JANUARY 2005

virus BULLETIN

The International Publication on Computer Virus Prevention, Recognition and Removal

CONTENTS

2 **COMMENT**
The buck stops here, there, everywhere

3 **NEWS**
Test files: straightening the record
Synchronized malware identification for the new year

3 **VIRUS PREVALENCE TABLE**

VIRUS ANALYSES
4 Time machine
6 Reduce, reuse, recycle: W32/Orpheus.A

FEATURES
9 Are metamorphic viruses really invincible? Part 2
12 The three faces of VBA: Part 1

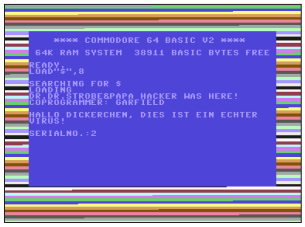
18 **CALL FOR PAPERS**
VB2005 Dublin

19 **CONFERENCE REPORT**
AVAR 2004: Eutaxy or chaos?

20 **END NOTES & NEWS**

ISSN 0956-9979

IN THIS ISSUE



GOING RETRO

Remember when Robert Palmer was *Addicted to Love*, Dire Straits were doing the *Walk of Life* and Lionel Richie was *Dancing on the Ceiling*? In 1986, Sir Richard Branson beat the Atlantic speed record, Prince Andrew married Sarah Ferguson and Sir Clive Sinclair sold his computer business to rival *Amstrad*. Meanwhile, in the AV world, C64/BHP.A quietly became the first full stealth file-infecting virus. Dust off your stone-washed jeans, get out your fingerless gloves and prepare to go retro!

page 4

THREE-FACED MACROS

Macros written in Visual Basic for Applications have three very different forms, any of which can be the one that is executed. To illustrate why this poses a problem, Dr Vesselin Bontchev lets us in on some of his specialist knowledge of macro viruses.

page 12

vbSpam supplement

This month: anti-spam news & events; John Graham-Cumming looks at the content obfuscation techniques of a spam program; and we review O'Reilly's *Spam Kings*.

'People want to know who should hold ultimate responsibility for ensuring that our lives are virus free.'

Andrew Radley, StreamShield Networks, UK

THE BUCK STOPS HERE, THERE, EVERYWHERE

Opening an email inbox first thing in the morning is one of the most stressful aspects of living in the 21st century. For one third of us this means facing a slew of email of which half is spam, according to a recent MORI poll. Add to that the concerns over malware and you have a headache we could all do without.

Users are faced with a plethora of firewall, anti-virus and anti-spam software, as well as a constant stream of patches and fixes, all to allow us to communicate safely. It is no wonder that people want to know who should hold ultimate responsibility for ensuring that our lives are virus free.

Against a background of growing cybercrime that already stretches government and police manpower and budgets, ISPs have begun integrating anti-virus solutions and spam blockers, whilst banks have had little choice, in the wake of recent phishing scams, but to place the responsibility for securing communications firmly on the shoulders of the users.

It seems that combating viruses and spam is still left to the individual business or home user. This is becoming an increasingly difficult task, and currently only one third of users trust that their anti-virus software will protect them when an attack comes (MORI).

Simply put, no single group has been willing to take the responsibility for preventing virus and spam traffic on the Internet. The MORI research indicates clearly where most users feel the responsibility lies: only 4 per cent felt it should be up to the individual, and whilst 10 per cent believed the government had a leading role to play, the majority (58 per cent) of those questioned said they would like their ISP to provide additional security services to protect them from spam, viruses and offensive content on the Internet.

But before the ISPs start to sweat, they should consider the advantages such a policy could deliver. Most of those questioned in the poll said they would be happy to pay a further £2 or more on top of their monthly Internet access charges to guarantee a clean Internet service. Critically, two thirds of respondents said they would switch service provider if an alternative provider offered protection capabilities as part of their Internet access portfolio.

As any army general would tell you, the best place to form a front line of defence is the place furthest away from the things you are trying to protect. Defend too close and you run the risk that the enemy will break through, leaving you with no defences at all. The best place to stop threats coming from the Internet, therefore, is in the Internet itself.

In the early days of the water industry people were so concerned about the water quality that they boiled it before use. Today, all water companies clean their water before delivery, and users know it is safe to drink straight out of the pipe. This is a model the ISPs need to emulate.

The first steps towards Internet-based content security have already been made by consumer ISPs offering anti-virus and anti-spam protection for their own email services, but this technology is already being outstripped. ISPs need to consider self-propagating worms that infect systems, embedded viruses that download from web pages, viruses in files downloaded directly from the Internet such as in peer-to-peer file sharing services, and instant messaging and text messaging spam. A good ISP will demonstrate responsibility for preventing all these threats.

All-encompassing Internet-based content security is the holy grail of many businesses and individuals that are struggling to defend themselves against the onslaught of attacks from the Internet. As technology makes these capabilities an affordable reality, it is clear that the ISPs must respond to market demands and finally accept responsibility for securing the Internet for all.

NEWS

TEST FILES: STRAIGHTENING THE RECORD

Andreas Clementi, of the University of Innsbruck, has asked *VB* to set the record straight regarding the collection of files referenced in Peter Morley's letter 'Generic detection – a specific case' (see *VB*, December 2004, p.14). In his letter, Peter writes: '*McAfee* received (from Andreas Clementi of the University of Innsbruck) a collection of some 1,350 *.HTM files … Should we detect these files, for any reason other than the fact that we may be reviewed against them?' Andreas says: 'None of those files were, are, or will be used in my tests – but the matter is out of my hands should any other reviewer use any of the files by mistake or through ignorance. I stated clearly when I sent the files that none of the files are used for my tests.' Consider the record straightened.

SYNCHRONIZED MALWARE IDENTIFICATION FOR THE NEW YEAR

Causing a stir in the anti-virus community last month was the announcement of a new US-led initiative whose aim is to achieve threat synchronization.

The US Department of Homeland Security's Computer Emergency Readiness Team, US-CERT, is set to coordinate a Common Malware Enumeration (CME) initiative, according to a letter sent to the SANS Institute and signed by representatives of the DHS, *Symantec*, *Microsoft*, *McAfee*, and *Trend Micro*. Rather like *Mitre Corp*'s Common Vulnerabilities and Exposures (CVE) list, US-CERT plans to maintain and coordinate a database of malware identifiers.

The letter stated: 'By building upon the success of CVE and applying the lessons learned, US-CERT, along with industry participants ... hopes to address many of the challenges that the anti-malware community currently faces.' The letter acknowledged that the task would not be a straightforward one, saying: 'There are significant obstacles to effective malware enumeration, including the large volume of malware and the fact that deconfliction [*sic*] can be difficult and time-consuming.'

With such an enormous task ahead, the enumeration project will make a start with just the 'major' threats. The initial proposal, therefore, is for representatives of the companies involved to forward samples that are submitted to AVED (Anti-Virus Emergency Discussion network) to US-CERT, allowing US-CERT to generate a CME number for each new threat.

Participants in the initiative acknowledge that this is not an 'end-all' solution to the malware-naming problem, but represents a helpful step forward. *VB* awaits the introduction of the scheme with interest.

Prevalence Table – November 2004

Virus	Type	Incidents	Reports
Win32/Netsky	File	104,451	50.95%
Win32/Bagle	File	64,526	31.48%
Win32/Sober	File	29,131	14.21%
Win32/Bagz	File	1,601	0.78%
Win32/Mydoom	File	943	0.46%
Win32/Mabutu	File	936	0.46%
Win32/Dumaru	File	744	0.36%
Win32/Zmist	File	619	0.30%
Win32/Funlove	File	585	0.29%
Win32/Zafi	File	371	0.18%
Win32/Klez	File	262	0.13%
Win32/Valla	File	116	0.06%
Win32/MyWife	File	87	0.04%
Win32/Parite	File	86	0.04%
Win32/Bugbear	File	85	0.04%
Win32/Lovgate	File	75	0.04%
Win95/Spaces	File	63	0.03%
Win32/Pate	File	52	0.03%
Win32/Mimail	File	44	0.02%
Win32/Kriz	File	40	0.02%
Win32/Swen	File	37	0.02%
Win95/Tenrobot	File	29	0.01%
Win32/Elkern	File	21	0.01%
Win95/CIH	File	16	0.01%
Win32/SirCam	File	10	0.00%
Win32/Evaman	File	9	0.00%
Win32/Torvil	File	8	0.00%
Win95/Kuang	File	8	0.00%
Psyme	Script	6	0.00%
Win32/Magistr	File	6	0.00%
IEStart	Script	4	0.00%
Win32/Hybris	File	4	0.00%
Others[1]		21	0.01%
Total		204,996	100%

[1]The Prevalence Table includes a total of 21 reports across 15 further viruses. Readers are reminded that a complete listing is posted at http://www.virusbtn.com/Prevalence/.

VIRUS ANALYSIS 1

TIME MACHINE

Peter Ferrie
Symantec Security Response, USA

It is commonly reported that the first known full stealth file-infecting virus was Frodo, in 1989. In fact, that is true only for the *IBM PC* world. The *Commodore 64* world had been infected three years earlier by what was perhaps *truly* the first full stealth file-infecting virus: C64/BHP.A (not to be confused with the boot-sector virus for the *Atari*, also known as BHP).

All of the descriptions of BHP that were published at the time were inaccurate, some of them even giving incorrect descriptions of how the infection worked. This article takes a look at what it really did.

BASIC INSTINCT

As with all *Commodore 64* programs, BHP began with some code written in Basic. This code consisted of a single line, a SYStem [*sic*] call to the assembler code, where the rest of the virus resided. Unlike many programs, the virus code built the address to call dynamically. This implies that the virus was perhaps written by a very careful coder, but it proved to be unnecessary because the address did not change in later versions of the machine.

Once the assembler code gained control, it placed itself in the block of memory that was normally occupied by the I/O devices when the ROM was banked-in.

At this point, it is necessary to describe some of the *Commodore 64* architecture in more detail.

DOWN MEMORY LANE

The *Commodore 64* used a MOS 6510 CPU, a later version of the MOS 6502 chip used by several competing machines of the time, including the *Apple II* series and the *Atari 400* and *800*.

Since the 6502's data bus (and therefore the 6510's data bus) was only 16 bits wide, the maximum directly addressable memory range was 64kb. In order to accommodate more memory, a 'banking' architecture was implemented, allowing different memory regions to be mapped in under the user's control, simply by writing the appropriate value to a specific memory-mapped port.

NOW YOU SEE ME ...

The *Commodore 64* allowed quite a large address space in comparison with other machines at that time: potentially eight banks of 64kb (a total of 512kb!) of memory were available, though most machines did not have the chips installed to provide that much.

Since the mapped regions all needed to be within the 64kb range, a few memory ranges provided the base for all of the banked memory, in order to give the maximum amount of memory that would always be available. This greatly reduced the complexity of the average program.

On the other hand, however, several steps were required for a program that ran in one memory bank to access data in another memory bank. The first step was to place code in non-banked memory and run it. The next steps were for that code to bank out the program, bank in the required data, access that data and save them, then bank out the data, bank in the program again, restore the data, and return control to the program.

... NOW YOU DON'T

A side effect of memory banking was that it was a great way to hide a program, since the program was not visible if its memory was not banked in. This is the reason why BHP placed its code in banked memory.

After copying itself to banked memory, the virus restored the host program to its original memory location and restored the program size to its original value. This allowed the host program to execute as though it were not infected. However, at this time the virus would verify the checksum of the virus's Basic code, and would overwrite the host memory if the checksum did not match.

An interesting note about the checksum routine is that it missed the first three bytes of the code, which were the line number and SYS command. This made the job easier for the person who produced the later variant of the virus. Although the later variant differed only in the line number, this was sufficient to defeat the BHP-Killer program, because BHP-Killer checked the entire Basic code, including the line number.

CAPTAIN HOOK

The virus checked whether it was running already by reading a byte from a specific memory location. If that value matched the expected value, the virus assumed that another copy was running. Thus, writing that value to that memory location would have been an effective inoculation method. Similar methods were used against viruses for the *Commodore Amiga* machines.

If no other copy of the virus was running, the virus would copy some code into a low address in non-banked memory, and hook several vectors, pointing them to the copied code.

Specifically, it hooked the ILOAD, ISAVE, MAIN, NMI, CBINV and RESET vectors.

The hooking of MAIN, NMI, CBINV and RESET made the virus Break-proof, Reset-proof, and Run/Stop-Restore-proof. These hooks ensured that the virus did not lose control while the machine restarted. This technique was similar to the Ctrl-Alt-Delete hooks that were used later in DOS viruses on the *IBM* PCs, or the Ctrl-Amiga-Amiga hooks that viruses used on the *Commodore Amiga*.

Once the hooks were in place, the virus ran the host code. The main virus code would be called on every request to load or save a file.

HEAVY LOAD

The ILOAD hook was reached when a disk needed to be searched. This happened whenever a directory listing was requested, and could happen when a search was made using a filename with wildcards, or the first time that a file was accessed. Otherwise, the drive hardware cached up to 2kb of data and returned it directly.

The virus called the original ILOAD handler, then checked whether an infected program had been loaded. If an infected program had been loaded, the virus restored the host program to its original memory location and restored the program size to its original value. Otherwise, even if no file had been loaded, the virus called the infection routine.

DON'T FORGET TO SAVE

The ISAVE hook was reached whenever a file was saved. The virus called the original ISAVE handler to save the file, then called the infection routine.

The infection routine began by checking that the requested device was a disk drive. If so, then the virus opened the first file in the cache. The first file in the cache would be the saved file if this code was reached via the ISAVE hook, otherwise it would be the first file in the directory listing.

If the file was a Basic program, then the virus performed a quick infection check by reading the first byte of the program and comparing it against the SYS command. The virus read only one byte initially, because disk drives were serial devices on the *Commodore 64*, and therefore very slow. However, if the SYS command was present, the virus verified the infection by reading and comparing up to 27 subsequent bytes. A file was considered infected if all 27 bytes matched.

If the file was not infected, the virus switched to reading data from the hardware cache. The first check was for a standard disk layout: the directory had to exist on track 18, sector 0, and the file to be infected had not to have resided on that track.

LESS TALK, MORE ACTION

If these checks passed, the virus searched the track list for free sectors. It began with the track containing the file to be infected, then moved outwards in alternating directions. This reduced the amount of seeking that the drive had to perform in order to read the file afterwards, and was a very interesting optimisation, given that some multi-sector boot viruses on the *IBM PC* placed their additional code at the end of the disk, leading to very obvious (read: audible) seeking by the drive.

If at least eight free sectors existed on the same track, then the virus allocated eight sectors for itself and updated the sector bitmap for that track. The code to update the sector bitmap was beautiful, allocating the sectors and creating the list of sector numbers at the same time. The code could have been shortened slightly, though, by reordering some of the instructions.

This was the case throughout the virus – overall, the code was very tight (as it needed to be), but there were some pieces of code that could have been optimised in very obvious (and some less obvious) ways. There were also a couple of harmless bugs. However, given the size of the code, the only resulting advantage would have been that the payload (see below) could have contained a longer message or more effects.

By comparison, the code used to write the virus to the disk was a horrible mess – suggesting, perhaps, that it was written by a co-author. The virus wrote itself to disk in the following manner: the first sector of the host was copied to the last sector allocated by the virus, then that first sector

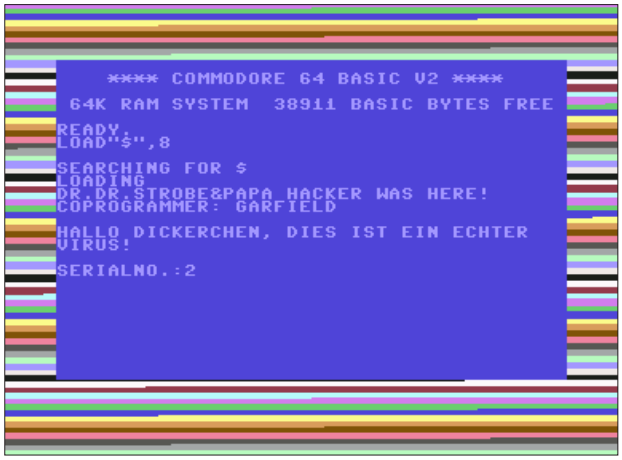


Figure 1. BHP's payload. The text was displayed one character at a time, while the colours of the border cycled.

was replaced by the first sector of the virus. After that, the remaining virus code was written to the remaining allocated sectors.

The directory stealth was present here, and it existed without any effort on the part of the virus writer(s). It was a side effect of the virus not updating the block count in the directory sector. The block count was not used by DOS to load files, its purpose was informational only, since it was displayed by the directory listing.

In fact, the same problem existed on DOS for the *Apple II* series of machines and such a virus would have been much easier to write there, since communication with the hardware is much simpler on those machines. The only obvious effect in the case of BHP was that the number of free blocks on the disk was visibly reduced, because the value was calculated using the sector bitmap, not the directory listing.

PAYLOAD

After any call to ILOAD or ISAVE, the virus checked whether the payload should activate. The conditions for the payload activation were the following: that the machine was operating in 'direct' mode (the command-prompt), that the seconds field of the jiffy clock was a value from 2–4 seconds, and that the current scan line of the vertical retrace was at least 128. This made the activation fairly random. The payload was to display a particular text, one character at a time, while cycling the colours of the border (see Figure 1).

The serial number that was displayed was the number of times the payload check was called. It was incremented once after each call, and it was carried in replications. It reset to zero only after 65,536 calls.

CONCLUSION

So now we know: BHP was a virus ahead of its time.

C64/BHP.A	
Size:	2030 bytes.
Type:	Memory-resident parasitic prepender.
Infects:	*Commodore 64* Basic files.
Payload:	Displays text under certain conditions.
Removal:	Delete infected files and restore them from backup.

VIRUS ANALYSIS 2

REDUCE, REUSE, RECYCLE: W32/ORPHEUS.A

John Canavan
Symantec, Ireland

The building of generic objects and structures for optimum code reuse is a goal of software developers throughout the IT industry. Effective reuse of code can significantly speed up development time, ease debugging and result in more consistent-looking and reliable applications. And it seems that even virus writers have recognised the benefits of reusing code – in recent times they have made an art of hacking together pieces of stolen code to suit their objectives.

With code available on the Internet for threats like Gaobot, Spybot and Beagle, we have seen thousands of samples that have been tailored specifically to fulfil the desires of script-kiddies.

Taking a unique perspective on code reuse, one of the latest of these Frankenstein creations is W32/Orpheus.A.

MYTHOLOGY

In ancient Greek mythology Cerberus was the gruesome many-headed dog that guarded the gates of Hades, the realm of the dead. Stories tell of two men great enough to have passed him, one of whom was Orpheus. When his wife died (having been bitten by a poisonous snake on their wedding day), Orpheus used his musical abilities to charm the savage dog, and make his way to Hades to claim back his wife.

W32/Orpheus.A does not have the musical charm of its namesake, however. Instead, it bypasses the computer system's watchdogs by using advanced rootkit techniques such as API patching and DLL injection.

The lion's share of the worm's code is taken from a publicly available Romanian DLL injection-based rootkit known as Vanquish, written by 'xShadow'. Hacked together with this, Orpheus makes use of named pipes for its backdoor communication, and uses the output of the system command 'net view' for enumeration of network resources during its propagation.

The original Vanquish rootkit comprised a DLL containing the main functionality of the rootkit and a small executable used to load it. The author of W32/Orpheus.A kept this model, but added some extra bells and whistles to the executable in an attempt to make it more difficult to remove from the infected system. An analysis of both modules of W32/Orpheus.A follows.

HOTPLUG.EXE

Hotplug.exe is the executable stub of W32/Orpheus.A. It is used to create services so that the virus is executed on system startup, and starts the DLL module of the virus by thread injection.

Upon execution, Orpheus checks the following registry value:

HKEY_LOCAL_MACHINE\Software\Cerberus\1.1\ "dontInstall" = "1"

If this value is present, the worm performs a basic uninstall routine, which attempts to stop and delete the 'Hotplug' service. If the value is not present, Orpheus proceeds to copy itself to the System directory of the infected host as 'Hotplug.exe'.

Orpheus also attempts to move its associated DLL to %System%\ntadint.dll, or to %System%\msvcr71.dll if the first fails. Hotplug.exe expects its DLL component to be present in the root directory of the drive from which it is executing.

Orpheus then makes the first of its clever social engineering moves. The worm creates a service named Hotplug, assigning it the following description:

'Enables automated driver loading for hotpluggable devices, including USB, FireWire and Hotplug PCI systems. If this service is stopped, hotplug devices will no longer function. If this service is disabled, any services that explicitly depend on it will fail to start.'

Orpheus then adds a DependOnService hook to the PlugPlay service key

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ Services\PlugPlay\"DependOnService"="hotplug"'

This ensures that the Hotplug service is loaded successfully before *Microsoft Plug n Play* services are initiated. The Hotplug service will also show up in the Dependencies tab of the Services Management Console for each of the service's properties, thus lending it further legitimacy.

Using CreateToolhelp32Snapshot, Process32First, and Process32Next, Orpheus iterates through the process list searching for 'lsass'. When a match is found the worm attempts to load its DLL into the matched process.

The most common means of DLL injection is through the use of the CreateRemoteThread() API. However, Orpheus implements a more innovative method: it uses register settings gathered from GetThreadContext() to force code execution. As is evident from the name of the mutex that is created during injection ('VanquishAutoInjectingDLL'), this portion of the code has been lifted from the Vanquish rootkit package.

After finding a process to infiltrate, Orpheus uses OpenProcess() and a ReadProcessMemory() call to find the ThreadID of the first thread in this process. The thread is then suspended and 0x2f bytes of memory are allocated within LSASS. The worm then calls GetThreadContext(), and learns the values of the registers EAX, EBX, ECX, EDX, ESI and EDI as well as the stack pointers (ESP, EBP) and the instruction pointer (EIP) from the returned CONTEXT structure.

Orpheus then uses this information to write the following code and data to the newly allocated 0x2f byte buffer in LSASS:

```
"szNTADINT.dll"
pushad
push <szNTADINT_dll>
mov esi, <LoadLibraryW>
call esi
popad
ret
```

This is the equivalent of calling LoadLibrary("NTADINT.dll").

If the buffer write succeeds, Orpheus must adjust the stack so that execution is not interrupted after the injected virus code has been run. This is achieved by adding the value of the thread CONTEXT instruction pointer, our new return address, to the stack. EIP is then set to the address of the 0x2F byte buffer just after the 'NTADINIT.DLL' string, so that the injected code is executed immediately once the thread resumes. Loose ends are then tied up as Orpheus updates the thread CONTEXT with the new values of ESP and EIP using SetThreadContext(), and the thread is resumed.

Before exiting Hotplug.exe will enumerate through all windowed processes using EnumWindows(), attempting to inject NTADINT.DLL into each.

NTADINT.DLL

Ntadint.dll contains the main functionality of W32/Orpheus.A. It patches APIs to hide itself, hosts the worm's backdoor functionality, logs keystrokes and attempts to spread to other vulnerable hosts on the network.

Initially executed by injection, ntadint.dll proceeds to patch 34 important system APIs in an attempt to hide its presence on the system.

API PATCHING – TRAMPOLINING

The effective API replace is done by overwriting the first five bytes of the API prologue with a 32-bit offset unconditional jmp to the new patched API that lives in the injected DLL (ntadint.dll).

First, Orpheus sets the access protection on the first 0xB bytes at the address of the API to be patched to PAGE_EXECUTE_READWRITE.

Once this is complete, the worm saves the first five bytes of the API for later use (they will need to be used when the API is called) and replaces them with the jump to its trampoline function. Finally, FlushInstructionCache() is called, as even though the instructions have changed in memory, old code may still run from the cache.

Each trampoline function contains code that will restore the first five bytes of the API, in a manner similar to that of the initial patch, and executes the clean version if required.

Patched API calls are filtered based on the arguments provided to hide the processes, files and registry settings associated with the worm. Orpheus will also attempt to inject ntadint.dll into the calling process of the patched API, using the DLL injection technique described above.

The APIs Orpheus has chosen to patch are selected carefully to make it as difficult as possible to remove the virus from the system. They are as follows:

CreateProcessA	CreateProcessW
LoadLibraryExW	FreeLibrary
CreateProcessAsUserA	CreateProcessAsUserW
FindFirstFileExW	FindNextFileW
RegCloseKey	RegEnumKeyW
RegEnumKeyA	RegEnumKeyExW
RegEnumKeyExA	RegEnumValueW
RegEnumValueA	RegOpenKeyExW
RegQueryValueExW	EnumServicesStatusW
EnumServicesStatusA	DispatchMessageW
DispatchMessageA	GetMessageW
GetMessageA	LogonUserA
RegQueryMultipleValuesA	LogonUserW
RegQueryMultipleValuesW	WlxLoggedOutSAS

The patched versions of these APIs allow the worm to hide the presence of its files, services and associated registry keys so that, essentially, it will be invisible both to the user and to any security products monitoring the system.

SO LONG, AND THANKS FOR ALL THE FISH!

When the worm is injected into Explorer.exe, it opens a back door on the system to accept unauthorised remote connections. The most orthodox backdoor is a bound and listening TCP port. However, following a technique pioneered by Backdoor.Fluxay, Orpheus uses a named pipe to accept its commands.

The worm creates the following named pipe and enables it to wait for a client process to connect by using ConnectNamedPipe():

\\.\pipe\cb_win_nt_proc_rpc_[current process id]

The current process id is retrieved using GetCurrentProcessID() before the pipe is created. Tellingly, however, the worm does nothing to notify the author of this pipe name or the machine's IP address.

The remote user can connect to the backdoor using CreateFile() or CallNamedPipe() and once connected has access to the following commands:

`version`	Responds with 'Cerberus 1.1'.
`details`	Responds '##Master, I am here to serve you...'.
`newuser`	Creates a new user on the system.
`changepassword`	Changes an account password on the system.
`deleteuser`	Deletes an account from the system.
`clearsystemlog`	Clears the system log – OpenEventLogA(), ClearEventLogA().
`showvanquishlog`	Shows the debugging logfile generated.
`showkeylog`	Shows the keystroke log.
`clearkeylog`	Clears the keystroke log.
`clearvanquishlog`	Clears its own log.
`remoteinstall`	Attempts to infect a specified remote machine via network shares.
`exec`	Executes a specified command – CreateProcessA().
`shellexec`	Executes a specified command – cmd / C start [command].
`killhost`	Terminates a specified process.
`insecure`	Sets ACLs on drives C through G so that everyone has full control access. Sets up open shares of drives C through G. Uses cacls and net share commands.
`doshost`	Performs a basic DoS attack on the infected host – enters a recursive infinite loop creating threads which attempt to allocate a 100,000 byte block of memory on the heap and initialise each byte to 0xff.

`exit`	Responds 'So long, and thanks for all the fish!', disconnects the named pipe and ends the backdoor process.

KEYLOGGING

As well as using its patched APIs to grab user login details the worm also sets a system-wide hook to monitor keystrokes. Each keystroke is recorded with the currently logged-in username, current time, and the active process.

If the active process is *Internet Explorer*, Orpheus also attempts to log the window title retrieved using GetClassName(). Logged keystrokes are stored in the file c:\irdos.sys.

NETWORK PROPAGATION

The worm's propagation routine iterates through each of the domains gathered from the output of 'net view /domain'. If the worm determines it has admin rights on the domain it will begin attempting to connect to listed hosts.

The remote installation procedure will copy ntadint.dll and hotplug.exe to the target host, dropping copies to both the system directory and the root C and D drives. It will then use the service control manager to create the services 'Hotplug Devices Manager' and 'Microsoft Windows Hotplug Service' so that they are executed on startup.

This method of infection means that the worm will spread very quickly if it is run on a domain with admin rights, but without any significant rise in network traffic. There is no pinging of random IPs or sending of the worm unnecessarily – however, in such a situation it is not likely to spread outside the context of that organisation.

CONCLUSION

The advanced DLL injection and API patching techniques demonstrated by W32/Orpheus.A mean that it would be extremely difficult to detect and remove the worm from an infected system if it were not caught before it was executed.

Its injection routines would allow it to bypass easily any system firewalls installed with outbound connections appearing to come from explorer.exe, lsass.exe or whatever executable the DLL is loaded in, and its patching techniques ensure its files are hidden comprehensively from the user – and, indeed, from most corporate anti-virus products.

Unless efficient memory scanning and rootkit bypass mechanisms are incorporated into anti-virus products in the near future, threats like W32/Orpheus.A will be able to have free run of the hapless users' systems, and indeed their entire networks.

FEATURE 1

ARE METAMORPHIC VIRUSES REALLY INVINCIBLE? PART 2

Arun Lakhotia, Aditya Kapoor, Eric Uday
University of Louisiana at Lafayette, USA

Metamorphic viruses thwart detection by signature-based (static) AV technologies by morphing their code as they propagate. The viruses can also thwart detection by emulation-based (dynamic) technologies. To do so they need to detect whether they are running in an emulator and change their behaviour. So, are metamorphic viruses really invincible?

In part one of this article (see *VB*, December 2004 p.5) we presented an overview of mutation engines, followed by a discussion of the Achilles' heel of a metamorphic virus: its need to analyse itself. In this part of the article we present a case study in which we look at the metamorphic engine of the virus W32/Evol. This leads to a discussion on developing 'reverse morphers' to undo the mutations performed by a mutation engine. The article closes with our conclusions.

W32/EVOL: A CASE STUDY

W32/Evol is a relatively simple metamorphic virus. Nonetheless, it is a good example for a case study since the virus demonstrates properties that are common to all metamorphic viruses, i.e. it obfuscates calls made to system libraries and it mutates its code prior to propagation.

The rest of this section describes the details of these methods.

OBFUSCATING SYSTEM CALLS

In order to perform a malicious act, a program must access the disk or the network. Access to these resources is controlled by the operating system. A quick way to determine whether a program is malicious is to look at the system calls it makes.

W32/Evol does not use a 'normal' procedure to make system calls – it obfuscates its calls, which means that a disassembler such as *IDAPro* cannot determine directly the system calls it makes. W32/Evol uses the following strategies to obfuscate its calls:

1. It computes the address of the kernel32.dll function GetProcAddress() by searching for the eight-byte sequence [0x55 00 01 F2 51 51 ec 8b] on *Windows 2000*. (The W32/Evol binary at http://vx.netlux.org/ looks for the byte sequence [0x55 00 00 0f 51 51 ec 8b], which is probably for a different version of *Windows*.)

2. It keeps the address of GetProcAddress() in its stack-based global data store, maintained at a certain distance from a magic marker pushed on the stack.
3. It uses a 'return' instruction to make a call to GetProcAddress().
4. It maintains the names of functions to be called as immediate, double-word operands of multiple instructions, not as strings in the data store.

MUTATION ENGINE

The mutation engine of W32/Evol is a function consisting of the Disassembly and Transform modules described in part 1 of this article (in *VB,* December 2004). It does not have a Reverse Engineering module since it transforms one instruction at a time.

The mutation engine is located at address 00401FD7. The engine takes three inputs (all values quoted here are the values recorded during a test run of W32/Evol in a debugger):

1. The Relocatable Virtual Address (RVA) of loaded virus code: RVA = 401000.
2. The length of the original virus code: LEN = arg_4 (1847).
3. Pointer to buffer (BUF1) to store the transformed code: (Max Size buffer = 4 x LEN = arg_8 (7F0000).

The output of the engine is the transformed program, which is placed in the buffer BUF1.

DISASSEMBLY MODULE

The disassembly module of W32/Evol uses the linear sweep algorithm. It checks whether a byte starts an instruction, if it does then it gets the size of the instruction, and disassembles the byte following the instruction. If, during disassembly, the program comes across a byte that is not an instruction, the mutation process is abandoned (see Figure 3).

The mutation engine processes only a limited range of opcodes of the x86 instruction set. For instance, it does not process floating-point instructions. The mutation is abandoned if an instruction outside its accepted range is encountered. Figure 4 shows the code fragment from W32/Evol performing the instruction range check.

TRANSFORM MODULE

The Transform module maps an instruction into one or more instructions. A detailed list of all the transformations is given in the appendix to this article, which can be found at http://www.virusbtn.com/magazine/articles/features/2005/01_01.xml.

```
Location    Instruction
0040227A    cmp    al, 0FEh
0040227C    jz     short loc_402282
            ; If the byte under analysis is FE
            ; goto 00402282
0040227E    cmp    al, 0FFh
            ; If the byte is FF goto 00402282
00402280    jnz    short loc_4022B5
            ; compare al with next opcode.
00402282    mov    al, [esi+1]
            ; If byte is either 0xFE or 0xFFload ModR/M
            ;  byte in al
00402285    and    al, 38h
00402287    ror    al, 3
0040228A    cmp    al, 7
0040228C    jz     loc_402532
            ; If value of bits 3, 4, 5 of ModR/M byte are
            ; 1 the instruction does not exist
            ;  Exit mutation process
```

Figure 3. Invalid instruction check.

```
Location    Instruction
00402118    cmp    al, 0Fh
                   ; Checking for two-byte opcode.
0040211A    jnz    short loc_402152
                   ;compare al with next opcode.
0040211C    mov    cl, [esi+1]
0040211F    cmp    cl, 80h
00402122    jb     loc_402532
            ; If byte following 0x0F is less than 0x80
            ; then exit mutation process
00402128    cmp    cl, 90h
0040212B    jnb    loc_402532
            ;  If byte following 0x0F is greater than
            ; 0x90 then exit mutation process
```

Figure 4. Invalid instruction 'range' check.

The transformation rules can be classified into two categories: deterministic and nondeterministic. A deterministic rule always transforms an instruction to the exact same sequence of instructions.

For example, the following rule for transforming the instruction movsb (opcode 0xA4) is a deterministic transformation rule:

```
movsb  ➔    push   eax
            mov    al, [esi]
            add    esi, 1
            mov    [edi], al
            add    edi, 1
            pop eax
```

Figure 5 shows the procedure for generating a fixed transformation for byte 0xA4 representing movsb.

A non-deterministic rule may transform an instruction to a different sequence of instructions. The following two rules demonstrate non-deterministic rules:

```
mov eax, [ebp+4]    ➔    push ecx
(8B 45 04)               mov  ecx, ebp
                         add  ecx, 41h
                         mov  eax, [ecx-3Dh]
                         pop  ecx
```

```
Location    Instructions
004023B0    cmp al, 0xA4
            ; If byte is not 0xA4 goto next step
004023B2    jnz  004023CE
004023B4    add esi,1
            ; Increment esito analyze next byte
004023B7    mov eax, 83068A50
004023BC    stos dword ptr es:[edi]
004023BD    mov eax, 78801C6
004023C2    stos dword ptr es:[edi]
004023C3    mov eax, 5801C783
004023C8    stos dword ptr es:[edi]
            ; If al contains 0xA4, insert the equivalent byte
            ; sequence 50 8A 06 83 C6 88 07 83 C7 01 58
            ; at  the buffer location pointed to by edi
004023C9    jmp 00401FF8
            ; goto analyze next byte
```

Figure 5. Transformation of byte 0xA4.

```
mov eax, [ebp+4]          ➔     push esi
(8B 45 04)                      mov  esi, [ebp+4]
                                mov  eax, esi
                                pop  esi
```

Whenever the code introduced by a rule modifies a register, say *reg*, which was not modified by the original instruction, the mutated code is wrapped between 'push *reg*' and 'pop *reg*' instructions.

PATCHING RELOCATABLE ADDRESSES

W32/Evol does not contain any jump and call instructions that use absolute addresses, rather all the branching instructions use relative jumps. The virus also contains no indirect jumps and calls, where the target address is available in a register or some other memory location.

Since the transformations replace one instruction with multiple instructions, the mutation engine must also modify the relative addresses of the jump and call instructions.

In order to update the relative addresses, the mutation engine maintains another buffer, BUF2, of size 16 x [length of virus code]. For each instruction of the virus program, BUF2 has four entries, as shown in Table 1.

The first entry of the table is Source, this points to the address of the n^{th} instruction in the virus code. The second entry, Dest, points to the address in BUF1 where the transformed virus code is stored. (*Note that the mutation engine takes BUF1 as input.*)

Entry 1 (DWord)	Entry 2 (DWord)	Entry 3 (DWord)	Entry 4 (DWord)
Source	Dest	Next address following opcode	Original offset

Table 1. A record in the buffer BUF2.

The other two entries in the table are zero unless the instruction carries a relocatable offset, in which case the third entry points to the address where the calculated offset is to be stored. The last entry stores the value of the current offset.

The change in the length of the code results in a change of relative addresses. To update the relative offsets, the algorithm searches for all the non-zero 'Entry 3' locations, i.e. instructions that have offsets.

If an instruction, *I*, with a non-zero offset is found, it adds the original offset (Entry 4) to Source (Entry 1), to obtain address *a*. Address *a* is the original destination address in the W32/Evol code. Since this destination address should start a valid instruction, there should be a valid record in BUF2 such that Source is equal to *a*. (*Note that BUF2 has records corresponding to each valid instruction in virus code.*)

The difference between the values of Dest at the location of instruction *I* and Dest at location *a* gives us the new offset. This offset gives the number of bytes that have been added in the transformed code. The offset is then patched back to the location pointed by Entry 3 at the location of instruction *I*.

DEFEATING W32/EVOL

W32/Evol is no longer considered to be a major threat – most current AV scanners can catch it owing to its relatively simple morphing engine. Yet it may be worth contemplating how this virus could be defeated. The insights could lead to the development of methods for defeating other metamorphic viruses.

W32/Evol uses some very interesting techniques to obfuscate system calls. It is probably beyond the scope of current static analysis techniques to undo these obfuscations and identify the system functions being called by the virus. It appears to be futile to follow that direction.

However, the limitations of the metamorphic engine of W32/Evol are clearly its weaknesses.

- It uses linear sweep for disassembling itself, the same method that is used by most disassemblers. Hence it can easily be disassembled.
- It cannot use indirect jumps and calls because it cannot transform them correctly. Thus, its control flow graph can be derived easily, thereby simplifying its reverse engineering.
- Its deterministic transformation rules essentially replace a certain byte with a certain fixed sequence of bytes. These rules can be applied in reverse.

- The code generated by non-deterministic transformation rules follows the pattern: push *reg*, *instructions*, pop *reg*, where the *instructions* does not contain *push* or *pop*. The *push* and *pop* instructions form a pair of parentheses. All such pairs are properly matched in the generated code. It should be possible to undo the transformation using a parenthesis-matching algorithm.

Now consider a program Undo.Evol that does the following: it disassembles a program using linear sweep and then applies the transformations of W32/Evol in reverse. The program continues to apply the transformations until none of the transformations can be applied. Will the Undo.Evol program help in detecting versions of W32/Evol?

Since the transformations of Win32.Evol always increase the code size, when applied in reverse they will always decrease the code size. Suppose Undo.Evol applies a reverse transformation repeatedly until no more transformations can be applied. Since each reverse transformation reduces the size of the program, Undo.Evol will always reach a state where no more transformation can be applied. If it did not, the process would reduce the size of the program to zero bytes, and at that point no more transformations can be applied anyway. Thus, Undo.Evol will always terminate.

It is a matter of further study whether Undo.Evol will always terminate on a single program. If it can be shown that Undo.Evol terminates on a single program, say Min.Evol, then to detect W32/Evol one may apply Undo.Evol on a binary and check for the signature of the Min.Evol.

CONCLUSIONS

Anti-virus scanner technology is constrained by the theoretical limits of program analysis techniques. A metamorphic virus is a manifestation of these limits. In fact, metamorphic viruses also depend on program analysis techniques, because in order to mutate, a metamorphic virus must analyse its own code. Thus a metamorphic virus cannot use tricks that will fool its own analyser.

This handicap of metamorphic viruses can potentially be exploited to develop AV scanners. However, to reverse the mutations in order to defeat a virus, the AV research community faces several key questions. For example, how does one extract the assumptions of a virus and the transformations it performs? Will reverting the transformations lead to a single result? Will the reverse transformations terminate in polynomial time? And how does one separate virus code from the code of the host?

The answers to some of these questions would be crucial in developing technology that takes advantage of the Achilles' heel of a virus.

FEATURE 2

THE THREE FACES OF VBA: PART 1

Dr Vesselin Bontchev
FRISK Software International

We live in the age of specialization. Sixteen years ago, when I started researching computer viruses, I quickly learned by heart what every known virus in existence did – either because I had analysed it myself, or because I had read everything published about it by somebody else.

When the number of known viruses reached a volume at which this became impossible, I began to specialize. I chose to specialize in macro viruses because it was an emerging threat, there were relatively few known macro viruses and virus-writing techniques and, in general, the field looked large, unexplored and promising.

I am a rather thorough and pedantic person; a perfectionist. While this makes it difficult to socialize with most people (who tend to find me dry, boring and, generally, insufferable), it means that when I decide to study a subject, I study it *really* thoroughly. As a result, I daresay that there are now few people in the world whose knowledge of macro viruses rivals mine. Of course, given that macro viruses are no longer the threat they once were, some might consider the usefulness of this knowledge arguable.

When one becomes highly knowledgeable in an extremely narrow field, one is prone to a kind of tunnel vision that tends to blind one to the notion that not everybody in the world perceives as obvious, elementary and self-evident the things one knows. Even if intellectually aware of this problem, one is still occasionally surprised when others show a lack of understanding of some obscure piece of knowledge from one's narrow field of expertise.

I had such a surprise recently, when reviewing the manuscript of a fellow anti-virus researcher. The colleague in question is a world-class expert – but he specializes in a *different* kind of virus. After correcting the particular problem in his work, I started researching it. To my great surprise, I found only one article relevant to the issue [1] – and that mentions the problem only fleetingly. The lack of a good description of the problem is disturbing and, as I have observed, it can be confusing for even the best among our profession. Therefore, I decided to write this article.

THE PROBLEM

Let us cut to the chase. The problem is caused by the fact that macros written in Visual Basic for Applications (VBA) have three completely different forms, any of which can be the one that is actually executed, depending on the

circumstances. In order to illustrate the point, there follows some background information.

OLE2, HOP!

The contemporary versions of the *Microsoft Office* applications (*Word*, *Excel*, etc. – but not *Access*) save their documents in files with the so-called 'OLE2 structure'. An OLE2 file is essentially a 'file system within a file' – it has a File Allocation Table (FAT), clusters of various sizes, a root directory (called 'root storage'), subdirectories (called 'storages') and files (called 'streams').

The information stored by the application in the document is contained in the various streams, while the storages are used to organize these streams – just as you would keep your information in files and organize these files in various directories (folders). Just like the contents of a logically continuous file can be split into clusters and these clusters themselves scattered all over the physical hard disk, the contents of a stream are also split into clusters and these clusters can be scattered all over the physical file.

So, in order to read the contents of an OLE2 stream in a logically continuous manner, you need to make sense of the underlying file structure and to locate the stream's clusters in the right order. This is a rather daunting task, given the complexity of the file structure and the lack of good documentation describing it. And a scanner must be able to do it, if it wants to scan for macro viruses properly.

Microsoft provides a set of APIs to perform the necessary operations in a transparent manner. Unfortunately, they are available only under *Windows*. Furthermore, they are rather buggy – a trivial corruption of the OLE2 file can easily cause an application that relies on *Microsoft*'s APIs (like the programs from the *Microsoft Office* suite) crash or hang when trying to process the offending file.

For the purposes of this article, we shall skip over the complexities of the OLE2 formats and consider the logical structure of these files. Fortunately, there is a wonderful little publicly available tool that can show this structure in an *Explorer*-like tree format. Its name is *eDoc* and it is made by a company called *eTree* [2].

Figure 1 shows the typical logical OLE2 structure of a document with a VBA macro in a module named 'Module1', as created by *Microsoft Word*. The subdirectory tree 'VBA' is present in every VBA-containing OLE2 file, while the other streams and storages are different, depending on the particular *Office* application that has created the document. For instance, *Excel* uses a storage named '_VBA_PROJECT_CUR' instead of 'Macros' and a stream named 'Workbook' instead of 'WordDocument'. For the rest of this article, we shall concentrate only on the streams in the VBA storage.

WELCOME TO VBA

It is worth noting that the problem we will discuss here exists only in VBA versions 5 and higher. VBA version 5 was introduced with *Office 97*. In the earlier versions of *Office*, only *Excel* supported VBA (version 3) and the problem did not exist.

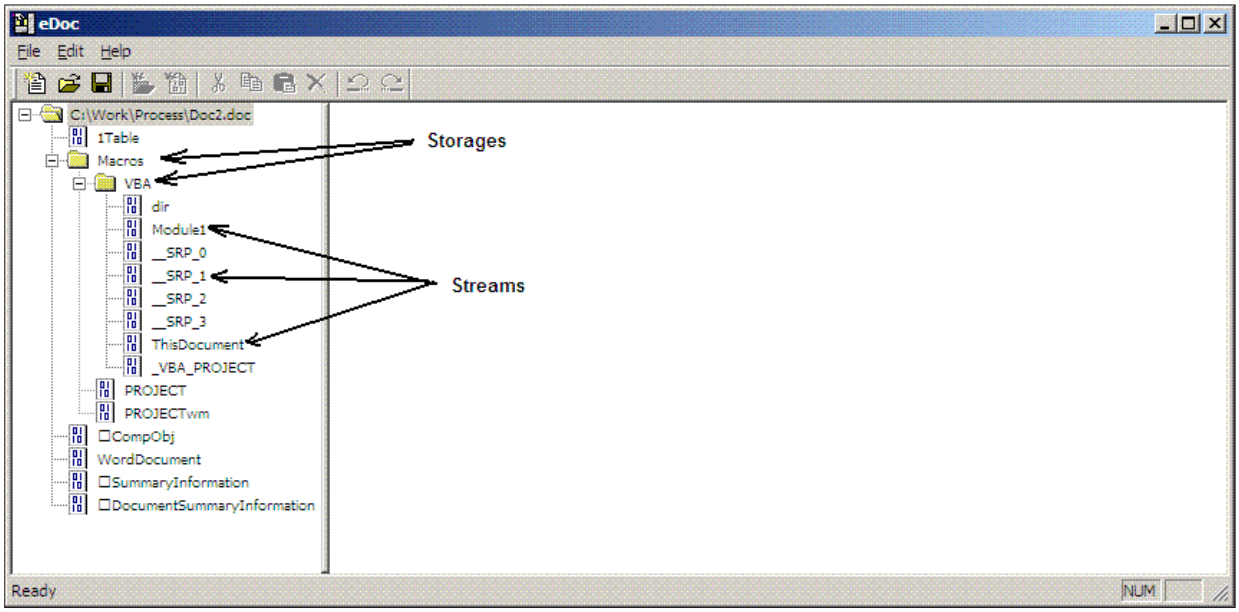


Figure 1. General OLE structure of a Word document with VBA macros.

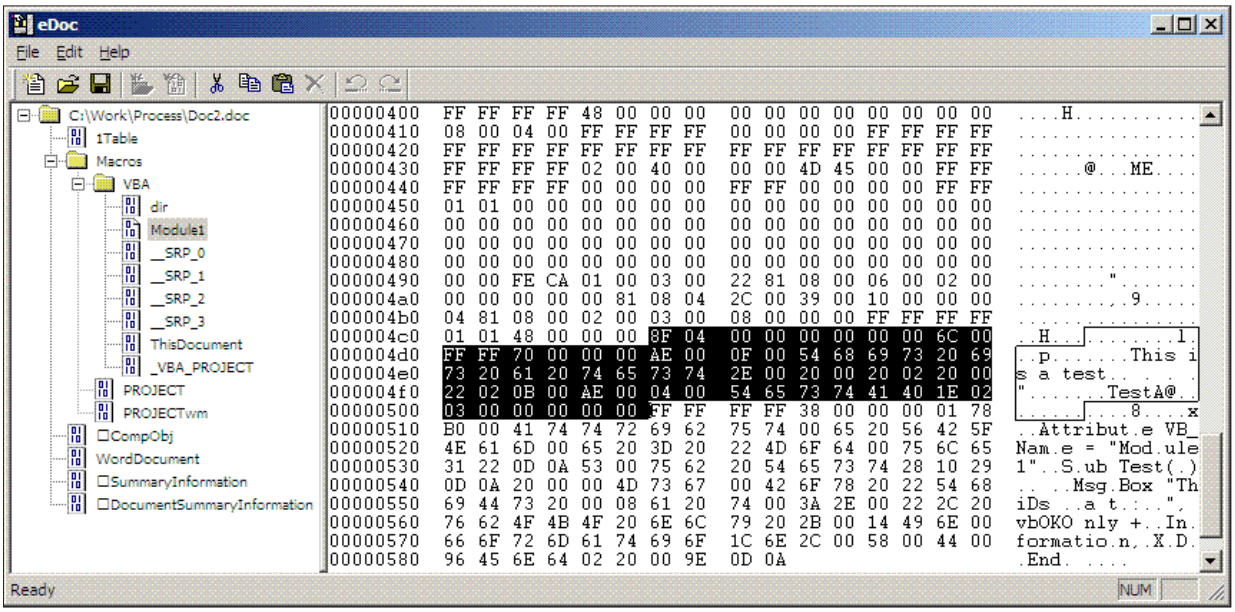


Figure 2. The p-code area in the module stream ('Module1') is highlighted.

Let us take another look at Figure 1. The streams in the VBA storage can be split into four different categories, depending on their contents.

The first category is the so-called *module streams*. There are two of them in our example: 'Module1' and 'ThisDocument'. A module stream is created for every module that exists in the VBA project – no matter whether it is a regular module (like 'Module1'), a class module (like 'ThisDocument'), or a user form (none are present in our example) – even if the module in question is empty (as 'ThisDocument' is, in our case).

The second category is the so-called *execode streams*. Their names always begin with '__SRP_', followed by a number. We shall discuss the purposes and the contents of these streams later.

The third category consists of the streams that contain *directory information*. There are two of them in our example: 'dir' and '_VBA_PROJECT'. These streams contain information about the names and properties of the modules of the VBA project. Some other information is stored there too – for instance, the '_VBA_PROJECT' stream contains all the identifiers (subroutine and variable names) used in all modules, as well as the paths of all add-ins used by the VBA project. An understanding of these streams is not important for the purposes of this article, so we shall ignore them.

The fourth category of streams are *informative streams* like 'PROJECT' and 'PROJECTwm'. They contain information such as whether the VBA project is password-protected (and, depending on the VBA version, the trivially encrypted password or a hash of it) and other such properties. They are similarly irrelevant for our purposes.

THE MODULE STREAMS

Part of the content of a module stream is shown in Figure 2. The module stream contains two of the VBA forms ('faces') about which this article is talking – the *p-code* and the *compressed source*.

The p-code area

The p-code area is highlighted in Figure 2. It is outside the scope of this article to explain exactly how the p-code area can be located in a module stream. Much of the necessary information has been received from *Microsoft* under a Non Disclosure Agreement (NDA) – suffice it to say that all legitimate anti-virus developers who have signed such an NDA with *Microsoft* can receive this information.

It is similarly outside the scope of this article to give a detailed explanation of how to read (and make sense of) the contents of the p-code area. For more information on this subject, the reader is referred to [3]. For the curious among you, in our example the VBA macro in Module1 contains:

```
Sub Test()
   MsgBox "This is a test.", vbOKOnly + vbInformation, "Test"
End Sub
```

The p-code highlighted in Figure 2 can be read as follows:

```
8F 04 00 00 00 00      ; Sub Test(). "0x00000000" is a
                       ; pointer to a structure,
```

```
                        ; describing the subroutine's
                        ; name and parameters.
00 00                   ; Filler. Each p-code line is
                        ; padded with garbage, so that
                        ; the number of bytes it consists
                        ; of is always a multiple of 8.
6C 00                   ; End Sub. The p-code lines
                        ; don't have to be physically in
                        ; order. Their logical order is
                        ; described elsewhere in the
                        ; module stream.
FF FF 70 00 00 00       ; Filler
AE 00 0F 00 5468697320697320612074657374E00 ; "This is a test."
                        ; Strings are padded with a zero,
                        ; if necessary, so that their
                        ; length is always a
                        ; multiple of 2.
20 00 20 02             ; Load the identifier "vbOKOnly".
                        ; "0x0220" is a reference to
                        ; the actual identifier, which
                        ; is stored elsewhere (in the
                        ; _VBA_PROJECT stream).
20 00 22 02             ; Load the identifier
                        ; "vbInformation".
0B 00                   ; +. The p-code interpreter is
                        ; a stack machine and works in
                        ; Reverse Polish Notation - i.e.,
                        ; the operands are followed by
                        ; the operator.
AE 00 04 00 5465737441 ; "Test"
40 1E 02 03             ; Subroutine call to MsgBox.
                        ; Again "0x0302" is a reference
                        ; to the table of identifiers in
                        ; the _VBA_PROJECT stream.
00 00 00 00 00          ; Filler
```

The source code area

Almost immediately after the p-code area, each module stream contains a source code area. The latter, unsurprisingly, contains the source code of the VBA program that resides in the module. The source code area is compressed using Lempel-Zev compression, which is why it is not readable immediately – but it is, nevertheless, easily recognizable by visual inspection.

THE EXECODE STREAMS

As mentioned earlier, the execodes reside in the __SRP_ streams. They are the third 'face' (i.e. executable form) of a VBA program – besides the p-code and the source code. (We shall explain why the source code is 'executable' in the next section.)

Unfortunately, very little is known about the exact purpose and format of the execodes. *Microsoft* has not been forthcoming on this subject either, simply because it does not *have* such information, surprising as it seems. As a rule, *Microsoft* does not document internally the formats it develops. Instead, it documents the API calls necessary to access them. While this might make sense in a *Windows* environment, it is rather useless to those who wish to develop and support multi-platform anti-virus programs.

So, the little information we have about the execode streams has come from reverse-engineering. Unfortunately, since this form is almost never used, and we've always had enough higher-priority *Microsoft* things to reverse-engineer, we do not really know much on the subject of execode formats.

What we do know is that the module named '__SRP_0' is a special one. It seems to contain, among other things, all

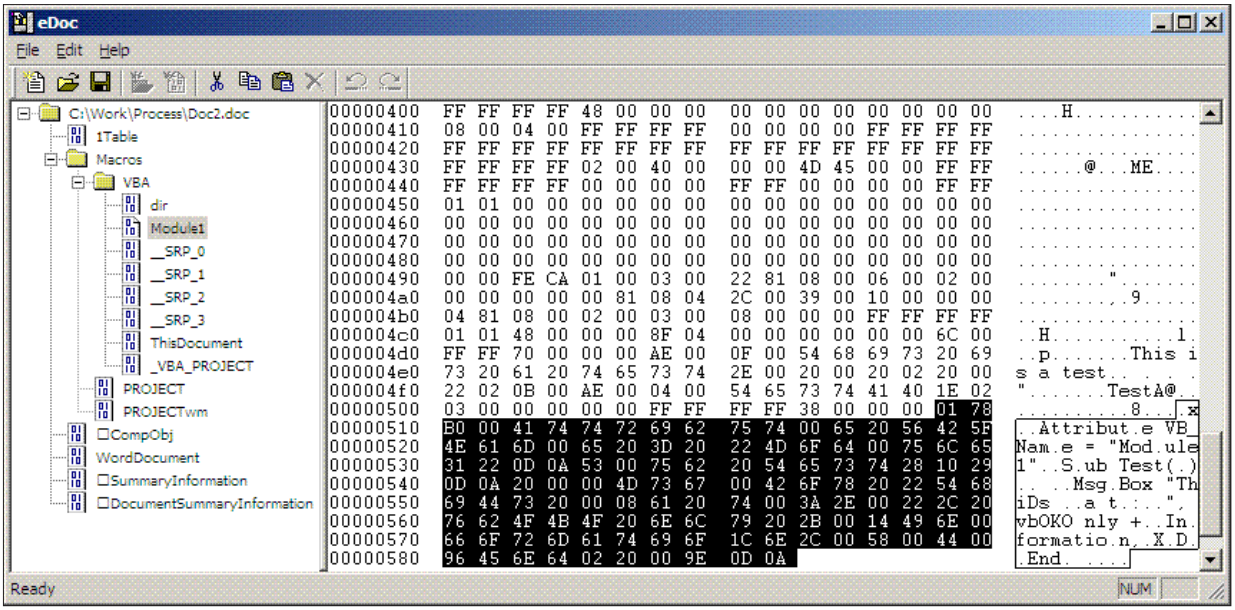


Figure 3. The source code area in the module stream ('Module1' in our example) is highlighted.

literal strings used in all VBA modules of the VBA project. Part of the content of this stream is shown in Figure 4.

We also know that every module stream has its separate corresponding __SRP_ execode stream – although presently we do not know how to determine which execode stream corresponds to which module stream, and we do not know the purpose of the additional execode streams.

The content of the execode stream corresponding to the 'Module1' module stream in our example is shown in Figure 5. If you cannot make sense of it, don't worry – you are not alone.

We do not reliably know even how to locate the beginning of the execodes in the execode stream – although we have developed a set of heuristics that seems to work most of the time. What we *do* know is that each subroutine or function of a module has its own area in the execode stream. These areas are adjacent and in the order in which the subroutines (or functions) appear in the module. Somewhere before them in the execode stream there is a two-byte word, containing the number of such areas – it is at offset 0x003C in the '__SRP_2' stream in our example and it contains 0x0001, for the 'Module1' stream contains only a single subroutine ('Test'). The corresponding single execodes area begins at offset 0x005A in the '__SRP_2' stream in our case.

Each such area seems to begin with two four-byte fields [4]. The first field seems to contain the full size of the area (0x00000080 in our case), while the second seems to contain the size of the non-variable part of it (0x00000040 in our case). This non-variable part is always located at the beginning of the area that corresponds to a particular subroutine or function. The rest of the area is the variable part – so called because its contents seem to be variable, although we don't have the slightest clue how to interpret it. In fact, we are not even certain that the contents of the so-called 'non-variable area' are always constant.

That, I am afraid, exhausts all that we currently know about the format of the execodes streams.

PIECING IT ALL TOGETHER

Most of the time, it is the p-code area that is executed when a VBA program is run. The execodes are not even always present – the streams containing them can safely be deleted and the VBA program will still run without problems. The source code area can be modified with a hex editor, so that its contents do not match what the p-code says (for instance, in our example we could patch the 'T' to 'P' at offset 0x054E in the stream named 'Module1') – and, if the patched macro is run, it will be the contents of the p-code that are executed. Worse, if you open the patched document with *Word* and inspect its VBA module with the VBA Editor, you will not see any effects from the patching – the VBA Editor cheerfully ignores the compressed source and decompiles the p-code into what it displays.

Seemingly, neither the source code, nor the execodes are of any importance and one should concentrate on the p-code alone when scanning for VBA macro viruses. But, when *Microsoft* provided us with the limited information it did on the subject of VBA, we were told two things explicitly.

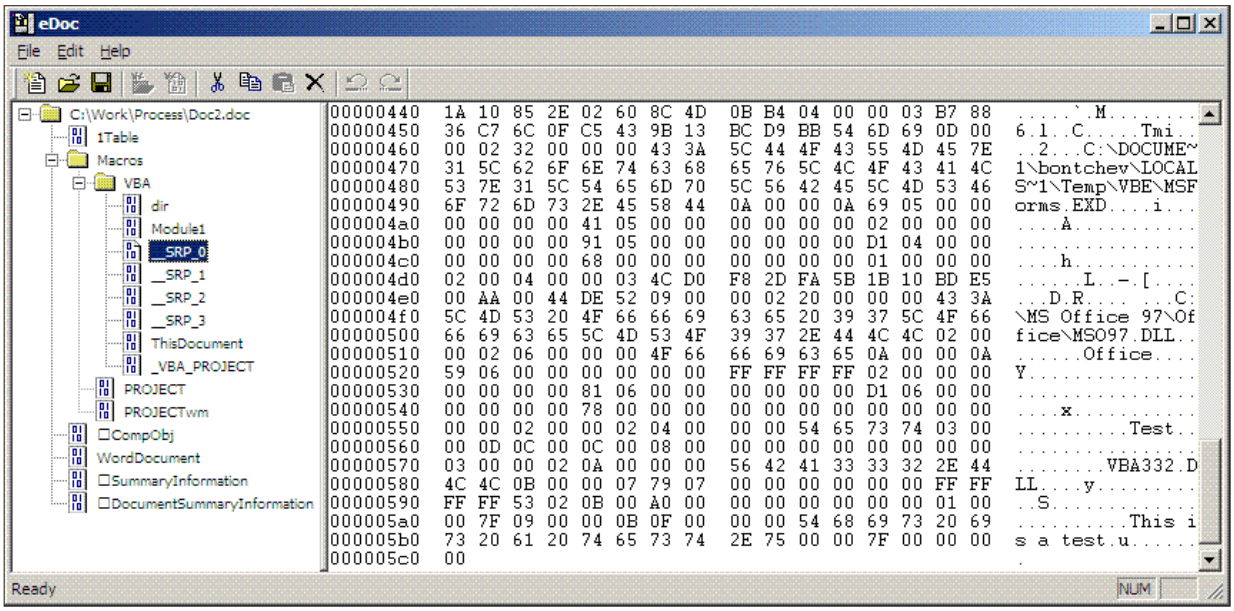


Figure 4. Part of the content of the '__SRP_0' stream in our example.

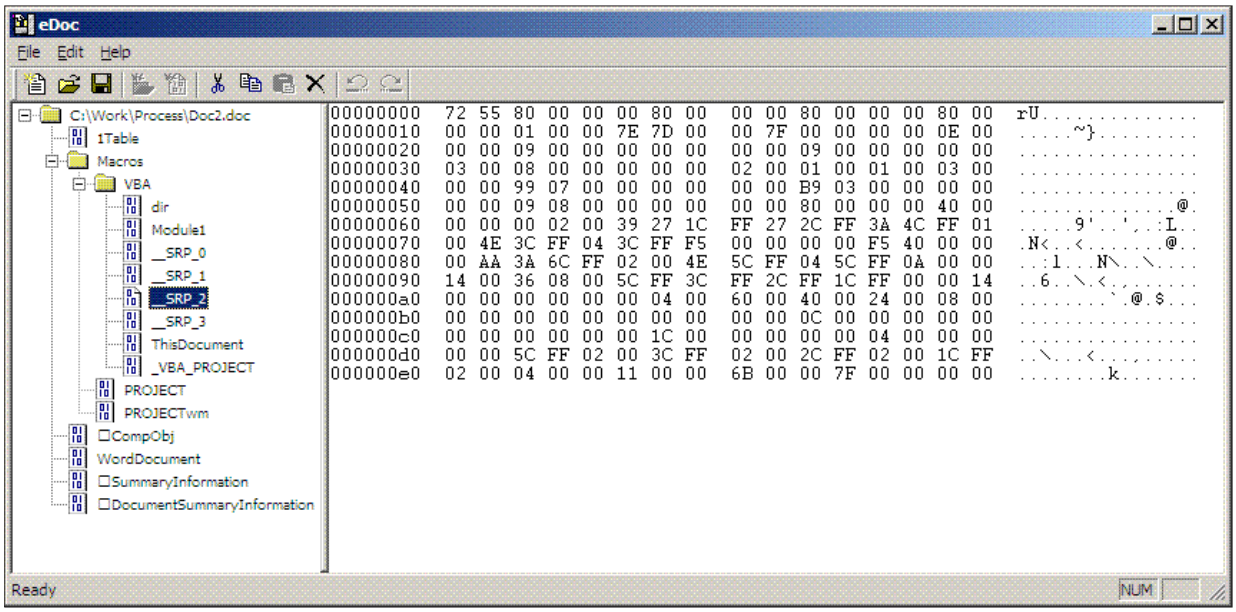


Figure 5. A regular execodes stream.

First, we were told to decompress the compressed source code area and to scan that. Second, we were told that, when disinfecting a VBA macro virus, we should delete all __SRP_* streams that are present.

True to form, there was no explanation of the reasoning behind these two recommendations. We cheerfully ignored *Microsoft*'s suggestions – although not all anti-virus companies handled them in the same way. Very few products implemented both of the recommendations, some implemented only the first, some only the second, and some (again, very few) ignored both of them.

I was responsible for implementing the macro scanning engine in our product and I took the decision to ignore the first suggestion but to implement the second. My reasoning was that, since the p-code was clearly what was executed, it made most sense to scan it instead of anything else. I decided to remove the __SRP_* streams when disinfecting because I could, and I didn't see any harm in doing so.

Much later, we discovered that there had been excellent reasons behind *Microsoft*'s suggestions. As it turns out, the algorithm used by VBA to interpret the p-code, the source code and the execodes is as follows:

```
IF (the execodes exist) AND
   (the execodes are created by exactly the same
   version of VBA as the one opening the
   VBA project)
THEN
   use the execodes
ELSE IF (the p-code is created by the same version of
        VBA as the one opening the VBA project)
THEN
   use the p-code
ELSE
   use the source code
END IF
```

By 'exactly the same version', I mean just that. For instance, even the same version numbers of *Windows Office* and *Mac Office* do not have exactly the same version of VBA, because one of them uses little-endian byte format and the other uses a big-endian one.

By 'the same version of VBA', I mean just the same version number. For instance, both *Office 97 for Windows* and *Office 98 for Macintosh* use the same version of VBA (5.0).

The execodes seem to be some kind of memory dump of the structures that the p-code interpreter creates in memory when loading the p-code instructions. Since we don't know exactly what the execodes are, you will have to use your imagination as to what exactly 'use the execodes' means.

By 'use the source code', I mean decompress the compressed source code area in memory, compile it into p-code, and start interpreting that p-code.

Why such a convoluted algorithm? Presumably, the purpose of the execodes is to speed up the loading and execution of the p-code. They do not *have* to be present and, even if they are present, they will not be used if the document containing them is opened with a different version of *Office*.

But if the execodes are present and the document is opened by exactly the same version of *Office* as the one used to create it, the execodes will be used and (presumably) will load and run faster than interpreting the p-code directly. The execodes will be present only if the document containing

the VBA project is saved *after* the VBA code in it has been executed at least once. Depending on how, exactly, a macro virus works (i.e., whether or not any of the viral code in the newly infected document is run before that document is saved), some will always contain execodes in their VBA project and some never will.

The reason for the presence (and usage) of the source code is a completely different one. It provides compatibility between the different VBA versions. For instance, VBA version 6.0 (used in *Office 2000* and above) is slightly different (it has some additional p-code instructions) from VBA version 5.0 (used in *Office 97*). If it weren't for the source code, *Office 2000* would not be able to execute directly any macros in the *Office 97* documents it opens – because the opcodes of the p-code instructions would be different and incompatible. Instead, it would need a special upconversion program. Furthermore, as new versions of VBA are introduced, the number of such converters would have to increase exponentially in order to cover all possible up- and downconversions between the various existing VBA versions.

Yet, with the algorithm described above, the problem of multiple converters is solved in a very elegant way: none are needed. If a document containing a VBA macro is opened with a version of *Office* that contains a version of VBA different from the one that has been used to create the macro in the document (whether it is a higher or a lower version number), the source code area in the module stream(s) will simply be decompressed, recompiled and executed. There is no need for p-code upconversion or downconversion; the existing p-code is simply ignored and recreated from the source.

It is a very elegant solution to a rather difficult problem. Unfortunately, whoever came up with that solution at *Microsoft* clearly wasn't thinking along the same lines as the designers of the anti-virus programs. As a result, the solution created a completely different set of problems.

[Vesselin explains all about those problems in the continuation of this article in the February 2005 issue of Virus Bulletin - Ed.]

REFERENCES

[1] Igor Muttik, 'A Portrait of Jini', *Virus Bulletin,* October 2000, p.7.

[2] http://www.etree.com/tech/freestuff/edoc/eDoc.exe.

[3] Vesselin Bontchev, 'Solving the VBA upconversion problem', *Proc. 10th Int. Virus Bull. Conf.*, pp.273–299.

[4] Costin Raiu, personal communication.

CALL FOR PAPERS

VB2005 DUBLIN

Virus Bulletin is seeking submissions from those wishing to present at VB2005, the Fifteenth Virus Bulletin International Conference, which will take place 5–7 October 2005 at The Burlington, Dublin, Ireland.

The conference will include two full days of 40-minute presentations running in concurrent streams: Technical Anti-Virus, Corporate Anti-Virus and Spam (both technical and corporate).

TOPICS

Submissions are invited on all subjects relevant to the anti-virus and anti-spam arenas.

A list of suggested topics elicited from attendees at VB2004 can be found at http://www.virusbtn.com/conference/vb2005. However, please note that this list is *not* exhaustive, and papers on these and any other AV and spam-related subjects will be considered.

VB welcomes the submission of papers that will provide delegates with ideas, advice and/or practical techniques, and encourages presentations that include practical demonstrations of techniques or new technologies.

HOW TO SUBMIT A PAPER

Abstracts of approximately 200 words must reach the Editor of *Virus Bulletin* no later than **Thursday 10 March 2005**. Submissions received after this date will not be considered. Abstracts should be sent as RTF or plain text files to editor@virusbtn.com. Please include full contact details with each submission.

Following the close of the call for papers all submissions will be anonymised before being reviewed by a selection committee; authors will be notified of the status of their paper by email.

Authors are advised in advance that, should their paper be selected for the conference programme, the deadline for submission of the completed papers will be Monday 6 June 2005 and that full papers should not exceed 8,000 words.

Further details of the paper submission and selection process are available at http://www.virusbtn.com/conference/vb2005/.

CONFERENCE REPORT

AVAR 2004: EUTAXY OR CHAOS?

Righard Zwienenberg
Norman, The Netherlands

The AVAR 2004 conference, entitled 'Eutaxy [*meaning good or established order - Ed*] or chaos?', was held on 25 and 26 November 2004 at the Sheraton Grande hotel in Tokyo, Japan.

AVAR 2004 chair Mr Shigeru Ishii opened the conference by welcoming some 200 delegates to Tokyo and reviewing the journey of viruses over the last 20 years from desktop, to LAN, to WAN. Mr Ishii was followed by Mr Seiji Murakami, chairman of AVAR, who confessed that he was glad to see the conference return to Japan, since it meant that he could present in Japanese, instead of struggling with English.

The first morning of the conference was taken up with presentations and panel discussions on the state of computer security in Japan, Korea and China. Of particular interest was a presentation by Zhang Jian, of CNCERT (the Chinese National Computer Network Emergency Response Technical Team), about the legal aspects of writing malicious code.

The afternoon's session started with Suguru Yamaguchi, advisor on information security for the IT Security Office of Japan's Cabinet Secretariat, who presented both a government and industry model for the Critical Infrastructure Protection (CIP) in Japan. Japan has just stepped into the challenge of combining CIP and Information and Communication Technology (ICT) and Suguru's presentation provided an excellent overview.

David Perry's presentation was, as usual, entertaining. David led the audience through some past and present threats and their appropriate responses. Those who have seen David present before will not be surprised to learn that he spoke too quickly for the translators providing simultaneous translation to keep up. Unfortunately, it proved to be a no-win situation for the translators (and for the non-English speakers for whose benefit they were translating) because, having recognised that he was speaking too fast, and slowed down, David proceeded to use a large number of untranslatable words!

A banquet held on the first evening of the conference saw some rather interesting and unexpected entertainment in the form of Seiji Murakami, who cast off his business attire, picked up the drumsticks and performed with a guitar band on stage. Those lucky enough to witness Seiji's closing drum solo will never look at him again in quite the same light – from now on he will be known as Seiji 'Mr Hit It!' Murakami.

Seiji 'Mr Hit It!' Murakami.

Jeannette Jarvis started the second day off by providing an insight into how *The Boeing Company* manages a global corporate protection infrastructure.

To follow, Randy Abrams and Andreas Marx joined forces to demonstrate how they script AV file signature updates and how they perform testing for their specific purposes.

Next, Eugene Kaspersky charted the evolution of the computer underground, in an in-depth retrospective of virus-writing – from hooliganism to organised crime.

Mikko Hypponen followed up by showing that the virus-writing scene has moved on from the days when youngsters were writing viruses for fun or ego-boosting purposes. Mikko described what we know of today's virus 'professionals', who use viruses, spam and spyware for the express purpose of gaining money and data. The data collected is subsequently sold on – a practice that has been in common use among spammers for some time.

Later in the afternoon, Jimmy Kuo related some of his experiences in his role as a Police Reserve Specialist for the Hillsboro Oregon Police Department in his presentation, 'Chasing the bad guys'. Jimmy described how the police, with the aid of the security industry, attempt to hunt down those who write and distribute malware. Sometimes they are successful, as in the case of W97/Melissa author David L. Smith, and sometimes they are not – as was the case with the author(s) of the Sobig.F virus. Although the police found the system from which the virus was launched, the elderly couple who owned the system had no idea that it had been infected and commandeered to send out the new virus.

In the final panel session of the conference, representatives of some of the conference sponsors were asked to give delegates some idea of their companies' future strategies. The common factor here was that each of the panel members mentioned phishing and mobile devices as a major focus of their company's future strategy.

Finally, the 7th AVAR conference was drawn to a close with the traditional closing ceremony, and the venue for the next AVAR conference was announced: AVAR 2005 will be held in the city of Tianjin, in the People's Republic of China (date TBA).

END NOTES & NEWS

Computer & Internet Crime 2005 will take place 24–25 January 2005 in London, UK. The conference is dedicated solely to the problem of cyber crime and the associated threat to business, government, public services and individuals. For more details and to register, see http://www.cic-exhibition.com/.

SecureLondon 2005 takes place 10 February 2005 in London, UK. Sessions include: Identity Management (Fred Piper, Royal Holloway), Crime on the Internet (Steve Santorelli, Scotland Yard) and the Future of Internet Security (Phil Cracknell, netSurity). For more information, and to register, visit https://www.isc2.org/events/.

The 14th annual RSA Conference will be held 14–19 February 2005 at the Moscone Center in San Francisco, CA, USA. For more information, including online registration and the conference agenda, see http://www.rsaconference.com/.

Websec 2005: i-Security World Conference takes place 15–17 March 2005 in London, UK. Optional workshops will be held on 14 and 18 March. The conference features three tracks: security policy, risk & governance; IT infrastructure security; and enterprise application security. For details see http://www.mistieurope.com/.

The E-crime and Computer Evidence conference ECCE 2005 takes place at the Columbus Hotel in Monaco from 29–30 March 2005. ECCE 2005 will consider aspects of digital evidence in all types of criminal activity, including timelines, methods of evidence deposition, use of computers for court presentation, system vulnerabilities, crime prevention etc. For more details see http://www.ecce-conference.com/.

Black Hat Europe takes place in Amsterdam, The Netherlands, from 29 March to 1 April 2005. Black Hat Europe Training runs from 29 to 30 March, with the Black Hat Europe Briefings following, from 31 March until 1 April.

Black Hat Asia takes place 5–8 April 2005 in Singapore. In this case the Briefings take place 5–6 April, with the training on 7–8 April. A call for papers for the Black Hat Briefings (both Europe and Asia) closes on 15 January 2005. For details and registration see http://www.blackhat.com/.

The first Information Security Practice and Experience Conference (ISPEC 2005) will be held 11–14 April 2005 in Singapore. ISPEC is intended to bring together researchers and practitioners to provide a confluence of new information security technologies, their applications and their integration with IT systems in various vertical sectors. For more information see http://ispec2005.i2r.a-star.edu.sg/.

Infosecurity Europe 2005 takes place 26–28 April 2005 in London, UK. Now in its tenth year, the exhibition will have over 250 exhibitors and its organisers anticipate over 10,000 visitors. See http://www.infosec.co.uk/.

The 14th EICAR conference will take place from 30 April to 3 May 2005 in Saint Julians, Malta. Authors are invited to submit papers for the conference. The deadlines for submissions are as follows: academic papers 14 January 2005; poster presentations 18 February 2005. For full details see http://conference.eicar.org/.

The sixth National Information Security Conference (NISC 6) will be held 18–20 May 2005 at the St Andrews Bay Golf Resort and Spa, Scotland. For details of the agenda (which includes a complimentary round of golf at the close of the conference) or to register online, see http://www.nisc.org.uk/.

The third International Workshop on Security in Information Systems, WOSIS-2005, takes place 24–25 May 2005 in Miami, USA. For full details see http://www.iceis.org/.

NetSec 2005 will be held 13–15 June 2005 in Scottsdale AZ, USA. The program covers a broad array of topics, including awareness, privacy, policies, wireless security, VPNs, remote access, Internet security and more. See http://www.gocsi.com/events/netsec.jhtml.

The 15th Virus Bulletin International Conference, VB2005, will take place 5–7 October 2005 in Dublin, Ireland. For conference registration, sponsorship and exhibition information and details of how to submit a paper see http://www.virusbtn.com/.

SUBSCRIPTION RATES

Subscription price for 1 year (12 issues) including first-class/airmail delivery: £195 (US$310)

Editorial enquiries, subscription enquiries, orders and payments:

Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

Tel: +44 (0)1235 555139 Fax: +44 (0)1235 531889
Email: editorial@virusbtn.com www.virusbtn.com

vb Spam supplement

CONTENTS

S1 **NEWS & EVENTS**

S1 **FEATURE**
A spam program's techniques

S4 **BOOK REVIEW**
Spammers revealed

NEWS & EVENTS

LYCOS ENDS DDoS STUNT

Lycos ended a controversial DDoS campaign against spammers last month, claiming that it had accomplished its objectives. The campaign, in which *Lycos* encouraged users to download a screensaver program which sent HTTP requests to spammers' servers, had been criticised for its vigilante nature. A *Lycos* spokesman explained: 'The aim of the campaign was to ignite a debate about anti-spam measures. We feel that we have achieved this through our activity and will continue that debate with others in the email industry.'

Unfortunately, the company's campaign may have started problems of a different kind – reports have been received of a fake *Lycos* screensaver circulating on the Internet. The file, which arrives as an attachment to an email with the subject line 'Be the first to fight spam with Lycos screen saver', is actually a RAR SFX archive with embedded keylogger Trojan components. More details of the story can be found at http://www.virusbtn.com/news/spam_news/.

EVENTS

The 2005 Spam Conference will be held in Cambridge, MA, USA on 21 January 2005. See http://spamconference.org/.

The ISIPP's National 'Spam and the Law' Conference will be held on 28 January 2005 in San Francisco, CA, USA. See http://www.isipp.com/.

The IQPC will hold a two-day conference on managing and securing corporate email from 1–2 February 2005 in Las Vegas, NV, USA. See http://www.iqpc.com/.

FEATURE

A SPAM PROGRAM'S TECHNIQUES

John Graham-Cumming
The POPFile Project, USA

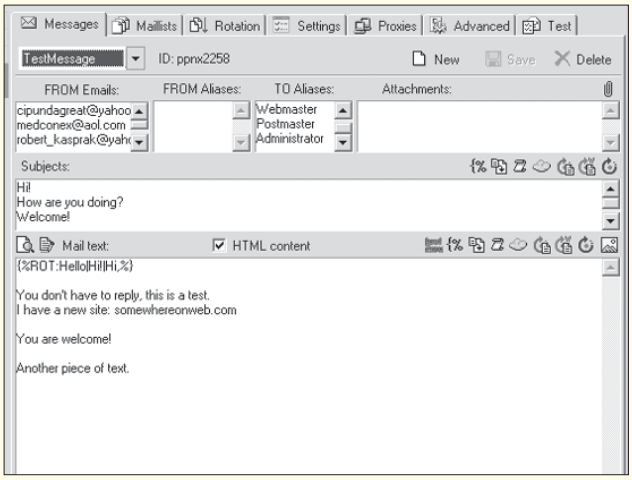
Send-Safe is a bulk email program and service. The *Send-Safe* application is freely available on the Internet from http://www.send-safe.com/. *Send-Safe* allows a user to create bulk email campaigns, manage mailing lists and send messages either directly through an SMTP server or through a list of open, anonymous proxies; proxies can be provided either by the user, or provided automatically by the *Send-Safe* service.

To send bulk email with *Send-Safe* the user must open an account with send-safe.com and purchase credits. Credits can be bought in blocks – prices start at $50 for 400,000 messages, and range up to $3,000 for 300 million messages.

Send-Safe has many capabilities, including the ability to select random From, To and Subject lines; insert random chains of Received headers; create a random From address; change the DNS name of the machine sending the mail, etc. This article looks at *Send-Safe*'s content obfuscation techniques.

UI AND MACRO LANGUAGE

When *Send-Safe* is run it presents a user interface (UI) in which the user composes an email message to form part of a bulk email campaign. Logging in with the demonstration account generates a sample message as shown here.



The sample message has a list of possible From, To and Subject lines and the following body:

```
{%ROT:Hello|Hi!|Hi,%}
You don't have to reply, this is a test.
I have a new site: somewhereonweb.com
You are welcome!
Another piece of text
```

The first line consists of {%ROT:Hello|Hi!|Hi,%}, which means that each time a message is sent, *Send-Safe* will pick 'Hello', 'Hi!' or 'Hi', for the start of the message. This is an example of the macro language that is built into *Send-Safe* for automatic message customization.

A variant of {%ROT%} is {%ROTF%}, which selects from a list of possible text in a file: each line of the file is a possible piece of text to insert and *Send-Safe* will select a line randomly. {%WROTF%} selects a single word from a file to insert into a sentence. These ROT macros provide a means of customizing messages so that each message is unique.

The {%RND%} macro is used to add random characters to an email. For example, specifying {%RND: ##^^**%} would insert two random numbers (#), two random upper case letters (^) and two lower case letters (*).

More powerfully, {%RND%} has a syntax for creating random sequences of characters with random lengths: {%RND:<d12><l10><L5><m5>%} means 'add up to 12 digits, followed by up to 10 lower case letters, followed by up to five upper case letters, followed by up to five digits, upper case or lower case letters.'

{%RND%} was clearly intended for adding text to break anti-spam measures that rely on checksumming or fingerprinting messages to detect bulk runs of the same message.

To hide URLs from filters, the {%URL%} tag will encode the URL. For example, suppose that I wanted to email the URL http://www.jgc.org/. I would write http://{%URL:www.jgc.org/%} and *Send-Safe* would insert http://%77%77%77%2E%6A%67%63%2E%6F%72%67%/ in the message. This form works in any email program and will take you to the website, but it will fool some spam-filtering software.

Furthermore, *Send-Safe* allows these macros to be nested inside one another, so it is possible, for example, to ask *Send-Safe* to obscure a URL with {%URL%} and within it generate a random part of the URL using {%RND%} or pick a random URL from a list with {%ROT%}. It is common for the URLs in spam messages to have a completely bogus part in the DNS name, and for the DNS server for the spammer's website to accept any name in that part. If it were the case that http://www.jgc.org/ would accept any sequence of characters in place of www (as is common among spammers' websites), the following macro could be used: http://{%URL:{%RND:<l10>%}.jgc.org/}. This would generate a new sequence of up to 10 letters followed by .jgc.org and then encode the entire URL.

RANDOMIZING HTML CONTENT

Another technique that *Send-Safe* uses to obfuscate the content of a message is the insertion of random HTML tags in a message. When this feature is enabled, and the message being sent is in HTML format, *Send-Safe* will insert random pairs of <FONT>, <SPAN>, <EM>, <STRONG>, <I> and other harmless HTML tags in the message. Each tag is paired with a matching end tag.

It is possible for *Send-Safe* to make a mistake and insert a tag inside a tag. While testing the software it generated a line of text containing <FONT<FONT>>, which resulted in the display of a '>' in the final email. This appears to be a bug.

HIDING BAD WORDS

No spammer wants a word like 'Viagra' to appear in simple ASCII text in their email, since many spam filters look for keywords such as this to filter out spam. *Send-Safe* has a specific feature to hide 'bad words'.

The user creates a file called 'badwords.txt', places it in C:\Program Files\Send-Safe and enables the 'Hide words from badwords.txt' function. If any of the words listed in this file are present in the message, the word(s) in question will be split at a random point and a piece of HTML inserted. If, for example, the word 'welcome' was listed in badwords.txt, each time it was found in the message a piece of HTML in the form:

```
<span style='display:none'>ghuwefeq</span>
```

would be inserted to break up the word (the letters in the middle are chosen at random). This will prevent some spam filters from seeing the sensitive word. The HTML does nothing when the email message is displayed and the word 'welcome' will appear in one piece.

MESSAGE PART CONTROL

Send-Safe also provides macros that control the application of specific techniques to regions of a message. It is possible, using these macros, to base 64 encode or quoted-printable encode part of a message, or enable hiding of bad words and randomizing HTML content.

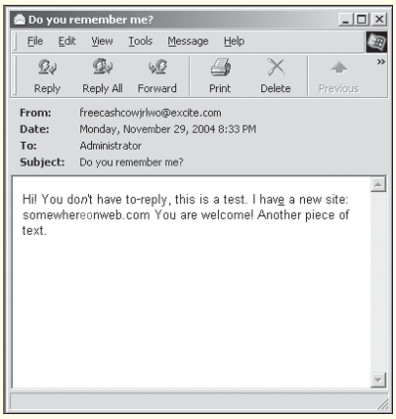
Base 64 encoding is used typically to encode binary content (e.g. a *Word* file attachment) in ASCII form so that it can be handled by mail servers. Quoted-printable encoding is used by *Microsoft* email programs like *Outlook Express* to

encode the content of an email message so that the message is unlikely to be modified by an email server.

{%BEGIN_BASE64%}/{%END_BASE64%} enables base 64 encoding of the enclosed region, and the macro {%BEGIN_QUOTEDPRINTABLE%}/{%END_QUOTEDPRINTABLE%} does the same for quoted-printable encoding.

{%BEGIN_RANDHTML%}/{%END_RANDHTML%} enables the insertion of random HTML tags in the enclosed region, just like the randomizing HTML feature. {%BEGIN_HIDEBADWORDS%}/{%END_HIDEBADWORDS%} enables the 'Hide words from badwords.txt' function between the macros.

PUTTING IT ALL TOGETHER



Send-Safe includes a preview function that allows the user to see what their message will look like when viewed with *Outlook Express*. Here's an example of previewing the default message with the following options: the 'Hide words from badwords.txt' function is enabled and set to obscure the word 'welcome', the 'Randomize body HTML' function is enabled, the message is in HTML form and a single {%ROT%} macro is included.

You can see that *Send-Safe* has picked a random From address (and further obscured it by inserting the random letters 'jrlwo'), and picked a random To and Subject. The body of the HTML shows some italics and underlining, which indicates that random HTML tags have been inserted.

The message source is as follows:

```
From: <freecashcowjrlwo@excite.com>
To: "Administrator" <freecashcowjrlwo@excite.com>
Subject: Do you remember me?
Date: Tue, 30 Nov 2004 10:33:07 +0600
MIME-Version: 1.0
Content-Type: multipart/alternative;
boundary="——=_NextPart_A11_623_DEA9B.72FB7"
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: Microsoft Outlook Express 6.00.2800.1106
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2800.1106

This is a multi-part message in MIME format.
---=_NextPart_A11_623_DEA9B.72FB7
Content-Type: text/plain;
   charset="us-ascii"
Content-Transfer-Encoding: quoted-printable

---=_NextPart_A11_623_DEA9B.72FB7
Content-Type: text/html;
   charset="us-ascii"
Content-Transfer-Encoding: quoted-printable

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional/
/EN">
<HTML><HEAD>
<META http-equiv=3DContent-Type content=3D"text/html;
charset=3Dus-asci=i">
<META content=3D"MSHTML 6.00.2800.1106" name=3DGENERATOR>
<STYLE></STYLE>
</HEAD><FONT face=3DArial><FONT size=3D2>
<BODY>
<DIV>
Hi!

You do<EM>n</EM>'t have to<STRIKE> </STRIKE>reply,
thi<SPAN>s is a</SPA=
N> test.
I hav<U>e</U> a<SPAN> new </SPAN>site: somewher<FONT
color=3D#2d33d3>eo=
</FONT>nweb.com

Yo<SPAN>u are </SPAN>we<SPAN
STYLE=3D"display:none">fldv</SPAN>lcome!

Another piece of text.

</DIV></BODY></HTML></FONT></FONT>

---=_NextPart_A11_623_DEA9B.72FB7-
```

Send-Safe has inserted the following pairs of tags: <EM>/</EM> around the 'n' in 'don't', <STRIKE>/</STRIKE> around a space, <SPAN>/</SPAN> around 's is a' in 'this is a test', <U>/</U> around the 'e' in 'have', <SPAN>/</SPAN> around 'new', a random font color around the letters 'eo' in 'somewhereonweb.com' and a <SPAN>/</SPAN> around 'u are' in 'you are'. All those tags were added by the Randomize body HTML function.

The 'Hide words from badwords.txt' function has split the word 'welcome' by inserting <SPAN STYLE="display:none"> fldv</SPAN> between 'we' and 'lcome'.

CONCLUSION

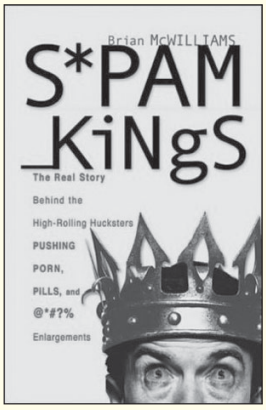
Send-Safe has quite extensive functionality for obfuscating the content of an email message, and it even has *SpamAssassin* built in so that the user can test whether the content of their message is a red-flag for *SpamAssassin*.

Older spam filters, and especially those that depend on recognizing keywords, will be fooled by the trickery available and spam filters that rely on checksumming messages will have a hard time if the {%RND%} and {%ROT%} macros are used carefully. However, up to date spam filters should easily be able to deal with these tricks. It is easy to spot and remove the useless HTML tags, and the word 'welcome' can easily be reconstructed by spotting the <SPAN STYLE="display:none">fldv</SPAN> that was inserted and ignoring the letters 'fldv'.

BOOK REVIEW

SPAMMERS REVEALED

Matt Ham



Title: Spam Kings,
Author: Brian McWilliams
Publisher: O'Reilly
ISBN: 0-596-00732-9

Spam Kings is described as 'The real story behind the high-rolling hucksters' – a grand claim indeed.

A large part of the book is devoted to the mini-biographies of two individuals in the spam/anti-spam arena, with a series of asides as notable characters are encountered by or contribute to the activities of the central two. The research that has been undertaken for the book is solid, with references available in many cases, and it is clear that McWilliams sought out and interviewed the main characters he describes.

The two central characters are Davis Hawke, spammer, and Susan Gunn, aka Shiksaa, 'anti-spammer'.

Davis Hawke must be every writer's dream character – not only does he have an extensive career as a spammer, but his personal views range from neo-Nazi to survivalist libertarian. At the start of the book Hawke comes across as an entirely unsympathetic character, yet by the end of the book – in my opinion at least – he seems more intriguing than villainous.

Having been on the receiving end of numerous of Hawke's penis-related spams, my change in attitude is certainly not due to a diminished sense of his ability to irritate vast swathes of Internet users. Hawke does, however, seem to have distanced himself from the net politics which dominate the tales of supporting characters in *Spam Kings* – among whom irritating personal traits seem to be the norm. Other spammers described in the book seem to be a mix of individuals ranging from the clinically insane to the career criminal, with the majority simply being those who see an opportunity for easy money.

In contrast, the other main character, Susan Gunn, seems a far more sympathetic character at the start of the book – reacting to being spammed by seeking out information about the perpetrators. However, a sizeable proportion of her activity as described in *Spam Kings* appears to be name-calling. Logs of personal exchanges between both spammers and anti-spammers play a prominent part in the book and, as is common in such exchanges on the Internet in general, these descend into pure childishness on numerous occasions. As a result, few of the participants are shown in an entirely positive light.

Direct action undertaken by anti-spammers is often dominated by what can be considered acts of a vigilante nature. DDoS attacks, the defacing of spammers' sites and publishing of personal information all seem to be considered fair game by the anti-spammers described in the book. Personal opinions may vary, but I believe that such behaviour simply lowers the anti-spammers to the same level as the spammers themselves. I had intended to be more positive about the ethos of anti-spamming activists, but the recent DDoS stunt performed by *Lycos* – in which users were invited to download a screensaver program which sent HTTP requests to spammers' sites – made this an impossible concession. It is unfortunate that the irresponsible actions of a few activists have often overshadowed the good work of the many sane anti-spammers.

Returning to the book: as a summary of the *dramatis personae*, few of the protagonists look good at all times. The plot of the book unfolds around the two central characters and their associates, and contains some notable clashes.

It was very interesting to learn about the various response rates to different spam messages, along with the views of spammers as to whether these response rates could be considered good, bad or indifferent. Many of the spams and products mentioned in the book seem almost like old friends – who could forget the mound of emails exhorting the reader to buy playing cards emblazoned with 'most-wanted' Iraqis as issued during the US invasion of Iraq? Being able to link these spams to a particular individual added greatly to my interest in the book.

From the outset I was drawn to the subject matter of the book – and *Spam Kings* is certainly relevant to any person who has been spammed. However, the language and concepts do not assume a high level of technical knowledge and the technical discussions avoid going into depth – which might disappoint those who are after a more technically detailed description of spammers' activities.

The reported exchanges between spammers and anti-spammers are rife with expletives – and, of course, there are numerous mentions of male genitalia, which might be a concern for some readers. That said, the ideal audience for this book would be net-obsessed folk – for whom reading the book would be a perfect way to spend a lazy afternoon.

Those expecting technical depth or a reference tome may be disappointed, but those who simply crave entertainment and an insight into the world of some of those on the other end of the spam in their inbox will enjoy *Spam Kings*.