SOPHOS
Security made simple.

PlugX – The Next Generation

By **Gabor Szappanos**, Principal Researcher

A SophosLabs technical paper - June 2014

Contents

Overview 2
Deployment 3
Payload 3
Final payload 6
Conclusion 10
Appendix 10
References 11

Overview

The ominous PlugX backdoor has been covered by numerous security blogs in the past[1,2]. There were a few variations in the distribution and the deployment of this backdoor, but the end result was always the same.

Three files are dropped on the infected computer[3]:

- A clean and digitally signed application, registered for startup as a service in the registry (e.g., in HKLM\SYSTEM\CurrentControlSet\Services\WS\ImagePath)
- A dynamically linked malicious library, loaded by the above application by DLL search order hijacking, and serves as the loader of the final payload
- The final payload, loaded by the above library, an encrypted data file

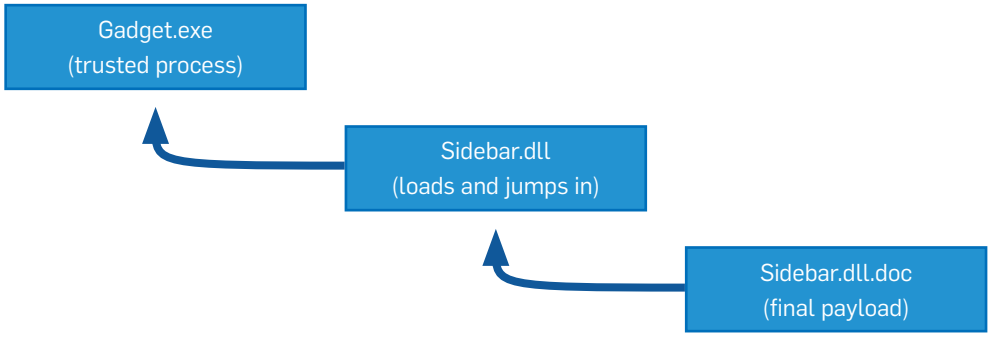


Figure 1. PlugX loading scheme

At the end of 2013, a brand new generation of the PlugX backdoor appeared on the scene. Our first encounter with it was in a distribution campaign which focused on exploiting the popular Japanese word processor Ichitaro[4], but other researchers observed the new generation from different campaigns[5].

What we didn't mention in our report back then was that we saw a single sample that broke the usual scheme described in Figure 1. This one did not use a signed executable for cover, and did not drop the payload into the infected system as a separate file. Instead, it decrypted and loaded it into the memory, without hitting the disk. At that point we couldn't be sure whether it was a single experiment or a development that was going to be consistently used.

As time passed, we found a handful of other samples that use the very same technique. We could therefore be sure that it was not just a one-time mistake, and thus, it makes sense to write about it. As the overall operation of the PlugX backdoor has already been documented in great detail[1,6], in this analysis we focus only on the differences in the new generation.

Deployment

The malware uses the traditional scheme in the sense that it is distributed in exploited Rich Text Format Word documents.

Other than that, it is rather widespread in its methods. We have seen the diskless PlugX in the following scenarios:

- Dropped by an exploited Ichitaro document
- Downloaded by a Word DOCX document exploiting the CVE-2013-3906 vulnerability
- Dropped by an Word Rich Text Format document exploiting CVE-2012-0158, using a fake ZIP object as storage

The shellcode activated by the exploit decrypts and unpacks the embedded dropper the same way the earlier variations of the same family[2] or closely related cousins[7] did.

Payload

The payload is bytewise XOR encrypted and LZNT compressed, but unlike the classic PlugX, the LCG algorithm is missing on top of it[8].

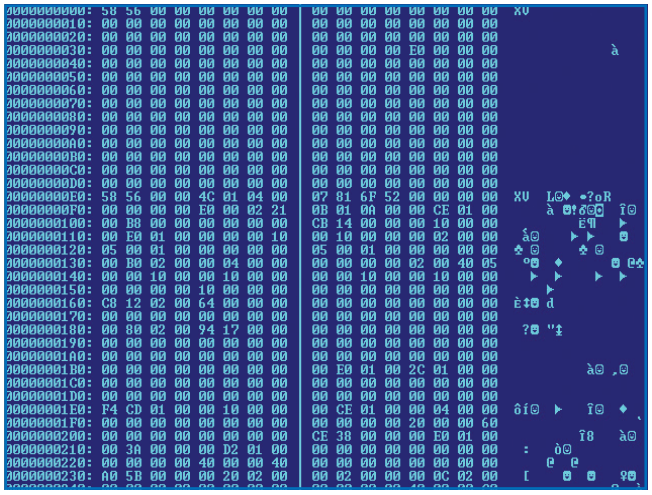
```
                push    eax
                mov     [ebp+var_B0], 'ldtn'
                mov     [ebp+var_AC], 'l'
                call    [ebp+var_24]
                mov     esi, eax
                test    esi, esi
                jnz     short loc_379
                push    7
                jmp     sub_6BF
; ---------------------------------------------------------------------------

loc_379:                                ; CODE XREF: sub_1F7+179↑j
                lea     eax, [ebp+var_44]
                push    eax
                push    esi
                mov     [ebp+var_44], 'DltR'
                mov     [ebp+var_40], 'moce'
                mov     [ebp+var_3C], 'serp'
                mov     [ebp+var_38], 'fuBs'
                mov     [ebp+var_34], 'ref'
                call    edi
                mov     [ebp+var_C], eax
                test    eax, eax
                jnz     short loc_3B1
                push    8
                jmp     sub_6BF
; ---------------------------------------------------------------------------

loc_3B1:                                ; CODE XREF: sub_1F7+1B1↑j
                lea     eax, [ebp+var_60]
                push    eax
                push    esi
                mov     [ebp+var_60], 'cmem'
                mov     [ebp+var_5C], 'yp'
                mov     [ebp+var_5A], 0
                call    edi
```

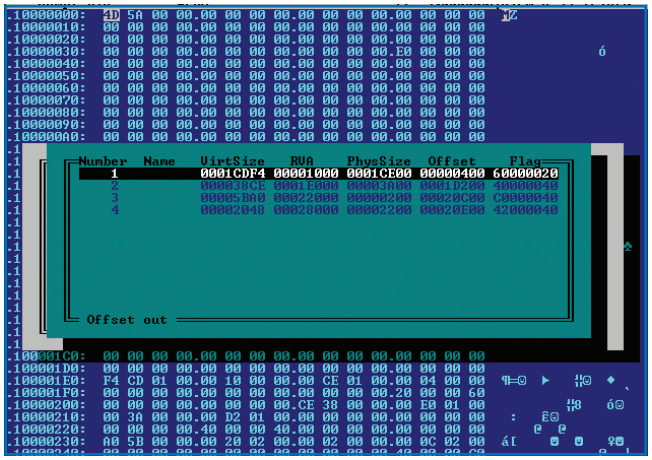
Decompressor shell code

The decompressed content is a sort of DLL file. The magic bytes, marking the start of the executable header ('MZ') and the start of the Portable Executable header ('PE') are both overwritten with 'XV'. It makes the unpacked image unexecutable for the operating system (and unsuitable for static analysis). Note that this is only a transient, short-lived form of the payload in the memory. It is created after unpacking, and will be erased when the loading process is finished and the execution is transferred to the backdoor code.



Decompressed payload DLL

Fixing the missing marker bytes, it makes proper looking PE files (though a bit odd as the sections have no names).



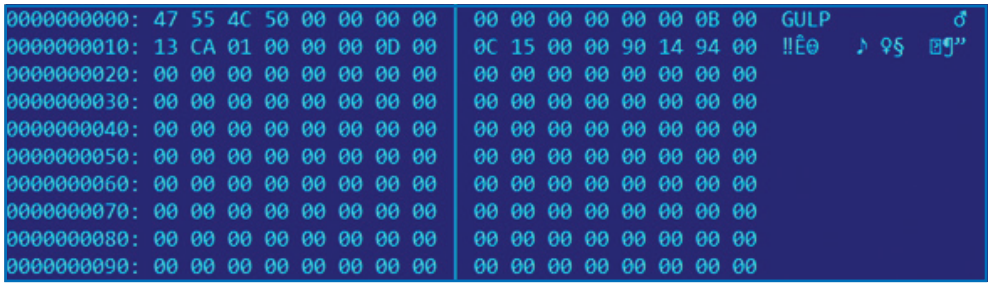
Decompressed and fixed payload DLL

The lack of the markers and inability of the OS loading the DLL does not bother the malware loader. It does the PE loading by itself, including the following operations:

- Allocates the memory areas for the PE section, sets the access rights for the regions and copies the section content there
- Performs the relocation of the absolute data offsets—an action needed for dynamically linked libraries
- Resolves the required API function addresses from the import table

It then overwrites the beginning of the file (including the PE header) with zeros. As a result, the dumped executable will be very difficult to analyze. (This step existed in older PlugX variants as well, but then it was done by the payload DLL itself during its bootload process.) Finally, the loader jumps to the entry point of the payload file and executes it.

This is already a deflection from the classic scheme, where the beginning of the unpacked PE image was overwritten with the 'GULP' marker.



Classis PlugX DLL memory image

Final payload

The final payload is a backdoor that connects to a remote server.

The backdoor collects system information and sends it to the command and control server. This information includes:

- Computer name
- Username
- CPU
- User token
- User group
- Total/free memory
- OS version
- System time

Functionalities are implemented in plugins, which are registered the following way in the classic version:

```
seg000:00951C8C 8B 00                mov     eax,    [eax]
seg000:00951C8E 56                   push    esi
seg000:00951C8F 68 10 53 96 00       push    offset  aScreen ; “Screen”
seg000:00951C94 68 F0 1C 95 00       push    offset  Screen
seg000:00951C99 68 20 02 12 20       push    20120220h
seg000:00951C9E 6A 08                push    8
seg000:00951CA0 6A FF                push    0FFFFFFFFh
seg000:00951CA2 FF D0                call    eax
```

This registration call connects the plugin name with the implementing function. In the case above, the Screen() function implements the screenshot capture functionality. This functionality is performed by a couple of subfunctions, and the Screen() procedure performs the registration of these subfunctions.

Using the information about the plugin names, it was possible to compile a table that contains the functionality implemented by each plugin.

Function name	Functionaity
Disk	Get drive information (type, free space) Enumerate files Create directory Create/modify file Copy/delete/move/rename files Execute files
KeyLog	Log keystrokes to file %ALLUSERSPROFILE%\SxS\NvSmart.hlp
Nethood	Enumerate shared network resources
Enumerate	Set TCP connection state Enumerate UDP and TCP connections
Option	Lock workstation Logoff/reboot/shutdown workstation Display messagebox
PortMap	Perform port map
Process	Terminate process Enumerate processes and modules Get process and module information
RegEdit	Enumerate/create/delete registry entries
Screen	Capture screenshot
Service	Get service information Change service configuration Start service Control service Delete service
Shell	Create remote shell
SQL	List SQL drivers List SQL data sources Execute SQL command
Telnet	Create telnet connection

The above call makes an easily recognizable structure for the backdoor. This was changed with v2, where the plugin registration call was removed, and the subfunctions are directly registered from the bootloader code.

The code of the payload shows a very high similarity with the classic PlugX backdoors—at least at the functionality level. But on the underlying code level, there are many differences. It looks as if the code was refactored using the specification of the original backdoor. At first, it looked like a completely different malware family. Further comparison revealed the similarities between the code of the old and the new PlugX generation.

The most characteristic similarity can be found at the initialization of plugins (public functions that provide backdoor functionality available to use remotely on an infected computer)[1].

Each public plugin can contain one or more internal functions that implement the functionality, and are started as separate threads. The plugin modules are begun using a function call, pairing the internal name of the function (bootproc in this case) with the address of the plugin functions. It then starts the new thread. This call look like this in the classic PlugX variants:

```
seg000:00941786 6A 00                 push      0
seg000:00941788 68 70 18 94 00        push      offset bootProc
seg000:0094178D 68 90 46 96 00        push      offset aBootproc ; "bootProc"
seg000:00941792 BB 68 BF 96 00        mov       ebx, offset hThread
seg000:00941797 E8 44 D7 01 00        call      call_CreateThread
```

The diskless new variants follow much the same scheme, with the minor difference that the internal plugin names are abbreviated:

```
seg000:008E16CC 57                    push      edi
seg000:008E16CD 68 E6 16 8E 00        push      offset bootproc
seg000:008E16D2 68 3C 11 8F 00        push      offset aBp    ; "BP"
seg000:008E16D7 BB 50 83 8F 00        mov       ebx, offset hThread
seg000:008E16DC E8 96 A7 00 00        call      call_CreateThread
```

Following that, the subfunctions of the particular plugin are also registered, using a call that takes a couple of numeric parameters. One of them is the unique numeric ID of the subfunction. The other looks like a date that is likely to be the date when the particular functionality was added to the PlugX arsenal.

In the classic PlugX it looks like this:

```
.text:1000C3C6 C7 06 23 01 12 20      mov       dword ptr [esi], 20120123h ; date
added
.text:1000C3CC C7 46 04 04 30 00 00   mov       dword ptr [esi+4], 3004h ;
function + subfunction
.text:1000C3D3 C7 46 08 2E 00 00 00   mov       dword ptr [esi+8], 2Eh
.text:1000C3DA C7 46 0C 00 00 00 00   mov       dword ptr [esi+0Ch], 0
.text:1000C3E1 E8 4A F3 FF FF         call      sub_1000B730
```

The new version uses almost exactly the same structure, both in code and parameters, but the date is modified:

```
seg000:1000806D C7 06 10 08 13 20     mov       dword ptr [esi], 20130810h ; date
added
seg000:10008073 C7 46 04 04 30 00 00  mov       dword ptr [esi+4], 3004h ;
function + subfunction
seg000:1000807A C7 46 08 2E 00 00 00  mov       dword ptr [esi+8], 2Eh
seg000:10008081 FF 15 14 21 02 10     call      ds:dword_10022114
```

Analyzing a lot of variants that arrived to our lab in 2013, the latest date we could observe was 20120325h. This indicates that nothing much was happening in the PlugX development since that time. That is, until the next-generation samples started to show up.

Now I have to assume, that a major refactoring went on, and finished at the end of the summer (10th August), resulting in this new variant. The parameter value was the same in all four of the diskless PlugX variants.

As it was mentioned, the names of the subfunctions changed in the v2 samples. This is summarized in the following table, along with the basic functionality of the subfunctions.

Functionality	Procedure (6.0 classic)	Disk-less set 1	Disk-less set 2
Initialize variables	bootProc,DoImpUserProc	BP, DIUP	BP, DIUP
	JoProc, JoProcAccept, JoProcBroadcast, JoProcBroadcatRecv, JoProcListen		JP, JPL, PBB, PBBR, JPA
Injects into services.exe	OlProc, OlProcNotify, OlProcManager	OP, OPM, OPN	OP, OPN, OPM
Shellcode to unpack and install main code in an injected process	LdrLoadShellCode		
Log keystrokes to file	KLProc	KP	KP
Capture screenshot	ScreenT1, ScreenT2	ST1, ST2	SC, ST1, ST2
Create remote shell	ShellT1, ShellT2		
Create a command shell to establish telnet connection, relay input/ output with the C&C server (Telnet)	TelnetT1, TelnetT2	TT1, TT2	TT1, TT2
Create elevated process and inject code	SiProc		
	SxWorkProc	SWP	SWP
	PlugProc	PP	PP
Display Message box	RtlMessageBoxProc		RMBP

Function sets in PlugX

It's worth noting that two basic plugin sets were observed, which represents two slightly different sets of functionalities. This is not a surprise when talking about a backdoor with modular plugin architecture. These two configurations existed already, among the classic samples as well as in the next-generation samples.

Conclusion

A year ago, PlugX development seemed to be stuck with only minor facelifts to the code. It appeared that the focus shifted to affiliate projects like Smoaler[7].

Now, it is clear that the development efforts continue and we can't expect the disappearance of this general purpose malware family.

Appendix

The following droppers were identified to belong to the diskless PlugX variation:

Dropper SHA1: e6281d74a8d874c5a46ec2c1c9c145aa60a4c886
Distribution method: Ichitaro exploit
C&C server: msn.catalogipdate.com

Dropper SHA1: ce60e8f27031126a680c90a664443f5cd85bb1e8
Distribution method: CVE-2013-3906
C&C server: av4.microsoftsp3.com
Loader SHA1: f0c0975f349f12cdbd39e00b151df07cd82ecf7d

Dropper SHA1: 3710eded0bc5bc5b3bf792834ac21f1452d4bc7b
Distribution method: CVE-2012-0158
C&C server: scqf.bacguarp.com, scqf.zuesinfo.com
Loader SHA1: 19957f4cc33d8676736756f81899a2fbd0586c1e

Dropper SHA1: ac321b556020061fae7bb35a79a692d7509c1bb8
Distribution method: CVE-2012-0158
C&C server: scqf.bacguarp.com, scqf.zuesinfo.com
Loader SHA1: 896f3711c4beca592127ace7615574e2b6024d07

References

1. http://lastline.com/an-analysis-of-Plugx.php
2. http://nakedsecurity.sophos.com/2013/05/20/inside-the-Plugx-malware-with-sophoslabs-a-fascinating-journey-into-a-malware-factory/
3. http://nakedsecurity.sophos.com/2013/02/27/targeted-attack-nvidia-digital-signature/
4. http://nakedsecurity.sophos.com/2013/12/04/new-Plugx-malware-variant-takes-aim-at-japan/
5. http://blog.cassidiancybersecurity.com/post/2014/01/Plugx-v2%3A-meet-SController
6. https://www.circl.lu/files/tr-12/tr-12-circl-Plugx-analysis-v1.pdf
7. http://nakedsecurity.sophos.com/2013/07/15/the-Plugx-malware-factory-revisited-introducing-smoaler
8. http://www.contextis.com/files/Plugx_-_Payload_Extraction_March_2013_1.pdf

United Kingdom and Worldwide Sales
Tel: +44 (0)8447 671131
Email: sales@sophos.com

North American Sales
Toll Free: 1-866-866-2802
Email: nasales@sophos.com

Australia and New Zealand Sales
Tel: +61 2 9409 9100
Email: sales@sophos.com.au

Asia Sales
Tel: +65 62244168
Email: salesasia@sophos.com

Oxford, UK | Boston, USA

1180-06.14DD.tpna.simple