Masking Malicious Memory Artifacts – Part II: Blending in with False Positives

forrest-orr.net/post/masking-malicious-memory-artifacts-part-ii-insights-from-moneta

Forrest Orr

July 16, 2020

Introduction

With fileless malware becoming a ubiquitous feature of most modern Red Teams, knowledge in the domain of memory stealth and detection is becoming an increasingly valuable skill to add to both an attacker and defender's arsenal. I've written this text with the intention of further improving the skill of the reader as relating to the topic of memory stealth on Windows both when designing and defending against such malware. First by introducing my open source memory scanner tool Moneta (on Github here), and secondly by exploring the topic of legitimate dynamic code allocation, false positives and stealth potential therein discovered through use of this scanner.

This is the second in a series of posts on malware forensics and bypassing defensive scanners, the part one of which can be found here. It was written with the assumption that the reader understands the basics of Windows internals, memory scanners and malware design.

Moneta

In order to conduct this research I wrote a memory scanner in C++ which I've named Moneta. It was designed both as an ideal tool for a security researcher designing malware to visualize artifacts relating to dynamic code operations, as well as a simple and effective tool for a defender to quickly pick up on process injections, packers and other types of malware in memory. The scanner maps relationships between the PEB, stack, heaps, CLR, image files on disk and underlying PE structures with the regions of committed memory within a specified process. It uses this information to identify anomalies, which it then uses to identify IOCs. It does all of this without scanning the contents of any of the regions it enumerates, which puts it in stark contrast to tools such as pe-sieve, which is also a usermode/runtime memory IOC scanner but which relies on byte patterns in addition to memory characteristics as its input. Both Moneta and pe-sieve have the shared characteristic of being usermode scanners designed for runtime analysis, as opposed to tools based on the Volatility framework which rely on kernel objects and which are generally intended to be used retrospectively on a previously captured memory dump file.

Moneta focuses primarily on three areas for its IOCs. The first is the presence of dynamic/unknown code, which it defines as follows:

1. Private or mapped memory with executable permissions.
2. Modified code within mapped images.
3. PEB image bases or threads with start addresses in non-image memory regions.
4. Unmodified code within unsigned mapped images (this is a soft indicator for hunting not a malware IOC).

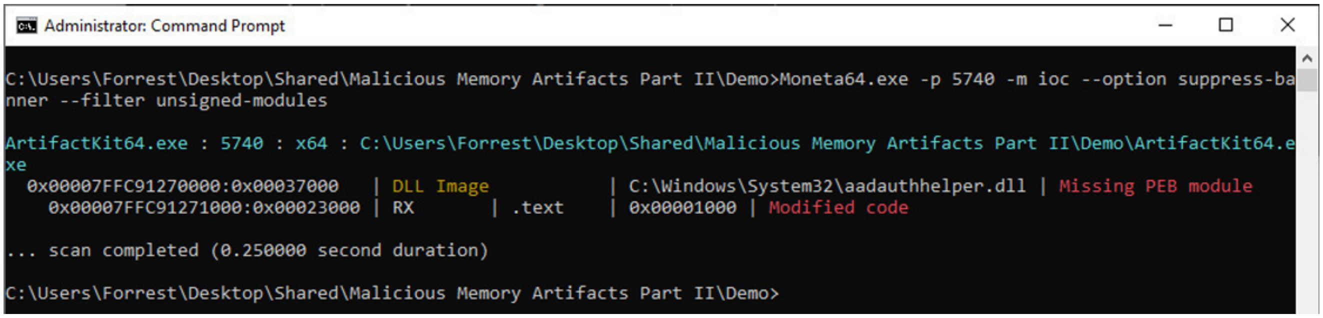
Secondly, Moneta focuses on suspicious characteristics of the mapped PE image regions themselves:

1. Inconsistent executable permissions between a PE section in memory and its counterpart on disk. For example a PE with a section which is *+RX* in memory but marked for *+R* in its PE header on disk.
2. Mapped images in memory with modified PE headers.
3. Mapped images in memory whose *FILE_OBJECT* attributes cannot be queried (this is an indication of phantom DLL hollowing).

Thirdly, Moneta looks at IOCs related to the process itself:

1. The process contains a mapped image whose base address does not have a corresponding entry in the PEB.
2. The process contains a mapped image whose base address corresponds to an entry in the PEB but whose name or path (as derived from its *FILE_OBJECT*) do not match those in the PEB entry.

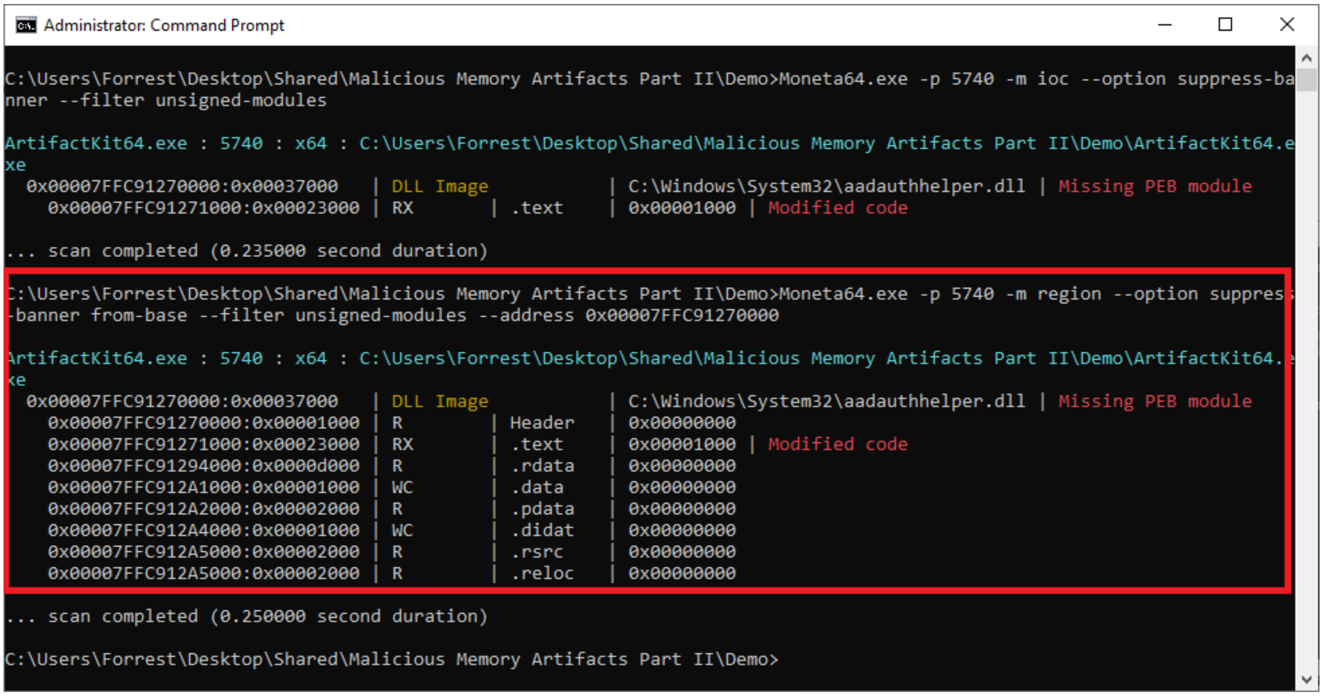
To illustrate the attribute-based approach to IOCs utilized by Moneta, a prime example can be found in the first part of this series, where classic as well as phantom DLL hollowing were described in detail and given as examples of lesser known and harder to detect alternatives to classic dynamic code allocation. In the example below, I've pointed Moneta at a process containing a classic DLL hollowing artifact being used in conjunction with a shellcode implant.



```
C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>Moneta64.exe -p 5740 -m ioc --option suppress-ba
nner --filter unsigned-modules

ArtifactKit64.exe : 5740 : x64 : C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\ArtifactKit64.e
xe
  0x00007FFC91270000:0x00037000   | DLL Image          | C:\Windows\System32\aadauthhelper.dll | Missing PEB module
    0x00007FFC91271000:0x00023000 | RX       | .text    | 0x00001000 | Modified code

... scan completed (0.250000 second duration)

C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>
```

Figure 1 - Moneta being used to select all committed memory regions associated with IOCs within a process containing a DLL hollowing artifact with a shellcode implant

The module **aadauthhelper.dll** at *0x00007FFC91270000* associated with the triggered IOC can be further enumerated by changing the selection type of Moneta from **ioc** to **region** and providing the exact address to select. The **from-base** option enumerates the entire region (from its allocation base) associated with specified address, not only its subregion (VAD).



```
C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>Moneta64.exe -p 5740 -m ioc --option suppress-ba
nner --filter unsigned-modules

ArtifactKit64.exe : 5740 : x64 : C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\ArtifactKit64.e
xe
  0x00007FFC91270000:0x00037000   | DLL Image           | C:\Windows\System32\aadauthhelper.dll | Missing PEB module
    0x00007FFC91271000:0x00023000 | RX        | .text     | 0x00001000 | Modified code

... scan completed (0.235000 second duration)

:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>Moneta64.exe -p 5740 -m region --option suppres
-banner from-base --filter unsigned-modules --address 0x00007FFC91270000

rtifactKit64.exe : 5740 : x64 : C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\ArtifactKit64.
e
  0x00007FFC91270000:0x00037000   | DLL Image           | C:\Windows\System32\aadauthhelper.dll | Missing PEB module
    0x00007FFC91270000:0x00001000 | R         | Header    | 0x00000000
    0x00007FFC91271000:0x00023000 | RX        | .text     | 0x00001000 | Modified code
    0x00007FFC91294000:0x0000d000 | R         | .rdata    | 0x00000000
    0x00007FFC912A1000:0x00001000 | WC        | .data     | 0x00000000
    0x00007FFC912A2000:0x00002000 | R         | .pdata    | 0x00000000
    0x00007FFC912A4000:0x00001000 | WC        | .didat    | 0x00000000
    0x00007FFC912A5000:0x00002000 | R         | .rsrc     | 0x00000000
    0x00007FFC912A5000:0x00002000 | R         | .reloc    | 0x00000000

... scan completed (0.250000 second duration)

C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>
```

Figure 2 - Moneta being used to enumerate the memory region associated with a hollowed DLL containing a shellcode implant

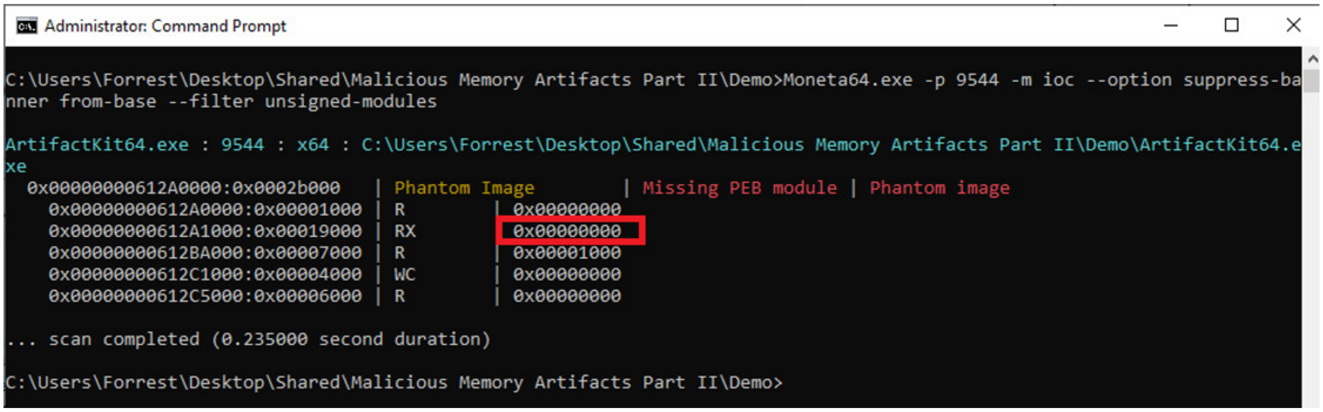
The two suspicions in ***Figure 2*** illustrate the strategy used by Moneta to detect DLL hollowing, as well as other (more common) malware stealth techniques such as Lagos Island (a technique often used to bypass usermode hooks). The **aadauthhelper.dll** module itself, having been mapped with NTDLL.DLL!NtCreateSection and NTDLL.DLL!NtMapViewOfSection as opposed to legitimately using NTDLL.DLL!LdrLoadDll, lacks an entry in the loaded modules list referenced by the PEB. In the event that the module had been legitimately loaded and added to the PEB, the shellcode implant would still have been detected due to the 0x1000 bytes (1 page) of memory privately mapped into the address space and retrieved by Moneta by querying its working set - resulting in a **modified code** IOC as seen above.

The C code snippet below, loosely based upon Moneta, illustrates the detection of classic DLL hollowing through use of both PEB discrepancy and working set IOCs:

```
uint8_t *pAddress = ...
MEMORY_BASIC_INFORMATION Mbi;

if (VirtualQueryEx(hProcess, pAddress, &Mbi, sizeof(MEMORY_BASIC_INFORMATION)) == sizeof(MEMORY_BASIC_INFORMATION)) {
if(Mbi.Type == MEM_IMAGE && IsExecutable(&Mbi)) {
wchar_t ModuleName[MAX_PATH + 1] = { 0 };

if (!GetModuleBaseNameW(hProcess, (static_cast<HMODULE>(Mbi.AllocationBase), ModuleName, MAX_PATH + 1)) {
// Detected missing PEB entry...
}

if (Mbi.State == MEM_COMMIT && Mbi.Protect != PAGE_NOACCESS) {
uint32_t dwPrivateSize = 0;
PSAPI_WORKING_SET_EX_INFORMATION WorkingSets= { 0 };
uint32_t dwWorkingSetsSize = sizeof(PSAPI_WORKING_SET_EX_INFORMATION);

for (uint32_t dwPageOffset = 0; dwPageOffset < Mbi.RegionSize; dwPageOffset += 0x1000) {
WorkingSets.VirtualAddress = (static_cast<uint8_t *>(Mbi.BaseAddress) + dwPageOffset);
if (K32QueryWorkingSetEx(this->ProcessHandle, &WorkingSets, dwWorkingSetsSize)) {
if (!WorkingSets.VirtualAttributes.Shared) {
dwPrivateSize += 0x1000;
}
}
```

```
}

if(dwPrivateSize) {
// Detected modified code...
}
}
}
}
```

In the example below, I’ve pointed Moneta at a process containing a phantom DLL hollowing artifact used in conjunction with a shellcode implant.



```
C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>Moneta64.exe -p 9544 -m ioc --option suppress-ba
nner from-base --filter unsigned-modules

ArtifactKit64.exe : 9544 : x64 : C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\ArtifactKit64.e
xe
  0x00000000612A0000:0x0002b000   | Phantom Image        | Missing PEB module | Phantom image
    0x00000000612A0000:0x00001000 | R          | 0x00000000
    0x00000000612A1000:0x00019000 | RX         | 0x00000000
    0x00000000612BA000:0x00007000 | R          | 0x00001000
    0x00000000612C1000:0x00004000 | WC         | 0x00000000
    0x00000000612C5000:0x00006000 | R          | 0x00000000

... scan completed (0.235000 second duration)

C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>
```

Figure 4 - Moneta being used to enumerate the memory region associated with a hollowed phantom DLL containing a shellcode implant

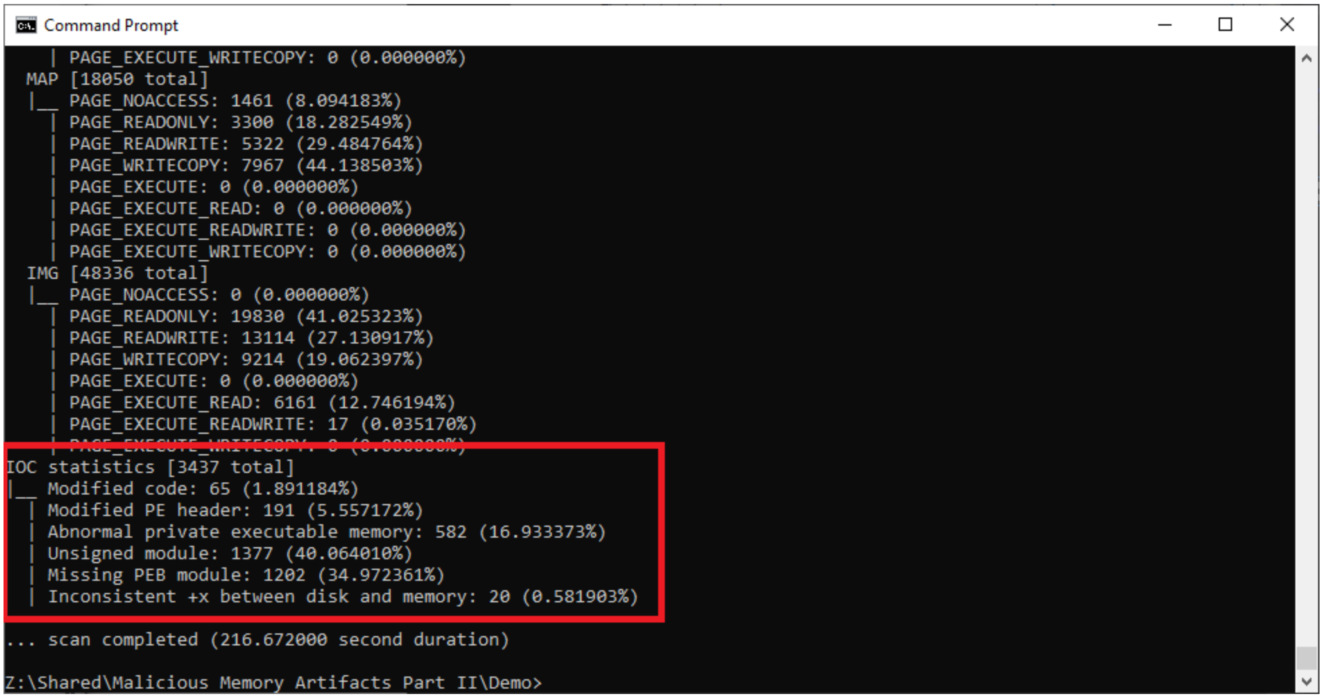
Notably in the image above, the missing PEB module suspicion persists (since the region in question is technically image memory without a corresponding PEB module entry) but the image itself is unknown. This is because TxF isolates its transactions from other processes, including in this case Moneta. When attempting to query the name of the file associated with the image region from its underlying *FILE_OBJECT* using the PSAPI.DLL!GetMappedFileNameW API, external processes will fail in the unique instance that the section underlying the image mapping view was generated using a transacted handle created by an external process. **This is the most robust method I’ve devised to reliably detect phantom DLL hollowing and process doppelganging**. This also results in the subregions of this image mapping region (distinguished by their unique VAD entries in the kernel) being unable to be associated with PE sections as they are in ***Figure 2***. Notably, phantom DLL hollowing has done a very nice job of hiding the shellcode implant itself. In the highlighted region of ***Figure 4*** above, the private bytes associated with the region (which should be 0x1000, or 1 page, due to the shellcode implant) is **zero**. There is no other method I am aware of powerful enough to hide modified ranges of executable image memory from working set scans. This is why the Moneta scan of the classic DLL hollowing artifact process seen in ***Figure 2*** yields a “modified code” suspicion, while phantom DLL hollowing does not.

The code snippet below, loosely based upon Moneta, illustrates the detection of phantom DLL hollowing through TxF file object queries:

```
uint8_t *pAddress = ...
MEMORY_BASIC_INFORMATION Mbi;

if (VirtualQueryEx(hProcess, pAddress, &Mbi, sizeof(MEMORY_BASIC_INFORMATION)) == sizeof(MEMORY_BASIC_INFORMATION)) {
if(Mbi.Type == MEM_IMAGE) {
wchar_t DevFilePath[MAX_PATH + 1] = { 0 };
if (!GetMappedFileNameW(hProcess, static_cast<HMODULE>(Mbi.AllocationBase), DevFilePath, MAX_PATH + 1)) {
// Detected phantom DLL hollowing...
}
}
}
```

Filters and False Positivies

With an understanding of the IOC criteria described in the previous section, a scan of my full Windows 10 OS would be expected to yield no IOCs, yet this is far from the reality in practice.

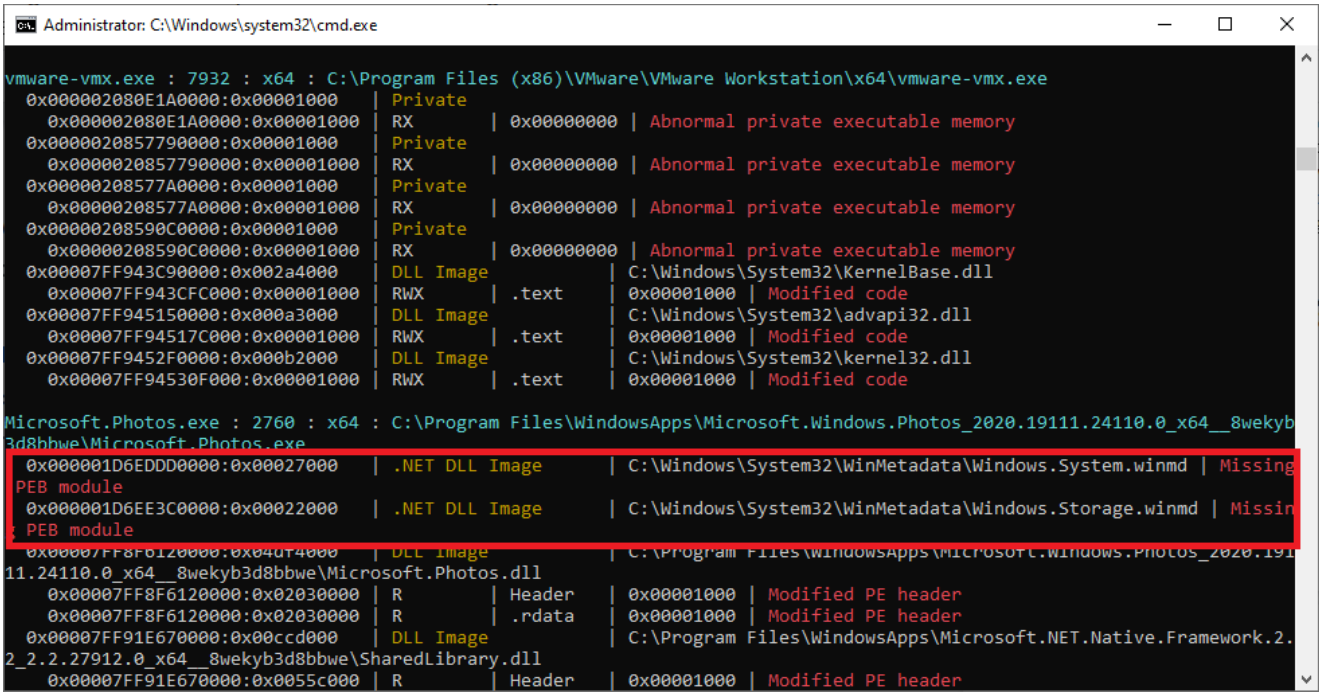


```
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
  MAP [18050 total]
  |__ PAGE_NOACCESS: 1461 (8.094183%)
    | PAGE_READONLY: 3300 (18.282549%)
    | PAGE_READWRITE: 5322 (29.484764%)
    | PAGE_WRITECOPY: 7967 (44.138503%)
    | PAGE_EXECUTE: 0 (0.000000%)
    | PAGE_EXECUTE_READ: 0 (0.000000%)
    | PAGE_EXECUTE_READWRITE: 0 (0.000000%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
  IMG [48336 total]
  |__ PAGE_NOACCESS: 0 (0.000000%)
    | PAGE_READONLY: 19830 (41.025323%)
    | PAGE_READWRITE: 13114 (27.130917%)
    | PAGE_WRITECOPY: 9214 (19.062397%)
    | PAGE_EXECUTE: 0 (0.000000%)
    | PAGE_EXECUTE_READ: 6161 (12.746194%)
    | PAGE_EXECUTE_READWRITE: 17 (0.035170%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
IOC statistics [3437 total]
|__ Modified code: 65 (1.891184%)
  | Modified PE header: 191 (5.557172%)
  | Abnormal private executable memory: 582 (16.933373%)
  | Unsigned module: 1377 (40.064010%)
  | Missing PEB module: 1202 (34.972361%)
  | Inconsistent +x between disk and memory: 20 (0.581903%)

... scan completed (216.672000 second duration)

Z:\Shared\Malicious Memory Artifacts Part II\Demo>
```

Figure 5 - IOC statistics generated by Moneta given a full OS memory space

With an astounding 3,437 IOCs on a relatively barren Windows 10 OS it quickly becomes clear why so many existing memory scanners rely so heavily on byte patterns and other less broad IOC criteria. I found these results fascinating when I first began testing Moneta, and I discovered many quirks, hidden details and abnormalities inherent to many subsystems in Windows which are of particular interest when designing both malware and scanners.

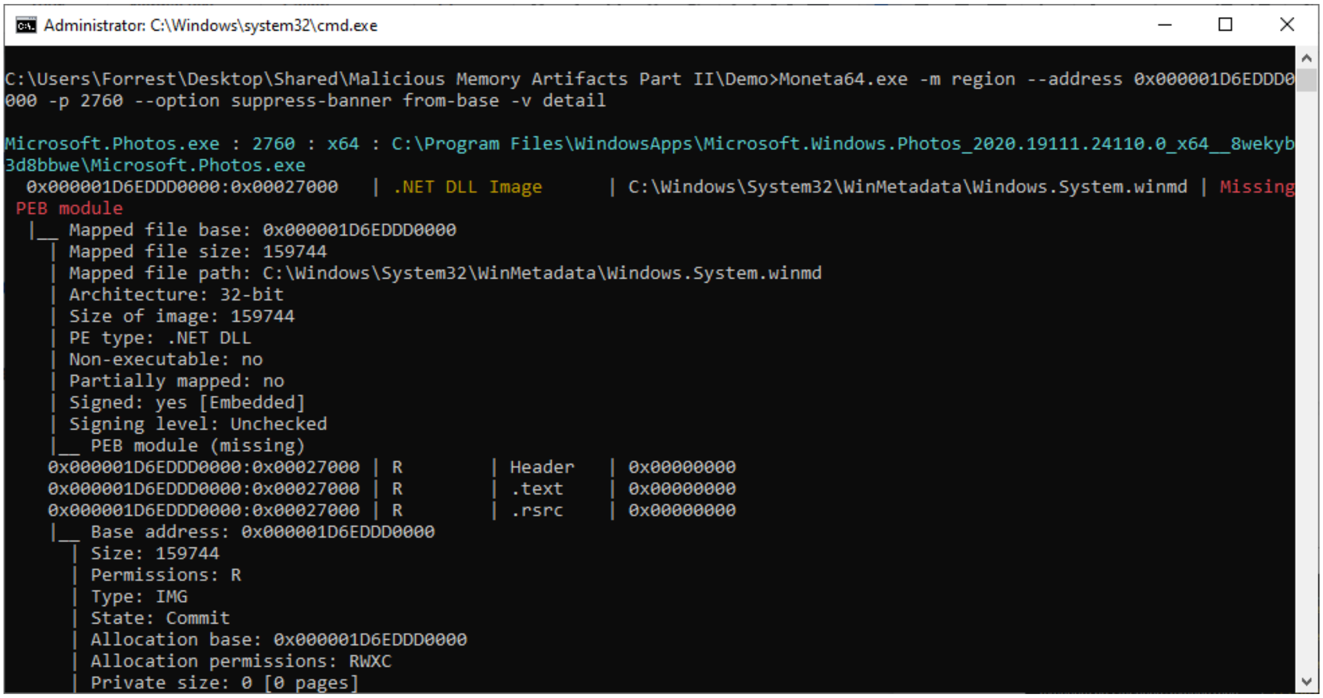
Let's begin by examining the 1202 missing PEB module IOCs. These IOCs are only generated when a PE is explicitly mapped into a process as an image using ***SEC_IMAGE*** with NTDLL.DLL!NtCreateSection and is not added to the loaded modules list in the PEB - something which would be done automatically if the PE had been loaded how it is supposed to be loaded via NTDLL.DLL!LdrLoadDll.



```
vmware-vmx.exe : 7932 : x64 : C:\Program Files (x86)\VMware\VMware Workstation\x64\vmware-vmx.exe
  0x000002080E1A0000:0x00001000    | Private
    0x000002080E1A0000:0x00001000 | RX         | 0x00000000 | Abnormal private executable memory
  0x0000020857790000:0x00001000    | Private
    0x0000020857790000:0x00001000 | RX         | 0x00000000 | Abnormal private executable memory
  0x00000208577A0000:0x00001000    | Private
    0x00000208577A0000:0x00001000 | RX         | 0x00000000 | Abnormal private executable memory
  0x00000208590C0000:0x00001000    | Private
    0x00000208590C0000:0x00001000 | RX         | 0x00000000 | Abnormal private executable memory
  0x00007FF943C90000:0x002a4000    | DLL Image             | C:\Windows\System32\KernelBase.dll
    0x00007FF943CFC000:0x00001000 | RWX        | .text      | 0x00001000 | Modified code
  0x00007FF945150000:0x000a3000    | DLL Image             | C:\Windows\System32\advapi32.dll
    0x00007FF94517C000:0x00001000 | RWX        | .text      | 0x00001000 | Modified code
  0x00007FF9452F0000:0x000b2000    | DLL Image             | C:\Windows\System32\kernel32.dll
    0x00007FF94530F000:0x00001000 | RWX        | .text      | 0x00001000 | Modified code

Microsoft.Photos.exe : 2760 : x64 : C:\Program Files\WindowsApps\Microsoft.Windows.Photos_2020.19111.24110.0_x64__8wekyb
3d8bbwe\Microsoft.Photos.exe
  0x000001D6EDDD0000:0x00027000    | .NET DLL Image        | C:\Windows\System32\WinMetadata\Windows.System.winmd | Missing
 PEB module
  0x000001D6EE3C0000:0x00022000    | .NET DLL Image        | C:\Windows\System32\WinMetadata\Windows.Storage.winmd | Missin
g PEB module
  0x00007FF8F6120000:0x04df4000    | DLL Image             | C:\Program Files\WindowsApps\Microsoft.Windows.Photos_2020.191
11.24110.0_x64__8wekyb3d8bbwe\Microsoft.Photos.dll
    0x00007FF8F6120000:0x02030000 | R          | Header     | 0x00001000 | Modified PE header
    0x00007FF8F6120000:0x02030000 | R          | .rdata     | 0x00001000 | Modified PE header
  0x00007FF91E670000:0x00ccd000    | DLL Image             | C:\Program Files\WindowsApps\Microsoft.NET.Native.Framework.2.
2_2.2.27912.0_x64__8wekyb3d8bbwe\SharedLibrary.dll
    0x00007FF91E670000:0x0055c000 | R          | Header     | 0x00001000 | Modified PE header
```

Figure 6 - The metadata false positive results of an IOC scan made by Moneta

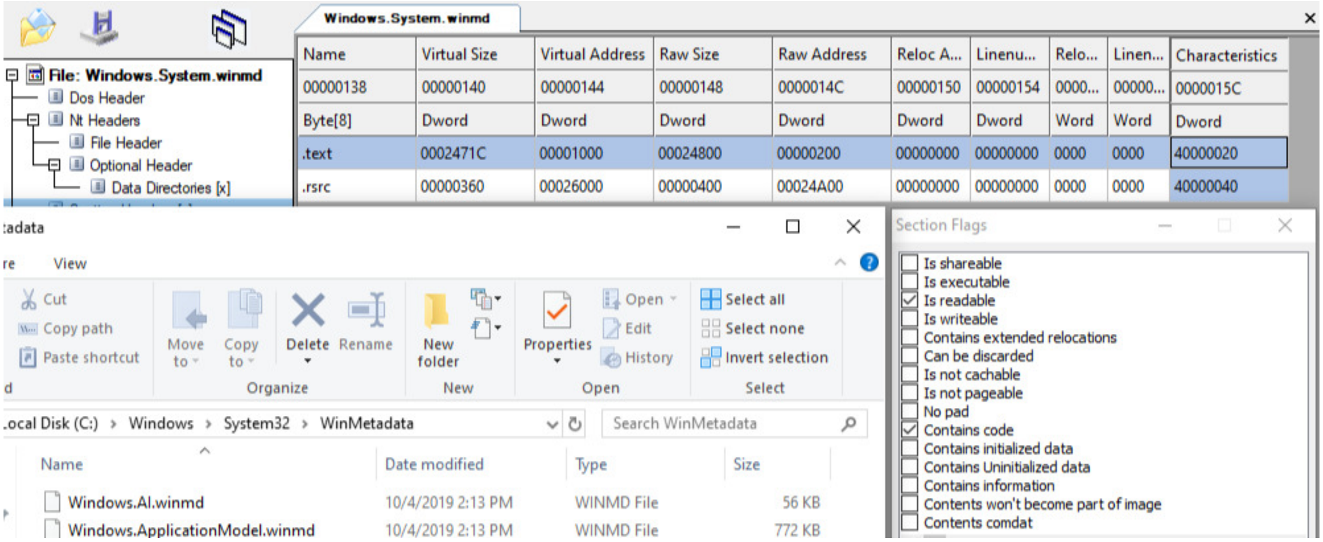
The region at *0x000001D6EDDD0000* corresponds to the base of a block of image memory within an instance of the **Microsoft.Photos.exe** process. At a glance, it shares characteristics in common with malicious DLL hollowing and Lagos Island artifacts. Further details of this region can be obtained through a subsequent scan of this exact address with a higher detail verbosity level:



```
C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>Moneta64.exe -m region --address 0x000001D6EDDD0
000 -p 2760 --option suppress-banner from-base -v detail

Microsoft.Photos.exe : 2760 : x64 : C:\Program Files\WindowsApps\Microsoft.Windows.Photos_2020.19111.24110.0_x64__8wekyb
3d8bbwe\Microsoft.Photos.exe
  0x000001D6EDDD0000:0x00027000    | .NET DLL Image        | C:\Windows\System32\WinMetadata\Windows.System.winmd | Missing
 PEB module
   |__ Mapped file base: 0x000001D6EDDD0000
     | Mapped file size: 159744
     | Mapped file path: C:\Windows\System32\WinMetadata\Windows.System.winmd
     | Architecture: 32-bit
     | Size of image: 159744
     | PE type: .NET DLL
     | Non-executable: no
     | Partially mapped: no
     | Signed: yes [Embedded]
     | Signing level: Unchecked
     |__ PEB module (missing)
    0x000001D6EDDD0000:0x00027000 | R          | Header     | 0x00000000
    0x000001D6EDDD0000:0x00027000 | R          | .text      | 0x00000000
    0x000001D6EDDD0000:0x00027000 | R          | .rsrc      | 0x00000000
     |__ Base address: 0x000001D6EDDD0000
       | Size: 159744
       | Permissions: R
       | Type: IMG
       | State: Commit
       | Allocation base: 0x000001D6EDDD0000
       | Allocation permissions: RWXC
       | Private size: 0 [0 pages]
```

Figure 7 - Detailed scan of the specific region associated with the metadata image

There are several interesting characteristics of this region. Prime among them, is the **Non-executable** attribute (queried through the NTDLL.DLL!NtQueryVirtualMemory API) set to **false** despite this image clearly not having been loaded with the intention of executing code. Non-executable image regions are a unique and undocumented feature of the NNTDLL.DLL!NtCreateSection API, which causes the resulting image to be immutably readonly but still of type *MEM_IMAGE*. Furthermore, use of the *SEC_IMAGE_NO_EXECUTE* flag when creating new sections allows for a bypass of the image load notification routine in the kernel. We would expect such a feature to have been used in the case of this metadata file, but it was not. There is a single VAD associated with the entire region, with PTE attributes of read-only even though the image was clearly loaded as a regular executable image (also evidenced by the initial permissions of *PAGE_EXECUTE_WRITECOPY*) and contains a ***.text*** section which would normally contain executable code.



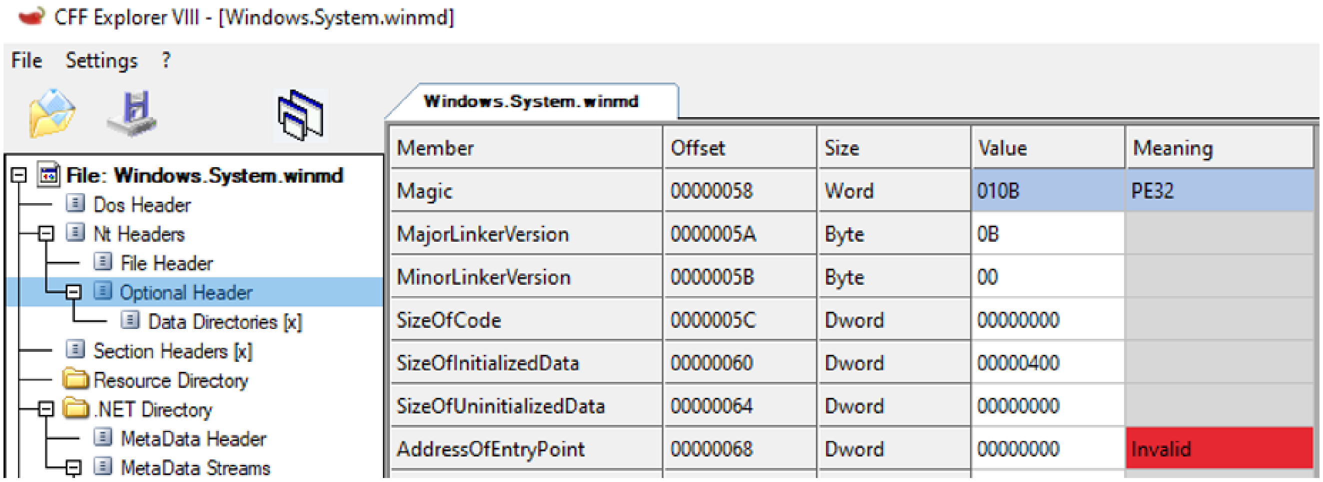
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc A...	Linenu...	Relo...	Linen...	Characteristics
00000138	00000140	00000144	00000148	0000014C	00000150	00000154	0000...	00000...	0000015C
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0002471C	00001000	00024800	00000200	00000000	00000000	0000	0000	40000020
.rsrc	00000360	00026000	00000400	00024A00	00000000	00000000	0000	0000	40000040

- ☐ Is shareable
- ☐ Is executable
- ☑ Is readable
- ☐ Is writeable
- ☐ Contains extended relocations
- ☐ Can be discarded
- ☐ Is not cachable
- ☐ Is not pageable
- ☐ No pad
- ☑ Contains code
- ☐ Contains initialized data
- ☐ Contains Uninitialized data
- ☐ Contains information
- ☐ Contents won't become part of image
- ☐ Contents comdat

Name	Date modified	Type	Size
Windows.AI.winmd	10/4/2019 2:13 PM	WINMD File	56 KB
Windows.ApplicationModel.winmd	10/4/2019 2:13 PM	WINMD File	772 KB

Figure 8 - PE sections and .text section attributes of Windows.System.winmd file in CFF explorer

As its name implies, this does appear to be a genuine metadata file which was not ever intended to be executed (despite being a valid PE, being loaded as an executable image and containing a ***.text*** section).



Member	Offset	Size	Value	Meaning
Magic	00000058	Word	010B	PE32
MajorLinkerVersion	0000005A	Byte	0B	
MinorLinkerVersion	0000005B	Byte	00	
SizeOfCode	0000005C	Dword	00000000	
SizeOfInitializedData	00000060	Dword	00000400	
SizeOfUninitializedData	00000064	Dword	00000000	
AddressOfEntryPoint	00000068	Dword	00000000	Invalid

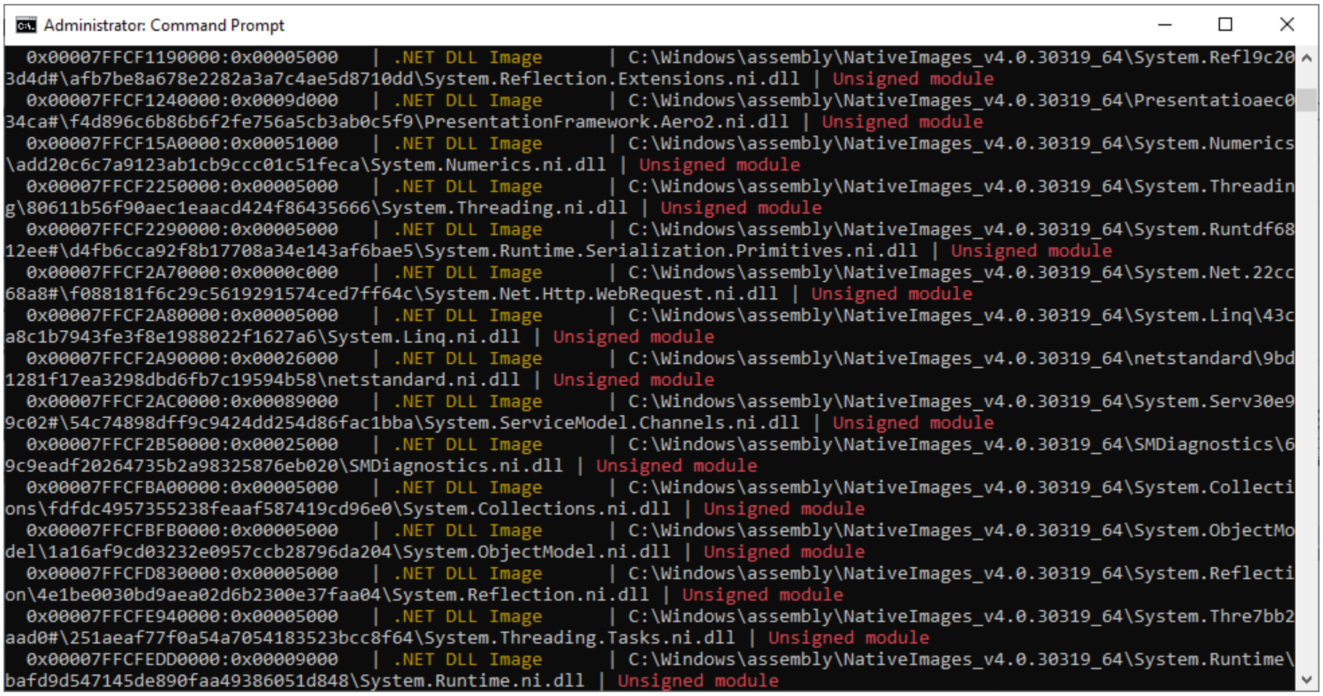
Figure 9 - The optional PE header of the Windows.System.winmd file in CFF explorer

The image above provides a definitive confirmation of the fact that this is a PE file which was never meant to execute: its *IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint* is zero. With no entry point and no exports, there are no conventional methods of executing this DLL, which explains why it was manually mapped in a way which made it appear as a malicious DLL hollowing or Lagos Island artifact.

Combining the criteria explored above, a filter rule was created within Moneta which removes **missing PEB module** IOCs associated with signed Windows metadata files with blank entry points. This methodology was repeated throughout the development of the scanner to eliminate false positives from its IOCs.

Windows metadata files are not alone in imitating Lagos Island IOCs: standard .NET assemblies have this same IOC as well, as they are not loaded via NTDLL.DLL!LdrLoadDll but rather are directly mapped using NTDLL.DLL!NtCreateSection with *SEC_IMAGE*. The exception to this rule is Native Image Generated (NGEN) .NET assemblies, which are loaded as standard native DLLs and therefore have corresponding links in the PEB. This phenomenon was first observed by Noora Hyvärinen of F-Secure in their post examining detection strategies for malicious .NET code.

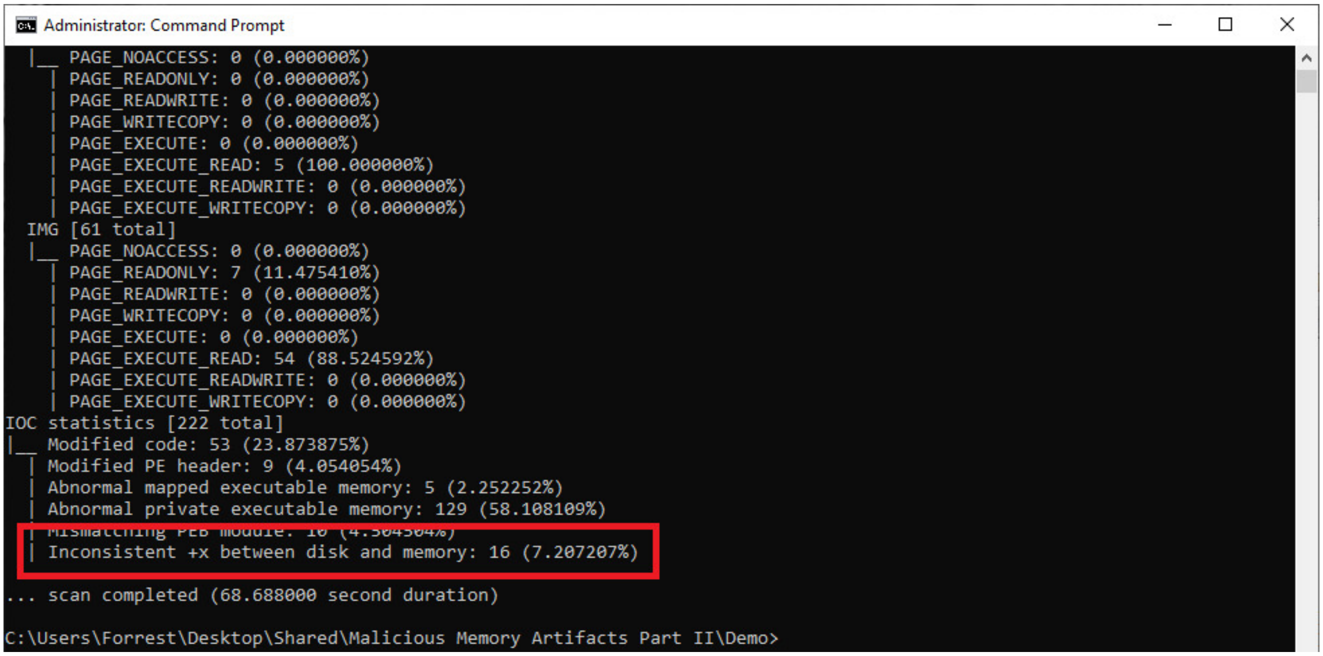
Another interesting detail of the statistics gathered in ***Figure 5*** are the 1377 unsigned modules, a total of about 40% of all IOCs on the OS. This large number is certainly inconsistent with what one would expect: for unsigned modules to be rarities associated exclusively with unsigned 3rd party applications. In reality, the vast majority of these unsigned images are derived from Microsoft DLLs, specifically, .NET NGEN assemblies. This is consistent with the concept of these DLLs being built dynamically, to eliminate the need for conversion of CIL to native code by JIT at runtime.



```
  0x00007FFCF1190000:0x00005000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Refl9c20
3d4d#\afb7be8a678e2282a3a7c4ae5d8710dd\System.Reflection.Extensions.ni.dll | Unsigned module
  0x00007FFCF1240000:0x0009d000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\Presentatioaec0
34ca#\f4d896c6b86b6f2fe756a5cb3ab0c5f9\PresentationFramework.Aero2.ni.dll | Unsigned module
  0x00007FFCF15A0000:0x00051000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Numerics
\add20c6c7a9123ab1cb9ccc01c51feca\System.Numerics.ni.dll | Unsigned module
  0x00007FFCF2250000:0x00005000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Threadin
g\80611b56f90aec1eaacd424f86435666\System.Threading.ni.dll | Unsigned module
  0x00007FFCF2290000:0x00005000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Runtdf68
12ee#\d4fb6cca92f8b17708a34e143af6bae5\System.Runtime.Serialization.Primitives.ni.dll | Unsigned module
  0x00007FFCF2A70000:0x0000c000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Net.22cc
68a8#\f088181f6c29c5619291574ced7ff64c\System.Net.Http.WebRequest.ni.dll | Unsigned module
  0x00007FFCF2A80000:0x00005000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Linq\43c
a8c1b7943fe3f8e1988022f1627a6\System.Linq.ni.dll | Unsigned module
  0x00007FFCF2A90000:0x00026000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\netstandard\9bd
1281f17ea3298dbd6fb7c19594b58\netstandard.ni.dll | Unsigned module
  0x00007FFCF2AC0000:0x00089000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Serv30e9
9c02#\54c74898dff9c9424dd254d86fac1bba\System.ServiceModel.Channels.ni.dll | Unsigned module
  0x00007FFCF2B50000:0x00025000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\SMDiagnostics\6
9c9eadf20264735b2a98325876eb020\SMDiagnostics.ni.dll | Unsigned module
  0x00007FFCFBA00000:0x00005000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Collecti
ons\fdfdc4957355238feaaf587419cd96e0\System.Collections.ni.dll | Unsigned module
  0x00007FFCFBFB0000:0x00005000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.ObjectMo
del\1a16af9cd03232e0957ccb28796da204\System.ObjectModel.ni.dll | Unsigned module
  0x00007FFCFD830000:0x00005000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Reflecti
on\4e1be0030bd9aea02d6b2300e37faa04\System.Reflection.ni.dll | Unsigned module
  0x00007FFCFE940000:0x00005000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Thre7bb2
aad0#\251aeaf77f0a54a7054183523bcc8f64\System.Threading.Tasks.ni.dll | Unsigned module
  0x00007FFCFEDD0000:0x00009000   | .NET DLL Image       | C:\Windows\assembly\NativeImages_v4.0.30319_64\System.Runtime\
bafd9d547145de890faa49386051d848\System.Runtime.ni.dll | Unsigned module
```

Figure 10 - Moneta IOC scan yielding over 1000 image memory regions connected to unsigned modules, the vast majority of them Windows .NET NGEN assemblies

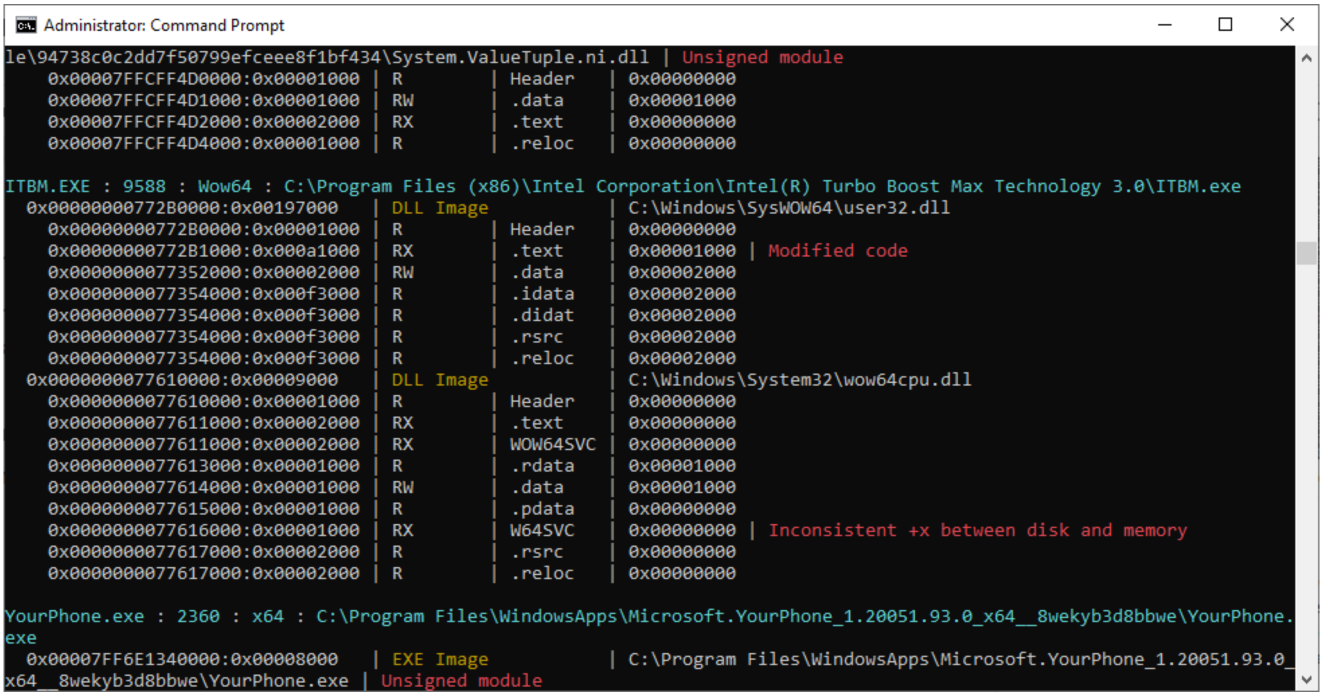
Shifting focus to other categories of IOC, another interesting genre appears as **inconsistent +x between disk and memory** at a total of 16 (7%) of the now drastically reduced IOC total of 222.



```
  |__ PAGE_NOACCESS: 0 (0.000000%)
    | PAGE_READONLY: 0 (0.000000%)
    | PAGE_READWRITE: 0 (0.000000%)
    | PAGE_WRITECOPY: 0 (0.000000%)
    | PAGE_EXECUTE: 0 (0.000000%)
    | PAGE_EXECUTE_READ: 5 (100.000000%)
    | PAGE_EXECUTE_READWRITE: 0 (0.000000%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
  IMG [61 total]
  |__ PAGE_NOACCESS: 0 (0.000000%)
    | PAGE_READONLY: 7 (11.475410%)
    | PAGE_READWRITE: 0 (0.000000%)
    | PAGE_WRITECOPY: 0 (0.000000%)
    | PAGE_EXECUTE: 0 (0.000000%)
    | PAGE_EXECUTE_READ: 54 (88.524592%)
    | PAGE_EXECUTE_READWRITE: 0 (0.000000%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
IOC statistics [222 total]
|__ Modified code: 53 (23.873875%)
  | Modified PE header: 9 (4.054054%)
  | Abnormal mapped executable memory: 5 (2.252252%)
  | Abnormal private executable memory: 129 (58.108109%)
  | Mismatching PEB module: 10 (4.504504%)
  | Inconsistent +x between disk and memory: 16 (7.207207%)

... scan completed (68.688000 second duration)

C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>
```

Figure 11 - Moneta IOC scan result statistics while filtering metadata and unsigned modules

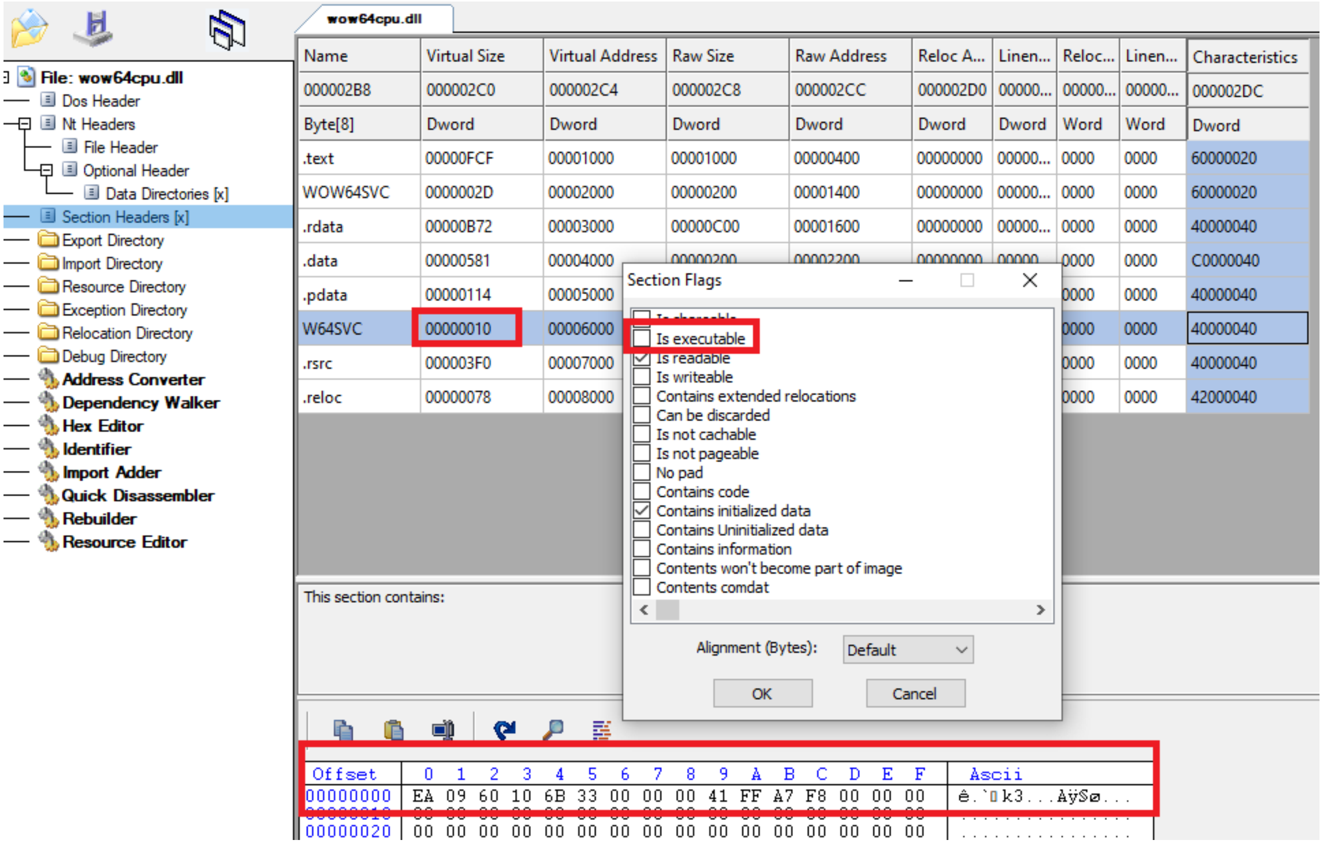
Interestingly, this number of 16 also matches the total number of Wow64 processes on the scanned OS. A further investigation yields the answer to why:



```
le\94738c0c2dd7f50799efceee8f1bf434\System.ValueTuple.ni.dll | Unsigned module
    0x00007FFCFF4D0000:0x00001000 | R         | Header    | 0x00000000
    0x00007FFCFF4D1000:0x00001000 | RW        | .data     | 0x00001000
    0x00007FFCFF4D2000:0x00002000 | RX        | .text     | 0x00000000
    0x00007FFCFF4D4000:0x00001000 | R         | .reloc    | 0x00000000

ITBM.EXE : 9588 : Wow64 : C:\Program Files (x86)\Intel Corporation\Intel(R) Turbo Boost Max Technology 3.0\ITBM.exe
  0x00000000772B0000:0x00197000   | DLL Image             | C:\Windows\SysWOW64\user32.dll
    0x00000000772B0000:0x00001000 | R         | Header    | 0x00000000
    0x00000000772B1000:0x000a1000 | RX        | .text     | 0x00001000 | Modified code
    0x0000000077352000:0x00002000 | RW        | .data     | 0x00002000
    0x0000000077354000:0x000f3000 | R         | .idata    | 0x00002000
    0x0000000077354000:0x000f3000 | R         | .didat    | 0x00002000
    0x0000000077354000:0x000f3000 | R         | .rsrc     | 0x00002000
    0x0000000077354000:0x000f3000 | R         | .reloc    | 0x00002000
  0x0000000077610000:0x00009000   | DLL Image             | C:\Windows\System32\wow64cpu.dll
    0x0000000077610000:0x00001000 | R         | Header    | 0x00000000
    0x0000000077611000:0x00002000 | RX        | .text     | 0x00000000
    0x0000000077611000:0x00002000 | RX        | WOW64SVC  | 0x00000000
    0x0000000077613000:0x00001000 | R         | .rdata    | 0x00001000
    0x0000000077614000:0x00001000 | RW        | .data     | 0x00001000
    0x0000000077615000:0x00001000 | R         | .pdata    | 0x00000000
    0x0000000077616000:0x00001000 | RX        | W64SVC    | 0x00000000 | Inconsistent +x between disk and memory
    0x0000000077617000:0x00002000 | R         | .rsrc     | 0x00000000
    0x0000000077617000:0x00002000 | R         | .reloc    | 0x00000000

YourPhone.exe : 2360 : x64 : C:\Program Files\WindowsApps\Microsoft.YourPhone_1.20051.93.0_x64__8wekyb3d8bbwe\YourPhone.
exe
  0x00007FF6E1340000:0x00008000   | EXE Image             | C:\Program Files\WindowsApps\Microsoft.YourPhone_1.20051.93.0_
x64__8wekyb3d8bbwe\YourPhone.exe | Unsigned module
```

Figure 12 - Inconsistent permission IOC stemming from wow64cpu.dll

Wow64cpu.dll is a module which is loaded into every Wow64 process in order to help facilitate the interaction between the 32-bit code/modules and 64-bit code/modules (Wow64 processes all have both 32 and 64-bit DLLs in them). Checking the PE sections attributes of the *W64SVC* section in **Wow64cpu.dll** on disk we can see that it should be read-only in memory:



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc A...	Linen...	Reloc...	Linen...	Characteristics
000002B8	000002C0	000002C4	000002C8	000002CC	000002D0	00000...	00000...	00000...	000002DC
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00000FCF	00001000	00001000	00000400	00000000	00000...	0000	0000	60000020
WOW64SVC	0000002D	00002000	00000200	00001400	00000000	00000...	0000	0000	60000020
.rdata	00000B72	00003000	00000C00	00001600	00000000	00000...	0000	0000	40000040
.data	00000581	00004000	00000200	00002200	00000000	00000...	0000	0000	C0000040
.pdata	00000114	00005000					0000	0000	40000040
W64SVC	00000010	00006000					0000	0000	40000040
.rsrc	000003F0	00007000					0000	0000	40000040
.reloc	00000078	00008000					0000	0000	42000040

Section Flags

☐ Is executable
☑ Is readable
☐ Is writeable
☐ Contains extended relocations
☐ Can be discarded
☐ Is not cachable
☐ Is not pageable
☐ No pad
☐ Contains code
☑ Contains initialized data
☐ Contains Uninitialized data
☐ Contains information
☐ Contents won't become part of image
☐ Contents comdat

Alignment (Bytes): Default

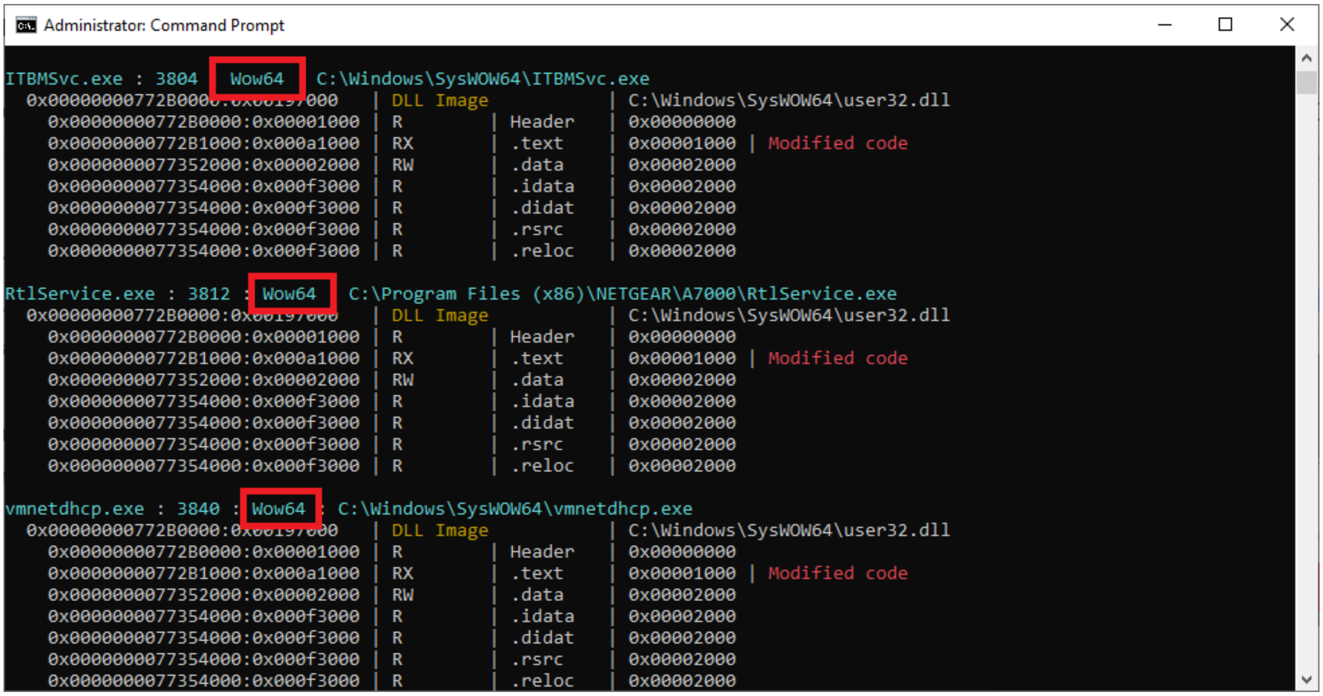
OK Cancel

```
Offset     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii
00000000  EA 09 60 10 6B 33 00 00 00 41 FF A7 F8 00 00 00   ê.`▯k3...Aÿ§ø...
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Figure 13 - Wow64cpu.dll W64SVC section in CFF Explorer

Another very interesting detail of the *W64SVC* section is that it contains only 0x10 bytes of data and is <u>**not**</u> modified after having its permissions changed from *+R* to *+RX* by Windows. This means that the content of the *W64SVC* section seen in ***Figure 13*** is meant to be executed at runtime as they appear in disk. The first byte of this region 0xEA is an intersegment far *CALL* instruction, the use of which is typically limited to x86/x64 mode transition in Wow64 processes (an attribute which is exploited by the classic Heaven's Gate technique).

Both the modified code within **User32.dll** (as well as occasionally the 32-bit version of **Kernel32.dll**) and the inconsistent permission IOCs seen in **Figure 12** are consistent side-effects of Wow64 initialization.



```
ITBMSvc.exe : 3804 : Wow64 : C:\Windows\SysWOW64\ITBMSvc.exe
  0x00000000772B0000:0x00197000    | DLL Image            | C:\Windows\SysWOW64\user32.dll
    0x00000000772B0000:0x00001000 | R          | Header    | 0x00000000
    0x00000000772B1000:0x000a1000 | RX         | .text     | 0x00001000 | Modified code
    0x0000000077352000:0x00002000 | RW         | .data     | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .idata    | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .didat    | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .rsrc     | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .reloc    | 0x00002000

RtlService.exe : 3812 : Wow64 : C:\Program Files (x86)\NETGEAR\A7000\RtlService.exe
  0x00000000772B0000:0x00197000    | DLL Image            | C:\Windows\SysWOW64\user32.dll
    0x00000000772B0000:0x00001000 | R          | Header    | 0x00000000
    0x00000000772B1000:0x000a1000 | RX         | .text     | 0x00001000 | Modified code
    0x0000000077352000:0x00002000 | RW         | .data     | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .idata    | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .didat    | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .rsrc     | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .reloc    | 0x00002000

vmnetdhcp.exe : 3840 : Wow64 : C:\Windows\SysWOW64\vmnetdhcp.exe
  0x00000000772B0000:0x00197000    | DLL Image            | C:\Windows\SysWOW64\user32.dll
    0x00000000772B0000:0x00001000 | R          | Header    | 0x00000000
    0x00000000772B1000:0x000a1000 | RX         | .text     | 0x00001000 | Modified code
    0x0000000077352000:0x00002000 | RW         | .data     | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .idata    | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .didat    | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .rsrc     | 0x00002000
    0x0000000077354000:0x000f3000 | R          | .reloc    | 0x00002000
```

Figure 14 - Modified code IOCs associated with user32 in Wow64 processes

They are actions taken at runtime by Windows, in both cases by manually changing the permissions of the ***.text*** and *W64SVC* sections using NTDLL.DLL!NtProtectVirtualMemory. A filter for both of these IOCs called **wow64-init** exists in Moneta.

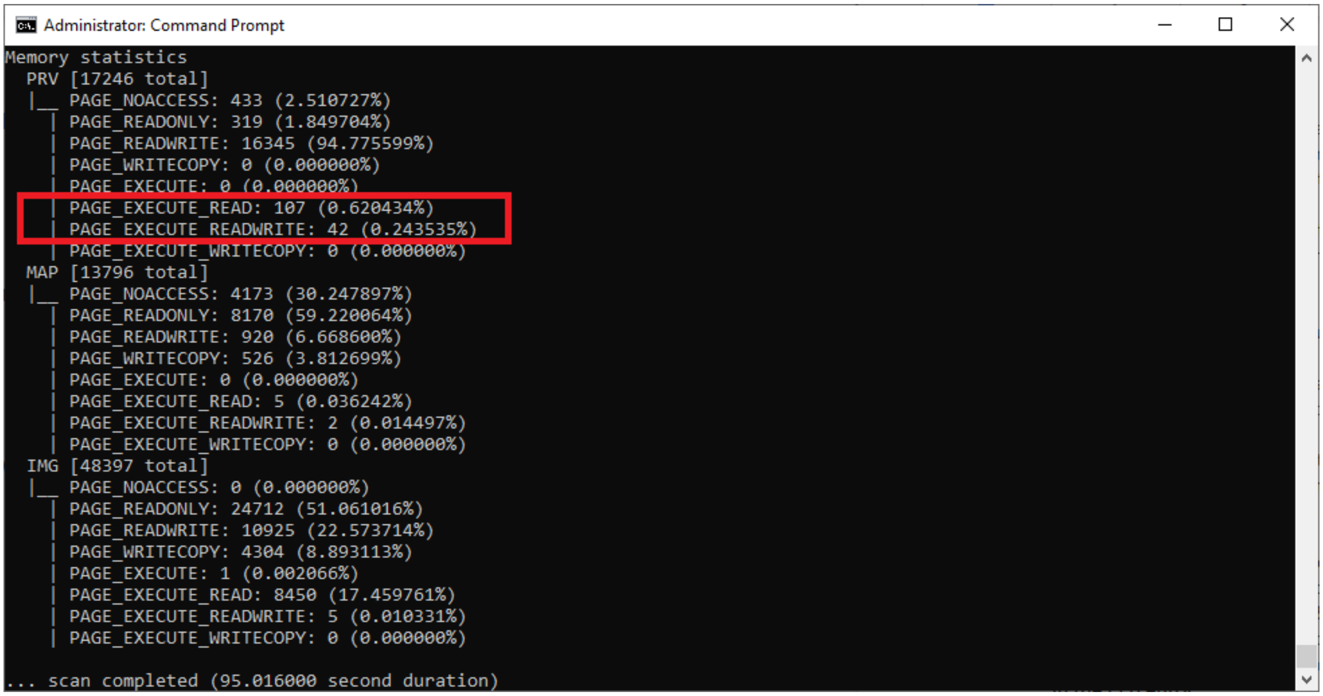
While there are many such false positives, many of which cannot be discussed here due to time and space constraints my conclusion is that they are distinctly finite. With the exception of 3rd party applications making use of usermode hooks, the IOCs which trigger false positives in Moneta are the result of specific subsystems within Windows itself and with sufficient time and effort can be universally eliminated through whitelisting.

Dynamic Code

Windows contains a seldomly discussed exploit mitigation feature called ***Arbitrary Code Guard (ACG)***. It is one of many process mitigation policies (most commonly known for ***DEP***, ***ASLR*** and ***CFG***) which makes its host process unable to "generate dynamic code or modify existing executable code." In practice this translates to a restriction on the NTDLL.DLL!NtAllocateVirtualMemory, NTDLL.DLL!NtProtectVirtualMemory, and NTDLL.DLL!NtMapViewOfSection APIs. In essence, it prevents all code which is not loaded via the mapping of a section created with the *SEC_IMAGE* flag from being allocated in the first place when the *PAGE_EXECUTE* permission is requested. It also prevents the addition of the *PAGE_EXECUTE* permission to any existing memory region regardless of its type. This information illustrates that Microsoft has its own definition of dynamic code and considers its definition sufficient for an exploit mitigation policy. Moneta, whose primary mechanism for creating IOC is the detection of dynamic code is based upon this same definition. In theory a combination of ***ACG*** and Code Integrity Guard (which prevents any unsigned image section from being mapped into the process) should make it impossible to introduce any unsigned code into memory, as there are only several ways to do so:

1. Allocating private or mapped memory as *+RWX*, writing code to it and executing. This technique is mitigated by A***CG***.
2. Allocating or overwriting existing private, mapped or image memory as *+RW*, writing code to it and then modifying it to be *+X* before executing. This technique is mitigated by ***ACG***.
3. Writing the code in the form of a PE file to disk and then mapping it into the process as an image. This technique is mitigated by ***Code Integrity Guard (CIG)***.
4. ***Recycling an existing +RWX region of mapped, image or private memory. Such memory regions can be considered to be pre-existing dynamic code.***
5. Phantom DLL hollowing - **the only technique which is capable of bypassing ACG and CIG** if there is no existing *+RWX* region available to recycle. Credit is due to Omer Yair, the Endpoint Team Lead at Symantec for making me aware of this potential use of phantom DLL hollowing in exploit writing. ***EDIT - 9/13/2020 - NtCreateSection now returns error 0xC0000428 (STATUS_INVALID_IMAGE_HASH) from CIG enabled processes if a modified TxF file handle is used.***

The remainder of this section will focus on the topic of recycling existing *+RWX* regions of dynamic code. While the pickings are relatively sparse, there are consistent phenomena within existing Windows subsystems which produce such memory. Those who remember the first post of this series may see this statement as a contradiction of one of the fundamental principles it was based upon, namely that legitimate executable memory within the average process is exclusively the domain of *+RX* image mappings associated with ***.text*** sections. Time has proven this assertion to be false, and Moneta clearly demonstrates this when asked to provide statistics on memory region types and their corresponding permissions on a Windows 10 OS:



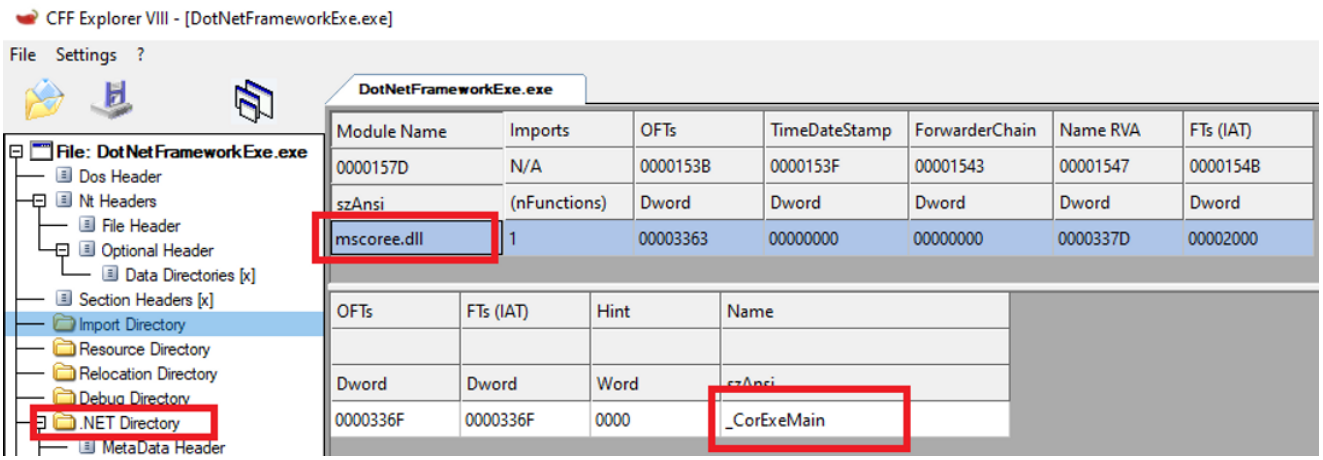
```
Memory statistics
  PRV [17246 total]
  |__ PAGE_NOACCESS: 433 (2.510727%)
    | PAGE_READONLY: 319 (1.849704%)
    | PAGE_READWRITE: 16345 (94.775599%)
    | PAGE_WRITECOPY: 0 (0.000000%)
    | PAGE_EXECUTE: 0 (0.000000%)
    | PAGE_EXECUTE_READ: 107 (0.620434%)
    | PAGE_EXECUTE_READWRITE: 42 (0.243535%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
  MAP [13796 total]
  |__ PAGE_NOACCESS: 4173 (30.247897%)
    | PAGE_READONLY: 8170 (59.220064%)
    | PAGE_READWRITE: 920 (6.668600%)
    | PAGE_WRITECOPY: 526 (3.812699%)
    | PAGE_EXECUTE: 0 (0.000000%)
    | PAGE_EXECUTE_READ: 5 (0.036242%)
    | PAGE_EXECUTE_READWRITE: 2 (0.014497%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
  IMG [48397 total]
  |__ PAGE_NOACCESS: 0 (0.000000%)
    | PAGE_READONLY: 24712 (51.061016%)
    | PAGE_READWRITE: 10925 (22.573714%)
    | PAGE_WRITECOPY: 4304 (8.893113%)
    | PAGE_EXECUTE: 1 (0.002066%)
    | PAGE_EXECUTE_READ: 8450 (17.459761%)
    | PAGE_EXECUTE_READWRITE: 5 (0.010331%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)

... scan completed (95.016000 second duration)
```

Figure 15 - Memory type/permission statistics from Moneta

Although this executable private memory accounts for less than 1% of the total private memory in all processes on the OS, at over 200 total regions it raises an extremely interesting question: if malware is not allocating these dynamic regions of memory, then who is?

When I first began testing Moneta this was the question that prompted me to begin reverse engineering the Common Language Runtime (CLR). The **clr.dll** module, I quickly observed, was a consistent feature of every single process I encountered which contained regions of private +*RWX* memory. The CLR is a framework that supports managed (.NET) code within a native process. Notably, there is no such thing as a "managed process" and all .NET code, whether it be C# or VB.NET runs within a virtualized environment within a normal Windows process supported by native DLLs such as **NTDLL.DLL**, **Kernel32.dll** etc.

A .NET EXE can load native DLLs and vice versa. .NET PEs are just regular PEs which contain a .NET metadata header as a data directory. All of the same concepts which apply to a regular EXE or DLL apply to their .NET equivalents. The key difference is that when any PE with a .NET subsystem is loaded and initialized (more on this shortly) either as the primary EXE of a newly launched process or a .NET DLL being loaded into an existing process, it will cause a series of additional modules to be loaded. These modules are responsible for initializing the virtual environment (CLR) which will contain the managed code. I’ve created one such .NET EXE in C# targeting .NET 4.8 for demonstrative purposes:

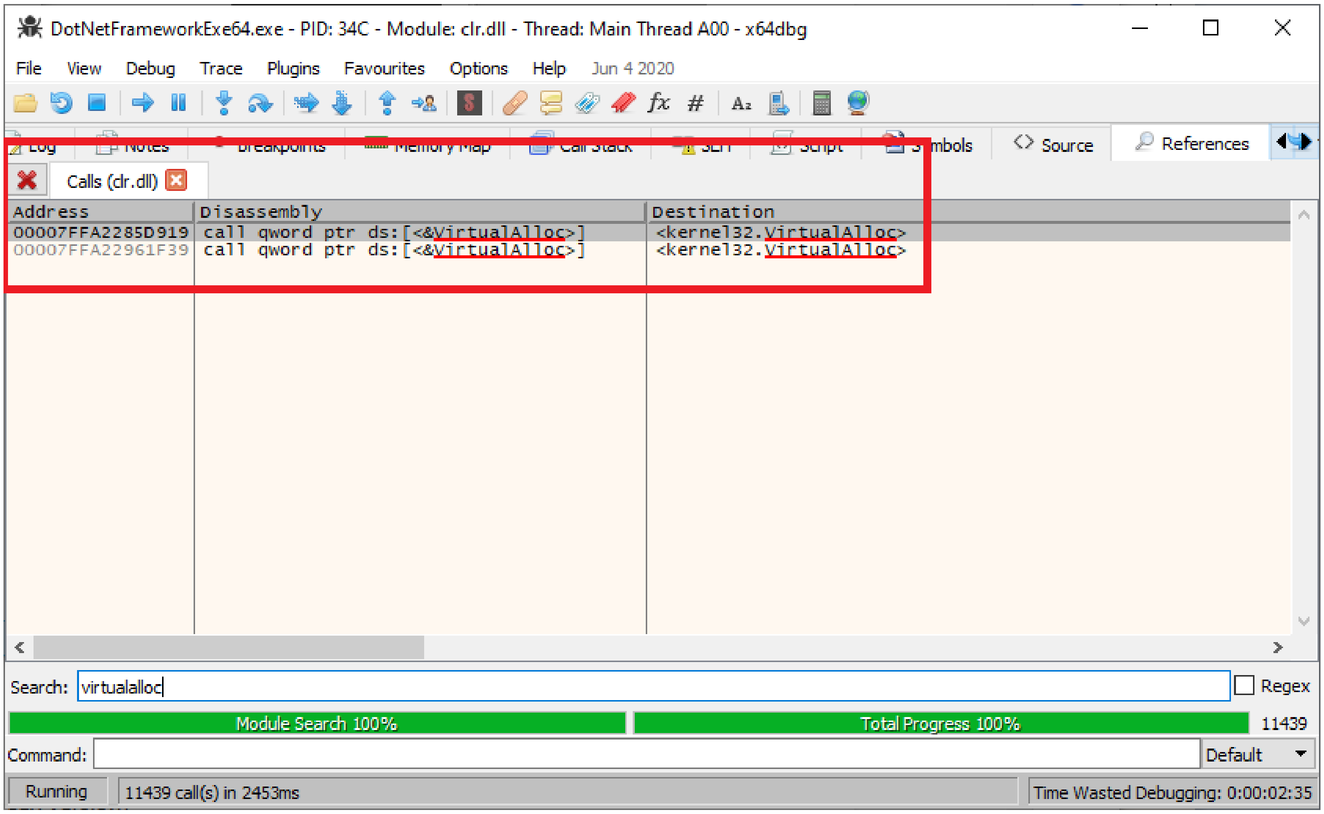


Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
0000157D	N/A	0000153B	0000153F	00001543	00001547	0000154B
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
mscoree.dll	1	00003363	00000000	00000000	0000337D	00002000

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
0000336F	0000336F	0000	_CorExeMain

Figure 16 - Import directory of .NET test EXE in CFF Explorer

.NET PEs contain a single native import, which is used to initialize the CLR and run their managed code. In the case of an EXE this function is _CorExeMain as seen above, and in the case of DLLs it is _CorDllMain. The native PE entry point specified in the *IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint* is simply a stub of code which calls this import. **clr.dll** has its own versions of these exports, for which the *_CorExeMain/_CorDllMain* exports of **mscoree.dll** are merely wrappers. It is within *_CorExeMain/_CorDllMain* in **clr.dll** that the real CLR initialization begins and the private *+RWX* regions begin to be created. When I began reverse engineering this code I initially set breakpoints on its references to KERNEL32.DLL!VirtualAlloc, of which there were two.



DotNetFrameworkExe64.exe - PID: 34C - Module: clr.dll - Thread: Main Thread A00 - x64dbg

Address	Disassembly	Destination
00007FFA2285D919	call qword ptr ds:[<&VirtualAlloc>]	<kernel32.VirtualAlloc>
00007FFA22961F39	call qword ptr ds:[<&VirtualAlloc>]	<kernel32.VirtualAlloc>

Search: virtualalloc

11439

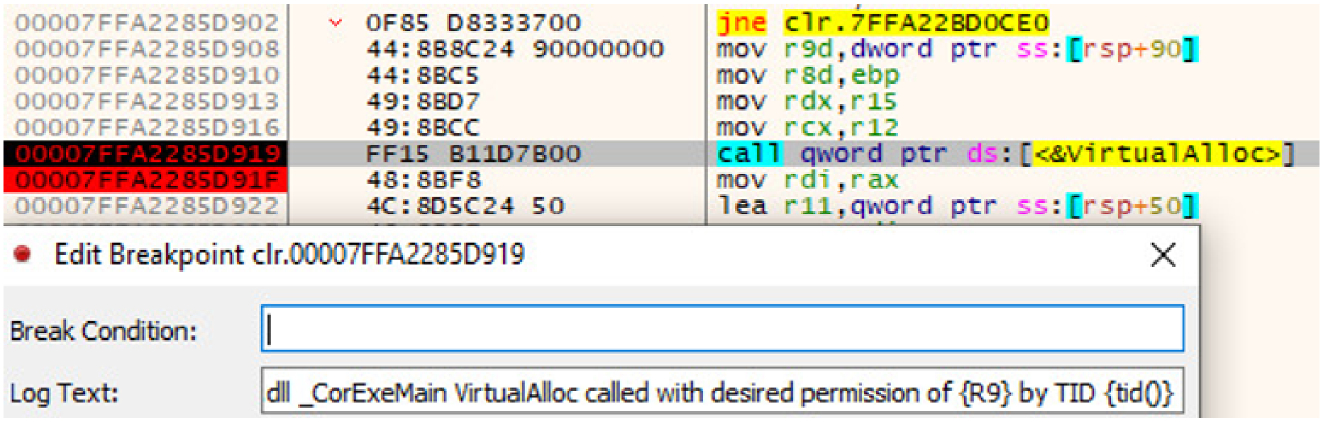
Command:

11439 call(s) in 2453ms

Time Wasted Debugging: 0:00:02:35

Figure 17 - Searching for intermodular references to KERNEL32.DLL!VirtualAlloc from clr.dll in memory within a .NET EXE being debugged from x64dbg

The first breakpoint records the permission KERNEL32.DLL!VirtualAlloc is called with (since this value is dynamic we can't simply read the assembly and know it). This is the 4th parameter and therefore is stored in the **R9** register.



```
00007FFA2285D902    0F85 D8333700        jne clr.7FFA22BD0CE0
00007FFA2285D908    44:8B8C24 90000000   mov r9d,dword ptr ss:[rsp+90]
00007FFA2285D910    44:8BC5              mov r8d,ebp
00007FFA2285D913    49:8BD7              mov rdx,r15
00007FFA2285D916    49:8BCC              mov rcx,r12
00007FFA2285D919    FF15 B11D7B00        call qword ptr ds:[<&VirtualAlloc>]
00007FFA2285D91F    48:8BF8              mov rdi,rax
00007FFA2285D922    4C:8D5C24 50         lea r11,qword ptr ss:[rsp+50]
```

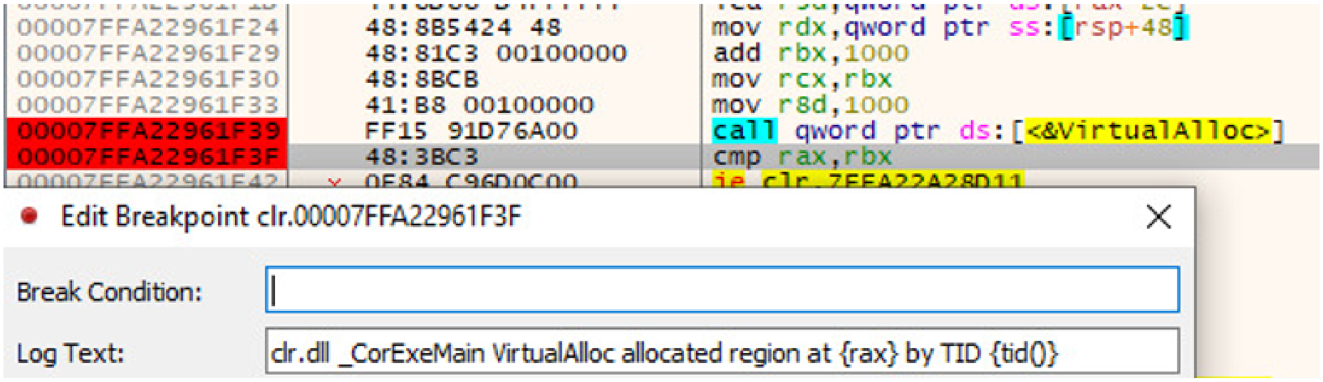
Edit Breakpoint clr.00007FFA2285D919

Break Condition:

Log Text: dll _CorExeMain VirtualAlloc called with desired permission of {R9} by TID {tid()}

Figure 18 - x64dbg instance of .NET EXE with a logging breakpoint on VirtualAlloc

The second breakpoint records the allocated region address returned by KERNEL32.DLL!VirtualAlloc in the **RAX** register.



```
00007FFA22961F24    48:8B5424 48        mov rdx,qword ptr ss:[rsp+48]
00007FFA22961F29    48:81C3 00100000    add rbx,1000
00007FFA22961F30    48:8BCB             mov rcx,rbx
00007FFA22961F33    41:B8 00100000      mov r8d,1000
00007FFA22961F39    FF15 91D76A00       call qword ptr ds:[<&VirtualAlloc>]
00007FFA22961F3F    48:3BC3             cmp rax,rbx
00007FFA22961F42  v 0F84 C96D0C00       je clr.7FFA22A28D11
```

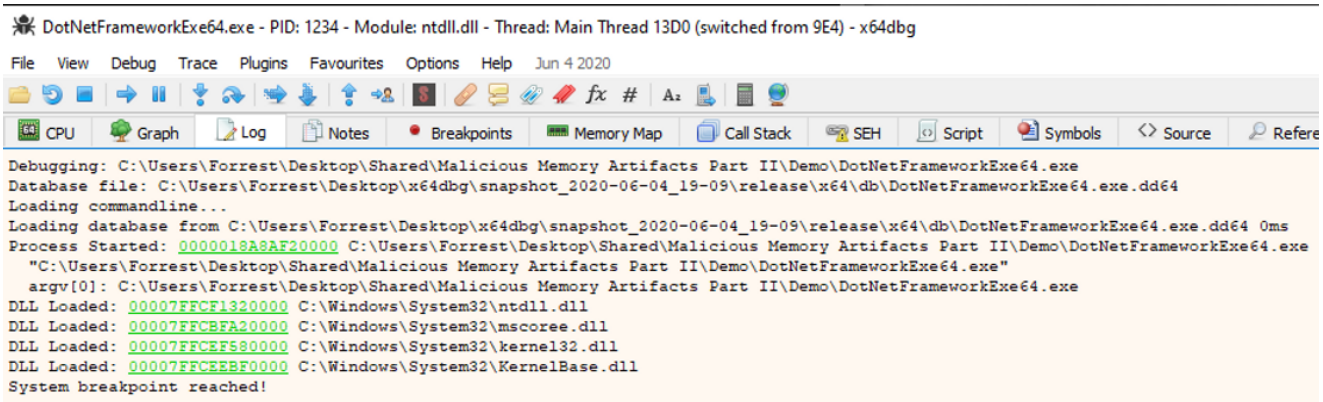
Edit Breakpoint clr.00007FFA22961F3F

Break Condition:

Log Text: clr.dll _CorExeMain VirtualAlloc allocated region at {rax} by TID {tid()}

Figure 19 - x64dbg instance of .NET EXE with a logging breakpoint after VirtualAlloc

An additional four breakpoints were set on the *_CorExeMain* start/return addresses in both **mscoree.dll** and **clr.dll**. Beginning the trace, the logs from x64dbg gradually illustrate what happens behind the scenes when a .NET EXE is loaded:

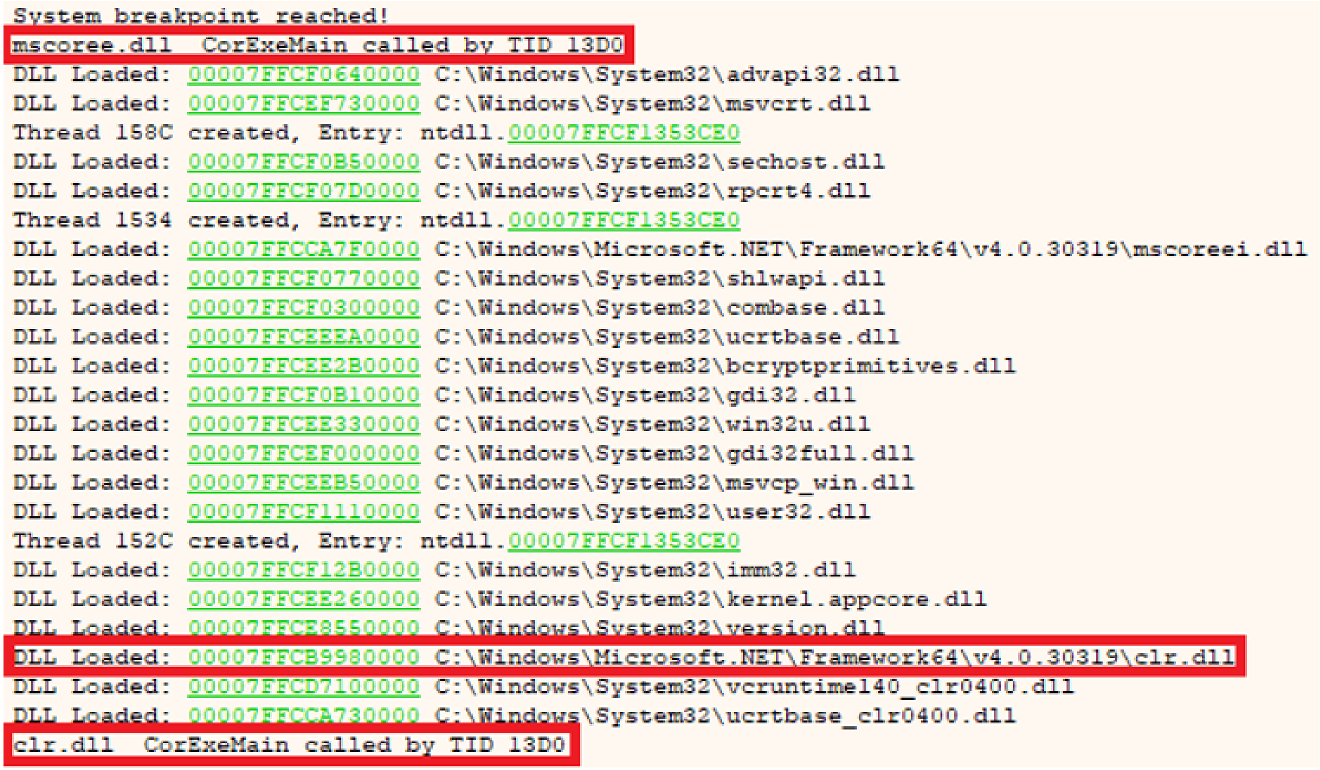


DotNetFrameworkExe64.exe - PID: 1234 - Module: ntdll.dll - Thread: Main Thread 13D0 (switched from 9E4) - x64dbg

```
Debugging: C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\DotNetFrameworkExe64.exe
Database file: C:\Users\Forrest\Desktop\x64dbg\snapshot_2020-06-04_19-09\release\x64\db\DotNetFrameworkExe64.exe.dd64
Loading commandline...
Loading database from C:\Users\Forrest\Desktop\x64dbg\snapshot_2020-06-04_19-09\release\x64\db\DotNetFrameworkExe64.exe.dd64 0ms
Process Started: 0000018A8AF20000 C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\DotNetFrameworkExe64.exe
  "C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\DotNetFrameworkExe64.exe"
  argv[0]: C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\DotNetFrameworkExe64.exe
DLL Loaded: 00007FFCF1320000 C:\Windows\System32\ntdll.dll
DLL Loaded: 00007FFCBFA20000 C:\Windows\System32\mscoree.dll
DLL Loaded: 00007FFCEF580000 C:\Windows\System32\kernel32.dll
DLL Loaded: 00007FFCEEBF0000 C:\Windows\System32\KernelBase.dll
System breakpoint reached!
```

Figure 20 - x64dbg log trace of .NET EXE

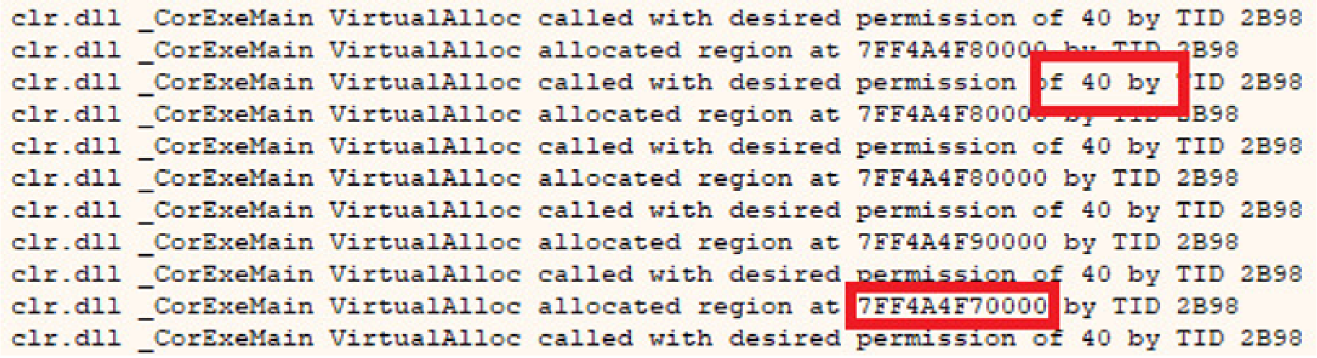
First, the main EXE loads its baseline native modules and primary import of **mscoree.dll**. At this point the default system breakpoint is hit.



```
System breakpoint reached!
mscoree.dll  CorExeMain called by TID 13D0
DLL Loaded: 00007FFCF0640000 C:\Windows\System32\advapi32.dll
DLL Loaded: 00007FFCEF730000 C:\Windows\System32\msvcrt.dll
Thread 158C created, Entry: ntdll.00007FFCF1353CE0
DLL Loaded: 00007FFCF0B50000 C:\Windows\System32\sechost.dll
DLL Loaded: 00007FFCF07D0000 C:\Windows\System32\rpcrt4.dll
Thread 1534 created, Entry: ntdll.00007FFCF1353CE0
DLL Loaded: 00007FFCCA7F0000 C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscoreei.dll
DLL Loaded: 00007FFCF0770000 C:\Windows\System32\shlwapi.dll
DLL Loaded: 00007FFCF0300000 C:\Windows\System32\combase.dll
DLL Loaded: 00007FFCEEEA0000 C:\Windows\System32\ucrtbase.dll
DLL Loaded: 00007FFCEE2B0000 C:\Windows\System32\bcryptprimitives.dll
DLL Loaded: 00007FFCF0B10000 C:\Windows\System32\gdi32.dll
DLL Loaded: 00007FFCEE330000 C:\Windows\System32\win32u.dll
DLL Loaded: 00007FFCEF000000 C:\Windows\System32\gdi32full.dll
DLL Loaded: 00007FFCEEB50000 C:\Windows\System32\msvcp_win.dll
DLL Loaded: 00007FFCF1110000 C:\Windows\System32\user32.dll
Thread 152C created, Entry: ntdll.00007FFCF1353CE0
DLL Loaded: 00007FFCF12B0000 C:\Windows\System32\imm32.dll
DLL Loaded: 00007FFCEE260000 C:\Windows\System32\kernel.appcore.dll
DLL Loaded: 00007FFCE8550000 C:\Windows\System32\version.dll
DLL Loaded: 00007FFCB9980000 C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
DLL Loaded: 00007FFCD7100000 C:\Windows\System32\vcruntime140_clr0400.dll
DLL Loaded: 00007FFCCA730000 C:\Windows\System32\ucrtbase_clr0400.dll
clr.dll  CorExeMain called by TID 13D0
```

Figure 21 - x64dbg log trace of .NET EXE

As seen in ***Figure 21*** the primary thread of the application calls through the *IMAGE_OPTIONAL_HEADER.AddressOfEntryPoint* into MSCOREE.DLL!_CorExeMain, which in turn loads the prerequisite .NET environment modules and calls CLR.DLL!_CorExeMain.

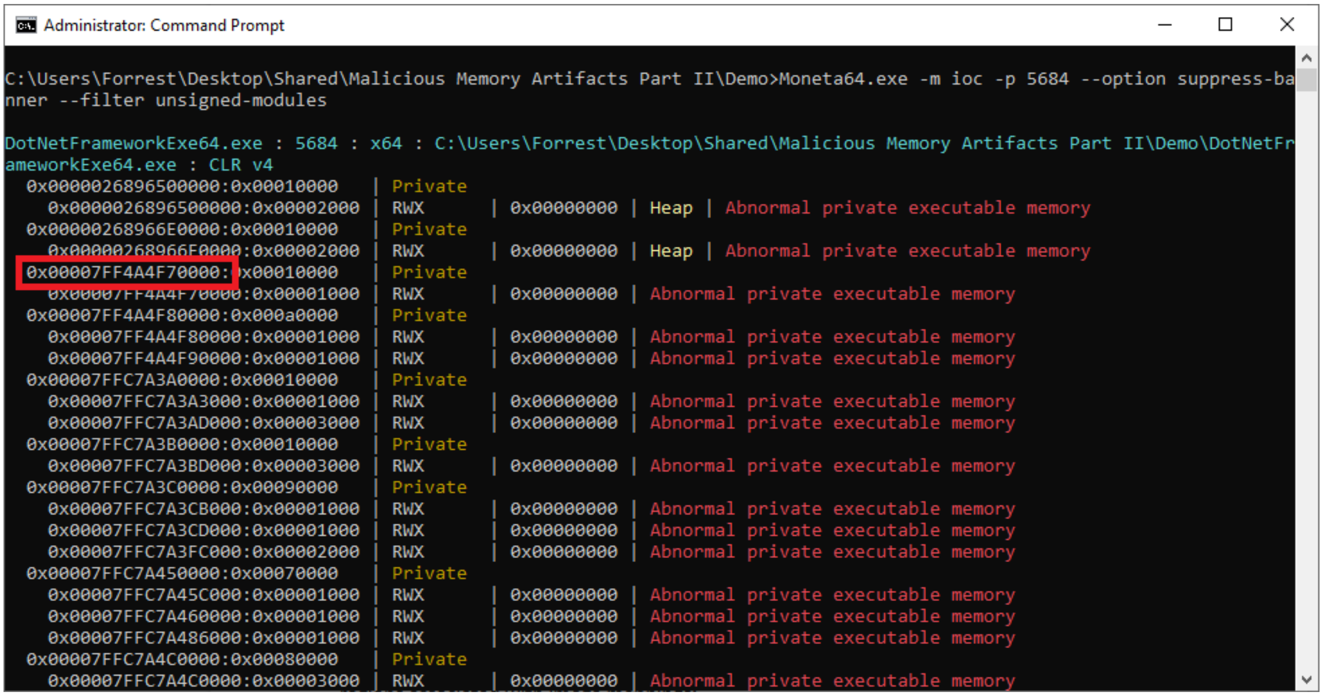


```
clr.dll _CorExeMain VirtualAlloc called with desired permission of 40 by TID 2B98
clr.dll _CorExeMain VirtualAlloc allocated region at 7FF4A4F80000 by TID 2B98
clr.dll _CorExeMain VirtualAlloc called with desired permission of 40 by TID 2B98
clr.dll _CorExeMain VirtualAlloc allocated region at 7FF4A4F80000 by TID 2B98
clr.dll _CorExeMain VirtualAlloc called with desired permission of 40 by TID 2B98
clr.dll _CorExeMain VirtualAlloc allocated region at 7FF4A4F80000 by TID 2B98
clr.dll _CorExeMain VirtualAlloc called with desired permission of 40 by TID 2B98
clr.dll _CorExeMain VirtualAlloc allocated region at 7FF4A4F90000 by TID 2B98
clr.dll _CorExeMain VirtualAlloc called with desired permission of 40 by TID 2B98
clr.dll _CorExeMain VirtualAlloc allocated region at 7FF4A4F70000 by TID 2B98
clr.dll _CorExeMain VirtualAlloc called with desired permission of 40 by TID 2B98
```

Figure 22 - x64dbg log trace of .NET EXE

While not all of the captured *VirtualAlloc* calls from CLR.DLL!_CorExeMain are requesting PAGE_EXECUTE_READWRITE memory, a substantial number are, as is shown in ***Figure 22*** above where a permission of *0x40* is being requested through the **R9** register.

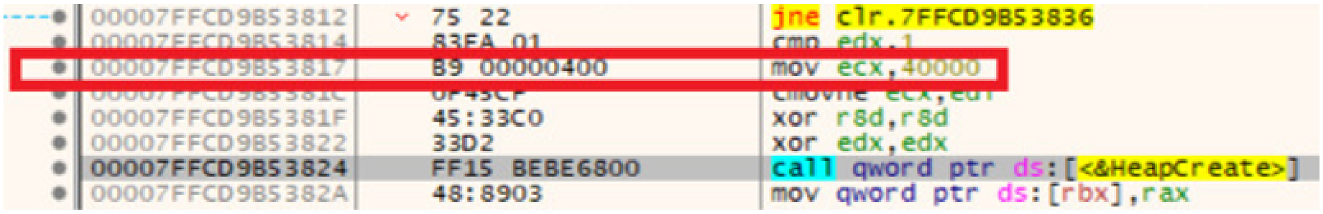
Enumerating the memory address space of this .NET EXE using Moneta we can see a great deal of the *+RWX* memory allocated in ***Figure 22*** appear as IOCs:



```
C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo>Moneta64.exe -m ioc -p 5684 --option suppress-ba
nner --filter unsigned-modules

DotNetFrameworkExe64.exe : 5684 : x64 : C:\Users\Forrest\Desktop\Shared\Malicious Memory Artifacts Part II\Demo\DotNetFr
ameworkExe64.exe : CLR v4
  0x0000026896500000:0x00010000   | Private
    0x0000026896500000:0x00002000 | RWX      | 0x00000000 | Heap | Abnormal private executable memory
  0x00000268966E0000:0x00010000   | Private
    0x00000268966F0000:0x00002000 | RWX      | 0x00000000 | Heap | Abnormal private executable memory
  0x00007FF4A4F70000:0x00010000   | Private
    0x00007FF4A4F70000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
  0x00007FF4A4F80000:0x000a0000   | Private
    0x00007FF4A4F80000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
    0x00007FF4A4F90000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
  0x00007FFC7A3A0000:0x00010000   | Private
    0x00007FFC7A3A3000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
    0x00007FFC7A3AD000:0x00003000 | RWX      | 0x00000000 | Abnormal private executable memory
  0x00007FFC7A3B0000:0x00010000   | Private
    0x00007FFC7A3BD000:0x00003000 | RWX      | 0x00000000 | Abnormal private executable memory
  0x00007FFC7A3C0000:0x00090000   | Private
    0x00007FFC7A3CB000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
    0x00007FFC7A3CD000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
    0x00007FFC7A3FC000:0x00002000 | RWX      | 0x00000000 | Abnormal private executable memory
  0x00007FFC7A450000:0x00070000   | Private
    0x00007FFC7A45C000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
    0x00007FFC7A460000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
    0x00007FFC7A486000:0x00001000 | RWX      | 0x00000000 | Abnormal private executable memory
  0x00007FFC7A4C0000:0x00080000   | Private
    0x00007FFC7A4C0000:0x00003000 | RWX      | 0x00000000 | Abnormal private executable memory
```

Figure 24 - Moneta IOC scan of the .NET EXE process open in x64dbg

Notably, upon closer inspection the *+RWX* regions shown as IOCs in the Moneta scan match those allocated by KERNEL32.DLL!VirtualAlloc from CLR.DLL!_CorExeMain (one such example is highlighted in ***Figures 22 and 24***). There are however two regions shown in the Moneta IOC results which do not correspond to any of the traced KERNEL32.DLL!VirtualAlloc calls. These are the two regions which appear near the top of ***Figure 24*** with the “Heap” attribute. Searching the code of **clr.dll** we can indeed see a reference to the KERNEL32.DLL!HeapCreate API:



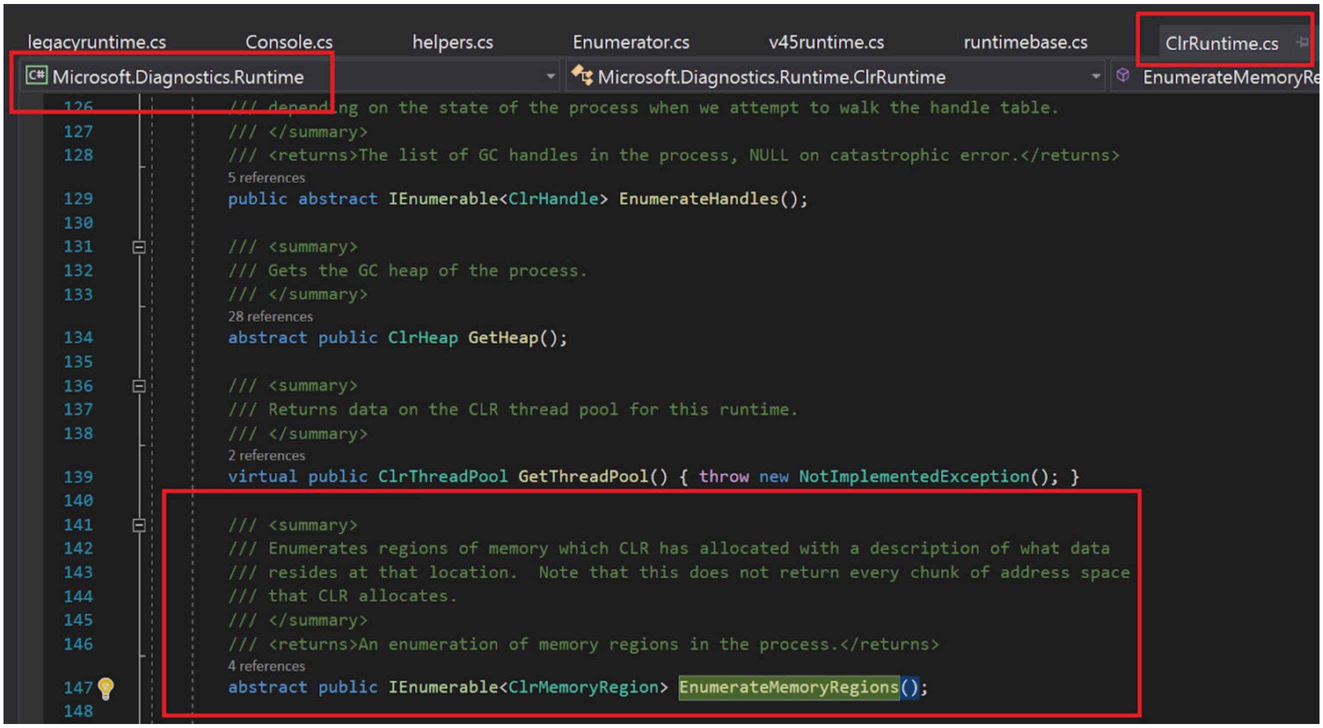
```
00007FFCD9B53812  75 22          jne clr.7FFCD9B53836
00007FFCD9B53814  83FA 01        cmp edx,1
00007FFCD9B53817  B9 00000400    mov ecx,40000
00007FFCD9B5381C  0F45CF         cmovne ecx,edi
00007FFCD9B5381F  45:33C0        xor r8d,r8d
00007FFCD9B53822  33D2           xor edx,edx
00007FFCD9B53824  FF15 BEBE6800  call qword ptr ds:[<&HeapCreate>]
00007FFCD9B5382A  48:8903        mov qword ptr ds:[rbx],rax
```

Figure 25 - Subroutine of clr.dll creating an executable heap

The key detail of this stub of code is the value that **ECX** (the first parameter of *HeapCreate*) is being initialized to which is *0x40000*. This constant corresponds to the *HEAP_CREATE_ENABLE_EXECUTE* option flag, which will cause the resulting heap to be allocated with *+RWX* permissions, explaining the *+RWX* heaps generated as a result of CLR initialization. These native heaps, recorded in the PEB, are notably distinct from the virtual CLR heaps which are only queryable through .NET debugging APIs.

This analysis explains the origins of the private *+RWX* regions but it doesn't explain their purpose - a detail which is key to whitelisting them to avoid false positives. After all, if we can programmatically query the regions of memory associated with the .NET subsystem in a process then we can use this data as a filter to distinguish between legitimately allocated dynamic code stemming from the CLR and unknown dynamic code to mark as an IOC. Answering this question proved to be an exceptionally time consuming and part of this research, and I believe some high-level details will help to enhance the knowledge of the reader in what has proven to be a very obscure and undocumented area of Windows.

Windows contains an obscure and poorly documented DLL called **mscoredacwks.dll** which hosts a ***Data Access Control*** (DAC) COM interface intended to allow native debugging of managed .NET code. Some cursory digging into the capabilities of these interfaces yields what appears to be promising results. One such example is the ICLRDataEnumMemoryRegions interface which purports to enumerate all regions of memory associated with the CLR environment of an attached process. This sounds like the perfect solution to developing an automated CLR whitelist, however in practice this interface proved to have a remarkably poor coverage of such memory (only enumerated about 20% of the *+RWX* regions we observed to be allocated by CLR.DLL!_CorExeMain). Seeking an alternative, I stumbled across ClrMD, a C# library designed for the specific purpose of interfacing with the ***DAC*** and containing what appeared to be a relevant code in the form of the *EnumerateMemoryRegions* method of its *ClrRuntime* class. Furthermore, this method does not rely upon the aforementioned *ICLRDataEnumMemoryRegions* interface and instead manually enumerates the heaps, app domains, modules and JIT code of its target.



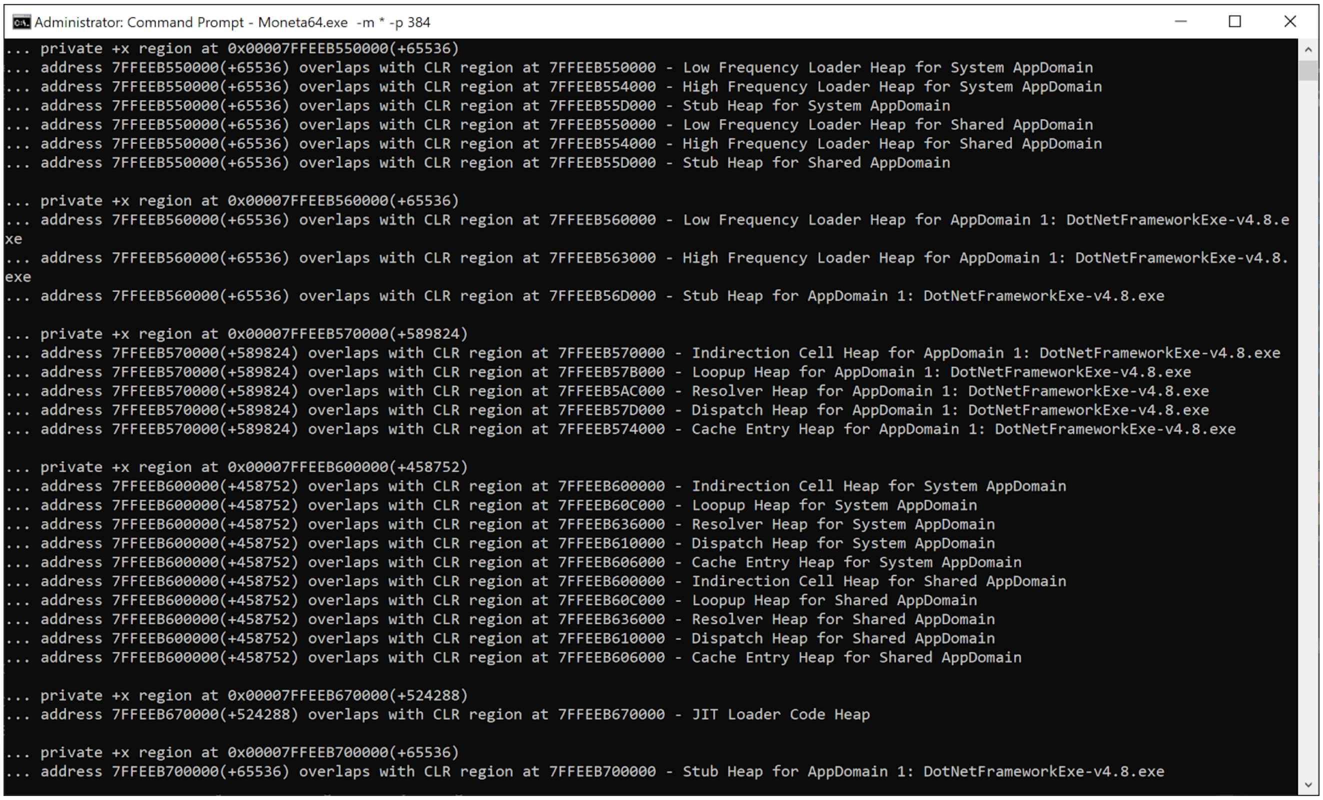
```
        /// depending on the state of the process when we attempt to walk the handle table.
        /// </summary>
        /// <returns>The list of GC handles in the process, NULL on catastrophic error.</returns>
        5 references
        public abstract IEnumerable<ClrHandle> EnumerateHandles();

        /// <summary>
        /// Gets the GC heap of the process.
        /// </summary>
        28 references
        abstract public ClrHeap GetHeap();

        /// <summary>
        /// Returns data on the CLR thread pool for this runtime.
        /// </summary>
        2 references
        virtual public ClrThreadPool GetThreadPool() { throw new NotImplementedException(); }

        /// <summary>
        /// Enumerates regions of memory which CLR has allocated with a description of what data
        /// resides at that location.  Note that this does not return every chunk of address space
        /// that CLR allocates.
        /// </summary>
        /// <returns>An enumeration of memory regions in the process.</returns>
        4 references
        abstract public IEnumerable<ClrMemoryRegion> EnumerateMemoryRegions();
```

Figure 27 - The definition of EnumerateMemoryRegions within ClrMD in Visual Studio

I wrote a small side project in C# (the same language as ClrMD) to interface between Moneta and the *EnumerateMemoryRegions* method over the command line, and created a modified version of the scanner to use this code to attempt to correlate the private *PAGE_EXECUTE_READWRITE* regions it enumerated with the CLR heaps described prior.

```
ulong Address = ...

using (var dataTarget = DataTarget.AttachToProcess(Pid, 10000, AttachFlag.Invasive))
{
ClrInfo clrVersion = dataTarget.ClrVersions[0];
ClrRuntime clrRuntime = clrVersion.CreateRuntime();

foreach (ClrMemoryRegion clrMemoryRegion in clrRuntime.EnumerateMemoryRegions())
{
```

```
if (RegionOverlap(Address, RegionSize, clrMemoryRegion.Address, clrMemoryRegion.Size))

{

Console.WriteLine("... address {0:X}(+{1}) overlaps with CLR region at {2:X} - {3}", Address, RegionSize, clrMemoryRegion.Address, clrMemoryRegion.ToString(true));

}

}

}
```



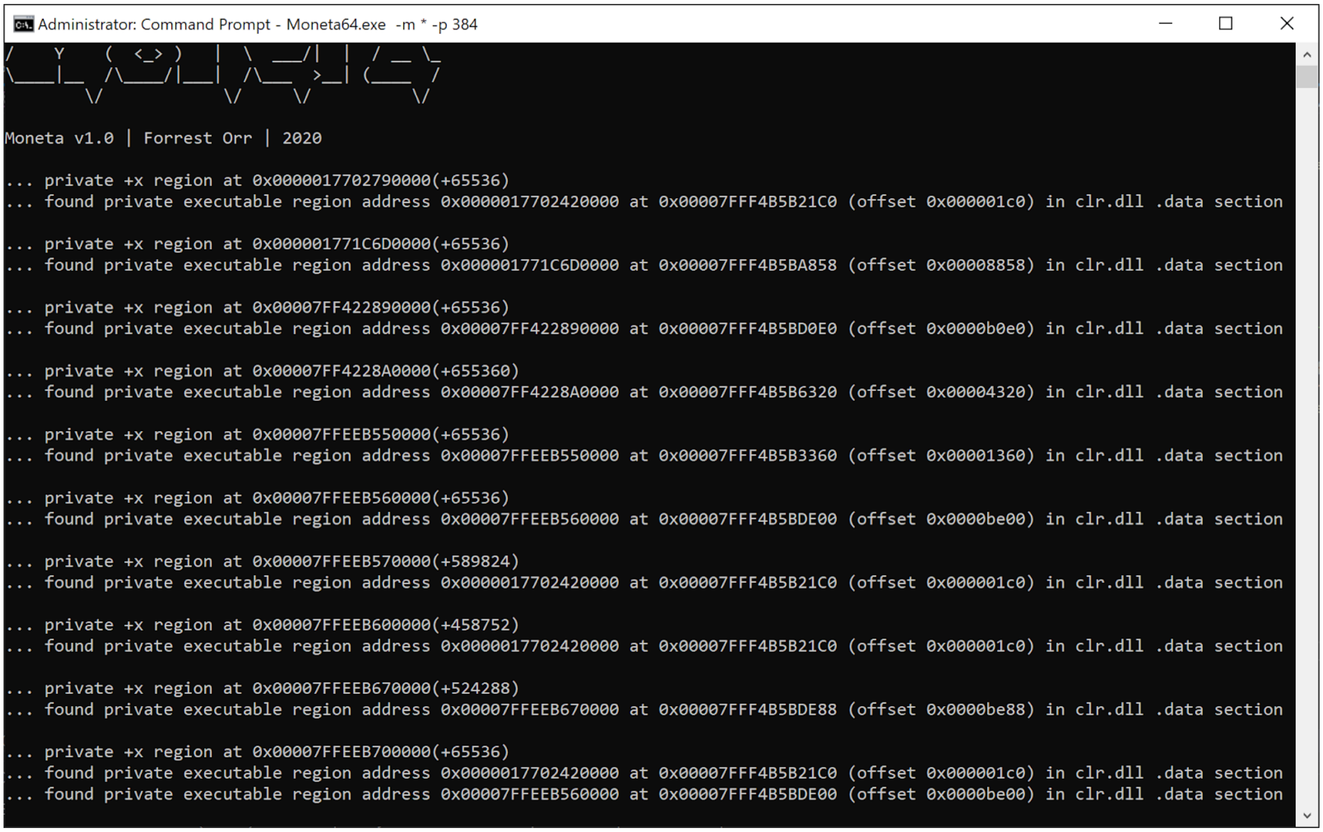
```
... private +x region at 0x00007FFEEB550000(+65536)
... address 7FFEEB550000(+65536) overlaps with CLR region at 7FFEEB550000 - Low Frequency Loader Heap for System AppDomain
... address 7FFEEB550000(+65536) overlaps with CLR region at 7FFEEB554000 - High Frequency Loader Heap for System AppDomain
... address 7FFEEB550000(+65536) overlaps with CLR region at 7FFEEB55D000 - Stub Heap for System AppDomain
... address 7FFEEB550000(+65536) overlaps with CLR region at 7FFEEB550000 - Low Frequency Loader Heap for Shared AppDomain
... address 7FFEEB550000(+65536) overlaps with CLR region at 7FFEEB554000 - High Frequency Loader Heap for Shared AppDomain
... address 7FFEEB550000(+65536) overlaps with CLR region at 7FFEEB55D000 - Stub Heap for Shared AppDomain

... private +x region at 0x00007FFEEB560000(+65536)
... address 7FFEEB560000(+65536) overlaps with CLR region at 7FFEEB560000 - Low Frequency Loader Heap for AppDomain 1: DotNetFrameworkExe-v4.8.e
xe
... address 7FFEEB560000(+65536) overlaps with CLR region at 7FFEEB563000 - High Frequency Loader Heap for AppDomain 1: DotNetFrameworkExe-v4.8.
exe
... address 7FFEEB560000(+65536) overlaps with CLR region at 7FFEEB56D000 - Stub Heap for AppDomain 1: DotNetFrameworkExe-v4.8.exe

... private +x region at 0x00007FFEEB570000(+589824)
... address 7FFEEB570000(+589824) overlaps with CLR region at 7FFEEB570000 - Indirection Cell Heap for AppDomain 1: DotNetFrameworkExe-v4.8.exe
... address 7FFEEB570000(+589824) overlaps with CLR region at 7FFEEB57B000 - Loopup Heap for AppDomain 1: DotNetFrameworkExe-v4.8.exe
... address 7FFEEB570000(+589824) overlaps with CLR region at 7FFEEB5AC000 - Resolver Heap for AppDomain 1: DotNetFrameworkExe-v4.8.exe
... address 7FFEEB570000(+589824) overlaps with CLR region at 7FFEEB57D000 - Dispatch Heap for AppDomain 1: DotNetFrameworkExe-v4.8.exe
... address 7FFEEB570000(+589824) overlaps with CLR region at 7FFEEB574000 - Cache Entry Heap for AppDomain 1: DotNetFrameworkExe-v4.8.exe

... private +x region at 0x00007FFEEB600000(+458752)
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB600000 - Indirection Cell Heap for System AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB60C000 - Loopup Heap for System AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB636000 - Resolver Heap for System AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB610000 - Dispatch Heap for System AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB606000 - Cache Entry Heap for System AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB600000 - Indirection Cell Heap for Shared AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB60C000 - Loopup Heap for Shared AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB636000 - Resolver Heap for Shared AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB610000 - Dispatch Heap for Shared AppDomain
... address 7FFEEB600000(+458752) overlaps with CLR region at 7FFEEB606000 - Cache Entry Heap for Shared AppDomain

... private +x region at 0x00007FFEEB670000(+524288)
... address 7FFEEB670000(+524288) overlaps with CLR region at 7FFEEB670000 - JIT Loader Code Heap

... private +x region at 0x00007FFEEB700000(+65536)
... address 7FFEEB700000(+65536) overlaps with CLR region at 7FFEEB700000 - Stub Heap for AppDomain 1: DotNetFrameworkExe-v4.8.exe
```

Figure 28 - Modified instance of Moneta designed to correlate private +x regions with CLR regions using ClrMD

The results, seen above in ***Figure 28*** show that these private *+RWX* regions correspond to the low frequency loader, high frequency loader, stub, indirection call, lookup, resolver, dispatch, cache entry and JIT loader heaps associated with all of the App Domains of the .NET process. In the case of this test EXE, this is only the ***System*** and ***Shared*** App Domains (which are present in all .NET environments) along with the App Domain corresponding to the main EXE itself. For a further explanation of App Domains and how managed assemblies are loaded I suggest reading XPN’s blog or the Microsoft documentation on the topic.

Despite the high rate of correlation, it was not 100%. There were consistently 2 or more private *+RWX* regions in every .NET process I analyzed which could not be accounted for using ClrMD. After a great deal of reversing and even manually fixing bugs in ClrMD I came to the conclusion that the documentation on the topic was too poor to fix this problem short of reversing the entire CLR, which I was not willing to do. There seems to be no existing API or project (not even written by Microsoft) which can reliably parse the CLR heap and enumerate its associated memory regions.

With this path closed to me I opted for a more simplistic approach to the issue, instead focusing on identifying references to these *+RWX* regions as global variables stored within the ***.data*** section of **clr.dll** itself. This proved to be a highly effective solution to the problem, allowing me to introduce a whitelist filter for the CLR which I called **clr-prvx**.



```
/    Y    (  <_> )   |  \  ___/|  |  / __ \_
\____|__  /\____/|___|  /\___  >__| (____  /
        \/            \/     \/          \/

Moneta v1.0 | Forrest Orr | 2020

... private +x region at 0x0000017702790000(+65536)
... found private executable region address 0x0000017702420000 at 0x00007FFF4B5B21C0 (offset 0x000001c0) in clr.dll .data section

... private +x region at 0x000001771C6D0000(+65536)
... found private executable region address 0x000001771C6D0000 at 0x00007FFF4B5BA858 (offset 0x00008858) in clr.dll .data section

... private +x region at 0x00007FF422890000(+65536)
... found private executable region address 0x00007FF422890000 at 0x00007FFF4B5BD0E0 (offset 0x0000b0e0) in clr.dll .data section

... private +x region at 0x00007FF4228A0000(+655360)
... found private executable region address 0x00007FF4228A0000 at 0x00007FFF4B5B6320 (offset 0x00004320) in clr.dll .data section

... private +x region at 0x00007FFEEB550000(+65536)
... found private executable region address 0x00007FFEEB550000 at 0x00007FFF4B5B3360 (offset 0x00001360) in clr.dll .data section

... private +x region at 0x00007FFEEB560000(+65536)
... found private executable region address 0x00007FFEEB560000 at 0x00007FFF4B5BDE00 (offset 0x0000be00) in clr.dll .data section

... private +x region at 0x00007FFEEB570000(+589824)
... found private executable region address 0x0000017702420000 at 0x00007FFF4B5B21C0 (offset 0x000001c0) in clr.dll .data section

... private +x region at 0x00007FFEEB600000(+458752)
... found private executable region address 0x0000017702420000 at 0x00007FFF4B5B21C0 (offset 0x000001c0) in clr.dll .data section

... private +x region at 0x00007FFEEB670000(+524288)
... found private executable region address 0x00007FFEEB670000 at 0x00007FFF4B5BDE88 (offset 0x0000be88) in clr.dll .data section

... private +x region at 0x00007FFEEB700000(+65536)
... found private executable region address 0x0000017702420000 at 0x00007FFF4B5B21C0 (offset 0x000001c0) in clr.dll .data section
... found private executable region address 0x00007FFEEB560000 at 0x00007FFF4B5BDE00 (offset 0x0000be00) in clr.dll .data section
```

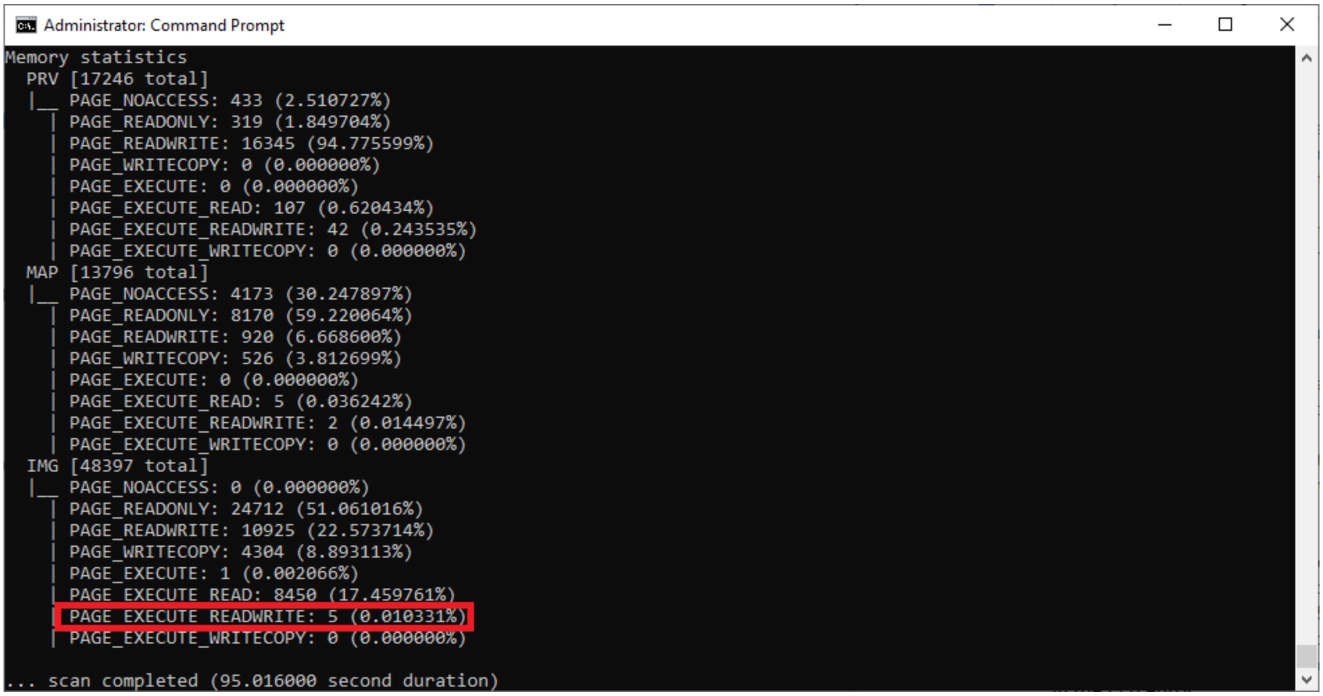
Figure 29 - Modified Moneta scanner enumerating references to all private +RWX memory regions in .NET EXE

Notably, in older versions of the .NET framework the **mscorwks.dll** module will be used for CLR initialization rather than **clr.dll** and will thus contain the references to globals in its own ***.data*** section. The only additional criteria needed to apply this CLR whitelist filter is to confirm that the process in question has had the CLR initialized in the first place. I discovered a nice trick to achieve this in the Process Hacker source code through use of a global section object, a technique which I adapted into my own routine used in Moneta:

```
int32_t nDotNetVersion = -1;
wchar_t SectionName[500] = { 0 };
static NtOpenSection_t NtOpenSection = reinterpret_cast<NtOpenSection_t>(GetProcAddress(GetModuleHandleW(L"ntdll.dll"), "NtOpenSection"));
static RtlInitUnicodeString_t RtlInitUnicodeString = reinterpret_cast<RtlInitUnicodeString_t>(GetProcAddress(GetModuleHandleW(L"ntdll.dll"), "RtlInitUnicodeString"));
UNICODE_STRING usSectionName = { 0 };
HANDLE hSection = nullptr;
OBJECT_ATTRIBUTES ObjAttr = { sizeof(OBJECT_ATTRIBUTES) };
NTSTATUS NtStatus;

_snwprintf_s(SectionName, 500, L"\\BaseNamedObjects\\Cor_Private_IPCBlock_v4_%d", dwPid);
RtlInitUnicodeString(&usSectionName, SectionName);
InitializeObjectAttributes(&ObjAttr, &usSectionName, OBJ_CASE_INSENSITIVE, nullptr, nullptr);
NtStatus = NtOpenSection(&hSection, SECTION_QUERY, &ObjAttr);

if (NT_SUCCESS(NtStatus)) {
nDotNetVersion = 4;
CloseHandle(hSection);
```

```
}
else if (NtStatus == 0xc0000022) { // Access denied also implies the object exists, which is all I care about.
nDotNetVersion = 4;
}

if (nDotNetVersion == -1) {
ZeroMemory(&usSectionName, sizeof(usSectionName));
ZeroMemory(&ObjAttr, sizeof(ObjAttr));
hSection = nullptr;

_snwprintf_s(SectionName, 500, L"\\BaseNamedObjects\\Cor_Private_IPCBlock_%d", dwPid);
RtlInitUnicodeString(&usSectionName, SectionName);
InitializeObjectAttributes(&ObjAttr, &usSectionName, OBJ_CASE_INSENSITIVE, nullptr, nullptr);
NtStatus = NtOpenSection(&hSection, SECTION_QUERY, &ObjAttr);

if (NT_SUCCESS(NtStatus)) {
nDotNetVersion = 2;
CloseHandle(hSection);
}
else if (NtStatus == 0xc0000022) {
nDotNetVersion = 2;
}
}
```

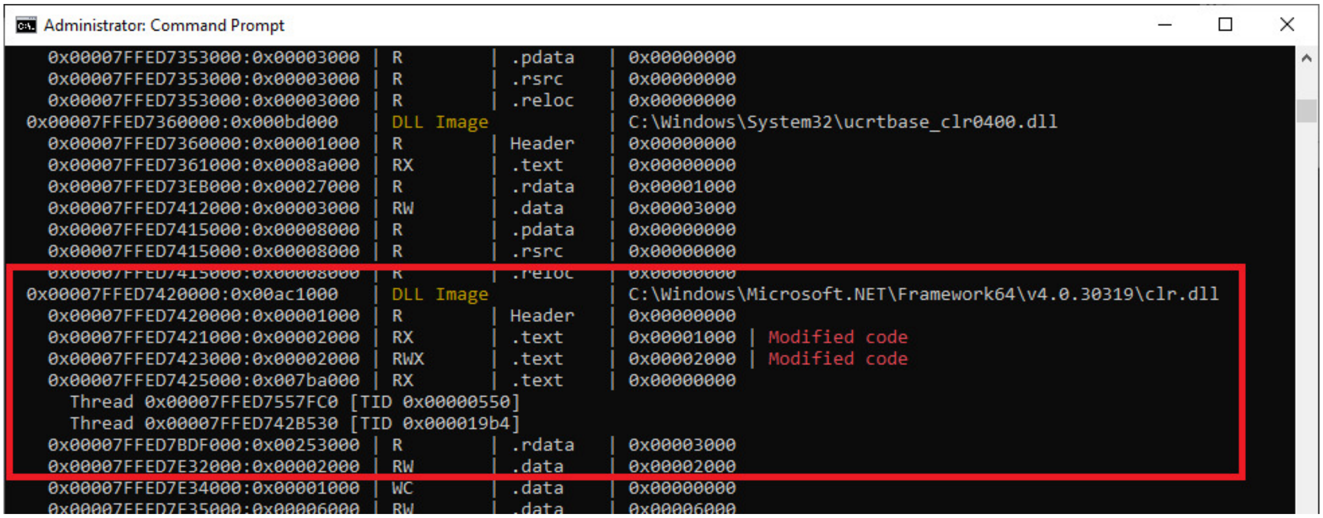
Private *+RWX* regions resulting from the CLR explain only a limited portion of the dynamic code which can appear as false positives. To describe them all is beyond the scope of this post, so I'll focus on one last interesting category of such memory - the *+RWX* regions associated with image mappings:



```
Memory statistics
  PRV [17246 total]
  |__ PAGE_NOACCESS: 433 (2.510727%)
    | PAGE_READONLY: 319 (1.849704%)
    | PAGE_READWRITE: 16345 (94.775599%)
    | PAGE_WRITECOPY: 0 (0.000000%)
    | PAGE_EXECUTE: 0 (0.000000%)
    | PAGE_EXECUTE_READ: 107 (0.620434%)
    | PAGE_EXECUTE_READWRITE: 42 (0.243535%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
  MAP [13796 total]
  |__ PAGE_NOACCESS: 4173 (30.247897%)
    | PAGE_READONLY: 8170 (59.220064%)
    | PAGE_READWRITE: 920 (6.668600%)
    | PAGE_WRITECOPY: 526 (3.812699%)
    | PAGE_EXECUTE: 0 (0.000000%)
    | PAGE_EXECUTE_READ: 5 (0.036242%)
    | PAGE_EXECUTE_READWRITE: 2 (0.014497%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)
  IMG [48397 total]
  |__ PAGE_NOACCESS: 0 (0.000000%)
    | PAGE_READONLY: 24712 (51.061016%)
    | PAGE_READWRITE: 10925 (22.573714%)
    | PAGE_WRITECOPY: 4304 (8.893113%)
    | PAGE_EXECUTE: 1 (0.002066%)
    | PAGE_EXECUTE_READ: 8450 (17.459761%)
    | PAGE_EXECUTE_READWRITE: 5 (0.010331%)
    | PAGE_EXECUTE_WRITECOPY: 0 (0.000000%)

... scan completed (95.016000 second duration)
```

Figure 30 - Moneta scan statistics highlighting +RWX image memory

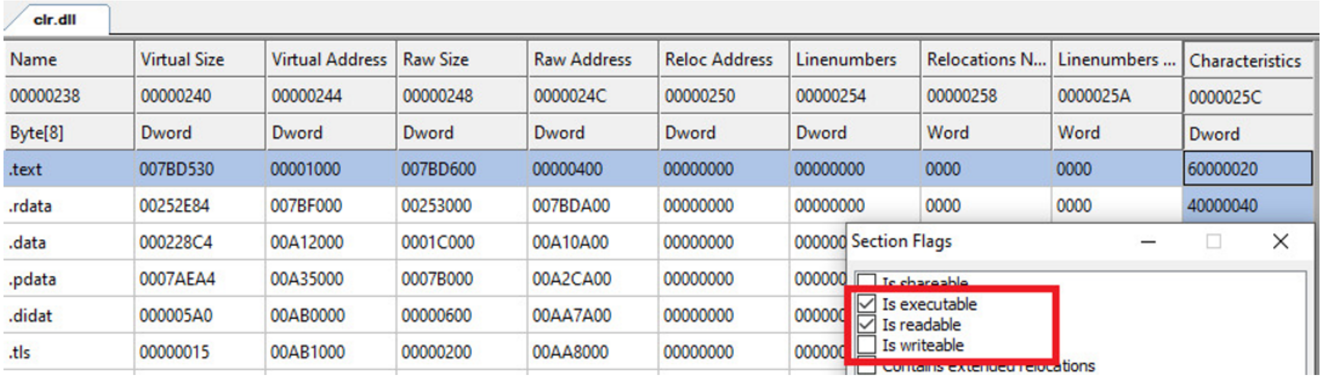
Although a rarity, some PEs contain *+RWX* sections. A prime example is the previously discussed **clr.dll**, a module which will consistently be loaded into processes targeting .NET framework 4.0+.



```
    0x00007FFED7353000:0x00003000 | R        | .pdata   | 0x00000000
    0x00007FFED7353000:0x00003000 | R        | .rsrc    | 0x00000000
    0x00007FFED7353000:0x00003000 | R        | .reloc   | 0x00000000
  0x00007FFED7360000:0x000bd000   | DLL Image           | C:\Windows\System32\ucrtbase_clr0400.dll
    0x00007FFED7360000:0x00001000 | R        | Header   | 0x00000000
    0x00007FFED7361000:0x0008a000 | RX       | .text    | 0x00000000
    0x00007FFED73EB000:0x00027000 | R        | .rdata   | 0x00001000
    0x00007FFED7412000:0x00003000 | RW       | .data    | 0x00003000
    0x00007FFED7415000:0x00008000 | R        | .pdata   | 0x00000000
    0x00007FFED7415000:0x00008000 | R        | .rsrc    | 0x00000000
    0x00007FFED7415000:0x00008000 | R        | .reloc   | 0x00000000
  0x00007FFED7420000:0x00ac1000   | DLL Image           | C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
    0x00007FFED7420000:0x00001000 | R        | Header   | 0x00000000
    0x00007FFED7421000:0x00002000 | RX       | .text    | 0x00001000 | Modified code
    0x00007FFED7423000:0x00002000 | RWX      | .text    | 0x00002000 | Modified code
    0x00007FFED7425000:0x007ba000 | RX       | .text    | 0x00000000
      Thread 0x00007FFED7557FC0 [TID 0x00000550]
      Thread 0x00007FFED742B530 [TID 0x000019b4]
    0x00007FFED7BDF000:0x00253000 | R        | .rdata   | 0x00003000
    0x00007FFED7E32000:0x00002000 | RW       | .data    | 0x00002000
    0x00007FFED7E34000:0x00001000 | WC       | .data    | 0x00000000
    0x00007FFED7E35000:0x00006000 | RW       | .data    | 0x00006000
```

Figure 31 - Dynamic code associated with clr.dll

The phenomena displayed above is a consistent attribute of **clr.dll**, appearing in every process where the CLR has been initialized. At *0x00007FFED7423000* two pages (0x2000 bytes) of memory has been privately paged into the host process, where an isolated enclave within the ***.text*** section has been made writable and modified at runtime. Interestingly, these *+RWX* permissions are not consistent with the **clr.dll** PE headers on disk.



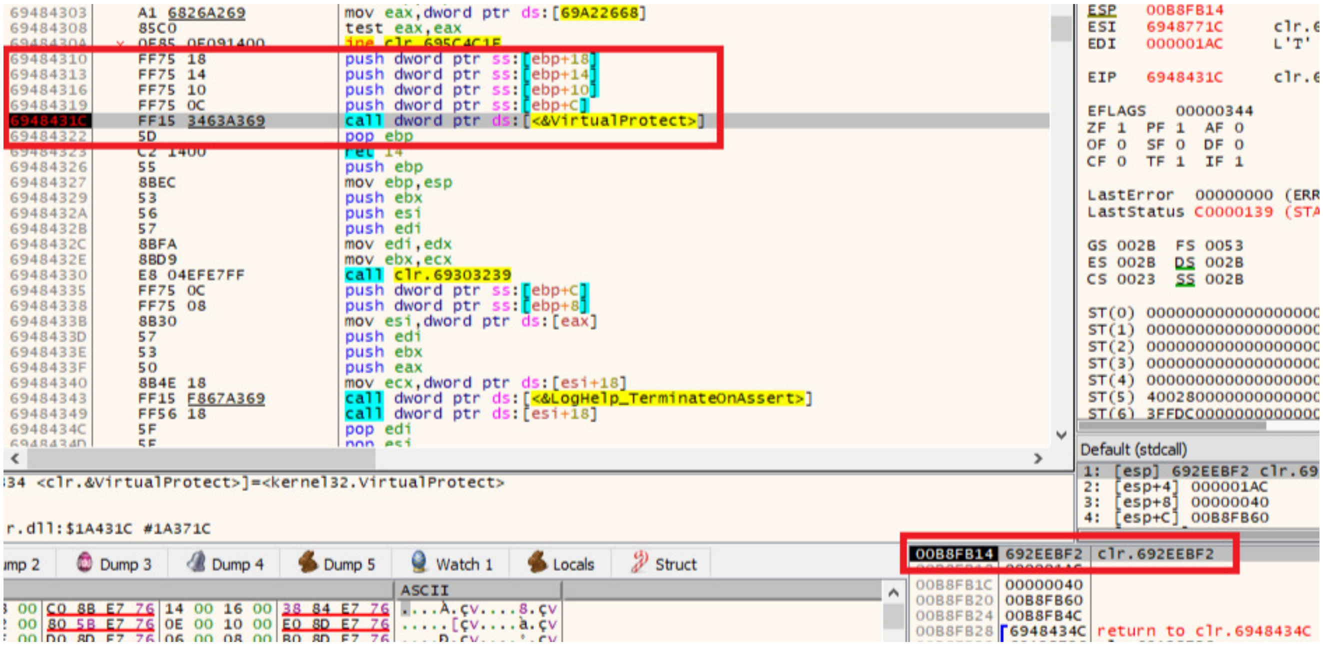
Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
00000238	00000240	00000244	00000248	0000024C	00000250	00000254	00000258	0000025A	0000025C
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	007BD530	00001000	007BD600	00000400	00000000	00000000	0000	0000	60000020
.rdata	00252E84	007BF000	00253000	007BDA00	00000000	00000000	0000	0000	40000040
.data	000228C4	00A12000	0001C000	00A10A00	00000000	000000			
.pdata	0007AEA4	00A35000	0007B000	00A2CA00	00000000	000000			
.didat	000005A0	00AB0000	00000600	00AA7A00	00000000	000000			
.tls	00000015	00AB1000	00000200	00AA8000	00000000	000000			

Section Flags

- ☐ Is shareable
- ☑ Is executable
- ☑ Is readable
- ☐ Is writeable
- ☐ Contains extended relocations

Figure 32 - clr.dll .text section permissions in CFF Explorer

This region is manually modified by CLR.DLL!_CorExeMain as part of the CLR initialization discussed earlier via a call to KERNEL32.DLL!VirtualProtect.



```
69484303     A1 6826A269      mov eax,dword ptr ds:[69A22668]
69484308     85C0             test eax,eax
6948430A   x 0F85 0F091400    jne clr.695C4C1E
69484310     FF75 18          push dword ptr ss:[ebp+18]
69484313     FF75 14          push dword ptr ss:[ebp+14]
69484316     FF75 10          push dword ptr ss:[ebp+10]
69484319     FF75 0C          push dword ptr ss:[ebp+C]
6948431C     FF15 3463A369    call dword ptr ds:[<&VirtualProtect>]
69484322     5D               pop ebp
69484323     C2 1400          ret 14
69484326     55               push ebp
69484327     8BEC             mov ebp,esp
69484329     53               push ebx
6948432A     56               push esi
6948432B     57               push edi
6948432C     8BFA             mov edi,edx
6948432E     8BD9             mov ebx,ecx
69484330     E8 04EFE7FF      call clr.69303239
69484335     FF75 0C          push dword ptr ss:[ebp+C]
69484338     FF75 08          push dword ptr ss:[ebp+8]
6948433B     8B30             mov esi,dword ptr ds:[eax]
6948433D     57               push edi
6948433E     53               push ebx
6948433F     50               push eax
69484340     8B4E 18          mov ecx,dword ptr ds:[esi+18]
69484343     FF15 F867A369    call dword ptr ds:[<&LogHelp_TerminateOnAssert>]
69484349     FF56 18          call dword ptr ds:[esi+18]
6948434C     5F               pop edi
6948434D     5E               pop esi

34 <clr.&VirtualProtect>]=<kernel32.VirtualProtect>

r.dll:$1A431C #1A371C
```

```
ESP    00B8FB14
ESI    6948771C      clr.6
EDI    000001AC      L'T'

EIP    6948431C      clr.6

EFLAGS    00000344
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

LastError  00000000 (ERR
LastStatus C0000139 (STA

GS 002B   FS 0053
ES 002B   DS 002B
CS 0023   SS 002B

ST(0) 000000000000000000
ST(1) 000000000000000000
ST(2) 000000000000000000
ST(3) 000000000000000000
ST(4) 000000000000000000
ST(5) 400280000000000000
ST(6) 3FFDC0000000000000
```

```
1: [esp] 692EEBF2 clr.69
2: [esp+4] 000001AC
3: [esp+8] 00000040
4: [esp+C] 00B8FB60
```

```
00B8FB14  692EEBF2  clr.692EEBF2
00B8FB1C  00000040
00B8FB20  00B8FB60
00B8FB24  00B8FB4C
00B8FB28  6948434C  return to clr.6948434C
```

```
3 00 C0 8B E7 76 14 00 16 00 38 84 E7 76  ....À.çv....8.çv
2 00 80 5B E7 76 0E 00 10 00 E0 8D E7 76  .....[çv....à.çv
```

Figure 33 - clr.dll using VirtualProtect on its own .text section at runtime in x32dbg

These types of dynamic *+RWX* image regions are rare and tend to stem from very specific modules such as **clr.dll** and **mscorwks.dll** (the legacy version of **clr.dll**, which also creates a *+RWX* enclave in its ***.text*** section). There are however an entire genre of PE (the aforementioned unsigned Windows NGEN assemblies) which contain a *+RWX* section called ***.xdata***. This makes them easy for Moneta to classify as false positives, but also easy for malware and exploits to hide their dynamic code in.

Last Thoughts

With fileless malware becoming ubiquitous in the Red Teaming world, dynamic code is a feature of virtually every single “malware” presently in use. Interestingly, the takeaway concept from this analysis seems to be that attempting to detect such memory is nearly impossible with IOCs alone when the malware writer understands the landscape he is operating in and takes care to camouflage his tradecraft in one of the many existing abnormalities in Windows. Prime among these being some of the false positives discussed previously, such as the OS-enacted DLL hollowing of **User32.dll** in Wow64 processes, or the *+RWX* subregions within CLR image memory. There were far too many such abnormalities to discuss within the scope of this text alone, and the list of existing filters for Moneta remains far from comprehensive.

Moneta provides a useful way for attackers to identify such abnormalities and customize their dynamic code to best leverage them for stealth. Similarly, it provides a valuable way for defenders to identify/dump malware from memory and also to identify the false positives they may be interested in using to fine-tune their own memory detection algorithms.

The remaining content in this series will be aimed at increasing the skill of the reader in the domain of bypassing existing memory scanners by understanding their detection strategies and exploring new stealth tradecraft still undiscussed in this series.