PlugX “malware factory” celebrates CVE-2012-0158 anniversary with Version 6.0

By **Gabor Szappanos**, Principal Researcher, SophosLabs

May 2013

Just over a year ago, in April 2012, Microsoft's Patch Tuesday [A] fixed an exploitable vulnerability [B] known as CVE-2012-0158. [C]

This was a remotely exploitable bug in the Windows Common Controls that was being used in the wild for drive-by attacks. Drive-bys are malware infections that happen merely by visiting a malicious website or opening a poisoned document, without any popup confirmations or security warnings.

At the beginning of 2013, we wrote about a recent variant of the PlugX malware [D], commonly seen in Tibetan-themed malware attacks, still using the same vulnerability.

The bad news is that the attackers have celebrated the anniversary of the CVE- 2012-0158 patch with yet another version of PlugX, which now reaches version 6.0, and is still spreading on unpatched computers. (Yes, malware has major version upgrade and point releases, too.)

The core of PlugX Version 6.0 is a remote-control backdoor that is similar to previous versions - a list of the commands available to attackers on infected computers can be found in our earlier analysis [D].

But Version 6.0 has some interesting new aspects, and also gives us a peek into the size and structure of the programming project behind the malware.

In this paper, we take apart the infection mechanism of the new version to give you an insight into how attackers divide the operation of malware into distinct steps.

Splitting up malware means that each step does only a small piece of the overall work, in order to avoid looking suspicious on its own. The aim is to reduce the chance of being flagged as dangerous by heuristic defences that expect more complex behaviour.

Delivery of the threat

The infected samples acquired by SophosLabs were delivered in RTF (rich text format) files that triggered the CVE-2012-0158 vulnerability [C] when opened on unpatched Windows computers.

Previous PlugX attacks have used Tibetan-themed decoys; this time the decoy document is a timeline of the alleged arrest and detention of Anne Zhang, the young daughter of a Chinese dissident.

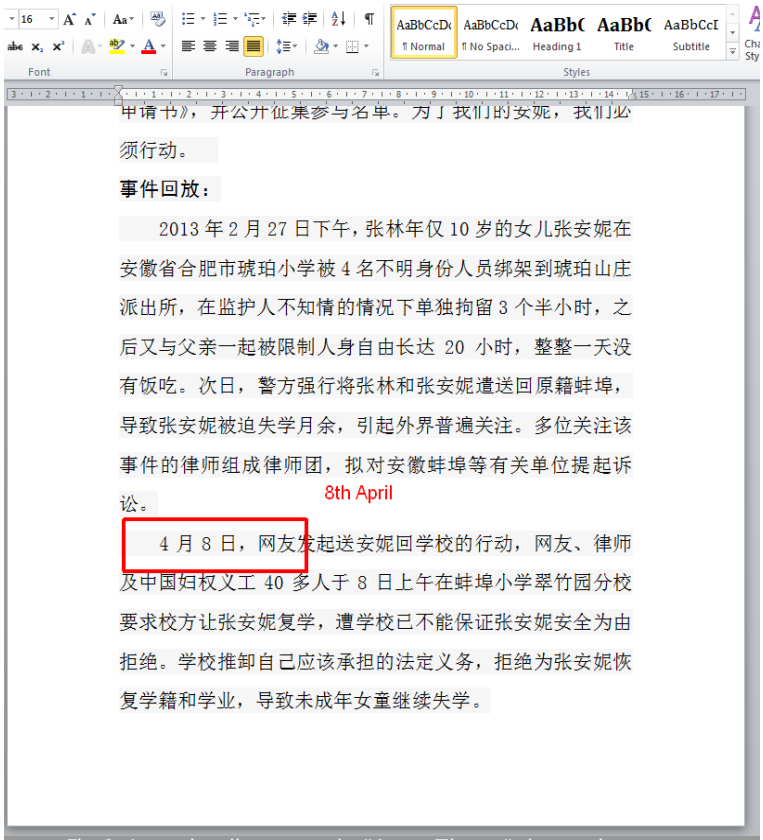

Fig 1: Last timeline entry in "Anne Zhang" decoy document

The last event listed in the decoy document is dated 08 April 2013, presumably close to the time that this particular PlugX attack package was put together.

This assumption is further strengthened by the timestamps in some of the executables files created during infection. For example, the files described below as the intermediate dropper and the final loader were compiled early in the morning of 09 April 2013.

The multi-step infection scheme

We have divided the description into seven parts, each involving a separate executable object.

Note that the malware components are not just regular Windows EXE files. This malware consists of:

- **Shellcode**, made up of raw Intel machine code embedded in the original RTF.
- **Several DLLs**, a special sort of executable usually loaded by other programs.
- **A clean EXE file** taken from a legitimate Chinese internet application.
- **A custom binary image**, used for the main payload and loaded directly by one of the malware DLLs.

Phase 1: RTF carrier

Filename	安妮活动.doc (“Anne activities.doc”)
File size	512422 bytes
SHA1	`3b4a6d4271df5276237185a642c7e00bb828f9fe`
MD5	`1d3c184dde74ac4ea8a25e57a40c6ce4`
Sophos detection	Exp/20120158-A

This file serves merely as a carrier. When it is opened on an unpatched computer, the CVE-2012-0158 bug is triggered. Execution flows into the phase 2 shellcode that is embedded inside the RTF.

Phase 2: Shellcode

The shellcode itself runs in two parts, a small loader/decryptor and a much larger main body. This is common practice in document exploits, because it reduces the amount of openly visible malicious machine code.

The loader/decryptor uses the standard method (`Process Environment Block → PEB_LDR_DATA → InMemoryModuleList`) to locate `kernel32.dll` in memory and read its export table to locate the Windows API functions it will use later.

As usual, the functions are identified by matching checksums of their names, not the names themselves. This avoids having rather obvious function name strings such as `LoadLibraryA` and `GetProcAddressA` visible in the decryptor's code.

```
   push    10FA6516h                       ; ReadFile
   call    Get_API_by_checksum
   mov     [edi+0B8h], eax
   xor     esi, esi

next_filehandle:                           ; CODE XREF: seg000:000048C9↓j
   add     esi, 4
   lea     eax, [edi+60h]
   push    eax
   push    esi
   call    ds:off_20[edi]                  ; GetFileSize
   cmp     eax, 0FFFFFFFFh
   jz      short next_filehandle
   cmp     eax, 7D1A6h                     ; compare with RTF size
   jnz     short loc_4903
   mov     [edi+60h], eax
   mov     [edi+64h], esi
   push    0
   push    0
   push    521Ch                           ; stage 2 offset
   push    dword ptr [edi+64h]             ; parent RTF handle
   call    dword ptr [edi+0B0h]            ; SetFilePointer
   push    40h ; '@'
   push    1000h
   mov     ebx, [edi+60h]
   sub     ebx, 521Ch
   push    ebx
   push    0
   call    dword ptr [edi+0B4h]            ; VirtualAlloc

loc_4903:                                  ; CODE XREF: seg000:000048D0↑j
   mov     [edi+0BCh], eax
   push    0
   push    0
   lea     ebx, [edi+0C0h]
   push    ebx
   mov     ebx, [edi+60h]
   sub     ebx, 521Ch
   push    ebx
   push    dword ptr [edi+0BCh]            ; allocated memory handle
   push    dword ptr [edi+64h]             ; parent RTF handle
   call    dword ptr [edi+0B8h]            ; ReadFile
```

Fig 2: Shellcode locating its parent RTF "carrier" file

The loader enumerates all legal file handles (for historical reasons these start at 4 and increase by 4 for each new handle) to find an open file that is 514222 bytes in size, the same as the RTF. In this way, the shellcode gains access to the RTF carrier file to read in the second stage of the shellcode.

The loader then does the following:

- Allocates a block of memory for the main body.
- Reads the main body from offset 0x521C in the RTF carrier file.
- Decrypts the main body using a simple XOR unscrambling algorithm.
- Transfers control to the main body.

The main body of the shellcode then:

- Reads in a compressed DLL and the decoy from the RTF carrier.
- Decompresses the DLL using the system function `RtlDecompressBuffer()`.
- Writes the DLL to `%TEMP\dw20.dll`.
- Writes the decoy document to `%TEMP%\~WINWORD`.
- Launches the DLL using LoadLibrary().
- Launches a new copy of Word to load the decoy.

The compression of the DLL and its decompression with the `RtlDecompressBuffer()` system function is unusual.

The shellcode then terminates the instance of Word in which it has grabbed control, but this is covered up by the newly-loaded decoy document opened in a fresh copy of Word.

Phase 3: Initial dropper

Filename	dw20.dll
File size	233472 bytes
SHA1	`5da9081013e6fc6ef33a47a46dceced8b847f543`
MD5	`e22b839f042c391fb022dc58fe7901b5`
Sophos detection	Troj/Plugx-I

This is the DLL that is stored in compressed form in the RFC carrier and extracted in phase 2.

Its purpose is to extract and drop the next file in the infection chain, which it does as follows:

- Reads from offset 0xA00 in its own executable, where another DLL is embedded.
- Writes the extracted data to a file in the `%TEMP%` directory.
- Launches the file with `WinExec()`.

This DLL is the intermediate dropper described in phase 4. Its filename is generated using `GetTempFileNameA()` [E]. That means it is different

every time the malware infects. The dropped file is then launched with `WinExec()`.

The initial dropper DLL exports a single function, blandly named `DoWork()`. This function is supposed to remove the file after it has run, but never gets called.

As a result, this component will be left behind in the temporary directory.

Phase 4: Intermediate dropper

Filename	generated by `GetTempFileNameA()`
File size	230912 bytes
SHA1	`9bb28483f4c32dec5f011b01f6e7e2984253ef54`
MD5	`83006ac9fb73bc2b891f36dd2f759230`
Sophos detection	Troj/Plugx-I

This is a regular EXE file that extracts and drops three more files:

- A copy of a clean EXE program file named `Gadget.exe`, created and digitally signed by Chinese social media outfit Tencent.
- A DLL named `Sidebar.dll` that the Gadget.exe program will be tricked into loading.
- A binary file named `Sidebar.dll.doc` that is an encrypted binary object containing the main malware components.

```
GetSystemTime(&SystemTime);
if ( (_WORD)SystemTime.wYear >= 2013u )
{
  v6 = LoadLibraryA("user32.dll");
  wsprintfW = GetProcAddress(v6, "wsprintfW");
  GetCurrentProcessId();
  ((int (__stdcall *)(char *, wchar_t *))wsprintfW)(&v9, L"Global\\DelSelf(%8.8X)");
  CreateMutexW(0, 0, &Name);
  GetTempPathW(0x800u, &Buffer);
  ((int (*)(const WCHAR *, const char *, ...))wsprintfW)(&dllfilename, (const char *)L"%s\\Sidebar.dll", &Buffer);
  ((int (*)(_UNKNOWN *, const char *, ...))wsprintfW)(&payloadfilename, (const char *)L"%s\\Sidebar.dll.doc", &Buffer);
  ((int (*)(WCHAR *, const char *, ...))wsprintfW)(&loaderfilename, (const char *)L"%s\\Gadget.exe", &Buffer);
  v2 = 0;
  hdllfile = CreateFileW(&dllfilename, 0x40000000u, 1u, 0, 2u, 0, 0);
  v3 = hdllfile;
  if ( hdllfile == (HANDLE)-1 )
  {
    v2 = GetLastError();
  }
  else
  {
    if ( !WriteFile(hdllfile, "MZÉ", 0xA400u, (DWORD *)&SystemTime.wDayOfWeek, 0) )
      v2 = GetLastError();
    CloseHandle(v3);
  }
  if ( v2 )
    ExitProcess(0);
  if ( writefile((const WCHAR *)&payloadfilename, v4, (int)&payloaddata, 124732) )
    ExitProcess(0);
  if ( writefile(&loaderfilename, v5, (int)"MZÉ", 26112) )
    ExitProcess(0);
```

Fig 3: Intermediate dropper creating payload files

The intermediate dropper deletes itself once it has done its work, so it is only briefly present on infected systems.

Phase 5: Clean loader

Filename	Gadget.exe
File size	26112 bytes
SHA1	8985c2394ed9a58c36f907962b0724fe66c204a6
MD5	6b97b3cd2fcfb4b74985143230441463

This is a clean file published and signed by Chinese social media company Tencent.

This very simple program uses `LoadLibrary()` to load a DLL called `Sidebar.dll`, usually another component of Tencent's application, and then calls an exported function named `Main()`.

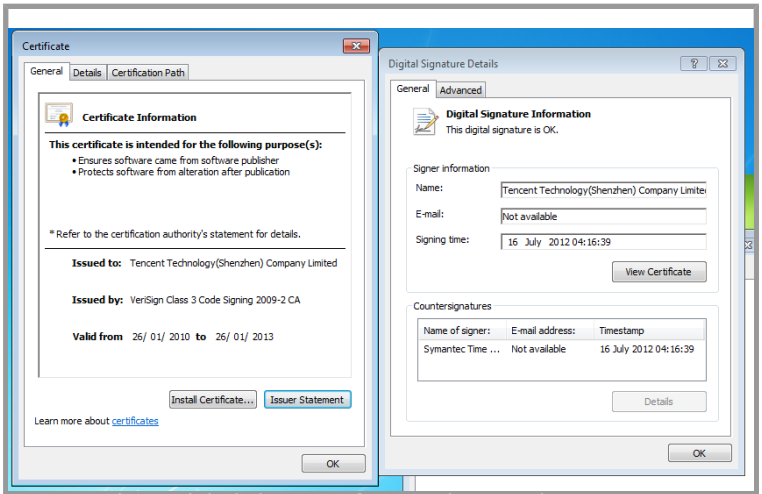

Fig 4: Digital signature from Gadget.exe by Tencent

Tencent's Gadget.exe fails to specify a full directory and filename for the DLL it loads, so it suffers from an insecure library loading vulnerability [F].

`Gadget.exe` will load any DLL named `Sidebar.dll` that happens to be in the current directory.

So, instead of directly loading the malicious `Sidebar.dll`, dropped in phase 4, the malware loads `Gadget.exe` and thereby indirectly loads the malicious DLL.

This has the effect of shrouding the Sidebar DLL in a veneer of legitimacy, because `Gadget.exe` is digitally signed by a trusted certificate authority.

This is the same trick described in our earlier PlugX analysis [D]; in the earlier example, however, the malware misused a program from graphics card vendor Nvidia instead of Tencent's `Gadget.exe`.

Phase 6: Final loader DLL

Filename	Sidebar.dll
File size	41984 bytes
SHA1	`22d7ab94fd7042a781c0bee992fc0bf25f3bd626`
MD5	`fe3548281f9716862ee6e614ae7a0e76`
Sophos detection	Troj/Plugx-I

`Sidebar.dll` is loaded (erroneously) by the Tencent `Gadget.exe`; if the year is 2013 or later, it performs the final file-based phase of PlugX's infection.

As explained below, the phase 7 payload file `Sidebar.dll.doc` is an encrypted binary object containing the main malware components. This means it cannot be loaded directly using standard Windows API functions such as `LoadLibrary()` or `WinExec()`.

```
  ExitProcess(0);
NumberOfBytesRead = GetModuleFileNameW(hModule, &Filename, 0x2000u);
lstrcatW(&Filename, L".doc");
v5 = CreateFileW(&Filename, 0xA0000000u, 1u, 0, 3u, 0, 0);
hpayloadfile = v5;
if ( v5 == (HANDLE)-1 )
{
  result = GetLastError();
}
else
{
  v3 = VirtualAlloc(0, 0x100000u, 0x1000u, 0x40u);
  mpayload = (int (*)(void))v3;
  if ( v3 && ReadFile(hpayloadfile, v3, 0x100000u, &NumberOfBytesRead, 0) )
  {
    CloseHandle(hpayloadfile);
    mpayload();
    Sleep(0xFFFFFFFFu);
    result = 0;
```

Call payload

Fig 5: Malicious Sidebar.dll loaded by clean Gadget.exe

So the purpose of the Sidebar DLL is to organise the customised loading of, and the transfer of execution to, the `Sidebar.dll.doc` file.

Phase 7: Payload

Filename	Sidebar.dll.doc
File size	124732 bytes
SHA1	`7d36518acf345794a0a6421542d1c6b8b052e58a`
MD5	`0475f406de14fdbca2ec542d6743e1c4`

Finally, we reach the component that the attackers wanted to activate in the first place.

It is not a Windows PE executable file, but a custom binary image consisting of a loader and a number of additional encrypted binary images that implement the functionality of the malware.

When the phase 6 final loader transfers control here, the loader code at the start of the payload file extracts, decrypts and decompresses the main backdoor code.

The backdoor is very similar to the version we analysed earlier [D]; even the "demo mode" of the earlier version is present in PlugX Version 6.0.

Differences from earlier PlugX versions

The major difference is that the payload file (`Sidebar.dll.doc`) that we acquired and analysed is clearly a debug build, featuring source logging.

The logging function is invoked like this:

```
plugx_log(sourcefile,linenumber,statuscode);
```

Logging produces as useful trail of information because the `plugx_log()` function is invoked after Windows API calls that return an unwanted status code (usually indicating an error).

The function builds a log message, which contains the source code file name, the current line number in the source file and the status code, as illustrated below.

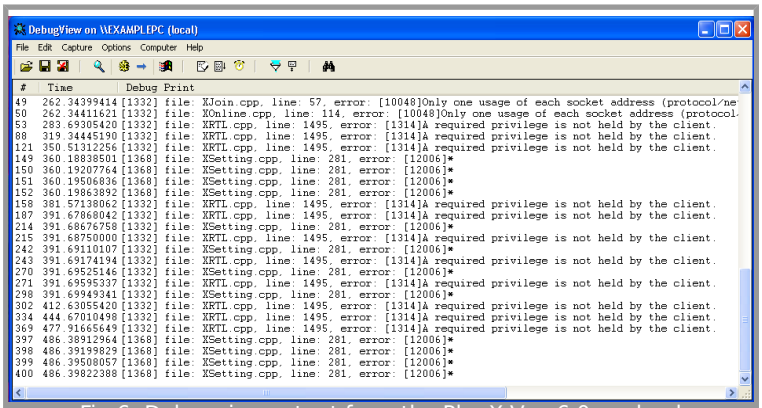

Fig 6: Debugging output from the PlugX Ver. 6.0 payload

The log message is passed to `OutputDebugStringA()`, a Windows function that usually generates no visible output. But debug output can easily be examined with a tool such as DebugView [G]; this gives

us a way to keep track of the activity of the malware on an infected system.

Since `plugx_log()` receives the source code file name and a line number as parameters, collecting logging data gives us an opportunity to map the source code tree, and even to estimate the size of the project.

Mapping the PlugX project

The following table contains a selection of known source file names, known functions implemented in each source file, and the maximum line number known in the file (probably the last API call in the the function):

Source file	Procedures	Functionality	Max line
XBoot.cpp	bootProc DoImpUserProc	Initialize variables and inject shellcode into svchost.exe process	615
XOnline.cpp	OlProc OlProcManager OlProcNotify	Injects into services.exe	1184
XPlgLoader.cpp	LdrLoadShellCode	Shellcode to unpack and install main code in an injected process	1111
XPlugNethood.cpp	Nethood	Enumerate shared network resources	213
XPlugNetstat.cpp	Netstat	Set TCP connection state; enumerate UDP and TCP connections	492
XPlugPortMap.cpp	PortMap	Perform port map	237
XPlugRegedit.cpp	RegEdit	Manage registry entries	719
XPlugScreen.cp	Screen ScreenT1 ScreenT2	Capture screenshot	1376

XPlugShell.cpp	Shell ShellT1 ShellT2	Create remote shell	603
XRTL.cpp		General helper functions for all other modules	1683
XSetting.cpp			722

Using statistical methods to estimate the actual number of lines [H] from the maximum known number in each file (the above table is only a partial list) gives us a total size of 19,771 lines.

Given that the header files in general are not accounted for, the entire project almost certainly exceeds 20,000 lines of code.

Interestingly, the filename `dllmain.cpp` in the phase 7 payload project shows that this component is developed as a DLL file.

Looking at the memory dump of the payload code it looks as though it is first decoded to a valid DLL during the load process, before the first 0x1000 bytes in the allocated memory area (containing the PE header and the section table) are overwritten.

The original DLL should therefore be available in memory as the malware starts up. Indeed, by setting a breakpoint immediately after the decompression stage of the phase 6 loader, we quickly revealed the untainted DLL component.

It has a single export, `DllEntryPoint()`, which starts `bootProc`, the initialisation function for the backdoor.

Digging deeper into the payload code revealed that during the start process the bootProc module calls the `LdrLoadShellcode` procedure. This is actually responsible for decrypting and decompressing the DLL, loading it, then overwriting the start of the DLL header in memory.

This frustrates research by wiping out data that would be useful for analysis, but leaving intact all the binary code that will actually run.

With a full copy of the DLL, the PE header of the payload is exposed. This, in turn, reveals the compile time of the project: 25 February 2013.

The code also includes 32-bit numeric parameters that are used when each function is initialised.

When written out in hexadecimal, these parameters represent dates, presumably denoting the date when each function was added to the PlugX functionality. The compile times of the program files used in the phases above are:

Initial dropper	08 Feb 2013 @ 03:55
Intermediate dropper	09 Apr 2013 @ 04:43
DLL loader	09 April 2013 @ 04:08
PlugX payload	25 Feb 2013 @ 15:26

The range of dates in use gives the impression that the PlugX package components are developed and released individually, rather than as a monolithic whole. That makes it easier for the attackers to update or modify individual parts of the malware.

Where is it from?

Interestingly, the only source file that appears with a full pathname is `XSetting.h`. The name turns out to be written in a Chinese-language encoding, giving a full path of:

```
d:\work\plug6.0(360)(gadget)(非洲来客)(正式)\
                  shellcode\shellcode\XSetting.h
```

The appearance of the string `(gadget)` is interesting, as it is the name of the digitally-signed clean file inside the malware. This suggests that the entire payload project is built around this particular clean loader file.

The first Chinese-language string within brackets above (非洲来客)means "Black hat hacker"; the second string (正式)means "Official".

You may read into that what you will.

What next?

There is no doubt that PlugX development will go on, and new features and tricks will be introduced.

We'll keep an eye on them, and if any interesting or important new features appear, we'll be sure to let you know.

References

[A] http://nakedsecurity.sophos.com/patch-tuesday-april-2012

[B] http://technet.microsoft.com/en-us/security/bulletin/ms12-027

[C] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0158

[D] http://nakedsecurity.sophos.com/targeted-attack-nvidia-digital-sig

[E] http://msdn.microsoft.com/library/windows/desktop/aa364991%28v=vs.85%29.aspx

[F] http://nakedsecurity.sophos.com/unsafe-windows-dll-loading

[G] http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx

[H] http://en.wikipedia.org/wiki/German_tank_problem

For further security information

http://www.sophos.com/why-sophos/our-people/technical-papers.aspx

http://nakedsecurity.sophos.com/

http://podcasts.sophos.com/

