virus BULLETIN Covering the global threat landscape

API-EPO

Raul Alvarez
Fortinet, Canada

Most file infectors attempt to avoid heuristic detection by implementing an EPO (entry-point obscuring) technique. EPO confuses anti-virus scanners by emulating the instructions from the beginning of the executable – which makes it look as if it is still operating within the host file. The technique varies slightly from malware to malware.

Expiro uses an EPO technique, as discussed in [1], in which it replaces a block of code from the entry point of the executable with its malicious binaries, which contain the initial decryption algorithm. This article focuses on an old file infector, but one which is still active in the wild. W32/Daum is a simple file infector, but it is worth looking at more closely for its unique EPO methodology.

SIMPLE DECRYPTION

Before we go any further, let's talk about the decryption algorithm.

A file infected with Daum contains an extra section named '.dspack', which contains the encrypted malware body as well as a simple decryptor. The decryptor uses a simple XOR instruction.

Initially, the decryptor looks for '00', which marks the beginning of the encrypted block of code, and replaces it with 'BS'. This is followed by getting the low-order byte of the SizeOfOptionalHeader field from the PE header. This byte is the decryption key used to decrypt 7,885 (0x1ECD) bytes of encrypted malware code.

Another pass using the same decryption routine produces the string 'kernel32.dll'. Daum converts this string to its Unicode equivalent, then uses the string while it parses the PEB (Process Environment Block) structure to locate the imagebase of kernel32.dll. Afterwards, the malware re-encrypts the string to remove any trace of it.

This is followed by decrypting the string 'CreateRemoteThread' and converting it to Unicode. The malware then parses the kernel32 export table to resolve the address of the CreateRemoteThread API. As with the 'kernel32.dll' string, the 'CreateRemoteThread' string is re-encrypted to its original form. Finally, Daum creates a new local thread using the CreateRemoteThread API with an hProcess parameter of INVALID_HANDLE_VALUE. (There is a similar effect if the CreateThread API is used.)

While the new thread executes, the main thread enters an infinite loop, checking to see if the marker has changed from 'BS' to 'SS'.

LOCAL THREAD

The initial routine of the newly created thread resolves all of the required APIs.

First, Daum decrypts all of the API names that will be used throughout the malware execution. This is followed by parsing the PEB again to get the imagebase of kernel32.dll.

Using kernel32's imagebase, the malware parses the kernel32 export table to resolve the APIs that correspond to the list of decrypted API names.

Once the kernel32-related APIs have been resolved, the imagebase for ws2_32.dll is produced by using the GetModuleHandleA API. Note that the GetModuleHandleA API produces the imagebase if the library is already loaded within the process. For user32.dll, the LoadLibraryA API is used. The APIs related to both DLLs are resolved using the GetProcAddress API.

Once all of the APIs have been resolved, the API names are re-encrypted to their original form and the marker is changed to 'SS' – in effect, releasing the main thread from its infinite loop.

Finally, Daum returns all of the original addresses of the APIs to the original host file. (What happens to these API addresses will be discussed later.)

By this point, the original executable should be running in parallel with the malware.

CODE INJECTION

Still within the context of the new thread, and while the original executable is running, the malware looks for a chance to inject its code into the explorer.exe process.

Daum parses the list of running processes in a quest to find 'explorer.exe', using a combination of the CreateToolhelp32Snapshot, Process32Next and Process32First APIs. The malware uses a simple call to the lstrcmp API to compare the name of each process on the list against 'explorer.exe'.

Once the explorer.exe process has been found, Daum opens it using the OpenProcess API with the parameter PROCESS_ALL_ACCESS. The malware reads two characters from a specific location in the DOS header area of the explorer.exe process and converts one of the characters to lower case. For *XP* machines, the specific characters are 'S' and (space) – in effect, converting ‘S’ to ‘s’. Then, it writes the lower-case ‘s’ back to the explorer.exe process, using the WriteProcessMemory API.

The lower-case ‘s’ serves as an infection marker for the process. If, by any chance, a non-infected explorer.exe process already has a lower-case ‘s’ in that specific location, the code injection will not be performed.

If everything is in order, Daum will write 8,181 (0x1FF5) bytes of malicious code to a newly allocated section of memory in explorer.exe, using another call to the WriteProcessMemory API.

To finalize the code injection routine, a remote thread is created within the explorer.exe process using the CreateRemoteThread API.

Daum then sleeps for 300,000 (0x493E0) ms before performing the code injection routine again. For the newly infected explorer.exe process, this routine will not be completed if it finds the process infection marker.

REMOTE THREAD

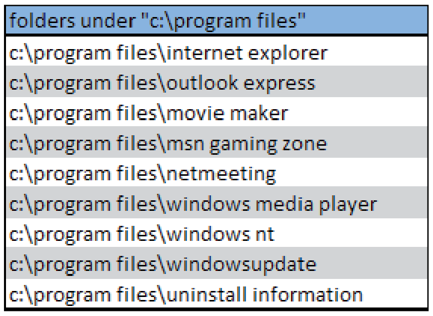
Following the remote thread in explorer.exe, Daum executes similar routines to the local thread. It decrypts the API names, gets the imagebases of the DLLs, and resolves all of the API addresses it needs. It also re-encrypts all of the API names.

After the preparations, Daum gets the list of available drives in the system by calling the GetLogicalDrives API. Then it creates another thread using the CreateRemoteThread API, running within the explorer.exe process.

The new thread loads shell32.dll using the LoadLibraryA API. This is followed by resolving the address of the SHGetFolderPathA API by calling the GetProcAddress API.

Daum generates a list of folders to avoid during the infection routine. First, it calls the SHGetFolderPathA API with a special CSIDL parameter. Then, it converts the pathnames to lower case and stores them in memory. Figure 1 shows the CSIDL parameters and the possible pathnames generated.

The malware stores several copies of ‘c:\program files’ for an additional set of pathnames. It stores additional paths in memory by adding strings to ‘c:\program files\’, producing the list of paths shown in Figure 2.



folders under "c:\program files"
c:\program files\internet explorer
c:\program files\outlook express
c:\program files\movie maker
c:\program files\msn gaming zone
c:\program files\netmeeting
c:\program files\windows media player
c:\program files\windows nt
c:\program files\windowsupdate
c:\program files\uninstall information

Figure 2: List of folders under c:\program files.

After setting up the folders and paths to avoid, the malware traverses the folders of a given drive acquired earlier from a call to the GetLogicalDrives API. Daum checks if the path appears in its list of paths to avoid. If the path is in the list, the malware skips the infection routine and moves onto the next path.

INFECTION ROUTINE

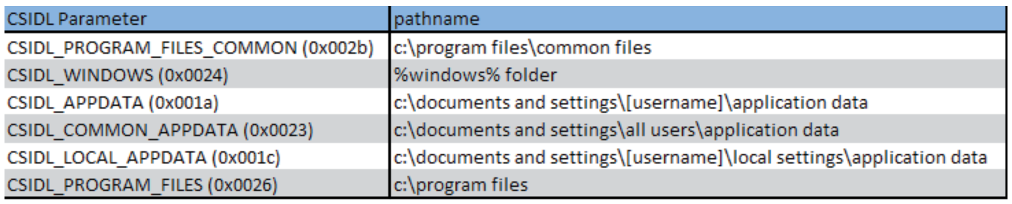
When a pathname has passed the inspection, Daum searches for host files to infect using a combination of calls to the FindFirstFileA and FindNextFileA APIs. It searches for all files available in the system, including folders, to enable the malware to skip the list of pathnames discussed earlier.

Next, the malware checks whether the host file has the extension name ‘.exe’ or ‘.scr’. If the file does have one of these extension names, Daum prepares for infection by changing the protection of the malware body in the explorer.exe process to PAGE_EXECUTE_READWRITE (0x40) using the VirtualProtectEx API.

This is followed by opening the new host executable file with GENERIC_READ | GENERIC_WRITE access using the CreateFileA API. It reads the first 63 (0x3f) bytes of the host file into memory using the ReadFile API, and checks if it has the ‘MZ’ marker.

From the MZ header, Daum sets the file pointer to the PE header (MZ+3C) and reads 1,024 (0x400) bytes using another call to the ReadFile API.

If the ‘PE’ marker is present, Daum will save the value of the file’s imagebase, the size of optional header, the relative



CSIDL Parameter	pathname
CSIDL_PROGRAM_FILES_COMMON (0x002b)	c:\program files\common files
CSIDL_WINDOWS (0x0024)	%windows% folder
CSIDL_APPDATA (0x001a)	c:\documents and settings\[username]\application data
CSIDL_COMMON_APPDATA (0x0023)	c:\documents and settings\all users\application data
CSIDL_LOCAL_APPDATA (0x001c)	c:\documents and settings\[username]\local settings\application data
CSIDL_PROGRAM_FILES (0x0026)	c:\program files

Figure 1: The CSIDL parameters used.

virtual addresses of the import tables, the section alignment, file alignment, and the number of sections.

As mentioned earlier, the low-order byte of the SizeOfOptionalHeader field serves as the key used in the decryption routine.

Meanwhile, the information relating to the first section is saved into a newly allocated section of memory, including section name, virtual size, relative virtual address, size of raw data, pointer to raw data, and the section's characteristics.

Then, the information for the other sections in the host file is saved in a separate memory allocation for each. A linked-list structure is created, in which each section knows the location of the next one.

To avoid reinfection, Daum parses the names of the sections to look for '.dspack', the name of the additional section inserted by the malware into the infected file. The whole infection routine will be skipped if '.dspack' is found.

EPO TECHNIQUE

In preparation for the EPO technique, Daum parses the linked list of section information. It searches for the section that contains the import table by checking the import table's RVA (relative virtual address) against the computed size and RVA of each section. Once the malware knows in which section the import table is located, it will get the pointer to raw data (file offset) of that section. This file offset will be used later.

Daum creates a new section named '.dspack'. The virtual size and the size of raw data are 8,181 (0x1ff5) bytes. The relative virtual address and pointer to raw data are computed beforehand, based on the size of the host file. Finally, the section's characteristics are set to (0xC0000060) WRITABLE | READABLE | CODE | INITIALIZED DATA.

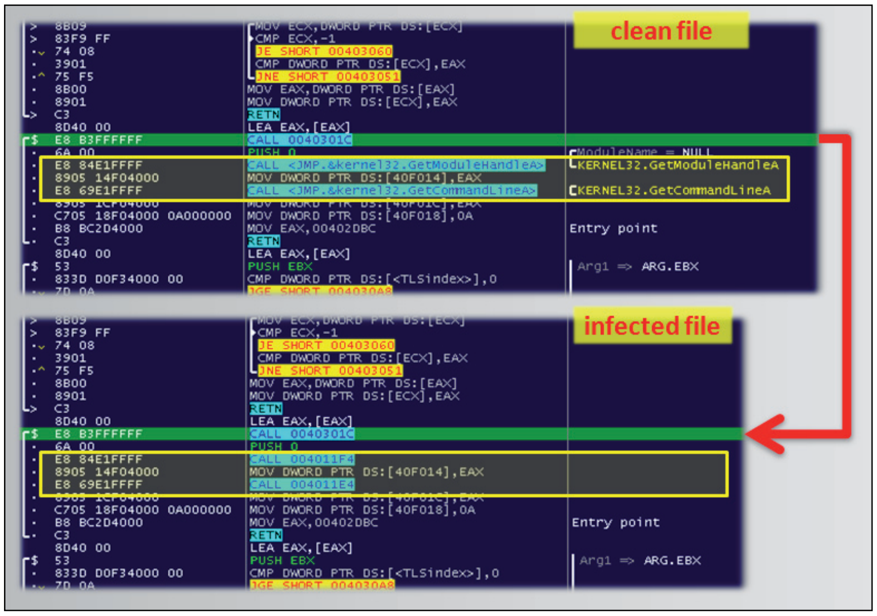
The malware increases the NumberOfSections field, from the PE header, by one. Then, it adds .dspack's information to the linked list of section information. Note that this data has not yet been committed to the physical file.

After setting up the required structure and data for the .dspack section, Daum reads the first 20 bytes of the host file's import table into memory using a combination of the SetFilePointer and ReadFile APIs. The malware computes the file offset of the first thunk from the import table structure. The first thunk is the pointer to the first imported API address of the host file.

Once the file offset of the first thunk has been acquired, the malware reads 4,096 (0x1000) bytes of the thunk table that will contain the imported API addresses. The thunk table is an array of DWORD locations that will hold the imported API addresses. The API addresses will be resolved once the host file is executed.

Daum only considers the thunk table of the first library (DLL), and copies it to the physical host file using a call to the WriteFile API. The thunk table is written at the end of the host file, which will be the starting location of the .dspack section.

This is followed by computing the offset of Daum's initial routine (the 'simple decryption' discussed at the beginning of this article), then the thunk table's addresses are overwritten



```
 >  8B09                     MOV ECX,DWORD PTR DS:[ECX]
 >  83F9 FF                  CMP ECX,-1
 .v 74 08                    JE SHORT 00403060
 .  3901                     CMP DWORD PTR DS:[ECX],EAX
 .^ 75 F5                    JNE SHORT 00403051
 .  8B00                     MOV EAX,DWORD PTR DS:[EAX]
 .  8901                     MOV DWORD PTR DS:[ECX],EAX
 >  C3                       RETN
    8D40 00                  LEA EAX,[EAX]
 $  E8 B3FFFFFF              CALL 0040301C
 .  6A 00                    PUSH 0                                     ModuleName = NULL
 .  E8 84E1FFFF              CALL <JMP.&kernel32.GetModuleHandleA>      KERNEL32.GetModuleHandleA
 .  8905 14F04000            MOV DWORD PTR DS:[40F014],EAX
 .  E8 69E1FFFF              CALL <JMP.&kernel32.GetCommandLineA>       KERNEL32.GetCommandLineA
 .  8905 1CF04000            MOV DWORD PTR DS:[40F01C],EAX
 .  C705 18F04000 0A000000   MOV DWORD PTR DS:[40F018],0A
 .  B8 BC2D4000              MOV EAX,00402DBC                           Entry point
 .  C3                       RETN
    8D40 00                  LEA EAX,[EAX]
 $  53                       PUSH EBX                                   Arg1 => ARG.EBX
 .  833D D0F34000 00         CMP DWORD PTR DS:[<TLSindex>],0
 .v 7D 0A                    JGE SHORT 004030A8
```

```
 >  8B09                     MOV ECX,DWORD PTR DS:[ECX]
 >  83F9 FF                  CMP ECX,-1
 .v 74 08                    JE SHORT 00403060
 .  3901                     CMP DWORD PTR DS:[ECX],EAX
 .^ 75 F5                    JNE SHORT 00403051
 .  8B00                     MOV EAX,DWORD PTR DS:[EAX]
 .  8901                     MOV DWORD PTR DS:[ECX],EAX
 >  C3                       RETN
    8D40 00                  LEA EAX,[EAX]
 $  E8 B3FFFFFF              CALL 0040301C
 .  6A 00                    PUSH 0
 .  E8 84E1FFFF              CALL 004011F4
 .  8905 14F04000            MOV DWORD PTR DS:[40F014],EAX
 .  E8 69E1FFFF              CALL 004011E4
 .  8905 1CF04000            MOV DWORD PTR DS:[40F01C],EAX
 .  C705 18F04000 0A000000   MOV DWORD PTR DS:[40F018],0A
 .  B8 BC2D4000              MOV EAX,00402DBC                           Entry point
 .  C3                       RETN
    8D40 00                  LEA EAX,[EAX]
 $  53                       PUSH EBX                                   Arg1 => ARG.EBX
 .  833D D0F34000 00         CMP DWORD PTR DS:[<TLSindex>],0
 .v 7D 0A                    JGE SHORT 004030A8
```

Figure 3: Comparison between the clean and infected files' API calls.

with offsets pointing to the initial routine. This effectively changes the jump locations of all imported API calls to the single address. Note that the APIs affected are only those related to the first DLL used by the host file.

To make sure that the imported APIs will still be resolved, Daum modifies the location of the first thunk in the import table to point to the .dspack section. Again, note that the first block of the .dspack section contains the original thunk table of the host file.

The actual EPO technique is the replacement of all API calls related to the first library of the import table with a call to the initial routine of the malware. Daum doesn't know which API will be called first, so in order to make sure that the malware code is triggered, all API calls are changed.

The original API addresses will be replaced after Daum has performed all the necessary routines (see Figure 3).

FINISHING TOUCHES

After setting up the EPO technique, the infection routine continues by copying 8,181 (0x1ff5) bytes of malware code to the newly allocated memory using a combination of the VirtualAllocEx and WriteProcessMemory APIs.

Daum encrypts 7,885 (0x1ECD) bytes of the same malware code using a simple XOR instruction. The key is taken from the low-order byte of the SizeOfOptionalHeader field from the PE header of the host file.

After encryption, the malware code is written to the end of the host file using the WriteFile API.

Next, Daum reads the value of the SizeOfImage field from the host's PE header, using the ReadFile API. It adds 0x1ff5 to increase the size of the image, and checks whether it has the proper file alignment. The new image size is then written back to the host file.

This is followed by allocating a block of memory to collect all of the section tables' data, taken from the linked list of section information, including the new '.dspack' section. Then it writes it back to the host file.

For the final touch, the NumberOfSections field in the host's PE header is also updated by increasing it by one.

Daum frees up all allocated memories and closes the handle for the newly infected file. It will sleep for 77 seconds then look for more files to infect.

WRAP UP

Strong encryption algorithms, along with anti-debugging and anti-analysis techniques, are some of the defensive measures used by malware to make analysis more difficult. Daum proves that encryption with a simple EPO technique still works.

By re-encrypting its strings, e.g. API names, Daum hides some of its details from post-infection analysis. While the EPO technique looks more like IAT hooking, for Daum, it takes only one API call to initiate its malicious routine. Once the routine is initiated, the correct addresses of the APIs will be replaced. Unlike IAT hooking, the hook remains even if the API has already been called.

From an external perspective, file infection and other activities performed by different malware might look very similar, but on the inside, there is always a hint of diversity.

Stay safe!

REFERENCE

[1] Alvarez, R. Not Expir-ed yet. Virus Bulletin, March 2014, p.17. https://www.virusbtn.com/virusbulletin/archive/2014/03/vb201403-Expiro.

Editor: Martijn Grooten
Chief of Operations: John Hawes
Security Test Engineers: Scott James, Tony Oliveira
Sales Executive: Allison Sketchley
Editorial Assistant: Helen Martin
Perl Developer: Tom Gracey
Consultant Technical Editors: Dr Morton Swimmer, Ian Whalley

© 2014 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England.
Tel: +44 (0)1235 555139. Fax: +44 (0)1865 543153
Email: editorial@virusbtn.com
Web: http://www.virusbtn.com/