

Malware Analysis – PlugX

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, 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 referred 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.

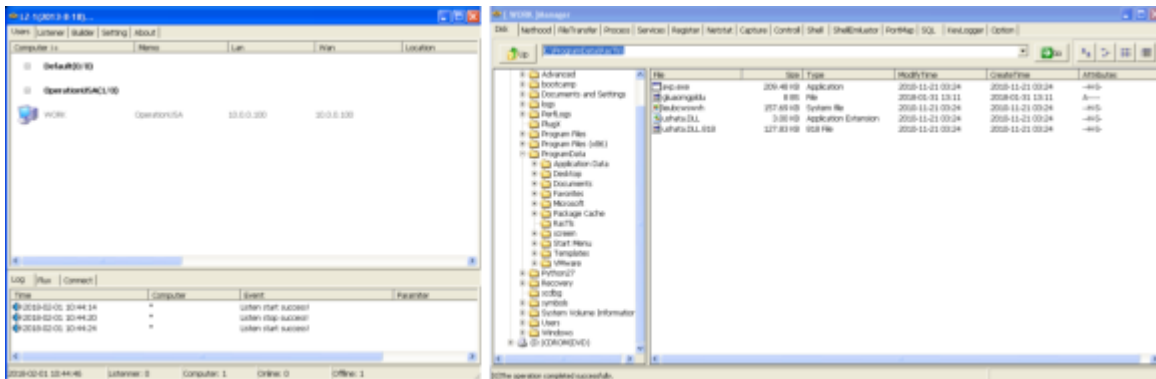
The patched builder, MD5 6aad032a084de893b0e8184c17f0376a, is an English version, from Q3 2013, of the featured-rich and modular command & control interface for PlugX that allows an operator to:

- 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 compromised systems per campaign.

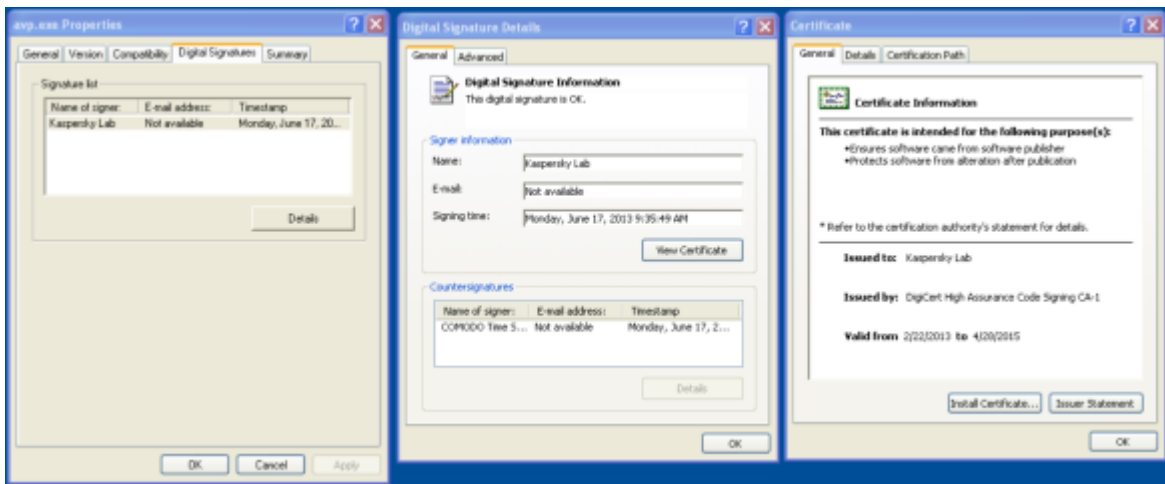
Then for each compromised system, the operator has extensive capabilities to interact with the systems over the controller that includes the following modules:

- 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	[]èÀÁtú»¼Póø
10003018	F7 C7 C6 60 5E DA F7 C7 6E 36 BA 9B 4B 81 C9 3F	÷ÇÆ^ú÷çñó»KÑÉ?
10003028	C4 81 D0 E9 01 00 00 00 E9 81 E1 47 39 4F F0 E9	ÀÑÐé»...é»áG90ðé
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~»ééé»..
10003058	00 E9 81 CB BE 19 92 C1 E9 01 00 00 00 E8 81 C9	.é»È»'Áé»...è»É
10003068	AE 78 EC 5D E8 00 00 00 00 4B 71 03 70 01 E9 49	@xijè»...KqppéI
10003078	F7 C7 BA 84 1F 4C 43 81 F9 03 EA E3 CB 5E E9 01	÷Ç»»LCù»é»É^é»
10003088	00 00 00 E9 7A 03 7B 01 E8 81 E9 5E A2 93 69 81	...é»z»{»è»è^ç»i»
10003098	F3 F1 B8 B1 DA E9 01 00 00 00 E9 81 C1 EC 5F FB	óñ ±úé»...é»Áì Ò
100030A8	86 81 C2 3E F0 D2 06 81 C2 01 65 9B BC 7A 03 7B	»»Á»>ð0»»Á»e»%z»{
100030B8	01 E9 81 FA 48 3C 57 69 81 E2 A5 02 46 3A 81 CB	éé»úH<Wi»â»¥»F:»È
100030C8	60 07 8B 91 47 E9 01 00 00 00 E9 81 EE 69 00 00	`»»'Gé»...é»i»..
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	..è»z»{»{»é»...é»ó
100030F8	72 DF 87 81 E1 11 B0 CE 4D E9 01 00 00 00 E9 81	r»»»á»»°îMé»...é»
10003108	C6 61 02 00 00 7D 03 7C 01 E8 81 F3 B8 47 5C 03	æa»..}»J»è»ó»G\»
10003118	81 F9 9C 94 29 60 81 CF 15 51 15 16 B8 B5 D7 01	»ù»»»)»î»Q»»»µ»x»
10003128	00 42 4B 81 CB EC CC 18 47 81 C7 08 36 58 BE 81	.BK»Èì»î»G»ç»6»X»»
10003138	FB 9C AE 0D 52 81 FA 90 46 7D 54 81 E7 A4 E3 B5	û»@.R»ú»F}T»ç»»âµ
10003148	9C 7A 03 7B 01 E9 4B 47 E9 01 00 00 00 E8 E9 01	z»z»{»é»KGé»...èé»
10003158	00 00 00 E8 7A 03 7B 01 74 E9 01 00 00 00 E8 81	...è»z»{»t»é»...è»
10003168	E9 DC D5 2C 57 E9 01 00 00 00 E9 E9 01 00 00 00	éúõ,Wé»...éé»...

Address	Hex dump	ASCII
00350000	58 56 00 00 00 00 00 00	XU.....
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 00à..
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	XU..L■■.
003500E8	DB 96 10 52 00 00 00 00	Û■■R....
003500F0	00 00 00 00 E0 00 02 21à.■?
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 9xM+ME
01D8F8A4 F7 39 78 8A 2B 89 CB 01 9xM+ME
01D8F8AC 58 9B 7A 8A 2B 89 CB 01 Xz+ME
01 NTFS standard information attribute
01 timestamps are manipulated to look
01 like the ones from ntdll.dll
01
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"
0225FB80 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\rjowfhxnzmdknsixt” 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	0242FAF0	0014863A	CALL to WriteProcessMemory from 00148635
00068261	E8 00 00 00 00 58 83 E8	è....Xè	0242FAF4	000000F4	hProcess = 000000F4
00068269	05 00 4C 24 04 51 88 40	ML\$ QH0	0242FAF8	00080000	Address = 80000
00068271	25 00 00 00 00 88 05 07 01	%..µpx	0242FAFC	00068261	Buffer = loader-u.00068261
00068279	00 51 68 96 02 01 00 80			FCF5	BytesToWrite = 1FCF5 (130293.)
00068281	88 1F 05 00 00 51 68 F5			FB2C	pBytesWritten = 0242FB2C
00068289	FC 01 00 00 00 00 00 00			0000	
00068291	00 51 54 E8 06 00 00 00			344C	ASCII "XBoot.cpp"
00068299	83 C4 1C C2 04 00 55 80	RR- .UM	0242FB10	00160000	UNICODE "%windir%\system32\svchost.exe"
000682A1	FC 6A 01 30 00 00 00 00	rd. . .	0242FB14	00000000	

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\System32\USER32.dll
76EE0000 Modules C:\Windows\System32\CRYP132.dll
770E0000 Modules C:\Windows\System32\USERHEL005F.dll
77160000 Modules C:\Windows\System32\Apsapi.dll
77340000 Modules C:\Windows\System32\USER32.dll
773D0000 Modules C:\Windows\System32\USER32.dll
774B0000 Modules C:\Windows\System32\USER32.dll
779B0000 Modules C:\Windows\System32\USER32.dll
779C0000 [11:24:00] Attached process paused at ntdll.DbgBreakPoint
Unrecognized PyCommand
00ADF000 ===== PlugX config/PE dumper =====
00ADF000 searching signature...
00ADF000 searching shellcode (getPC) at section start ...
00ADF000 found get PC code at section start(0xC0000)
00ADF000 current process: 2 (0 = installed malware, 2 = 1st injected, 3 = 2nd injected, 4 = iexplore.exe (3rd injected))
00ADF000 PlugX encrypt data table address: 0x0010C06C without GMLP signature
00ADF000 -----
00ADF000 config: addr=0x00000785, len=0x2540
00ADF000 decrypting...
00ADF000 src=0x2540, payload=0xffffffff, uncompressed=0xffffffff
00ADF000 saved: PyCommands\config.bin
00ADF000 parsing config...
00ADF000 PlugX config (0x2540)
00ADF000 C2 server entry 0: protocol flag (TCP:1,HTTP:2,UDP:4,ICMP:8) = 3, hostname = www.builder.com, port = 80
00ADF000 C2 server entry 1: protocol flag (TCP:1,HTTP:2,UDP:4,ICMP:8) = 15, hostname = 127.0.0.1, port = 12345
00ADF000 C2 server entry 2: protocol flag (TCP:1,HTTP:2,UDP:4,ICMP:8) = 15, hostname = 127.0.0.1, port = 12345
00ADF000 C2 server entry 3: protocol flag (TCP:1,HTTP:2,UDP:4,ICMP:8) = 15, hostname = 127.0.0.1, port = 12345
00ADF000 C2Setting URL entry 0: URL =
00ADF000 C2Setting URL entry 1: URL =
00ADF000 C2Setting URL entry 2: URL =
00ADF000 C2Setting URL entry 3: URL =
00ADF000 proxy server entry 0: proxy type (0:none,1:TCP,2:TCPwithAuth,3:HTTP) = 1, hostname = , port = 0, username = , password =
00ADF000 proxy server entry 1: proxy type (0:none,1:TCP,2:TCPwithAuth,3:HTTP) = 1, hostname = , port = 0, username = , password =
00ADF000 proxy server entry 2: proxy type (0:none,1:TCP,2:TCPwithAuth,3:HTTP) = 1, hostname = , port = 0, username = , password =
00ADF000 proxy server entry 3: proxy type (0:none,1:TCP,2:TCPwithAuth,3:HTTP) = 1, hostname = , port = 0, username = , password =
00ADF000 install option (0=1-service,2=registry,3=already_registered_by_loader?): 0
00ADF000 install folder path: %AOTI%\Rast1s
00ADF000 service name: Rast1s
00ADF000 service display name: Rast1s
00ADF000 service description: Symantec 002.1x Supplicant
00ADF000 registry hive type(e.g., HKEY_CURRENT_USER=0x00000001): 0x00000000
00ADF000 registry key: Software\Microsoft\Windows\CurrentVersion\Run
00ADF000 registry value: Rast1s
00ADF000 injection target:
00ADF000 injection target2:
00ADF000 injection target3:
00ADF000 injection target4: %windir%\system32\svchost.exe
00ADF000 receive magic word: randompassword
00ADF000 send magic word: Operation05A
00ADF000 mutex name in injected process: PlugX
00ADF000 screen capture flag: 1
00ADF000 expire days used in http header: 3
00ADF000 the folder path saving screenshots: %AOTI%\screen
00ADF000 saved: PyCommands\config.txt
00ADF000 -----
00ADF000 original PE: addr=0x000C051F, len=0x10296
00ADF000 src=0x10296, payload=0x10292, uncompressed=0x28400
00ADF000 uncompressing...
00ADF000 saved: PyCommands\PlugX.PE
00ADF000 -----
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]
jnl    Decrypt
    
```

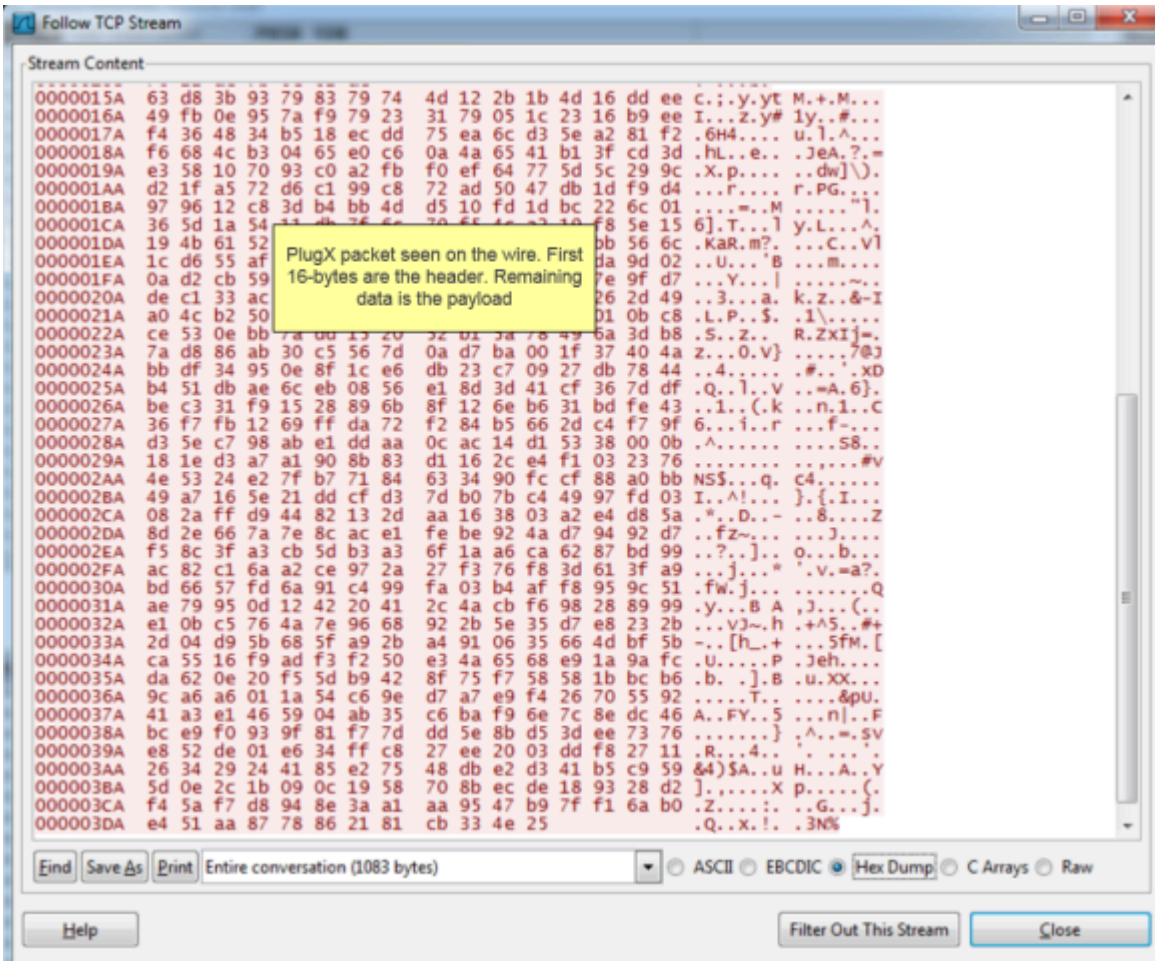
So, from a debugger view, with the right breakpoints we could start to observe what is happening. In the picture below, on the left-hand side it shows the packet before encoding and compression. It contains a 16-byte header, where the first 4-bytes are the key for the custom encoding routine. The next 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	Address	Hex dump	ASCII
000F5F20	43 08 38 93 03 78 00 00 2E 07 00 00 00 00 00 00	...M.L.n.d.o.o...	000F5F20	63 08 38 93 79 03 79 7A 4D 12 2B 1B AD 16 D0 EE	CR;RyRyTH:--H-VI
000F5F28	00 04 08 00 57 00 69 00 6E 00 6A 00 6F 00 77 00	...i.P..C.o.n...	000F5F28	49 FB 0E 95 78 F9 23 31 79 05 1C 23 16 09 EE	IQRzRyRyTj B.-i
000F5F30	73 00 20 00 49 00 50 00 20 00 43 00 6F 00 6E 00	F.i.g.u.r.a.t.i...	000F5F30	FA 36 A8 34 05 18 EC 00 75 EA 6C 03 5E 82 81 F2	60Hq:19uF16"0
000F5F38	66 00 69 00 67 00 75 00 72 00 61 00 74 00 69 00	O.o.n.....	000F5F38	F4 68 AC 03 04 65 E0 C0 0A 40 65 41 81 3F CD 3D	0Hl'-e3d.Je0e7i-
000F5F40	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	E.t.h.e.r.n.a.t...	000F5F38	E3 58 10 70 93 C0 A2 F0 F0 EF 64 77 5D 5C 29 9C	3x-p00000Y0j)1
000F5F48	00 70 00 74 00 05 00 72 00 00 00 00 00 00 00	a.d.a.p.t.e.r...	000F5F38	B2 1F 85 72 D6 C1 99 C8 72 A0 50 A7 D0 10 F9 D4	0Vr000Er-P000
000F5F50	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	O.b.i.u.e.t.o.o...	000F5F40	97 96 12 C8 30 04 00 A0 D5 10 F0 10 0C 22 6C 01	00;E="n0-0a"1
000F5F58	74 00 68 00 20 00 4E 00 65 00 74 00 77 00 6F 00	t.h..M.e.t.w.o.o...	000F5F40	36 5D 18 54 11 0B 7F 6C 79 F5 AC 82 19 F8 5E 15	0j)-I-001y0Lc a"-
000F5F60	72 00 60 00 20 00 43 00 6F 00 6E 00 6E 00 65 00	P.k..C.o.m.m.a.n...	000F5F40	19 A0 81 52 00 6D 3F EB 93 C2 E7 43 FF 00 56 6C	-Ran.m7000c0j=01
000F5F68	43 00 74 00 69 00 6F 00 6E 00 3A 00 80 00 00 00	E.t.i.o.n.....	000F5F40	1C D6 55 AF F0 D7 27 42 0E 99 98 40 94 04 9D 02	0u'0'-000000-
000F5F70	00 00 00 00 20 00 20 00 20 00 4D 00 65 00 64 00	..a..M.e.d...	000F5F40	08 D2 C8 59 C3 C3 C6 7C 00 FC DF 1E CD 7E 9F 07	0E'00000000"0x
000F5F78	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	i.a..S.t.a.t.e...	000F5F40	0E C1 33 AC 00 00 00 00 00 00 00 00 00 00 00 00	b43-Eagk:z900-I
000F5F80	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	00 0F 26 20 49 00 00 00 00 00 00 00 00 00 00 00	L'P00\$'1\00' 1
000F5F88	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	CE 53 0E 00 00 00 00 00 00 00 00 00 00 00 00	I5z00- Rzxi1j=
000F5F90	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	78 08 86 A0 20 C5 56 7D 0A 07 00 00 1F 37 4A 04	z000000).x0.70J
000F5F98	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	88 DF 34 95 0E 8F 1C E6 D8 23 C7 09 27 08 78 44	00000000.0x0
000F5FA0	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	84 51 08 AE 6C EB 08 56 E1 8D 3D A1 CF 36 7D 0F	00010000-0160
000F5FA8	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	0E C3 31 F9 15 28 89 68 8F 12 6E 86 31 80 FE 43	0010-(00:0100c
000F5FB0	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	36 F7 F8 12 69 FF 08 72 F2 84 85 44 2D CA F7 9F	0:0 1j000000-00
000F5FB8	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	02 5E C7 98 08 E1 D0 00 0C 0E 14 91 53 28 00 00	0'000000-0000-0
000F5FC0	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	18 1E 03 A7 81 90 00 8D 16 2C EA F1 03 23 76 05	050000-0000
000F5FC8	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	0E 53 24 E2 7F 07 71 84 63 34 90 FC CF 88 0A 08	NSS00000000
000F5FD0	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	A9 A7 16 5E 21 0D CF 03 70 80 78 CA 49 97 FD 03	18.-"110"(0100-
000F5FD8	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	00 2A FF D9 4A 82 13 2D A0 16 38 03 A2 EA D8 5A	000000--0-0002
000F5FE0	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	00 2E 66 7A 7E 8C AC E1 FE 0E 92 A0 D7 94 92 D7	-fz"-00000000"x
000F5FE8	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	F5 8C 3F 83 C8 5D 83 83 6F 1A 86 CA 62 87 8D 99	0000E}Eo-1E0000
000F5FF0	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	8C 82 C1 6A 82 CE 97 28 27 F3 76 F8 3D 61 3F 89	-00j0000000=070
000F5FF8	00 20 00 2E 00 20 00 2E 00 20 00 2E 00 20 00 2E 00	..c.o.m.p.r.e.s.s.i.o.n. C.o.n.t.a.i.n.s a 15-b.y.t.e.s h.e.a.d.e.r f.o.l.l.o.w.i.n.g b.u.t t.h.e o.u.t.p.u.t o.f t.h.e c.o.m.m.a.n.d.	000F5F40	00 66 57 FD 68 91 CA 99 FA 03 84 8F F8 95 9C 51	0000j'000"00000
000F6000	65 00 63 00 00 74 00 69 00 6F 00 6E 00 6E 00 65 00	...t.e.s..C.o.m.m.a.n.d...	000F5F40	0E 79 95 00 12 42 20 41 2C 48 C8 F4 98 28 99 99	0000..0..000000

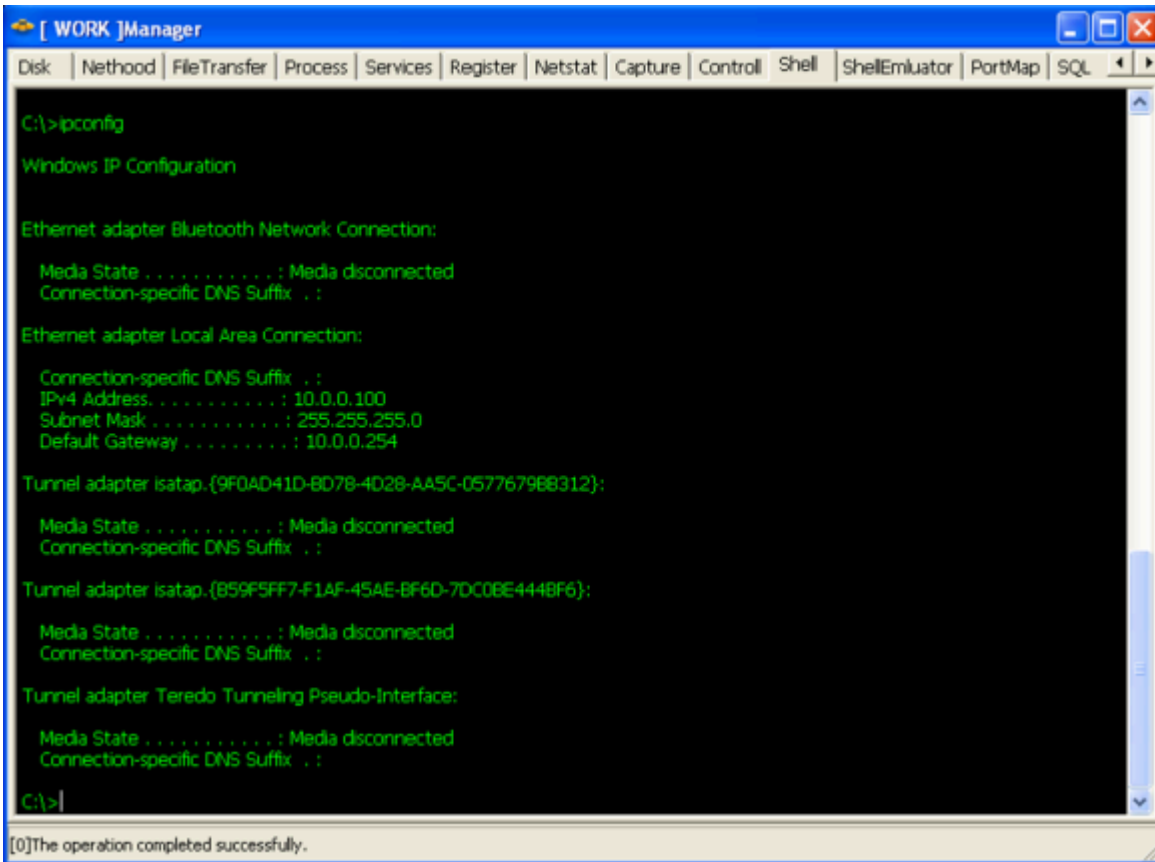
Then, the malware uses WSASend API to send the traffic.

Address	Hex dump	ASCII	00139C33
03EAFD44	8c 02 00 00 28 5f bf 03	...(\.?)a<?>	CALL to WSASend from 00139C31
03EAFD54	28 5f bf 03 0c 02 00 00	...a...<?>	Socket = 520
03EAFD64	20 89 13 00 f8 44 58 00	...<?>	nBuffers = 03EAFD44
03EAFD74	28 5f bf 03 28 5f bf 03	...<?>	nBytesSent = 03EAFD54
03EAFD84	C4 FD E4 03 92 8C 13 00	...<?>	Flags = 0
03EAFD94	88 FD E4 03 30 75 00 00	...<?>	pOverlapped = 00504514
03EAFDAA	2E 07 00 00 10 08 13 20	...<?>	Callback = NULL

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.

```
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 throughout 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.

Source: <https://countuponsecurity.com/2018/02/04/malware-analysis-plugx/>