virus BULLETIN Covering the global threat landscape

NESTING DOLL: UNWRAPPING VAWTRAK

Raul Alvarez
Fortinet, Canada

A lot of malware families perform their malicious activities within a single executable. Complete with body armour and shields, they load themselves into memory, decrypt just enough binaries, and then carry out their malicious actions under the protection of encryption and other stealth techniques. Thus it is hard to get a complete binary dump for simple analysis.

It is less common for a piece of malware to wrap itself in layers and at the heart of those layers, expose a simple complete binary executable – but that is exactly what a piece of malware known as Vawtrak does. In this article we will unravel Vawtrak's layers, each of which gives rise to the next, just like a nesting doll.

Vawtrak, which is also known as Neverquest or Snifula, is a banking trojan that recently made the headlines because of its high level of sophistication. Initially it targeted only Japanese systems, but it has recently broadened its geographic scope [1]. Vawtrak spreads through drive-by downloads, attached to spam emails, and is also downloaded by other malware.

Other articles, such as [2], have looked at how the malware communicates with its C&C server and how it targets various banks. This paper looks specifically at how the malware installs itself persistently on a machine, frustrating both researchers and anti-virus products along the way.

LAYER 1 (THE OUTER DOLL)

In this section, we will look into the different algorithms performed by the outer layer (i.e. the malware itself) and at how it generates the executable binary that forms the next layer. The outer layer also contains the image file that is displayed once the malware is executed, to trick the victim into believing that is all the file does.

Simple anti-debugging trick

Anti-debugging tricks are common in modern malware. The variant of Vawtrak we looked at uses what appears to be a simple anti-debugging trick. There is nothing significant in its execution, and it doesn't have a sneaky trick up its sleeve. However, when we look at how it is delivered, we see that it can easily stop heuristic scanning of the executable.

In its initial execution, Vawtrak calls the following APIs: GetModuleHandleA, InitCommonControlsEx and GetCommandLineA. None of these suggest that anything malicious is going to take place.

These instructions are followed by 1,600 (0x640) zeros ('0'), where each pair of zeros is interpreted as 'ADD BYTE PTR DS:[EAX],AL', an irrelevant instruction that is repeated 800 times. This is a simple trick that could stop a simple anti-virus engine from emulating the malware.

Next, Vawtrak gets the address of the PEB (Process Environment Block). The third byte of the PEB is the 'BeingDebugged' flag, which has a value of TRUE if the process is run within the context of a debugger.

The malware doesn't check specifically for the 'BeingDebugged' flag byte. Instead, it checks for the DWORD value from the third byte of the PEB, which contains the 'BeingDebugged' flag, the 'SpareBool' flag (reserved), and 'Mutant' DWORD value (reserved). Since the 'Mutant' value is a DWORD, only half of it is checked.

If the malware is being debugged, the DWORD value from the location PEB+2 will contain 0xFFFF0001. If this is not the case, the malware will continue execution.

After performing its anti-debugging check, the malware parses the stack memory to locate a return address that is located within kernel32.dll. Another check is then performed against the 'BeingDebugged' flag – in case the first anti-debug trick was unsuccessful. After performing a series of simple computations, the ImageBase of kernel32.dll is found.

String generator and API resolution

After finding kernel32's ImageBase, the malware resolves the APIs it requires by performing the following routine:

1. A simple string generator is used to produce the names of the APIs and libraries that it needs.

   The string generator starts by pushing DWORD values into the stack memory and adds each DWORD to another group of DWORD values. These DWORDs are constant in nature, pre-computed for specific strings such as 'GetModuleFileNameA'.

2. The malware parses the export table of the kernel32.dll library to look for an API name that matches the aforementioned string. When a match is found, the index of the API name is used to locate the address of the corresponding API.

Using the above routine, Vawtrak also resolves the following APIs: GetModuleFileNameA, CreateFileA, SetFilePointer, CloseHandle, ReadFile, WriteFile, GetTempPathA, LoadLibraryA and lstrcat.

Using the newly resolved LoadLibraryA API, Vawtrak loads the shell32.dll library into memory. This library name is also generated using the string generator. Using the technique described above, the ShellExecuteA API is also resolved from the shell32.dll library.

Simple API calling

When a regular application performs an API call, it does so by pushing all required parameters to the stack memory and then calling the API.

When, throughout its three layers, Vawtrak performs an API call, it first pushes all the required parameters to the stack, then it pushes the address of the API. Finally, it executes a RETN instruction, effectively jumping to the last value pushed, which is the address of the API.

At random locations between these instructions, NOP, DEC EDX and INC EDX instructions can be seen, which is another trick to make detection of the malware more difficult.

Main module

After a series of calls to the CreateFileA, SetFilePointer and ReadFile APIs that yield no meaningful results (which could either be a bug, or these calls could be intended as garbage code), Vawtrak gets the pathname of the current module using a call to the GetModuleFileNameA API and opens this path for reading using a call to the CreateFileA API.

This is followed by reading 260 (0x104) bytes from the physical file using a combination of the SetFilePointer and ReadFile APIs. These bytes hold the encrypted filename 'mainOUT-crypted-5.exe'[1], which will be used later for the dropped file. The filename is obfuscated using the single-byte XOR key 0xAA.

Using a combination of the GetTempPathA, lstrcat and CreateFileA APIs, Vawtrak creates the file '%temp%\ mainOUT-crypted-5.exe' with GENERIC_WRITE access.

To fill this newly created file, the malware needs to decrypt a large block of data from the 'overlay' area of the original malware file. (The 'overlay' area of an executable file is not loaded into memory when the operating system executes the file.)

Vawtrak's overlay area holds an encrypted copy of the executable binary that is used in the next layer. It is to be transferred and decrypted into the malware's virtual memory space.

Decryption and the dropped executable file

Initially, the malware reads 1,024 (0x400) bytes of data from the overlay area into memory. Then it decrypts the bytes using a simple decryption algorithm with four byte-wise instructions: 'SUB AH, 0x63', 'XOR AH, 0x42', 'XOR AH, 0x63' and 'NOT AH'. A variable number of NOP instructions are embedded in-between these instructions.

Upon decryption, the newly decrypted bytes are written to the previously created file, using the WriteFile API.

This routine is repeated until the full overlay area has been read and the decrypted file has been stored. It is then closed using the CloseHandle API.

Finally, Vawtrak executes '%temp%\ mainOUT-crypted-5.exe' by calling the ShellExecuteA API.

Dropped image file

While '%temp%\ mainOUT-crypted-5.exe' runs in the background, Vawtrak reads another 260 (0x104) bytes from the original malware into memory. Using the same simple XOR key, 0xAA, it generates the string 'Diana-23.jpg'. The previously discussed set of instructions is then used to create an image file with this name in the temporary folder.

Finally, Vawtrak displays the image by calling the ShellExecuteA API; the operating system will automatically associate the file with the default image viewer. Figure 1 shows part of the image which is displayed once the main malware is executed. It also resembles the icon used by the original malware.

After displaying the image, the malware tries to generate another file but there seems to be a bug in generating the third filename, causing this part to fail.

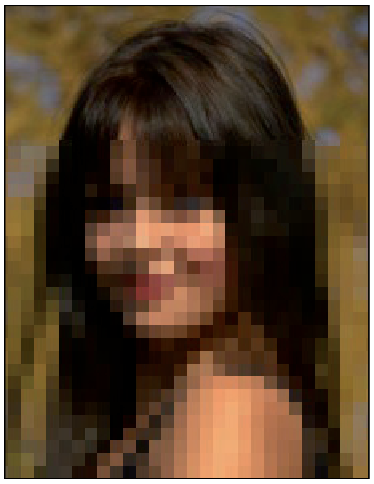


Figure 1: Pixellated and cropped version of the image displayed.

LAYER 2 (THE SECOND DOLL)

In this section, we will discuss the algorithms performed during the execution of the '%temp%\ mainOUT-crypted-5.exe' file

[1] The same filename has been observed among various samples, but it is possible that other variants use different filenames.

(the second 'doll' in our nesting doll analogy), which has been dropped by the outer layer.

Executing 'mainOUT-crypted-5.exe'

First, Vawtrak resolves the GetProcessHeap and HeapAlloc APIs using the API resolution algorithm discussed below. This algorithm is also used to resolve the rest of the APIs needed by the malware.

API resolution algorithm

The API resolution algorithm starts with a series of function calls to locate the initial location of the PEB (Process Environment Block). Once the PEB has been located, Vawtrak parses every module name through the PEB's linked list structure. Each module name is hashed using a simple mathematical calculation. Then, every computed hash value is compared against the hard-coded hash value (0xD56131B3) of 'kernel32.dll'. Once the correct hash is found, the ImageBase of kernel32.dll is derived from the current structure of the PEB.

Next, the malware decrypts the name of the required API using a rotating key string (0xFA A7 E9 F4 44 9C DF 43 5D C8 FD) similar to the one used during the decryption of the large data block (which will be discussed in the next section).

Then Vawtrak gets the ImageBase of the ntdll.dll library using the same routine as was used to find the ImageBase of kernel32.dll; in this case the hard-coded hash value is 0xA6196EA7. The same routine is also used to find the address of the LdrGetProcedureAddress API (hash value 0xB110618C).

Finally, the LdrGetProcedureAddress API is used to get the addresses of the APIs needed by Vawtrak (e.g. GetProcessHeap and HeapAlloc).

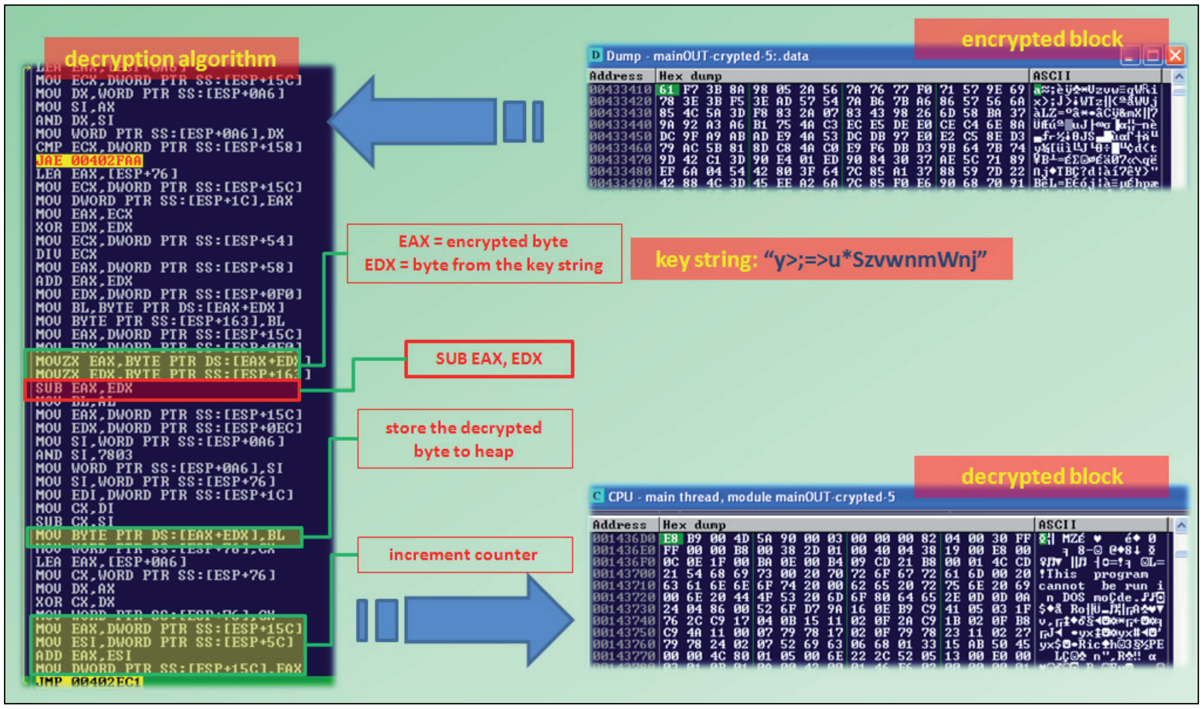
Decrypting the large data block

Vawtrak allocates 200,192 (0x30E00) bytes of heap memory using a combination of the GetProcessHeap and HeapAlloc (which is similar to RtlAllocateHeap) APIs. It decrypts and copies a large chunk of memory from the '.data' section of the malware.

The malware decrypts the aforementioned large block of data using the key string 'y>;=>u*SzvwnmWnj' (0x79 3E 3B 3D 3E 75 2A 53 7A 76 77 6E 6D 57 6E 6A).

The decryption algorithm looks complicated as a result of there being garbage code embedded within the routine. However, on looking closely, one sees that the only relevant computation is the 'SUB EAX, EDX' instruction. It simply subtracts the key byte (taken from the rotating key) from the encrypted byte. The rest of the instructions (the majority) are irrelevant to the actual algorithm.

This decryption algorithm is similar to the one discussed earlier for decrypting the API name. Figure 2 shows



```
LEA EAX,[ESP+0A6]
MOV ECX,DWORD PTR SS:[ESP+15C]
MOV DX,WORD PTR SS:[ESP+0A6]
MOV SI,AX
AND DX,SI
MOV WORD PTR SS:[ESP+0A6],DX
CMP ECX,DWORD PTR SS:[ESP+158]
JAE 00402FAA
LEA EAX,[ESP+76]
MOV ECX,DWORD PTR SS:[ESP+15C]
MOV DWORD PTR SS:[ESP+1C],EAX
MOV EAX,ECX
XOR EDX,EDX
MOV ECX,DWORD PTR SS:[ESP+54]
DIV ECX
MOV EAX,DWORD PTR SS:[ESP+58]
ADD EAX,EDX
MOV EDX,DWORD PTR SS:[ESP+0F0]
MOV BL,BYTE PTR DS:[EAX+EDX]
MOV BYTE PTR SS:[ESP+163],BL
MOV EAX,DWORD PTR SS:[ESP+15C]
MOV EDX,DWORD PTR SS:[ESP+0F0]
MOVZX EAX,BYTE PTR DS:[EAX+EDX]
MOVZX EDX,BYTE PTR SS:[ESP+163]
SUB EAX,EDX
MOV BL,AL
MOV EAX,DWORD PTR SS:[ESP+15C]
MOV EDX,DWORD PTR SS:[ESP+0EC]
MOV SI,WORD PTR SS:[ESP+0A6]
AND SI,7803
MOV WORD PTR SS:[ESP+0A6],SI
MOV SI,WORD PTR SS:[ESP+76]
MOV EDI,DWORD PTR SS:[ESP+1C]
MOV CX,DI
SUB CX,SI
MOV BYTE PTR DS:[EAX+EDX],BL
MOV WORD PTR SS:[ESP+76],CX
LEA EAX,[ESP+0A6]
MOV CX,WORD PTR SS:[ESP+76]
MOV DX,AX
XOR CX,DX
MOV WORD PTR SS:[ESP+76],CX
MOV EAX,DWORD PTR SS:[ESP+15C]
MOV ESI,DWORD PTR SS:[ESP+5C]
ADD EAX,ESI
MOV DWORD PTR SS:[ESP+15C],EAX
JMP 00402EC1
```

```
Dump - mainOUT-crypted-5:.data
Address   Hex dump                                         ASCII
00433410  61 F7 3B 8A 98 05 2A 56 7A 76 77 F0 71 57 9E 69  a≈;èÿ♣*Vzvw≡qWR i
00433420  78 3E 3B F5 3E AD 57 54 7A B6 7B A6 86 57 56 6A  x>;J>¡WTz╢{ª åWVj
00433430  85 4C 5A 3D F8 83 2A 07 83 43 98 26 6D 58 BA 37  àLZ=°â*•âCÿ&mX║7
00433440  9A 92 A3 A6 B1 75 4A C3 EC E5 DE E0 CE C4 6E 8A  ÜÆúª░uJ├∞σ▐α╬─nè
00433450  DC 9F A9 AB AD E9 4A 53 DC DB 97 E0 E2 C5 8E D3  ▄ƒ⌐½¡ΘJS▄█ùαΓ┼Ä╙
00433460  79 AC 5B 81 8D C8 4A C0 E9 F6 DB D3 9B 64 7B 74  y¼[üì╚J└Θ÷█╙¢d{t
00433470  9D 42 C1 3D 90 E4 01 ED 90 84 30 37 AE 5C 71 89  ¥B┴=ÉΣ☺φÉä07«\që
00433480  EF 6A 04 54 42 80 3F 64 7C 85 A1 37 88 59 7D 22  ∩j♦TBÇ?d|àí7êY}"
00433490  42 88 4C 3D 45 EE A2 6A 7C 85 F0 E6 90 68 70 91  BêL=EεójÍà≡µÉhpæ
```

```
CPU - main thread, module mainOUT-crypted-5
Address   Hex dump                                         ASCII
001436D0  E8 B9 00 4D 5A 90 00 03 00 00 00 82 04 00 30 FF  Φ╣ MZÉ ♥   é♦ 0
001436E0  FF 00 00 B8 00 38 2D 01 00 40 04 38 19 00 E8 00    ╕ 8-☺ @♦8↓ Φ
001436F0  0C 0E 1F 00 BA 0E 00 B4 09 CD 21 B8 00 01 4C CD  ♀♫▼ ║♫ ┤○═!╕ ☺L═
00143700  21 54 68 69 73 00 20 70 72 6F 67 72 61 6D 00 20  !This  program
00143710  63 61 6E 6E 6F 74 20 00 62 65 20 72 75 6E 20 69  cannot  be run i
00143720  00 6E 20 44 4F 53 20 6D 6F 80 64 65 2E 0D 0D 0A   n DOS moÇde.♪♪◙
00143730  24 04 86 00 52 6F D7 9A 16 0E B9 C9 41 05 03 1F  $♦å Ro╫Ü▬♫╣╔A♣♥▼
00143740  76 2C C9 17 04 0B 15 11 02 0F 2A C9 1B 02 0F B8  v,╔↨♦♂§◄☻☼*╔←☻☼╕
00143750  C9 4A 11 00 07 79 78 17 02 0F 79 78 23 11 02 27  ╔J◄ •yx↨☻☼yx#◄☻'
00143760  79 78 24 02 07 52 69 63 06 68 01 33 15 AB 50 45  yx$☻•Ric♠h☺3§½PE
00143770  00 00 4C 80 01 05 00 6E 22 2C 52 05 13 00 E0 00    LÇ☺♣ n",R♣‼ α
```

Figure 2: Decryption algorithm, part of the encrypted block and part of the decrypted block in heap memory. Only the highlighted instructions are relevant; the rest are garbage instructions.

the decryption algorithm with the relevant instructions highlighted.

Decompressing the decrypted data

The decrypted data is now located in the heap memory, albeit in a compressed form. In order to decompress it to its normal state, the malware needs to call the RtlDecompressBuffer API. This API is resolved using the same technique as was used to resolve the LdrGetProcedureAddress API, as explained previously.

After getting the address of the RtlDecompressBuffer API, Vawtrak prepares the heap by calling the RtlAllocateHeap API. This is followed by decompressing the decrypted data using the RtlDecompressBuffer API with the COMPRESSION_FORMAT_LZNT1 format.

Decompressing the data produces an executable file image with the proper MZ/PE header, the sections, and the rest of the complete unencrypted, unpacked executable.

Moving from heap to virtual memory

After the decompression, Vawtrak resolves the following APIs using the same API resolution algorithm as used earlier: LoadLibraryA, GetProcAddress, VirtualAlloc, VirtualProtect, VirtualFree and UnmapViewOfFile.

Then, Vawtrak parses the PE header of the newly decompressed executable file image to acquire the SizeOfImage value. After that, it allocates a section of virtual memory of this size by calling the VirtualAlloc API.

This is followed by using SizeOfRawData to compute the size of each section of the executable file image. Then, the malware copies the contents of each section into the newly allocated virtual memory (at address 0x8a0000).

After carefully copying the decompressed executable file image, byte by byte, section by section, adjusted using SizeOfRawData to obtain the physical file size, a researcher can dump it to a file.

Fixing the IAT

When a given application runs, the operating system is responsible for filling the IAT (Import Address Table) with the corresponding API addresses. However, if the binary file is loaded without proper Import Address translation, the IAT will contain RVAs (Relative Virtual Addresses) instead of the actual API addresses, and any call that is executed to an RVA will not work properly.

The malware's executable file image that was just copied to the virtual memory has an IAT that only contains RVAs. Vawtrak fills the IAT with the actual API addresses by performing the following routine:

Initially, the malware parses the PE header to locate the import table. This is followed by loading the first library that is found, kernel32.dll, using the LoadLibraryA API. Then it gets the address of each API using the GetProcAddress API and saves them to the corresponding locations in the IAT. Once all the kernel32-related APIs have been resolved, it goes back to load the rest of the libraries and thus resolves all the required APIs.

Overwriting 'mainOUT-crypted-5.exe'

Once the IAT has been filled with the proper API addresses, Vawtrak copies another 4,096 (0x1000) bytes of code to a different section of allocated virtual memory (0x890000) and transfers control to it.

Within the execution of the newly copied binaries, the malware changes the protection of the 'mainOUT-crypted-5.exe' module to PAGE_READWRITE using the VirtualProtect API. Then it clears the memory by overwriting the location at which 'mainOUT-crypted-5.exe' was stored with zeros, effectively removing it from memory.

This is followed by copying the contents of the virtual memory at 0x8a0000 to the original memory location of the 'mainOUT-crypted-5.exe' module. To recap: the content of the memory at 0x8a0000 is the decompressed executable image with properly filled IAT, that has just been generated.

After copying every byte from the virtual memory (0x8a0000), the section characteristics of the newly copied module are adjusted to take account of the new memory location.

Finally, the malware transfers control to the newly created module, which is the next layer of the malware.

LAYER 3 (THE THIRD DOLL)

In this section, we will look into the third layer (the third 'doll' in our nesting doll analogy) – an executable binary produced from the second layer. The new executable binary contains no protection at all. The execution of this module is straightforward. Neither decryption nor hashing are used, and there is no digging around in a garbage bin.

In the malware's current state, we can easily get a dumped copy of a complete executable binary file – but the malware is not done yet: it still has to reveal the last layer of the nesting doll.

Removing restrictions

Within the context of the third layer, Vawtrak's first order of business is to remove software restrictions. These restrictions can be set through the software restriction policies of the operating system.

Some of these policies are accessed via the registry key 'HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers'.

Using the RegSetValueExA API, the following subkeys are set by Vawtrak:

1. DefaultLevel = 0x40000 (unrestricted). This allows an administrator to define exceptions.
2. TransparentEnabled = 1. This skips DLL checking.
3. PolicyScope = 0. This makes policies applicable to all users.

After doing this, the malware deletes the key 'HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers\0\Paths\', using the SHDeleteKeyA API. This basically removes any restrictions on any file under the 'Path Rule', which identifies whether an application is restricted or not.

Anti-anti-malware

After removing restrictions for any application, Vawtrak tries to restrict possible execution of some anti-malware applications using the routine described below.

Initially, it gets the %system% folder using the GetSystemDirectoryA API to extract the drive name, such as drive 'C'. Then it adds the string ':\Program Files\' to the drive name. Finally, it adds the anti-malware software name, resulting in the following string format: '[drivename]:\Program Files\[anti-malware name]' (e.g. 'C:\Program Files\[anti-malware name]').

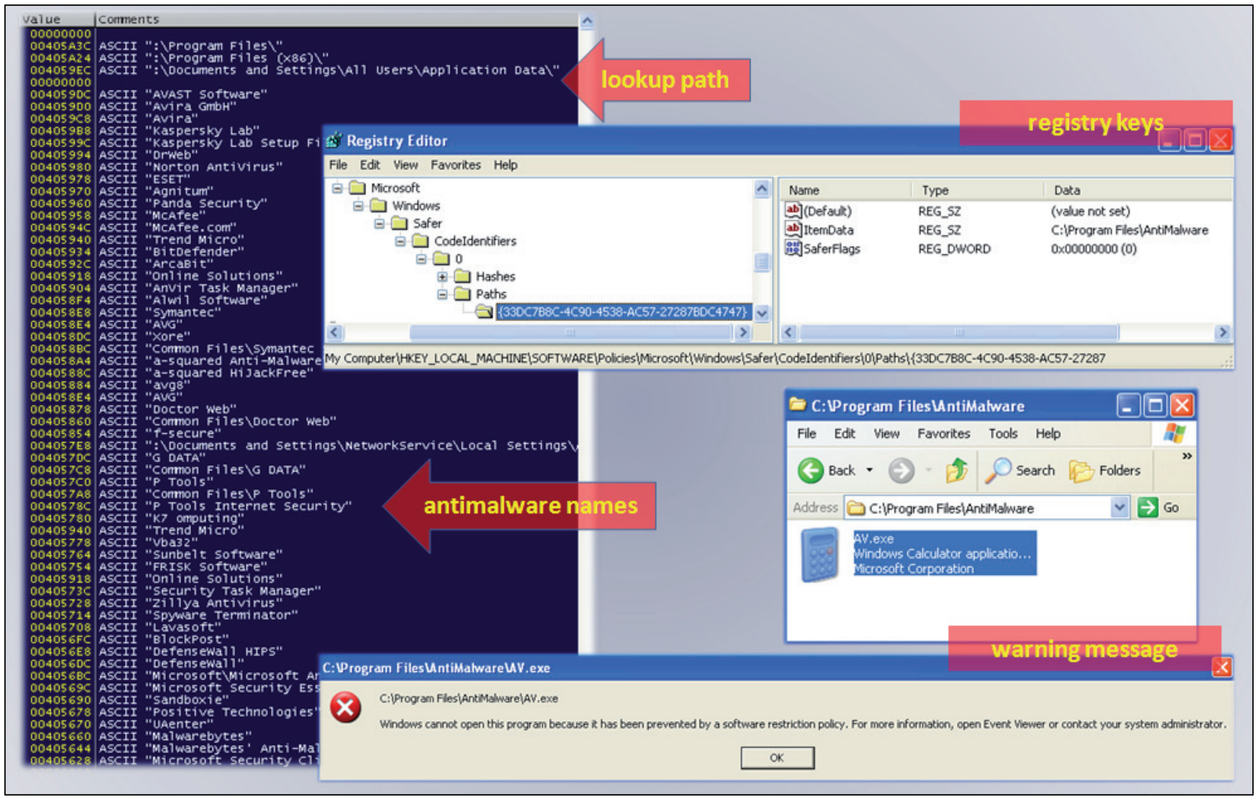
Vawtrak also looks into different folders by replacing the string ':\Program Files\' with either ':\Program Files (x86)\' or ':\Documents and Settings\All Users\Application Data\'.

Vawtrak calls the GetFileAttributesA API to check if the derived anti-malware's pathname exists. If the pathname exists, the malware will generate a hash value for it. Then it will create the registry key 'HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers\0\Paths\[hash value]' with the subkeys 'SaferFlags' set to 0 and 'ItemData' set to the anti-malware's pathname.

This newly created key will restrict the privileges and permissions of the given anti-malware application. It can also restrict execution of any applications under a given pathname.

Vawtrak will try to restrict the permissions granted to any anti-malware applications in its list.

Figure 3 shows a list of anti-malware names, the registry entries, and the warning message that appears when attempting to execute an application from a restricted folder.



```
Value    |Comments
00000000
00405A3C ASCII ":\Program Files\"
00405A24 ASCII ":\Program Files (x86)\"
004059EC ASCII ":\Documents and Settings\All Users\Application Data\"
00000000
004059DC ASCII "AVAST Software"
004059D0 ASCII "Avira GmbH"
004059C8 ASCII "Avira"
004059B8 ASCII "Kaspersky Lab"
0040599C ASCII "Kaspersky Lab Setup Fi
00405994 ASCII "DrWeb"
00405980 ASCII "Norton AntiVirus"
00405978 ASCII "ESET"
00405970 ASCII "Agnitum"
00405960 ASCII "Panda Security"
00405958 ASCII "McAfee"
0040594C ASCII "McAfee.com"
00405940 ASCII "Trend Micro"
00405934 ASCII "BitDefender"
0040592C ASCII "ArcaBit"
00405918 ASCII "Online Solutions"
00405904 ASCII "AnVir Task Manager"
004058F4 ASCII "Alwil Software"
004058E8 ASCII "Symantec"
004058E4 ASCII "AVG"
004058DC ASCII "Xore"
004058BC ASCII "Common Files\Symantec
004058A4 ASCII "a-squared Anti-Malware
0040588C ASCII "a-squared HiJackFree"
00405884 ASCII "avg8"
004058E4 ASCII "AVG"
00405878 ASCII "Doctor Web"
00405860 ASCII "Common Files\Doctor Web"
00405854 ASCII "f-secure"
004057E8 ASCII ":\Documents and Settings\NetworkService\Local Settings\
004057DC ASCII "G DATA"
004057C8 ASCII "Common Files\G DATA"
004057C0 ASCII "P Tools"
004057A8 ASCII "Common Files\P Tools"
0040578C ASCII "P Tools Internet Security"
00405780 ASCII "K7 omputing"
00405940 ASCII "Trend Micro"
00405778 ASCII "Vba32"
00405764 ASCII "Sunbelt Software"
00405754 ASCII "FRISK Software"
00405918 ASCII "Online Solutions"
0040573C ASCII "Security Task Manager"
00405728 ASCII "Zillya Antivirus"
00405714 ASCII "Spyware Terminator"
00405708 ASCII "Lavasoft"
004056FC ASCII "BlockPost"
004056E8 ASCII "DefenseWall HIPS"
004056DC ASCII "DefenseWall"
004056BC ASCII "Microsoft\Microsoft An
0040569C ASCII "Microsoft Security Ess
00405690 ASCII "Sandboxie"
00405678 ASCII "Positive Technologies"
00405670 ASCII "UAenter"
00405660 ASCII "Malwarebytes"
00405644 ASCII "Malwarebytes' Anti-Mal
00405628 ASCII "Microsoft Security Cli
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
ItemData	REG_SZ	C:\Program Files\AntiMalware
SaferFlags	REG_DWORD	0x00000000 (0)

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\Safer\CodeIdentifiers\0\Paths\{33DC7B8C-4C90-4538-AC57-27287

Address C:\Program Files\AntiMalware

AV.exe
Windows Calculator applicatio...
Microsoft Corporation

C:\Program Files\AntiMalware\AV.exe

C:\Program Files\AntiMalware\AV.exe

Windows cannot open this program because it has been prevented by a software restriction policy. For more information, open Event Viewer or contact your system administrator.

Figure 3: A list of anti-malware names, the registry entries and the warning message displayed when attempting to execute an application from a restricted folder.

Generating the last doll

After restricting the execution of anti-malware applications, the malware is ready to unwrap its final component – the last ‘doll’.

First, Vawtrak accesses its resource section by calling the FindResourceA API with an RT_RCDATA (raw data) type parameter. This is followed by getting the size and the handle of the only resource found for this module, using a combination of the SizeofResource and LoadResource APIs.

After a new heap memory has been allocated using the HeapCreate and RtlAllocateHeap APIs, the malware copies 184,258 (0x2cfc2) bytes of raw data resource to the heap memory, byte by byte. Upon finishing this, the resource is set free by calling the FreeResource API.

A marker (‘AP32’) at the beginning of the raw data is checked to make sure that the right binaries are copied to the heap memory. Another check is made by calculating the hash of the whole raw data and comparing it to the value 0x24D2EDEA.

If these checks are successful, another block of heap memory is allocated. Vawtrak decrypts the raw data and copies it to the newly allocated heap. Another hash computation is performed against the newly decrypted data and the resulting hash is compared against the value 0x52194545.

The new heap now contains a new executable binary – the last ‘doll’ – in the form of a dynamic link library (DLL) file.

A new file, with a random filename and an extension name of ‘dat’, is created in the %appdata% folder using a combination of the SHGetFolderPathA and CreateFileW APIs. Afterwards, Vawtrak copies the contents of the new heap memory to the newly created file.

This is followed by creating an autostart registry entry, ‘[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run]’ with value ‘[regsvr32.exe /s ‘C:\Documents and Settings\All Users\Application\{random filename}.dat’]’, using a combination of the RegCreateKeyA and RegSetValueExW APIs. This makes sure that the malware maintains persistence upon a restart.

Finally, to enable the current module to use the functions exported by the dropped DLL file, Vawtrak loads it into memory using a call to the LoadLibraryW API with the parameter ‘C:\Documents and Settings\All Users\Application\{random filename}.dat’.

The usage of the DLL file is not covered in this article. In principle, any DLL file could be included.

CONCLUSION

Like a nesting doll, the first executable binary (outer doll) generates the second executable binary from its overlay section; the second executable binary (second doll) decompresses a big chunk of data to generate the third executable binary; the third executable binary (third doll) uses its resource section to generate the final executable binary (last doll).

Each ‘doll’ (executable binary) has its own set of algorithms and functions that leads to the unwrapping of the next one. Every binary (except for the last one) has an important role to perform in generating the next one.

The ingenuity and skills shown by Vawtrak are not simple, but concise. Do not be deceived by a nice picture. Be vigilant and stay safe.

REFERENCES

[1] Leyden, J. Vawtrak challenges almighty ZeuS as king of the botnets. The Register. December 2014. http://www.theregister.co.uk/2014/12/27/vawtrak_challenges_almighty_zeus_as_king_of_the_botnets/.

[2] Wyke, J. Vawtrak – International Crimeware as a Service. Sophos. December 2014. http://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/sophos-vawtrak-international-crimeware-as-a-service-tpna.pdf.

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

© 2015 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/