VIRUS ANALYSIS

TUMOURS AND POLIPS

Peter Ferrie
Symantec Security Response, USA

It seems that we have reached the stage where a parasitic virus has become a novelty. That might explain why the W32/Polip virus caught us by surprise recently – we didn't expect to see one, and we certainly didn't expect to see anything of such apparent complexity. However, looks can be deceiving.

The virus author chose the name 'Polipos', which is the Spanish word for polyp, a non-malignant growth. Perhaps the virus author wanted to suggest that the virus was harmless.

While the virus certainly was written carefully, its author was not careful enough. The virus author favoured function over form, so the code is far from optimised, but it works well enough.

EXIT, STAGE LEFT

The virus begins by checksumming itself, and branches to an exit routine if the checksum does not match the expected value. This is where we encounter the first bug. The exit routine is intended to restore the patched host bytes. It requires the VirtualProtect() API to have been retrieved from kernel32.dll – however at this point no APIs have been retrieved. The virus is aware of this possibility and checks whether the address is zero.

The bug is the fact that the address is never initialised, so it always contains a non-zero value. Additionally, the virus assumes that the host module handle has been retrieved, but again, this has not occurred yet. However, neither of these problems causes a crash, since the virus uses Structured Exception Handling to trap the errors, and simply skips restoring the bytes.

The virus then copies the host bytes into a special buffer and executes them from there. This means that if the host bytes are never restored, the virus code could be called repeatedly, as often as the patched bytes are reached.

HAPI HAPI, JOY JOY

If the checksum matches, the virus will retrieve some API addresses from kernel32.dll. The APIs are located by checksum, instead of by name.

While there is nothing new about this idea, the API resolver in this virus is aware of import forwarding. This is new code for a virus, even though the problem has long been known about and documented by virus writers. It is also a requirement for the virus to work with *Windows XP* and later, since some functions, such as GetLastError(), are forwarded into ntdll.dll as RtlGetLastWin32Error().

Function forwarding exists in all 32-bit *Windows* versions, including *Windows 9x/Me*, but the forwarded functions on those platforms are not used by the virus.

Interestingly, the checksum routine is the same 16-bit CRC32 routine that has been used by a number of viruses previously. Given the technical level of the rest of the code, this routine seems a very strange choice.

The first set of APIs that the virus retrieves are related to file management. The virus branches to the exit routine if any API is not retrievable. The second of the bugs in the virus occurs here, and it is the fact that the check for retrieving all of the APIs successfully appears only after one of the functions has already been used.

The virus then retrieves a second set of APIs, most of which are related to thread management. It branches to the exit routine if more than nine APIs are not retrievable. If that check passes, the virus makes certain assumptions about which of those APIs have been retrieved successfully.

BAD SEED

At this point, the virus calls the GetTickCount() API to initialise the random number generator. The generator is seeded further by the entry point address of the virus.

There is some unused code here, which perhaps is left over from an earlier version, since a text string suggests that this is 'version 1.2'. The code loads kernel32.dll again, even though it has been used already.

The virus then retrieves a set of APIs from user32.dll, which are related to window messaging. It branches to the exit routine if any API is not retrievable.

At this point, the virus considers itself sufficiently initialised to choose a different exit routine in the event of failure. That function supports the repair of files that have data appended to their executable image, typically application installers and self-extracting archives.

STRATA MANAGER

The repair function begins by copying the infected file to the '%temp%' directory, as 'ptf[random].tmp'. The file is checksummed and compared with the checksum that the virus carries. If the checksums do not match, the virus terminates and does not even run the host.

Otherwise, the virus restores the host bytes, as before. Additionally, the virus carries a table that contains the

addresses of the cavities that the host contained, in which the virus placed some of the decryptor code. The virus erases the content of those cavities, and restores the section sizes to their original values.

The virus removes the unnamed section that contains the virus body, and moves back all of the data directories that were present. It also restores the security table if it existed previously. The virus relocates all debug and/or resource information properly, if they existed. It also rewrites the file header information to remove all traces of the added section.

The virus recalculates the SizeOfCode, SizeOfInitializedData and SizeOfUnitializedData values to place into the PE header. However, these values are used only if the SizeOfCode value was zero in the original file – which can never happen, since the virus avoids such files. Otherwise, the virus uses the values that it reserved prior to the infection.

If the PE checksum field was non-zero previously, the virus checksums the file again and compares it with the checksum of the original file that the virus carries. If they match, the virus uses that checksum, otherwise the virus uses the CheckSumMappedFile() API, if it is available, to calculate a new checksum.

The repair function is capable of returning three different result codes, one of which indicates complete success. The result is checked at the end of the function, but no action is taken. That check appears to be from older code. The result is also checked again later, and if repair was a complete success, the repaired file is executed. Once the repaired file terminates, the virus waits three seconds, then deletes the file and terminates the infected process.

NEW VERSION

The virus collects information about the operating system of the victim machine, the amount of memory present, as well as the CPU family and its capabilities. Specifically, the virus retrieves the *Windows* version number, and branches to the repair routine if it finds it is running on *Windows NT*. The virus accepts all *Windows 9x* versions (it has code devoted to the special handling required there), including *Windows Me*, and *Windows 2000* and later.

The virus calls the GlobalMemoryStatus() API to find out how much physical memory exists, and exits if there are less than 64Mb. The documentation for the API states that the size field must be set first, but this is not true, and the virus author knows it.

The virus checks the CPU flags for the presence of the CPUID instruction, and if available it uses the CPUID instruction to query the CPU family and for the presence of two recent instructions. The virus requires an *Intel 80486* or better CPU, but also requires support for the CMPXCHG8B instruction (introduced in the *Intel Pentium* 1) and CMOV instruction (introduced in the *Intel Pentium 2*). The virus branches to the repair routine if one or more of these three instructions is not available.

The virus write-enables its own module header in order to place an infection marker there if one is not present already. Since this operation is supported only on *Windows NT* and later, the virus achieves this by using the undocumented VxDCall function if it is run on *Windows 9x/Me*. If the infection marker was already present, the virus branches to the repair routine.

The virus also checks if the system is shutting down, by querying the GetSystemMetrics() API, and branches to the repair routine if so. This check is supported only by *Windows XP* and later. Conveniently, however, the return value is the same if the request is unsupported, and if the system is not shutting down. As such, it is unclear whether the virus author intended to support *Windows 2000*, or was targeting *Windows XP* and later.

If all of these checks pass, the virus retrieves from advapi32.dll a set of APIs that are related to security tokens and registry key manipulation. It branches to the repair routine if any API is not retrievable.

The virus queries the 'SCRNSAVE.EXE' value of 'HKCU\Control Panel\Desktop' key. A bug exists here that results in a handle leak if the value does not exist. The returned filename is a candidate for infection.

TERMINAL DISEASE

The virus attempts to acquire the 'SeDebugPrivilege' and 'SeCreateGlobalPrivilege' privileges. The 'SeDebugPrivilege' is required for process enumeration, while the 'SeCreateGlobalPrivilege' is required by *Terminal Services* applications in order to create a file-mapping object, which the virus uses for several purposes. This is the first known virus that is aware of *Terminal Services*.

The virus creates a file-mapping object in the global namespace, whose name is the entry point code of the host. The name is adjusted to remove all zeroes. Additionally, the attributes are adjusted so that they also work on *Windows XP SP2*. Within this map, the virus creates three randomly named global namespace objects, and marks the map with the string 'JIPC' ('gypsy').

On *Windows 9x/Me*, the virus allocates memory using an undocumented flag to create a shared memory region. On *Windows 2000* and later, the memory region is already

shared. The virus then copies itself into the shared memory region. This copy of the virus code is used when the virus injects itself into other processes.

The virus also acquires a security descriptor to achieve full access to objects that require ACLs. This is very uncommon – other viruses simply allow *Windows* to supply the default security descriptor, with the potential associated access limitations.

The virus then checksums the current process filename if it has been run either from a subdirectory from the following list or from within the '%ProgramFiles%' or '%SystemRoot%' directories (which might be different from the list below), regardless of the drive:

\program files
\windows
\win98
\win98se
\winxp
\win2000
\winnt
\winme

Based on that, the virus intends to check the checksum against a list of 37 special filenames. The filenames belong to network-aware applications such as *Windows Messenger*, *MSN* and *NetMeeting*. However, a bug exists here – this code is reached regardless of the execution location, so the register that should hold the checksum could hold another value, and it is possible that this value can match something in the list.

If the checksum was not found in the list, the virus enumerates the windows of the current process to see if one of them corresponds to *Windows Explorer*.

If the checksum was found, or if the current process is *Windows Explorer*, the virus retrieves from wininet.dll a set of APIs related to remote file retrieval. If any APIs cannot be retrieved, the virus 'forgets' that it found any of the APIs.

ON A TIGHT SCHEDULE

The virus uses its own thread scheduler, which works across process boundaries. The reason for this is that multiple threads will be injected into remote processes, and they must be coordinated to prevent resource conflict and to synchronise their behaviour. This appears to be the work of a professional programmer.

The scheduler begins by checking whether the filename of the current process can be found in a list carried by the virus. The list is composed of names of a large number of anti-malware products, and several other applications that are known to perform self-checking. The virus disables the file infection if any of them are found.

The virus retrieves the address of the undocumented SfcTerminateWatcherThread() API from sfc.dll. The virus uses the GetProcAddress() API because its import resolver does not support functions that are imported by ordinal only. If the current process filename is 'winlogon.exe', the virus calls the SfcTerminateWatcherThread() API to disable the System File Checker.

HOOKED ON CLASSICS

The virus then retrieves the following API addresses from kernel32.dll – it retrieves only the first five if it is running on *Windows 9x/Me*, or all of them if it is running on *Windows 2000* or later:

ExitProcess
CreateProcessA
CreateFileA
LoadLibraryExA
SearchPathA
CreateProcessW
CreateFileW
LoadLibraryExW
SearchPathW

The code in these functions is parsed, instruction by instruction, using what appears to be a home-made length disassembler engine.

At 778 bytes long, this is surely one of the largest and most inefficient assembler length disassembler engines in existence. The champion of those was published in *29A*#7, and is more functional, yet only 339 bytes long (and it can even be shortened by one byte!). However, as noted previously, the author of this virus favoured function over form, so the code is far from optimised.

The disassembler is used to copy code from the API, until five bytes have been copied, or an e8 or e9 opcode is seen. In either case, if the API address could be retrieved, then it will be hooked to point to code within the virus body.

Since the data to be modified exist in a shared memory region, the virus uses a multiprocessor-compatible method to write the required number of bytes in one pass. The hooked APIs allow the virus to infect files as they are accessed, or, in the case of ExitProcess, once the process has terminated.

After hooking the APIs, the virus queues three files for later infection. Those files are the values of the 'SCRNSAVE.EXE' registry key, '%system%\logonui.exe' and '%system%\logon.scr'.

Then the scheduler enters its idle loop. Periodically, the idle loop creates a thread that checks for the presence of a debugger. If one is found, the virus stops all activity until the debugger exits.

Additionally, the virus checksums itself to ensure that two specific routines (the scheduler and detection of *VMWare*) have not been changed. A change to either of these routines will also cause the virus to stop all activity.

TIME PASSES...

The idle loop periodically calls the routine to perform the thread injection into other processes. The injection routine enumerates all running processes within the current session if running in *Terminal Services*.

The routine ignores processes whose names are any one of the following:

savedump
dumprep
dwwin
drwtsn32
drwatson
kernel32.dll
smss
csrss
spoolsv
ctfmon
temp

It also ignores the current process. While searching, the routine attempts to detect the presence of *SoftICE* and *VMWare*. The enumeration exits if *SoftICE* is found, but due to a bug, the detection of *VMWare* does not work.

For any other process found, the routine enumerates the threads within the process, looking for threads that have been created by that process (i.e. ignoring injected threads). For each of these threads, the routine suspends the thread, then sends it a message to see if the thread wakes up. If the thread does not respond, the routine injects the virus code into the remote process and redirects execution to the injected code.

The injected code then begins the whole process again (including unpacking, which is the reason for the large size of virus – the virus carries a packed version of itself). Finally, control returns to the original code in the thread.

If no thread could be suspended, the routine attempts to create a new thread within the remote process. If that is successful, the routine injects the virus code as described above.

SAY YES TO GNUTELLA

After some time, the scheduler will start the backdoor thread. First, the backdoor checks for an active network connection. If an active connection is found, the backdoor will create a hidden window, which is used to control the network activity.

The virus then retrieves a set of APIs from ws2_32.dll, if available, and otherwise from wsock32.dll. The APIs are related to network management. The backdoor exits if any of these APIs are not retrievable.

The backdoor understands the *Gnutella* 0.6 protocol, as used by *Gnucleus* and *BearShare*, among others. It watches for the arrival of *Gnutella*-specific strings, and responds appropriately.

This is not as impressive as it sounds – the protocol is open, and the source code is available freely. However, it is significant in one way: the virus can spread through the P2P network, from a compromised machine that does not have the P2P software installed.

The *Gnutella* routine works by contacting a *Gnutella* web caching server selected at random from a list carried by the virus, and retrieving the current list of connected clients. The routine then connects to these clients, so now it will be contacted if a query is made. The routine responds to queries by offering a file called 'dmckaziejdntb'. This is the virus.

The routine keeps the current contact information in the '{1DF41E2A-DA21-0412-829E-240A8C38F7A1}' value of the 'HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths' registry key.

Periodically, infected machines will communicate with each other, by sending a special packet. These packets all start with the string 'VPacket'. For any query that contains the string 'cmdp', a particular 'VPacket' will be sent, which will cause the virus to connect back to the sender on the specified port, and download an updated version of the virus.

INFECTIOUS GROOVES

In addition to the specific files queued for infection, the virus is interested in the subkeys in the

'HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths' registry key. The default value of each subkey is queried. If the filename in the data has an extension of either '.exe' or '.scr', then the file will be considered a candidate for infection.

The infection routine begins by checking if flags in the PE header specify a file of executable type, that is neither a DLL nor a system file. Additionally, the file must be for the command-line or GUI subsystem. The file will not be infected if it appears on the same anti-malware products list that is used to avoid thread injection.

The infection marker is the presence of an unnamed section. The virus adds this section during infection, in which to place the virus body. Files are also avoided if they contain only one section, or more than 11 sections. The virus also deletes the integrity-checking databases of several anti-virus products, if any of those files exist in the same directory as the file to infect.

The virus breaks its code into a random number of blocks, which it places into various areas of the file, including unused space at the end of other sections, and the unnamed section that the virus added. These blocks are then encrypted using a strong algorithm. While the algorithm resembles XTEA, it can probably no longer be called XTEA, since all of the important characteristics of XTEA have been changed. Specifically, XTEA is a 64-bit block Feistel network with a 128-bit key and 64 rounds.

The sum and delta values are C6EF3720 and 9E3779B9, respectively. Polip, on the other hand, uses a 32-bit block Feistel network with a 32-bit key and only 10 rounds. Additionally, the sum and delta values have been changed to 1717E09D and 9E37F9B9, respectively. Despite this weaker encryption strength, cracking the encryption is still infeasible within a reasonable time.

The decryptor is embedded within a highly polymorphic layer, which is also spread over the file. While most of it is appended to the body in the unnamed section, some parts of the decryptor are placed at the end of executable sections in the file.

The polymorphic engine itself presents nothing really new – it supports random register assignment, dummy loops and subroutines, and dummy references to the BSS section, all of which are fairly standard these days.

However, one interesting feature relates to the dummy subroutines themselves – the engine can produce subroutines that support fastcall, stdcall, and cdecl-format parameter passing, and the routines can even operate on the parameters. The results are always discarded, though.

The key weakness in the decryptor is the linear nature of its caller – the block decryption parameters are all passed from the same subroutine, so once that subroutine is found, the parameters can be retrieved and the virus code decrypted, without any significant time penalty.

The decryptor decrypts a stub, which decrypts the rest of the code and host bytes using a 32-bit xor key. Underneath that is the packed virus body. The packing algorithm is JCALG1. Underneath the packing is another layer of 8-bit xor encryption. JCALG1 is an unusual choice. It seems that it has been used by only one other virus – W32/Fizzer – which also appeared to be the work of a professional programmer.

CONCLUSION

This virus fuelled some unpleasantness in the anti-virus community: intentional withholding of samples until detection was completed.

The first company to detect the virus claimed that it had updated its product to provide full detection of the virus at that time. However, it was almost three weeks before any other company obtained samples of the virus, and a further week before everyone had received samples. For what purpose? During that time, it was demonstrated that none of the companies had managed to provide full detection of the virus, not even the first company (which updated its detection silently when the misses were found).

In fact, this virus is trivial to detect. To untrained eyes (i.e. those of the virus author), the polymorphic layer does look very complex and difficult. However, that layer contains so many constant operations, that the real instructions are recognisable instantly once the algorithm is understood. A simple repair is also quite easy.

For the really hard-core coders out there, it requires fewer than 1,000 lines of assembler to find and decrypt the virus, and restore the host bytes, using only static analysis. No emulation or debugging tricks are required.

Perhaps now the problem has been solved once and for all, and we can all get back to other work.

W32/Polip

Aliases:	W32/Polipos-A, PE_POLIP.A, p2p-worm.win32.polip.a.
Type:	Polymorphic memory-resident file infector.
Payload:	Infects .exe and .scr files; deletes integrity-checking databases of several anti-virus products.