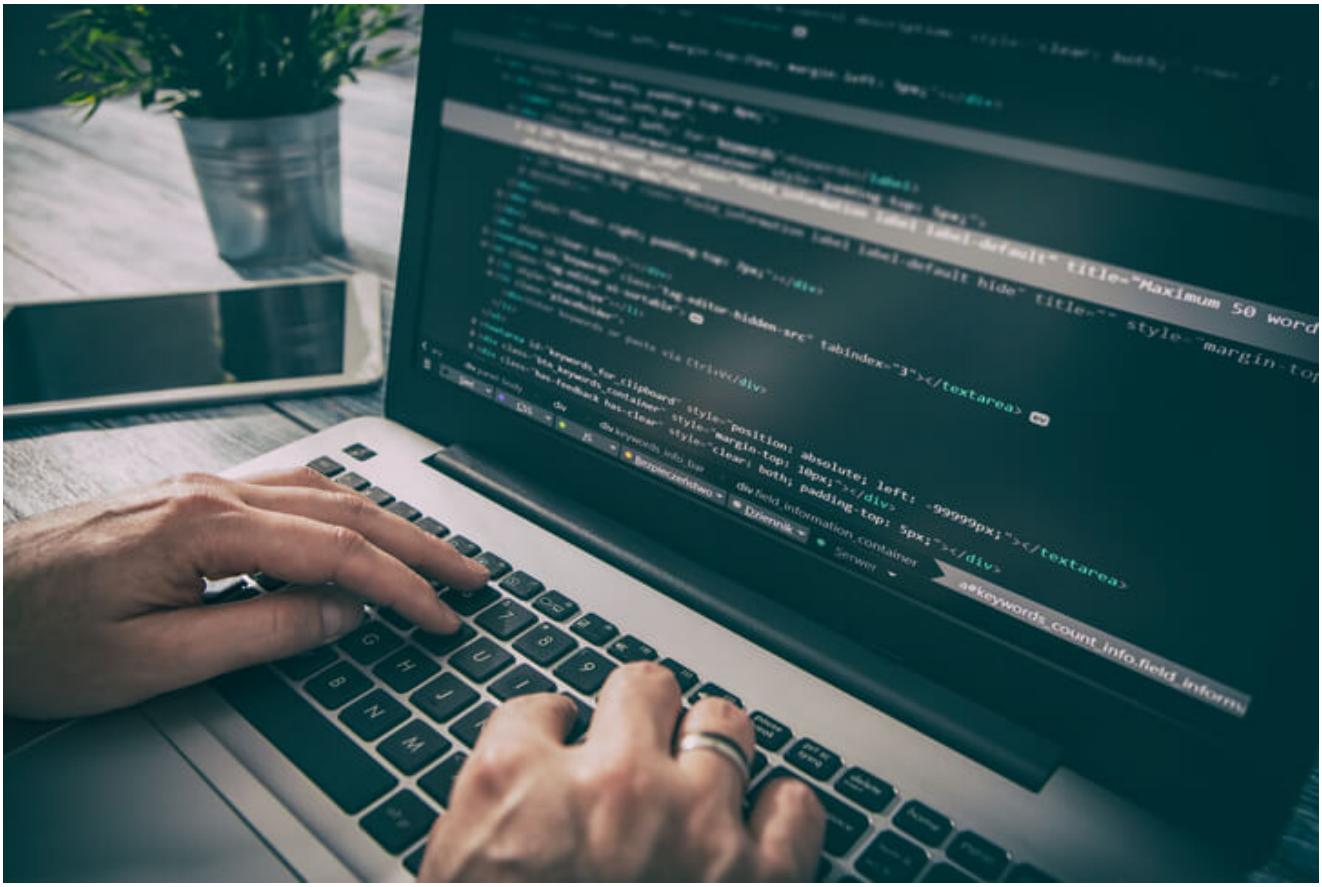
Deep Analysis of New Poison Ivy Variant

blog.fortinet.com/2017/08/23/deep-analysis-of-new-poison-ivy-variant

August 23, 2017



Threat Research

By Xiaopeng Zhang | August 23, 2017

Recently, the FortiGuard Labs research team observed that a new variant of Poison Ivy was being spread through a compromised PowerPoint file. We captured a PowerPoint file named Payment_Advice.ppsx, which is in OOXML format. Once the victim opens this file using the MS PowerPoint program, the malicious code contained in the file is executed. It downloads the Poison Ivy malware onto the victim's computer and then launches it. In this blog, I'll show the details of how this happens, what techniques are used by this malware, as well as what it does to the victim's computer.

The PowerPoint Sample

Figure 1 shows a screenshot of when the ppsx file is opened.

Figure 1. Open Payment_Advice.ppsx

As you can see, the ppsx file is played automatically. The “ppsx” extension stands for “PowerPoint Show,” which opens the file in presentation mode. This allows the malicious code to be executed automatically. The warning message box alerts the user that it might run an unsafe external program. Usually, the implied content of the document beguiles the user into pressing the Enable button.

Let’s take a look at the malicious code embedded inside this PowerPoint file.

OOXML file is a zip format file. By decompressing this file we can see the file/folder structure, shown below.

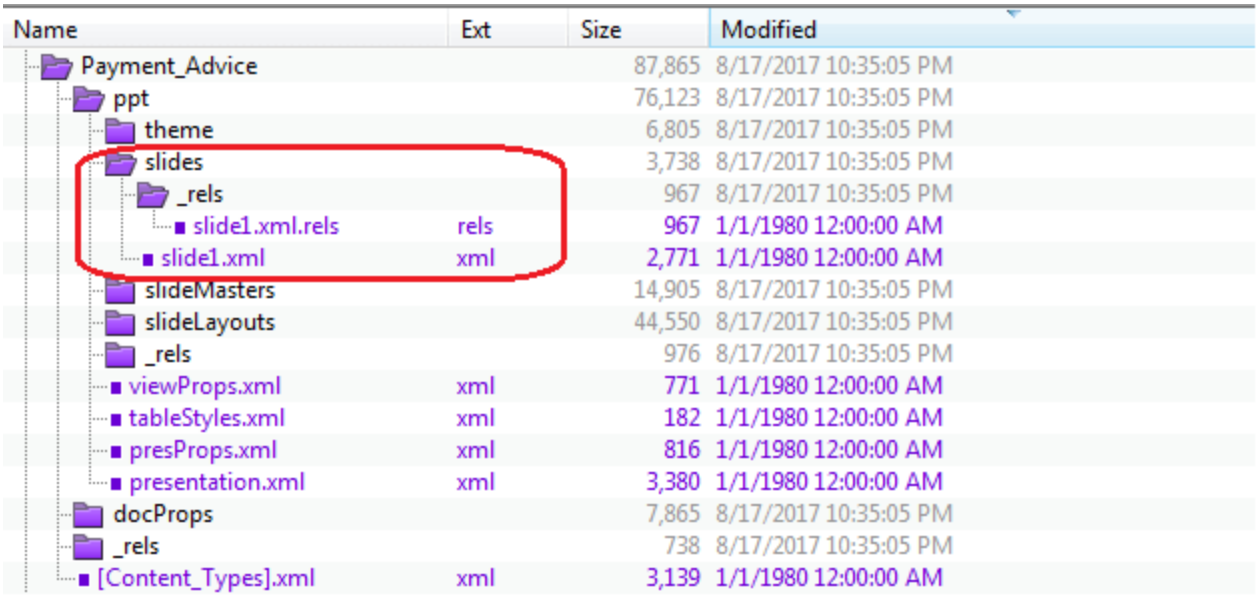
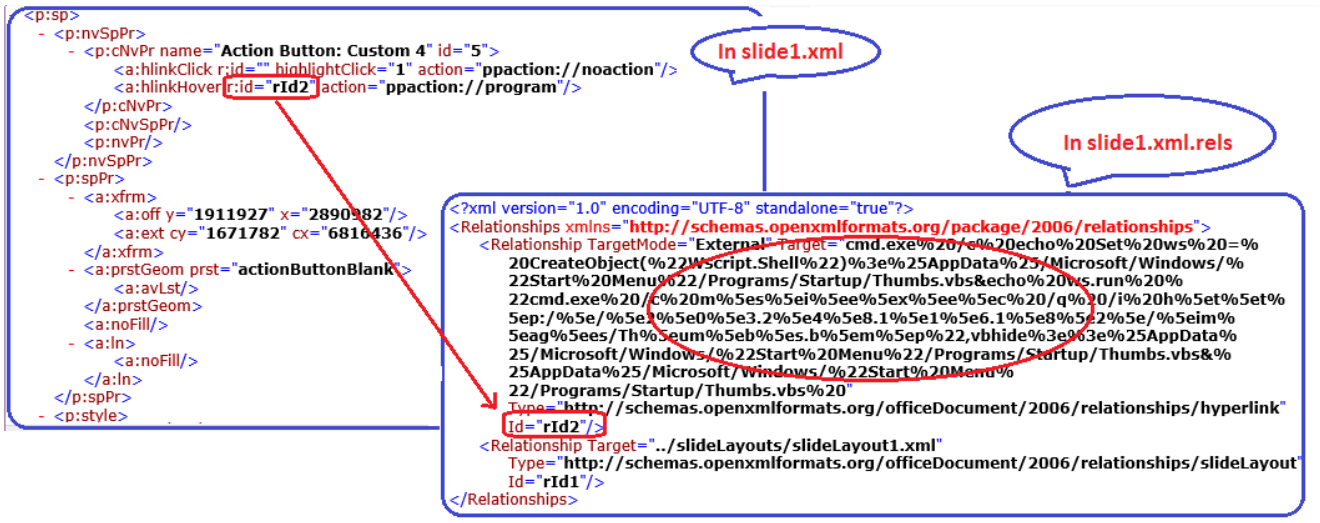


Figure 2. PPSX file structure

Going into its .\ppt\slides\ subfolder, slide1.xml is the slide automatically shown in Figure 1. The file "._rels\slide1.xml.rels" is the relationship file where the resources used in slide1.xml are defined. In slide1.xml, I found the xml code:



.

This means that when the user's mouse hovers over this element, something named "rId2" in slide1.xml.rels file is executed.

Figure 3 shows the relationship between them.

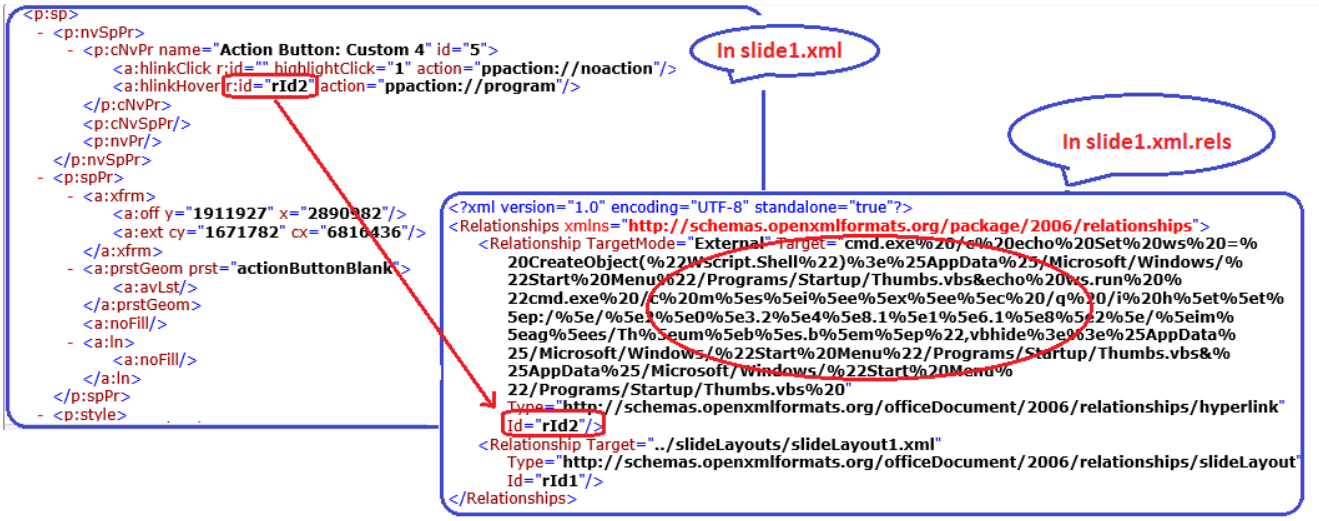


Figure 3. The code defined in “rId2”

Being Added into the Startup Group

The code defined in “rId2” uses an echo command of cmd.exe to output vbs codes into the Thumbs.vbs file in the “Startup” folder of the Start menu. This allows the Thumbs.vbs file to be executed when the victim’s system starts. We’ll take a look at the content of this Thumb.vbs file below.

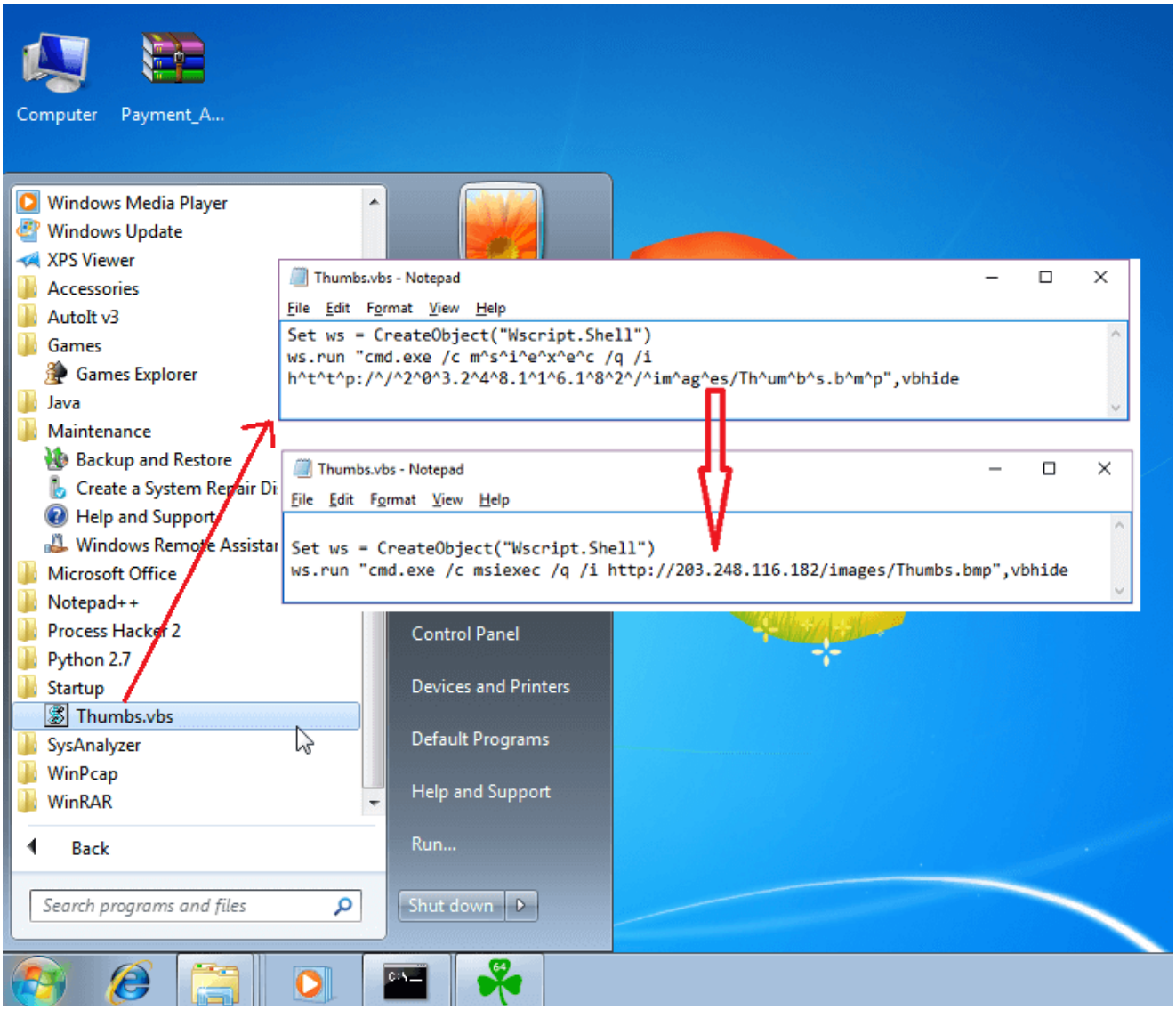


Figure 4. Thumb.vbs in the Startup folder and its content

The Downloaded File

Thumbs.vbs downloads a file from hxxp://203.248.116.182/images/Thumbs.bmp and runs it using msiexec.exe. As you may know, msiexec.exe is the Microsoft Windows Installer program, which is the default handler of .MSI files. Msiexec.exe can be used to install/uninstall/update software on Windows. The MSI file is an Installer Package. It contains a PE file (in a stream) that is executed when it's loaded by msiexec.exe. This PE file could be replaced with malware to bypass any AV software detection. We have also observed that more and more malware authors have started using this method to run their malware. The MSI file is in the Microsoft OLE Compound File format. In Figure 5 we can see the downloaded Thumbs.bmp file content in the DocFile Viewer.

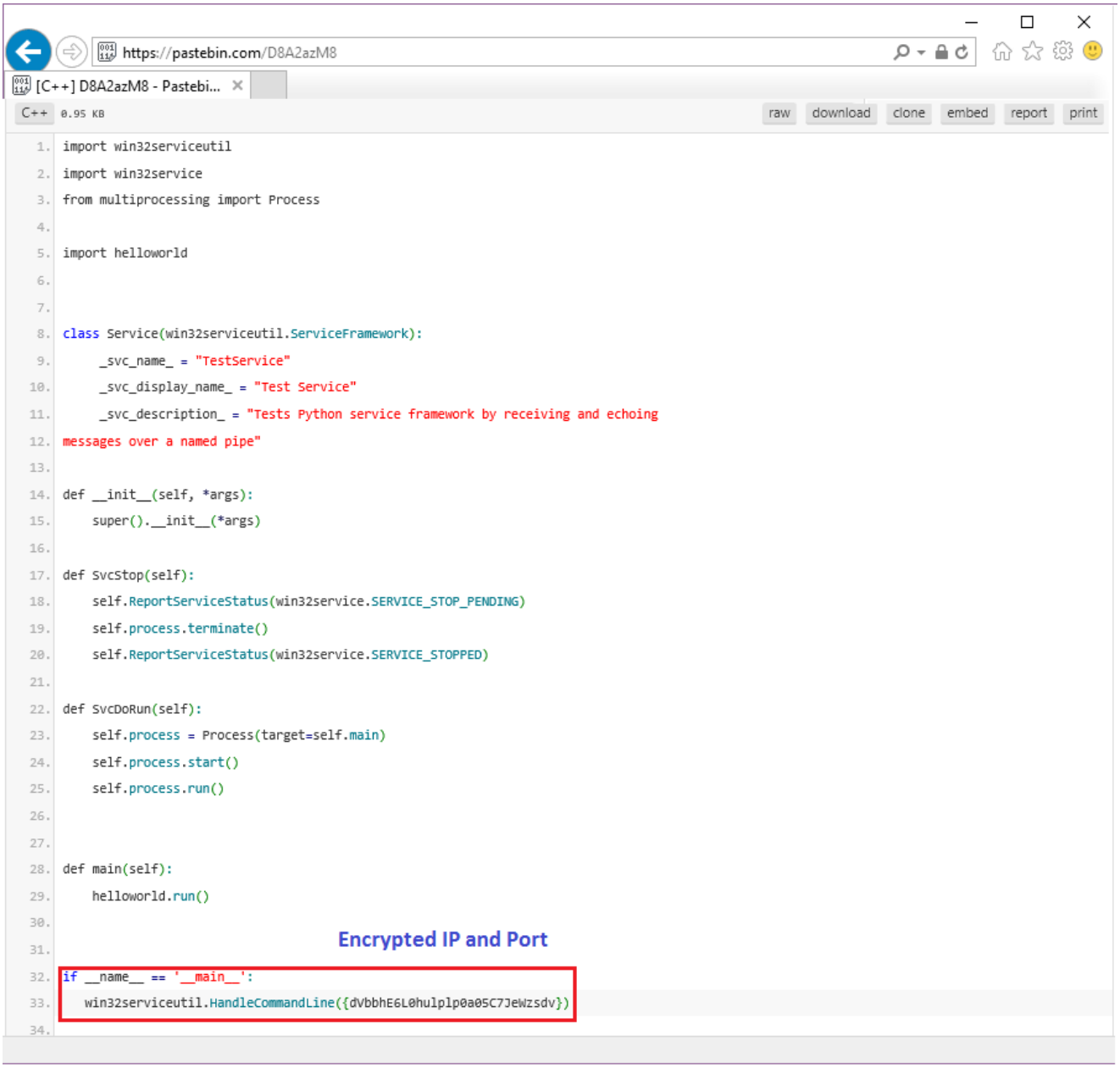


Figure 5. The downloaded Thumb.bmp in DocFile viewer

Next, I'm going to extract this PE file from the stream into a file (exported_thumbs). By checking with a PE analysis tool, we can see that it's a 64-bit .Net program. This means that this malware only afftects 64bit Windows.

Analyzing the .Net code and Running It

After putting this extracted file into dnSpy to be analyzed, we can see the entry function *Main()*, as shown in Figure 6.

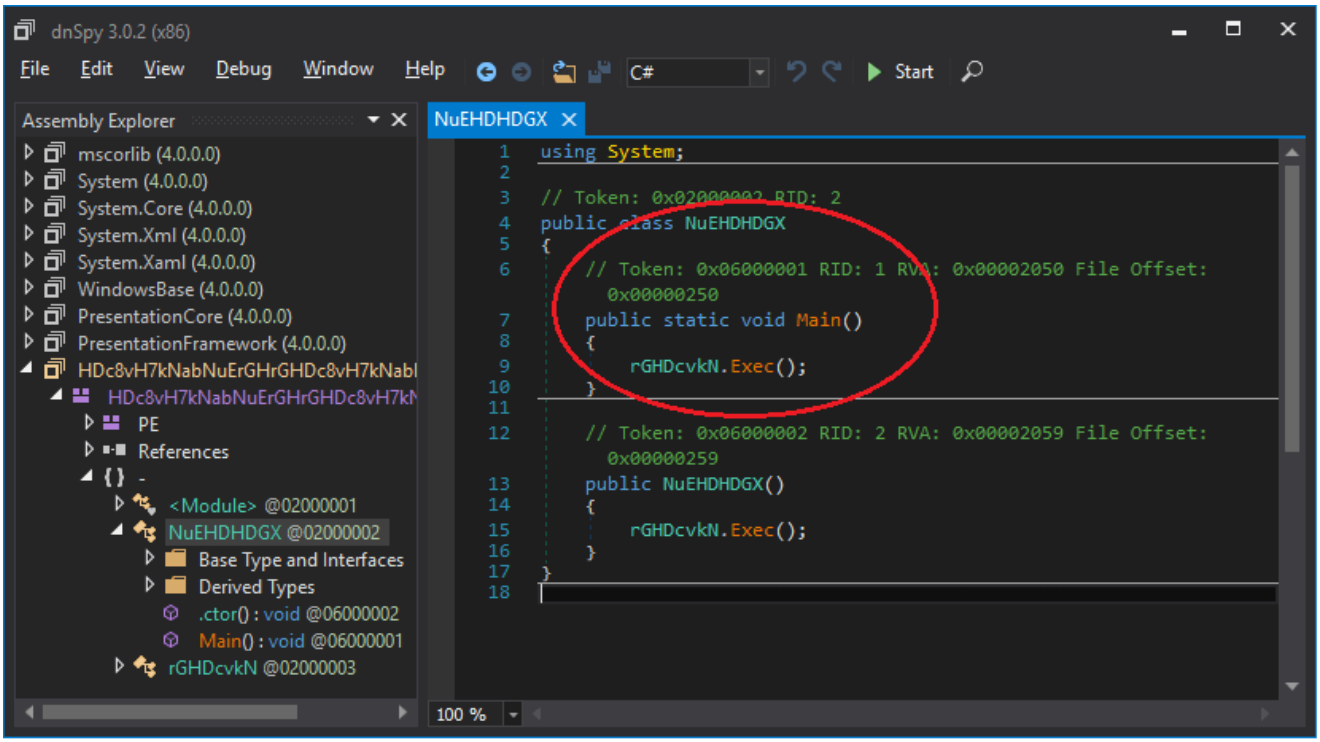


Figure 6. Main function

It then calls the *rGHDcvkN.Exec()* function in Main(), which contains a huge array. Actually, the data in the array is the code that is executed as a thread function by a newly-created thread.

Figure 7 clearly shows how the code in the array is executed.

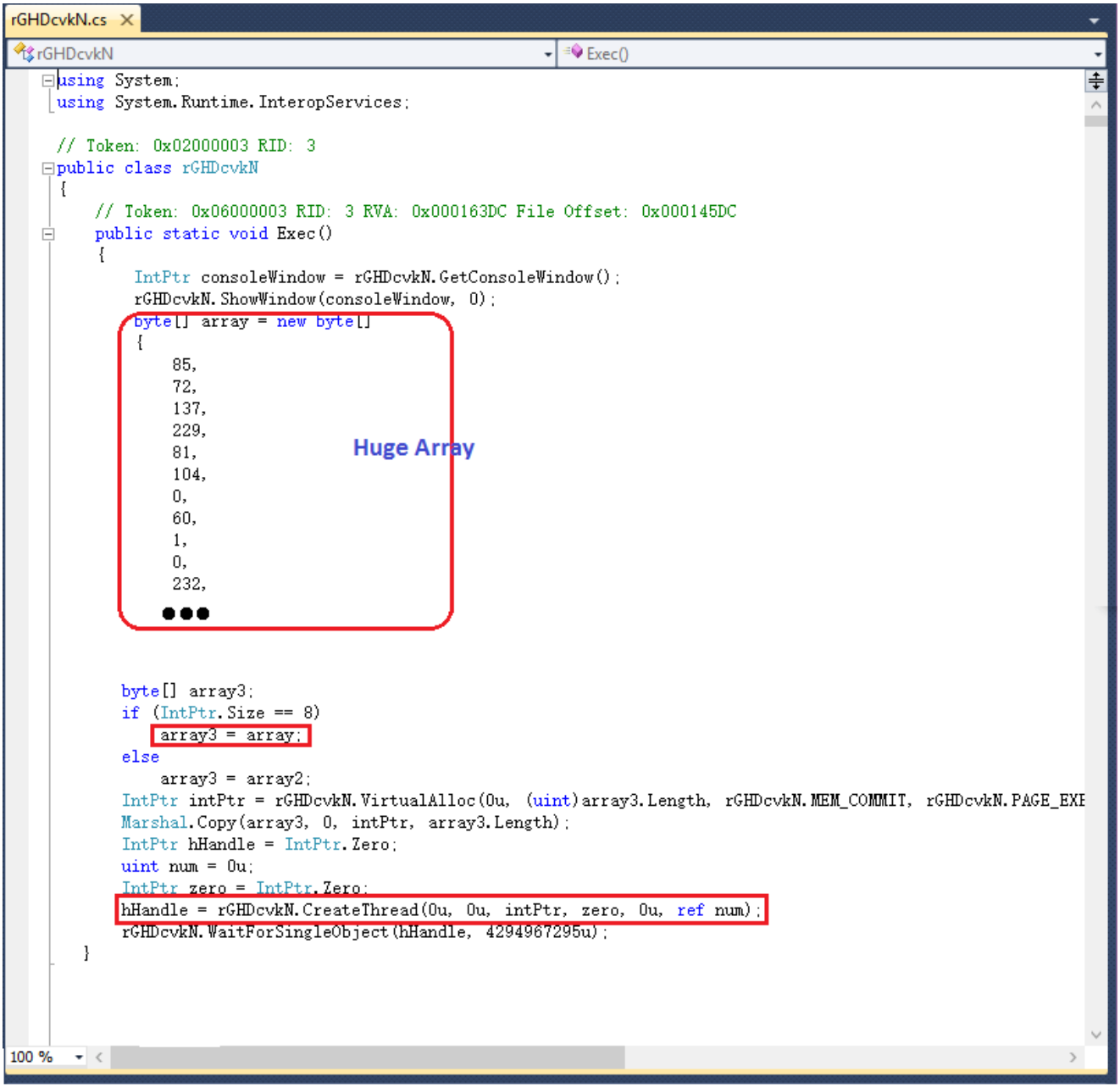


Figure 7. .Net program runs a thread to execute the code in a huge array

If the code is run on a 64-bit platform, IntPtr.Size is 8. So the huge array is passed to array3. It then allocates memory buffer by calling *rGHDcvkN.VirtualAlloc()* and copies the code from array3 into the new memory by calling *Marshal.Copy()*. It eventually calls *rGHDcvkN.CreateThread()* to run the code up.

I started the .Net program in the debugger, and set a breakpoint on CreateThread API to see what the array code would do when it's hit. Per my analysis of the array code, it is a kind of loader. Its main purpose is to dynamically load the main part of the malware code from the memory space into a newly-allocated memory buffer. It then repairs any relocation issues according to the new base address and repairs APIs' offset for the main part code. Finally, the main code's entry function is called.

Anti-Analysis Techniques

1. All APIs are hidden. They are restored when being called. The snippet below is the hidden CreateRemoteThread call.

```
sub_1B0E6122 proc near
   mov   rax, 0FFFFFFFF88E23B10h
   neg   rax
   jmp   rax  ;; CreateRemoteThread
sub_1B0E6122 endp
```

1. All strings are encrypted. They are decrypted before using. For example, this is the encrypted “ntdll” string.

```
unk_1AFD538C  db 54h, 0B2h, 9Bh, 0F1h, 47h, 0Ch  ; ==> "ntdll"
```

1. It runs a thread (I named it ThreadFun6) to check if the API has been set as a breakpoint. If yes, it calls TerminateProcess in another thread to exit the process immediately. The thread function checks all APIs in the following modules: “ntdll”, “kernel32”, “kernelbase” and “user32”. In Figure 8, you can see how this works:

```
●●●
mov     [rsp-8+arg_8], rbx
push    rbp
mov     rbp, rsp
sub     rsp, 50h
lea     rdx, unk_1AFD538C ; ==> "ntdll"
lea     rcx, [rbp+var_20]
mov     r8d, 724D023Ah
call    Decrypt_String_fun
mov     rcx, rax
call    sub_1AFD1000
mov     rcx, rax
call    cs:call_GetModuleHandleA ; GetModuleHandleA
lea     r8, ThreadFun6  ;; To check if the exported API of ntdll is set breakpoint.
xor     edx, edx
mov     r9, rax
lea     rax, [rbp+arg_0]
xor     ecx, ecx
mov     [rsp+50h+var_28], rax
and     [rsp+50h+var_30], 0
call    cs:call_CreateThread ; ;;;ThreadFun6
mov     rcx, rax
call    cs:call_CloseHandle ; CloseHandle
lea     rcx, [rbp+var_20]
test    eax, eax
●●●
```

```
ThreadFun6      proc near               ; DATA XREF: sub_1AFD2264+34↑o
                                        ; sub_1AFD2264+A3↑o ...

var_28          = dword ptr -28h
var_20          = qword ptr -20h
arg_0           = byte ptr  8

                mov     r11, rsp
                mov     [r11+10h], rbx
                mov     [r11+18h], rbp
                push    rsi
                push    rdi
                push    r12
                sub     rsp, 30h
                mov     rbx, rcx
                test    rcx, rcx        ; ;;;;; module base address
                jnz     short loc_1AFD2452
                mov     rbx, [r11+10h]
                mov     rbp, [r11+18h]
```

```
                xor     eax, eax
                add     rsp, 30h
                pop     r12
                pop     rdi
                pop     rsi
                retn
; ---------------------------------------------------------------------------

loc_1AFD2452:                           ; CODE XREF: ThreadFun6+19↑j
                movsxd  rax, dword ptr [rcx+3Ch]
                xor     edi, edi
                mov     ebp, [rax+rcx+88h]
                mov     r12d, [rbp+rcx+1Ch]
                add     r12, rcx
                xor     esi, esi

loc_1AFD2469:                           ; CODE XREF: ThreadFun6+A5↓j
                xor     ecx, ecx
                cmp     [rbp+rbx+14h], ecx
                jle     short loc_1AFD248C
                mov     r8d, [rbp+rbx+14h]
                mov     rdx, r12

loc_1AFD2479:                           ; CODE XREF: ThreadFun6+66↓j
                mov     eax, [rdx]
                cmp     byte ptr [rax+rbx], 0CCh ;0CCH is binary of instruction int3. ie. soft breakpoint.
                jnz     short loc_1AFD2483
                inc     ecx

loc_1AFD2483:                           ; CODE XREF: ThreadFun6+5B↑j
                add     rdx, 4
                dec     r8
                jnz     short loc_1AFD2479 ; ;;go through all export APIs in this module.

loc_1AFD248C:                           ; CODE XREF: ThreadFun6+4B↑j
                test    esi, esi
                jnz     short loc_1AFD2497
                mov     edi, ecx
                mov     esi, 1

loc_1AFD2497:                           ; CODE XREF: ThreadFun6+6A↑j
                cmp     edi, ecx
                jz      short loc_1AFD24BE
                mov     r8, cs:call_RtlExitUserProcess;Exit process when soft breakpoint detected on APIs.
                lea     rax, [rsp+48h+arg_0]
                xor     r9d, r9d
                mov     [rsp+48h+var_20], rax
                and     [rsp+48h+var_28], 0
                xor     edx, edx
                xor     ecx, ecx        ; ;;exit if error
                call    cs:call_CreateThread ; CreateThread

loc_1AFD24BE:                           ; CODE XREF: ThreadFun6+75↑j
                mov     ecx, 3E8h
                call    cs:call_Sleep   ; Sleep
                jmp     short loc_1AFD2469
ThreadFun6      endp
```

Figure 8. Checking for breakpoints on exported APIs in "ntdll"

1. It runs a thread to check if any analysis tools are running. It does this by creating specially named pipes that are created by some analysis tools. For example, "\\.\Regmon" for registry monitor tool RegMon; "\\.\FileMon" for local file monitor tool FileMon; "\\.\NTICE" for SoftIce, so on.

If one of the named pipes cannot be created, it means one of the analysis tools is running. It then exits process soon thereafter.

1. It then goes through all the running program windows to check if any windows class name contains a special string to determine if an analysis tool is running. For example, “WinDbgFrameClass” is Windbg main window’s class name. This check runs in a thread as well (I named it as Threadfun3). Below, Figure 9 shows how this thread function works.

```
ThreadFun3      proc near               ; DATA XREF: main+1E5↑o
                sub     rsp, 28h

loc_1AFD223C:                           ; CODE XREF: ThreadFun3+28↓j
                call    cs:call_GetForegroundWindow ; GetForegroundWindow
                lea     rdx, sub_1AFD2064 ; ;;;;Function to check windows class name. for exmaple windbg
                mov     r8, rax
                mov     rcx, rax
                call    cs:call_EnumChildWindows ; EnumChildWindows
                mov     ecx, 3E8h
                call    cs:call_Sleep   ; Sleep
                jmp     short loc_1AFD223C
ThreadFun3      endp
```

Figure 9. Check Windows’ Class Name

1. By checking to see if the “Wireshark-is-running-{…}” named mutex object exists (by calling OpenMutex), it could implement anti-WireShark.
2. By calling the API “IsDebuggerPresent”, it can check to see] if this process is running in a debugger (returns with 1). It’s a kind of anti-debugging check. It also checks how much time is spent by calling IsDebuggerPresent. If the time is more than 1000ms, it means that the process runs in a debugger or VM, and it then exits the process.

These are all the ways that this malware performs anti-analysis. Most of these checks run in their own threads, and are called every second. It then exits the process if any check is matched.

To continue the analysis of this malware, we have to first skip these checks. We can dynamically modify its code to do so. For example, changing “IsDebuggerPresent”’s return value as 0 allows us to bypass the running-in-debugger detection.

Generating A Magic String from a Decrypted String

By decrypting three strings and putting them together, we get the magic string "Poison Ivy C++", which will be saved in a global variable qword_1B0E4A10. From the code snippet below you can see how it makes this string.

```
                        ; CODE XREF: main+4AC↑j
lea     rdx, unk_1AFD5268 ; ;;;"Poison "
lea     rcx, [rbp+var_20]
mov     r8d, 0C95F4308h
call    Decrypt_String_fun
mov     rcx, rax
call    sub_1AFD1000    ; ;WideCharToMultiByte
mov     rcx, cs:qword_1B0E4A10
mov     rdx, rax
call    cs:call_lstrcat ; lstrcat
lea     rcx, [rbp+var_20]
call    sub_1AFD4C54    ; ;;calling HeapFree
lea     rdx, unk_1AFD5274 ; ;;; "Ivy "
lea     rcx, [rbp+var_20]
mov     r8d, 0F70B83DDh
call    Decrypt_String_fun
mov     rcx, rax
call    sub_1AFD1000    ; ;WideCharToMultiByte
mov     rcx, cs:qword_1B0E4A10
mov     rdx, rax
call    cs:call_lstrcat ; lstrcat
lea     rcx, [rbp+var_20]
call    sub_1AFD4C54    ; ;;calling HeapFree
lea     rdx, unk_1AFD527C ; ;; "C++"
lea     rcx, [rbp+var_20]
mov     r8d, 21E0ED40h
call    Decrypt_String_fun
mov     rcx, rax
call    sub_1AFD1000    ; ;WideCharToMultiByte
mov     rcx, cs:qword_1B0E4A10 ; ;;;; "Poison Ivy C++"
mov     rdx, rax
call    cs:call_lstrcat ; lstrcat
lea     rcx, [rbp+var_20]
call    sub_1AFD4C54    ; ;;calling HeapFree
```

Figure 10. Generating the magic string

Hiding Key-functions in Six Different Modules

It next loads several modules from its encrypted data. It creates a doubly-linked list, which is used to save and manage these loaded modules. There are many export functions from each of these modules that achieve the malware's main work. In this way, it's also a challenge for dynamic debugging. The variable qword_1AFE45D0 saves the header of that doubly-linked list. Each object in the list has the structure below:

```
+00H  pointer to previous object in the list
+08H  pointer to next object in the list
+18H  for Critical Section object use
+28H  the base address of the module this object is related to
+30H  pointer to export function table
```

It then decrypts and decompresses six modules one by one, and adds each of them into the doubly-linked list. Figure 11 shows a code snippet from decrypting these six modules.

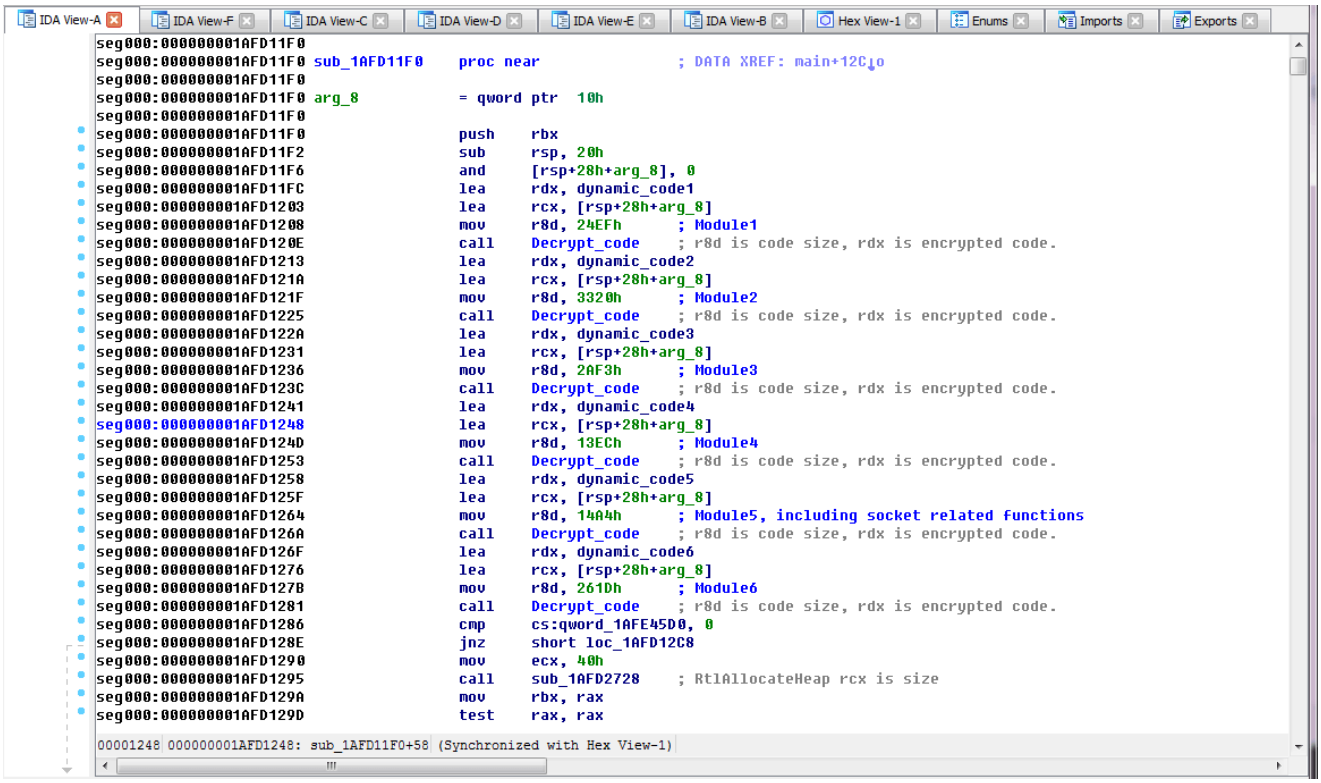


Figure 11. Decrypting and decompressing modules

Each module has an Initialization function (like DllMain function for Dll files) that is called once the module is completely decrypted and decompressed. Three of these modules have an anti-analysis ability similar to the one I described in the Anti-Analysis section above. So to continue the analysis of this malware, I needed to modify their codes to bypass their detection function.

After that it calls the export functions of those modules. It decrypts the configuration data from the buffer at unk_1AFE3DA0. This configuration data is decrypted many times during the process running, and it tells the malware how to work. I'll talk more about the configuration data in a later section.

The malware then picks a string from the configuration data, which is "%windir%\system32\svchost.exe". It later calls CreatProcess to run svchost.exe, and then injects some code and data from malware memory into the newly-created svchost.exe. It finally calls the injected code and exits its current process. The malware's further work is now done in the svchost.exe side.

Starting over in SVCHOST.exe

Through my analysis I could see that the injected codes and data represent the entire malware. It all starts over again in the svchost.exe process. Everything I have reviewed about is repeated in svchost.exe. For example, executing the anti-analysis detection code,

getting the magic string, creating a doubly-linked list, decrypting six modules and adding them into the doubly-linked list, and so on.

It then goes to different code branch when executing the instruction 01736C2 cmp dword ptr [rdi+0Ch], 1 in module2. [rdi+0ch] is a flag that was passed when the entire code was initialized. When the flag is 0, it takes the code branch to run svchost.exe and inject code in it; when it's 1, it takes the code branch to connect to the C&C server. Before the injected code in svchost.exe is executed, the flag is set to 1. Figure 12 shows the code branches.

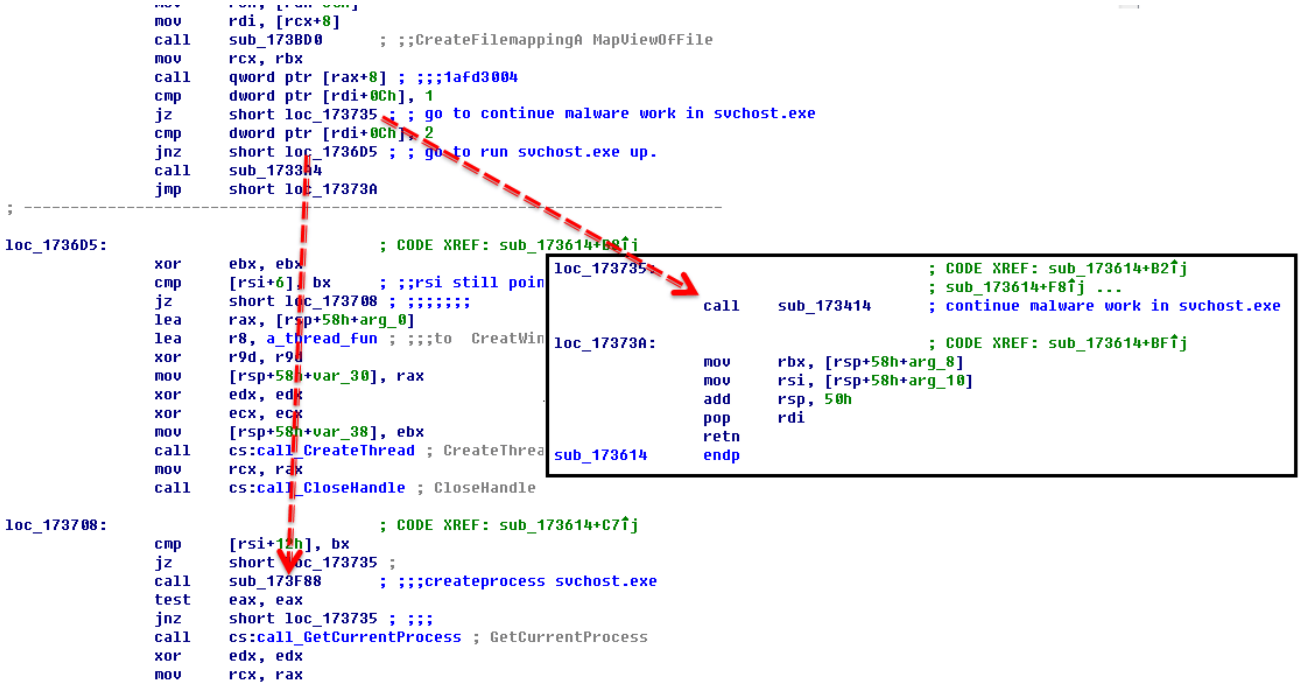


Figure 12. Snippet of code branches

Obtaining the C&C Server from PasteBin

The C&C server's IP addresses and ports are encrypted and saved on the PasteBin website. PasteBin is a text code sharing website. A registered user can paste text code on it in order to share the text content to everyone. The malware author created 4 such pages, and put the C&C server IP addresses and ports there. Do you remember when I talked previously about encrypted configuration data? It contains the 4 PasteBin URLs. They are

```
hxxps://pastebin.com/Xhpmhhuy
hxxps://pastebin.com/m3TPwxQs
hxxps://pastebin.com/D8A2azM8
hxxps://pastebin.com/KQAxvdvJ
```

Figure 13 shows the decrypted configuration data.

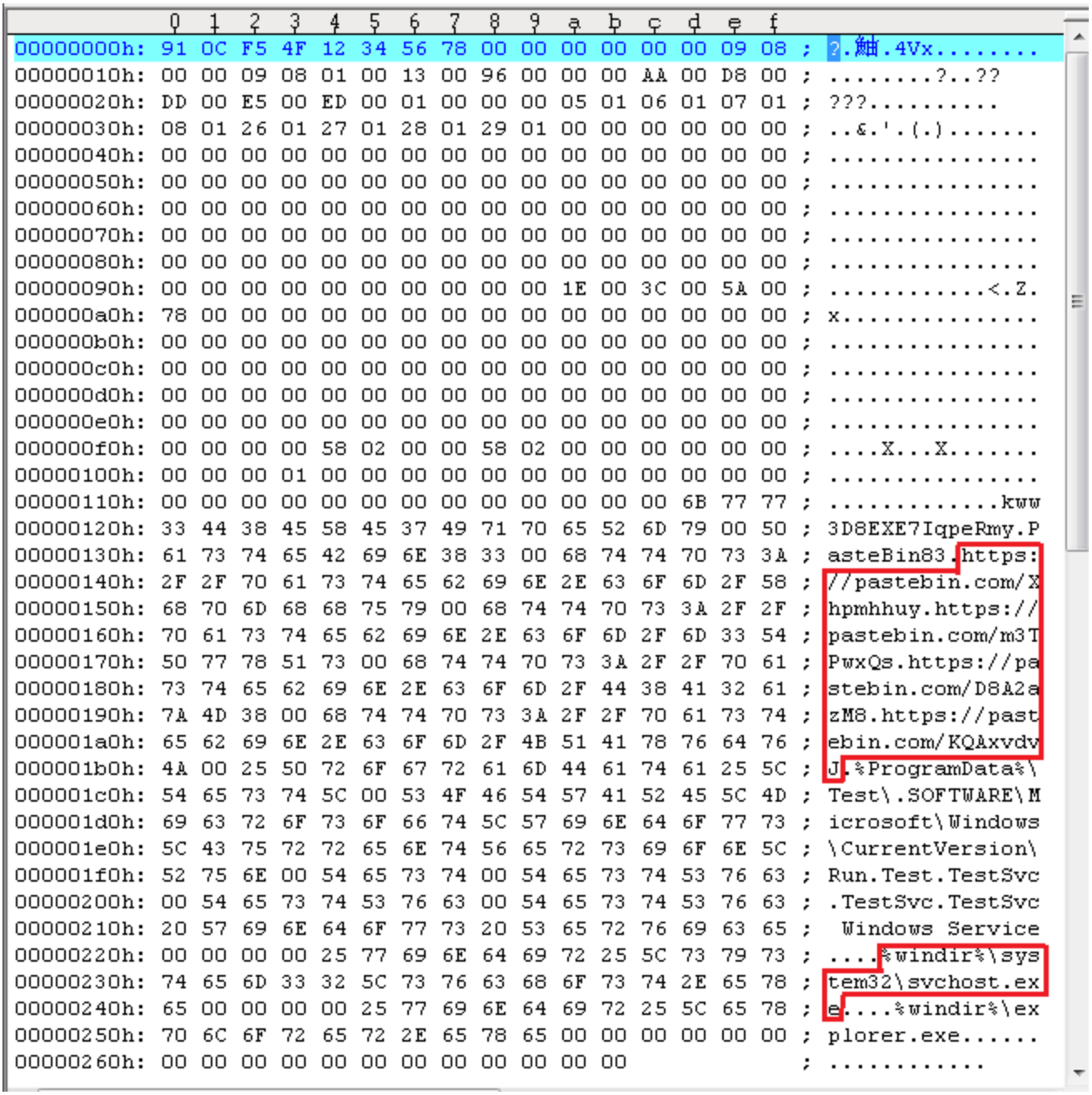


Figure 13. Decrypted configuration data

If you access any one of these URLs, you will find there are normal Python codes on it. The encrypted server IP address and port are hidden in the normal python code. Let's take a look.

While looking at the main function you will find the code below:

win32serviceutil.HandleCommandLine({65YbRI+gEtvlZpo0qw6CrNdWDoev}), the data between "{" and "}", is the encrypted IP address and port. See Figure 14 for more information.

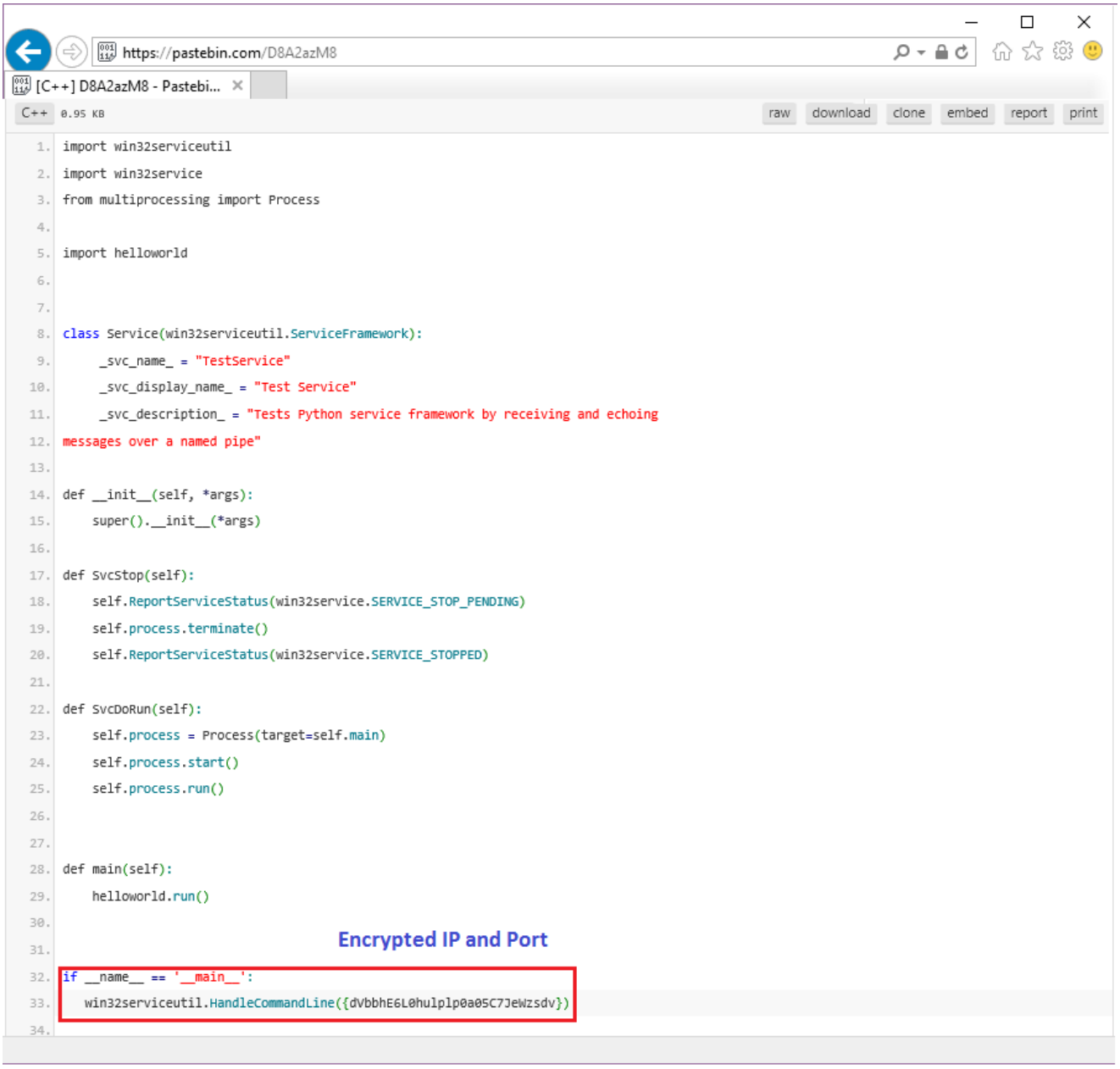


Figure 14. Encrypted C&C IP address and Port on PasteBin

Let's see what we can see after decryption in Figure 15.

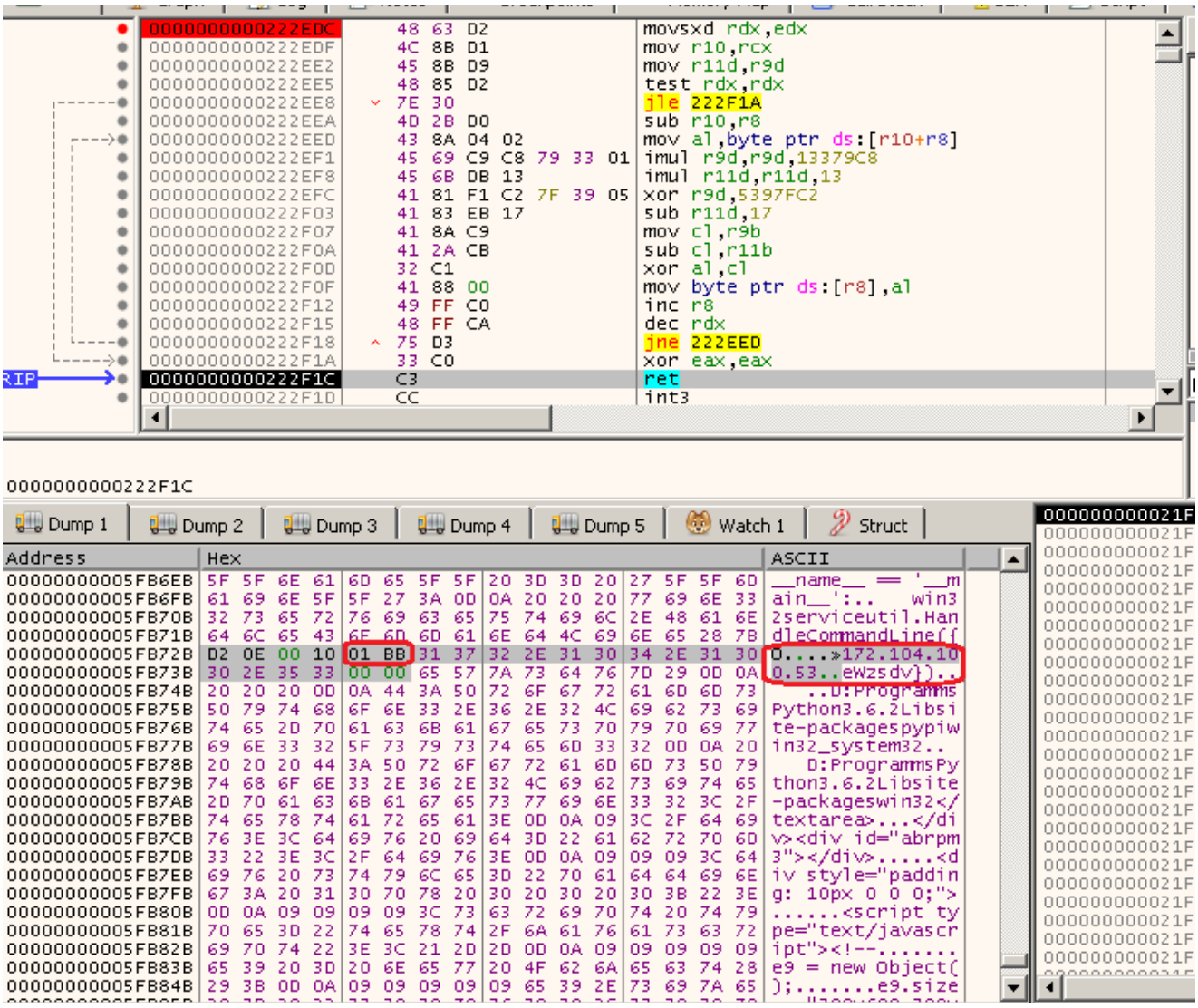


Figure 15. Decrypted IP address and Port

From Figure 15, we can determine that the decrypted C&C server IP address is 172.104.100.53 and the Port is 1BBH i.e. 443. It should be noted that the IP addresses and Ports on the four pages are not the same. The author of this malware can update these IP addresses and Ports by simply updating the python codes on the four PasteBin pages.

Communicating with the C&C server

The malware starts connecting and sending data to its C&C server once it gets the IP address and Port. All the packets traveling between the malware and its server are encrypted using a private algorithm. The structure of the packet is like this: (the first 14H bytes is the header part, from 14H on is the data part)

```
+00 4 bytes are a key for encryption or decryption.
+04 4 byte, are the packet command.
+0c 4 bytes is the length in bytes of the data portion of the packet.
+14 4 bytes. From this point on is the real data.
```

Once the malware has connected to the server, it first sends a “30001” command, and the server replies with command “30003”. The command “30003” requests the client to collect the victim’s system information. Once the malware receives this command, it calls tons of APIs to collect the system information.

- It gathers the system's current usage of both physical and virtual memory by calling GlobalmemoryStatusEx.
- It gets the CPU speed from the system registry from “HKLM\HARDWARE\DESCRIPTION\SYSTEM\CENTRALPROCESSOR\0\~MHz".
- It gets the free disk space of all partitions by calling GetDiskFreeSpaceExA.
- It gets the CPU architecture by calling GetNativeSysstemInfo.
- It collects display settings by calling EnumDisplaySetting.
- It collects file information from kernel32.dll.
- It gets the current computer name and user name by calling GetComputerName and GetUserName.
- It also gets the System time by calling GetSystemTime, and the system version by calling GetVersionEx.
- Finally, it copies the svchost.exe’s full path and a constant string, “PasteBin83”, which is from the decrypted configuration data (see Figure 13 again).

In Figure 16 you can see the collected system information before encryption. Figure 17 shows the data after encryption as it’s about to be sent to the C&C server. The first four bytes are used to encrypt or decrypt the following data.

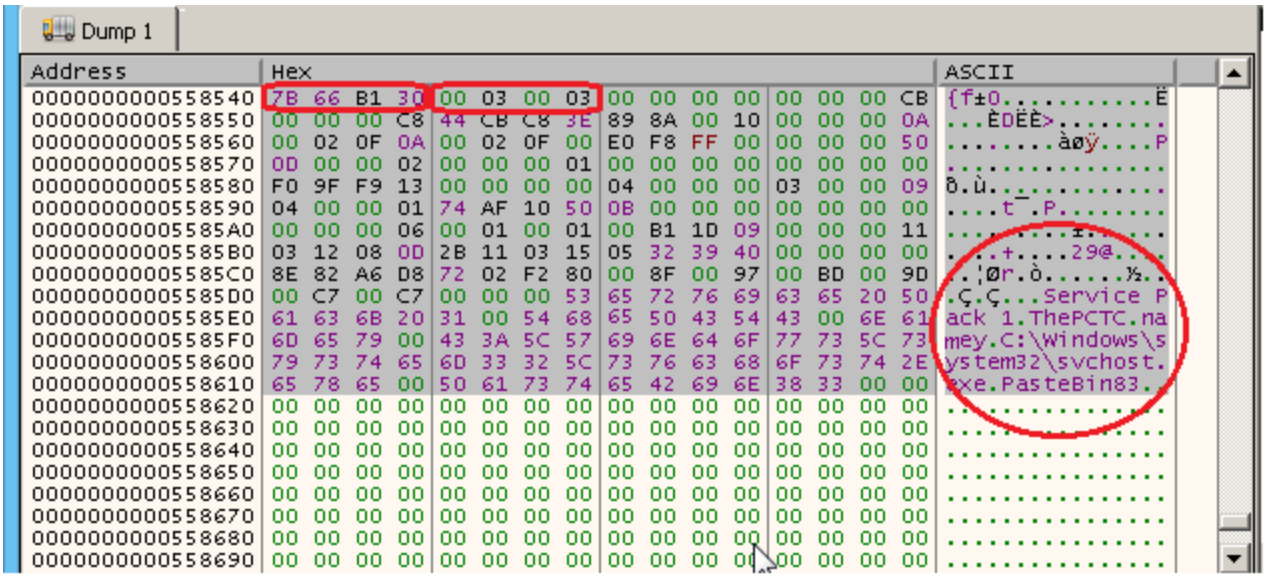


Figure 16. Collected information from the victim’s system

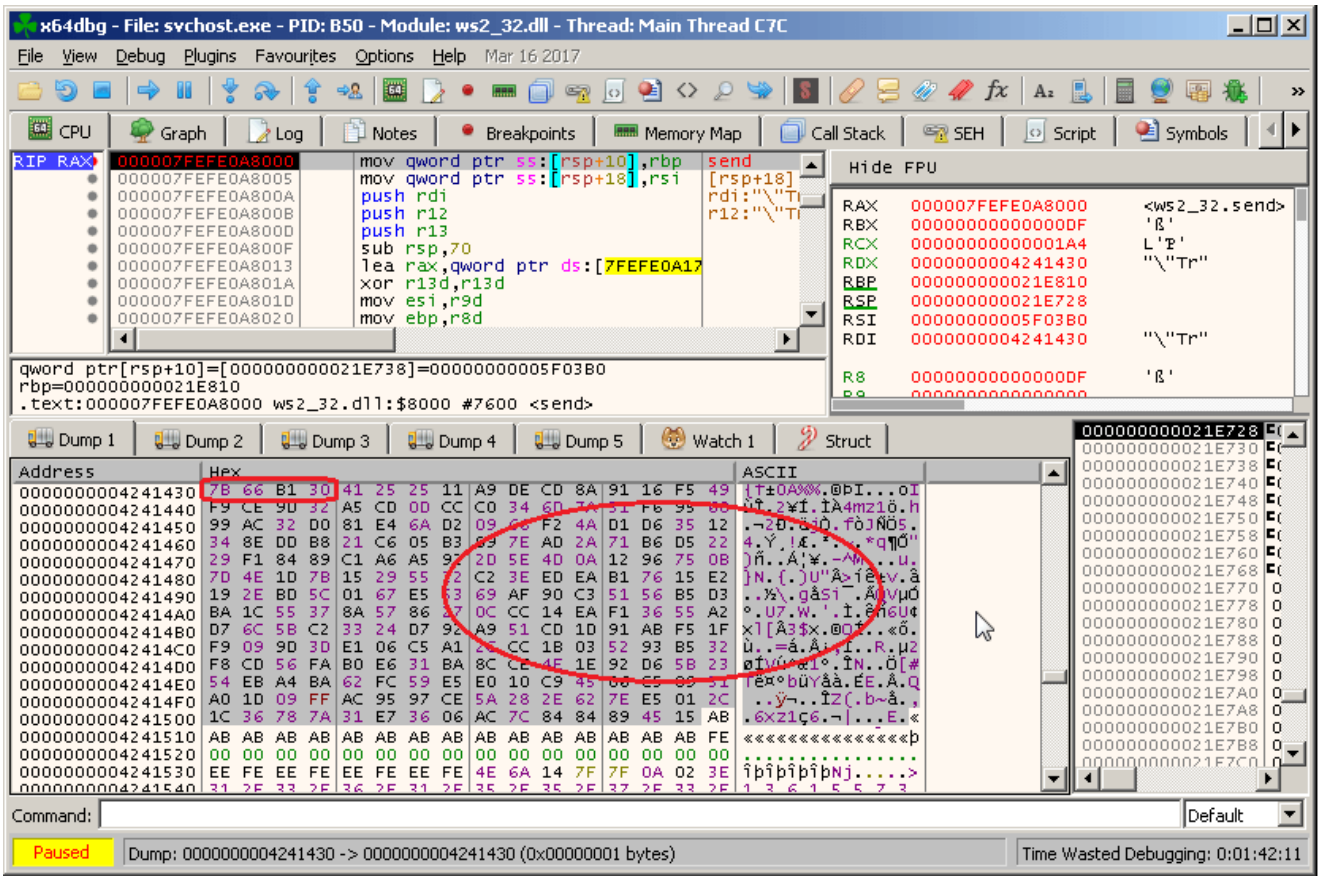


Figure 17. Encrypted system information from victim's system

From my analysis during the malware runtime, I could determine that the malware keeps obtaining the C&C server's IP address from PasteBin and communicating with the C&C server in an infinite loop (by calling Sleep(1000) to suspend the execution).

So far, I only saw that the commands "030001" and "030003" are used. I'll continue to monitor and analyze the malware's behavior to see what else it will do.

Solution

The FortiGuard Antivirus service has detected the files "Payment_Advice.ppsx" as MSOFFICE/PoisonIvy.A!tr.dldr and "Thumbs.bmp" as MSOFFICE/PoisonIvy.A!tr.

IOC

URL:

hxxp://203.248.116.182/images/Thumbs.bmp

Sample SHA-256 hashes:

Payment_Advice.ppsx

E7931270A89035125E6E6655C04FEE00798C4C2D15846947E41DF6BBA36C75AE

Thumbs.bmp

A3E8ECF21D2A8046D385160CA7E291390E3C962A7107B06D338C357002D2C2D9

Sign up for weekly Fortinet FortiGuard Labs Threat Intelligence Briefs and stay on top of the newest emerging threats.

Related Posts