatory Level

https://msdn.microsoft.com/en-us/library/bb625963.aspx


LET'S GET IT ON!

Discretionary Access Controls - Tokens

- Targets for exploitation in your token
  - Simplest way: we can just snag the token from another process and associate it with our process
  - We then have the same privileges as that process
  - This also changes our integrity level

Discretionary Access Controls - Tokens

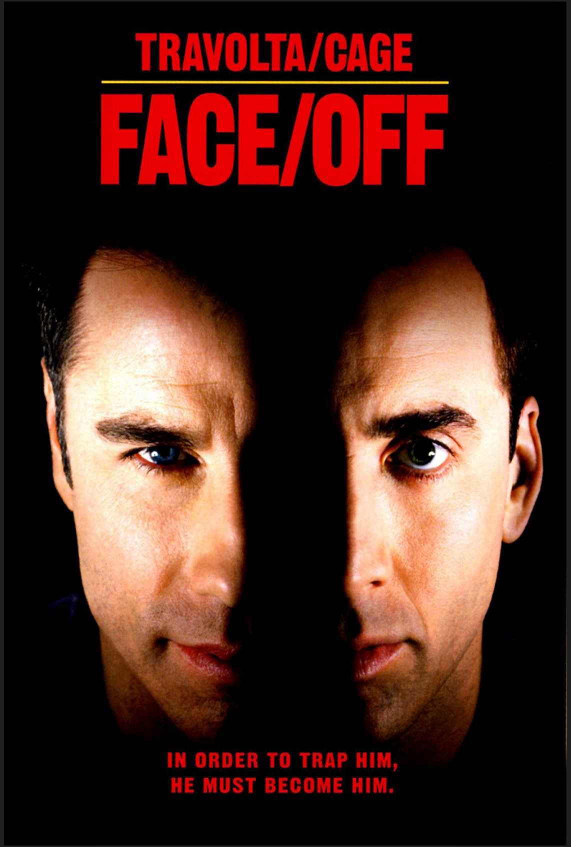
- Targets for exploitation in your token
  - In practice, we overwrite the pointer to our token with a pointer to a more powerful token
  - Found at an offset in the _EPROCESS kernel structure
  - Has changed since Cerrudo's paper described it for Win7

```
0: kd> dt nt!_EPROCESS -n Token
   +0x358 Token : _EX_FAST_REF
```

Discretionary Access Controls - Tokens

- Targets for exploitation in your token
    a. Find location of our eprocess struct in kernel memory
        i. We will discuss KASLR/kernel info leaks later
    b. Find location of one for a more privileged process, like lsass
    c. Overwrite our token pointer with the lsass token pointer

TOKEN SWAP DEMO



Discretionary Access Controls - Tokens

- Targets for exploitation in your token
  - Increase your privileges
  - Wouldn’t it be nice to have all the privileges?
  - Just a bitmask, so we can overwrite it with all 1’s

Discretionary Access Controls - Tokens

- Targets for exploitation in your token

```
0: kd> dt nt!_TOKEN -n Privileges

   +0x040 Privileges : _SEP_TOKEN_PRIVILEGES
0: kd> dt nt!_SEP_TOKEN_PRIVILEGES

   +0x000 Present            : Uint8B
   +0x008 Enabled            : Uint8B
   +0x010 EnabledByDefault   : Uint8B
```

Discretionary Access Controls - Tokens

- Targets for exploitation in your token
    a. Find location of our token structure in kernel memory
    b. Overwrite the privileges with 0xFFFFFFFFFFFFFFFF

PRIVILEGE INCREASE DEMO

Discretionary Access Controls - Security Descriptors

- What if we leave our token the same, and target another object
- Perhaps remove all the access controls for that target
- Then we can read the object, inject code into it (if it's a process), whatever

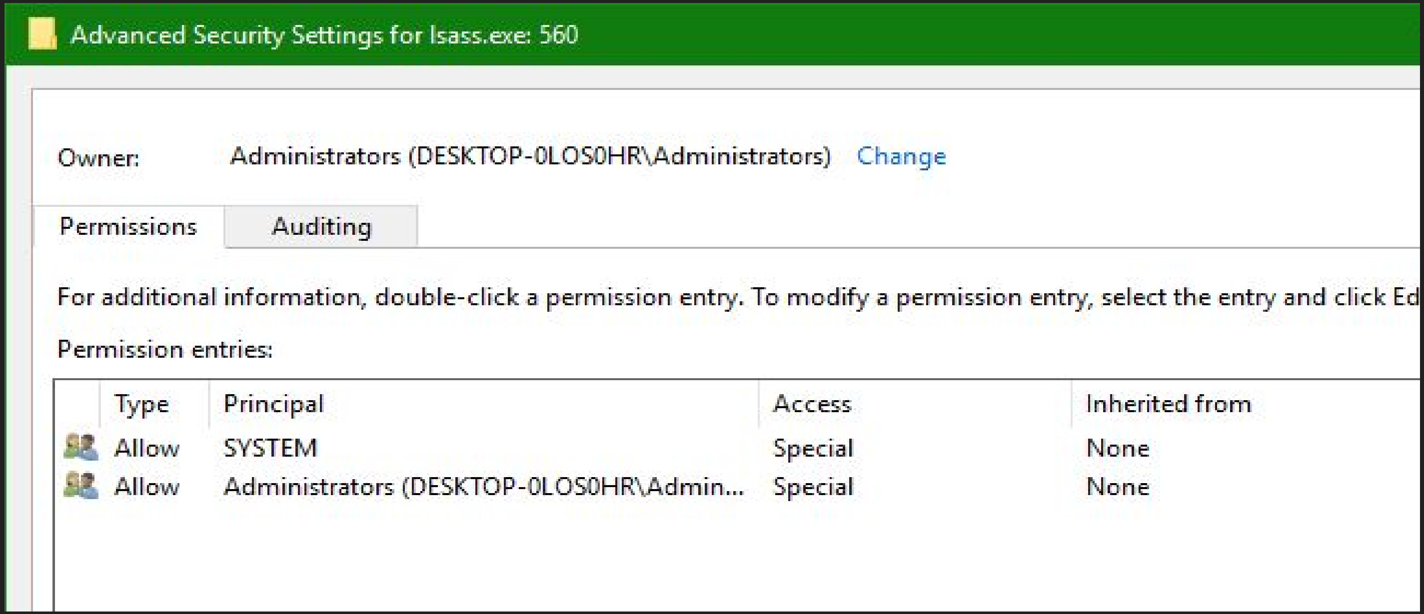
Discretionary Access Controls - Security Descriptors

- Security Descriptors
  - Contains security information about a securable object
    - Securable Objects: All named objects & some unnamed like processes/threads
    - Unsecurable Objects: Unnamed section objects,Win32k stuff (GUI windows), hardware registers, things like that

Discretionary Access Controls - Security Descriptors

- Security Descriptors
  - Contains Discretionary Access Control List (DACL)
    - Specifies which users and groups can access the object
  - Contains System Access Control List (SACL)
    - Controls generation of audit messages for access attempts

Discretionary Access Controls - Security Descriptors



Discretionary Access Controls - Security Descriptors

Discretionary Access Controls - Security Descriptors

- Targets for exploitation in a security descriptor
  - What if an object had no security descriptor?
  - Cerrudo's trick: Replace with NULL

Discretionary Access Controls - Security Descriptors

- Targets for exploitation in a security descriptor
  - Lives in _OBJECT_HEADER, which is just above our _EPROCESS struct

```
0: kd> dt nt!_OBJECT_HEADER
    ---snipped---
   +0x028 SecurityDescriptor : Ptr64 Void
   +0x030 Body             : _QUAD
```

Discretionary Access Controls - Security Descriptors

- Targets for exploitation with your security descriptor
  a. Find the pointer to the security descriptor for a target object
     i. We choose lsass in this case
  b. Overwrite the pointer with NULL
  c. We now have full access to lsass
     i. We could dump its memory and look for secrets
     ii. Inject code into it and elevate privileges that way
     iii. More easily steal its system token like in the previous attack

Discretionary Access Controls - Security Descriptors

```
2: kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

BAD_OBJECT_HEADER (189)
The OBJECT_HEADER has been corrupted
Arguments:
Arg1: ffffd184913e2750, Pointer to bad OBJECT_HEADER
Arg2: ffffd18490276f20, Pointer to the resulting OBJECT_TYPE based on the TypeIndex in the OBJECT_HEADER
Arg3: 0000000000000001, The object security descriptor is invalid.
Arg4: 0000000000000000, Reserved.
```

Discretionary Access Controls - Security Descriptors

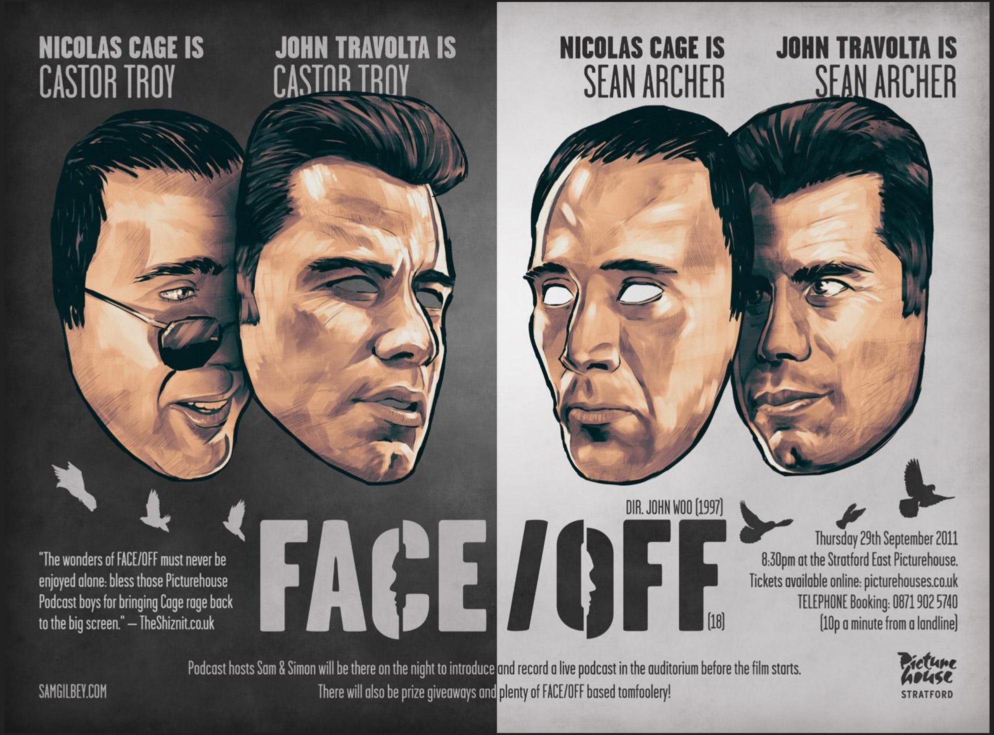
Discretionary Access Controls - Security Descriptors

- Vector is removed, checks now exist for NULL SD pointers
- That’s OK, we can just swap it with a security descriptor more favorable to us, similar to the token trick
    - explorer.exe is a good target
    - We could also just make an object with a wide open security descriptor and use that

Discretionary Access Controls - Tokens

- Targets for exploitation with your security descriptor
    a. Find the pointer to the lsass.exe security descriptor
    b. Find the pointer to the security descriptor of explorer.exe
    c. Overwrite lsass security descriptor pointer with explorer security descriptor pointer
    d. We now have full access

DACL SWAP DEMO



Quick Recap

- Overwrite pointer to our low privilege token with pointer to a high privilege token
  - Still works. Offset of token has changed, but no big deal
- Overwrite privilege “Enabled” field in token
  - Still works as advertised
- Overwrite security descriptor pointer with NULL
  - No longer works, will bugcheck
  - Can overwrite pointer with a more favorable SD pointer though

Kernel Protections Transition

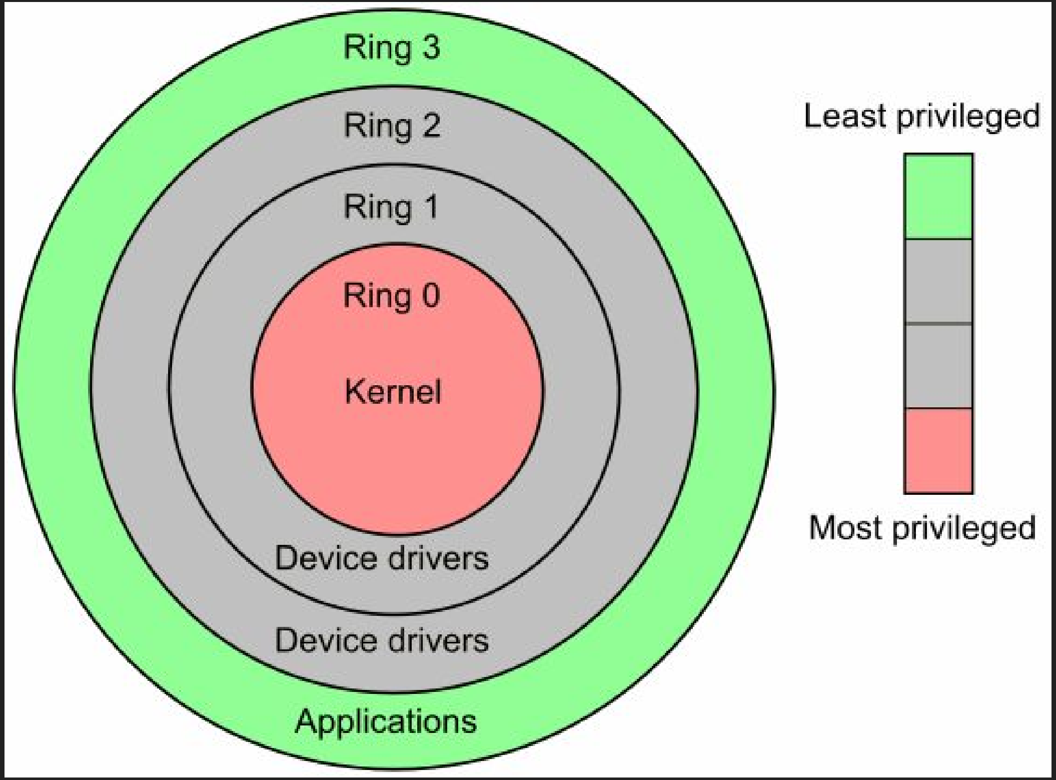
- Cut out the middleman and just execute as the kernel

Kernel Protections

- Protection Rings
- NX/DEP
- KASLR
- Authenticode
- PatchGuard
- IUM/SKM
- DeviceGuard & CredentialGuard
- SMEP

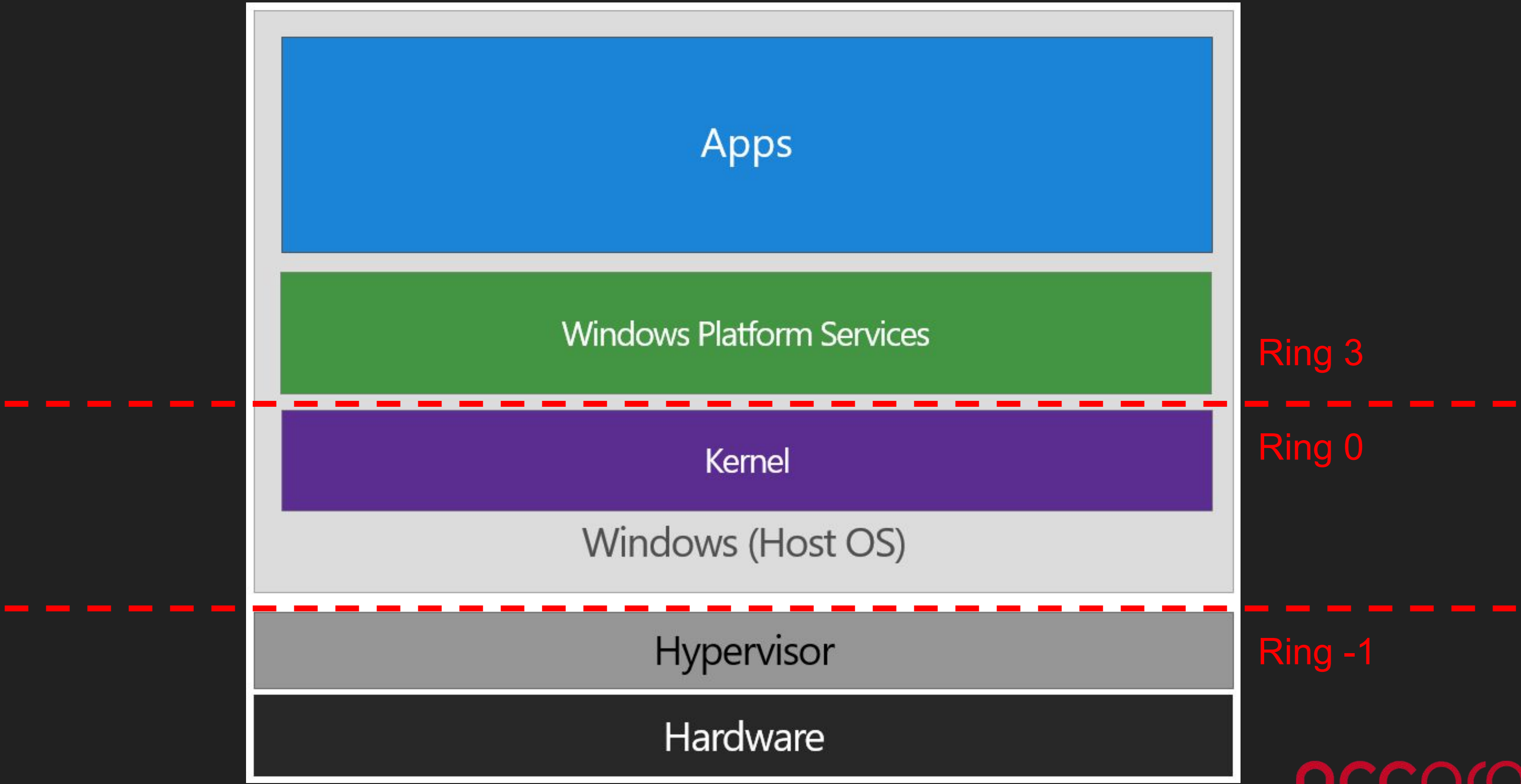
Protection Rings

Protection Rings



https://fawzi.files.wordpress.com/2009/05/four-protection-rings.png

Protection Rings



https://msdnshared.blob.core.windows.net/media/TNBlogsFS/prod.evol.blogs.technet.com/CommunityServer.Blogs.Components.WeblogFiles/00/00/01/03/19/metablogapi/image_7598A837.png

Protection Rings

- Restrict access to resources based on privilege level
    - Current privilege level stored in cs register
- Ring 3
    - User mode
- Ring 0
    - Kernel mode
    - Device drivers
- Ring -1
    - Hypervisor
- Ring -2
    - System Management Mode (SMM)

NX/DEP

- Data Execution Prevention (DEP)
    - Enabled by default in kernel around Windows Vista x64
    - Memory marked by NX bit
        - Two main kernel pools
            - NonPaged Pool/NonPaged Pool Nx
                - Always in physical pages
                - Prior to win 8 was executable
            - Paged Pool
                - Allows paging to disk

Exploitable?

- Some drivers still use executable pool
- ROP
  - Must find ROP gadgets
  - Deal with KASLR

KASLR

- Kernel Address Space Layout Randomization (KASLR)
  - Enabled by default in kernel starting with Win 7 x64
  - Randomizes objects/modules location
  - Bits of entropy dependent on OS
  - Makes using a fixed address harder
    - Like with ROP
    - Or write-what-where

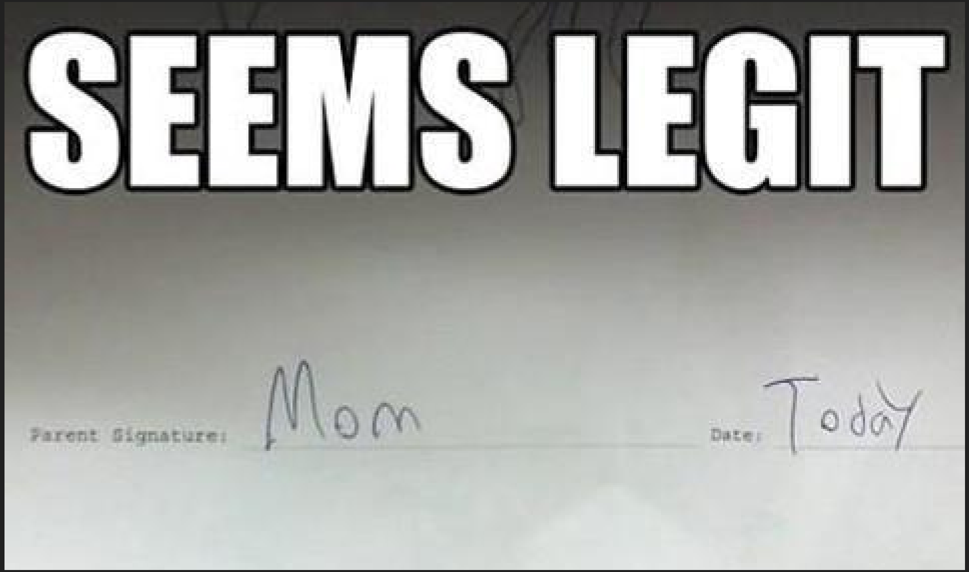
Windows 8		
32-bit	64-bit	64-bit (HE)
8	8	24
17	17	33
8	8	24
8	17	17
8	17	17
8	17*	17*
8	19*	19*
8	8	24

https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

Exploitable?

- Information leaks
    - Undocumented APIs
        - NtQuerySystemInformation
            - This is what we used earlier
        - Can only be called by medium+ integrity processes now
    - Traditional C-style infoleaks, not kernel specific
    - Drivers, particularly third-party, often leak kernel addresses
        - https://support.lenovo.com/us/en/product_security/len_6027

Authenticode

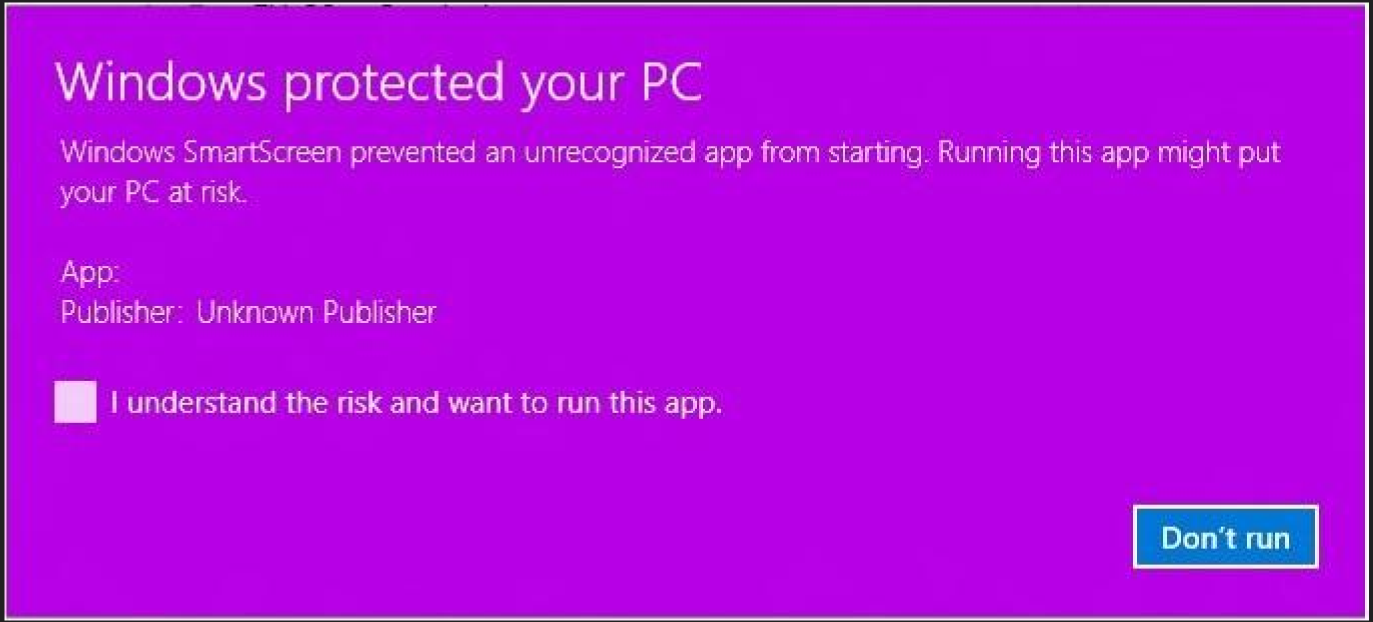


Authenticode

- Code signing
  - Signcode.exe - old school, separate private key + publisher cert
  - Signtool.exe - modern, uses .pfx file
  - All win 10 kernel drivers must be signed by WHDCDP after v1607
  - SHA-1 signed certs deprecated as of 2/14/17

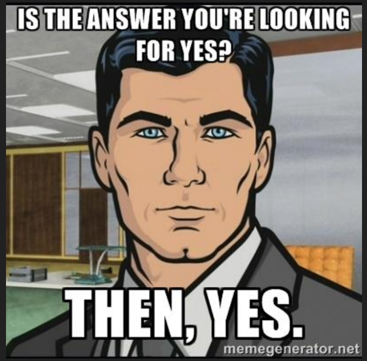
Authenticode

Authenticode



Exploitable?

- Code signing checks maintained by global vars
  - These vars are checked by PatchGuard (will discuss next)
    - Could disable PatchGuard
  - Or map driver functionality into kernel with buffer of shellcode
    - TDL (Turla Driver Loader)

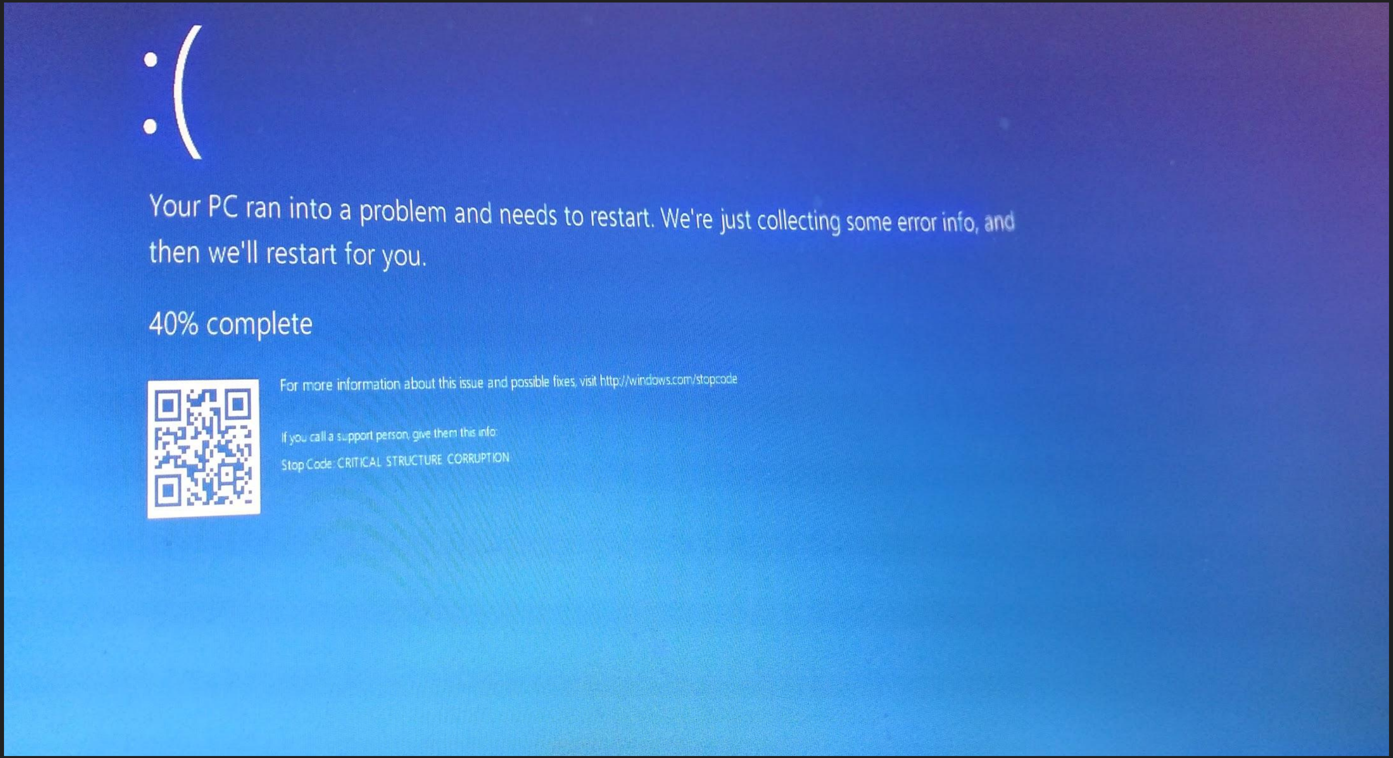


PatchGuard

PatchGuard

- Integrity check of various vars/registers/objects/structures
  - Released 2005
  - Does not apply to 32bit Windows
  - Created to prevent A/V from hooking kernel
    - Side effect of making exploits harder
  - If checksum fails...

PatchGuard


:(
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.
40% complete
For more information about this issue and possible fixes, visit http://windows.com/stopcode
If you call a support person, give them this info:
Stop Code: CRITICAL STRUCTURE CORRUPTION

Exploitable?

- PatchGuard implemented in kernel
  - Patch the PatchGuard
    - Several features make this more challenging
      - Symbol stripping
      - Obfuscation
      - Anti-debug
  - Modify protected var/structure/reg, run exploit, modify back
  - Or don’t, turns out PatchGuard can take a while to notice

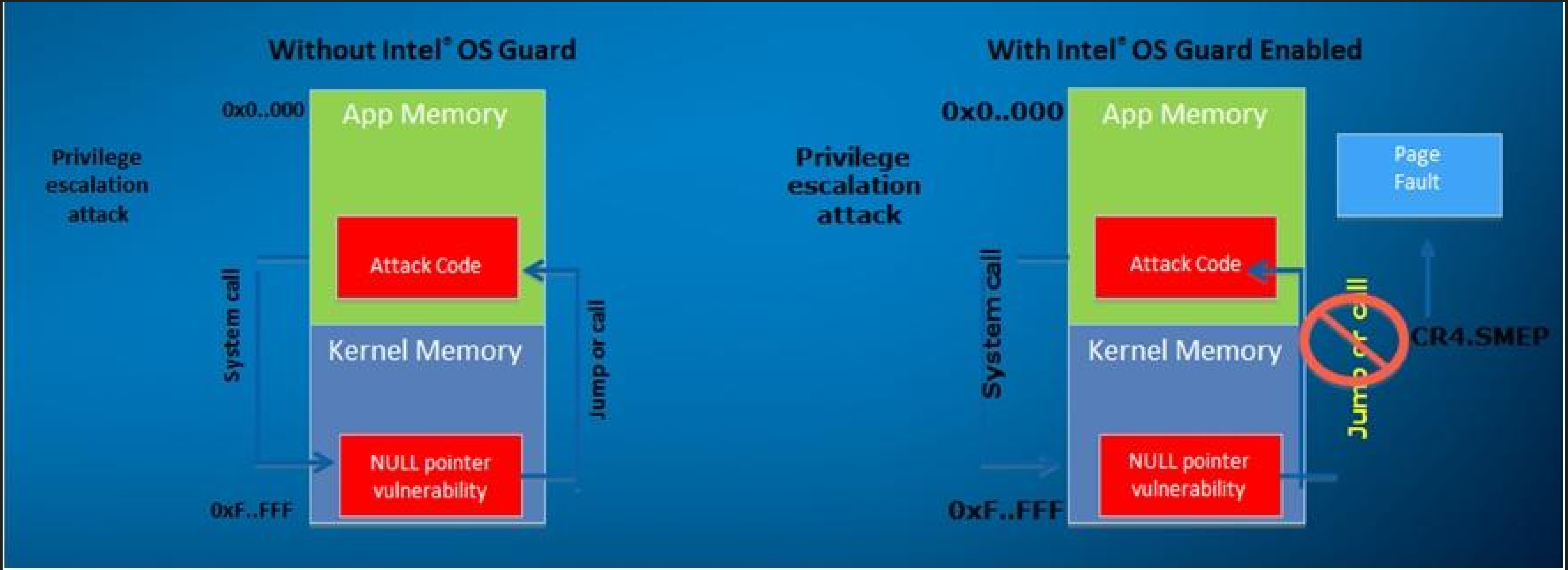
SMEP - Supervisor Mode Execution Prevention

SMEP

- Supervisor Mode Execution Protection (SMEP): Prevents kernel from executing code in userland
  - CR4 register holds SMEP status
    - Flip 20th bit to disable/enable SMEP
  - Supported on intel CPU's since IvyBridge
  - Former bypass methods
    - Paging table abuse* - Randomized now
    - nt!MmUserProbeAddress - PatchGuard

*https://www.coresecurity.com/system/files/publications/2016/05/Windows%20SMEP%20bypass%20U%3DS.pdf

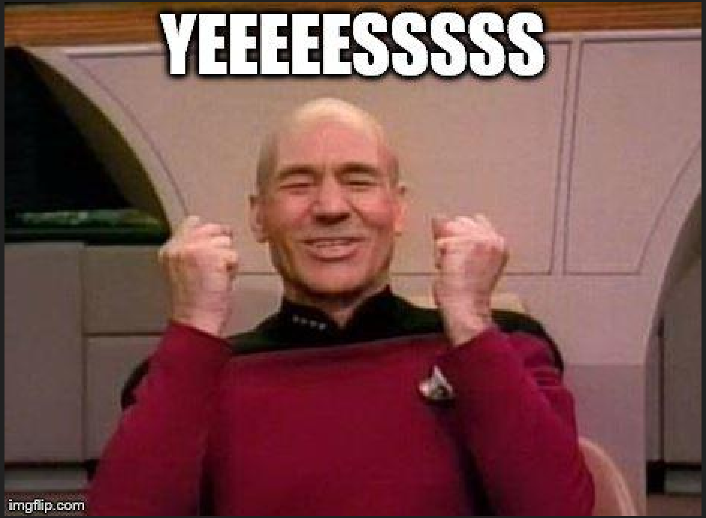
SMEP



https://software.intel.com/sites/default/files/6_9.jpg

Exploitable?

- SMEP controlled by CR4
  - Modify 20th bit of CR4
    - Can’t modify from userland
      - Need kernel to turn it off
      - PatchGuard strikes again
        - Restore original CR4 state after exploit



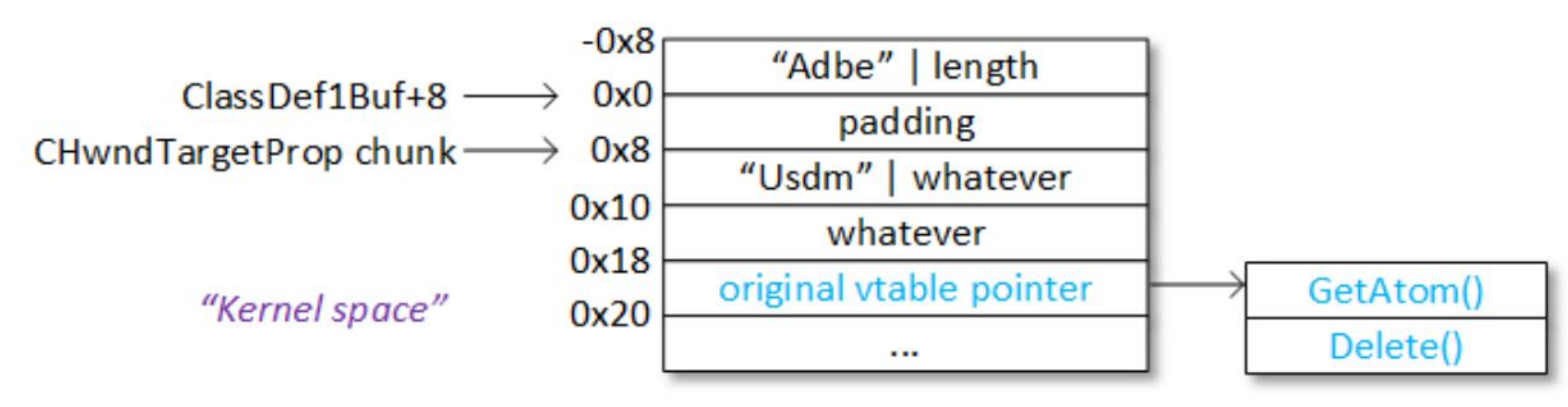
nccgroup

Case Study: CVE-2015-2426

- OpenType Font Driver Vulnerability
  - ATMFD.DLL
    - Adobe Type Manager Font Driver
  - Buffer overflow in kernel object
    - Object allocated with EngAllocMem a Win32k.sys call
    - Fails to check for length of 0
    - Copies 0x20 bytes anyways

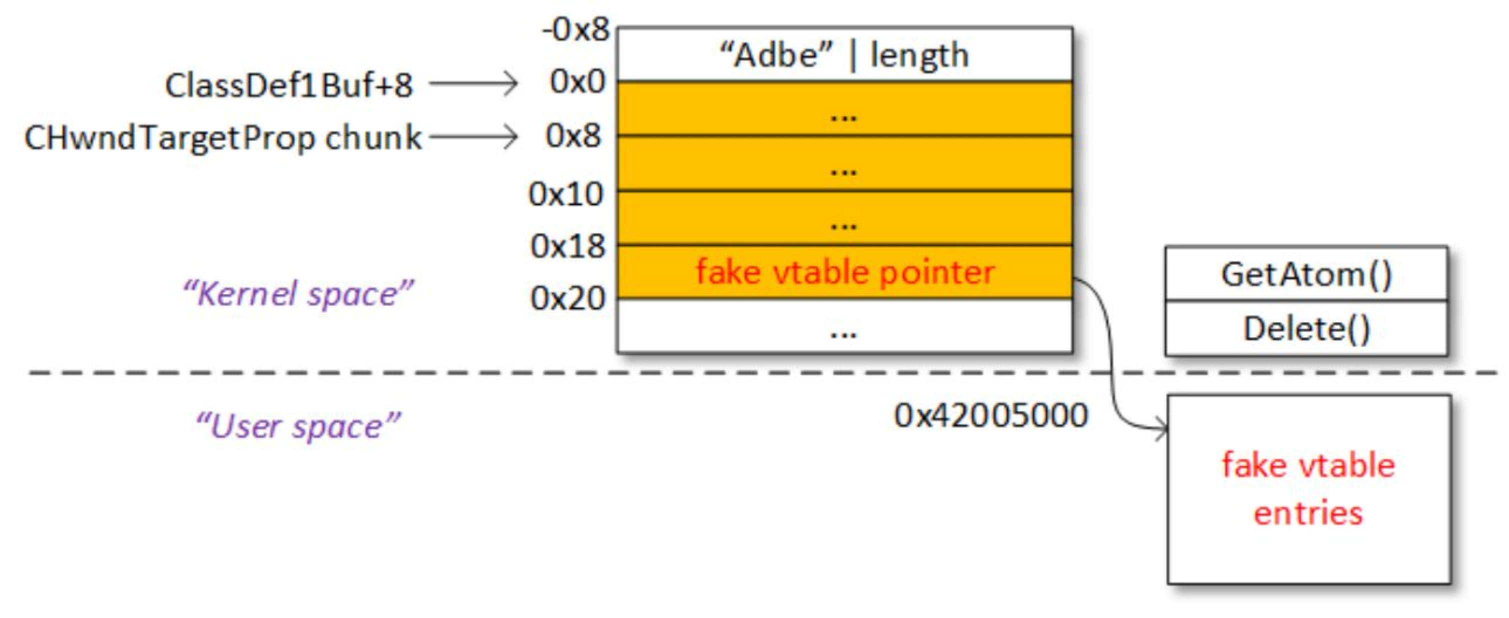
nccgroup

Case Study: CVE-2015-2426



https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/09/2015-08-28_-_ncc_group_-_exploiting_cve_2015_2426_-_release.pdf

Case Study: CVE-2015-2426



https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/09/2015-08-28_-_ncc_group_-_exploiting_cve_2015_2426_-_release.pdf

Case Study: CVE-2015-2426

- Exploit plan:
    a. Spray pool full of same sized kernel objects
    b. Create userland vtable and write address with overflow
    c. Free an object from middle of pool
    d. Call object method from userland
    e. ROP to bypass SMEP
    f. Redirect execution to userland shellcode

SMEP Bypass Demo

- Turn off SMEP
- Redirect kernel to userland shellcode
  - Steal SYSTEM token
  - Overwrite current process token with SYSTEM token
  - system(‘cmd’)

SMEP Bypass Demo

- Turn off SMEP
  a. If we can execute code directly
     - mov rax, cr4
     - btr rax, 0x14 // flip 20th bit
     - mov cr4, rax // disable SMEP
  b. Otherwise we got to ROP
     - How to find gadgets?

SMEP Bypass Demo

- We can use Windbg to find SMEP ROP gadget

```
kd> u nt!KiConfigureDynamicProcessor+0x33
nt!KiConfigureDynamicProcessor+0x33:
fffff802`603f678b 0f22e0          mov     cr4,rax
fffff802`603f678e 4883c428        add     rsp,28h
fffff802`603f6792 c3              ret
```

SMEP Bypass Demo

- Steal SYSTEM token
    a. Since we are executing in Ring 0 let’s use the _KPCR
    b. Locate the _EPROCESS structure
    c. Enumerate all processes until we find one with SYSTEM token
    d. Overwrite that our process token with SYSTEM token

SMEP Bypass Demo

- GS[0] should point to _KPCR
  a. Cannot be read
  b. Windbg can be confusing

```
kd> dg gs
                                                    P Si Gr Pr Lo
Sel        Base              Limit          Type    l ze an es ng Flags
---- ----------------- ----------------- ---------- - -- -- -- -- --------
002B 00000000`00000000 00000000`ffffffff Data RW Ac 3 Bg Pg P  Nl 00000cf3
```

SMEP Bypass Demo

- We can read certains offsets of gs though
    a. Grab the address of _KPCR

```
kd> !pcr
KPCR for Processor 0 at fffff8026035e000:
```

SMEP Bypass Demo

- Determine the offset to the _KPRCB

```
kd> dt nt!_KPCR fffff8026035e000
   +0x000 NtTib            : _NT_TIB
   +0x000 GdtBase          : 0xfffff802`61ddb000 _KGDTENTRY64
   +0x008 TssBase          : 0xfffff802`61ddc070 _KTSS64
   ...
   ...

   +0x180 Prcb             : _KPRCB
```

SMEP Bypass Demo

- Find the offset of current thread
    a. _KPCR + 0x180 = _KPRCB
    b. _KPRCB + 0x8 = CurrentThread
    c. If gs[0] = _KPCR then gs[0x188] = CurrentThread

```
kd> dt nt!_KPRCB fffff8026035e000+0x180
   +0x000 MxCsr            : 0x1f80
   +0x004 LegacyNumber     : 0 ''
   +0x005 ReservedMustBeZero : 0 ''
   +0x006 InterruptRequest : 0x1 ''
   +0x007 IdleHalt         : 0x1 ''
   +0x008 CurrentThread    : 0xfffff802`603d9940 _KTHREAD
```

SMEP Bypass Demo

- Then we need offset to _EPROCESS
    a. CurrentThread + 0x98

```
kd> dt nt!_KTHREAD 0xfffff802`603d9940
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : (null)
   +0x020 QuantumTarget    : 0x791ddc0
   +0x028 InitialStack     : 0xfffff802`61de3c90 Void
  ...
  ...
  ...
+0x098 ApcState          : _KAPC_STATE
```

SMEP Bypass Demo

- Then we need offset to _EPROCESS
    a. _KAPC_STATE + 0x20
    b. _EPROCESS = CurrentThread+0x98+0x20 = CurrentThread+0xb8

```
kd> dt nt!_KAPC_STATE 0xfffff802`603d9940+0x98
   +0x000 ApcListHead      : [2] _LIST_ENTRY [ 0xfffff802`603d99d8 - 0xfffff802`603d99d8 ]
   +0x020 Process          : 0xffff8784`e1454440 _KPROCESS
```

SMEP Bypass Demo

- Next we need offset to _EPROCESS.ActiveProcessLinks and _EPROCESS.UniqueProcessId

```
kd> dt nt!_EPROCESS 0xffff8784`e1454440
   +0x000 Pcb                : _KPROCESS
   +0x2d8 ProcessLock        : _EX_PUSH_LOCK
   +0x2e0 RundownProtect     : _EX_RUNDOWN_REF
   +0x2e8 UniqueProcessId    : 0x00000000`00000004 Void
   +0x2f0 ActiveProcessLinks : _LIST_ENTRY [ 0xffff8784`e2498330 - 0xfffff802`6031b470 ]
```

SMEP Bypass Demo

- Finally we need the offset to the _EPROCESS.Token

```
kd> dt nt!_EPROCESS 0xffff8784`e1454440
   +0x000 Pcb              : _KPROCESS
   +0x2d8 ProcessLock      : _EX_PUSH_LOCK
   +0x2e0 RundownProtect   : _EX_RUNDOWN_REF
   +0x2e8 UniqueProcessId  : 0x00000000`00000004 Void
   ...
   …
 +0x358 Token              : _EX_FAST_REF
```

SMEP Bypass Demo

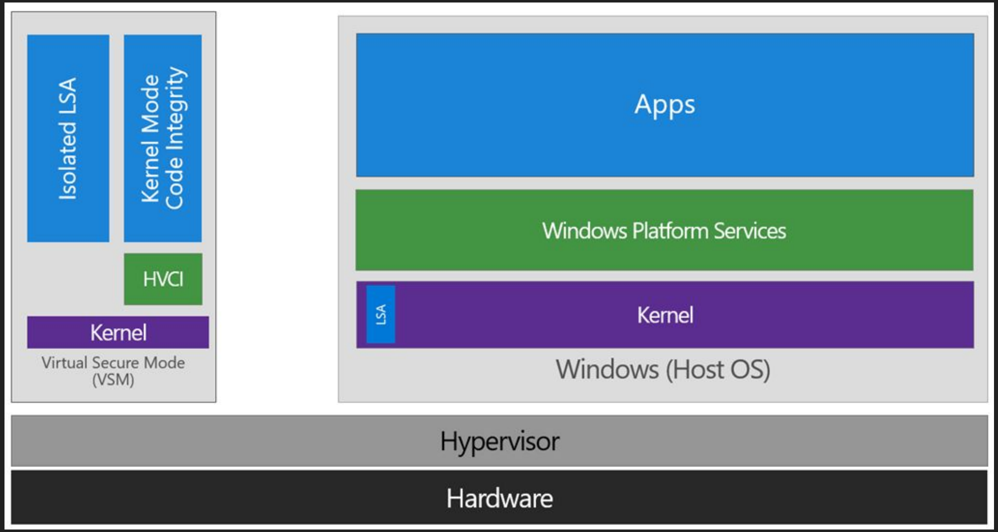
- With all the offsets obtained we are ready to roll
    a. Current thread offset gs:[0x188]
    b. _EPROCESS offset [current thread+0xb8]
    c. Token offset [_EPROCESS+0x358]
    d. ActiveProcessLinks offset [_EPROCESS+0x2f0]
    e. UniqueProcessId offset [_EPROCESS+0x2e8]

SMEP Bypass Demo

IUM/SKM

IUM/SKM

- Two security features added
  - DeviceGuard & CredentialGuard
    - Isolated in special processes called Trustets inside the new VSM



https://msdnshared.blob.core.windows.net/media/TNBlogsFS/prod.evol.blogs.technet.com/CommunityServer.Blogs.Components.WeblogFiles/00/00/01/03/19/metablogapi/image_1DF21D50.png

Exploitable?

- Couldn’t test DeviceGuard or CredentialGuard
  - Laptops too old, hardware didn’t meet specifications

Future Work

- Investigate Hyper-V/Virtualization based security
- Investigate Windows Subsystem for Linux

Thanks

- Thanks to the NCC Group research team, including Joel St. John and Jesse Burns
- Thanks to Nicolas Guigo for providing tech support
- Thanks to ToorCon for the opportunity to share with y’all

References and Additional Reading

- Cerrudo paper that the first few demos were based on: https://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf
- Further reading on KASLR leaks from Alex Ionescu's talk: https://recon.cx/2013/slides/Recon2013-Alex%20Ionescu-I%20got%2099%20problems%20but%20a%20kernel%20pointer%20ain't%20one.pdf
- White paper by Cedric Halbronn the case study is based on: https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/09/2015-08-28_-_ncc_group_-_exploiting_cve_2015_2426_-_release.pdf
- Additional reading on IUM/SKM from Ionescu's talk: http://www.alex-ionescu.com/blackhat2015.pdf
- J00ru's Paper about SMEP bypass: http://j00ru.vexillium.org/?p=783
- MSDN article on DeviceGuard/CredentialGuard: https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/
- Exploit Mitigation Improvements on Windows 8 Ken Johnson Matt Miller: https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf