SECURITY OVER-ENGINEERING

Caches of caches of chaos

BIO

- Reverse engineered Windows kernel since 1999
  - Became lead kernel developer for ReactOS Project
- Interned at Apple for a few years
  - Worked on "Core Platform Team" – firmware, boot loader, kernel, drivers
- Co-author of Windows Internals 5th and 6th Edition
- Founded Winsider Seminars & Solutions Inc.,
  - Provides services and Windows Internals training for enterprise/government
- Now Chief Architect at CrowdStrike, a security company
  - Architecting a to security solution that has deep ties to the operating system

INTRODUCTION

- Windows has been one of the most secure general purpose operating systems from the beginning
  - C2 certification with US Government – EAL Level 3 internationally
- Each release has introduced additional security mechanisms and features
  - Jobs in Windows 2000
  - DEP/NX, Software Restriction Policies (SRP), Callback Access Control in XP
  - Blocking \Device\PhysicalMemory and PatchGuard in Server 2003
  - BitLocker, UIPI, Code Integrity, ASLR, Service SIDs, Protected Processes in Vista
  - AppLocker in Windows 7
  - AppContainer, User-Mode Code Integrity, ELAM, SecureBoot in Windows 8
  - Protected Process Light in Windows 8.1

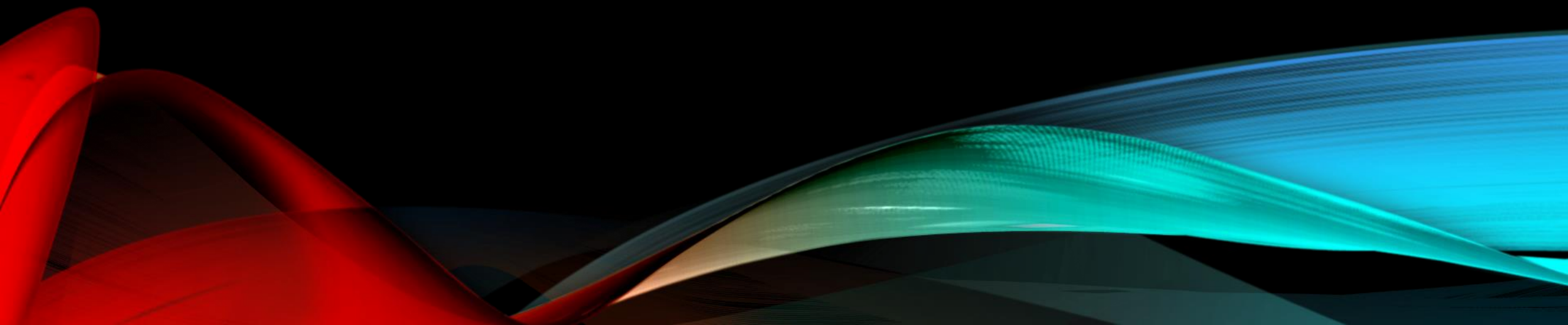
INTRODUCTION

- I see three distinct phases of security design/concerns
- Phase 1 was about protecting the kernel boundary against user applications (NT -> XP)
- Phase 2 was about protecting the kernel boundary even against administrators (2003->Windows 7)
- Phase 3 is about protecting the device and applications against administrators (Windows 8-ongoing)
- Initially, security was driven by government standards, hardware and data integrity, and mitigation against security attacks
- Security is increasingly being driven by DRM and hardware control demands, especially in the mobile market

INTRODUCTION

- We have to keep in mind that Windows 8.1 is still Windows NT (6.3)
- The first release of NT was in 1993 – development started in 1989.
- In other words, we are running a 25 year old operating system!
- What was considered important 25 years ago is not necessarily important today
  - And vice-versa!
- Assumptions made by core system components can be violate security assertions that are expected by new code
- Most of the people who wrote the code have long left Microsoft
  - Except Dave Cutler ☺
- Emergent behavior occurs (deadly to security)

WINDOWS SECURITY DESIGN



WINDOWS 8.1 DEVICE SECURITY

- Like many other vendors, Microsoft wanted to have control over mobile devices running Windows RT
- AppContainer provides iOS/Android-like entitlement system and generic application sandboxing
- Code Signing provides strong guarantees against unofficial code but also flexibility for OEM/Partner/3rd party integration
- Software Licensing manages “sideloading” permissions and Enterprise deployment
- Secure Boot provides strong root chain of trust from the firmware up
- Code Signing and Secure Boot also provide device identification and personalization

UEFI AND SECURE BOOT

- UEFI is modern replacement of BIOS
  - Collaboration between multiple industry partners
- First UEFI systems were Itanium (IA64) – Windows Server 2003 & EFI 1.12
- Apple used EFI for Intel Macintosh from the start
- All new Windows 8/RT machines are UEFI
  - Windows RT was first big deployment of UEFI for ARM Platform
- Secure Boot is UEFI 2.3.1 Errata C Extension for UEFI that provides Authenticode support
  - Enforces signing of boot firmware, option ROMs, OS boot code, …
  - Enforces signing and protection of critical variables
- All new Windows 8/RT machines are UEFI

CHAIN OF TRUST

- Secure Boot provides 4 key control mechanisms/databases:
  - PK is the Platform Key from the vendor. It can turn off Secure Boot & update KEK
  - KEK is the Key Enrollment Key from Microsoft. It can update the db and dbx
  - db is the Signature Database. Code must be signed with a key from here to run
  - dbx is the Revoked Signature Database. Code signed with a key here will not run
- By trusting that all boot code is signed correctly, when Windows Boot Manager (bootmgr) runs, the system should be in a known "clean" state
  - No more bootkits!
- Windows Boot Manager enforces signing of Windows Boot Loader (winload)
- Windows Boot Loader enforces signing of Kernel (ntoskrnl)
- Kernel enforces signing of drivers and applications

AUTHENTICODE CODE SIGNING

- Code Signing in UEFI and Windows is based on Portable Executable (PE) format's Authenticode Standard
- Original implementation calls for an embedded certificate in the file, along with the correct PE header information to point to it
- File is hashed, hash is compared with the one in the certificate, and the certificate is signed and issued by a trusted root authority (CA)
- For size optimization, catalog files (.cat) can be used
  - Catalog file contains the hashes of multiple executables (nt.cat has all of Windows, for example)
  - Catalog file is then signed and has embedded certificate
  - Binaries only have a hash that is compared to the catalog entry's hash

CODE INTEGRITY (CI)

- Code Integrity (Ci.dll) is a Kernel Mode Library that enforces signing on Windows
  - Introduced in Windows Vista, as part of Kernel Mode Code Signing (KMCS) Guidelines
- All x64 Drivers on Windows Vista+ **must** be signed with an Authenticode certificate
  - CI does not use the system certificate store – administrator cannot override/force launch of driver by importing a new trusted CA
  - Can only be partially disabled by booting into test-signing mode – but image must still be signed with a valid test certificate
- In Windows 8.1, if Secure Boot is enabled, even x86 drivers must be signed!
- Windows 8 also introduced User Mode Code Signing (UMCI)…

UMCI

- In Windows 8, UMCI is only enabled for Windows RT
- Prevents execution of unsigned code in user-mode as well (using the same certificate chain as KMSC)
  - Also used to check DLL loads inside signed applications
- UMCI does allow execution of unsigned "AppContainer" applications…
  - But Microsoft signs all binaries on the Windows Store
  - So this is only useful if sideloading is enabled (and/or a developer license is installed)
- UMCI implements "signing levels" as well
  - Authenticode Signed vs Microsoft Signed vs Windows Signed vs Windows TCB Signed
  - Levels are used to create strong boundaries even between signed components

UMCI IN WINDOWS 8.1

- In Windows 8.1, UMCI is also enabled on desktop devices
- But no longer applied blindly to block all unsigned desktop applications
  - Instead, kernel determines a "required signing level" for the binary…
  - … and CI determines if a signature exists, and if the signature is eligible for the required signing level
- On RT devices, the required signing level is hardcoded to "8", which is the Microsoft level
  - Means that –some- Microsoft signature is required to run desktop code
- On desktop, it's hardcoded to "0", but another mechanism can cause the "required signing level" to be raised
  - Such as the new Protected Process Light functionality (TBD)

PROTECTED PROCESSES

- Protected Process concept was first introduced in Windows Vista for DRM
- Audiodg.exe and Mfpmp.exe are "protected" against administrators
  - Cannot be debugged, memory cannot be read/written, information cannot be queried/set
  - Even if SYSTEM
  - Even if special debug privilege is enabled
- In Windows 8.1, the system was extended to apply beyond DRM processes
  - A Protected Process "light" has the same anti-Administrator protections
  - And there are different levels of protection
    - LSASS is protected against Pass-the-Hash attacks
    - CSRSS, SMSS, SCM are protected against administrator exploits (prevents Jailbreak)
    - Windows Defender and other anti-malware tools are protected against malware

SIGNING LEVELS AND EKUS

- A table the kernel reads tell it which signing level to request from UMCI based on the protection level
  - "14" is for Windows TCB (Trusted Computing Base) binaries
  - "12" is for Windows binaries
  - "8" is for Microsoft binaries
  - "5" is for Anti-Malware Binaries
  - "4" is for any Authenticode-signed Binary
- UMCI has different logic for deciding which certificate requirements are needed for each signing level
- Certificate requirements are primarily based on Enhanced Key Usages (EKU)

SIGNING POLICIES

- In Windows 8, most of the UMCI system was hard-coded
  - Only thing that mattered is for a binary to be signed or unsigned
- But in Windows 8.1, addition of UMCI on desktop, increased devices and security requirements, as well as protected processes, add complexity
  - How to define default signing levels?
    - Kernel drivers must be signed on x86 + Secure Boot, but not on x86 + BIOS
  - How to decide requirements/EKUs for each signing level?
    - For example, Windows Store App for Phone vs for Desktop
  - How to decide minimum hashing levels?
    - For example SHA256 is required for ARM Drivers, but SHA1 is accepted for x86/x64
- Protected UEFI variable called "CurrentPolicy" has a blob that is read by Boot Manager

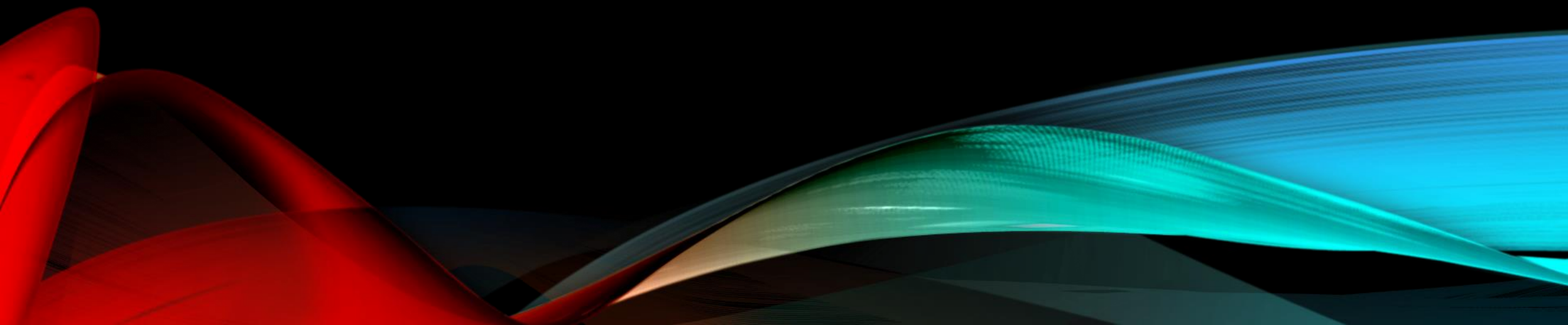
DIFFERENT SIGNING POLICIES

- Production Policy is embedded in Bootmgr.efi x86, x64, and ARM (no need for UEFI variable)
  - Prevents certain BCD options such as Test Signing and Debugging Mode
  - Sets flags and options to enforce Code Integrity decisions
    - For example, ARM policy sets the default signing level for desktop to “8”
- Windows Kits Policy can be downloaded with Windows Driver Kit
  - Installs a new policy in “CurrentPolicy” that has a special GUID and a “kits policy” flag
  - The effect is that Windows Debugger and other developer tools are allowed to run
    - UMCI will block tools that have “Windows Kits Signer” EKU when the kits policy is not installed
    - Certain Windows debugging APIs check the current policy GUID to see if it matches

WHAT ABOUT DEBUGGING?

- Debugging is officially only possible by turning off SecureBoot, which turns off the signing policy (even the embedded one)
- Microsoft has a private "debug signing policy" which removes DEBUG and TESTSIGNING from the list of BCD options, so they can be enabled
- Other types of signing policies are also probably given out to partners
  - To allow running restricted number of tools/driver during device development
- Signing policy can also completely disable UMCI/CI and set special testing options
- Signing policies are not only signed (and must be signed with a special key), but they can also have a Device ID
  - In this way, a signing policy can be bound to only a single device

WINDOWS SECURITY FAILURES



NTFS USN CHANGE JOURNAL

- Modern Windows uses a file system called NTFS (NT File System – very original)
- NTFS is a journaled file system – all changes are recorded as transactions, which allows rollback/recovery during unexpected reboots
- But Microsoft quickly realized that developers might also want access to the log of changes
  - Allows immediate response to file system events
  - Allows optimizing code to quickly see if a file has been modified or not
  - Any other kind of applications that needs a list of change operations done to a file/directory (which are also files)
- This is called the USN Change Journal
  - And every file receives a USN ID that describes the latest change in the journal
  - Any change to a file causes a change in its USN ID

NTFS USN CHANGE JOURNAL

- The USN Journal itself has a USN Journal ID
- The Journal ID is used to ensure that the change in USN IDs is localized to a specific journal – user could reformat the disk, or recreate the journal, and new USNs would now be created
- "fsutil usn" command as well as equivalent FSCTL commands can be used to create, query, modify, and delete the USN journal
- System components typically used the USN as an optimization before checking if a file has changed
  - MSN Messenger uses it for "Shared Folders" implementation
- USN Journal is protected so that only Administrators can delete/create it
  - But it is not meant as a generic security boundary – we'll see why this matters

CI CACHE EA

- Checking the hash of a file every time it executes can be slow
  - Especially with DLLs → launching Office can cause 400-500 hashes to be computed each time
- A hash cache mechanism is used to record the results of a previous hash check
- Data is encoded in an NTFS extended attribute called $Kernel.PurgeESB
  - Because the attribute starts with $Kernel, NTFS will not allow it to be set from user-mode
  - Because the attribute has "Purge" in it, NTFS will not allow it to be carried when a file is copied or modified in any way (the attribute will disappear)
- The extended attribute has the signing level that this file has been blessed with in the past, as well as the USN that matches the last file change

WHEN IS CI CACHE USED?

- It looks like at least some of the designers behind CI knew the risks involved:
  - CI EA Cache is not trusted for kernel drivers
  - Or for protected processes
  - Or for anything loaded into a protected process
  - Or for Windows Store applications
  - Or for any signing level above 11 (i.e.: “Windows” and higher)
- So the cache is mostly only used for standard DLLs and executables when signing checks are required
  - But on x64 and x86, UMCI is not used (except for Store/protected applications), so the cache is hit very rarely
- But what if you map a driver binary… in user-mode?
  - Or a normally protected DLL, in a non-protected process

FORCING A CI CACHE RECORD

- When a driver is being loaded as a user-mode mapping, it goes through the standard UMCI logic and EA cache…
  - But on x64/x86, the default signing level for user-mode mappings is 0, so no signature check is done
  - On ARM, signing level of 8 is required, so if the driver is signed by Microsoft, it will load as a DLL, and a CI Cache entry will be written in the EA
- On x64/x86, we can enable "Load MS Signed Images Only" process mitigation
  - Once we do this, all user-mode DLLs must be signed with a level of 8
  - So CI Cache entries will be written for any driver being loaded as a user-mode DLL, as long as it's signed by Microsoft
- OK, so we can force UMCI to *write* to the cache, but the driver still has to be signed, and the cache still won't be trusted in a protected process/driver

CI CACHE LIVABILITY BUG

- The CI EA Cache is trusted **forever** unless the file has changed (EA Purge) or the USN ID is different…
- A user can boot their computer in test signing mode – in which test certificates are accepted by CI as being valid
- One can then load an unsigned driver as a user-mode DLL using the trick in the previous page
  - Unsigned driver is trusted because test mode is on
  - CI EA Cache Entry writes a signing policy of 8 because "Microsoft" is the requirement when the mitigation is enabled
- Can do the same thing to any other binary – put a test-signed certificate and map in user-mode
  - Now we can essentially write "fake" CI EA Cache Entries... trusted at reboot!

CONTROL AREA CACHING BUG

- But we know that the cache is **not** trusted when loading a driver or a DLL in a protected process (the two cases when signing is used)
  - So this is only useful on Windows RT, to launch unsigned desktop applications?
- And Windows RT doesn't even have test signing? So what have we achieved?
  - This must be where Microsoft stopped to analyze the security design.
- Strange behavior was observed:
  - Load unsigned driver as a DLL inside "Microsoft-only" mitigated process
    - Works, because of the EA Cache being trusted
  - Now load the driver as a normal driver…
    - Works?!?!
- Turns out the memory manager has a cache of loaded images!

CONTROL AREA CACHING BUG

- When you load any kind of PE image in windows, this causes the creation of a Control Area
- The Control Area has a pointer to the Segment
  - Multiple handles to the same image (or multiple loads) all use the same Segment
- In Windows 8, if you load a driver in user-mode, this creates a Segment…
  - And if you try to load it in kernel-mode, it reuses the Segment…
  - But there is a check if this is a driver load or protected image load…
  - … If it is, the cached Segment is checked for a digital signature with CI.
- In Windows 8.1, the Segment now has a "Cached Signing Level" – what was the signing level at which this segment was created at runtime?
  - If the signing level of the segment is higher or equal then the one requested, the image signing check is ignored – the Segment is trusted

USN0 BACKDOOR

- Whenever an Extended Attribute Hash Cache query fails, CI also checks if the system is currently in "USN0 Trust Mode"
  - 0 -> USN0 trust state is uninitialized (boot)
  - 1 -> USN0 trust state is false (untrusted/mismatch)
  - 2 -> USN0 trust state is true (trusted/match)
- Then USN0 Trust Mode is set based on:
  - Registry key that enables USN0 backdoor (set by default!)
  - Protected variable that stores the USN Journal ID
- If the current USN journal ID matches the protected variable, USN0 trust is set.
- From this point on, any file that has a USN of 0 is allowed to execute without a signature!
  - Deleting USN Journal causes USN 0 to be set for all binaries – but new Journal ID!

UEFI & BIOS PROTECTED VARIABLES

- Since the USN Journal ID is used to ensure the integrity of the Extended Attribute Cache, it must be stored somewhere "offline" to detect changes in the USN Journal itself (such as new journal being created)
- On BIOS systems, this is called "CI Protected Storage", and is in the registry
  - Uses Windows API "NtLockRegistryKey" which prevents modification of the value
  - But registry could be edited offline (but offline attack is possible against the data on disk anyway)
- On UEFI systems, this is stored in the protected UEFI variable space
  - Name is "Kernel_USN", and because it starts with Kernel_, user-mode is not allowed writing to it
  - What about modification from the kernel?
  - There is a *secondary* value called "Kernel_USNCopy" that is only boot-accessible!

PROTECTED VARIABLE BUG - BIOS

- NTFS USN Journal is only enabled after SMSS runs autochk.exe (first user-mode process)
  - Writes to disk are not possible before
- CI library wants to make sure that USN journal is enabled, and that USN journal ID is trusted – but only once SMSS has run
  - So when the image hash of SMSS is computed, it runs the USN Journal ID logic check
- But in Windows, there is an undocumented setting to change the "Initial User Process"
  - Attacker can copy Smss.exe to Smss2.exe and use the undocumented setting
- USN Journal ID will not be read from protected variable, and variable will not be locked!

PROTECTED VARIABLE BUG - UEFI

- On UEFI, the variable is always "locked" because it starts with Kernel ("Kernel_USN").
- But for some reason, the Kernel_USN variable is only read/used by CI if the system is in Secure Boot Mode
- So with UEFI, but Secure Boot disabled, the system is less secure than in BIOS!
  - Registry key value is ignored/not protected (because system is not BIOS)
  - Variable is not read because system is not Secure Boot
- OK but… if Journal ID variable cannot be retrieved, doesn't the system go in "untrusted USN0 mode"?
  - Nope… function returns STATUS_NOT_IMPLEMENTED
  - STATUS_NOT_IMPLMENETED is not treated as trust failure!

USN JOURNAL ID COLLISION BUG

- So on BIOS or UEFI systems (without Secure Boot) we can use USN0 backdoor without a problem.
- But Secure Boot, only the initial system Journal ID will be trusted – and only OEM reset clears it
  - On most systems, OEM has reformatted/reinstalled/recreated USN Journal ID, so the current one is no longer trusted
- But what exactly is a USN Journal ID?
  - fsutil usn queryjournal c: => Usn Journal ID : 0x01ccb88645057745
- Hmm, familiar format?
  - lkd> !filetime 0x01ccb88645057745 => **12/12/2011 13:26:51.001**
- The USN Journal ID is the time when it was created!
  - Administrator can simply set the time back and recreated the USN Journal!

CONCLUSION

- Don’t take things that were designed for one security environment, and use them in another environment
- Revisit caching technologies whenever possible
- Don’t take assumptions and protections against one type of attacker to work against another type of attacker
- Don’t ship backdoors, even if you think you’ve protected them
- If you don’t understand the legacy codebase, don’t use the legacy codebase!

QA TIME

Thank you!