memory. This error appeared on PCs with 24 MB of system memory installed, but there was no such error when replicating the virus on PCs with 64 MB of RAM.

From Word to Access

The virus uses a method analogous to that just described when infecting *Access* from *Word*. It creates the temporary DATA.COM file with DOS DEBUG, writes its hexadecimal data there, and unpacks it with the Extract utility to the infected DATA.MDB *Access* database. The virus then executes *Access* with the START command, passing the infected DATA.MDB file as a parameter. As a result, the virus takes control and infects other *Access* databases.

Conclusion

The evolution of computer viruses continues. DOS is dying, and DOS viruses will die with it, but a great number of *Windows* and macro viruses are replacing them, often repeating the evolutionary history of 'good old' DOS viruses. They occupy new 'ecological niches' and are spreading from one platform to another – from *Access* to *Word* and back in Cross' case.

We are probably fortunate that some of the more sophisticated cross-*Office* infection mechanisms available are not used in Cross. It can only be a matter of time, however, before we see viruses that utilize some of these options, breaking the shackles (and limitations) of DEBUG and spawning batch files to achieve their trickery.

Another problem is that few anti-virus vendors yet support *Access* database formats. If this kind of virus had its bugs fixed and appeared in the wild, it may be a real shock to the PC world. Let's be ready – ready for a *Word-Excel-Access-Windows* stealth and polymorphic virus that also infects MBRs and drops itself on CDs, if a CD-writer is installed…

Cross

Aliases:	None known.
Type:	*Microsoft Word 97* and *Access 97* macro infector which 'cross-infects'.
Self-recognition in files:	*Word* documents containing a module named X are assumed infected.
Intercepts:	*Word*: AutoClose, AutoExec, AutoExit and AutoOpen macros. *Access*: Autoexec macro.
Payload:	None – see text for bugs.
Removal:	In a clean *Word* environment, delete modules from Organizer. In *Access*, reset the Toolbars on the Toolbar tab of View/Toolbars/Customize, then follow the procedure for AccessiV (see *Virus Bulletin*, April 1998, p.15).

VIRUS ANALYSIS 2

HPS

Péter Ször
Data Fellows

The first *Windows 95* virus with oligomorphic capabilities was Memorial, which appeared in 1997 (see *VB*, September 1997, p.6). It caused infections in the wild and we suspected that the next step would be a fully polymorphic virus for *Windows 95*. Virus writers had to extend their knowledge significantly to reach that level of coding under *Windows 95*, but we did not have long to wait.

It is not surprising that the person to create such a virus comes from a strong DOS virus-writing background, specifically in polymorphic/multi-partite viruses. GriYo is a notorious member of the 29A virus-writing group and the author of complex viruses like Implant. As part of 29A, he has others around him, such as the author of Cabanas (see *VB*, November 1997, p.10), to help his creations work efficiently in *Windows* environments.

The pioneer of 32-bit polymorphic *Windows* viruses is called HPS (or Hantavirus Pulmonary Syndrome). The virus has been developed specifically for *Windows 98*, but is compatible with *Windows 95*. Fortunately, it does not work under *Windows NT*, because it is built on top of undocumented *Windows 9x* functionalities. HPS.5124 is a slow polymorphic, Portable Executable (PE) infector with directory stealth. Not only does it employ a new method of hooking the *Windows 9x* file system, it also has retro virus features. Its polymorphic engine is quite advanced, but worse news is that it is very stable. Despite making some extreme hacks in the heart of *Windows 9x*, the OS seems to accept the virus as part of the system. HPS is one of the first *Windows 9x* viruses with a graphical activation routine.

Initialization

When an infected PE program is executed, HPS' polymorphic decryptor takes control. The entry-point of the PE header points to the last section of the executable where the virus code is placed. The decryptor is at the end of the virus, in the last few hundred bytes of the infected program. The encrypted body precedes the decryptor with a fixed size of 5124 bytes. However, the overall size (including the decryptor) is variable, so HPS saves its size at the end of the infected program for use by its directory stealth routine.

The first byte of the virus body takes control when decryption is complete. First, the virus calculates the real entry point to the host program and saves its value to the stack. HPS' novelty is illustrated in this initial routine, which attempts to determine the address in memory of the already loaded KERNEL32.DLL in memory. Its purpose is similar to that of the Win32 GetModuleHandleA function that

returns the base address of a DLL in memory. These acrobatics allow the virus to access any exported APIs from there, without thunking through itself in the host's import section. The search routine is protected by Structured Exception Handling (SEH) as it would certainly cause General Protection faults otherwise. SEH saves the function from application level debugging.

It seems that KERNEL32.DLL is unlike any other DLL, since it is presented for all processes (under Windows NT, too) even if the actual program does not have imports for it. Thus, the virus can search for it in the usual shared memory address space. Since this is different from one system to another, the virus tries different addresses, scanning the memory backwards in 64KB steps for the 'MZ' marker representative of a loaded DLL (EXE) program. When the marker is detected, the virus looks for the 'PE' identifier. If found, HPS checks that the name of the module in the export table is KERNEL32.DLL. The virus uses some complex calculations for these signatures, and thus was tricky to analyse at first glance.

If the base of KERNEL32.DLL is not detected, HPS simply executes the host program. Otherwise, it checks for the VxDCall address at the beginning of the KERNEL32.DLL export table. A VxDCall is an undocumented *Windows 9x* API. The entries of VxDCalls (there are several of them to accept different parameters from the caller) were documented in the first beta release of *Windows 95*.

Microsoft decided to remove the names of these entries from the exported function table since Andrew Schulman used them in 'Unauthorized *Windows 95*' to show how KERNEL32.DLL calls VWIN32_Int21Dispatch services. Moreover, *Microsoft* added special code to the *Windows 95* GetProcAddress API to disallow imports by ordinal values. This is part of everyday business, the usual 'Where do you HAVE to go today?' ideology. Obviously, *Microsoft* did not want to allow developers the use of VxDCalls. The rationale behind this is probably the prevention of solutions incompatible with *Windows NT*.

Since the virus cannot use GetProcAddress to obtain the VxDCall address, it picks it up from the export table of the loaded KERNEL32.DLL. Then it makes its 'Are you there?' call (a VxDCall) – VWIN32_Int21Dispatch get date services with ESI=48505321h ('HPS!') and EDI=5453523Fh ('TSR?'). If, on return, ESI=59455321h ('YES!') the virus assumes it is already resident and executes the host program. Otherwise, it checks if it is Saturday and saves a flag to use later.

The use of a VxDCall is not new. Xine used one similarly in its infection routines, but HPS is the first virus to hook a VxDCall. The implementation of the hook requires a good understanding of *Windows 9x* system internals. Since a VxDCall is used by all applications through system DLLs (like KERNEL32.DLL), the implemented hook cannot be placed anywhere but in the shared memory area for all processes under *Windows 95*.

In order to be able to place itself here, the virus uses undocumented Win32 VxD services provided by VMM – PageReserve and PageCommit – again using a VxDCall. Thus, the virus allocates four pages of shared memory. If the start address of the allocated pages points lower than the usual shared memory range, HPS uses PageFree to free them. This is because the hook would certainly hang the system if the memory allocation were not able to reserve pages from the shared memory area.

Next, HPS scans 256 bytes from the entry point of the VxDCall in KERNEL32.DLL for the signature 2EFF1Dh. These are the bytes followed by an address that can be patched to the virus' own VxDCall entry point. (Basically, the patching method is very similar to the one Matt Pietrek used in one of his sample programs in his excellent book 'Windows 95 System Programming Secrets'.) How is that patch possible at all? How can a Ring 3 program overwrite some bytes in a shared system DLL? The answer is that this area is write-enabled. Thus, the patch works without even using the WriteProcessMemory debug API.

Before the actual hook, the virus calls its polymorphic decryptor generator. Since polymorphism happens only at installation, the virus is a slow polymorphic. HPS' body is copied to the beginning of the allocated shared memory area and the decryptor is created after that. Finally, the virus hooks the VxDCall and executes the host. From then on, the virus actively monitors VWIN32_Int21Dispatch services without having any specific VxD part. This simplifies the virus structure significantly.

Infection/Own VxDCall Handler.

The new VxDCall handler monitors all the important, *Windows 95* long filename functions (7143h, 714Eh, 714Fh, 7156h, 716Ch, 71A8h) coming in as an Int21_Dispatch service code (2A0010h) and all file open requests for read or write. The virus also monitors the get date function for its 'Are you there?' call.

The virus carefully saves the host program's original attributes and its date-stamp, resetting them after infection. In EXE, SCR, BMP and SYS files, the virus looks for the extension of the accessed file. The above files are checked for their internal structure. If the first bytes are 'MZ' and the 'PE' mark is in place, the virus tries to infect the host program (if it has 386-compatible code). If the file starts with 'BM', the virus checks if the actual image is a non-compressed bitmap file. If so, the virus performs its activation routine.

Files whose sizes can be divided exactly by 101 are assumed to be infected. This is the same self-check that Cabanas uses. The 29A group may have decided to use this simple trick to avoid double infections by their different viruses. During the infection process, the virus uses only Int21_Dispatch service routines. It modifies the entry-point of the PE header to point to the end of the program. It also alters the last section header to fit its own code, and

changes the flags in order to execute code in that section. Then it calls its generic encryption function and encrypts its image with this routine. After that, it writes itself to the end of the host program with the polymorphic decryptor created at installation and some additional bytes at the end to make its size exactly divisible by 101. HPS saves the entire virus size as the last four bytes of the file for later use by its directory stealth handler.

When infection is complete, HPS looks for various anti-virus checksum files such as ANTI-VIR.DAT, AVP.CRC, CHKLIST.MS, IVB.NTZ. If these are in the same directory as the host program, the virus deletes the file, after taking the precaution to clear the file's attribute. Finally, it gives control to the original VxDCall.

Directory Stealth Handler

HPS is a directory stealth virus. It does not hide file size changes from the DIR command, as it only monitors 714Eh, 714Fh LFN FindFirst/FindNext functions.

Implementation of the stealth handler is unique. On the fly, the virus patches the return address of FindFirst/FindNext functions on the stack to its own handler. This handler checks whether the file size is exactly divisible by 101. If so, the virus opens the program with an extended open LFN function. It reads the virus size from the last four bytes of the infected program and subtracts this value from the value returned by FindFirst/FindNext. This way the virus is able to hide its file size changes from most applications, despite having a variable infective length.

Polymorphic Engine

HPS has a polymorphic engine as powerful and advanced as that of Implant. It supports subroutines (using CALLs and RETs) and conditional jumps with non-zero displacement. The polymorphic engine comprises about half the virus code. There are blocks of random bytes inserted in the decryptor's generated code chain. The full decryptor is built up during the first initialization phase which makes the virus a slow polymorphic. This means that anti-virus vendors cannot test their scanner's detection rate against this virus efficiently, because the infected PC has to be rebooted in order to create a new decryptor. The decryptor consists of 386-specific instructions. The image is encrypted and decrypted by different methods including XOR, INC, DEC, NOT, ADD and SUB instructions with 8-, 16- or 32-bit keys. This reduces the range of detection shortcuts rather drastically. I am sad to say that the polymorphic engine part is well written, just like the rest of the virus. It was certainly not created by a beginner.

Activation Routine

Since the virus only checks the date during initialization, it will only activate when the infected PC is booted on a Saturday (or the virus is executed on a Saturday for the first time). If a non-compressed bitmap file has been opened, the virus allocates enough memory for the image and horizontally flips the picture. HPS patches the value DEADBABEh into the end of the bitmap header area to avoid flipping the same image again. Since non-compressed bitmap files are frequently used by *Windows 95*, this causes all kinds of weird effects.

Conclusion

I had not finished writing this article when another polymorphic virus called Win95/Marburg, also the work of GriYo, was discovered. Marburg has a similar infection method to Cabanas, but it only works under *Windows 95* due to a small bug in the replication code. It would be easy to fix this bug, thus I suspect we will see Win32 (*NT*-compatible) polymorphic viruses in the near future.

We also anticipate the emergence of a polymorphic mutation engine library, callable by Win32 viruses in a few months' time. Moreover, we will have to face more and more weird graphical effects under 32-bit *Windows* systems caused by *Windows 95* and Win32 viruses. It seems virus writing is going back to its roots. While Cascade exploited the possibilities of DOS by way of graphics, *Windows* viruses have endless resources for hair-raising activation routines. These will most certainly challenge us in 1998.

HPS

Aliases:	Win95/HPS, Hanta.
Type:	*Windows 95/98* PE infector, hooks VxDCall, directory stealth.
Self-recognition in Files:	Files whose size is exactly divisible by 101 are assumed infected.
Self-recognition in Memory:	VxDCall (VWIN32_Int21Dispatch get date) on call ESI=48505321h, EDI=5453523Fh returns ESI=59455321h if the virus is active.
Hex Pattern in PE files:	Not possible.
Intercepts:	VxDCall, VWIN32_Int21Dispatch services.
Payload:	Flips the image in non-compressed BMP files horizontally on Saturdays.
Removal:	Under clean system conditions, restore infected files from backups or replace with originals.