virus BULLETIN Covering the global threat landscape

THE HULK

Raul Alvarez
Fortinet, Canada

In most cases, infected files have an increased file size due to the nature of their infection: the infection process either results in the addition of a new section, or the expansion of the file's own last section. Either way, the infected file is larger than the original.

However, there is another group of file infectors known as 'cavity file infectors', which can infect files without increasing their size. We seldom see this kind of infector due to the complexity of their algorithms, or perhaps because they are more time consuming to create.

In this article, we will look into how Win32/Huhk implements cavity infection.

CONTEXT #1: RUNNING WITHIN AN INFECTED FILE (NOT EXPLORER.EXE)

Huhk's execution depends on the context in which it runs. If it is running within any infected file, other than explorer.exe, its only goal is to make sure that explorer.exe is infected. Both '%system%\dllcache\explorer.exe' and '%windows%\explorer.exe' must be infected.

The following sub-sections describe what happens within this context.

Collecting bytes

When an application runs, the initial address of the SEH (Structured Exception Handling) chain points to a procedure in kernel32.dll. The malware makes use of this fact by subtracting 0x1000 from the initial address until it reaches the ImageBase of the kernel32 library.

Once the kernel32 library has been found, Huhk computes the hash of each API name in the kernel's export table to search for 'VirtualAlloc' (0xA5171D00). (The hash algorithm only uses 'ADC EDI,ECX' and 'ROL ECX,8' instructions.) Then it computes the equivalent API address based on the index of the API name.

This is followed by allocating a section of virtual memory, for byte collection, using the newly resolved VirtualAlloc API.

The malware has a table of addresses and sizes. Let's call it the 'cavity table'. The addresses point to the malware bytes scattered throughout the infected module, while the sizes determine the number of malware bytes at a given address.

To collect these bytes, the malware performs the following steps:

1. The first address in the cavity table points to the unencrypted bytes of the malware. These bytes are copied to the newly allocated virtual memory without decryption, with the number of bytes copied determined by the first size entry in the table.
2. The next address points to the encrypted malware bytes. The decryption algorithm is a simple XOR using a key taken from the first byte of the TimeDateStamp value of the infected module. After a byte has been decrypted, it is copied to the virtual memory.

   The malware bytes are decrypted and copied to the virtual memory, byte by byte, until the count reaches the given size value.
3. Step 2 is repeated for as long as the next address in the table is not zero. These variations depend on the available free spaces in any given host file.

Once all malware bytes have been collected in virtual memory, Huhk transfers control to the newly copied code.

Resolving APIs

Within the context of the new malware code in the newly allocated memory, Huhk resolves its APIs using a similar method to that discussed above.

The API resolution algorithm is as follows:

1. In a given list, if the hash value is 0xFFFFFFFF, what follows will be a pointer to a library (DLL) name. The library is loaded using a call to the LoadLibraryA API.

   (The default library is kernel32 and the first group of hash values belongs to it.)
2. If a hash value is for an API, Huhk will compute the hash value of each API name in the library's export table to look for the equivalent hash value (see Figure 1).
3. It then gets the corresponding API address based on the index of the API name.

Each hash value in the list (see Figure 1) passes through the above steps.

Restoring the original bytes

After resolving all the required APIs, the malware restores the first five bytes at the infected module's entry point.

The routine that restores the original bytes is as follows:

hash	API	hash	API
7DDF0CDC	kernel32.LoadLibraryA	5AC3B09E	kernel32.WaitForSingleObject
93F50CDC	kernel32.LoadLibraryW	47B5C7A4	kernel32.FindFirstFileW
1479946F	kernel32.CreateFileW	DE4C5E3B	kernel32.FindNextFileW
4788654	kernel32.GetFileAttributesW	F66A783F	kernel32.GetCurrentDirectoryW
EE628654	kernel32.GetFileAttributesA	F66A784B	kernel32.SetCurrentDirectoryW
4788660	kernel32.SetFileAttributesW	FFFFFFFF	
E3486339	kernel32.CreateFileMappingW	004015A1	pointer to "ADVAPI32"
D444401D	kernel32.MapViewOfFile	3CABD9B9	ADVAPI32.LookupPrivilegeValueA
A6131C00	kernel32.UnmapViewOfFile	53B8BA9F	ADVAPI32.OpenProcessToken
1E92925C	kernel32.GetFileSize	7EE8D9AE	ADVAPI32.AdjustTokenPrivileges
1286865D	kernel32.GetFileTime	FFFFFFFF	
12868669	kernel32.SetFileTime	004015AE	pointer to "WS2_32"
2599996D	kernel32.GetFileType	AC1A1513	WS2_32.connect
27969D71	kernel32.CloseHandle	F63719F9	WS2_32.WSAStartup
5ED2C494	kernel32.GetProcAddress	45A8A3B0	WS2_32.socket
3BADB19F	kernel32.VirtualFree	3EB2B2A8	WS2_32.gethostbyname
980BFCD2	kernel32.GetTickCount	64D2D3D8	WS2_32.send
FE728047	kernel32.GetWindowsDirectoryW	76D9C8D7	WS2_32.recv
C4384622	kernel32.GetModuleFileNameW	FFFFFFFF	
7DF1FFDB	kernel32.GetTempPathW	401599	pointer to "mpr"
67DBFFDB	kernel32.GetTempPathA	E0455238	mpr.WNetOpenE

Figure 1: Partial table with hashes and the equivalent APIs.

1. Initially, it changes the protection of the bytes to PAGE_EXECUTE_READWRITE using the VirtualProtect API.
2. This is followed by opening the current process using a combination of the GetCurrentProcessId and OpenProcess APIs.
3. Finally, it writes the original bytes using the WriteProcessMemory API.

Using the same routine as described above, the malware restores another seven bytes of the infected module. These bytes are used to jump to the first malware function (see the section 'Collecting bytes').

Using the information from the cavity table, Huhk restores the memory locations at which the scattered pieces of malware are located. To quickly refill these locations, the malware allocates a section of virtual memory full of zeros using the VirtualAlloc API. Then, using the zeros and the routine above, the memory locations pointed to by the cavity table are restored.

Determining which context

After restoring the necessary bytes of the host file, Huhk checks if the current infected module is explorer.exe. If it is not, it will continue with the sequence of events described in the following sub-sections, otherwise it will perform a different sequence of events (as described in the section 'Context #2').

To continue running in this context, the malware gets the temp path folder using the GetTempPathW API, and checks whether '%temp%\lorer.exe' exists using the GetFileAttributesW API.

If 'lorer.exe' (derived from 'explorer.exe') exists, the malware will restore the original bytes of the current module and transfer control to it. Checking for the presence of '%temp%\lorer.exe' is another form of checking whether the system is already infected.

Otherwise, the malware performs the following routine:

First, it tries to disable the Windows File Protection of '%windows%\explorer.exe'. Then it moves '%windows%\explorer.exe' to '%temp%\lorer.exe' using the MoveFileW API. This is followed by overwriting '%system%\dllcache\explorer.exe' with '%temp%\lorer.exe' using the CopyFileW API with the bFailIfExists parameter set as FALSE.

Next, it gets the file attributes of '%system%\dllcache\explorer.exe' and saves them for later use. The new attributes are set to FILE_ATTRIBUTE_NORMAL using the SetFileAttributesW API.

The file '%system%\dllcache\explorer.exe' is opened with GENERIC_READ|GENERIC_WRITE access using the CreateFileW API. To make sure that it is a disk file, the malware calls the GetFileType API.

The file's size and time stamp are also saved for later use, using the GetFileSize and GetFileTime APIs.

Finally, the file '%system%\dllcache\explorer.exe' is loaded into the memory, ready for reading and writing, using a combination of the CreateFileMappingW and MapViewOfFile APIs.

Infection routine

As we have observed so far, Huhk is polymorphic in nature. Besides being a cavity file infector, it can infect files with different binary versions of itself, making it harder to detect.

Although it uses a simple XOR algorithm for encryption and decryption, the generation of the key is a bit more interesting.

Generating the encryption key and the infection marker

The infection marker is used to avoid re-infection of host files, while the decryption key is used to expose the actual binary. It is rare for both of them to be located in the same place.

In checking for the infection marker, straight after the mapping of the cached version of explorer.exe, the malware gets the TimeDateStamp DWORD value from the PE header. If the sum of the first and second bytes of the TimeDateStamp DWORD is 0xFF, the file is already infected. The malware then unmaps and closes '%system%\dllcache\explorer.exe' using the UnmapViewOfFile and CloseHandle APIs, and exits from this routine.

If, on the other hand, '%system%\dllcache\explorer.exe' is not yet infected, the malware will generate the encryption key and infection marker by performing the following routine:

Initially, the malware checks whether the file is a DLL. If it is, it skips the infection routine, unmaps explorer.exe, and exits from this routine.

Otherwise, it gets a new DWORD value by calling the GetTickCount API. It divides the DWORD by three, changes the second byte to 0xFF, subtracts the first byte from the second byte, and replaces the second byte with the difference. The final value of the DWORD now contains the infection marker (the sum of the first and second bytes) and the encryption key (first byte). It is saved to a memory location for later use.

Looking for free space

After generating the marker and the key, the malware traverses the content of the mapped explorer.exe, starting at the PE header. It looks for free spaces (memory locations filled with zeros) for its cavity infection.

It checks each DWORD memory for 0x00000000. When a DWORD with 0x00000000 is found, it marks it as 'startingLocation'. Then, it looks for a non-zero DWORD and marks it as 'endingLocation'. The size of the free space is determined by the difference between the 'endingLocation' and the 'startingLocation'.

If the size is equal to or greater than 0xFA, the malware will add it as an entry in a temporary table in stack memory. Each entry consists of an address pointing to a free space, and the size of the free space. This temporary table will serve as the cavity table in a successful infection.

The malware continues to search for every available space until it reaches the end of the mapped explorer.exe file. Every suitable free space will be referenced in the cavity table.

Once all available free spaces have been referenced, the malware sums up all the sizes in the table and checks whether the total is enough for the malware. The total size must be greater than 7,005 bytes (0x1B5D).

Cavity infection

The first block of code is important to the malware, so it looks for enough free space to place the initial code. It searches the cavity table for a size entry which is equal to or greater than 551 bytes (0x227), then moves that entry to the top of the table.

The infection begins by copying the malware code as is (no encryption) to the free space pointed to by the address at the top of the cavity table.

Afterwards, the rest of the malware code is copied and scattered amongst the various free spaces referenced by the table (see Figure 2). Each and every byte is encrypted with a simple XOR using the key that was generated earlier.

Finalizing the infection

After filling the free spaces of the mapped '%system%\dllcache\explorer.exe' file, the malware readjusts the addresses in the cavity table relative to the ImageBase of explorer.exe. Then it copies the cavity table from the stack memory to the mapped file.

This is followed by overwriting the first five bytes of the entry point of the mapped explorer.exe. These bytes are the initial call to the malware routine.

The TimeDateStamp field of explorer.exe's PE header is also changed to the DWORD (encryption key and infection marker) that was generated previously.

Afterwards, Huhk generates the checksum value of the newly infected explorer.exe, using the 'ADC DX,AX' instruction. Each WORD value is computed, starting from the MZ header, and the final WORD value is copied to the Checksum field of explorer.exe's PE header. The generated checksum is different for each infection due to the polymorphic nature of the malware.

To flush all the changes and modifications to the physical file of explorer.exe, the malware calls the UnmapViewOfFile API.

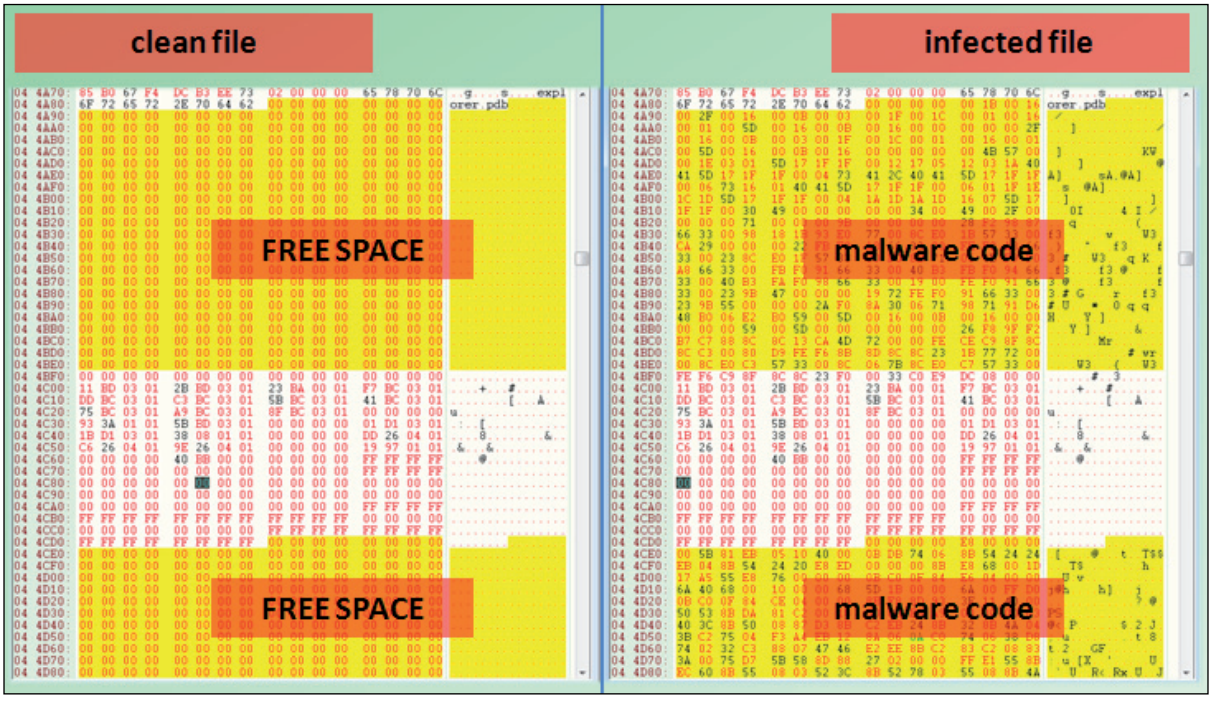

Figure 2: A cavity-infected file using the free spaces.

Finally, Huhk restores the original time stamp and attributes of '%system%\dllcache\explorer.exe' by calling the SetFileTime and SetFileAttributesW APIs.

Passing the infection

After the infection routine, Huhk tries to move the infected '%system%\dllcache\explorer.exe' to '%windows%\explorer.exe', then it tries to copy '%windows%\explorer.exe' back to '%system%\dllcache\explorer.exe', using the MoveFileW and CopyFileW APIs, respectively. This is the malware's attempt to make sure that both copies of 'explorer.exe' files are infected.

Hooking WS2_32.connect

For Huhk's final trick, it hooks the 'connect' API of WS2_32.DLL in memory, by changing the first five bytes of the API to a call to the malware's code. The hook function for the connect API is discussed in the next section.

Hooking the connect API is only implemented in the infection of non-explorer executable files.

After hooking the connect API, the malware transfers execution to the host file.

Activating the connect API

The hook function, which is activated when the connect API is called, attempts to connect to 'http://vampire00[--REMOVED-]info' and 'http://c34.statcoun[-REMOVED-]unter.php?sc_project=3034266&java=0&[-REMOVED-]&invisible=0', and tries to download another piece of malware. (For safety reasons, parts of the links have been removed.)

The URLs are the decrypted version of the hard-coded strings 'cpqn9/0xdqumug107rt*stuw-iohr' and 'cpqn9/0e683wwcucnskp`n+anm0erysxht/pgn<o^[mpnjfewA8466365$g]q]:.%sfexvnx|?396/-.\b#gmvjulfqi@2', respectively. At the time of writing this article, the links were no longer active.

CONTEXT #2: RUNNING IN THE CONTEXT OF EXPLORER.EXE

In this context, the malware goes through the same steps as described in the preceding sub-sections, until it reaches the point of determining in which context it is running.

Since it is running in the context of explorer.exe, the malware hooks the CreateProcessW API by activating Thread #1 (see below). Afterwards, it transfers control to *Explorer*'s main module.

The malware now sits and waits for the hooked CreateProcessW API to be called.

Thread #1

This thread is responsible for installing the hook for the CreateProcessW API.

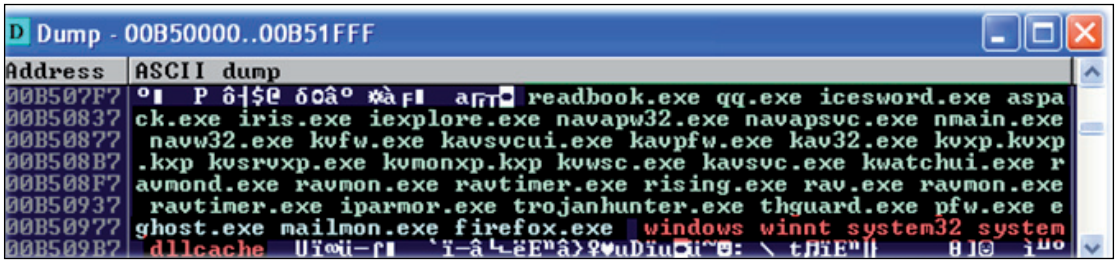

Figure 3: Folder names and filenames to avoid.

When the thread is activated, it sleeps for 30,000ms, before waking up to place the hook to the CreateProcessW API using the following routine:

The malware changes the protection of the CreateProcessW API to PAGE_EXECUTE_READWRITE using the VirtualProtect API, opens the current process using a combination of the GetCurrentProcessId and OpenProcess APIs, then modifies the first five bytes using a combination of the ReadProcessMemory and WriteProcessMemory APIs. The first five bytes are now a call to the hook function.

After setting the hook, the thread terminates.

Activating the CreateProcessW API

The hook function is triggered when the hooked CreateProcessW API is called.

Initially, Huhk restores the original bytes of the CreateProcessW API, using a routine similar to that used in Thread #1. The only difference is that it restores the original bytes of the API instead of hooking it.

This is followed by extracting the folder names from the pathname of the application, taken from one of the parameters from the CreateProcessW API when it was triggered. The folder names are converted to lower case and the malware checks whether any of the following strings are present: 'windows', 'winnt', 'system32', 'system' and 'dllcache'.

Every folder name is extracted from the pathname, and if any of them match any of the aforementioned strings, the malware will skip other checks, re-hook the CreateProcessW API by activating Thread #1, and exit the current function.

If the folder names pass the checks, the filename is also checked against the strings shown in Figure 3. If any of the strings match the filename, it will re-hook the API and exit. The strings resemble the filenames of some anti-malware and security applications.

If, on the other hand, both the folder names and filenames pass the checks, the malware will perform the infection routine, which is similar to the infection of '%system%\dllcache\explorer.exe'. Once the infection is finished, it will activate Thread #1 to re-hook the CreateProcessW API and exit the function.

WRAP UP

In the case of this piece of malware, explorer.exe is always infected, while the infection of other executable files only happens if the malware runs in the context of the aforementioned critical file.

Due to its infection criteria, the malware only infects a handful of executable files. In this regard, it has unintentionally created a stealth technique.

In addition to its polymorphic nature, Huhk's cavity infection technique and the small number of infected files help it to avoid detection.

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
Perl Developer: Tom Gracey
Consultant Technical Editors: Dr Morton Swimmer, Ian Whalley

Tel: +44 (0)1235 555139. Fax: +44 (0)1865 543153
Email: editorial@virusbtn.com
Web: http://www.virusbtn.com/