Malware Analysis – PlugX

countuponsecurity.com/2018/02/04/malware-analysis-plugx/

By Luis Rocha February 4, 2018

[The PlugX malware family has always intrigued me. I was curious to look at one variant. Going over the Internet and the research articles and blogs about it I came across the research made by Fabien Perigaud. From here I got an old PlugX builder. Then I set a lab that allowed me to get insight about how an attacker would operate a PlugX campaign. In this post, I will cover a brief overview about the PlugX builder, analyze and debug the malware installation and do a quick look at the C2 traffic. ~LR]

PlugX is commonly used by different threat groups on targeted attacks. PlugX is also refered as KORPLUG, SOGU, DestroyRAT and is a modular backdoor that is designed to rely on the execution of signed and legitimated executables to load malicious code. PlugX, normally has three main components, a DLL, an encrypted binary file and a legitimate and signed executable that is used to load the malware using a technique known as DLL search order hijacking. But let's start with a quick overview about the builder.

The patched builder, MD5 6aad032a084de893b0e8184c17f0376a, is an English version, from Q3 2013, of the featured-rich and modular command & control interface for PlugX that allows an operator to:

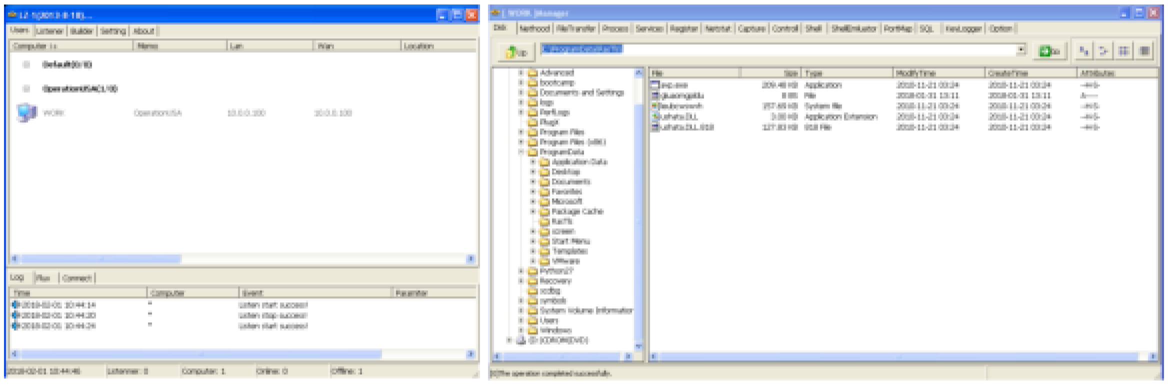
- Build payloads, set campaigns and define the preferred method for the compromised hosts to check-in and communicate with the controller.
- Proxy connections and build a tiered C2 communication model.
- Define persistence mechanisms and its attributes.
- Set the process(s) to be injected with the payload.
- Define a schedule for the C2 call backs.
- Enable keylogging and screen capture.
- Manage compromises systems per campaign.

Then for each compromised system, the operator has extensive capabilities to interact with the systems over the controller that includes the following modules:

- Disk module allows the operator to write, read, upload, download and execute files.

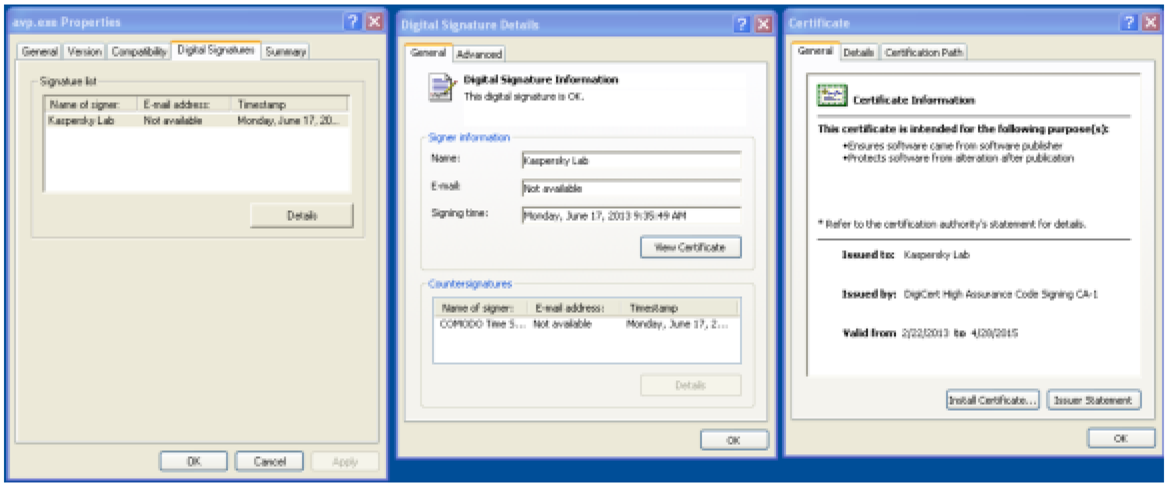
- Networking browser module allows the operator to browse network connections and connect to another system via SMB.
- Process module to enumerate, kill and list loaded modules per process.
- Services module allows the operator to enumerate, start, stop and changing booting properties
- Registry module allows the operator to browse the registry and create, delete or modify keys.
- Netstat module allows the operator to enumerate TCP and UDP network connections and the associated processes
- Capture module allows the operator to perform screen captures
- Control plugin allows the operator to view or remote control the compromised system in a similar way like VNC.
- Shell module allows the operator to get a command line shell on the compromised system.
- PortMap module allows the operator to establish port forwarding rules.
- SQL module allows the operator to connect to SQL servers and execute SQL statements.
- Option module allows the operator to shut down, reboot, lock, log-off or send message boxes.
- Keylogger module captures keystrokes per process including window titles.

The picture below shows the Plug-X C2 interface.



So, with this we used the builder functionality to define the different settings specifying C2 comms password, campaign, mutex, IP addresses, installation properties, injected binaries, schedule for call-back, etc. Then we build our payload. The PlugX binary produced by this version of the builder (LZ 2013-8-18) is a self-extracting RAR archive that contains three files. This is sometimes referred in the literature as the PlugX trinity payload. Executing the self-extracting RAR archive will drop the three files to the directory chosen during the process. In this case “%AUTO%/RasTls”. The files are: A legitimate signed executable from Kaspersky AV solution named “avp.exe”, MD5 e26d04cecd6c7c71cfbb3f335875bc31, which is susceptible to DLL search order hijacking . The file “avp.exe” when executed will load the second file: “ushata.dll”, MD5 728fe666b673c781f5a018490a7a412a, which in this case is a

DLL crafted by the PlugX builder which on is turn will load the third file. The third file: “ushata.DLL.818”, MD5 “21078990300b4cdb6149dbd95dff146f” contains obfuscated and packed shellcode.



Signature list

Name of signer:	E-mail address:	Timestamp
Kaspersky Lab	Not available	Monday, June 17, 20...

Digital Signature Information
This digital signature is OK.

Signer information
Name: Kaspersky Lab
E-mail: Not available
Signing time: Monday, June 17, 2013 9:35:49 AM

Countersignatures

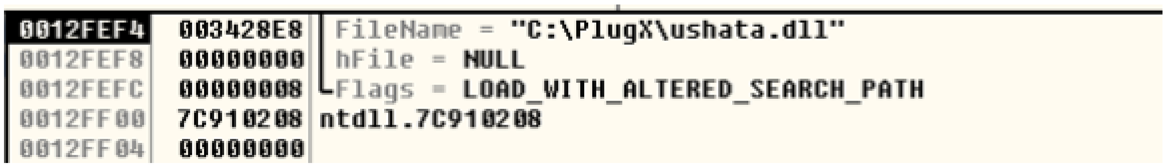
Name of signer:	E-mail address:	Timestamp
COMODO Time S...	Not available	Monday, June 17, 2...

Certificate Information
This certificate is intended for the following purpose(s):
•Ensures software came from software publisher
•Protects software from alteration after publication

* Refer to the certification authority's statement for details.

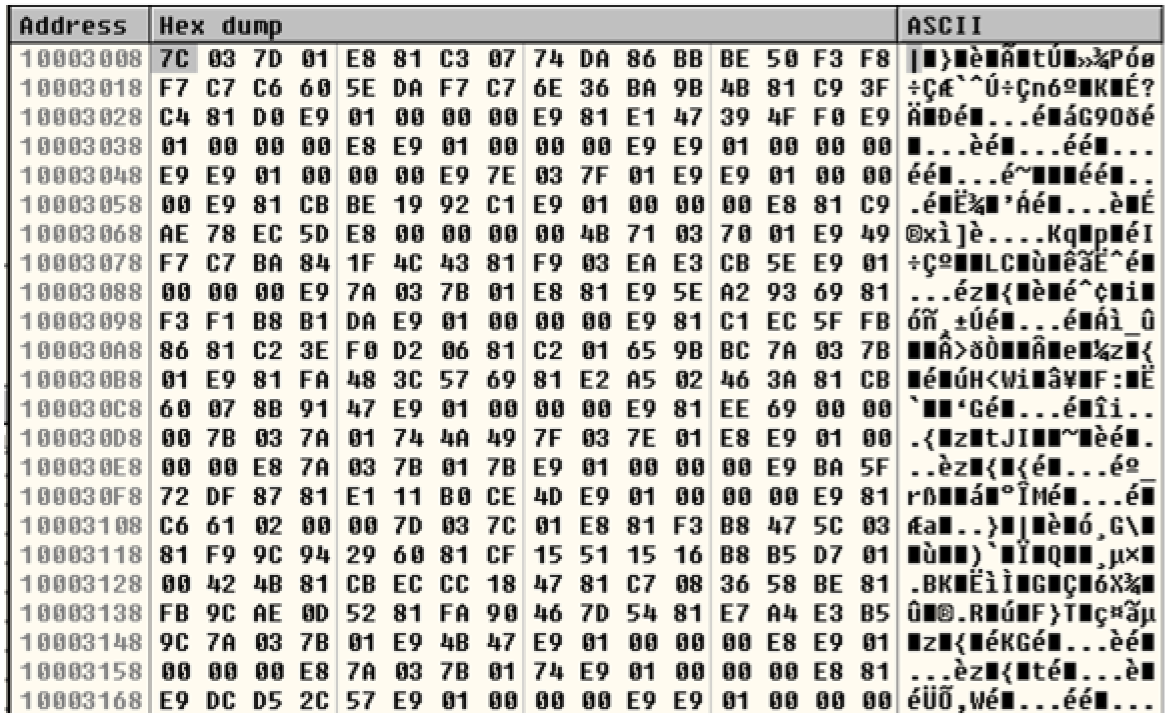
Issued to: Kaspersky Lab
Issued by: DigiCert High Assurance Code Signing CA-1
Valid from 2/22/2013 to 4/20/2015

So, let’s look at the mechanics of what happens when the self-extracting archive is executed. The three files are extracted to a temporary directory and “avp.exe” is executed. The “avp.exe” when executed will load “ushata.dll” from the running directory due to the DLL search order hijacking using Kernel32.LoadLibrary API.



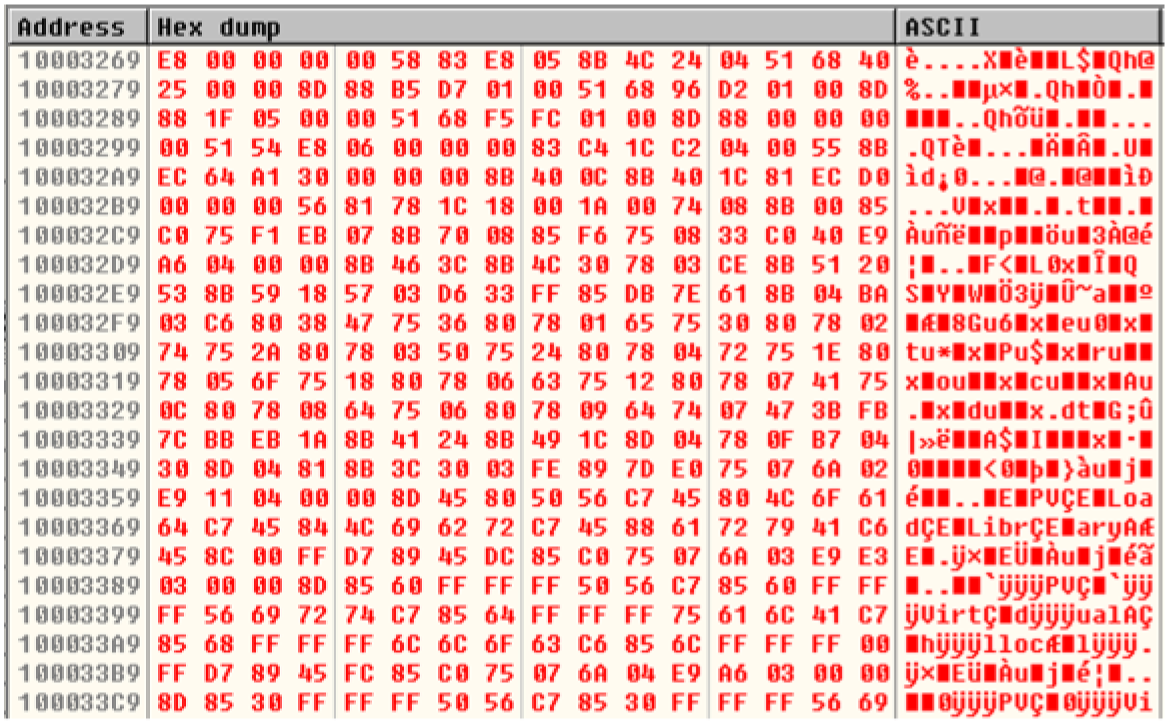
```
0012FEF4  003428E8  FileName = "C:\PlugX\ushata.dll"
0012FEF8  00000000  hFile = NULL
0012FEFC  00000008 └Flags = LOAD_WITH_ALTERED_SEARCH_PATH
0012FF00  7C910208 ntdll.7C910208
0012FF04  00000000
```

Then “ushata.dll” DLL entry point is executed. The DLL entry point contains code that verifies if the system date is equal or higher than 20130808. If yes it will get a handle to “ushata.DLL.818”, reads its contents into memory and changes the memory address segment permissions to RWX using Kernel32.VirtualProtect API. Finally, returns to the first instruction of the loaded file (shellcode). The file “ushata.DLL.818” contains obfuscated shellcode. The picture below shows the beginning of the obfuscated shellcode.



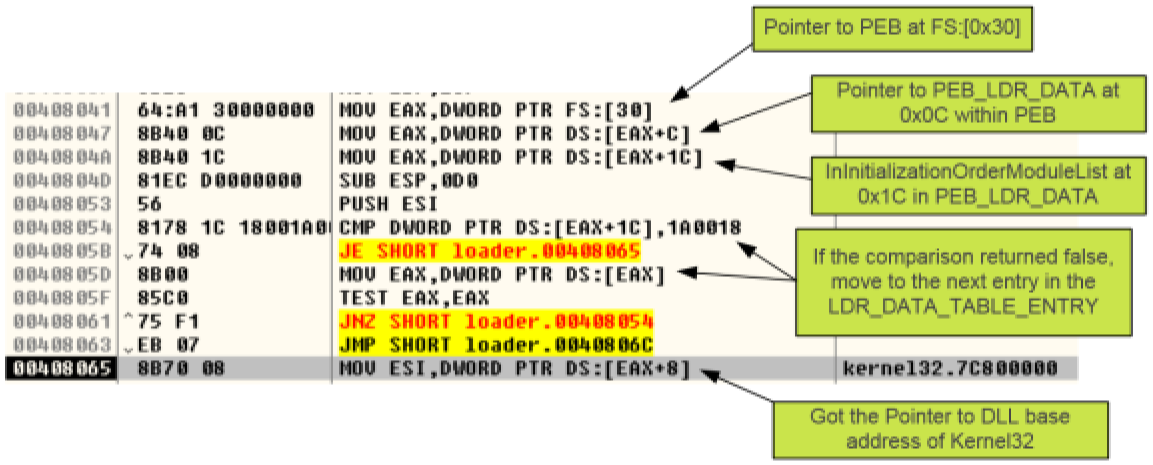
```
Address   Hex dump                                         ASCII
10003008  7C 03 7D 01 E8 81 C3 07 74 DA 86 BB BE 50 F3 F8  |■}■è■Ã■tÚ■»¾Póø
10003018  F7 C7 C6 60 5E DA F7 C7 6E 36 BA 9B 4B 81 C9 3F  ÷ÇÆ`^Ú÷Çn6º■K■É?
10003028  C4 81 D0 E9 01 00 00 00 E9 81 E1 47 39 4F F0 E9  Ä■Ðé■...é■áG9Oðé
10003038  01 00 00 00 E8 E9 01 00 00 00 E9 E9 01 00 00 00  ■...èé■...éé■...
10003048  E9 E9 01 00 00 00 E9 7E 03 7F 01 E9 E9 01 00 00  éé■...é~■■■éé■..
10003058  00 E9 81 CB BE 19 92 C1 E9 01 00 00 00 E8 81 C9  .é■Ë¾■'Áé■...è■É
10003068  AE 78 EC 5D E8 00 00 00 00 4B 71 03 70 01 E9 49  ®xì]è....Kq■p■éI
10003078  F7 C7 BA 84 1F 4C 43 81 F9 03 EA E3 CB 5E E9 01  ÷Çº■■LC■ù■êãË^é■
10003088  00 00 00 E9 7A 03 7B 01 E8 81 E9 5E A2 93 69 81  ...éz■{■è■é^¢■i■
10003098  F3 F1 B8 B1 DA E9 01 00 00 00 E9 81 C1 EC 5F FB  óñ¸±Úé■...é■Áì_û
100030A8  86 81 C2 3E F0 D2 06 81 C2 01 65 9B BC 7A 03 7B  ■■Â>ðÒ■■Â■e■¼z■{
100030B8  01 E9 81 FA 48 3C 57 69 81 E2 A5 02 46 3A 81 CB  ■é■úH<Wi■â¥■F:■Ë
100030C8  60 07 8B 91 47 E9 01 00 00 00 E9 81 EE 69 00 00  `■■'Gé■...é■îi..
100030D8  00 7B 03 7A 01 74 4A 49 7F 03 7E 01 E8 E9 01 00  .{■z■tJI■■~■èé■.
100030E8  00 00 E8 7A 03 7B 01 7B E9 01 00 00 00 E9 BA 5F  ..èz■{■{é■...éº_
100030F8  72 DF 87 81 E1 11 B0 CE 4D E9 01 00 00 00 E9 81  rß■■á■°ÎMé■...é■
10003108  C6 61 02 00 00 7D 03 7C 01 E8 81 F3 B8 47 5C 03  Æa■..}■|■è■ó¸G\■
10003118  81 F9 9C 94 29 60 81 CF 15 51 15 16 B8 B5 D7 01  ■ù■■)`■Ï■Q■■¸µ×■
10003128  00 42 4B 81 CB EC CC 18 47 81 C7 08 36 58 BE 81  .BK■ËìÌ■G■Ç■6X¾■
10003138  FB 9C AE 0D 52 81 FA 90 46 7D 54 81 E7 A4 E3 B5  û■®.R■ú■F}T■ç¤ãµ
10003148  9C 7A 03 7B 01 E9 4B 47 E9 01 00 00 00 E8 E9 01  ■z■{■éKGé■...èé■
10003158  00 00 00 E8 7A 03 7B 01 74 E9 01 00 00 00 E8 81  ...èz■{■té■...è■
10003168  E9 DC D5 2C 57 E9 01 00 00 00 E9 E9 01 00 00 00  éÜÕ,Wé■...éé■...
```

The shellcode unpacks itself using a custom algorithm. This shellcode contains position independent code. Figure below shows the unpacked shellcode.



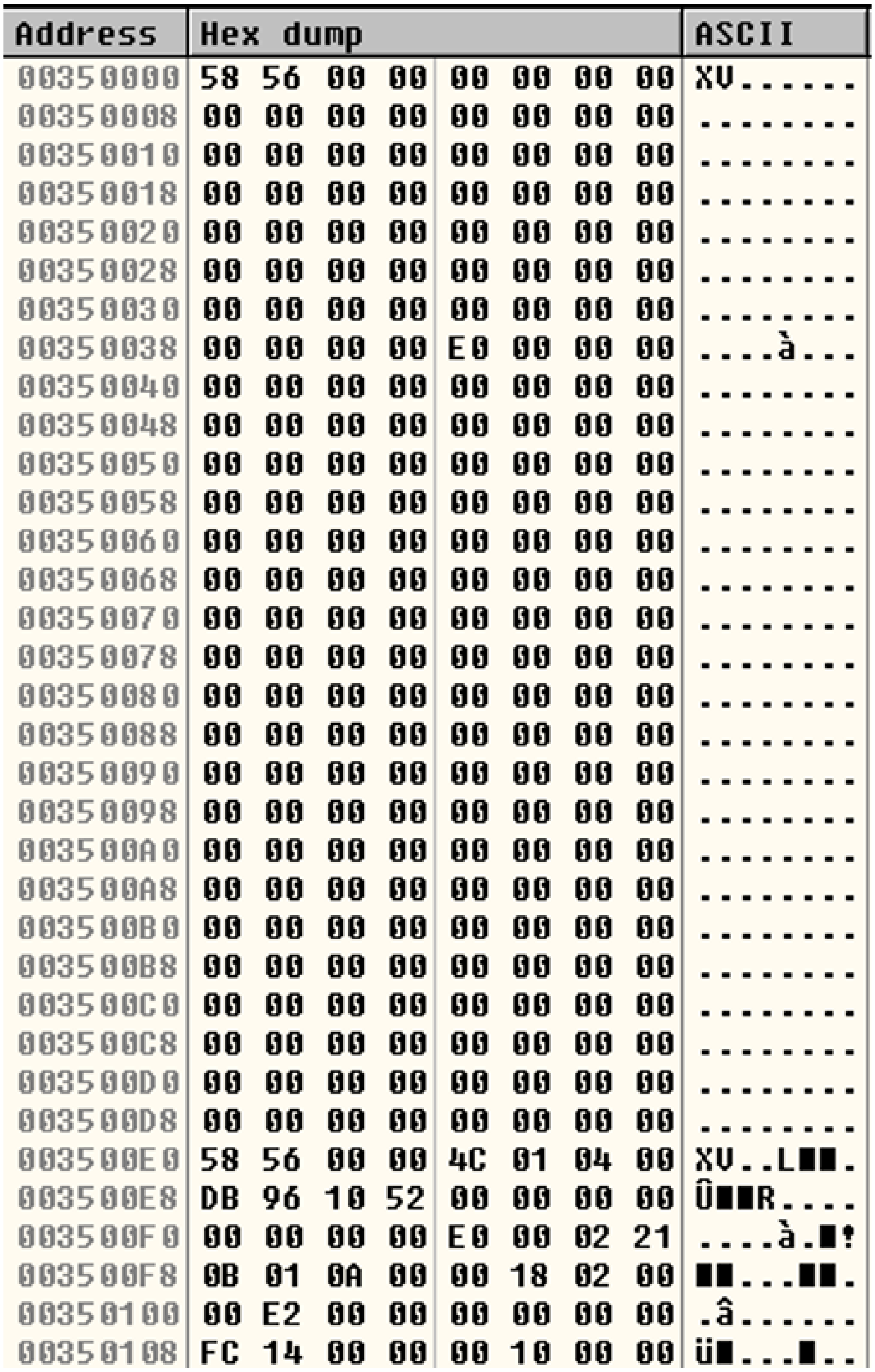
```
Address   Hex dump                                         ASCII
10003269  E8 00 00 00 00 58 83 E8 05 8B 4C 24 04 51 68 40  è....X■è■■L$■Qh@
10003279  25 00 00 8D 88 B5 D7 01 00 51 68 96 D2 01 00 8D  %..■■µ×■.Qh■Ò■.■
10003289  88 1F 05 00 00 51 68 F5 FC 01 00 8D 88 00 00 00  ■■■..Qhõü■.■■...
10003299  00 51 54 E8 06 00 00 00 83 C4 1C C2 04 00 55 8B  .QTè■...■Ä■Â■.U■
100032A9  EC 64 A1 30 00 00 00 8B 40 0C 8B 40 1C 81 EC D0  ìd¡0...■@.■@■■ìÐ
100032B9  00 00 00 56 81 78 1C 18 00 1A 00 74 08 8B 00 85  ...V■x■■.■.t■■.■
100032C9  C0 75 F1 EB 07 8B 70 08 85 F6 75 08 33 C0 40 E9  Àuñë■■p■■öu■3À@é
100032D9  A6 04 00 00 8B 46 3C 8B 4C 30 78 03 CE 8B 51 20  ¦■..■F<■L0x■Î■Q 
100032E9  53 8B 59 18 57 03 D6 33 FF 85 DB 7E 61 8B 04 BA  S■Y■W■Ö3ÿ■Û~a■■º
100032F9  03 C6 80 38 47 75 36 80 78 01 65 75 30 80 78 02  ■Æ■8Gu6■x■eu0■x■
10003309  74 75 2A 80 78 03 50 75 24 80 78 04 72 75 1E 80  tu*■x■Pu$■x■ru■■
10003319  78 05 6F 75 18 80 78 06 63 75 12 80 78 07 41 75  x■ou■■x■cu■■x■Au
10003329  0C 80 78 08 64 75 06 80 78 09 64 74 07 47 3B FB  .■x■du■■x.dt■G;û
10003339  7C BB EB 1A 8B 41 24 8B 49 1C 8D 04 78 0F B7 04  |»ë■■A$■I■■■x■·■
10003349  30 8D 04 81 8B 3C 30 03 FE 89 7D E0 75 07 6A 02  0■■■■<0■þ■}àu■j■
10003359  E9 11 04 00 00 8D 45 80 50 56 C7 45 80 4C 6F 61  é■■..■E■PVÇE■Loa
10003369  64 C7 45 84 4C 69 62 72 C7 45 88 61 72 79 41 C6  dÇE■LibrÇE■aryAÆ
10003379  45 8C 00 FF D7 89 45 DC 85 C0 75 07 6A 03 E9 E3  E■.ÿ×■EÜ■Àu■j■éã
10003389  03 00 00 8D 85 60 FF FF FF 50 56 C7 85 60 FF FF  ■..■■`ÿÿÿPVÇ■`ÿÿ
10003399  FF 56 69 72 74 C7 85 64 FF FF FF 75 61 6C 41 C7  ÿVirtÇ■dÿÿÿualAÇ
100033A9  85 68 FF FF FF 6C 6C 6F 63 C6 85 6C FF FF FF 00  ■hÿÿÿllocÆ■lÿÿÿ.
100033B9  FF D7 89 45 FC 85 C0 75 07 6A 04 E9 A6 03 00 00  ÿ×■Eü■Àu■j■é¦■..
100033C9  8D 85 30 FF FF FF 50 56 C7 85 30 FF FF FF 56 69  ■■0ÿÿÿPVÇ■0ÿÿÿVi
```

The shellcode starts by locating the kernel32.dll address by accessing the Thread Information Block (TIB) that contains a pointer to the Process Environment Block (PEB) structure. Figure below shows a snippet of the shellcode that contains the different sequence of assembly instructions for the code to find the Kernel32.dll.


Pointer to PEB at FS:[0x30]
Pointer to PEB_LDR_DATA at 0x0C within PEB
InInitializationOrderModuleList at 0x1C in PEB_LDR_DATA
If the comparison returned false, move to the next entry in the LDR_DATA_TABLE_ENTRY
Got the Pointer to DLL base address of Kernel32

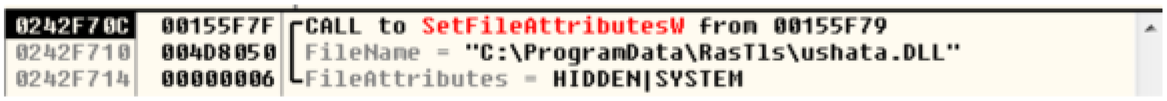
```
00408041   64:A1 30000000   MOV EAX,DWORD PTR FS:[30]
00408047   8B40 0C          MOV EAX,DWORD PTR DS:[EAX+C]
0040804A   8B40 1C          MOV EAX,DWORD PTR DS:[EAX+1C]
0040804D   81EC D0000000    SUB ESP,0D0
00408053   56               PUSH ESI
00408054   8178 1C 18001A0  CMP DWORD PTR DS:[EAX+1C],1A0018
0040805B  ⌄74 08            JE SHORT loader.00408065
0040805D   8B00             MOV EAX,DWORD PTR DS:[EAX]
0040805F   85C0             TEST EAX,EAX
00408061  ^75 F1            JNZ SHORT loader.00408054
00408063  ⌄EB 07            JMP SHORT loader.0040806C
00408065   8B70 08          MOV ESI,DWORD PTR DS:[EAX+8]                kernel32.7C800000
```

It then reads kernel32.dll export table to locate the desired Windows API’s by comparing them with stacked strings. Then, the shellcode decompresses a DLL (offset 0x784) MD5 333e2767c8e575fbbb1c47147b9f9643, into memory using the LZNT1 algorithm by leveraging ntdll.dll.RtlDecompressBuffer API. The DLL contains the PE header replaced with the “XV” value. Restoring the PE header signature allows us to recover the malicious DLL.



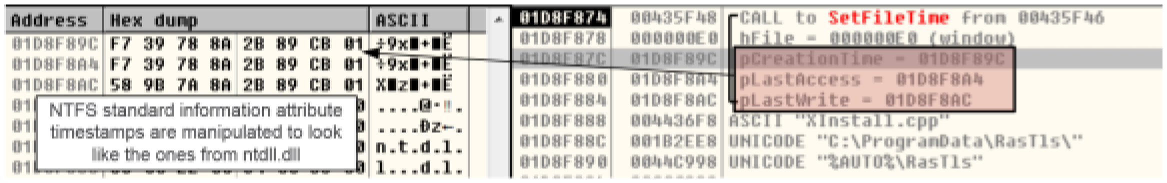
```
Address   Hex dump                 ASCII
00350000  58 56 00 00 00 00 00 00  XV......
00350008  00 00 00 00 00 00 00 00  ........
00350010  00 00 00 00 00 00 00 00  ........
00350018  00 00 00 00 00 00 00 00  ........
00350020  00 00 00 00 00 00 00 00  ........
00350028  00 00 00 00 00 00 00 00  ........
00350030  00 00 00 00 00 00 00 00  ........
00350038  00 00 00 00 E0 00 00 00  ....à...
00350040  00 00 00 00 00 00 00 00  ........
00350048  00 00 00 00 00 00 00 00  ........
00350050  00 00 00 00 00 00 00 00  ........
00350058  00 00 00 00 00 00 00 00  ........
00350060  00 00 00 00 00 00 00 00  ........
00350068  00 00 00 00 00 00 00 00  ........
00350070  00 00 00 00 00 00 00 00  ........
00350078  00 00 00 00 00 00 00 00  ........
00350080  00 00 00 00 00 00 00 00  ........
00350088  00 00 00 00 00 00 00 00  ........
00350090  00 00 00 00 00 00 00 00  ........
00350098  00 00 00 00 00 00 00 00  ........
003500A0  00 00 00 00 00 00 00 00  ........
003500A8  00 00 00 00 00 00 00 00  ........
003500B0  00 00 00 00 00 00 00 00  ........
003500B8  00 00 00 00 00 00 00 00  ........
003500C0  00 00 00 00 00 00 00 00  ........
003500C8  00 00 00 00 00 00 00 00  ........
003500D0  00 00 00 00 00 00 00 00  ........
003500D8  00 00 00 00 00 00 00 00  ........
003500E0  58 56 00 00 4C 01 04 00  XV..L■■.
003500E8  DB 96 10 52 00 00 00 00  Û■■R....
003500F0  00 00 00 00 E0 00 02 21  ....à.■!
003500F8  0B 01 0A 00 00 18 02 00  ■■...■■.
00350100  00 E2 00 00 00 00 00 00  .â......
00350108  FC 14 00 00 00 10 00 00  ü■...■..
```

Next, the payload will start performing different actions to achieve persistence. On Windows 7 and beyond, PlugX creates a folder “%ProgramData%\RasTl” where “RasTl” matches the installation settings defined in the builder. Then, it changes the folder attributes to “SYSTEM|HIDDEN” using the SetFileAttributesW API. Next, copies its three components into the folder and sets all files with the “SYSTEM|HIDDEN” attribute.



```
0242F70C  00155F7F  ┌CALL to SetFileAttributesW from 00155F79
0242F710  004D8050  │FileName = "C:\ProgramData\RasTls\ushata.DLL"
0242F714  00000006  └FileAttributes = HIDDEN|SYSTEM
```

The payload also modifies the timestamps of the created directory and files with the timestamps obtained from ntdll.dll using the SetFileTime API.

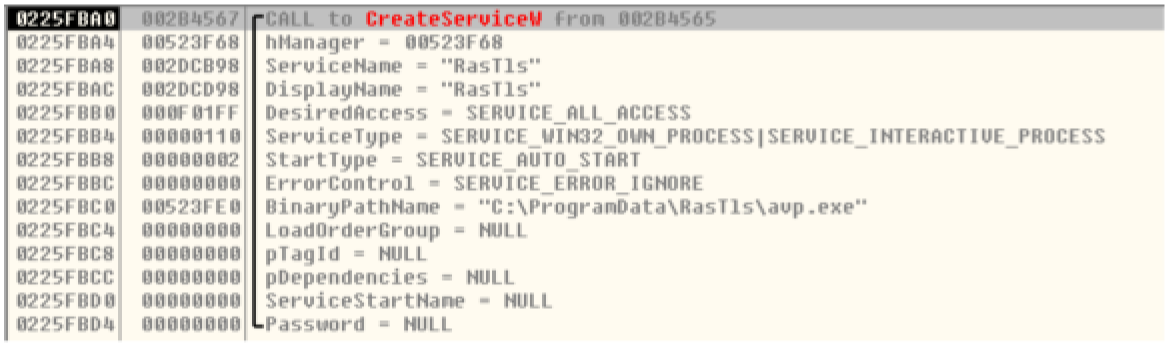


```
Address   Hex dump                  ASCII
01D8F89C  F7 39 78 8A 2B 89 CB 01  ÷9x■+■Ë
01D8F8A4  F7 39 78 8A 2B 89 CB 01  ÷9x■+■Ë
01D8F8AC  58 9B 7A 8A 2B 89 CB 01  X■z■+■Ë
```

NTFS standard information attribute timestamps are manipulated to look like the ones from ntdll.dll

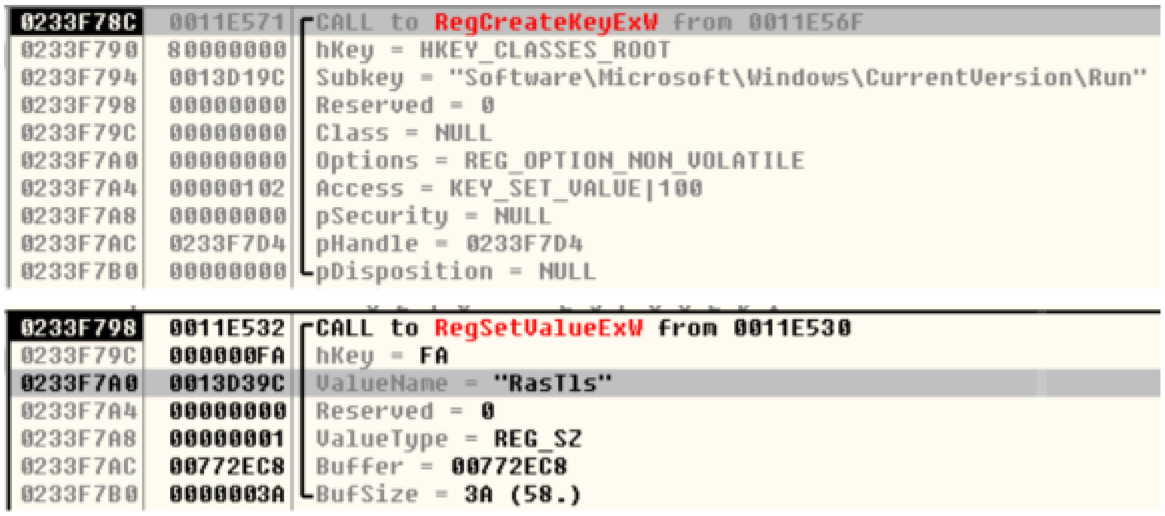
```
01D8F874  00435F48  ┌CALL to SetFileTime from 00435F46
01D8F878  000000E0  │hFile = 000000E0 (window)
01D8F87C  01D8F89C  │pCreationTime = 01D8F89C
01D8F880  01D8F8A4  │pLastAccess = 01D8F8A4
01D8F884  01D8F8AC  └pLastWrite = 01D8F8AC
01D8F888  004436F8  ASCII "XInstall.cpp"
01D8F88C  001B2EE8  UNICODE "C:\ProgramData\RasTls\"
01D8F890  0044C998  UNICODE "%AUTO%\RasTls"
```

Then it creates the service “RasTl” where the ImagePath points to “%ProgramData%\RasTl\avp.exe”



```
0225FBA0  002B4567  ┌CALL to CreateServiceW from 002B4565
0225FBA4  00523F68  │hManager = 00523F68
0225FBA8  002DCB98  │ServiceName = "RasTls"
0225FBAC  002DCD98  │DisplayName = "RasTls"
0225FBB0  000F01FF  │DesiredAccess = SERVICE_ALL_ACCESS
0225FBB4  00000110  │ServiceType = SERVICE_WIN32_OWN_PROCESS|SERVICE_INTERACTIVE_PROCESS
0225FBB8  00000002  │StartType = SERVICE_AUTO_START
0225FBBC  00000000  │ErrorControl = SERVICE_ERROR_IGNORE
0225FBC0  00523FE0  │BinaryPathName = "C:\ProgramData\RasTls\avp.exe"
0225FBC4  00000000  │LoadOrderGroup = NULL
0225FBC8  00000000  │pTagId = NULL
0225FBCC  00000000  │pDependencies = NULL
0225FBD0  00000000  │ServiceStartName = NULL
0225FBD4  00000000  └Password = NULL
```

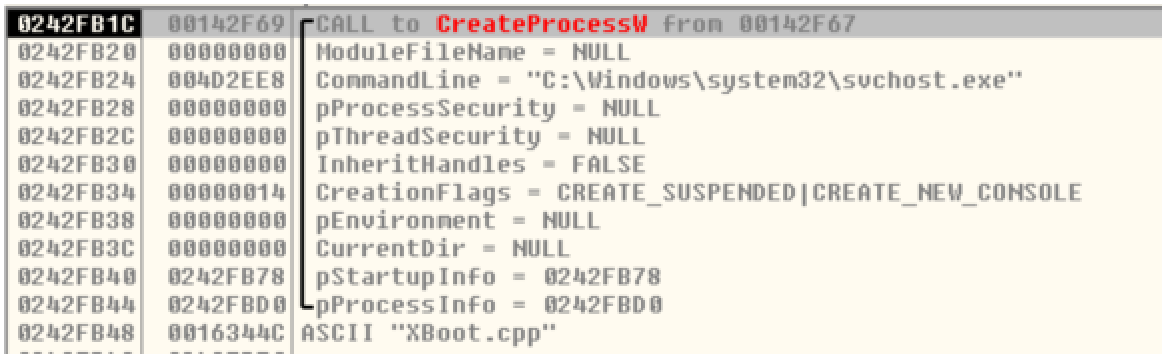
If the malware fails to start the just installed service, it will delete it and then it will create a persistence mechanism in the registry by setting the registry value “C:\ProgramData\RasTls\avp.exe” to the key “HKLM\SOFTWARE\Classes\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\RasTls” using the RegSetValueExW API.



```
0233F78C  0011E571  ┌CALL to RegCreateKeyExW from 0011E56F
0233F790  80000000  │hKey = HKEY_CLASSES_ROOT
0233F794  0013D19C  │Subkey = "Software\Microsoft\Windows\CurrentVersion\Run"
0233F798  00000000  │Reserved = 0
0233F79C  00000000  │Class = NULL
0233F7A0  00000000  │Options = REG_OPTION_NON_VOLATILE
0233F7A4  00000102  │Access = KEY_SET_VALUE|100
0233F7A8  00000000  │pSecurity = NULL
0233F7AC  0233F7D4  │pHandle = 0233F7D4
0233F7B0  00000000  └pDisposition = NULL
```

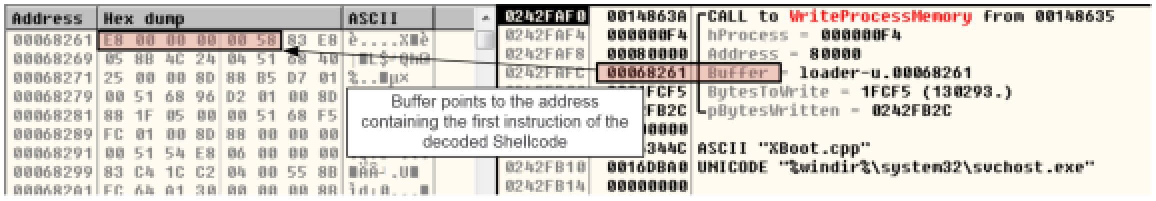
```
0233F798  0011E532  ┌CALL to RegSetValueExW from 0011E530
0233F79C  000000FA  │hKey = FA
0233F7A0  0013D39C  │ValueName = "RasTls"
0233F7A4  00000000  │Reserved = 0
0233F7A8  00000001  │ValueType = REG_SZ
0233F7AC  00772EC8  │Buffer = 00772EC8
0233F7B0  0000003A  └BufSize = 3A (58.)
```

If the builder options had the Keylogger functionality enabled, then it may create a file with a random name such as “%ProgramData%\RasTl\rjowfhxnzmdknsixtx” that stores the key strokes. If the payload has been built with Screen capture functionality, it may create the folder “%ProgramData%\RasTl \RasTl\Screen” to store JPG images in the format <datetime>.jpg that are taken at the frequency specified during the build process. The payload may also create the file “%ProgramData%\DEBUG.LOG” that contains debugging information about its execution (also interesting that during execution the malware outputs debug messages about what is happening using the OutputDebugString API. This messages could be viewed with DebugView from SysInternals). The malicious code completes its mission by starting a new instance of “svchost.exe” and then injects the malicious code into svchost.exe process address space using process hollowing technique. The pictures below shows the first step of the process hollowing technique where the payload creates a new “svchost.exe” instance in SUSPENDED state.



```
0242FB1C  00142F69  ┌CALL to CreateProcessW from 00142F67
0242FB20  00000000  │ModuleFileName = NULL
0242FB24  004D2EE8  │CommandLine = "C:\Windows\system32\svchost.exe"
0242FB28  00000000  │pProcessSecurity = NULL
0242FB2C  00000000  │pThreadSecurity = NULL
0242FB30  00000000  │InheritHandles = FALSE
0242FB34  00000014  │CreationFlags = CREATE_SUSPENDED|CREATE_NEW_CONSOLE
0242FB38  00000000  │pEnvironment = NULL
0242FB3C  00000000  │CurrentDir = NULL
0242FB40  0242FB78  │pStartupInfo = 0242FB78
0242FB44  0242FBD0  └pProcessInfo = 0242FBD0
0242FB48  0016344C  ASCII "XBoot.cpp"
```

and then uses WriteProcessMemory API to inject the malicious payload

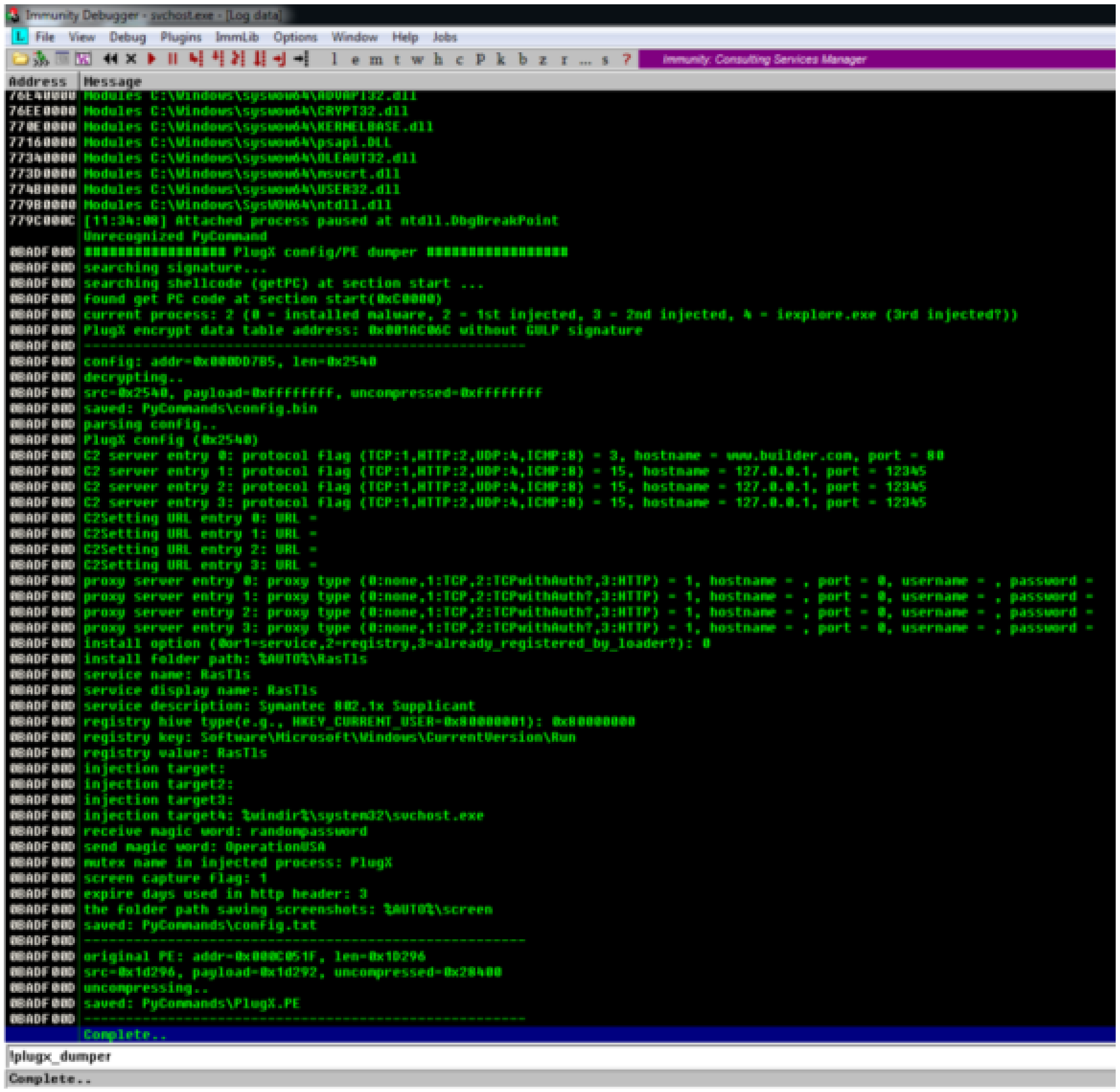


```
Address   Hex dump                 ASCII
00068261  E8 00 00 00 00 58 83 E8  è....XⅡè
00068269  05 8B 4C 24 04 51 68 40  ⅡL$◆QhⅡ
00068271  25 00 00 8D 88 B5 D7 01  %..Ⅱµx
00068279  00 51 68 96 D2 01 00 8D
00068281  88 1F 05 00 00 51 68 F5
00068289  FC 01 00 8D 88 00 00 00
00068291  00 51 54 E8 06 00 00 00
00068299  83 C4 1C C2 04 00 55 8B  ⅡÄ◆.UⅡ
```

```
0242FAF0  0014863A  ┌CALL to WriteProcessMemory from 00148635
0242FAF4  000000F4  │hProcess = 000000F4
0242FAF8  00080000  │Address = 80000
0242FAFC  00068261  │Buffer = loader-u.00068261
               1FCF5  │BytesToWrite = 1FCF5 (130293.)
               FB2C   └pBytesWritten = 0242FB2C
               0000
               344C   ASCII "XBoot.cpp"
0242FB10  0016DBA0  UNICODE "%windir%\system32\svchost.exe"
0242FB14  00000000
```

Buffer points to the address containing the first instruction of the decoded Shellcode

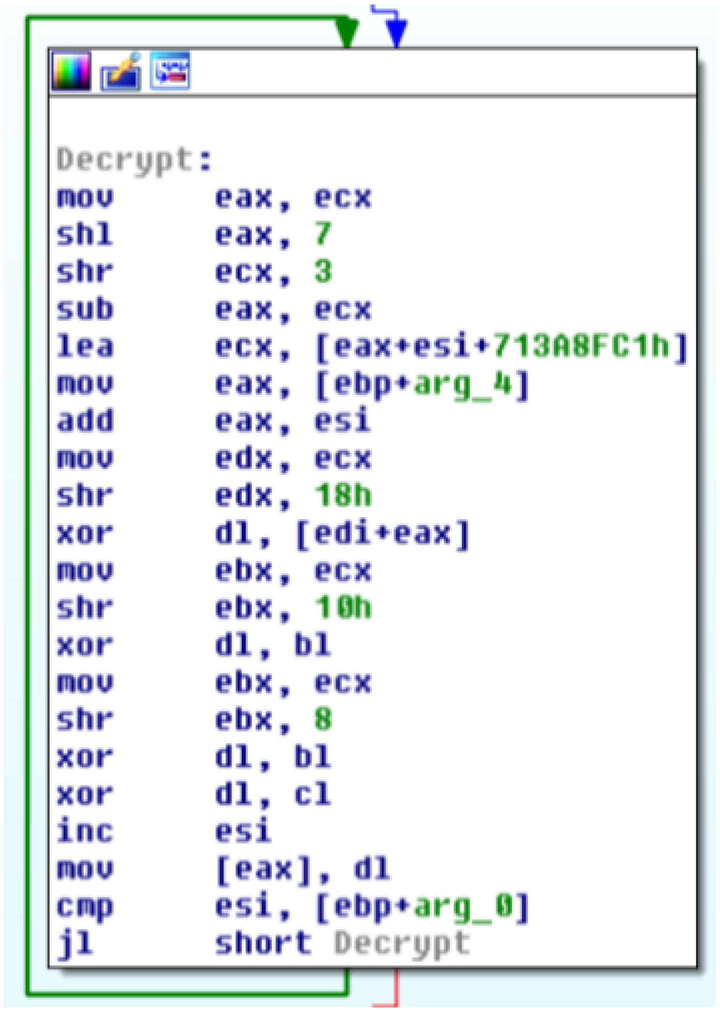
Then the main thread, which is still in suspended state, is changed in order to point to the entry point of the new image base using the SetThreadContext API. Finally, the ResumeThread API is invoked and the malicious code starts executing. The malware also has the capabilities to bypass User Account Control (UAC) if needed. From this moment onward, the control is passed over “svchost.exe” and Plug-X starts doing its thing. In this case we have the builder so we know the settings which were defined during building process. However, we would like to understand how could we extract the configuration settings. During Black Hat 2014, Takahiro Haruyama and Hiroshi Suzuki gave a presentation titled “I know You Want Me – Unplugging PlugX” where the authors go to great length analyzing a variety of PlugX samples, its evolution and categorizing them into threat groups. But better is that the Takahiro released a set of PlugX parsers for the different types of PlugX samples i.e, Type I, Type II and Type III. How can we use this parser? The one we are dealing in this article is considered a PlugX type II. To dump the configuration, we need to use Immunity Debugger and use the Python API. We need to place the “plugx_dumper.py” file into the “PyCommands” folder inside Immunity Debugger installation path. Then attached the debugger to the infected process e.g, “svchost.exe” and run the plugin. The plugin will dump the configuration settings and will also extract the decompressed DLL


Immunity Debugger - svchost.exe - [Log data]
File View Debug Plugins ImmLib Options Window Help Jobs
Immunity: Consulting Services Manager

```
Address  Message
76E40000 Modules C:\Windows\sysWOW64\ADVAPI32.dll
76EE0000 Modules C:\Windows\syswow64\CRYPT32.dll
770E0000 Modules C:\Windows\syswow64\KERNELBASE.dll
77160000 Modules C:\Windows\syswow64\psapi.DLL
77340000 Modules C:\Windows\syswow64\OLEAUT32.dll
773D0000 Modules C:\Windows\syswow64\msvcrt.dll
77480000 Modules C:\Windows\syswow64\USER32.dll
779B0000 Modules C:\Windows\SysWOW64\ntdll.dll
779C000C [11:34:08] Attached process paused at ntdll.DbgBreakPoint
         Unrecognized PyCommand
0BADF00D ################## PlugX config/PE dumper ##################
0BADF00D searching signature...
0BADF00D searching shellcode (getPC) at section start ...
0BADF00D found get PC code at section start(0xC0000)
0BADF00D current process: 2 (0 - installed malware, 2 - 1st injected, 3 - 2nd injected, 4 - iexplore.exe (3rd injected?))
0BADF00D PlugX encrypt data table address: 0x001AC06C without GULP signature
0BADF00D -----------------------------------------------------
0BADF00D config: addr=0x000DD7B5, len=0x2540
0BADF00D decrypting..
0BADF00D src=0x2540, payload=0xffffffff, uncompressed=0xffffffff
0BADF00D saved: PyCommands\config.bin
0BADF00D parsing config..
0BADF00D PlugX config (0x2540)
0BADF00D C2 server entry 0: protocol flag (TCP:1,HTTP:2,UDP:4,ICMP:8) = 3, hostname = www.builder.com, port = 80
0BADF00D C2 server entry 1: protocol flag (TCP:1,HTTP:2,UDP:4,ICMP:8) = 15, hostname = 127.0.0.1, port = 12345
0BADF00D C2 server entry 2: protocol flag (TCP:1,HTTP:2,UDP:4,ICMP:8) = 15, hostname = 127.0.0.1, port = 12345
0BADF00D C2 server entry 3: protocol flag (TCP:1,HTTP:2,UDP:4,ICMP:8) = 15, hostname = 127.0.0.1, port = 12345
0BADF00D C2Setting URL entry 0: URL =
0BADF00D C2Setting URL entry 1: URL =
0BADF00D C2Setting URL entry 2: URL =
0BADF00D C2Setting URL entry 3: URL =
0BADF00D proxy server entry 0: proxy type (0:none,1:TCP,2:TCPwithAuth?,3:HTTP) = 1, hostname = , port = 0, username = , password =
0BADF00D proxy server entry 1: proxy type (0:none,1:TCP,2:TCPwithAuth?,3:HTTP) = 1, hostname = , port = 0, username = , password =
0BADF00D proxy server entry 2: proxy type (0:none,1:TCP,2:TCPwithAuth?,3:HTTP) = 1, hostname = , port = 0, username = , password =
0BADF00D proxy server entry 3: proxy type (0:none,1:TCP,2:TCPwithAuth?,3:HTTP) = 1, hostname = , port = 0, username = , password =
0BADF00D install option (0or1=service,2=registry,3=already_registered_by_loader?): 0
0BADF00D install folder path: %AUTO%\RasTls
0BADF00D service name: RasTls
0BADF00D service display name: RasTls
0BADF00D service description: Symantec 802.1x Supplicant
0BADF00D registry hive type(e.g., HKEY_CURRENT_USER=0x80000001): 0x80000000
0BADF00D registry key: Software\Microsoft\Windows\CurrentVersion\Run
0BADF00D registry value: RasTls
0BADF00D injection target:
0BADF00D injection target2:
0BADF00D injection target3:
0BADF00D injection target4: %windir%\system32\svchost.exe
0BADF00D receive magic word: randompassword
0BADF00D send magic word: OperationUSA
0BADF00D mutex name in injected process: PlugX
0BADF00D screen capture flag: 1
0BADF00D expire days used in http header: 3
0BADF00D the folder path saving screenshots: %AUTO%\screen
0BADF00D saved: PyCommands\config.txt
0BADF00D -----------------------------------------------------
0BADF00D original PE: addr=0x000C051F, len=0x1D296
0BADF00D src=0x1d296, payload=0x1d292, uncompressed=0x28400
0BADF00D uncompressing..
0BADF00D saved: PyCommands\PlugX.PE
0BADF00D -----------------------------------------------------
         Complete..
!plugx_dumper
Complete..
```

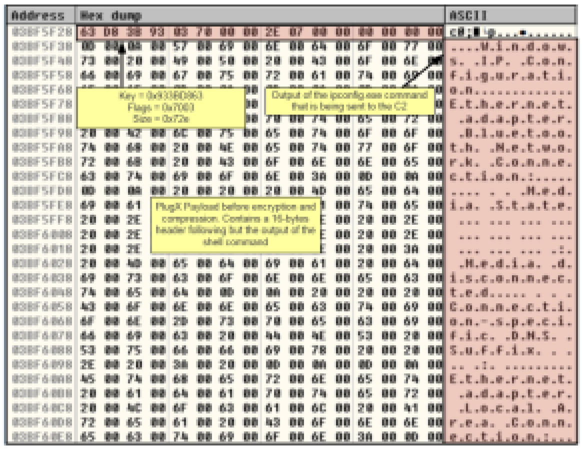
We can see that this parser is able to find the injected shellcode, decode its configuration and all the settings an attacker would set on the builder and also dump the injected DLL which contains the core functionality of the malware.

In terms of networking, as observed in the PlugX controller, the malware can be configured to speak with a controller using several network protocols. In this case we configured it to speak using HTTP on port 80. The network traffic contains a 16-byte header followed by a payload. The header is encoded with a custom routine and the payload is encoded and compressed with LZNT1. Far from a comprehensive analysis we launched a Shell prompt from the controller, typed command “ipconfig” and observed the network traffic. In parallel, we attached a debugger to “svchost.exe” and set breakpoints: on Ws2_32.dll!WSASend and Ws2_32.dll!WSARecv to capture the packets ; on ntdll.dll!RtlCompressBuffer and ntdll.dll!RtlDecompressBuffer to view the data before and after compression. ; On custom encoding routine to view the data before and after. The figure below shows a disassemble listing of the custom encoding routine.



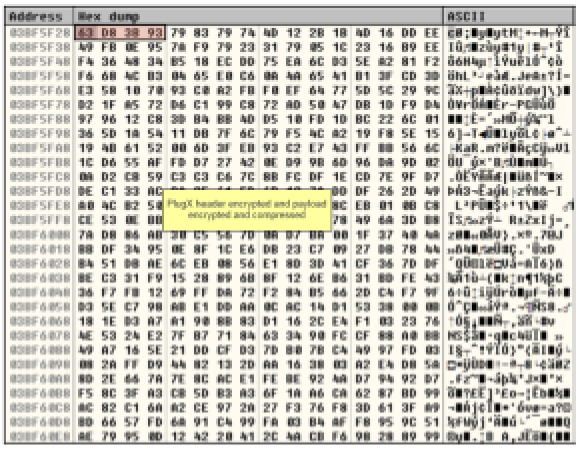
```
Decrypt:
mov     eax, ecx
shl     eax, 7
shr     ecx, 3
sub     eax, ecx
lea     ecx, [eax+esi+713A8FC1h]
mov     eax, [ebp+arg_4]
add     eax, esi
mov     edx, ecx
shr     edx, 18h
xor     dl, [edi+eax]
mov     ebx, ecx
shr     ebx, 10h
xor     dl, bl
mov     ebx, ecx
shr     ebx, 8
xor     dl, bl
xor     dl, cl
inc     esi
mov     [eax], dl
cmp     esi, [ebp+arg_0]
jl      short Decrypt
```

So, from a debugger view, with the right breakpoints we could start to observe what is happening. In the picture below, on the left-hand side it shows the packet before encoding and compression. It contains a 16-byte header, where the first 4-bytes are the key for the custom encoding routine. The next 4-bytes are the flags which contain the commands/plugins being used. Then the next 4-bytes is the size. After the header there is the payload which in this case contains is output of the ipconfig.exe command. On the right-hand side, we have the packet after encoding and compressing. It contains the 16-byte header encoded following by the payload encoded and compressed.



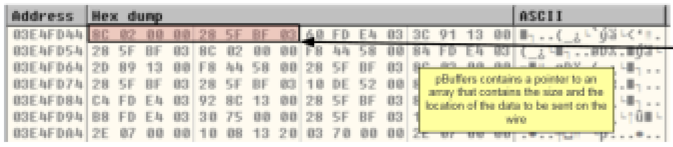
```
Address   Hex dump                                          ASCII
03BF5F28  63 D8 3B 93 03 70 00 00 2E 07 00 00 00 00 00 00  cØ;..p..........
03BF5F38  0D 00 0A 00 57 00 69 00 6E 00 64 00 6F 00 77 00  ....W.i.n.d.o.w.
03BF5F48  73 00 20 00 49 00 50 00 20 00 43 00 6F 00 6E 00  s. .I.P. .C.o.n.
03BF5F58  66 00 69 00 67 00 75 00 72 00 61 00 74 00 69 00  f.i.g.u.r.a.t.i.
03BF5F68                                                    o.n.............
03BF5F78                                                    E.t.h.e.r.n.e.t.
03BF5F88                                   70 00 74 00 65 00 72 00   .a.d.a.p.t.e.r.
03BF5F98  20 00 42 00 6C 00 75 00 65 00 74 00 6F 00 6F 00   .B.l.u.e.t.o.o.
03BF5FA8  74 00 68 00 20 00 4E 00 65 00 74 00 77 00 6F 00  t.h. .N.e.t.w.o.
03BF5FB8  72 00 6B 00 20 00 43 00 6F 00 6E 00 6E 00 65 00  r.k. .C.o.n.n.e.
03BF5FC8  63 00 74 00 69 00 6F 00 6E 00 3A 00 0D 00 0A 00  c.t.i.o.n.:.....
03BF5FD8  0D 00 0A 00 20 00 20 00 20 00 4D 00 65 00 64 00  .... . . .M.e.d.
03BF5FE8  69 00 61                          00 74 00 65 00  i.a. .S.t.a.t.e.
03BF5FF8  20 00 2E                          00 20 00 2E 00   . . . . . . . .
03BF6008  20 00 2E                          00 20 00 2E 00   . . . . . . . .
03BF6018  20 00 2E                          00 20 00 3A 00   . . . . . . :.
03BF6028  20 00 4D 00 65 00 64 00 69 00 61 00 20 00 64 00   .M.e.d.i.a. .d.
03BF6038  69 00 73 00 63 00 6F 00 6E 00 6E 00 65 00 63 00  i.s.c.o.n.n.e.c.
03BF6048  74 00 65 00 64 00 0D 00 0A 00 20 00 20 00 20 00  t.e.d..... . . .
03BF6058  43 00 6F 00 6E 00 6E 00 65 00 63 00 74 00 69 00  C.o.n.n.e.c.t.i.
03BF6068  6F 00 6E 00 2D 00 73 00 70 00 65 00 63 00 69 00  o.n.-.s.p.e.c.i.
03BF6078  66 00 69 00 63 00 20 00 44 00 4E 00 53 00 20 00  f.i.c. .D.N.S. .
03BF6088  53 00 75 00 66 00 66 00 69 00 78 00 20 00 20 00  S.u.f.f.i.x. . .
03BF6098  2E 00 20 00 3A 00 20 00 0D 00 0A 00 0D 00 0A 00  .. .:. .........
03BF60A8  45 00 74 00 68 00 65 00 72 00 6E 00 65 00 74 00  E.t.h.e.r.n.e.t.
03BF60B8  20 00 61 00 64 00 61 00 70 00 74 00 65 00 72 00   .a.d.a.p.t.e.r.
03BF60C8  20 00 4C 00 6F 00 63 00 61 00 6C 00 20 00 41 00   .L.o.c.a.l. .A.
03BF60D8  72 00 65 00 61 00 20 00 43 00 6F 00 6E 00 6E 00  r.e.a. .C.o.n.n.
03BF60E8  65 00 63 00 74 00 69 00 6F 00 6E 00 3A 00 0D 00  e.c.t.i.o.n.:...
```

Key = 0x933BD863
Flags = 0x7003
Size = 0x72e
Output of the ipconfig.exe command that is being sent to the C2
PlugX Payload before encryption and compression. Contains a 16-bytes header following but the output of the shell command



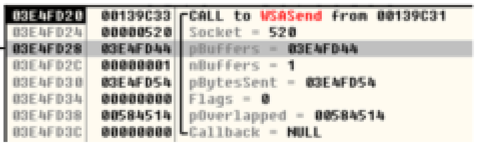
```
Address   Hex dump
03BF5F28  63 D8 3B 93 79 83 79 74 4D 12 2B 1B 4D 16 DD EE
03BF5F38  49 FB 0E 95 7A F9 79 23 31 79 05 1C 23 16 B9 EE
03BF5F48  F4 36 48 34 B5 18 EC DD 75 EA 6C D3 5E A2 81 F2
03BF5F58  F6 68 4C B3 04 65 E0 C6 0A 4A 65 41 B1 3F CD 3D
03BF5F68  E3 58 10 70 93 C0 A2 FB F0 EF 64 77 5D 5C 29 9C
03BF5F78  D2 1F A5 72 D6 C1 99 C8 72 AD 50 47 DB 1D F9 D4
03BF5F88  97 96 12 C8 3D B4 BB 4D D5 10 FD 1D BC 22 6C 01
03BF5F98  36 5D 1A 54 11 DB 7F 6C 79 F5 4C A2 19 F8 5E 15
03BF5FA8  19 4B 61 52 00 6D 3F EB 93 C2 E7 43 FF 08 56 6C
03BF5FB8  1C D6 55 AF FD D7 27 42 0E D9 9B 6D 96 DA 9D 02
03BF5FC8  0A D2 CB 59 C3 C3 C6 7C 8B FC DF 1E CD 7E 9F D7
03BF5FD8  DE C1 33 AC                         00 DF 26 2D 49
03BF5FE8  A0 4C B2 50                         8C EB 01 0B C8
03BF5FF8  CE 53 0E BB                         7A 49 6A 3D B8
03BF6008  7A D8 86 AB 30 C5 56 7D 0A D7 BA 00 1F 37 40 4A
03BF6018  BB DF 34 95 0E 8F 1C E6 DB 23 C7 09 27 DB 78 44
03BF6028  B4 51 DB AE 6C EB 08 56 E1 8D 3D 41 CF 36 7D DF
03BF6038  BE C3 31 F9 15 28 89 6B 8F 12 6E B6 31 BD FE 43
03BF6048  36 F7 FB 12 69 FF DA 72 F2 84 B5 66 2D C4 F7 9F
03BF6058  D3 5E C7 98 AB E1 DD AA 0C AC 14 D1 53 38 00 0B
03BF6068  18 1E D3 A7 A1 90 8B 83 D1 16 2C E4 F1 03 23 76
03BF6078  4E 53 24 E2 7F B7 71 84 63 34 90 FC CF 88 A0 BB
03BF6088  49 A7 16 5E 21 DD CF D3 7D B0 7B C4 49 97 FD 03
03BF6098  08 2A FF D9 44 82 13 2D AA 16 38 03 A2 E4 D8 5A
03BF60A8  8D 2E 66 7A 7E 8C AC E1 FE BE 92 4A D7 94 92 D7
03BF60B8  F5 8C 3F A3 CB 5D B3 A3 6F 1A A6 CA 62 87 BD 99
03BF60C8  AC 82 C1 6A A2 CE 97 2A 27 F3 76 F8 3D 61 3F A9
03BF60D8  BD 66 57 FD 6A 91 C4 99 FA 03 B4 AF F8 95 9C 51
03BF60E8  AE 79 95 0D 12 42 20 41 2C 4A CB F6 98 28 89 99
```

Then, the malware uses WSASend API to send the traffic.



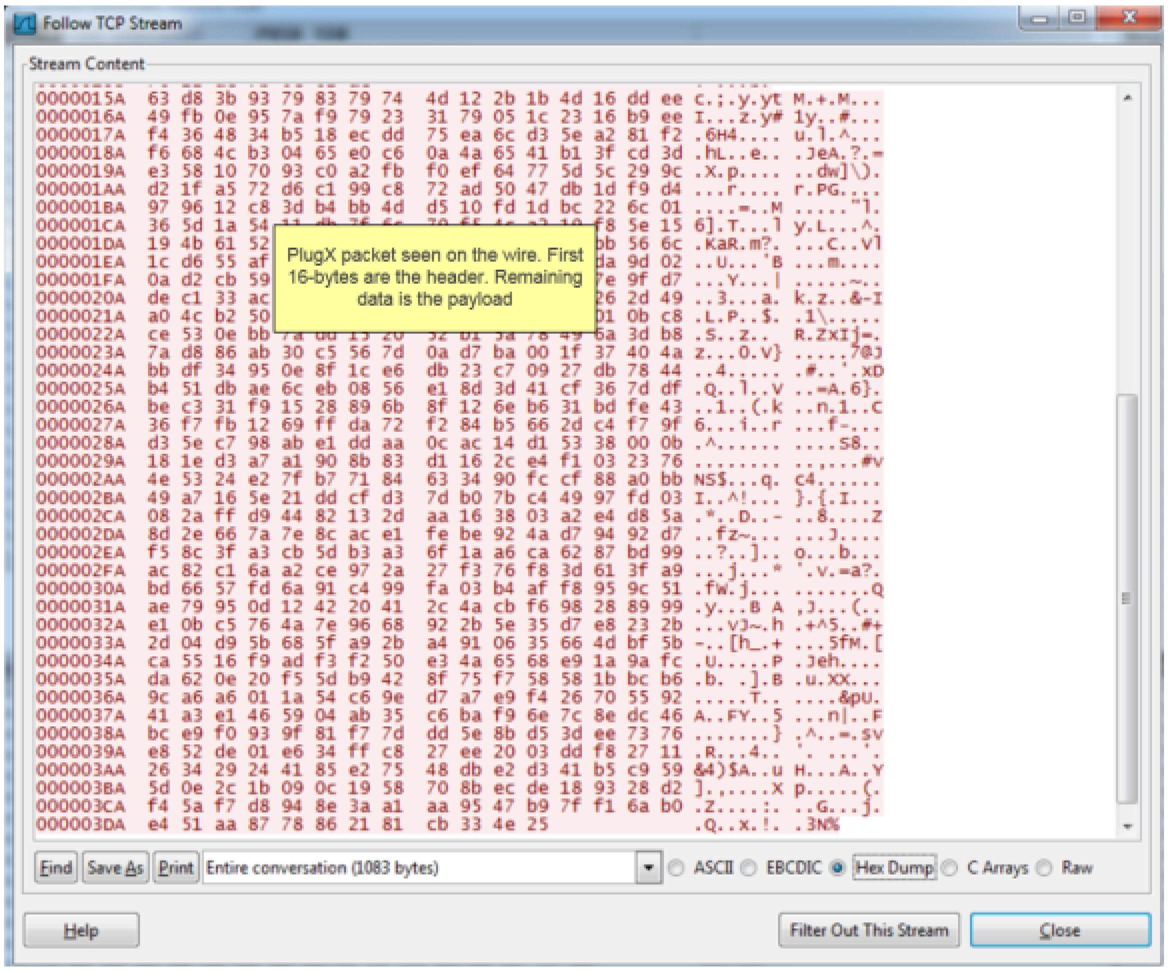
```
Address   Hex dump
03E4FD44  8C 02 00 00 28 5F BF 03 60 FD E4 03 3C 91 13 00
03E4FD54  28 5F BF 03 8C 02 00 00 F8 44 58 00 84 FD E4 03
03E4FD64  2D 89 13 00 F8 44 58 00 28 5F BF 03
03E4FD74  28 5F BF 03 28 5F BF 03 10 DE 52 00
03E4FD84  C4 FD E4 03 92 8C 13 00 28 5F BF 03
03E4FD94  B8 FD E4 03 30 75 00 00 28 5F BF 03
03E4FDA4  2E 07 00 00 10 08 13 20 03 70 00 00
```

pBuffers contains a pointer to an array that contains the size and the location of the data to be sent on the wire



```
03E4FD20  00139C33  ┌CALL to WSASend from 00139C31
03E4FD24  00000520  │Socket = 520
03E4FD28  03E4FD44  │pBuffers = 03E4FD44
03E4FD2C  00000001  │nBuffers = 1
03E4FD30  03E4FD54  │pBytesSent = 03E4FD54
03E4FD34  00000000  │Flags = 0
03E4FD38  00584514  │pOverlapped = 00584514
03E4FD3C  00000000  └Callback = NULL
```

Capturing the traffic, we can observe the same data.



Follow TCP Stream

Stream Content

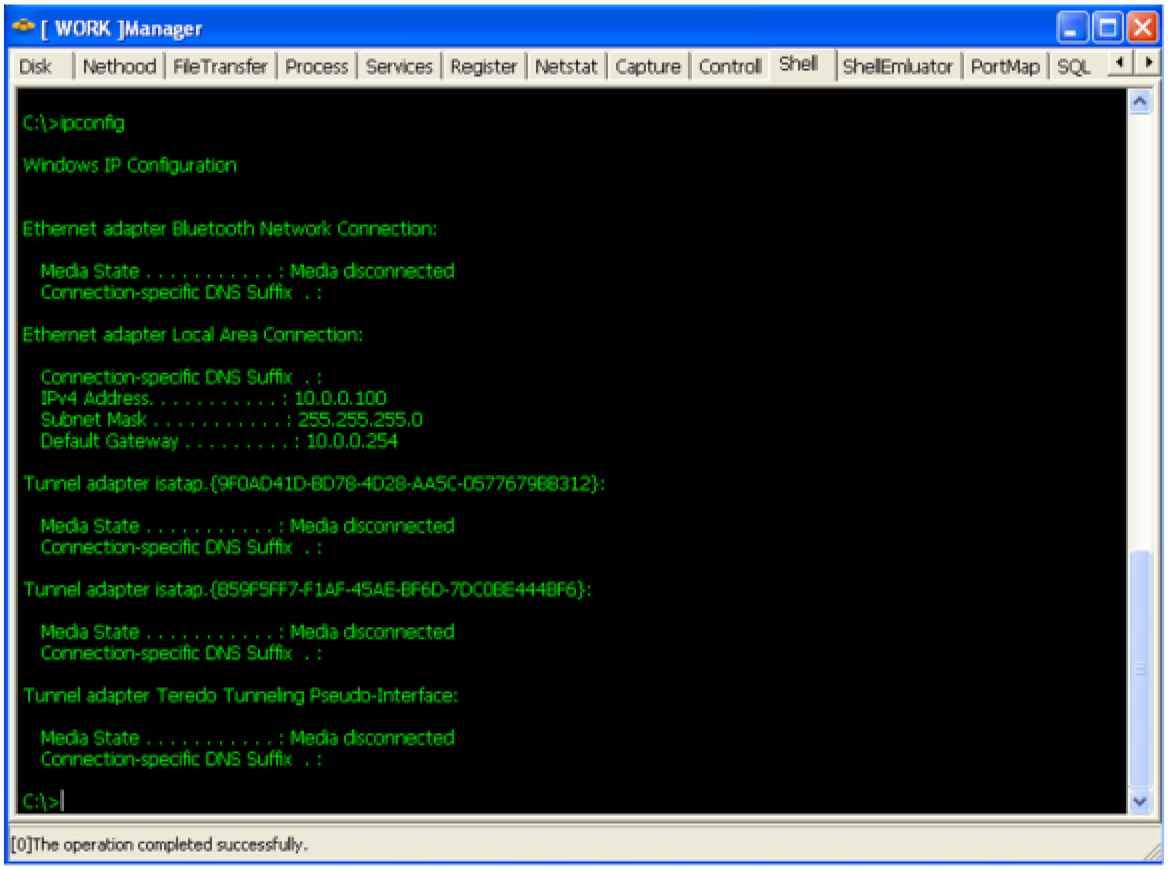
```
0000015A  63 d8 3b 93 79 83 79 74  4d 12 2b 1b 4d 16 dd ee c.;.y.yt M.+.M...
0000016A  49 fb 0e 95 7a f9 79 23  31 79 05 1c 23 16 b9 ee I...z.y# 1y..#...
0000017A  f4 36 48 34 b5 18 ec dd  75 ea 6c d3 5e a2 81 f2 .6H4.... u.l.^...
0000018A  f6 68 4c b3 04 65 e0 c6  0a 4a 65 41 b1 3f cd 3d .hL..e.. .JeA.?.=
0000019A  e3 58 10 70 93 c0 a2 fb  f0 ef 64 77 5d 5c 29 9c .X.p.... ..dw]\).
000001AA  d2 1f a5 72 d6 c1 99 c8  72 ad 50 47 db 1d f9 d4 ...r.... r.PG....
000001BA  97 96 12 c8 3d b4 bb 4d  d5 10 fd 1d bc 22 6c 01 ....=..M ....."l.
000001CA  36 5d 1a 54                            f8 5e 15 6].T...l y.L...^.
000001DA  19 4b 61 52                            bb 56 6c .KaR.m?. ...C..Vl
000001EA  1c d6 55 af                            da 9d 02 ..U...'B ...m....
000001FA  0a d2 cb 59                            7e 9f d7 ...Y...| ......~..
0000020A  de c1 33 ac                            26 2d 49 ..3...a. k.z..&-I
0000021A  a0 4c b2 50                            01 0b c8 .L.P..$. .1\.....
0000022A  ce 53 0e bb                            6a 3d b8 .S..z..  R.ZxIj=.
0000023A  7a d8 86 ab 30 c5 56 7d  0a d7 ba 00 1f 37 40 4a z...0.V} .....7@J
0000024A  bb df 34 95 0e 8f 1c e6  db 23 c7 09 27 db 78 44 ..4..... .#..'.xD
0000025A  b4 51 db ae 6c eb 08 56  e1 8d 3d 41 cf 36 7d df .Q..l..V ..=A.6}.
0000026A  be c3 31 f9 15 28 89 6b  8f 12 6e b6 31 bd fe 43 ..1..(.k ..n.1..C
0000027A  36 f7 fb 12 69 ff da 72  f2 84 b5 66 2d c4 f7 9f 6...i..r ...f-...
0000028A  d3 5e c7 98 ab e1 dd aa  0c ac 14 d1 53 38 00 0b .^...... ....S8..
0000029A  18 1e d3 a7 a1 90 8b 83  d1 16 2c e4 f1 03 23 76 ........ ..,...#v
000002AA  4e 53 24 e2 7f b7 71 84  63 34 90 fc cf 88 a0 bb NS$...q. c4......
000002BA  49 a7 16 5e 21 dd cf d3  7d b0 7b c4 49 97 fd 03 I..^!... }.{.I...
000002CA  08 2a ff d9 44 82 13 2d  aa 16 38 03 a2 e4 d8 5a .*..D..- ..8....Z
000002DA  8d 2e 66 7a 7e 8c ac e1  fe be 92 4a d7 94 92 d7 ..fz~... ...J....
000002EA  f5 8c 3f a3 cb 5d b3 a3  6f 1a a6 ca 62 87 bd 99 ..?..].. o...b...
000002FA  ac 82 c1 6a a2 ce 97 2a  27 f3 76 f8 3d 61 3f a9 ...j...* '.v.=a?.
0000030A  bd 66 57 fd 6a 91 c4 99  fa 03 b4 af f8 95 9c 51 .fW.j... .......Q
0000031A  ae 79 95 0d 12 42 20 41  2c 4a cb f6 98 28 89 99 .y...B A ,J...(..
0000032A  e1 0b c5 76 4a 7e 96 68  92 2b 5e 35 d7 e8 23 2b ...vJ~.h .+^5..#+
0000033A  2d 04 d9 5b 68 5f a9 2b  a4 91 06 35 66 4d bf 5b -..[h_.+ ...5fM.[
0000034A  ca 55 16 f9 ad f3 f2 50  e3 4a 65 68 e9 1a 9a fc .U.....P .Jeh....
0000035A  da 62 0e 20 f5 5d b9 42  8f 75 f7 58 58 1b bc b6 .b. .].B .u.XX...
0000036A  9c a6 a6 01 1a 54 c6 9e  d7 a7 e9 f4 26 70 55 92 .....T.. ....&pU.
0000037A  41 a3 e1 46 59 04 ab 35  c6 ba f9 6e 7c 8e dc 46 A..FY..5 ...n|..F
0000038A  bc e9 f0 93 9f 81 f7 7d  dd 5e 8b d5 3d ee 73 76 .......} .^..=.sv
0000039A  e8 52 de 01 e6 34 ff c8  27 ee 20 03 dd f8 27 11 .R...4.. '. ...'.
000003AA  26 34 29 24 41 85 e2 75  48 db e2 d3 41 b5 c9 59 &4)$A..u H...A..Y
000003BA  5d 0e 2c 1b 09 0c 19 58  70 8b ec de 18 93 28 d2 ].,....X p.....(.
000003CA  f4 5a f7 d8 94 8e 3a a1  aa 95 47 b9 7f f1 6a b0 .Z....:. ..G...j.
000003DA  e4 51 aa 87 78 86 21 81  cb 33 4e 25             .Q..x.!. .3N%
```

Entire conversation (1083 bytes)

ASCII [ ] EBCDIC [ ] Hex Dump [✓] C Arrays [ ] Raw [ ]

PlugX packet seen on the wire. First 16-bytes are the header. Remaining data is the payload
Find
Save As
Print
Help
Filter Out This Stream
Close

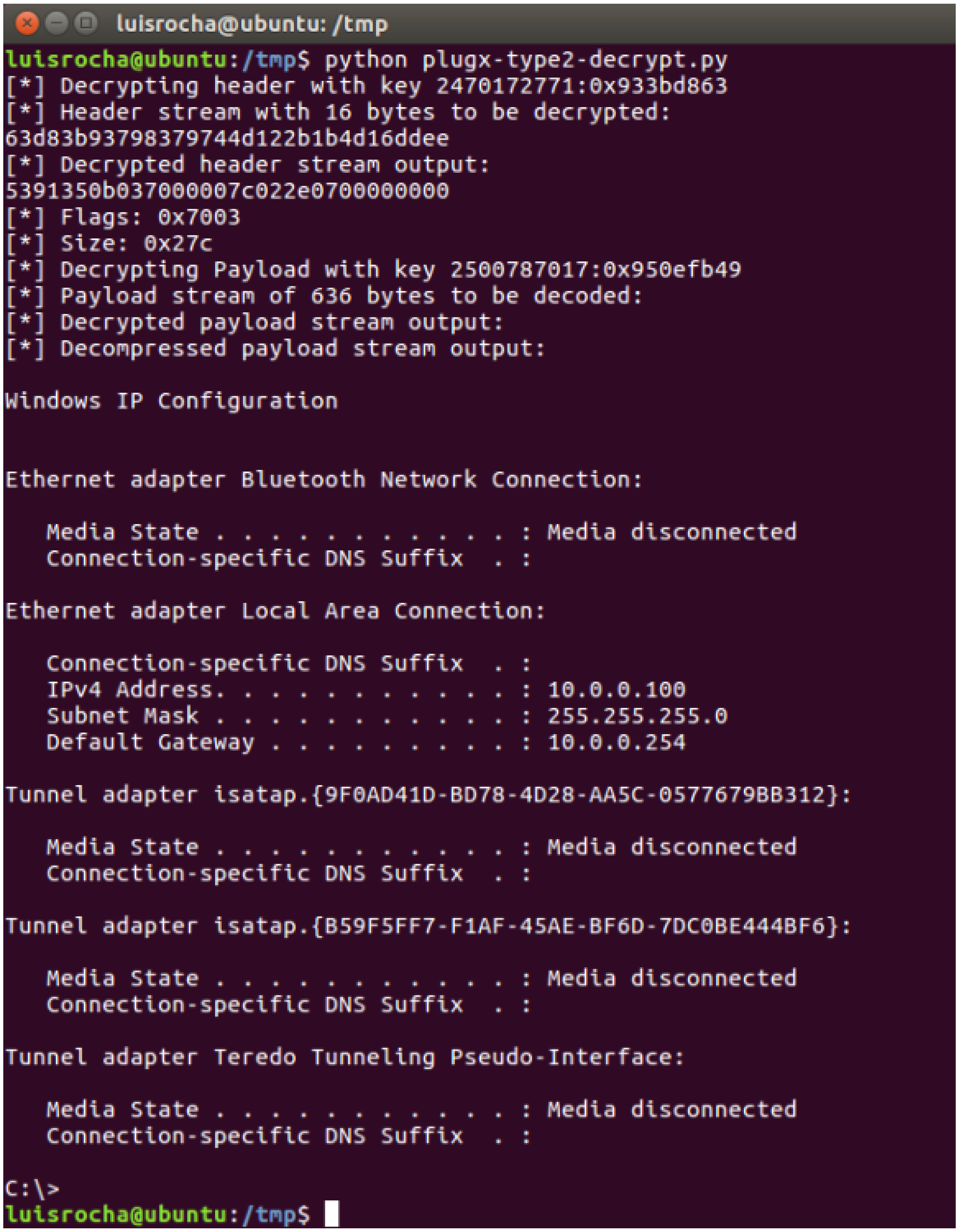
On the controller side, when the packet arrives, the header will be decoded and then the payload will be decoded and decompressed. Finally, the output is showed to the operator.


[WORK]Manager
Disk | Nethood | FileTransfer | Process | Services | Register | Netstat | Capture | Controll | Shell | ShellEmluator | PortMap | SQL

```
C:\>ipconfig

Windows IP Configuration


Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . :
   IPv4 Address. . . . . . . . . . . : 10.0.0.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.0.0.254

Tunnel adapter isatap.{9F0AD41D-BD78-4D28-AA5C-057767988312}:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Tunnel adapter isatap.{B59F5FF7-F1AF-45AE-BF6D-7DC0BE4448F6}:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Tunnel adapter Teredo Tunneling Pseudo-Interface:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

C:\>
```

```
[0]The operation completed successfully.
```

Now that we started to understand how C2 traffic is handled, we can capture it and decode it. Kyle Creyts has created a PlugX decoder that supports PCAP's. The decoder supports decryption of PlugX Type I.But Fabien Perigaud reversed the Type II algorithm and implemented it in python. If we combine Kyle's work with the work from Takahiro Haruyama and Fabien Perigaud we could create a PCAP parser to extract PlugX Type II and Type III. Below illustrates a proof-of-concept for this exercise against 1 packet. We captured the traffic and then used a small python script to decrypt a packet. No dependencies on Windows because it uses the herrcore's standalone LZNT1 implementation that is based on the one from the ChopShop protocol analysis and decoder framework by MITRE.


```
luisrocha@ubuntu: /tmp
luisrocha@ubuntu:/tmp$ python plugx-type2-decrypt.py
[*] Decrypting header with key 2470172771:0x933bd863
[*] Header stream with 16 bytes to be decrypted:
63d83b93798379744d122b1b4d16ddee
[*] Decrypted header stream output:
5391350b037000007c022e0700000000
[*] Flags: 0x7003
[*] Size: 0x27c
[*] Decrypting Payload with key 2500787017:0x950efb49
[*] Payload stream of 636 bytes to be decoded:
[*] Decrypted payload stream output:
[*] Decompressed payload stream output:

Windows IP Configuration


Ethernet adapter Bluetooth Network Connection:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter Local Area Connection:

   Connection-specific DNS Suffix  . :
   IPv4 Address. . . . . . . . . . . : 10.0.0.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 10.0.0.254

Tunnel adapter isatap.{9F0AD41D-BD78-4D28-AA5C-0577679BB312}:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Tunnel adapter isatap.{B59F5FF7-F1AF-45AE-BF6D-7DC0BE444BF6}:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Tunnel adapter Teredo Tunneling Pseudo-Interface:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

C:\>
luisrocha@ubuntu:/tmp$ 
```

That’s it for today! We build a lab with a PlugX controller, got a view on its capabilities. Then we looked at the malware installation and debugged it in order to find and interpret some of its mechanics such as DLL search order hijacking, obfuscated shellcode, persistence mechanism and process hollowing. Then, we used a readily available parser to dump its configuration from memory. Finally, we briefly looked the way the malware communicates with the C2 and created a small script to decode the traffic. Now, with such environment ready, in a controlled and isolated lab, we can further simulate different tools and techniques and observe how an attacker would operate compromised systems. Then we can learn, practice at our own pace and look behind the scenes to better understand attack methods and ideally find and implement countermeasures.

References:
Analysis of a PlugX malware variant used for targeted attacks by CRCL.lu
Operation Cloud Hopper by PWC
PlugX Payload Extraction by Kevin O’Reilly
Other than the authors and articles cited troughout the article, a fantastic compilation about PlugX articles and papers since 2011 is available here.

Credits: Thanks to Michael Bailey who showed me new techniques on how to deal with shellcode which I will likely cover on a post soon.