virus BULLETIN Covering the global threat landscape

HOW IT WORKS: STEGANOGRAPHY HIDES MALWARE IN IMAGE FILES

Lordian Mosuela
CYREN

One of the more insidious new obfuscation techniques identified by our research team in the past year, which we believe will grow in popularity, uses a new 'digital steganography', or concealment technique, to evade detection by conventional security tools. Digital steganography is a method of concealing a file, message, image or video within another file, message, image or video. In this case the technique is used by a piece of malware popularly known as Stegoloader (detected by *CYREN* as W32/Gatak), which is a trojan or downloader for stealing data and delivering ransomware.

OVERVIEW: 'NEW AND IMPROVED' HIDING TECHNIQUE

Gatak/Stegoloader is a good illustration of the ongoing arms race between malware writers and your Internet security. A precursor to the new digital steganography technique was seen in the Duqu [1] malware (discovered in 2011), which was found to transmit encrypted data appended to an image (.JPG) file. The Zeus/Zbot [2] malware also uses similar tactics, appending its encrypted configuration file to an image file for exfiltration. However, this technique has proven easy to block using content filtering rules, because the configuration file is simply appended to the image file.

The Gatak/Stegoloader malware, which emerged in 2015 [3], improves on this steganography technique – it completely hides its malicious code within an image (.PNG) file. So far, we have seen this threat bundled in software licence cracking tools that are used (illegally) to generate software licence keys (typically to extend software trials or unlock software features without payment), but there is a high probability that new distribution mechanisms will appear.

HOW IT WORKS

Figure 1 shows a Gatak sample (SHA256: 0a58b98205c8542ae0516b4fe3ff8a4a6d6e9c199ec2d4e0de0aa8f9e1290328) in which two executable files are included inside a compressed archive.

Upon execution of the installer file, the software licence cracking tool (1237.exe) runs and the window shown in

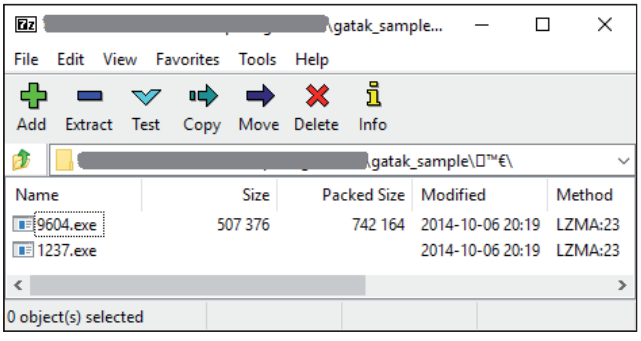

Figure 1: Gatak sample showing two executable files inside a compressed archive.

Figure 2: The cracking tool.

Figure 2 is displayed – this is used to generate software keys for a specific software program. Of course, the Gatak malware (9604.exe) is also executed, but without the user's knowledge.

The malware installation is complex, with many steps, but there are two main parts:

1. Initial decryption and installation of the malware.
2. Download of the image and use of the hidden image data to establish encrypted communications with the C&C server and to download/upload data and/or further malware.

Step 1: Initial installation of the malware

The Gatak malware first assembles its code into memory by decrypting nine segments of the encrypted code embedded in the malware itself, and then transfers its execution to the decrypted code in memory. Figure 3 shows the size and virtual address of nine parts of encrypted code that are included in the three sections of the malware: .data, .adata and sync.

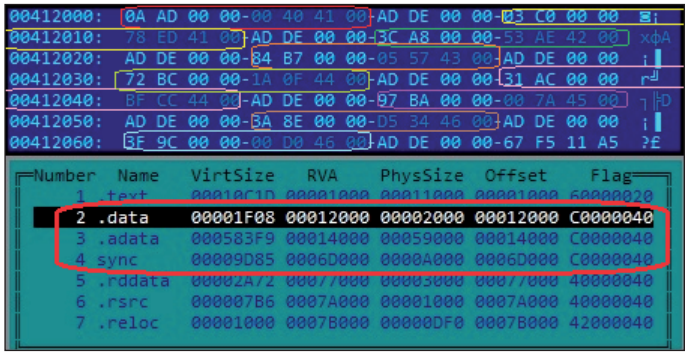

Figure 3: Size and virtual address of nine parts of encrypted code that are included in three sections of the malware.

The key for decryption is two bytes in size and is hard coded in the malware body. Decryption is accomplished by alternating the two bytes in the decryption key and subtracting them from the encrypted code:

1. Subtract a byte of encrypted code using the byte key.
2. Shift the byte key and use another byte key.
3. Store the decrypted byte in the allocated buffer.
4. Increment a pointer to the next byte of encrypted code.
5. Increment a pointer to the next byte of allocated buffer.
6. Loop to #1.

The decrypted code then rebuilds itself by first obtaining the location of the process environment block (PEB) using the FS:[30] register. It then acquires the address of InLoadOrderModuleList (see Figure 4) in order to find the virtual address of ntdll.dll. This procedure is used to construct the required import addresses of the ZwAllocateVirtualMemory, ZwFreeVirtualMemory and LdrLoadDll APIs, which are used to load more DLLs and retrieve the required APIs from them through their API hashes.

Figure 5 shows the list of APIs gathered with their corresponding hashes.

The malware then gets the ftCreationTime.dwLowDateTime of the %USERPROFILE% folder of the infected system using the FindFirstFileA API and stores it as a variable for later use.

Next, it gets the ftCreationTime of the Windows folder using the GetWindowsDirectoryA and FindFirstFileA APIs, and tries to compare it with the hard-coded array of the Windows timestamp. If a match is found, it will not infect the system. It also tries to determine whether the system is based on *Linux* running a *Windows* compatibility layer (Wine) by checking for the existence of the following registry key:

HKEY_CURRENT_USER\Software\Wine

Hash of API Name	API function
E0762FEB	ntdll.ZwAllocateVirtualMemory
E9D6CE5E	ntdll.ZwFreeVirtualMemory
183679F2	ntdll.LdrLoadDll
C97C1FFF	kernel32.GetProcAddress
B1866570	kernel32.GetModuleHandleA
251097CC	kernel32.ExitProcess
A9290135	ADVAPI32.RegCloseKey
DB58C4B1	ADVAPI32.RegOpenKeyA
8DE030C1	ADVAPI32.RegCreateKeyA
9F19DC76	ADVAPI32.SystemFunction036
9C343B9A	ole32.CreateStreamOnHGlobal
C1698B74	USER32.FindWindowA
F8EE80EC	USER32.GetWindowThreadProcessId
201D0DD6	USER32.wsprintfW
D4C9B887	USER32.wsprintfA
B09315F4	kernel32.CloseHandle
CCE95612	kernel32.WriteFile
553B5C78	kernel32.CreateFileA
09CE0D4A	kernel32.VirtualAlloc
D82BF69A	kernel32.FindClose
C9EBD5CE	kernel32.FindFirstFileA
A6BAB89C	kernel32.ExpandEnvironmentStringsA
636B1E9D	kernel32.GlobalFree
BB44F8FE	kernel32.GlobalUnlock
A9CA445B	kernel32.GlobalLock
7FBC7431	kernel32.GlobalAlloc
919B6BCB	kernel32.DeleteFileA
E058BB45	kernel32.WaitForSingleObject
AE672C54	kernel32.GlobalFindAtomA
9B3D61A0	kernel32.GetProcessId
ADECCF27	kernel32.GlobalAddAtomA
BA83C4F6	kernel32.GetComputerNameA
FFF372BE	kernel32.GetWindowsDirectoryA
251097CC	kernel32.ExitProcess
40F6426D	kernel32.GetProcessHeap
B0F6E8A9	ntdll.RtlFreeHeap
5EDB1D72	ntdll.RtlAllocateHeap
5B4219F8	kernel32.GetTickCount
72F11E39	kernel32.MultiByteToWideChar
9A80E589	kernel32.WideCharToMultiByte
CEF2EDA8	kernel32.Sleep
F3771641	kernel32.GetTempPathA
FF808C10	kernel32.CreateRemoteThread
4F58972E	kernel32.WriteProcessMemory
E62E824D	kernel32.VirtualAllocEx
DF27514B	kernel32.OpenProcess
A851D916	kernel32.CreateProcessA
E2892308	kernel32.GetTempFileNameA
EB64C435	kernel32.GetNativeSystemInfo
DF87764A	kernel32.GetVersionExA
08BFF7A0	kernel32.GetModuleFileNameA
3872BEB9	kernel32.ResumeThread
5DA70FB4	kernel32.SetThreadPriority
844FBD37	kernel32.InitializeCriticalSection
A3589103	ntdll.RtlDeleteCriticalSection
973CA9E2	gdiplus.GdipBitmapGetPixel
F7239947	gdiplus.GdipCloneImage
9533D257	gdiplus.GdiplusStartup
3402F03E	gdiplus.GdiplusShutdown
7E279C5A	gdiplus.GdipAlloc
23F273DC	gdiplus.GdipFree
D48C8AC5	gdiplus.GdipDisposeImage
2576C798	gdiplus.GdipCreateBitmapFromStreamICM
F78FFE59	gdiplus.GdipGetImageWidth
8EE7A83E	gdiplus.GdipCreateBitmapFromStream
4C50A4BD	gdiplus.GdipGetImageHeight
016505E0	WININET.InternetOpenUrlA
E5191D24	WININET.InternetCloseHandle
1AF971E4	WININET.InternetQueryDataAvailable
6CC098F5	WININET.InternetReadFile
DA16A83D	WININET.InternetOpenA

Figure 5: List of APIs gathered with their corresponding hashes.

```
MOV EAX,DWORD PTR FS:[30]          ; pointer to the PEB
MOV DWORD PTR SS:[EBP-4],EAX
MOV EAX,DWORD PTR SS:[EBP-4]
MOV EAX,DWORD PTR DS:[EAX+C]       ; get PEB->PEB_LDR_DATA
MOV EAX,DWORD PTR DS:[EAX+C]       ; get PEB->PEB_LDR_DATA.InLoadOrderModuleList (First entry)
```

Figure 4: Code snippet of getting InLoadOrderModuleList.

If this registry key exists, it will not infect the system, it will terminate and delete itself using the following command (where %% is the path and filename of the malware):

```
CMD /C SYSTEMINFO && SYSTEMINFO && SYSTEMINFO &&
SYSTEMINFO && SYSTEMINFO && DEL "%s"
```

It tries to get the processID of the explorer.exe process in the system by using a combination of the FindWindowA and GetWindowThreadProcessId APIs with the following parameters:

- lpClassName = "Shell_TrayWnd"
- lpWindowName = NULL

If this method fails, it will create a rundll32.exe process using the following command:

```
rundll32.exe shell32.dll,Control_RunDLL
```

It gets the processID of the newly created process using the CreateProcessA and GetProcessId APIs, then transfers its execution to the injected code in either the system's explorer.exe or the newly created rundll32.exe process using the CreateRemoteThread API and proceeds to terminate and delete itself from the system.

The malware checks whether it is already installed in the system using the following conditions:

- Finding the AtomName = "1234554321" using the GlobalFindAtomA API
- Opening the registry key:

  HKEY_CURRENT_USER\Software\Microsoft\[unique_string_through_hash_of_computer_name]

As shown in Figure 6, the malware sends a step-by-step status update to its C&C server.

It then activates InternetOpenA with the User-Agent:

```
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
Trident/4.0)
```

and InternetOpenUrlA with the following URL string structure:

```
http://207.36.---.49/report_[HEX1]_[HEX2]_[COUNTER]_
[STATUS]
```

Where:

- [HEX1] is the hex string of the computer name
- [HEX2] is the ftCreationTime.dwLowDateTime of the %USERPROFILE% folder
- [COUNTER] is the status counter of the procedure
- [STATUS] is the string that represents the status of the procedure used.

Step 2: Downloading the image with the encrypted message

The malware then attempts to download an image file from one of the following URLs:

- http://www.imagesup.net/?di=1514---3927715
- http://hostthenpost.org/uploads/a7f5c7e67-----8ff9dbfac6a3f28d3be.png

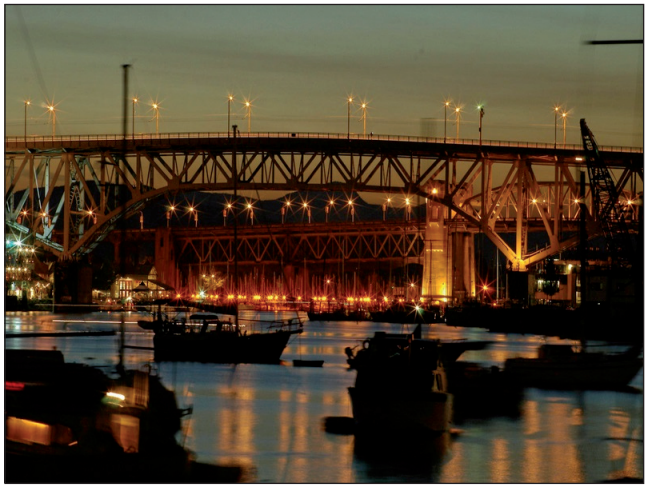

Figure 7: Gatak/Stegoloader's downloaded image file including hidden data.

Protocol	Info
HTTP	GET /report_[redacted]_0_started HTTP/1.1
HTTP	GET /report_[redacted]_1_already_ok HTTP/1.1
HTTP	GET /report_[redacted]_2_mark_ok HTTP/1.1
HTTP	GET /report_[redacted]_0_started HTTP/1.1
HTTP	GET /report_[redacted]_1_already_ok HTTP/1.1
HTTP	GET /report_[redacted]_2_mark_ok HTTP/1.1
HTTP	GET /report_[redacted]_3_http___www_[redacted]_com__di_15141253927715 HTTP/1.1
HTTP	GET /?di=15141253927715 HTTP/1.1
HTTP	GET /report_[redacted]_4_page_ok HTTP/1.1
HTTP	GET /report_[redacted]_5_image_size_ok HTTP/1.1
HTTP	GET /report_[redacted]_6_image_type_not_ok HTTP/1.1
HTTP	GET /report_[redacted]_7_payload_not_ok HTTP/1.1
HTTP	GET /report_[redacted]_8_http___[redacted]org_uploads_a7f5c7e6785598ff9dbfac6a3f28d3be_png
HTTP	GET /uploads/a7f5c7e6785598ff9dbfac6a3f28d3be.png HTTP/1.1
HTTP	GET /report_57494E5850_E6EA8CE3_9_page_ok HTTP/1.1
HTTP	GET /report_57494E5850_E6EA8CE3_10_image_size_ok HTTP/1.1

Figure 6: The malware sends a step-by-step status update to its C&C server.

After downloading the image file and before proceeding to the steganography algorithm to retrieve the hidden encypted data, the malware runs through the following integrity checklist:

1. It verifies that the image is a PNG file by checking that the header starts with '0x89PNG' bytes.
2. It checks whether it properly initializes *Windows* GDI+ via the GdiplusStartup API.
3. It checks whether it can create a bitmap object for the downloaded image to retrieve the pixel data using the CreateStreamOnHGlobal and GdipCreateBitmapFromStream APIs.

The malware then retrieves the hidden encrypted data from the image by getting the pixel data of the image. The pixel data serves as variables in a stream generator algorithm to form the hidden encrypted data. 'Pixel data' is the colour of the pixel at position (x,y) in a bitmap object. The malware uses a combination of the GdipGetImageHeight, GdipGetImageWidth and GdipBitmapGetPixel APIs to obtain the pixel data of the image.

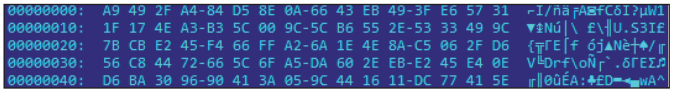

Figure 8: Image's hidden encrypted stream.

It decrypts the encrypted stream in the image with an RC4 algorithm and a hard-coded eight-byte key at offset 0x19 of the injected code in either explorer.exe or rundll32.exe (see below for description).

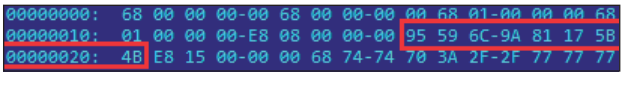

Figure 9: Start of injected code.

Next, it checks the CRC32 of the decrypted stream to verify the correctness of the resolved stream. This resolved stream is a shellcode. The shellcode contains:

- A command code located at offset 0x4 which instructs the injected remote code on how to process the hidden data.
- The length (located at offset zero DWORD size) and CRC32 value (located at offset [length_of_shellcode + 4] DWORD size) of the shellcode that serves as a variable for its CRC32 checker function.
- The hard-coded byte key (located at offset 0xb) that will be used to encrypt/decrypt the communicated message in the malware's C&C server.
- The hard-coded byte key ID of the image shellcode located at offset 0x21.

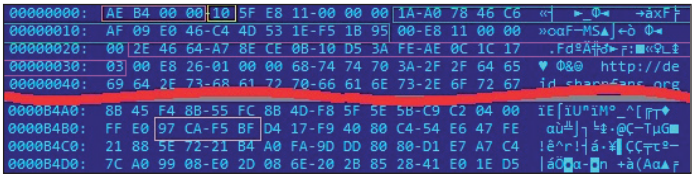

Figure 10: Image's hidden decrypted stream.

The command codes and their descriptions are as follows:

- 0x10: Execute payload code in memory
- 0x20: Create then execute payload code in [binary file]
- 0x21: Create then execute then delete payload code in [binary file]

where [binary file] has the form %temp%\~XX[random_number].tmp.

The image shellcode is a backdoor that communicates with the following command and control servers:

- http://deid.sharpfans.org/calibre/view?present=09---67
- http://bpp.bppharma.com/calibre/view?present=09---67
- http://reader.lifeacademyinc.com:80/encourage/help?pointed=85----4
- http://5.135.233.16:80/file/photos?handle=68--77
- http://cod.chezsimone971.com:80/encourage/help?pointed=85----4

The message sent by the image shellcode is encrypted with an RC4 algorithm prepended with a 16-byte key ID. The key ID serves as the RC4 key for the encrypted message. The structure of the decrypted message sent to the C&C server is shown in Figure 11.

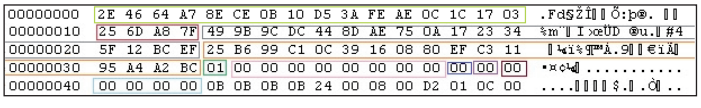

Figure 11: Structure of the decrypted message sent to the C&C server.

- @ Offset 0 = 16-byte key ID of the shellcode.
- @ Offset 0x10 = 4-byte CRC32 value starting at offset 0x14 until the end of the message.
- @ Offset 0x14 = 16-byte command ID of the shellcode, randomly generated using the SystemFunction036 API.
- @ Offset 0x24 = 16-byte function ID of shellcode, randomly generated using the SystemFunction036 API at the start of the shellcode execution.
- @ Offset 0x34 = byte header for the message.
- @ Offset 0x35 = 8-byte session ID of the message, which initializes to zero in the first contact with the command and control server.

- @ Offset 0x3D = BOT command.
- @ Offset 0x3E = success/error flag, set to zero for success of operation.
- @ Offset 0x3F = information flag, set to one if information length is greater than 1,024 bytes.
- @ Offset 0x40 = length of information.
- @ Offset 0x44 = start of the information.

Information greater than 1,024 bytes will be compressed with the LZMA algorithm.

The information sent to the C&C server includes sensitive information stolen from the infected system in the following format: "{"[Information Description]": "[Base64 Encoded Information]"}".

Figure 12 shows the structure of the decrypted message received from the C&C server.

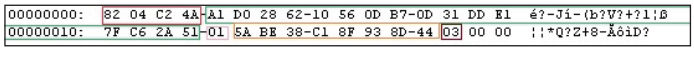

Figure 12: Structure of the decrypted message received from the C&C server.

- @ Offset 0 = 4-byte CRC32 value starting at offset 4 until the end of the message.
- @ Offset 4 = 16-byte command ID of the C&C server.
- @ Offset 0x14 = byte header for the message.
- @ Offset 0x15 = 8-byte session ID of the message.
- @ Offset 0x1D = BOT command.

Examples of BOT commands include, but are not limited to, the following:

- 0x01 = No operation, just contact the C&C server
- 0x02 = Execute payload via shellcode or [binary file]
- 0x03 = Retrieve system information (e.g. InternalIP, DomainName, Processes, etc.)
- 0x04 = Retrieve software installed
- 0x05 = Retrieve web browser history
- 0x64 = Execute shellcode
- 0xDC = Retrieve Windows folder timestamp.

SUMMARY

Gatak/Stegoloader may install other modules or malware for stealing sensitive information. Some variants have been found to install the Vundo malware family, which installs adware, ransomware and scareware. The improved digital steganography technique demonstrated by this piece of malware will surely be further adopted and/or enhanced by cybercriminals thanks to its efficiency in hiding code.

REFERENCES

[1] https://en.wikipedia.org/w/index.php?title=Duqu&oldid=712083675.

[2] https://en.wikipedia.org/w/index.php?title=Zeus_(malware)&oldid=711753219.

[3] http://www.securityweek.com/information-stealing-malware-%E2%80%9Cstegoloader%E2%80%9D-hides-image-file.

Editor: Martijn Grooten

Chief of Operations: John Hawes

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca, Ionuţ Răileanu, Chris Stock

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

Consultant Technical Editor: Dr Morton Swimmer

Tel: +44 (0)1235 555139 Fax: +44 (0)1865 543153

Email: editorial@virusbtn.com Web: https://www.virusbulletin.com/