UNREAL MODE: BREAKING PROTECTED PROCESSES

Alex Ionescu
NSC 2014
http://www.alex-ionescu.com
@aionescu

ABOUT ALEX IONESCU

- Chief Architect at CrowdStrike, a security startup
- Previously worked at Apple on iOS Core Platform Team
- Co-author of *Windows Internals 5th and 6th Editions*
- Reverse engineering NT since 2000 – main kernel developer of ReactOS
- Instructor of worldwide Windows internals classes
- Conference speaking:
  - SyScan 2014-2012
  - NoSuchCon 2014-2013, Breakpoint 2012
  - Recon 2014-2010, 2006
  - Blackhat 2013, 2008
- For more info, see www.alex-ionescu.com
- Twitter: @aionescu

INTRODUCTION

- Windows Vista introduced core changes to the kernel to allow atomic, kernel-driven process creation inside of a "protected environment"
  - Used to protect access to the DRM keys and to secure the System process
- Windows 8.1 extends that model in order to protect key non-DRM system processes even from Admin, and to mitigate against pass-the-hash attacks
- Digital signatures and code signing now add an additional boundary of protection beyond load/don't load
  - Similar to the iOS Entitlement Model
- Mechanisms change a few core security paradigms:
  - Admin == Kernel is something that Microsoft has sometimes disagreed with, especially in light of PatchGuard, Code Signing and DRM. Now it's really !=
  - Unkillable processes and unstoppable services are now something supported and documented for developer (mis)use

OUTLINE

- Introduction
- Code Signing 101
- Signature Levels
- Process and Service Protection
- UEFI Variables 101
- LSASS Pass-the-Hash Protection
- Windows 10 Hyper-V Process Containers
- Conclusion

CODE SIGNING 101

AUTHENTICODE

- For almost two decades, Portable Executable format (PE) supports Digital Code Signing through the Microsoft Authenticode standard
- Digital certificate (X.509 standard) contains a signature that ties the application to a publisher (or "signer") that is identified by a public key.
- Public key is used to compute a hash of the executable
- Hash is compared to the hash in the certificate itself
- Over the years, extended with
  - Embedded catalog support (to avoid bloating executable size)
  - Support for page hashes (so that each page can be hashed at runtime)
  - Stricter requirements (now SHA-256 is used and keys are 2048-bit or more)

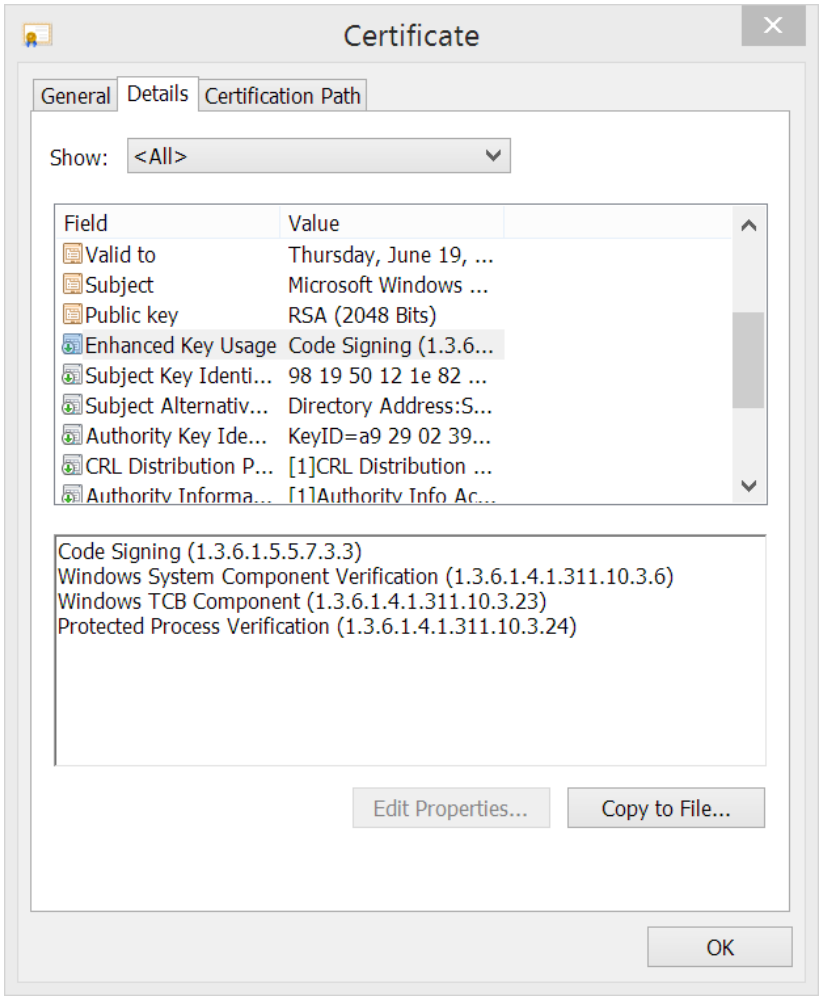
ENHANCED KEY USAGE

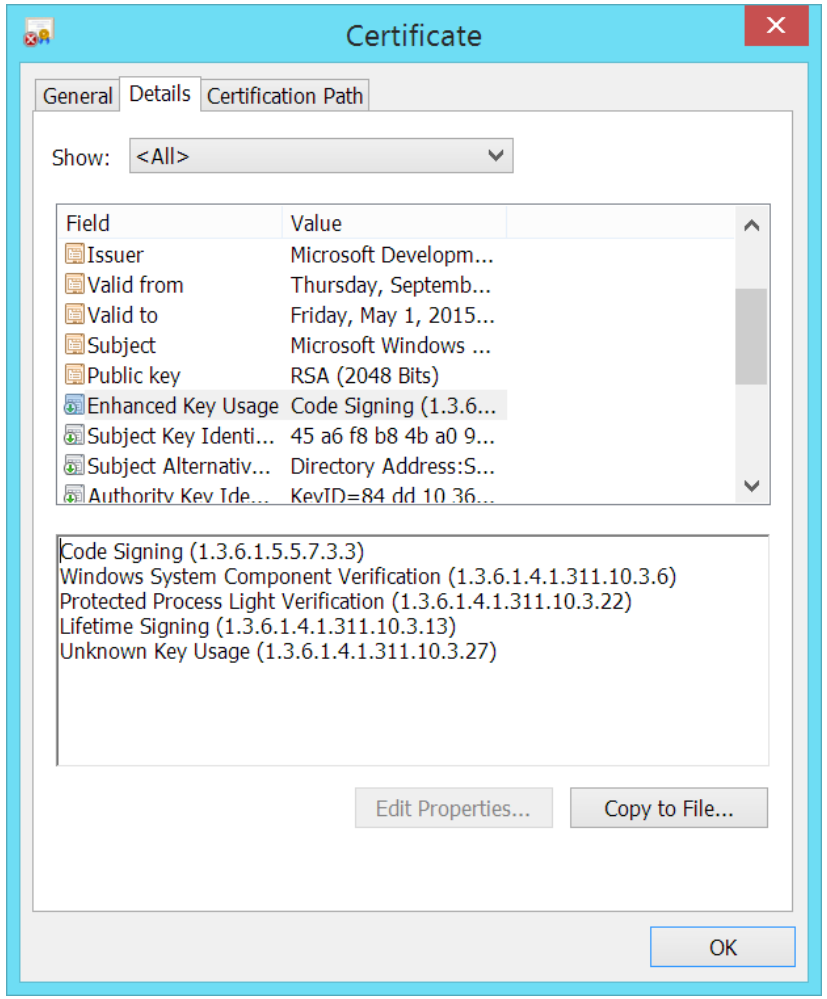
- X.509 certificates are used by different applications for different *intended purposes*
  - SSL Encryption
  - EFS Encryption
  - Java Applet Signing
  - Windows Kernel Mode Driver Signing
  - …etc
- The "EKUs" in a certificate, which are requested during the certificate creation through the CA process, allow the signature validation algorithm to make sure that the certificate is being used in accordance to its issuance rules
  - For example, if a Root CA issued a 19$ SSL Certificate, it should not be used in order to authenticate a Windows kernel-mode driver load

WINDOWS EKU USAGE

- Windows traditionally only used one EKU to validate code signing requirements:
  - Code Signing (1.3.6.1.5.5.7.3.3)
- A few other EKUs were used for WHQL and determining OS files
  - Microsoft Publisher (1.3.6.1.4.1.311.76.8.1)
- Now this has changed with additional support for ELAM:
  - Early Launch Antimalware Driver (1.3.6.1.4.1.311.61.4.1)
- And new Windows 8.1 PPL support:
  - Windows System Component Verification (1.3.6.1.4.1.311.10.3.6)
  - Protected Process Light Verification (1.3.6.1.4.1.311.10.3.22)
  - Windows TCB Component (1.3.6.1.4.1.311.10.3.23)
- Windows 10 adds 2 additional EKUs

SAMPLE EKUS


Certificate
General
Details
Certification Path
Show:
<All>
Field
Value
Valid to
Thursday, June 19, ...
Subject
Microsoft Windows ...
Public key
RSA (2048 Bits)
Enhanced Key Usage
Code Signing (1.3.6...
Subject Key Identi...
98 19 50 12 1e 82 ...
Subject Alternativ...
Directory Address:S...
Authority Key Ide...
KeyID=a9 29 02 39...
CRL Distribution P...
[1]CRL Distribution ...
Code Signing (1.3.6.1.5.5.7.3.3)
Windows System Component Verification (1.3.6.1.4.1.311.10.3.6)
Windows TCB Component (1.3.6.1.4.1.311.10.3.23)
Protected Process Verification (1.3.6.1.4.1.311.10.3.24)
Edit Properties...
Copy to File...
OK


Certificate
General
Details
Certification Path
Show:
<All>
Field
Value
Issuer
Microsoft Developm...
Valid from
Thursday, Septemb...
Valid to
Friday, May 1, 2015...
Subject
Microsoft Windows ...
Public key
RSA (2048 Bits)
Enhanced Key Usage
Code Signing (1.3.6...
Subject Key Identi...
45 a6 f8 b8 4b a0 9...
Subject Alternativ...
Directory Address:S...
Code Signing (1.3.6.1.5.5.7.3.3)
Windows System Component Verification (1.3.6.1.4.1.311.10.3.6)
Protected Process Light Verification (1.3.6.1.4.1.311.10.3.22)
Lifetime Signing (1.3.6.1.4.1.311.10.3.13)
Unknown Key Usage (1.3.6.1.4.1.311.10.3.27)
Edit Properties...
Copy to File...
OK

SIGNATURE LEVELS

SIGNING LEVEL NUMBER

- In Windows 8, the system needed to differentiate between Windows, Microsoft, and 3rd party signed files, instead of the black and white "signed or unsigned" choice
  - A system of integers was added in which each successive higher level indicates an increased amount of trust for the signer
- 0->Unchecked, 1->Unsigned, 4->Authenticode, 8->Microsoft, 12->Windows
- In Windows 8.1, this was extended to differentiate against the new types of processes and their protection levels
  - 6->Store, 7->Anti Malware, 14->Windows TCB
  - Other custom levels can be defined with a *signing policy*
- Signing level is granted based on the root key, the EKUs, and the operating system usage/requirement

SIGNING LEVEL CHECKS

- On Windows 8.1, signing levels are computed only for user-mode binaries that are run as protected processes
- On Windows 8.1 RT, they are computed anytime a process launches, based on SeILSigningPolicy
  - Defaults to 8, can be configured through different signing policy
- Process launch is not allowed if the User-Mode Code Integrity (UMCI) runtime does not allow the given certificate to grant the required signing level
  - This is called the Exe Signature Level
- Each successful process launch also carries with it the Dll Signature Level
  - Image load is not allowed if UMCI runtime does not allow the certificate of any loaded DLL to grant the required section signing level

LEVELS DEMO

SECURE BOOT POLICY

- The default signing levels are embedded inside the Windows Code Integrity Library (CI.DLL)
  - You can see their names by dumping the strings/resources inside Ci.dll
  - Documented on my blog
- However, signing levels can be customized through the use of a custom Secure Boot Policy
  - Also allows customizing hash requirements, mapping to protected process signers, and more
- Finally, signing levels can also be "runtime" registered
  - See blog for more details – used to allow 3rd party Anti-Malware developers to access level 7
- Policy is loaded from EFI\Microsoft\Boot\SecureBootPolicy.p7b or UEFI Authenticated Variable "CurrentPolicy" or embedded inside Bootmgfw.efi

PROCESS PROTECTION

VISTA PROTECTED PROCESS

- In Windows Vista, the term *protected process* was first introduced as a mechanism to protect the DRM requirements of the system
  - "Protected Media Path", PMP requires trusted components all the way to the output source
- Protected Processes had one bit set in the kernel EPROCESS structure at process launch time, if
  - CREATE_PROTECTED_PROCESS was passed to CreateProcess
  - Correct Windows Media DRM Certificate is present
- Media Foundation API took care of implementation details
  - Protected Environment Authorization (Peauth.sys) handles the obfuscation, key exchange, etc…
- *PsIsProtectedProcess* is documented for kernel-mode drivers and *NtQueryInformationProcess* can be used to check the PEB field as well

PROTECTED PROCESS RIGHTS

- Once a process runs with the bit set, all of its threads and processes are protected against accesses except:
  - PROCESS/THREAD_QUERY_LIMITED_INFORMATION
  - PROCESS_TERMINATE
  - PROCESS/THREAD_SUSPEND_RESUME
  - SYNCHRONIZE
- Only other protected processes can bypass the checks and use ACL instead
- Debug/TCB privilege, SYSTEM account, does not carry any weight here!
- In-box processes that run protected include
  - Mfpmp.exe (if playing protected media with WMP)
  - Audiodg.exe (Windows Audio Service Device Graph)

WINDOWS 8

- Minor change introduced in Windows 8, the ProtectedProcess bit disappears
- OR'ing bit 1 to the SignatureLevel makes the process protected
  - Thus, Audiodg.exe is 5 on Windows 8, 9 on Windows 8 RT
- Continues to be a DRM-enforcement mechanism only, however
- Note that System process also runs as Protected
  - This prevents Administrators from accessing the handle table or user-mode virtual address space of the process
  - ASLR relocation information is stored in user-mode
  - Code Integrity catalogs, driver images, and other data, temporarily stored in user-mode as well!
  - Some DRM/crypto temporary data as well
  - Handles could be duplicated and lead to access to protected kernel resources, such as hibernation file, page file, registry hives and transactional logs

WINDOWS 8.1

- Adds a new “Protection” field to EPROCESS composed of three elements
  - Protected Signer
  - Protected Type
  - Auditing Mode
- The Protected Signer is one of the root keys accepted by UMCI, possibly combined with one or more required EKUs
  - PsProtectedSignerAuthenticode = 0n1
  - PsProtectedSignerCodeGen = 0n2
  - PsProtectedSignerAntimalware = 0n3
  - PsProtectedSignerLsa = 0n4
  - PsProtectedSignerWindows = 0n5
  - PsProtectedSignerWinTcb = 0n6

PROTECTED PROCESS LIGHT

- The Protected Type is one of:
  - PsProtectedTypeProtectedLight = 0n1 OR PsProtectedTypeProtected = 0n2
- Combined together this creates a number such as 0x31
  - Anti-Malware Protected Light
- New API introduced for kernel: *PsIsProcessProtectedLight*.
  - User mode can also check PEB, or PROCESS_EXTENDED_BASIC_INFORMATION
- Protected processes can only be accessed by Protected processes
  - Protected Light processes can only be accessed by Protected Light or higher processes
  - Each signer dominates the other and creates an access hierarchy
- Different signers receive different access mask protections
  - See RtlProtectedAccess array (used by *RtlTestProtectedAccess*)

SYSTEM EFFECTS

- LSA processes, Anti-Malware processes, and Windows TCB processes can no longer be killed
  - But suspension is still possible
- Deprecates previous "Critical Process" functionality used to protect CSRSS and SMSS
- Makes Windows 8 RT public jailbreak impossible to execute, as it requires injection of code into CSRSS
- Makes injection/memory-based techniques for dumping LSA hashes, passwords, and secrets impossible
  - But only if LSA protection is enabled
- Also makes it that much harder to steal (impersonate) a SYSTEM token

PROTECTION ATTRIBUTE

- In Vista, CREATE_PROTECTED_PROCESS was the only flag needed for *CreateProcess* to do the right thing
- But in Windows 8.1, how to specify the actual protection level required (type and signer?)
  - Using the new Protection Level Attribute (0x2000B) in the Process/Thread Attribute List
- At the Win32 level, this is based on an undocumented enumeration:
  - 1 – Windows Signer, Protected Type
  - 2 – Windows Signer, Protected Light Type
  - 3 – Antimalware Signer, Protected Light Type
- At the NT level, it is converted into the actual PS_PROTECTED_SIGNER and PS_PROTECTED_TYPE number we saw earlier

PROTECTION CHAIN

- Recall that Windows 8.1 only checks signature levels if and only if a process is created requesting a protection level
  - Double-clicking a file on Explorer doesn't do this
- Therefore, there exists an implied "protection chain" that makes the system work:
  - The System process marks itself as protected
  - It launches SMSS requesting protection 0x61
  - SMSS launches CSRSS requesting protection 0x61
  - SMSS launches Wininit and Winlogon with no protection (Win 10: Wininit is 0x61)
  - Wininit launches LSASS with protection 0x41 (if configured)
  - Wininit launches SCM with protection 0x61
- But what about Anti-Malware?

PROTECTION DEMO

WINDBG ANALYSIS DEMO

SERVICE PROTECTION

PROTECTED SERVICES

- Since Services.exe (aka the SCM/Service Control Manager) launches highly protected, this might imply that it can manage protected light processes…
- Since Anti-Malware applications run protected, this implies something must have had launched them requesting a protection level…
- Indeed, the key to Process Protection Light is that it must also allow 3rd party and/or non-System processes to also run protected
  - This is done by enhancing services to support a protection level of their own
- In Windows 8,1 SERVICE_CONFIG_LAUNCH_PROTECTED can be sent to *ChangeServiceConfig2*, with a matching SERVICE_LAUNCH_PROTECTED_INFO structure and level
- The SCM sends this protection level request in the *CreateProcess* call that it makes

PROTECTED SERVICES

- The SCM API ultimately ends up setting the key "LaunchProtected" in the Services database entry in the registry
- The following calls/commands immediately become blocked by non-protected callers:
  - *ChangeServiceConfig(2)* / sc config
  - *ControlService(Ex)* / sc start, stop, pause
  - *DeleteService* / sc delete
  - *SetServiceObjectSecurity* / sc privs
- Check is done by the RPC Server inside SCM, using *RtlTestProtectedAccess* and by asking the kernel for the caller's protection level
- AppXSvc, sppsvc, WdNisSvc, WinDefend and WSService run protected in Windows 8.1
  - Can check with "sc qprotection"

PROTECTED APPLICATIONS

- Therefore, services ultimately are in charge of the last chain of protected process launches as they can use *CreateProcess* to spawn children that are also protected
- Recall that the enumeration is undocumented, however!
- Microsoft expects developers to thus use PROTECTION_LEVEL_SAME, which is documented as the only valid type for PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL
- In this way, Anti-Malware services will spawn Anti-Malware processes, etc…
- User-launched applications will never run protected, unless the application employs a type of self-launcher and/or COM activation launcher method and requests a protection level for its spawned clone

PROTECTED SERVICES DEMO

UEFI VARIABLES 101

UEFI VARIABLES (SPECIFICATION)

- UEFI variables are used to control behavior of UEFI Boot Manager, Drivers, and Applications
  - Can also be used to define OS settings
  - *Boot####* and *Driver####* define drivers and boot applications.
  - *BootCurrent, BootNext, BootOrder* influence boot manager selection
  - *DriverOrder* influences driver ordering
- Three types of UEFI variables exist
  - Boot Variables can only be seen while in "Boot Services" mode (in TSL or earlier)
  - Runtime Variables can be seen while in "Runtime" mode (RT or later)
  - Authenticated Variables are Authenticode-signed binary blobs that must be signed as per Secure Boot policy
- Variables can also be volatile and/or read-only vs read-write
- Variables live in different namespaces, identified by a GUID

UEFI VARIABLES (WINDOWS)

- Windows provides user and kernel APIs for Firmware Variables
  - User-mode: *Get/SetFirmwareEnvironmentVariableEx*
    - Requires SE_SYSTEM_ENVIRONMENT_PRIVILEGE
  - Kernel-mode: *ExGet/SetFirmwareEnvironmentVariable*
- Internally, kernel calls *IoGet/SetEnvironmentVariable*
  - Can use either the \Device\Sysenv device if custom NVRAM is present, or the HAL in standard scenarios
- Microsoft owns the following Vendor GUID. Used as KEK Signature Owner, Secure Boot Policy GUID, and also UEFI Variable Namespace GUID
  - {77fa9abd-0359-4d32-bd60-28f4e78f784b} [*ExpSecureBootVendorGuid*]
  - Variables that start with "Kernel_" are implicitly part of this namespace
- User-mode uses *NtSetSystemEnvironmentValueEx* which prevents the above

LSASS PASS-THE-HASH PROTECTION

LSASS PROTECTED MODE

- LSASS uses new Process Protection Level to prevent attacks from user-mode elevated caller
  - Runs as Protected Process Light, Protected Signer LSA
- Disabled by default, since would now require DLLs to also be signed at least with the LSA or higher protected signer flag
  - To enable, set RunAsPPL in SYSTEM\CurrentControlSet\Control\Lsa
- On UEFI systems, variable is mirrored into the UEFI firmware database
  - Kernel_Lsa_Ppl_Config is used as a Runtime EFI Variable
  - Microsoft provides reset UEFI binary to interactively remove UEFI protection
- Both system variable and registry key must be deleted for LSA to revert to old behavior
- Prevents operations against LSA except PROCESS_TERMINATE

DISABLING LSA PROTECTION

- mountvol X: /s
- copy C:\LSAPPLConfig.efi X:\EFI\Microsoft\Boot\LSAPPLConfig.efi /Y
- bcdedit /create {0cb3b571-2f2e-4343-a879-d86a476d7215} /d "DebugTool" /application osloader
- bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} path "\EFI\Microsoft\Boot\LSAPPLConfig.efi"
- bcdedit /set {bootmgr} bootsequence {0cb3b571-2f2e-4343-a879-d86a476d7215}
- bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} loadoptions %1
- bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} device partition=X:
- mountvol X: /d

LSAPPLCONFIG.EFI VIDEO

PROTECTED PROCESSES AND CRASHES

TAKING A DUMP

- When a Windows process crashes, the Windows Error Reporting infrastructure is activated in order to generate a mini-dump
  - Programs can do this on their own by using MiniDumpWriteDump
  - Procdump.exe, Taskmgr.exe, Windbg.exe all use this API
- Quote "Windows Internals, 6th Edition":

> The Audio Device Graph process (Audiodg.exe) is a protected process because protected music content can be decoded through it. Similarly, the Windows Error Reporting (or WER, discussed in Chapter 3) client process (Werfault.exe) can also run protected because it needs to have access to protected processes in case one of them crashes. Finally, the System process itself is protected

- "Can also run protected" ???

WER FOR PROTECTED PROCESSES

- If a protected process crashes, WER launches a special version of the WerFault.exe process, which has the correct EKUs for running as a protected process
  - It's launched as Protected TCB (0x62) – or the highest possible level!
- Because of its protection level, this version of WER is able to open handles to processes such as Csrss.exe and Smss.exe
  - And more interestingly, Lsass.exe if running with LSA Protection
- It's then able to call MiniDumpWriteDump without any problems
- So we can we just force a protected process to crash?
  - Not obvious
  - But let's cheat and use the kernel debugger…

PROTECTED CRASH DEMO

PROTECTED CRASH DEMO 2

PROTECTED CRASH ANALYSIS

- By following the Procmon output we've just seen, the following command-line is visible:

- WER parses the following options:

  - /h /s – Secure Dump
  - /pid – Crashing PID
  - /tid – Crashing TID
  - /type – Minidump Type (see MiniDumpWriteDump)
  - /file – Dump File Handle
  - /encfile – Encrypted Dump File Handle
  - /cancel – Event to Cancel Dump

TRICKING WER

- By controlling the parameters that are passed to WER and using a flaw in the dumping process, one can cause WER to reveal the unencrypted contents of the dump
  - Patch for the issue is delayed to January – cannot reveal exact design flaw
- Resulting dump is the same type of dump that ProcDump generates and which Windbg can open
- And, more concretely, the same format dump that Mimikatz can parse when using the sekurlsa::minidump option
- Thus, by leveraging a WER design flaw, the LSA Protection Mode can essentially be ignored/bypassed, and Mimikatz can continue to be used without requiring a driver

MIMIKATZ DEMO

OTHER ISSUES

- As mentioned, LSASS is not the only process that leverages process protection levels
  - Other system issues can arise from the ability to dump protected processes
- WER can do more than just create a dump of a process
  - Other potential issues can exist due to the ability to control WER
- What about writing into a protected process?
  - No current public flaw exists. My RPC vulnerability presented at SyScan would be used to target protected processes that contain an RPC server and spray data inside of their memory
    - Fixed in October Patch Tuesday
  - However, I will be presenting new ALPC vulnerabilities at a later time
- Ultimately, an admin can still cause dangerous environment changes which can affect protected processes

WINDOWS 10 AND CONCLUSION

WINDOWS 10: THE FUTURE

- From Critical Process to Protected Process to Protected Process Light to ???
  - "VSM" – Virtual Secure Machine?
  - "Hyper-V Secure Containers"?
  - "Isolated Process"?
  - "Secure Process"?
- Windows 10 blog announced that in Windows 10, LSA credential information will be stored in a secured container/enclave leveraging Hyper-V Client
  - Latest builds of Windows 10 contain an "Lsalso.exe" process which is an early prototype of an isolated version of LSASS
    - Calls itself "VSMified LSASS"
  - Latest kernel contains "IsSecureProcess" in PROCESS_EXTENDED_INFORMATION and "SecureProcess" in KPROCESS->Flags
  - New Process Creation Attribute -- *PsAttributeSecureProcess*

CONCLUSION

- Protected Processes in Windows 8.1 significantly increase the overall platform security
  - Less exploitable vulnerabilities
  - Tighter control of core system binaries
  - Anti-malware-type applications can survive manipulation and termination by admin-level malware
- They do not protect against kernel attackers –boundary is at the user level
- Desire of supporting customer crash scenarios through WER has resulted in a "backdoor" into the memory of protected processes
  - Will be fixed in January 2015
- Upcoming Windows 10 platform features will prevent this type of attack
  - Even from a kernel attacker!