Malware Analysis – PlugX

By By Luis Rocha

Published: 2018-02-04 · Archived: 2026-04-02 11:31:31 UTC

[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, l 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.

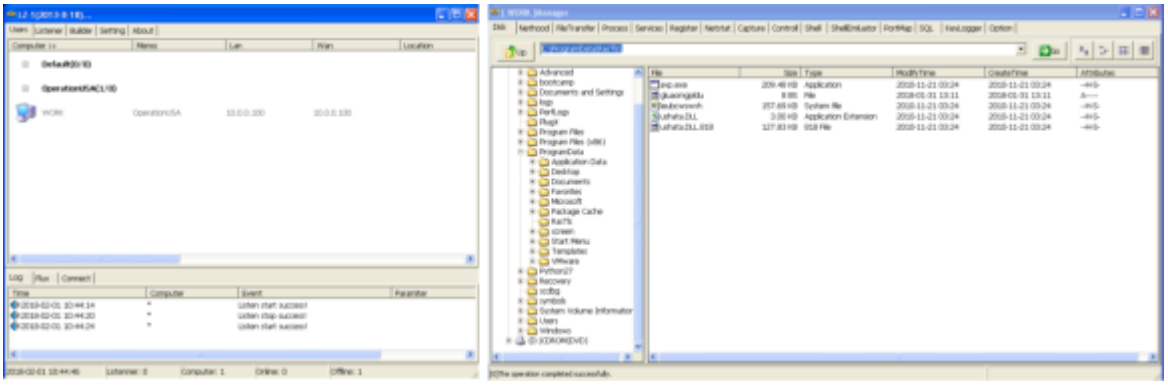
The patched builder, MD5 6aad032a084de893b0e8184c17f0376a, is an English version, from Q3 2013, of the featured-rich and modular command & control interface for PlugX that allows an operator to:

- 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.

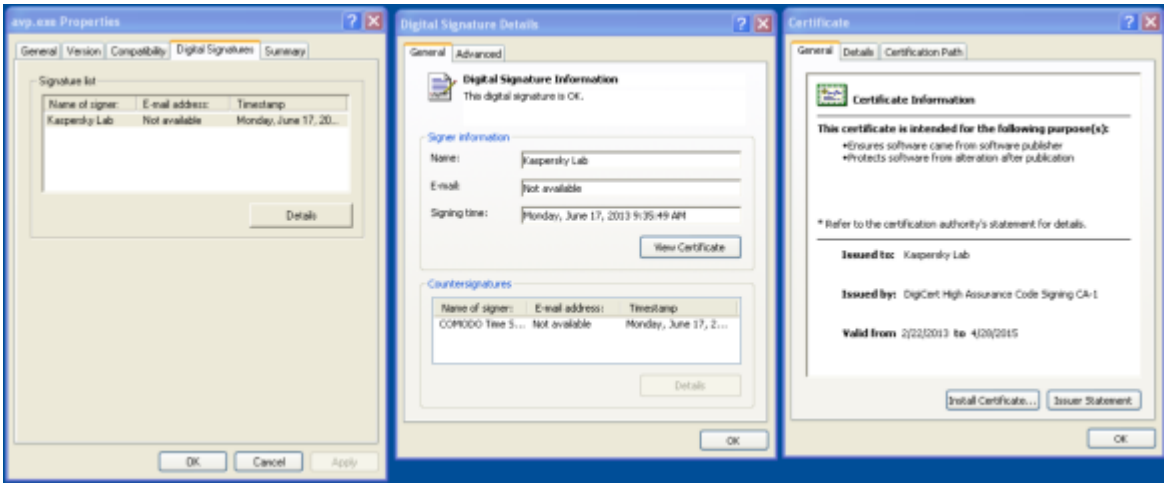
Then for each compromised system, the operator has extensive capabilities to interact with the systems over the controller that includes the following modules:

- 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.



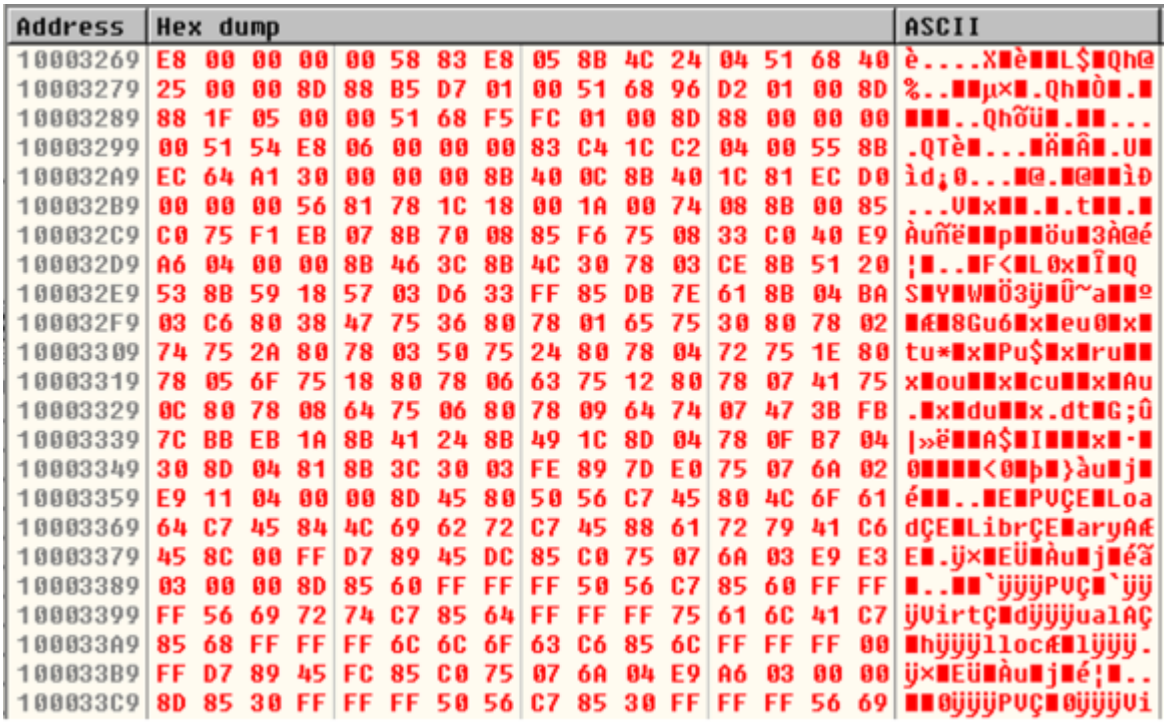
So, let's look at the mechanics of what happens when the self-extracting archive is executed. The three files are extracted to a temporary directory and "avp.exe" is executed. The "avp.exe" when executed will load "ushata.dll" from the running directory due to the DLL search order hijacking using Kernel32.LoadLibrary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

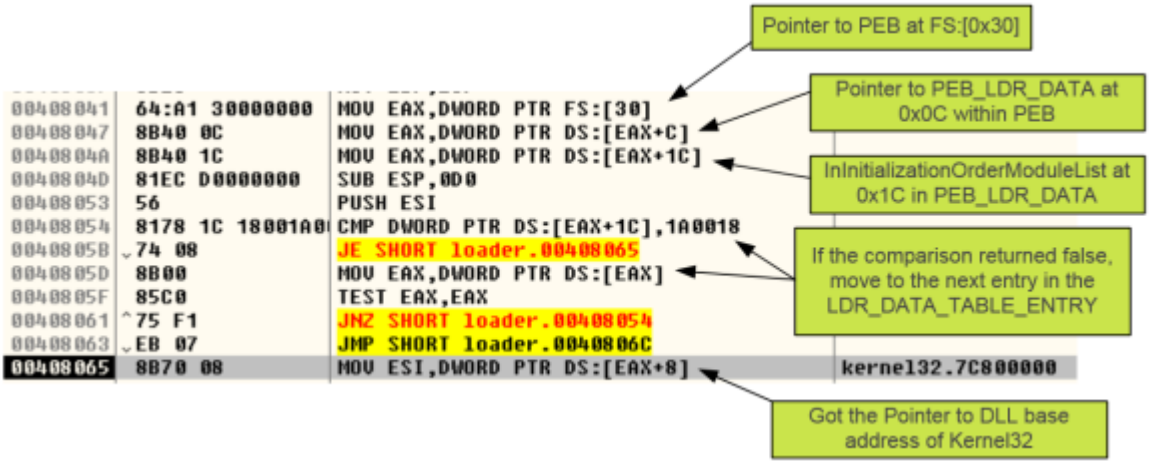
```
Address   Hex dump                                          ASCII
10003008  7C 03 7D 01 E8 81 C3 07 74 DA 86 BB BE 50 F3 F8  |■}■è■Ã■tÚ■»¾Pó■
10003018  F7 C7 C6 60 5E DA F7 C7 6E 36 BA 9B 4B 81 C9 3F  ÷ÇÆ`^Ú÷Çn6º■K■É?
10003028  C4 81 D0 E9 01 00 00 00 E9 81 E1 47 39 4F F0 E9  Ä■Ðé■...é■áG9Oðé
10003038  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^ç■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■Ëì Ì■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.

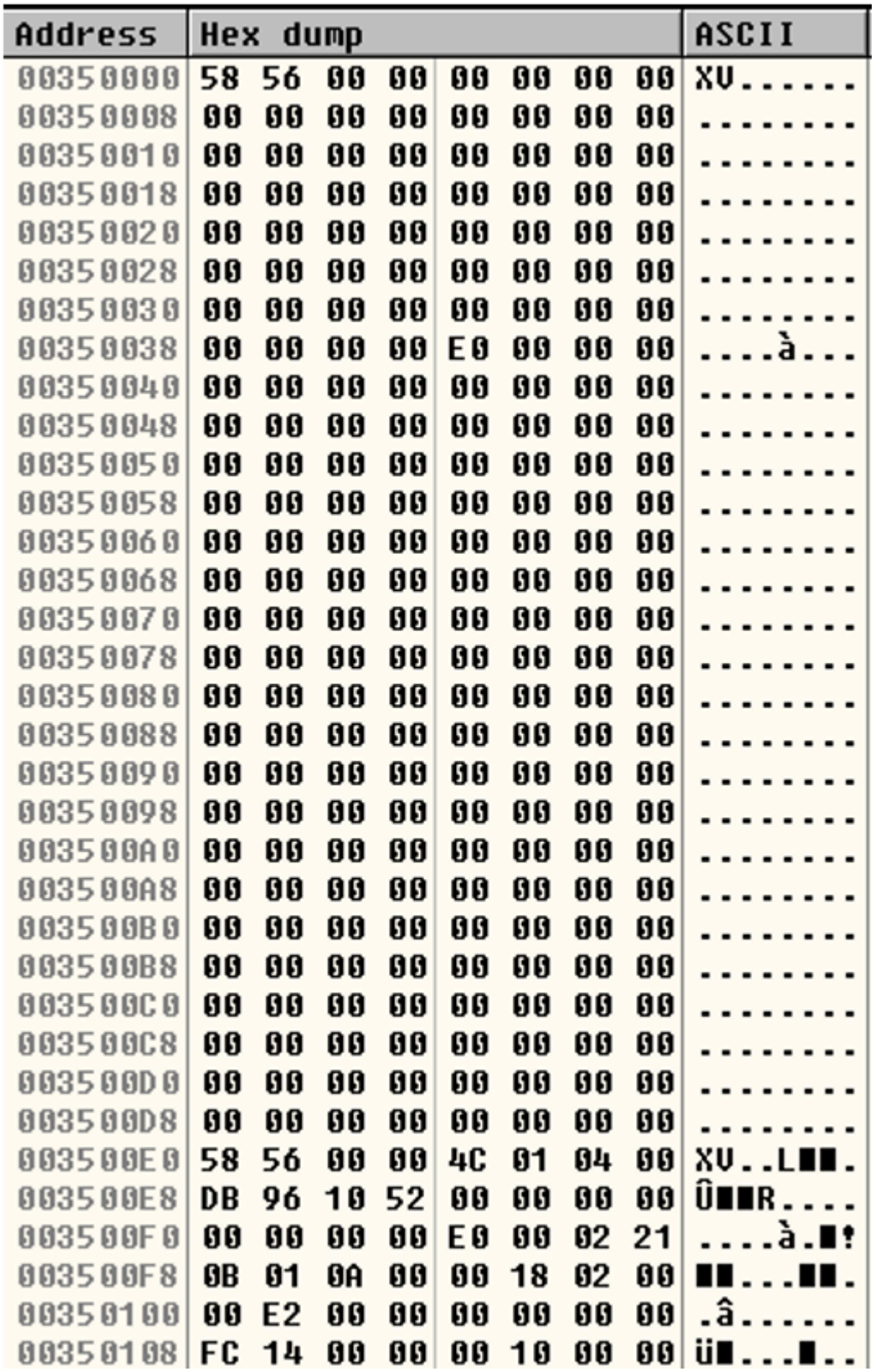


Address	Hex dump	ASCII
10003269	E8 00 00 00 00 58 83 E8 05 8B 4C 24 04 51 68 40	è....X■è■■L$■Qh@
10003279	25 00 00 8D 88 B5 D7 01 00 51 68 96 D2 01 00 8D	%..■■µ×■.Qh■Ò■.■
10003289	88 1F 05 00 00 51 68 F5 FC 01 00 8D 88 00 00 00	■■■..Qhõü■.■■...
10003299	00 51 54 E8 06 00 00 00 83 C4 1C C2 04 00 55 8B	.QTè■...■Ä■Â■.U■
100032A9	EC 64 A1 30 00 00 00 8B 40 0C 8B 40 1C 81 EC D0	ìd¡0...■@.■@■■ìÐ
100032B9	00 00 00 56 81 78 1C 18 00 1A 00 74 08 8B 00 85	...V■x■■.■.t■■.■
100032C9	C0 75 F1 EB 07 8B 70 08 85 F6 75 08 33 C0 40 E9	Àuñë■■p■■öu■3À@é
100032D9	A6 04 00 00 8B 46 3C 8B 4C 30 78 03 CE 8B 51 20	¦■..■F<■L0x■Î■Q
100032E9	53 8B 59 18 57 03 D6 33 FF 85 DB 7E 61 8B 04 BA	S■Y■W■Ö3ÿ■Û~a■■º
100032F9	03 C6 80 38 47 75 36 80 78 01 65 75 30 80 78 02	■Æ■8Gu6■x■eu0■x■
10003309	74 75 2A 80 78 03 50 75 24 80 78 04 72 75 1E 80	tu*■x■Pu$■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.



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.

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

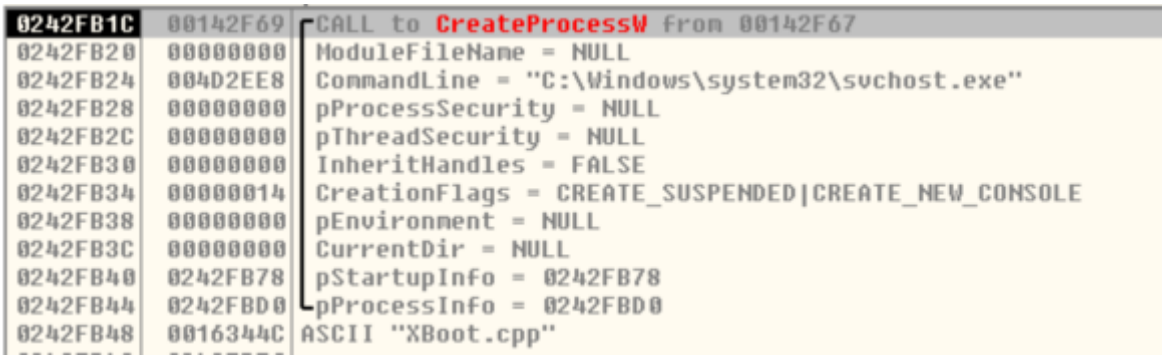
```
0225FBA0  002B4567  CALL to CreateServiceW from 002B4565
0225FBA4  00523F68  hManager = 00523F68
0225FBA8  002DCB98  ServiceName = "RasTls"
0225FBAC  002DCD98  DisplayName = "RasTls"
0225FBB0  000F01FF  DesiredAccess = SERVICE_ALL_ACCESS
0225FBB4  00000110  ServiceType = SERVICE_WIN32_OWN_PROCESS|SERVICE_INTERACTIVE_PROCESS
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.

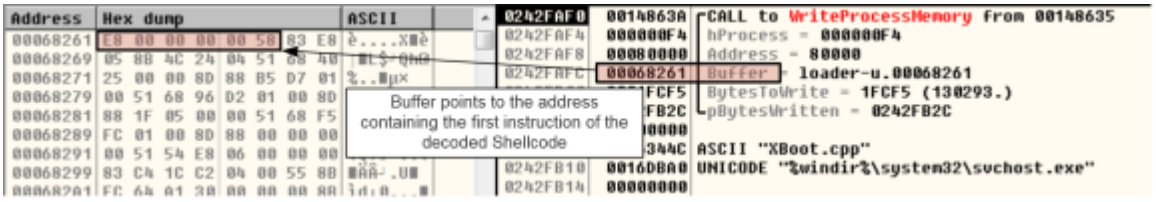
```
0233F78C  0011E571  CALL to RegCreateKeyExW from 0011E56F
0233F790  80000000  hKey = HKEY_CLASSES_ROOT
0233F794  0013D19C  Subkey = "Software\Microsoft\Windows\CurrentVersion\Run"
0233F798  00000000  Reserved = 0
0233F79C  00000000  Class = NULL
0233F7A0  00000000  Options = REG_OPTION_NON_VOLATILE
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

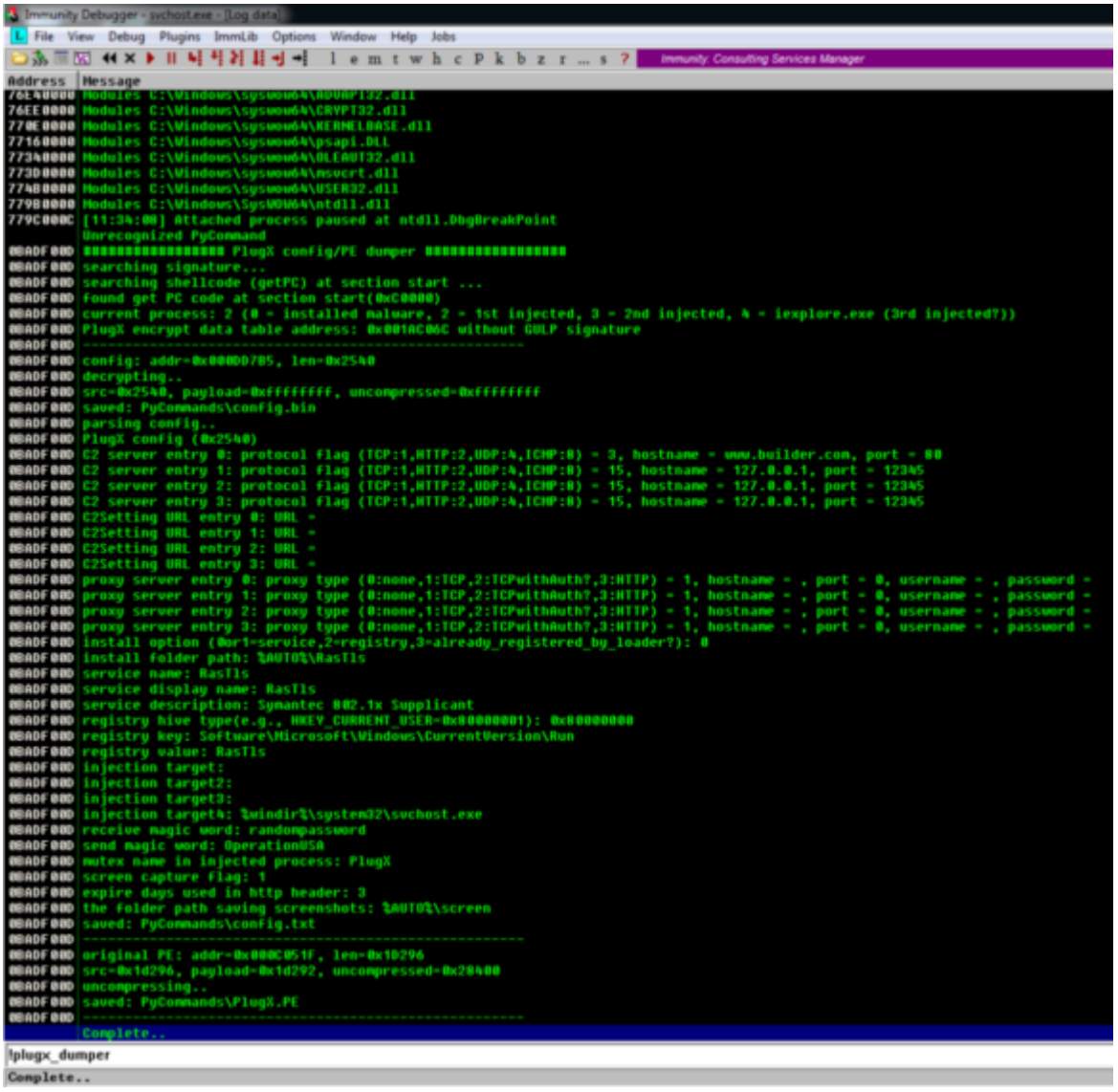
(also interesting that during execution the malware outputs debug messages about what is happening using the OutputDebugString API. This messages could be viewed with DebugView from SysInternals). The malicious code completes its mission by starting a new instance of “svchost.exe” and then injects the 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.



and then uses WriteProcessMemory API to inject the malicious payload



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

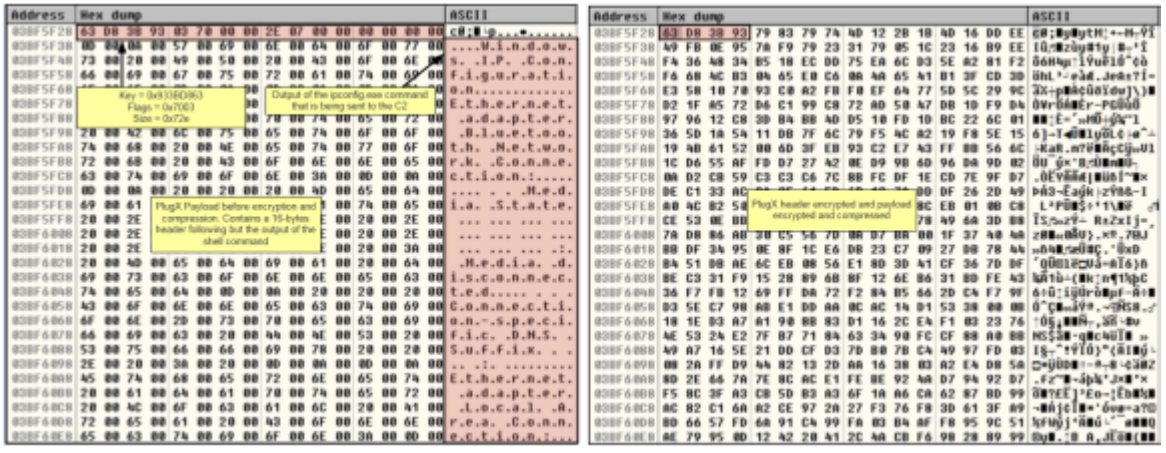


We can see that this parser is able to find the injected shellcode, decode its configuration and all the settings an attacker would set on the builder and also dump the injected DLL which contains the core functionality of the malware.

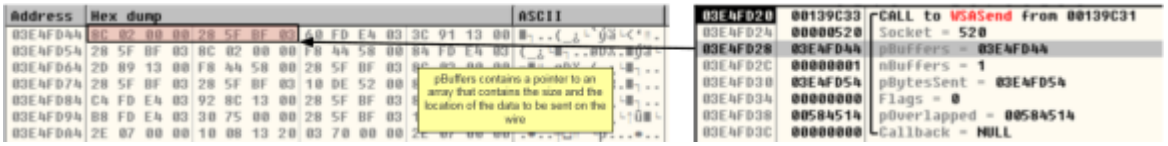
In terms of networking, as observed in the PlugX controller, the malware can be configured to speak with a controller using several network protocols. In this case we configured it to speak using HTTP on port 80. The network traffic contains a 16-byte header followed by a payload. The header is encoded 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
```

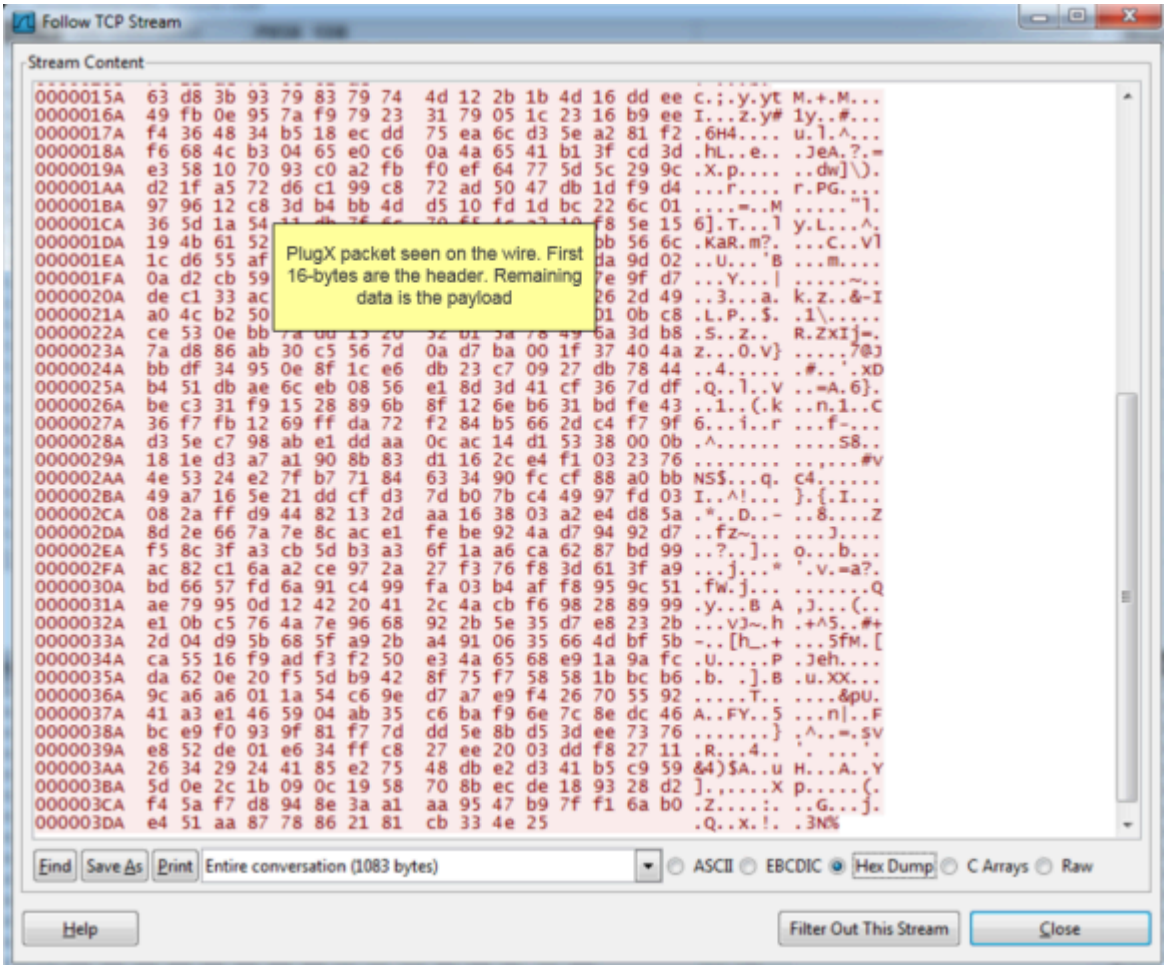
So, from a debugger view, with the right breakpoints we could start to observe what is happening. In the picture below, on the left-hand side it shows the packet before encoding and compression. It contains a 16-byte header, where the first 4-bytes are the key for the custom encoding routine. The next 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.



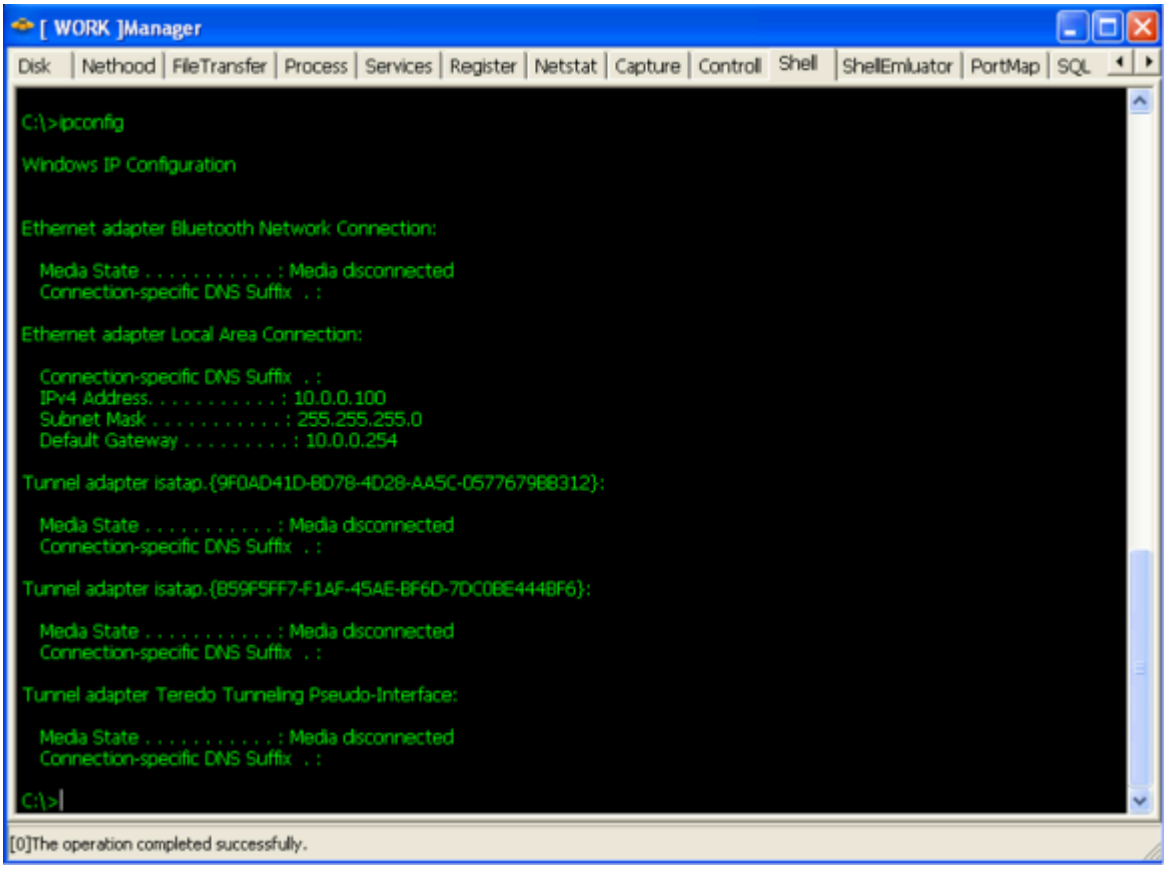
Then, the malware uses WSASend API to send the traffic.



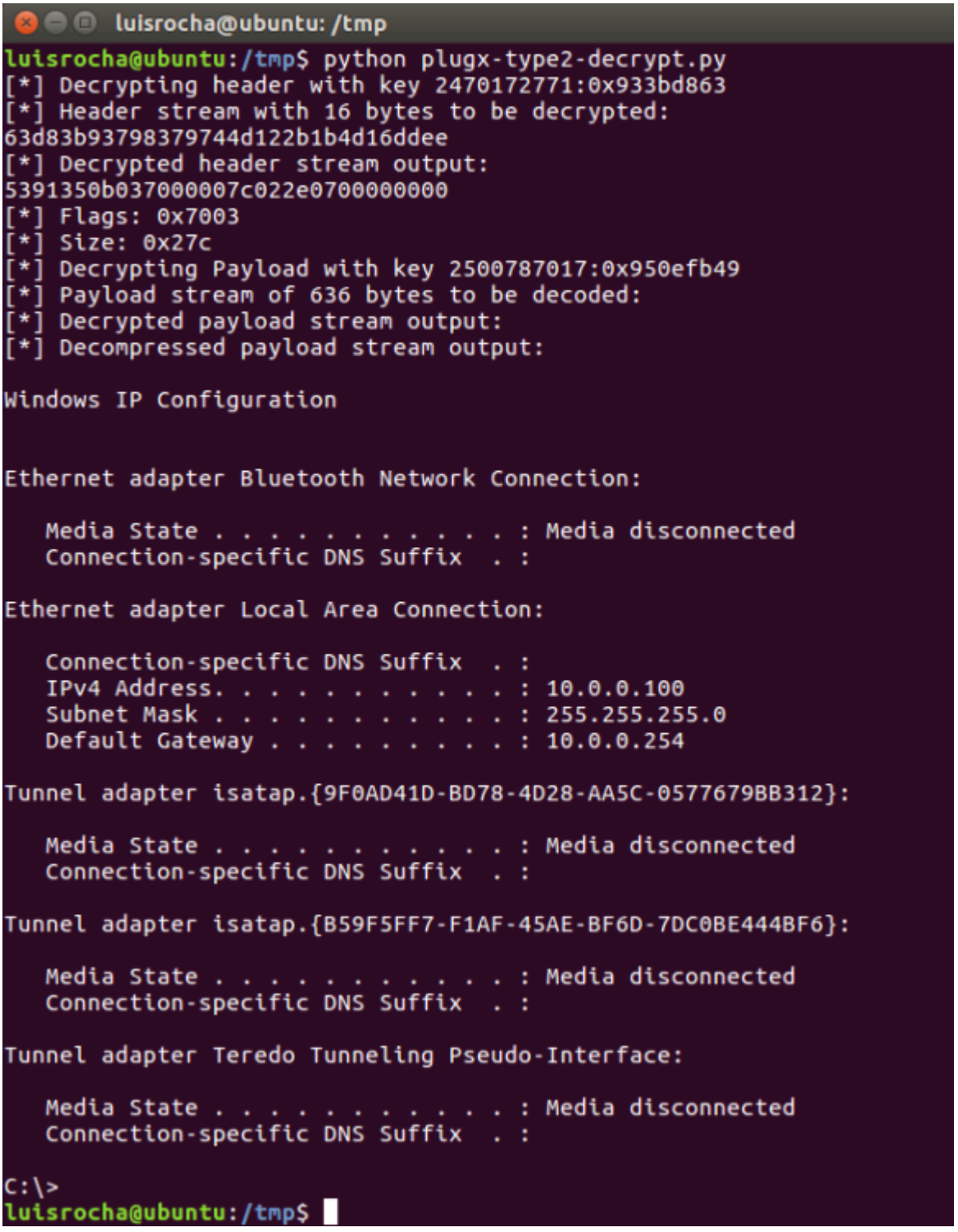
Capturing the traffic, we can observe the same data.



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.



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.



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.
